Problems with solutions in the Analysis of Algorithms

Draft date March 7, 2012

Contents

Chapter 1

Notations: Θ, O, Ω, o, and ω

The functions we consider are assumed to have positive real domains and real codomains unless specified otherwise. Furthermore, the functions are assumed to be *asymptotically positive*. The function $f(n)$ is asymptotically positive iff $\exists n_0 \colon \forall n \geq n_0, f(n) > 0$.

Basic definitions:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 : \ \forall n \geq n_0, \ 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\} \tag{1.1}$$

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \ \forall n \geq n_0, \ 0 \leq f(n) \leq c.g(n)\} \tag{1.2}$$

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 : \ \forall n \geq n_0, \ 0 \leq c.g(n) \leq f(n)\} \tag{1.3}$$

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \ \forall n \geq n_0, \ 0 \leq f(n) < c.g(n)\} \tag{1.4}$$

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 : \ \forall n \geq n_0, \ 0 \leq c.g(n) < f(n)\} \tag{1.5}$$

1.4 is equivalent to:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \tag{1.6}$$

if the limit exists. 1.5 is equivalent to:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \tag{1.7}$$

if the limit exists.

It is universally accepted to write "$f(n) = \Theta(g(n))$" instead of the formally correct "$f(n) \in \Theta(g(n))$".

Let us define the binary relations $\approx$, $\preceq$, $\prec$, $\succeq$, and $\succ$ over functions as follows. For any two functions $f(n)$ and $g(n)$:

$$f(n) \approx g(n) \quad \Leftrightarrow \quad f(n) = \Theta(g(n)) \tag{1.8}$$

$$f(n) \preceq g(n) \quad \Leftrightarrow \quad f(n) = O(g(n)) \tag{1.9}$$

$$f(n) \prec g(n) \quad \Leftrightarrow \quad f(n) = o(g(n)) \tag{1.10}$$

$$f(n) \succeq g(n) \quad \Leftrightarrow \quad f(n) = \Omega(g(n)) \tag{1.11}$$

$$f(n) \succ g(n) \quad \Leftrightarrow \quad f(n) = \omega(g(n)) \tag{1.12}$$

When the relations do not hold we write $f(n) \not\approx g(n)$, $f(n) \not\preceq g(n)$, *etc.*

Properties of the relations:

1. Reflexivity: $f(n) \approx f(n)$, $f(n) \preceq f(n)$, $f(n) \succeq f(n)$.
2. Symmetry: $f(n) \approx g(n) \;\Leftrightarrow\; g(n) \approx f(n)$.

   Proof: Assume $\exists c_1, c_2, n_0 > 0$ as necessitated by (1.1), so that $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$. Then $0 \leq \frac{1}{c_2} f(n) \leq g(n)$ and $g(n) \leq \frac{1}{c_1} f(n)$. Overall, $0 \leq \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$. So there exist positive constants $k_1 = \frac{1}{c_2}$ and $k_2 = \frac{1}{c_1}$, such that $0 \leq k_2.f(n) \leq g(n) \leq k_1.f(n)$ for all $n \geq n_0$. □
3. Transitivity:

   $f(n) \approx g(n)$ and $g(n) \approx h(n) \;\Rightarrow\; f(n) \approx h(n)$

   $f(n) \preceq g(n)$ and $g(n) \preceq h(n) \;\Rightarrow\; f(n) \preceq h(n)$

   $f(n) \prec g(n)$ and $g(n) \prec h(n) \;\Rightarrow\; f(n) \prec h(n)$

   $f(n) \succeq g(n)$ and $g(n) \succeq h(n) \;\Rightarrow\; f(n) \succeq h(n)$

   $f(n) \succ g(n)$ and $g(n) \succ h(n) \;\Rightarrow\; f(n) \succ h(n)$.
4. Transpose symmetry:

   $f(n) \succeq g(n) \;\Leftrightarrow\; g(n) \preceq f(n)$

   $f(n) \succ g(n) \;\Leftrightarrow\; g(n) \prec f(n)$.

5. $f(n) \prec g(n) \Rightarrow f(n) \preceq g(n)$
   $f(n) \preceq g(n) \not\Rightarrow f(n) \prec g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \succeq g(n)$
   $f(n) \succeq g(n) \not\Rightarrow f(n) \succ g(n)$
6. $f(n) \approx g(n) \Rightarrow f(n) \not\prec g(n)$
   $f(n) \approx g(n) \Rightarrow f(n) \not\succ g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\approx g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\succ g(n)$
   $f(n) \prec g(n) \Rightarrow f(n) \not\succeq g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\approx g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\prec g(n)$
   $f(n) \succ g(n) \Rightarrow f(n) \not\preceq g(n)$
7. $f(n) \approx g(n) \Leftrightarrow f(n) \preceq g(n)$ and $f(n) \succeq g(n)$
8. There do not exist functions $f(n)$ and $g(n)$, such that $f(n) \prec g(n)$ and $f(n) \succ g(n)$
9. Let $f(n) = f_1(n) \pm f_2(n) \pm f_3(n) \pm \ldots \pm f_k(n)$. Let

   $$f_1(n) \succ f_2(n)$$
   $$f_1(n) \succ f_3(n)$$
   $$\ldots$$
   $$f_1(n) \succ f_k(n)$$

   Then $f(n) \approx f_1(n)$.
10. Let $f(n) = f_1(n) \times f_2(n) \times \ldots \times f_k(n)$. Let some of the $f_i(n)$ functions be *positive* constants. Say, $f_1(n) = \text{const}$, $f_2(n) = \text{const}$, $\ldots$, $f_m(n) = \text{const}$ for some m such that $1 \leq m \leq n$. Then $f(n) \approx f_{m+1}(n) \times f_{m+2}(n) \times \ldots \times f_k(n)$.
11. The statement "$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is equal to some L such that $0 < L < \infty$" is stronger than "$f(n) \approx g(n)$":

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \quad \Rightarrow \quad f(n) \approx g(n) \tag{1.13}$$
$$f(n) \approx g(n) \quad \not\Rightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ exists.}$$

    To see why the second implication does not hold, suppose $f(n) = n^2$ and $g(n) = (2 + \sin(n))n^2$. Obviously $g(n)$ oscillates between n^2 and $3n^2$ and thus $f(n) \approx g(n)$ but $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist.

Problem 1 ([CLR00], pp. 24–25)**.** *Let* $f(n) = \frac{1}{2}n^2 - 3n$. *Prove that* $f(n) \approx n^2$.

Solution:
For a complete solution we have to show some concrete positive constants c_1 and c_2 and a concrete value n_0 for the variable, such that for all $n \geq n_0$,

$$0 \leq c_1.n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2.n^2$$

Since $n > 0$ this is equivalent to (divide by n^2):

$$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

What we have here are in fact three inequalities:

$$0 \leq c_1 \tag{1.14}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \tag{1.15}$$

$$\frac{1}{2} - \frac{3}{n} \leq c_2 \tag{1.16}$$

(1.14) is trivial, any $c_1 > 0$ will do. To satisfy (1.16) we can pick $n_0' = 1$ and then any positive c_2 will do; say, $c_2 = 1$. The smallest integer value for n that makes the right-hand side of (1.15) positive is 7; the right-hand side becomes $\frac{1}{2} - \frac{3}{7} = \frac{7}{14} - \frac{6}{14} = \frac{1}{14}$. So, to saisfy (1.15) we pick $c_1 = \frac{1}{14}$ and $n_0'' = 7$. The overall n_0 is $n_0 = \max\{n_0', n_0''\} = 7$. The solution $n_0 = 7$, $c_1 = \frac{1}{14}$, $c_2 = 1$ is obviously not unique. □

Problem 2. *Is it true that* $\frac{1}{1000}n^3 \preceq 1000n^2$*?*

Solution:
No. Assume the opposite. Then $\exists c > 0$ and $\exists n_0$, such that for all $n \geq n_0$:

$$\frac{1}{1000}n^3 \leq c.1000n^2$$

It follows that $\forall n \geq n_0$:

$$\frac{1}{1000}n \leq 1000.c \Leftrightarrow n \leq 1000000.c$$

That is clearly false. □

Problem 3. *Is it true that for any two functions, at least one of the five relations* $\approx$*,* $\preceq$*,* $\prec$*,* $\succeq$*, and* $\succ$ *holds between them?*

Solution:
No. Proof by demonstrating an counterexample ([CLR00, pp. 31]): let $f(n) = n$ and $g(n) = n^{1+\sin n}$. Since $g(n)$ oscillates between $n^0 = 1$ and n^2, it cannot be the case that $f(n) \approx g(n)$ nor $f(n) \preceq g(n)$ nor $f(n) \prec g(n)$ nor $f(n) \succeq g(n)$ nor $f(n) \succ g(n)$.

However, this argument from [CLR00] holds only when $n \in \mathbb{R}^+$. If $n \in \mathbb{N}^+$, we cannot use the function $g(n)$ directly, *i.e.* without proving additional stuff. Note that $\sin n$ reaches its extreme values -1 and 1 at $2k\pi + \frac{3\pi}{2}$ and $2k\pi + \frac{\pi}{2}$, respectively, for integer k. As these are irrational numbers, the integer n cannot be equal to any of them. So, it is no longer true that $g(n)$ oscillates between $n^0 = 1$ and n^2. If we insist on using $g(n)$ in our counterexample we have to argue, for instance, that:

- for infinitely many (positive) values of the integer variable, for some constant $\epsilon > 0$, it is the case that $g(n) \geq n^{1+\epsilon}$;
- for infinitely many (positive) values of the integer variable, for some constant $\sigma > 0$, it is the case that $g(n) \leq n^{1-\sigma}$.

An alternative is to use the function $g'(n) = n^{1+\sin(\pi n+\pi/2)}$ that indeed oscillates between $n^0 = 1$ and n^2 for integer n. Another alternative is to use

$$g''(n) = \begin{cases} n^2, & \text{if } n \text{ is even,} \\ 1, & \text{else.} \end{cases}$$

□

Problem 4. *Let* $p(n)$ *be any univariate polynomial of degree* k*, such that the coefficient in the higherst degree term is positive. Prove that* $p(n) \approx n^k$.

Solution:
$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ with $a_k > 0$. We have to prove that there exist positive constants c_1 and c_2 and some n_0 such that for all $n \geq n_0$, $0 \leq c_1 n^k \leq p(n) \leq c_2 n^k$. Since the leftmost inequality is obvious, we have to prove that

$$c_1 n^k \leq a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} \ldots + a_1 n + a_0 \leq c_2 n^k$$

For positive n we can divide by n^k, obtaining:

$$c_1 \leq a_k + \underbrace{\frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \ldots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}}_{T} \leq c_2$$

Now it is obvious that any c_1 and c_2 such that $0 < c_1 < a_k$ and $c_2 > a_k$ are suitable because $\lim_{n\to\infty} T = 0$.

□

Problem 5. *Let* $a \in \mathbb{R}$ *and* $b \in \mathbb{R}^+$*. Prove that* $(n + a)^b \approx n^b$

Solution:
Note that this problem does not reduce to Problem 4 except in the special case when b is integer. We start with the following trivial observations:

$n + a \leq n + |a| \leq 2n$, provided that $n \geq |a|$
$n + a \geq n - |a| \geq \frac{n}{2}$, provided that $\frac{n}{2} \geq |a|$, that is, $n \geq 2|a|$

It follows that:

$$\frac{1}{2} n \leq n + a \leq 2n, \text{ if } n \geq 2|a|$$

By raising to the b^{th} power we obtain:

$$\left(\frac{1}{2}\right)^b n^b \leq (n + a)^b \leq 2^b n^b$$

So we have a proof with $c_1 = \left(\frac{1}{2}\right)^b$, $c_2 = 2^b$, and $n_0 = \lceil 2|a| \rceil$.

Alternatively, solve this problem trivially using Problem 6. □

Problem 6. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$ *and any constant* $k \in \mathbb{R}^+$,

$$f(n) \approx g(n) \Leftrightarrow (f(n))^k \approx (g(n))^k$$

Solution:
In one direction, assume

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for some positive constants c_1 and c_2 and for all $n \geq n_0$ for some $n_0 > 0$. Raise the three inequalities to the k-th power (recall that k is positive) to obtain

$$0 \leq c_1^k (g(n))^k \leq (f(n))^k \leq c_2^k (g(n))^k, \text{ for all } n \geq n_0$$

Conclude that $(f(n))^k \approx (g(n))^k$ since c_1^k and c_2^k are positive constants.
In the other direction the proof is virtually the same, only raise to power $\frac{1}{k}$. □

Problem 7. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$*, it is the case that* $\max(f(n), g(n)) \approx f(n) + g(n)$.

Solution:
We are asked to prove there exist positive constants c_1 and c_2 and a certain n_0, such that for all $n \geq n_0$:

$$0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$$

As $f(n)$ and $g(n)$ are asymptotically positive,

$$\exists n_0' : \forall n \geq n_0', f(n) > 0$$
$$\exists n_0'' : \forall n \geq n_0'', g(n) > 0$$

Let $n_0''' = \max\{n_0', n_0''\}$. Obviously,

$$0 \leq c_1(f(n) + g(n)) \text{ for } n \geq n_0''', \text{ if } c_1 > 0$$

It is also obvious that when $n \geq n_0'''$:

$$\frac{1}{2} f(n) + \frac{1}{2} g(n) \leq \max(f(n), g(n))$$
$$f(n) + g(n) \geq \max(f(n), g(n)),$$

which we can write as:

$$\frac{1}{2}(f(n) + g(n)) \leq \max(f(n), g(n)) \leq f(n) + g(n)$$

So we have a proof with $n_0 = n_0'''$, $c_1 = \frac{1}{2}$, and $c_2 = 1$. □

Problem 8. *Prove or disprove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$ *such that* $f(n) - g(n)$ *is asymptotically positive, it is the case that* $\max(f(n), g(n)) \approx f(n) - g(n)$.

Solution:
The claim is false. As a counterexample consider $f(n) = n^3 + n^2$ and $g(n) = n^3 + n$. In this case, $\max(f(n), g(n)) = n^3 + n^2 = f(n)$ for all sufficiently large n. Clearly, $f(n) - g(n) = n^2 - n$ which is asymptotically positive but $n^3 + n^2 \not\approx n^2 - n$. □

Problem 9. *Which of the following are true:*

$$2^{n+1} \approx 2^n$$
$$2^{2n} \approx 2^n$$

Solution:
$2^{n+1} \approx 2^n$ is true because $2^{n+1} = 2.2^n$ and for any constant c, $c.2^n \approx 2^n$. On the other hand, $2^{2n} \approx 2^n$ is not true. Assume the opposite. Then, having in mind that $2^{2n} = 2^n.2^n$, it is the case that for some constant c_2 and all $n \to +\infty$:

$$2^n.2^n \leq c_2.2^n \quad \Leftrightarrow \quad 2^n \leq c_2$$

That is clearly false. □

Problem 10. *Which of the following are true:*

$$\frac{1}{n^2} \prec \frac{1}{n} \tag{1.17}$$
$$2^{\frac{1}{n^2}} \prec 2^{\frac{1}{n}} \tag{1.18}$$

Solution:
(1.17) is true because

$$0 \leq \frac{1}{n^2} < c.\frac{1}{n} \quad \Leftrightarrow \quad 0 \leq \frac{1}{n} < c$$

is true for every positive constant c and sufficiently large n. (1.18), however, is not true. Assume the opposite. Then:

$$\forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq 2^{\frac{1}{n^2}} < c.2^{\frac{1}{n}} \quad \Leftrightarrow \quad 0 \leq \frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}} < c \tag{1.19}$$

But

$$\lim_{n\to\infty}\left(\frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}}\right) = \lim_{n\to\infty}\left(2^{\frac{1}{n^2}-\frac{1}{n}}\right) = 1 \text{ because} \tag{1.20}$$

$$\lim_{n\to\infty}\left(\frac{1}{n^2} - \frac{1}{n}\right) = \lim_{n\to\infty}\left(\frac{1-n}{n^2}\right) = \lim_{n\to\infty}\left(\frac{\frac{1}{n}-1}{n}\right) = 0 \tag{1.21}$$

It follows that (1.19) is false. □

Problem 11. *Which of the following are true:*

$$\frac{1}{n} \prec 1 - \frac{1}{n} \tag{1.22}$$

$$2^{\frac{1}{n}} \prec 2^{1-\frac{1}{n}} \tag{1.23}$$

Solution:
(1.22) is true because

$$\lim_{n\to\infty} \left(\frac{\frac{1}{n}}{1-\frac{1}{n}} \right) = \lim_{n\to\infty} \left(\frac{\frac{1}{\not{n}}}{\frac{n-1}{\not{n}}} \right) = \lim_{n\to\infty} \frac{1}{n-1} = 0 \tag{1.24}$$

(1.23) is false because

$$\lim_{n\to\infty} \left(\frac{2^{\frac{1}{n}}}{2^{1-\frac{1}{n}}} \right) = \lim_{n\to\infty} \left(\frac{2^{\frac{2}{n}}}{2^1} \right) = \text{const} \tag{1.25}$$

Problem 12. *Let* a *be a constant such that* $a > 1$*. Which of the following are true:*

$$f(n) \approx g(n) \Rightarrow a^{f(n)} \approx a^{g(n)} \tag{1.26}$$

$$f(n) \preceq g(n) \Rightarrow a^{f(n)} \preceq a^{g(n)} \tag{1.27}$$

$$f(n) \prec g(n) \Rightarrow a^{f(n)} \prec a^{g(n)} \tag{1.28}$$

for all asymptotically positive functions $f(n)$ *and* $g(n)$*.*

Solution:
(1.26) is not true – Problem 9 provides a counterexample since $2n \approx n$ and $2^{2n} \not\approx 2^n$. The same counterexample suffices to prove that (1.27) is not true – note that $2n \preceq n$ but $2^{2n} \not\preceq 2^n$.

Now consider (1.28).

case 1, $g(n)$ **is increasing and unbounded:** The statement is true. We have to prove that

$$\forall c > 0, \exists n' : \forall n \geq n',\ 0 \leq a^{f(n)} < c.a^{g(n)} \tag{1.29}$$

Since the constant c is positive, we are allowed to consider its logarithm to base a, namely $k = \log_a c$. So, $c = a^k$. Of course, k can be positive or negative or zero. We can rewrite (1.29) as

$$\forall k, \exists n' : \forall n \geq n',\ 0 \leq a^{f(n)} < a^k a^{g(n)} \tag{1.30}$$

Taking logarithm to base a of the two inequalities, we have

$$\forall k, \exists n' : \forall n \geq n',\ 0 \leq f(n) < k + g(n) \tag{1.31}$$

If we prove (1.31), we are done. By definition ((1.4) on page 2), the premise is

$$\forall c > 0, \exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) < c.g(n)$$

Since that holds for any $c > 0$, in particular it holds for $c = \frac{1}{2}$. So, we have

$$\exists n_0 : \forall n \geq n_0,\ 0 \leq f(n) < \frac{g(n)}{2} \tag{1.32}$$

But $g(n)$ is increasing and unbounded. Therefore,

$$\forall k, \exists n_1 : \forall n \geq n_1,\ 0 < k + \frac{g(n)}{2} \tag{1.33}$$

We can rewrite (1.33) as

$$\forall k, \exists n_1 : \forall n \geq n_1,\ \frac{g(n)}{2} < k + g(n) \tag{1.34}$$

From (1.32) and (1.34) we have

$$\forall k, \exists n'' : \forall n \geq n'',\ 0 \leq f(n) < k + g(n) \tag{1.35}$$

Since (1.35) and (1.31) are the same, the proof is completed.

case 2, $g(n)$ is increasing but bounded: In this case (1.28) is not true. Consider Problem 11. As it is shown there, $\frac{1}{n} \prec 1 - \frac{1}{n}$ but $2^{\frac{1}{n}} \not\asymp 2^{1-\frac{1}{n}}$.

case 3, $g(n)$ is not increasing: In this case (1.28) is not true. Consider Problem 10. As it is shown there, $\frac{1}{n^2} \prec \frac{1}{n}$ but $2^{\frac{1}{n^2}} \not\asymp 2^{\frac{1}{n}}$. □

Problem 13. *Let a be a constant such that $a > 1$. Which of the following are true:*

$$a^{f(n)} \approx a^{g(n)} \Rightarrow f(n) \approx g(n) \tag{1.36}$$

$$a^{f(n)} \preceq a^{g(n)} \Rightarrow f(n) \preceq g(n) \tag{1.37}$$

$$a^{f(n)} \prec a^{g(n)} \Rightarrow f(n) \prec g(n) \tag{1.38}$$

for all asymptotically positive functions $f(n)$ and $g(n)$.

Solution:
(1.36) is true, if $g(n)$ is increasing and unbounded. Suppose there exist positive constants c_1 and c_2 and some n_0 such that

$$0 \leq c_1.a^{g(n)} \leq a^{f(n)} \leq c_2.a^{g(n)}, \forall n \geq n_0$$

Since $a > 1$ and $f(n)$ and $g(n)$ are asymptotically positive, for all sufficiently large n, the exponents have strictly larger than one values. Therefore, we can take logarithm to base a (ignoring the leftmost inequality) to obtain:

$$\log_a c_1 + g(n) \leq f(n) \leq \log_a c_2 + g(n)$$

First note that, provided that $g(n)$ is increasing and unbounded, for any constant k_1 such that $0 < k_1 < 1$, $k_1.g(n) \leq \log_a c_1 + g(n)$ for all sufficiently large n, regardless of whether the logarithm is positive or negative or zero. Then note that, provided that $g(n)$ is increasing and unbounded, for any constant k_2 such that $k_2 > 1$, $\log_a c_2 + g(n) \leq k_2.g(n)$ for all sufficiently large n, regardless of whether the logarithm is positive or negative or zero. Conclude there exists n_1, such that

$$k_1.g(n) \leq f(n) \leq k_2.g(n),\ \forall n \geq n_1$$

However, if $g(n)$ is increasing but bounded, (1.36) is not true. We already showed $2^{\frac{1}{n}} \approx 2^{1-\frac{1}{n}}$ (see 1.25). However, since $\lim_{n\to\infty}\left(\frac{\frac{1}{n}}{1-\frac{1}{n}}\right) = 0$ (see (1.24)), it is the case that $\frac{1}{n} \prec 1 - \frac{1}{n}$ according to (1.6).
Furthermore, if $g(n)$ is not increasing, (1.36) is not true. We already showed (see (1.20)) that $\lim_{n\to\infty}\left(\frac{2^{\frac{1}{n^2}}}{2^{\frac{1}{n}}}\right) = 1$. According to (1.13), it is the case that $2^{\frac{1}{n^2}} \approx 2^{\frac{1}{n}}$. However, $\frac{1}{n^2} \not\approx \frac{1}{n}$ (see (1.21)).
Consider (1.37). If $g(n)$ is increasing and unbounded, it is true. The proof can be done easily as in the case with (1.36). If $g(n)$ is increasing but bounded, the statement is false. Let $g(n) = \frac{1}{n}$. As shown in Problem 11, $2^{1-\frac{1}{n}} \approx 2^{\frac{1}{n}}$, therefore $2^{1-\frac{1}{n}} \preceq 2^{\frac{1}{n}}$, but $\frac{1}{n} \prec 1 - \frac{1}{n}$, therefore $1-\frac{1}{n} \not\preceq \frac{1}{n}$. Suppose $g(n)$ is not increasing. Let $g(n) = \frac{1}{n}$. We know that $2^{\frac{1}{n^2}} \preceq 2^{\frac{1}{n}}$ but $\frac{1}{n^2} \not\preceq \frac{1}{n}$.

Now consider (1.38). It is not true. As a counterexample, consider that $2^n \prec 2^{2n}$ but $n \not\prec 2n$. □

Problem 14. *Let a be a constant such that $a > 1$. Which of the following are true:*

$$\log_a \phi(n) \approx \log_a \psi(n) \Rightarrow \phi(n) \approx \psi(n) \quad (1.39)$$
$$\log_a \phi(n) \preceq \log_a \psi(n) \Rightarrow \phi(n) \preceq \psi(n) \quad (1.40)$$
$$\log_a \phi(n) \prec \log_a \psi(n) \Rightarrow \phi(n) \prec \psi(n) \quad (1.41)$$
$$\phi(n) \approx \psi(n) \Rightarrow \log_a \phi(n) \approx \log_a \psi(n) \quad (1.42)$$
$$\phi(n) \preceq \psi(n) \Rightarrow \log_a \phi(n) \preceq \log_a \psi(n) \quad (1.43)$$
$$\phi(n) \prec \psi(n) \Rightarrow \log_a \phi(n) \prec \log_a \psi(n) \quad (1.44)$$

for all asymptotically positive functions $\phi(n)$ and $\psi(n)$.

Solution:
Let $\phi(n) = a^{f(n)}$ and $\psi(n) = a^{g(n)}$, which means that $\log_a \phi(n) = f(n)$ and $\log_a \psi(n) = g(n)$. Consider (1.26) and conclude that (1.39) is not true. Consider (1.36) and conclude that (1.42) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.27) and conlude that (1.40) is not true. Consider (1.37) and conclude that (1.43) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.28) and conclude that (1.41) is true if $\psi(n)$ is increasing and unbounded, and false otherwise. Consider (1.38) and conlude that (1.44) is not true. □

Problem 15. *Prove that for any two asymptotically positive functions* $f(n)$ *and* $g(n)$, $f(n) \approx g(n)$ *iff* $f(n) \preceq g(n)$ *and* $f(n) \succeq g(n)$.

Solution:
In one direction, assume that $f(n) \approx g(n)$. Then there exist positive constants c_1 and c_2 and some n_0, such that:

$$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n),\ \forall n \geq n_0$$

It follows that,

$$0 \leq c_1.g(n) \leq f(n),\ \forall n \geq n_0 \tag{1.45}$$
$$0 \leq f(n) \leq c_2.g(n),\ \forall n \geq n_0 \tag{1.46}$$

In the other direction, assume that $f(n) \preceq g(n)$ and $f(n) \succeq g(n)$. Then there exists a positive constant c' and some n_0', such that:

$$0 \leq f(n) \leq c'.g(n), \forall n \geq n_0'$$

and there exists a positive constant c'' and some n_0'', such that:

$$0 \leq c''.g(n) \leq f(n), \forall n \geq n_0''$$

It follows that:

$$0 \leq c'.g(n) \leq f(n) \leq c''.g(n), \forall n \geq \max\{n_0', n_0''\}$$

□

Lemma 1 (Stirling's approximation)**.**

$$n! = \sqrt{2\pi n}\,\frac{n^n}{e^n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) \tag{1.47}$$

□

Here, $\Theta\left(\frac{1}{n}\right)$ means any function that is in the set $\Theta\left(\frac{1}{n}\right)$.

Problem 16. *Prove that*

$$\lg n! \approx n \lg n \tag{1.48}$$

Solution:
Use Stirling's approximation, ignoring the $\left(1 + \Theta\left(\frac{1}{n}\right)\right)$ factor, and take logarithm of both sides to obtain:

$$\lg(n!) = \lg(\sqrt{2\pi}) + \lg n + n \lg n - n \lg e$$

By Property 9 of the relations, $\lg(\sqrt{2\pi}) + \lg n + n \lg n - n \lg e \approx n \lg n$. □

Problem 17. *Prove that for any constant* $a > 1$,

$$a^n \prec n! \prec n^n \tag{1.49}$$

Solution:
Because of the factorial let us restrict n to positive integers.

$$\lim_{n\to\infty}\left(\frac{n.(n-1).(n-2)\ldots 2.1}{\underbrace{a\,.\,a\,.\,a\,\ldots\,a\,.\,a}_{n \text{ times}}}\right) = \infty$$

$$\lim_{n\to\infty}\left(\frac{n.(n-1).(n-2)\ldots 2.1}{\underbrace{n\,.\,n\,.\,n\,\ldots\,n\,.\,n}_{n \text{ times}}}\right) = 0$$

□

Problem 18 (polylogarithm versus constant power of n)**.** *Let* a, k *and* ϵ *be any constants, such that* $k > 0$, $a > 1$, *and* $\epsilon > 0$. *Prove that:*

$$(\log_a n)^k \prec n^\epsilon \tag{1.50}$$

Solution:

$$\lim_{n\to\infty}\frac{n^\epsilon}{(\log_a n)^k} = \qquad \text{let } b \leftarrow \frac{\epsilon}{k}$$

$$\lim_{n\to\infty}\frac{(n^b)^k}{(\log_a n)^k} =$$

$$\lim_{n\to\infty}\left(\frac{n^b}{\log_a n}\right)^k = \qquad k \text{ is positive}$$

$$\lim_{n\to\infty}\frac{n^b}{\log_a n} = \qquad \text{use l'Hôpital's rule}$$

$$\lim_{n\to\infty}\frac{bn^{b-1}}{\left(\frac{1}{\ln a}\right)\left(\frac{1}{n}\right)} =$$

$$\lim_{n\to\infty}(\ln a)b\,n^b = \infty$$

□

Problem 19 (constant power of n versus exponent)**.** *Let* a *and* ϵ *be any constants, such that* $a > 1$ *and* $\epsilon > 0$. *Prove that:*

$$n^\epsilon \prec a^n \tag{1.51}$$

Solution:
Take $\log_a$ of both sides. The left-hand side yields $\epsilon.\log_a n$ and the right-hand side yields n. But $\epsilon.\log_a n \prec n$ because of Problem 18. Conclude immediately the desired relation holds. □

Definition 1 (log-star function, [CLR00], pp. 36)**.** *Let the function* $\lg^{(i)} n$ *be defined recursively for nonnegative integers* i *as follows:*

$$\lg^{(i)} n = \begin{cases} n, & \text{if } i = 0 \\ \lg\left(\lg^{(i-1)} n\right), & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0 \\ \text{undefined}, & \text{if } i > 0 \text{ and } \lg^{(i-1)} n < 0 \text{ or } \lg^{(i-1)} n \text{ is undefined} \end{cases}$$

Then

$$\lg^* n = \min\left\{i \geq 0 \,|\, \lg^{(i)} n \leq 1\right\}$$

□

According to this definition,

$$\begin{aligned} \lg^* 2 &= 1, \text{ since } \lg^{(0)} 2 = 2 \text{ and } \lg^{(1)} 2 = \lg\left(\lg^{(0)} 2\right) = \lg(2) = 1 \\ \lg^* 3 &= 2, \text{ since } \lg^{(0)} 3 = 3 \text{ and } \lg\left(\lg^{(0)} 3\right) = \lg(\lg 3) = 0.6644\ldots \\ \lg^* 4 &= 2 \\ \lg^* 5 &= 3 \\ &\ldots \\ \lg^* 16 &= 3 \\ \lg^* 17 &= 4 \\ &\ldots \\ \lg^* 65536 &= 4 \\ \lg^* 65537 &= 5 \\ &\ldots \\ \lg^* 2^{65536} &= 5 \\ \lg^*\left(2^{65536} + 1\right) &= 6 \\ &\ldots \end{aligned}$$

Obviously, every real number t can be represented by a *tower of twos*:

$$t = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^s}}}}}}$$

where s is a real number such that $1 < s \leq 2$. The *height of the tower* is the number of elements in this sequence. For instance,

number	*its tower of twos*	*the height of the tower*
2	2	1
3	$2^{1.5849625007...}$	2
4	2^2	2
5	$2^{2^{1.2153232957...}}$	3
16	2^{2^2}	3
17	$2^{2^{2^{1.0223362884...}}}$	4
65536	$2^{2^{2^2}}$	4
65537	$2^{2^{2^{1.0000051642167...}}}$	5

Having that in mind, it is trivial to see that $\lg^* n$ is the height of the tower of twos of n.

Problem 20 ([CLR00], problem 2-3, pp. 38–39)**.** *Rank the following thirty functions by order of growth. That is, find the equivalence classes of the "$\approx$" relation and show their order by "$\succ$".*

$\lg(\lg^* n)$	$2^{\lg^* n}$	$\left(\sqrt{2}\right)^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{\frac{1}{\lg n}}$
$\ln\ln n$	$\lg^* n$	$n.2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n\lg n$	$2^{2^{n+1}}$

Solution:

$2^{2^{n+1}} \succ 2^{2^n}$ because $2^{2^{n+1}} = 2^{2.2^n} = 2^{2^n} \times 2^{2^n}$.

$2^{2^n} \succ (n+1)!$ To see why, take logarithm to base two of both sides. The left-hand side becomes 2^n, the right-hand side becomes $\lg((n+1)!)$ By (1.47), $\lg((n+1)!) \approx (n+1)\lg(n+1)$, and clearly $(n+1)\lg(n+1) \approx n\lg n$. As $2^n \succ n\lg n$, by (1.41) we have $2^{2^n} \succ (n+1)!$

$(n+1)! \succ n!$ because $(n+1)! = (n+1) \times n!$

$n! \succ e^n$ by (1.49).

$e^n \succ n.2^n$. To see why, consider:

$$\lim_{n\to\infty} \frac{n.2^n}{e^n} = \lim_{n\to\infty} \frac{n}{\frac{e^n}{2^n}} = \lim_{n\to\infty} \frac{n}{\left(\frac{e}{2}\right)^n} = 0$$

$n.2^n \succ 2^n$

$2^n \succ \left(\frac{3}{2}\right)^n$. To see why, consider:

$$\lim_{n\to\infty} \frac{\left(\frac{3}{2}\right)^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{4}\right)^n = 0$$

$\left(\frac{3}{2}\right)^n \succ n^{\lg(\lg n)}$. To see why, take lg of both sides. The left-hand side becomes $n.\lg\left(\frac{3}{2}\right)$, the right-hand side becomes $\lg n.\lg(\lg n)$. Clearly, $\lg^2 n \succ \lg n.\lg(\lg n)$ and $n \succ \lg^2 n$ by (1.50). By transitivity, $n \succ \lg n.\lg(\lg n)$, and so $n.\lg\left(\frac{3}{2}\right) \succ \lg n.\lg(\lg n)$. Apply (1.41) and the desired conclusion follows.

$(\lg n)^{\lg n} = n^{\lg(\lg n)}$, which is obvious if we take lg of both sides. So, $(\lg n)^{\lg n} \approx n^{\lg(\lg n)}$.

$(\lg n)^{\lg n} \succ (\lg n)!$ To see why, substitute $\lg n$ with m, obtaining $m^m \succ m!$ and apply (1.49).

$(\lg n)! \succ n^3$. Take lg of both sides. The left-hand side becomes $\lg((\lg n)!)$. Substitute $\lg n$ with m, obtaining $\lg(m!)$. By (1.48), $\lg(m!) \approx m\lg m$, therefore $\lg((\lg n)!) \approx (\lg n).(\lg(\lg n))$. The right-hand side becomes $3.\lg n$. Compare $(\lg n).(\lg(\lg n))$ with $3.\lg n$:

$$\lim_{n\to\infty} \frac{3.\lg n}{(\lg n).(\lg(\lg n))} = \lim_{n\to\infty} \frac{3}{\lg(\lg n)} = 0$$

It follows that $(\lg n).(\lg(\lg n)) \succ 3.\lg n$. Apply (1.41) to draw the desired conclusion.

$n^3 \succ n^2$.

$n^2 \approx 4^{\lg n}$ because $4^{\lg n} = 2^{2\lg n} = 2^{\lg n^2} = n^2$ by the properties of the logarithm.

$n^2 \succ n\lg n$.

$\lg n! \approx n\lg n$ (see (1.48)).

$n\lg n \succ n$.

$n \approx 2^{\lg n}$ because $n = 2^{\lg n}$ by the properties of the logarithm.

$n \succ (\sqrt{2})^{\lg n}$ because $(\sqrt{2})^{\lg n} = 2^{\frac{1}{2}\lg n} = 2^{\lg\sqrt{n}} = \sqrt{n}$ and clearly $n \succ \sqrt{n}$.

$(\sqrt{2})^{\lg n} \succ 2^{\sqrt{2\lg n}}$. To see why, note that $\lg n \succ \sqrt{\lg n}$, therefore $\frac{1}{2}.\lg n \succ \sqrt{2}.\sqrt{\lg n} = \sqrt{2\lg n}$. Apply (1.28) and conclude that $2^{\frac{1}{2}.\lg n} \succ 2^{\sqrt{2\lg n}}$, *i.e.* $(\sqrt{2})^{\lg n} \succ 2^{\sqrt{2\lg n}}$.

$2^{\sqrt{2\lg n}} \succ \lg^2 n$. To see why, take lg of both sides. The left-hand side becomes $\sqrt{2\lg n}$ and the right-hand side becomes $\lg(\lg^2 n) = 2.\lg(\lg n)$. Substitute $\lg n$ with m: the left-hand side becomes $\sqrt{2m} = \sqrt{2}\sqrt{m} = \sqrt{2}.m^{\frac{1}{2}}$ and the right-hand side becomes $2\lg m$. By (1.50) we know that $m^{\frac{1}{2}} \succ \lg m$, therefore $\sqrt{2}.m^{\frac{1}{2}} \succ 2\lg m$, therefore $\sqrt{2m} \succ 2\lg m$, therefore $\sqrt{2\lg n} \succ \lg(\lg^2 n)$. Having in mind (1.41) we draw the desired conclusion.

$\lg^2 n \succ \ln n$. To see this is true, observe that $\ln n = \frac{\lg n}{\lg e}$.

$\ln n \succ \sqrt{\lg n}$.

$\sqrt{\lg n} \succ \ln \ln n$. The left-hand side is $\sqrt{\frac{\ln n}{\ln 2}}$. Substitute $\ln n$ with m and the claim becomes $\frac{1}{\sqrt{\ln 2}} . \sqrt{m} \succ \ln m$, which follows from (1.50).

$\ln \ln n \succ 2^{\lg^* n}$. To see why this is true, note that $\ln \ln n \approx \lg \lg n$ and rewrite the claim as $\lg \lg n \succ 2^{\lg^* n}$. Take lg of both sides. The left-hand side becomes $\lg \lg \lg n$, *i.e.* a triple logarithm. The right-hand side becomes $\lg^* n$. If we think of n as a tower of twos, it is obvious that the triple logarithm decreases the height of the tower with three, while, as we said before, the log-star measures the height of the tower. Clearly, the latter is *much* smaller than the former.

$2^{\lg^* n} \succ \lg^* n$. Clearly, for any increasing function $f(n)$, $2^{f(n)} \succ f(n)$.

$\lg^* n \approx \lg^* (\lg n)$. Think of n as a tower of twos and note that the difference in the height of n and $\lg n$ is one. Therefore, $\lg^* (\lg n) = (\lg^* n) - 1$.

$\lg^* n \succ \lg (\lg^* n)$. Substitute $\lg^* n$ with $f(n)$ and the claim becomes $f(n) \succ \lg f(n)$ which is clearly true since $f(n)$ is increasing.

$\lg (\lg^* n) \succ 1$.

$1 \approx n^{\frac{1}{\lg n}}$. Note that $n^{\frac{1}{\lg n}} = 2$: take lg of both sides, the left-hand side becomes $\lg \left(n^{\frac{1}{\lg n}} \right) = \frac{1}{\lg n} . \lg n = 1$ and the right-hand side becomes $\lg 2 = 1$. □

Problem 21. *Give an example of a function* $f(n)$, $n \in \mathbb{N}^+$, *such that for function* $g(n)$ *among the thirty functions from Problem 20,* $f(n) \not\preceq g(n)$ *and* $f(n) \not\succeq g(n)$.

Solution:
For instance,

$$f(n) = \begin{cases} 2^{2^{n+2}}, & \text{if } n \text{ is even} \\ \frac{1}{n}, & \text{if } n \text{ is odd} \end{cases}$$

□

Problem 22. *Is it true that for any asymptotically positive functions* $f(n)$ *and* $g(n)$, $f(n) + g(n) \approx \min (f(n), g(n))$*?*

Solution:
No. As a counterexample, consider $f(n) = n$ and $g(n) = 1$. Then $\min (f(n), g(n)) = 1$, $f(n) + g(n) = n + 1$, and certainly $n + 1 \not\approx 1$. □

Problem 23. *Is it true that for any asymptotically positive function* $f(n)$, $f(n) \preceq (f(n))^2$*?*

Solution:
If $f(n)$ is increasing, it is trivially true. If it is decreasing, however, it may not be true: consider (1.17). □

Problem 24. *Is it true that for any asymptotically positive function* $f(n)$, $f(n) \approx f(\frac{n}{2})$*?*

Solution:
No. As a counterexample, consider $f(n) = 2^n$. Then $f(\frac{n}{2}) = 2^{\frac{n}{2}}$. As we already saw, $2^n \not\approx 2^{\frac{n}{2}}$. □

Problem 25. *Compare the growth of* $n^{\lg n}$ *and* $(\lg n)^n$.

Solution:
Take logarithm of both sides. The left-hand side becomes $(\lg n)(\lg n) = \lg^2 n$, the right-hand side, $n.\lg(\lg n)$. As $n.\lg(\lg n) \succ \lg^2 n$, it follows that $(\lg n)^n \succ n^{\lg n}$. □

Problem 26. *Compare the growth of* $n^{\lg\lg\lg n}$ *and* $(\lg n)!$

Solution:
Take lg of both sides. The left-hand side becomes $(\lg n).(\lg\lg\lg n)$, the right-hand side becomes $\lg((\lg n)!)$. Substitute $\lg n$ with m is the latter expression to get $\lg((m)!) \approx m \lg m$. And that is $(\lg n).(\lg\lg n)$. Since $(\lg n).(\lg\lg n) \succ (\lg n).(\lg\lg\lg n)$, it follows that $(\lg n)! \succ n^{\lg\lg\lg n}$. □

Problem 27. *Let* $n!! = (n!)!$. *Compare the growth of* $n!!$ *and* $(n-1)!! \times ((n-1)!)^{n!}$.

Solution:
Let $(n-1)! = \nu$. Then $n! = n\nu$. We compare

$$n!! \quad \text{vs} \quad (n-1)!! \times ((n-1)!)^{n!}$$

$$(n\nu)! \quad \text{vs} \quad \nu! \times \nu^{n\nu}$$

Apply Stirling's approximation to both sides to get:

$$\sqrt{2\pi n\nu}\,\frac{(n\nu)^{n\nu}}{e^{n\nu}} \quad \text{vs} \quad \sqrt{2\pi\nu}\,\frac{\nu^\nu}{e^\nu} \times \nu^{n\nu}$$

$$\sqrt{2\pi n\nu}\,(n\nu)^{n\nu} \quad \text{vs} \quad \sqrt{2\pi\nu}\,e^{(n-1)\nu} \times \nu^\nu \times \nu^{n\nu}$$

Divide by $\sqrt{2\pi\nu}\,\nu^{n\nu}$ both sides:

$$\sqrt{n}\,n^{n\nu} \quad \text{vs} \quad e^{(n-1)\nu} \times \nu^\nu$$

Ignore the $\sqrt{n}$ factor on the left. If we derive without it that the left side grows faster, surely it grows even faster with it. So, consider:

$$n^{n\nu} \quad \text{vs} \quad e^{(n-1)\nu} \times \nu^\nu$$

Raise both sides to $\frac{1}{\nu}$:

$$n^n \quad \text{vs} \quad e^{n-1} \times \nu$$

That is,

$$n^n \quad \text{vs} \quad e^{n-1} \times (n-1)!$$

Apply Stirling's aproximation second time to get:

$$n^n \quad \text{vs} \quad e^{n-1} \times \sqrt{2\pi(n-1)}\,\frac{(n-1)^{n-1}}{e^{n-1}}$$

That is,

$$n^n \quad \text{vs} \quad \sqrt{2\pi(n-1)}\,(n-1)^{n-1}$$

Since $\sqrt{2\pi(n-1)}\,(n-1)^{n-1} \approx (n-1)^{\left(n-\frac{1}{2}\right)}$, we have

$$n^n \quad \text{vs} \quad (n-1)^{\left(n-\frac{1}{2}\right)}$$

Clearly, $n^n \succ (n-1)^{\left(n-\frac{1}{2}\right)}$, therefore $n!! \succ (n-1)!! \times ((n-1)!)^{n!}$. □

Lemma 2. *The function series:*

$$S(x) = \frac{\ln x}{x} + \frac{\ln^2 x}{x^2} + \frac{\ln^3 x}{x^3} + \ldots$$

is convergent for $x > 1$. *Furthermore,* $\lim_{x\to\infty} S(x) = 0$.

Proof:
It is well known that the series

$$S'(x) = \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \ldots$$

called *geometric series* is convergent for $x > 1$ and $S'(x) = \frac{1}{x-1}$ when $x > 1$. Clearly, $\lim_{x\to\infty} S'(x) = 0$. Consider the series

$$S''(x) = \frac{1}{\sqrt{x}} + \frac{1}{(\sqrt{x})^2} + \frac{1}{(\sqrt{x})^3} + \ldots \tag{1.52}$$

It is a geometric series and is convergent for $\sqrt{x} > 1$, *i.e.* $x > 1$, and $\lim_{x\to\infty} S''(x) = 0$. Let us rewrite $S(x)$ as

$$S(x) = \frac{1}{\sqrt{x}.\frac{\sqrt{x}}{\ln x}} + \frac{1}{(\sqrt{x})^2.\left(\frac{\sqrt{x}}{\ln x}\right)^2} + \frac{1}{(\sqrt{x})^3.\left(\frac{\sqrt{x}}{\ln x}\right)^3} + \ldots \tag{1.53}$$

For each term $f_k(x) = \frac{1}{(\sqrt{x})^k.\left(\frac{\sqrt{x}}{\ln x}\right)^k}$ of $S(x)$, $k \geq 1$, for large enough x, it is the case that $f_k(x) < g_k(x)$ where $g_k(x) = \frac{1}{\left(\sqrt{x}\right)^k}$ is the k^{th} term of $S''(x)$. To see why this is true, consider (1.50). Then the fact that $S''(x)$ is convergent and $\lim_{x\to\infty} S''(x) = 0$ implies the desired conclusion. □

Problem 28 ([Knu73], pp. 107)**.** *Prove that* $\sqrt[n]{n} \approx 1$.

Solution:
We will show an even stronger statement: $\lim_{n\to\infty} \sqrt[n]{n} = 1$. It is known that:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Note that $\sqrt[n]{n} = e^{\ln \sqrt[n]{n}} = e^{\left(\frac{\ln n}{n}\right)}$.

$$e^{\left(\frac{\ln n}{n}\right)} = 1 + \underbrace{\frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots}_{T(n)}$$

Lemma 2 implies $\lim_{n\to\infty} T(n) = 0$. It follows that $\lim_{n\to\infty} \sqrt[n]{n} = 1$. □
We can also say that $\sqrt[n]{n} = 1 + O\left(\frac{\lg n}{n}\right)$, $\sqrt[n]{n} = 1 + \frac{\lg n}{n} + O\left(\frac{\lg^2 n}{n^2}\right)$, *etc*, where the big-Oh notation stands for any function of the set.

Problem 29 ([Knu73], pp. 107)**.** *Prove that* $n\left(\sqrt[n]{n} - 1\right) \approx \ln n$.

Solution:
As

$$\sqrt[n]{n} = 1 + \frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots$$

it is the case that:

$$\sqrt[n]{n} - 1 = \frac{\ln n}{n} + \frac{\left(\frac{\ln n}{n}\right)^2}{2!} + \frac{\left(\frac{\ln n}{n}\right)^3}{3!} + \dots$$

Multiply by n to get:

$$n\left(\sqrt[n]{n} - 1\right) = \ln n + \underbrace{\frac{(\ln n)^2}{2!n} + \frac{(\ln n)^3}{3!n^2} + \dots}_{T(n)}$$

Note that $\lim_{n\to\infty} T(n) = 0$ by an obvious generalisation of Lemma 2. The claim follows immediately. □

Problem 30. *Compare the growth of* n^n, $(n+1)^n$, n^{n+1}, *and* $(n+1)^{n+1}$.

Solution:
$n^n \approx (n+1)^n$ because

$$\lim_{n\to\infty} \frac{(n+1)^n}{n^n} = \lim_{n\to\infty} \left(\frac{n+1}{n}\right)^n = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$$

Clearly, $n^n \prec n^{(n+1)} = n.n^n$. And $n^{(n+1)} \approx (n+1)^{(n+1)}$:

$$\lim_{n\to\infty} \frac{(n+1)^{n+1}}{n^{n+1}} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^{n+1} = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \lim_{n\to\infty} \left(1 + \frac{1}{n}\right) = e.1 = e$$

□

Problem 31. *Let* k *be a constant such that* $k > 1$*. Prove that*

$$1 + k + k^2 + k^3 + \ldots + k^n = \Theta(k^n)$$

Solution:
First assume n is an integer variable. Then

$$1 + k + k^2 + k^3 + \ldots + k^n = \frac{k^{n+1} - 1}{k - 1} = \Theta(k^n)$$

The result can obviously be extended for real n, provided we define appropriately the sum. For instance, if $n \in \mathbb{R}^+ \setminus \mathbb{N}$ let the sum be

$$S(n) = 1 + k + k^2 + k^3 + \ldots + k^{\lfloor n-1 \rfloor} + k^{\lfloor n \rfloor} + k^n$$

By the above result, $S(n) = k^n + \Theta\left(k^{\lfloor n \rfloor}\right) = \Theta(k^n)$. □

Problem 32. *Let* k *be a constant such that* $0 < k < 1$*. Prove that*

$$1 + k + k^2 + k^3 + \ldots + k^n = \Theta(1)$$

Solution:

$$1 + k + k^2 + k^3 + \ldots + k^n < \sum_{t=0}^{\infty} k^t = \frac{1}{1 - k} = \Theta(1)$$ □

Corollary 1.

$$1 + k + k^2 + k^3 + \ldots + k^n = \begin{cases} \Theta(1), & \text{if } 0 < k < 1 \\ \Theta(n), & \text{if } k = 1 \\ \Theta(k^n), & \text{if } k > 1 \end{cases}$$ □

Problem 33. *Let* $f(x) = 2^{2^{\lfloor x \rfloor}}$ *and* $g(x) = 2^{2^{\lceil x \rceil}}$ *where* $x \in \mathbb{R}^+$*. Determine which of the following are true and which are false:*

1. $f(x) \approx g(x)$
2. $f(x) \preceq g(x)$
3. $f(x) \prec g(x)$
4. $f(x) \succeq g(x)$
5. $f(x) \succ g(x)$

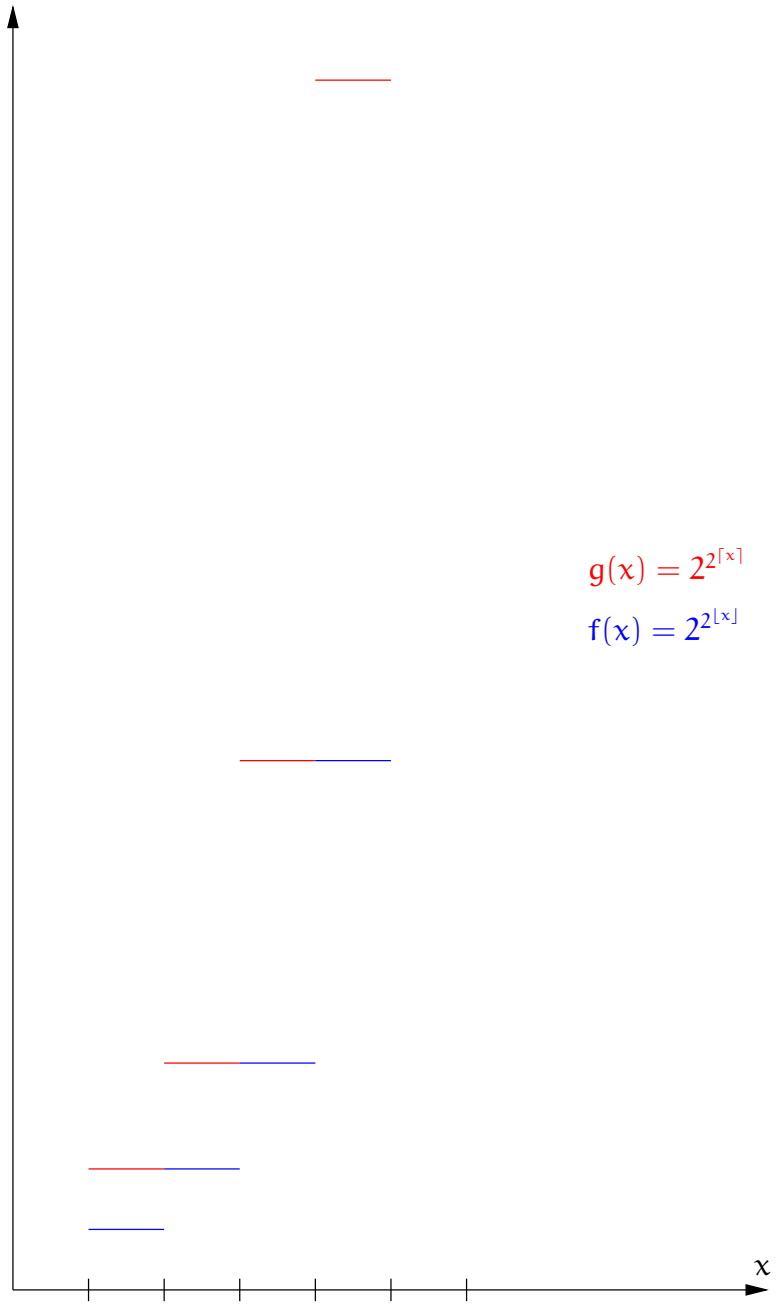


Figure 1.1: $f(x)$ and $g(x)$ from Problem 33.

Solution:
Note that $\forall x \in \mathbb{N}^+, \lfloor x \rfloor = \lceil x \rceil$, therefore $f(x) = g(x)$ whenever $x \in \mathbb{N}^+$. On the other hand, $\forall x \in \mathbb{R}^+ \setminus \mathbb{N}^+, \lceil x \rceil = \lfloor x \rfloor + 1$, therefore $g(x) = 2^{2^{\lfloor x \rfloor + 1}} = 2^{2.2^{\lfloor x \rfloor}} = \left(2^{2^{\lfloor x \rfloor}}\right)^2 = (f(x))^2$ whenever $x \in \mathbb{R}^+ \setminus \mathbb{N}^+$. Figure 1.1 illustrates the way that $f(x)$ and $g(x)$ grow.

