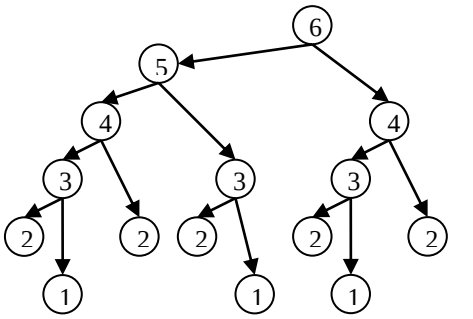
Динамично програмиране. Оценки за сложност.

Същност на алгоритмичната схема „динамично програмиране" – свеждане на задача със зададен размер към задачи от същия вид с по-малки размери и „мемоизация".

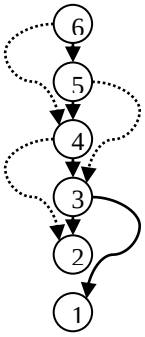
Алгоритмичната схема "динамично програмиране" може да се прилага в задачи, които имат рекурсивно решение. Ако в дървото на рекурсията има много еднакви подзадачи, то тогава би могло да се "запомни" решението на дадена подзадача и следващия път, когато се наложи да се решава тази задача, вместо това да се вземе резултатът наготово. Това "запомняне" наричаме "мемоизация" (memoization). Важно условие за прилагане на "мемоизация" е рекурсивните извиквания да бъдат "контекстно-свободни", т.е. да зависят само от параметрите си и всеки път при едни и същи параметри да се получава един и същ резултат. Така е сигурно, че колкото и да се решава дадената подзадача, винаги ще се получава резултатът, който е "запомнен" в началото.

За пример може да се вземе задачата за намиране на F_n – n-тото число на Фибоначи. Самите числа на Фибоначи са дефинирани рекурсивно, така че рекурсивното решение е ясно:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & ако\ n > 2 \\ 1 \qquad\qquad , & ако\ n = 1,2 \end{cases}$$



фиг. 1



фиг. 2

На фигура 1 е дадено дървото на рекурсията за n = 6. Ясно се вижда, как има голям брой повтарящи се подзадачи. Например, F_3 се изчислява три пъти, а колкото и пъти да се изчисли, резултатът ще е един и същ. Затова може да се приложи "мемоизация" на резултатите от решението на всяка подзадача и да се избегне рекурсивното извикване за нейното решаване, когато резултатът е наличен. Така всяка подзадача се решава само по един път, а броят на рекусивните извиквания намалява драстично. В примера с числото на Фибоначи F_6 извикванията ще бъдат тези, показани на фигура 2 с непрекъснатите стрелки. Потенциалните извиквания показани с пунктирани стрелки ще бъдат избегнати, защото съответните числа вече ще са пресметнати. Така рекурсивните извиквания на функции намаляват от 14 на 5. Чрез прилагане на "мемоизация" алгоритми с експоненциална сложност могат да бъдат сведени до такива с полиномиална. В случая на числата на Фибоначи, сложността става линейна.

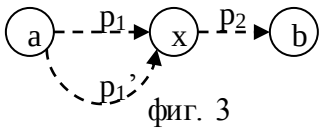
Дървото на рекурсията може да се сортира топологически. С други думи, може да се избере такъв ред за решаване на отделните подзадачи, че за намирането на дадено решение всички необходими резултати да са вече получени. В случая на числата на Фибоначи, подредбата е по номера – 1, 2, 3, 4, 5, 6. Така за пресмятането на F_4 ще са необходими F_3 и F_2, а те вече са известни. По този начин, ако съществува лесен начин да се подредят отделните подзадачи, отпада нуждата от рекурсия и поддържане на стек за изчисленията й. Вместо това се правят последователни изчисления, а резултатите се пазят в обикновена таблица.

