Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture – 3 : Divide and Conquer Strategy

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

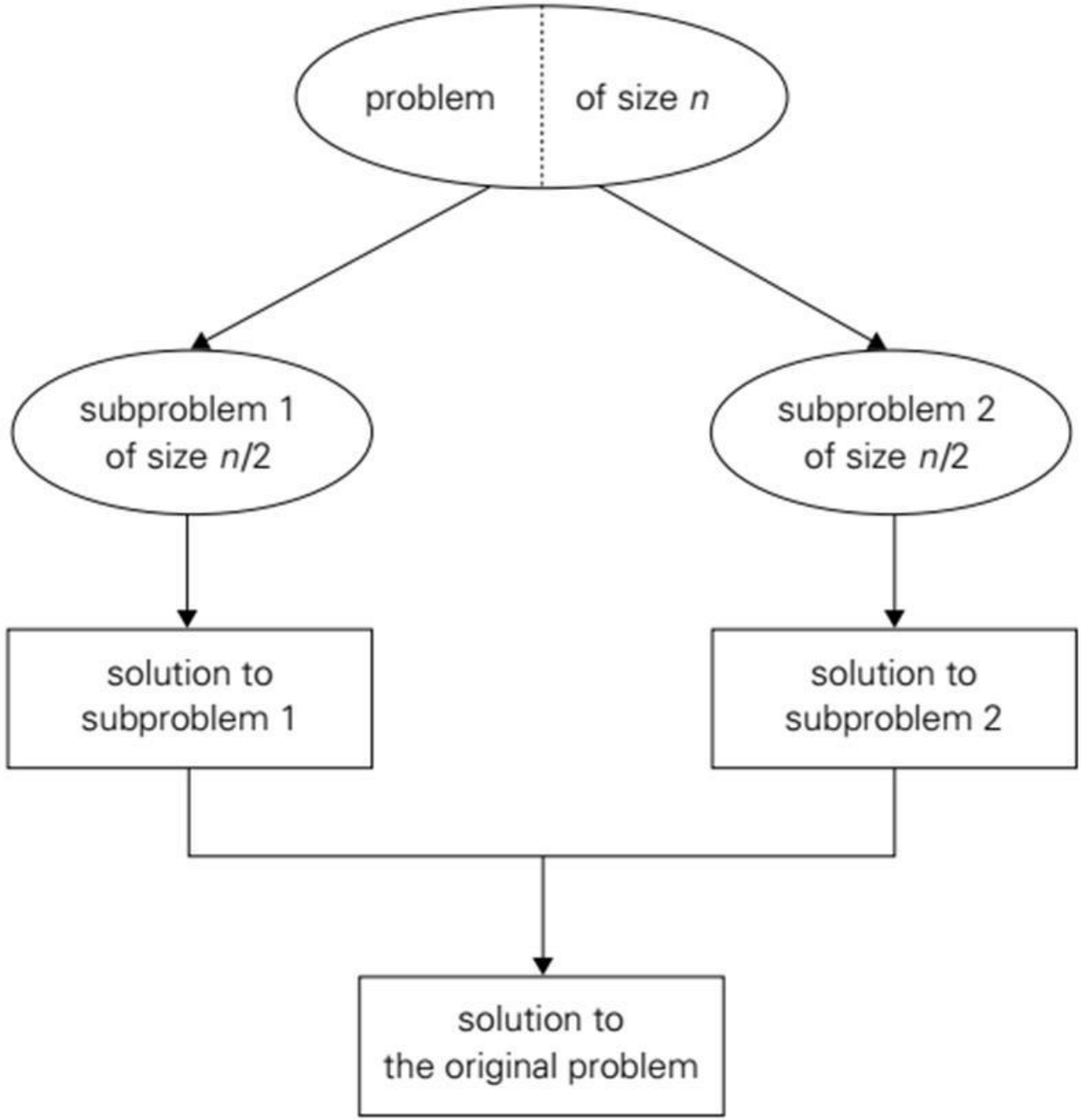
Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- Divide-and-conquer is probably the best-known general algorithm design technique.
- Quite a few very efficient algorithms are specific implementations of this general strategy.
- Divide-and-conquer algorithms work according to the following general plan:
  1. A problem is divided into several subproblems of the same type, ideally of about equal size.
  2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
  3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem

Divide-and-conquer technique



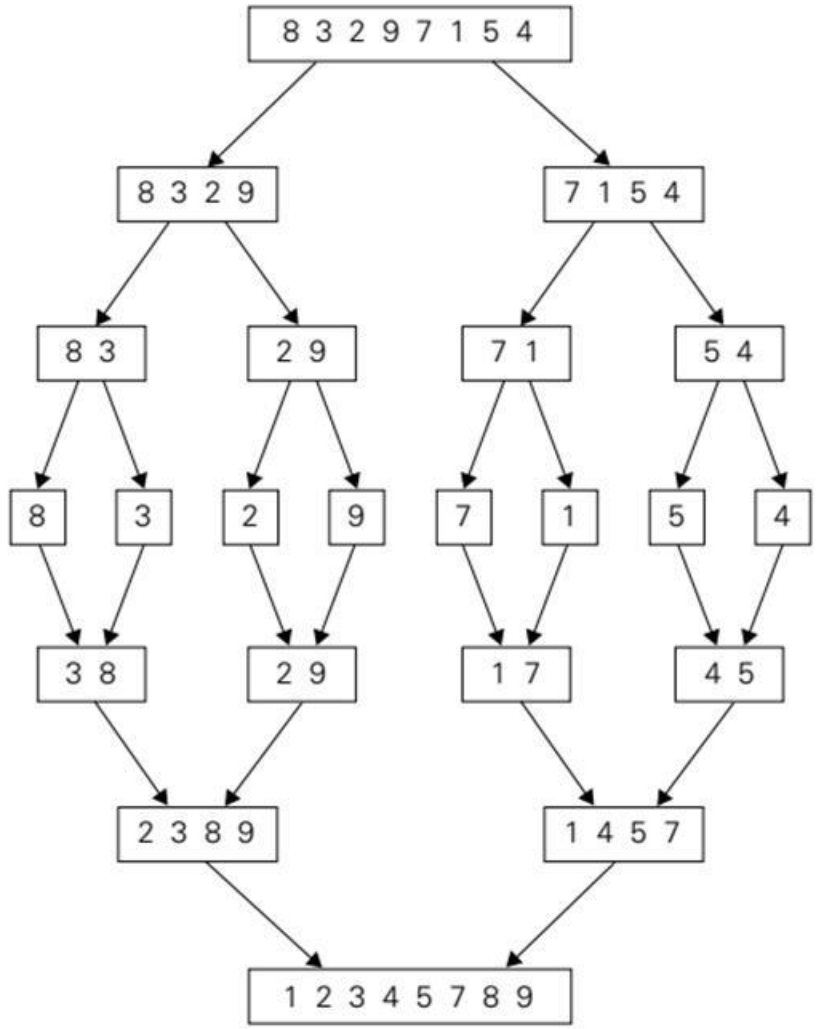
Divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor

Comparative Sorting the Array Elements

- Insertion sort –$O(n^2)$
- Selection sort - $O(n^2)$
- Bubble sort - $O(n^2)$
- Heap sort – O(nlogn)
- Merge sort - O(nlogn)
- Quick sort - $O(n^2)$
- Non –comparative Sorting
  - Counting sort – O(n)
  - Radix sort – O(n)
  - Bucket sort – O(n)

Mergesort

- Mergesort is a perfect example of a successful application of the divide-andconquer technique.
- It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0.....(n/2-1)]$ and $A[n/2....(n-1)]$,
  - sorting each of them recursively.
  - merging the two smaller sorted arrays into a single sorted one.

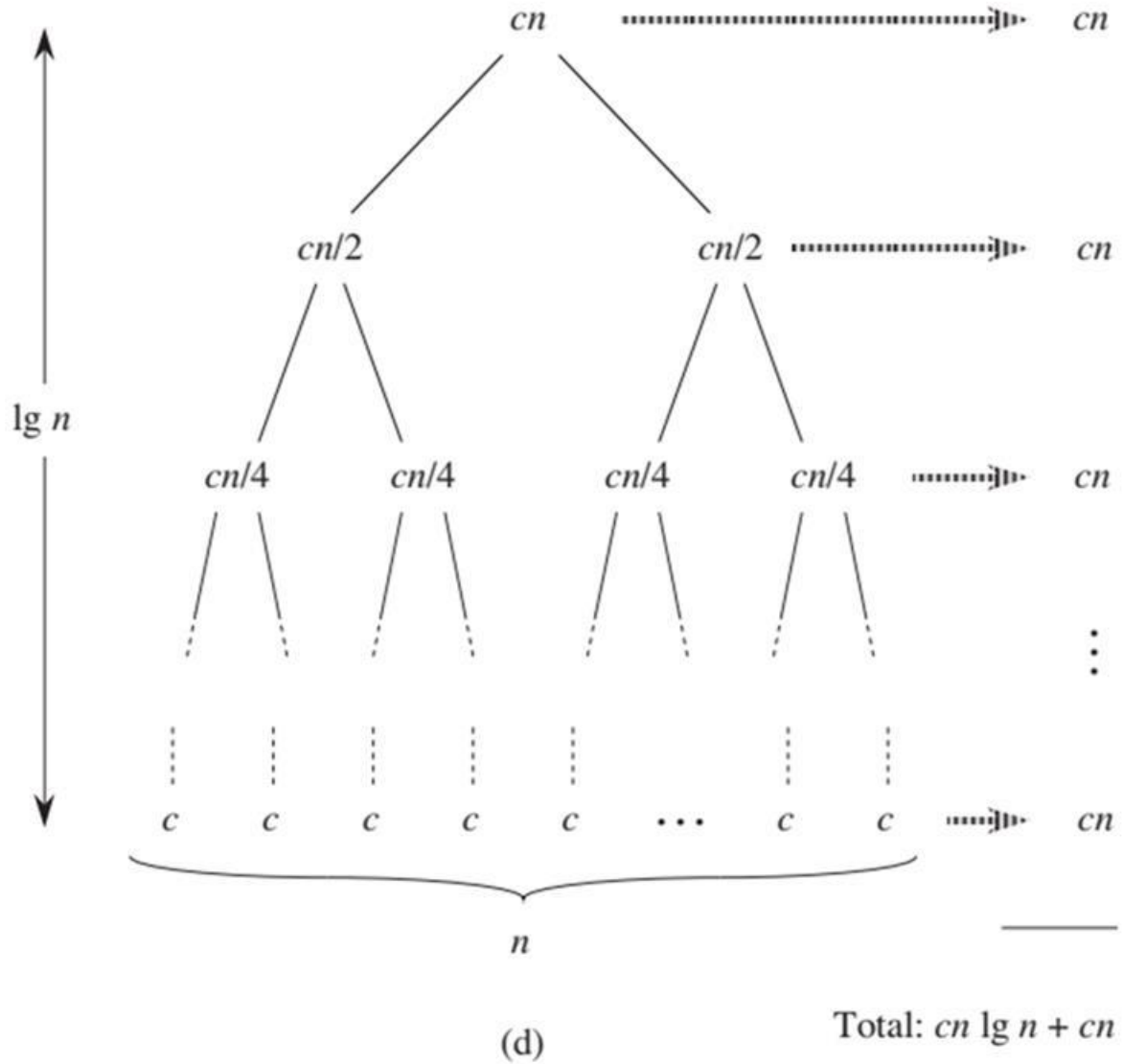


Mergesort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

T(n)= 2T(n/2)+θ(n)

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```



Efficiency!

O(nlogn)

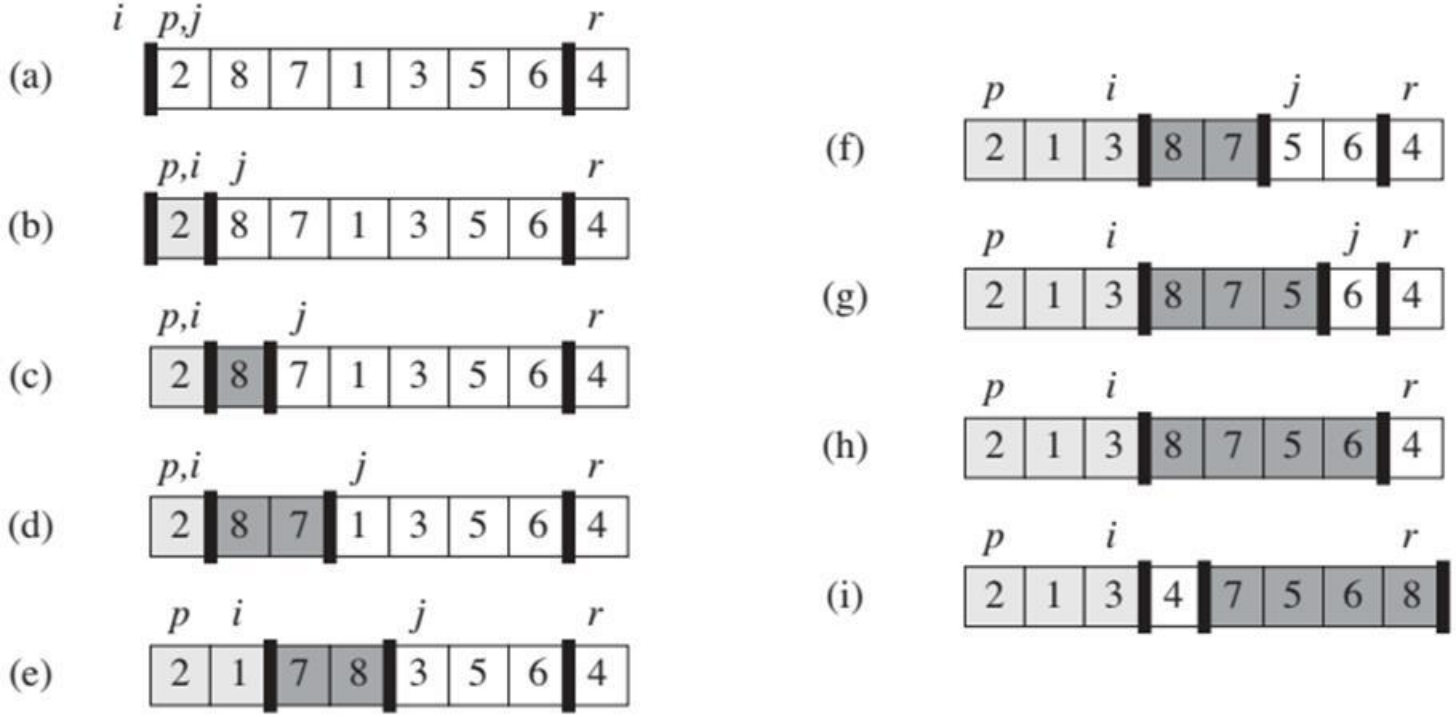
Quicksort

- Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- **Divide:**
  - Partition (rearrange) the array A[p…r] into two (possibly empty) subarrays A[p….(q - 1)] and A[(q + 1) …. r]
  - Each element of A[p …. (q -1)] is less than or equal to A[q],
  - A[q] is less than or equal to each element of A[(q + 1) …. r] .
  - Compute the index q as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays A[p : : (q -1)] and A[(q + 1) …. r] by recursive calls to quicksort.
- **Combine:** Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p…r] is now sorted

Quick Sort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



Quick Sort

The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Worst-case partitioning

- The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements.
- The partitioning costs θ(n) time.
- Recurrence relationship is as follows

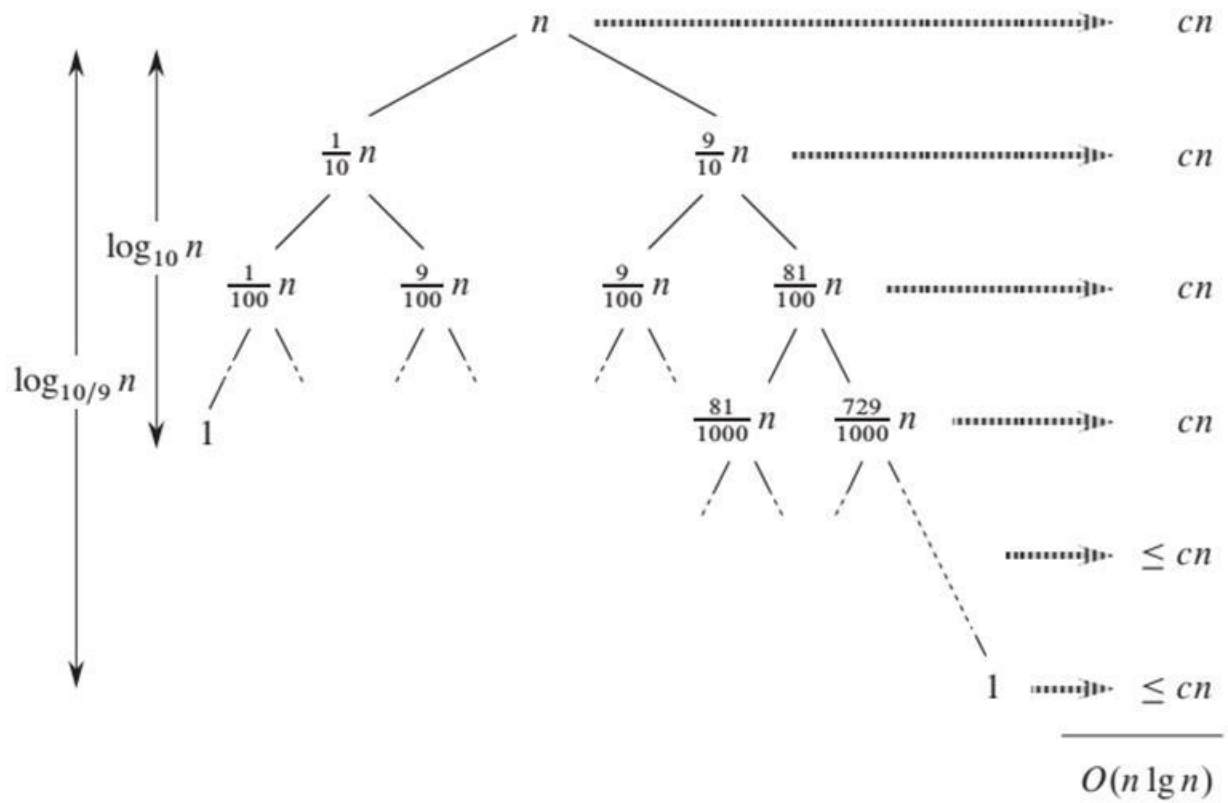
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned}$$

- Above recurrence yields to arthematic series leads to $\theta(n^2)$

Best-case partitioning

- In the most even possible split, PARTITION produces two subproblems, each of size no more than n/2, since one is of size n/2 and one of size (n/2)-1.
- In this case, quicksort runs much faster.
- The recurrence for the running time is then

$$T(n) = 2T(n/2) + \Theta(n) ,$$



Quick Sort Observations

- Choosing a pivot element is crucial in quicksort
- Complexity of the quick sort will always yield O(nlogn) complexity if you choose the median of the given array.
- The catch is if you have to know the median, the elements should be in sorted order!!!
- The average case of quick sort is O(nlogn), using a clear notion of the randomized behavior.

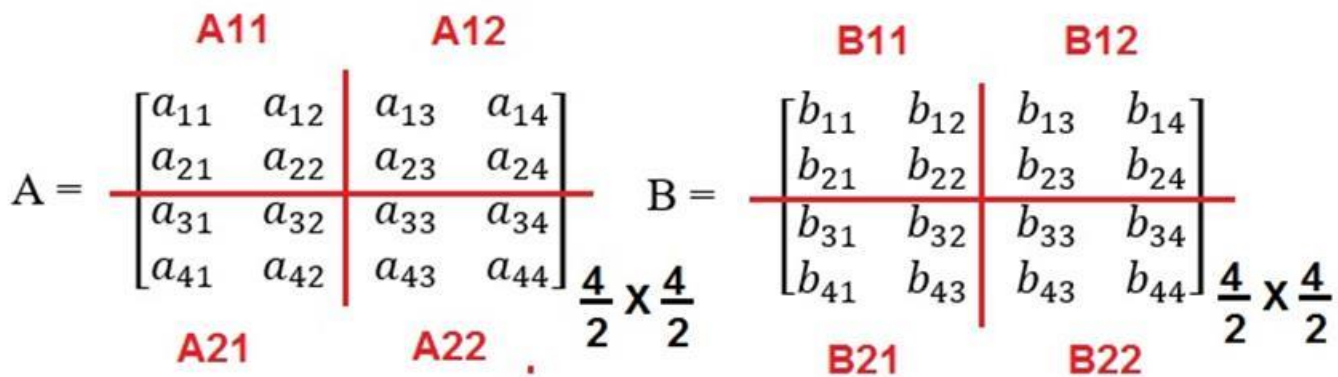
Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

- SQUARE-MATRIX-MULTIPLY procedure takes $\theta(n^3)$ time.
- Strassen's remarkable recursive algorithm for multiplying nxn matrices runs in $\theta(n^{\lg 7})$ time.
- Since lg 7 lies between 2:80 and 2:81, Strassen's algorithm runs in $O(n^{2.81})$ time.
- Asymptotically Strassen's Matrix Multiplication better than the simple SQUARE-MATRIXMULTIPLY procedure.

Matrix Multiplication

- Divide-and-conquer algorithm to compute the matrix product C = A . B, only if n is an exact power of 2 in each of the nxn matrices.
- This assumption is necessary because in each divide step, the matrix divide nxn matrices into four n/2xn/2 matrices, and by assuming that n is an exact power of 2.



Matrix Multiplication

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Matrix Multiplication

- Time Complexity
  - In the base case, when n = 1, we perform just the one scalar multiplication i.e., c1=[a1].[b1] => c1=[a1*b1] => T (1)= θ(1).
  - The recursive case occurs when n > 1.
    - Recursive function called 8 times => T(n')=8T(n/2).
    - number of matrix additions is a constant => complexity => T (n")= θ(n^2).
  -

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) . \end{aligned} \quad \Rightarrow \quad T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases}$$

$$T(n) = \Theta(n^3)$$

Strassen's Method
Matrix Multiplication

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4 \qquad C_{22} = P_5 + P_1 - P_3 - P_7$$

Strassen's Method
Matrix Multiplication

$$T(n) = 7T(n/2) + \Theta(n^2)$$

which describes the running time of Strassen's algorithm. Here, we have $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, and thus $n^{\log_b a} = n^{\log_2 7}$. Rewriting $\log_2 7$ as $\lg 7$ and recalling that $2.80 < \lg 7 < 2.81$, we see that $f(n) = O(n^{\lg 7 - \epsilon})$ for $\epsilon = 0.8$. Again, case 1 applies, and we have the solution $T(n) = \Theta(n^{\lg 7})$.

Strassen's Method
Matrix Multiplication

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$\begin{array}{l}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
\qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \\
\qquad - A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad - A_{12} \cdot B_{22} \\
\qquad\qquad\qquad\qquad - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad\qquad\qquad + A_{12} \cdot B_{21} \ ,
\end{array}$$

$$C_{12} = P_1 + P_2$$

$$\begin{array}{l}
A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
\qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
\hline
A_{11} \cdot B_{12} \qquad\qquad + A_{12} \cdot B_{22} \ ,
\end{array}$$

Strassen's Method
Matrix Multiplication

$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \ , \end{array}$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 \ ,$$

so that C_{22} equals

$$\begin{array}{l} A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\ \qquad - A_{11} \cdot B_{22} \qquad\qquad + A_{11} \cdot B_{12} \\ \qquad\qquad - A_{22} \cdot B_{11} \qquad\qquad - A_{21} \cdot B_{11} \\ - A_{11} \cdot B_{11} \qquad\qquad - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12} \\ \hline \qquad\qquad A_{22} \cdot B_{22} \qquad\qquad + A_{21} \cdot B_{12} \ , \end{array}$$

Strassen's Method
Matrix Multiplication

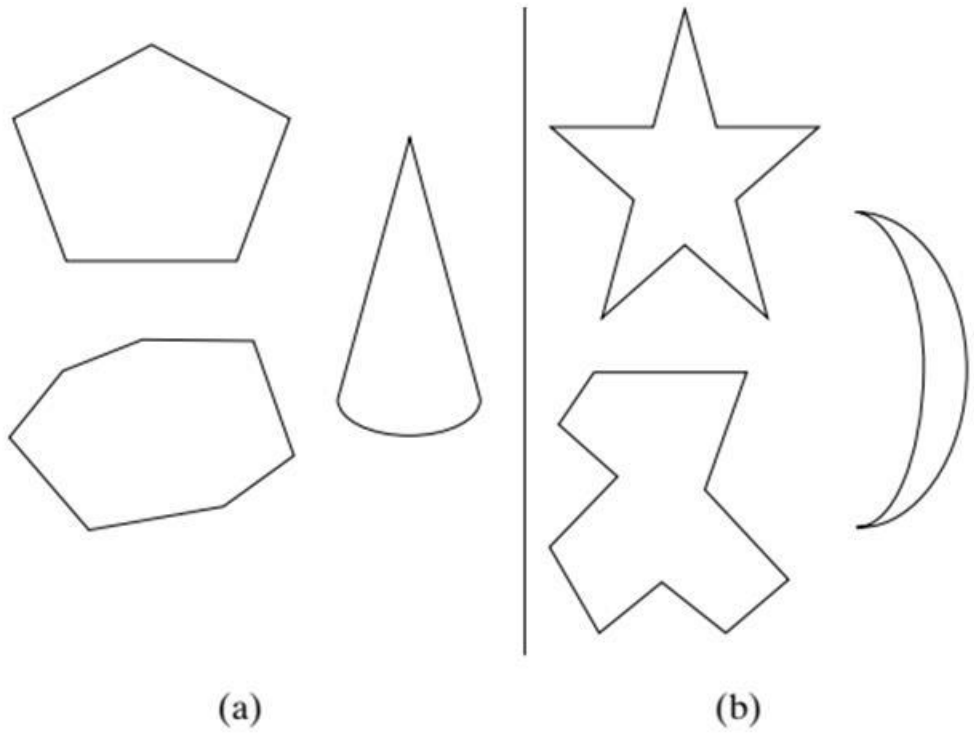
1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation . This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Convex Hull Problem

- Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important—problems in computational geometry.
- Several such applications are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given.
- For example,
  - In computer animation, replacing objects by their convex hulls speeds up collision detection;
  - In path planning for Mars mission rovers.
  - In computing accessibility maps produced from satellite images by Geographic Information Systems.
  - For detecting outliers by some statistical techniques.
  - For computing a diameter of a set of points,
    - which is the largest distance between two of the points, needs the set's convex hull to find the largest distance between two of its extreme points.
  - For solving many optimization problems, because their extreme points provide a limited set of solution candidates.

