Algorithm Analysis (II)

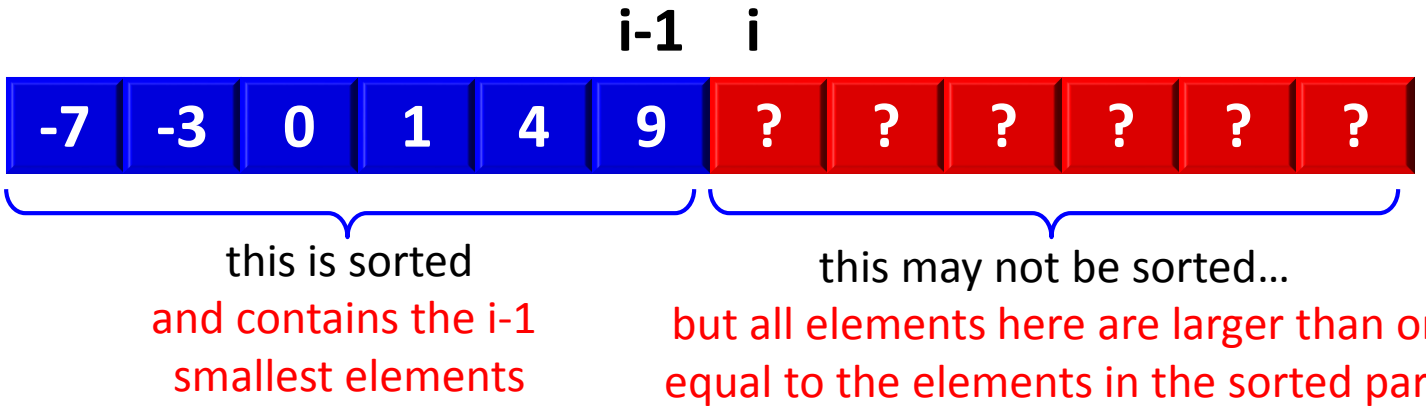
Jordi Cortadella and Jordi Petit
Department of Computer Science

Examples

- Selection sort
- Insertion sort
- The Maximum Subsequence Sum Problem
- Convex Hull

Selection Sort

- Selection sort uses this invariant:



Selection Sort

```
void selection_sort(vector<elem>& v) {
    int last = v.size() - 1;                    // v.size() = n
    for (int i = 0; i < last; ++i) {            // 0..n-2
        int k = i;
        for (int j = i + 1; j <= last; ++j) { // i+1..n-1
            if (v[j] < v[k]) k = j;
        }
        swap(v[k], v[i]);
    }
}
```

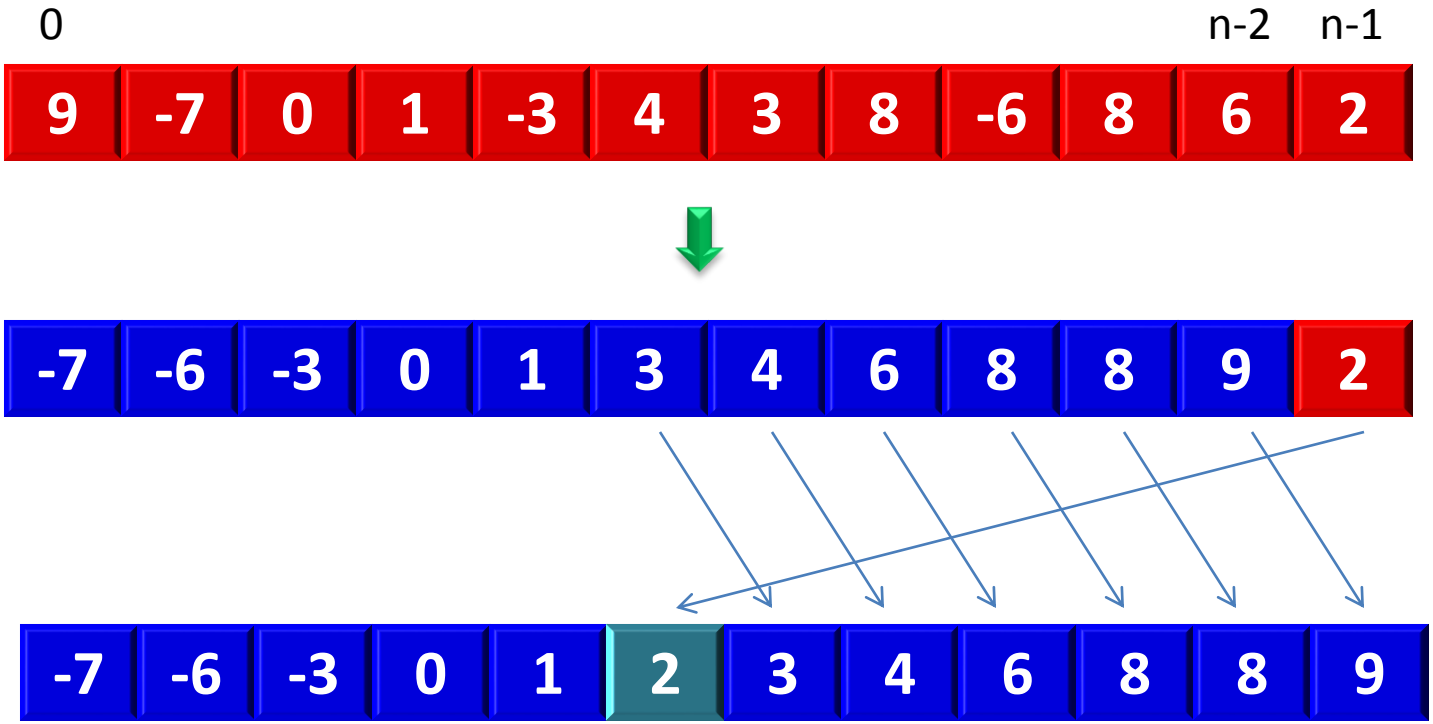
$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \mathrm{O}(1) = \mathrm{O}(1) \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \mathrm{O}(1) \sum_{i=0}^{n-2} (n-i-1)$$

