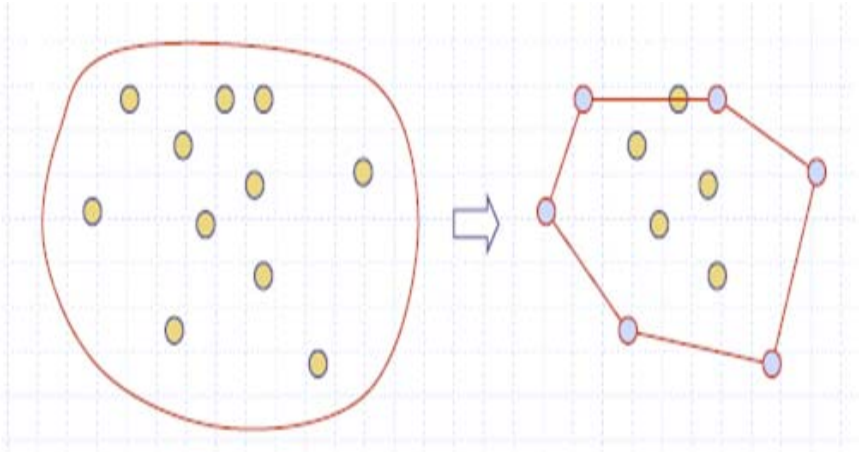
Space Efficient Convex Hull Algorithms

Rifat Shahriyar
100705037p

What is Convex Hull ?

- Let S={s[0],s[1],……,s[n-1]} be a set of n distinct points in the Euclidean space.
- The convex hull of S is the minimal convex region that contains every point of S.
- So the convex hull of S is a convex polygon whose vertices are points of S.



Various Algorithms in 2D

- Easy algorithm – $O(n^3)$
- Jarvi's March – $O(n^2)$
- Preparata's and Shamos's Quick Hull – $O(n^2)$
- Graham's Scan – O(n logn) ['72]
- **Kirkpatrick's and Seidel's algorithm – O(n logh) ['83]**
- **Chan's output sensitive algorithm – O(n logh)['96]**
- Incremental algorithm – $O(n^2)$ but can be reduced to O(n logn)
- Divide and Conquer algorithm – O(n logn)

Space Efficient Algorithms

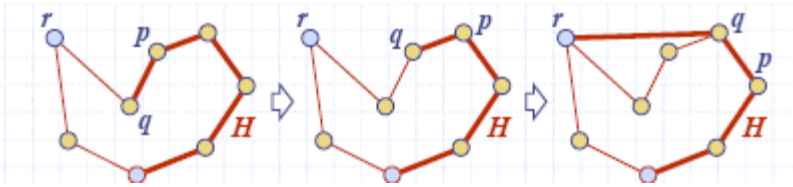
- An algorithm is space efficient if its implementation requires ***little or no extra memory*** except to store the input
- For convex hull , these algorithms take the input points as an array and output the vertices of the convex hull ***in the same array***
- The additional working storage is kept to a minimum
- Two types
  - In-place - input array and O(1) extra memory
  - In situ - input array and O(log n) extra memory

Why Space Efficient Algorithms ?

- They allow for the processing of larger data sets.
- They don't need to store input and output points separately which requires storing 2n points.
- They typically avoid virtual memory paging and external I/O bottlenecks.
- They exhibit greater locality of reference so it is practical for implementation in modern computer architectures with memory hierarchies.

