Algorithm Analysis (I)

Jordi Cortadella and Jordi Petit
Department of Computer Science

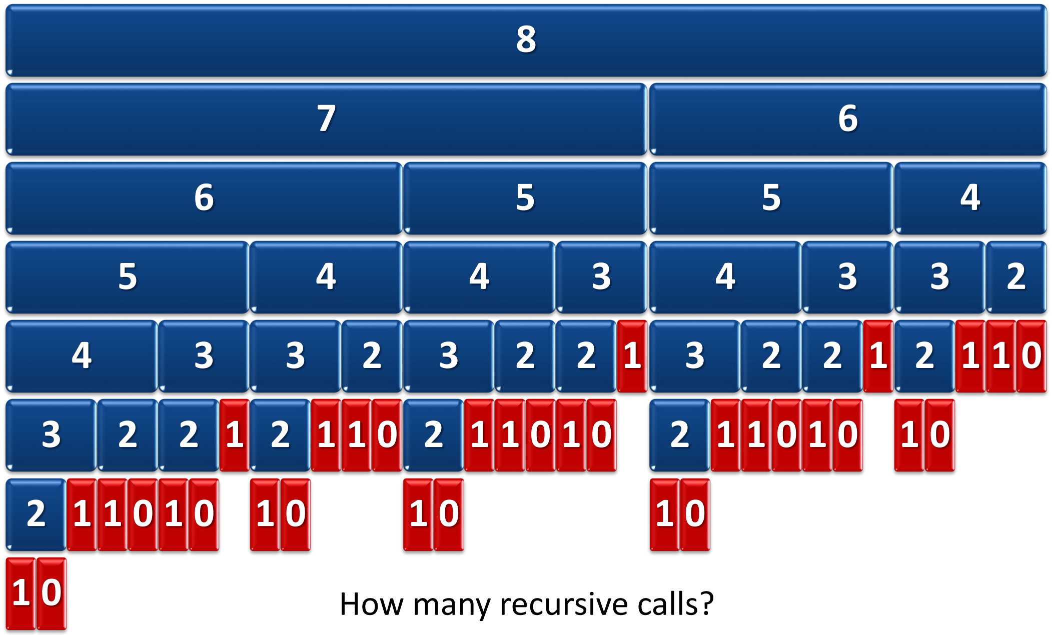
What do we expect from an algorithm?

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

Fibonacci: recursive version

```python
def fib(n: int) -> int:
    """Returns the Fibonacci number of order n
       Pre: n ≥ 0
    """
    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 $T(100) \approx 2.6 \cdot 10^{20}\text{ns} > 8000$ yrs.

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

Fibonacci numbers: iterative version

