Analysis of Algorithms
Script of Lecture I

Amalia Duch

September 18, 2019

Contents

Cost in time and space

* The question we always ask ourselves as programmers is: Can I do it better? In order to answer we must be able to compare programs/algorithms among them and determine which one is better.

* But what does it mean that an algorithm is better than another? In order to be able to answer, we have to define what it means that an algorithm is good/bad, that is, how efficient it is. This has sense only in relative terms (for comparison): the answer is not absolute.

* We define the efficiency of an algorithm as the amount of resources that it requires from the system and these resources are measured in time (how fast it is) and in space (what is the amount of memory that it requires).

* To the efficiency in time we say: cost in time, to the efficiency in space: cost in space.

* Obviously these costs can be measured in many ways. A possibility is experimentally, scheduling the algorithm and measuring the time it takes to solve each instance of the problem it solves. What are the disadvantages of doing it this

way? that we depend on the programming language used, the operating system we use, the type of machine, etc ...

* Since we want our measures to be independent of the above factors and in addition we want to predict the behaviour of our algorithms, we will measure the efficiency of our algorithms in a formal way, mathematically.

* How do we measure the cost of an algorithm? The cost in time: as the number of elemental operations that are required to run it. The cost in space: as the amount of memory that is required to execute it.

* What are the basic operations? Sums, substractions, multiplications, divisions, comparisons, assignments, parameters-passing, reading/writing etc... of basic elements, (indivisible elements) such as integers, real characters, etc....

* Example 1): Selection sorting algorithm. Input: V, vector of n elements; Output: V, with its n elements is increasing order. We perform basically three types of operations with the elements of the vector: comparisons, exchanges, in addition to loops we do assignments, increments and exchanges. If we count the number of comparisons: $n-1$ to find the first element, $n-2$ to find the second, etc. for a total of the order of n^2. If we count the number of exchanges, there are of the order of n exchanges For big ns, the number of comparisons dominates and we will say that the cost of the algorithm is of the order of n^2. The cost in space is of order of n.

Generally, given an algorithm A with input data set $\mathcal{A}$ its blueefficiency or bluecost (in time, space, number of I/O operations, etc.) is a function T from $\mathcal{A}$ to $\mathbb{N}$ (or $\mathbb{Q}$ or $\mathbb{R}$):

$$\begin{aligned} T : \mathcal{A} &\to \mathbb{N} \\ \alpha &\to T(\alpha) \end{aligned}$$

* But if we leave it that way it is too general and we would be comparing costs on vectors of different sizes, which would be unhelpful.

*This is the reason to define three functions that depend only on the input size and that summarize the properties of T.

Let $\mathcal{A}_n$ be the input set of size n and $T_n : \mathcal{A}_n \to \mathbb{N}$ the function T restricted to $\mathcal{A}_n$.

- *Best case cost*:
$$T_{\text{best}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$
- *Worst case cost*:
$$T_{\text{worst}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Average case cost*:

$$T_{\text{avg}}(n) = \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha)\, T_n(\alpha)$$
$$= \sum_{k \geq 0} k\ \Pr(T_n = k).$$

* Example 2): Sorting algorithm for insertion. Best case: $n-1$ comparisons and no exchanges. Worst case: from the order of n^2 comparisons and n exchanges.

* Example 3): Binary search. Best case: constant number of comparisons. Worst case: order of $\log n$ comparisons.

1. For all $n \geq 0$ and for any $\alpha \in \mathcal{A}_n$

$$T_{\text{best}}(n) \leq T_n(\alpha) \leq T_{\text{worst}}(n).$$

2. For all $n \geq 0$

$$T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n).$$

* In addition, we need a more practical way of referring to the cost when we say "proportional to" or "of the order of", and thus the asymptotic notation is used.

Given a function $f : \mathbb{R}^+ \to \mathbb{R}^+$ the class $\mathcal{O}(f)$ (big O of f) is

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0\ \exists c\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

And it reads, a function g is in $\mathcal{O}(f)$ if there is a constant c such that $g < c \cdot f$ for all n greater than n_0.

* Examples: (Explain annotation abuse)

$O(1) = \{1/n, 57, 1024, ...\}$ $O(n) = \{n, sqrt(n), n+12, 2n-52, log(n), 45, ...\}$ $O(n^3) = \{n, nsqrt(n), n^2 - 17n - 3, log^2(n), ...\}$

Basic properties of $\mathcal{O}$:

1. If $\lim_{n\to\infty} g(n)/f(n) < +\infty$ then $g = \mathcal{O}(f)$
2. Reflexivity: for every function $f : \mathbb{R}^+ \to \mathbb{R}^+$, $f = \mathcal{O}(f)$
3. Transitivity: if $f = \mathcal{O}(g)$ y $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$
4. For every constant $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

* $\Omega(f)$: symmetrical of $O(f)$.

$$\Omega(f)=\{g:\mathbb{R}^+\to\mathbb{R}^+ \mid \exists n_0 \, \exists c>0 \, \forall n\geq n_0 : g(n)\geq c\cdot f(n)\}$$

Ω is reflexive and transitive; if $\lim_{n\to\infty} g(n)/f(n) > 0$ then $g = \Omega(f)$. Besides, if $f = \mathcal{O}(g)$ then $g = \Omega(f)$ and viceversa.

* $\Theta(f)$ = $O(f) \cap \Omega(f)$.

is the class of functions with the same rate growth than f.

Θ is also reflexive and transitive. Furthermore, it is symmetric: $f = \Theta(g)$ if and only if $g = \Theta(f)$. If $\lim_{n\to\infty} g(n)/f(n) = c$ with $0 < c < \infty$ then $g = \Theta(f)$.