In-place O(n logn) algorithm

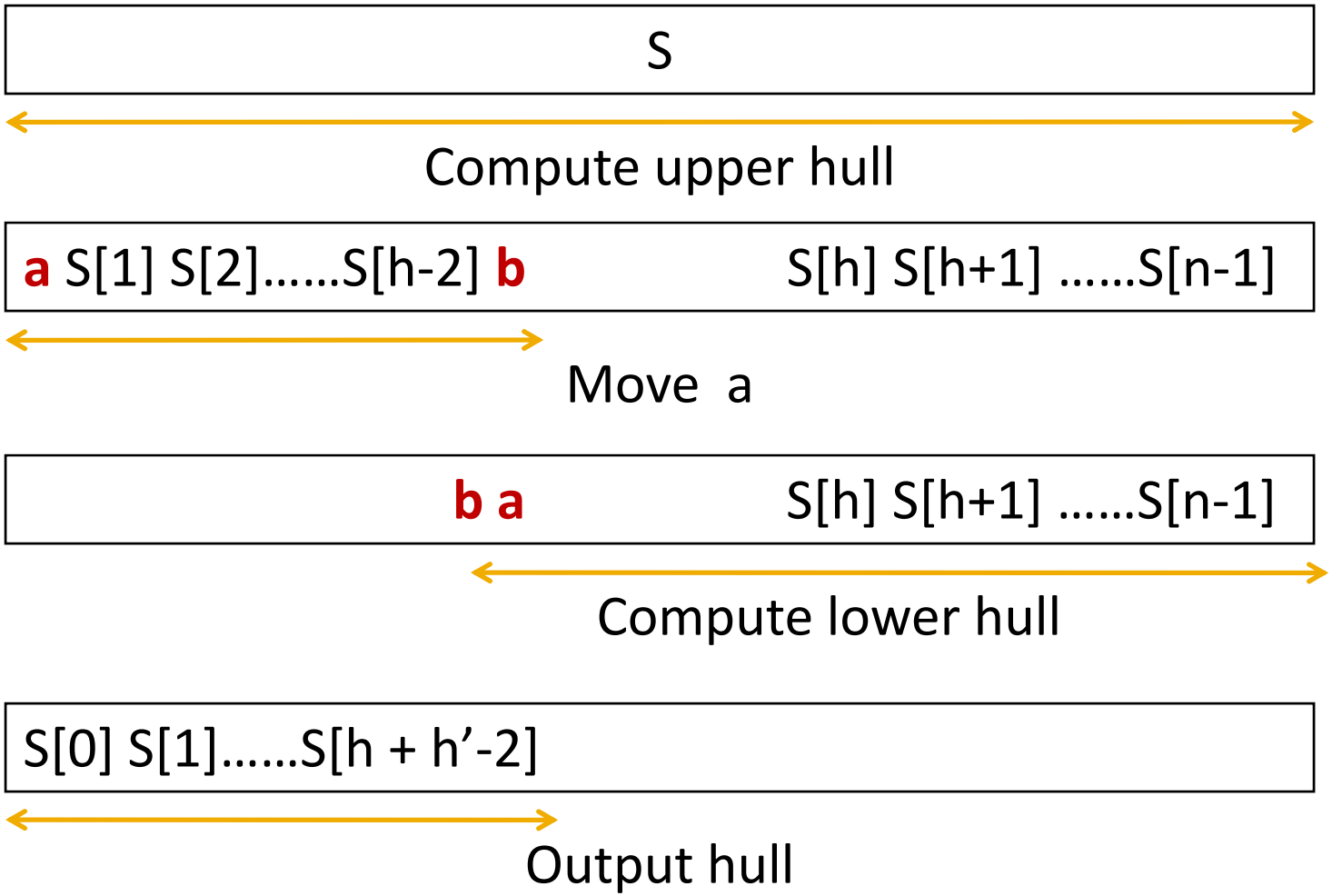
- Based on Graham's algorithm
- Uses in-place sorting algorithm
- Uses the concept of upper and lower convex hull
- It determines the upper hull first and then the lower hull
- The convex hull vertices of both hulls are stored in the same array where the input resides



GRAHAM-INPLACE-SCAN(S, n, d)

1: INPLACE-SORT(S, n, d)
2: $h \leftarrow 1$
3: for $i \leftarrow 1 \ldots n-1$ do
4: while $h \geq 2$ and not right_turn$(S[h-2], S[h-1], S[i])$ do
5: $h \leftarrow h-1$ { pop top element from the stack }
6: swap $S[i] \leftrightarrow S[h]$
7: $h \leftarrow h+1$
8: return h