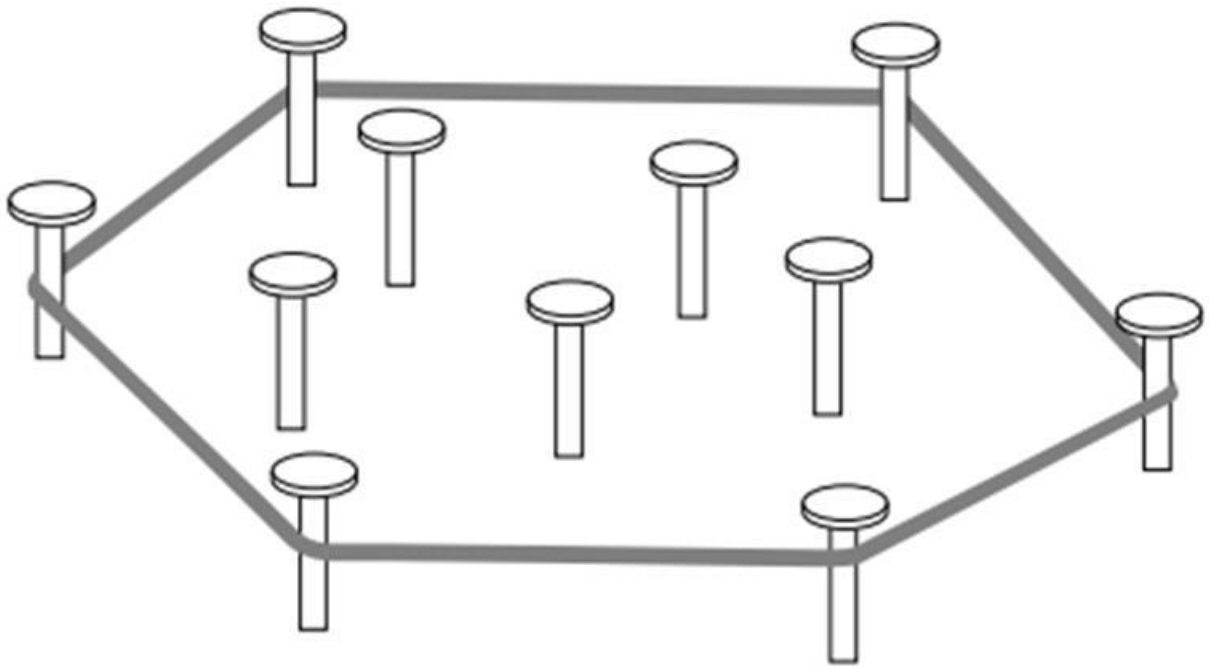
Convex Hull Problem

- **DEFINITION-Convex:** A set of points (finite or infinite) in the plane is called ***convex*** if for any two points *p* and *q* in the set, the entire line segment with the endpoints at *p* and *q* belongs to the set.
- **DEFINITION-Convex hull:** The ***convex hull*** of a set *S* of points is the smallest convex set containing *S*. (The "smallest" requirement means that the convex hull of *S* must be a subset of any convex set containing *S*.)

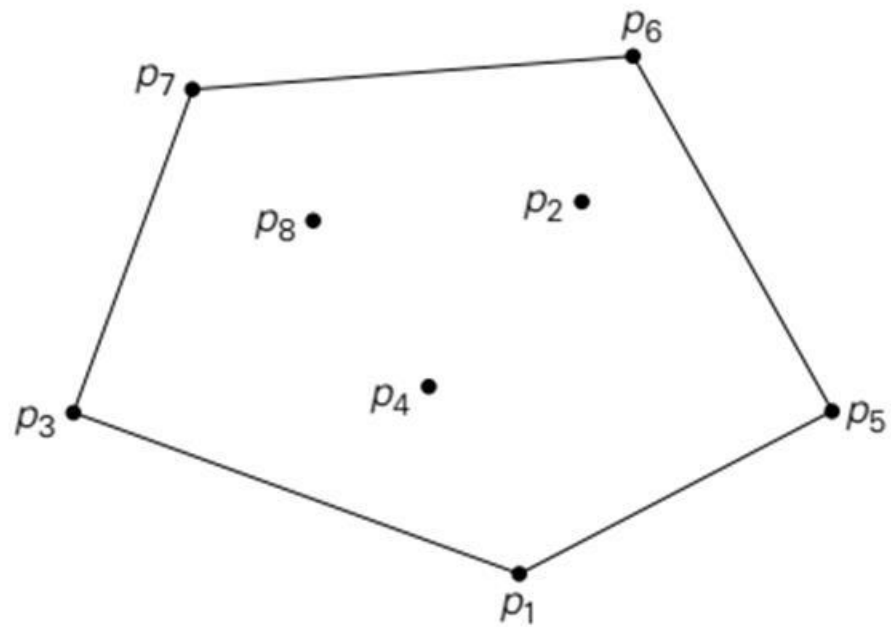


(a) Convex sets. (b) Sets that are not convex.

Convex Hull Problem



Rubber-band interpretation of the convex hull.



The convex hull for this set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, and p_3.

Convex Hull Problem

- Brute Force Algorithm
  - A line segment connecting two points p_i and p_j of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points.
  - Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.
  - First, the straight line through two points $(x_1, y_1), (x_2, y_2)$ in the coordinate plane can be defined by the equation

$$\boldsymbol{ax + by = c},$$

where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.

Convex Hull Problem

- Brute Force Algorithm
  - Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.)
  - Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.

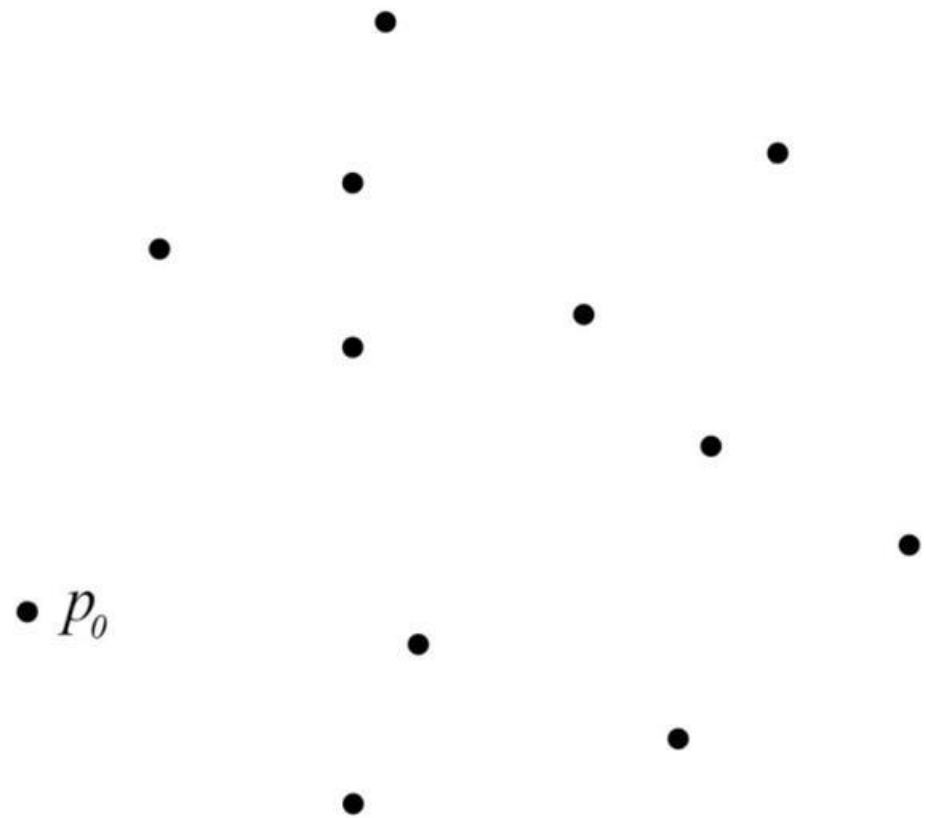
Types of Algorithms

- Graham Scan (1972) - Sort by angle measures
- Jarvis March (1973, Gift Wrapping) - Wrap the set
- Quickhull (1977/1978) - Divide and conquer by splitting the set using the farthest available points
- Other Algorithms - Chan’s Algorithm (1996) is one of the current best 2D algorithms. Combination of multiple simpler algorithms

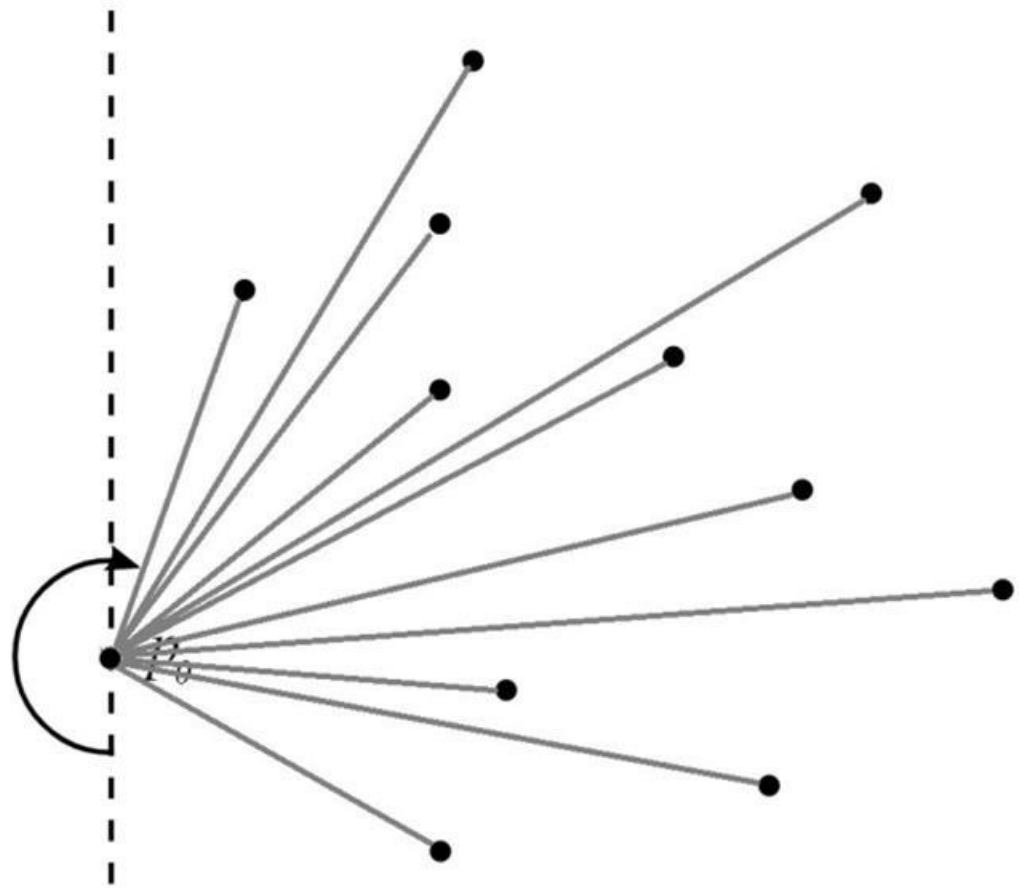
Algorithm - Jarvis March (2D)

- Also known as the Gift Wrapping Algorithm.
- One of the first modern algorithms for finding Convex Hulls, published in 1973.
- Designed by R. (Ray) A. Jarvis - Australian National University
- Algorithm Outline:
  - Take as input a set of points S, and begin with an empty ordered hull set H = ∅.
  - Take the leftmost point in S and put it in the set H. (or Lowest Point)
  - Pick a second point in S and draw a line between the two points.
  - Iterate through the remaining points in S until you find the leftmost line. All points in the set S are to the right of the line. Put the point in the set H.
  - Repeat the last two steps until you reach the starting hull point.

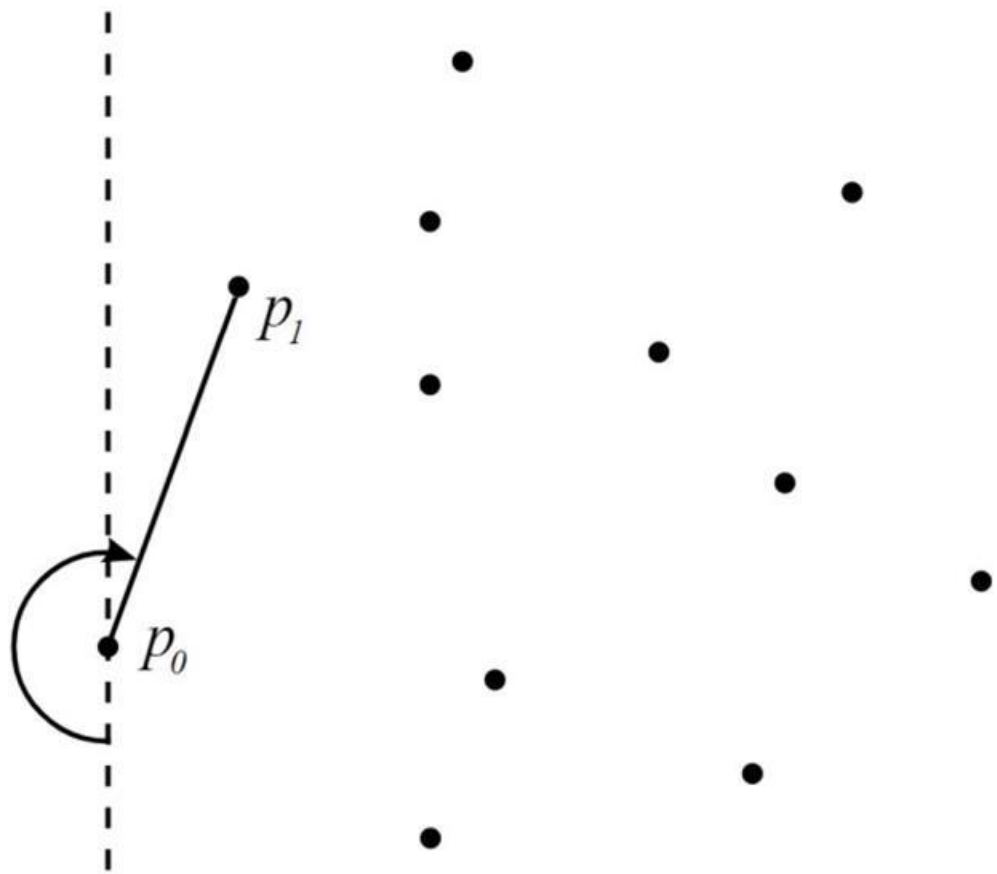
Algorithm - Jarvis March (2D)
Step 1: Pick the leftmost point



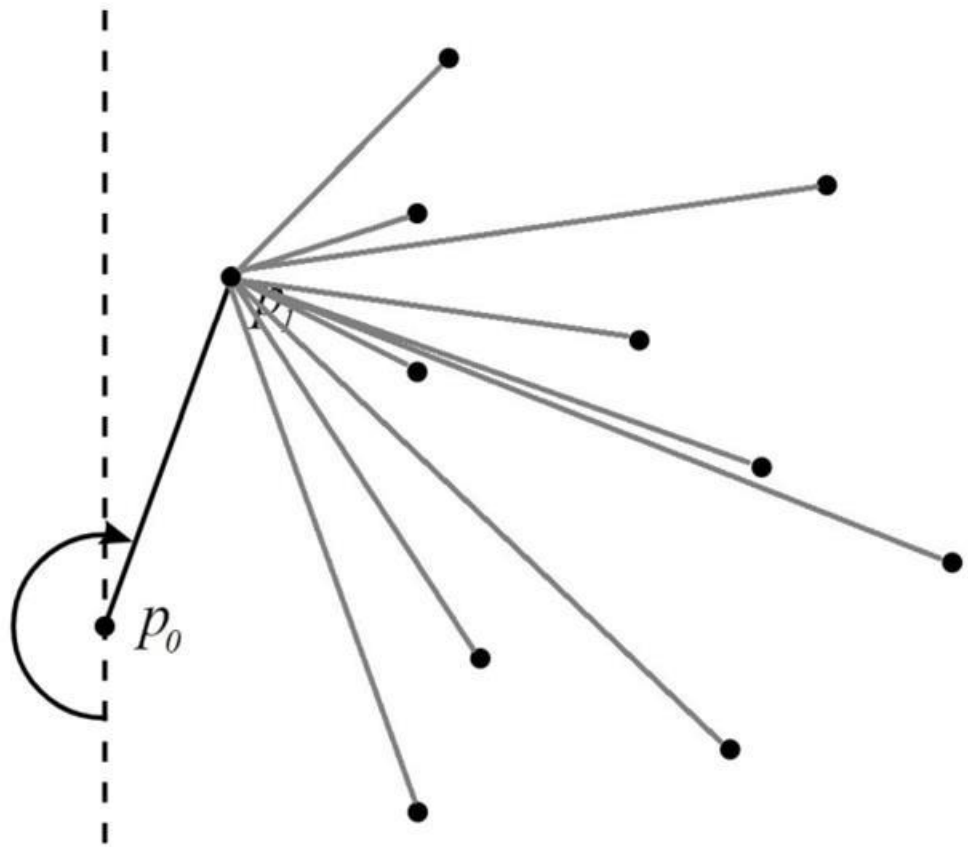
Algorithm - Jarvis March (2D)
Step 2: Pick the “leftmost line”



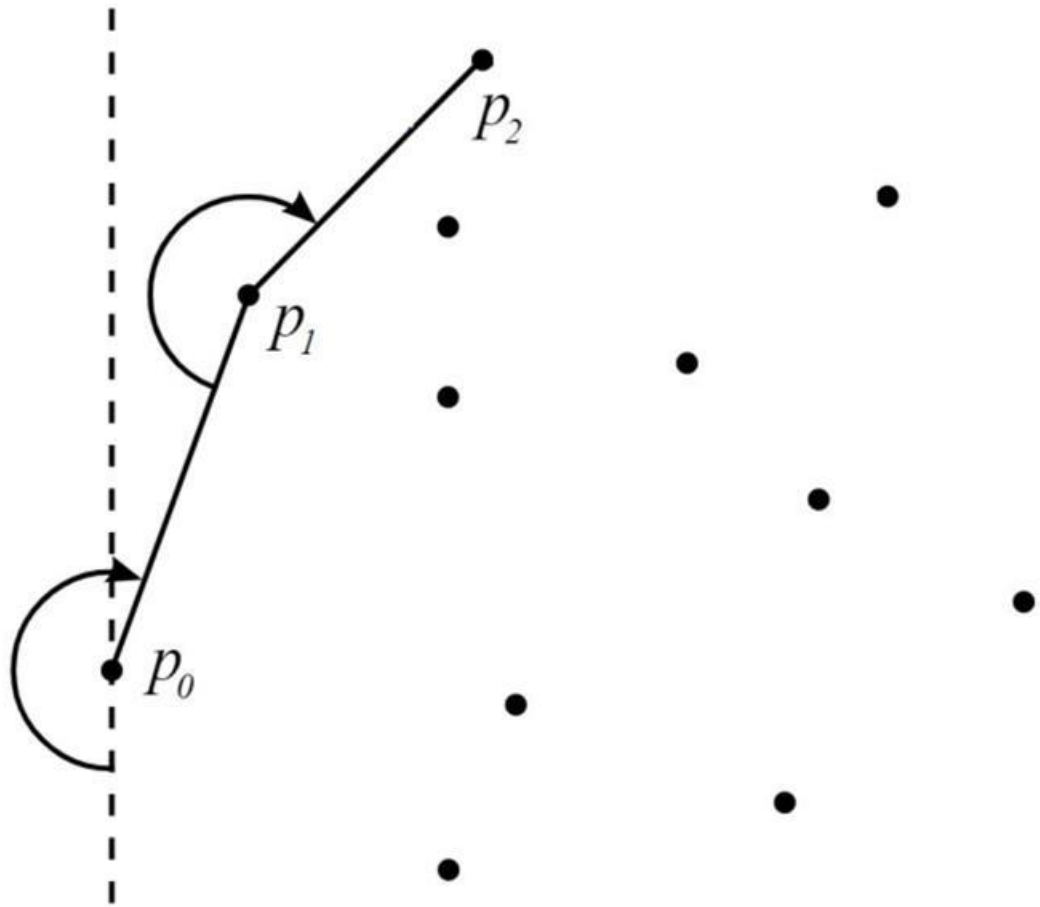
Algorithm - Jarvis March (2D)
Step 2: Pick the “leftmost line”



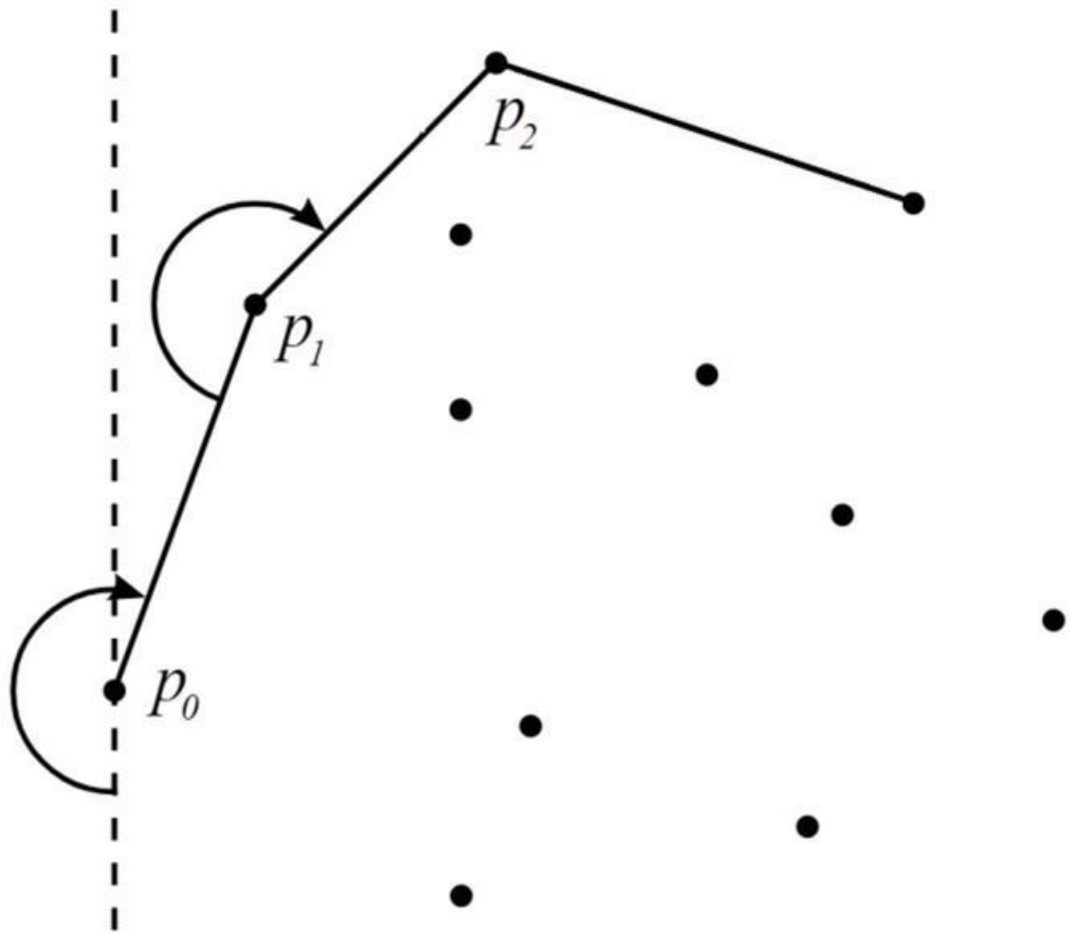
Algorithm - Jarvis March (2D)
Step 3: Repeat



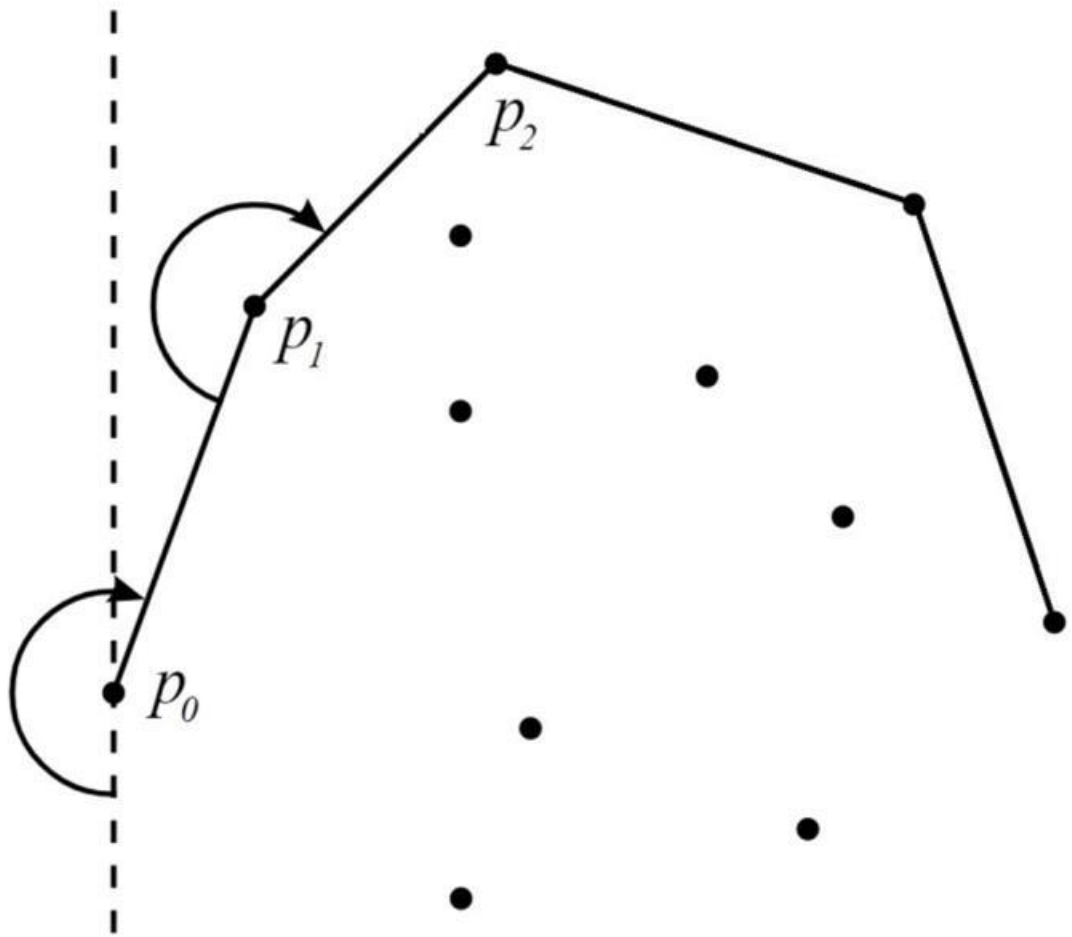
Algorithm - Jarvis March (2D)
Step 3: Repeat