$$= \mathrm{O}(1) \left(\frac{n}{2}(n-1) \right) = \mathrm{O}(1) \cdot \mathrm{O}(n^2) = \mathrm{O}(n^2)$$

Observation: notice that $T(n) \in \Omega(n^2)$, also. Therefore, $T(n) \in \Theta(n^2)$.

Insertion Sort

- Let us use inductive reasoning:
  - If we know how to sort arrays of size n-1,
  - do we know how to sort arrays of size n?



Insertion Sort

```
void insertion_sort(vector<elem>& v) {
    for (int i = 1; i < v.size(); ++i) {  // n-1 times
        elem x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > x) {  // 0..i times
            v[j] = v[j - 1];
            --j;
        }
        v[j] = x;
    }
}
```

$$\boxed{\begin{aligned} T(n) &= \Omega(n) \\ T(n) &= \mathrm{O}(n^2) \end{aligned}}$$

$$T_{\text{worst}}(n) = \sum_{i=1}^{n-1} i \cdot \mathrm{O}(1) = \mathrm{O}(n^2) \qquad \Rightarrow \text{sorted in reverse order}$$

$$T_{\text{best}}(n) = \sum_{i=1}^{n-1} \mathrm{O}(1) = \mathrm{O}(n) \qquad \Rightarrow \text{already sorted}$$

The Maximum Subsequence Sum Problem

- Given (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum value of $\sum_{k=i}^{j} A_k$.
  (the max subsequence sum is 0 if all integers are negative).

- Example:
  - Input: -2, 11, -4, 13, -5, -2
  - Answer: 20 (subsequence 11, -4, 13)

(extracted from M.A. Weiss, Data Structures and Algorithms in C++, Pearson, 2014, 4th edition)

The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {
  int maxSum = 0;
  // try all possible subsequences
  for (int i = 0; i < a.size(); ++i)
    for (int j = i; j < a.size(); ++j) {
      int thisSum = 0;
      for (int k = i; k <= j; ++k) thisSum += a[k];
      if (thisSum > maxSum) maxSum = thisSum;
    }
  return maxSum;
}
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1$$

The Maximum Subsequence Sum Problem

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$$

$$= \sum_{i=0}^{n-1} \frac{(n - i + 1)(n - i)}{2} = \cdots$$

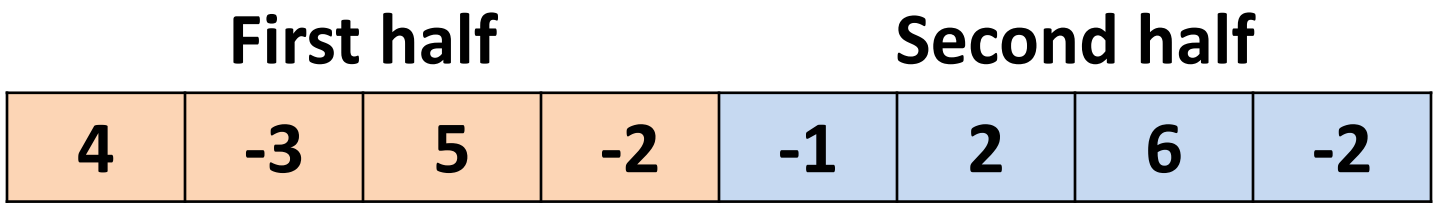
$$= \frac{n^3 + 3n^2 + 2n}{6} = \mathrm{O}(n^3)$$

The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {
  int maxSum = 0;
  // try all possible subsequences
  for (int i = 0; i < a.size(); ++i) {
    int thisSum = 0;
    for (int j = i; j < a.size(); ++j) {
      thisSum += a[j]; // reuse computation
      if (thisSum > maxSum) maxSum = thisSum;
    }
  }
  return maxSum;
}
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = \mathrm{O}(n^2)$$

Max Subsequence Sum: Divide&Conquer



The max sum can be in one of three places:

- 1st half
- 2nd half
- Spanning both halves and crossing the middle

In the 3rd case, two max subsequences must be found starting from the center of the vector (one to the left and the other to the right)

Max Subsequence Sum: Divide&Conquer

```
int maxSumRec(const vector<int>& a,
              int left, int right) {
  // base cases
  if (left == right)
    if (a[left] > 0) return a[left];
    else return 0;

  // Recursive cases: left and right halves
  int center = (left + right)/2;
  int maxLeft = maxSumRec(a, left, center);
  int maxRight = maxSumRec(a, center + 1, right);

                    :
```