First assume that $f(x) \prec g(x)$. By definition, for every constant $c > 0$ there exists x_0, such that $\forall x \geq x_0$, $f(x) < c.g(x)$. It follows for $c = 1$ there exists a value for the variable, say x', such that $\forall x \geq x'$, $f(x) < g(x)$. However,

$$\lceil x' \rceil \geq x'$$

Therefore,

$$f(\lceil x' \rceil) < g(\lceil x' \rceil)$$

On the other hand,

$$\lceil x' \rceil \in \mathbb{N}^+ \;\Rightarrow\; f(\lceil x' \rceil) = g(\lceil x' \rceil)$$

We derived

$$f(\lceil x' \rceil) = g(\lceil x' \rceil) \text{ and } f(\lceil x' \rceil) < g(\lceil x' \rceil) \;\lightning$$

We derived a contradiction, therefore

$$f(x) \not\prec g(x)$$

Analogously we prove that

$$f(x) \not\succ g(x)$$

To see that $f(x) \not\approx g(x)$, note that $\forall \tilde{x} \in \mathbb{R}^+, \exists x'' \geq \tilde{x}$, such that $g(x'') = (f(x''))^2$. As $f(x)$ is a growing function, its square must have a higher asymptotic growth rate.

Now we prove that $f(x) \preceq g(x)$. Indeed,

$$\begin{aligned}
&\forall x \in \mathbb{R}^+,\ \lfloor x \rfloor \leq \lceil x \rceil \Rightarrow \\
&\forall x \in \mathbb{R}^+,\ 2^{\lfloor x \rfloor} \leq 2^{\lceil x \rceil} \Rightarrow \\
&\forall x \in \mathbb{R}^+,\ 2^{2^{\lfloor x \rfloor}} \leq 2^{2^{\lceil x \rceil}} \Rightarrow \exists c > 0, c = \text{const},\ \text{such that } \forall x \in \mathbb{R}^+,\ 2^{2^{\lfloor x \rfloor}} \leq c.2^{2^{\lceil x \rceil}}
\end{aligned}$$

Finally we prove that $f(x) \not\succeq g(x)$. Assume the opposite. Since $f(x) \preceq g(x)$, by property 7 on page 4 we derive $f(x) \approx g(x)$ and that contradicts our result that $f(x) \not\approx g(x)$. □

Chapter 2

Iterative Algorithms

In this section we compute the asymptotic running time of algorithms that use the **for** and **while** statements but make no calls to other algorithms or themselves. The time complexity is expressed as a function of the size of the input, in case the input is an array or a matrix, or as a function of the upper bound of the loops. Consider the time complexity of the following trivial algorithm.

```
ADD-1(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      a ← a + i
4  return a
```

We make the folloing assumptions:

- the expression at line 3 is executed in constant time regardless of how large n is,
- the expression at line 1 is executed in constant time, and
- the loop control variable check and assignment of the **for** loop at line 2 are executed in constant time.

Since we are interested in the asymptotic running time, not in the precise one, it suffices to find the number of times the expression inside the loop (line 3 in this case) is executed as a function of the upper bound on the loop control variable n. Let that function be $f(n)$. The time complexity of ADD-1 will then be $\Theta(f(n))$. We compute $f(n)$ as follows. First we substitute the expression inside the loop with $a \leftarrow a + 1$ where a is the counter variable that is set to zero initially. Then find the value of a after the loop finishes as a function of n where n is the upper bound n of the loop control variable i. Using that approach, algorithm ADD-1 becomes ADD-1-MODIFIED as follows.

```
ADD-1-MODIFIED(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      a ← a + 1
4  return a
```

The value that ADD-1-MODIFIED outputs is $\sum_{i=1}^{n} 1 = n$, therefore its time complexity is $\Theta(n)$. Now consider another algorithm:

```
ADD-2(n: nonnegative integer)
1  return n
```

Clearly, ADD-2 is equivalent to ADD-1 but the running time of ADD-2 is, under the said assumptions, constant. We denote constant running time by $\Theta(1)$[†]. It is not incorrect to say the running time of both algorithms is $O(n)$ but the big-Theta notation is superior as it grasps precisely—in the asymptotic sense—the algorithm's running time.

Consider the following iterative algorithm:

```
ADD-3(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← 1 to n
4          a ← a + 1
5  return a
```

The value it outputs is $\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$, therefore its time complexity is $\Theta(n^2)$.

Algorithm ADD-3 has two *nested cycles*. We can generalise that the running time of k nested cycles as follows.

```
ADD-GENERALISED(n: nonnegative integer)
1  for i_1 ← 1 to n
2      for i_2 ← 1 to n
3          ...
4          for i_k ← 1 to n
5              expression
```

where **expression** is computed in $\Theta(1)$, has running time $\Theta(n^k)$.

Let us consider a modification of ADD-3:

```
ADD-4(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i to n
4          a ← a + 1
5  return a
```

[†]All constants are bit-Theta of each other so we might have as well used $\Theta(1000)$ or $\Theta(0.0001)$ but we prefer the simplest form $\Theta(1)$.

The running time is determined by the output a and that is:

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i-1} 1}_{i-1}\Big) = \sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n}(n+1) - \sum_{i=1}^{n} i =$$

$$n(n+1) - \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$ (see Problem 4 on page 6.)

It follows that asymptotically ADD-4 has the same running time as ADD-3. Now consider a modification of ADD-4.

```
ADD-5(n: nonnegative integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i+1 to n
4          a ← a+1
5  return a
```

The running time is determined by the output a and that is:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}\Big(\underbrace{\sum_{j=1}^{n} 1}_{n} - \underbrace{\sum_{j=1}^{i} 1}_{i}\Big) = \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n}(n) - \sum_{i=1}^{n} i =$$

$$n^2 - \frac{n(n+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$$

Consider the following algorithm:

```
A2(n: positive integer)
1  a ← 0
2  for i ← 1 to n-1
3      for j ← i+1 to n
4          for k ← 1 to j
5              a ← a+1
6  return a
```

We are asked to determine a that A2 returns as a function of n. The answer clearly is

$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1$, we just need to find an equivalent closed form.

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n} j - \sum_{j=1}^{i} j\right) =$$
$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1) - \frac{1}{2}i(i+1)\right) =$$
$$\sum_{i=1}^{n-1}\left(\frac{1}{2}n(n+1)\right) - \frac{1}{2}\sum_{i=1}^{n-1}(i^2+i) =$$
$$\frac{1}{2}n(n+1)(n-1) - \frac{1}{2}\sum_{i=1}^{n-1} i^2 - \frac{1}{2}\sum_{i=1}^{n-1} i$$

But $\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1)$, therefore $\sum_{i=1}^{n-1} i^2 = \frac{1}{6}(n-1)n(2n-1)$. Further, $\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$, so we have

$$\frac{1}{2}n(n-1)(n+1) - \frac{1}{12}n(n-1)(2n-1) - \frac{1}{4}n(n-1) =$$
$$\frac{1}{2}n(n-1)\left(n+1-\frac{1}{6}(2n-1)-\frac{1}{2}\right) =$$
$$\frac{1}{12}n(n-1)(6n+3-2n+1) = \frac{1}{12}n(n-1)(4n+4) =$$
$$\frac{1}{3}n(n-1)(n+1)$$

That implies that the running time of A2 is $\Theta(n^3)$. Clearly A2 is equivalent to the following algorithm.

```
A3(n: positive integer)
1  return n(n-1)(n+1)/3
```

whose running time is $\Theta(1)$.

```
A4(n: positive integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i+1 to n
4          for k ← i+j-1 to n
5              a ← a+1
6  return a
```

Problem 34. *Find the running time of algorithm* A4 *by determining the value of* a *it returns as a function of* n, f(n). *Find a closed form for* f(n).

Solution:

$$f(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=i+j-1}^{n} 1$$

Let us evaluate the innermost sum $\sum_{k=i+j-1}^{n} 1$. It is easy to see that the lower boundary $i+j-1$ may exceed the higher boundary n. If that is the case, the sum is zero because the index variable takes values from the empty set. More precisely, for any integer t,

$$\sum_{i=t}^{n} 1 = \begin{cases} n-t+1 & \text{, if } t \le n \\ 0 & \text{, else} \end{cases}$$

It follows that

$$\sum_{k=i+j-1}^{n} 1 = \begin{cases} n-i-j+2 & \text{, if } i+j-1 \le n \Leftrightarrow j \le n-i+1 \\ 0 & \text{, else} \end{cases}$$

Then

$$f(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n-i+1} (n+2-(i+j))$$

Now the innermost sum is zero when $i+1 > n-i+1 \Leftrightarrow 2i > n \Leftrightarrow i > \lfloor \frac{n}{2} \rfloor$, therefore

the maximum i we have to consider is $\lfloor \frac{n}{2} \rfloor$:

$$f(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} (n+2-(i+j)) =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} 1 - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i \left(\sum_{j=i+1}^{n-i+1} 1 \right) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{j=i+1}^{n-i+1} j =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-i+1-(i+1)+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-i+1-(i+1)+1) -$$

$$\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\sum_{j=1}^{n-i+1} j - \sum_{j=1}^{i} j \right) =$$

$$(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (n-2i+1) - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i(n-2i+1) -$$

$$\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\frac{(n-i+1)(n-i+2)}{2} - \frac{i(i+1)}{2} \right) =$$

$$(n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - 2(n+2) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i - (n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 -$$

$$\frac{1}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left((n+1)(n+2) - i(2n+3) + \cancel{i^2} - \cancel{i^2} - i \right) =$$

$$(n+2)(n+1) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 - (3n+5) \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i + 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i^2 -$$

$$\frac{(n+1)(n+2)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 1 + \frac{(2n+4)}{2} \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} i =$$

$$\left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) - (3n+5) \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} + 2 \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{6} -$$

$$\frac{1}{2} \left\lfloor \frac{n}{2} \right\rfloor (n+1)(n+2) + (n+2) \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right)}{2} =$$

$$\frac{\lfloor \frac{n}{2} \rfloor (n+1)(n+2)}{2} - \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) (2n+3)}{2} + \frac{\lfloor \frac{n}{2} \rfloor \left(\lfloor \frac{n}{2} \rfloor + 1 \right) \left(2 \lfloor \frac{n}{2} \rfloor + 1 \right)}{3}$$

When n is even, *i.e.* $n = 2k$ for some $k \in \mathbb{N}^+$, $\left\lfloor \frac{n}{2} \right\rfloor = k$ and so

$$f(n) = \frac{k(2k+1)(2k+2)}{2} - \frac{k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$\frac{k(k+1)(4k+2) - k(k+1)(4k+3)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$k(k+1)\left(-\frac{1}{2} + \frac{2k+1}{3}\right) = \frac{k(k+1)(4k-1)}{6}$$

When n is odd, *i.e.* $n = 2k+1$ for some $k \in \mathbb{N}$, $\left\lfloor \frac{n}{2} \right\rfloor = k$ and so

$$f(n) = \frac{k(2k+2)(2k+3)}{2} - \frac{k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$\frac{k(k+1)(4k+6) - k(k+1)(4k+5)}{2} + \frac{k(k+1)(2k+1)}{3} =$$
$$k(k+1)\left(\frac{1}{2} + \frac{2k+1}{3}\right) = \frac{k(k+1)(4k+5)}{6}$$

Obviously, $f(n) = \Theta(n^3)$. □

```
A5(n: positive integer)
1  a ← 0
2  for i ← 1 to n
3      for j ← i to n
4          for k ← n+i+j−3 to n
5              a ← a+1
6  return a
```

Problem 35. *Find the running time of algorithm* A5 *by determining the value of* a *it returns as a function of* n, $f(n)$. *Find a closed form for* $f(n)$.

Solution:
We have three nested **for** cycles and it is certainly true that $f(n) = O(n^3)$. However, now $f(n) \neq \Theta(n^3)$. It is easy to see that for any large enough n, line 5 is executed for only three values of the ordered triple $\langle i, j, k\rangle$. Namely,

$$\langle i, j, k\rangle \in \{\langle 1, 1, n-1\rangle, \langle 1, 1, n\rangle, \langle 1, 2, n-1\rangle\}$$

because the condition in the innermost loop (line 5) requires that $i + j \leq 3$. So, $f(n) = 3$, thus $f(n) = \Theta(1)$. □
Problem 35 raises a question: does it make sense to compute the running time of an iterative algorithm by counting how many time the expression in the innermost loop is executed? At lines 2 and 3 of A5 there are condition evaluations and variable increments – can we assume they take no time at all? Certainly, if that was a segment of a real-world program, the outermost two loops would be executed $\Theta(n^2)$ times, unless some sort of optimisation

was applied by the compiler. Anyway, we *postulate* that the running time is evaluated by counting how many times the innermost loop is executed. Whether that is a realistic model for real-world computation or not, is a side issue.

```
A6(a_1, a_2, ... a_n: array of positive distinct integers, n ≥ 3)
 1  S: a stack of positive integers
 2  (* P(S) is a predicate that is evaluated in Θ(1) time. *)
 3  (* If there are less than two elements in S then P(S) is false. *)
 4  push(a_1, S)
 5  push(a_2, S)
 6  for i ← 3 to n
 7      while P(S) do
 8          pop(S)
 9      push(a_i, S)
```

Problem 36. *Find the asymptotic growth rate of running time of* A6. *Assume the predicate* P *is evaluated in* $\Theta(1)$ *time and the push and pop operations are executed in* $\Theta(1)$ *time.*

Solution:
Certainly, the running time is $O(n^2)$ because the outer loop runs $\Theta(n)$ times and the inner loop runs in $O(n)$ time: note that for each concrete i, the inner loop (line 8) cannot be executed more than $n-2$ times sinse there are at most n elements in S and each execution of line 8 removes one element from S.

However, a more precise analysis is possible. Observe that each element of the array is being pushed in S and *may be* popped out of S later but only once. It follows that line 8 cannot be exesuted more than n times altogether, *i.e.* for all i, and so the algorithm runs in $\Theta(n)$ time. □

```
A7(a_1, a_2, ... a_n: array of positive distinct integers, x: positive integer)
 1  i ← 1
 2  j ← n
 3  while i ≤ j do
 4      k ← ⌊(i+j)/2⌋
 5      if x = a_k
 6          return k
 7      else if x < a_k
 8          j ← k − 1
 9      else i ← k + 1
10  return −1
```

Problem 37. *Find the asymptotic growth rate of running time of* A7.

Solution:
The following is a loop invariant for A7:

For every iteration of the ***while*** *loop of* A7 *it is the case that:*

$$j - i + 1 \leq \frac{n}{2^t} \tag{2.1}$$

where the iteration number is t*, for some* $t \geq 0$*.*

We prove it by induction on t. The basis is $t = 0$, *i.e.* the first time the execution reaches line 3. Then j is n, i is 1, and indeed $n - 1 + 1 \leq \frac{n}{2^0}$ for all sufficiently large n. Assume that at iteration t, $t \geq 1$, (2.1) holds, and there is yet another iteration to go through. Ignore the possibility $x = a_k$ (line 5) because, if that is true then iteration $t+1$ never takes place. There are exactly two ways to get from iteration t to iteration $t+1$ and we consider them in separate cases.

Case I: the execution reaches line 8 Now j becomes $\left\lfloor \frac{i+j}{2} \right\rfloor - 1$ and i stays the same.

$$\begin{aligned}
\frac{j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}} && \text{divide (2.1) by 2}\\
\frac{j+i-2i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - i + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - i + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - 1 - i + \frac{1}{2} + 1 &\leq \frac{n}{2^{t+1}}\\
\frac{j+i}{2} - 1 - i + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \frac{1}{2} > 0\\
\underbrace{\left\lfloor \frac{j+i}{2} \right\rfloor - 1}_{\text{the new } j} - i + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \lfloor m \rfloor \leq m,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

And so the induction step follows from the induction hypothesis.

Case II: the execution reaches line 9 Now j stays the same and i becomes $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$.

$$\begin{aligned}
\frac{j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}} && \text{divide (2.1) by 2}\\
\frac{2j-j-i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
j - \frac{j+i}{2} + \frac{1}{2} &\leq \frac{n}{2^{t+1}}\\
j - \frac{i+j}{2} + \frac{1}{2} - 1 + 1 &\leq \frac{n}{2^{t+1}}\\
j - \frac{i+j}{2} - \frac{1}{2} + 1 &\leq \frac{n}{2^{t+1}}\\
j - \left(\frac{i+j}{2} + \frac{1}{2} \right) + 1 &\leq \frac{n}{2^{t+1}}\\
j - \underbrace{\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1 \right)}_{\text{the new } i} + 1 &\leq \frac{n}{2^{t+1}} && \text{since } \left\lfloor \frac{i+j}{2} \right\rfloor + 1 \geq \frac{i+j}{2} + \frac{1}{2},\ \forall i, j \in \mathbb{N}^+
\end{aligned}$$

And so the induction step follows from the induction hypothesis.

Having proven (2.1), we consider the maximum value that t reaches, call it t_{max}. During that last iteration of the loop, the values of i and j become equal, because the loop stops executing when $j < i$. Therefore, $j - i = 0$ during the execution of iteration t_{max}—before i gets incremented or j gets decremented. So, substituting t with t_{max} and $j - i$ with 0 in the invariant, we get $2^{t_{max}} \leq n \Leftrightarrow t_{max} < \lceil \lg n \rceil$. It follows that the running time of A7 is $O(\lg n)$.

The following claim is a loop invariant for A7:

> *For every iteration of the* ***while*** *loop of* A7*, if the iteration number is* t*,* $t \geq 0$*, it is the case that:*
>
> $$\frac{n}{2^{t+1}} - 4 < j - i \tag{2.2}$$

We prove it by induction on t. The basis is $t = 0$, *i.e.* the first time the execution reaches line 3. Then j is n, i is 1, and indeed $\frac{n}{2^{1+0}} - 4 = \frac{n}{2} - 4 < n - 1$, for all sufficiently large n. Assume that at iteration t, $t \geq 1$, (2.2) holds, and there is yet another iteration to go through. Ignore the possibility $x = a_k$ (line 5) because, if that is true then iteration $t + 1$ never takes place. There are exactly two ways to get from iteration t to iteration $t + 1$ and we consider them in separate cases.

Case I: the execution reaches line 8 Now j becomes $\left\lfloor \frac{i+j}{2} \right\rfloor - 1$ and i stays the same.

$$\begin{aligned}
&\frac{n}{2^{t+2}} - 2 < \frac{j-i}{2} \quad \text{divide (2.2) by 2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{j+i-2i}{2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{j+i}{2} - i \\
&\frac{n}{2^{t+2}} - 4 < \frac{j+i}{2} - 2 - i \\
&\frac{n}{2^{t+2}} - 4 < \underbrace{\left\lfloor \frac{j+i}{2} \right\rfloor - 1}_{\text{the new } j} - i \quad \text{since } m - 2 \leq \lfloor m \rfloor - 1,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

Case II: the execution reaches line 9 Now j stays the same and i becomes $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$.

$$\begin{aligned}
&\frac{n}{2^{t+2}} - 2 < \frac{j-i}{2} \\
&\frac{n}{2^{t+2}} - 2 < \frac{2j-j-i}{2} \\
&\frac{n}{2^{t+2}} - 2 < j - \frac{j+i}{2} \\
&\frac{n}{2^{t+2}} - 4 < j - \frac{j+i}{2} - 2 \\
&\frac{n}{2^{t+2}} - 4 < j - \left(\frac{j+i}{2} + 2 \right) \\
&\frac{n}{2^{t+2}} - 4 < j - \underbrace{\left(\left\lfloor \frac{j+i}{2} \right\rfloor + 1 \right)}_{\text{the new i}} \quad \text{since } m + 2 \geq \lfloor m \rfloor + 1,\ \forall m \in \mathbb{R}^+
\end{aligned}$$

Having proven (2.2), it is trivial to prove that in the worst case, *e.g.* when x is not in the array, the loop is executed $\Omega(\lg n)$ times. □

Problem 38. *Determine the asymptotic running time of the following programming segment:*

```
s = 0;
for(i = 1; i * i <= n; i ++)
  for(j = 1; j <= i; j ++)
   s += n + i - j;
return s;
```

Solution:
The segment is equivalent to:

```
s = 0;
for(i = 1; i <= floor(sqrt(n)); i ++)
  for(j = 1; j <= i; j ++)
   s += n + i - j;
return s;
```

As we already saw, the running time is $\Theta\left(\left(\sqrt{n}\right)^2\right)$ and that is $\Theta(n)$. □

Problem 39. *Assume that* $A_{n\times n}$, $B_{n\times n}$, *and* $C_{n\times n}$ *are matrices of integers. Determine the asymptotic running time of the following programming segment:*

```
for(i = 1; i <= n; i ++)
 for(j = 1; j <= n; j ++) {
  s = 0;
  for(k = 1; k <= n; k ++)
    s += A[i][k] * B[k][j];
  C[i][j] = s; }
return s;
```

Solution:
Having in mind the analysis of ADD-3 on page 25, clearly this is a $\Theta(n^3)$ algorithm. However, if consider the order of growth as a function of the *length of the input*, the order of growth is $\Theta\left(m^{\frac{3}{2}}\right)$, where m is the length of the input, *i.e.* m is the order of the number of elements in the matrices and $m = \Theta(n^2)$. □

```
A8(a_1, a_2, ... a_n: array of positive integers)
1  s ← 0
2  for i ← 1 to n − 4
3      for j ← i to i + 4
4          for k ← i to j
5              s ← s + a_i
```

Problem 40. *Determine the running time of algorithm* A8.

Solution:
The outermost loop is executed $n-4$ times (assume large enough n). The middle loop is executed 5 times precisely. The innermost loop is executed 1, 2, 3, 4, or 5 times for j equal to i, $i+1$, $i+2$, $i+3$, and $i+4$, respectively. Altogether, the running time is $\Theta(n)$. □

```
A9(n: positive integer)
1  s ← 0
2  for i ← 1 to n − 4
3      for j ← 1 to i + 4
4          for k ← i to j
5              s ← s + 1
6  return s
```

Problem 41. *Determine the running time of algorithm* A9. *First determine the value it returns as a function of* n.

Solution:
We have to evaluate the sum:

$$\sum_{i=1}^{n-4} \sum_{j=1}^{i+4} \sum_{k=i}^{j} 1$$

Having in mind that

$$\sum_{k=i}^{j} 1 = \begin{cases} j-i+1, & \text{if } j \geq i \\ 0, & \text{else} \end{cases}$$

we rewrite the sum as:

$$\sum_{i=1}^{n-4}\left(\underbrace{\sum_{j=1}^{i-1}\sum_{k=i}^{j}1}_{\text{this is }0}+\sum_{j=i}^{i+4}\sum_{k=i}^{j}1\right)=$$

$$\sum_{i=1}^{n-4}\sum_{j=i}^{i+4}(j-i+1)=$$

$$\sum_{i=1}^{n-4}\big((i-i+1)+(i+1-i+1)+(i+2-i+1)+(i+3-i+1)+(i+4-i+1)\big)=$$

$$\sum_{i=1}^{n-4}(1+2+3+4+5)=15(n-4)$$

So, algorithm A9 returns $15(n-4)$. The time complexity, though, is $\Omega(n^2)$ because the outer two loops require $\Omega(n^2)$ work. □

```
A10(n: positive integer)
1  a ← 0
2  for i ← 0 to n − 1
3      j ← 1
4      while j < 2n do
5          for k ← i to j
6              a ← a + 1
7          j ← j + 2
8  return a
```

Problem 42. *Find the running time of algorithm* A10 *by determining the value of* a *it returns as a function* $f(n)$ *of* n. *Find a closed form for* $f(n)$.

Solution:

$$f(n)=\sum_{i=0}^{n-1}\sum_{j\in\{1,3,5,\ldots,2n-1\}}\sum_{k=i}^{j}1$$

Let $n'=n-1$. Then

$$j\in\{1,3,5,\ldots,2n-1\}\Leftrightarrow j\in\{1,3,5,\ldots,2n'+1\}$$

But $\{1,3,5,\ldots,2n'+1\}=\{2\times 0+1,2\times 1+1,2\times 2+1,\ldots,2\times n'+1\}$. So we can rewrite the sum as:

$$f(n)=\sum_{i=0}^{n-1}\sum_{l=0}^{n-1}\sum_{k=i}^{2l+1}1$$

We know that

$$\sum_{k=i}^{2l+1}1=\begin{cases}2l+1-i+1, & \text{if } 2l+1\geq i\Leftrightarrow l\leq\left\lceil\frac{i-1}{2}\right\rceil\\ 0, & \text{otherwise}\end{cases}$$

Let $\left\lceil \frac{i-1}{2} \right\rceil$ be called i'. it must be case that

$$
\begin{aligned}
f(n) &= \sum_{i=0}^{n-1} \left(\sum_{l=0}^{i'-1} \underbrace{\sum_{k=i}^{2l+1} 1}_{0} \quad + \quad \sum_{l=i'}^{n-1} \underbrace{\sum_{k=i}^{2l+1}}_{2l+2-i} \right) \\
&= \sum_{i=0}^{n-1} \sum_{l=i'}^{n-1} (2l+2-i) \\
&= \sum_{i=0}^{n-1} \left((2-i) \sum_{l=i'}^{n-1} 1 + 2 \sum_{l=i'}^{n-1} l \right) = \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-1-i'+1) + 2 \sum_{l=i'}^{n-1} l \right) = \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-i') + 2 \sum_{l=i'}^{n-1} l \right) = \quad /\!/\!/ \; since \sum_{k=p}^{q} k = \frac{1}{2}(q+p)(q-p+1) \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-i') + 2 \times \frac{1}{2} \times (n-1+i')(n-1-i'+1) \right) = \\
&= \sum_{i=0}^{n-1} \left((2-i)(n-i') + (n-1+i')(n-i') \right) = \\
&= \sum_{i=0}^{n-1} \Big((n-i') \big((2-i) + (n-1+i') \big) \Big) = \\
&= \sum_{i=0}^{n-1} (n-i')(n+(-i+1+i')) \qquad (2.3)
\end{aligned}
$$

But

$$
\begin{aligned}
&-i+1+i' = -i+1+\left\lceil \frac{i-1}{2} \right\rceil = \\
&\left\lceil -i+1+\frac{i-1}{2} \right\rceil = \left\lceil \frac{-2i+2+i-1}{2} \right\rceil = \left\lceil \frac{-i+1}{2} \right\rceil = \left\lceil -\frac{i-1}{2} \right\rceil = -\left\lfloor \frac{i-1}{2} \right\rfloor
\end{aligned}
$$

since $\forall x \in \mathbb{R}, \lceil -x \rceil = -\lfloor x \rfloor$. Therefore, (2.3) equals

$$\sum_{i=0}^{n-1} \left(n - \left\lceil \frac{i-1}{2} \right\rceil \right) \left(n - \left\lfloor \frac{i-1}{2} \right\rfloor \right) =$$

$$\sum_{i=0}^{n-1} \left(n^2 - n \left\lfloor \frac{i-1}{2} \right\rfloor - n \left\lceil \frac{i-1}{2} \right\rceil + \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor \right) =$$

$$n^2 \sum_{i=0}^{n-1} 1 - n \sum_{i=0}^{n-1} \left(\underbrace{\left\lfloor \frac{i-1}{2} \right\rfloor + \left\lfloor \frac{i-1}{2} \right\rfloor}_{i-1} \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^2(n) - n \sum_{i=0}^{n-1} (i-1) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - n \left(\sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - n \left(\frac{(n-1)n}{2} - n \right) + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$n^3 - \frac{n^2(n-3)}{2} + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor =$$

$$\frac{n^3 + 3n^2}{2} + \sum_{i=0}^{n-1} \left\lceil \frac{i-1}{2} \right\rceil \left\lfloor \frac{i-1}{2} \right\rfloor \tag{2.4}$$

By (6.33) on page 174,

$$\sum_{i=0}^{n-1} \left\lfloor \frac{i-1}{2} \right\rfloor \left\lceil \frac{i-1}{2} \right\rceil = \begin{cases} \dfrac{(n-2)n(2n-5)}{24}, & n-1 \text{ odd } \Leftrightarrow n \text{ even} \\ \dfrac{(n-3)(n-1)(2n-1)}{24}, & n-1 \text{ even} \Leftrightarrow n \text{ odd} \end{cases}$$

I. Suppose n is even. Then

$$\begin{aligned} f(n) &= \frac{n^3 + 3n^2}{2} + \frac{(n-2)n(2n-5)}{24} \\ &= \frac{12(n^3 + 3n^2) + (2n^3 - 9n^2 + 10n)}{24} \\ &= \frac{14n^3 + 27n^2 + 10n}{24} \\ &= \frac{n(2n+1)(7n+10)}{24} \end{aligned}$$

II. Suppose n is odd. Then

$$\begin{aligned} f(n) &= \frac{n^3+3n^2}{2} + \frac{(n-3)(n-1)(2n-1)}{24} \\ &= \frac{12n^3+36n^2+2n^3-9n^2+10n-3}{24} \\ &= \frac{(n+1)(14n^2+13n-3)}{24} \end{aligned}$$

Obviously, $f(n) = \Theta(n^3)$ in either case. □

Asymptotics of bivariate functions

Our notations from Chapter 1 can be generalised for two variables as follows. A bivariate function $f(n,m)$ is asymptotically positive iff

$$\exists n_0 \exists m_0 : \forall n \geq n_0 \forall m \geq m_0, f(n,m) > 0$$

Definition 2. *Let* $g(n,m)$ *be an asymptotically positive function with real domain and codomain. Then*

$$\begin{aligned} \Theta(g(n,m)) = \big\{ f(n,m) \mid \exists c_1, c_2 > 0,\ \exists n_0, m_0 > 0 : \\ \forall n \geq n_0, \forall m \geq m_0,\ 0 \leq c_1.g(n,m) \leq f(n,m) \leq c_2.g(n,m) \big\} \end{aligned}$$ □

Pattern matching is a computational problem in which we are given a *text* and a *pattern* and we compute how many times or, in a more elaborate version, at what *shifts*, the pattern occurs in the text. More formally, we are given two arrays of characters $T[1..n]$ and $P[1..m]$, such that $n \geq m$. For any k, $1 \leq k \leq n-m+1$, we have a shift at position k iff:

$$\begin{aligned} T[k] &= P[1] \\ T[k+1] &= P[2] \\ &\dots \\ T[k+m-1] &= P[m] \end{aligned}$$

The problem then is to determine all the valid shifts. Consider the following algorithm for that problem.

```
NAIVE-PATTERN-MATHING(T[1..n]: characters, P[1..m]: characters)
1  (* assume n ≥ m *)
2  for i ← 1 to n − m + 1
3       if T[i, i + 1, ..., i + m − 1] = P
4            print "shift at" i
```

Problem 43. *Determine the running time of algorithm* NAIVE-PATTERN-MATHING.

Solution:
The algorithm is ostensibly $\Theta(n)$ because it has a single loop with the loop control variable running from 1 to n. That analysis, however, is wrong because the comparison at line 3 cannot be performed in constant time. Have in mind that m can be as large as n. Therefore, the algorihm is in fact:

```
NAIVE-PATTERN-MATHING-1(T[1..n]: characters, P[1..m]: characters)
 1  (* assume n ≥ m *)
 2  for i ← 1 to n − m + 1
 3      Match ← TRUE
 4      for j ← 1 to m
 5          if T[i + j − 1] ≠ P[j]
 6              Match ← FALSE
 7      if Match
 8          print "shift at" i
```

For obvious reasons this is a $\Theta((n-m).m)$ algorithm: both the best-case and the worst-case running times are $\Theta((n-m).m)$[†]. Suppose we improve it to:

```
NAIVE-PATTERN-MATHING-2(T[1..n]: characters, P[1..m]: characters)
 1  (* assume n ≥ m *)
 2  for i ← 1 to n − m + 1
 3      Match ← TRUE
 4      j ← 1
 5      while Match AND j ≤ m do
 6          if T[i + j − 1] = P[j]
 7              j ← j + 1
 8          else
 9              Match ← FALSE
10      if Match
11          print "shift at" i
```

NAIVE-PATTERN-MATHING-2 has the advantage that once a mismatch is found (line 9) the inner loop "breaks". Thus the best-case running time is $\Theta(n)$. A best case, for instance, is:

$$T = \underbrace{a\,a\,\ldots\,a}_{n \text{ times}} \quad \text{and} \quad P = \underbrace{b\,b\,\ldots\,b}_{m \text{ times}}$$

However, the worst case running time is still $\Theta((n-m).m)$. A worst case is, for instance:

$$T = \underbrace{a\,a\,\ldots\,a}_{n \text{ times}} \quad \text{and} \quad P = \underbrace{a\,a\,\ldots\,a}_{m \text{ times}}$$

It is easy to prove that $(n-m).m$ is maximised when m varies and n is fixed for $m \approx \frac{n}{2}$ and achieves maximum value $\Theta(n^2)$. It follows that all the naive string matchings are, at worst, quadratic algorithms. □

[†]Algorithms that have the same—in asymptotic terms—running time for all inputs of the same length are called *oblivious*.

It is known that faster algorithms exist for the pattern matching problem. For instance, the Knuth-Morris-Pratt [KMP77] algorithm that runs in $\Theta(n)$ in the worst case.

Problem 44. *For any two strings* x *and* y *of the same length, we say that* x is a circular shift of y *iff* y *can be broken into substrings—one of them possibly empty—*y_1 *and* y_2*:*

$$y = y_1 y_2$$

such that $x = y_2 y_1$. *Find a linear time algorithm,* i.e. $\Theta(n)$ *in the worst case, that computes whether* x *is a circular shift of* y *or not. Assume that* $x \neq y$.

Solution:
Run the linear time algorithm for string matching of Knuth-Morris-Pratt with input $y\,y$ (y concatenated with itself) as text and x as pattern. The algorithm will output one or more valid shifts iff x is a circular shift of y, and zero valid shifts, otherwise. To see why, consider the concatenation of y with itself when it is a circular shift of x for some y_1 and y_2, such that $y = y_1 y_2$ and $x = y_2 y_1$:

$$y \quad y \quad = \quad y_1 \quad \underbrace{y_2 \quad y_1}_{\text{this is } x} \quad y_2$$

The running time is $\Theta(2n)$, *i.e.* $\Theta(n)$, at worst. □

Chapter 3

Recursive Algorithms and Recurrence Relations

3.1 Preliminaries

A *recursive algorithm* is an algorithm that calls itself, one or more times on smaller inputs. To prevent an infinite chain of such calls there has to be a value of the input for which the algorithm does not call itself.

A *recurrence relation in one variable* is an equation, *i.e.* there is an "=" sign "in the middle", in which a function of the variable is equated to an expression that includes the same function on smaller value of the variable. In addition to that for some basic value of the variable, typically one or zero, an explicit value for the function is defined – that is the initial condition[†]. The variable is considered by default to take nonnegative integer values, although one can think of perfectly valid recurrence relations in which the variable is real.

Typically, in the part of the relation that is not the initial condition, the function of the variable is written on the left-hand side of the "=" sign as, say, $T(n)$, and the expression, on the right-hand side, *e.g.* $T(n) = T(n-1) + 1$. If the initial condition is, say, $T(0) = 0$, we typically write:

$$\begin{aligned} &T(n) = T(n-1) + 1,\ \forall n \in \mathbb{N}^+ \\ &T(0) = 0 \end{aligned} \tag{3.1}$$

It is not formally incorrect to write the same thing as:

$$\begin{aligned} &T(n-1) = T(n-2) + 1,\ \forall n \in \mathbb{N}^+, n \neq 1 \\ &\quad T(0) = 0 \end{aligned}$$

The equal sign is interpreted as an assignment from right to left, just as the equal sign in the C programming language, so the following "unorthodox" way of describing the same

[†]Note there can be more than one initial condition as in the case with the famous Fibonacci numbers:

$$\begin{aligned} &F(n) = F(n-1) + F(n-2),\ \forall n \in \mathbb{N}^+, n \neq 1 \\ &F(1) = 1 \\ &F(0) = 0 \end{aligned}$$

The number of initial conditions is such that the initial conditions prevent "infinite descent".

relation is *discouraged*:

$$T(n-1)+1 = T(n),\ \forall n \in \mathbb{N}^+$$
$$0 = T(0)$$

Each recurrence relation defines an infinite numerical sequence, provided the variable is integer. For example, (3.1) defines the sequence $0, 1, 2, 3, \ldots$. Each term of the relation, except for the terms defined by the initial conditions, is defined recursively, *i.e.* in terms of smaller terms, hence the name. To *solve* a recurrence relation means to find a non-recursive expression for the same function – one that defines the same sequence. For example, the solution of (3.1) is $T(n) = n$.

It is natural to describe the running time of a recursive algorithm by some recurrence relation. However, since we are interested in asymptotic running times, we do not need the precise solution of a "normal" recurrence relation as described above. A normal recurrence relation defines a sequence of numbers. If the time complexity of an algorithm as a worst-case analysis was given by a normal recurrence relation then the number sequence a_1, a_2, a_3, ..., defined by that relation, would describe the running time of algorithm precisely, *i.e.* for input of size n, the maximum number of steps the algorithm makes over all inputs of size n is precisely a_n. We do not need such a precise analysis and often it is impossible to derive one. So, the recurrence relations we use when analysing an algorithm typically have bases $\Theta(1)$, for example:

$$T(n) = T(n-1)+1,\ n \geq 2 \tag{3.2}$$
$$T(1) = \Theta(1)$$

Infinitely many number sequences are solutions to (3.2). To solve such a recurrence relation means to find the asymptotic growth of any of those sequences. The best solution we can hope for, asymptotically, is the one given by the Θ notation. If we are unable to pin down the asymptotic growth in that sense, our second best option is to find functions $f(n)$ and $g(n)$, such that $f(n) = o(g(n))$ and $T(n) = \Omega(f(n))$ and $T(n) = O(g(n))$. The best solution for the recurrence relation (3.2), in the asymptotic sense, is $T(n) = \Theta(n)$. Another solution, not as good as this one, is, for example, $T(n) = \Omega(\sqrt{n})$ and $T(n) = O(n^2)$.

In the problems that follow, we distinguish the two types of recurrence relation by the initial conditions. If the initial condition is given by a precise expression as in (3.1) we have to give a precise answer such as $T(n) = n$, and if the initial condition is $\Theta(1)$ as in (3.2) we want only the growth rate.

It is possible to omit the initial condition altogether in the description of the recurrence. If we do so we assume tacitly the initial condition is $T(c) = \Theta(1)$ for some positive constant c. The reason to do that may be that it is pointless to specify the usual $T(1)$; however, it may be the case that the variable never reaches value one. For instance, consider the recurrence relation

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor + 17\right) + n$$

which we solve below (Problem 51 on page 57). To specify "$T(1) = \Theta(1)$" for it is *wrong*.

3.1.1 Iterators

The recurrence relations can be partitioned into the following two classes, assuming T is the function of the recurrence relations as above.

1. The ones in which T appears only once on the right-hand side as in (3.1).
2. The ones in which T appears mutiple times on the right-hand side, for instance:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + n \tag{3.3}$$

We will call them relations with *single occurrence* and with *multiple occurrences*, respectively. We find it helpful to make that distinction because in general only the relations with single occurrence are ameaneable to the method of unfolding (see below). If the relation is with single occurrence we define *the iterator* of the relation as the iterative expression that shows how the variable decreases. For example, the iterator of (3.1) is:

$$n \to n-1 \tag{3.4}$$

It is not practical to define iterators for relations with multiple occurrences. If we wanted to define iterators for them as well, they would have a set of functions on the right-hand side, for instance the iterator of (3.3) would be

$$n \to \{n-1, n-2, n-3\}$$

and that does help the analysis of the relation. So, we define iterators only for relations with single occurrence. The iterators that are easiest to deal with (and, fortunately, occur often in practice) are the ones in which the function on the right-hand side is subtraction or division (by constant > 1):

$$n \to n - c,\ c > 0 \tag{3.5}$$

$$n \to \frac{n}{b},\ b > 1 \tag{3.6}$$

Another possibility is that function to be some root of n:

$$n \to \sqrt[d]{n},\ d > 1 \tag{3.7}$$

Note that the direction of the assignment in the iterator is the opposite to the one in the recurrence relation (compare (3.1) with (3.4)). The reason is that a recurrence has to phases: descending and ascending. In the descending phase we start with some value n for the variable and decrease it in successive steps till we reach the initial condition; in the ascending phase we go back from the initial condition "upwards". The left-to-right direction of the iterator refers to the descending phase, while the right-to-left direction of the assignment in the recurrence refers to the ascending phase.

It is important to be able to estimate the number of times an iterator will be executed before its variable becomes 1 (or whatever value the initial conditions specify). If the variable n is integer, the iterator $n \to n-1$ is the most basic one we can possibly have. The number of times it is executed before n becomes any *a priori* fixed constant is $\Theta(n)$. That has to be obvious. Now consider (3.5). We ask the same question: how many times it is executed before n becomes a constant. Substitute n by cm and (3.5) becomes:

$$cm \to c(m-1) \tag{3.8}$$

The number of times (3.8) is executed (before m becomes a constant) is $\Theta(m)$. Since $m = \Theta(n)$, we conclude that (3.5) is executed $\Theta(n)$ times.

Consider the iterator (3.6). To see how many times it is executed before n becomes a constant (fixed *a priori*)) can be estimated as follows. Substitute n by b^m and (3.6) becomes

$$b^m \to b^{m-1} \tag{3.9}$$

(3.9) is executed $\Theta(m)$ times because $m \to m-1$ is executed $\Theta(m)$ times. Since $m = \log_b n$, we conclude that (3.6) is executed $\Theta(\log_b n)$ times, *i.e.* $\Theta(\lg n)$ times. We see that the concrete value of b is immaterial with respect to the asymptotics of the number of executions, provided $b > 1$.

Now consider (3.7). To see how many times it is executed before n becomes a constant, substitute n by d^{d^m}. (3.7) becomes

$$d^{d^m} \to d^{\frac{d^m}{d}} = d^{d^{m-1}} \tag{3.10}$$

(3.10) is executed $\Theta(m)$ times. As $m = \log_d \log_d n$, we conclude that (3.7) is executed $\Theta(\log_d \log_d n)$ times, *i.e.* $\Theta(\lg \lg n)$ times. Again we see that the value of the constant in the iterator, namely d, is immaterial as long as $d > 1$.

Let us consider an iterator that decreases even faster than (3.7):

$$n \to \lg n \tag{3.11}$$

The number of times it is executed before n becomes a constant is $\lg^* n$, which follows right from Definion 1 on page 14.

Let us summarise the rates of decrease of the iterators we just considered assuming the mentioned "constants of decrease" b and d are 2.

iterator	*asymptotics of the number executions*	*alternative form (see Definion 1)*
$n \to n-1$	n	$\lg^{(0)} n$
$n \to n/2$	$\lg n$	$\lg^{(1)} n$
$n \to \sqrt{n}$	$\lg \lg n$	$\lg^{(2)} n$
$n \to \lg n$	$\lg^* n$	$\lg^* n$

There is a gap in the table. One would ask, what is the function $f(n)$, such that the iterator $n \to f(n)$ is executed, asymptotically, $\lg \lg \lg n$ times, *i.e.* $\lg^{(3)} n$ times. To answer that question, consider that $f(n)$ has to be such that if we substitute n by 2^m, the number of executions is the same as in the iterator $m \to \sqrt{m}$. But $m \to \sqrt{m}$ is the same as $\lg n \to \sqrt{\lg n}$, *i.e.* $n \to 2^{\sqrt{\lg n}}$. We conclude that $f(n) = 2^{\sqrt{\lg n}}$. To check this, consider the iterator

$$n \to 2^{\sqrt{\lg n}} \tag{3.12}$$

Substitute n by $2^{2^{2^m}}$ in (3.12) to obtain:

$$2^{2^{2^m}} \to 2^{\sqrt{\lg 2^{2^{2^m}}}} = 2^{\sqrt{2^{2^m}}} = 2^{2^{\frac{2^m}{2}}} = 2^{2^{2^{m-1}}} \tag{3.13}$$

Clearly, (3.13) is executed $m = \lg\lg\lg n = \lg^{(3)} n$ times.

A further natural question is, what the function $\phi(n)$ is, such that the iterator $n \to \phi(n)$ is executed $\lg^{(4)} n$ times. Applying the reasoning we used to derive $f(n)$, $\phi(n)$ has to be such that if we substitute n by 2^m, the number of executions is the same as in $m \to 2^{\sqrt{\lg m}}$. As $m = \lg n$, the latter becomes $\lg n \to 2^{\sqrt{\lg\lg n}}$, *i.e.* $n \to 2^{2^{\sqrt{\lg\lg n}}}$. So, $\phi(n) = 2^{2^{\sqrt{\lg\lg n}}}$. We can fill in two more rows in the table:

iterator	*asymptotics of the number executions*	*alternative form (see Definion 1)*
$n \to n-1$	n	$\lg^{(0)} n$
$n \to n/2$	$\lg n$	$\lg^{(1)} n$
$n \to \sqrt{n}$	$\lg\lg n$	$\lg^{(2)} n$
$n \to 2^{\sqrt{\lg n}}$	$\lg\lg\lg n$	$\lg^{(3)} n$
$n \to 2^{2^{\sqrt{\lg\lg n}}}$	$\lg\lg\lg\lg n$	$\lg^{(4)} n$
$n \to \lg n$	$\lg^* n$	$\lg^* n$

Let us define, analogously to Definion 1, the function base-two iterated exponent.

Definition 3 (iterated exponent)**.** *Let* i *be a nonnegative integer.*

$$\text{itexp}^{(i)}(n) = \begin{cases} n, & \text{if } i = 0 \\ 2^{\text{itexp}^{(i-1)}(n)}, & \text{if } i > 0 \end{cases}$$ □

Having in mind the results in the table, we conjecture, and it should not be too difficult to prove by induction, that the iterator:

$$n \to \text{itexp}^{(k)}\left(\sqrt{\lg^{(k)} n}\right) \tag{3.14}$$

is executed $\lg^{(k+2)} n$ times for $k \in \mathbb{N}$.

3.1.2 Recursion trees

Assume we are given a recurrence relation of the form:

$$T(n) = k_1 T(f_1(n)) + k_2 T(f_2(n)) + \ldots + k_p T(f_p(n)) + \phi(n) \tag{3.15}$$

where k_i, $i \le i \le p$ are positive integer constants, $f_i(n)$ for $1 \le i \le p$ are integer-valued functions such that $n > f(n)$ for all $n \ge n_0$ where n_0 is the largest (constant) value of the argument in any initial condition, and $\phi(n)$ is some positive function. It is not necessary $\phi(n)$ to be positive as the reader will see below; however, if $T(n)$ describes the running time of a recursive algorithm then $\phi(n)$ has to be positive. We build a special kind of rooted tree that corresponds to our recurrence relation. Each node of the tree corresponds to one particular value of the variable that appears in the process of unfolding the relation, the value that corresponds to the root being n. That value we call *the level* of the node. Further, with each node we associate $\phi(m)$ where m is the level of that node. We call that, *the cost* of the node. Further, each node—as long as no initial condition has been reached

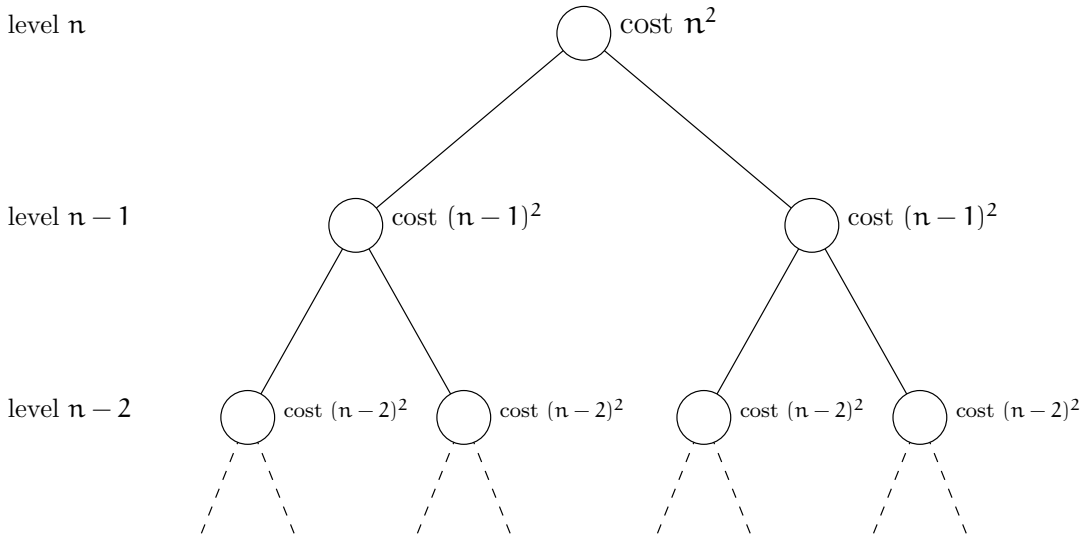


Figure 3.1: The recursion tree of $T(n) = 2T(n-1) + n^2$.

yet—has $k_1 + k_2 + \ldots + k_p$ children, k_i of them being at level defined by f_i for $1 \leq i \leq p$. For example, if our recurrence is

$$T(n) = 2T(n-1) + n^2$$

the recursion tree is as shown on Figure 3.1. It is a complete binary tree. It is binary because there are two invocations on the right side, *i.e.* $k_1 + k_2 + \ldots + k_p = 2$ in the above terminology. And it is complete because it is a recurrence with a single occurrence. Note that if $k_1 + k_2 + \ldots + k_p$ equals 1 then the recursion tree degenerates into a path.

The size of the tree depends on n so we can not draw the whole tree. The figure is rather a suggestion about it. The bottom part of the tree is missing because we have not mentioned the initial conditions. The solution of the recursion—and that is the goal of the tree, to help us solve the recursion—is the total sum of all the costs. Typically we sum by levels, so in the current example the sum will be

$$n^2 + 2(n-1)^2 + 4(n-2)^2 + \ldots$$

The general term of this sum is $2^k(n-k)^2$. The "..." notation hides what happens at the right end, however, we agreed the initial condition is for some, it does not matter, what constant value of the variable. Therefore, the sum

$$\sum_{k=0}^{n} 2^k(n-k)^2$$

has the same growth rate as our desired solution. Let us find a closed form for that sum.

$$\sum_{k=0}^{n} 2^k(n-k)^2 = n^2 \sum_{k=0}^{n} 2^k - 2n \sum_{k=0}^{n} 2^k k + \sum_{k=0}^{n} 2^k k^2$$

Having in mind Problem 122 on page 158 and Problem 123 on page 158, that expression

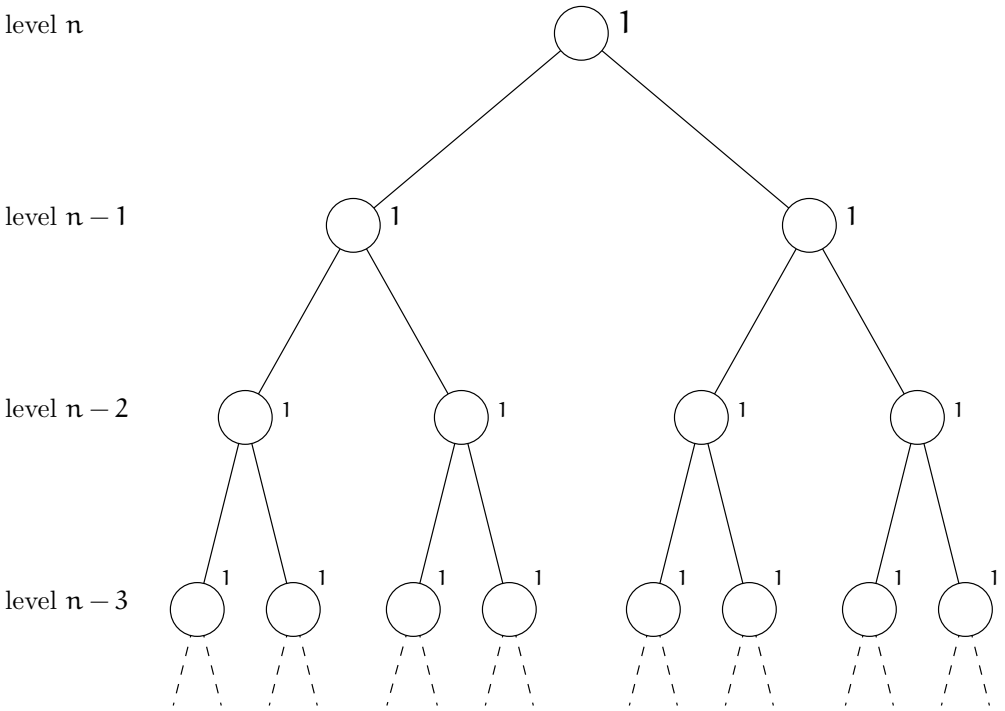


Figure 3.2: The recursion tree of $T(n) = 2T(n-1) + 1$.

becomes

$$n^2(2^{n+1} - 1) - 2n((n-1)2^{n+1} + 2) + n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$n^2.2^{n+1} - n^2 - 2n(n.2^{n+1} - 2^{n+1} + 2) + n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$2.n^2.2^{n+1} - n^2 - 2.n^2.2^{n+1} + 2n.2^{n+1} - 4n - 2n2^{n+1} + 4.2^{n+1} - 6 =$$
$$4.2^{n+1} - n^2 - 4n - 6$$

It follows that $T(n) = \Theta(2^n)$.

