Algorithm Analysis

Jordi Cortadella and Jordi Petit
Department of Computer Science

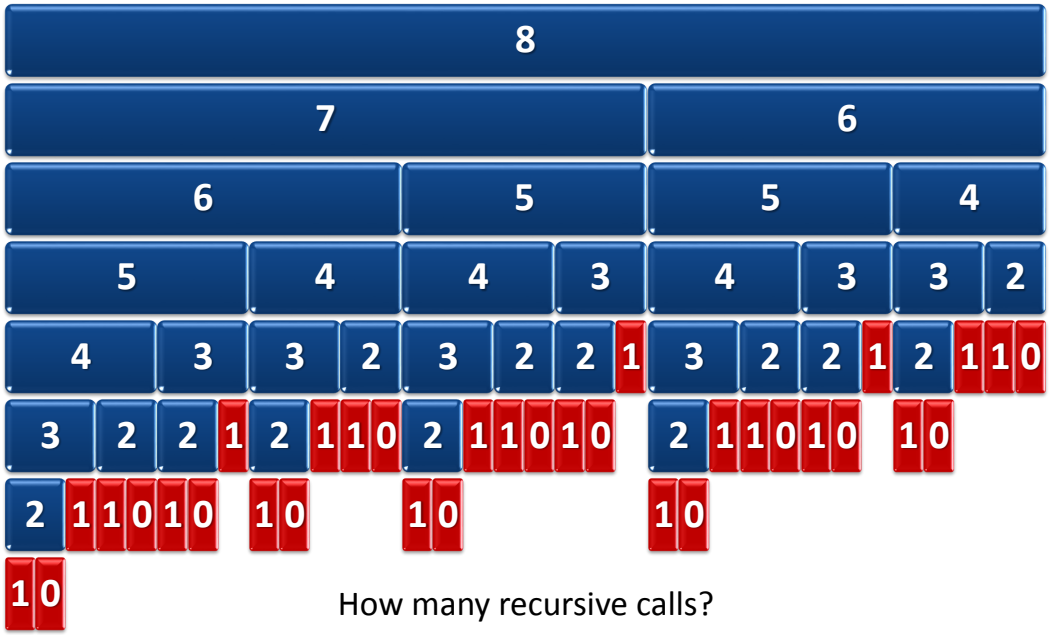
What do we expect from an algorithm?

- Correct
- Easy to understand
- Easy to implement
- Efficient:
  - Every algorithm requires a set of resources
    - Memory
    - CPU time
    - Energy

Fibonacci: recursive version

```
// Pre: n ≥ 0
// Returns the Fibonacci number of order n.

int fib(int n) { // Recursive solution
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```


Fibonacci



How many recursive calls?

Fibonacci: runtime

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = T(n-1) + T(n-2)$$

Let us assume that $T(n) = a^n$ for some constant a. Then,

$$a^n = a^{n-1} + a^{n-2} \quad \Rightarrow \quad a^2 = a + 1$$

$$a = \frac{1+\sqrt{5}}{2} = \varphi \approx 1.618 \qquad \text{(golden ratio)}$$

Therefore, $T(n) \approx 1.6^n$.

If $T(0) = 1$ ns, then $T(100) \approx 2.6 \cdot 10^{20}$ns > 8000 yrs.

With the age of Universe ($14 \cdot 10^9$ yrs),
we could compute up to `fib(128)`.

Fibonacci numbers: iterative version