Доста често топологичното сортиране на задачите се свежда до простото сортиране на аргументите им – както е и в случая с числата на Фибоначи. Обикновено самата рекурсивна дефиниция определя, че за решаване на задача с дадени аргументи са нужни резултатите от задачи с по-малки аргументи. Примерни аргументи могат да бъдат – пореден номер, максимална разрешена стойност, време, височина, брой, обем и т.н. Казва се, че свеждаме задача с даден размер към "по-малки" задачи (с по-малък размер), защото самите аргументи имат естествена наредба помежду си. Въвеждането на такова нереждане, планиране, програмиране на изчисленията едно след друго дава името "програмиране". Определението "динамично" идва от физичното понятие за динамика и набляга на това, че изчисленията имат развитие във времето, има определен ред, и не може да се реши една задача преди да се измине пътя по решаване на всички предхождащи я. Така създателят на този метод, Ричард Белман (Richard Bellman), се спира на името "динамично програмиране" (dynamic programming), което най-добре описва същността на алгоритмичната му схема.

Принцип за оптималност и конструиране на решението на задачата от решенията на подзадачите.

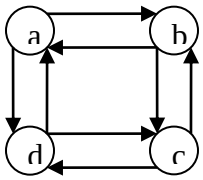
Първата стъпка при решаването на дадена оптимизационна задача чрез динамично програмиране е да се определи структурата на оптималното решение. Ако за намиране на оптимално решение се използват оптимални решения на по-малки задачи, тогава може да се прилага динамично програмиране. Нещо повече – ако това условие не е налице, тогава не е възможно прилагането на динамично програмиране, защото принципът е да свеждаме решаването на задачата до решаване на подзадачи от същия вид. Търси се оптимално решение, значи и подзадачите трябва да имат оптимални решения. Тази необходимост оптималното решение да се получава от оптимални решения на по-малки задачи се нарича "принцип за оптималност".

Нека разгледаме две примерни задачи – за най-къс и най-дълъг прост път в граф. Задачата за най-къс път следва принципа за оптималност. Нека вземем един най-къс път p между два различни върха a и b. Нека върхът x лежи на пътя p. Тогава частите от $p - p_1$, от a до x, и p_2, от x до b, също са най-къси пътища: Да допуснем, че съществува път p_1', от a до x, който е по-къс от p_1. Тогава пътят $p_1'p_2$ ще е по-къс от p_1p_2 и, следователно, съществува път p', който е по-къс от p, което е противоречие. Тези разсъждения дават не само увереност, че може да се използва динамично програмиране за намиране на най-къс път в граф, а дават и идея как точно да стане това.



фиг. 3

В задачата за най-дълъг прост път, обаче, не може да се приложи този принцип. Нека разгледаме ориентирания граф на фигура 4. Най-дългият прост път между a и c е $a \to b \to c$. Със сигурност, обаче, той не е съставен от най-дълги прости пътища. Например между a и b най-дългият прост път е $a \to d \to c \to b$, а не $a \to b$. Тъй като принципът за оптималност не е спазен, тук не може да се използва динамично програмиране. Всъщност, задачата за най-дълъг прост път в граф е NP-пълна и полиномиално решение не е известно засега.



фиг. 4

Задачи с линейна таблица на подзадачите (най-дълга растяща подредица).

Да разгледаме следната задача (задача *): Дадена е крайна редица от числа $\{a_i\}_1^n$. Растящи подредици на редицата $a=\{a_i\}_1^n$, ще наричаме редиците $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$, за които a_i са от a, $a_{i_1} < a_{i_2} < ... < a_{i_k}$ и $i_1<i_2<\ldots<i_k$. За дадена редица $\{a_i\}_1^n$, да се намери нейна растяща подредица с възможно най-голяма дължина.

Преди всичко трябва да забележим, че тук всъщност се търси подмножество на елементите, участващи в редицата. С други думи за всеки един елемент a_i има точно по две възможности – или е част от отговора на задачата, или не е. Нека вземем последния елемент a_n. Ще разгледаме двата варианта – ако участва в решението или не, и ще изберем по-добрия. Ако a_n не участва в подредицата-отговор, тогава, ако намерим оптималното решение за редицата $\{a_i\}_1^{n-1}$, то ще е решение и за $\{a_i\}_1^n$. Ако пък участва в най-дългата растяща подредица, тогава трябва да решим задачата (задача **) "да се намери най-дългата растяща подредица, която завършва в a_n". Задача * е частен случай на **, защото винаги можем да добавим едно число a_{n+1} след всички останали, което да е по-голямо от тях и когато решим **, махаме a_{n+1} и получаваме решение на *. Нека сега си представим, че имаме решение на задачата ** за всички числа от a_1 до a_{n-1}. Тогава за a_n решението е лесно: разглеждаме всички a_i, които са по-малки от a_n. Ако вземем това, за което сме намерили най-дългата нарастваща редица и "закачим" за нея a_n, тогава получената редица ще е решение на ** за a_n. Така достигаме и до решението:

Решение: Ще поддържаме вектор v (таблица само с един ред), в който на позиция с индекс i ще стои дължината на най-дългата подредица, която завършва с числото a_i. Инициализираме първата позиция с 1, защото ако се вземе само a_1, тогава най-дългата подредица е самият елемент a_1, а редицата има дължина 1. Последователно изчисляваме стойностите от втората до последната по формулата:

$v_i = \max_{1\le k<i}^{a_k<a_i}\{v_k\}+1$, а ако няма числа по-малки от a_i преди него, тогава $v_i = 1$.

Дължината на най-дългата нарастваща подредица е v_{n+1} - 1. За да намерим самата редица, трябва да поддържаме втори вектор p, в който на позиция p_i ще стои индекса на предшественика на a_i в редицата, която завършва с a_i. След намирането на всички стойности редицата може да се възстанови, като се тръгне от p_{n+1}.

Решението на задачата изисква линейна таблица на подзадачите – в случая два вектора. Сложността по време, обаче, е квадратна, защото за изчисляването на всяка стойност на вектора v, е необходимо да се обходят всички предишни стойности, за да се вземе най-голямата от тях.

Задачи с триъгълна таблица на подзадачите (оптимално разбиване на редица, разпознаване на КСЕ в НФ на Чомски).

Задача: N клиента c_1, c_2,…, c_N са се наредили на опашка за обслужване. Сървърът може да обслужи едновременно произволен брой K стоящи един след друг клиенти c_i, c_{i+1},…, c_{i+K-1}, i = 1,2,…,N; k = 1,2,…,N–i+1, като е известно времето $p_{i,i+K-1}$ необходимо на сървъра за да обслужи тези K клиенти едновременно. Да се намери такова разбиване на редицата в групи от последоватлно стоящи клиенти, че общото време за обслужване да бъде минимално, като всяка група на разбиването се обслужва едновременно.

Решение: Ще поддържаме таблица t с размер N на N. В клетката с координати (x,y) – $t_{x;y}$, ще пазим минималното време за обслужването на клиентите от c_x до c_y. Инициализираме всяка от клетките с координати $(x;x)$ с $p_{x,x}$. Попълваме таблицата, като разглеждаме всички подредици започвайки от тези с дължина 2 ($\{c_x, c_{x+1}\}$, за всички x от 1 до N-1) и стигнем до цялата редица $\{c_1, c_2,\ldots, c_N\}$. За да определим $t_{x;y}$ разглеждаме всички разбивания $t_{x;x+k}$, $t_{x+k+1;y}$ на редицата:

$t_{x;y} = \min\{t_{x;x+k} + t_{x+k+1;y}\}$, $k = 0,1,\ldots,y–x–1$. Ако полученият минимум за $t_{x;y}$ е по-голям от $p_{x;y}$, тогава $t_{x;y} = p_{x;y}$ (Това е случаят, в който направо обслужваме всички заедно, без да разбиваме на по-малки редици).

Накрая в $t_{1;N}$ получаваме търсеното минимално време. За да намерим самото разбиване, трябва да пазим още една таблица, в която за всяка група $c_x,\ldots, c_y$ да пазим къде сме разделили (ако сме я разделили изобщо). Накрая, тръгвайки от клетката (1, N), можем чрез обхождане в дълбочина да възстановим разделянето.

Нека забележим, че няма смисъл да се попълват множествата $t_{x;y}$, за x>y. По тази причина ще използваме само половината от таблицата t – онези клетки, които са над единия диагонал. Затова и имаме триъгълна таблица на подзадачите.