The correspondence between a recurrence relation and its recursion tree is not necessarily one-to-one. Consider the recurrence relation

$$\begin{aligned} T(n) &= 2T(n-1) + 1,\ n \geq 2 \\ T(1) &= \Theta(1) \end{aligned} \tag{3.16}$$

and its recursion tree (Figure 3.2). The cost at level n is 1, at level $n-1$ is 2, at level $n-2$ is 4, at level $n-3$ is 8, *etc.* The tree is obviously complete. Let us now rewrite (3.2) as follows.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \ \Leftrightarrow\ T(n) = T(n-1) + T(n-1) + 1 \\ T(n-1) &= 2T(n-2) + 1 \\ T(n) &= T(n-1) + 2T(n-2) + 2 \end{aligned}$$

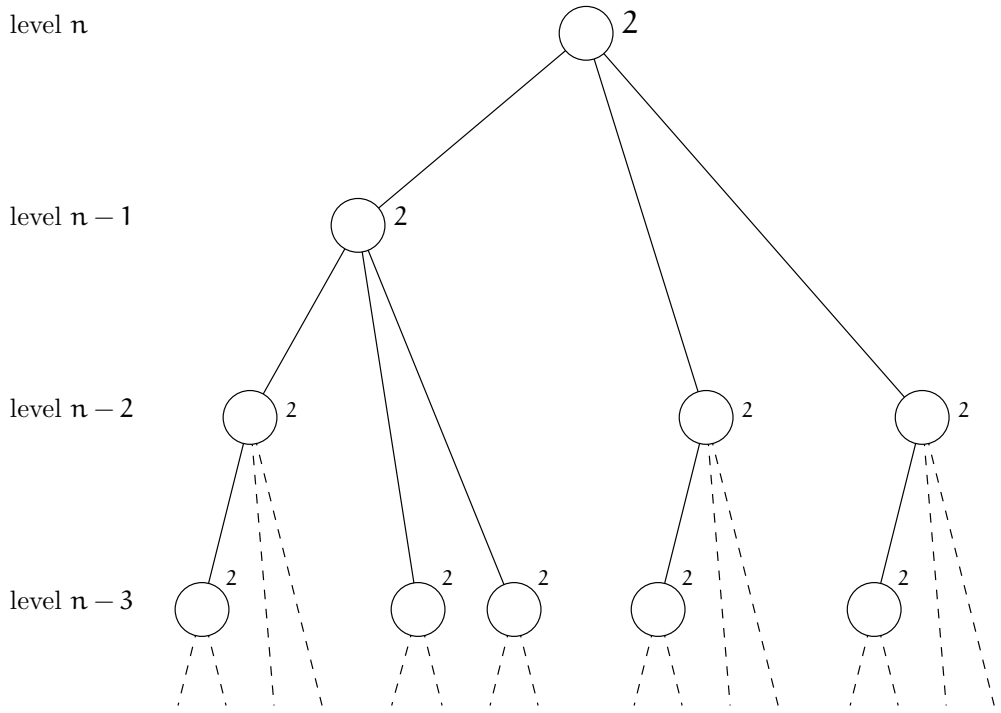


Figure 3.3: The recursion tree of $T(n) = T(n-1) + 2T(n-2) + 2$.

We have to alter the initial conditions for this rewrite, adding $T(2) = 3$. Overall the recurrence becomes

$$\begin{aligned} &T(n) = T(n-1) + 2T(n-2) + 2 \\ &T(2) = 3 \\ &T(1) = 1 \end{aligned} \tag{3.17}$$

Recurrences (3.2) and (3.17) are equivalent. One can say these are different ways of writing down the same recurrence because both of them define one and the same sequence, namely $1, 3, 7, 15, \ldots$ However, their recursion trees are neither the same nor isomorphic. Figure 3.3 shows the tree of (3.17). To give a more specific example, Figure 3.4 shows the recursion tree of (3.17) for $n = 5$. It shows the whole tree, not just the top, because the variable has a concrete value. Therefore the initial conditions are taken into account. The reader can easily see the total sum of the costs over the tree from Figure 3.4 is 31, the same as the tree from Figure 3.2 for $n = 5$. However, the sum 31 on Figure 3.2 is obtained as $1+2+4+8+16$, if we sum by levels. In the case with Figure 3.4 we do not have obvious definition of levels.

- If we define the levels as the vertices that have the same value of the variable, we have 5 levels and the sum is derived, level-wise, as $2 + 2 + 6 + 15 + 6 = 31$.
- If we define the levels as the vertices that are at the same distance to the root, we have only 4 levels and the sum is derived, level-wise, as $2 + 6 + 18 + 5 = 31$.

Regardless of how we define the levels, the derivation is not $1 + 2 + 4 + 8 + 16$.

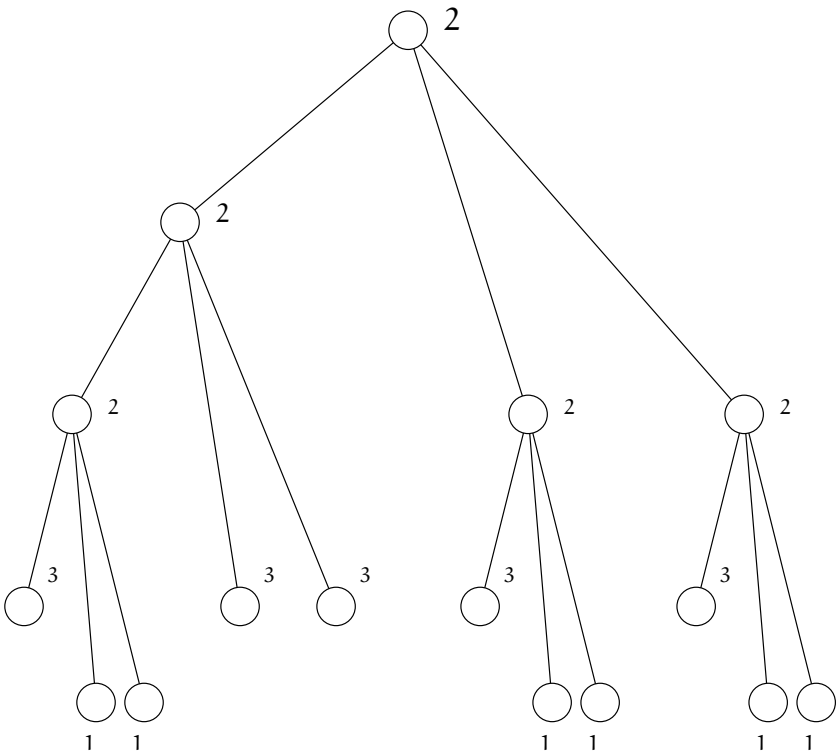


Figure 3.4: The recursion tree of $T(n) = T(n-1) + 2T(n-2) + 2$, $T(2) = 3$, $T(1) = 1$, for $n = 5$.

3.2 Problems

Our repertoire of methods for solving recurrences is:

- by induction,
- by unfolding,
- by considering the recursion tree,
- by the Master Theorem, and
- by the method of the characteristic equation.

3.2.1 Induction, unfolding, recursion trees

Problem 45. *Solve*

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ T(0) &= 0 \end{aligned} \tag{3.18}$$

Solution:
We guess that $T(n) = 2^n - 1$ for all $n \geq 1$ and prove it by induction on n.

Basis: $n = 1$. We have $T(1) = 2T(0) + 1$ by substituting n with 1. But $T(0) = 0$, thus $T(1) = 2 \times 0 + 1 = 1$. On the other hand, substituting n with 1 in our guessed solution, we have $2^1 - 1 = 1$.

Inductive hypothesis: assume $T(n) = 2^n - 1$ for some $n > 1$.

Inductive step: $T(n+1) = 2T(n)+1$ by definition. Apply the inductive hypothesis to obtain $T(n+1) = 2(2^n - 1) + 1 = 2^{n+1} - 1$. □

The proofs by induction have one major drawback – making a good guess can be a form of art. There is no recipe, no algorithm for making a good guess in general. It makes sense to compute several initial values of the sequence defined by the recurrence and try to see a pattern in them. In the last problem, $T(1) = 1$, $T(2) = 3$, $T(3) = 7$ and it is reasonable to assume that $T(n)$ is $2^n - 1$. Actually, if we think about (3.18) in terms of the binary representation of $T(n)$, it is pretty easy to spot that (3.18) performs a shift-left by one position and then turns the least significant bit from 0 into 1. As we start with $T(1) = 1$, clearly

$$T(n) = \underbrace{111\ldots1}_{n \text{ times}}{}_b$$

For more complicated recurrence relations, however, seeing a pattern in the initial values of the sequence, and thus making a good guess, can be quite challenging. If one fails to see such a pattern it is a good idea to check if these numbers are found in *The On-Line Encyclopedia of Integer Sequences* [Slo]. Of course, this advice is applicable when we solve precise recurrence relations, not asymptotic ones.

Problem 46. *Solve by unfolding*

$$\begin{aligned} T(n) &= T(n-1) + n \\ T(0) &= 1 \end{aligned} \tag{3.19}$$

Solution:
By unfolding (also called unwinding) of the recurrence down to the initial condition.

$$\begin{aligned} T(n) &= T(n-1) + n \quad \text{directly from (3.19)} \\ &= T(n-2) + n - 1 + n \quad \text{substitute } n \text{ with } n-1 \text{ in (3.19)} \\ &= T(n-3) + n - 2 + n - 1 + n \quad \text{substitute } n-1 \text{ with } n-2 \text{ in (3.19)} \\ &\ldots \\ &= T(0) + 1 + 2 + 3 + \ldots + n - 2 + n - 1 + n = \\ &= 1 + 1 + 2 + 3 + \ldots + n - 2 + n - 1 + n = \\ &= 1 + \frac{n(n+1)}{2} \end{aligned}$$

This method is considered to be not as formally precise as the induction. The reason is that we inevitably skip part of the derivation—the dot-dot-dot "..." part—leaving it to the imagination of the reader to verify the derived closed formula. Problem 46 is trivially simple and it is certain beyond any doubt that if we start with $T(n-3) + n - 2 + n - 1 + n$ and systematically unfold $T(i)$, decrementing by one values of i, eventually we will "hit"

the initial condition $T(0)$ and the "tail" will be $1+2+3+\ldots+n-2+n-1+n$. The more complicated the expression is, however, the more we leave to the imagination of the reader when unfolding.

One way out of that is to use the unfolding to derive a closed formula and then prove it by induction. □

Problem 47. *Solve*

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \tag{3.20}$$

$$T(1) = \Theta(1) \tag{3.21}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$.

Part I: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.22}$$

There is a potential problem with the initial condition because for $n = 1$ the right-hand side of (3.22) becomes $c.1. \lg 1 = 0$, and $0 \neq \Theta(1)$. However, it is easy to deal with that issue, just do not take $n = 1$ as basis. Taking $n = 2$ as basis works as $c.2. \lg 2$ is not zero. However, note that $n = 2$ is not sufficient basis! There are certain values for n, for example 3, such that the iterator of this recurrence, namely

$$n \to \left\lfloor \frac{n}{2} \right\rfloor$$

"jumps over" 2, having started from one of them. Indeed, $\left\lfloor \frac{3}{2} \right\rfloor = 1$, therefore the iterator, starting from 3, does

$$3 \to 1$$

and then goes infinite descent. The solution is to take two bases, for both $n = 2$ and $n = 3$. It is certain that no matter what n is the starting one, the iterator will at one moment "hit" either 2 or 3. So, the bases of our proof are:

$$T(2) = \Theta(1) \tag{3.23}$$

$$T(3) = \Theta(1) \tag{3.24}$$

Of course, that does not say that $T(2) = T(3)$, it says there exist constants c_2 and c_3, such that:

$$c_2 \leq c.2 \lg 2$$
$$c_3 \leq c.3 \lg 3$$

Our induction hypothesis is that relative to some sufficiently large n, (3.22) holds for some positive constant c all values of the variable between 3 and n, excluding n. The induction step is to prove (3.22), using the hypothesis. So,

$$\begin{aligned}
T(n) &= 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n \qquad \text{this is the defintion of } T(n)\\
&\le 2.c.\left\lfloor\frac{n}{2}\right\rfloor \lg\left\lfloor\frac{n}{2}\right\rfloor + n \qquad \text{from the inductive hypothesis}\\
&\le 2.c.\frac{n}{2}\lg\frac{n}{2} + n\\
&= cn(\lg n - 1) + n\\
&= cn\lg n + (1-c)n \qquad (3.25)\\
&\le cn\lg n \qquad \text{provided that } (1-c) \le 0 \;\Leftrightarrow\; c \ge 1 \qquad (3.26)
\end{aligned}$$

If $c \ge 1$, the proof is valid. If we want to be perfectly precise we have to consider the two bases as well to find a value for c that works. Namely,

$$c = \max\left\{1, \frac{c_2}{2\lg 2}, \frac{c_3}{3\lg 3}\right\}$$

In our proofs from now on we will not consider the initial conditions when choosing an appropriate constant.

Part II: Proof that $T(n) = \Omega(n\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \ge n_1$,

$$T(n) \ge dn\lg n \qquad (3.27)$$

We will ignore the basis of the induction and focus on the hypothesis and the inductive step only. Applying the inductive hypothesis to (3.27), we get:

$$\begin{aligned}
T(n) &\ge 2d\left\lfloor\frac{n}{2}\right\rfloor \lg\left\lfloor\frac{n}{2}\right\rfloor + n \qquad \text{from the inductive hypothesis}\\
&\ge 2d\left(\frac{n}{2} - 1\right)\lg\left\lfloor\frac{n}{2}\right\rfloor + n\\
&= d(n-2)\lg\left\lfloor\frac{n}{2}\right\rfloor + n\\
&\ge d(n-2)\lg\left(\frac{n}{4}\right) + n\\
&= d(n-2)\left(\lg n - 2\right) + n\\
&= dn\lg n + n(1-2d) - 2d\lg n + 4d\\
&\ge dn\lg n \qquad \text{provided that } n(1-2d) - 2d\lg n + 4d \ge 0
\end{aligned}$$

So (3.27) holds when

$$n(1-2d) - 2d\lg n + 4d \ge 0 \qquad (3.28)$$

Observe that for $d = \frac{1}{4}$ inequality (3.28) becomes

$$\frac{n}{2} + 1 \ge \frac{1}{2}\lg n$$

It certainly holds $\forall n \ge 2$, therefore the choice $d = \frac{1}{4}$ and $n_1 = 2$ suffices for our proof. □

Problem 48. *Solve*

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \tag{3.29}$$

$$T(1) = \Theta(1) \tag{3.30}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$. We ignore the basis of the induction – the solution of Problem 47 gives us enough confidence that we can handle the basis if we wanted to.

Part I: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.31}$$

From the inductive hypothesis

$$\begin{aligned}
T(n) &\leq 2.c. \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\
&\leq 2.c. \left(\frac{n}{2} + 1\right) \lg \left\lceil \frac{n}{2} \right\rceil + n && (3.32) \\
&\leq 2.c. \left(\frac{n}{2} + 1\right) \lg \left(\frac{3n}{4}\right) + n \qquad \text{because } \frac{3n}{4} \geq \left\lceil \frac{n}{2} \right\rceil \ \forall n \geq 2 && (3.33) \\
&= c(n+2)(\lg n + \lg 3 - 2) + n \\
&= cn \lg n + cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \\
&\leq cn \lg n \qquad \text{if } \ cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n \leq 0
\end{aligned}$$

Consider

$$cn(\lg 3 - 2) + 2c \lg n + 2c(\lg 3 - 2) + n = (c(\lg 3 - 2) + 1)n + 2c \lg n + 2c(\lg 3 - 2)$$

Its asymptotic growth rate is determined by the linear term. If the constant $c(\lg 3 - 2) + 1$ is negative then the whole expression is certainly negative for all sufficiently large values of n. In other words, for the sake of brevity we do not specify precisely what n_0 is. In order to have $c(\lg 3 - 2) + 1 < 0$ it must be the case that $c > \frac{1}{2 - \lg 3}$. So, any $c > \frac{1}{2 - \lg 3}$ works for our proof.

⁘ NB ⁘ *In (3.32) we substitute $\left\lceil \frac{n}{2} \right\rceil$ with $\frac{3n}{4}$. We could have used any other fraction $\frac{pn}{q}$, provided that $\frac{1}{2} < \frac{p}{q} < 1$. It is easy to see why it has to be the case that $\frac{1}{2} < \frac{p}{q}$: unless that is fulfilled we cannot claim there is a "$\leq$" inequality between (3.32) and (3.33). Now we argue it has to be the case that $\frac{p}{q} < 1$. Assume that $\frac{p}{q} = 1$,* i.e., *we substitute $\left\lceil \frac{n}{2} \right\rceil$ with n. Then (3.33) becomes:*

$$c(n+2)(\lg n) + n = cn \lg n + 2c \lg n + n$$

Clearly, that is bigger than $cn \lg n$ for all sufficiently large n and we have no proof.

Part II: Proof that $T(n) = \Omega(n \lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn \lg n \tag{3.34}$$

From the inductive hypothesis

$$\begin{aligned} T(n) &\geq 2.d. \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + n \\ &\geq 2.d. \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) + n \\ &= dn(\lg n - 1) + n \\ &= dn \lg n + (1-d)n \\ &\geq dn \lg n \qquad \text{provided that } (1-d)n \geq 0 \end{aligned} \tag{3.35}$$

It follows that any d such that $0 < d \leq 1$ works for our proof. □

:: NB :: *As explained in [CLR00, pp. 56–57], it is easy to make a wrong "proof" of the growth rate by induction if one is not careful. Suppose one "proves" the solution of (3.20) is* $T(n) = O(n)$ *by first guessing (incorrectly) that* $T(n) \leq cn$ *for some positive constant* c *and then arguing*

$$\begin{aligned} T(n) &\leq 2c \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq cn + n \\ &= (c+1)n \\ &= O(n) \end{aligned}$$

While it is certainly true that $cn + n = O(n)$*, that is irrelevant to the proof. The proof started relative to the constant* c *and has to finish relative to it. In other words, the proof has to show that* $T(n) \leq cn$ *for the choice of* c *in the inductive hypothesis, not that* $T(n) \leq dn$ *for some positive constant* d *which is not* c*. Proving that* $T(n) \leq (c+1)n$ *does* not *constitute a proof of the statement we are after.*

Problem 49. *Solve*

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \tag{3.36}$$

$$T(1) = \Theta(1) \tag{3.37}$$

Solution:
We prove that $T(n) = \Theta(\lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$.

Part I: Proof that $T(n) = O(\lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq c \lg n \tag{3.38}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\le c \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&\le c \lg\left(\frac{n}{2}\right) + 1 \\
&= c(\lg n - 1) + 1 \\
&= c \lg n + 1 - c \\
&\le c \lg n \qquad \text{provided that } 1 - c \le 0 \Leftrightarrow c \ge 1
\end{aligned}$$

Part II: Proof that $T(n) = \Omega(\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \ge n_1$,

$$T(n) \ge d \lg n \tag{3.39}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\ge d \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&\ge d \lg\left(\frac{n}{4}\right) + 1 \qquad \text{since } \frac{n}{4} \le \left\lfloor \frac{n}{2} \right\rfloor \text{ for all sufficiently large } n \\
&= d \lg n - 2d + 1 \\
&\ge d \lg n \qquad \text{provided that } -2d + 1 \ge 0 \Leftrightarrow d \le \frac{1}{2}
\end{aligned}$$

□

Problem 50. *Solve*

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \tag{3.40}$$

$$T(1) = \Theta(1) \tag{3.41}$$

Solution:
We prove that $T(n) = \Theta(\lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$.

Part I: Proof that $T(n) = O(\lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \ge n_0$,

$$T(n) \le c \lg n \tag{3.42}$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\le c \lg\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\
&\le c \lg\left(\frac{3n}{4}\right) + 1 \\
&= c(\lg n + \lg 3 - 2) + 1 \\
&= c \lg n + c(\lg 3 - 2) + 1 \\
&\le c \lg n \qquad \text{provided that } c(\lg 3 - 2) + 1 \le 0 \Leftrightarrow c \ge \frac{1}{2 - \lg 3}
\end{aligned}$$

Part II: Proof that $T(n) = \Omega(\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq d \lg n \tag{3.43}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq d \lg \left(\left\lceil \frac{n}{2} \right\rceil \right) + 1 \\ &\geq d \lg \left(\frac{n}{2} \right) + 1 \\ &= d \lg n - d + 1 \\ &\geq d \lg n \qquad \text{provided that } -d + 1 \geq 0 \Leftrightarrow d \leq 1 \end{aligned}$$

□

Problem 51. *Solve*

$$T(n) = 2T \left(\left\lfloor \frac{n}{2} + 17 \right\rfloor \right) + n \tag{3.44}$$

Solution:
We prove that $T(n) = \Theta(n \lg n)$ by induction on n. To accomplish that we prove separately that $T(n) = O(\lg n)$ and $T(n) = \Omega(\lg n)$. Note that the initial condition in this problem is *not* $T(1) = \Theta(1)$ because the iterator

$$n \to \left\lfloor \frac{n}{2} \right\rfloor + 17$$

never reaches 1 when starting from any sufficiently large n. Its fixed point is 34 but we avoid mentioning the awkward initial condition $T(34) = \Theta(1)$.

Part I: Proof that $T(n) = O(n \lg n)$, that is, there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \lg n \tag{3.45}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c \left(\left\lfloor \frac{n}{2} \right\rfloor + 17 \right) \lg \left(\left\lfloor \frac{n}{2} \right\rfloor + 17 \right) + n \\ &= 2c \left(\frac{n}{2} + 17 \right) \lg \left(\frac{n}{2} + 17 \right) + n \\ &= c(n + 34) \lg \left(\frac{n + 34}{2} \right) + n \\ &= c(n + 34) \big(\lg (n + 34) - 1 \big) + n \\ &\leq c(n + 34) \big(\lg (\sqrt{2}n) - 1 \big) + n \end{aligned} \tag{3.46}$$

because for all sufficiently large values of n, say $n \geq 100$, it is the case that $\sqrt{2}n \geq n + 34$.

$$\begin{aligned}
T(n) &\leq c(n+34)\big(\lg(\sqrt{2}n) - 1\big) + n \qquad \text{from (3.46)} \\
&= c(n+34)\left(\lg n + \frac{1}{2}\lg 2 - 1\right) + n \\
&= c(n+34)\left(\lg n - \frac{1}{2}\right) + n \\
&= cn\lg n + 34c\lg n - \frac{cn}{2} - 17c + n \\
&\leq cn\lg n \qquad \text{provided that } 34c\lg n - \frac{cn}{2} - 17c + n \leq 0
\end{aligned}$$

In order $34c\lg n - \frac{cn}{2} - 17c + n = n\left(1 - \frac{c}{2}\right) + 34c\lg n - 17c$ to be non-positive for all sufficiently large n it suffices $\left(1 - \frac{c}{2}\right)$ to be negative because the linear function dominated the logarithmic function. A more detailed analysis is the following. Fix $c = 4$. The expression becomes $(-1)n + 136\lg n - 136$.

$$(-1)n + 136\lg n - 136 \leq 0 \;\Leftrightarrow\; n \geq 136(\lg n - 1) \;\Leftrightarrow\; \frac{n}{\lg n - 1} \geq 136$$

For $n = 65536 = 2^{16}$ the inequality holds:

$$\frac{2^{16}}{15} \geq 136$$

so we can finish the proof with choosing $n_0 = 65536$ and $c = 4$.

Part II: Proof that $T(n) = \Omega(n\lg n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn\lg n$$

By the inductive hypothesis,

$$\begin{aligned}
T(n) &\geq 2d\left(\left\lfloor\frac{n}{2}\right\rfloor + 17\right)\lg\left(\left\lfloor\frac{n}{2}\right\rfloor + 17\right) + n \\
&\geq 2d\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + n \\
&= dn(\lg n - 1) + n \\
&= dn\lg n + (1-d)n \\
&\geq dn\lg n \qquad \text{provided that } 1 - d \geq 0 \;\Leftrightarrow\; d \leq 1 \qquad \square
\end{aligned}$$

Problem 52. *Solve*

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \qquad (3.47) \\
T(1) &= \Theta(1)
\end{aligned}$$

by the method of the recursion tree.

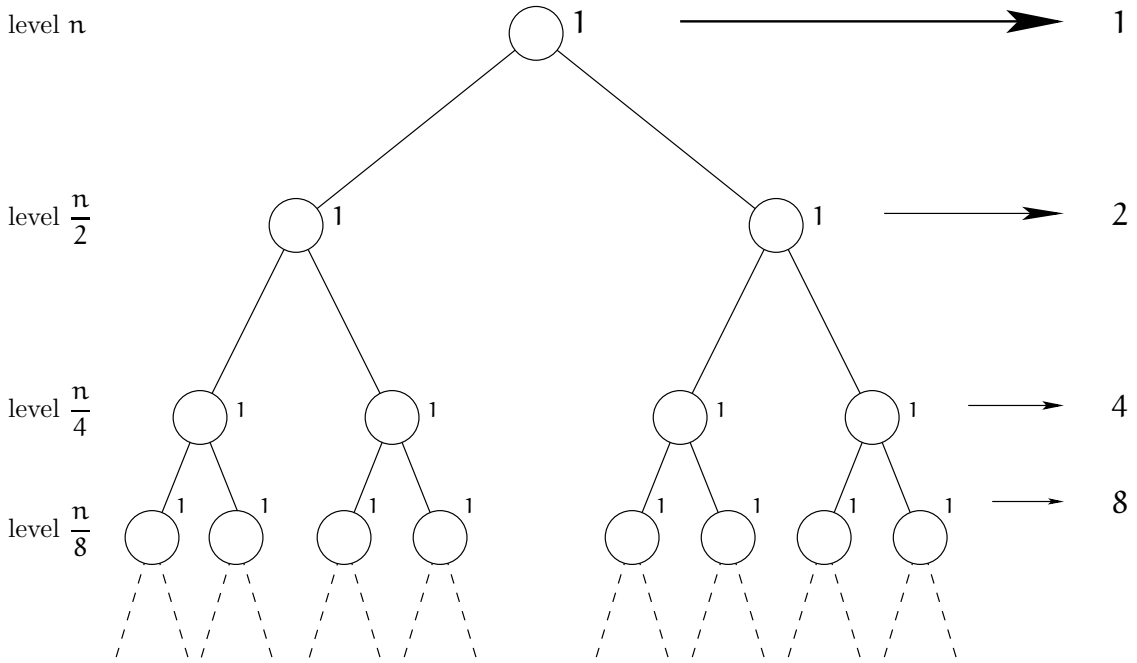


Figure 3.5: The recursion tree of $T(n) = 2T\left(\frac{n}{2}\right) + 1$.

Solution:
The recursion tree is shown on Figure 3.5. The solution is the sum over all levels:

$$T(n) = \underbrace{1 + 2 + 4 + 8 + \ldots}_{\text{the number of terms is the height of the tree}} \tag{3.48}$$

The height of the tree is the number of times the iterator

$$n \to \frac{n}{2}$$

is executed before the variable becomes 1. As we already saw, that number is $\lg n$[†]. So, (3.48) in fact is

$$\begin{aligned} T(n) &= \underbrace{1 + 2 + 4 + 8 + \ldots}_{(\lg n + 1)\text{ terms}} = 1 + 2 + 4 + 8 + \ldots + \frac{n}{2} + n \\ &= \sum_{i=0}^{\lg n} \frac{n}{2^i} = n \left(\sum_{i=0}^{\lg n} \frac{1}{2^i} \right) \leq n \underbrace{\left(\sum_{i=0}^{\infty} \frac{1}{2^i} \right)}_{2} = 2n \end{aligned}$$

We conclude that $T(n) = \Theta(n)$. □

However, that proof by the method of the recursion tree can be considered insufficiently precise because it involves several approximations and the use of imagination—the dot-dot-dot notations. Next we demonstrate a proof by induction of the same result. We may think

[†]Actually it is $\lfloor \lg n \rfloor$ but that is immaterial with respect to the asymptotic growth of $T(n)$.

of the proof with recursion tree as a mere way to derive a good guess to be verified formally by induction.

Problem 53. *Prove by induction on* n *that the solution to*

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\ T(1) &= \Theta(1) \end{aligned} \tag{3.49}$$

is $T(n) = \Theta(n)$.

Solution:
We prove separately that $T(n) = O(n)$ and $T(n) = \Omega(n)$.

Part I: Proof that $T(n) = O(n)$. For didactic purposes we will first make an unsuccessful attempt.

Part I, try 1: Assume there exists a positive constant c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn \tag{3.50}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2} + 1 \\ &= cn + 1 \end{aligned}$$

Our proof ran into a problem: no matter what positive c we choose, it is not true that $cn + 1 \leq cn$, and thus (3.50) cannot be shown to hold. Of course, that failure does *not* mean our claim $T(n) = \Theta(n)$ is false. It simply means that (3.50) is inappropriate. We amend the situation by a technique known as *strenthening the claim*. It consists of stating an appropriate claim that is stronger than (3.50) and then proving it by induction. Intuitively, that stronger claim has to contain some minus sign in such a way that after applying the inductive hypothesis, there is a term like $-c$ that can "cope with" the $+1$.

Part I, try 2: Assume there exists positive constants b and c and some n_0, such that for all $n \geq n_0$,

$$T(n) \leq cn - b \tag{3.51}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2\left(c\frac{n}{2} - b\right) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \qquad \text{for any } b \text{ such that } -b + 1 \leq 0 \Leftrightarrow b \geq 1 \end{aligned}$$

Part II: Proof that $T(n) = \Omega(n)$, that is, there exists a positive constant d and some n_1, such that for all $n \geq n_1$,

$$T(n) \geq dn$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq 2\left(d\frac{n}{2}\right) + 1 \\ &= dn + 1 \\ &\geq dn \end{aligned}$$

□

Problem 54. *Prove by induction on* n *that the solution to*

$$\begin{aligned} T(n) &= 2T(n-1) + n \\ T(1) &= \Theta(1) \end{aligned} \tag{3.52}$$

is $T(n) = \Theta(2^n)$.

Solution:
We prove separately that $T(n) = O(2^n)$ and $T(n) = \Omega(2^n)$.

Part I: Proof that $T(n) = O(2^n)$. For didactic purposes we will first make several unsuccessful attempts.

Part I, try 1: Assume there exists a positive constant c such that for all large enough n,

$$T(n) \leq c2^n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2c2^{n-1} + n \\ &= c2^n + n \\ &\not\leq c2^n \text{ for any choice of positive } c \end{aligned}$$

Our proof failed so let us strenghten the claim.

Part I, try 2: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^n - b$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b) + n \\ &= c2^n - 2b + n \\ &\not\leq c2^n - b \text{ for any choice of positive } c \end{aligned}$$

The proof failed once again so let us try another strenghtening of the claim.

Part I, try 3: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^{n-b}$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-b-1}) + n \\ &= c2^{n-b} + n \\ &\not\leq c2^{n-b} \text{ for any choice of positive } c \end{aligned}$$

Yet another failure and we try yet another strenghtening of the claim.

Part I, try 4: Assume there exists a positive constant c such that for all large enough n,

$$T(n) \leq c2^n - n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - (n-1)) + n \\ &= c2^n - n + 2 \\ &\not\leq c2^n - n \text{ for any choice of positive } c \end{aligned}$$

Part I, try 5: Assume there exist positive constants b and c such that for all large enough n,

$$T(n) \leq c2^n - bn$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - b(n-1)) + n \\ &= c2^n - 2bn + 2b + n \\ &= c2^n - bn + (1-b)n + 2b \\ &\leq c2^n - bn \text{ for any choice of } c > 0 \text{ and } b > 1 \end{aligned}$$

Success! At last we have managed to formulate a provable hypothesis.

Part II: Proof that $T(n) = \Omega(n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \geq d2^n$$

By the inductive hypothesis,

$$\begin{aligned} T(n) &\geq 2(d2^{n-1}) + n \\ &= d2^n + n \\ &\geq d2^n \end{aligned}$$

Success! Again we see that the strengthening of the claim is required only in one direction of the proof. □

The next three problems have the iterator

$$n \to \sqrt{n}$$

According to the table on page 46, that number of times this iterator is executed before n becomes some fixed constant is $\Theta(\lg\lg n)$. Note, however, that unless n is integer, this constant cannot be 1 because for real n, it is the case that $n > 1$ after any iteration. Therefore "$T(1) = \Theta(1)$" cannot be the initial condition if n is real. One way out of that is to change the initial conditions to

$$T(n) = \Theta(1) \text{ for } 2 \le n \le 4$$

Problem 55. *Solve*

$$T(n) = 2T(\sqrt{n}) + 1 \tag{3.53}$$

Solution:
Substitute n by 2^{2^m}, *i.e.* $m = \lg\lg n$ and $2^m = \lg n$. Then (3.53) becomes

$$T\left(2^{2^m}\right) = 2T\left(2^{\frac{2^m}{2}}\right) + 1$$

which is

$$T\left(2^{2^m}\right) = 2T\left(2^{2^{m-1}}\right) + 1 \tag{3.54}$$

Further substitute $T\left(2^{2^m}\right)$ by $S(m)$ and (3.54) becomes

$$S(m) = 2S(m-1) + 1 \tag{3.55}$$

But we know the solution to that recurrence. According to Problem 45, $S(m) = \Theta(2^m)$. Let us go back now to the original n and $T(n)$.

$$S(m) = \Theta(2^m) \;\Leftrightarrow\; T\left(2^{2^m}\right) = \Theta(\lg n) \;\Leftrightarrow\; T(n) = \Theta(\lg n)$$

□

Problem 56. *Solve*

$$T(n) = 2T(\sqrt{n}) + \lg n \tag{3.56}$$

Solution:
Substitute n by 2^m, *i.e.* $m = \lg n$. Then (3.56) becomes

$$T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m \tag{3.57}$$

Further substitute $T(2^m)$ by $S(m)$ and (3.57) becomes

$$S(m) = 2S\left(\frac{m}{2}\right) + m \tag{3.58}$$

Consider Problem 47 and Problem 48. They have solve the same recurrence, differing from (3.58) only in the way the division is rounded to integer. In Problem 47 the iterator is

$$n \to \left\lfloor \frac{n}{2} \right\rfloor$$

and in Problem 48 the iterator is

$$n \to \left\lceil \frac{n}{2} \right\rceil$$

Both Problem 47 and Problem 48 have $\Theta(n \lg n)$ solutions. We conclude the solution of (3.58) is $S(m) = \Theta(m \lg m)$, which is equivalent to $T(n) = \Theta(\lg n \lg\lg n)$. □

Problem 57. *Solve*

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.59}$$

Solution:
Let us unfold the recurrence:

$$T(n) = n + n^{\frac{1}{2}}T\left(n^{\frac{1}{2}}\right) \tag{3.60}$$
$$= n + n^{\frac{1}{2}}\left(n^{\frac{1}{2}} + n^{\frac{1}{4}}T\left(n^{\frac{1}{4}}\right)\right) \tag{3.61}$$
$$= 2n + n^{\frac{3}{4}}T\left(n^{\frac{1}{4}}\right) \tag{3.62}$$
$$= 2n + n^{\frac{3}{4}}\left(n^{\frac{1}{8}} + T\left(n^{\frac{1}{8}}\right)\right) \tag{3.63}$$
$$= 3n + n^{\frac{7}{8}}T\left(n^{\frac{1}{8}}\right) \tag{3.64}$$
$$\dots \tag{3.65}$$
$$= in + n^{\left(1-\frac{1}{2^i}\right)}T\left(n^{\frac{1}{2^i}}\right) \tag{3.66}$$

As we already said, the maximum value of i, call it $i_{\max}$, is $i_{\max} = \lg\lg n$. But then $2^{i_{\max}} = \lg n$, therefore

$$n^{\left(1-\frac{1}{2^{i_{\max}}}\right)} = \frac{n}{n^{\frac{1}{2^{i_{\max}}}}} = \frac{n}{n^{\frac{1}{\lg n}}} = \frac{n}{2}$$

The derivation of the fact that $n^{\frac{1}{\lg n}} = 2$ is on page 17. So, for $i = i_{\max}$,

$$T(n) = (\lg\lg n)n + \frac{n}{2}T(c) \qquad c \text{ is some number such that } 2 \le c \le 4$$

But $T(c)$ is a constant, therefore $T(n) = \Theta(n \lg\lg n)$.

Let us prove the same result by induction.
Part 1: Prove that $T(n) = O(n \lg\lg n)$, that is, there exists a positive constant c such that for all sufficiently large n,

$$T(n) \le cn \lg\lg n \tag{3.67}$$

Our inductive hypothesis then is

$$T(\sqrt{n}) \le c\sqrt{n} \lg\lg \sqrt{n} \tag{3.68}$$

We know by the definition of the problem that

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.69}$$

Apply (3.68) to (3.69) to get

$$\begin{aligned}
T(n) &\leq \sqrt{n}(c\sqrt{n}\lg\lg\sqrt{n}) + n \\
&= cn\lg\lg\sqrt{n} + n \\
&= cn\lg\left(\frac{1}{2}\lg n\right) + n \\
&= cn\lg\left(\frac{\lg n}{2}\right) + n \\
&= cn(\lg\lg n - 1) + n \\
&= cn\lg\lg n - cn + n \\
&\leq cn\lg\lg n \qquad \text{provided that } -cn + n \leq 0 \Leftrightarrow c \geq 1
\end{aligned}$$

Part 2: Prove that $T(n) = \Omega(n\lg\lg n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \geq dn\lg\lg n \tag{3.70}$$

Our inductive hypothesis then is

$$T(\sqrt{n}) \geq d\sqrt{n}\lg\lg\sqrt{n} \tag{3.71}$$

We know by the definition of the problem that

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \tag{3.72}$$

Apply (3.71) to (3.72) to get

$$\begin{aligned}
T(n) &\geq \sqrt{n}(d\sqrt{n}\lg\lg\sqrt{n}) + n \\
&= dn\lg\lg\sqrt{n} + n \\
&= dn\lg\left(\frac{1}{2}\lg n\right) + n \\
&= dn\lg\left(\frac{\lg n}{2}\right) + n \\
&= dn(\lg\lg n - 1) + n \\
&= dn\lg\lg n - dn + n \\
&\geq dn\lg\lg n \qquad \text{provided that } -dn + n \geq 0 \Leftrightarrow d \leq 1
\end{aligned}$$

□

Problem 58. *Solve*

$$T(n) = n^2T\left(\frac{n}{2}\right) + 1 \tag{3.73}$$

Solution:
Unfold the recurrence:

$$\begin{aligned}
T(n) &= n^2 T\left(\frac{n}{2}\right) + 1 \\
&= n^2\left(\frac{n^2}{4}T\left(\frac{n}{4}\right) + 1\right) + 1 \\
&= \frac{n^4}{2^2}T\left(\frac{n}{2^2}\right) + n^2 + 1 \\
&= \frac{n^4}{2^2}\left(\frac{n^2}{2^4}T\left(\frac{n}{2^3}\right) + 1\right) + n^2 + 1 \\
&= \frac{n^6}{2^6}T\left(\frac{n}{2^3}\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^6}{2^6}\left(\frac{n^2}{2^6}T\left(\frac{n}{2^4}\right) + 1\right) + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}}T\left(\frac{n}{2^4}\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^8}{2^{12}}\left(\frac{n^2}{2^8}T\left(\frac{n}{2^5}\right) + 1\right) + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}}T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + n^2 + 1 \\
&= \frac{n^{10}}{2^{20}}T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{12}} + \frac{n^6}{2^6} + \frac{n^4}{2^2} + \frac{n^2}{2^0} + \frac{n^0}{2^0} \\
&= \frac{n^{10}}{2^{5.4}}T\left(\frac{n}{2^5}\right) + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}} \\
&= \dots \\
&= \underbrace{\frac{n^{2i}}{2^{i(i-1)}}T\left(\frac{n}{2^i}\right)}_{A} + \underbrace{\frac{n^{2(i-1)}}{2^{(i-1)(i-2)}} + \frac{n^{2(i-2)}}{2^{(i-2)(i-3)}} \dots + \frac{n^8}{2^{4.3}} + \frac{n^6}{2^{3.2}} + \frac{n^4}{2^{2.1}} + \frac{n^2}{2^{1.0}} + \frac{n^0}{2^{0.(-1)}}}_{B}
\end{aligned}$$

The maximum value of i is $i_{max} = \lg n$. First we compute A with $i = \lg n$, having in mind $T(1)$ is some positive constant c.

$$A = \frac{n^{2\lg n}}{2^{\lg^2 n - \lg n}}T(1) = \frac{c2^{\lg n}n^{2\lg n}}{2^{\lg^2 n}} = \frac{c.n.n^{2\lg n}}{2^{\lg^2 n}}$$

But

$$2^{\lg^2 n} = 2^{\lg n.\lg n} = 2^{\lg\left(n^{\lg n}\right)} = n^{\lg n} \tag{3.74}$$

Therefore

$$A = \frac{c.n.n^{2\lg n}}{n^{\lg n}} = \Theta(n^{1+\lg n}) \tag{3.75}$$

Consider B. Obviously, we can represent it as a sum in the following way:

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{n^{2((\lg n)-j)}}{2^{((\lg n)-j)((\lg n)-j-1)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{n^{2\lg n}}{2^{(\lg^2 n - j\lg n - j\lg n + j^2 - \lg n + j)}} \\ &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(2^{2j\lg n}\right)\left(2^{\lg n}\right)}{\left(2^{\lg^2 n}\right)\left(2^{j^2+j}\right)} \end{aligned} \tag{3.76}$$

But

$$2^{2j\lg n} = 2^{\lg\left(n^{2j}\right)} = n^{2j} \tag{3.77}$$

and

$$2^{\lg n} = n \tag{3.78}$$

Apply (3.74), (3.77), and (3.78) on (3.76) to obtain

$$\begin{aligned} B &= \sum_{j=1}^{\lg n} \frac{1}{n^{2j}} \frac{\left(n^{2\lg n}\right)\left(n^{2j}\right)(n)}{\left(n^{\lg n}\right)\left(2^{j^2+j}\right)} \\ &= \sum_{j=1}^{\lg n} \frac{n^{1+\lg n}}{2^{j^2+j}} \\ &= \left(n^{1+\lg n}\right) \sum_{j=1}^{\lg n} \frac{1}{2^{j^2+j}} \\ &\le \left(n^{1+\lg n}\right) \sum_{j=1}^{\infty} \frac{1}{2^{j^2+j}} \\ &\le n^{1+\lg n} \qquad \text{since we know that } \sum_{j=1}^{\infty} \frac{1}{2^j} = 1 \\ &= \Theta\left(n^{1+\lg n}\right) \end{aligned} \tag{3.79}$$

From (3.75) and (3.79) it follows that

$$T(n) = \Theta\left(n^{1+\lg n}\right)$$

□

Problem 59. *Solve by unfolding*

$$T(n) = T(n-2) + 2\lg n \tag{3.80}$$

Solution:
Let us unfold the recurrence:

$$\begin{aligned} T(n) &= T(n-2) + 2\lg n \\ &= T(n-4) + 2\lg(n-2) + 2\lg n \\ &= T(n-6) + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \\ &= \ldots \\ &= T(c) + \ldots + 2\lg(n-4) + 2\lg(n-2) + 2\lg n \end{aligned} \tag{3.81}$$

where c is either 1 or 2[†].

Case I: n is odd. Then $c = 1$ and (3.81) is:

$$2\lg n + 2\lg(n-2) + 2\lg(n-4) + \ldots + 2\lg 3 + T(1) \tag{3.82}$$

We approximate $T(1)$ with $0 = \lg 1$, which does not alter the asymptotic growth rate of (3.82), and thus (3.82) becomes:

$$\begin{aligned} &\lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \ldots + \lg 3^2 + \lg 1 = \\ &\lg\left(n^2(n-2)^2(n-4)^2 \ldots 3^2.1\right) = \\ &\lg\Big(\underbrace{n.n(n-2)(n-2)(n-4)(n-4)\ldots 5.5.3.3.1}_{n \text{ factors}}\Big) = T(n) \end{aligned} \tag{3.83}$$

Define

$$X(n) = \lg\Big(\underbrace{n(n-1)(n-2)(n-3)\ldots 3.2.1}_{n \text{ factors}}\Big) = \lg n!$$
$$Y(n) = \lg\Big(\underbrace{(n+1)n(n-1)(n-2)\ldots 4.3.2}_{n \text{ factors}}\Big) = \lg(n+1)!$$

and note that

$$X(n) \le T(n) \le Y(n) \tag{3.84}$$

because of the following inequalities between the corresponding factors inside the logarithms

$$\begin{array}{lccccccccc} X(n) = \lg(& n & n-1 & n-2 & n-3 & \cdots & 3 & 2 & 1 &) \\ & /\!/\backslash & /\!/\backslash & /\!/\backslash & /\!/\backslash & & /\!/\backslash & /\!/\backslash & /\!/\backslash & \\ T(n) = \lg(& n & n & n-2 & n-2 & \cdots & 3 & 3 & 1 &) \\ & /\!/\backslash & /\!/\backslash & /\!/\backslash & /\!/\backslash & & /\!/\backslash & /\!/\backslash & /\!/\backslash & \\ Y(n) = \lg(& n+1 & n & n-1 & n-2 & \cdots & 4 & 3 & 2 &) \end{array}$$

However, $X(n) = \Theta(n\lg n)$ and $Y(n) = \Theta((n+1)\lg(n+1)) = \Theta(n\lg n)$ by (1.48). Having in mind that and (3.84), $T(n) = \Theta(n\lg n)$ follows immediately.

[†]The initial conditions that define $T(1)$ and $T(2)$ are omitted.

Case II: n is even. Then $c = 2$ and (3.81) is:

$$2\lg n + 2\lg(n-2) + 2\lg(n-4) + \ldots + 2\lg 4 + T(2) \tag{3.85}$$

We approximate $T(2)$ with $1 = \lg 2$, which does not alter the asymptotic growth rate of (3.82), and thus (3.82) becomes:

$$\begin{aligned}
&\lg n^2 + \lg(n-2)^2 + \lg(n-4)^2 + \ldots + \lg 4^2 + \lg 2 = \\
&\lg\left(n^2(n-2)^2(n-4)^2 \ldots 4^2.2\right) = \\
&\lg\Big(\underbrace{n.n(n-2)(n-2)(n-4)(n-4)\ldots 6.6.4.4.2}_{n-1 \text{ factors}}\Big) = T(n)
\end{aligned} \tag{3.86}$$

Define

$$X(n) = \lg\Big(\underbrace{n(n-1)(n-2)(n-3)\ldots 4.3.2}_{n-1 \text{ factors}}\Big) = \lg n!$$

$$Y(n) = \lg\Big(\underbrace{(n+1)n(n-1)(n-2)\ldots 5.4.3}_{n-1 \text{factors}}\Big) = \lg\frac{(n+1)!}{2} = \lg(n+1)! - 1$$

and note that

$$X(n) \le T(n) \le Y(n) \tag{3.87}$$

because of the following inequalities between the corresponding factors inside the logarithms

$$\begin{array}{lccccccccc}
X(n) = \lg(& n & n-1 & n-2 & n-3 & \cdots & 4 & 3 & 2 &) \\
 & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & \\
T(n) = \lg(& n & n & n-2 & n-2 & \cdots & 4 & 4 & 2 &) \\
 & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & & /\!/\!\backslash & /\!/\!\backslash & /\!/\!\backslash & \\
Y(n) = \lg(& n+1 & n & n-1 & n-2 & \cdots & 5 & 4 & 3 &)
\end{array}$$

However, $X(n) = \Theta(n\lg n)$ and $Y(n) = \Theta((n+1)\lg(n+1)) = \Theta(n\lg n)$ by (1.48). Having in mind that and (3.84), $T(n) = \Theta(n\lg n)$ follows immediately. □

Problem 60. *Solve by induction*

$$T(n) = T(n-2) + 2\lg n \tag{3.88}$$

Solution:
We use Problem 59 to guess the solution $T(n) = \Theta(n\lg n)$.

Part I: Proof that $T(n) = O(n\lg n)$, that is, there exists a positive constant c such that for all sufficiently large n,

$$T(n) \le cn\lg n \tag{3.89}$$

The following inequalities hold

$$\begin{aligned}
T(n) &\le c(n-2)\lg(n-2) + 2\lg n \qquad \text{from the induction hypothesis} \\
&\le c(n-2)\lg n + 2\lg n \\
&= cn\lg n - 2c\lg n + 2\lg n \\
&\le cn\lg n \qquad \text{provided that } -2c\lg n + 2\lg n \le 0 \;\Leftrightarrow\; c \ge 1
\end{aligned}$$

Part II: Proof that $T(n) = \Omega(n\lg n)$, that is, there exists a positive constant d such that for all sufficiently large n,

$$T(n) \ge dn\lg n \tag{3.90}$$

It is the case that

$$\begin{aligned}
T(n) &\ge d(n-2)\lg(n-2) + 2\lg n \qquad \text{from the induction hypothesis} \\
&= (dn - 2d)\lg(n-2) + 2\lg n \\
&= dn\lg(n-2) + 2(\lg n - d\lg(n-2)) \qquad (3.91)
\end{aligned}$$

Having in mind (3.90) and (3.91), our goal is to show that

$$\begin{aligned}
dn\lg(n-2) + 2(\lg n - d\lg(n-2)) &\ge dn\lg n \;\Leftrightarrow \\
dn\lg(n-2) - dn\lg n + 2(\lg n - d\lg(n-2)) &\ge 0 \;\Leftrightarrow \\
\underbrace{d\lg\left(\frac{n-2}{n}\right)^n}_{A} + \underbrace{2\lg\frac{n}{(n-2)^d}}_{B} &\ge 0 \qquad (3.92)
\end{aligned}$$

Let us first evaluate A when n grows infinitely:

$$\lim_{n\to\infty} d\lg\left(\frac{n-2}{n}\right)^n = d\lim_{n\to\infty}\lg\left(1+\frac{-2}{n}\right)^n = d\lg\lim_{n\to\infty}\left(1+\frac{-2}{n}\right)^n = d\lg e^{-2} = -2d\lg e$$

Now consider B when n grows infinitely:

$$\lim_{n\to\infty} 2\lg\frac{n}{(n-2)^d} = 2\lg\lim_{n\to\infty}\frac{n}{(n-2)^d} \tag{3.93}$$

Note that for any d such that $0 < d < 1$, (3.93) is $+\infty$. For instance, for $d = \frac{1}{2}$, (3.93) becomes

$$2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\frac{n^{\frac{1}{2}}}{(n-2)^{\frac{1}{2}}}\right) =$$

$$2\lg\lim_{n\to\infty}\left(n^{\frac{1}{2}}\left(\frac{n}{n-2}\right)^{\frac{1}{2}}\right) =$$

$$2\lg\left(\left(\lim_{n\to\infty} n^{\frac{1}{2}}\right)\underbrace{\left(\lim_{n\to\infty}\left(\frac{1}{1-\frac{2}{n}}\right)^{\frac{1}{2}}\right)}_{1}\right) = +\infty$$

It follows inequality (3.92) is true for any choice of d such that $0 < d < 1$, say, $d = \frac{1}{2}$, because A by absolute value is limited by a constant, and B grows infinitely. And that concludes the proof of (3.89). □

⁉ NB ⁉ *The proof by induction in* ***Part II*** *of the solution to Problem 59 is tricky. Consider (3.91):*

$$dn \lg (n-2) + 2(\lg n - d \lg (n-2))$$

Typically, we deal with logarithms of additions or differences by approximating the additions or differences with multiplications or fractions in such a way that the inequality holds in the desired direction. But notice that if we approximate $n-2$ *inside the above logarithms with* any *fraction* $\frac{n}{\alpha}$*, for any positive constant* α*, it must be the case that* $\alpha > 1$*, otherwise the inequality would not be in the direction we want. Here is what happens when we substitute* $n-2$ *with* $\frac{n}{\alpha}$ *in the logarithm on the left:*

$$dn \lg \frac{n}{\alpha} + 2(\lg n - d \lg (n-2)) = dn \lg n - dn \lg \alpha + 2(\lg n - d \lg (n-2))$$

To accomplish the proof, we have to show the latter is greater than or equal to $dn \lg n$*; and to show that, we have to show that the term* $-dn \lg \alpha + 2(\lg n - d \lg (n-2))$ *is positive. But that is not true!* $d > 0$ *and* $\alpha > 1$*, therefore* $-dn \lg \alpha < 0$ *for all* $n > 0$*. And the asymptotic behaviour of* $-dn \lg \alpha + 2(\lg n - d \lg (n-2))$ *is determined by* $-dn \lg \alpha$ *because the linear function dominates the logarithmic function for all sufficiently large* n*. Therefore, we need a more sophisticated technique, based on analysis.*

Problem 61. *Solve by unfolding*

$$T(n) = T(n-1) + \lg n$$

Solution:

$$\begin{aligned}
T(n) &= T(n-1) + \lg n \\
&= T(n-2) + \lg (n-1) + \lg n \\
&= T(n-3) + \lg (n-2) + \lg (n-1) + \lg n \\
&\dots \\
&= \underbrace{T(1)}_{\Theta(1)} + \lg 2 + \lg 3 + \ldots + \lg (n-2) + \lg (n-1) + \lg n \\
&= \Theta(1) + \lg (2.3 \ldots (n-2)(n-1)n) \\
&= \Theta(1) + \lg n! \\
&= \Theta(1) + \Theta(n \lg n) \qquad \text{by (1.48)} \\
&= \Theta(n \lg n)
\end{aligned}$$

□

Problem 62. *Solve by unfolding*

$$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n \tag{3.94}$$

Solution:

$$\begin{aligned}
T(n) &= n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) \\
&= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{\left\lfloor \frac{n}{4} \right\rfloor}{4} \right\rfloor\right)\right) \\
&= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{n}{16} \right\rfloor\right)\right) \qquad \text{because } \left\lfloor \frac{\left\lfloor \frac{n}{4} \right\rfloor}{4} \right\rfloor = \left\lfloor \frac{n}{16} \right\rfloor \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9T\left(\left\lfloor \frac{n}{16} \right\rfloor\right) \\
&= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9\left\lfloor \frac{n}{16} \right\rfloor + 27T\left(\left\lfloor \frac{n}{64} \right\rfloor\right) \\
&\dots \\
&= \underbrace{3^0\left\lfloor \frac{n}{4^0} \right\rfloor + 3^1\left\lfloor \frac{n}{4^1} \right\rfloor + 3^2\left\lfloor \frac{n}{4^2} \right\rfloor + \dots + 3^{i-1}\left\lfloor \frac{n}{4^{i-1}} \right\rfloor}_{P(n)} + \underbrace{3^i T\left(\left\lfloor \frac{n}{4^i} \right\rfloor\right)}_{\text{remainder}} \qquad (3.95)
\end{aligned}$$

The maximum value for i, let us call it i_{max}, is achieved when $\left\lfloor \frac{n}{4^i} \right\rfloor$ becomes 1. It follows $i_{max} = \lfloor \log_4 n \rfloor$. Let us estimate the main part $P(n)$ and the remainder of (3.95) for $i = i_{max}$.

- To estimate $P(n)$, define

$$\begin{aligned}
X(n) &= 3^0\left(\frac{n}{4^0}\right) + 3^1\left(\frac{n}{4^1}\right) + 3^2\left(\frac{n}{4^2}\right) + \dots + 3^{i_{max}-1}\left(\frac{n}{4^{i_{max}-1}}\right) \\
Y(n) &= 3^0\left(\frac{n}{4^0} - 1\right) + 3^1\left(\frac{n}{4^1} - 1\right) + 3^2\left(\frac{n}{4^2} - 1\right) + \dots + 3^{i_{max}-1}\left(\frac{n}{4^{i_{max}-1}} - 1\right)
\end{aligned}$$

  Clearly, $X(n) \geq P(n) \geq Y(n)$. But

$$\begin{aligned}
X(n) &= n\left(\sum_{j=0}^{i_{max}-1}\left(\frac{3}{4}\right)^j\right) \\
&\leq n\sum_{j=0}^{\infty}\left(\frac{3}{4}\right)^j \\
&= n\frac{1}{1-\frac{3}{4}} \\
&= \Theta(n)
\end{aligned}$$

  and

$$\begin{aligned}
Y(n) &= n\left(\sum_{j=0}^{i_{max}-1}\left(\frac{3}{4}\right)^j\right) - \sum_{j=0}^{i_{max}-1} 3^j \\
&= n\,\Theta(1) - \Theta\left(3^{i_{max}-1}\right) \qquad \text{by Corollary 1 on page 21} \\
&= \Theta(n) - \Theta\left(3^{\log_4 n}\right) \qquad \text{since } i_{max} = \lfloor \log_4 n \rfloor \\
&= \Theta(n) - \Theta(n^{\log_4 3}) = \Theta(n)
\end{aligned}$$

  Then it has to be the case that $P(n) = \Theta(n)$.