```
// Pre: n ≥ 0
// Returns the Fibonacci number of order n.
int fib(int n) { // iterative solution
    int f_i = 0;
    int f_i1 = 1;
    // Inv: f_i is the Fibonacci number of order i.
    //      f_i1 is the Fibonacci number of order i+1.
    for (int i = 0; i < n; ++i) {
        int f = f_i + f_i1;
        f_i = f_i1;
        f_i1 = f;
    }
    return f_i;
}
```

Runtime: **n** iterations

Fibonacci numbers

Algebraic solution: find matrix A such that

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = A^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Fibonacci numbers

$$A^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \qquad A^8 = \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix}$$

$$A^{16} = \begin{bmatrix} 1597 & 987 \\ 987 & 610 \end{bmatrix} \quad \cdots A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Runtime ≈ *$\log_2 n$ 2x2 matrix multiplications*

Algorithm analysis

Given an algorithm that reads inputs from a domain D, we want to define a cost function C:

$$C : D \to \mathbb{R}^+$$
$$x \mapsto C(x)$$

where $C(x)$ represents the cost of using some resource (CPU time, memory, energy, …).

Analyzing $C(x)$ for every possible x is impractical.

Algorithm analysis: simplifications

- Analysis based on the size of the input: $|x| = n$
- Only the best/average/worst cases are analyzed:

$$\begin{aligned} C_{\text{worst}}(n) &= \max\{C(x) : x \in D, |x| = n\} \\ C_{\text{best}}(n) &= \min\{C(x) : x \in D, |x| = n\} \\ C_{\text{avg}}(n) &= \sum_{x \in D, |x| = n} p(x) \cdot C(x) \end{aligned}$$

$p(x)$: probability of selecting input x among all the inputs of size n.

Algorithm analysis

- Properties:

$$\begin{aligned} \forall n \geq 0 : &\quad C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n) \\ \forall x \in D : &\quad C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|) \end{aligned}$$

- We want a notation that characterizes the cost of algorithms independently from the technology (CPU speed, programming language, efficiency of the compiler, etc.).
- Runtime is usually the most important resource to analyze.

Asymptotic notation

Let us consider all functions $f : \mathbb{R}^+ \to \mathbb{R}^+$