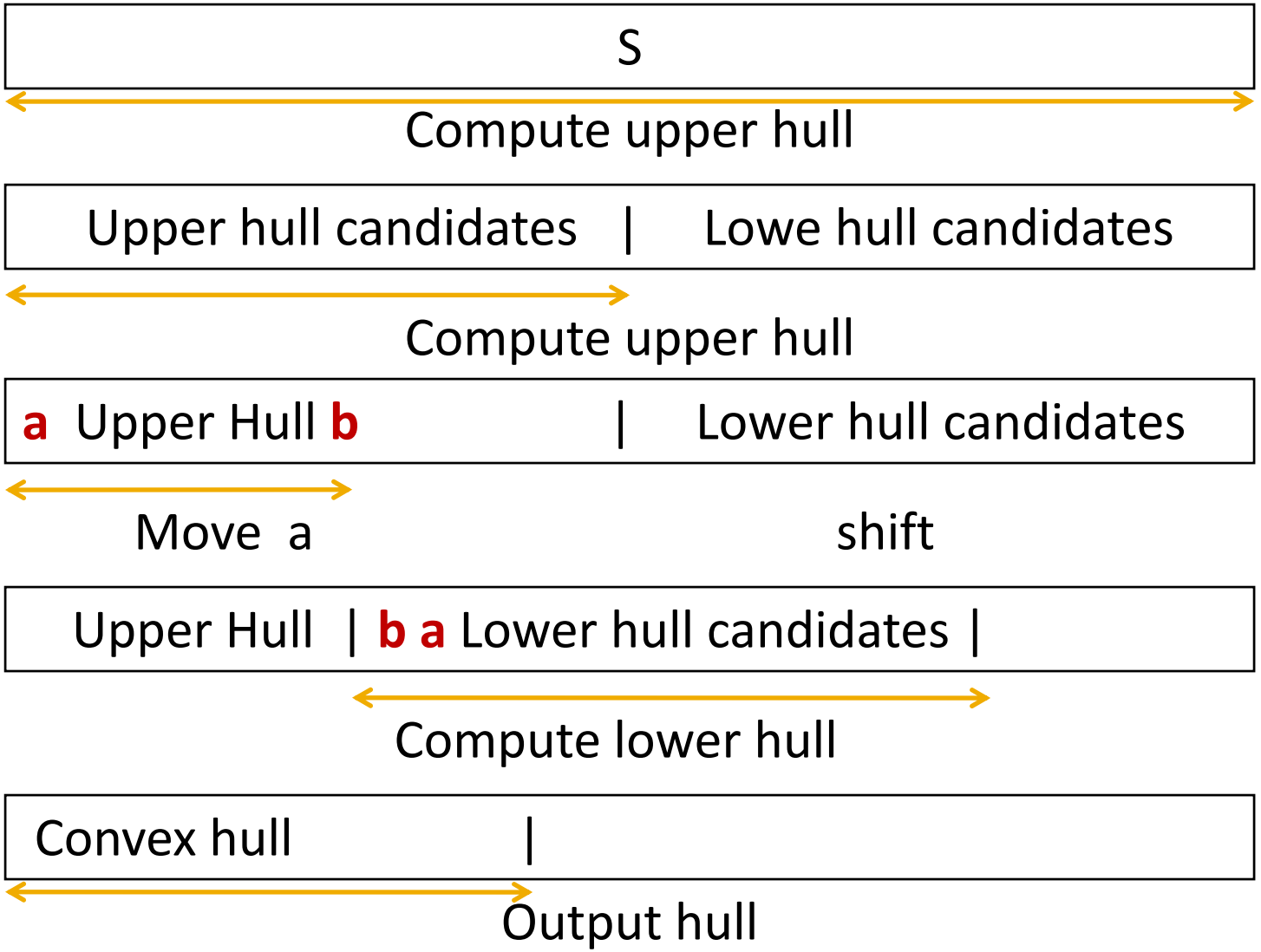
In-place O(n logn) algorithm



Optimized In-place O(n logn) algorithm

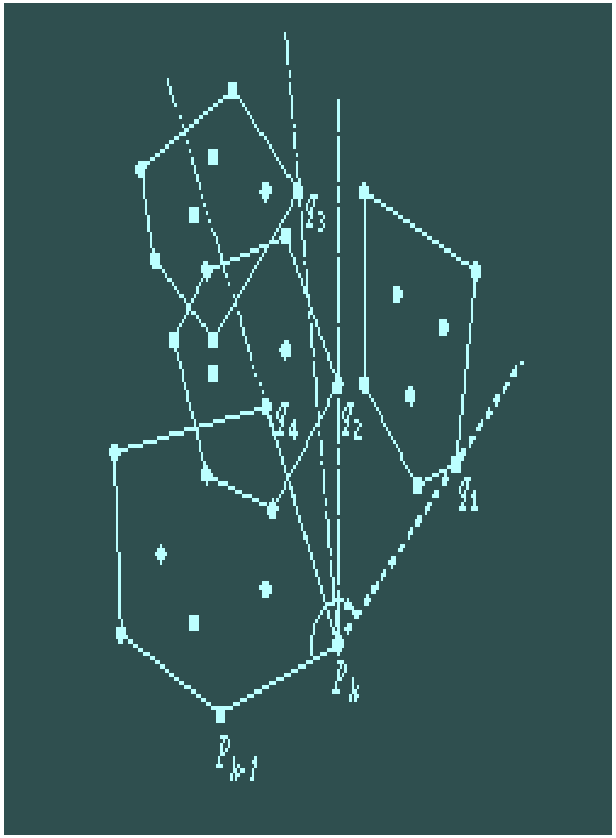
- Find the two extreme points **a** and **b**
- Partition the input array into two parts
- One part contains vertices that can only appear on the upper hull
- Same is true for lower hull
- Each point (except **a** and **b**) is examined only one call of Graham's algorithm

Optimized In-place O(n logn) algorithm



In-place O(n logh) algorithm

- Based on Chan's algorithm
- It runs in rounds and during the i^{th} round it finds the first $g_i=2^{2^i}$ points on the convex hull.
- During round i, it partitions the input points into n/g_i groups of size g_i and compute the convex hull of each group.
- The vertices are output in clockwise order beginning with the leftmost.
- Each successive vertex is obtained by finding tangents from the previous vertex to each of the n/g_i convex hull.



In-place O(n logh) algorithm

- Complexity : O(n logh)
- We want a space efficient implementation
- n/g_i groups of size g_i are build by taking groups of consecutive elements in S