Max Subsequence Sum: Divide&Conquer

```
                                    ⋮
  int maxRCenter = 0, rightSum = 0;
  for (int i = center; i >= left; --i) {
    rightSum += a[i];
    if (rightSum > maxRCenter) maxRCenter = rightSum;
  }

  int maxLCenter = 0, leftSum = 0;
  for (int i = center + 1; i <= right; ++i) {
    leftSum += a[i];
    if (leftSum > maxLCenter) maxLCenter = leftSum;
  }

  int maxCenter = maxRCenter + maxLCenter;
  return max3(maxLeft, maxRight, maxCenter);
}
```

Max Subsequence Sum: Divide&Conquer

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + \mathrm{O}(n) \end{aligned}$$

We will see how to solve this equation formally in the next lesson (Master Theorem). Informally:

$$\begin{aligned} T(n) &= 2T(n/2) + n = 2\left(2(T(n/4) + n/2)\right) + n \\ &= 4T(n/4) + n + n = 8T(n/8) + n + n + n = \cdots \\ &= 2^k T(n/2^k) + \underbrace{n + n + \cdots + n}_{k} \end{aligned}$$

when $n = 2^k$ we have that $k = \log_2 n$

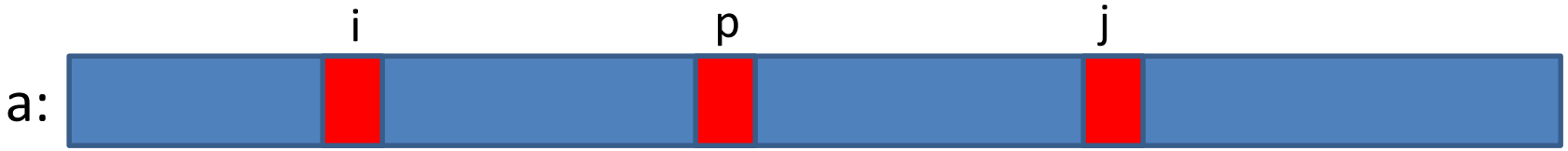
$$T(n) = 2^k T(1) + kn = n + n\log_2 n = \mathrm{O}(n \log n)$$

But, can we still do it faster?

The Maximum Subsequence Sum Problem

- Observations:
  - If a[i] is negative, it cannot be the start of the optimal subsequence.
  - Any negative subsequence cannot be the prefix of the optimal subsequence.

- Let us consider the inner loop of the $O(n^2)$ algorithm and assume that all prefixes of a[i..j-1] are positive and a[i..j] is negative:



  - If p is an index between i+1 and j, then any subsequence from a[p] is not larger than any subsequence from a[i] and including a[p-1].
  - If a[j] makes the current subsequence negative, we can advance i to j+1.