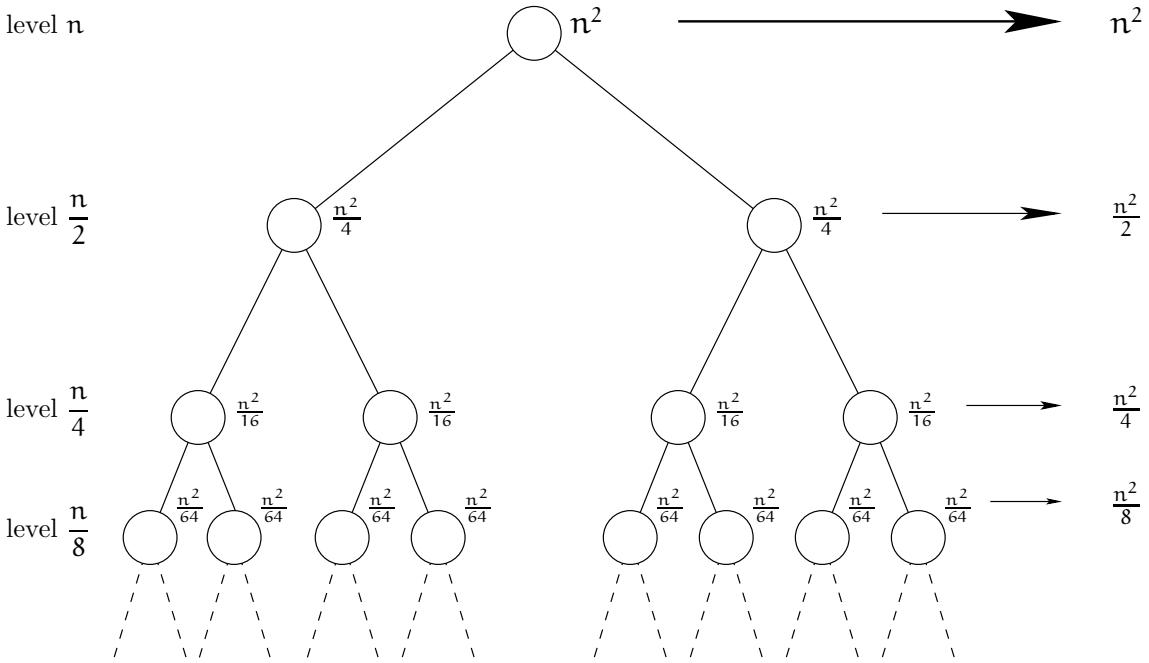


Figure 3.6: The recursion tree of $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

- To estimate the remainder, consider the two factors in it:

$$3^{i_{\max}} = 3^{\lfloor \log_4 n \rfloor} = \Theta(3^{\log_4 n}) = \Theta(n^{\log_3 4})$$
$$T\left(\left\lfloor \frac{n}{4^{i_{\max}}} \right\rfloor\right) = T(1) = \Theta(1)$$

  It follows the remainder is $\Theta(3^{\log_4 n}) = o(n)$.

Therefore, $T(n) = \Theta(n) + o(n) = \Theta(n)$. □

Problem 63. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

by the method of the recursion tree.

Solution:
The recursion tree is shown on Figure 3.6. The solution is the sum

$$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \ldots \leq n^2 \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n^2$$

It follows $T(n) = \Theta(n^2)$. □

Problem 64. *Solve*

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

by the method of the recursion tree.

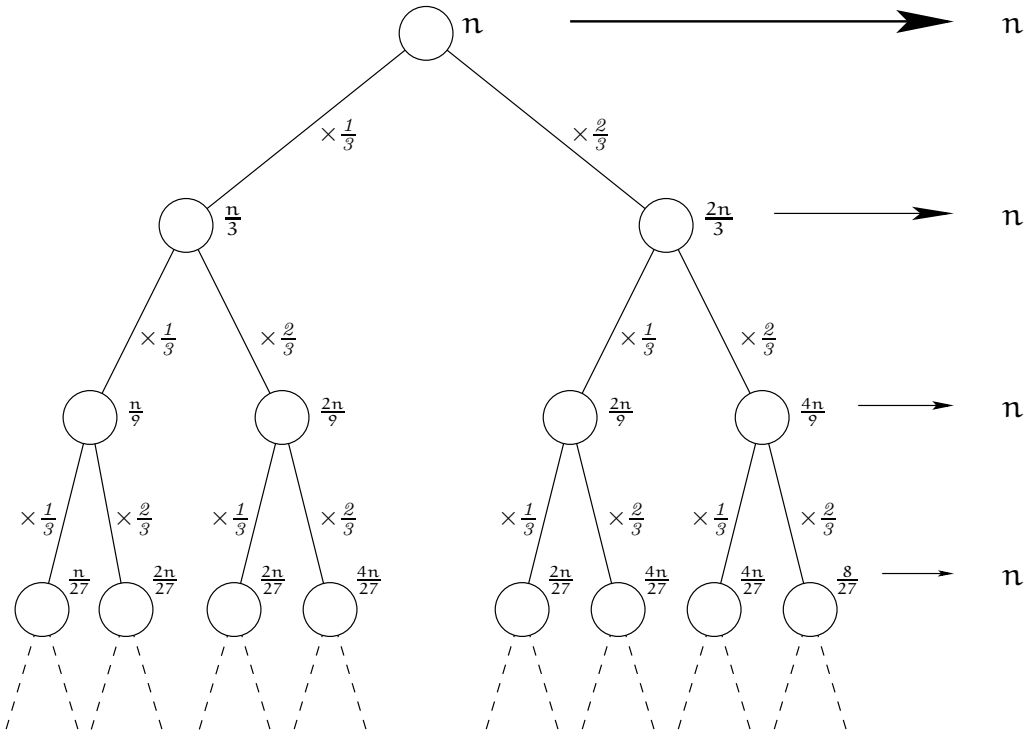


Figure 3.7: The recursion tree of $\mathsf{T}\left(\frac{n}{3}\right) + \mathsf{T}\left(\frac{2n}{3}\right) + n$.

Solution:
The recursion tree is shown on Figure 3.7. This time the tree is not complete so we do not write the levels on the left side in terms of n (as we did on Figure 3.6). Rather, the level of each node is the distance between it and the root. Thus the equidistant with respect to the root nodes are at the same level. Think of the tree as an ordered tree. That is, if a node has any children we distinguish between the left and the right child. The value of the left child is the value of the parent multiplied by $\frac{1}{3}$ and the value of the right child is the value of the parent multiplied by $\frac{2}{3}$. It is trivial to prove by induction that for each level such that all the nodes at this level exist, the sum of the values at that level is n. However, we cannot obtain the answer immediately through mulitplying n by the height because the tree is not balanced. The maximum distance between the root and any leaf is achieved along the rightmost path (starting at the root, always take the right choice; see Figure 3.7) and the minimum distance, by the leftmost path. The length of the leftmost path is determined by the iterator

$$n \to \frac{n}{3}$$

which is executed $\Theta(\log_3 n)$ times before reaching any fixed in advance constant. The length of the rightmost path is determined by the iterator

$$n \to \frac{2n}{3} = \frac{n}{\frac{3}{2}}$$

which is executed $\Theta\left(\log_{\frac{3}{2}} n\right)$ times before reaching any fixed in advance constant.

Let $\mathcal{T}$ be the recursion tree. Construct two balanced trees $\mathcal{T}_1$ and $\mathcal{T}_2$ such that the height of $\mathcal{T}_1$ is $\Theta(\log_3 n)$ and the height of $\mathcal{T}_2$ is $\Theta\left(\log_{\frac{3}{2}} n\right)$. Suppose that each level in $\mathcal{T}_1$ and $\mathcal{T}_2$ is associated with some value n – it does not matter for what reason, just assume each level "costs" n. Let $S_i(n)$ be the sum of those costs in $\mathcal{T}_i$ over all levels for $i = 1, 2$. Clearly,

$$S_1(n) = n \times \Theta(\log_3 n) = \Theta(n \log_3 n) = \Theta(n \lg n)$$
$$S_2(n) = n \times \Theta\left(\log_{\frac{3}{2}} n\right) = \Theta\left(n \log_{\frac{3}{2}} n\right) = \Theta(n \lg n)$$

To conlude the solution, note that $S_1(n) \leq T(n) \leq S_2(n)$ because $\mathcal{T}_1$ can be considered a subtree of $\mathcal{T}$ and $\mathcal{T}$ can be considered a subtree of $\mathcal{T}_2$. Then $T(n) = \Theta(n \lg n)$. □

Problem 65. *Solve by unfolding*

$$T(n) = T(n-a) + T(a) + n \qquad a = const,\ a \geq 1$$

Solution:
We assume a is integer[†] and the initial conditions are

$$T(1) = \Theta(1)$$
$$T(2) = \Theta(1)$$
$$\cdots$$
$$T(a) = \Theta(1)$$

Let us unfold the recurrence.

$$\begin{aligned}
T(n) &= T(n-a) + T(a) + n \\
&= (T(n-2a) + T(a) + n - a) + T(a) + n \\
&= T(n-2a) + 2T(a) + 2n - a \\
&= (T(n-3a) + T(a) + n - 2a) + 2T(a) + 2n - a \\
&= T(n-3a) + 3T(a) + 3n - 3a \\
&= (T(n-4a) + T(a) + n - 4a) + 3T(a) + 3n - 3a \\
&= T(n-4a) + 4T(a) + 4n - 6a \\
&= (T(n-5a) + T(a) + n - 4a) + 4T(a) + 4n - 6a \\
&= T(n-5a) + 5T(a) + 5n - 10a \\
&\cdots \\
&= T(n-ia) + iT(a) + in - \frac{1}{2}i(i-1)a
\end{aligned} \tag{3.96}$$

Let the maximum value i takes be $i_{\max}$. Consider the iterator

$$n \to n - a$$

[†]It is not essential to postulate a is integer. The problems makes sense even if a is just a positive real. If that is the case the initial conditions have to be changed to cover some interval with length a, *e.g.* $T(i) = \text{const.}$ if $i \in (0, a]$.

It maps every $n > a$, $n \in \mathbb{N}$, to a unique number from $\{1, 2, \ldots, a\}$. Let that number be called k. So $i_{\max}$ is the number of times the iterator is executed until the variable becomes k. If $n \bmod a \neq 0$ then k is $n \bmod a$, otherwise k is a[†]. It follows that

$$i_{\max} = \begin{cases} \left\lfloor \frac{n}{a} \right\rfloor, & \text{if } n \bmod a \neq 0 \\ \frac{n}{a} - 1, & \text{else} \end{cases}$$

That is equivalent to

$$i_{\max} = \left\lceil \frac{n}{a} \right\rceil - 1$$

Subsituting i with $\left\lceil \frac{n}{a} \right\rceil - 1$ in (3.96), we get

$$T(k) + \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) T(a) + \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) n - \frac{1}{2} \left(\left\lceil \frac{n}{a} \right\rceil - 1 \right) \left(\left\lceil \frac{n}{a} \right\rceil - 1 - 1 \right) a \tag{3.97}$$

The growth rate of (3.97) is determined by

$$n \left\lceil \frac{n}{a} \right\rceil - \frac{1}{2} \left\lceil \frac{n}{a} \right\rceil \left\lceil \frac{n}{a} \right\rceil = \Theta(n^2)$$

It follows $T(n) = \Theta(n^2)$. □

Problem 66. *Solve*

$$T(n) = T(\alpha n) + T((1-\alpha)n) + n, \qquad \alpha = const.,\ 0 < \alpha < 1 \tag{3.98}$$

by the method of the recursion tree.

Solution:
Define that $1 - \alpha = \beta$. Obviously, $0 < \beta < 1$ and (3.98) becomes

$$T(n) = T(\alpha n) + T(\beta n) + n \tag{3.99}$$

The recursion tree of (3.99) is shown on Figure 3.8. The solution is completely analogous to the solution of Problem 64. The level of each node is the distance between it and the root. The sum of the costs at every level such that all nodes at that levels exist, is n. More precisely, at level i the sum is $(\alpha + \beta)^i n = n$. The tree is not complete. Assume without loss of generality that $\alpha \leq \beta$ and think of the tree as an ordered tree. The shortest path from the root to any leaf is the leftmost one, *i.e.* "follow the alphas", and the longest path is the rightmost one. The length of the shortest path is $\log_{\left(\frac{1}{\alpha}\right)} n$ and of the longest path, $\log_{\left(\frac{1}{\beta}\right)} n$. We prove that $T(n) = \Theta(n \lg n)$ just as in Problem 64 by considering two other trees, one that is a subgraph of the current one and one that isa supergraph of the current one. Since the first of then has sum of the costs $n \times \Theta\left(\log_{\left(\frac{1}{\alpha}\right)} n\right) = \Theta(n \lg n)$ and the second one, $n \times \Theta\left(\log_{\left(\frac{1}{\beta}\right)} n\right) = \Theta(n \lg n)$, it follows $T(n) = \Theta(n \lg n)$. □

[†]Not $n \bmod a$, which is 0.

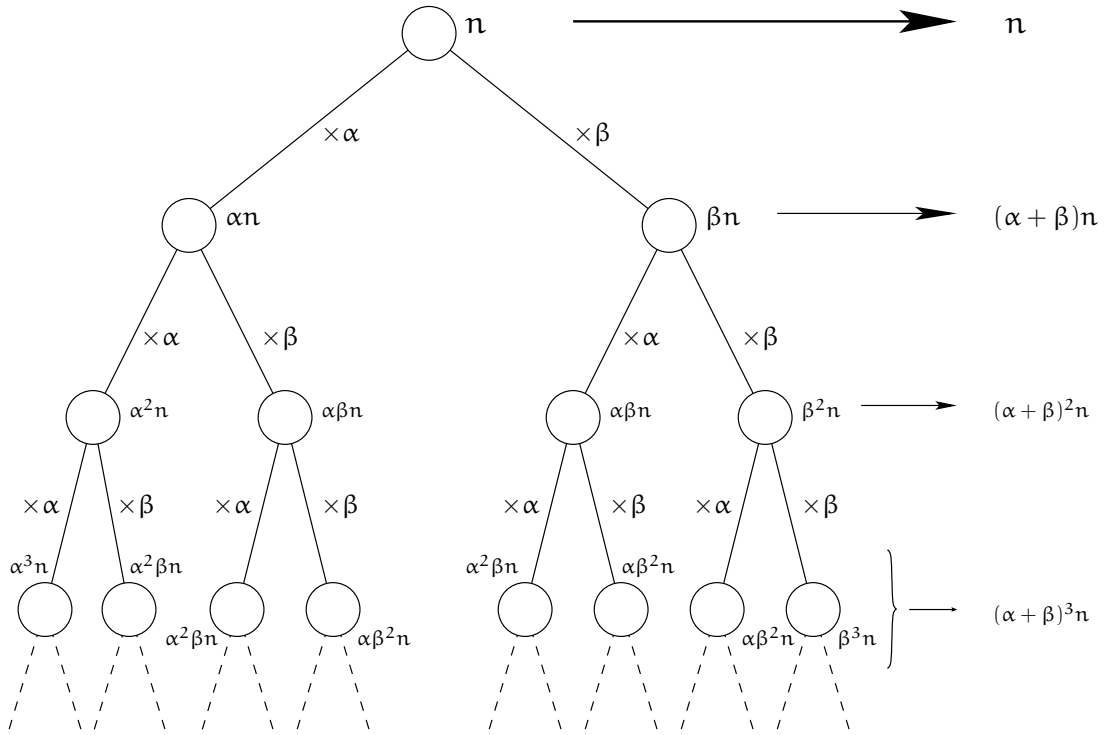


Figure 3.8: The recursion tree of $T(n) = T(\alpha n) + T(\beta n) + n$ where $0 < \alpha, \beta < 1$ and $\alpha + \beta = 1$.

Problem 67. *Solve by unfolding*

$$T(n) = T(n-1) + \frac{1}{n} \tag{3.100}$$

Solution:
Before we commence the unfolding check the definition of the harmonic series, the partial sum H_n of the harmonic series, and its order of growth $\Theta(\lg n)$ on page 168.

$$\begin{aligned}
T(n) &= T(n-1) + \frac{1}{n} \\
&= T(n-2) + \frac{1}{n-1} + \frac{1}{n} \\
&= T(n-3) + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n} \\
&\dots \\
&= T(1) + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n} \\
&= T(1) - 1 + \underbrace{1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n}}_{H_n} \\
&= O(1) + H_n \\
&= O(1) + \Theta(\lg n) \\
&= \Theta(\lg n)
\end{aligned}$$

□

Problem 68. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + 1$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + 1 \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + 1\right) + 1 \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1} + 1 \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + 1\right) + \frac{n}{n+1} + 1 \\
&= \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4) + 1\right) + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{n-3}{n+1}T(n-4) + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \qquad (3.101)
\end{aligned}$$

If we go on like that down to $T(1)$, (3.101) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \ldots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + 1 \\
&= \frac{2}{n+1}T(1) + \frac{3}{n+1} + \frac{4}{n+1} + \ldots + \frac{n-2}{n+1} + \frac{n-1}{n+1} + \frac{n}{n+1} + \frac{n+1}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=3}^{n+1} i \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n+1} i\right) - 3\right) \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\frac{(n+1)(n+2)}{2} - 3\right) \\
&= \frac{1}{n+1}\left(4T(1) + (n^2+3n+2) - 6\right) \\
&= \frac{n^2+3n+4T(1)-4}{n+1} \\
&= \underbrace{\frac{n^2}{n+1}}_{\Theta(n)} + \underbrace{\frac{3n}{n+1}}_{\Theta(1)} + \underbrace{\frac{4T(1)-4}{n+1}}_{O(1)} \\
&= \Theta(n)
\end{aligned}$$

So, $T(n) = \Theta(n)$. □

Problem 69. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + n^2$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + n^2 \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + (n-1)^2\right) + n^2 \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + (n-2)^2\right) + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-2}{n+1}T(n-3) + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4) + (n-3)^2\right) + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{n-3}{n+1}T(n-4) + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2
\end{aligned} \tag{3.102}$$

If we go on like that down to $T(1)$, (3.102) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3.2^2}{n+1} + \frac{4.3^2}{n+1} + \ldots \\
&\qquad + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + n^2 \\
&= \frac{2}{n+1}T(1) + \frac{3.2^2}{n+1} + \frac{4.3^2}{n+1} + \ldots \\
&\qquad + \frac{(n-2)(n-3)^2}{n+1} + \frac{(n-1)(n-2)^2}{n+1} + \frac{n(n-1)^2}{n+1} + \frac{(n+1)n^2}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=3}^{n+1} i(i-1)^2 \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n+1} i(i-1)^2\right) - 2\right) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n+1} i(i-1)^2 \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n+1} (i^3 - 2i^2 + i) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\sum_{i=1}^{n+1} i^3 - 2\sum_{i=1}^{n+1} i^2 + \sum_{i=1}^{n+1} i\right)
\end{aligned} \tag{3.103}$$

Having in mind (6.21), (6.22), and (6.23) on page 169, (3.103) becomes

$$\begin{aligned}
&\frac{2T(1)-2}{n+1}+\frac{1}{n+1}\left(\frac{(n+1)^2(n+2)^2}{4}-2\frac{(n+1)(n+2)(2n+3)}{6}+\frac{(n+1)(n+2)}{2}\right)\\
&=\underbrace{\frac{2T(1)-2}{n+1}}_{O(1)}+\underbrace{\frac{(n+1)(n+2)^2}{4}}_{\Theta(n^3)}-\underbrace{\frac{(n+2)(2n+3)}{3}}_{\Theta(n^2)}+\underbrace{\frac{n+2}{2}}_{\Theta(n)}\\
&=\Theta(n^3)
\end{aligned}$$

So, $T(n)=\Theta(n^3)$. □

Problem 70. *Solve by unfolding*

$$T(n)=\frac{n}{n+1}T(n-1)+\sqrt{n} \tag{3.104}$$

where $\sqrt{n}$ *stands for either* $\lfloor\sqrt{n}\rfloor$ *or* $\lceil\sqrt{n}\rceil$.

Solution:

$$\begin{aligned}
T(n)&=\frac{n}{n+1}T(n-1)+\sqrt{n}\\
&=\frac{n}{n+1}\left(\frac{n-1}{n}T(n-2)+\sqrt{n-1}\right)+\sqrt{n}\\
&=\frac{n-1}{n+1}T(n-2)+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3)+\sqrt{n-2}\right)+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-2}{n+1}T(n-3)+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-2}{n+1}\left(\frac{n-3}{n-2}T(n-4)+\sqrt{n-3}\right)+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}\\
&=\frac{n-3}{n+1}T(n-4)+\frac{(n-2)\sqrt{n-3}}{n+1}+\frac{(n-1)\sqrt{n-2}}{n+1}+\frac{n\sqrt{n-1}}{n+1}+\sqrt{n}
\end{aligned} \tag{3.105}$$

If we go on like that down to $T(1)$, (3.105) unfolds into

$$\begin{aligned}
T(n) &= \frac{2}{n+1}T(1) + \frac{3\sqrt{2}}{n+1} + \frac{4\sqrt{3}}{n+1} + \ldots \\
&\qquad + \frac{(n-2)\sqrt{n-3}}{n+1} + \frac{(n-1)\sqrt{n-2}}{n+1} + \frac{n\sqrt{n-1}}{n+1} + \sqrt{n} \\
&= \frac{2}{n+1}T(1) + \frac{3\sqrt{2}}{n+1} + \frac{4\sqrt{3}}{n+1} + \ldots \\
&\qquad + \frac{(n-2)\sqrt{n-3}}{n+1} + \frac{(n-1)\sqrt{n-2}}{n+1} + \frac{n\sqrt{n-1}}{n+1} + \frac{(n+1)\sqrt{n}}{n+1} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\sum_{i=2}^{n}(i+1)\sqrt{i} \\
&= \frac{2T(1)}{n+1} + \frac{1}{n+1}\left(\left(\sum_{i=1}^{n}(i+1)\sqrt{i}\right) - 2\right) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n}(i+1)\sqrt{i} \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\sum_{i=1}^{n}(i\sqrt{i}+\sqrt{i}) \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\sum_{i=1}^{n} i\sqrt{i} + \sum_{i=1}^{n}\sqrt{i}\right) \qquad (3.106)
\end{aligned}$$

But we know that

$$\sum_{i=1}^{n}\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{3}{2}}\right) \qquad \text{by (6.5) on page 160.}$$

$$\sum_{i=1}^{n}\left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{3}{2}}\right) \qquad \text{by (6.7) on page 162.}$$

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{5}{2}}\right) \qquad \text{by (6.10) on page 164.}$$

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{5}{2}}\right) \qquad \text{by (6.14) on page 167.}$$

Therefore, regardless of whether "$\sqrt{n}$" in (3.104) stands for $\lfloor\sqrt{n}\rfloor$ or $\lceil\sqrt{n}\rceil$,

$$\begin{aligned}
T(n) &= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\Theta\left(n^{\frac{5}{2}}\right) + \Theta\left(n^{\frac{5}{2}}\right)\right) \text{ by substituting into (3.106)} \\
&= \frac{2T(1)-2}{n+1} + \frac{1}{n+1}\left(\Theta\left(n^{\frac{5}{2}}\right)\right) \\
&= O(1) + \Theta\left(n^{\frac{3}{2}}\right)
\end{aligned}$$

So, $T(n) = \Theta\left(n^{\frac{3}{2}}\right)$. □

Problem 71. *Solve by unfolding*

$$T(n) = \frac{n}{n+1}T(n-1) + \lg n \tag{3.107}$$

Solution:

$$\begin{aligned}
T(n) &= \frac{n}{n+1}T(n-1) + \lg n \\
&= \frac{n}{n+1}\left(\frac{n-1}{n}T(n-2) + \lg(n-1)\right) + \lg n \\
&= \frac{n-1}{n+1}T(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \frac{n-1}{n+1}\left(\frac{n-2}{n-1}T(n-3) + \lg(n-2)\right) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \frac{n-2}{n+1}T(n-3) + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n \\
&= \dots \\
&= \underbrace{\frac{2}{n+1}T(1)}_{A} + \underbrace{\frac{3}{n+1}\lg 2 + \frac{4}{n+1}\lg 3 + \ldots + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \lg n}_{B}
\end{aligned}$$

Clearly, $A = O(1)$. Consider B.

$$\begin{aligned}
B &= \frac{3}{n+1}\lg 2 + \frac{4}{n+1}\lg 3 + \ldots + \frac{n-1}{n+1}\lg(n-2) + \frac{n}{n+1}\lg(n-1) + \frac{n+1}{n+1}\lg n \\
&= \frac{1}{n+1}\underbrace{(3\lg 2 + 4\lg 3 + \ldots + (n-1)\lg(n-2) + n\lg(n-1) + (n+1)\lg n)}_{C}
\end{aligned}$$

Now consider C.

$$\begin{aligned}
C &= 3\lg 2 + 4\lg 3 + \ldots + (n-1)\lg(n-2) + n\lg(n-1) + (n+1)\lg n \\
&= \underbrace{\lg 2 + \lg 3 + \ldots + \lg(n-1) + \lg n}_{D} + \underbrace{2\lg 2 + 3\lg 3 + \ldots + (n-1)\lg(n-1) + n\lg n}_{E}
\end{aligned}$$

But $D = \Theta(n\lg n)$ (see Problem 120 on page 156) and $E = \Theta(n^2\lg n)$ (see Problem 121 on page 156). It follows that $C = \Theta(n^2\lg n)$, and so $B = \Theta(n\lg n)$. We conclude that

$$T(n) = \Theta(n\lg n) \qquad \square$$

Problem 72. *Solve*

$$T(1) = \Theta(1) \tag{3.108}$$
$$T(2) = \Theta(1) \tag{3.109}$$
$$T(n) = T(n-1).T(n-2) \tag{3.110}$$

Solution:
Unlike the problems we encountered so far, the aymptotic growth rate of $T(n)$ in this problem depends on the concrete values of the constants in (3.108) and (3.109). It is easy to see that if $T(1) = T(2) = 1$ then $T(n) = 1$ for all positive n. So let us postulate that

$$T(1) = c \tag{3.111}$$
$$T(2) = d \tag{3.112}$$

where c and d are some positive constants. Then

$$T(3) = T(2).T(1) = cd$$
$$T(4) = T(3).T(2) = cd^2$$
$$T(5) = T(4).T(3) = c^2d^3$$
$$T(6) = T(5).T(4) = c^3d^5$$
$$T(7) = T(6).T(5) = c^5d^8$$
$$\dots$$

The degrees that appear in this sequence look like the Fibonacci number (see the definition on page 167). Indeed, it is trivial to prove by induction that

$$T(1) = c$$
$$T(n) = d^{F_{n-1}}c^{F_{n-2}}, \text{ for all } n > 1 \tag{3.113}$$

Define

$$a = c^{\frac{1}{\sqrt{5}}}$$
$$b = d^{\frac{1}{\sqrt{5}}}$$

and derive

$$\begin{aligned} T(n) &= \Theta\left(b^{\phi^{n-1}}\right)\Theta\left(a^{\phi^{n-2}}\right) \qquad \text{applying (6.15) on page 168 on (3.113)} \\ &= \Theta\left(b^{\phi^{n-1}}a^{\phi^{n-2}}\right) \\ &= \Theta\left(b^{\phi.\phi^{n-2}}a^{\phi^{n-2}}\right) \\ &= \Theta\left(k^{\phi^{n-2}}a^{\phi^{n-2}}\right) \qquad \text{defining that } b^{\phi} = k \\ &= \Theta\left((ak)^{\phi^{n-2}}\right) \end{aligned} \tag{3.114}$$

Depending on how detalied analysis we need, we may stop right here. However, we can go on a little further because depending on what a and k are, (3.113) can have dramatically different asymptotic growth.

- If $ak > 1$, $T(n) \xrightarrow[n \to +\infty]{} \infty$.
- If $ak = 1$, $T(n) = 1$ for all positive n, thus $T(n) = \Theta(1)$.
- If $ak < 1$, $T(n) \xrightarrow[n \to +\infty]{} 0$, thus $T(n) = O(1)$. □

Problem 73. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{n-1} T(i) + 1$$

Solution:
By definition,

$$T(n) = T(n-1) + T(n-2) + \ldots + T(2) + T(1) + 1 \quad (3.115)$$
$$T(n-1) = \quad T(n-2) + \ldots + T(2) + T(1) + 1 \quad (3.116)$$

Subtract 3.116 from 3.115 to obtain

$$T(n) - T(n-1) = T(n-1)$$

So, the original recurrence is equivalent to the following one:

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n-1)$$

It is trivial to show that $T(n) = \Theta(2^n)$, either by induction or by the method with the characteristic equation. □

Problem 74. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = \sum_{i=1}^{n-1} (T(i) + T(n-i)) + 1$$

Solution:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} (T(i) + T(n-i)) + 1 \\ &= \underbrace{\sum_{i=1}^{n-1} T(i)}_{T(1)+T(2)+\ldots+T(n-1)} + \underbrace{\sum_{i=1}^{n-1} T(n-i)}_{T(n-1)+T(n-2)+\ldots+T(1)} + 1 \\ &= 2\sum_{i=1}^{n-1} T(i) + 1 \end{aligned}$$

Having in mind the latter result, we proceed as in the previous problem.

$$T(n) = 2T(n-1) + 2T(n-2) + \ldots + 2T(2) + 2T(1) + 1 \quad (3.117)$$
$$T(n-1) = \quad 2T(n-2) + \ldots + 2T(2) + 2T(1) + 1 \quad (3.118)$$

Subtract 3.118 from 3.117 to obtain

$$T(n) - T(n-1) = 2T(n-1)$$

So, the original recurrence is equivalent to the following one:

$$T(1) = \Theta(1)$$
$$T(n) = 3T(n-1)$$

It is trivial to show that $T(n) = \Theta(3^n)$, either by induction or by the method with the characteristic equation. □

Problem 75. *Solve*

$$T(1) = \Theta(1)$$
$$T(n) = nT(n-1) + 1$$

Solution:

$$\begin{aligned}
T(n) &= nT(n-1) + 1 \\
&= n((n-1)T(n-2) + 1) + 1 \\
&= n(n-1)T(n-2) + n + 1 \\
&= n(n-1)((n-2)T(n-3) + 1) + n + 1 \\
&= n(n-1)(n-2)T(n-3) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)((n-3)T(n-4) + 1) + n(n-1) + n + 1 \\
&= n(n-1)(n-2)(n-3)T(n-4) + n(n-1)(n-2) + n(n-1) + n + 1 \\
&= \dots \\
&= \frac{n!}{(n-i)!}T(n-i) + \frac{n!}{(n-i+1)!} + \frac{n!}{(n-i+2)!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \qquad (3.119)
\end{aligned}$$

Clearly, the maximum value i achieves is $i_{\max} = n-1$. For $i = i_{\max}$, (3.119) becomes:

$$\begin{aligned}
T(n) &= \frac{n!}{1!}T(1) + \frac{n!}{2!} + \frac{n!}{3!} + \dots + \frac{n!}{(n-1)!} + \frac{n!}{n!} \\
&= n! \times \underbrace{\left(\frac{T(1)}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{(n-1)!} + \frac{1}{n!}\right)}_{A}
\end{aligned}$$

We claim A is bounded by a constant. To see why, note that the series $\sum_{i=1}^{\infty} \frac{1}{i!}$ is convergent because the geometric series $\sum_{i=1}^{\infty} \frac{1}{2^i}$ is convergent and $i! > 2^i$ for all $i > 3$. Therefore,

$$T(n) = \Theta(n!)$$

□

3.2.2 The Master Theorem

There are several theoretical results solving a broad range of recurrences corresponding to divide-and-conquer algorithms that are called master theorems. The one stated below is due to [CLR00]. There is a considerately more powerful master theorem due to Akra and Bazzi [AB98]. See [Lei96] for a detailed explanation.

Theorem 1 (Master Theorem, [CLR00], pp. 62)**.** *Let* $a \geq 1$ *and* $b > 1$ *be constants, let* $k = \lg_b a$*, and let* $f(n)$ *be a positive function. Let*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = \Theta(1)$$

where $\frac{n}{b}$ *is interpreted either as* $\left\lfloor \frac{n}{b} \right\rfloor$ *or* $\left\lceil \frac{n}{b} \right\rceil$*. Then* $T(n)$ *can be bounded asymptotically as follows.*

Case 1 *If* $f(n) = O\left(n^{k-\epsilon}\right)$ *for some positive constant* ϵ *then* $T(n) = \Theta(n^k)$.

Case 2 *If* $f(n) = \Theta(n^k)$ *then* $T(n) = \Theta\left(n^k . \lg n\right)$.

Case 3 *If both*

1. $f(n) = \Omega\left(n^{k+\epsilon}\right)$ *for some positive constant* ϵ*, and*
2. $a.f\left(\frac{n}{b}\right) \leq c.f(n)$ *for some constant* c *such that* $0 < c < 1$ *and for all sufficiently large* n*,*

then $T(n) = \Theta(f(n))$. □

Condition 3-2 is known as *the regularity condition.*

Note that the condition $f(n) = O\left(n^{k-\epsilon}\right)$ is stronger than $f(n) = o(n^k)$ and $f(n) = \Omega\left(n^{k+\epsilon}\right)$ is stronger than $f(n) = \omega\left(n^k\right)$:

$$\begin{aligned} f(n) = O(n^{k-\epsilon}) &\Rightarrow f(n) = o(n^k) \\ f(n) = o(n^k) &\not\Rightarrow f(n) = O(n^{k-\epsilon}) \\ f(n) = \Omega(n^{k+\epsilon}) &\Rightarrow f(n) = \omega(n^k) \\ f(n) = \omega(n^k) &\not\Rightarrow f(n) = \Omega(n^{k+\epsilon}) \end{aligned}$$

For example, consider that

$$n \lg n = \omega(n) \tag{3.120}$$
$$n \lg n \neq \Omega(n^{1+\epsilon}) \text{ for any } \epsilon > 0 \text{ because } \lg n \neq \Omega(n^{\epsilon}) \text{ by (1.50)} \tag{3.121}$$
$$\frac{n}{\lg n} = o(n) \tag{3.122}$$
$$\frac{n}{\lg n} \neq O(n^{1-\epsilon}) \text{ for any } \epsilon > 0 \text{ because } \frac{1}{\lg n} \neq O(n^{-\epsilon}) \tag{3.123}$$

To see why $\frac{1}{\lg n} \neq O(n^{-\epsilon})$ in (3.123) consider that

$$\lim_{n\to\infty} \frac{\lg n}{n^\epsilon} = 0 \Rightarrow \lim_{n\to\infty} \left(\frac{\frac{1}{n^\epsilon}}{\frac{1}{\lg n}} \right) = 0 \Rightarrow \frac{1}{n^\epsilon} = o\left(\frac{1}{\lg n}\right) \text{ by (1.6)} \Rightarrow$$

$$\frac{1}{\lg n} = \omega\left(\frac{1}{n^\epsilon}\right) \text{ by the transpose symmetry}$$

Problem 76. *Solve by the Master Theorem*

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Solution:
Using the terminology of the Master Theorem, a is 4, b is 2, thus $\log_b a$ is $\log_2 4 = 2$ and $n^{\log_b a}$ is n^2. The function $f(n)$ is n. The theorem asks us to compare $f(n)$ and $n^{\log_b a}$, which, in the current case, is to compare n with n^2. Clearly, $n = O(n^{2-\epsilon})$ for some $\epsilon > 0$, so Case 1 of the Master Theorem is applicable and $T(n) = n^2$. □

Problem 77. *Solve by the Master Theorem*

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

Solution:
Rewrite the recurrence as

$$T(n) = 1.T\left(\frac{n}{\frac{3}{2}}\right) + 1$$

Using the terminology of the Master Theorem, a is 1, b is $\frac{3}{2}$, thus $\log_b a$ is $\log_{\frac{3}{2}} 1 = 0$ and $n^{\log_b a}$ is $n^0 = 1$. The function $f(n)$ is n. Clearly, $1 = \Theta(n^0)$, so Case 2 of the Master Theorem is applicable. Assording to it, $T(n) = \Theta(1.\lg n) = \Theta(\lg n)$. □

Problem 78. *Solve*

$$T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$$

Solution:
Using the terminology of the Master Theorem, a is 3, b is 4, thus $\log_b a$ is $\log_4 3$, which is approximately 0.79, and the function $f(n)$ is $n\lg n$. It certainly is true that $n\lg n = \Omega(n^{\log_4 3+\epsilon})$ for some $\epsilon > 0$, for instance $\epsilon = 0.1$. However, we have to check the regularity condition to see if Case 3 of the Master Theorem is aplicable. The regularity condition in this case is:

$$\exists c \text{ such that } 0 < c < 1 \text{ and } 3\frac{n}{4}\lg\frac{n}{4} \leq cn\lg n \text{ for all sufficiently large } n$$

The latter clearly holds for, say, $c = \frac{3}{4}$, therefore Case 3 is applicable and according to it, $T(n) = \Theta(n\lg n)$. □

Problem 79. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$

Solution:
Let us the try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 2 and b is 2, thus $\log_b a$ is $\log_2 2 = 1$, therefore $n^{\log_b a}$ is $n^1 = n$. The function $f(n)$ is $n \lg n$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $n \lg n = O(n^{1-\epsilon})$ for some $\epsilon > 0$? No, because $n \lg n = \omega(n^1)$.

try Case 2 Is it true that $n \lg n = \Theta(n^1)$? No, because $n \lg n = \omega(n^1)$.

try Case 3 Is it true that $n \lg n = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$? No, see (3.121).

Therefore this problem cannot be solved using the Master Theorem as stated above. We solve it by Theorem 2 on page 91 and the answer is $T(n) = \Theta(n \lg^2 n)$. □

Problem 80. *Solve*

$$T(n) = 4T\left(\frac{n}{2}\right) + n \tag{3.124}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \tag{3.125}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \tag{3.126}$$

$$\tag{3.127}$$

Solution:
Using the terminology of the Master Theorem, a is 4 and b is 2, thus $\log_b a$ is $\log_4 2 = 2$, therefore $n^{\log_b a}$ is n^2. With respect to (3.124), it is the case that $n = O(n^{2-\epsilon})$ for some $\epsilon > 0$, therefore the solution of (3.124) is $T(n) = \Theta(n^2)$ by Case 1 of the Master Theorem. With respect to (3.125), it is the case that $n^2 = \Theta(n^2)$, therefore the solution of (3.125) is $T(n) = \Theta(n^2 \lg n)$ by Case 2 of the Master Theorem. With respect to (3.126), it is the case that $n^3 = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$, therefore the solution of (3.126) is $T(n) = \Theta(n^3)$ by Case 3 of the Master Theorem, *provided* the regularity condition holds. The regularity condition here is

$$\exists c \text{ such that } 0 < c < 1 \text{ and } 4\left(\frac{n}{2}\right)^3 \leq cn^3 \text{ for all sufficiently large } n$$

Clearly that holds for any c such that $\frac{1}{2} \leq c < 1$. Therefore, by Case 3 of the Master Theorem, the solution of (3.126) is $T(n) = \Theta(n^3)$. □

Problem 81. *Solve*

$$T(n) = T\left(\frac{n}{2}\right) + \lg n \tag{3.128}$$

Solution:
Let us try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 1 and b is 2, thus $\log_b a$ is $\log_2 1 = 0$, therefore $n^{\log_b a}$ is $n^0 = 1$. The function $f(n)$ is $\lg n$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $\lg n = O(n^{0-\epsilon})$ for some $\epsilon > 0$? No, because $\lg n$ is an increasing function and $n^{-\epsilon} = \frac{1}{n^\epsilon}$ is a decreasing one.

try Case 2 Is it true that $\lg n = \Theta(n^0)$? No, because $\lg n = \omega(n^0)$.

try Case 3 Is it true that $\lg n = \Omega(n^{0+\epsilon})$ for some $\epsilon > 0$? No, see (1.50) on page 13.

So the Master Theorem is not applicable and we seek other methods for solving. Substitute n by 2^m, *i.e.* $m = \lg n$ and $m = \lg n$. Then (3.128) becomes

$$T(2^m) = T\left(2^{m-1}\right) + m \tag{3.129}$$

Further substitute $T(2^m)$ by $S(m)$ and (3.129) becomes

$$S(m) = S(m-1) + m \tag{3.130}$$

But that recurrence is the same as (3.19), therefore its solution is $S(m) = \Theta(m^2)$. Let us go back now to the original n and $T(n)$.

$$S(m) = \Theta(m^2) \;\Leftrightarrow\; T(2^m) = \Theta(\lg^2 n) \;\Leftrightarrow\; T(n) = \Theta(\lg^2 n)$$

□

Problem 82. *Solve by the Master Theorem*

$$T(n) = 2T\left(\frac{n}{2}\right) + n^3 \tag{3.131}$$

$$T(n) = T\left(\frac{9n}{10}\right) + n \tag{3.132}$$

$$T(n) = 16T\left(\frac{n}{4}\right) + n^2 \tag{3.133}$$

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \tag{3.134}$$

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \tag{3.135}$$

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} \tag{3.136}$$

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n} \tag{3.137}$$

$$T(n) = 8T\left(\frac{n}{2}\right) + n^3 \tag{3.138}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + 2n^2 \tag{3.139}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + n\lg n \tag{3.140}$$

Solution:
(3.131): as $n^3 = \Omega\left(n^{\log_2 2+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $2\left(\frac{n}{2}\right)^3 \leq cn^3 \Leftrightarrow \frac{1}{4} \leq c$. So, any c such that $\frac{1}{4} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^3)$.

(3.132): rewrite the recurrence as $T(n) = 1.T\left(\frac{n}{\frac{10}{9}}\right) + n$. As $n = \Omega\left(n^{\left(\log_{\frac{10}{9}} 1\right)+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $1\left(\frac{n}{\frac{10}{9}}\right) \leq cn \Leftrightarrow \frac{9}{10} \leq c$. So, any c such that $\frac{9}{10} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n)$.

(3.133): As $n^2 = \Theta\left(n^{\log_4 16}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = n^2 \lg n$.

(3.134): as $n^2 = \Omega\left(n^{\log_3 7+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $7\left(\frac{n}{3}\right)^2 \leq cn^2 \Leftrightarrow \frac{7}{9} \leq c$. So, any c such that $\frac{7}{9} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^2)$.

(3.135): as $n^2 = O\left(n^{\log_2 7-\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 1 of the Master Theorem and so $T(n) = \Theta\left(n^{\log_2 7}\right)$.

(3.136): as $\sqrt{n} = \Theta\left(n^{\log_4 2}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = \Theta(\sqrt{n} \lg n)$.

(3.137): as $n^{\frac{5}{2}} = \Omega\left(n^{\log_2 4+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $4\left(\frac{n}{2}\right)^{\frac{5}{2}} \leq cn^{\frac{5}{2}} \Leftrightarrow \frac{1}{\sqrt{2}} \leq c$. So, any c such that $\frac{1}{\sqrt{2}} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(n^2\sqrt{n})$.

(3.138): As $n^3 = \Theta\left(n^{\log_2 8}\right)$, we classify the problem into Case 2 of the Master Theorem and so $T(n) = n^3 \lg n$.

(3.139): as $2n^2 = \Omega\left(n^{\log_2 3+\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 3 of the Master Theorem. To apply Case 3, we have to check the regularity condition holds. Namely, there is a constant c such that $0 < c < 1$ and $3\left(2\left(\frac{n}{2}\right)^2\right) \leq c2n^2 \Leftrightarrow 3 \leq 4c$. So, any c such that $\frac{3}{4} \leq c < 1$ will do, therefore the regularity condition holds, therefore Case 3 is applicable, therefore $T(n) = \Theta(2n^2) = \Theta(n^2)$.

(3.140): as $n \lg n = O\left(n^{\log_2 3-\epsilon}\right)$ for some $\epsilon > 0$, we classify the problem into Case 1 of the Master Theorem and so $T(n) = \Theta\left(n^{\log_2 3}\right)$. □

The following result extends Case 2 of the Master Theorem.

Theorem 2. *Under the premises of Theorem 1, assume*

$$f(n) = \Theta(n^k \lg^t n) \tag{3.141}$$

for some constant $t \geq 0$. *Then*

$$T(n) = \Theta(n^k \lg^{t+1} n)$$

Proof:
Theorem 1 itself is not applicable because the recurrence for the said $f(n)$ cannot be classified into any of the three cases there. To solve the problem we use unfolding. For simplicity we assume that n is an exact power of b, *i.e.* $n = b^m$ for some integer $m > 0$. The same technique is used in [CLR00] for proving the Master Theorem: first prove it for exact powers of b and then prove the result holds for any positive n. Here we limit our proof to the case that n is an exact power of b and leave it to the reader to generalise for any positive n.

Assume that the logarithm in (3.141) is base-b and note we can rewrite what is inside the Θ-notation on the right-hand side of (3.141) in the following way:

$$n^k \log_b^t n = n^{\log_b a} (\log_b b^m)^t = b^{(m \log_b a)} m^t = b^{(\log_b a^m)} m^t = a^m m^t \tag{3.142}$$

Then (3.141) is equivalent to saying that

$$c_1 a^m m^t \leq f(b^m) \leq c_2 a^m m^t$$

for some positive constants c_1 and c_2 and all sufficiently large values of m. However, for the sake of simplicity, we will assume in the remainder of the proof that

$$f(b^m) = a^m m^t \tag{3.143}$$

The reader is invited to construct a proof for the general case.

By the definition of the Master Theorem, $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Using (3.143) we rewrite that as follows.

$$\begin{aligned}
T(b^m) &= aT\left(\frac{b^m}{b}\right) + a^m m^t \\
&= aT(b^{m-1}) + a^m m^t \;\Leftrightarrow \\
S(m) &= aS(m-1) + a^m m^t \qquad \text{substituting } T(b^m) \text{ with } S(m) \\
&= a\big(aS(m-2) + a^{m-1}(m-1)^t\big) + a^m m^t \\
&= a^2 S(m-2) + a^m (m-1)^t + a^m m^t \\
&= a^2\big(aS(m-3) + a^{m-2}(m-2)^t\big) + a^m (m-1)^t + a^m m^t \\
&= a^3 S(m-3) + a^m (m-2)^t + a^m (m-1)^t + a^m m^t \\
&\dots \\
&= a^{m-1} S(1) + a^m 2^t + a^m 3^t + \ldots + a^m (m-2)^t + a^m (m-1)^t + a^m m^t \\
&= a^{m-1} S(1) - a^m + a^m \underbrace{(1^t + 2^t + 3^t + \ldots + (m-2)^t + (m-1)^t + m^t)}_{\Theta(m^{t+1}) \text{ by (6.20) on page 168}} \\
&= a^{m-1} S(1) - a^m + a^m \Theta(m^{t+1}) \\
&= a^{m-1} S(1) - a^m + \Theta(a^m m^{t+1})
\end{aligned} \tag{3.144}$$

But (3.144) is $\Theta(a^m m^{t+1})$ because $a^m m^{t+1} = \omega(|a^{m-1}S(1) - a^m|)$. So,

$$S(m) = \Theta(a^m m^{t+1}) \Leftrightarrow T(n) = \Theta\left(a^{\log_b n}(\log_b n)^{t+1}\right)$$

Having in mind that $a^{\log_b n} = n^{\log_b a}$ and $\log_b n = \Theta(\lg n)$, we conclude that

$$T(n) = \Theta\left(n^{\log_b a} \lg^{t+1} n\right)$$

□

Problem 83. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n$$

Solution:
Since $\lg n = O\left(n^{\log_2 2-\epsilon}\right)$ for some $\epsilon > 0$, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$ by Case 1 of the Master Theorem. □

Problem 84. *Solve*

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$$

Solution:
Let us the try to solve it using the Master Theorem. Using the terminology of the Master Theorem, a is 2 and b is 2, thus $\log_b a$ is $\log_2 2 = 1$, therefore $n^{\log_b a}$ is $n^1 = n$. The function $f(n)$ is $\frac{n}{\lg n}$. Let us see if we can classify that problem in one of the three cases of the Master Theorem.

try Case 1 Is it true that $\frac{n}{\lg n} = O(n^{1-\epsilon})$ for some $\epsilon > 0$? No, see (3.123) on page 86.

try Case 2 Is it true that $\frac{n}{\lg n} = \Theta(n^1)$? No, because $n \lg n = o(n^1)$.

try Case 3 Is it true that $\frac{n}{\lg n} = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$? No, because $n \lg n = o(n^1)$.

Therefore this problem cannot be solved using the Master Theorem as stated above. Furthermore, Theorem 2 on the preceding page cannot be applied either because it is not true that $\frac{n}{\lg n} = \Theta(n^{\log_2 2} \lg^t(n))$ for any $t \geq 0$.

We solve the problem by unfolding.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{\frac{n}{2}}{\lg\frac{n}{2}}\right) + \frac{n}{\lg n} \\
&= 4T\left(\frac{n}{4}\right) + \frac{n}{(\lg n)-1} + \frac{n}{\lg n} \\
&= 4\left(2T\left(\frac{n}{8}\right) + \frac{\frac{n}{4}}{\lg\frac{n}{4}}\right) + \frac{n}{(\lg n)-1} + \frac{n}{\lg n} \\
&= 8T\left(\frac{n}{8}\right) + \frac{n}{(\lg n)-2} + \frac{n}{(\lg n)-1} + \frac{n}{\lg n} \\
&\dots \\
&= nT(1) + \frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{(\lg n)-2} + \frac{n}{(\lg n)-1} + \frac{n}{\lg n} \\
&= nT(1) + \underbrace{n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{(\lg n)-2} + \frac{1}{(\lg n)-1} + \frac{1}{\lg n}\right)}_{B}
\end{aligned}$$

Clearly, $|A| = O(n)$. Now observe that $B = n.H_{\lg n}$ because inside the parentheses is the $(\lg n)^{\text{th}}$ partial sum of the harmonic series (see 6.16 on page 168). By (6.17), $H_{\lg n} = \Theta(\lg\lg n)$, therefore $B = \Theta(n \lg\lg n)$, therefore $T(n) = \Theta(n \lg\lg n)$. □

Problem 85 ([CLR00], Problem 4.4-3)**.** *Show that case 3 of the master theorem is overstated, in the sense that the regularity condition* $af\left(\frac{n}{b}\right) \le cf(n)$ *for some constant* $c < 1$ *implies that there exists a constant* $\epsilon > 0$ *such that* $f(n) = \Omega(n^{\log_b a+\epsilon})$*.*

Solution:
Assume for some constant c such that $0 < c < 1$,

$$f(n) \ge \frac{a}{c} f\left(\frac{n}{b}\right)$$

Then

$$\begin{aligned}
f(n) &\ge \frac{a^2}{c^2} f\left(\frac{n}{b^2}\right) \\
f(n) &\ge \frac{a^3}{c^3} f\left(\frac{n}{b^3}\right) \\
&\dots \\
f(n) &\ge \frac{a^t}{c^t} f\left(\frac{n}{b^t}\right) \qquad (3.145)
\end{aligned}$$

For some constant value n_0 for n, that process stops. So, $\frac{n}{b^t} = n_0$, therefore $t = \log_b\left(\frac{n}{n_0}\right) = \log_b n - \log_b n_0$. Let $s = \frac{1}{c}$. Since $c < 1$, it is the case that $s > 1$. Substitute t, $\frac{n}{b^t}$, and c in (3.145) to obtain

$$f(n) \ge \frac{a^{\log_b n}}{a^{\log_b n_0}} \times \frac{s^{\log_b n}}{s^{\log_b n_0}} \times f(n_0)$$

Note that $\frac{1}{a^{\log_b n_0}} \times \frac{1}{s^{\log_b n_0}} \times f(n_0)$ is a constant. Call that constant, β. So,

$$f(n) \geq a^{\log_b n} \times s^{\log_b n} \times \beta$$

Having in mind that $a^{\log_b n} = n^{\log_b a}$ and $s^{\log_b n} = n^{\log_b s}$, we see that

$$f(n) \geq \beta \times n^{\log_b a} \times n^{\log_b s}$$

Since $s > 1$, $\log_b s > 0$. Let $\epsilon = \log_b s$. We derive the desired result: for all sufficiently large n and some constant β

$$f(n) \geq \beta n^{\log_b a + \epsilon} \Rightarrow f(n) = \Omega(n^{\log_b a + \epsilon})$$

□

3.2.3 The Method with the Characteristic Equation

Theorem 3 ([Man05], pp. 57). *Let the infinite sequence* $\widetilde{a} = a_0, a_1, a_2, \ldots$ *be generated by the linear recurrence relation*

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_{r-1} a_{n-(r-1)} + c_r a_{n-r} \tag{3.146}$$

Let α_1, α_2, *...*, α_s *be the distinct complex roots of the* characteristic *equation*

$$x^r - c_1 x^{r-1} - c_2 x^{r-2} - \ldots - c_{r-1} x - c_r = 0 \tag{3.147}$$

where α_i *has multiplicity* k_i *for* $1 \leq i \leq s$[†]. *Then*

$$a_n = P_1(n)\alpha_1^n + P_2(n)\alpha_2^n + \ldots + P_s(n)\alpha_s^n \tag{3.148}$$

where $P_i(n)$ *is a polynomial of* n *of degree* $< k_i$. *The polynomials* $P_1(n)$, $P_2(n)$, *...*, $P_s(n)$ *have* r *coefficients altogether which coefficients are determined uniquely by the first* r *elements of* $\widetilde{a}$. □

Using our terminology, (3.146) would be rewritten as

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \ldots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \tag{3.149}$$

This is a generic instance of *homogeneous* linear recurrence. It is hardly possible such a recurrence relation to describe the running time of a recursive algorithm since after the recursive calls finish, at least some constant-time work must be perfomed. Therefore, we will consider more general recurrence relations that are *nonhomogeneous* and whose generic form is:

$$\begin{aligned} T(n) = {} & c_1 T(n-1) + c_2 T(n-2) + \ldots + c_{r-1} T(n-(r-1)) + c_r T(n-r) \\ & + b_1^n Q_1(n) + b_2^n Q_2(n) + \ldots + b_m^n Q_m(n) \end{aligned} \tag{3.150}$$

$b_1, b_2, \ldots, b_m$ are distinct positive constants and $Q_1(n), Q_2(n), \ldots, Q_m(n)$ are polynomials of n of degrees $d_1, d_2, \ldots, d_m$, respectively.