- Compute hull using Graham-Inplace-Hull
- ***Two questions***
  - When tangent finding started where to put the founded hull vertices ?
  - To find the tangent from a point to a group how to find the size of the group's hull ?

Question-1 Solution

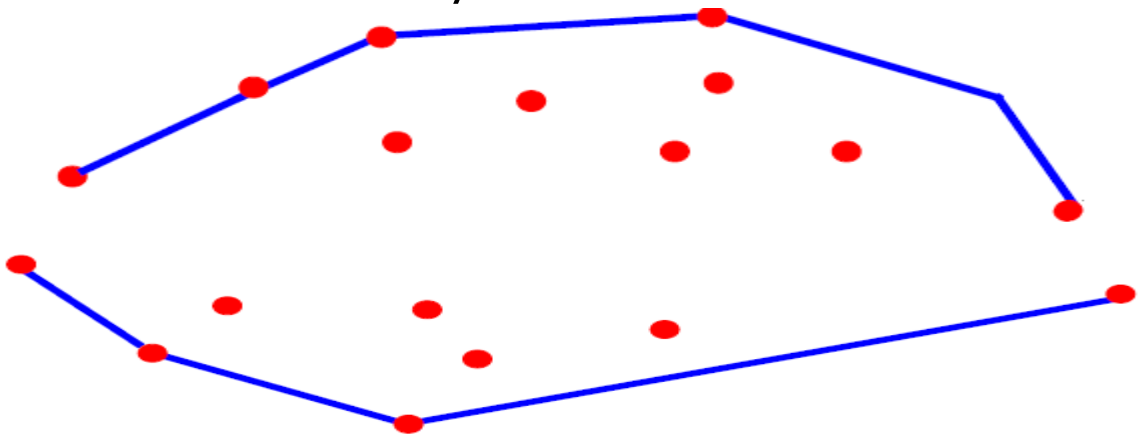
- Store the convex hull vertices at the beginning of S in the order they are found.
  - When finding the k^{th} vertex swap it with S[k-1]
- But due to this ***two changes*** occur.
  - Convex hull of the 1^{st} group is changed
  - Convex hull of the group containing the newly found vertex is changed
- What to do ??
  - Re-compute their convex hulls.

Question-1 Solution

- But what is the effect of the re-computations in the overall running time ?
- During one step at round i that has at most g_i steps
  - One convex hull vertex is found.
  - Two re-computation is done that takes $O(g_i \log g_i)$
  - Total cost of re-computation is $O(g_i^2 \log g_i)$
  - Total cost of round i is $O(g_i^2 \log g_i + n \log g_i)$ which is bounded by $O(n \log g_i)$.
- So overall complexity O(n logh).