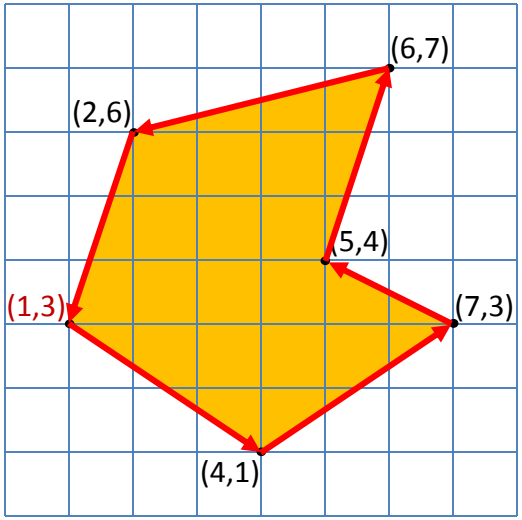
The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {
  int maxSum = 0, thisSum = 0;
  for (int i = 0; i < a.size(); ++i) {
    int thisSum += a[i];
    if (thisSum > maxSum) maxSum = thisSum;
    else if (thisSum < 0) thisSum = 0;
  }
  return maxSum;
}
```

$$T(n) = \mathrm{O}(n)$$

a:	4	-3	5	-4	-3	-1	5	-2	6	-3	2
thisSum:	4	1	6	2	0	0	5	3	9	6	8
maxSum:	4	4	6	6	6	6	6	6	9	9	9

Representation of polygons

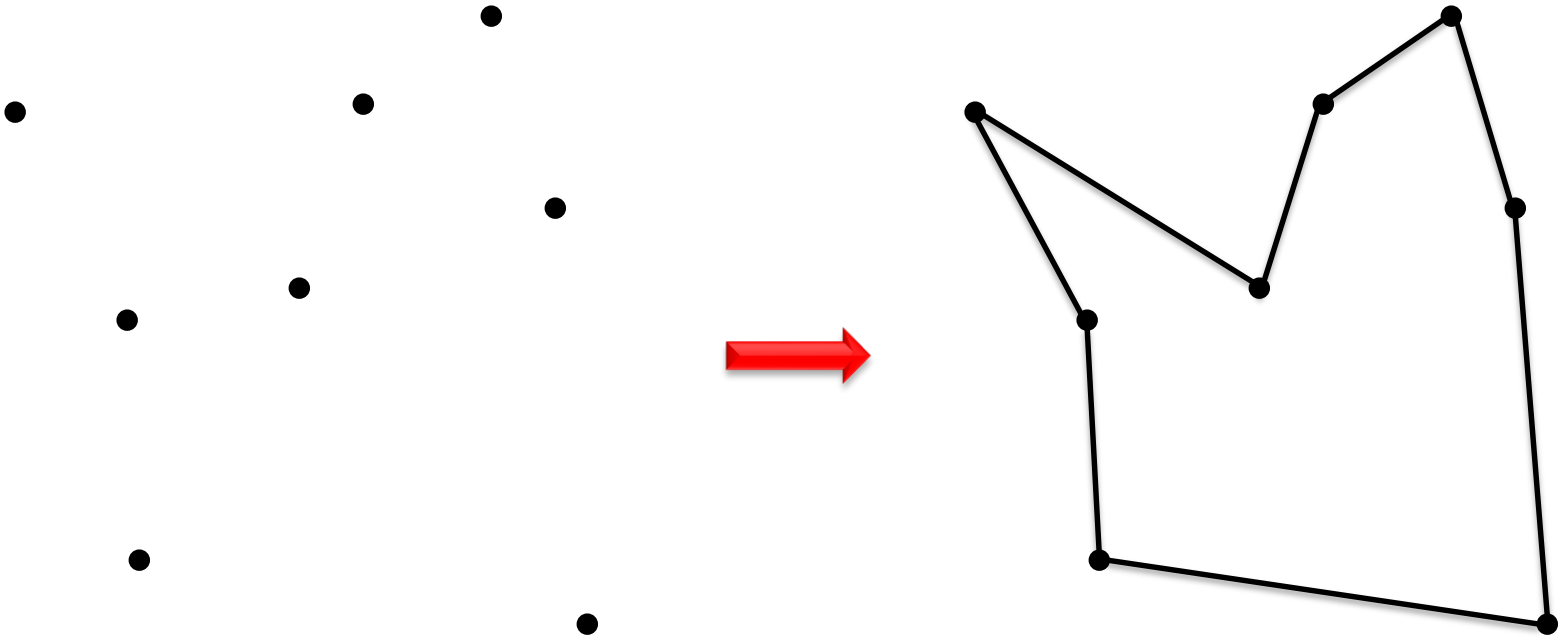


- A polygon can be represented by a sequence of vertices.
- Two consecutive vertices represent an edge of the polygon.
- The last edge is represented by the first and last vertices of the sequence.

```
Vertices:  (1,3) (4,1) (7,3) (5,4) (6,7) (2,6)