[†]Clearly, $k_i \geq 1$ for $1 \leq i \leq s$ and $k_1 + k_2 + \ldots + k_s = r$.

Let us denote multisets by $\{\quad\}_M$ brackets, *e.g.* $\{1,1,2,3,3,3\}_M$. For each element a from some mulitset A, let $\#(a,A)$ denote the number of occurrences of a in A. For example, $\#(1,\{1,1,2,3,3,3\}) = 2$. The union of two multisets A and B adds the multiplicities of the elements, that is,

$$A \cup B = \{x \,|\, (x \in A \text{ or } x \in B) \text{ and } (\#(x, A \cup B) = \#(x, A) + \#(x, B))\}_M$$

The cardinality of a multiset A is the sum of the multiplicities of its elements and is denoted by $|A|$. For example, $|\{1,1,2,3,3,3\}_M| = 6$.

The solution of (3.150) is the following. Let the multiset of the roots of the characteristic equation be A. Clearly, $|A| = r$. Let $B = \{b_i \,|\, \#(b,B) = d_i + 1\}_M$. Let $Y = A \cup B$. Clearly, $|Y| = r + \sum_{i=1}^{m}(d_i+1)$. Let us rename the distinct elements of Y as $y_1, y_2, \ldots, y_t$ and define that $\#(y_i, Y) = z_i$, for $1 \le i \le t$. Then

$$\begin{aligned} T(n) = {} & \beta_{1,1}\, y_1^n + \beta_{1,2}\, n\, y_1^n + \ldots + \beta_{1,z_1}\, n^{z_1}\, y_1^n + \\ & \beta_{2,1}\, y_2^n + \beta_{2,2}\, n\, y_2^n + \ldots + \beta_{2,z_2}\, n^{z_2}\, y_2^n + \\ & \ldots \\ & \beta_{t,1}\, y_1^n + \beta_{t,2}\, n\, y_t^n + \ldots + \beta_{t,z_t}\, n^{z_t}\, y_1^n \end{aligned} \tag{3.151}$$

The indexed β's are constants, $|Y|$ in number. Since we are interested in the asymptotic growth rate of $T(n)$ we do not care what are the precise values of those constants. As we do not specify concrete initial conditions we are not able to compute them precisely anyways. The asymptotic growth rate is determined by precisely one of all those terms—namely, the biggest y_i, multiplied by the biggest degree of n.

Problem 86. *Solve*

$$T(n) = T(n-1) + 1$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^0$ to make sure its form is as required by (3.150). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 0$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1,1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n$ for some constants A and B, therefore $T(n) = \Theta(n)$. □

Problem 87. *Solve*

$$T(n) = T(n-1) + n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^1$ to make sure its form is as required by (3.150). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the

multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n$ for some constants A, B, and C, therefore $T(n) = \Theta(n^2)$. □

Problem 88. *Solve*

$$T(n) = T(n-1) + n^4$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n n^4$ to make sure its form is as required by (3.150). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 4$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 5$ to $\{1\}_M$, obtaining $\{1, 1, 1, 1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n + D\,n^3\,1^n + E\,n^4\,1^n + F\,n^5\,1^n$ for some constants A, B, C, D, and F, therefore $T(n) = \Theta(n^5)$. □

Problem 89. *Solve*

$$T(n) = T(n-1) + 2n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 1^n(2n^1)$ to make sure its form is as required by (3.150). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 1\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,n^2\,1^n$ for some constants A, B, and C, therefore $T(n) = \Theta(n^2)$. □

Problem 90. *Solve*

$$T(n) = T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = T(n-1) + 2^n n^0$ to make sure its form is as required by (3.150). The characteristic equation is $x - 1 = 0$ with a single root $x_1 = 1$. So, the multiset of the roots of the characteristic equation is $\{1\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1, 2\}_M$. Then $T(n) = A\,1^n + B\,2^n$ for some constants A and B, therefore $T(n) = \Theta(2^n)$. □

Problem 91. *Solve*

$$T(n) = 2T(n-1) + 1$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 1^n n^0$ to make sure its form is as required by (3.150). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 0$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 1$ to $\{1\}_M$, obtaining $\{1, 2\}_M$. Then $T(n) = A\,1^n + B\,2^n$ for some constants A and B, therefore $T(n) = \Theta(2^n)$. □

Problem 92. *Solve*

$$T(n) = 2T(n-1) + n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 1^n n^1$ to make sure its form is as required by (3.150). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 1$, and $d_1 = 1$. So we add $b_1 = 1$ with multiplicity $d_1 + 1 = 2$ to $\{1\}_M$, obtaining $\{1, 1, 2\}_M$. Then $T(n) = A\,1^n + B\,n\,1^n + C\,2^n$ for some constants A, B, and C, therefore $T(n) = \Theta(2^n)$. □

Problem 93. *Solve*

$$T(n) = 2T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 2T(n-1) + 2^n n^0$ to make sure its form is as required by (3.150). The characteristic equation is $x - 2 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{2\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 0$ to $\{2\}_M$, obtaining $\{2, 2\}_M$. Then $T(n) = A\,2^n + B\,n\,2^n$ for some constants A and B, therefore $T(n) = \Theta(n2^n)$. □

Problem 94. *Solve*

$$T(n) = 3T(n-1) + 2^n$$

using the method of the characteristic equation.

Solution:
Rewrite the recurrence as $T(n) = 3T(n-1) + 2^n n^0$ to make sure its form is as required by (3.150). The characteristic equation is $x - 3 = 0$ with a single root $x_1 = 2$. So, the multiset of the roots of the characteristic equation is $\{3\}_M$. In the naming convention of (3.150), $m = 1$, $b_1 = 2$, and $d_1 = 0$. So we add $b_1 = 2$ with multiplicity $d_1 + 1 = 0$ to $\{3\}_M$, obtaining $\{2, 3\}_M$. Then $T(n) = A\,2^n + B\,3^n$ for some constants A and B, therefore $T(n) = \Theta(3^n)$. □

Problem 95. *Solve*

$$T(n) = T(n-1) + T(n-2)$$

using the method of the characteristic equation.

Solution:
This recurrence is homogeneous but is nevertheless interesting. The characteristic equation is $x^2 - x - 1 = 0$. The two roots are $x_1 = \frac{1+\sqrt{5}}{2}$ and $x_2 = \frac{1-\sqrt{5}}{2}$. Then $T(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$ for some constants A and B. Note that $\left|\frac{1-\sqrt{5}}{2}\right| < 1$ therefore $T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. □

Problem 96. *Solve*

$$T(n) = T(n-1) + 2T(n-2)$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - x - 2 = 0$. The two roots are $x_1 = \frac{1+3}{2} = 2$ and $x_2 = \frac{1-3}{2} = -1$. Then $T(n) = A\,2^n + B\,(-1)^n$ for some constants A and B. Therefore, $T(n) = \Theta(2^n)$. □

Problem 97. *Solve*

$$T(n) = 3T(n-1) + 4T(n-2) + 1 \tag{3.152}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2 - 3x - 4 = 0$. The two roots are $x_1 = \frac{3+5}{2} = 4$ and $x_2 = \frac{3-5}{2} = -1$. Then $T(n) = A\,4^n + B\,(-1)^n + C\,1^n$ for some constants A, B, and C. Therefore, $T(n) = \Theta(4^n)$. □

Problem 98. *Solve*

$$T(n) = 4T(n-1) + 3T(n-2) + 1 \tag{3.153}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2-4x-3=0$. The two roots are $x_1=2+\sqrt{7}$ and $x_2=2-\sqrt{7}$. Then $T(n)=A\,(2+\sqrt{7})^n+B\,(2-\sqrt{7})^n+C\,1^n$ for some constants A, B, and C. Therefore, $T(n)=\Theta((2+\sqrt{7})^n)$. □

Problem 99. *Solve*

$$T(n)=5T(n-1)+6T(n-2)+1 \tag{3.154}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is $x^2-5x-6=0$. The two roots are $x_1=\frac{5+\sqrt{5^2+24}}{2}=\frac{5+\sqrt{49}}{2}=\frac{5+7}{2}=6$ and $x_2=\frac{5-7}{2}=-1$. Then $T(n)=A.6^n+B.(-1)^n+C.1^n$. for some constants A, B, and C. Therefore, A, B C. Therefore, $T(n)=\Theta(6^n)$. □

Problem 100. *Solve*

$$T(n)=4T(n-3)+1 \tag{3.155}$$

using the method of the characteristic equation.

Solution:
The characteristic equation is

$$x^3-4=0$$

Its roots are

$$x_1=\sqrt[3]{4}$$
$$x_2=\sqrt[3]{4}\,e^{i\frac{\pi}{3}}$$
$$x_3=\sqrt[3]{4}\,e^{i\frac{-\pi}{3}}$$

If A, B, C, and D are some constants that can be complex numbers, the solution is

$$\begin{aligned}T(n)&=A\left(\sqrt[3]{4}\right)^n+B\left(\sqrt[3]{4}\right)^n e^{\frac{n\pi i}{3}}+C\left(\sqrt[3]{4}\right)^n e^{\frac{n\pi i}{3}}+D1^n=\\&=A\left(\sqrt[3]{4}\right)^n+B\left(\sqrt[3]{4}\right)^n\left(\cos\left(\frac{n\pi}{3}\right)+i\sin\left(\frac{n\pi}{3}\right)\right)+\\&\quad C\left(\sqrt[3]{4}\right)^n\left(\cos\left(\frac{-n\pi}{3}\right)+i\sin\left(\frac{-n\pi}{3}\right)\right)+D\\&=A\left(\sqrt[3]{4}\right)^n+\left(\sqrt[3]{4}\right)^n\cos\left(\frac{n\pi}{3}\right)(B+C)+\\&\quad\left(\sqrt[3]{4}\right)^n\sin\left(\frac{n\pi}{3}\right)(B-C)i+D\end{aligned}$$

If we take $B=C=\frac{1}{2}$, we get one solution

$$T_1(n)=A\left(\sqrt[3]{4}\right)^n+\left(\sqrt[3]{4}\right)^n\cos\left(\frac{n\pi}{3}\right)+D$$

If we take $B = -\frac{1}{2}i$ and $C = \frac{1}{2}i$, we get another solution

$$T_2(n) = A\left(\sqrt[3]{4}\right)^n + \left(\sqrt[3]{4}\right)^n \sin\left(\frac{n\pi}{3}\right) + D$$

By the superposition principle[†], we have a general solution

$$T(n) = A_1\left(\sqrt[3]{4}\right)^n + A_2\left(\sqrt[3]{4}\right)^n \cos\left(\frac{n\pi}{3}\right) + A_3\left(\sqrt[3]{4}\right)^n \sin\left(\frac{n\pi}{3}\right) + A_4$$

for some constants A_1, A_2, A_3, A_4. The asymptotics of the solution is $T(n) = \Theta\left(\left(\sqrt[3]{4}\right)^n\right)$
□

[†]The superposition principle says that if we have a linear recurrence and we know that some functions $g_i()$, $1 \leq i \leq k$, are solutions to it, then any linear combination of them is also a solution. See [Bal91], pp. 97.

Chapter 4

Proving the correctness of algorithms

4.1 Preliminaries

Every algorithm implements a total function that maps the set of the inputs to the set of the outputs. To prove the algorithm is correct is to prove that

1. the algorithm halts on every input, and
2. for every input, the corresponding output is precisely the one that is specified by the said function.

Proving facts about the "behaviour" of algorithms is not easy even for trivial algorithms. It is well known that in general such proofs cannot be automated. For instance, provably there does not exist an algorithm that, given as input a program and its input, always determines (using, of course, a finite number of steps) whether that program with that input halts. For details, see the HALTING PROBLEM and Theorem 4 on page 174.

That famous undecidability result, due to Alan Turing, sets certain limitations on the power of computation in general. Now we show that even for a specific simple program deciding whether it halts or not can be extremely difficult. Consider the following program.

```
k = 3;
for (;;) {
  for(a = 1; a <= k; a ++)
   for(b = 1; b <= k; b ++)
    for(c = 1; c <= k; c ++)
     for(n = 3; n <= k; n ++)
      if (pow(a,n) + pow(b,n) == pow(c,n))
       exit(); }
```

Clearly, the program does not halt if and only if Fermat's Last Theorem[†] is true. However, for several hundred years some of the best mathematicians in the world were unable to prove that theorem. The theorem was indeed proved after a huge effort by Sir Andrew Wiles, an effort that spanned many years and led to some initial frustrations as the first

[†]It says: "$a^n + b^n = c^n$ has no positive integer solutions for $n \geq 3$".

proof turned out to be incorrect. For a detailed account of these events, see [CSS97] or the review of that book by Buzzard [Buz99].

We emphasise that to prove Fermat's Last Theorem and to prove that the abovementioned program halts are essentially the same thing. Furthermore, determining whether an algorithm halts or not is but only one aspect of the analysis of algorithms. If telling whether a well-defined sequence of instructions halts or not is hard, determining whether it indeed returns the desired output cannot be any easier in general. So, determining the behaviour of even simple algorithms can necessitate profound mathematical knowledge and skills.

4.2 Loop Invariants – An Introduction

It is possible to prove assertions about algorithms—correctness or time complexity—using *loop invariants*. That technique is applicable when the algorithm is iterative. It may be a simple (not nested) loop or something more complicated. The gist of the algorithm has to be a **for** or **while** loop.

Proving assertions with loop invariants is essentially proving assertions by induction, with the notable exception that normally proofs by induction are done for infinite sequence, while the loop invariants concern algorithm that take only finite number of steps. We first give and example of a proof of correctness with a loop invariant of a very simple algorithm. That first example is very detailed and the invariant itself is outlined as a nested statement (sub-lemma).

MAXIMAL SEQUENTIAL($A[1, 2, \ldots, n]$: array of integers)

```
1  max ← A[1]
2  i ← 2
3  while i ≤ n do
4      if A[i] > max
5          max ← A[i]
6      i ← i + 1
7  return max
```

Lemma 3. *Algorithm* MAXIMAL SEQUENTIAL *returns the value of a maximum element of* $A[1, 2, \ldots, n]$.

Proof:
In order to prove the desired desired result we first prove a sub-lemma:

> **Sub-lemma:** *Every time the execution of* MAXIMAL SEQUENTIAL *is at line 3,* max *contains the value of a maximum element in the subarray* $A[1, \ldots, i-1]$.
>
> The proof of the sub-lemma is done by induction on the number of times the execution reaches line 3.
>
> **Basis.** The first time the execution is at line 3, i equals 2 (because of the previous assignment at line 2). So, the subarray $A[1, \ldots, i-1]$ is in fact $A[1, \ldots, 1]$. But max is $A[1]$ (because of the previous assignment at line 1), and indeed this is a maximum element in $A[1, \ldots, 1]$.

Inductive Hypothesis. Assume the claim is true at a certain moment when the execution is at line 3, such that the loop is to be executed at least once more[†]. Let us define that the current value of $\mathtt{i}$ is called $\mathtt{i}'$.

Induction Step. The following two possibilities are exhaustive.

- $A[\mathtt{i}']$ is the maximum element in $A[1,\ldots,\mathtt{i}']$. By the inductive hypothesis, *max* equals a maximum element in $A[1,\ldots,\mathtt{i}'-1]$. It must be the case that $\mathit{max} < A[\mathtt{i}']$. So, the condition at line 4 is TRUE and the assignment at line 5 takes place. Thus *max* becomes equal to the maximum element in $A[1,\ldots,\mathtt{i}']$ immediately after that assignment. Then $\mathtt{i}$ gets incremenented to $\mathtt{i}'+1$ (line 6). So, the next time the execution reaches line 3 again, *max* is indeed equal to a maximum element in $A[1,\ldots,(\mathtt{i}'+1)-1]$. We see the invariant is preserved in the current case.
- It is not the case that $A[\mathtt{i}']$ is the maximum element in $A[1,\ldots,\mathtt{i}']$. Then $A[\mathtt{i}']$ is smaller than or equal to a maximum element in $A[1,\ldots,\mathtt{i}'-1]$, which in its turn is equal to *max* by the inductive hypothesis. It follows *max* is equal to a maximum element in $A[1,\ldots,\mathtt{i}']$. The condition at line 4 is FALSE and the assignment at line 5 does not take place. Then $\mathtt{i}$ gets incremenented to $\mathtt{i}'+1$ (line 6). So, the next time the execution reaches line 3 again, *max* is indeed equal to a maximum element in $A[1,\ldots,(\mathtt{i}'+1)-1]$. We see the invariant is preserved in the current case, too.

That concludes the proof of the sublemma. Now consider the last time line 3 is executed. Then $\mathtt{i}$ equals $\mathtt{n}+1$. Substitute $\mathtt{i}$ with $\mathtt{n}+1$ in the sub-lemma and conclude that *max* contains the value of a maximum element in $A[1,\ldots,(\mathtt{n}+1)-1]$, *i.e.* *max* equals a maximum element in $A[1,\ldots,\mathtt{n}]$. We observe that at line 7 the algorithm returns *max*, and that concludes the proof of the lemma. □

In the subsequent proofs we will not maintain as separate claims the invariant and the conclusion "what does the invariant imply when the loop condition is evaluated for the last time". The latter will be the last step of the proof, called *termination*. The proofs will have the following structure (after [CLR00]). The claim that is to be proved is a one-place predicate. Let us call that predicate, $\mathtt{P}()$. Typically, that predicate's variable is the loop control variable. If the loop control variable is $\mathtt{i}$, the predicate is $\mathtt{P}(\mathtt{i})$.

1. **Initialisation.** It correspond to the basis in the proofs by induction. In this step we verify that the claim holds at the moment immediately prior to the first iteration of the loop. That is, consider the moment when the execution reaches the line with the **for** or **while** for the first time. If the valus of the loop control variable is $\mathtt{i}_0$ at that moment, show $\mathtt{P}(\mathtt{i}_0)$.
2. **Maintenance.** On the assumption that the claim holds before some iteration of the loop, prove the claim holds at the end of that iteration. More precisely, if $\mathtt{k}$ is

[†]It would be an error to omit the proviso "the loop is to be executed at least once more". We have to prove the invariant is preserved during any execution of the loop. Therefore, if we consider the very last execution of line 3, we cannot establish the preservation of the invariant in the body of the loop.

the value of the loop control variable at certain moment when the **for** or **while** line is reached, prove the implication $P(k) \Rightarrow P(k+1)$[†].

If there are multiple ways to go through the body of the loop (for instance, in case the body of the loop has **if**–**else** structure), the proof has to follow all of them.

3. **Termination.** The invariant is already proved. Consider the very last time the execution reaches the top of the loop, *i.e.* the line with **for** or **while** condition. Let the value of the loop control variable at that moment be, say, t. The statement $P(t)$ must prove *directly* the property of the algorithm that we are interested in.

4.3 Proving the Correctness of Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, and Quick Sort

Consider the following proof of correctness by loop invariant of INSERTION SORT, based on [CLR00]. The goal is to prove that INSERTION SORT indeed sorts its input.

```
INSERTION SORT(A[1,2,...n]: array of integers)
1  for i ← 2 to n
2      key ← A[i]
3      j ← i − 1
4      while j > 0 and A[j] > key do
5          A[j + 1] ← A[j]
6          j ← j − 1
7      A[j + 1] ← key
```

There are two loops in that algorithm. A very precise formal proof should consist of two separate invariants:

- an invariant concerning the inner **while** loop—it should be stated and proved relative to j, the control variable of the inner loop;
- an invariant concerning the outer **for** loop—it should be stated and proved relative to i.

The algorithm moves the elements of $A[\,]$ around. Because of that, making a precise argument can be tricky: for instance, when we mention $A[i]$ in Lemma 4, do we mean the element of A at position i before, or after, the inner **while** loop has shifted a certain subarray "upwards" by one position? Position i can be affected by the shift, so in general those are different elements. We overcome that potential ambiguity by giving different names to the whole array, or parts of it, in different moments of the execution.

Lemma 4. *Consider algorithm* INSERTION SORT. *Relative to any execution of the* ***for*** *loop (lines 1–7), let us call the subarray* $A[1,\ldots,i]$ *before the* ***while*** *loop (lines 4–6) starts being executed,* $A'[1,\ldots,i]$*, and let us call it* $A''[1,\ldots,i]$*, after it is executed. Assume the subarray* $A'[1,\ldots,i-1]$ *is sorted. The* ***while*** *loop has the following effects:*

[†]Assuming that the loop control variable is always incremented by one.

- j *is assigned the biggest number from* $\{1, 2, \ldots, i-1\}$ *such that* $A'[j] \leq$ ***key****, if such a number exists, or is assigned* 0*, otherwise.*
- *with respect to that value of* j*, it shifts the subarray* $A'[j+1, \ldots, i-1]$ *by one position upwards.*

Proof: The following is a loop invariant for the **while** loop:

Every time the execution reaches line 4:

- for every element x of the current subarray $A[j+2, \ldots, i]$, $x >$ ***key***, and
- the current subarray $A[j+2, \ldots, i]$ is the same as $A'[j+1, \ldots, i-1]$.

Basis. The first time the execution reaches line 4, it is the case that $j = i-1$. So, the current subarray $A[j+2, \ldots, i]$ is in fact $A[i+1, \ldots, i]$. Since that is an empty subarray, the first part of the invariant is vacuously true. The second part of the invariant is true as well because both subarrays it concerns are empty.

Maintenance. Assume the claim holds at a certain moment t when the execution is at line 4 and the **while** loop is to be executed at least once more. The latter means that $j > 0$ and $A[j] >$ ***key***. After line 5, it is the case that $A[j+1] >$ ***key***, and that is relative to the value of j that the current iteration began with. By the first part of the invariant, for every element x of the current subarray $A[j+2, \ldots, i]$, $x >$ ***key***. We conclude that for every element x of the current subarray $A[j+1, \ldots, i]$, $x >$ ***key***. At line 6, j is decremented. Relative to the new value of j, the previously stated conclusion becomes, for every element x of the current subarray $A[j+2, \ldots, i]$, $x >$ ***key***. We proved the first part of the invariant.

Consider the execution at moment t again. Consider the second part of the invariant. According to it, the current subarray $A[j+2, \ldots, i]$ is the same as $A'[j+1, \ldots, i-1]$. Then the execution of the loop body commences. At line 5, the value of the current $A[j]$ is assigned to $A[j+1]$. But element $A[j]$ has not been modified by the **while** loop so far[†]—a fact which is fairly obvious and does not necessitate inclusion in the invariant—so $A[j]$ is in fact $A'[j]$. It follows the current subarray $A[j+1, \ldots, i]$ is the same as $A'[j, \ldots, i-1]$ after the assignment at line 5 takes place. Then j gets decremented (line 6). When the execution is at line 4 again, relative to the new value of j, it is the case that $A[j+2, \ldots, i]$ is the same as $A'[j+1, \ldots, i-1]$. We proved the second part of the invariant.

Termination. Consider the moment when the execution is at line 4 and the condition there is **False**. That is, $j \leq 0$ or $A[j] \leq$ ***key***.

Case i. First assume that $j \leq 0$. Since j is decremented by one, it cannot be negative, so it is the case that $j = 0$. Plug the value 0 for j in the invariant to obtain that:

- for every element x of the current subarray $A[2, \ldots, i]$, $x >$ ***key***, and
- the current subarray $A[2, \ldots, i]$ is the same as $A'[1, \ldots, i-1]$.

The first part of the invariant means there is no number j from $\{1, 2, \ldots, i-1\}$ such that $A'[j] \leq$ ***key***. But j is assigned 0 and so the first claim of this Lemma is true. The

[†]Remember that here we consider only the executions of the **while** loop relative to the current execution of the outer **for** loop, not all executions of the **while** loop since the start of the algorithm.

second part of the invariant means that, relative to the value 0 for j, the original subarray $A[j+1,\ldots,i-1]$ has been shifted one position upwards. So, the Lemma holds when $j = 0$.

Case ii. Now assume that $j > 0$ and $A[j] \leq \texttt{key}$. But $A[j]$ has never been modified by the **while** loop. Therefore, it is the case that $A'[j] \leq \texttt{key}$.

The invariant says that, on the one hand, $A[j+2] > \texttt{key}$, $A[j+3] > \texttt{key}$, *etc.*, $A[i] > \texttt{key}$, and on the other hand, that $A[j+2] = A'[j+1]$, $A[j+3] = A'[j+2]$, $A[i] = A'[i-1]$. It follows that $A'[j+1] > \texttt{key}$, $A'[j+2] > \texttt{key}$, *etc.*, $A'[i-1] > \texttt{key}$.

The two facts above imply that indeed j is assigned the biggest number from $\{1, 2, \ldots, i-1\}$ such that $A'[j] \leq \texttt{key}$. So, the first claim of the Lemma holds. The second claim of the Lemma is literally the same as the the second part of the invariant. That concludes the proof of the Lemma. □

Lemma 5. *Algorithm* Insertion Sort *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1,\ldots,n]$, and let us call $A''[1,\ldots,n]$ the array after the algorithm halts. The following is a loop invariant for the **for** loop:

> Every time the execution of Insertion Sort is at line 1, the current subarray $A[1,\ldots,i-1]$ consists of the same elements as $A'[1,\ldots,i-1]$, but in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i = 2$. The subarray $A[1,\ldots,1]$ consists of a single element that is clearly the same as $A'[1]$ and it is, in a trivial sense, sorted.

Maintenance. Assume the claim holds at a certain execution of line 1 and the **for** loop is to be executed at least once more. Let $\widetilde{A}[1,\ldots,i]$ be the name of $A[\,]$ when the execution of the **for** loop commences. The current value of $A[i]$, *i.e.* $\widetilde{A}[i]$, is stored in $\texttt{key}$, j is set to $i-1$, and the inner **while** loop is executed. By Lemma 4, the effects of the **while** loop are the following:

- j is assigned the biggest number from $\{1, 2, \ldots, i-1\}$ such that $\widetilde{A}[j] \leq \texttt{key}$, if such a number exists, or is assigned 0, otherwise.
- with respect to that value of j, the subarray $\widetilde{A}[j+1,\ldots,i-1]$ is shifted by one position upwards.

If there are elements in $\widetilde{A}[1,\ldots,i-1]$ that are bigger than $\texttt{key} = \widetilde{A}[i]$, they are stored in a contiguous sorted subsequence – that follows from the assumption at the beginning of the **Maintenance** phase. Lemma 4 implies that the index of the smallest of those is $j+1$. Lemma 4 further implies that $\widetilde{A}[j+1,\ldots,i-1]$ is shifted into the current $A[j+2,\ldots,i]$. Thus in the current $A[\,]$, $A[j+1] = A[j+2]$ and therefore the assignment at line 7 overwrites a value that has already been copied into another position. Clearly, at the end of the **for** loop, $A[1,\ldots,i]$ consists of the same elements as $\widetilde{A}[1,\ldots,i]$ but in sorted order.

It remains to consider the case when no elements in $\widetilde{A}[1,\ldots,i-1]$ are bigger than $\texttt{key} = \widetilde{A}[i]$. By Lemma 4, j equals $i-1$ at the end of the **while** loop and nothing has been shifted upwards, thus the assignment at line 7 overwrites the i-th element of A with the value it had at the start of the **for** loop, namely $\widetilde{A}[i]$. Clearly, at the end of the **for** loop, $A[1,\ldots,i]$ consists of the same elements as $\widetilde{A}[1,\ldots,i]$ but in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, i equals $n+1$. Plug the value $n+1$ in place of i in the invariant to obtain "the current subarray $A[1,\ldots,(n+1)-1]$ consists of the same elements as $A'[1,\ldots,(n+1)-1]$, but in sorted order". □

```
SELECTION SORT(A[1,2,...n]: array of integers)
1  for i ← 1 to n − 1
2      for j ← i + 1 to n
3          if A[j] < A[i]
4              swap(A[i], A[j])
```

The following two lemmas concern the correctness of SELECTION SORT. It is obvious that SELECTION SORT permutes the elements of the input because the only changes is does to A[] happen at line 4, using swaps. That is unlike INSERTION SORT where it is not (so) obviuos that at the end, the elements in the array are the same as the original ones. So in the analysis of INSERTION SORT we did concern ourselves with proving that no original element gets overwritten before its value is stored safely somewhere else. In the analysis of SELECTION SORT we have no such concerns.

Lemma 6. *With respect to a particular execution of the outer* ***for*** *loop (lines 1–4) of* SELECTION SORT*, the execution of the inner* ***for*** *loop (lines 2–4) has the effect that* $A[i]$ *is a smallest element in* $A[i,\ldots,n]$.

Proof:
With respect to a certain execution of the outer **for** loop, the following is a loop invariant for the inner **for** loop:

> Every time the execution reaches line 2, the current $A[i]$ holds the value of a minimum element from $A[i,\ldots,j-1]$.

Basis. The first time the execution of the inner **for** loop is at line 2, it is the case that $j = i+1$. Then $A[i]$ is trivially a minimum element in $A[i,\ldots,(i+1)-1]$.

Maintenance. Assume the claim is true at some moment when the execution is at line 2 and the inner **for** loop is to be executed once more. The following two cases are exhaustive.

Case i. $A[j] < A[i]$. The condition at line 3 is TRUE and the swap at line 4 takes place. By the maintenance hypothesis, $A[i]$ is a minimum in $A[i,\ldots,j-1]$. Since $A[j] < A[i]$, by the transitivity of the $<$ relation, $A[j]$ is the minimum element in $A[i,\ldots,j]$ before the swap. We conclude $A[i]$ is the minimum element in $A[i,\ldots,j]$ after that swap. Then j gets incremented by one and the execution goes to line 2. With respect to the new value of j, it is the case that $A[i]$ is the minimum element in $A[i,\ldots,j-1]$.

Case ii. $A[j] \not< A[i]$. Then the condition at line 3 is FALSE and the swap at line 4 does not take place. By the maintenance hypothesis, $A[i]$ is a minimum element in $A[i,\ldots,j-1]$. Since $A[i] \leq A[j]$, clearly $A[i]$ is a minimum element in $A[i,\ldots,j]$. Then j gets incremented by one and the execution goes to line 2. With respect to the new value of j, it is the case that $A[i]$ is a minimum element in $A[i,\ldots,j-1]$.

Termination. Consider the moment when the execution is at line 2 for the last time. Clearly, j equals $n+1$. Plug the value $n+1$ in place of j in the invariant to obtain "the current $A[i]$ holds the value of a minimum element from $A[i,\ldots,(n+1)-1]$". □

Lemma 7. *Algorithm* SELECTION SORT *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1,\ldots,n]$. The following is a loop invariant for the outer **for** loop (lines 1–4):

> Every time the execution of SELECTION SORT is at line 1, the current subarray $A[1,\ldots,i-1]$ consists of $i-1$ in number smallest elements from $A'[1,\ldots,n]$, in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i=1$. The current subarray $A[1,\ldots,i-1]$ is empty and thus, vacuously, it consists of the zero in number smallest elements from $A'[1,\ldots,n]$, in sorted order.

Maintenance. Assume the claim holds at a certain execution of line 1 and the outer **for** loop is to be executed at least once more. Let us call the array $A[]$ at that moment, $A''[]$. By Lemma 6, the effect of the inner **for** loop is that it stores into the i^{th} position a smallest value from $A''[i,\ldots,n]$. On the other hand, by the maintenance hypothesis, $A''[1,\ldots,i-1]$ consists of $i-1$ in number, smallest elements from $A'[1,\ldots,n]$, in sorted order. We conclude that at the end of that execution of the outer **for** loop, the current $A[1,\ldots,i]$ consists of i in number, smallest elements from $A'[1,\ldots,n]$, in sorted order. Then i gets incremented by one and the execution goes to line 1. With respect to the new value of i, it is the case that the current $A[1,\ldots,i-1]$ consists of $i-1$ in number, smallest elements from $A'[1,\ldots,n]$, in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, i equals n. Plug the value n in place of i in the invariant to obtain "the current subarray $A[1,\ldots,n-1]$ consists of the smallest, $n-1$ in number, elements from $A'[1,\ldots,n]$, in sorted order". But then $A[n]$ has to be a maximum element from $A'[1,\ldots,n]$. And that concludes the proof of the correctness of SELECTION SORT. □

```
BUBBLE SORT(A[1,2,...n]: array of integers)
1  for i ← 1 to n
2      for j ← n downto i + 1
3          if A[j − 1] > A[j]
4              swap(A[j − 1], A[j])
```

Lemma 8. *With respect to a particular execution of the outer* ***for*** *loop (lines 1–4) of* BUBBLE SORT*, the execution of the inner* ***for*** *loop (lines 2–4) has the effect that* $A[i]$ *is a smallest element in* $A[i,\ldots,n]$.

Proof:
With respect to a certain execution of the outer **for** loop, the following is a loop invariant for the inner **for** loop:

> Every time the execution reaches line 2, the current $A[j]$ holds the value of a minimum element from $A[j,\ldots,n]$.

Basis. The first time the execution of the inner **for** loop is at line 2, it is the case that $j=n$. Then $A[n]$ is trivially a minimum element in $A[n,\ldots,n]$.

Maintenance. Assume the claim is true at some moment when the execution is at line 2 and the inner **for** loop is to be executed once more. The following two cases are exhaustive.

Case i. $A[j-1] > A[j]$. The condition at line 3 is TRUE and the swap at line 4 takes place. By the maintenance hypothesis, $A[j]$ is a minimum in $A[j, \ldots, n]$ at the before the swap. Since $A[j-1] > A[j]$, $A[j]$ is a minimum element in $A[j-1, \ldots, n]$ before the swap. After the swap, clearly $A[j-1]$ is a minimum element in $A[j-1, \ldots, n]$. Since j is decremented by one the next time the execution is at line 2, with respect to the new j, it is the case that $A[j]$ is a minimum element from $A[j, \ldots, n]$.

Case ii. $A[j-1] \not> A[j]$, *i.e.* $A[j-1] \leq A[j]$. The condition at line 3 is FALSE and the swap at line 4 does not take place. By the maintenance hypothesis, $A[j]$ is a minimum in $A[j, \ldots, n]$ at the before the evaluation at line 3. By the transitivity of the $\leq$ relation, $A[j-1]$ is a minimum element in $A[j-1, \ldots, n]$. Since j is decremented by one the next time the execution is at line 2, with respect to the new j, it is the case that $A[j]$ is a minimum element from $A[j, \ldots, n]$.

Termination. Consider the moment when the execution is at line 2 for the last time. Then j equals i. Plug the value i in place of j in the invariant to obtain "$A[i]$ holds the value of a minimum element from $A[i, \ldots, n]$". □

Lemma 9. *Algorithm* BUBBLE SORT *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1, \ldots, n]$. The following is a loop invariant for the outer **for** loop (lines 1–4):

> Every time the execution of BUBBLE SORT is at line 1, the current subarray $A[1, \ldots, i-1]$ consists of $i-1$ in number smallest elements from $A'[1, \ldots, n]$, in sorted order.

Basis. The first time the execution reaches line 1, it is the case that $i = 1$. Obviously, the current $A[1, \ldots, i-1]$ is empty and the claim is vacuously true.

Maintenance. Assume the claim holds at a certain execution of line 1 and the outer **for** loop is to be executed at least once more. Let us call the array $A[\,]$ at that moment, $A''[\,]$. By Lemma 8, the effect of the inner **for** loop is that it stores into the i^{th} position a smallest value from $A''[i, \ldots, n]$. On the other hand, by the maintenance hypothesis, $A''[1, \ldots, i-1]$ consists of $i-1$ in number, smallest elements from $A'[1, \ldots, n]$, in sorted order. We conclude that at the end of that execution of the outer **for** loop, the current $A[1, \ldots, i]$ consists of i in number, smallest elements from $A'[1, \ldots, n]$, in sorted order. Then i gets incremented by one and the execution goes to line 1. With respect to the new value of i, it is the case that the current $A[1, \ldots, i-1]$ consists of $i-1$ in number, smallest $i-1$ elements from $A'[1, \ldots, n]$, in sorted order.

Termination. Consider the moment when the execution is at line 1 for the last time. Clearly, j equals $n+1$. Plug the value n in place of i in the invariant to obtain "the current subarray $A[1, \ldots, (n+1)-1]$ consists of the smallest, $(n+1)-1$ in number, elements from $A'[1, \ldots, n]$, in sorted order.". □

```
MERGE(A[1,2,...n]: array of integers; l, mid, h: indices in A[])
  1  (* the subarrays A[l,...,mid] and A[mid+1,...,h] are sorted *)
```

```
2   n1 ← mid − l + 1
3   n2 ← h − mid
4   create L[1, ..., n1 + 1] and R[1, ..., n2 + 1]
5   L ← A[l, ..., mid]
6   R ← A[mid + 1, ..., h]
7   L[n1 + 1] ← ∞
8   R[n2 + 1] ← ∞
9   i ← 1
10  j ← 1
11  for k ← l to h
12      if L[i] ≤ R[j]
13          A[k] ← L[i]
14          i ← i + 1
15      else
16          A[k] ← R[j]
17          j ← j + 1
```

Lemma 10. *On the assumption that the subarrays* $A[l, \ldots, mid]$ *and* $A[mid+1, \ldots, h]$ *are sorted, the whole array* $A[l, \ldots, h]$ *is sorted after* MERGE *terminates.*

Proof:
The following is a loop invariant for the **for** loop (lines 11–17):

> **part i** Every time the execution of MERGE is at line 11, $A[l, \ldots, k-1]$ contains $k - l$ smallest elements of L[] and R[], in sorted order.
>
> **part ii** Furthermore, L[i] and R[j] are smallest elements in L[] and R[], respectively, that have not been copied into A[] yet.

Basis. When the execution is at line 11 for the first time, it is the case that $k = l$. Then the subarray $A[l, \ldots, k-1]$ is in fact $A[l, \ldots, l-1]$, *i.e.* an empty subarray. Vacuously, it is sorted and contains the $k - l = 0$ smallest elements of L[] and R[], in sorted order. Furthermore, L[1] and R[1] are smallest elements of L[] and R[], respectively, that have not been copied into A[].

Maintenance. Assume the claim holds at a certain execution of line 11 and the **for** loop is to be executed at least once more. There are two alternative ways the execution can take through the body of the loop. We consider them both. Before we do that we prove an useful auxilliary result; both L[] and R[] contain a ∞ value so we want to be sure that those are never compared.

> *In the comparison at line 12, it cannot be the case that two* ∞ *values are compared.*
>
> **Proof:** Assume the opposite. By the maintenance hypothesis, $k - l$ elements are copied into A from L[] and R[]. By the maintenance hypothesis, the body of the loop is to be executed once more, so the number of the copied elements is $\leq h - l$. By the assumption we made, we have copied all $n_1 + n_2$ elements that are not ∞, so the number of copied elements is $\geq n_1 + n_2 = mid - l + 1 + h - mid = h - l + 1 > h - l$. $\lightning$

Consider the comparison at line 12. Since both L[i] and R[j] cannot be ∞, the result of the comparison is always defined. First assume L[i] $\leq$ R[j]. Clearly, L[i] $< \infty$. By **part ii** of the maintenance hypothesis and the assumption that L[i] $\leq$ R[j] we conclude L[i] is a smallest element in L[] and R[] that has not been copied yet. By **part i** of the maintenance hypothesis, L[i] is not smaller than any element in $A[l, \ldots, k-1]$. The execution goes to line 13. So, now $A[k]$ is not smaller that any element in $A[l, \ldots, k-1]$. It follows $A[l, \ldots, k]$ is sorted and contains the $k - l + 1 = (k + 1) - l$ smallest elements of L[] and R[]. But k gets incremented the next time the execution is at line 11. Relative to the new value of k, it is the case that $A[l, \ldots, k-1]$ contains $k - l$ smallest elements of L[] and R[], in sorted order. So, **part i** of the invariant holds.

Now we prove that **part ii** of the invariant holds as well. By assumption, L[] and R[] are sorted. Before the assignment at line 13, L[i] was a smallest element from L[] not copied into A[] yet. After that assignment, L[i + 1] is a smallest element from L[] not copied into A[] yet. But i get incremented at line 14. With respect to the new value of i, L[i] is a smallest element from L[] not copied into A[] yet.

Now assume the execution is still at line 12 and L[i] $\not\leq$ R[j], *i.e.* L[i] $>$ R[j]. The proof is completely analogous to the proof above.

Termination. The loop control variable k equals $h + 1$ the last time the execution is at line 11. Plug that value into the invariant to obtain "the subarray $A[l, \ldots, h - 1]$ contains $k - l$ smallest elements of L[] and R[], in sorted order". □

MERGE SORT(A[1, 2, ..., n]: array of integers; l, h: indices in A[])
1 **if** $l < h$
2 $\quad mid \leftarrow \lfloor \frac{l+h}{2} \rfloor$
3 $\quad$ MERGE SORT(A, l, mid)
4 $\quad$ MERGE SORT(A, $mid + 1$, h)
5 $\quad$ MERGE(A, l, mid, h)

Lemma 11. *Algorithm* MERGE SORT *is a correct sorting algorithm if the initial call is* MERGE SORT(A, 1, n).

Proof:
By induction on the difference $h - l$[†]. We consider it obvious that $h - l$ can get as small as zero but not any smaller. So, the basis is $h - l = 0$.

Basis. $h = l$. On the one hand, the array $A[l]$ is trivially sorted. On the other hand, MERGE SORT does nothing when $h = l$. So, the one element array remains sorted at the end.

Maintenance. Assume that MERGE SORT sorts correctly the subarrays $A[l, \ldots, mid]$ and $A[mid + 1, \ldots, h]$ (lines 3 and 4) during any recursive call such that $h > l$. Use Lemma 10 to conclude that at the end of the current call, the whole $A[l, \ldots, h]$ is sorted.

Termination. When proving facts about recursive algorithms, using induction rather than loop invariant, the termination step of the proof concerns the termination of the very first recursive call. In this proof that is MERGE SORT(A[1, ..., n]). With MERGE SORT,

[†]Note that this is not a proof with loop invariant because MERGE SORT is not an iterative algorithm.

this step is trivial: simply observe that after the algorithm finishes altogether, $A[1,\ldots,n]$ is sorted. □

```
PARTITION(A[1,2,...,n]: array of integers; l, h: indices in A[])
1  pivot ← A[h]
2  pp ← l
3  for i ← l to h − 1
4      if A[i] < pivot
5          swap(A[i], A[pp])
6          pp ← pp + 1
7  swap(A[pp], A[h])
8  return pp
```

Lemma 12. *The value* pp *returned by* PARTITION *is such that* $l \le pp \le h$ *and*

$$\forall x \in A[l,\ldots,pp-1], \forall y \in A[pp+1,\ldots,h] : x < A[pp] \le y$$

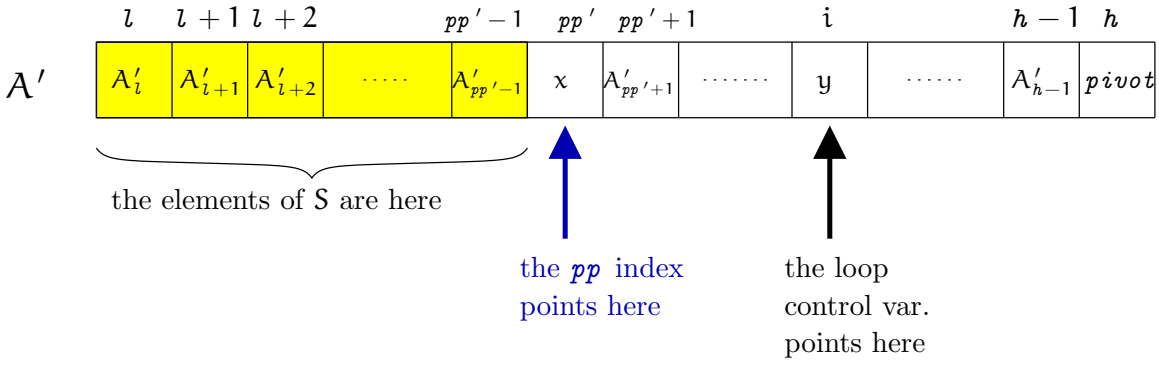
Proof:
At any moment, let S denote the current set $\{x \in A[l,\ldots,i-1] \mid x < pivot\}$. The following is a loop invariant for the **for** loop (lines 3–6):

> Every time the execution of PARTITION is at line 3, $pp \le i$ and the elements of S are precisely the elements in $A[l,\ldots,pp-1]$.

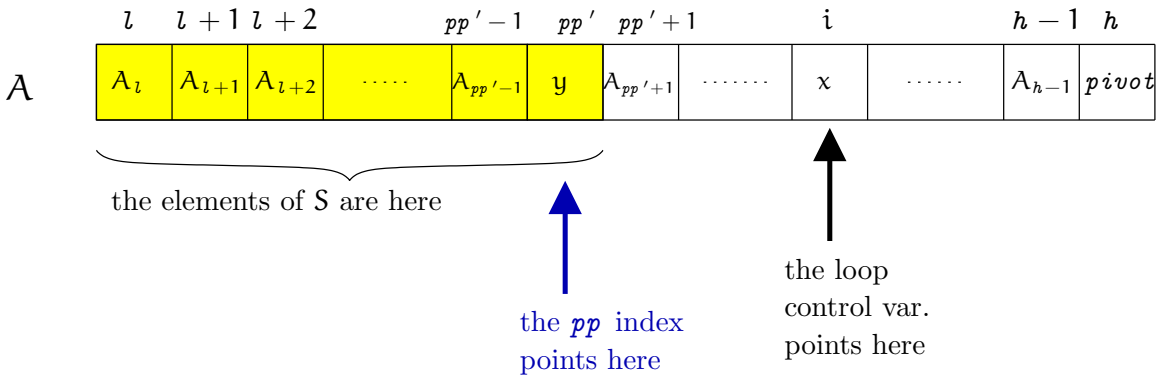
Basis: $i = l$. Because of the assignement at line 2, $pp \le i$ holds. The subarray $A[l,\ldots,i-1]$ is $A[l,\ldots,l-1]$, an empty subarray, therefore $S = \emptyset$, so it is vacuously true that the elements of S are precisely the elements in $A[l,\ldots,pp-1]$.

Maintenance: Assume the claim holds at a certain moment when the execution is at line 3 and $i \le h-1$. Let $A'[]$ and pp' denote the array A[] and the index variable pp, respectively, at the beginning of that the **for** loop execution.

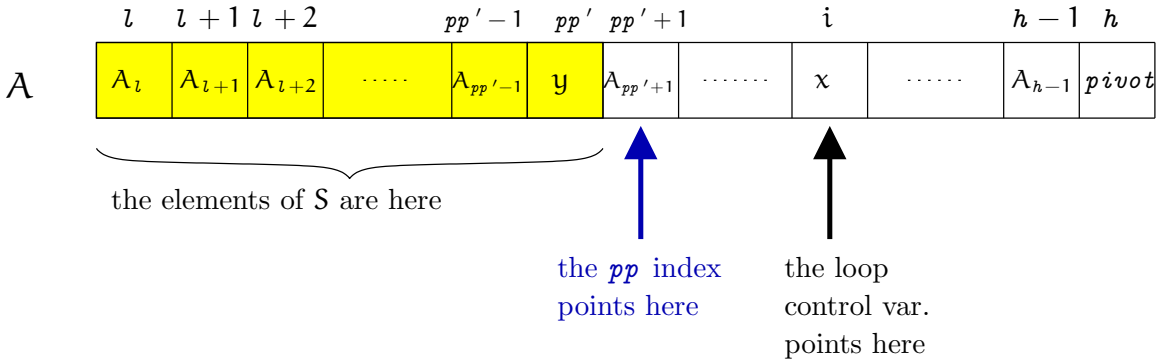
Case 1: $A'[i] < pivot$. By the maintenance hypothesis, $A'[pp]$ is the leftmost element that is left of the current i^{th} position and is not smaller than $pivot$. All the elements that are smaller than $pivot$ and left of the current i^{th} position—namely, the elements of S—consitute the subarray left of the current pp^{th} position. Let the values of $A'[pp]$ and $A'[i]$ be called x and y, respectively. See the following figure. The elements of S are outlined in yellow. For brevity we write A_l rather than $A[l]$, *etc.*



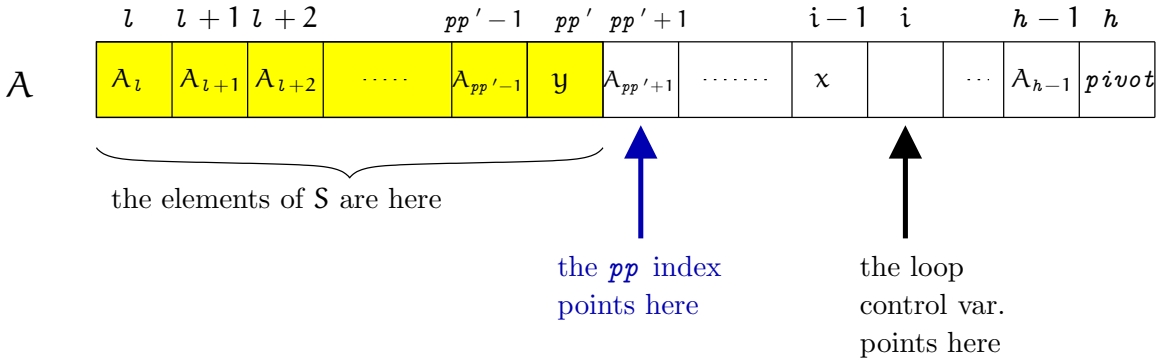
The condition at line 4 is TRUE and so the execution proceeds to line 5 where x and y get swapped. Note that S "grows" by one element, namely y:



At line 6, pp is incremented by one:

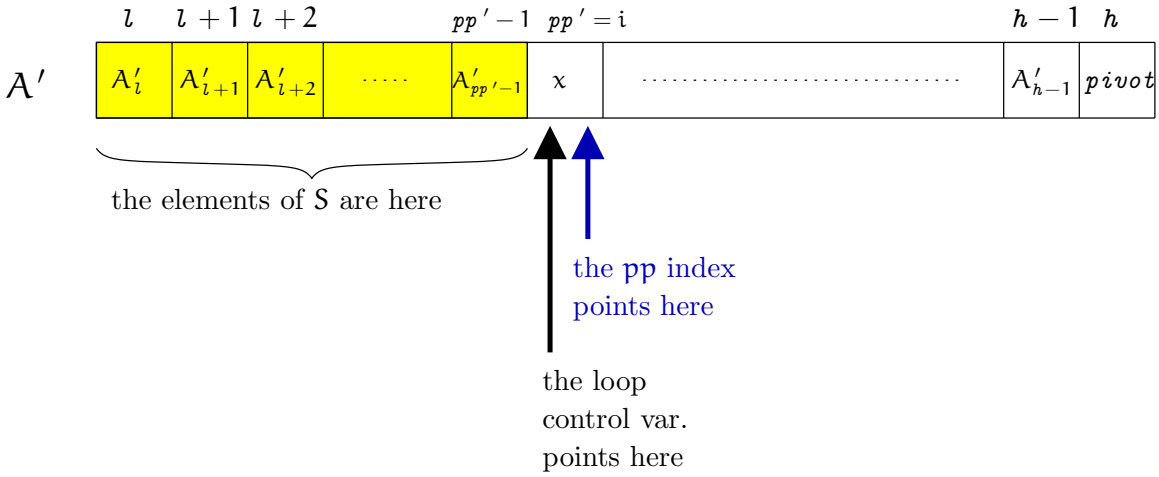


Next the execution is at line 3 again, with i incremented by one. Relative to the current i, we have the following picture:

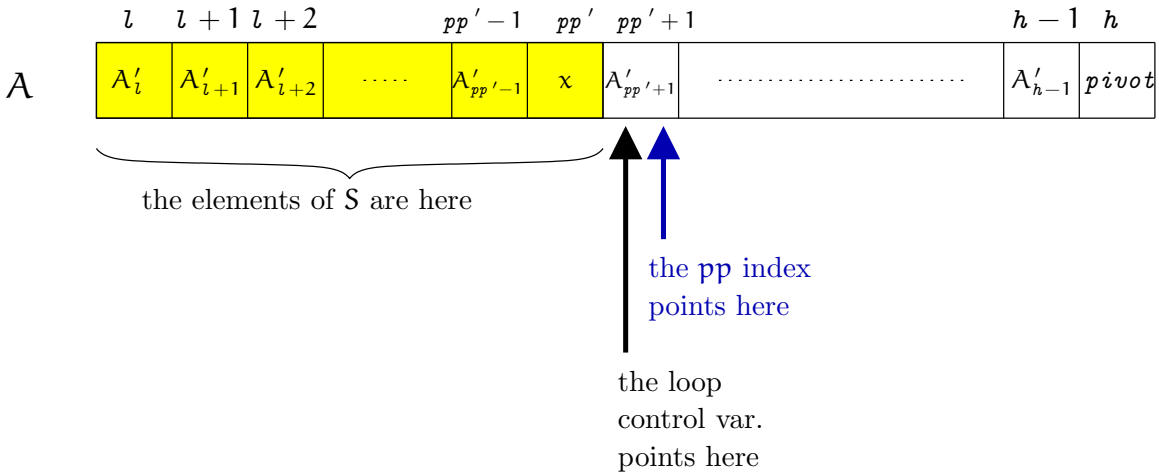


Relative to the current values of i and pp, the elements in $A[l, \ldots, i-1]$ smaller than $pivot$ are precisely the elements in $A[l, \ldots, pp-1]$. Furthermore, $pp \leq i$. And so the invariant holds.

There is no loss of generality in considering the case $pp' < i$ as we did here. However, if the reader is not convinced, here is what happens when $pp' = i$ at the start of the iteration whose execution we follows. Let x be the value of the element at position $pp' = i$:



The condition at line 4 is true under the premise of **Case 1**, so the swap at line 5 takes place. However, nothing changes because x is swapped with itself. Then both pp and i get incremented by one. Relative to their new values, S is one element bigger than it was at the beginning of the current iteration – it has "grown", x being added to it. However, it is still the case that $pp \leq i$ and the elements of S are precisely the elements in $A[l, \ldots, pp - 1]$:



Case 2: $A'[i] \geq pivot$. In this case, the condition at line 4 is FALSE and so the execution proceeds directly to line 3 with the loop control variable being incremented by one. The invariant holds because of the maintenance hypothesis and the facts that no element in the array is moved, pp equals pp', and $pp \leq i$ relative to the new value of i, and S is the same relative to the new value of i.