Question-2 Solution

- We need keep track the size of the convex hull of each group without storing them.
- How it can be done ?? Using Reordering trick.
- G[0],G[1].....G[g_i-1] denotes the elements of group G.
- The ***sign of G[j] is +1 if G[j] < G[j+1]*** and – otherwise [where < denotes lexicographic comparison].
- So first elements G[0]....G[h-2] of convex hull of G form a sequence of 1 or more +'s followed by 0 or more –'s.



Other space efficient algorithm

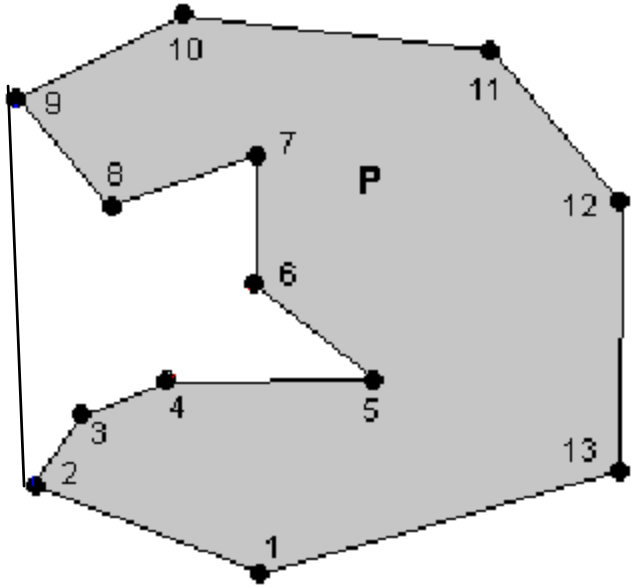
- **Chan, Snoeyink and Yap's algorithm**
  - Recursive
  - Partition the problem into two roughly equal size partitions
  - Finds a point p on the convex hull that leaves roughly the same no of vertices on each side.
  - O(n logh) algorithm which can be implemented using O(logn) additional storage for median finding.
  - So in-situ algorithm.

Other space efficient algorithm

- **Kirkpatrick and Seidel's algorithm**
  - Recursive
  - Partition the problem into two roughly equal size partitions
  - Finds an edge of the upper hull that leaves approximately the same no of points on each side.
  - O(n logh) algorithm which can be implemented using O(logn) additional storage
  - So in-situ algorithm.

Convex Hull of a Polygonal Line

- The polygon is given instead of the points
- So no need to sort the input points.
- Algorithms
  - Sklansky [‘72]
  - McCallum , Avis [’79]
  - Lee [‘83]
  - Graham , Yao [’83]
  - Bhattacharya , ElGindy [‘84]
  - Preparata and Shamos [‘85]
  - Shin , Woo [’86]
  - Melkman [’87]