Edges: (1,3)-(4,1)-(7,3)-(5,4)-(6,7)-(2,6)-(1,3)
```

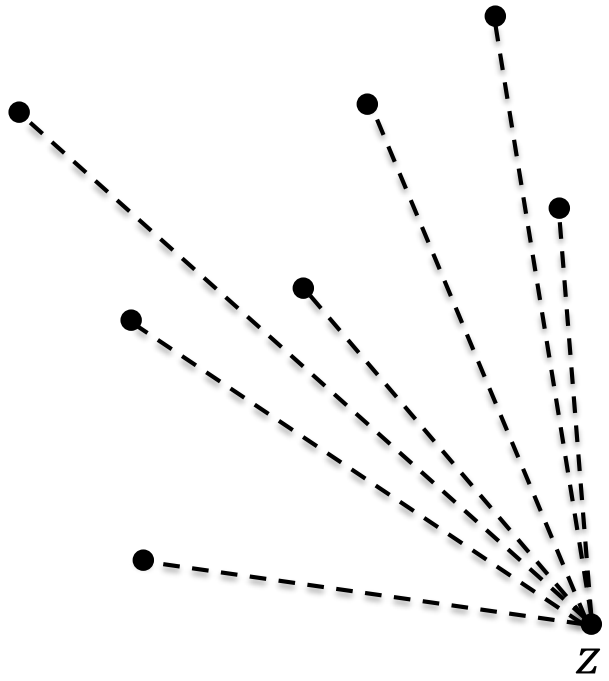
```
// A polygon (an ordered set of vertices)
using Polygon = vector<Point>;
```

Create a polygon from a set of points



Given a set of n points in the plane, connect them in a simple closed path.

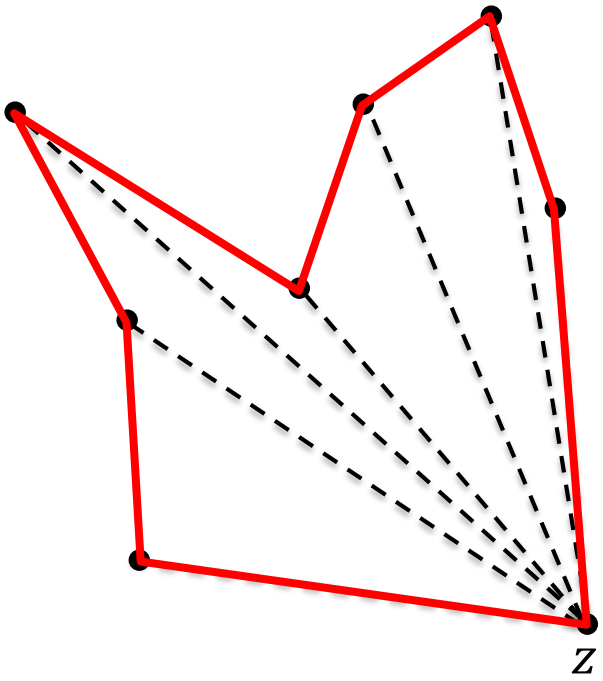
Simple polygon



- **Input:** $p_1, p_2, \ldots, p_n$ (points in the plane).
- **Output:** P (a polygon whose vertices are $p_1, p_2, \ldots, p_n$ in some order).

- Select a point z with the largest x coordinate (and smallest y in case of a tie in the x coordinate). Assume $z = p_1$.
- For each $p_i \in \{p_2, \ldots, p_n\}$, calculate the angle α_i between the lines $z - p_i$ and the x axis.
- Sort the points $\{p_2, \ldots, p_n\}$ according to their angles. In case of a tie, use distance to z.

Simple polygon

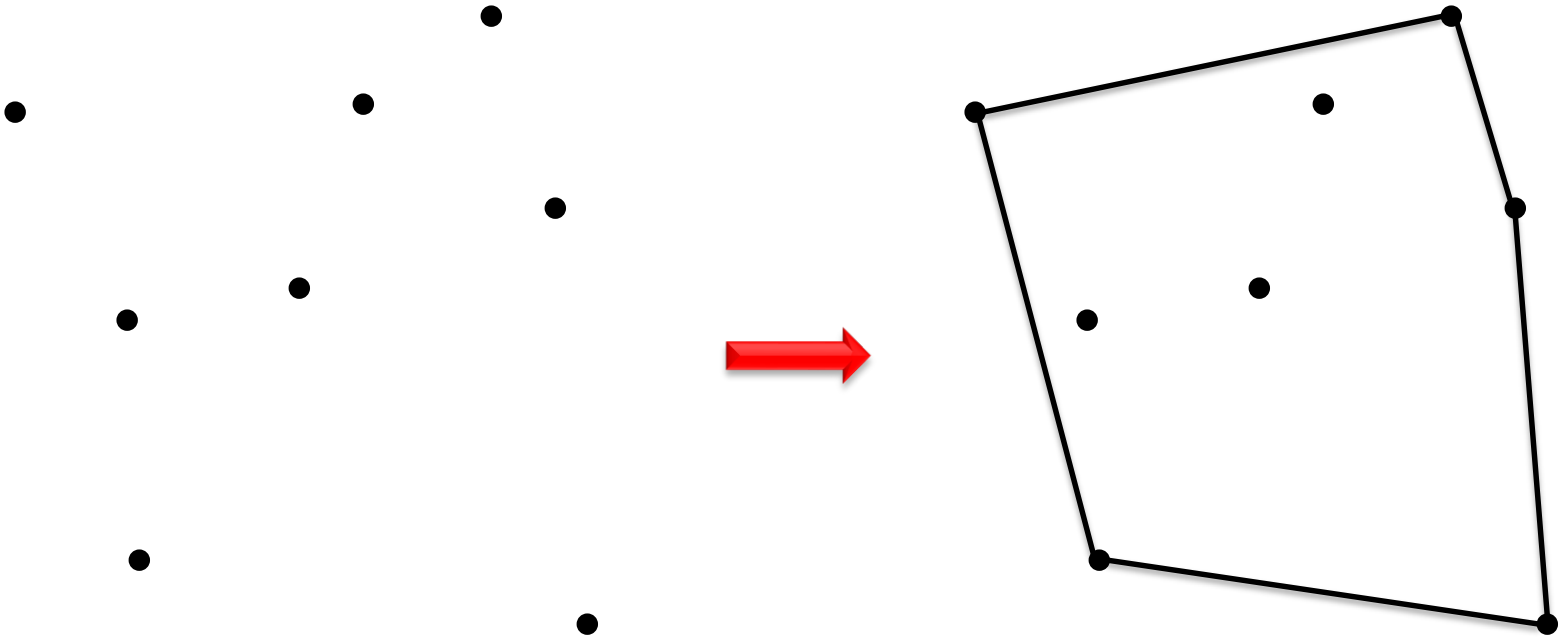


Implementation details:

- There is no need to calculate angles (requires arctan). It is enough to calculate slopes ($\Delta y/\Delta x$).