Задача: Дадена е контекстно-свободна граматика в нормална форма на Чомски (дадени са множествата от терминали и нетерминали, както и множеството от правила. Правилата са от вида "$A \to a$" или "$A \to BC$"). Да се определи дали дадена дума s принадлежи на езика, определен от дадената граматика.

Решение: Ще поддържаме таблица t с размер |s| на |s| (|s| е броят букви в думата). Нека $s_{x;y}$ е думата, която се получава като от s се вземе само частта от x-тата до y-тата буква а $s_{x;x}$.е просто x-тата буква. В клетката с координати (x,y) – $t_{x;y}$, ще пазим множеството от нетерминали, от които е изводима $s_{x;y}$. Инициализираме всяка от клетките с координати $(x;x)$, като намираме множеството от всички нетерминали A_i, участващи в правила от вида $A_i \to s_{x;x}$. Попълваме таблицата, като разглеждаме всички поддуми започвайки от тези с дължина 2 и стигнем до самата дума s. За да определим множеството $t_{x;y}$ разглеждаме всички разбивания $s_{x;x+k}$, $s_{x+k+1;y}$ на $s_{x;y}$, $k = 0,1,\ldots,y–x–1$. Сега $t_{x;y} = \{ A_i \mid \exists\, A_i \to BC, B\in t_{x;x+k}, C\in t_{x+k+1;y}\}$. Ако след попълването на таблицата началният нетерминал (аксиомата) S принадлежи на $t_{1;|s|}$, думата s е изводима от S и значи принадлежи на езика на граматиката.

Тук отново ще използваме само половината от таблицата t – онези клетки, които са над единия диагонал. Затова и имаме триъгълна таблица на подзадачите.

Въпреки че можем да спестим половината памет, като не заделяме за неизползваните клетки от таблицата, сложността по памет си остава асимптотично равна на $|s|^2$. Тук големината на всяка клетка от таблицата (всяко множество) зависи от броя на нетерминалите и той би могъл да даде отражение върху използваната памет. Сложността по време е $u.|s|^2$, където u зависи от начина на реализация на множествата от правила и сложността, с която търсим и добавяме в тях.

Задачи с правоъгълна таблица на подзадачите (най-дълга обща подредица на две редици, задача за раницата)

Задача (най-дълга обща подредица на две редици): Дадени са редиците $a = \{a_i\}_1^n$ и $b = \{b_i\}_1^m$. Да се намери най-дългата редица $\{c_i\}_1^k$, която е подредица едновременно на редиците a и b.

Решение: Ще поддържаме таблица t с размер n на m. В клетката с координати $(x;y)$ – $t_{x;y}$, ще пазим дължината на най-дългата обща подредица на $\{a_i\}_1^x$ и $\{b_i\}_1^y$. Нулевите ред и стълб инициализираме с нули, защото празна редица винаги има нула общи елемента с която и да е друга. Последователно, от първата към последната колона, изчисляваме стойностите във всеки ред, като започнем от първия и стигнем до последния (от малки към големи индекси). Всяка стойност в таблицата пресмятаме по следната формула:

$$t_{x;y} = \begin{cases} t_{x-1;y-1} + 1, & a_x = b_y \\ \max\{t_{x-1;y}, t_{x;y-1}\} & a_x \neq b_y \end{cases}$$

Формулата накратко казва, че ако последните елементи на редиците са равни, със сигурност ще участват в най-дългата обща подредица, а ако пък са различни със сигурност поне един от двата няма да участва в крайния резултат.

За да намерим самата подредица, трябва да пазим още една таблица, в която на позиция $(x;y)$ да записваме от коя клетка е дошъл резултатът в $t_{x;y}$: от $t_{x-1;y-1}$, от $t_{x-1;y}$ или от $t_{x;y-1}$. След изчисляването на максимално възможната дължина на обща подредица в $t_{n;m}$, можем да тръгнем от клетката $(n;m)$ назад и да възстановим подредицата, от която сме получили оптималната дължина – ако стойността в клетка $t_{x;y}$ се е получила от $t_{x-1;y}$ или от $t_{x;y-1}$ не правим нищо, а ако се е получила от $t_{x-1;y-1}$ – добавяме в построяваната отзад напред редица буквата a_x (или, което е същото, b_y).