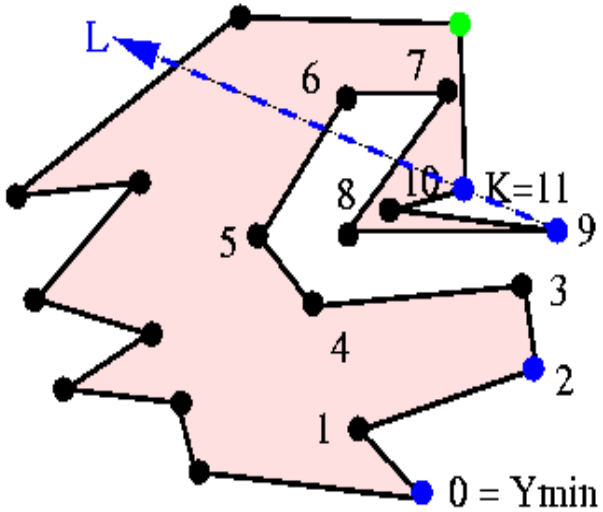


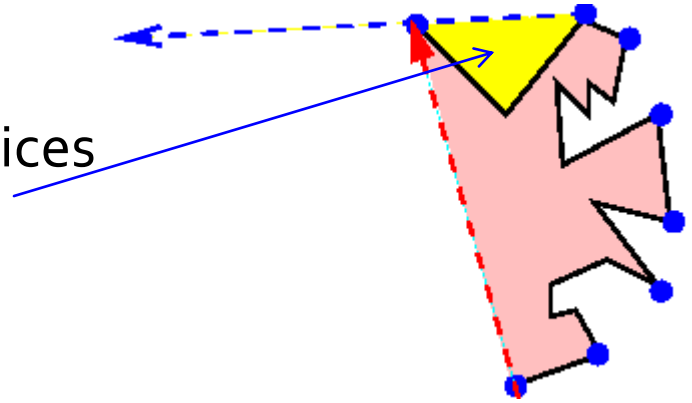
Lee's Algorithm

- Works for closed polygonal line
- A **lobe** of a polygon P is a region exterior to P, which is bounded by P and one line segment L between two vertices of the convex hull of P.
- After any iteration the stack contains the convex hull of all vertices that have been examined, *except* vertices which are known to be in the body of a lobe.
- Locate the $\mathbf{y}_{\mathbf{min}}$ and push it on the stack. Then push the next vertex counterclockwise.

Lee's Algorithm

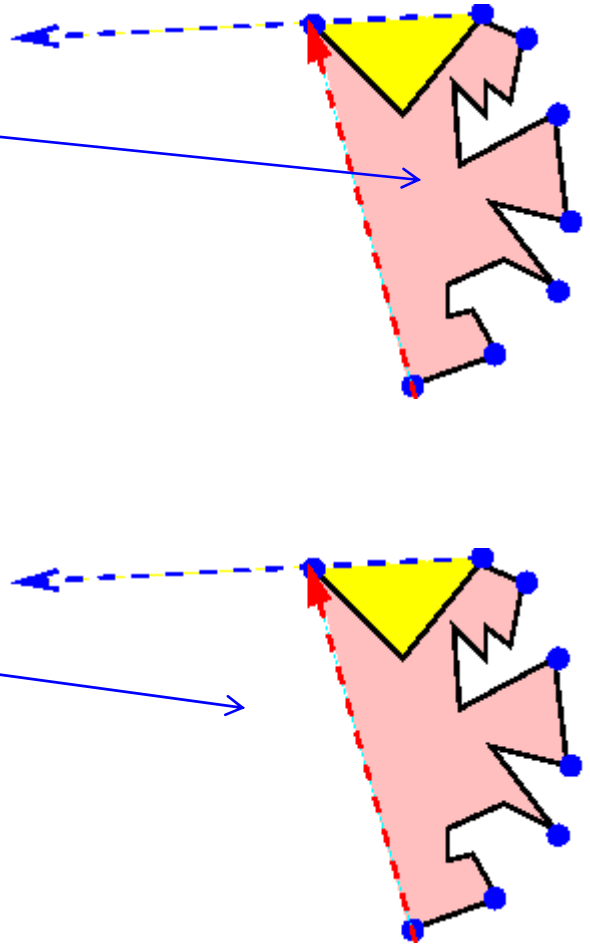
- Stack contains the blue points . Call the next vertex as the active vertex.
- The line L is formed by the top two vertices on the stack.
- **Step-1 : If the vertex is not to the left of L**
  - Delete the top element of the stack.
  - If y_{min} is left, push the vertex and get the next one. Compute L. Go back to [1].
- **If the vertex is to the left of L**
  - **Case-1:** In the lobe of the top two vertices of the stack.
    - Ignore the vertex, make the next vertex active and go back to [1].





Lee's Algorithm

- **Case 2:** Not in the lobe, but inside the convex polygon of the stack.
  - Ignore the vertex, make the next vertex active and go back to [1].
- **Case 3:** Is not in yellow or pink region (it's to the left of both dashed lines).
  - Push the vertex onto the stack. Compute L and go back to [1].
- **Terminate when y_{min} is examined again.**