- There is not need to calculate distances. It is enough to calculate the square of distances (no sqrt required).

Complexity: $\mathrm{O}(n \log n)$.
The runtime is dominated by the sorting algorithm.

Convex hull

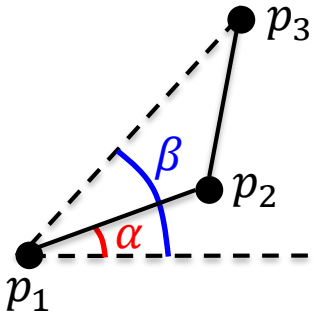


Compute the convex hull of n given points in the plane.

Clockwise and counter-clockwise

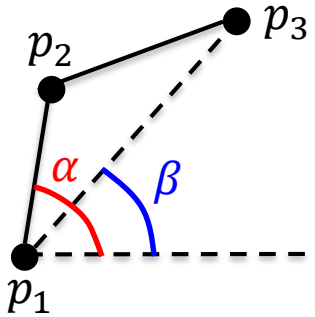
How to calculate whether three consecutive vertices are in a **clockwise** or ***counter-clockwise*** turn.

counter-clockwise
(p_3 at the left of $\overrightarrow{p_1p_2}$)



$\alpha < \beta$

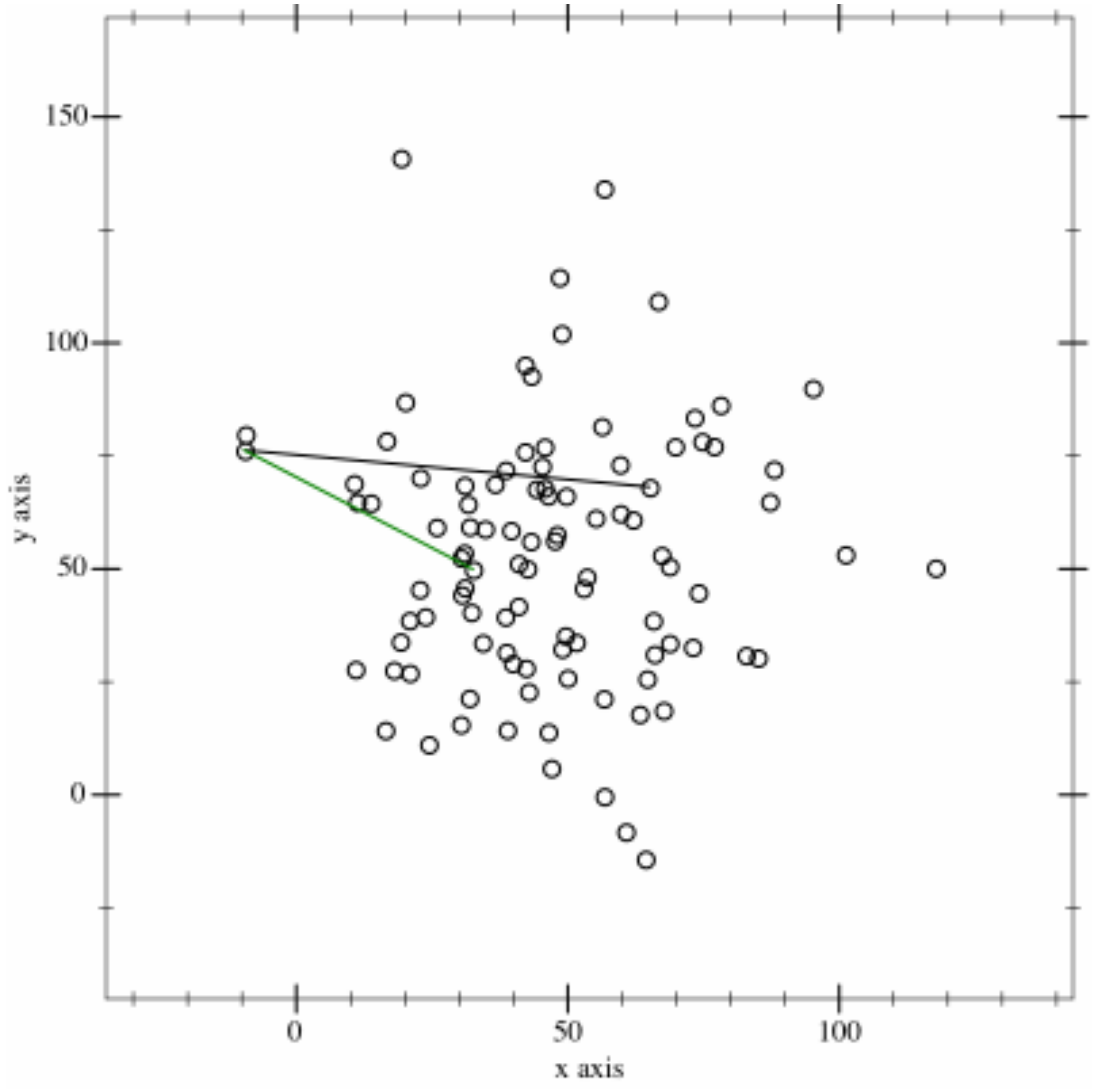
clockwise
(p_3 at the right of $\overrightarrow{p_1p_2}$)



$\alpha > \beta$

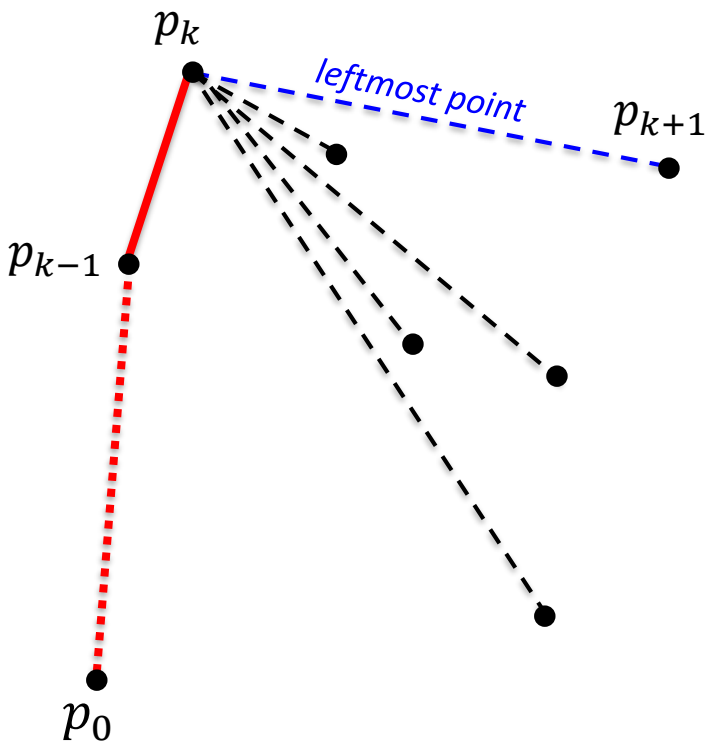
```
// Returns true if p3 is at the left of p1p2
bool leftof(p1, p2, p3) {
  return (p2.x − p1.x) · (p3.y − p1.y) > (p2.y − p1.y) · (p3.x − p1.x);
}
```

Convex hull: gift wrapping algorithm