```
def fib(n: int) -> int:
    """Returns the Fibonacci number of order n
       Pre: n ≥ 0
    """
    f_i, f_i1 = 0, 1
    # Inv: f_i is the Fibonacci number of order i
    #      f_i1 is the Fibonacci number of order i+1
    for i in range(n):
        f_i, f_i1 = f_i1, f_i+f_i1
    return f_i
```

Runtime: **n** iterations

Fibonacci numbers

Algebraic solution: find matrix A such that

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

$$\Downarrow$$

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

$$\Downarrow$$

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

Fibonacci numbers

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

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

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

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 $f : \mathbb{R}^+ \to \mathbb{R}^+$

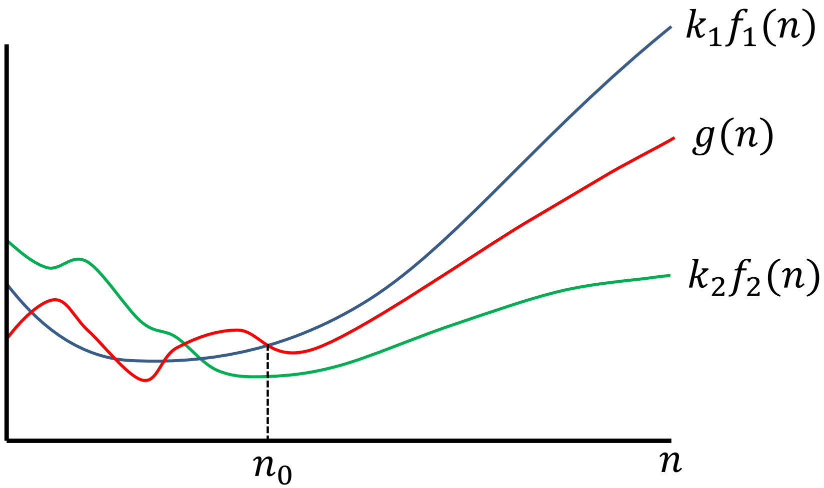
Definitions:

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

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

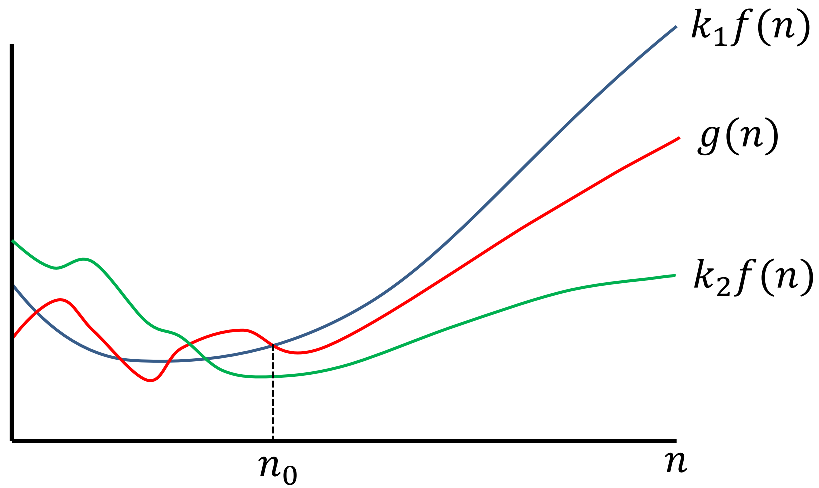
$$\Theta(f(n)) = \mathrm{O}\big(f(n)\big) \ \cap \ \Omega(f(n))$$

Asymptotic notation



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

Asymptotic notation



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

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

Big-O

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

Big-Ω

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

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:

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

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

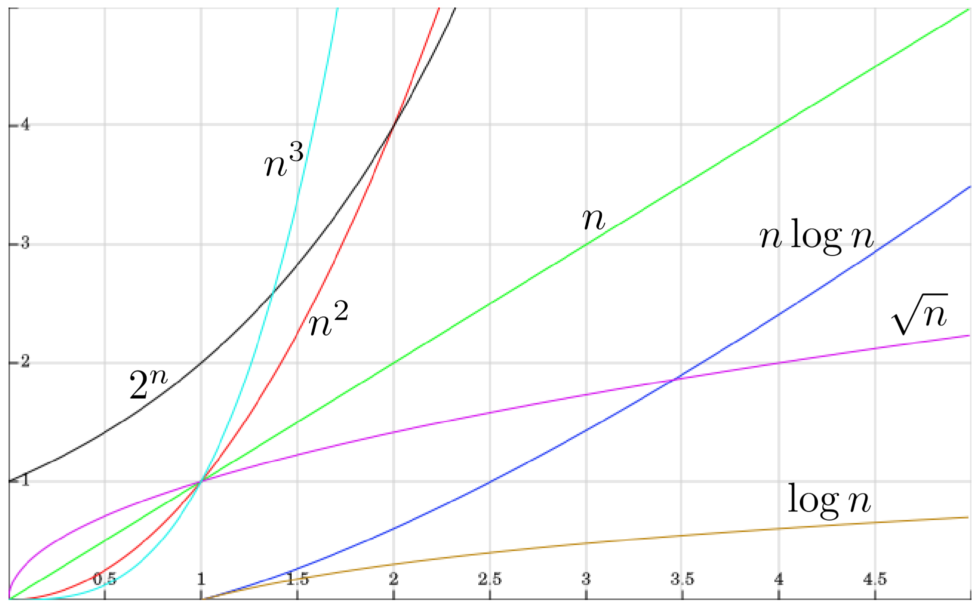
Note: If both limits are ∞ or 0, use L'Hôpital rule:

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

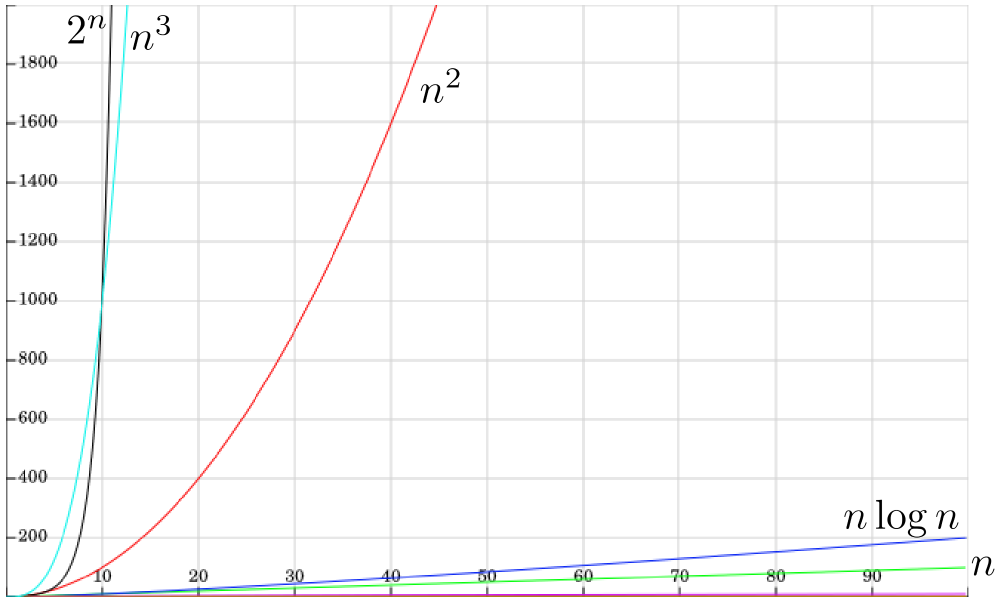
Properties

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

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"?

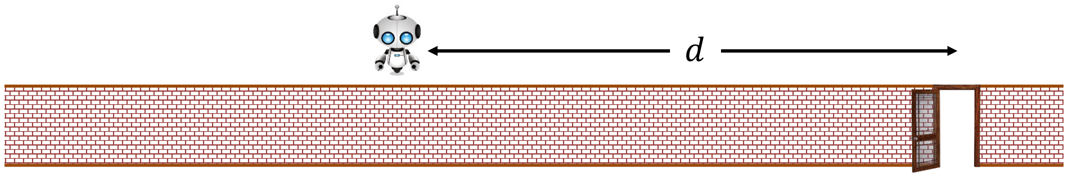
Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

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

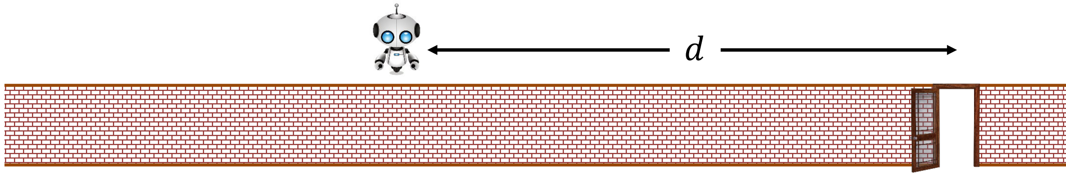