How to make space efficient

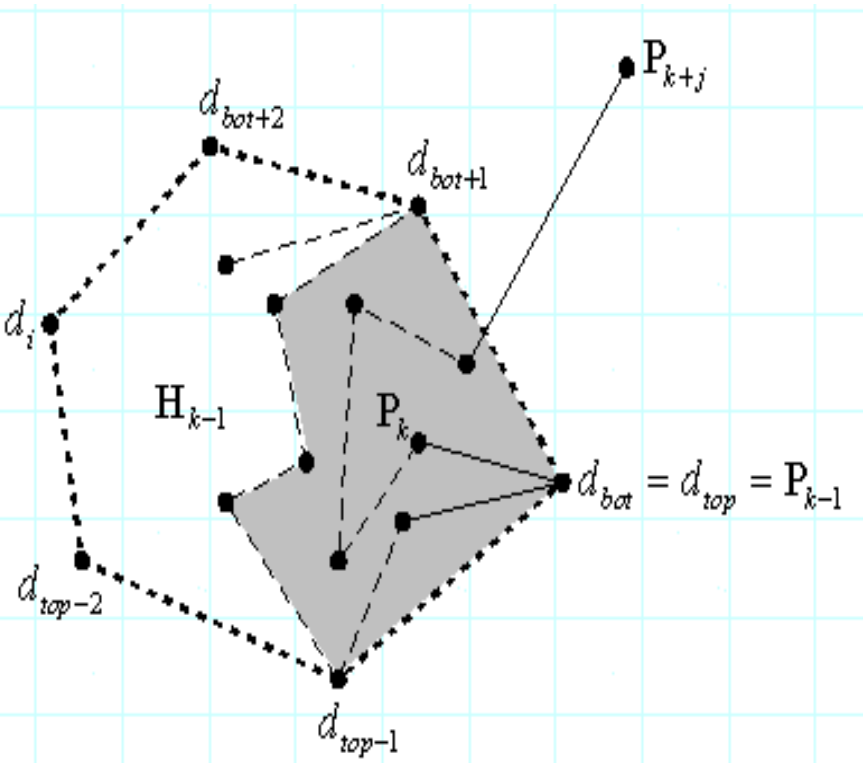
- Store the stack in the prefix of the array.
- The points inside the convex hull are stored in the same array.
- To do this swap is performed instead of assignment during push and pop operations.
- The algorithm produces a permutation of the input and index h such that
  - **S[1]…S[h] form a convex polygon**
  - **S[h+1]….S[n] are inside the convex hull**.
- Runtime is linear using O(1) extra memory because the y_{min} can be found in linear time with O(1) memory.

Melkman's algorithm

- Works for open and closed polygonal line.
- It determines and stored those vertices that form the hull for all vertices considered so far.
- When the next vertex is encountered:
  - If it is inside then ignore.
  - If it is outside then it becomes a new hull vertex.
- It uses deque that has both top and bottom and given by $D = \{ d_{bot}, \ldots, d_{top} \}$
- The elements d_i are vertices that form a polyline. When $\boldsymbol{d_{top} = d_{bot}}$, then D forms a polygon.

Melkman's algorithm

- In the Melkman hull algorithm, after processing vertex P_k, the deque D_k satisfies:
  - The polygon D_k is the ccw convex hull H_k of the vertices $W_k = \{P_0, \ldots, P_k\}$ already processed.
  - $d_{top} = d_{bot}$ is the most recent vertex processed that was added to D_k.
- **If P_k is inside H_{k-1}**, then **$D_k = D_{k-1}$**, and no processing.



Melkman's algorithm

- **When P_k is exterior to H_{k-1},** we change D_{k-1} to produce a new deque D_k by adding P_k to both the bottom and top of the deque and P_k will be inside the new polygon defined by D_k.
- Oher points already in D_{k-1} may get absorbed into the new hull H_k and they need to be removed before P_k is added to the deque ends.
- Vertices are removed from the two ends of D_{k-1} until the lines from P_k to the remaining deque endpoints form tangents to H_{k-1}.
- Complexity is O(n) because it consists just a series of simple isLeft () tests.

How to make space efficient

- The main problem is how to implement a deque of **n** elements in place.
- That means use of only the first **n** cells of the array when **n** points have been processed.
- This is as hard as stable partitioning.
- So techniques for stable partitioning can be adapted here.
- Double linked list may be used for constant time deque operations but extra spaces are needed by the pointers.
- So pointers are not stored explicitly but encoded implicitly via permutations of the input elements.
- The scheme is complex so there is a scope to find an efficient way to provide space efficiency here.

Thanks