https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

Convex hull: gift wrapping algorithm



- **Input:** $p_1, p_2, \ldots, p_n$ (points in the plane).
- **Output:** P (the convex hull of $p_1, p_2, \ldots, p_n$).

- **Initial points:**
  p_0 with the smallest x coordinate.

- **Iteration:** Assume that a partial path with k points has been built (p_k is the last point). Pick some arbitrary $p_{k+1} \neq p_k$. Visit the remaining points. If some point q is at the left of $\overrightarrow{p_k p_{k+1}}$ redefine $p_{k+1} = q$.

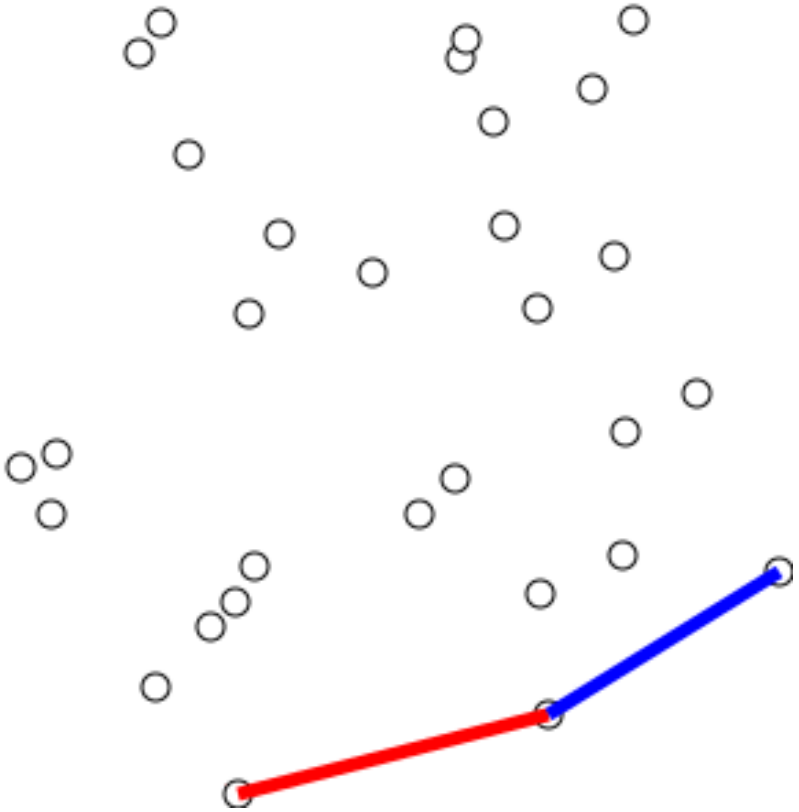
- Stop when P is complete (back to point p_0).

Complexity: At each iteration, we calculate n angles. $T(n) = O(hn)$, where h is the number of points in the convex hull. In the worst case, $T(n) = O(n^2)$.

Convex hull: gift wrapping algorithm

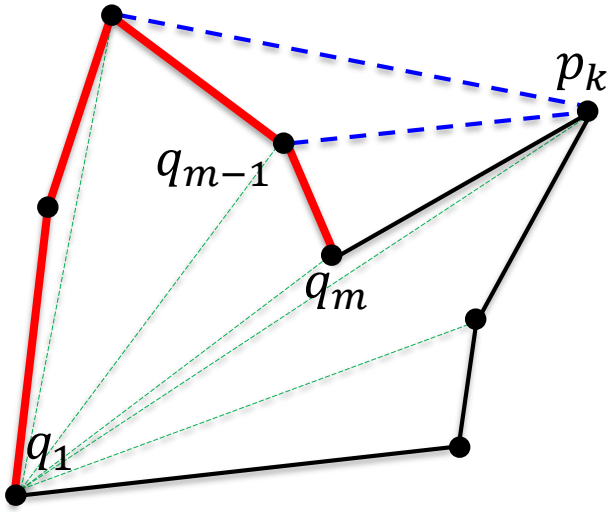
```
vector<Point> convexHull(vector<Point> points) {
  int n = points.size();
  vector<Point> hull;