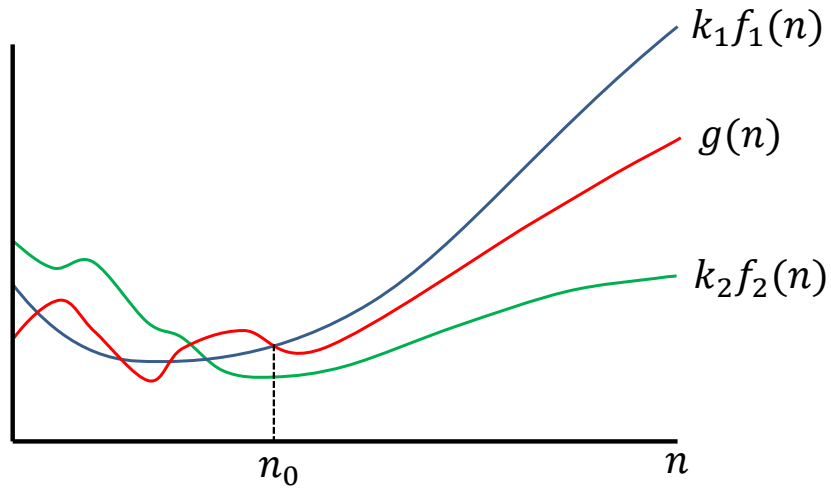
Definitions:

$$\mathrm{O}\big(f(n)\big) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \leq k \cdot f(n)\}$$

$$\Omega(f(n)) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \geq k \cdot f(n)\}$$

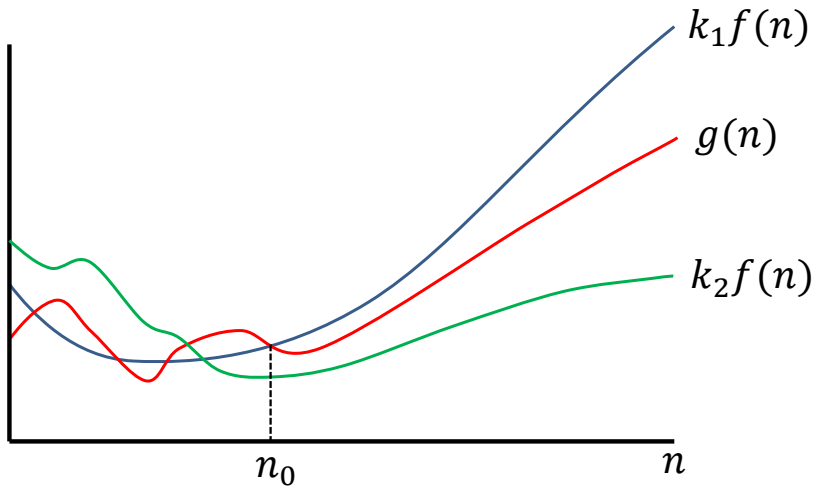
$$\Theta(f(n)) = \mathrm{O}\big(f(n)\big) \cap \Omega(f(n))$$

Asymptotic notation



$$g(n) \in \mathrm{O}\big(f_1(n)\big)$$
$$g(n) \in \Omega(f_2(n))$$

Asymptotic notation



$$g(n) \in \Theta(f(n))$$

Asymptotic notation: example

$$\begin{aligned}\mathrm{O}(n^2) = \{&3n^2 - n + 20,\\ &0.5n^2 - 3,\\ &4n\log_2 n,\\ &2n - 5,\\ &n < 20\ ?\ 2^n : 2n^2 + 1000,\\ &\dots\\ \}&\end{aligned}$$

$$\begin{aligned}\Omega(n^2) = \{&3n^2 - n + 20,\\ &0.5n^2 - 3,\\ &3n^3\log_2 n,\\ &2^n - 4n^3,\\ &n < 20\ ?\ 2^n : 2n^2 + 1000,\\ &\dots\\ \}&\end{aligned}$$

$$\begin{aligned}\Theta(n^2) = \mathrm{O}(n^2) \cap \Omega(n^2) = \{&3n^2 - n + 20,\\ &0.5n^2 - 3,\\ &n < 20\ ?\ 2^n : 2n^2 + 1000,\\ &\dots\\ \}&\end{aligned}$$

Examples for Big-O and Big-Ω

$$13n^3 - 4n + 8 \in \mathrm{O}\big(n^3\big)$$
$$2n - 5 \in \mathrm{O}\big(n\big)$$
$$n^2 \notin \mathrm{O}\big(n\big)$$
$$2^n \in \mathrm{O}\big(n!\big)$$
$$3^n \notin \mathrm{O}\big(2^n\big)$$
$$3\log_2 n \in \mathrm{O}\big(\log n\big)$$
$$3n\log_2 n \in \mathrm{O}\big(n^2\big)$$
$$\mathrm{O}\big(n^2\big) \subseteq \mathrm{O}\big(n^3\big)$$

$$13n^3 - 4n + 8 \in \Omega(n^3)$$
$$n^2 \in \Omega(n)$$
$$n^2 \notin \Omega(n^3)$$
$$n! \in \Omega(2^n)$$
$$3^n \in \Omega(2^n)$$
$$3\log_2 n \in \Omega(\log n)$$
$$n\log_2 n \in \Omega(n)$$
$$\mathrm{O}\big(n^3\big) \subseteq \Omega(n^2)$$

Complexity ranking

Function	Common name
$n!$	factorial
2^n	exponential
$n^d,\ d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	quasi-linear
n	linear
$\sqrt{n}$	root - n
$\log n$	logarithmic
1	constant

The limit rule

Let us assume that L exists (may be ∞) such that:

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$\begin{cases} \text{if } L = 0 & \text{then} \quad f \in \mathrm{O}(g) \\ \text{if } 0 < L < \infty & \text{then} \quad f \in \Theta(g) \\ \text{if } L = \infty & \text{then} \quad f \in \Omega(g) \end{cases}$$