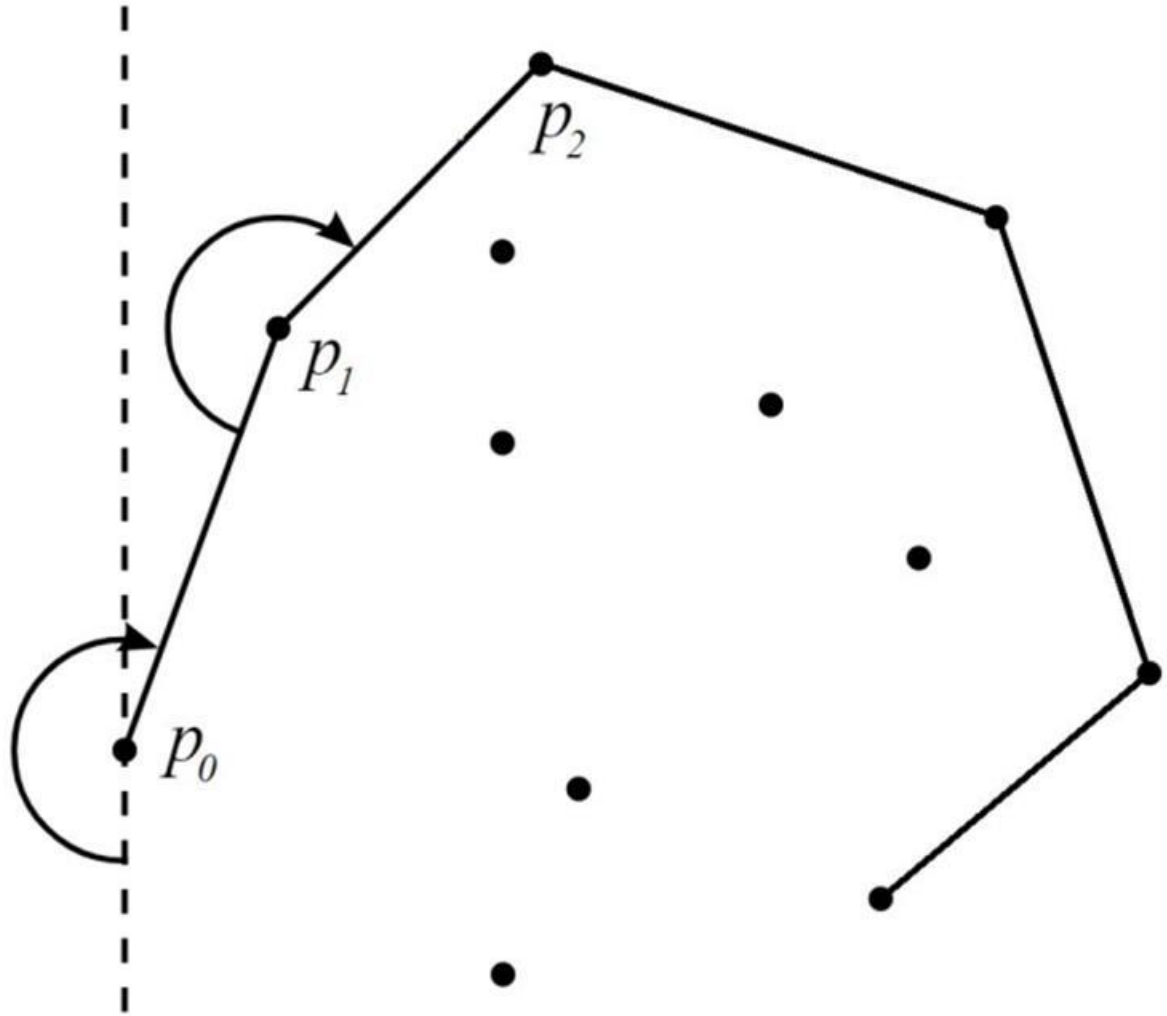
Algorithm - Jarvis March (2D)
Step 3: Repeat



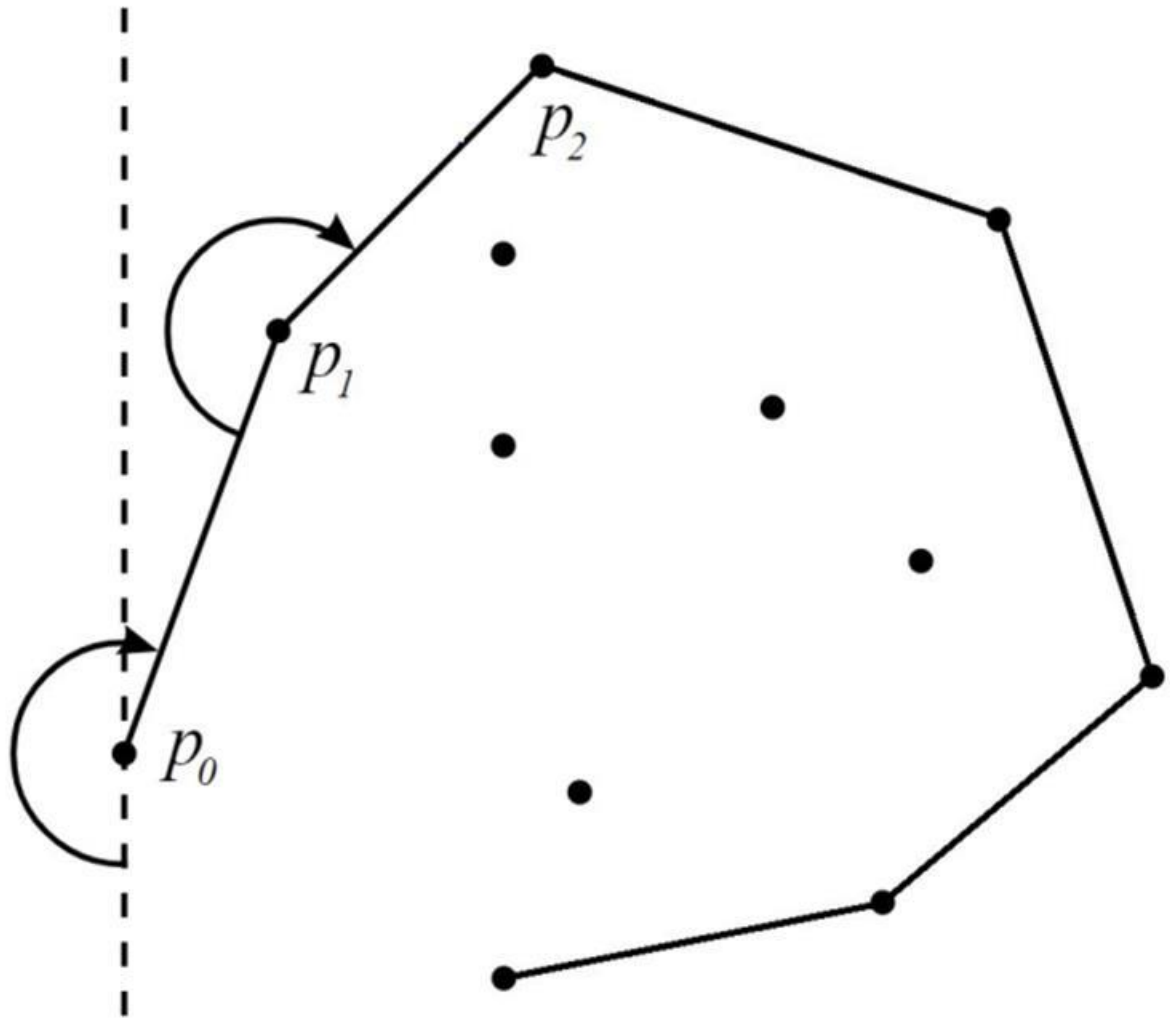
Algorithm - Jarvis March (2D)
Step 3: Repeat



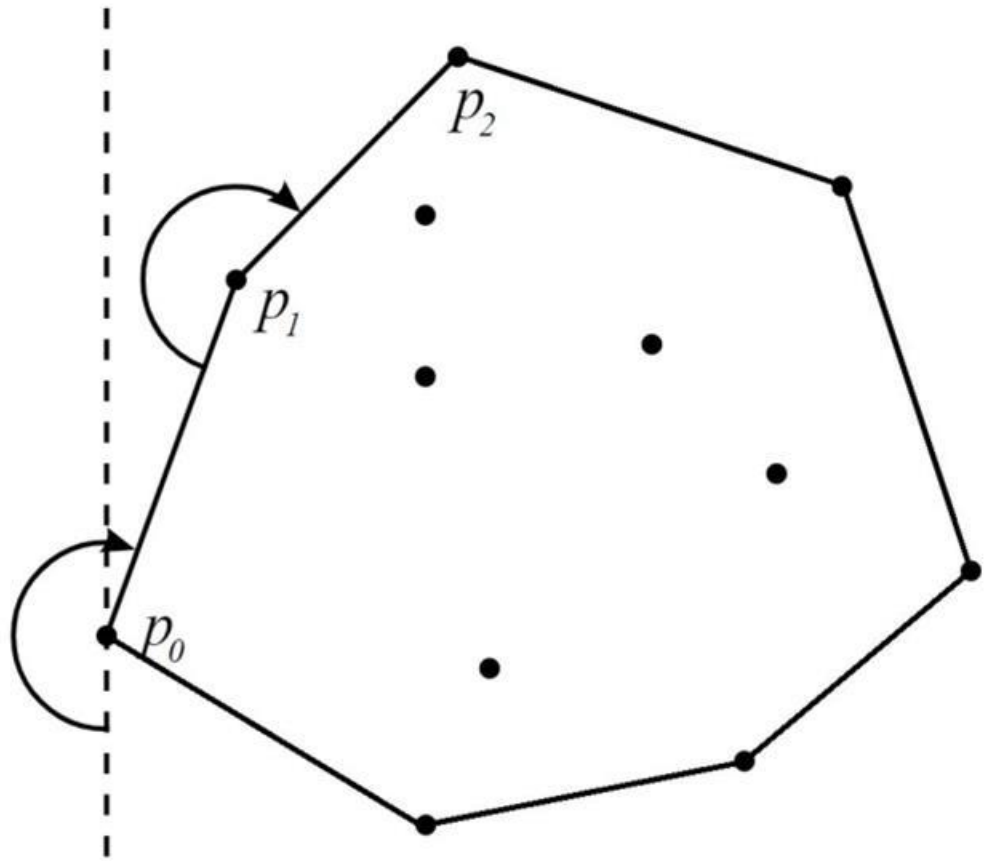
Algorithm - Jarvis March (2D)
Step 3: Repeat



Algorithm - Jarvis March (2D)
Step 3: Repeat



Algorithm - Jarvis March (2D)
Step 3: Repeat until you add p0



Algorithm - Jarvis March (2D)
Step 3: Repeat until you add p0

- Time Complexity:
  - Iterate through the set $|S|=n$, as many times as you have hull points $|H|=h$. Therefore, the time complexity is $O(nh)$.
  - Worst Case Complexity: If $|H|=n$, then $O(n^2)$.
  - Space Complexity: $O(n)$

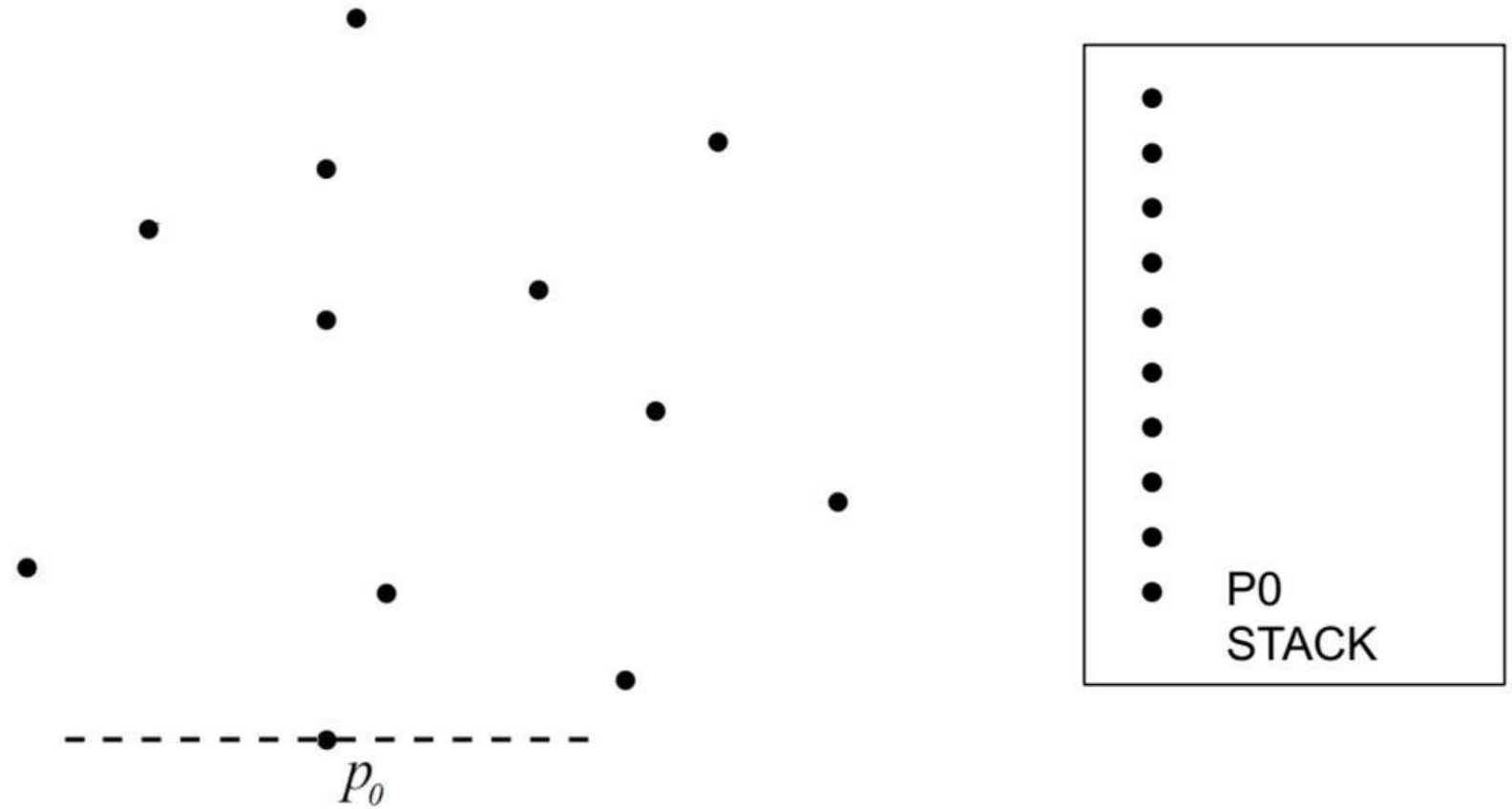
Algorithm - Graham Scan

- Published in 1972 by Ronald Graham, one year before the Jarvis March.
- More complex than the Jarvis March, but has a much better worst case time complexity.
- Average Case Time Complexity = O(n log n) and can be improved even further by pre-sorting to get O(n) complexity.

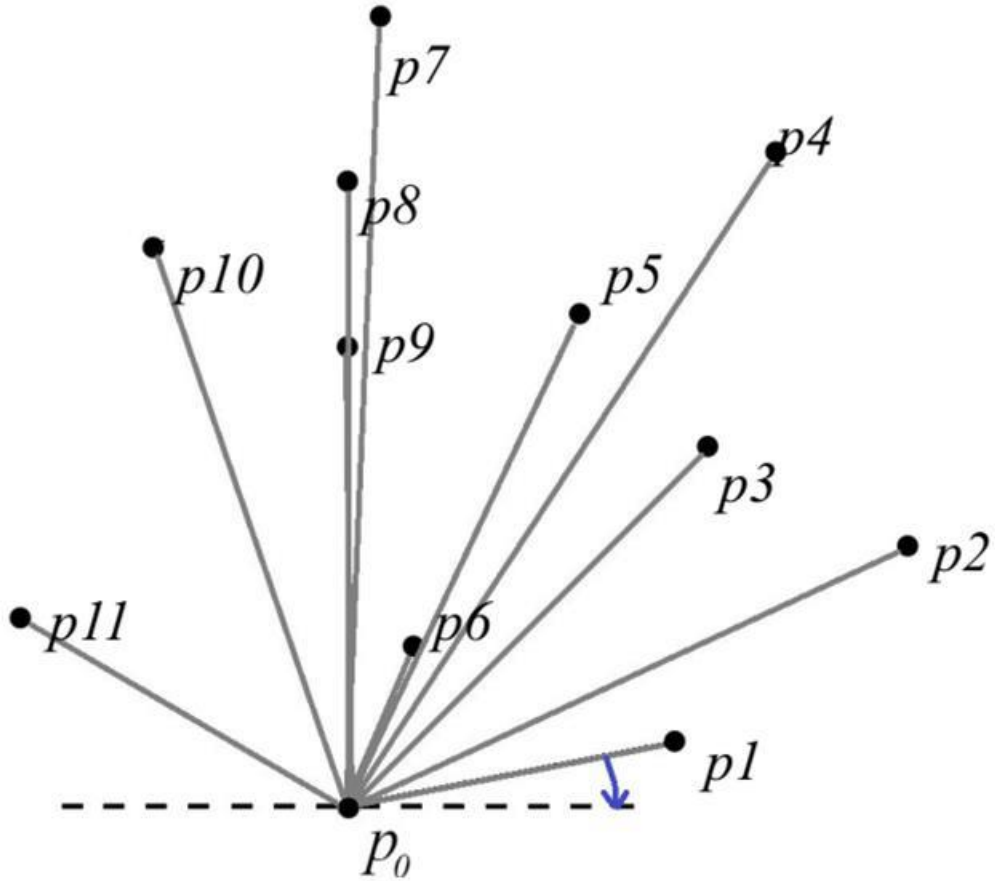
Algorithm - Graham Scan

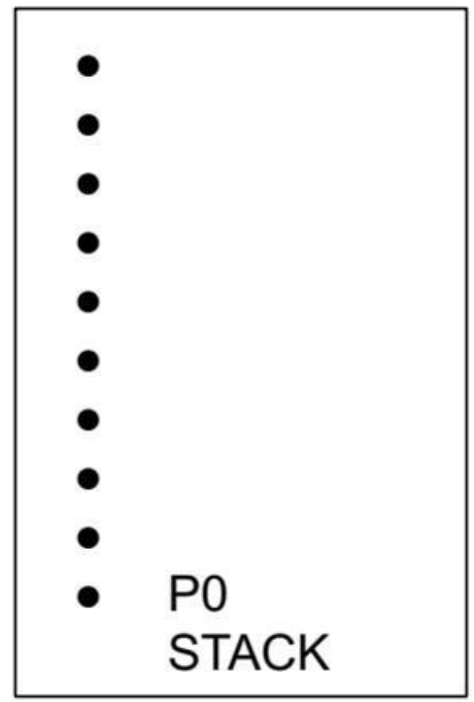
- Algorithm Outline:
  - Take as input a set of points S, and begin with an empty stack H.
  - Take the lowest point in S, call it p0, and add it to the stack H.
  - Take a horizontal line L through p0 and sort the set S by the angle from L.
  - Push points onto the stack if they make a left turn from the previous point.
  - Pop the previous point from the stack if the next point would make a right turn.

Algorithm - Graham Scan
Step 1: Pick the lowest point

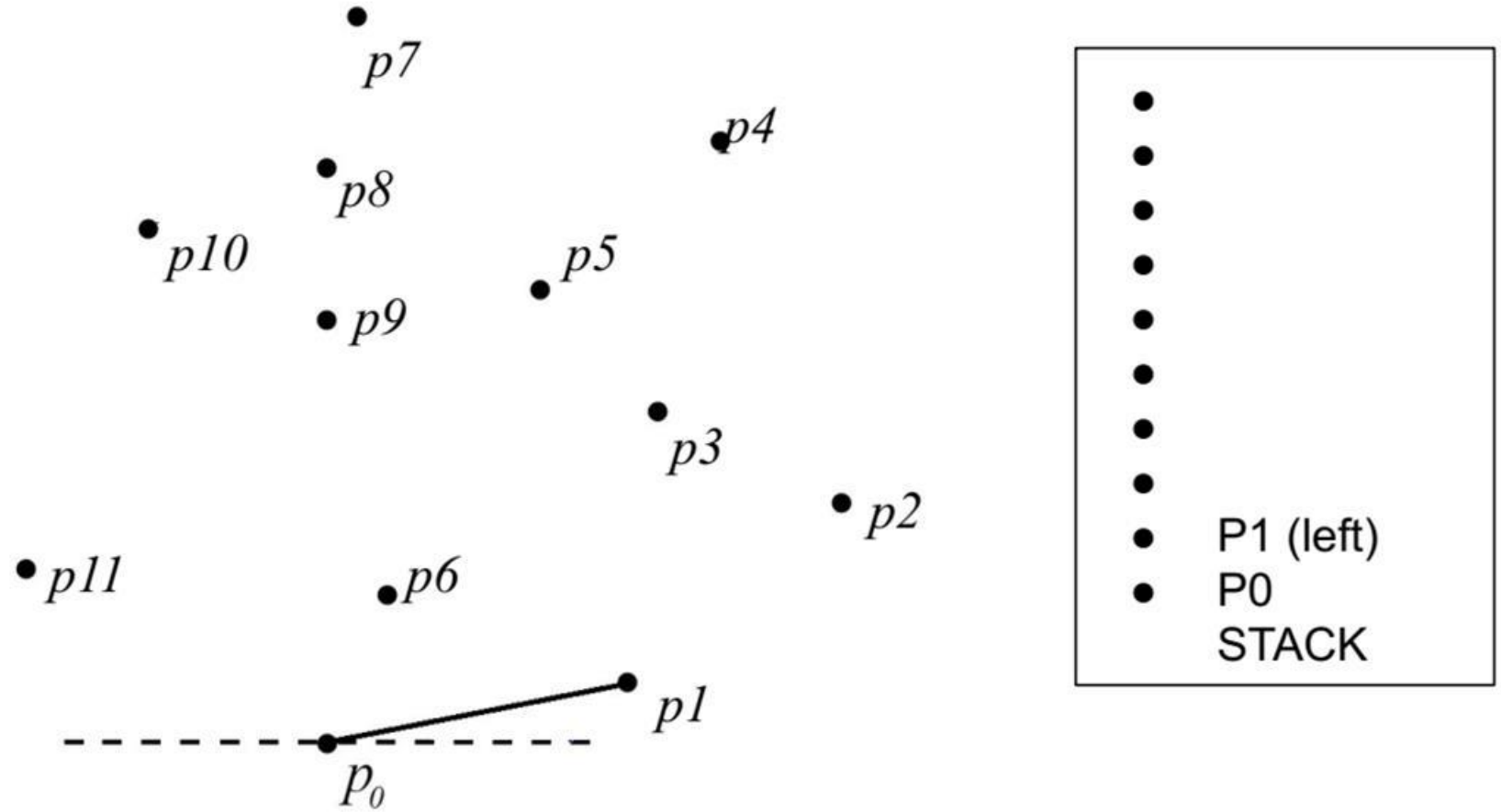


Algorithm - Graham Scan
Step 2: Sort the points by angles

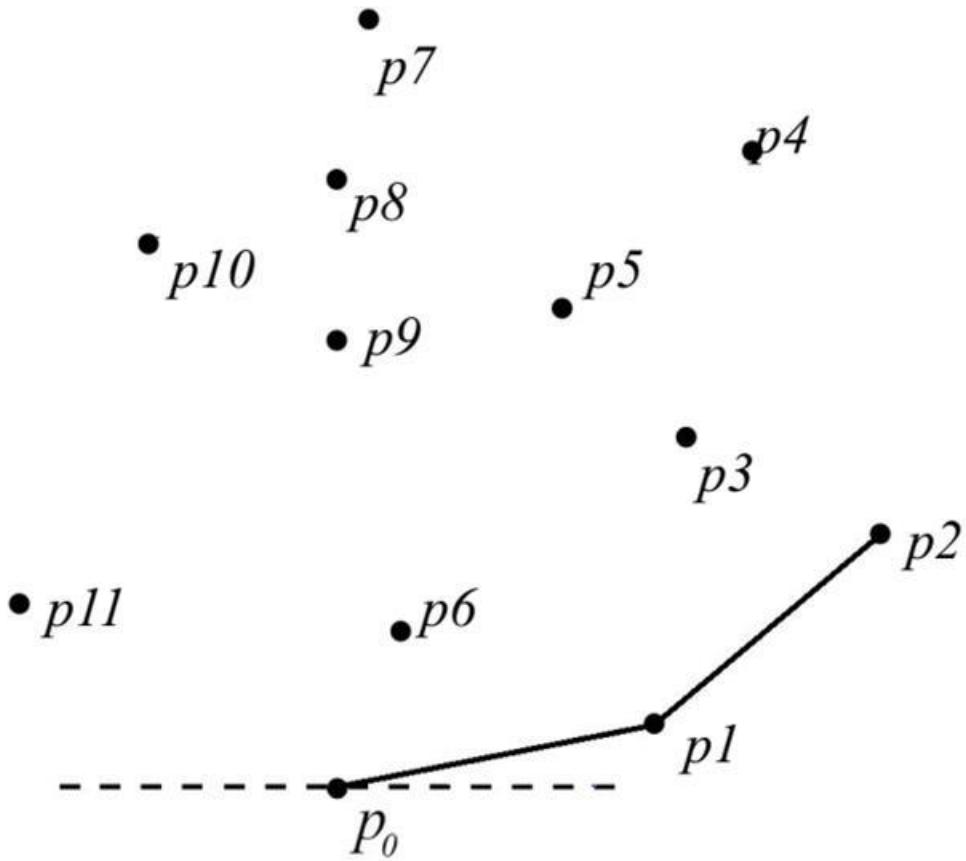


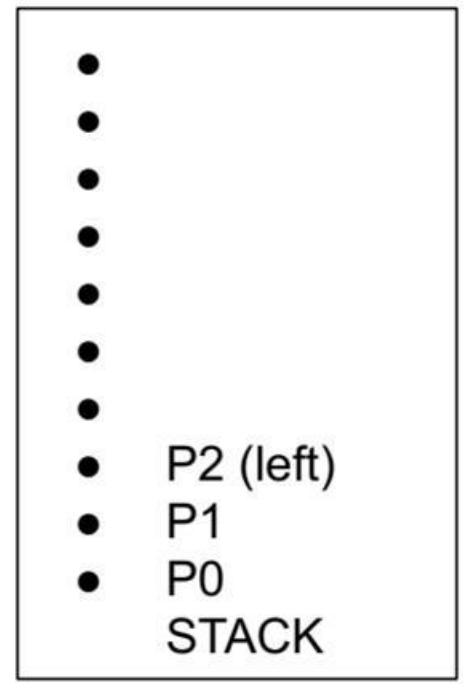


Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn

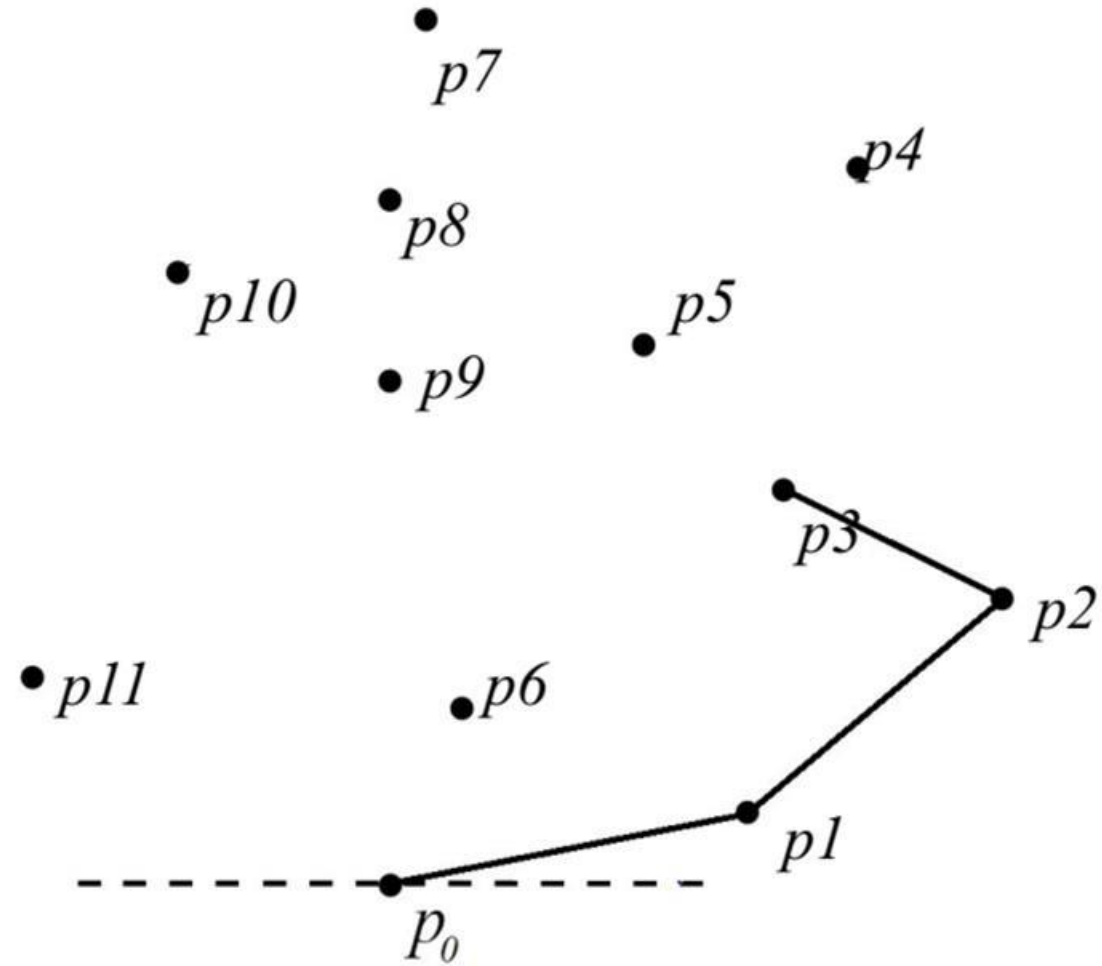


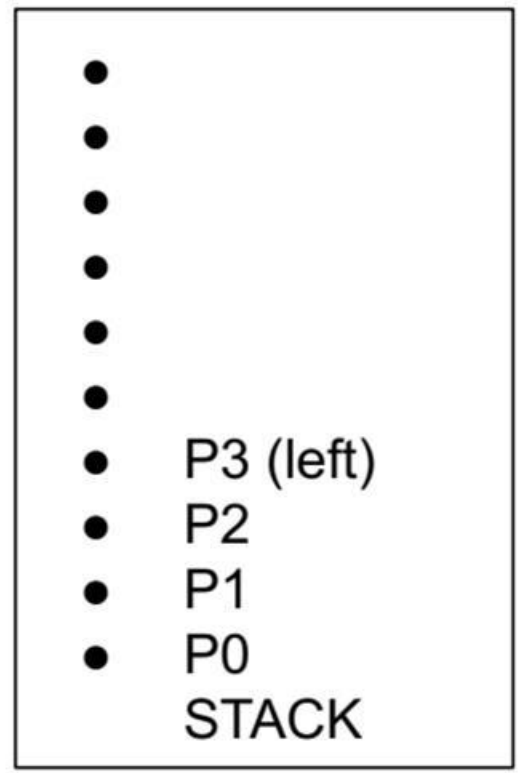
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