Termination: When the **for** loop terminates, it is the case that $i = h$. Now S is the set of the elements in $A[l, \ldots, h - 1]$ smaller than $pivot$. But it is also true that S consists

of the elements in $A[l, \ldots, h]$ smaller than $pivot$. By the loop invariant, the elements of S form the contiguous subarray $A[l, \ldots, pp-1]$. After the assignment at line 7, it is the case that

$$\forall x \in A[l, \ldots, pp-1], \forall y \in A[pp+1, \ldots, h] : x < A[pp] \leq y$$

And finally at line 8, PARTITION returns pp, so the claim of this Lemma is true. □

QUICK SORT($[A1, 2, \ldots, n]$: array of integers; l, h: indices in A[])
1 **if** $l < h$
2 $\quad mid \leftarrow$ PARTITION(A, l, h)
3 $\quad$ QUICK SORT($A, l, mid-1$)
4 $\quad$ QUICK SORT($A, mid+1, h$)

Lemma 13. *Algorithm* QUICK SORT *is a correct sorting algorithm if the initial call is* QUICK SORT($A, 1, n$).

Proof:
By induction on the difference $h - l$. We consider it obvious that $h - l$ can get as small as zero but not any smaller. So, the basis is $h - l = 0$.

Basis. $h = l$. On the one hand, the array $A[l]$ is trivially sorted. On the other hand, QUICK SORT does nothing when $h = l$. So, the one element array remains sorted at the end.

Maintenance. Assume that $h > l$. Use Lemma 12 to conclude that after the call to PARTITION (line 2) returns mid, it is the case that A[] is modified in such a way that left of $A[mid]$ are precisely the elements of the original A[] smaller than $A[mid]$, and right of $A[mid]$ are the elements not smaller than it. Assuming that the two recursive calls at lines 3 and 4 sort correctly the respective subarrays, clearly $A[l, \ldots, h]$ is sorted at the end of the current recursive call.

Termination. Consider the termination of the first recursive call QUICK SORT($A[1, \ldots, n]$) and conclude that $A[1, \ldots, n]$ is sorted. □

4.4 The Correctness of Algorithms on Binary Heaps

First let us clarify that by heaps we mean binary max heaps. The algorithms concerning heaps use the following primitive operations:

PARENT(i)
return $\lfloor \frac{i}{2} \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

In the following definitions and discussion we assume the array $A[]$ is used to represent a complete binary tree. *A complete binary tree* is a binary tree such that every level, except possibly for the last level, is completely filled. If the last level is incomplete then its nodes must be as left as possible. *A perfect binary tree* is a complete binary tree in which the last level is complete. For the classification of trees as data structures, see [NIS]. When we say tree, we always mean binary tree.

Definition 4. *Let* $A[1, 2, \ldots, n]$ *be an array of integers*[†] *and* i *be an index such that* $1 \leq i \leq n$. *We call* "the complete subtree in $A[]$ rooted at i" *—denoted by* $A[\![i]\!]$ *—the not necessarily contiguous subarray of* $A[]$ *induced by the indices generated by the following function:*

```
GENERATE INDICES(A[1,...,n]: array of integers, i: index in A[])
1  print i
2  left ← LEFT(i)
3  right ← RIGHT(i)
4  if left ≤ n
5      GENERATE INDICES(A[], left )
6  if right ≤ n
7      GENERATE INDICES(A[], right)                                  □
```

Clearly, $A[\![1]\!]$ is the array $A[1, \ldots, n]$ itself. Since $A[]$ represents a tree, $A[1]$ is the root and $A[\![i]\!]$ is the subtree rooted at vertex $A[i]$. When we use Graph Theory terminology and more specifically, terminology that concerns rooted trees, on arrays, we of course have in mind that the array represents a rooted tree. So the leaves of $A[]$ are the elements that correspond to the leaves of the tree that $A[]$ represents. The leaves of $A[1, \ldots, n]$ are precisely $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 1\right]$, $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 2\right]$, $\ldots$, $A[n]$ (see Problem 131 on page 170).

Definition 5. *Let* T *be any rooted tree and* u *be any vertex in it.* The height *of* u *is the distance between it and the root. The height of* T *is the maximum height of any vertex in* T. The depth[‡] *of* u *is the length of any longest path between* u *and a leaf which path does not contain vertices of height smaller than the height of* u. □

When we speak of the depth of an element of $A[]$ we have in mind the depth of the corresponding vertex of the tree $A[]$ represents. Of course, the elements of depth 0 are the leaves. As an example consider the following 26-element array $A[]$:

A	6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

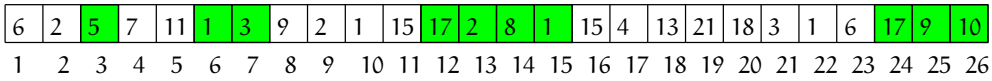
The leaves of $A[]$ are elements $A[13]$–$A[26]$, shown in yellow:

A	6	2	5	7	11	1	3	9	2	1	15	17	2	8	1	15	4	13	21	18	3	1	6	17	9	10
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

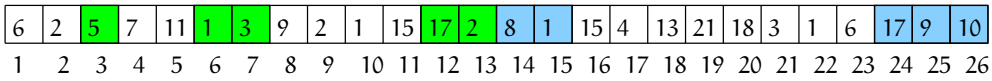
Suppose $i = 3$. $A[\![3]\!]$ is the (non-contiguous) subarray of $A[]$ outlined in green:

[†]The heaps we consider have elements–integers. Of course, any other data type whose elements take $O(1)$ memory and allows comparing and moving elements around in $O(1)$ time could be used.

[‡]In [CLR00], the authors call "height of a node" what we call "depth".



The leaves of $A[\![3]\!]$ are the five elements $A[14]$, $A[15]$, $A[24]$–$A[26]$, outlined in blue:



Definition 6 (heap)**.** $A[\![i]\!]$ *is a heap iff:*

- $A[\![\textsc{Left}(i)]\!]$*, if it exists, is a heap,*
- $A[\![\textsc{Right}(i)]\!]$*, if it exists, is a heap,*
- $A[i] \geq A[\textsc{Left}(i)]$*, if the latter exists, and*
- $A[i] \geq A[\textsc{Right}(i)]$*, if the latter exists.* □

We consider the following fact obvious.

Fact: If $A[\![i]\!]$ is a heap then for any index j such that $A[j]$ is in $A[\![i]\!]$, $A[\![j]\!]$ is a heap. □

In the context of heaps, relative to some $A[\,]$ and index i in it, let us call *a heap obstruction* every pair of indices $\langle j, k\rangle$ such that:

- $A[j]$ and $A[k]$ are in $A[\![i]\!]$,
- $A[j] < A[k]$, and
- $k = \textsc{Left}(j)$ or $k = \textsc{Right}(j)$.

Clearly, $A[\![i]\!]$ is a heap iff it has no heap obstructions.

```
RECURSIVE HEAPIFY(A[1,2,...,n]: array of integers, i: index in A[])
 1  left ← LEFT(i)
 2  right ← RIGHT(i)
 3  if left ≤ n  and A[left] > A[i]
 4      largest ← left
 5  else
 6      largest ← i
 7  if right ≤ n  and A[right] > A[largest]
 8      largest ← right
 9  if largest ≠ i
10      swap(A[i], A[largest])
11      RECURSIVE HEAPIFY(A[], largest)
```

Lemma 14. *Under the assumption that* $A[1,\ldots,n]$ *and* i *are such that each of* $A[\![\textsc{Left}(i)]\!]$ *and* $A[\![\textsc{Right}(i)]\!]$*, if it exists, is a heap, the effect of algorithm* RECURSIVE HEAPIFY *is that* $A[\![i]\!]$ *is heap.*

Proof:
By induction on the height of $A[\![i]\!]$. The height can be as small as zero and that will be our basis. Before we go on with our proof, a word of caution: the assumption about $A[\![\text{LEFT}(i)]\!]$ and $A[\![\text{RIGHT}(i)]\!]$ is *entirely different* from the inductive hypothesis. The hypothesis is the claim we are proving, though formulated in terms of a particular value of the parameter – it is not a premise of this Lemma. On the other hand, the said assumption is part of the premises of the Lemma.
Let h be the height of $A[\![i]\!]$.

Basis. $h = 0$. That means that $A[\![i]\!]$ consists of a single element from $A[]$. So, both LEFT(i) and RIGHT(i) point outside $A[]$. Let us follow the execution of RECURSIVE HEAPIFY: the condition at line 3 is false and so the assignment at line 6 takes place. The condition at line 7 is false, too, so line 8 is not executed and the execution goes to line 9. The evaluation there yields FALSE and so the current recursive call terminates. Clearly, $A[\![i]\!]$ is a heap when that recursive call terminates.

Inductive Hypothesis. Assume that for every $A[\![j]\!]$ of height $\leq h-1$ rooted at some j such that A[j] belongs to $A[\![i]\!]$, it is the case that RECURSIVE HEAPIFY$(A[], j)$ constructs a heap out of $A[\![j]\!]$.

Inductive Step. Consider the execution of RECURSIVE HEAPIFY$(A[], i)$. Without loss of generality, assume that LEFT(i) $\leq n$ and RIGHT(i) $\leq n$ point, so lines 3 and 7 are

if $A[left] > A[i]$

and

if $A[right] > A[largest]$

respectively.
Case i: Both $A[left]$ and $A[right]$ are bigger than A[i] at the beginning and $A[right] > A[left]$. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place. The evaluation at line 7 yields TRUE, too, so the assignment at line 8 takes place and when the execution is at line 9, $largest$ equals $right$. The evaluation at line 9 yields TRUE and at line 10, A[i] and $A[largest]$ get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $right$. By the inductive hypothesis, that recursive call constructs a heap out of $A[\![right]\!]$. On the other hand, $A[\![left]\!]$ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial $A[right]$, is

- bigger than the current $A[left]$ because of the premises
- not smaller than the current $A[right]$ for the following reasons. At the beginning, $A[\![right]\!]$ was a heap so the former $A[right]$ was not smaller than any other element of $A[\![right]\!]$ then (at the beginning). Now, at the end, the elements of $A[\![right]\!]$ consist of the initial elements minus the initial $A[right]$ plus the initial A[i]. Since the initial $A[right]$ is bigger than the initial A[i] and not smaller that any other element of $A[\![right]\!]$, the current A[i] is not smaller than the current $A[right]$.

Therefore, by definition the current $A[\![i]\!]$ is a heap.
Case ii: Both $A[left]$ and $A[right]$ are bigger than A[i] at the beginning and $A[right] \not> A[left]$. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place.

The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $largest$ equals $left$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $left$. By the inductive hypothesis, that recursive call constructs a heap out of A$[\![left]\!]$. On the other hand, A$[\![right]\!]$ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$left$], is

- not smaller than the current A[$right$] because of the premises
- not smaller than the current A[$left$] for the following reasons. At the beginning, A$[\![left]\!]$ was a heap so the former A[$left$] was not smaller than any other element of A$[\![left]\!]$ then (at the beginning). Now, at the end, the elements of A$[\![left]\!]$ consist of the initial elements minus the initial A[$left$] plus the initial A[i]. Since the initial A[$left$] is bigger than the initial A[i] and not smaller that any other element of A$[\![left]\!]$, the current A[i] is not smaller than the current A[$left$].

Therefore, by definition the current A$[\![$i$]\!]$ is a heap.
Case iii: A[$left$] $\not>$ A[i] and A[$right$] $>$ A[i]. The evaluation at line 3 yields FALSE and the assignment at line 6 takes place. The evaluation at line 7 yields TRUE, so the assignment at line 8 takes place and when the execution is at line 9, $largest$ equals $right$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $right$. By the inductive hypothesis, that recursive call constructs a heap out of A$[\![right]\!]$. On the other hand, A$[\![left]\!]$ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$right$], is

- bigger than the current A[$left$] because of the premises and the transitivity of the inequalities
- not smaller than the current A[$right$] for the following reasons. At the beginning, A$[\![right]\!]$ was a heap so the former A[$right$] was not smaller than any other element of A$[\![right]\!]$ then (at the beginning). Now, at the end, the elements of A$[\![right]\!]$ consist of the initial elements minus the initial A[$right$] plus the initial A[i]. Since the initial A[$right$] is bigger than the initial A[i] and not smaller that any other element of A$[\![right]\!]$, the current A[i] is not smaller than the current A[$right$].

Therefore, by definition the current A$[\![$i$]\!]$ is a heap.
Case iv: A[$left$] $>$ A[i] and A[$right$] $\not>$ A[i]. The evaluation at line 3 yields TRUE and the assignment at line 4 takes place. The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $largest$ equals $left$. The evaluation at line 9 yields TRUE and at line 10, A[i] and A[$largest$] get exchanged. The recursive call at line 11 takes place with the former A[i] now at position $left$. By the inductive hypothesis, that recursive call constructs a heap out of A$[\![left]\!]$. On the other hand, A$[\![right]\!]$ is a heap because it has not been modified in any way. On the third hand, the current A[i], *i.e.* the initial A[$left$], is

- bigger than the current A[$right$] because of the premises and the transitivity of the inequalities

- not smaller than the current $A[\mathit{left}]$ for the following reasons. At the beginning, $A[\![\mathit{left}]\!]$ was a heap so the former $A[\mathit{left}]$ was not smaller than any other element of $A[\![\mathit{left}]\!]$ then (at the beginning). Now, at the end, the elements of $A[\![\mathit{left}]\!]$ consist of the initial elements minus the initial $A[\mathit{left}]$ plus the initial $A[i]$. Since the initial $A[\mathit{left}]$ is bigger than the initial $A[i]$ and not smaller that any other element of $A[\![\mathit{left}]\!]$, the current $A[i]$ is not smaller than the current $A[\mathit{left}]$.

Therefore, by definition the current $A[\![i]\!]$ is a heap.
Case v: $A[\mathit{left}] > A[i]$ and $A[\mathit{right}] \not> A[i]$. The evaluation at line 3 yields FALSE and the assignment at line 6 takes place. The evaluation at line 7 yields FALSE, so the assignment at line 8 does not take place and when the execution is at line 9, $\mathit{largest}$ equals i. The evaluation at line 9 yields FALSE and the execution terminates, leaving $A[\![i]\!]$ untouched. By the premises, $A[\![\mathit{left}]\!]$ and $A[\![\mathit{right}]\!]$ are heaps and $A[\mathit{left}] > A[i]$ and $A[\mathit{right}] \not> A[i]$, so the current $A[\![i]\!]$ is a heap by definition. □

ITERATIVE HEAPIFY($A[1, 2, \ldots, n]$: array of integers, i: index in $A[\,]$)

```
1  j ← i
2  while j ≤ ⌊n/2⌋ do
3      left ← LEFT(j)
4      right ← RIGHT(j)
5      if left ≤ n and A[left] > A[j]
6          largest ← left
7      else
8          largest ← j
9      if right ≤ n and A[right] > A[largest]
10         largest ← right
11     if largest ≠ j
12         swap(A[j], A[largest])
13         j ← largest
14     else
15         break
```

Lemma 15. *Under the assumption that* $A[1, \ldots, n]$ *and* i *are such that each of* $A[\![\text{LEFT}(i)]\!]$ *and* $A[\![\text{RIGHT}(i)]\!]$*, if it exists, is a heap, the effect of algorithm* ITERATIVE HEAPIFY *is that* $A[\![i]\!]$ *is heap at its termination.*

Proof:
The following is a loop invariant for the **while** loop (lines 1–15):

> Every time the execution is at line 2, the only possible heap obstructions in $A[\![i]\!]$ are $\langle j, \text{LEFT}(j)\rangle$ and $\langle j, \text{RIGHT}(j)\rangle$, if $A[\text{LEFT}(j)]$ and $A[\text{RIGHT}(j)]$ exist.

Basis. $j = i$. By the premises, each of $A[\![\text{LEFT}(i)]\!]$ and $A[\![\text{RIGHT}(i)]\!]$, if it exists, is a heap, so the claim holds trivially.

Maintenance. Assume that the claim holds at some moment when the execution is at line 2 and execution will reach line 2 at least once more.

⁉ NB ⁉ *The latter proviso implies line 15 is not reached during the current execution.*

So, it is the case that $j \leq \left\lfloor \frac{n}{2} \right\rfloor$, and thus at least one of $\textsc{Left}(j) \leq n$ and $\textsc{Right}(j) \leq n$ is true, namely $\textsc{Left}(j) \leq n$. So, at least $A[\textsc{Left}(j)]$ is defined. Without loss of generality, assume both $\textsc{Left}(j) \leq n$ and $\textsc{Right}(j) \leq n$ to avoid considering unnecessary subcases.
Case i: Both $A[\mathit{left}]$ and $A[\mathit{right}]$ are bigger than A[j] at the beginning and $A[\mathit{right}] > A[\mathit{left}]$. The evaluation at line 5 yields True and the assignment at line 6 takes place. The evaluation at line 9 yields True, too, so the assignment at line 10 takes place and when the execution is at line 11, $\mathit{largest}$ equals right. The evaluation at line 11 yields True and at line 12, A[j] and $A[\mathit{largest}]$ get exchanged. We claim the only possible obstructions in $A[\![i]\!]$ after the exchange are $\langle \mathit{right}, \textsc{Left}(\mathit{right}) \rangle$ and $\langle \mathit{right}, \textsc{Right}(\mathit{right}) \rangle$:

- no element in $A[\![i]\!]$ outside $A[\![j]\!]$ has been changed;
- currently $A[j] > A[\mathit{left}]$ so $\langle j, \textsc{Left}(j) \rangle$ cannot be an obstruction;
- currently $A[j] > A[\mathit{right}]$ so $\langle j, \textsc{Right}(j) \rangle$ cannot be an obstruction;
- $A[\![\mathit{left}]\!]$ has not been modified.
- none of $A[\![\textsc{Left}(\mathit{right})]\!]$ and $A[\![\textsc{Right}(\mathit{right})]\!]$ has been modified.

But right is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.
Case ii: Both $A[\mathit{left}]$ and $A[\mathit{right}]$ are bigger than A[j] at the beginning and $A[\mathit{right}] \not> A[\mathit{left}]$. The evaluation at line 5 yields True and the assignment at line 6 takes place. The evaluation at line 9 yields False, so the assignment at line 10 does not take place and when the execution is at line 11, $\mathit{largest}$ equals left. The evaluation at line 11 yields True and at line 12, A[j] and $A[\mathit{largest}]$ get exchanged. We claim the only possible obstructions in $A[\![i]\!]$ after the exchange are $\langle \mathit{left}, \textsc{Left}(\mathit{left}) \rangle$ and $\langle \mathit{left}, \textsc{Right}(\mathit{left}) \rangle$:

- no element in $A[\![i]\!]$ outside $A[\![j]\!]$ has been changed;
- currently $A[j] > A[\mathit{left}]$ so $\langle j, \textsc{Left}(j) \rangle$ cannot be an obstruction;
- currently $A[j] \geq A[\mathit{right}]$ so $\langle j, \textsc{Right}(j) \rangle$ cannot be an obstruction;
- $A[\![\mathit{right}]\!]$ has not been modified.
- none of $A[\![\textsc{Left}(\mathit{left})]\!]$ and $A[\![\textsc{Right}(\mathit{left})]\!]$ has been modified.

But left is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.
Case iii: $A[\mathit{left}] \not> A[j]$ and $A[\mathit{right}] > A[j]$ at the beginning. The evaluation at line 5 yields False and the assignment at line 8 takes place. The evaluation at line 9 yields True, so the assignment at line 10 takes place and when the execution is at line 11, $\mathit{largest}$ equals right. The evaluation at line 11 yields True and at line 12, A[j] and $A[\mathit{largest}]$ get exchanged. We claim the only possible obstructions in $A[\![i]\!]$ after the exchange are $\langle \mathit{right}, \textsc{Left}(\mathit{right}) \rangle$ and $\langle \mathit{right}, \textsc{Right}(\mathit{right}) \rangle$. The proof is precisely the same as the proof of the analogous claim is **Case i**. But $\mathit{largest}$ is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.
Case iv: $A[\mathit{left}] > A[j]$ and $A[\mathit{right}] \not> A[j]$ at the beginning. The evaluation at line 5 yields True and the assignment at line 6 takes place. The evaluation at line 9

yields FALSE, so the assignment at line 10 does not take place and when the execution is at line 11, *largest* equals *left*. The evaluation at line 11 yields TRUE and at line 12, A[j] and A[*largest*] get exchanged. We claim the only possible obstructions in A⟦i⟧ after the exchange are ⟨*left*, LEFT(*left*)⟩ and ⟨*left*, RIGHT(*left*)⟩. The proof is precisely the same as the proof of the analogous claim is **Case ii**. But *left* is assigned to j at line 13. So the invariant holds the next time the execution is at line 2.

Case v: $A[left] \not> A[j]$ and $A[right] \not> A[j]$ at the beginning. But that is impossible under the current assumptions, because clearly line 15 would be reached and the execution would never get to line 2 again.

Termination. Unlike the examples we have see so far, this loop can be exited in two ways: via line 2 when $j > \left\lfloor \frac{n}{2} \right\rfloor$ and via line 15. Let us consider the first possibility. Then both LEFT(j) and RIGHT(j) point outside $A[1, \ldots, n]$. The invariant, therefore, says there are no heap obstructions in A⟦i⟧ at all since A[LEFT(j)] and A[RIGHT(j)] do not those exist. And so A⟦i⟧ is heap.

Consider the second possibility, *viz.* the **while** loop starts executing but the execution reaches line 15. It is obvious that, in order line 15 to be reached, it has to be the case that *largest* = j at line 11. In order that to happen, both $A[left] \leq A[j]$ and $A[right] \leq A[j]$ must be true at the beginning of that execution since that is the only way that line 8 is reached and line 10 is not reached. But if $A[left] \leq A[j]$ and $A[right] \leq A[j]$, there are no heap obstructions in A⟦i⟧ at all, so A⟦i⟧ is a heap. □

The function HEAPIFY in the following algorithm is either one of RECURSIVE HEAPIFY or ITERATIVE HEAPIFY.

```
BUILD HEAP(A[1,2,...,n]: array of integers)
1  for i ← n downto 1
2      HEAPIFY(A[], i)
```

Lemma 16. *When* BUILD HEAP *terminates,* A[] *is a heap.*

Proof:
The following is a loop invariant for the **while** loop (lines 1–2):

> Every time the execution is at line 1 and d is the depth of A[i], then for every element A[j] of depth d − 1, A⟦j⟧ is a heap.

The make use the of the fact—without proving is—that for every depth d, the elements of A[] of depth d form a continuous subarray in A[]. If we call that subarray, *the* d-*block*, clearly those blocks appear in reverse sorted order:

- A[1] is the $\lfloor \lg n \rfloor$-block†.
- ...
- $A\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n\right]$ is the 0-block (the leaves).

†Recall that the height of any n element heap is $\lfloor \lg n \rfloor$ (see eq. (6.26) on page 170) and note that the height of the heap equals the depth of the root.

So, as the index i starts from n and goes down to 1, the depths of the $A[i]$ elements take all values from $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ in ascending order. Furthermore, Lemma 28 on page 172 implies that the d-block is precisely $A\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1, \ldots, \left\lfloor \frac{n}{2^d} \right\rfloor\right]$.

In order to make the proof work, assume that $A[\,]$ has an dummy element at position 0. The value of that $A[0]$ does not concern us. What is important is to consider the remainder of the array, namely $A[1, \ldots, n]$, as a subtree of $A[0]$. In other words, the depth of $A[0]$ is $\lfloor \lg n \rfloor + 1$. Without that technicality, the termination phase of the proof could not be formulated because i equals 0 when the execution is at line 2 for the last time.

Basis. Unlike the previous proofs, here the basis is not for a single value of the loop control variable but for a multitude of values. To see why, note that the induction is on the depth of the elements, not on the number of times line 2 is reached. It is natural to take for basis depth 0. But that means that the basis is over all leaves of $A[1, \ldots, n]$. The leaves are precisely the elements $A[i]$ such that $\lfloor \frac{n}{2} \rfloor + 1 \leq i \leq n$ (see Problem 131 on page 170).

Consider any i be such that $\lfloor \frac{n}{2} \rfloor + 1 \leq i \leq n$. Then $A[i]$ has depth 0. But elements of depth -1 do not exist. So the claim is vacuously true.

Maintenance. Relative to some number d such that $0 \leq d \leq \lfloor \lg n \rfloor$[†], consider the first time the execution is at line 2 and the depth of the current $A[i]$ is d. Since the depths of the $A[i]$ elements take all values from $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ in ascending order, such a moment will occur. As implied by Lemma 28 on page 172, at that moment $i = \lfloor \frac{n}{2^d} \rfloor$. Assume that the claim holds at that moment.

⁉ NB ⁉ *Our goal now is **not** to prove, using this assumption, that the claim holds the next time the execution is at line 2. Such an implication **does not exist**.*

Our goal is to prove that at the subsequent moment when the execution is at line 2 and for the first time, the depth of the current $A[i]$ is $d + 1$, it is the case that for all indices j such that $A[j]$ is at depth d, $A[\![j]\!]$ is a heap. As implied by Lemma 28 on page 172, that subsequent moment is when $i = \lfloor \frac{n}{2^{d+1}} \rfloor$.

In other words, the maintenance phase consists of the following:

- assuming that

$$\underbrace{A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor + 1\right]\!\right], A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor + 2\right]\!\right], \ldots, A\left[\!\left[\left\lfloor \frac{n}{2^{d-1}} \right\rfloor\right]\!\right]}_{\text{the indices here are from the } d-1 \text{ block}}$$

  are heaps,

- show that

$$\underbrace{A\left[\!\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1\right]\!\right], A\left[\!\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 2\right]\!\right], \ldots, A\left[\!\left[\left\lfloor \frac{n}{2^d} \right\rfloor\right]\!\right]}_{\text{the indices here are from the } d \text{ block}}$$

  are heaps.

[†]In other words, d can be the depth of any vertex from $A[1, \ldots, n]$.

It is rather obvious that the the map

$$j \to \{\textsc{Left}(j), \textsc{Right}(j)\}$$

maps the d-block on the $(d-1)$-block in the sense that the sets $\{\textsc{Left}(j), \textsc{Right}(j)\}$ are partitioning of the indices of the $(d-1)$-block, if j takes its values from the set of indices of the d-block. Of course, that holds if $d > 0$.

Apply Lemma 15 or Lemma 14, whichever one is applicable (depending on whether the recursive or the iterative HEAPIFY is used) on every one of

$$A\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1\right], A\left[\left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 2\right], \ldots, A\left[\left\lfloor \frac{n}{2^{d}} \right\rfloor\right]$$

For each index

$$j \in \left\{ \left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 1, \left\lfloor \frac{n}{2^{d+1}} \right\rfloor + 2, \ldots, \left\lfloor \frac{n}{2^{d}} \right\rfloor \right\}$$

the premises of the Lemma include the assumption that $A[\![\textsc{Left}(j)]\!]$ an $A[\![\textsc{Right}(j)]\!]$ are heaps. But those two subtrees are from the set

$$\left\{ A\left[\left\lfloor \frac{n}{2^{d}} \right\rfloor + 1\right], A\left[\left\lfloor \frac{n}{2^{d}} \right\rfloor + 2\right], \ldots, A\left[\left\lfloor \frac{n}{2^{d-1}} \right\rfloor\right] \right\}$$

and we did assume the elements of this set are heaps. So, the said Lemma provides the desired result.

Termination. When the execution is at line 2 for the last time, $i = 0$. We agreed that the dummy $A[0]$ is of depth $\lfloor \lg n \rfloor + 1$. Plug that value in the invariant to derive that every element of $A[\,]$ of depth $\lfloor \lg n \rfloor$, and the only such element is $A[1]$, it is the case that $A[\![1]\!]$ is a heap. □

The following pseudocode uses the notation "A.size". Assume that is a number such that $1 \le A.\text{size} \le n$, and HEAPIFY works using it as an upper index of the array–heap, **not** n as the pseudocode of HEAPIFY says.

```
HEAP SORT(A[1, 2, ..., n])
1  BUILD HEAP(A[])
2  A.size ← n
3  for i ← n downto 2
4      swap(A[1], A[i])
5      A.size ← A.size − 1
6      HEAPIFY(A[], 1)
```

Lemma 17. HEAP SORT *is a sorting algorithm.*

Proof:
Let us call the original array, $A'[1, \ldots, n]$. The following is a loop invariant for the **for** loop (lines 3–6):

> Every time the execution of HEAP SORT is at line 3, the current subarray $A[i+1, \ldots, n]$ consists of $n - i$ in number biggest elements of $A'[1, \ldots, n]$. Furthermore, the current $A[1, \ldots, i]$ is a heap.

Basis. The first time the execution reaches line 3, it is the case that $i = n$. The current subarray $A[i+1, \ldots, n]$ is empty and thus, vacuously, it consists of zero in number biggest elements from $A'[1, \ldots, n]$, in sorted order. $A[1, \ldots, n]$ is a heap by Lemma 17, applied to line 1.

Maintenance. Assume the claim holds at a certain execution of line 3 and the **for** loop is to be executed at least once more. Let us call the array $A[]$ at that moment, $A''[]$. By the maintenance hypothesis, $A''[i+1, \ldots, n]$ are $n-i$ in number maximum elements of $A'[]$, in sorted order. By the maintenance hypothesis again, $A''[1]$ is a maximum element of $A''[1, \ldots, i]$. After the swap at line 4, $A''[i, \ldots, n]$ are $n-i$ in number maximum elements of $A'[]$, in sorted order. Relative to the new value of i the next time the execution is at line 3, the first sentence of the invariant holds.

The second sentence holds, too, by applying Lemma 15 or Lemma 14, whichever one is applicable (depending on whether the recursive or the iterative HEAPIFY is used), at line 6. Just keep in mind that HEAPIFY considers the heap to be $A[1, \ldots, i-1]$ because i equals A.size when the execution is at line 6; note that because of line 5, A.size is $i-1$ at line 6. Thus at line 6, the current $A[i]$ is outside the scope of the current heap.

Termination. Consider the moment when the execution is at line 3 for the last time. Clearly, i equals 1. Plug the value 1 in place of i in the invariant to obtain "the current subarray $A[2, \ldots, n]$ consists of $n-1$ in number biggest elements of $A'[1, \ldots, n]$.". But then $A[1]$ has to be a minimum element from $A'[1, \ldots, n]$. And that concludes the proof of the correctness of HEAP SORT. □

Chapter 5

Algorithmic Problems

5.1 Programming fragments

Problem 101. *Determine the asymptotic running time of the following programming fragment. Function* `q()` *is an unspecified function, it works in* $\Theta(1)$ *time and can swap elements of the array* `A`. `Heapsort(A, i, j)` *sorts* `A[i..j]` *by the eponymous sorting algorithm.*

```
int A[MAXINT];

int p(int, int);
void q(int, int);

int main() { return p(1, n); }

int p(int i, int j) {
  int mid, a, b;
  if (j > i) {
   Heapsort(A, i, j);
   q(i, j);
   mid = (i+j)/2;
   a = p(i, mid);
   b = p(mid + 1, j);
   return a + b; }
  return 1; }
```

Solution:
It is well known that `Heapsort` has $\Theta(n \lg n)$ worst case time complexity. We have to assume that at every recursive level its complexity is the worst possible because we do not know how `q()` works. There are two resursive calls at each execution of `p()`, therefore the running time is determined by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \lg n)$$

Using Theorem 2 on page 91, we conclude $T(n) = \Theta(n \lg^2 n)$. □

Problem 102. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
void r(int, int, int);

int main() { return r(1, n, n*n); }

void r(int a, int b, int c) {
  int k;
  if (a + b + c > a + b + 1) {
   for(k = 1; k < a + b + c; k = (k << 2) - 1) {
    if(k % 3 == 0) break;
    r(a, b, c - 1); }
   for(k = 1; k < a + b + c; k <<= a + b + c)
    r(a, b, c - 1); } }
```

Solution:
Only the third parameter of `r()` determines the recursion; the first two parameters are insignificant. The body of the first `for` loop is executed precisely once because the second value that `k` gets is $(1 * (2^2) - 1) = 3$, then the condition of the `if` operator is evaluated to TRUE, and after the `break` operator the loop is executed no more.

The second `for` loop is executed only once, too: the second value that `k` gets is 2^{a+b+c}, and certainly $2^{a+b+c} > a + b + c$. There are two resursive calls at each execution of `r()`, therefore the running time is determined by the recurrence

$$T(m) = 2T(m-1) + \Theta(1)$$

where m is the size of the input. The solution is known to be $T(m) = \Theta(2^m)$. Having in mind that $m = n^2$, it is easy to see that $T(n) = \Theta\left(2^{n^2}\right)$. □

Problem 103. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int foo(int n) {
  int s;
  if (n == 0) s = 1;
  else s = foo(n - 1) * 2;
  bar(s);
  return s; }

void bar(int m) {
  int i;
  for (i = m; i > 0; i /= 2)
    cout << i % 2; }
```

Solution:
First we prove that `foo(n)` returns 2^n. Note that the `bar()` function, while affecting the running time, does not affect the value of `s` is any way. We prove the claim by induction

on `n`. For $n = 0$ it is obviously the case that `foo(n)` returns $1 = 2^0$. Assuming `foo(n-1)` returns 2^{n-1}, it is clear `foo(n)` returns $2 \times 2^{n-1} = 2^n$.

Note that `bar()` runs in time $\Theta(\lg m)$. Having in mind that `s` is 2^n and `bar()` runs in logarithmic time with respect to its input, it is clear that `bar(s)` runs in time $\Theta(\lg(2^n)) = \Theta(n)$. We conclude the time complexity of `foo()` is determined by the recurrence

$$T(n) = T(n-1) + n$$

According to (3.19) on page 51, $T(n) = \Theta(n^2)$. □

Problem 104. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int recf(int n) {
  int s;
  if (n == 0) s = 1;
  else s = foo(n - 1) * 2;
  bar(s);
  return s; }

void bar(int m) {
  int i;
  for (i = m; i > 0; i /= 2)
    cout << i % 2; }
```

Solution:
First we prove that `foo(n)` returns 2^n. Note that the `bar()` function, while affecting the running time, does not affect the value of `s` is any way. We prove the claim by induction on `n`. For $n = 0$ it is obviously the case that `foo(n)` returns $1 = 2^0$. Assuming `foo(n-1)` returns 2^{n-1}, it is clear `foo(n)` returns $2 \times 2^{n-1} = 2^n$.

Note that `bar()` runs in time $\Theta(\lg m)$. Having in mind that `s` is 2^n and `bar()` runs in logarithmic time with respect to its input, it is clear that `bar(s)` runs in time $\Theta(\lg(2^n)) = \Theta(n)$. We conclude the time complexity of `foo()` is determined by the recurrence

$$T(n) = T(n-1) + n$$

According to (3.19) on page 51, $T(n) = \Theta(n^2)$. □

Problem 105. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int recf(int n) {
  int i, s = 1;
  if (n == 1) return s;
  for (i = 0; i < 3; i ++) s += recf(n-1) * (i + 1);
  for (i = 0; i < 4; i ++) s += recf(n-2) * (i + 1);
  return s; }
```

Solution:
The time complexity of `recf()` is determined by the recurrence

$$T(n) = 3T(n-1) + 4T(n-2) + 1$$

To see why, note there are three recursive calls with input of size $n-1$ and four, with input of size $n-2$. According to Problem 97 on page 98, $T(n) = \Theta(4^n)$. □

⁘ NB ⁘ *We should not try to "optimise" the number of recursive calls. One may indeed be tempted to think we can make only one recursive call with input $n-1$ and then use the obtained value three times, rather than making three consecutive recursive calls (and likewise, only one call with input $n-2$ and then use the result four times. Such "optimisations" are not allowed: the algorithm should be investigated as it is. Furthermore, it is possible that the shown fragment is an abbreviated version of a program that does a lot more, for instance it may change a global variable. If that is the case, we cannot substitute a multitude of recursive calls by a single call and claim that the new "optimised" program is necessarily equivalent.*

Problem 106. *Determine the asymptotic running time of the following programming fragment as a function of n.*

```
int r(int n, int m) {
  int i, s = 0;
  if (n < 2) return n*m + 1;
  for(i = 0; i < 3; i ++) {
    s += r(n-1,m+i) * r(n-2,m-i); }
  s += r(n-1,m);
  return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 4T(n-1) + 3T(n-2) + 1$$

To see why that is true note that the variable that determines the recursive calls is n. The other variable m does not affect the time complexity, though it surely affects the returned quantity. There are three recursive calls with $n-2$ and four with $n-1$. The fact that there is a multiplication $r(n-1, m+i) * r(n-2, m-i)$ is immaterial with respect to the structure of the recursive calls. According to Problem 98 on page 98, $T(n) = \Theta((2+\sqrt{7})^n)$. □

Problem 107. *Determine the asymptotic running time of the following programming fragment as a function of n.*

```
int f(int n, int m) {
  int i, s = 0;
  if (n == 0 || n == 1)
    return m;
```

```
  for(i = 0; i < 5; i++) {
    s += f(n-1, m + i);
    s += f(n-2, m + 2*i); }
  s += f(n-2, 2*m)*3;
 return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 5T(n-1) + 6T(n-2) + 1$$

To see why, note that the variable that controls the recursive calls is n. The other variable m does not affect the time complexity, though it surely affects the returned quantity. There are six recursive calls with $n-2$ and five with $n-1$. According to Problem 99 on page 99, $T(n) = \Theta(6^n)$. □

Problem 108. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int r(int n) {
  int i, s = 2;
  if (n == 1)
    return 2;
  for(i = n; i > 0; i /= 2) {
    s += 2; }
  s += r(n/2)*r(n/2);
  return s; }
```

Solution:
The recurrence determining the asymptotic time complexity is

$$T(n) = 2T\left(\frac{n}{2}\right) + \lg n$$

To see why, note the `for` loop runs in $\Theta(\lg n)$ time and afterwards two recursive calls are made, each one on an input that is half the size of the original one. According to Problem 83 on page 92, $T(n) = \Theta(n)$. □

Problem 109. *Determine the asymptotic running time of the following programming fragment as a function of* n.

```
int bf(int m, int v) {
  if (m == 1)
    return v;
  return bf(m - 1, 1) * v && bf(m-1, 0) * (1 - v); }

int main() {
  return bf(n, x); }
```

Solution:
At a first glance, it seems the fragment has exponential time complexity: two recursive calls are made, each with input of size $n-1$, and only constant work is done in addition to them, therefore the recurrence $T(n) = 2T(n-1)+1$ determines the complexity, and we know (see Problem 91 on page 97) this recurrence has solution $T(n) = \Theta(2^n)$.

However, if we take into account the operator precedence of the `C` language[†] and the evaluation of the logical operators[‡], the analysis is quite different. Suppose we make the initial call `bf(n, x)` with a sufficiently large value of `n`. The first recursive call is `bf(n - 1, 1)`. Inside it, the first recursive call is `bf(n - 2, 1)`, *etc.*, until `bf(2, 1)` calls `bf(1, 1)`. Clearly, `bf(1, 1)` returns `1` and then `bf(2, 1)` makes its second recursive call, `bf(1, 0)`. The latter returns `0` and the `&&` operator within `bf(2, 1)` evaluates `1 && 0` to `0` and passes `0` upwards to `bf(3, 1)`. Now `bf(3, 1)` does not call `bf(2, 0)` because, by the rules of `C`, the evaluation of the `&&` operator is left to right and it stops once the value is known—`bf(2, 1)` returning `0` implies the result within `bf(3, 1)` is necessarily `0`. Thus `bf(3, 1)` passes `0` upwards, `bf(4, 1)` does not call `bf(3, 0)` but passes `0` upwards directly, *etc.*, until `bf(n, x)` returns `0` without calling `bf(n - 1, 0)`. In other words, the recursion tree is in fact a path, except that at the very bottom, `bf(2, 1)` has two children. It follows that the time complexity is linear in `n` rather than exponential. The recursion tree is shown on Figure 5.1.

Of course, that is the case only under the assumption that the rules of the `C` language apply. If that fragment were written in pseudocode the time complexity would be proved to be exponential since we have no standard rules for the direction and early stopping of the evaluation of logical expressions in pseudocode. □

Problem 110. *Determine the asymptotic running time of the following programming fragment as a function of* `n`.

```
unsigned n;
int main() {
  return recf(1, n); }

int recf(unsigned i, unsigned j) {
  int k, s = 0;
  if (j - i > 3) {
    for(k = 0; k < 4; k ++) {
      s += recf(i + k, j + k - 3); }
    return s; }
  else
  return 1; }
```

Solution:
The execution of the recursion is controlled by the difference `j-i` of the two input variables. Let us call it, *the control difference*. For all large enough values of the control difference, *i.e.* whenever $j - i > 3$, there are exactly four recursive calls, each one having control difference

[†]See [KR88], pp. 53.

[‡]*ibid.*, pp. 41: "*More interesting are the logical operators* `&&` *and* `||`. *Expressions connected by* `&&` *or* `||` *are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.*"

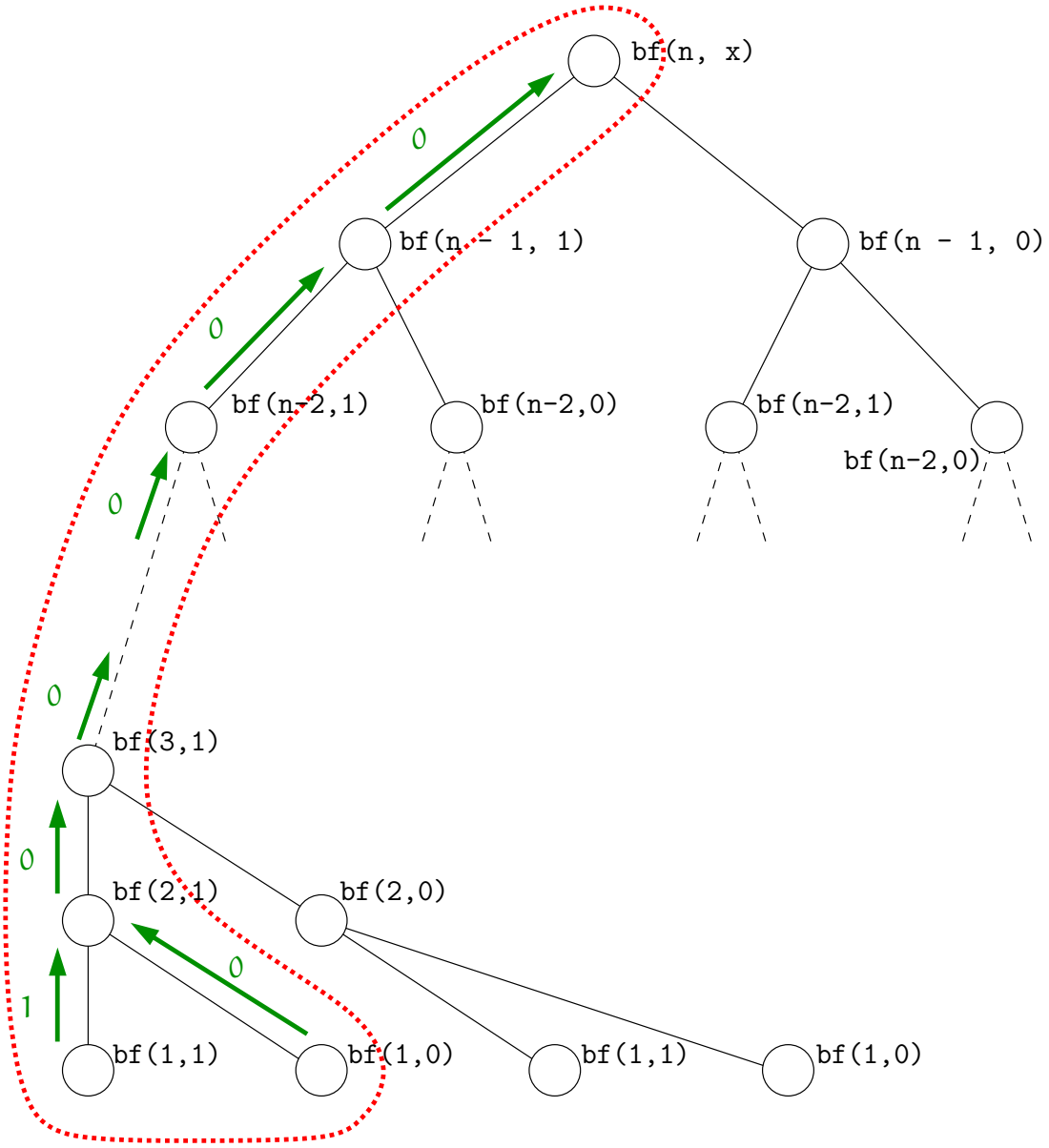


Figure 5.1: The recursion tree of the fragment in Problem 109. We draw the whole tree, *i.e.* what the tree would be if the considerations about the `&&` operator were not taken into account. The part of the tree that corresponds to the execution of the fragment when these considerations are taken into account, is outlined with dashed red line. The flow of the returned values is drawn in green.

that is smaller by three since $j+k-3-(i+k) = j-i-3$ for $k \in \{0,1,2,3\}$. Therefore, the following recurrence relation determines the asymptotic running time:

$$T(n) = 4T(n-3) + 1$$

According to Problem 100 on page 99, $T(n) = \Theta\left(4^{\frac{n}{3}}\right)$. □

5.2 Arrays and sortings

Problem 111. *Let* C_1, C_2, ..., C_n *be* n *cities placed along a straight line. Let* d_i *be the distance between* C_i *and* C_{i+1}, *for* $1 \leq i < n$. *At some moment, each city* C_i *possesses some amount* $x_i \in \mathbb{R}$ *of a certain resource, say water. Since* x_i *can be negative, what* C_i *possesses can be not physical water but deficiency of water,* e.g. *if* C_i *has* -5.5 *units of water and we transport* 6 *units of water to it, it is going to have* $+0.5$ *units. Each city* C_i *needs some amount* $l_i \in \mathbb{R}^+$ *of water. We say* C_i *is* satisfied *iff* $l_i \leq x_i$.

Water can be transported between any two adjacent cities but the transportation is lossy: if we start transporting amount z *from* C_i *to* C_{i+1} *or* vice versa, *the amount that is going to be delivered is* $\max\{z - d_i, 0\}$.

Design an algorithm that outputs TRUE, *in case there is a way to transport water between the cities so that every city is satisfied, or* FALSE, *otherwise. Assume that the amount of water in any city is constant unless water is transported to, or from, it. Prove the correctness of your algorithm and analyse its time complexity.*

Solution: Define d_n to be zero. Consider the following algorithm:

```
TRANSPORT(x_1, ..., x_n, d_1, ..., d_n, l_1, ..., l_n)
 1  s ← 0
 2  for i ← 1 to n
 3       Δ ← x_i − l_i
 4       if s + Δ ≥ 0
 5            s ← max{s + Δ − d_i, 0}
 6       else
 7            s ← s + Δ − d_i
 8  if s ≥ 0
 9       return TRUE
10  else
11       return FALSE
```

For any i such that $1 \leq i \leq n$, we say the subarray $X_i = [x_1, x_2, \ldots, x_i]$ is *good* if there is a way to satisfy the cities C_1, C_2, ..., C_i by transporting water only between them; we say the subarray $X_i = [x_1, x_2, \ldots, x_i]$ is *wanting* if there is no way to satisfy the cities C_1, C_2, ..., C_i by transporting water only between them. Suppose q is a positive amount. When we say X_i is q-*redundant* we mean:

- there is a way to satisfy the cities C_1, C_2, ..., C_i by transporting water only between them, under the premise that the amount in C_i is decreased by q units beforehand.

- there is no way to satisfy the cities C_1, C_2, ..., C_i by transporting water only between them, under the premise that the amount in C_i is decreased by $q+\epsilon$ units beforehand, for any $\epsilon > 0$.

When we say X_i is *closed* we mean the cities C_1, C_2, ..., C_i are satisfied but if $i < n$ and we transport $d_i + \epsilon$ units of water, for any $\epsilon > 0$, from C_i to C_{i+1} then there is no way they can be satisfied. When we say X_i is q-*deficient* we mean:

- there is a way satisfy the cities C_1, C_2, ..., C_i by transporting water only between them, under the premise that the amount in C_i is increased by q units beforehand.
- there is no way satisfy the cities C_1, C_2, ..., C_i by transporting water only between them, under the premise that the amount in C_i is increased by $q-\epsilon$ units beforehand, for any $\epsilon > 0$.

Note that if X_i is q-redundant or closed then it is good, and if X_i is q-deficient then it is wanting.

The following is a loop invariant for TRANSPORT.

Lemma 18. $\forall t \in \{2, 3, \ldots, n+1\}$, *when the execution reaches line 2 for the* t*-th time, it is the case that :*

- *if* $s \geq 0$ *then* X_{t-1} *is* $(s + d_t)$*-redundant.*
- *if* $s = 0$ *then* X_{t-1} *is closed.*
- *if* $s < 0$ *then* X_{t-1} *is* $|s + d_t|$*-deficient.*

Proof:
By induction on t.

Basis: The basis is for $t = 2$: suppose the first execution of the body of the **for** loop (lines 2–7) has just completed and currently the execution is at line 2 for the second time. Before the first execution of the **for** loop, s is initialised to zero (line 1). During the first execution of the **for** loop, $s + \Delta$ at line 4 is in fact Δ. Call s_{new} the value of s after the assignment at line 5 or line 7, whichever is applicable. Call s_{old}, the former value of s. As mentioned, $s_{old} = 0$. Consider the three possibilities for s_{new}.

Case I: Suppose $s_{new} > 0$.

- Suppose the execution went through line 5. That means the condition in the **if** instruction at line 4 was TRUE, *i.e.* $\Delta \geq 0 \Leftrightarrow x_1 \geq l_1$. We prove X_1 is $(s_{new} + d_1)$-redundant. Clearly, X_1 being $(s_{new}+d_1)$-redundant is the same as $x_1-l_1-(s_{new}+d_1) = 0$ and $x_1 - l_1 - (s_{new} + d_1 + \epsilon) < 0$ for any $\epsilon > 0$.
  - If d_1 is such that $s_{old}+\Delta-d_1 \geq 0 \Leftrightarrow 0+x_1-l_1-d_1 \geq 0$ then $s_{new} = x_1-l_1-d_1$, thus:

    $$x_1 - l_1 - (s_{new} + d_1) = x_1 - l_1 - (x_1 - l_1 - d_1 + d_1) = 0 \text{ , and}$$
    $$\forall \epsilon > 0, x_1 - l_1 - (s_{new} + d_1 + \epsilon) = x_1 - l_1 - (x_1 - l_1 - d_1 + d_1 + \epsilon) = -\epsilon$$

    so X_1 is $(s_{new} + d_t)$-redundant.

– Assuming d_1 is such that $s_{old} + \Delta - d_1 = 0$ implies $s_{new} = 0$, which conradicts our previous assumptions.
– Assuming d_1 is such that $s_{old} + \Delta - d_1 < 0$ implies $s_{new} = 0$, which conradicts our previous assumptions.

Clearly, X_1 is $(s_{new} + d_t)$-redundant.

- Suppose the execution went through line 7. But in this case, the condition in the **if** instruction at line 4 was FALSE, *i.e.* $\Delta < 0$. It follows at line 7, s is assigned a negative value because $d_1 > 0$ and therefore $\Delta - d_1 < 0$. Our supposition is wrong: s could not possibly get its positive value s_{new} at line 7.

Case II: Suppose $s_{new} = 0$. It is trivial, in a way analogous to above, to show that the execution could not have gone through line 7 because, if that were the case, s_{new} would be negative. Therefore, the execution must have gone through line 5. It is clear that $\Delta - d_1 \leq 0$ in this case. It follows $\Delta - d_1 - \epsilon < 0$ for any $\epsilon > 0$, therefore there is no way to transport $d_1 + \epsilon$ units out of C_1 and keep C_1 satisfied. So, X_1 is closed.

Case III: Suppose $s_{new} < 0$. Obviously, that value was not assigned to s at line 5, so it must be the case the negative value was assigned to s at line 7. It follows $s_{old} + \Delta < 0 \Leftrightarrow 0 + \Delta < 0$. We prove X_1 is $|s_{new} + d_1|$-deficient. On the one hand:

$$x_1 - l_1 + |s_{new} + d_1| =$$
$$x_1 - l_1 + |\Delta - d_1 + d_1| =$$
$$x_1 - l_1 + |\Delta| =$$
$$x_1 - l_1 + (l_1 - x_1) = 0$$

On the other hand:

$$\forall \epsilon > 0, x_1 - l_1 + |s_{new} + d_1| - \epsilon = -\epsilon < 0$$

It follows X_1 is $|s_{new} + d_1|$-deficient.

Inductive hypothesis: The claim is true for some $t \leq n$.

Inductive step: Consider all the possible ways to execute the body of the **for** loop. Note that $i = t$ and Δ is set to $x_t - l_t$ at line 3. Let us call the value of s prior to the execution of line 5 or line 7, s_{old}, and the value of s after that execution, s_{new}.

Case I: Suppose $s_{old} > 0$. By the induction hypothesis, X_{t-1} is $(s_{old} + d_{t-1})$-redundant. It follows we can deliver s_{old} units of water to C_t, by transporting $s_{old} + d_{t-1}$ from C_{t-1} to C_t, and still C_1, C_2, ..., C_{t-1} be satisfied. However, if we transport any more water, the cities C_1, C_2, ..., C_{t-1} will not be satisfied.

Case I.1: Suppose $s_{old} + \Delta \geq 0$. Then the assignment at line 5 takes place.

Case I.1.a: Suppose $s_{old} + \Delta - d_t > 0$. Then $s_{new} = s_{old} + \Delta - d_t$. The amount of water in C_t before any transportation to or from that city takes place, is Δ. To decrease the amount of water in C_t by $s_{new} + d_t$ units means to make it $\Delta - (s_{new} + d_t) = \Delta - (s_{old} + \Delta - d_t + d_t) = -s_{old}$. As mentioned, it is possible to transport s_{old} units of water to C_t and still C_1, C_2, ..., C_{t-1} be satisfied, but if we transport any less water, the cities C_1, C_2, ..., C_{t-1} will not be satisfied. It follows X_t is $(s_{new} + d_t)$-redundant.

Case I.1.b: Suppose $s_{old} + \Delta - d_t = 0$. Then $s_{new} = 0$. The fact that $s_{old} + \Delta - d_t = 0$ has two implications. First, $s_{old} + \Delta > 0$, therefore C_1, C_2, ..., C_{t-1}, C_t can be satisfied

because we can transport s_{old} units of water from C_{t-1} to C_t. Second, $s_{old} + \Delta - d_t - \epsilon < 0$ for any $\epsilon > 0$. It follows X_t is closed.

Case I.2: Suppose $s_{old} + \Delta < 0$, which means $\Delta < 0$ since $s_{old} > 0$. Then the assignment at line 7 takes place and $s_{new} = s_{old} + \Delta - d_t$, which is a negative amount since $d_t > 0$. Clearly, $|s_{new} + d_t| = |s_{old} + \Delta| = -\Delta - s_{old} > 0$. To increase the amount of water in C_t by $|s_{new} + d_t|$ units means to make it $\Delta + (-\Delta - s_{old}) = -s_{old}$. As mentioned, it is possible to transport s_{old} units of water to C_t and still $C_1, C_2, \ldots, C_{t-1}$ be satisfied, but if we transport any less water, the cities $C_1, C_2, \ldots, C_{t-1}$ will not be satisfied. It follows X_t is $|s_{new} + d_t|$-deficient.

Case II: Suppose $s_{old} = 0$. By the induction hypothesis, X_{t-1} is closed. It follows $C_1, C_2, \ldots, C_{t-1}$ are satisfied but we cannot transport $d_{t-1} + \epsilon$ units of water from C_{t-1} to C_t, for any $\epsilon > 0$, and keep $C_1, C_2, \ldots, C_{t-1}$ satisfied. So, the status of $C_1, C_2, \ldots, C_{t-1}, C_t$ depends entirely on C_t and d_t.

Case II.1: Suppose $s_{old} + \Delta \geq 0 \Leftrightarrow \Delta \geq 0$. Then the assignment at line 5 takes place.

Case II.1.a: Suppose $\Delta - d_t > 0$. Then $s_{new} = \Delta - d_t$. To decrease the amount of water in C_t by $s_{new} + d_t$ units means to make it $\Delta - (s_{new} + d_t) = \Delta - (\Delta - d_t + d_t) = 0$. It immediately follows X_t is $(s_{new} + d_t)$-redundant.

Case II.1.b: Suppose $\Delta - d_t = 0$. Then $s_{new} = 0$. So, C_t is satisfied since $\Delta > 0$ but, if $t < n$, we cannot transport any amount of water more than d_t from C_t to C_{t+1}, keeping C_t satisfied. It follows X_t is closed.

Case II.2: Suppose $s_{old} + \Delta < 0 \Leftrightarrow \Delta < 0$. Then the assignment at line 7 takes place and $s_{new} = \Delta - d_t$, which is a negative amount since $d_t > 0$. Clearly, $|s_{new} + d_t| = |\Delta| = -\Delta > 0$. To increase the amount of water in C_t by $|s_{new} + d_t|$ units means to make it $\Delta + (-\Delta) = 0$. It follows X_t is $|s_{new} + d_t|$-deficient.

Case III: Suppose $s_{old} < 0$. By the induction hypothesis, X_{t-1} is $|s_{old} + d_{t-1}|$-deficient. It follows $C_1, C_2, \ldots, C_{t-1}$ are not satisfied but if we deliver at least $|s_{old} + d_{t-1}|$ units of water to C_{t-1} they can be satisfied. To deliver at least $|s_{old} + d_{t-1}|$ units of water to C_{t-1} means to transport at least $|s_{old} + d_{t-1}| + d_{t-1}$ units of water from C_t to C_{t-1}, in order to compensate for the loss of d_{t-1} units along the way.

We claim that $|s_{old}| > d_{t-1}$. To see why, note that the negative value s_{old} was assigned to s during the previous execution of the **for**-loop at line 7: in order to reach that line, it must have been the case that $s + \Delta < 0$. Since $-d_i$ at line 7 is negative too, it follows the absolute value of what is assigned to s at line 7 is strictly larger than the current d_i. With respect to the current discussion, i is $t - 1$.

Having proved that $|s_{old}| > d_{t-1}$ and having in mind that $s_{old} < 0$, it is obvious that $|s_{old} + d_{t-1}| + d_{t-1} = |s_{old}|$. So, we can satisfy $C_1, C_2, \ldots, C_{t-1}$ by transporting $|s_{old}|$ units of water from C_t to them, and if we transport any less water then they cannot be satisfied.

Case III.1: Suppose $s_{old} + \Delta \geq 0$. Then the assignment at line 5 takes place.

Case III.1.a: Suppose $s_{old} + \Delta - d_t > 0$. Then $s_{new} = s_{old} + \Delta - d_t$. To decrease the amount of water in C_t by $s_{new} + d_t$ units means to make it $\Delta - (s_{new} + d_t) = \Delta - (s_{old} + \Delta - d_t + d_t) = -s_{old}$, which is a positive amount. As we said, in order to make $C_1, C_2, \ldots, C_{t-1}$ satisfied, we have to transport at least $|s_{old}|$ units of water from C_t to them. Suppose we transport precisely $|s_{old}|$ units of water: then $C_1, C_2, \ldots, C_{t-1}$ will be satisfied, and C_t will be satisfied, too, having zero water in it. However, if we transport any more water to them, then C_t will be left with a negative amount and will not be satisfied. It follows X_t is $(s_{new} + d_t)$-redundant.

Case III.1.b: Suppose $s_{old} + \Delta - d_t = 0$. Then $s_{new} = 0$. We prove that X_t is closed.

Note that $\Delta = -s_{old} + d_t$ and $-s_{old}$ is a positive amount. It follows that if we transport $|s_{old}|$ units of water from C_t back to C_{t-1}, it will be the case that C_1, C_2, ..., C_{t-1} will be satisfied, and C_t will be satisfied, too, having d_t water in it. However, if $t < n$, it is not possible to both satisfy C_1, C_2, ..., C_{t-1}, because we have to transport at least $|s_{old}|$ out of C_t to them, and be able to deliver positive amount of water to C_{t+1} from C_t, because we have only d_t units at our disposal for that purpose, and d_t units will be lost along the way from C_t to C_{t+1}. It follows X_t is closed.

Case III.2: Suppose $s_{old} + \Delta < 0$. Then the assignment at line 7 takes place and $s_{new} = s_{old} + \Delta - d_t$, which is a negative amount since $d_t > 0$. We prove X_t is $|s_{new} + d_t|$-deficient. Clearly, $|s_{new} + d_t| = |s_{old} + \Delta| = -\Delta - s_{old} > 0$. To increase the amount of water in C_t by $|s_{new} + d_t|$ units means to make it $\Delta + (-\Delta - s_{old}) = -s_{old}$, which is a positive amount. It follows that if we transport $|s_{old}|$ units of water from C_t back to C_{t-1}, it will be the case that C_1, C_2, ..., C_{t-1} will be satisfied, and C_t will be satisfied, too, having zero water in it. However, if we transport back any less water, the cities C_1, C_2, ..., C_{t-1} will not be satisfied. It follows X_t is $|s_{new} + d_t|$-deficient.

Termination: Having proved the loop invariant, suppose we are the beginning of the **for**-loop when $t = n + 1$ and the loop is to be executed no more. If the current s is non-negative then $X_{t-1} = X_n$ is either s-redundant (recall then d_n is defined to be zero) or closed, therefore there is a way to transport water so that all cities are satisfied. Accordingly, the retuned value (line 9) is TRUE. If s is negative then there is no such way. Accordingly, the retuned value (line 11) is FALSE.

The time complexity is obviously $\Theta(n)$. □

Problem 112. *Imagine two rooms with no visibility between them. In one room there* n *numbered light switches* s_1, s_2, ..., s_n. *In the other room there are* n *numbered light bulbs* l_1, l_2, ..., l_n. *It is known that each switch turns on and off to exactly one bulb but we do not know anything about the wiring between the switches and the bulbs. Initially we are in the room with the switches. Our job is to tell the exact wiring,* i.e. *which switch operates which bulb. We are allowed to press any switches and then go to the room with the bulbs and perform an observation. We are not allowed to touch the bulbs – our only source of information is the observation of the bulbs.*

The switches are such that their physical appearance does not change when toggled so we have no way of knowing beforehand whether pressing a certain switch leads to turning on or turning off of a bulb. Every swtich has, of course, two states only, as any normal light switch.

Describe an algorithm that discovers the wiring with minimum number of observations, i.e. *with minimum visits to the room with the bulbs. The algorithm should work iteratively, at each iteration simulating toggling some switches and then simulating an observation by calling some function* OBSERVE. *The toggling is simulated by writing into a* 0-1 *array* $P[1, \ldots, n]$. *Say,* $P[i] = 1$ *means* s_i *is toggled, and* $P[i] = 0$ *means* s_i *is not toggled. The result of the "observation" is written in some* 0-1 *array* $L[1, \ldots, n]$. *Say,* $L[i] = 1$ *means* l_i *is on, and* $L[i] = 0$ *means* l_i *is off. After every call to* OBSERVE, *the algorithm temporarily halts, the execution is supposed to be transfered to an outside agent and the algorithm resumes after the outside agent finishes writing into* L.

Prove an asymptotic lower bound for the number of observations. Is your algorithm optimal in the asymptotic sense?

Solution:
Our algorithm maintains the following data structures:

- a 0-1 array $A[1,\ldots,n]$ to keep the result of the previous observation,
- an array $S[1,\ldots,n]$ that refers to the switches. Every element of S is a pointer. Namely, $S[i]$ points to a (doubly) linked list that represents the set of the bulbs, each of which can possibly be connected to s_i according to currently available information. We call this set of bulbs, *the candidate set for* s_i.
- an array B of positive integers of size $2n$. During iteration i, B contains $2i$ elements that represent nonintersecting subarrays of S; the set of all those subarrays must cover S. $B[2j]$ and $B[2j+1]$ are numbers such that $B[2j] \leq B[2j+1]$ and the pair $\langle B[2j], B[2j+1]\rangle$ represents the subarray $S[\,B[2j],\ldots,B[2j+1]\,]$.
- a multitide of doubly linked lists with n elements altogether. They represent a partition of the set of the bulbs. Each element contains an integer that corresponds to the ID of precisely one bulb, and each list represents precisely one candidate set. Initially, there is only one list in this multitude, call this list C. That reflects the fact that at the beginning we have no restrictions on the possible connections between bulbs and switches. At the end, there are n non-empty lists in this multitude. That reflects the fact that at the end we know precisely the wiring between the switches and the bulbs.

Here is the pseudocode. Initially $P[\,]$ is arbitrary.

```
SWITCHES AND BULBS()
 1  create doubly linked list C of n elements, one for each bulb
 2  create S and set every pointer in it to C
 3  B ← [1, n]
 4  OBSERVE()
 5  copy L into A
 6  while the are less than 2n entities in B do
 7      foreach pair ⟨B[2j], B[2j + 1]⟩ such that B[2j] < B[2j + 1]
 8          mid ← ½(B[2j] + B[2j + 1])
 9          set P[ B[2j], ..., mid ] to ones
10          set P[ mid + 1, ..., B[2j + 1] ] to zeros
11      update B so that for each applicable pair ⟨B[2j], B[2j + 1]⟩, it
12            is substituted by two pairs ⟨B[2j], mid⟩ and ⟨mid + 1, B[2j + 1]⟩
13      OBSERVE()
14      for i ← 1 to n
15          if A[i]! = L[i]
16              mark bulb i as changed
17          foreach list of bulbs
18              split the list, if necessary, into two lists: changed and unchanged bulbs
19          foreach element of S
20              update the pointer to the relevant list of bulbs
21      copy L into A
22  for i ← 1 to n
23      print the sole element of the list pointed to by S[i]
```

The query complexity of the algorithm, *i.e.* the number of calls of OBSERVE, is the number of executions of the **while** loop plus one. The number of executions of the **while** loop is logarithmic in n because we split each subarray, delineated by a couple from B, roughly in half, with each execution. So, the number of queries is $\Theta(\lg n)$.

Now we prove an $\Omega(\lg n)$ lower bound of the number of such queries. We use the decision tree model. The decision tree model is used, for instance, for proving an $\Omega(n \lg n)$ lower bound for comparison based sortings (see [CLR00]). However, the decision trees for comparison based sortings are binary because there are precisely two possible outcomes of each comparison of the kind $a_i \stackrel{?}{<} a_j$. In contrast to that, any decision tree that corresponds to the current problem of switches and bulbs has branching factor of 2^n. To why this is true, note that there are precisely 2^n possible outcomes from each observation of the n bulbs.

The current problem is, essentially, computing a permutation, because the mapping from switches to bulbs is a bijection. It follows that any decision tree for that problem has to distinguish all possible $n!$ permutations of n elements: if the decision tree has a leaf labeled by at least two permutations then the corresponding algorithm is not correct. It follows that the leaves must be at least $n!$.

The height of the tree is approximately logarithm to base the branching factor of the number of leaves:

$$\log_{2^n} n! = \frac{\log_2 n!}{\log_2 2^n} = \frac{\Theta(n \lg n)}{n} = \Theta(\lg n)$$

The height of the tree is a lower bound for the query complexity of any observation-based algorithm for the problem of switches and bulbs. It follows that $\Theta(\lg n)$ observations are required if the only testing allowed is direct observation. It follows that algorithm SWITCHES AND BULBS is asymptotically optimal with respect to the number of performed ovservations. □

Problem 113 ([CLR00], Problem 4-2, Finding the missing integer)**.** *An array* $A[1, \ldots, n]$ *contains all the integers from* 0 *to* n *except one. It would be easy to determine the missing integer in* $O(n)$ *time by using an auxiliary array* $B[0, \ldots, n]$ *to record which numbers appear in* A*. In this problem, however, we cannot access an entire integer in* A *with a single operation. The elements of* A *are represented in binary, and the only operation we can use to access them is "fetch the* j*-th bit of* $A[i]$*," which takes constant time. Show that if we use only this operation, we can still determine the missing integer in* $O(n)$ *time.*

Solution:
Let m be the number of bits required to represent n in binary. It is well known that $m = \lfloor \log_2 n \rfloor + 1$. In this problem we think of A as an $m \times n$, 0-1 matrix. Row number m of A consists of the least significant bits of the numbers in A, row number $m - 1$ consists of the second least significant bits, *etc.*, row number 1 consists of the most significant bits. For instance, if $n = 10$ and the missing number is $6 = 0110_b$, A may look like:

$A =$

0	0	1	0	0	0	0	1	1	0
0	0	0	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0	0	1
1	1	0	1	0	1	0	1	0	0

The only constant time access to it is of the form A[j][i], which the authors of [CLR00] call "fetch the j-th bit of A[i]", assuming the first bit is the most significant, *etc.*

Consider the following program in C.

```
int m = floor(logb(n)) + 1;
int A[m][n];

int main() {
  int i, j, t, numrow, n0, n1, ntemp = n;
  int B[n], res[m];
  for(i = 0; i < n; i ++)
    B[i] = i;
  for(i = m - 1; i >= 0; i --) {
    n0 = n1 = 0;
    for(j = 0; j < ntemp; j ++) {
      if (A[i][B[j]] == 0) n0 ++;
      else n1 ++; }
    if(n0 - n1 == 2 || n0 - n1 = 1) res[i] = 1;
    if(n0 - n1 == 0 || n0 - n1 == -1) res[i] = 0;
    for(j = 0, t = 0; j < ntemp; j ++)
      if((res[i] == A[i][B[j]])) {
        B[t] = B[j];
        t ++; }
    if(ntemp % 2 == 0) ntemp = (ntemp / 2);
    else if(res[i] == 0) ntemp = floor(ntemp / 2);
    else ntemp = ceil(ntemp / 2);
  }
  for(i = 0; i < n; i ++)
    printf("%d", res[i]);
  }
```

We claim the algorithm implemented by this program solves correctly the problem of determining the missing bit. First we prove it is correct.

Define that a *complete array* of size n is a two dimensional bit array similar to the above A but without any missing column from it. Clearly, such an array has $n + 1$ columns. For instance, a complete array of size 10 would be the following:

0	0	1	0	0	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	1
1	1	0	1	0	1	0	1	0	0	0

Define that an *almost complete array* of size n is a two dimensional bit array with precisely one missing column from it[†]. Now consider any complete array $\tilde{A}$ of size n. Call $\tilde{L}$ the bottom row of $\tilde{A}$. That $\tilde{L}$ consists of the least significant bits of the numbers fron $\tilde{A}$. Let $\tilde{n_0}$ be the number of zeros and $\tilde{n_1}$, the number of ones, in $\tilde{L}$. Let $\tilde{\Delta} = \tilde{n_0} - \tilde{n_1}$. We claim

[†]It follows the array A in the current problem is an almost complete array of size n.

that:

$$\tilde{\Delta} = \begin{cases} 0, & \text{if } n \text{ is odd} \\ 1, & \text{if } n \text{ is even} \end{cases}$$

Indeed, it is trivial to prove by induction that if the number $n+1$ of columns in $\tilde{A}$ is even then $\tilde{\Delta} = 0$ and if it is odd, $\tilde{\Delta} = 1$.

Now consider A: any almost complete array of size n, obtained from $\tilde{A}$ by deleting a column, *i.e.*, the missing number. Let L be the bottom row of A. Let n_0 be the number of zeros and n_1, the number of ones, in L. Let $\Delta = n_0 - n_1$. We claim that:

$$\Delta = \begin{cases} \tilde{\Delta} + 1, & \text{if the missing number is odd} \\ \tilde{\Delta} - 1, & \text{if the missing number is even} \end{cases}$$

Indeed, if the missing number is even there is a 0 less in L in comparison with $\tilde{L}$, while the number of ones is the same; that is, $n_0 = \tilde{n_0} - 1$ and $n_1 = \tilde{n_1}$. Likewise, if the missing number is odd there is a 1 less in L in comparison with $\tilde{L}$, while the number of zeros is the same; that is, $n_0 = \tilde{n_0}$ and $n_1 = \tilde{n_1} - 1$. Having in mind the above considerations, it is clear that:

$$\Delta = \begin{cases} 2, & \text{if } n \text{ is even and the missing number is odd} \\ 1, & \text{if } n \text{ is odd and the missing number is odd} \\ 0, & \text{if } n \text{ is even and the missing number is even} \\ -1, & \text{if } n \text{ is odd and the missing number is even} \end{cases}$$

We conclude that:

$$\Delta \in \{1, 2\} \Rightarrow \text{the least significant bit of the missing number is } 1 \tag{5.1}$$

$$\Delta \in \{-1, 0\} \Rightarrow \text{the least significant bit of the missing number is } 0 \tag{5.2}$$

So, with one linear scan along the bottom row of A we can compute Δ and then in constant time we can compute the least significant bit of the missing number. However, if we attempt a similar approach for the other bits of the missing number, we will end up with $\Omega(n \lg n)$ computation because the number of rows is logarithmic in n. The key observation is that in order to derermine the second least significant bit of the missing number, we need to scan approximately half the columns of A. Namely, if the least significant bit was determined to be 1, for the computation of the second least significant bit we need to scan only the columns having 1 at the bottom row. Likewise, if the least significant bit was determined to be 0, for the computation of the second least significant bit we need to scan only the columns having 0 at the bottom row. Next we explain why this is true.

The number of rows of A is $m = \lfloor \log_2 n \rfloor + 1$. Define that $A_0^{(m-1)}$ is the two dimensional array obtained from A by deleting the columns that have 0 in row $m-1$, and then deleting row $m-1$. Define that $A_1^{(m-1)}$ is the two dimensional array obtained from A by deleting the columns that have 1 in row $m-1$, and then deleting row $m-1$. Call the process of deriving $A_0^{(m-1)}$ and $A_1^{(m-1)}$, *the reduction of* A, and the two obtained arrays, *the reduced arrays*. Let b be the least significant bit of the missing number and $\overline{b}$ be its complement.

Lemma 19. *Under the current naming conventions,* $A_b^{(m-1)}$ *is complete, and* $A_{\overline{b}}^{(m-1)}$ *is almost complete. Furthermore, the missing number in* $A_{\overline{b}}^{(m-1)}$ *is obtained from the missing number in* A *by removing the least significant bit (*i.e., *shift right).*

Proof:
First we will see an example and then make a formal proof. To use the previously given example with $n = 10$, $m = 4$, missing number $6 = 0110_b$, and

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 0&0&1&0&0&0&0&1&1&0\\\hline 0&0&0&1&1&1&0&0&0&0\\\hline 1&0&1&1&0&0&0&0&0&1\\\hline 1&1&0&1&0&1&0&1&0&0\\\hline\end{array}$$

the two derived subarrays are

$$A_0^{(3)} = \begin{array}{|c|c|c|c|c|}\hline 0&0&0&0&1\\\hline 0&0&1&1&0\\\hline 1&0&1&0&0\\\hline\end{array} \qquad A_1^{(3)} = \begin{array}{|c|c|c|c|c|}\hline 1&0&0&1&0\\\hline 0&1&0&0&0\\\hline 1&0&0&0&1\\\hline\end{array}$$

Obviously, $A_0^{(3)}$ is complete and $A_1^{(3)}$ is almost complete: column $\begin{smallmatrix}0\\1\\1\end{smallmatrix}$ is missing from it. The least significant bit of the missing number in A is 0; if we did not know which is the missing number, we could deduce that its least significant bit is 0 by computing the aforementioned $\Delta = 5 - 5 = 0$. Indeed $A_{\overline{0}}^{(3)} = A_1^{(3)}$ is the array that is almost complete. And indeed the missing number in it 011 is obtained from 0110 by shift right.

Let us prove the lemma. It is clear that if $\tilde{A}$ is a complete array of size n, both $\tilde{A}_0^{(m-1)}$ and $\tilde{A}_1^{(m-1)}$ are complete. If n is odd then they contain the same columns (possibly in different order left to right), otherwise $\tilde{A}_1^{(m-1)}$ contains one column more and the other columns are the same. Now imagine A—the array obtained from $\tilde{A}$ by the deletion of precisely one column. If the bit at the bottom row of the deleted column is 0 then $A_0^{(m-1)}$ is the same as $\tilde{A}_0^{(m-1)}$, so $A_0^{(m-1)}$ is complete. However, $A_1^{(m-1)}$ is not the same as $\tilde{A}_1^{(m-1)}$: $\tilde{A}_1^{(m-1)}$ contains one more column that corresponds to the missing number. It follows that $A_1^{(m-1)}$ is almost complete. Alternatively, if the bit at the bottom row of the deleted column is 1 then $A_1^{(m-1)}$ is the same as $\tilde{A}_1^{(m-1)}$, so $A_1^{(m-1)}$ is complete, but $A_0^{(m-1)}$ is almost complete. That concludes the proof of the lemma. □

Having all that in mind, the verification of the algorithm is straightforward. `m` is the number of bits, *i.e.* the number of rows of A. The array `res` is the output: at each iteration of the main **for** loop, one bit of `res` is computed, the direction being from the least significant bit "upwards". `B` is an auxilliary array of integers. The definition of the problem requires bitwise access only to the elements of `A`; the array `B` can be accessed "normally". `B` keeps the indices of the columns whose `i`-th row we scan at every iteration of the main **for** loop. Initially, of course, `B` contains the indices `0`, `1`, ..., `n-1`, in that order, so when `i` is `m-1` we simply scan the bottom row of `A`. At every iteration of the main **for** loop, `ntemp` is the number of columns in the almost complete array whose last row we scan. Initially, `ntemp` is `n`, which reflects the fact that at the first iteration we scan the bottom row of `A`. We will verify the assignment of new value to `ntemp` later on.

Within the main **for** loop, the first nested **for** loop simply counts the zeros and ones and stores the results in `n0` and `n1`, respectively.

The difference `n0 - n1` determines the `i`-th least significant bit `res[i]` according to 5.1 and 5.2.

The second nested **for** loop discovers the indices of the columns that correspond to the columns of the next reduced array that is almost complete. To see why we consider only values (of the last row of `A`) equal to `res[i]`, check the above Lemma.

Finally, the asignment of new value to `ntemp` is done in accordance to the following considerations. If `ntemp` is even then both derived arrays have the same length `ntemp / 2`. Otherwise, note that we are interested in that derived subarray that is almost complete. If `res[i]` is one then that subarray is the one obtained by deleting the columns with zeros at the bottom; it has one more column than the complete derived subarray, so `res[i]` should be `ceil(ntemp / 2)`. Analogously, if `res[i]` is zero then `res[i]` should be `floor(ntemp / 2)`. That concludes the verification of the algorithm. The reader is invited to make an even more rigorous proof of correctness using loop invariant.

The number of accesses to `A` at each iteration of the main **for** loop is proportional to the current value of `ntemp`. Clearly, the total number of accesses is proportional to

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \leq 2n = \Theta(n)$$

□

Problem 114. *Consider Problem 113 under the additional assumption that the numbers in* A*, that is, the columns, appear in sorted order. Find the missing number with* $O(\lg n)$ *bit accesses to* A*.*

Solution:

If the numbers in A are sorted the problem can be solved by first determining the most significant bit, then the second most significant bit, *etc.*, the least significant bit, of the missing number, with precisely one access to A for each bit.

Suppose $\tilde{A}$ is the complete array of size n (see the definition of "complete array" in the solution to Problem 113), *i.e.* there is no missing number. For instance, if $n = 10$ then $\tilde{A}$ is:

$\tilde{A} =$

0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1	0	0	0
0	0	1	1	0	0	1	1	0	0	1
0	1	0	1	0	1	0	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

Consider the boundary on the top row between the zeros and the ones. The rightmost zero is in column 7 ($7 = 2^{\lfloor \log_2 10 \rfloor} - 1$) and the leftmost one is in column 8 ($8 = 2^{\lfloor \log_2 10 \rfloor}$). It is easy to generalise that the boundary is between columns $2^{\lfloor \log_2 n \rfloor} - 1$ and $2^{\lfloor \log_2 n \rfloor}$, provided the leftmost column is number 0.

Now consider the boundary on the top row between the zeros and the ones in an almost complete array A of size n. For instance, if $n = 10$ and the missing number is $6 = 0110_b$ then A is:

$A =$

0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	0	0	0
0	0	1	1	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Consider positions $7 = 2^{\lfloor \log_2 10 \rfloor} - 1$ and $8 = 2^{\lfloor \log_2 10 \rfloor}$ on the top row. Now the boundary is not between them because the missing number has most significant bit 0, so the boundary is "shifted" one position to the left in comparison with $\tilde{A}$. However, if we do not know what the missing number's most significant bit is, we can deduce it is 0 from the fact that there are two 1's at positions 7 and 8 on the top row.

Clearly, if there were 0 and 1 at positions 7 and 8, respectively, on the top row, that would mean the missing number's most significant bit is 1, as in the following example where the array is called B, $n = 10$ and the missing number is $9 = 1001_b$:

$B =$

0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	1	1	1	0	0
0	0	1	1	0	0	1	1	0	1
0	1	0	1	0	1	0	1	0	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

So, by inspecting only position $2^{\lfloor \log_2 n \rfloor} - 1$ on the top row, we can deduce the most significant bit of the missing number as follows:

$$\text{the missing number's most significant bit} = \begin{cases} 0, & \text{if } A[0][2^{\lfloor \log_2 n \rfloor} - 1] = 1 \\ 1, & \text{if } A[0][2^{\lfloor \log_2 n \rfloor} - 1] = 0 \end{cases}$$

Having computed the most significant bit of the missing number in $\Theta(1)$ time, we compute the second most significant bit in $\Theta(1)$ time, *etc.*, until we compute all bits of the missing number with $\Theta(\lg n)$ attempts, each taking $\Theta(1)$ time.

Case I: If the most significant bit is 0, to compute the second most significant bit we consider only the subarray of columns $0, 1, \ldots, 2^{\lfloor \log_2 n \rfloor} - 2$ and rows $1, 2, \ldots, \lfloor \log_2 n \rfloor$. Using the above A as an example, that subarray is, say, A':

$A =$

0	0	0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	0	0	0
0	0	1	1	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$\longmapsto$ $A' =$

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*

We proceed recursively with A' exactly as with A because A' is an almost complete array. Let us compute the size of A'. The initial A can be of any size but the size of A' is uniquely determined by n. Let $m = \lfloor \log_2 n \rfloor + 1$, that is, the number of bits necessary to represent n in binary (thus m is the number of rows in A). Let $m' = m - 1$. The derived A' is of size $2^{m'} - 1$. The -1 comes from the fact that A' has a missing number.

Case II: If the most significant bit is 1, to compute the second most significant bit we consider only the subarray whose columns have 1's in the top row. Using the above B as an example, that subarray is, say, B':

$$B = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&0&0&0&0&1&1\\\hline 0&0&0&0&1&1&1&1&0&0\\\hline 0&0&1&1&0&0&1&1&0&1\\\hline 0&1&0&1&0&1&0&1&0&0\\\hline\end{array}\begin{array}{cccccccccc} 0&1&2&3&4&5&6&7&8&9\end{array} \longmapsto B' = \begin{array}{|c|c|}\hline 0&0\\\hline 0&1\\\hline 0&0\\\hline\end{array}\begin{array}{cc} 0&1\end{array}$$

The number of columns, call it c', in B' is easy to compute: it is $c' = n - 2^{m'}$ where $m' = \lfloor \log_2 n \rfloor$. In the concrete example, $n = 10$, $m' = 3$, thus $c' = n - 2^{m'} = 10 - 8 = 2$. However, B' is not necessarily an almost complete array—in order to be an almost complete array it has to have at least one 1 at its top row. In fact, B' is an almost complete array only when $c' > \frac{1}{2}(2^{m'}) = 2^{m'-1}$. In this example, B' is not an almost complete array, it has one row too many. We can conlude the second most significant bit of the missing number is 0 (the missing number is $9 = 1001_b$) just by knowing the dimensions of B'; we do not have to scan, or even examine bits of, the top row to make that conclusion. The rule is, while $c' \leq 2^{m'-1}$, write 0's to into the missing number's bit positions and perform $m' \leftarrow m' - 1$. This process is equivalent to removing the necessary number of top rows from B'. Once the process is over and B' is reduced as necessary, it can be dealt with recursively.

Consider the following program in C.

```
int m = floor(logb(n)) + 1;
int A[m][n], res[m];
void find(int, int, int) ;

int main() {
  find(0, n-1, 0);
  return 0; }

void find(int low, int high, int row) {
  int j, c, n1, n2, numel = high - low + 1;
  if(row == m-1) {
    res[row] = !(A[i][0]);
    PrintResult();
    return; }
  n1 = floor(logb(numel));
  n2 = 1 << n1;
  if(A[row][low+n2-1] == 1) {
    res[row] = 0;
    find(low, n2-2, row+1); }
  else {
    j = 1;
    res[row] = 1;
    c = numel - n2;
    while((n1 >= 0) & & c <= (1<<(--n1))) {
      j ++;
      res[row+j] = 0; }
    if(row+j == m-1) {
      PrintResult();
```

```
        return; }
    find(n2, high, row+j); } }
```

The correctness of the fragment follows from the previous discussion. The time complexity is obviously $\Theta(\lg n)$. □

Problem 115. A circular array $A[1,\ldots,n]$ *is an array such that* $n \geq 3$ *and* $A[1]$ *and* $A[n]$ *are considered to be adjacent elements just like* $A[1]$ *and* $A[2]$, $A[2]$ *and* $A[3]$, etc., *are adjacent. We are given a circular array* $A[1,\ldots,n]$ *of nonnegative integers. For any* $i,j \in \{1,2,\ldots,n\}$ *such that* $i \neq j$, $dist(i,j) = \max\{|i-j|, n-|i-j|\}$. *Design a linear time algorithm that computes a maximum number* t *such that for some* $i,j \in \{1,2,\ldots,n\}, i \neq j$, $t = A[i] + A[j] + dist(i,j)$.

Solution:
Consider the following algorithm, due to Mugurel Ionuţ Andreica [MAMc].

CIRCULAR ARRAY($A[1,\ldots,n]$: circular array of nonnegative integers)
1 let $B[0\ldots n]$ and $C[1\ldots n]$ be linear arrays of nonnegative integers
2 $B[0] \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** n
4 $\quad B[i] \leftarrow \max\{B[i-1], A[i] - (i-1)\}$
5 $\quad C[i] \leftarrow B[i-1] + A[i] + (i-1)$
6 $x \leftarrow \max\{C[i] \mid 1 \leq i \leq n\}$
7 **for** $i \leftarrow 1$ **to** n
8 $\quad B[i] \leftarrow \max\{B[i-1], A[i] + (i-1)\}$
9 $C[n] \leftarrow A[n] + 1$
10 **for** $i \leftarrow n-1$ **downto** 2
11 $\quad C[i] \leftarrow \max\{C[i+1], A[i] + n - (i-1)\}$
12 $y \leftarrow \max\{B[i] + C[i+1] \mid 1 \leq i \leq n-1\}$
13 **return** $\max\{x, y\}$

It is obvious that the time complexity is $\Theta(n)$. Now we prove the correctness.

Lemma 20. *Whenever the execution of the first* ***for*** *loop (lines 3–5) of* CIRCULAR ARRAY *is at line 5 and* $i \geq 2$, $C[i]$ *is assigned* $\max\{A[k] + A[i] + i - k \mid 1 \leq k < i\}$.

Proof:
It is fairly obvious that at line 5 the value $B[i-1]$ is such that

$$B[i-1] = \begin{cases} 0, \text{ if } A[k] - (k-1) \leq 0 \ \ \forall k \text{ such that } 1 \leq k < i \\ \max\{A[k] - (k-1) \mid 1 \leq k < i\}, \text{ else} \end{cases}$$

However, $A[1] - (1-1)$ cannot be negative, therefore there is at least one non-negative value in the sequence $A[k] - (k-1), 1 \leq k < i$, so we can say simply that $B[i-1]$ at line 5 is $B[i-1] = \max\{A[k] - k + 1 \mid 1 \leq k < i\}$. It follows that $C[i]$ is assigned the value $\max\{A[k] - k + 1 \mid 1 \leq k < i\} + A[i] + i - 1 = \max\{A[k] + A[i] + i - k \mid 1 \leq k < i\}$. □

It follows that x is assigned the value $\max\{A[i] + A[j] + j - i \mid 1 \leq i < j \leq n\}$ at line 6 of CIRCULAR ARRAY.

Lemma 21. y *is assigned the value* $\max\{A[i] + A[j] + n - (j - i) \mid 1 \le i < j \le n\}$ *at line 12 of* Circular Array.

Proof:
Consider the second **for** loop (lines 7–8). Since $A[1] + (1 - 1) \ge 0$, it is the case that $\forall i, 1 \le i \le n, B[i] = \max\{A[k] + (k - 1) \mid 1 \le k \le i\}$ after the second **for** loop terminates. Now consider the third **for** loop at lines 10–11. Think of the assignment at line 9 as $C[n] = A[n] + n - (n - 1)$. Having that in mind, it is fairly obvious that after that **for** loop terminates, it is the case that $C[i] = \max\{A[k] + n - (k - 1) \mid i \le k \le n\}$, $\forall i, 2 \le i \le n$ From these two considerations it follows immediately that at line 12, y is assigned the value

$$\max\{A[i] + (i - 1) + A[j] + n - (j - 1) \mid 1 \le i < j \le n\} =$$
$$\max\{A[i] + A[j] + n - (j - i) \mid 1 \le i < j \le n\}$$

□

It follows immediately that Circular Array indeed returns the maximum number t such that for some $i, j \in \{1, 2, \ldots, n\}, i \ne j$, $t = A[i] + A[j] + \text{dist}(i, j)$. □

Problem 116 ([CLR00], Problem 10.3-8)**.** *Let* $X[1, \ldots, n]$ *and* $Y[1, \ldots, n]$ *be two arrays, each containing* n *numbers already in sorted order. Give an* $O(\lg n)$*-time algorithm to find the median of all* $2n$ *elements in arrays* X *and* Y*.*

Solution:
Assume that when n is even the median of X is $X\left[\frac{n}{2} + 1\right]$. If the arrays are of equal size, and that is the current case, we can solve the problem by a divide and conquer algorithm that compares the medians of the two arrays and then discards the lower half of the array with the smaller median and the upper half of the array with the bigger median. The algorithm proceeds likewise until both arrays are reduced to 2 elements each. Then we solve the reduced problem in constant time. In case the size is odd, by upper and lower half we mean, the subarray from one end until and excluding the median. It is easy to show this dichotomy brings the size of the array down to 2 regardless of what the initial n is, because the iterator $n \to \left\lceil \frac{n}{2} \right\rceil$ reaches 2 regardless of the starting value of n.

Now consider a more general version of this problem where the arrays are $X[1, \ldots, p]$ and $Y[1, \ldots, q]$ for possibly unequal values of p and q. The following solution is based on [LD05]. Let us call Z the array that would be obtained if we merged X and Y. Let $m = p + q$. The essence is the fact that we can check in $\Theta(1)$ time whether $X[i]$ is the median of Z, for any i such that $1 \le i \le p$. According to our definition of median, the median is greater than or equal to $\left\lfloor \frac{m}{2} \right\rfloor$ elements of an m-element array. Having that in mind, clearly if $X[i]$ is the median then:

- $X[i]$ is greater than or equal to $i - 1$ elements of X.
- $X[i]$ is greater than or equal to $j = \left\lfloor \frac{m}{2} \right\rfloor - i + 1$ elements of Y.

It takes only constant time to check if $Y[j] \le X[i] \le Y[j + 1]$[†] If that is fulfilled we have found the median and it is $X[i]$. Otherwise, we binary search in X to see if the median is in

[†]To avoid excessive boundary checks, pad X and Y at the left side with $-\infty$ and with ∞ at the right side.

X. If that fails, the median must be from Y, and we can repeat the analogous process with X and Y swapped.

```
COMMON MEDIAN(X[1,...,p], Y[1,...,q]: sorted arrays)
 1  m ← p + q
 2  k ← MEDIAN BIN SEARCH(X, Y, 1, p)
 3  if k > 0
 4      return X, k
 5  k ← MEDIAN BIN SEARCH(Y, X, 1, q)
 6  return Y, k
```

```
MEDIAN BIN SEARCH(A, B: sorted arrays, l, r: integers)
 1  if l > r
 2      return −1
 3  i ← ⌊(l+r)/2⌋
 4  j ← ⌊m/2⌋ − i + 1
 5  if B[j] ≤ A[i] ≤ B[j + 1]
 6      return i
 7  if A[i] < B[j]
 8      MEDIAN BIN SEARCH(A, B, l, i)
 9  if A[i] > B[j + 1]
10      MEDIAN BIN SEARCH(A, B, i + 1, r)
```

A not too formal proof of correctness of COMMON MEDIAN is simply pointing out the preceding discussion and knowing that the binary search idea is correct. The time complexity is obviously $\Theta(\lg m)$. Alternatively, we can say the complexity is $\Theta(\max\{\lg p, \lg q\})$. □

5.3 Graphs

Whenever we say "graph" without any qualifiers, we mean undirected graph without loops and without edge weights. Whenever we say "weighted graph" we mean that the edges have positive weights and the vertices, no weights. Unless otherwise specified, n is the number of vertices of the graph under consideration and m is the number of its edges. If G is a graph, we denote its vertex set by $V(G)$ and its edge set, by $E(G)$. $\text{adj}(u)$ denotes the adjacency list of vertex u.

To *delete* a vertex v from a graph $G(V, E)$ means to delete v from V and to delete all edges with one endpoint v from E. The vertex deletion operation is denoted by $G - v$. To *remove* an edge e from a graph $G(V, E)$ means to delete e from E without deleting its endpoints from V. The edge deletion operation is denoted by $G - e$. To *add* an edge $e' = (u, v)$ to G means that $(u, v) \notin E$ and then the operation $E \leftarrow E \cup \{(u, v)\}$ is performed. The edge addition operation is denoted by $G + e'$.

By "path" we mean a simple path, *i.e.* without repeating vertices. Likewise, by "cycle" we mean a simple cycle. The *degree* of a vertex u in undirected graph G is denoted by $\deg(u)$ and is defined as $\deg(u) = |\text{adj}(u)|$. If u is a vertex in multiple graphs, we write $\deg_G(u)$ to emphasise we mean the degree of u in G.

5.3.1 Graph traversal related algorithms

Definition 7. *Let* $G(V, E, w)$ *be a weighted connected graph. The* eccentricity *of any vertex* $v \in V$ *is* $ecc(v) = \max\{dist(v, u) \mid u \in V \setminus \{v\}\}$. *The* diameter *of* G *is* $diam(G) = \max\{ecc(v) \mid v \in V\}$. □

The term "diameter" is overloaded, meaning either the maximum eccentricity, or any path of such length whose endpoints are two vertices of maximum eccentricity. We encourage the reader to have in mind that diameter is completely different from longest path: the diameter is the longest one among the shortest paths between any two vertices in the graph, while the longest path is the longest one among the longest paths between any two vertices in the graph. As an extreme example, consider the complete graph K_n (with edge weights ones). Its diameter is 1 because every vertex is connected to every other vertex but its longest path is of length $n - 1$ because K_n is Hamiltonian. A notable exception are trees. In any tree, "diameter" and "longest path" are the same thing, *i.e.* paths of the same length. To see why, note that in trees there is a unique path between any two vertices. Therefore, the longest path between any two vertices u and v has the same length as the shortest path between u and v, because there is only one such path to begin with.

A *cut vertex* in a graph G with k connected components is any vertex $u \in V(G)$ such that the deletion of u leads to graph G' with $\geq k + 1$ connected components. A *bridge* in a graph G with k connected components is any edge $e \in V(G)$ such that the deletion of u leads to graph G' with $k + 1$ connected components.

Now we present the well known algorithm DFS for graph traversal, in the version of Cormen *et al.* [CLR00].

```
DFS(G(V, E): directed graph)
 1  foreach u ∈ V
 2      color[u] ← WHITE
 3      π[u] ← NIL
 4  time ← 0
 5  foreach u ∈ V
 6      if color[u] = WHITE
 7          DFS VISIT(G, u)
```

```
DFS VISIT(G(V, E): directed graph, u: vertex from V)
 1  color[u] ← GRAY
 2  time ← time + 1
 3  d[u] ← time
 4  foreach v ∈ adj(u)
 5      if color[v] = WHITE
 6          π[v] ← u
 7          DFS VISIT(G, v)
 8  color[u] ← BLACK
 9  time ← time + 1
10  f[u] ← time
```

It is well known that DFS on undirected graphs partitions the edges into *tree edges* and *back edges* according to the following rules: if the colour of v at line 5 is WHITE then (u, v) is a tree edge and if the colour of v at line 5 is GRAY then (u, v) is a back edge. Also, it is known it is not possible the said colour to be BLACK, so no other type of edge is possible in undirected graphs.

Problem 117. *Design a fast algorithm to compute the diameter of weighted tree. Analyse its correctness and time complexity.*

Solution:
We use a modified DFS as follows. The original DFS on the preceding page works on nonweighted graphs. Assume the input graph is weighted and connected. Let the algorithm use an additional array dist$[1, \ldots, n]$. Consider the following modification of DFS VISIT that does not use d[ ], f[ ], and the variable time.

```
ECCENTRICITY(T(V, E, w): weighted tree, u: vertex from V)
 1  (* Returns an ordered pair ⟨α, β⟩ where α = ecc(u) and *)
 2  (* β is a vertex at distance α from u. *)
 3  foreach x ∈ V
 4      color[x] ← WHITE
 5      π[x] ← NIL
 6      dist[x] ← 0
 7  ECC1(T, u)
 8  α ← max{dist[x] | x ∈ V}
 9  β ← any x ∈ V such that dist[x] = α
10  return ⟨α, β⟩
```

```
ECC1(T(V, E, w): weighted tree, u: vertex from V)
1  color[u] ← GRAY
2  foreach v ∈ adj(u)
3      if color[v] = WHITE
4          π[v] ← u
5          dist[v] ← dist[u] + w(u, v)
6          ECC1(T, v)
7  color[u] ← BLACK
```

It is trivial to prove by induction that after the call at line 7 in ECCENTRICITY, for every vertex $x \in V$, dist[x] contains the distances betwen x and u in T. Using the definition of ecc(u), conclude that ECCENTRICITY returns the eccentricity of u and a vertex that is at that distance from u. ECCENTRICITY has the time complexity of DFS and that is $\Theta(m+n)$.

Lemma 22. *Let* G *be a connected graph, weighted or non-weighted. Any two paths of maximum length in* G *share a vertex.*

Proof:
Assume there are two paths of maximum length that are independent. It is not difficult to show there is a path of even greater length, contrary to the assumption just made. We leave the details to the reader. □

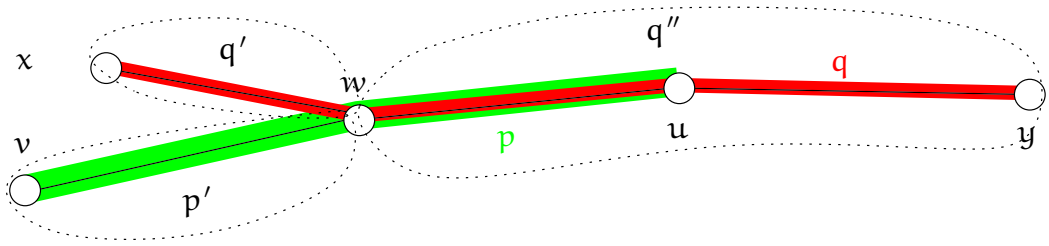


Figure 5.2: The paths p, q, p', and p'' under the assumption that u is an internal vertex of the backbone T'.

Corollary 2. *Let* T *be a tree, weighted or non-weighted. Any two diameters in* T *share a vertex.* □

Corollary 3. *Let* T *be a tree, weighted or non-weighted. Let* V' *be the union of the vertex sets of all diameters in* T. V' *induces a subtree* T' *of* T. *Every leaf of* T' *is a leaf of* T, *too.* □

We call the subtree T' from Corollary 3, *the backbone of T*. Note that T can coincide with its backbone, *e.g.* if T is a star.

Lemma 23. *Let* T *be a tree, weighted or non-weighted. Let* T' *be the backbone of* T. *For any vertex* $\mathsf{u} \in \mathsf{V}(\mathsf{T})$, *the eccentricity of* u *is the length of a path* p *such that one endpoint of* p *is* u *and the other endpoint, call it* v, *of* p *is some leaf of* T'.

Proof:
Assume the opposite. First assume u is a leaf of T'. By the definition of backbone, all vertices at maximum distance from u are endpoints of some diameters in T, *i.e.* they are leaves of T, hence the contradiction. Assume u is an internal vertex of T'. Then there is a diameter q of T such that u is an internal vertex in q. Having assumed that v is not a leaf of T', it follows v is not a vertex of q. But p and q share at least one vertex, namely u. Let the maximum contiguous subsequence (subpath) of p and q be $\mathsf{u}, \ldots, \mathsf{w}$. Let p' be the subpath $\mathsf{w}, \ldots, \mathsf{v}$ of p. Let q' be the subpath $\mathsf{w}, \ldots, \mathsf{x}$ of q such that x is am endpoint of q and u is not an internal vertex of q'. Let q'' be the subpath of q from w to y where y is the other endpoint of q (*i.e.*, not x). It must be case that $|\mathsf{p}'| > |\mathsf{q}'|$ according to the current assumptions. It follows that using p' and q'' we can construct a path in T that is longer than the diameter q, contrary to the fact that q is a longest path in T. See Figure 5.2.

Finally, assume u is not in T'. It is clear there exists a unique vertex z in T' such that, for every other vertex a from T', z is an internal vertex in the path between a and u. The proof reduces to the previous one with z instead of u. See Figure 5.3. □

Having in mind Lemma 23 and the correctness of algorithm Eccentricity, it is obvious that Eccentricity returns a vertex that is a leaf of the backbone of T. Now consider the following algorithm.

```
Diameter of Tree(T(V, E, w): weighted tree)
1  u ← arbitrary vertex from V
2  ⟨x, y⟩ ← Eccentricity(T, u)
3  ⟨w, z⟩ ← Eccentricity(T, y)
4  return ⟨w, z⟩
```

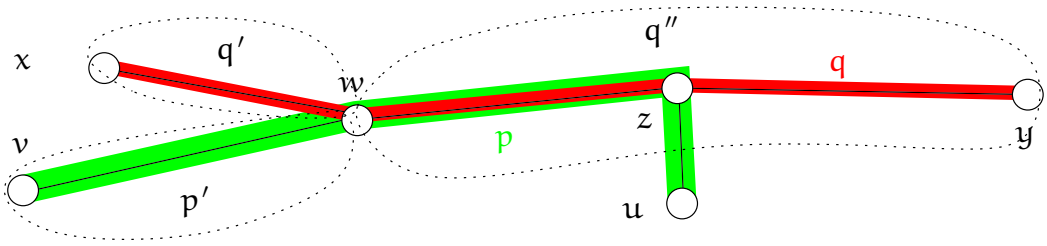


Figure 5.3: The case when u is not in the backbone T'.

Using tha names of DIAMETER OF TREE, y is a leaf of the backbone of T. Then the second call of ECCENTRICITY, namely ECCENTRICITY(T, y), returns $\langle w, z\rangle$ such that z is another leaf of the backbone, one that is at maximum distance from y, and w is the distance between them, *i.e.* the diameter. That proves the correctness of the algorithm. Clearly, the time complexity is $\Theta(n + m)$. □

Problem 118. *Design a fast algorithm to output all cut vertices of a connected graph. Analyse its correctness and time complexity.*

Solution:
The solution is based on the solution in [Ski08, Section 5.9.2, pp. 173–177]. The main idea is to modify DFS and thus to use its optimal time complexity.

Each execution of DFS on a connected graph $G(V, E)$ generates a tree T. $V(T) = V$ and $E(T) = \{e \in E \mid e$ is classified as a tree edge by DFS.$\}$. Clearly, the leaves of T cannot be cut vertices. To see why, consider any vertex $v \in V$ that is a leaf of T. If $\deg_G(v) = 1$ then the deletion of u from G leads to one connected component because such a vertex connects only itself to the remainder of the graph. If $\deg_G(v) \geq 2$, the tree edges are sufficient to hold the remainder of the graph together.

The root r of the tree may or may not be a cut vertex of G: if $\deg_T(r) = 1$ then r is not a cut vertex, otherwise it is a cut vertex.

Each of the remaining vertices may or may not be a cut vertex according to the following lemma.

Lemma 24. *Using the above names, for every vertex u that is not the root and is not a leaf of T, u is a cut vertex iff there exists a child v of u in T such that the subtree of T_v rooted at v is such that there is no back edge from any vertex from T_v to a proper ancestor of u.*

Proof:
Suppose there is no back edge from a vertex from $V(T_v)$ to a proper ancestor of u. Then for every path p with endpoints x and y such that $x \in V(T_v)$ and $y \in V(G) \setminus (V(T_v) \cup \{u\})$, u is an internal vertex in p. So, u is a cut vertex in G. Suppose the opposite: there is a back edge from a vertex from $V(T_v)$ to a proper ancestor of u. Then there exists a path p with endpoints x and y such that $x \in V(T_v)$ and $y \in V(G) \setminus (V(T_v) \cup \{u\})$, such that u is not in p. So, u is not a cut vertex in G. □

The algorithm that implements these ideas is a modifed DFS. The arrays $d[\,]$, $f[\,]$, and $\pi[\,]$, and the variable "time" are not used. There is, however, an array level$[1, \ldots, n]$ of natural

numbers that keeps record of the levels of the vertices in the DFS tree. That is, level[i] means the distance between vertex i and the root r. Obviously, the level of the root is 0.

```
FIND CUT VERTICES(G(V,E): undirected graph)
 1  foreach u ∈ V
 2      color[u] ← WHITE
 3  let u be an arbitrary vertex from V
 4  level[u] ← 0
 5  FCV1(G, u, 0);


FCV1(G(V,E): undirected graph, u: vertex from V, l: integer)
 1  color[u] ← GRAY
 2  level[u] ← l
 3  minback ← level[u]
 4  if level[u] = 0
 5      count ← 0
 6  IsCut ← FALSE
 7  foreach v ∈ adj(u)
 8      if color[v] = WHITE {
 9          if level[u] = 0
10              count ← count + 1
11          x ← FCV1(G, v, l + 1)
12          if x ≥ level[u] and level[u] ≥ 1
13              IsCut ← TRUE
14          minback ← min{minback, x} }
15      if color[v] = GRAY {
16          if level[v] < minback and level[v] ≠ level[u] − 1
17              minback ← level[v] }
18  if IsCut
19      print u
20  if level[u] = 0 and count ≥ 2
21      print u
22  color[u] ← BLACK
23  return minback
```

We argue that the printing of cut vertices (line 19 or line 21) is correct. First, recall that the root vertex is treated differently: the root, *i.e.* the starting vertex of the DFS, is a cut vertex iff there are at least two tree edges incident to it. The number of tree edges incident to the root is recorded in the `count` variable[†]. It is incremented at line 10 precisely when the current u is 0, *i.e.* u is the root, and v is WHITE, *i.e.* (u, v) is a tree egde. It follows that after the **for** loop (lines 7–17) finishes, `count` ≥ 2 iff the current u is the root of the DFS tree and it is indeed a cut vertex, and so the printing at line 21 is correct.

On the other hand, line 13 is reached iff

[†]We emphasise that the number of tree edges incident to that vertex is completely different from the degree of that vertex in G.

- vertex u is not the root, because level[u] ≥ 1 implies that, and
- there is no back edge from any vertex from T_v to a proper ancestor of u, because at line 11, x is assigned the number of the lowest level proper ancestor of v that is incident to a vertex from T_v; if that number is $\geq$ level[u] that vertex must be u or a vertex from T_v, so by Lemma 24, u is a cut vertex.

Those two conditions imply u is a cut vertex. Of course, in order to give a complete formal proof one has to prove by induction that FCV1 returns the number of the lowest level proper ancestor of u that is incident to a vertex from T_u. We leave that job to the inquisitive reader.

We point out that the variable minback at line 17 is set to level[v] only if level[v] $\neq$ level[u] $+ 1$ for the following reason. We know that DFS in the current implementation visits every edge of an undirected graph twice because that edge is in two different adjacency lists (one list for each endpoint). So, it makes sense to consider (u, v) at lines 16 and 17 as a back edge only when u is not the immediate ancestor of v. In other words, when level[v] $\neq$ level[u] $+ 1$. We also point out that the code at lines 16–17 is executed for each back edge (u, v).

That concludes the proof of the correctness of algorithm Find Cut Vertices. The time complexity is, obviously, the same as that of DFS: $\Theta(m + n)$. □

Problem 119. *Design a fast algorithm to output all bridges of a connected graph. Analyse its correctness and time complexity.*

Solution:
This problem is similar to the previous one and the solution is quite close to algorithm Find Cut Vertices. Again we use a modification of DFS and again we consider the partition of the edges into tree edges and back edges.

Lemma 25. *With respect to the work of DFS and the classification of edges into tree edges and back edges, any edge (u, v) is a bridge iff there exists a child v of u in T such that the subtree T_v rooted at v is such that there is no back edge from a vertex from T_v to u or a proper ancestor of u.*

Proof:
The claim is obvious, having in mind that there is no back edge from a vertex from T_v to u or a proper ancestor of u if and only if every path from a vertex from T_v to a vertex outside T_v must contain the edge (u, v). □

Unlike the problem of finding the cut vertices, now the root and the leaves of the DFS tree do not have to be treated differently from the other vertices.