Note: If both limits are ∞ or 0, use L'Hôpital rule:

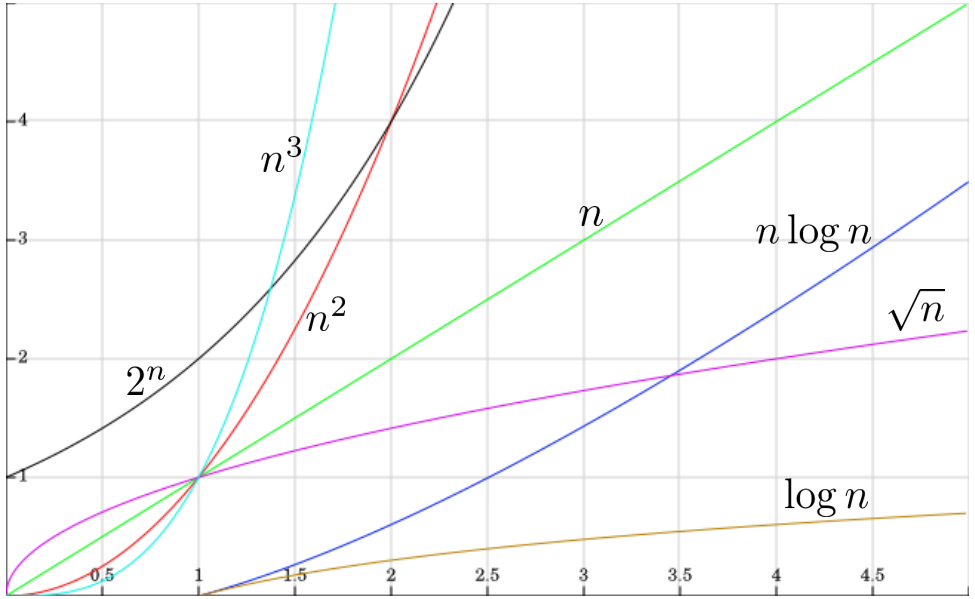
$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Properties

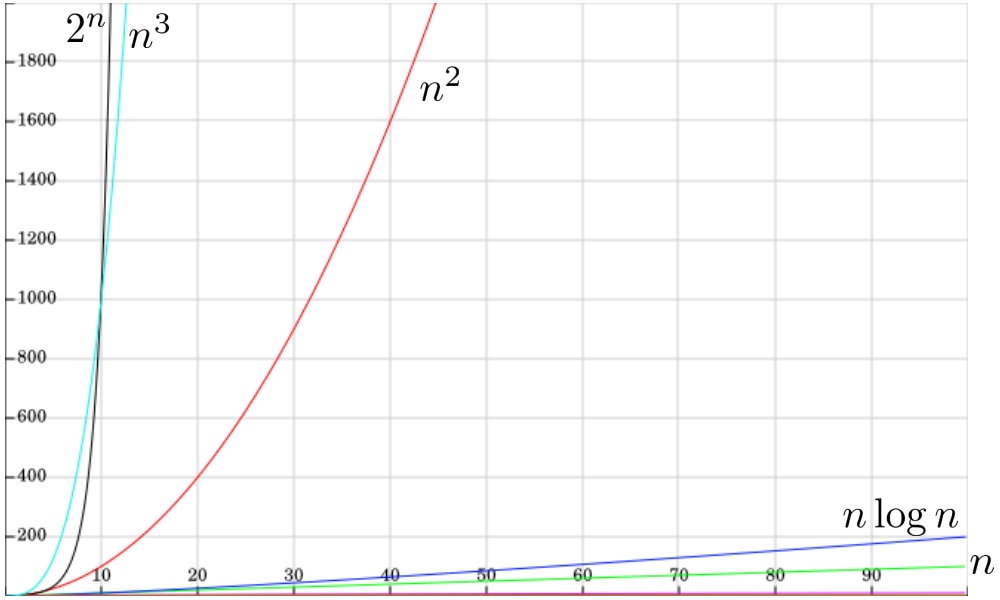
All rules (except the last one) also apply for Ω and Θ:

- $f \in \mathrm{O}(f)$
- $\forall c > 0, \quad \mathrm{O}(f) = \mathrm{O}(c \cdot f)$
- $f \in \mathrm{O}(g) \ \wedge \ g \in \mathrm{O}(h) \Rightarrow f \in \mathrm{O}(h)$
- $f_1 \in \mathrm{O}(g_1) \wedge \ f_2 \in \mathrm{O}(g_2)$
  $\Rightarrow f_1 + f_2 \in \mathrm{O}(g_1 + g_2) = \mathrm{O}(\max\{g_1, g_2\})$
- $f \in \mathrm{O}(g) \Rightarrow f + g \in \mathrm{O}(g)$
- $f_1 \in \mathrm{O}(g_1) \wedge f_2 \in \mathrm{O}(g_2) \Rightarrow f_1 \cdot f_2 \in \mathrm{O}(g_1 \cdot g_2)$
- $f \in \mathrm{O}(g) \Leftrightarrow g \in \Omega(f)$

Asymptotic complexity (small values)



Asymptotic complexity (larger values)



Execution time: example

Let us consider that every operation can be executed in 1 ns (10^{-9} s).

	Time		
Function	$n = 1,000$	$n = 10,000$	$n = 100,000$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

How about “big data”?

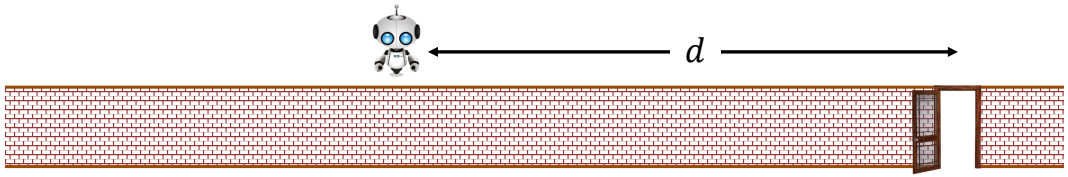
Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

This is often the practical limit for big data

The robot and the door in an infinite wall

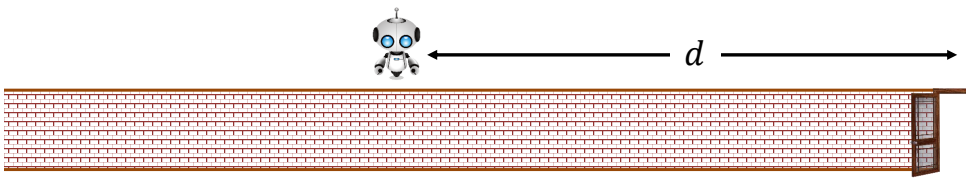


A robot stands in front of a wall that is infinitely long to the right and left side. The wall has a door somewhere and the robot has to find it to reach the other side. Unfortunately, the robot can only see the part of the wall in front of it.

The robot does not know neither how far away the door is nor what direction to take to find it. It can only execute moves to the left or right by a certain number of steps.

Let us assume that the door is at a distance d. How to find the door in a minimum number of steps?

The robot and the door in an infinite wall



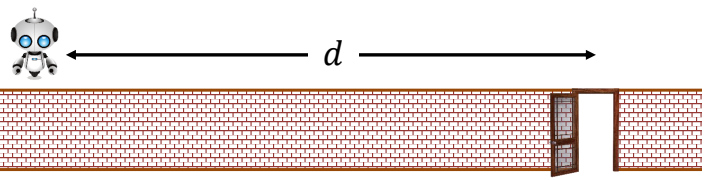
Algorithm 1:

- Pick one direction and move until the door is found.

Complexity:

- If the direction is correct → $O(d)$.
- If incorrect → the algorithm does not terminate.

