# Algorithm Analysis in a Nutshell

## Algorithm Analysis

**Algorithm Analysis** is

1. proof that the algorithm solves the given problem;
2. prediction of resources required by the algorithm.

**Algorithm correctness.** Algorithm correctness is typically stated as follows:

> *For each instance of the computational program given as input to the algorithm, the algorithm outputs the solution to the problem.*

Proving this is a challenge.

### Prediction of Resources

**RAM: Random Access Machine.** A computational model with the following properties:

- **A single processor.**
- **Sequential execution of instructions.** Instructions are executed *sequentially* on a single processor, until the end of the algorithm.
- **Infinite Random Access Memory.** The number of memory cells is not limited. It takes the same amount of time to access information stored in any memory cell.
- **Constant operation cost.** Each operation executed by the process has a specific **cost** associated with it (e.g., time it takes to complete). While costs may be different for different types of operation, operations of the same type *always have the same cost.*

- **Operations/Instructions specified by pseudocode.** The list of instructions/operations performed by the RAM is limited to what is used in our pseudocode:
  - addition
  - subtraction
  - multiplication
  - integer division
  - remainder
  - unary negation
  - comparison (of two numbers)
  - logical `and` (conjunction)
  - logical `or` (disjunction)
  - logical `not` (negation)
  - assignment
  - function call
  - procedure call
  - return value from function
  - return from procedure

**Note:** all other pseudocode intructions/statements can be represented as sequences of instructions/operations from the list above (e.g., **if** statements involve comparisons, possibly together with computation of logical operations)

**Resources used by an algorithm.** There are two main resources used by the algorithm:

1. Processor time.
2. Memory used.

Analysis of the processor time needed to complete the algorithm is called the analysis of the **computational complexity** (a.k.a. *running time*) of the algorithm.

Analysis of the memory used by the algorithm is called the analysis of the **space complexity** of the algorithm.

**Complexity of algorithm vs. complexity of problem.** We can analyze **both** the complexity of a computational problem and the complexity of a specific algorithm solving the problem.

Remember:

- The (space/running time) *complexity of an* ***algorithm*** is the amount of the resources needed for the algorithm to complete.
- The (space/running time) *complexity of a* ***problem*** is the **smallest** amount of the resources needed by **some** algorithm that solves the problem.

**Expressing complexity of an algorithm/problem.** Complexity of an algorithm/problem is usually expressed as a **function of the size of the input.**

**Input Size.** May differ for different problems. Variants are:

- Number of items in the input.
- Number of bytes in the input.
- Number of bits in the intput.
- Multiple numbers repsenting respective sizes of different portions of the input (e.g., number of vertices and number of edges in a graph).

## Simplistic Algorithm Analysis

Consider our algorithm FindMax for finding the largest number in an array. With each operation/instruction in the program we associate a known cost $c_o$. According to our model, this cost will always be the same for that specific instruction/operation.

| ALGORITHM FindMax(N, A[1..N]) | cost | freq |
|---|---|---|
| **begin** | | |
| tmpMax← $A[1]$; | $c_1$, | 1 time |
| **for** i ← 2 **to** N **do** | $c_2$, | $N-1$ times |
| **if** tmpMax< $A[i]$ **then** | $c_3$, | $N-1$ times |
| tmpMax← $A[i]$; | $c_4$, | 0–$N-1$ times |
| **endif** | | |
| **endfor** | | |
| **return** tmpMax; | $c_5$ | 1 time |
| **end** | | |

The running time of FindMax can be computed as the sum of all the costs for all operations/instructions executed in the algorithm:

$$runningTime = \sum_{i=1}^{5} freq(c_i) \cdot c_i.$$

In our case, the analysis is complicated by the fact that we may not know how many times tmpMax← $A[i]$ statement is executed. We can only estimate this number as follows:

- Best case: freq($c_4$) = 0. This happens when $A[1]$ is the largest number in the array.
- Worst case: freq($c_4$)= N-1. This happens when the array $A$ is sorted in ascending order (then, each array element is greater than the temporary maximum found on the previous step).

We will, this estimate, both the best and the worst case scenarios:

$runningTime(bestcase) = c_1 \cdot 1 + c_2 \cdot (N-1) + c_3 \cdot (N-1) + c_4 \cdot 0 + c_5 = (c_2 + c + 3) \cdot N + (c_1 + c_5 - c_2 - c_3)$.

$runningTime(worstcase) = c_1 \cdot 1 + c_2 \cdot (N-1) + c_3 \cdot (N-1) + c_4 \cdot (N-1) + c_5 = (c_2 + c + 3 + c_4) \cdot N + (c_1 + c_5 - c_4 - c_2 - c_3)$.

