# Cost Analysis of Algorithms

Master MoSIG — Algorithms and Program Design

Florent Bouchez Tichadou — Gwenaël Delaval
{florent.bouchez-tichadou,gwenael.delaval}@imag.fr

September 25th, 2015

**Objectives:**

- Analyze the cost of an algorithm;
- Compare the cost of different algorithms.

## To remember

**Cost of an algorithm.** Resources that the algorithm requires in function of its input $i$: $\hat{C}(i)$. Most often the number of *primitive operations* or steps executed (*time* cost, $T(i)$) or the *memory* used (*space* cost, $S(i)$).

**Best-case. Worst-case. Average-case.** If $\mathcal{I}$ is the set of all possible inputs and $|i|$ is the size of input $i$ then

$$\hat{C}_{\text{best}}(n) = \min_{i \in \mathcal{I}, |i|=n} \hat{C}(i)$$
$$\hat{C}_{\text{worst}}(n) = \max_{i \in \mathcal{I}, |i|=n} \hat{C}(i)$$
$$\hat{C}_{\text{average}}(n) = \sum_{i \in \mathcal{I}, |i|=n} \hat{C}(i) \times \text{probability that input } i \text{ occurs}$$

**Recurrence equations.** If a recursive function divides its input of size $n$ in $a$ subproblems of size $n/b$ (with $a$ and $b$ constants, $a \geqslant 1$, $b \geqslant 2$), then the cost of this function can be expressed by the recurrence relation:

$$\hat{C}(n) = a.\hat{C}\left(\frac{n}{b}\right) + f(n)$$
$$\hat{C}(1) = O(1).$$

**Asymptotic notation.** Often also called simply the *complexity* of an algorithm. Drop lower-order terms and ignore the constant coefficient in the leading term. We are interested in the behaviour when $n$ tends to be really big.

$$f = O(g) \iff \exists C > 0, \exists n_0 > 0 \mid \forall n > n_0 \; f(n) \leqslant C.g(n)$$
$$f = \Omega(g) \iff g = O(f)$$
$$f = \Theta(g) \iff f = O(g) \text{ and } f = \Omega(g)$$

# 1 Asymptotic notation

In asymptotic notation, we are interested in the behaviour of programs when inputs tend to be quite large, hence we are only interested in the higher order terms, and the constants are not relevant.

$$\begin{aligned} 2n^2 + 3n + \log n + 12 &= O(n^2) + O(n) + O(\log n) + O(1) \\ &= O(n^2) \end{aligned}$$

Notes:

- $\log$ is the binary logarithm, i.e., $\log n = \log_2 n = \frac{\ln n}{\ln 2}$.
- the polynomial is also $O(n^3)$ (but it is less precise)

Three notations are commonly used:

- $f(n) = O(g(n))$ means that $f$ *is dominated by* $g$: for sufficiently large $n$, we can find a positive constant $C$ such that $f(n) \leqslant C.g(n)$.
- $f(n) = \Omega(g(n))$ means that $f$ *dominates* $g$, and is equivalent to $g(n) = O(f(n))$.
- $f(n) = \Theta(g(n))$ is equivalent to $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Here is a (non-exhaustive) list of the asymptotic notation of commonly found in complexities of algorithms. Complexities are ranked from the "fastest" (at the top) to the "slowest" at the bottom, along with some examples of the approximate number of computations required for different input sizes. Note how quickly it can become impractical when the size is large.

| $n$ | 2 | 10 | 100 | 1000 | $10^9$ |
|---|---|---|---|---|---|
| $O(1)$ | 1 | 1 | 1 | 1 | 1 |
| $O(\log n)$ | 1 | 3.32 | 6.64 | 10 | 29.9 |
| $O(n)$ | 2 | 10 | 100 | $10^3$ | $10^9$ |
| $O(n \log n)$ | 2 | 33.2 | 664 | $10^4$ | $3 \cdot 10^{10}$ |
| $O(n^2)$ | 4 | 100 | $10^4$ | $10^6$ | $10^{18}$ |
| $O(n^3)$ | 8 | 1000 | $10^6$ | $10^9$ | $10^{27}$ |
| $O(2^n)$ | 4 | 1024 | $10^{30}$ | $10^{300}$ | $10^{300000000}$ |

## 1.1 Analysis of insertion sort

**Algorithm 1:** Insertion sort

**Input**: An array $A$ of $n$ integers
**Result**: The array $A$ is sorted

```
1 for j = 2 to n do
2     key ← A[j] ;
3     i ← j − 1 ;
4     while i > 0 and A[i] > key do
5         A[i + 1] ← A[i] ;
6         i ← i − 1 ;
7     A[i + 1] ← key ;
```

**Best-case:** the array is already sorted (in increasing order).

- Always find that $A[i] \leqslant key$ upon the first time the while loop test is run (when $i = j - 1$) (so the while loop is not run).
- The body of the for loop (lines 2-8) has cost $O(1)$.
- The for loop is executed $O(n)$ times.
- $\hat{C}_{\text{best}}(n) = O(n)$.

**Worst-case:** the array is sorted in decreasing order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare key with all elements to the left of the $j$th position: compare with $j - 1$ elements.
- The body of the for loop has cost $O(j)$.
- $\hat{C}_{\text{worst}}(n) = \sum_{j=2}^{j=n} O(j) = O(n^2)$.