The robot and the door in an infinite wall



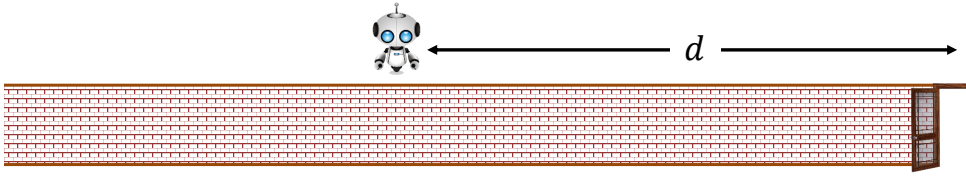
Algorithm 2:

- 1 step to the left,
- 2 steps to the right,
- 3 steps to the left, …
- … increasing by one step in the opposite direction.

Complexity:

$$T(d) = \sum_{i=1}^{2d} i = 2d\frac{1+2d}{2} = 2d^2 + d = O(d^2)$$

The robot and the door in an infinite wall



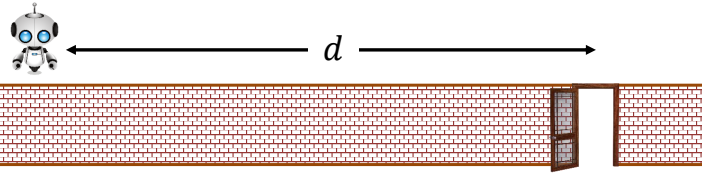
Algorithm 3:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 3 steps to the left and return to origin,…
- … increasing by one step in the opposite direction.

Complexity:

$$T(d) = d + 2\sum_{i=1}^{d} i = d + 2\frac{d(d+1)}{2} = d^2 + 2d = O(d^2)$$

The robot and the door in an infinite wall



Algorithm 4:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 4 steps to the left and return to origin,…
- … doubling the number of steps in the opposite direction.

Complexity (assume that $d = 2^n$)**:**

$$T(d) = d + 2\sum_{i=0}^{n} 2^i = d + 2(2^{n+1} - 1) = 5d - 2 = O(d)$$

Runtime analysis rules

- Variable declarations cost no time.
- ***Elementary operations*** are those that can be executed with a ***small number of basic computer steps*** (an assignment, a multiplication, a comparison between two numbers, etc.).
- Vector sorting or matrix multiplication are not elementary operations.
- We consider that the cost of elementary operations is $\mathrm{O}(1)$.

Runtime analysis rules

- Consecutive statements:
  - If S1 is $\mathrm{O}(f)$ and S2 is $\mathrm{O}(g)$, then S1;S2 is $\mathrm{O}(\max\{f, g\})$
- Conditional statements:
  - If S1 is $\mathrm{O}(f)$, S2 is $\mathrm{O}(g)$ and B is $\mathrm{O}(h)$, then if (B) S1; else S2; is $\mathrm{O}(\max\{f + h, g + h\})$, or also $\mathrm{O}(\max\{f, g, h\})$.

Runtime analysis rules

- For/While loops:
  - Running time is at most the running time of the statements inside the loop times the number of iterations
- Nested loops:
  - Analyze inside out: running time of the statements inside the loops multiplied by the product of the sizes of the loops

Nested loops: examples

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        DoSomething(); // O(1)
```