Let $c_2 + c_3 = a$, $c_2 + c + 3 + c_4 = a'$, $c_1 + c_5 - c_2 - c_3 = b$, $c_1 + c_5 - c_4 - c_2 - c_3 = b'$. Then, we get:

$$runningTime(bestcase) = aN + b$$
$$runningTime(worstcase) = a'N + b'$$

I both, the best, and the worst case, the running time of the algorithm grows **linearly** with the size of the input $N$!

## Asymptotic Algorithm Analysis

We claim that in the algorithm analysis, the knowledge of the specific constants $a$, $a'$, $b$, $b'$ is not important. The important part is to know **how fast the running time of the algorithm ioncreases with the increase in the size of the input.**

The "constants" do not matter because:

- The exact estimates for individual constants (e.g., $c_1$, $c_2$, etc.) are hard to obtain.
- Different computational models may have different costs associated with individual operations/instructions.

### Asymptotic growth of functions.

We discuss some mathematical notation needed in order to analyze algorithms.

**$\Theta$-notation.** Let $g(n)$ be a function. The set of functions $\Theta(g(n))$ is defined as follows:

$$\Theta(g(n)) = \{f(n) | (\exists c_1 \exists c_2 \exists n_0 > 0) \forall (n > n_0)(0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))\}.$$

*Intuitively:* $f(n)$ is in $\Theta(g(n))$ if, starting from some point ($n_0$), it can be "sandwiched" between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ for some values of $c_1$ and $c_2$.

*Example.* Let $g(n) = 5n + 3$. We claim that $f_1(n) = n + 100$ and $f_2(n) = 40n - 4$ both are in $\Theta(g(n))$.

For $f_1(n)$: consider $c_1 = 0.2$ and $c_2 = 1$. $c_1 \cdot g(n) = n + 0.6$. For all $n > 0$, $c_1 \cdot g(n) = n + 0.6 < n + 100 = f_1(n)$. Take $n_0 = 25$. For all $n > n_0$, $c_2 \cdot g(n) = 5n + 3 > n + 100$ ( $5n + 3 - n - 100 = 4n - 97$. therefore $5n + 3 > n + 100$ as long as $n \geq 25$.) Therefore $f_1(n) \in \Theta(g(n))$.

For $f_2(n)$: consider $c_1 = 1$, $c_2 = 8$, $n_0 = 2$. The arithmetics is left to you.

> If $f(n) \in \Theta(g(n))$, we say that $g(n)$ is an **asymptotically tight bound** on $f(n)$.

**Note:** We can show that $f(n) \in \Theta(g(n))$ **iff** $g(n) in \Theta(f(n))$. (Can you prove it?)

*Example.* Consider $g(n) = a \cdot n$. We can show that **any** linear function $f(n) = b \cdot N + c \in \Theta(g(n))$.

*Example.* Consider $g(n) = a \cdot n^2$. We can show that **any** quadratic function $f(n) = bn^2 + cn + d \in \Theta(g(n))$.

**Observation.** For any polynomial $\sum_{i=0}^{k} c_i \cdot n^k$, its **asymptotically tight bound** is a function $g_k(n) = n^k$.

## O-notation

Let $g(n)$ be a function. The set of functions for which $g(n)$ is an **(upper) asymptotic bound**, denoted $O(g(n))$, is defined as

$$O(g(n)) = \{f(n) | (\exists c > 0)(\exists n_0 > 0)(\forall n > n_0)(0 \leq f(n) \leq c \cdot g(n))\}$$

We write $f(n) = O(g(n))$ to mean $f(n) \in O(g(n))$.

*Example.* Let $g(n) = n^2$. We can see that $f(n) = 5n^2 + 3 = O(g(n))$. Also, $f'(n) = an + b = O(n^2)$.

**Observation.** Given a function $g(n)$, $\Theta(g(n)) \subset O(g(n))$.

## Ω-notation

Let $g(n)$ be a function. The set of functions that are **asymptotic upper bounds** of $g(n)$, (i.e., functions, for which $g(n)$ is an **asymptotic lower bound**), denoted $\Omega(g(n))$, is defined as

$$\Omega(g(n)) = \{f(n) | (\exists c > 0 \exists n_0 > 0)(\forall n > n_0)(0 \leq c \cdot g(n) \leq \cdot f(n))\}.$$

We write $f(n) = \Omega(g(n))$ to mean $f(n) \in \Omega(g(n))$.

*Example.* Let $g(n) = n^2$. $f(n) = 2n^2 + n + 5 = \Omega(g(n))$. In addition, $f'(n) = n^3 + n^2 + n + 1 = \Omega(g(n))$. On the other hand, if $h(n) = an + b$, then $g(n) = n^2 = \Omega(h(n))$.