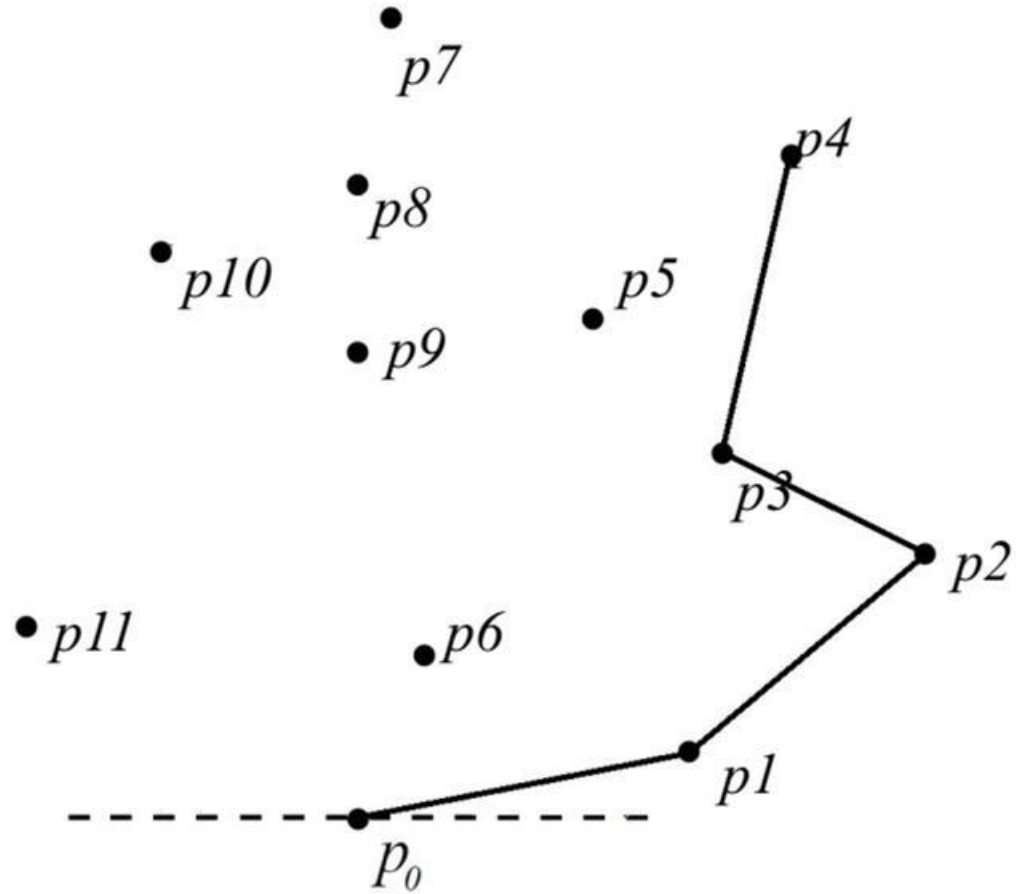


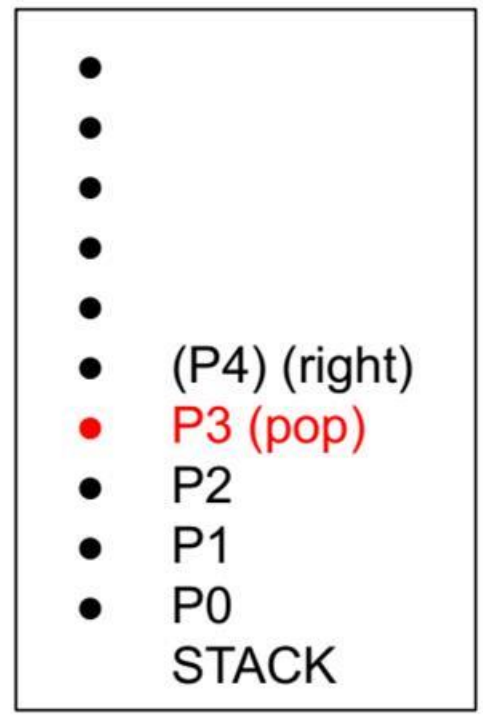
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



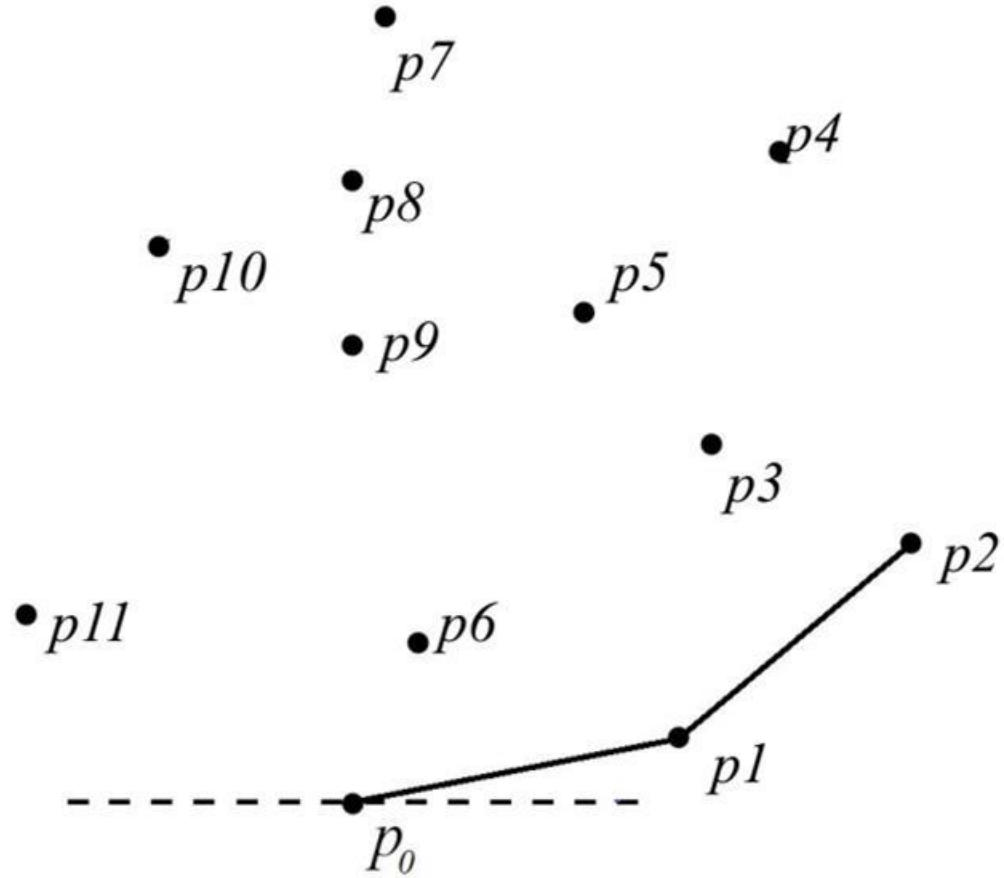


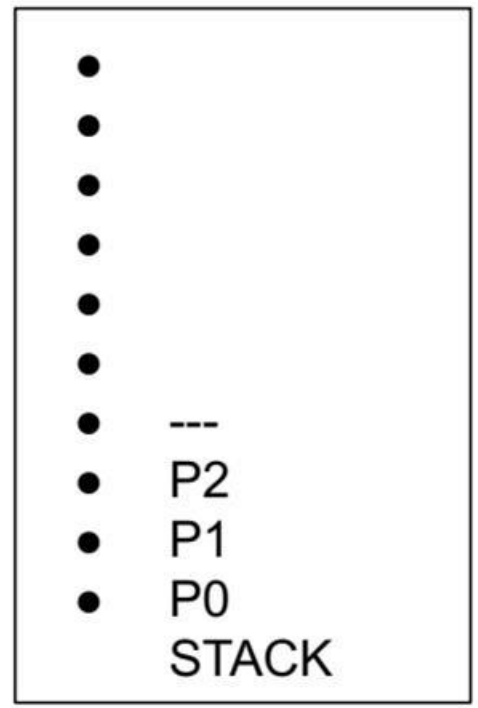
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



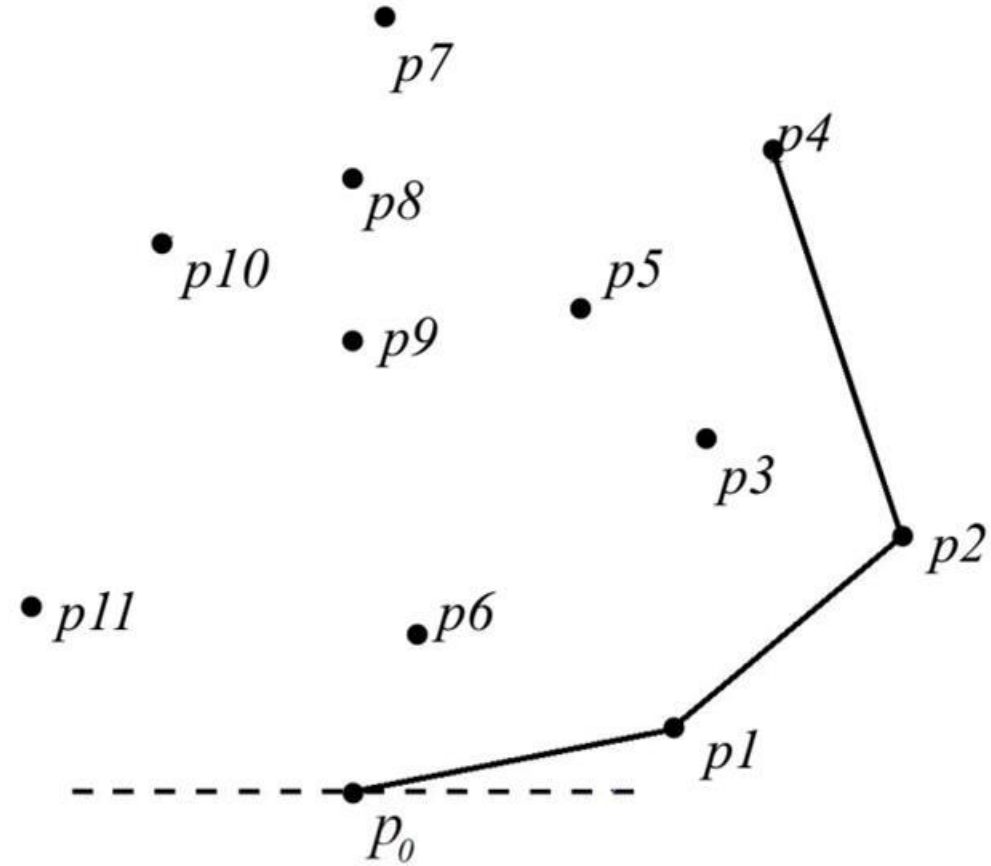


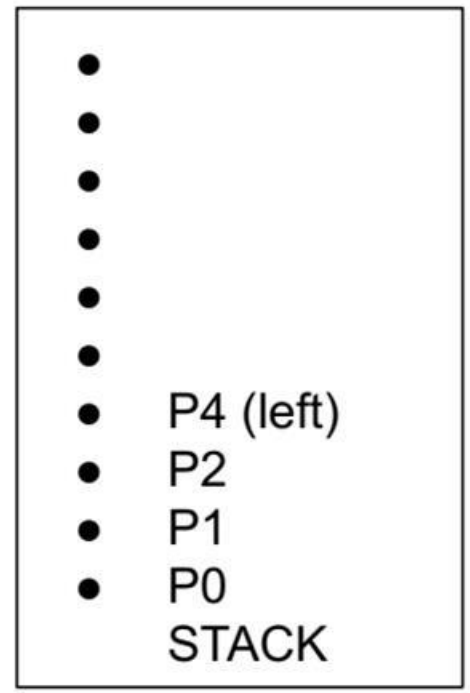
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



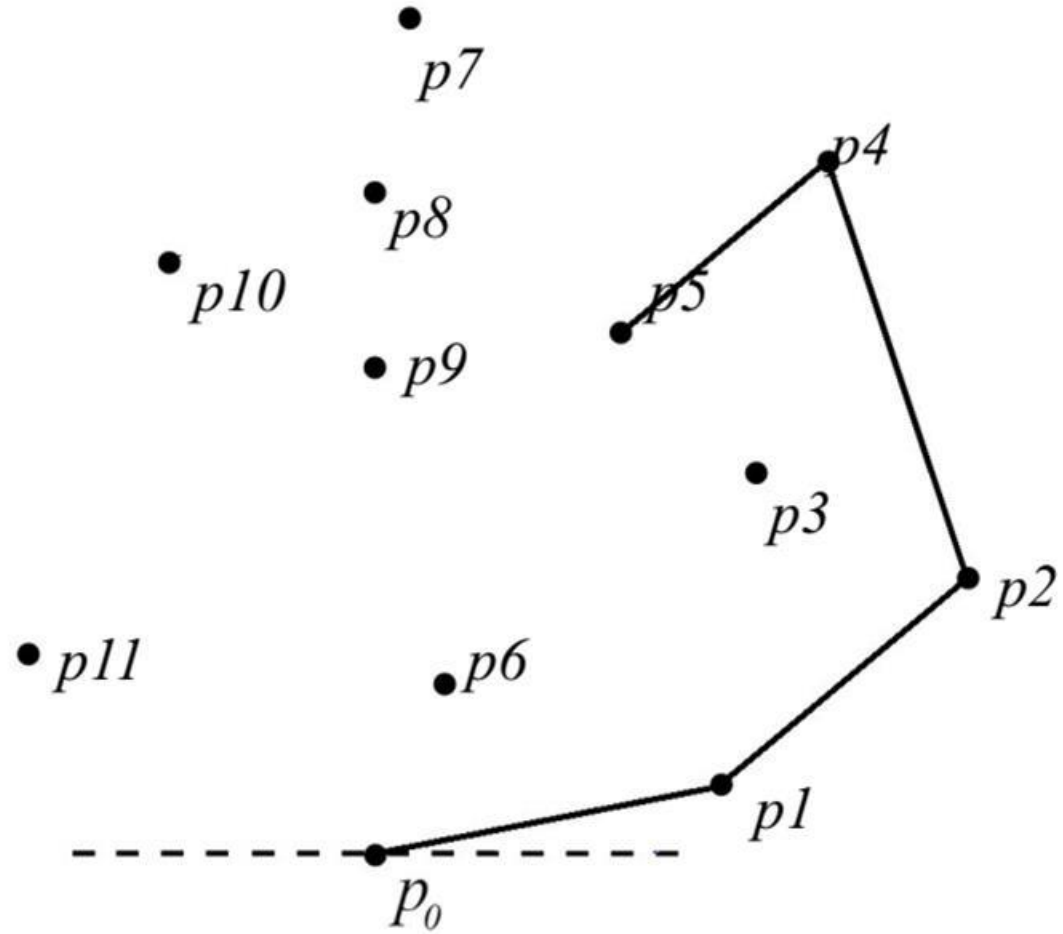


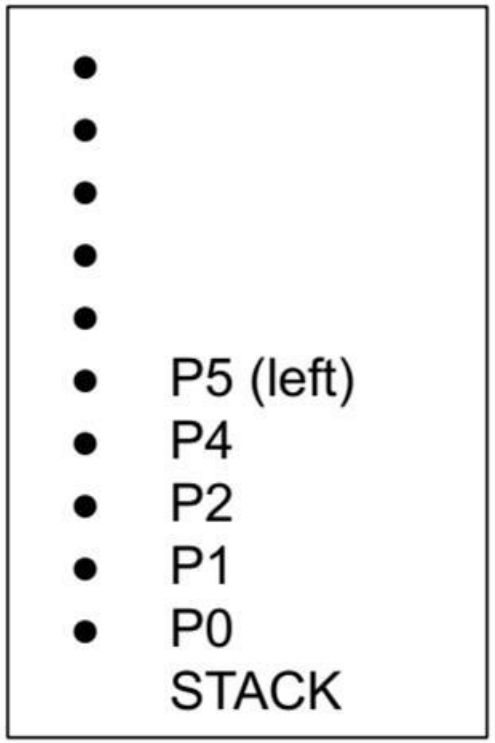
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



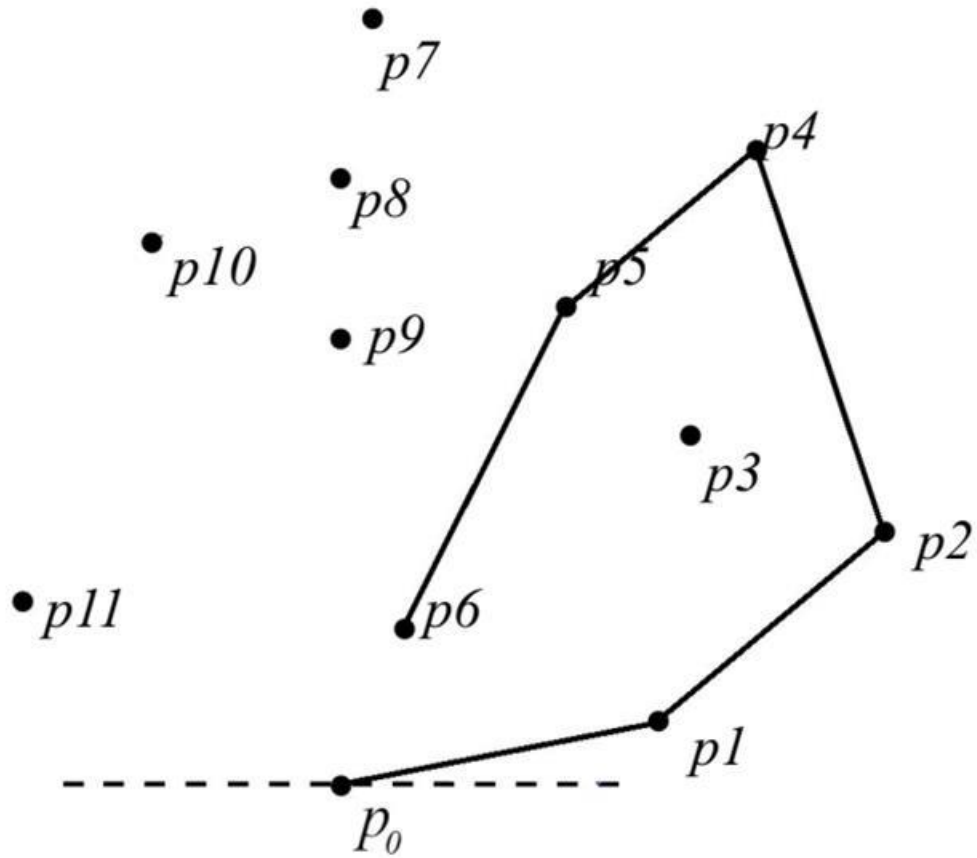


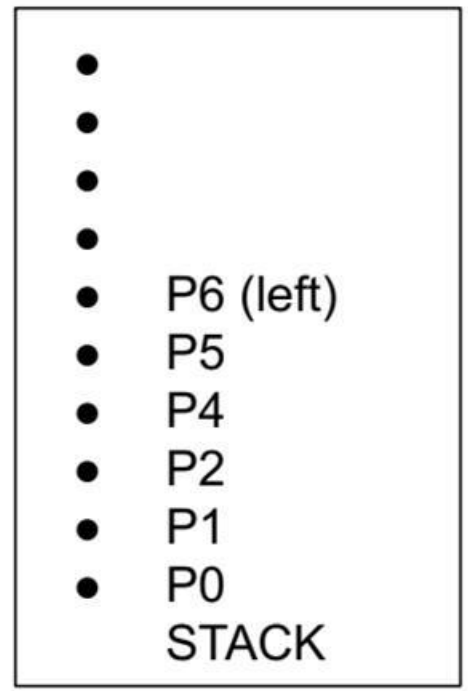
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



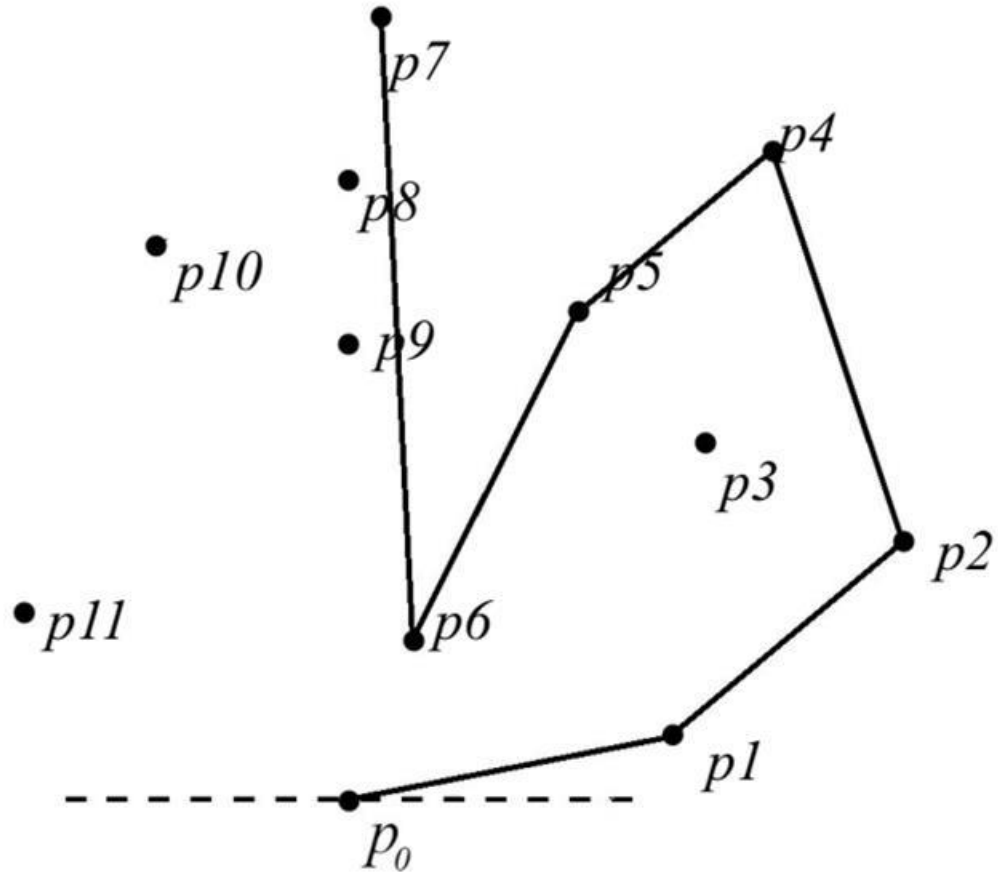


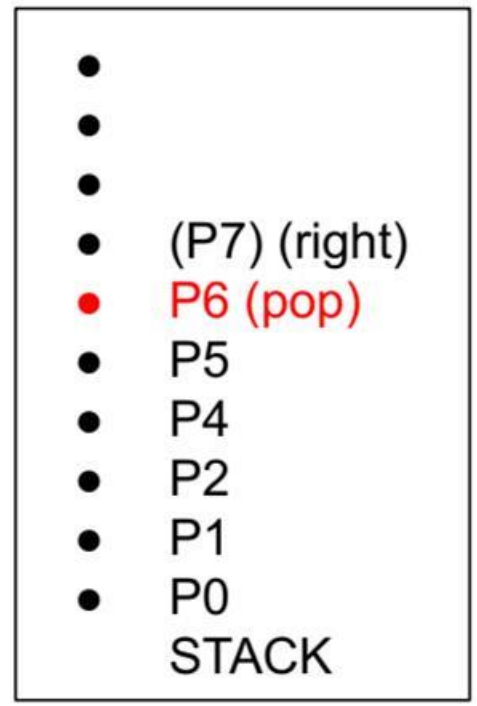
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