**Average-case:** assume each input is equiprobable.

- Equiprobable inputs imply for each element, rank among elements so far is equiprobable.
- When inserting element in j$th$ position, the expected number of times the while loop is executed is $\sum_{k=1}^{k=j} k/j = O(j)$.
- $\hat{C}_{\text{average}}(n) = \sum_{j=2}^{j=n} O(j) = O(n^2)$.

## 1.2 Recurrence analysis: binary search

The binary search algorithm is used to efficiently search a value in a sorted array:

**Algorithm 2:** Binary search

```
1 BS(T, X)
    Data: T[1..n] : array of n sorted elements, X : an element
    Result: true iff X is in T[bi..bs]
2   if n = 1 then
3     return T[1] = X
4   else
5     m ← ⌊n/2⌋;
6     if T[m] > X then
7       return BS (T[1..m − 1],X)
8     else
9       return BS (T[m..n],X)
```

### 1.2.1 Recurrence equations

The cost can be analyzed by following the structure of the recursive function. Let $\hat{C}(n)$ be the cost of BS applied to an array of size $n$. Then, this cost comprises a constant (test of the value of $n$, computation of $m$), added to the cost of BS on an array of size $\frac{n}{2}$:

$$\hat{C}(n) = \begin{cases} O(1) & \text{if } n = 1 \\ \hat{C}\left(\frac{n}{2}\right) + O(1) & \text{else} \end{cases} \quad (1)$$

### 1.2.2 General form of recurrence relations

A recursive function can generally be expressed in the form:

**Algorithm 3:** General form of recursive function

```
1 F(X(n))
2   if n > c then
3     Operations of cost f(n)...
      // division of input X into a subproblems of size n/b
4     (X_1, ..., X_a) ← Divide(X)
      // recursive calls on subproblems
5     r_1 ← F(X_1(n/b))
6     ⋮
7     r_a ← F(X_a(n/b))
8     return Assemble (r_1, ..., r_a)
```

Where:

- $a$ is the number of subproblems created by dividing the initial input $X$
- $n/b$ is the size of each subproblem.

`F` is then called $a$ times on inputs of size $n/b$.

Therefore, the recurrence equation for the cost of `F` is :

$$\hat{C}(n) = a.\hat{C}\left(\frac{n}{b}\right) + f(n)$$
$$\hat{C}(1) = O(1)$$

### 1.2.3 Master theorem

The Master Theorem[1] can be used to solve recurrence equations comprising asymptotic notations.

This theorem can be expressed as:

**Theorem 1** (Master theorem) Let $T$ be an increasing function satisfying

$$T(n) = a.T\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = O(1)$$

where $a \geqslant 1$, $b \geqslant 2$, $c > 0$.

Then :

1. if $f(n) = O(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$
2. if $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log^{k+1} n)$
3. if $f(n) = \Omega(n^c)$ where $c > \log_b a$ and $f$ follows the regularity condition, then $T(n) = \Theta(f(n))$.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. MIT Press, 2001. Sections 4.3 and 4.4 for the Master theorem and its proof.

### 1.2.4 Application to binary search

The Master theorem can be applied to the recurrence relation defining the cost of the binary search function :

$$\hat{C}(n) = \begin{cases} O(1) & \text{if } n = 1 \\ \hat{C}\left(\frac{n}{2}\right) + O(1) & \text{else} \end{cases}$$

This relation corresponds to the case 2 of the Master theorem, with $c = 0$, $a = 1$, $b = 2$ and $k = 0$. Then, we have:

$$\hat{C}(n) = \Theta(\log n).$$

# 2 Exercises

### Exercise 1 (Counting letters)

What is the worst-case *exact* number of comparisons of the count_letters function? Exhibit a family of examples reaching this worst case.

```
unsigned int count_letters (char *a) {
        unsigned int letters = 0;
        unsigned int i = 0;
        while (a[i] != 0) {
                if ((a[i] >= 'a') && (a[i] <= 'z')) letters ++;
                i++;
        }
        return letters ;
}
```

### Exercise 2 (A recursive function)

Give the number of calls to the printf function. What is the explore_tree function doing?

```
typedef struct s_btree {
        int value;
        struct s_btree * left ;
        struct s_btree * right;
} btree;

void explore_tree(btree *tree) {
        if (tree != NULL) {
                printf("value: %d\n", tree->value);
                explore_tree(tree->left);
                explore_tree(tree->right);
        }
}
```

### Exercise 3 (Selection sort)

Consider sorting $n$ numbers stored in array $A$ by first finding the smallest element of $A$ and exchanging it with the element in $A[1]$. Then finding the second smallest element of $A$, and exchanging it with $A[2]$. Continue in this manner for the first $n-1$ elements of $A$. Write the pseudocode for this algorithm, which is known as selection sort. Why does it need to run for only the first $n-1$ elements, rather than for all $n$ elements? Give the best-case and worst-case running times of the selection sort in $O$ notation.

### Exercise 4 (Linear search)

Consider the searching problem:
**Input:** An array $A$ of $n$ numbers and a value $v$.
**Output:** An index $i$ such that $v = A[i]$ or the special value $-1$ if $v$ does not appear in $A$.

Write the pseudocode for the linear search, which scans through the sequence looking for $v$. How many elements of the input sequence need to be checked on average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in $O$ notation?