```
Find Bridges(G(V, E): undirected graph)
 1  foreach u ∈ V
 2      color[u] ← White
 3  let u be an arbitrary vertex from V
 4  level[u] ← 0
 5  FBR1(G, u, 0);
```

```
FBR1(G(V,E): undirected graph, u: vertex from V, l: integer)
 1  color[u] ← GRAY
 2  level[u] ← l
 3  minback ← level[u]
 4  foreach v ∈ adj(u)
 5      if color[v] = WHITE {
 6          x ← FBR1(G, v, l + 1)
 7          if x > level[u]
 8              print (u, v)
 9          else if x < minback
10                  minback ← x }
11      else if level[v] < minback and level[v] ≠ level[u] − 1
12              minback ← level[v]
13  color[u] ← BLACK
14  return  minback
```

The proof of the correctness of algorithm FIND BRIDGES is simpler than that of FIND CUT VERTICES. The edge (u, v) is printed (line 8) iff the condition specified in Lemma 25 is fulfilled. Actually, the condition at line 7 is the main difference betweem this algorithm and FIND CUT VERTICES (see line 12 there). The time complexity is, obviously, the same as that of DFS: $\Theta(m + n)$. □

Chapter 6

Appendix

Problem 120.

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

Solution:
One way to solve it is to see that the sum is $\sum_{k=1}^{n} \lg k = \lg(n!)$ and then use Problem 1.48 on page 12. There is another way. Let $m = \lfloor \frac{n}{2} \rfloor$, $A = \sum_{k=1}^{m-1} \lg k$, and $B = \sum_{k=m}^{n} \lg k$. First we prove that $B \approx n \lg n$.

$$\sum_{k=m}^{n} \lg m \leq \sum_{k=m}^{n} \lg k \leq \sum_{k=m}^{n} \lg n \quad \Leftrightarrow$$

$$(\lg m) \sum_{k=m}^{n} 1 \leq B \leq (\lg n) \sum_{k=m}^{n} 1 \quad \Leftrightarrow$$

$$\underbrace{\left(\lg \left\lfloor \frac{n}{2} \right\rfloor\right)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{C} \leq B \leq \underbrace{(\lg n)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{D}$$

Clearly, $C \approx n \lg n$ and $D \approx n \lg n$. It must be the case that $B \approx n \lg n$. Now we prove that $A \preceq B$. A has $\frac{n-2}{2}$ terms, in case n is even, and $\frac{n-3}{2}$ terms, in case n is odd. In any event, A has less terms than B. Furthermore, every term of A is smaller than any term of B. It follows $A \preceq B$. Since $\sum_{k=1}^{n} \lg k = A + B$, it must be the case that $\sum_{k=1}^{n} \lg k \approx n \lg n$. □

Problem 121.

$$\sum_{k=1}^{n} k \lg k \approx n^2 \lg n$$

Solution:

Let $m = \lfloor \frac{n}{2} \rfloor$, $A = \sum_{k=1}^{m-1} k \lg k$, and $B = \sum_{k=m}^{n} k \lg k$. First we prove that $B \approx n^2 \lg n$.

$$\sum_{k=m}^{n} m \lg m \leq \sum_{k=m}^{n} k \lg k \leq \sum_{k=m}^{n} n \lg n \quad \Leftrightarrow$$

$$(m \lg m) \sum_{k=m}^{n} 1 \leq B \leq (n \lg n) \sum_{k=m}^{n} 1 \quad \Leftrightarrow$$

$$\underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor\right)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{C} \leq B \leq \underbrace{(n \lg n)\left(n - \left\lfloor \frac{n}{2} \right\rfloor + 1\right)}_{D}$$

Clearly, $C \approx n^2 \lg n$ and $D \approx n^2 \lg n$. It must be the case that $B \approx n^2 \lg n$. Now we prove that $A \preceq B$. A has $\frac{n-2}{2}$ terms, in case n is even, and $\frac{n-3}{2}$ terms, in case n is odd. In any event, A has less terms than B. Furthermore, every term of A is smaller than any term of B. It follows $A \preceq B$. Since $\sum_{k=1}^{n} k \lg k = A + B$, it must be the case that $\sum_{k=1}^{n} k \lg k \approx n \lg n$. □

Problem 122. *Find a closed formula for*

$$\sum_{k=0}^{n} 2^k k$$

Solution:
Let

$$S_n = \sum_{k=0}^{n} 2^k k$$

Then

$$S_n + (n+1)2^{n+1} = \sum_{k=0}^{n} 2^k k + (n+1)2^{n+1} = \sum_{k=0}^{n} 2^{k+1}(k+1) = 2\sum_{k=0}^{n} 2^k k + 2\sum_{k=0}^{n} 2^k$$

Since $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$,

$$S_n + (n+1)2^{n+1} = \underbrace{2\sum_{k=0}^{n} 2^k k}_{2S_n} + 2(2^{n+1} - 1) = 2S_n + 2.2^{n+1} - 2$$

Then

$$S_n = n2^{n+1} + 2^{n+1} - 2.2^{n+1} + 2 = n2^{n+1} - 2^{n+1} + 2$$

So,

$$S_n = (n-1)2^{n+1} + 2 \tag{6.1}$$

□

Problem 123. *Find a closed formula for*

$$\sum_{k=0}^{n} 2^k k^2$$

Solution:
Let

$$S_n = \sum_{k=0}^{n} 2^k k^2$$

Then

$$\begin{aligned} S_n + 2^{n+1}(n+1)^2 &= \sum_{k=0}^{n} 2^k k^2 + 2^{n+1}(n+1)^2 = \sum_{k=0}^{n} 2^{k+1}(k+1)^2 \\ &= 2\sum_{k=0}^{n} 2^k(k^2+2k+1) \\ &= \underbrace{2\sum_{k=0}^{n} 2^k k^2}_{2S_n} + \underbrace{4\sum_{k=0}^{n} 2^k k}_{4(n-1)2^{n+1}+8} + \underbrace{2\sum_{k=0}^{n} 2^k}_{2.2^{n+1}-2} \end{aligned}$$

Then

$$S_n + n^2 2^{n+1} + 2n2^{n+1} + 2.2^{n+1} = 2S_n + 4n2^{n+1} - 4.2^{n+1} + 8 + 2.2^{n+1} - 2$$

So,

$$S_n = n^2 2^{n+1} - 2n2^{n+1} + 4.2^{n+1} - 6 \tag{6.2}$$

□

Problem 124. *Find a closed formula for the sum of the first* n *odd numbers*

$$S_n = 1 + 3 + 5 + \ldots + 2n - 1$$

Solution:
It is trivial to prove by induction on n that $S_n = n^2$.
Basis: $S_1 = 1^2$.
Induction hypothesis: assume $S_n = n^2$.
Induction step:

$$\begin{aligned} S_{n+1} &= 1 + 3 + 5 + \ldots + 2n - 1 + 2n + 1 \\ &= S_n + 2n + 1 \qquad \text{by definition} \\ &= n^2 + 2n + 1 \qquad \text{by the induction hypothesis} \\ &= (n+1)^2 \end{aligned}$$

Indeed,

$$S_n = n^2 \tag{6.3}$$

There is a geometric proof of the same fact, illustrated on Figure 6.1. □

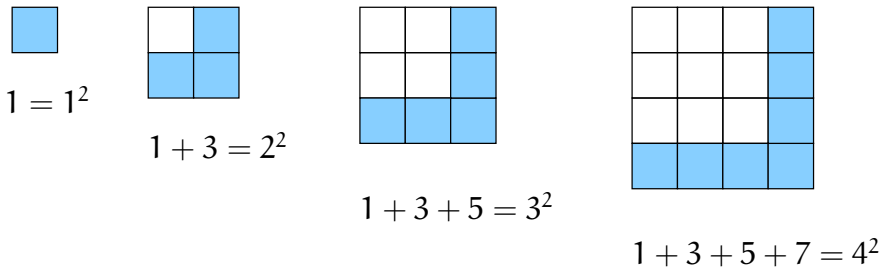


Figure 6.1: A geometric proof that the sum of the first n odd numbers is the n^{th} square n^2.

Problem 125. *Find a closed formula for*

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor$$

Solution:
To gain some intuition, let us write down the sum explicitly, *i.e.* all the terms, for some small n, say $n = 17$. For clarity put boxes around the terms whose positions are perfect squares, *i.e.* around the first, fourth, ninth, and sixtieth term.

$$\sum_{i=1}^{17} \left\lfloor \sqrt{i} \right\rfloor = \underbrace{\boxed{1} + 1 + 1}_{\text{run 1}} + \underbrace{\boxed{2} + 2 + 2 + 2 + 2}_{\text{run 2}} + \underbrace{\boxed{3} + 3 + 3 + 3 + 3 + 3 + 3}_{\text{run 3}} + \underbrace{\boxed{4} + 4}_{\text{run 4}}$$

The pattern is clear: the sum is the first n, in this case $n = 17$, terms of a series whose terms are the consecutive positive integers grouped in *runs*, run j being the sum of $2j + 1$ in number j's. Naturally, each run starts at a term whose position in the series is a perfect square: run 1 starts at position 1, run 2 starts at position 4, run 3 starts at position 9, *etc.* Problem 124 explains why the runs, except possibly for the last run, have lengths that are the consecutive odd numbers—since the first j odd numbers sum precisely to a perfect square, *viz.* j^2, it follows the difference between the two consecutive perfect squares $(j + 1)^2 - j^2$ is an odd number, *viz.* $2j + 1$.

The run with the largest number can be incomplete, as is the case when $n = 17$—run number 4 has only two terms. Let us call the number of complete runs, *i.e.* the ones that have all the terms, k_n. For instance, $k_{17} = 3$. We claim that

$$k_n = \lfloor \sqrt{n + 1} \rfloor - 1$$

To see why, imagine that n decreases one by one and think of the moment when k_n decreases. That is not when n becomes a perfect square minus one but when n becomes a perfect square minus two. For instance, $k_{15} = 3$ but $k_{14} = 2$. Hence we have $\sqrt{n + 1}$, not $\sqrt{n}$.

Having all that in mind we break the desired sum down into two sums:

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run. $S_2 = 0$ if and only if n is a perfect square minus one. More precisely, if we denote the number of

terms in S_2 by l_n,

$$l_n = n - \lfloor\sqrt{n+1}\rfloor^2 + 1$$

For instance, $l_{17} = 2$ as seen above and indeed $17 - \lfloor\sqrt{17+1}\rfloor^2 + 1 = 17 - 4^2 + 1 = 2$; $l_{15} = 0$ as seen above and indeed $15 - \lfloor\sqrt{15+1}\rfloor^2 + 1 = 15 - 4^2 + 1 = 0$.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.3 + 2.5 + 3.7 + 4.9 + 5.11 + \ldots + k(n)(2k(n)+1) \\
&= \sum_{i=1}^{k(n)} i(2i+1) \\
&= 2\sum_{i=1}^{k(n)} i^2 + \sum_{i=1}^{k(n)} i \\
&= 2\frac{k(n).(k(n)+1).(2k(n)+1)}{6} + \frac{k(n).(k(n)+1)}{2} \quad \text{by (6.21) and (6.22)} \\
&= k(n).(k(n)+1)\left(\frac{4k(n)+2}{6} + \frac{3}{6}\right) \\
&= \frac{1}{6}k(n).(k(n)+1).(4k(n)+5) \\
&= \frac{1}{6}(\lfloor\sqrt{n+1}\rfloor - 1)(\lfloor\sqrt{n+1}\rfloor - 1 + 1)(4\lfloor\sqrt{n+1}\rfloor - 4 + 5) \\
&= \frac{1}{6}(\lfloor\sqrt{n+1}\rfloor - 1)\lfloor\sqrt{n+1}\rfloor(4\lfloor\sqrt{n+1}\rfloor + 1)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{3}{2}}\right)$. S_2 is easier to compute, it has $l(n)$ terms, each term being $k(n)+1$.

$$\begin{aligned}
S_2 &= l_n(k_n + 1) \\
&= (n - \lfloor\sqrt{n+1}\rfloor^2 + 1)(\lfloor\sqrt{n+1}\rfloor - 1 + 1) \\
&= (n - \lfloor\sqrt{n+1}\rfloor^2 + 1)\lfloor\sqrt{n+1}\rfloor
\end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{3}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{3}{2}}\right) + O\left(n^{\frac{3}{2}}\right) = \Theta\left(n^{\frac{3}{2}}\right)$.

Let us denote $\lfloor\sqrt{n+1}\rfloor$ by $\widetilde{n}$. It follows that

$$\begin{aligned}
S_1 &= \frac{\widetilde{n}(\widetilde{n}-1)(4\widetilde{n}+1)}{6} \\
S_2 &= (n - \widetilde{n}^2 + 1)\widetilde{n}
\end{aligned}$$

$$\sum_{i=1}^{n} \left\lfloor\sqrt{i}\right\rfloor = \widetilde{n}\left(\frac{(\widetilde{n}-1)(4\widetilde{n}+1)}{6} + (n - \widetilde{n}^2 + 1)\right) \tag{6.4}$$

and

$$\sum_{i=1}^{n} \left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{3}{2}}\right) \tag{6.5}$$

□

Problem 126. *Find a closed formula for*

$$\sum_{i=1}^{n} \left\lceil \sqrt{i} \right\rceil$$

Solution:
Let us start with a small example as in Problem 125, say for $n = 17$. For clarity put boxes around the terms whose positions are perfect squares, *i.e.* around the first, fourth, ninth, and sixtieth term.

$$\sum_{i=1}^{17} \left\lceil \sqrt{i} \right\rceil = \underbrace{\boxed{1}}_{\text{run 1}} + \underbrace{2+2+\boxed{2}}_{\text{run 2}} + \underbrace{3+3+3+3+\boxed{3}}_{\text{run 3}} + \underbrace{4+4+4+4+4+4+\boxed{4}}_{\text{run 4}} + \underbrace{5}_{\text{run5}}$$

The pattern is quite similar to the one in Problem 125. We sum the first n terms of a series whose terms are the consecutive positive integers grouped in runs, run j being the sum of $2j-1$ in number j's.

The run with the largest number can be incomplete. For instance, if $n = 17$ then run number 5 has only one term. Let us call the number of complete runs, *i.e.* the ones that have all the terms, s_n. For instance, $s_{17} = 4$. It is obvious that

$$s_n = \lfloor \sqrt{n} \rfloor$$

We break the desired sum down into two sums:

$$\sum_{i=1}^{n} \left\lfloor \sqrt{i} \right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run. $S_2 = 0$ if and only if n is a perfect square. We denote the number of terms in S_2 by t_n.

$$t_n = n - \lfloor \sqrt{n} \rfloor^2$$

For instance, $t_{17} = 1$ as seen above and indeed $17 - \lfloor \sqrt{17} \rfloor^2 = 17 - 4^2 = 1$; $t_{16} = 0$ as seen above and indeed $16 - \lfloor \sqrt{16} \rfloor^2 = 16 - 4^2 = 0$.

Let us compute S_1.

$$\begin{aligned}
S_1 &= 1.1 + 2.3 + 3.5 + 4.7 + 5.9 + \ldots + s_n(2s_n - 1) \\
&= \sum_{i=1}^{s_n} i(2i-1) \\
&= 2\sum_{i=1}^{s_n} i^2 - \sum_{i=1}^{s_n} i \\
&= 2\frac{s_n.(s_n+1).(2s_n+1)}{6} - \frac{s_n.(s_n+1)}{2} \quad \text{by (6.21) and (6.22)} \\
&= s_n.(s_n+1)\left(\frac{4s_n+2}{6} - \frac{3}{6}\right) \\
&= \frac{1}{6}s_n.(s_n+1).(4s_n-1) \\
&= \frac{1}{6}(\lfloor \sqrt{n} \rfloor)(\lfloor \sqrt{n} \rfloor + 1)(4\lfloor \sqrt{n} \rfloor - 1)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{3}{2}}\right)$. Now we compute S_2. It has t_n terms, each term being $s_n + 1$.

$$\begin{aligned} S_2 &= t_n.(s_n + 1) \\ &= (n - \lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor + 1) \end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{3}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{3}{2}}\right) + O\left(n^{\frac{3}{2}}\right) = \Theta\left(n^{\frac{3}{2}}\right)$.

It follows that

$$\sum_{i=1}^{n} \left\lceil\sqrt{i}\right\rceil = (\lfloor\sqrt{n}\rfloor + 1)\left(\frac{\lfloor\sqrt{n}\rfloor(4\lfloor\sqrt{n}\rfloor - 1)}{6} + n - \lfloor\sqrt{n}\rfloor^2\right) \tag{6.6}$$

and

$$\sum_{i=1}^{n} \left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{3}{2}}\right) \tag{6.7}$$

□

Problem 127. *Find a closed formula for*

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor$$

Solution:
The line of reasoning is very similar to the one in Problem 125. We sum the first n terms of a series, the series being the one mentioned in the solution of Problem 125 with each term multiplied by its position. Consider for example $n = 17$. The terms whose positions are perfect squares are boxed.

$$\sum_{i=1}^{17} i\left\lfloor\sqrt{i}\right\rfloor = \underbrace{\boxed{1} + 2 + 3}_{\text{run 1}} + \underbrace{\boxed{8} + 10 + 12 + 14 + 16}_{\text{run 2}} + \underbrace{\boxed{27} + 30 + 33 + 36 + 39 + 42 + 45}_{\text{run 3}} + \underbrace{\boxed{64} + 68}_{\text{run 4}}$$

Unlike Problem 125, now the runs consist of those consecutive terms whose differences are equal (and equal to the number of the run). Just as in Problem 125, all the runs but the last one are complete, the last run being either complete or incomplete. We denote the number of the complete runs with k_n and the number of terms in the incomplete run by l_n. It is the case that

$$\begin{aligned} k_n &= \lfloor\sqrt{n+1}\rfloor - 1 \\ l_n &= n - \lfloor\sqrt{n+1}\rfloor^2 + 1 \end{aligned}$$

the reasoning being exactly the same as in Problem 125. We break the desired sum down into two sums:

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.(1+2+3) + 2.(4+5+6+7+8) + 3.(9+10+11+12+13+14+15) \\
&\quad + 4.(16+17+18+19+20+21+22+23+24) \\
&\quad + 5.(25+26+27+28+29+30+31+32+33+34+35) \\
&\quad + \ldots \\
&\quad + k_n\big(k_n^2 + (k_n^2+1) + (k_n^2+2) + \ldots + \underbrace{((k_n+1)^2-1)}_{k_n^2+2k_n}\big) \\
&= \sum_{i=1}^{k_n} i \sum_{j=i^2}^{i^2+2i} j \\
&= \sum_{i=1}^{k_n} i \left(\sum_{j=1}^{i^2+2i} j - \sum_{j=1}^{i^2-1} j \right) \\
&= \sum_{i=1}^{k_n} i \left(\frac{(i^2+2i)(i^2+2i+1)}{2} - \frac{(i^2-1)i^2}{2} \right) \\
&= \frac{1}{2} \sum_{i=1}^{k_n} i \left(i^4 + 2i^3 + i^2 + 2i^3 + 4i^2 + 2i - i^4 + i^2 \right) \\
&= \frac{1}{2} \sum_{i=1}^{k_n} i \left(4i^3 + 6i^2 + 2i \right) \\
&= 2\sum_{i=1}^{k_n} i^4 + 3\sum_{i=1}^{k_n} i^3 + \sum_{i=1}^{k_n} i^2 \quad \text{apply (6.22), (6.23), and (6.24)} \\
&= 2\frac{k_n(k_n+1)(2k_n+1)(3k_n^2+3k_n-1)}{30} + 3\frac{k_n^2(k_n+1)^2}{4} + \frac{k_n(k_n+1)(2k_n+1)}{6} \\
&= \frac{k_n(k_n+1)}{2} \left(\frac{(4k_n+2)(3k_n^2+3k_n-1)}{15} + \frac{3k_n(k_n+1)}{2} + \frac{2k_n+1}{3} \right) \\
&= \frac{k_n(k_n+1)}{60} \left((8k_n+4)(3k_n^2+3k_n-1) + 45k_n(k_n+1) + 20k_n + 10 \right) \\
&= \frac{k_n(k_n+1)}{60} \left(24k_n^3 + 24k_n^2 - 8k_n + 12k_n^2 + 12k_n - 4 + 45k_n^2 + 45k_n + 20k_n + 10 \right) \\
&= \frac{k_n(k_n+1)}{60} \left(24k_n^3 + 81k_n^2 + 69k_n + 6 \right) \\
&= \frac{k_n(k_n+1)(8k_n^3+27k_n^2+23k_n+2)}{20} \qquad (6.8)
\end{aligned}$$

Substitute k_n with $\lfloor\sqrt{n+1}\rfloor - 1$ in (6.8) to obtain

$$\begin{aligned}
S_1 &= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8(\lfloor\sqrt{n+1}\rfloor - 1)^3 + \\
&\qquad 27(\lfloor\sqrt{n+1}\rfloor - 1)^2 + 23(\lfloor\sqrt{n+1}\rfloor - 1) + 2\big) \\
&= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8\lfloor\sqrt{n+1}\rfloor^3 - 24\lfloor\sqrt{n+1}\rfloor^2 + 24\lfloor\sqrt{n+1}\rfloor - 8 \\
&\qquad 27\lfloor\sqrt{n+1}\rfloor^2 - 54\lfloor\sqrt{n+1}\rfloor + 27 + 23\lfloor\sqrt{n+1}\rfloor - 23 + 2\big) \\
&= \frac{1}{20}\lfloor\sqrt{n+1}\rfloor(\lfloor\sqrt{n+1}\rfloor - 1)\big(8\lfloor\sqrt{n+1}\rfloor^3 + 3\lfloor\sqrt{n+1}\rfloor^2 - 7\lfloor\sqrt{n+1}\rfloor - 2\big)
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{5}{2}}\right)$. Now we compute S_2. It has l_n terms, the first term is $(k_n+1)^3$, and the difference between every two consecutive terms is (k_n+1).

$$\begin{aligned}
S_2 &= \sum_{i=1}^{l_n}(k_n+1)^3 + (i-1)(k_n+1) \\
&= (k_n+1)^3\sum_{i=1}^{l_n} 1 + (k_n+1)\sum_{i=1}^{l_n}(i-1) \\
&= (k_n+1)^3 l_n + \frac{(k_n+1)(l_n-1)l_n}{2} \\
&= \lfloor\sqrt{n+1}\rfloor^3(n - \lfloor\sqrt{n+1}\rfloor^2 + 1) + \frac{\lfloor\sqrt{n+1}\rfloor(n - \lfloor\sqrt{n+1}\rfloor^2)(n - \lfloor\sqrt{n+1}\rfloor^2 + 1)}{2}
\end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{5}{2}}\right)$, therefore $S_1 + S_2 = \Theta\left(n^{\frac{5}{2}}\right) + O\left(n^{\frac{5}{2}}\right) = \Theta\left(n^{\frac{5}{2}}\right)$.

Let us denote $\lfloor\sqrt{n+1}\rfloor$ by $\widetilde{n}$. It follows that

$$\begin{aligned}
S_1 &= \frac{\widetilde{n}(\widetilde{n}-1)(8\widetilde{n}^3 + 3\widetilde{n}^2 - 7\widetilde{n} - 2)}{20} \\
S_2 &= \widetilde{n}^3(n - \widetilde{n}^2 + 1) + \frac{\widetilde{n}(n-\widetilde{n}^2)(n-\widetilde{n}^2+1)}{2}
\end{aligned}$$

and

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \frac{\widetilde{n}(\widetilde{n}-1)(8\widetilde{n}^3 + 3\widetilde{n}^2 - 7\widetilde{n} - 2)}{20} + \widetilde{n}^3(n - \widetilde{n}^2 + 1) + \frac{\widetilde{n}(n-\widetilde{n}^2)(n-\widetilde{n}^2+1)}{2} \tag{6.9}$$

and

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \Theta\left(n^{\frac{5}{2}}\right) \tag{6.10}$$

□

Problem 128. *Find a closed formula for*

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil$$

Solution:
The solution of this problem is quite similar to the solution of Problem 126. We sum the first n terms of a series, the series being the one mentioned in the solution of Problem 126 with each term multiplied by its position. Consider for example $n = 17$. The terms whose positions are perfect squares are boxed.

$$\sum_{i=1}^{17} i\left\lceil\sqrt{i}\right\rceil = \underbrace{\boxed{1}}_{\text{run 1}} + \underbrace{4+6+\boxed{8}}_{\text{run 2}} + \underbrace{15+18+21+24+\boxed{27}}_{\text{run 3}} + \underbrace{40+44+48+52+56+60+\boxed{64}}_{\text{run 4}} + \underbrace{85}_{\text{run5}}$$

Unlike Problem 126, now the runs consist of those consecutive terms whose differences are equal (and equal to the number of the run). Just as in Problem 126, all the runs but the last one are complete, the last run being either complete or incomplete. We denote the number of the complete runs with $s(n)$ and

$$s(n) = \lfloor\sqrt{n}\rfloor$$

the reasoning being exactly the same as in Problem 126. The number of terms in the incomplete run is

$$t(n) = n - \lfloor\sqrt{n}\rfloor^2$$

We break the desired sum down into two sums:

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil = S_1 + S_2$$

where S_1 is the sum of the terms of the complete runs and S_2, of the incomplete run.

Let us first compute S_1.

$$\begin{aligned}
S_1 &= 1.1 + 2.(2+3+4) + 3.(5+6+7+8+9) \\
&\quad + 4.(10+11+12+13+14+15+16) \\
&\quad + 5.(17+18+19+20+21+22+23+24+25) \\
&\quad + \ldots \\
&\quad + s_n\big(((s_n-1)^2+1) + ((s_n-1)^2+2) + \ldots + \underbrace{s_n^2}_{(s_n-1)^2+2s_n-1}\big) \\
&= \sum_{i=1}^{s_n} i \sum_{j=1}^{2i-1} (i-1)^2 + j \qquad (6.11) \\
&= \sum_{i=1}^{s_n} i \left(\sum_{j=1}^{2i-1} (i-1)^2 + \sum_{j=1}^{2i-1} j \right) \\
&= \sum_{i=1}^{s_n} i \left((i-1)^2(2i-1) + \frac{(2i-1)2i}{2} \right) \\
&= \sum_{i=1}^{s_n} i \left((i^2-2i+1)(2i-1) + 2i^2 - i \right) \\
&= \sum_{i=1}^{s_n} i(2i^3 - i^2 - 4i^2 + 2i + 2i - 1 + 2i^2 - i) \\
&= \sum_{i=1}^{s_n} i(2i^3 - 3i^2 + 3i - 1) \\
&= 2\sum_{i=1}^{s_n} i^4 - 3\sum_{i=1}^{s_n} i^3 + 3\sum_{i=1}^{s_n} i^2 - \sum_{i=1}^{s_n} i \quad \text{apply (6.21), (6.22), (6.23), and (6.24)} \\
&= 2\frac{s_n(s_n+1)(2s_n+1)(3s_n^2+3s_n-1)}{30} - 3\frac{s_n^2(s_n+1)^2}{4} + \\
&\qquad 3\frac{s_n(s_n+1)(2s_n+1)}{6} - \frac{s_n(s_n+1)}{2} \\
&= \frac{s_n(s_n+1)}{2}\left(\frac{2(2s_n+1)(3s_n^2+3s_n-1)}{15} - \frac{3s_n(s_n+1)}{2} + \frac{6s_n+3}{3} - 1 \right) \qquad (6.12)
\end{aligned}$$

Simplify (6.12) to obtain

$$\begin{aligned}
&\frac{s_n(s_n+1)}{2}\left(\frac{12s_n^3+12s_n^2-4s_n+6s_n^2+6s_n-2}{15} - \frac{3s_n^2+3s_n}{2} + \frac{6s_n+3}{3} - 1 \right) = \\
&\frac{s_n(s_n+1)}{2}\left(\frac{24s_n^3+36s_n^2+4s_n-4}{30} - \frac{45s_n^2+45s_n}{30} + \frac{60s_n+30}{30} - \frac{30}{30} \right) = \\
&\frac{s_n(s_n+1)}{60}(24s_n^3+36s_n^2+4s_n-4-45s_n^2-45s_n+60s_n+30-30) = \\
&\frac{s_n(s_n+1)(24s_n^3-9s_n^2+19s_n-4)}{60} = \\
&\frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+1)(24\lfloor\sqrt{n}\rfloor^3-9\lfloor\sqrt{n}\rfloor^2+19\lfloor\sqrt{n}\rfloor-4)}{60}
\end{aligned}$$

Clearly, $S_1 = \Theta\left(n^{\frac{5}{2}}\right)$. Now we compute S_2. It has t_n terms, the first term is $(s_n^2+1)(s_n+1)$, and the difference between every two consecutive terms is (s_n+1).

$$\begin{aligned}
S_2 &= \sum_{i=1}^{t_n}(s_n^2+1)(s_n+1)+(i-1)(s_n+1) = \\
&= (s_n^2+1)(s_n+1)\sum_{i=1}^{t_n}1+(s_n+1)\sum_{i=1}^{t_n}(i-1) \\
&= t_n(s_n^2+1)(s_n+1)+\frac{(s_n+1)(t_n-1)t_n}{2} = \\
&= \frac{t_n(s_n+1)}{2}\left(2s_n^2+2+t_n-1\right) = \\
&= \frac{t_n(s_n+1)(2s_n^2+t_n+1)}{2} \\
&= \frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(2\lfloor\sqrt{n}\rfloor^2+n-\lfloor\sqrt{n}\rfloor^2+1)}{2} \\
&= \frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(n+\lfloor\sqrt{n}\rfloor^2+1)}{2}
\end{aligned}$$

Clearly, $S_2 = O\left(n^{\frac{5}{2}}\right)$, therefore $S_1+S_2 = \Theta\left(n^{\frac{5}{2}}\right)+O\left(n^{\frac{5}{2}}\right) = \Theta\left(n^{\frac{5}{2}}\right)$. It follows that

$$\sum_{i=1}^{n} i\left\lfloor\sqrt{i}\right\rfloor = \frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+1)(24\lfloor\sqrt{n}\rfloor^3-9\lfloor\sqrt{n}\rfloor^2+19\lfloor\sqrt{n}\rfloor-4)}{60}+ \frac{(n-\lfloor\sqrt{n}\rfloor^2)(\lfloor\sqrt{n}\rfloor+1)(n+\lfloor\sqrt{n}\rfloor^2+1)}{2} \tag{6.13}$$

and

$$\sum_{i=1}^{n} i\left\lceil\sqrt{i}\right\rceil = \Theta\left(n^{\frac{5}{2}}\right) \tag{6.14}$$

□

Fact: The Fibonacci numbers are the natural numbers defined by the recurrence relation

$$\begin{aligned}
&F_0 = 0 \\
&F_1 = 1 \\
&F_n = F_{n-1}+F_{n-2}, \text{ for all } n > 1
\end{aligned}$$

The first several elements of the sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The asymptotic growth rate of F_n is determined by the following equality [GKP94, pp. 300]

$$F_n = \left\lfloor\frac{\phi^n}{\sqrt{5}}+\frac{1}{2}\right\rfloor = \frac{\phi^n}{\sqrt{5}}, \text{ rounded to the nearest integer}$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the so called "golden ratio", the positive root of $\phi^2 = \phi + 1$. Clearly, for any positive constant c,

$$c^{F_n} = \Theta\left(c^{\frac{\phi^n}{\sqrt{5}}}\right) = \Theta\left(k^{\phi^n}\right), \text{ where } k = c^{\frac{1}{\sqrt{5}}} \tag{6.15}$$

□

Fact: The harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots = \sum_{i=1}^{\infty} \frac{1}{i}$$

is divergent. Its n^{th} partial sum is denoted by H_n.

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} \tag{6.16}$$

It is known that

$$H_n = \Theta(\lg n) \tag{6.17}$$

Furthermore, $\ln n < H_n < \ln n + 1$ for $n > 1$. For details, see [GKP94, pp. 272–278]. □

Fact: The sum of the first k^{th} powers for some integer constant $k \geq 1$ is

$$1^k + 2^k + \ldots + n^k = \sum_{i=0}^{n} i^k \tag{6.18}$$

It is well known that

$$\sum_{i=0}^{n} i^k = \frac{1}{k+1} \sum_{j=0}^{k} \binom{k+1}{j} B_j (n+1)^{k+1-j} \tag{6.19}$$

where B_j is the j^{th} *Bernoulli number*. The Bernolli numbers are defined with the recurrence

$$B_0 = 1$$
$$B_m = -\frac{1}{m} \sum_{j=0}^{m-1} \binom{m+1}{j} B_j, \text{ for } m \in \mathbb{N}^+$$

For details on the summation formula (6.19) and plenty of information on the Bernoulli numbers, see [GKP94, pp. 283–290]. Just keep in mind that Knuth *et al.* denote the sum by $S_k(n)$ and define it as

$$S_k(n) = 0^k + 1^k + 2^k + \ldots + (n-1)^k$$

For our purposes in this manual it is sufficient to know that

$$1^k + 2^k + \ldots + n^k = \Theta(n^{k+1}) \tag{6.20}$$

which fact follows easily from (6.19). In fact, (6.19) is a polynomial of degree $k+1$ of n because the $\binom{k+1}{j}$ factor and the Bernoulli numbers are just constants and clearly the highest degree of n is $k+1$. Strictly speaking, we have not proved here formally that (6.19) is a degree $k+1$ polynomial of n because we have not shown that the coefficient before n^{k+1} is not zero. But that is indeed the case—see for instance [GKP94, (6.98), pp. 288].

⁉ NB ⁉ *Be careful to avoid the error of thinking that*

$$1^k + 2^k + \ldots + n^k$$

is a degree k *polynomial of* n *and thus erroneosly concluding that its order of growth is* $\Theta(n^k)$. *It is* not *a polynomial of* n *because a polynomial has an* a priori *fixed number of terms, while the above sum has* n *terms where* n *is the variable.*

Using (6.19), we can easily derive

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} \tag{6.21}$$

$$1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6} \tag{6.22}$$

$$1^3 + 2^3 + \ldots + n^3 = \frac{n^2(n+1)^2}{4} \tag{6.23}$$

$$1^4 + 2^4 + \ldots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \tag{6.24}$$

□

Lemma 26.

$$\sum_{k=1}^{n} k(k+1) = \frac{p(p+1)(p+2)}{3}$$

Proof:

$$\sum_{k=1}^{n} k(k+1) = 1 \times (1+1) + 2 \times (2+1) + \ldots + n(n+1) =$$

$$1 \times 1 + 1 + 2 \times 2 + 2 + \ldots n \times n + n =$$

$$1 \times 1 + 2 \times 2 + \ldots + n \times n + 1 + 2 + \ldots + n = \quad \text{by (6.22) and (6.23)}$$

$$\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} =$$

$$\frac{n(n+1)}{2}\left(\frac{2n+1}{3} + 1\right) =$$

$$\frac{n(n+1)(n+2)}{3}$$

□

Problem 129. *Let* T *be a binary heap of height* h *vertices. Find the minimum and maixmum number of vertices in* T.

Solution:
The vertices of any binary tree are partitioned into *levels*, the vertices from level number i being the ones that are at distance i from the root. By definition, every level i in T, except possibly for level h, is complete in the sense it has all the 2^i vertices possible. The last level (number h) can have anywhere between 1 and 2^h vertices inclusive. If n denotes the number of vertices in the heap, it is the case that

$$\underbrace{2^0+2^1+2^2+\ldots+2^{h-1}}_{\text{the number of vertices in the complete levels}}+1 \le n \le \underbrace{2^0+2^1+2^2+\ldots+2^{h-1}}_{\text{the number of vertices in the complete levels}}+2^h$$

Since $2^0+2^1+2^2+\ldots+2^{h-1}=\frac{2^h-1}{2-1}=2^h-1$, it follows that

$$\begin{aligned} 2^h-1+1 \le n &\le 2^h-1+2^h \\ 2^h \le n &\le 2^{h+1}-1 \end{aligned} \tag{6.25}$$

□

Problem 130. *Let T be a binary heap with n vertices. Find the height h of T.*

Solution:

$$\begin{aligned} &2^h \le n \le 2^{h+1}-1 \quad \text{see Problem 129, (6.25)} \\ &2^h \le n < 2^{h+1} \\ &h \le \lg n < h+1 \quad \text{take lg of both sides} \end{aligned}$$

Clearly,

$$h=\lfloor \lg n \rfloor \tag{6.26}$$

□

Problem 131. *Let T be a binary heap with n vertices. Prove that the number of leaves of T is $\lceil \frac{n}{2} \rceil$ and the number of internal vertices in T is $\lfloor \frac{n}{2} \rfloor$.*

Solution:
Let h be the height of T. We know (6.26) that $h=\lfloor \lg n \rfloor$. Let V' be the vertices of T at level h. Let T'' be obtained from T by deleting V' (see Figure 6.2). Clearly, T'' is a complete binary tree of height $h-1=\lfloor \lg n \rfloor -1$. The number of its vertices is

$$2^{\lfloor \lg n \rfloor -1+1}-1=2^{\lfloor \lg n \rfloor}-1 \tag{6.27}$$

It follows

$$|V'|=n-(2^{\lfloor \lg n \rfloor}-1)=n+1-2^{\lfloor \lg n \rfloor} \tag{6.28}$$

The vertices at level $h-1$ are $2^{h-1}=2^{\lfloor \lg n \rfloor -1}$. Those vertices are partitioned into V'', the vertices that have no children, and V''', the vertices that have a child or two children (see Figure 6.2). So,

$$|V''|+|V'''|=2^{\lfloor \lg n \rfloor -1} \tag{6.29}$$

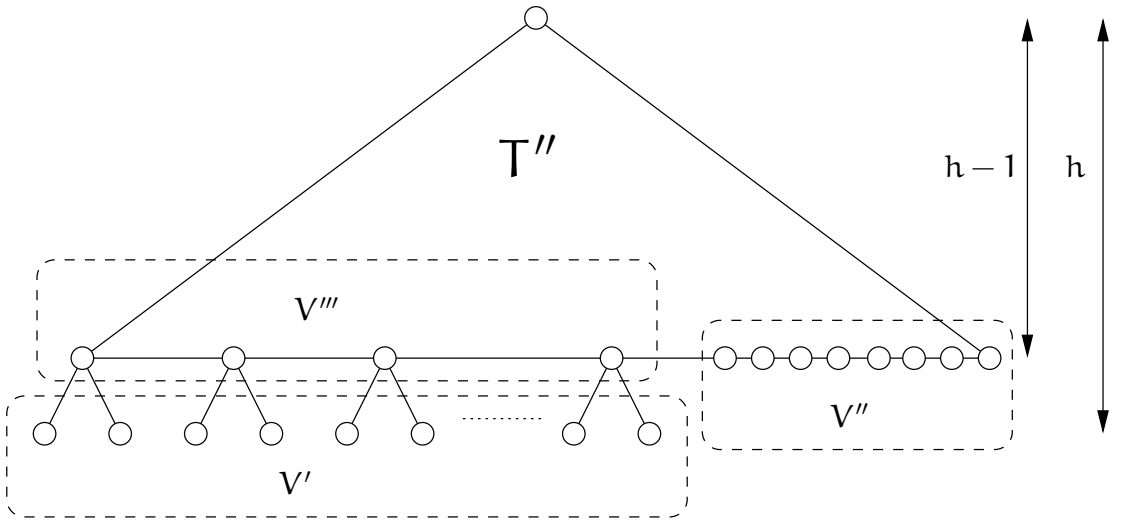


Figure 6.2: The heap in Problem 131.

Note that $|V'''| = \left\lceil \frac{|V'|}{2} \right\rceil$. Having in mind (6.28), it follows that

$$|V'''| = \left\lceil \frac{n+1-2^{\lfloor \lg n \rfloor}}{2} \right\rceil = \left\lceil \frac{n+1}{2} - \frac{2^{\lfloor \lg n \rfloor}}{2} \right\rceil = \left\lceil \frac{n+1}{2} - 2^{\lfloor \lg n \rfloor - 1} \right\rceil =$$
$$\left\lceil \frac{n+1}{2} \right\rceil - 2^{\lfloor \lg n \rfloor - 1} \qquad \text{since } 2^{\lfloor \lg n \rfloor - 1} \text{ is integer} \tag{6.30}$$

Use (6.29) and (6.30) to conclude that

$$\begin{aligned} |V''| &= 2^{\lfloor \lg n \rfloor - 1} - \left(\left\lceil \frac{n+1}{2} \right\rceil - 2^{\lfloor \lg n \rfloor - 1} \right) \\ &= 2^{\lfloor \lg n \rfloor - 1} - \left\lceil \frac{n+1}{2} \right\rceil + 2^{\lfloor \lg n \rfloor - 1} \\ &= 2.2^{\lfloor \lg n \rfloor - 1} - \left\lceil \frac{n+1}{2} \right\rceil \\ &= 2^{\lfloor \lg n \rfloor} - \left\lceil \frac{n+1}{2} \right\rceil \end{aligned} \tag{6.31}$$

It is obvious the leaves of T are $V' \cup V''$. Use (6.28) and (6.31) to conclude that

$$\begin{aligned} |V'| + |V''| &= n + 1 - 2^{\lfloor \lg n \rfloor} + 2^{\lfloor \lg n \rfloor} - \left\lceil \frac{n+1}{2} \right\rceil \\ &= n + 1 - \left\lceil \frac{n+1}{2} \right\rceil = n + 1 + \left\lfloor -\frac{n+1}{2} \right\rfloor \\ &= \left\lfloor n + 1 - \frac{n+1}{2} \right\rfloor \qquad \text{since } n+1 \text{ is integer} \\ &= \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil \end{aligned} \tag{6.32}$$

Then the internal vertices of T must be $\left\lfloor \frac{n}{2} \right\rfloor$ since $m = \left\lfloor \frac{m}{2} \right\rfloor + \left\lceil \frac{m}{2} \right\rceil$ for any natural number m. □

Lemma 27 ([GKP94], pp. 71)**.** *Let* $f(x)$ *be any continuous, monotonically increasing function with the property that*

$$f(x) \text{ is integer} \Rightarrow x \text{ is integer}$$

Then,

$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \quad \text{and} \quad \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

□

Corollary 4.

$$\forall x \in \mathbb{R}^+ \, \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor x \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{b} \right\rfloor \quad \text{and} \quad \left\lceil \frac{\lceil x \rceil}{b} \right\rceil = \left\lceil \frac{x}{b} \right\rceil \right)$$

Proof:
Apply Lemma 27 with $f(x) = \frac{x}{b}$. □

The equalities in Corollary 5 are presented in [CLR00] but without any proof.

Corollary 5.

$$\forall x \in \mathbb{R}^+ \, \forall a \in \mathbb{N}^+ \, \forall b \in \mathbb{N}^+ : \left(\left\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor \quad \text{and} \quad \left\lceil \frac{\lceil \frac{x}{a} \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil \right)$$

Proof:
Apply Corollary 4 with $\frac{x}{a}$ instead of x. □

Lemma 28. *In any binary heap* T *of* n *vertices, there are precisely* $\left\lceil \frac{\lfloor \frac{n}{2^d} \rfloor}{2} \right\rceil$ *vertices of depth* d *and* $\left\lfloor \frac{n}{2^d} \right\rfloor$ *vertices of depth* $\geq d$.

Proof:
Depth is defined as follows: any vertex u has depth d if the longest path p—that does not contain the parent of u if one exists—between u and any leaf is of length d (see also Definition 5 on page 116). The proof is by induction on d.

Basis. $d = 0$. The vertices of depth 0 are precisely the leaves of T. But there are precisely $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{\lfloor \frac{n}{2^0} \rfloor}{2} \right\rceil$ leaves in T (see Problem 131 on page 170) and $n = \left\lfloor \frac{n}{2^0} \right\rfloor$ vertices in T.

Induction Hypothesis. Assume the claim holds for some depth d.

Induction Step. Delete all vertices of depth $< d$ from T. In the obtained tree, call it T':

- the vertices of depth 0 are precisely the vertices of depth d in T, and
- all vertices are precisely the vertices of depth $\geq d$ in T.

We know (see Problem 131 on page 170) there are $\left\lceil \frac{n'}{2} \right\rceil$ leaves and $\left\lfloor \frac{n'}{2} \right\rfloor$ internal vertices in T', where n' is the number of vertices in T'. By the induction hypothesis, there are $\left\lfloor \frac{n}{2^d} \right\rfloor$ vertices of depth d in T, so $n' = \left\lfloor \frac{n}{2^d} \right\rfloor$. It follows there are $\left\lceil \frac{\left\lfloor \frac{n}{2^d} \right\rfloor}{2} \right\rceil$ vertices of depth d and $\left\lfloor \frac{\left\lfloor \frac{n}{2^d} \right\rfloor}{2} \right\rfloor$ vertices of depth $\geq d$ in T. By Corollary 5, $\left\lfloor \frac{\left\lfloor \frac{n}{2^d} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{2 \times 2^d} \right\rfloor$, and certainly $\left\lfloor \frac{n}{2 \times 2^d} \right\rfloor = \left\lfloor \frac{n}{2^{d+1}} \right\rfloor$. □

Problem 132. *Find a closed formula for*

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil$$

Solution:

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil =$$
$$\left\lfloor \frac{0-1}{2} \right\rfloor \left\lceil \frac{0-1}{2} \right\rceil + \left\lfloor \frac{1-1}{2} \right\rfloor \left\lceil \frac{1-1}{2} \right\rceil + \left\lfloor \frac{2-1}{2} \right\rfloor \left\lceil \frac{2-1}{2} \right\rceil + \left\lfloor \frac{3-1}{2} \right\rfloor \left\lceil \frac{3-1}{2} \right\rceil +$$
$$\ldots + \left\lfloor \frac{(n-1)-1}{2} \right\rfloor \left\lceil \frac{(n-1)-1}{2} \right\rceil + \left\lfloor \frac{n-1}{2} \right\rfloor \left\lceil \frac{n-1}{2} \right\rceil =$$
$$(-1) \times 0 + 0 \times 0 + 0 \times 1 + 1 \times 1 + 1 \times 2 +$$
$$\ldots + \left\lfloor \frac{(n-1)-1}{2} \right\rfloor \left\lceil \frac{(n-1)-1}{2} \right\rceil + \left\lfloor \frac{n-1}{2} \right\rfloor \left\lceil \frac{n-1}{2} \right\rceil$$

Suppose n is odd, *i.e.* $n = 2t+1$ for some $t \in \mathbb{N}$. We have to evaluate the sum

$$1 \times 1 + 1 \times 2 + 2 \times 2 + 2 \times 3 + \ldots + (t-1) \times t + t \times t =$$
$$\underbrace{1 \times 1 + 2 \times 2 + \ldots + (t-1) \times (t-1) + t \times t}_{A} + \underbrace{1 \times 2 + 2 \times 3 + \ldots + (t-2) \times (t-1) + (t-1) \times t}_{B}$$

By (6.23) on page 169, $A = \frac{t(t+1)(2t+1)}{6}$, and by Lemma 26 on page 169, $B = \frac{(t-1)t(t+1)}{3}$. So,

$$A + B = \frac{t(t+1)(2t+1)}{6} + \frac{(t-1)t(t+1)}{3} = \frac{t(t+1)}{3}\left(\frac{2t+1}{2} + (t-1)\right) =$$
$$\frac{t(t+1)}{3}\left(\frac{2t+1+2t-2}{2}\right) = \frac{t(t+1)(4t-1)}{6}$$

Now suppose n is even, *i.e.* $n = 2t$ for some $t \in \mathbb{N}$. We have to evaluate the sum

$$1 \times 1 + 1 \times 2 + 2 \times 2 + 2 \times 3 + \ldots + (t-1) \times (t-1) + (t-1) \times t =$$
$$\underbrace{1 \times 1 + 2 \times 2 + \ldots + (t-1) \times (t-1)}_{A} + \underbrace{1 \times 2 + 2 \times 3 + \ldots + (t-2) \times (t-1) + (t-1) \times t}_{B}$$

By (6.23) on page 169, $A = \frac{(t-1)t(2t-1)}{6}$, and by Lemma 26 on page 169, $B = \frac{(t-1)t(t+1)}{3}$. So,

$$A + B = \frac{(t-1)t(2t-1)}{6} + \frac{(t-1)t(t+1)}{3} = \frac{t(t-1)}{3}\left(\frac{2t-1}{2} + (t+1)\right) = \frac{t(t-1)}{3}\left(\frac{2t-1+2t+2}{2}\right) = \frac{t(t-1)(4t+1)}{6}$$

Overall,

$$\sum_{k=0}^{n} \left\lfloor \frac{k-1}{2} \right\rfloor \left\lceil \frac{k-1}{2} \right\rceil = \begin{cases} \dfrac{(n-1)(n+1)(2n-3)}{24}, & n \text{ odd} \\ \dfrac{(n-2)n(2n+1)}{24}, & n \text{ even} \end{cases} \tag{6.33}$$

□

Computational Problem HALTING PROBLEM
Generic Instance: $\langle \mathfrak{P}, \mathfrak{I} \rangle$ where $\mathfrak{P}$ is a computer program and $\mathfrak{I}$ its input
Question: Does $\mathfrak{P}(\mathfrak{I})$ halt? □

The following is a simplistic version of the proof of the famous undecidability result. For a more thorough treatment see, for instance, [Sip06].

Theorem 4. *The* HALTING PROBLEM *is algorithmically unsolvable.*

Proof:
Assume the opposite. Then there exists an program $\mathfrak{Q}$ with input an ordered pair $\langle \mathfrak{P}, \mathfrak{I} \rangle$ of (the encoding of) a computer program $\mathfrak{P}$ and its input $\mathfrak{I}$ (*i.e.*, $\mathfrak{I}$ is input to $\mathfrak{P}$), such that $\mathfrak{Q}(\mathfrak{P}, \mathfrak{I})$ returns TRUE if $\mathfrak{P}(\mathfrak{I})$ halts, and FALSE otherwise. Define program $\mathfrak{S}(\mathfrak{P})$ as $\mathfrak{Q}(\mathfrak{P}, \mathfrak{P})$. That is, $\mathfrak{S}(\mathfrak{P})$ consists of the single line

```
return Q(P, P)
```

Define yet another program $\mathfrak{T}(\mathfrak{P})$ as follows:

```
if S(P) then loop forever
else return TRUE
```

Analyse $\mathfrak{T}(\mathfrak{T})$. If it goes into infinite loop, it must be the case that $\mathfrak{S}(\mathfrak{T})$ returns TRUE. Then it must be the case that $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ returns TRUE. Then it must be the case that $\mathfrak{T}$ halts with input $\mathfrak{T}$. This is a contradiction.

If $\mathfrak{T}$ halts, it returns TRUE. Then it must be the case that $\mathfrak{S}(\mathfrak{T})$ returns FALSE. Then it must be the case that $\mathfrak{Q}(\mathfrak{T}, \mathfrak{T})$ returns FALSE. Then it must be the case that $\mathfrak{T}$ does not halt with input $\mathfrak{T}$. This is a contradiction, too.

□

Chapter 7

Acknowledgements

I express my gratitude to:

> **Zornitsa Kostadinova**, **Iskren Chernev**, **Stoyan Dimitrov**, **Martin Toshev**, **Georgi Georgiev**, **Yordan Stefanov**, **Mariya Zhelezova**, and **Nikolina Eftimova**

for all the errors and typos they discovered and corrected in this manual. In addition, I thank Georgi Georgiev for suggesting solutions and improvement to several problems.

Bibliography

[AB98] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.

[Bal91] V. K. Balakrishnan. *Introductory Discrete Mathematics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1991. Available online at `http://books.google.com/books?id=pOBXUoVZ9EEC&printsec=frontcover&dq=Introductory+discrete+mathematics++By+V.+K.+Balakrishnan&source=bl&ots=1lYLvMpVfY&sig=Jwklfma4Zf3EIvNC0UH-fmI5JPA&hl=en&ei=1MCkTf2II4_1sgaR0ISBBw&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBcQ6AEwAA#v=onepage&q&f=false`.

[Buz99] Kevin Buzzard. Review of *Modular forms and Fermat's Last Theorem, by G. Cornell, J. H. Silverman, and G. Stevens, Springer-Verlag, New York, 1997, xix + 582 pp., $49.95, ISBN 0-387-94609-8*. Bulletin (New Series) of the American Mathematical Society, Volume 36, Number 2, Pages 261–266, 1999.

[CLR00] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, first edition, 2000.

[CSS97] G. Cornell, J.H. Silverman, and G. Stevens. *Modular forms and Fermat's last theorem*. Springer, 1997. Available online at `http://books.google.com/books?id=Va-quzVwtMsC` .

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, second edition, 1973.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[LD05] Charles Leiserson and Erik Demaine. Assignments with solutions to free online course "Introduction to Algorithms", 2005. Available online at `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/assignments/`.

[Lei96] Leighton. Note on Better Master Theorems for Divide-and-Conquer Recurrences, 1996. Available online at `http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf`.

[MAMc] Minko Markov, Mugurel Ionuţ Andreica, Krassimir Manev, and Nicolae Ţăpuş. A linear time algorithm for computing longest paths in cactus graphs. Submitted for publication in Journal of Discrete Algorithms.

[Man05] Krasimir Manev. *Uvod v Diskretnata Matematika*. KLMN – Krasimir Manev, fourth edition, 2005.

[NIS] tree (data structure). National Institute of Standards and Technology's web site. URL `http://xlinux.nist.gov/dads//HTML/tree.html` .

[Sip06] Michael Sipser. *Introduction to the theory of computation: second edition*. PWS Pub., Boston, 2 edition, 2006.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[Slo] N. J. Sloane. The on-line encyclopedia of integer sequences. maintained by N. J. A. Sloane `njas@research.att.com`, available at `http://www.research.att.com/~njas/sequences/`.