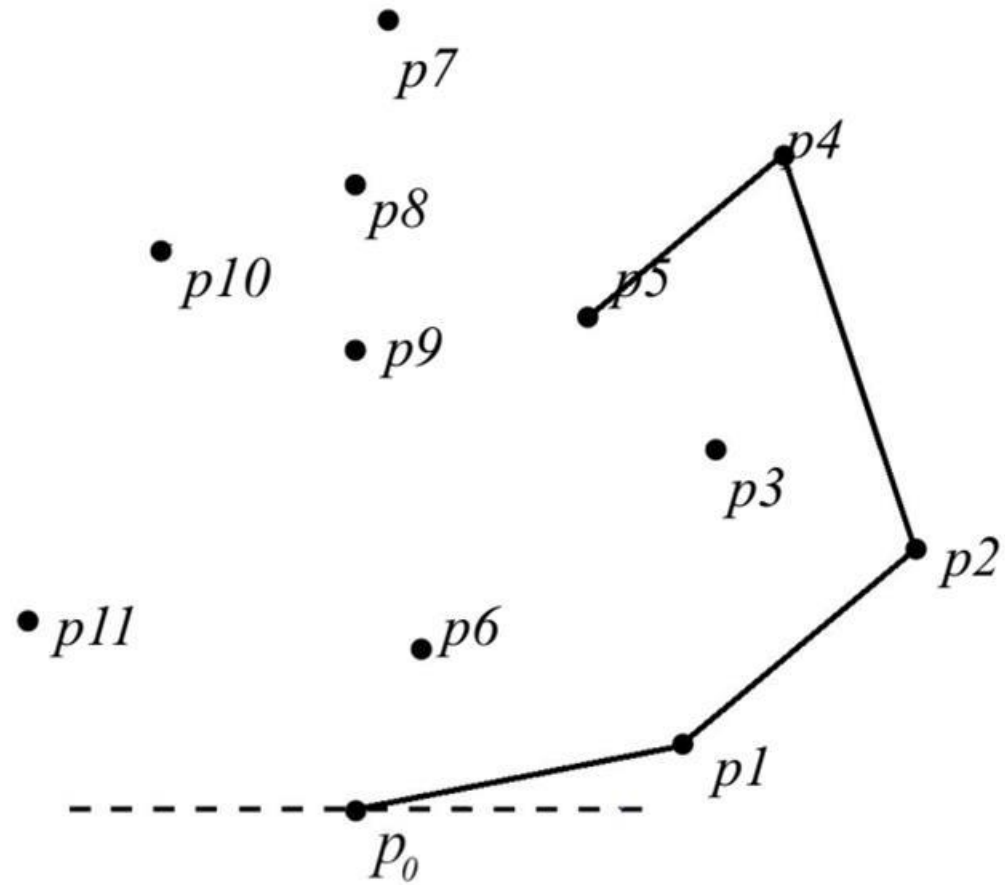


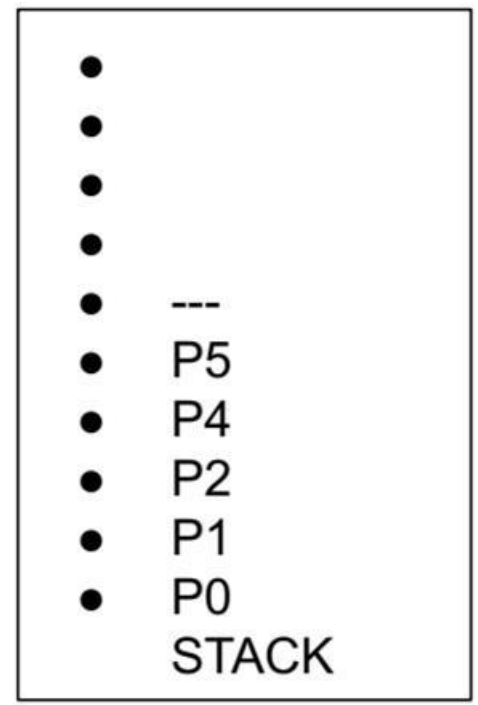
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



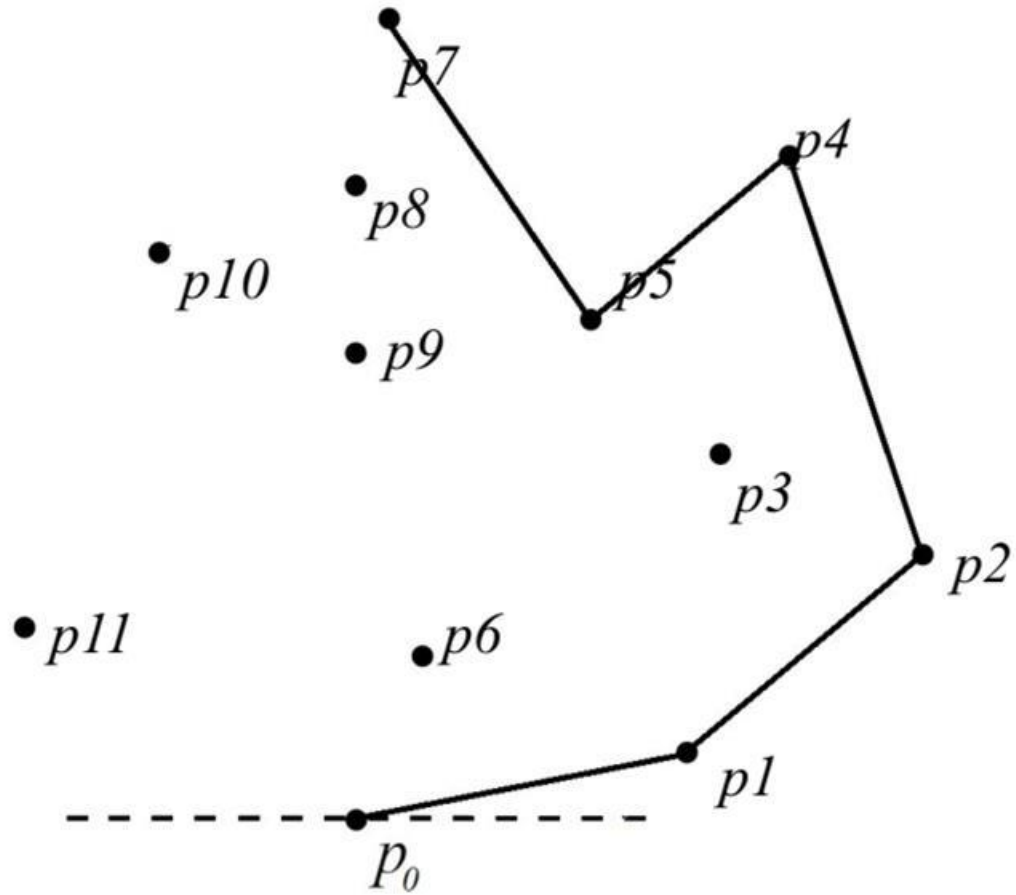


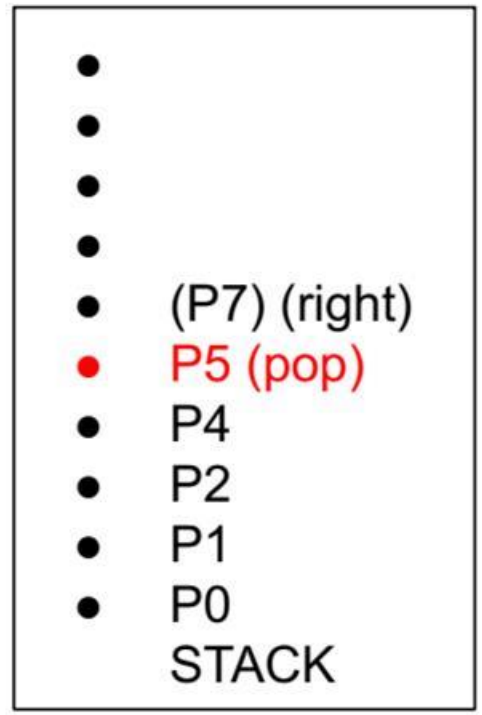
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



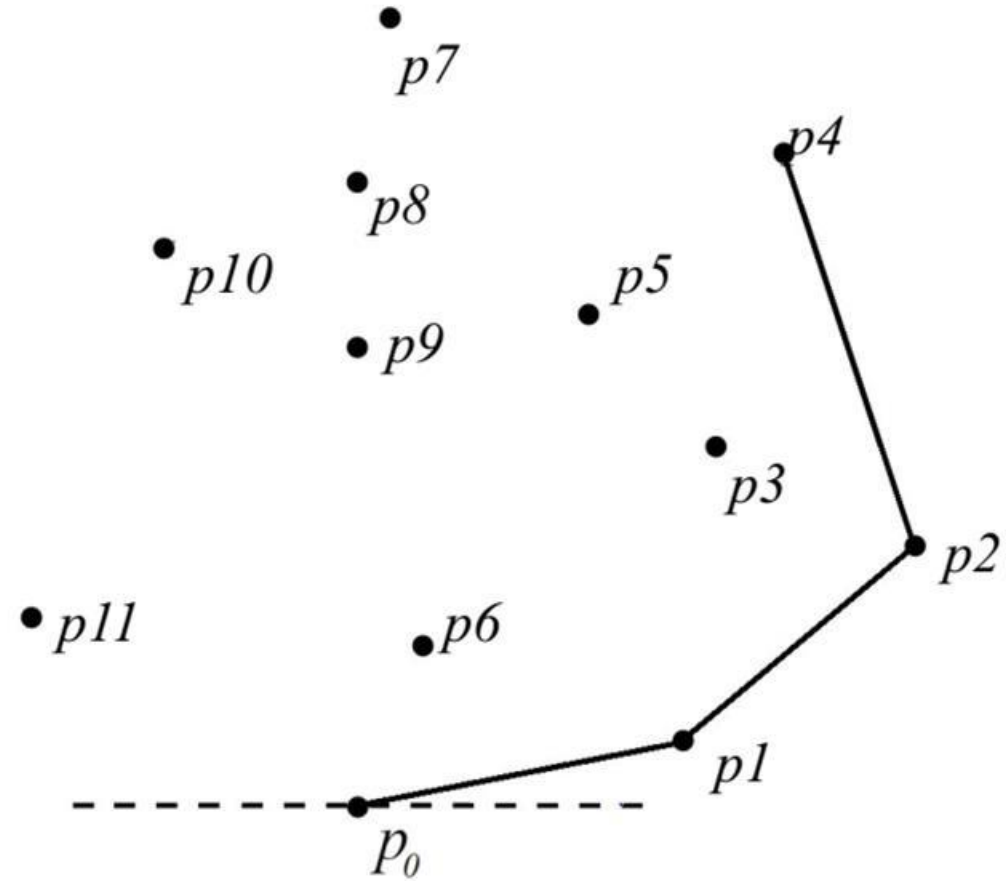


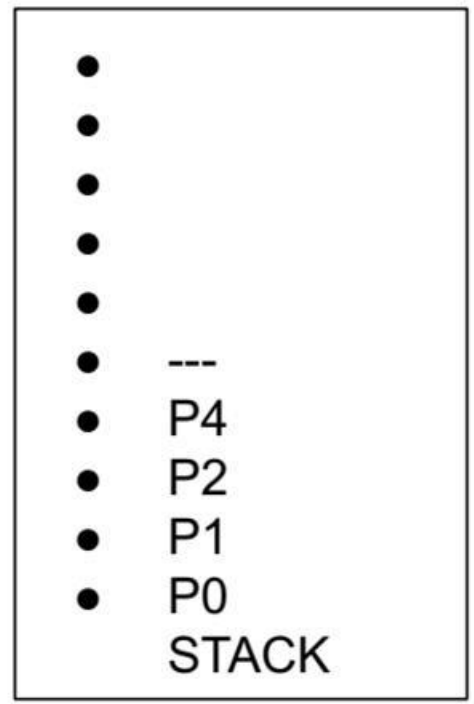
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn



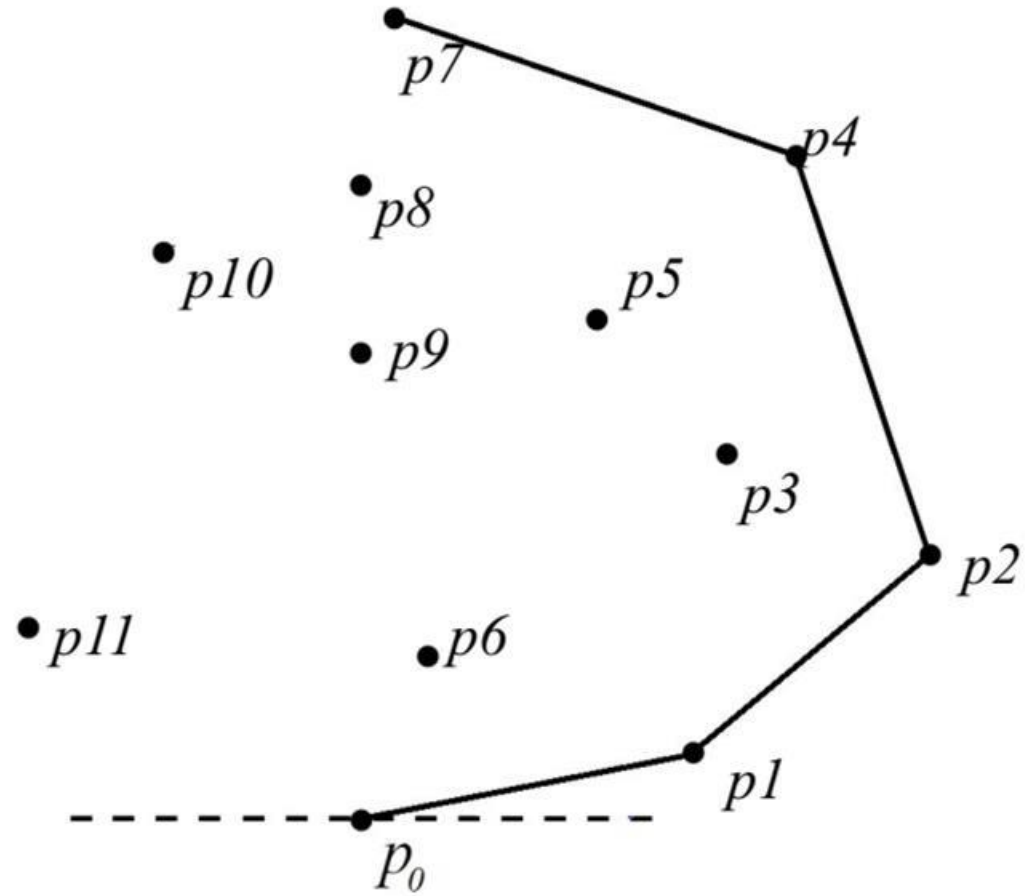


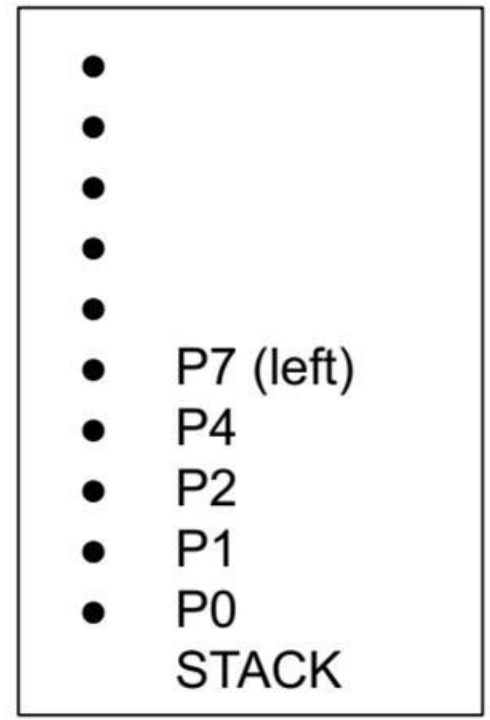
Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn





Algorithm - Graham Scan
Step 3: Push points onto the stack Pop on a right turn

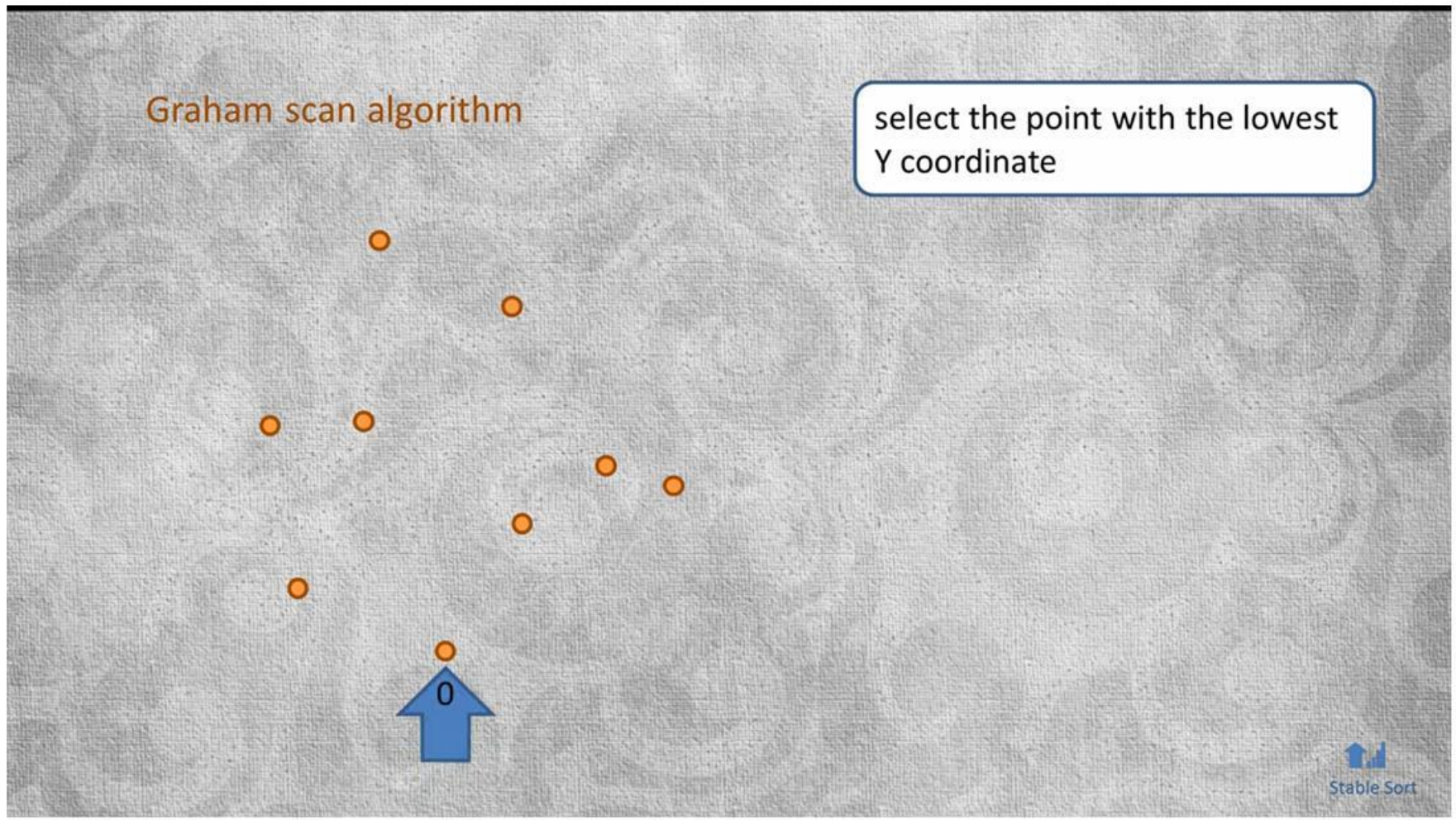




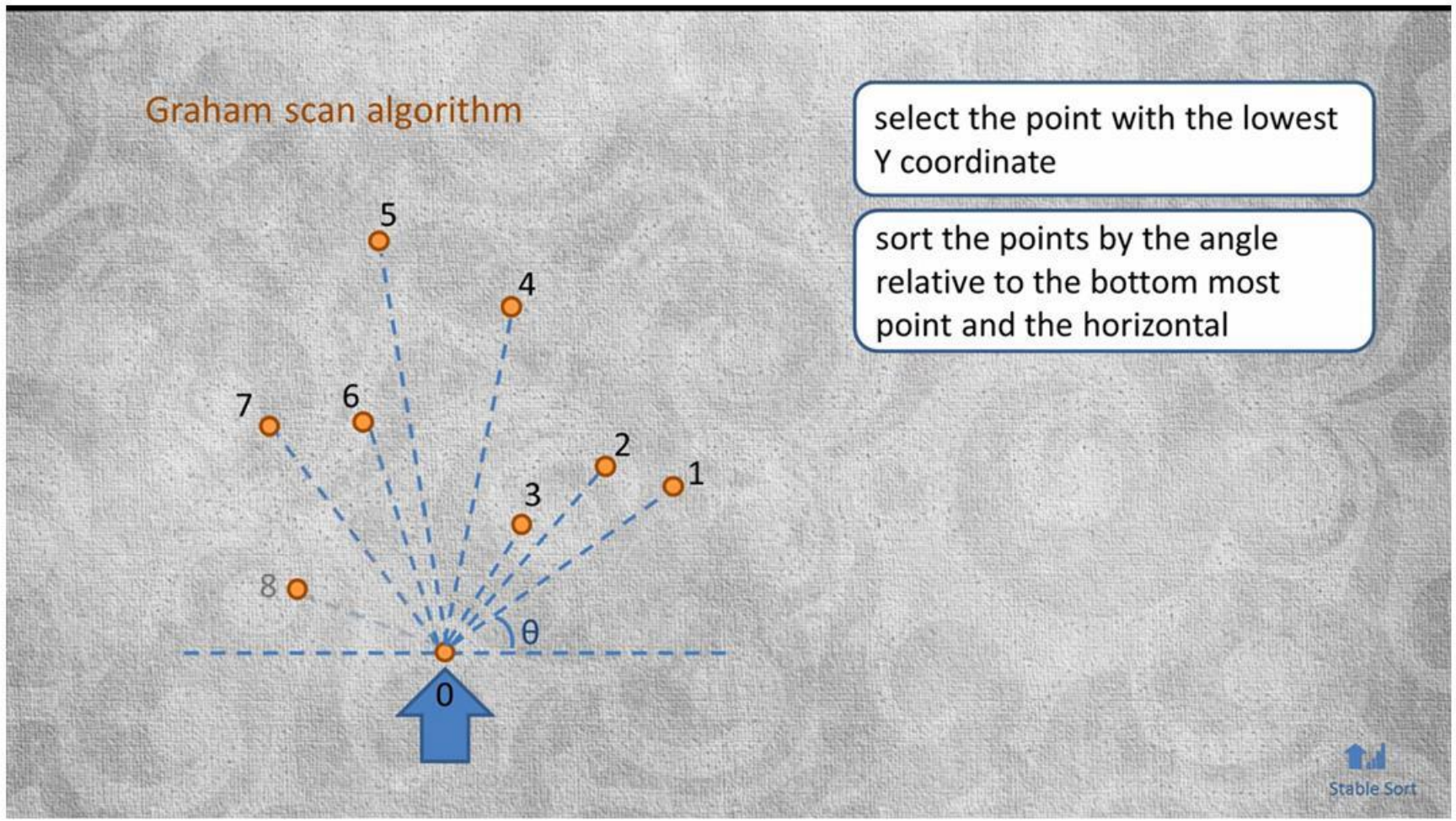
Algorithm - Graham Scan

- Multiple Lowest Points - Pick the leftmost point first.
- Sorting Multiple Equal Angles - Pick the point that is farthest from P0.
- Time Complexity: O(n log n)
- If the set is pre-sorted (by angle or vertices) complexity is O(n).
- Generally faster than Jarvis March. Jarvis's is faster if $h < \log n$.
- Space Complexity: O(n)

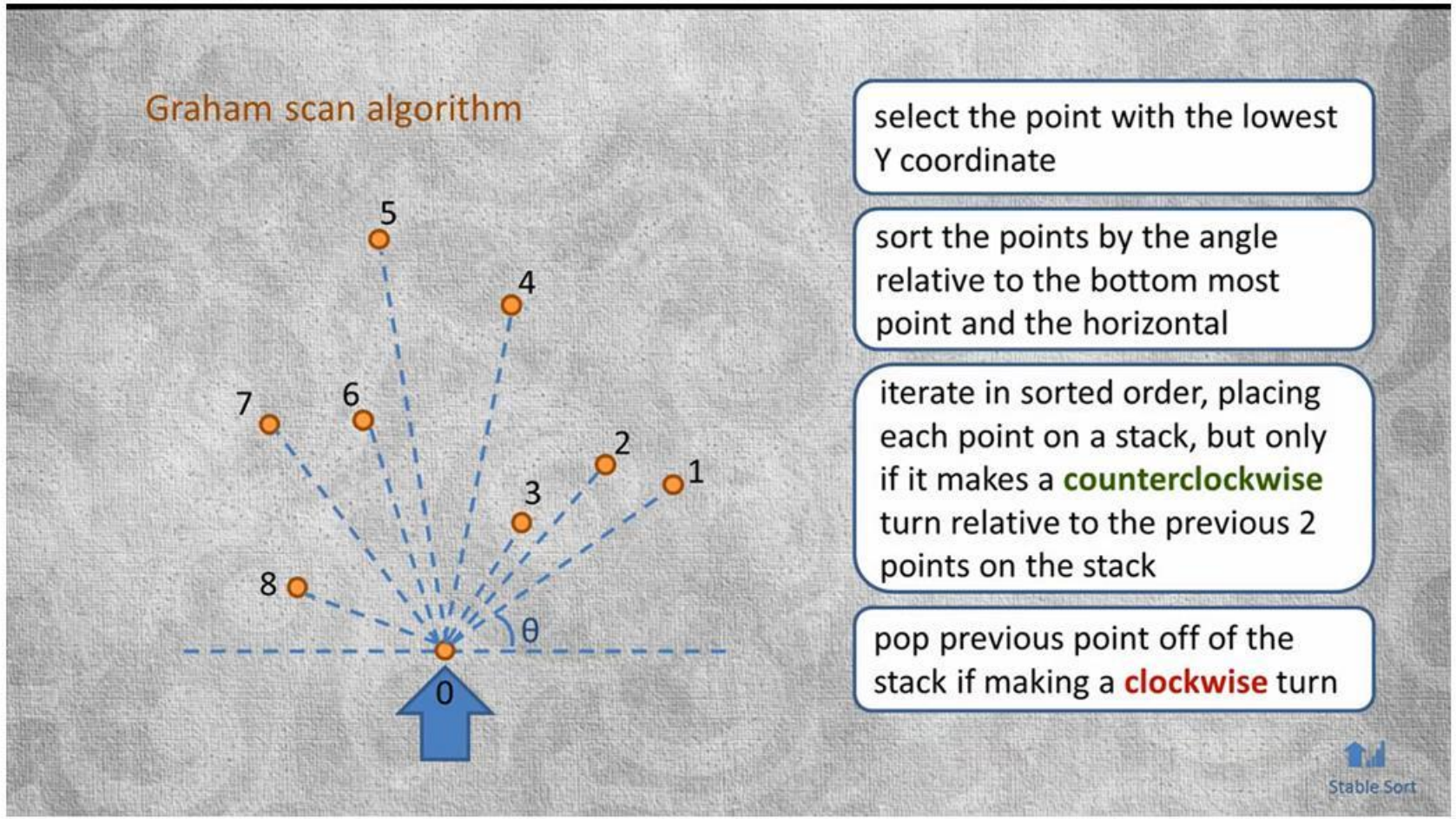
Convex Hull Algorithm - Graham Scan and Jarvis March