**Theorm:** $f(n) = \Theta(g(n))$ **iff**:

1. $f(n) = O(g(n))$;
2. $f(n) = \Omega(g(n))$;

## *o*-notation

Let $g(n)$ and $f(n)$ be two functions. We say that $f(n) = o(g(n))$ **iff**

- $f(n) = O(g(n))$;
- $f(n) \neq \Theta(g(n))$;

*o*-notation represents *upper bounds that are not tight.*

*Example:* $f(n) = n = o(n^2)$. $f(n) = n \cdot log(n) = o(n^2)$. $f(n) = n^3 = o(2^n)$.

### $\omega$-notation

Let $g(n)$ and $f(n)$ be two functions. We say that $f(n) = \omega(g(n))$ **iff**

- $f(n) = \Omega(g(n))$;
- $f(n) \neq \Theta(g(n))$;

Alternatively: $f(n) = \omega(g(n))$ **iff** $g(n) = o(f(n))$.

$\omega$-notation represents *lower bounds that are not tight.*

*Example:* $f(n) = n^2 = \omega(n)$. $f(n) = n \cdot log(n) = \omega(n \cdot log(log(n)))$. $f(n) = 2^n = \omega(n^7)$.

**Theorem:** $f(n) = o(g(n))$ **iff**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

### Properties of asymptotic notation

**Transitivity:** The transitivity properties are as follows:

$$\begin{array}{lcl}
f(n) = \Theta(g(n)),\ g(n) = \Theta(h(n)) & \Rightarrow & f(n) = \Theta(h(n)) \\
f(n) = O(g(n)),\ g(n) = O(h(n)) & \Rightarrow & f(n) = O(h(n)) \\
f(n) = \Omega(g(n)),\ g(n) = \Omega(h(n)) & \Rightarrow & f(n) = \Omega(h(n)) \\
f(n) = o(g(n)),\ g(n) = o(h(n)) & \Rightarrow & f(n) = o(h(n)) \\
f(n) = \omega(g(n)),\ g(n) = \omega(h(n)) & \Rightarrow & f(n) = \omega(h(n))
\end{array}$$

**Reflexivity:** The reflexivity properties:

$$\begin{array}{l}
f(n) = \Theta(f(n)) \\
f(n) = O(f(n)) \\
f(n) = \Omega(f(n))
\end{array}$$

**Symmetry:** $\Theta$ is symmetric:

$$f(n) = \Theta(g(n)) \quad \textbf{iff} \quad g(n) = \Theta(f(n))$$

**Transpose Symmetry:** The other four are *anti-symmetric*:

$$\begin{array}{ll}
f(n) = O(g(n)) & \textbf{iff} \quad g(n) = \Omega(f(n)) \\
f(n) = o(g(n)) & \textbf{iff} \quad g(n) = \omega(f(n))
\end{array}$$

**Note:** All real numbers are **comparable**: for two numbers $a, b$, either $a > b$ or $a = b$ or $a < b$. But not all functions $f(n)$ and $g(n)$ are **comparable** in the sense of being asymptotic bounds of each other. In particular, **periodic** functions (or *oscilating* functions) are incomparable with other functions.

Consider $f(n) = n^2$; $g(n) = n^{1+2\cdot(n \bmod 2)}$. Essentially, $g(n) = n$ when $n$ is even and $g(n) = n^3$ when $n$ is odd. This means, that $f(n)$ and $g(n)$ are incomparable. (we can actually prove formally that neither $f(n) = \Omega(g(n))$ nor $g(n) = \Omega(f(n))$, for example, using the definition of $\Omega$.)

## Growth of functions

The following "classes" of functions appear commonly in the analysis of algorithms. They are listed in the **ascending order** of asymptotic growth.

| Category | Simple form | Representative |
|---|---|---|
| Constant | $const$ | $f(n) = 5$ |
| Logarithmic | $\log(n)$ (a.k.a $\log_2(n)$) | $f(n) = log(n) + 1$ |
| Polylog | $\sum_{i=0}^{k} c_i \cdot log^i(n)$ | $f(n) = log^2(n) + 3log(n) - 1$ |
| Square root | $\sqrt{n}$ | $f(n) = \sqrt{n+3} + log(n) + 1$ |
| Linear | $n$; $an + b$ | $f(n) = 5n + 56$ |
| NlogN | $n \cdot log(n)$ | $f(n) = nlog(n) + 4n - 5$ |
| Quadratic | $n^2$ | $f(n) = 3n^2 + n + 2log(n) - 4$ |
| Cubic | $n^3$ | $f(n) = 5n^3 - 5n^2 + sqrt(n) + 5log(n)$ |
| Polynomial | $n^k$ | $f(n) = n^k + 1$ |
| Factorial | $n!$ | $f(n) = n! + n^5 - log(n)$ |
| Exponential | $2^n$ | $f(n) = 2^n + n^4 + nlog(n)$ |