A robot stands in front of a wall that is infinitely long to the right and left side. The wall has a door somewhere and the robot has to find it to reach the other side. Unfortunately, the robot can only see the part of the wall in front of it.

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



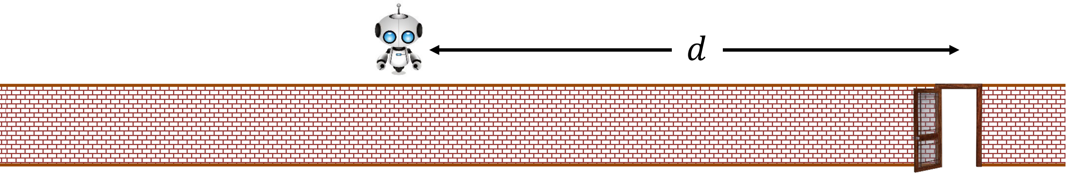
Algorithm 1:

- 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



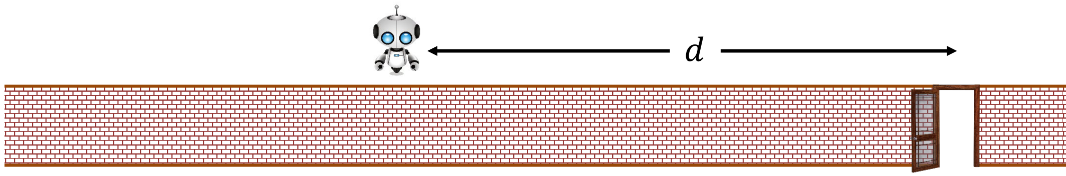
Algorithm 2:

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

Complexity:

$$T(d) = 3d + \sum_{i=1}^{d-1} 4i = 3d + 4\frac{d(d-1)}{2} = 2d^2 + d = O(d^2)$$

The robot and the door in an infinite wall



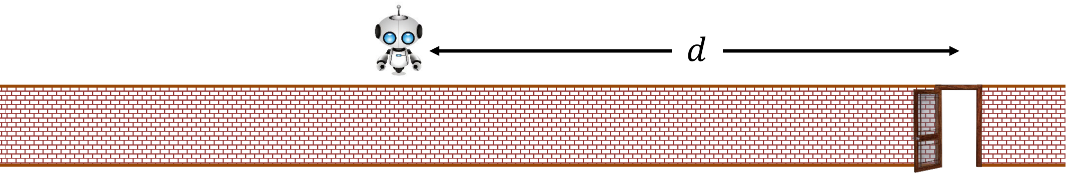
Algorithm 3:

- 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:

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

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 i in range(n):
    for j in range(n):
        do_something()   # O(1)
```

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

```
for i in range(n):
    for j in range(i, n):
        do_something()   # O(1)
```

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

```
for i in range(n):
    for j in range(m):
        for k in range(p):
            do_something()   # O(1)
```

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

Linear time: $O(n)$

Running time proportional to input size

```python
# Compute the maximum of a vector with n numbers
m = a[0]
for i in range(1, len(a)):
    m = max(m, a[i])

# Equivalent way in Python (same complexity)
m = max(a)
```


Linear time: $O(n)$

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: $O(\log n)$

- 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

```python
def power(x: int, y: int) -> int:
    """Returns x^y. Pre: x ≠ 0, y ≥ 0"""
    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)
```

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

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

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

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

- ***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?
  - $\mathrm{O}(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.