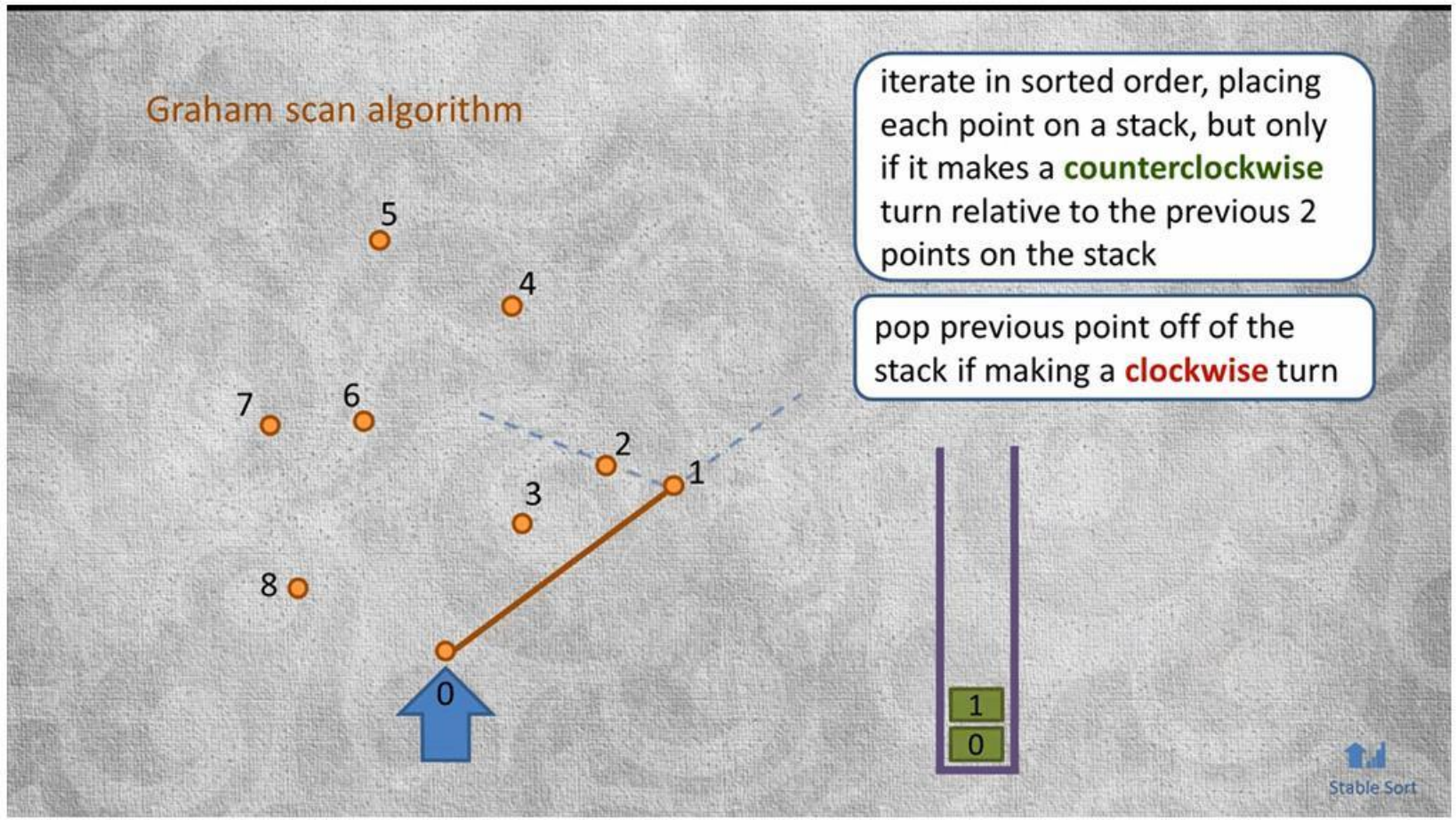
Convex Hull Algorithm - Graham Scan and Jarvis March



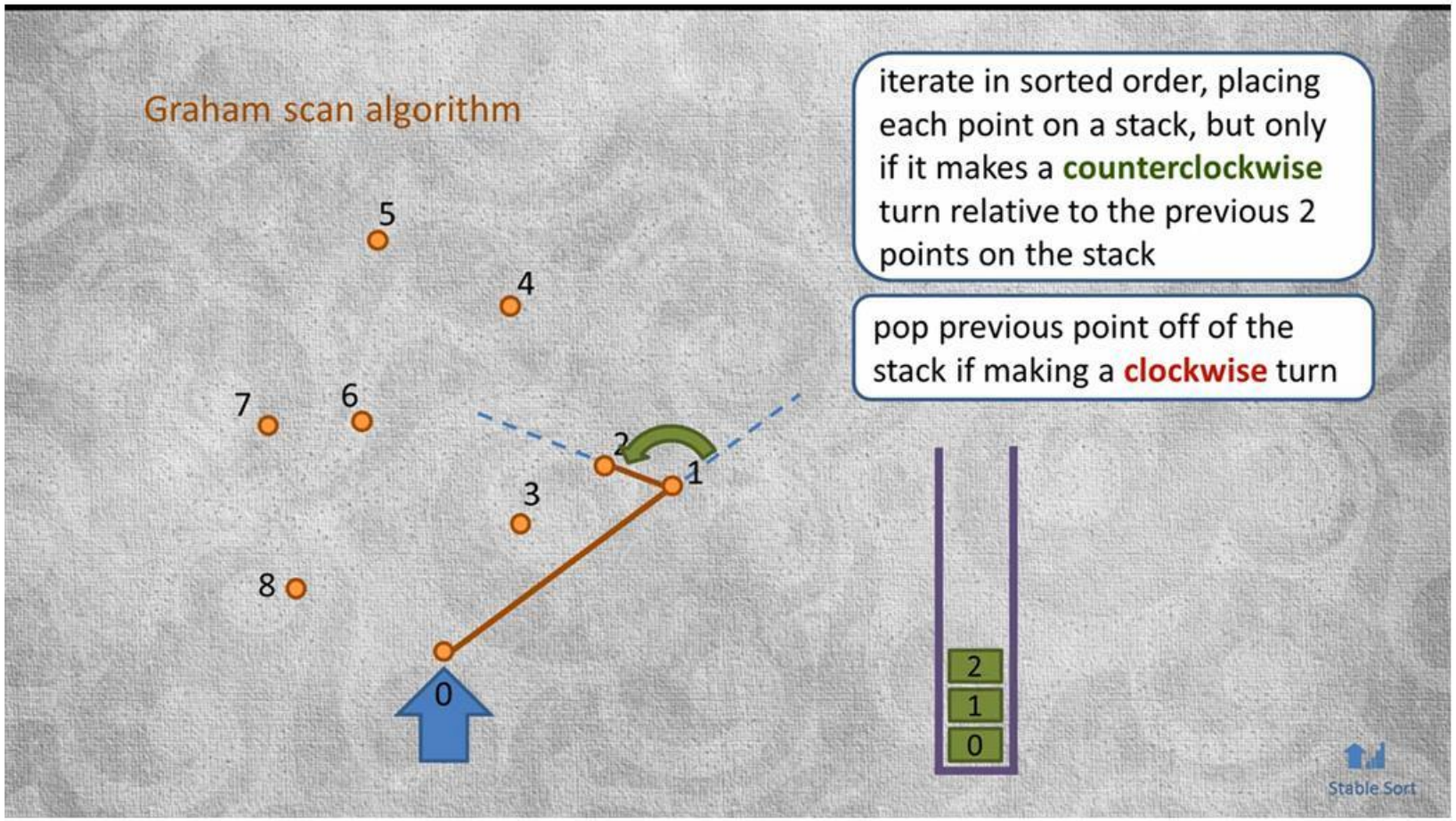
Convex Hull Algorithm - Graham Scan and Jarvis March



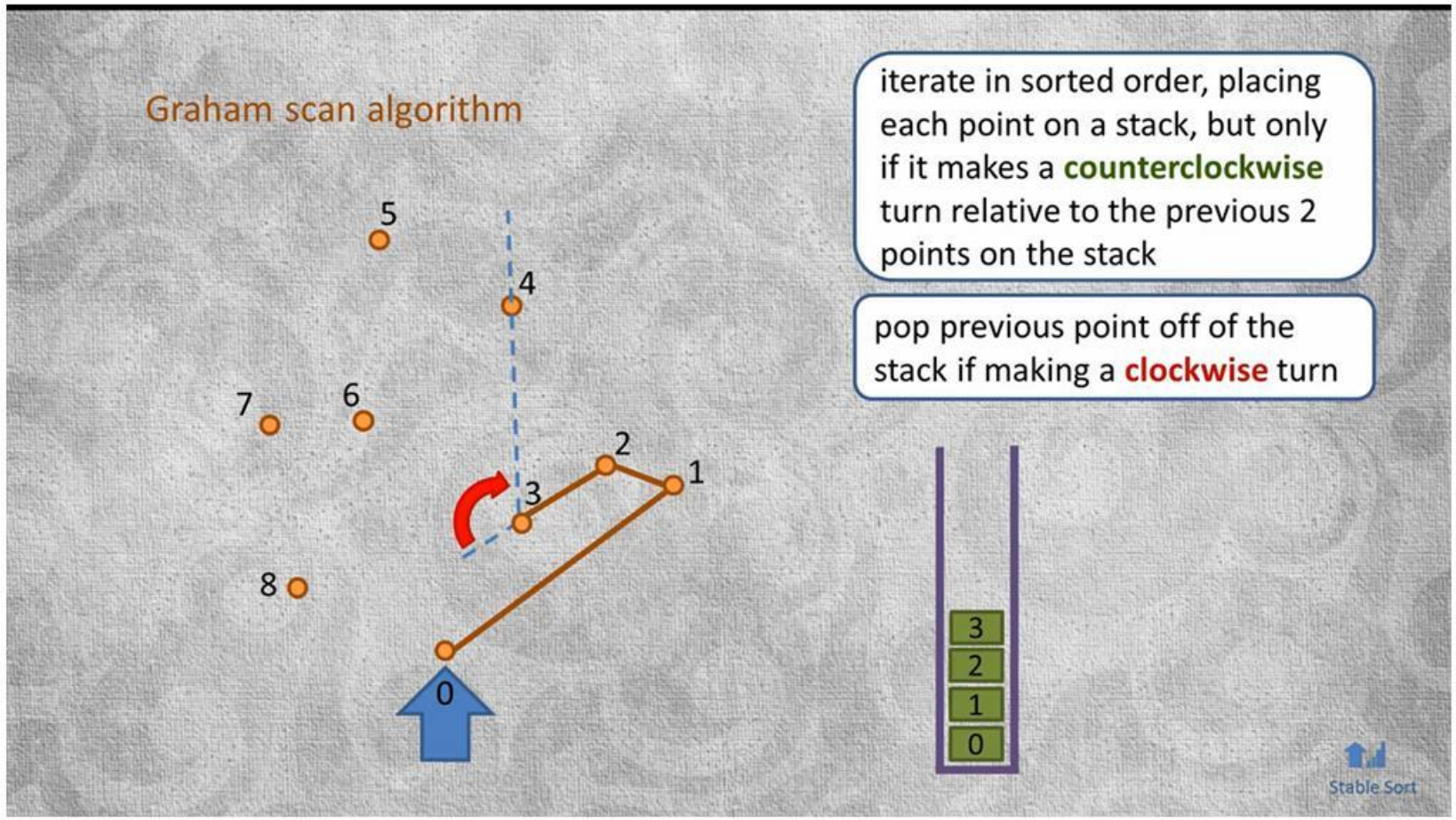
Convex Hull Algorithm - Graham Scan and Jarvis March



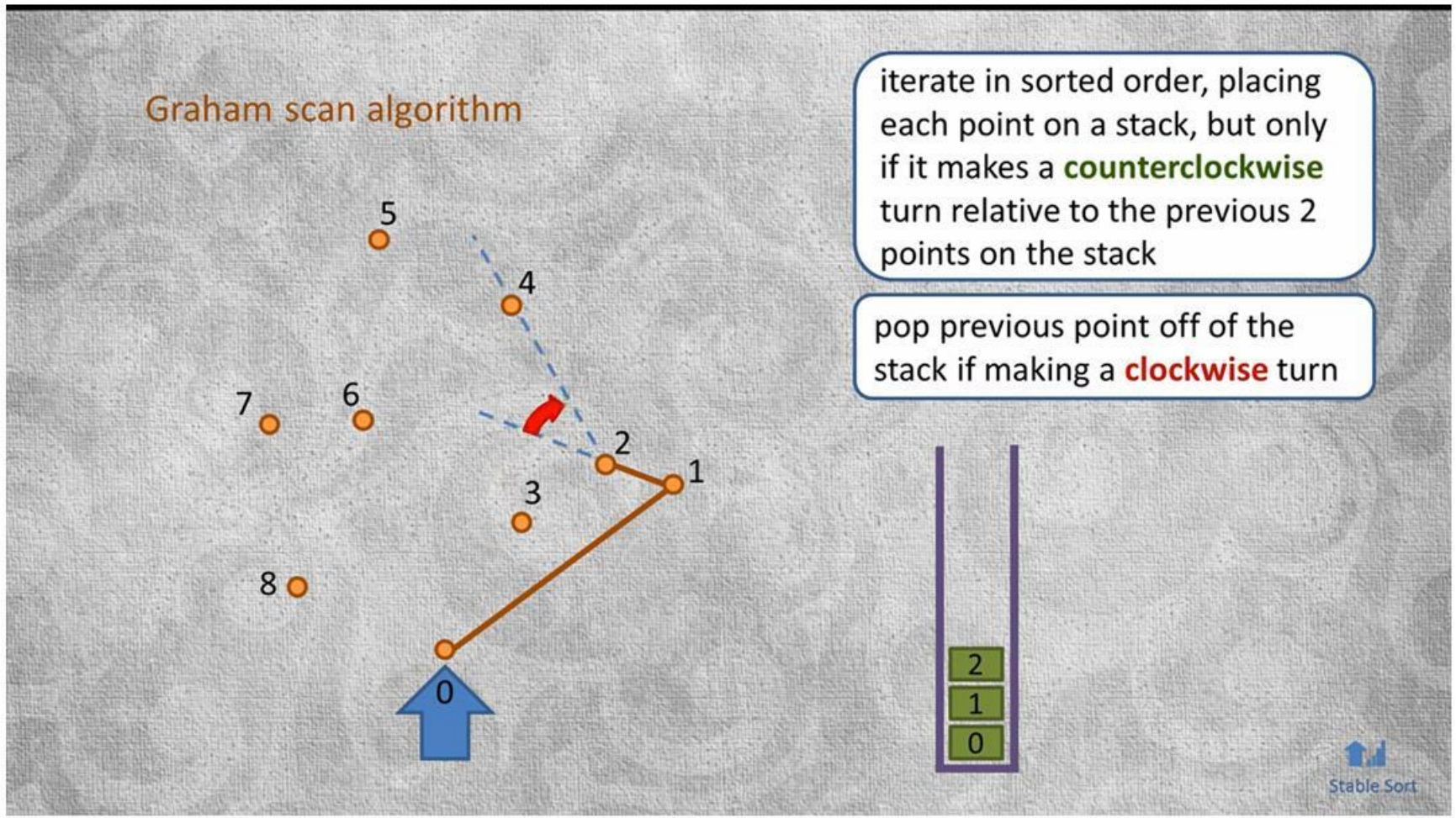
Convex Hull Algorithm - Graham Scan and Jarvis March



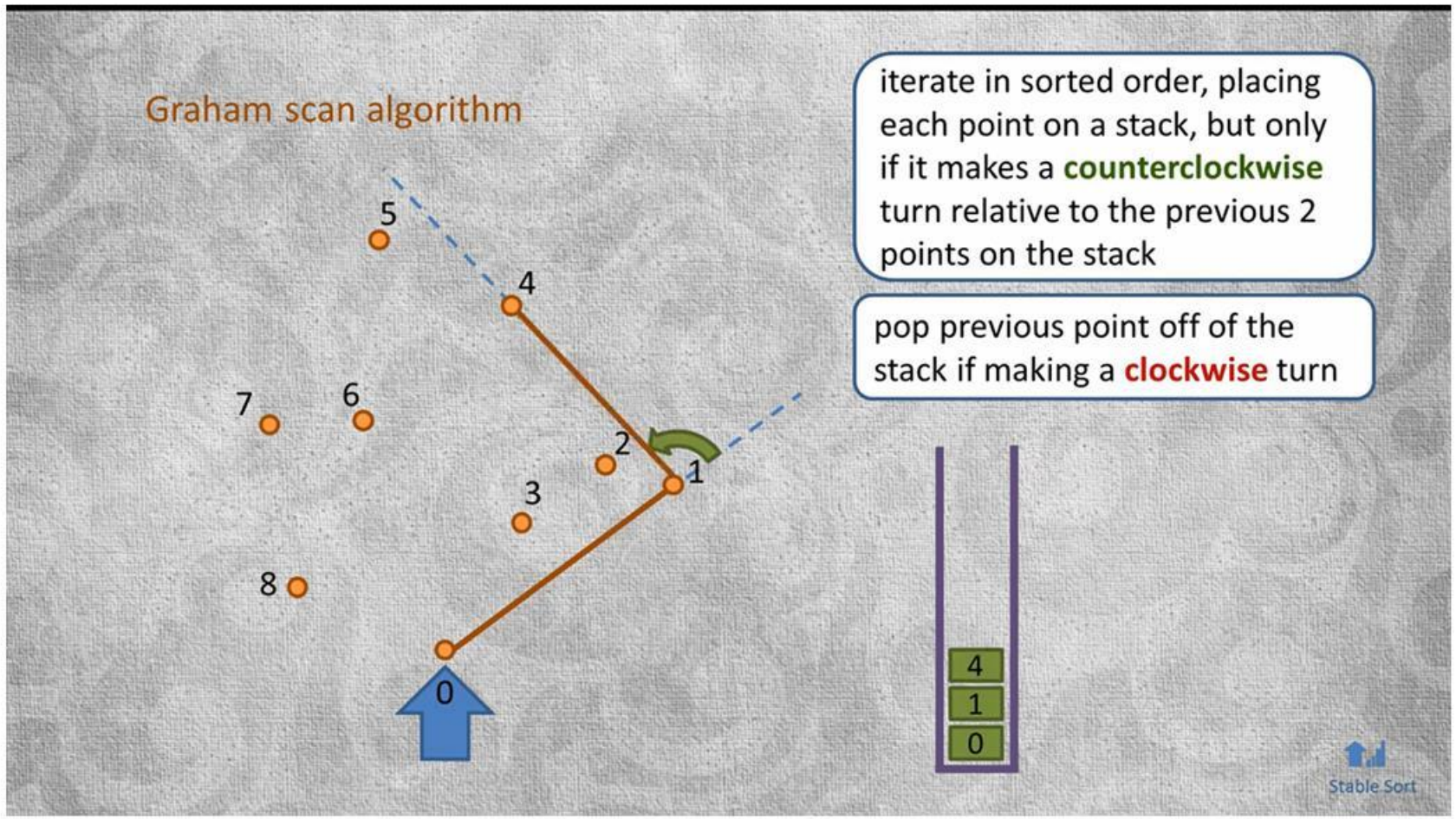
Convex Hull Algorithm - Graham Scan and Jarvis March



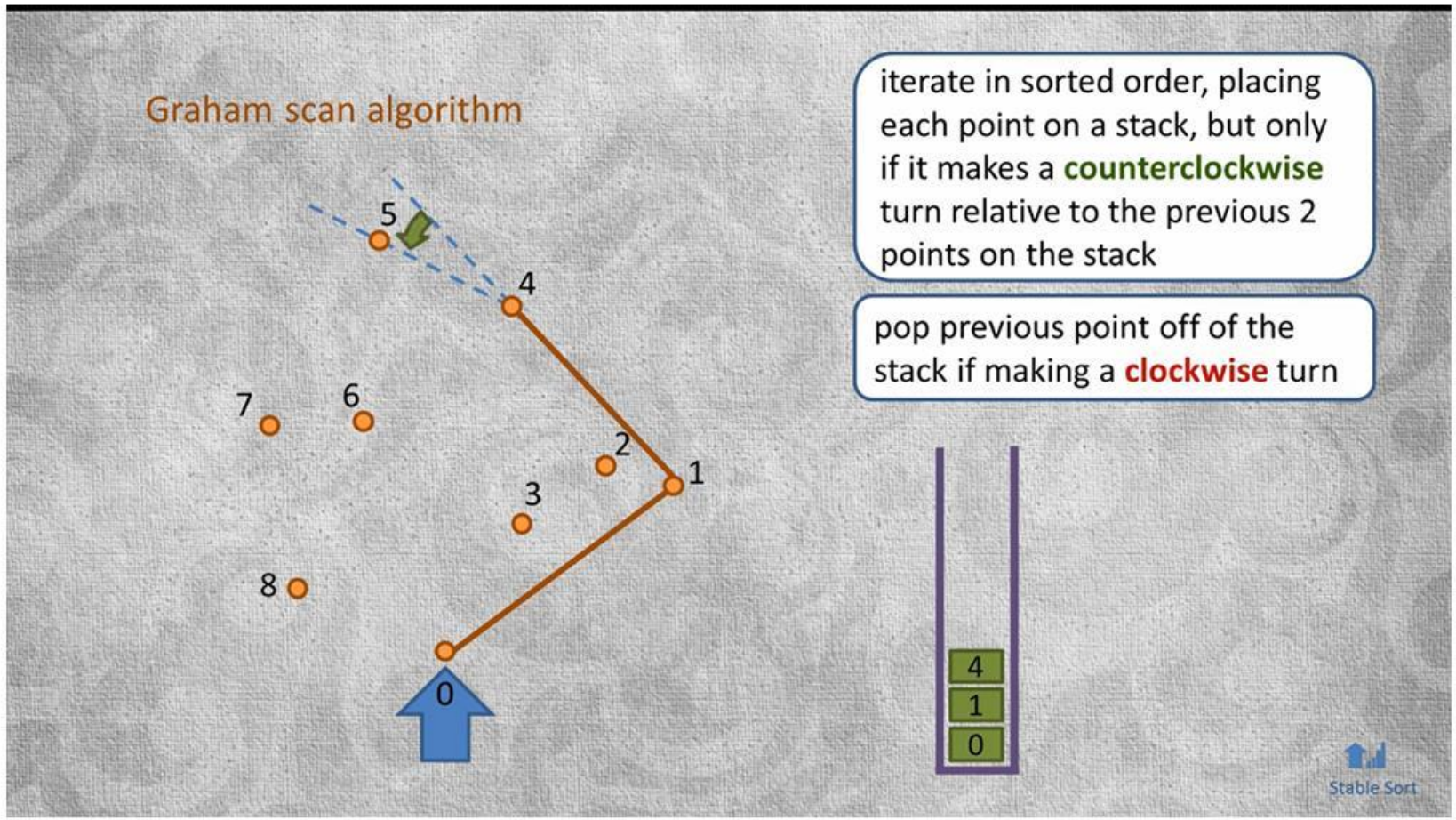
Convex Hull Algorithm - Graham Scan and Jarvis March



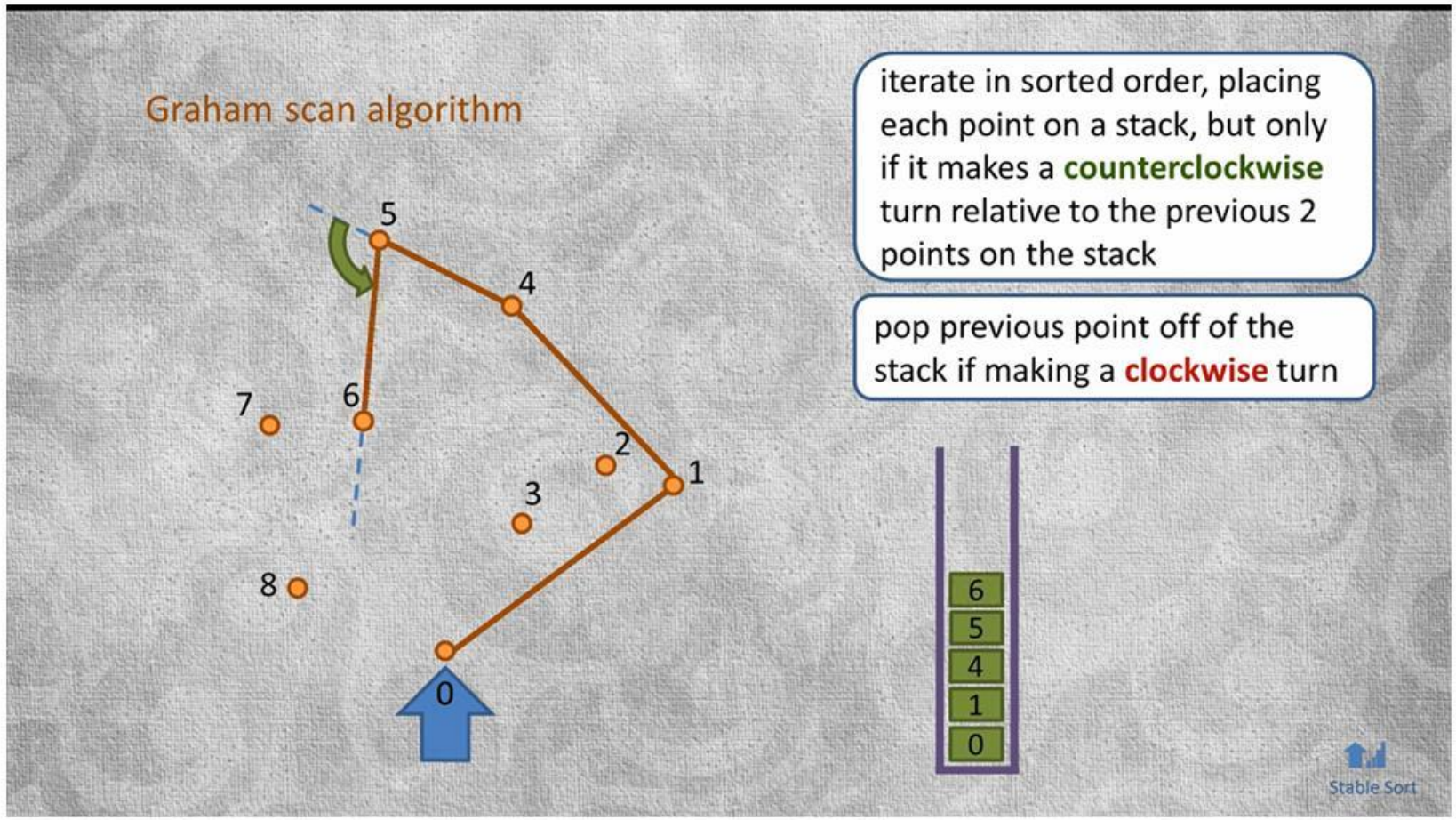
Convex Hull Algorithm - Graham Scan and Jarvis March