Тук използваната памет е съизмерима с $n.m$. Сложността по време също е такава, защото за изчисляването на всяка клетка от таблицата използваме константен брой операции.

Задача за раницата: Разполагаме с раница, която може да побере определен обем v (или тегло) и n на брой предмета (нека за определеност са номерирани с числата от 1 до n), всеки от които има определен целочислен обем и определена стойност. Задачата е да изберем такива предмети, че общата им стойност да е възможно най-голяма и в същото време общият им обем да не надхвърля обема на раницата.

Решение: Ще поддържаме таблица t с размер n на v. В клетката с координати $(x;y)$ – $t_{x;y}$, ще пазим максималната цена на предметите, които можем да поберем в раница с обем y, като имаме право да избираме само измежду първите x предмета. Нулевите ред и стълб инициализираме с нули, защото в раница с обем нула нищо не може да се сложи, а ако нямаме предмети общата им стойност е винаги нула. Последователно, от първата към последната колона, изчисляваме стойностите във всеки ред, като започнем от първия и стигнем до последния (от малки към големи индекси). Всяка стойност в таблицата пресмятаме по следната формула:

$$t_{x;y} = \begin{cases} t_{x-1;y} & v_x > y \\ \max\{t_{x-1;y}; c_x + t_{x-1;y-v_x}\} & v_x \le y \end{cases}$$

Формулата накратко казва, че ако даден предмет не може да се побере в раницата, изобщо не го разглеждаме, а ако може, сравняваме кое е по-добре: да го сложим и да ни остане по-малко място за другите предмети или да не го сложим и обемът на раницата да се запази.

За да намерим номерата на използваните предмети, трябва да поддържаме още една таблица, където в клетката $(x;y)$ да пазим от коя клетка на t сме получили резултата. След изчисляването на $t_{n;m}$ можем да тръгнем от клетката $(n;m)$ назад и да възстановим направения избор.

И тук използваната памет е съизмерима с $n.m$. Сложността по време също е такава, защото за изчисляването на всяка клетка от таблицата използваме константен брой операции.

Сложности

В разгледаните задачи намирахме размера на решението, а за самото решение се налагаше да поддържаме допълнителна таблица. В много случаи ни интересува само размера, без да има нужда да даваме конкретен пример за решение. Тогава, често можем да съкратим използваната памет.

Да вземем за пример дефиницията на числата на Фибоначи:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & ако\ n > 2 \\ 1 \quad\quad\quad\quad , & ако\ n = 1,2 \end{cases}$$

Вижда се, че за изчисляването на следващото число ни трябва информация само за предишните две числа, а не за абсолютно всички намерени до момента. Затова вместо вектор от стойности можем да "помним" само последните две намерени стойности. По този начин, ако трябва да решим еднократно задача за конкретно n, можем да сведем използваната памет до константа, независеща от n. При това, сложността по време не се повлиява от тази оптимизация на паметта.

В задачата за най-дълга растяща подредица не можем да приложим същата оптимизация, защото при изчисляването на поредната стойност имаме нужда от всички предишни резултати.

По същият начин – в задачите за оптимално разбиване на редица и за разпознаване на КС език, за да определим $t_{x;y}$, имаме нужда от информация за редици/думи с всякакви дължини. Затова тук няма как да направим оптимизация в използваната памет.

Формулата за решение на задачата за раницата, обаче, дава възможност за оптимизиране:

$$t_{x;y} = \begin{cases} t_{x-1;y} & v_x > y \\ \max\{t_{x-1;y}; c_x + t_{x-1;y-v_x}\} & v_x \le y \end{cases}$$

Вижда се, че винаги използваме за решението с предмет номер x, решения за x–1. Това означава, че ако искаме да получим само стойността на предметите, без самите предмети, можем да пазим само информацията за решенията за x–1, което е линейно по m и използваната памет става n пъти по-малко.

Като цяло изводът е, че ако не търсим структурата на оптималното решение, бихме могли **в някои от случаите** да редуцираме сложността по памет до размерност по-малка от сложността по време – линеен алгоритъм с константна сложност по памет, квадратичен – с линейна сложност по памет и т.н.