  // Pick the leftmost point
  int left = 0;
  for (int i = 1; i < n; i++)
    if (points[i].x < points[left].x) left = i;

  int p = left;
  do {
    hull.push_back(points[p]);  // Add point to the convex hull

    int q = (p + 1)%n; // Pick a point different from p
    for (int i = 0; i < n; i++)
      if (leftof(points[p], points[q], points[i])) q = i;

    p = q; // Leftmost point for the convex hull
  } while (p != left);  // While not closing polygon

  return hull;
}
```

Convex hull: Graham Scan



https://en.wikipedia.org/wiki/Graham_scan

Convex hull: Graham scan



Input: $p_1, p_2, \dots, p_n$ (points in the plane).
Output: $q_1, q_2, \dots, q_m$ (the convex hull).

Initially:
Create a simple polygon P (complexity $O(n \log n)$).
Assume the order of the points is $p_1, p_2, \dots, p_n$.

```
// Q = (q1, q2, ...) is a vector where the points
// of the convex hull will be stored.
q1 = p1; q2 = p2; q3 = p3; m = 3;
for k = 4 to n:
  while leftof(qm-1, qm, pk): m = m − 1;
  m = m + 1;
  qm = pk;
```

Observation: each point p_k can be included in Q and deleted at most once.
The main loop of Graham scan has linear cost.
Complexity: dominated by the creation of the simple polygon → $O(n \log n)$.

EXERCISES

Summations

Prove the following equalities:

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

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

For loops: analyze the cost of each code

```
// Code 1
int s = 0;
for (int i = 0; i < n; ++i) ++s;

// Code 2
int s = 0;
for (int i = 0; i < n; i += 2) ++s;

// Code 3
int s = 0;
for (int i = 0; i < n; ++i) ++s;
for (int j = 0; j < n; ++j) ++s;

// Code 4
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) ++s;
}

// Code 5
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i; ++j) ++s;
}
```

For loops: analyze the cost of each code

```
// Code 6
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = i; j < n; ++j) ++s;
}

// Code 7
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) ++s;
  }
}

// Code 8
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i; ++j) {
    for (int k = 0; k < j; ++k) ++s;
  }
}
```

For loops: analyze the cost of each code

```
// Code 9
int s = 0;
for (int i = 1; i <= n; i *= 2) ++s;

// Code 10
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i*i; ++j) {
    for (int k = 0; k < n; ++k) ++s;
  }
}

// Code 11
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i*i; ++j) {
    if (j%i == 0) {
      for (int k = 0; k < n; ++k) ++s;
    }
  }
}
```

O, Ω or Θ?

The following statements refer to the ***insertion sort*** algorithm and the X's hide an occurrence of O, Ω or Θ. For each statement, find which options for $X \in \{O, \Omega, \Theta\}$ make the statement true or false. Justify your answers.

1. The worst case is $X(n^2)$
2. The worst case is $X(n)$
3. The best case is $X(n^2)$
4. The best case is $X(n)$
5. For every probability distribution, the average case is $X(n^2)$
6. For every probability distribution, the average case is $X(n)$
7. For some probability distribution, the average case is $X(n \log n)$

Primality

The following algorithms try to determine whether $n \geq 0$ is prime.
Find which ones are correct and analyze their cost as a function of n.

```
bool isPrime1(int n) {
  if (n <= 1) return false;
  for (int i = 2; i < n; ++i) if (n%i == 0) return false;
  return true;
}

bool isPrime2(int n) {
  if (n <= 1) return false;
  for (int i = 2; i*i < n; ++i) if (n%i == 0) return false;
  return true;
}

bool isPrime3(int n) {
  if (n <= 1) return false;
  for (int i = 2; i*i <= n; ++i) if (n%i == 0) return false;
  return true;
}

bool isPrime4(int n) {
  if (n <= 1) return false;
  if (n == 2) return true;
  if (n%2 == 0) return false;
  for (int i = 3; i*i <= n; i += 2) if (n%i == 0) return false;
  return true;
}
```

The Sieve of Eratosthenes

The following program is a version of the Sieve of Eratosthenes. Analyze its complexity.

```
vector<bool> Primes(int n) {
  vector<bool> p(n + 1, true);
  p[0] = p[1] = false;
  for (int i = 2; i*i <= n; ++i) {
    if (p[i]) {
      for (int j = i*i; j <= n; j += i) p[j] = false;
    }
  }
  return p;
}
```

You can use the following equality, where $p \leq x$ refers to all primes $p \leq x$:

$$\sum_{p \leq x} \frac{1}{p} = \log\log x + O(1)$$

The Cell Phone Dropping Problem

n

- You work for a cell phone company which has just invented a new cell phone protector and wants to advertise that it can be dropped from the f^{th} floor without breaking.
- If you are given 1 or 2 phones and an n story building, propose an algorithm that minimizes the worst-case number of trial drops to know the highest floor it won't break.
- Assumption: a broken cell phone cannot be used for further trials.
- How about if you have p cell phones?

(Source: Wood & Yasskin, Texas A&M University)