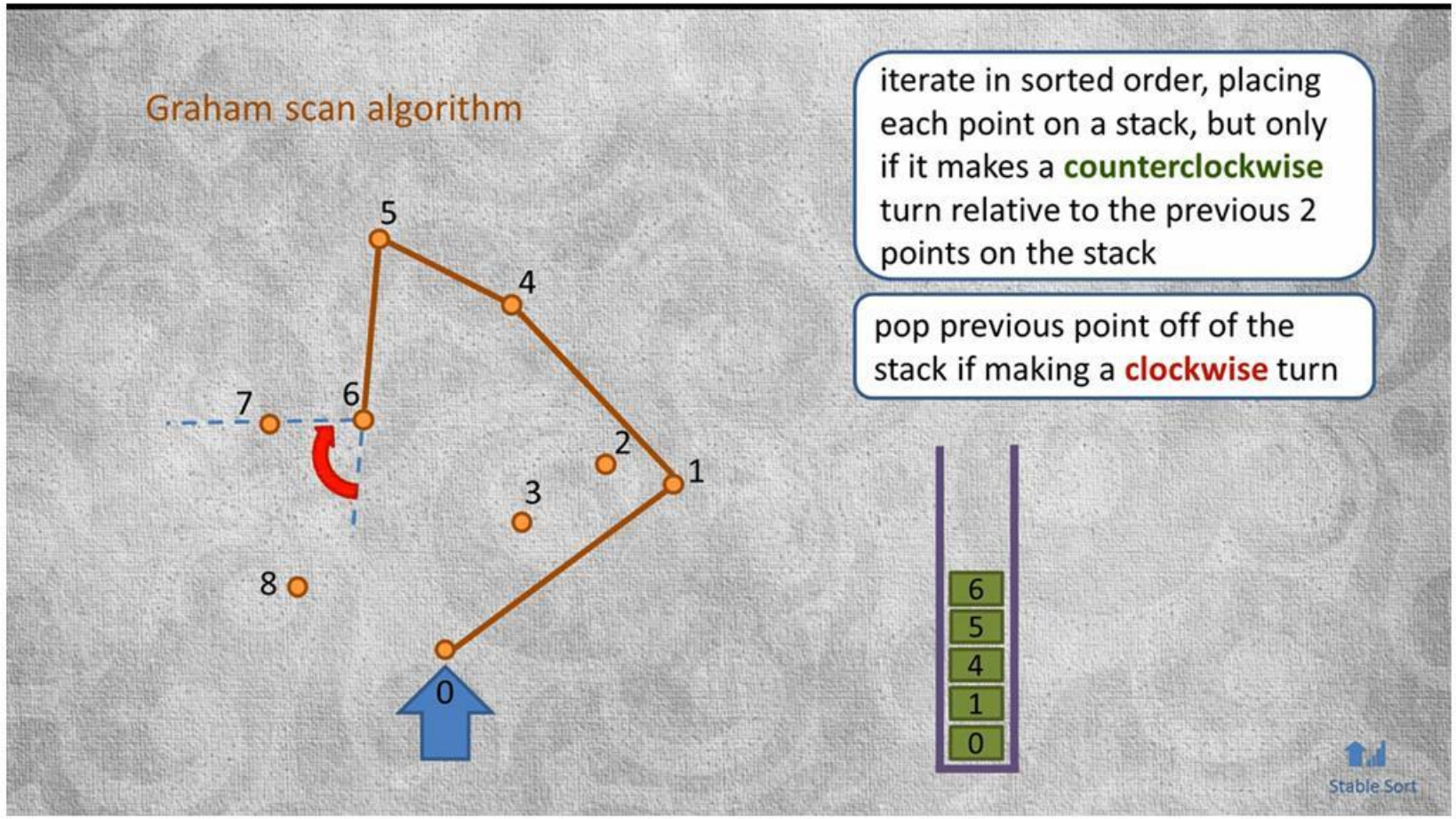
Convex Hull Algorithm - Graham Scan and Jarvis March



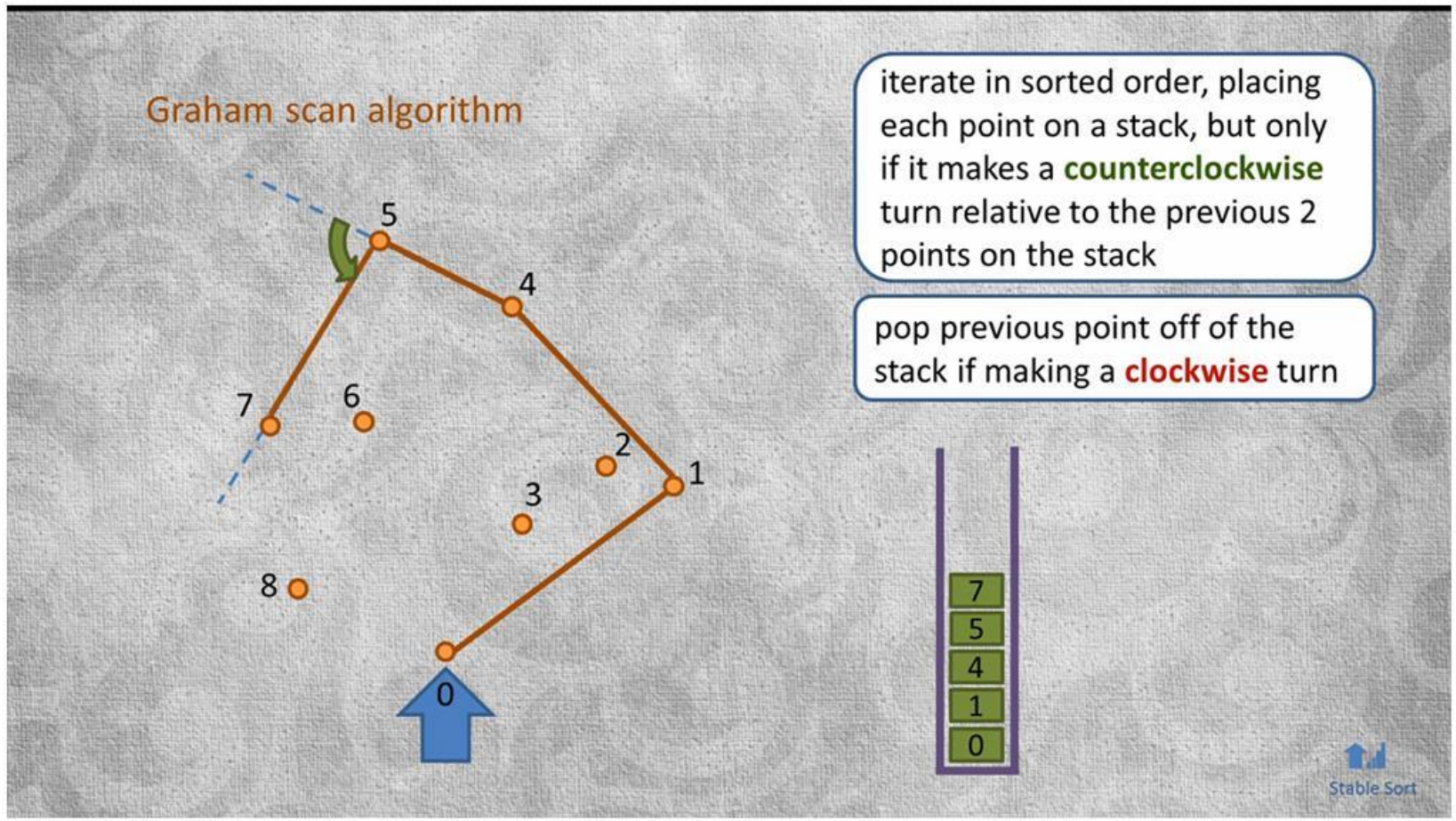
Convex Hull Algorithm - Graham Scan and Jarvis March



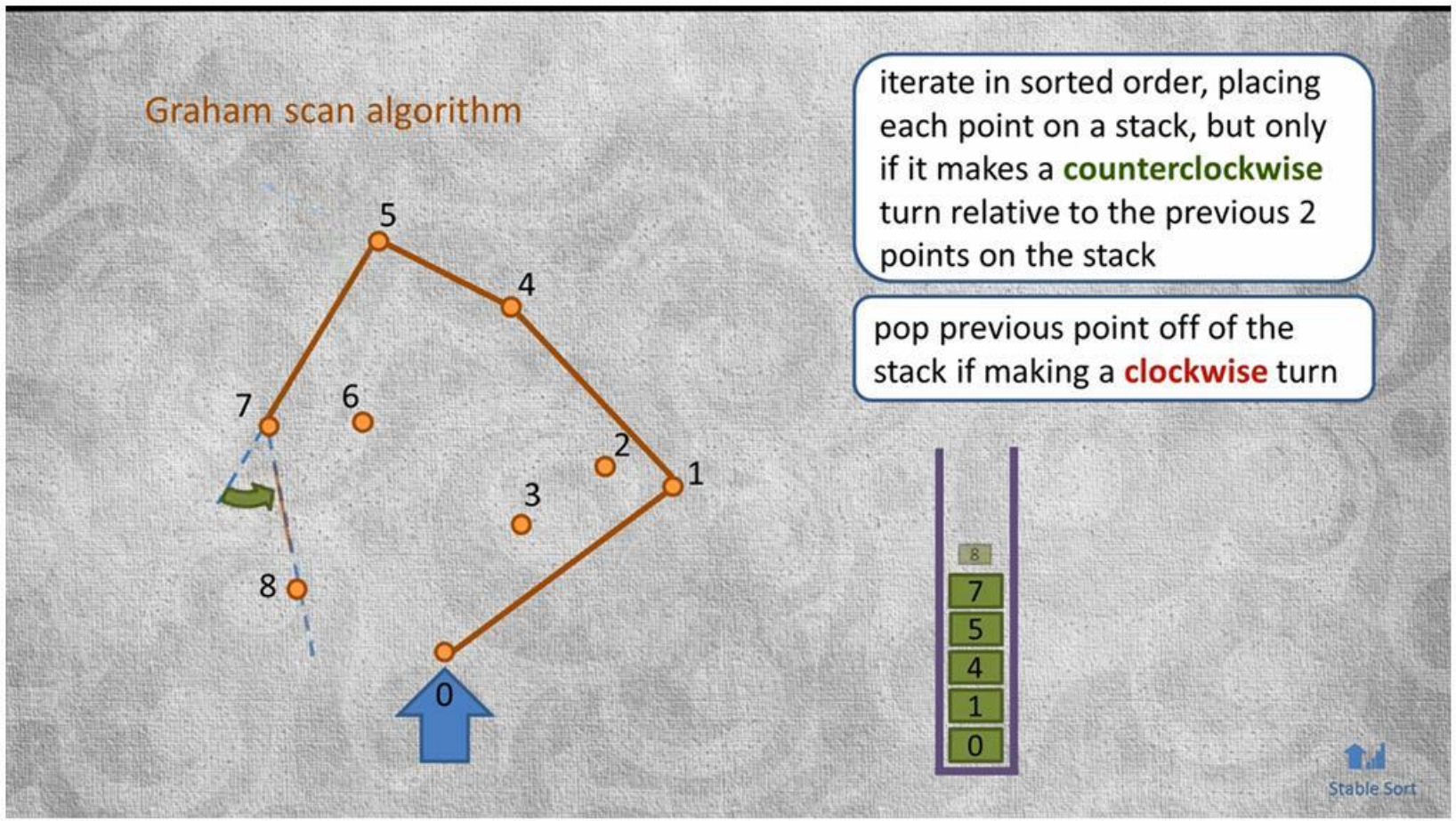
Convex Hull Algorithm - Graham Scan and Jarvis March



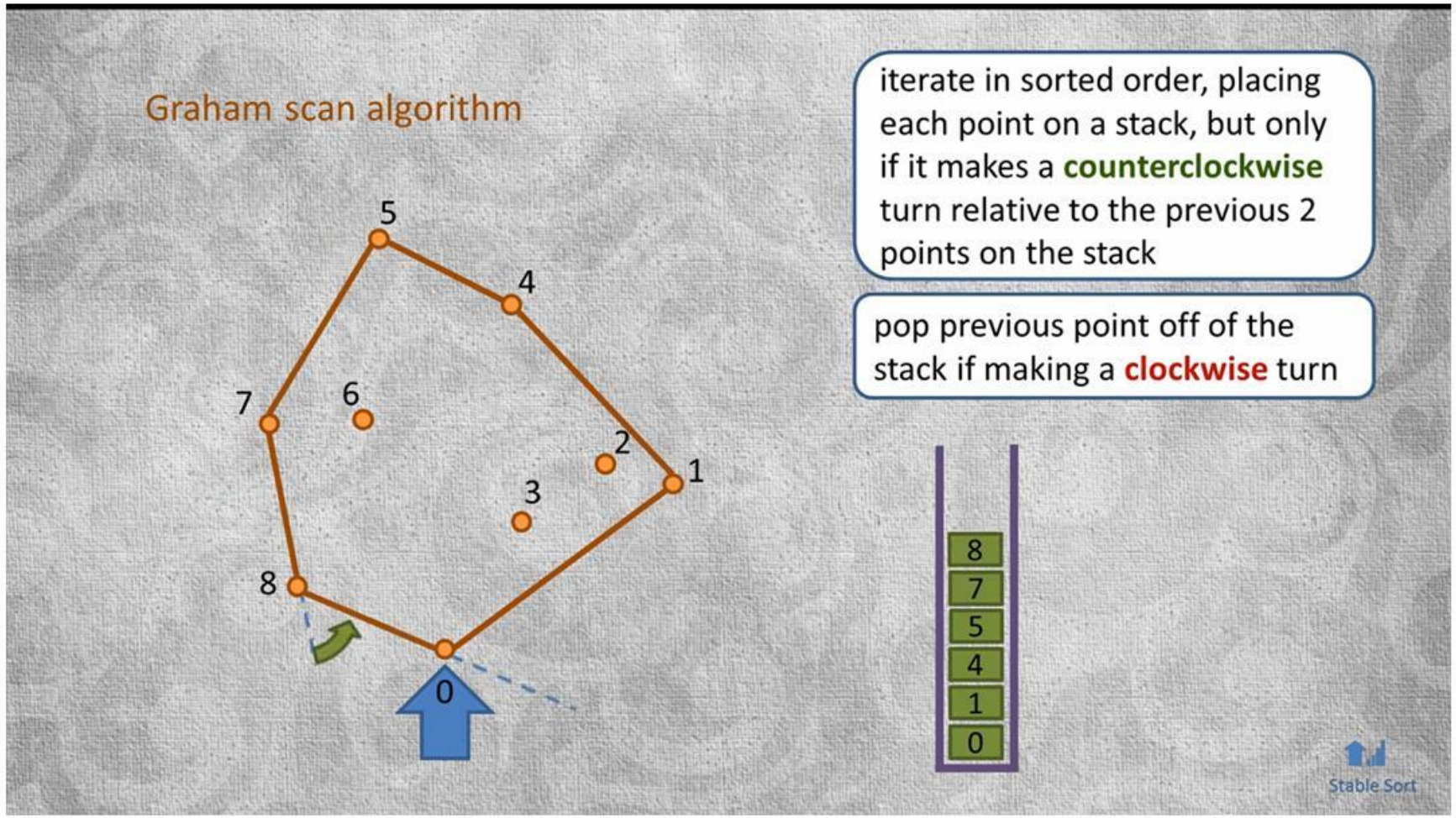
Convex Hull Algorithm - Graham Scan and Jarvis March



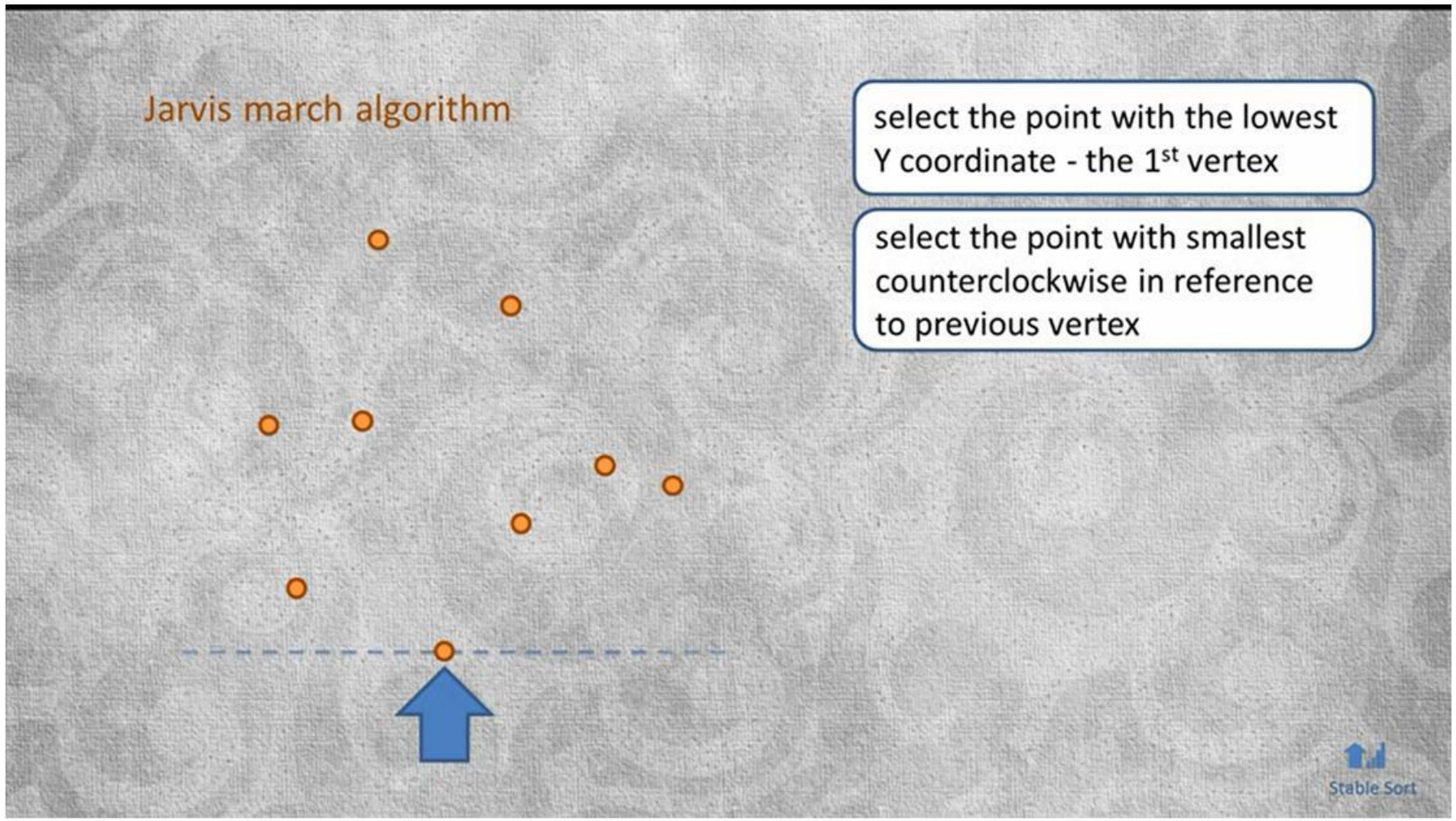
Convex Hull Algorithm - Graham Scan and Jarvis March



Convex Hull Algorithm - Graham Scan and Jarvis March

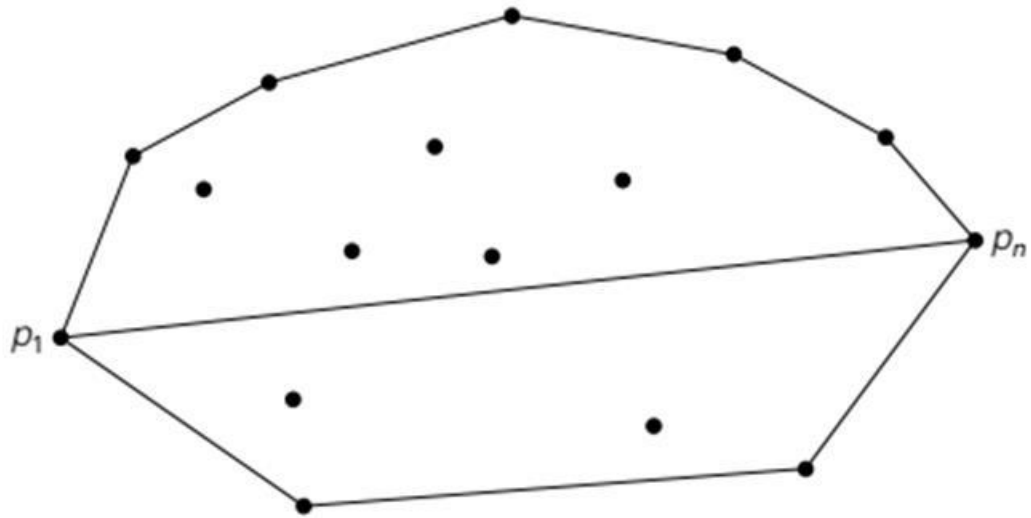


Convex Hull Algorithm - Graham Scan and Jarvis March

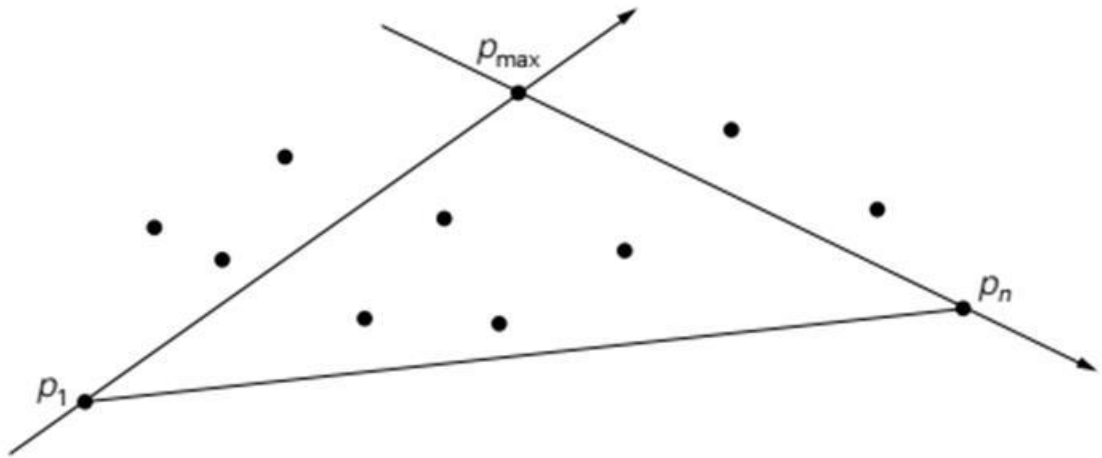


Convex Hull Problem

- Divide-and-conquer algorithm called ***quickhull***
  - Let *S* be a set of $n > 1$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$ in the Cartesian plane.
  - We assume that the points are sorted in nondecreasing order of their *x* coordinates, with ties resolved by increasing order of the *y* coordinates of the points involved.
  - Leftmost point p_1 and the rightmost point *pn* are two distinct extreme points of the set's convex hull
  - Let $\overrightarrow{p_1 p_n}$ be the straight line through points p_1 and p_n directed from p_1 to p_n.

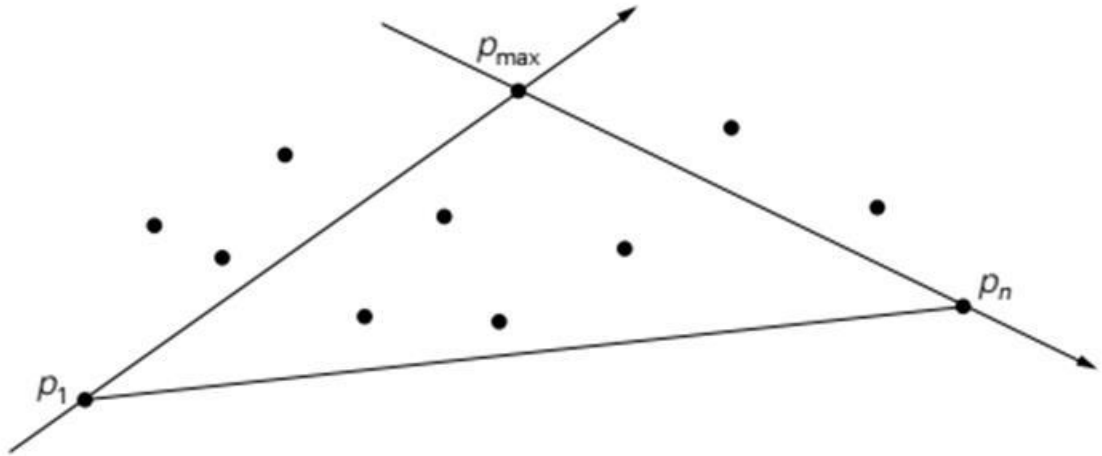


Convex Hull Problem



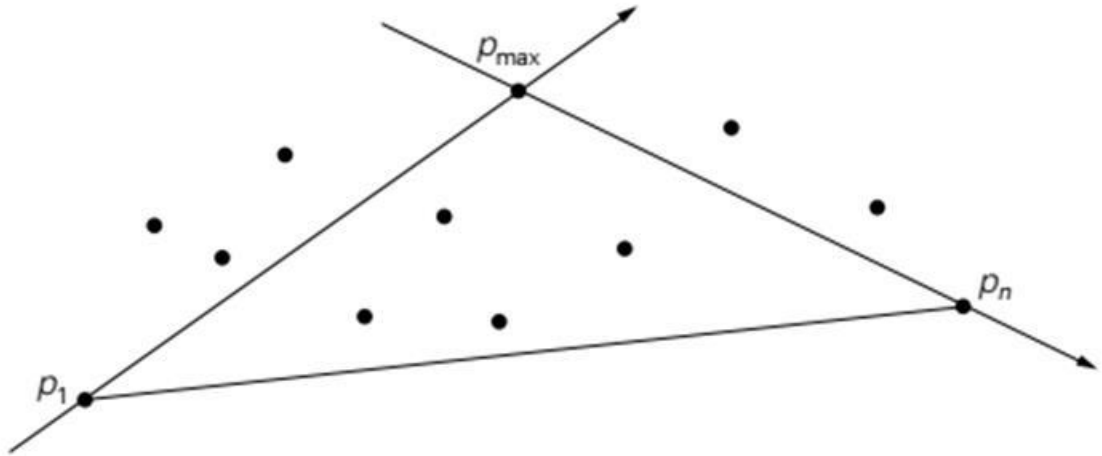
- Quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner.
- This line separates the points of S into two sets:
  - S_1 is the set of points to the left of this line, and
  - S_2 is the set of points to the right of this line
- If S_1 is empty, the upper hull is simply the line segment with the endpoints at p_1 and p_n.
- If S_1 is not empty, the algorithm identifies point p_{max} in S_1, which is the farthest from the line $\overrightarrow{p_1p_n}$

Convex Hull Problem



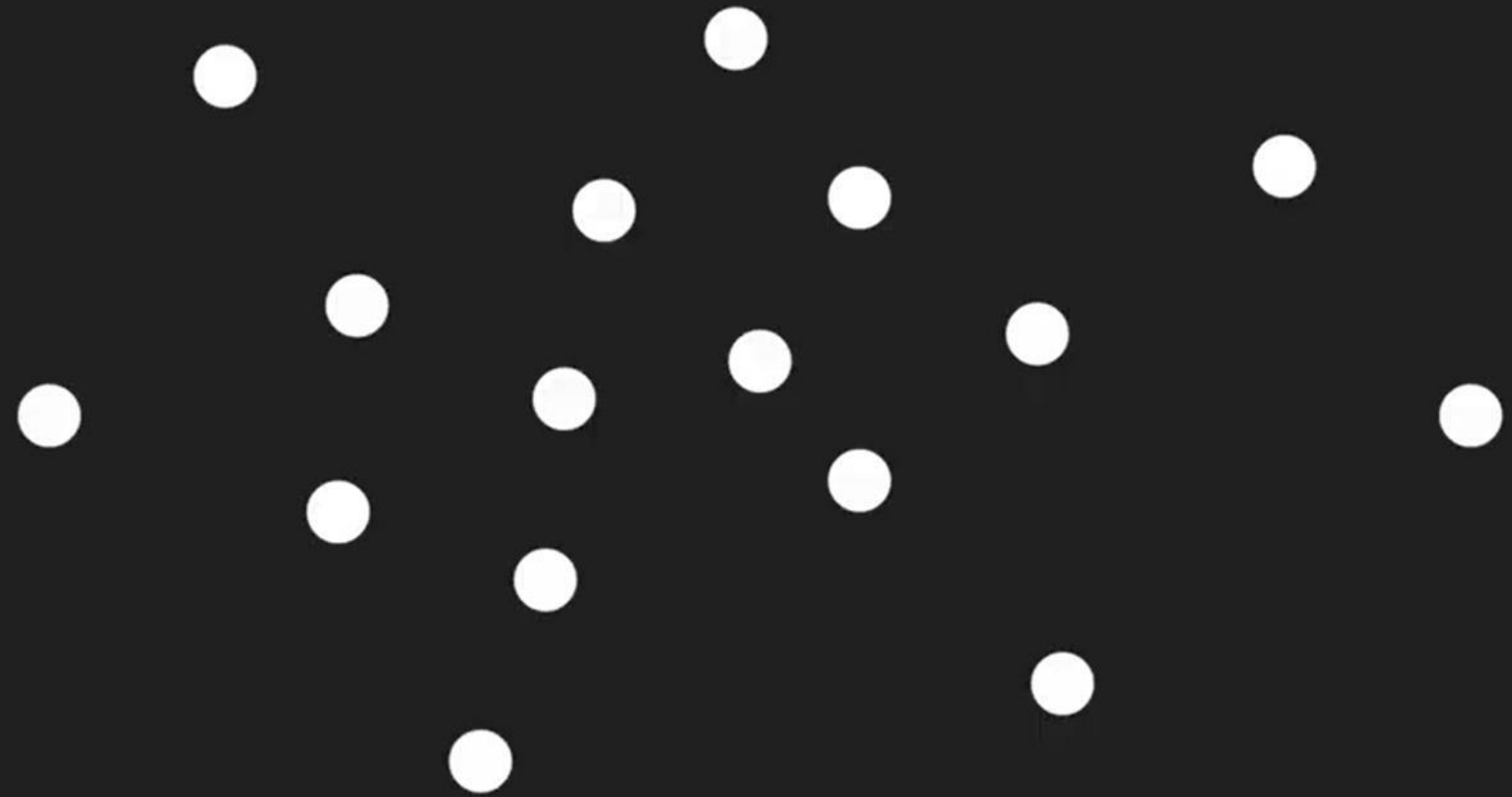
- If there is a tie, the point that maximizes the angle $\angle p_{max}pp_n$ can be selected.
- Then the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{p_1p_{max}}$, these are the points that will make up the set $S_{1,1}$.
- The points of S_1 to the left of the line $\overrightarrow{p_{max}p_n}$ will make up the set $S_{1,2}$.