$\Longrightarrow \mathrm{O}(n^2)$

```
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
        DoSomething(); // O(1)
```

$\Longrightarrow \mathrm{O}(n^2)$

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        for (int k = 0; k < p; ++k)
            DoSomething(); // O(1)
```

$\Longrightarrow \mathrm{O}(n \cdot m \cdot p)$

Linear time: $\mathrm{O}(n)$

Running time proportional to input size

```
// Compute the maximum of a vector
// with n numbers

int m = a[0];
for (int i = 1; i < a.size(); ++i) {
    if (a[i] > m) m = a[i];
}
```


Linear time: $\mathrm{O}(n)$

Other examples:

– Reversing a vector

– Merging two sorted vectors

– Finding the largest null segment of a sorted vector:
a linear-time algorithm exists
(a null segment is a compact sub-vector in which the sum of all the elements is zero)

Logarithmic time: $\mathrm{O}(\log n)$

- Logarithmic time is usually related to divide-and-conquer algorithms
- Examples:
  – Binary search
  – Calculating x^n
  – Calculating the n-th Fibonacci number

Example: recursive x^y

```
// Pre: x ≠ 0, y ≥ 0
// Returns x^y
int power(int x, int y) {
    if (y == 0) return 1;
    if (y%2 == 0) return power(x*x, y/2);
    return x*power(x*x, y/2);
}

// Assumption: each */% takes O(1)
```

$$T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \quad \cdots$$

$$T(x^y) \leq \underbrace{4 + 4 + \cdots + 4}_{\log_2 y \ \ times} \quad \Longrightarrow \quad \mathrm{O}(\log y)$$

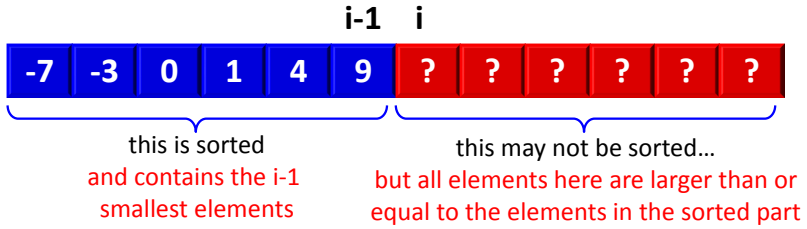
Linearithmic time: $O(n \log n)$

- ***Sorting:*** Merge sort and heap sort can be executed in $O(n \log n)$.

- ***Largest empty interval:*** Given n time-stamps $x_1, \cdots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?
  - $O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

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]);
    }
}
```

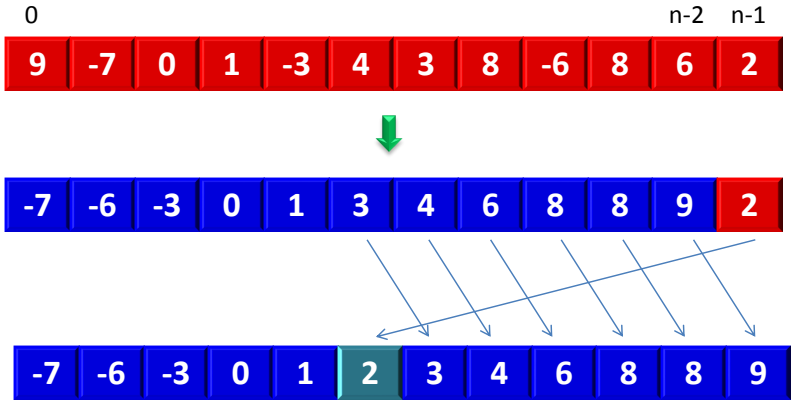
$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1) = O(1) \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = O(1) \sum_{i=0}^{n-2} (n-i-1)$$

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

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

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 \\ &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\ &= \sum_{i=0}^{n-1} \frac{(n-i+1)(n-i)}{2} = \cdots \\ &= \frac{n^3 + 3n^2 + 2n}{6} = \mathrm{O}(n^3) \end{aligned}$$

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

First half				Second half			
4	-3	5	-2	-1	2	6	-2

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:

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

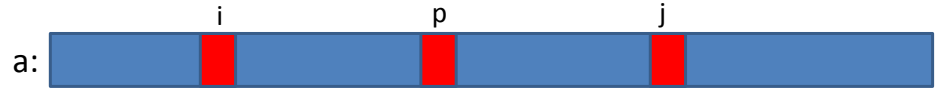
when $n = 2^k$ we have that $k = \log_2 n$

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

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 $\mathrm{O}(n^2)$ 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.