Convex Hull Problem

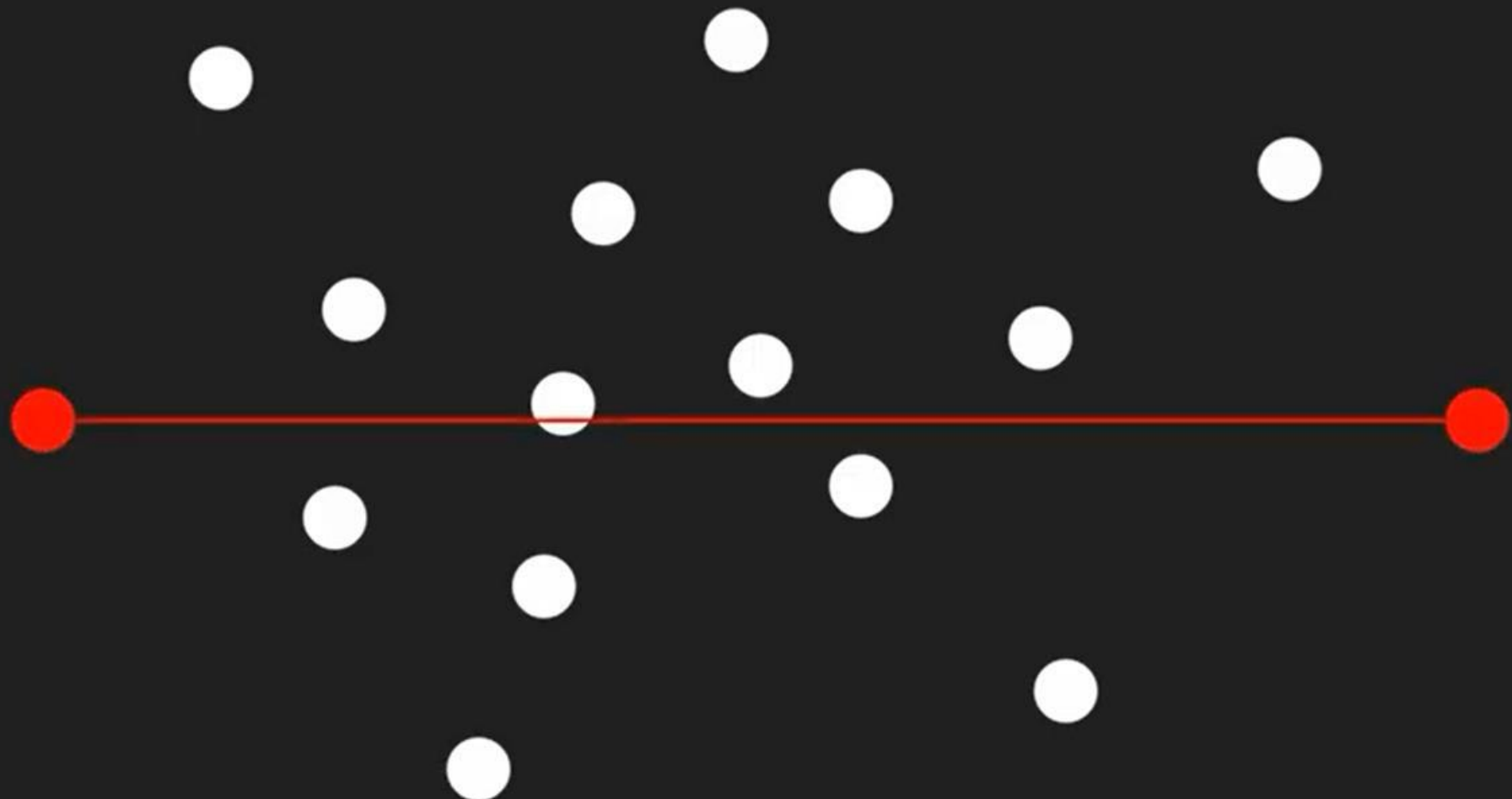


- Therefore, the algorithm can continue constructing the upper hulls of $p_1 \cup S_{1,1} \cup p_{max}$ and $p_{max} \cup S_{1,2} \cup p_n$ recursively and then simply concatenate them to get the upper hull of the entire set $p_1 \cup S_1 \cup p_n$.

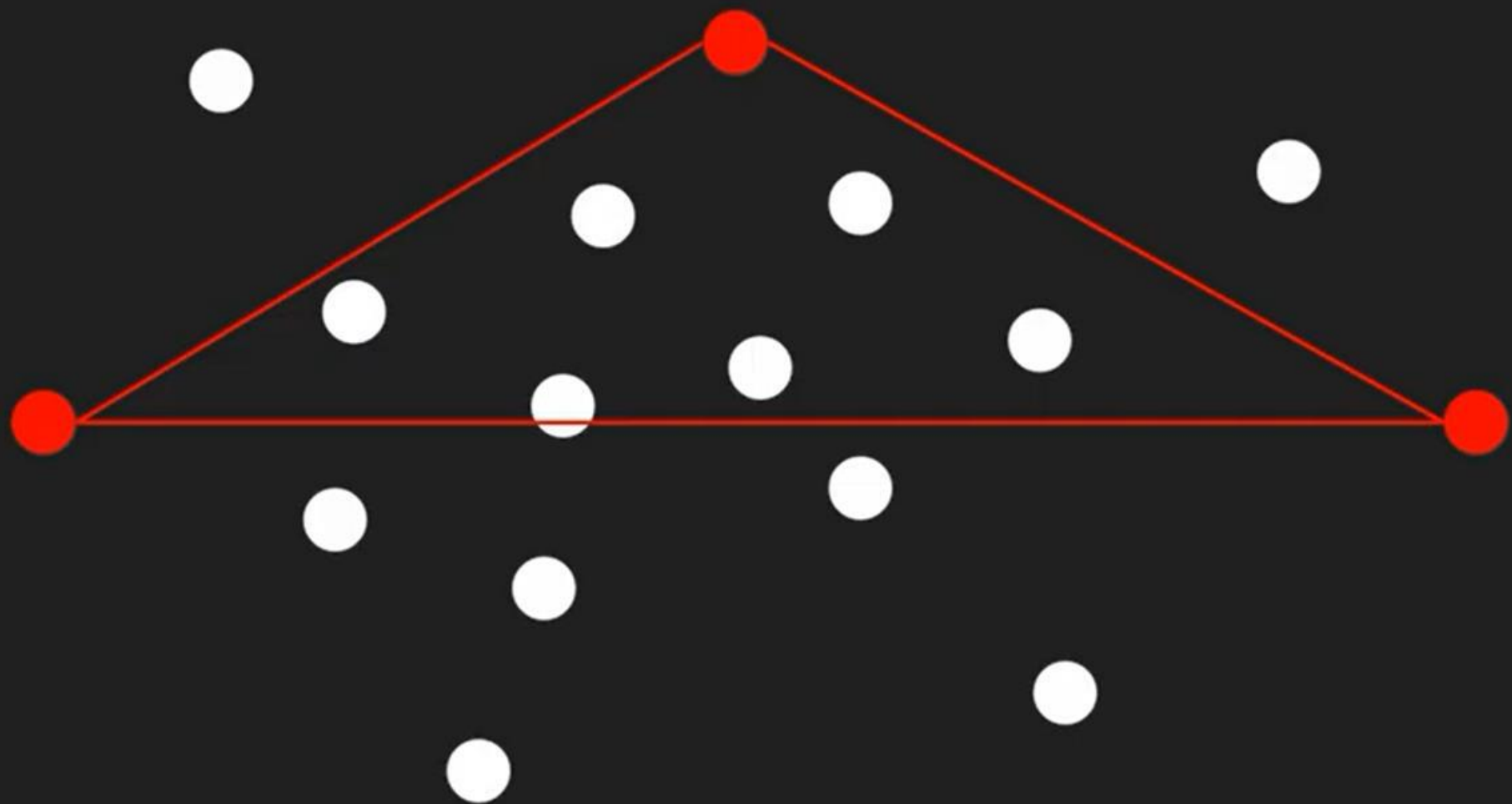
Quickhull: Divide and Conquer



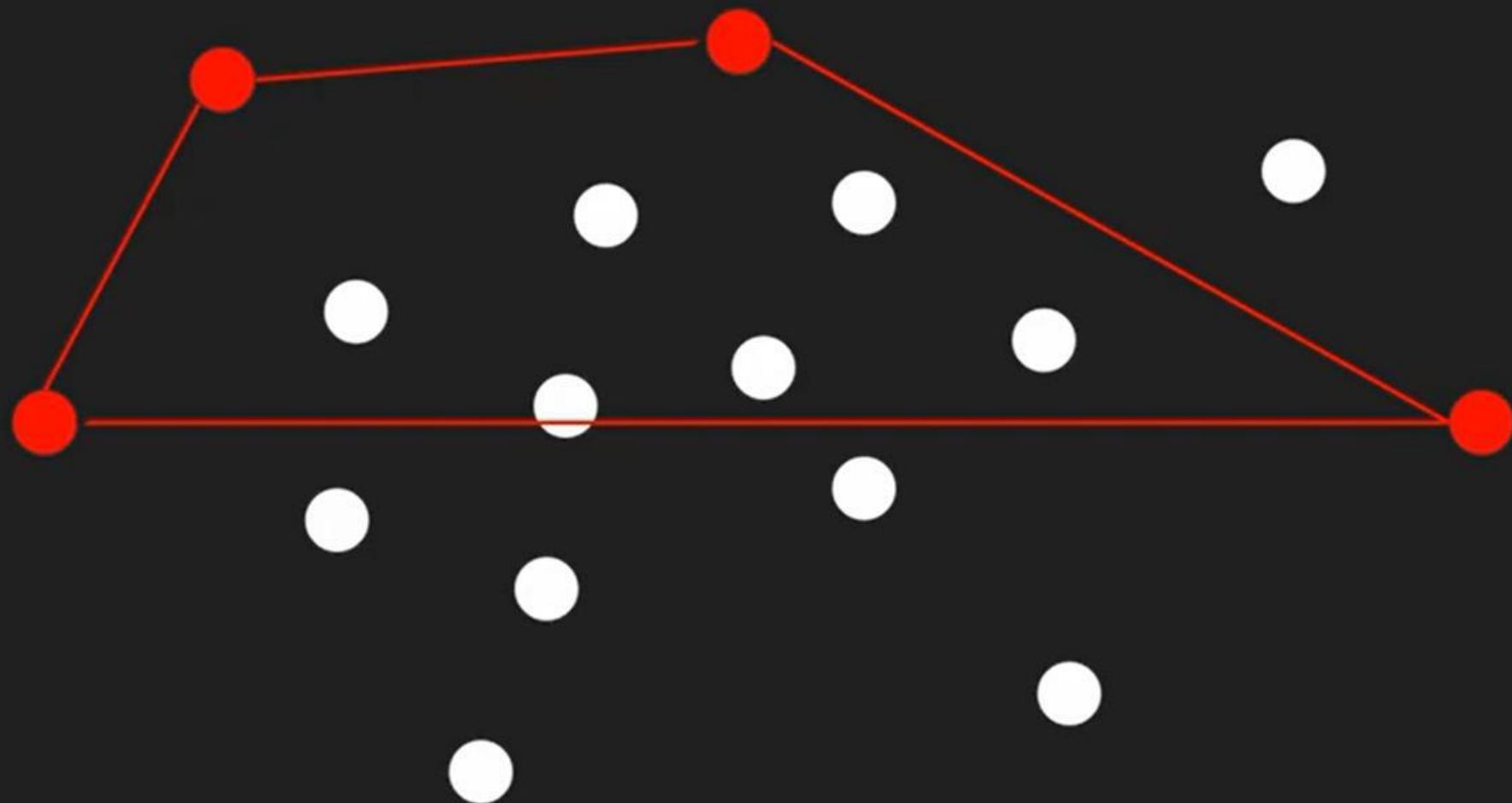
Quickhull: Divide and Conquer



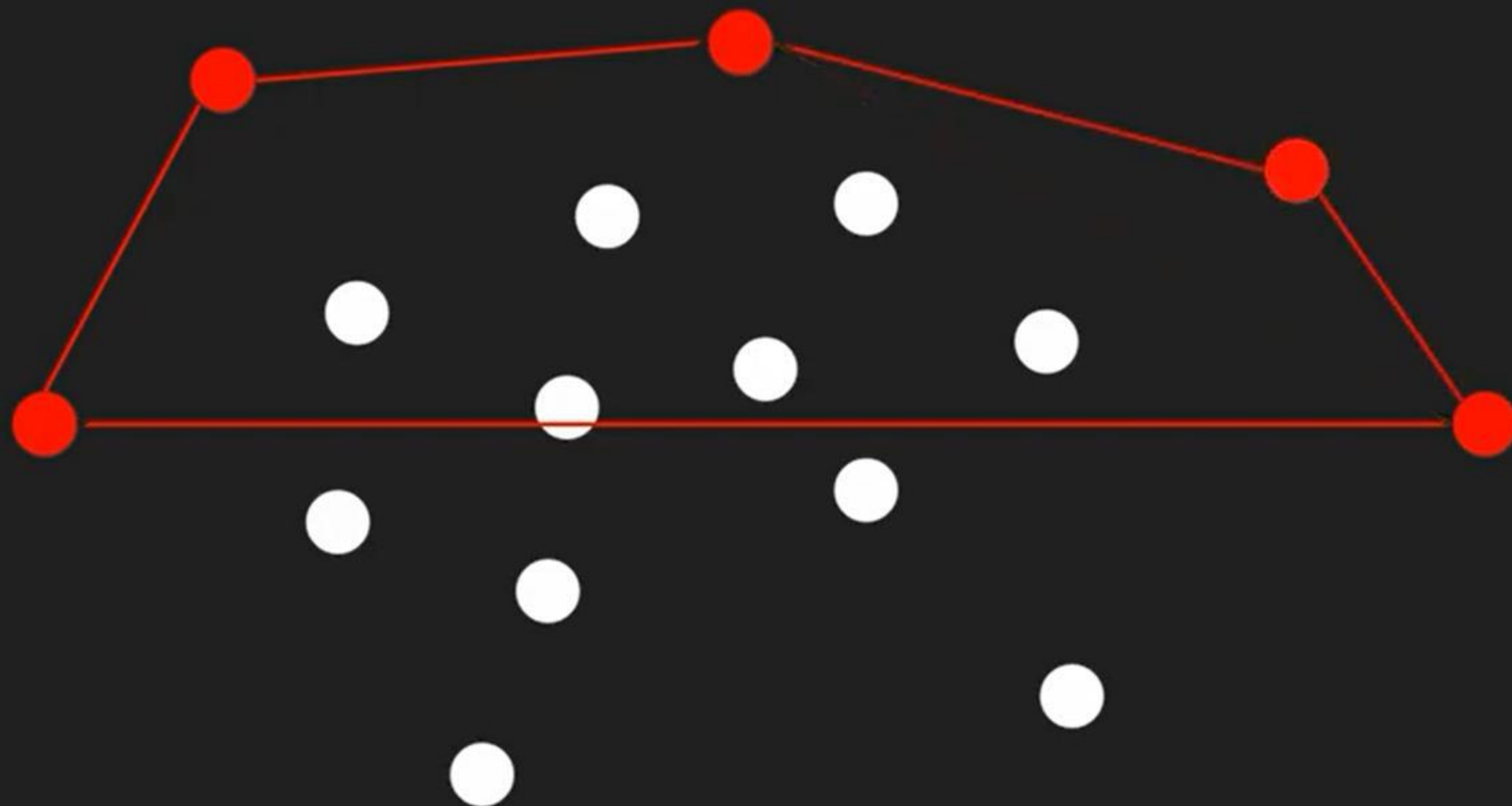
Quickhull: Divide and Conquer



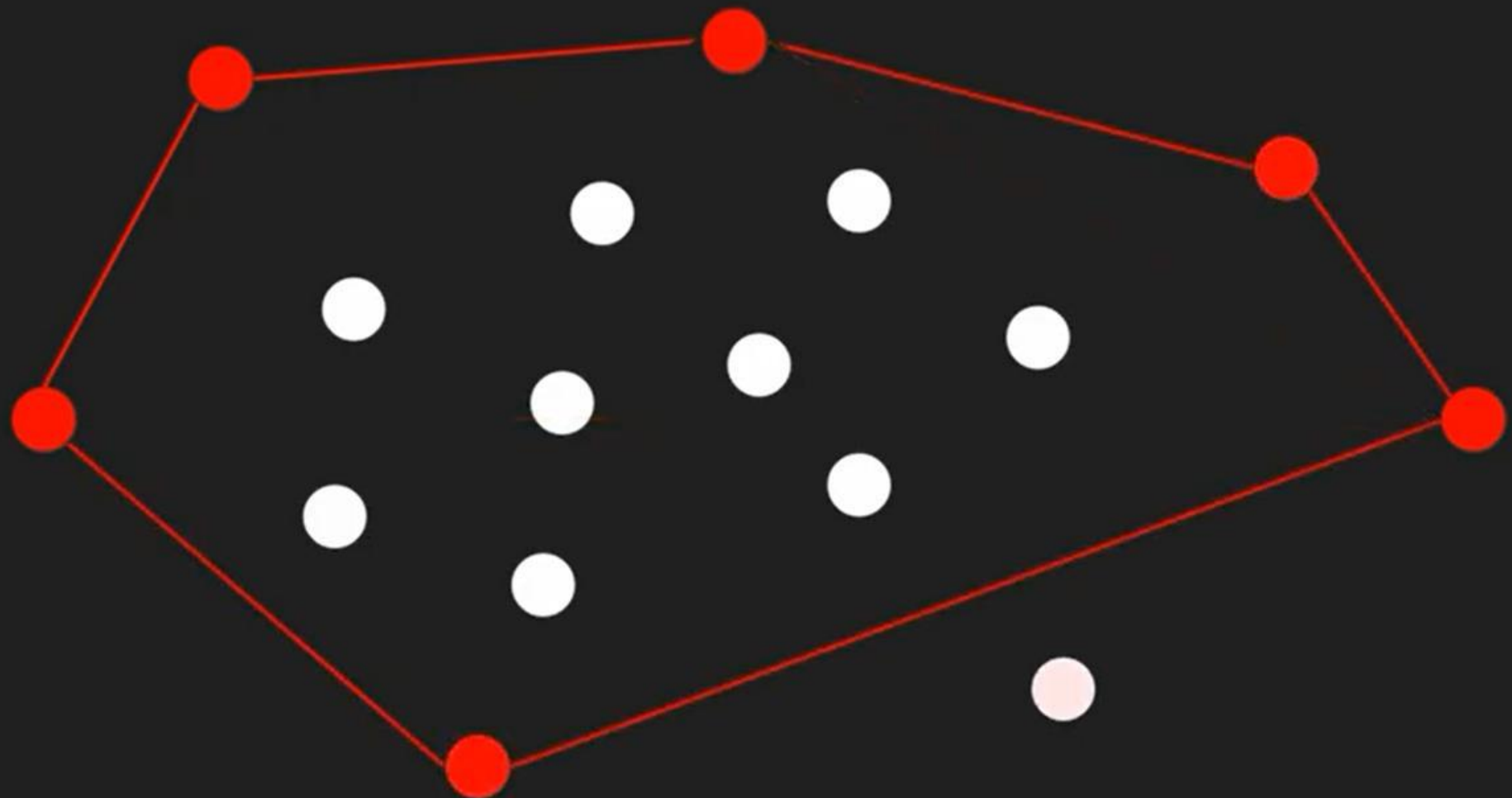
Quickhull: Divide and Conquer



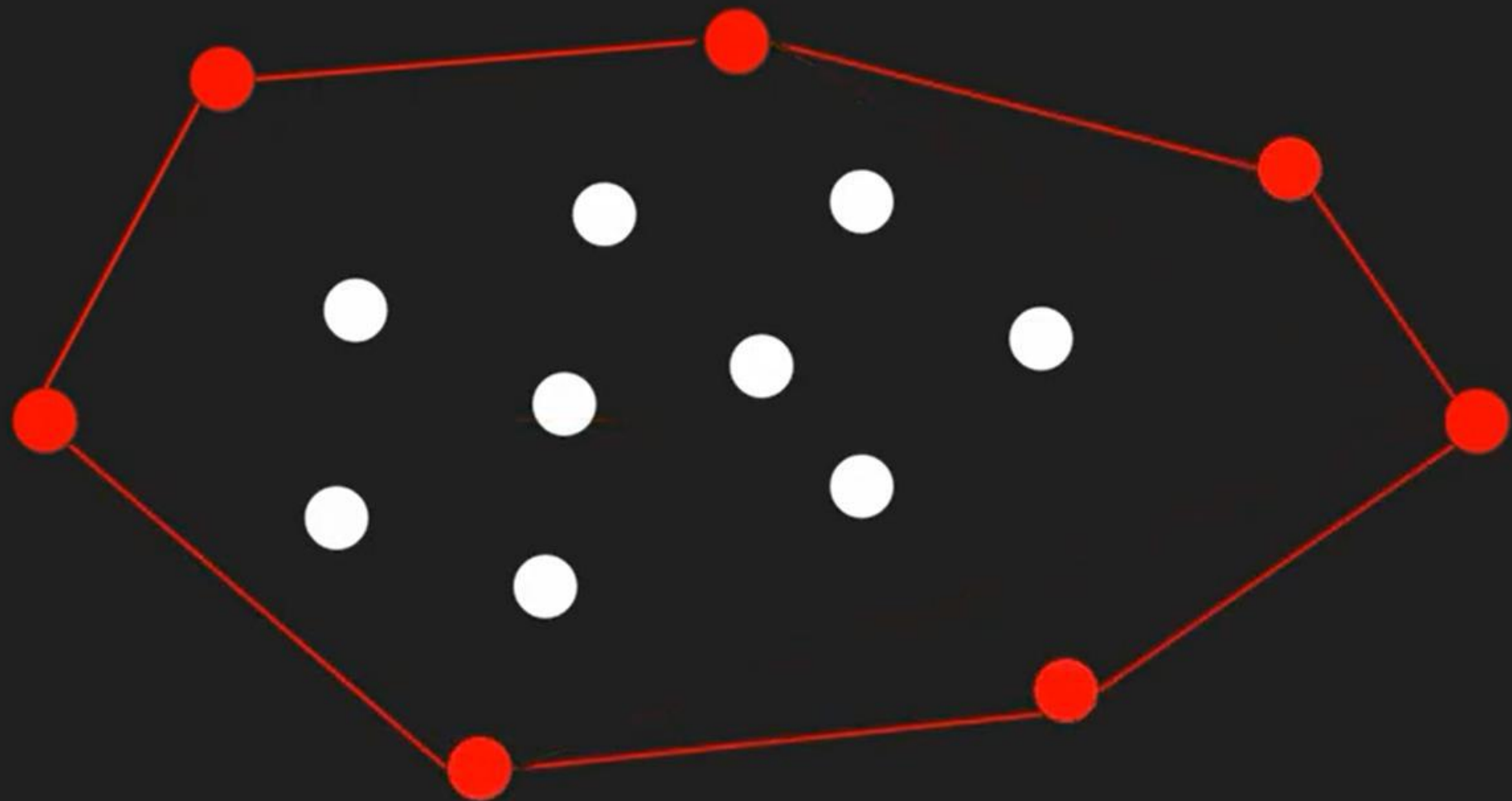
Quickhull: Divide and Conquer



Quickhull: Divide and Conquer



Quickhull: Divide and Conquer



Pseudo Code

```
Input = a set S of n points
Assume that there are at least 2 points in the input set S of points
QuickHull (S)
{
   // Find convex hull from the set S of n points
   Convex_Hull := {}
   Find left and right most points, say A & B
   Add A & B to convex hull
   Segment AB divides the remaining (n-2) points  into 2 groups S1 and S2
            where S1 are points in S  that are on the right side of the oriented line from A to B, and
           S2 are points in S that are on the right side of the oriented line  from B to A
     FindHull (S1, A, B)
    FindHull (S2, B, A)
}
```

Pseudo Code

```
FindHull (Sk, P, Q)
{
   // Find points on convex hull from the set Sk of points that are on the right side of the oriented line from P to Q
   If Sk has no point, then return.
   From the given set of points in Sk, find farthest point, say C, from segment PQ
    Add point C to convex hull at the location between P and Q .
    Three points P, Q and C partition the remaining points of Sk into 3 subsets: S0, S1, and S2
        where S0 are points inside triangle PCQ,
            S1 are points on the right side of the oriented line from  P to C
            S2 are points on the right side of the oriented line from C to Q
   FindHull(S1, P, C)
   FindHull(S2, C, Q)
}
```

Complexity

- $T(n) = T(n_1) + T(n_2) + O(n)$

  where, n_1 and n_2 are two resultant parts such that $n_1 + n_2 \leq n$

- Above recurrence yields to arthematic series leads to $\theta(n^2)$, similar to quicksort

Time complexity of algorithms

- Time complexity of algorithms is measured in the number of vertices n and the number of hull vertices h.
  - Jarvis March = O(nh)
  - Graham Scan = O(n log n)
  - Quickhull/Divide and Conquer = O(n log n) → $O(n^2)$ worst case
  - Best algorithms = O (n log h)

Acknowledgement

- Christian Atwater Levente Dojcsak, The University of Tennessee, Knoxville
- Andre Violentyev, Convex Hull Algorithm - Graham Scan and Jarvis March tutorial, Youtube Lectures. Available=> https://www.youtube.com/watch?v=B2AJoQSZf4M
- The Quickhull Algorithm, Youtube Lectures

  https://www.youtube.com/watch?v=2EKIZrimeuk