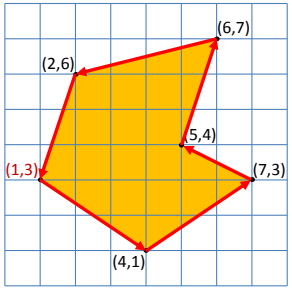
The Maximum Subsequence Sum Problem

```
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



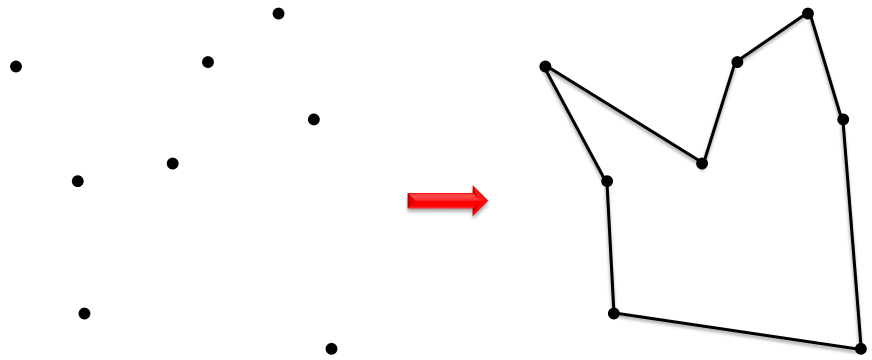
- A polygon can be represented by a sequence of vertices.
- Two consecutive vertices represent an edge of the polygon.
- The last edge is represented by the first and last vertices of the sequence.

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)

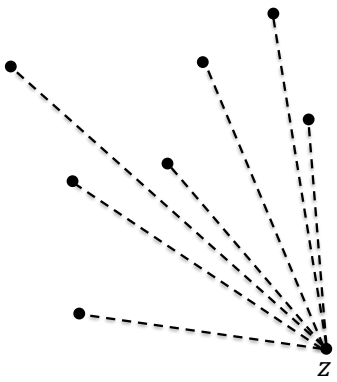
```
// A polygon (an ordered set of vertices)
using Polygon = vector<Point>;
```

Create a polygon from a set of points



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

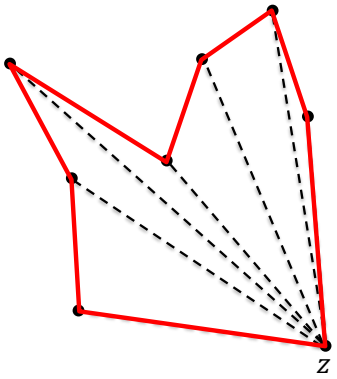
Simple polygon



- **Input:** $p_1, p_2, \dots, p_n$ (points in the plane).
- **Output:** P (a polygon whose vertices are $p_1, p_2, \dots, p_n$ 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 $p_i \in \{p_2, \dots, p_n\}$, calculate the angle α_i between the lines $z - p_i$ and the x axis.
- Sort the points $\{p_2, \dots, p_n\}$ according to their angles. In case of a tie, use distance to z.

Simple polygon



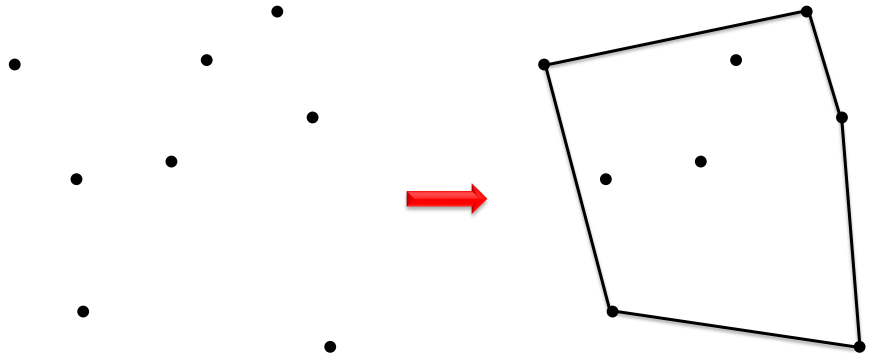
Implementation details:

- 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: $O(n \log n)$.
The runtime is dominated by the sorting algorithm.

Convex hull

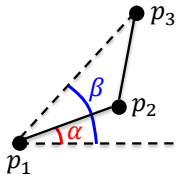


Compute the convex hull of n given points in the plane.

Clockwise and counter-clockwise

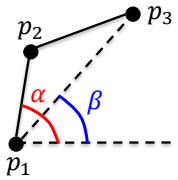
How to calculate whether three consecutive vertices are in a **clockwise** or ***counter-clockwise*** turn.

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



$\alpha < \beta$

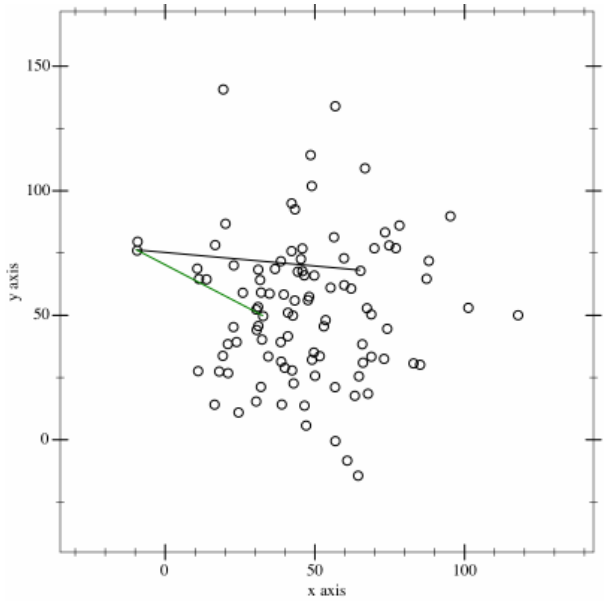
clockwise
(p_3 at the right of $\overrightarrow{p_1p_2}$)



$\alpha > \beta$

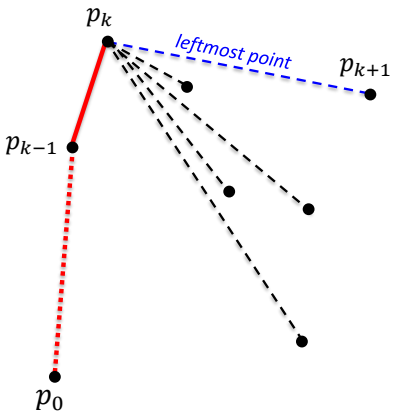
```
// Returns true if p3 is at the left of p1p2
bool leftof(p1,p2,p3) {
  return (p2.x − p1.x) · (p3.y − p1.y) > (p2.y − p1.y) · (p3.x − p1.x);
}
```


Convex hull: gift wrapping algorithm



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

Convex hull: gift wrapping algorithm



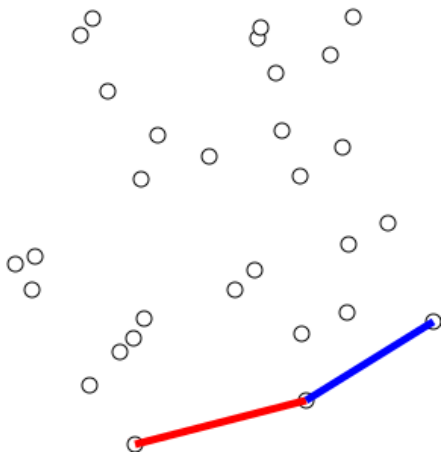
- **Input:** $p_1, p_2, \ldots, p_n$ (points in the plane).
- **Output:** P (the convex hull of $p_1, p_2, \ldots, p_n$).
- **Initial points:**
  p_0 with the smallest x coordinate.
- **Iteration:** Assume that a partial path with k points has been built (p_k is the last point). Pick some arbitrary $p_{k+1} \neq p_k$. Visit the remaining points. If some point q is at the left of $\overrightarrow{p_kp_{k+1}}$ 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

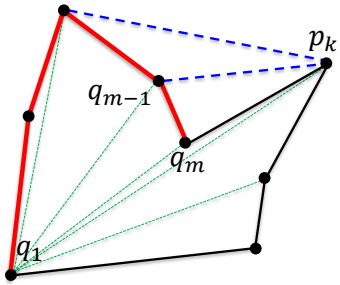
```
vector<Point> convexHull(vector<Point> points) {
  int n = points.size();
  vector<Point> hull;

  // 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)