INMA2111 Discrete mathematics II : Algorithms and complexity

Course notes

Cyril de Bodt, Dounia Mulders

April 25, 2015

Reminders : Asymptotic notation

Let $\begin{cases} f: & \mathbb{N} \to \mathbb{R}^+ \\ g: & \mathbb{N} \to \mathbb{R}^+ \end{cases}$

- $f \in \mathcal{O}(g)$, often written $f(n) = \mathcal{O}(g(n))$, means: $\exists n_0, c : \forall n \geq_0, f(n) \leq c \cdot g(n)$.
- $f \in \Theta(g)$ $\Leftrightarrow f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$
  $\Leftrightarrow \exists n_0, c_1, c_2 : \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
  (f and g are equivalent in complexity)
- $f \in o(g) \Leftrightarrow f \in (O)(g)$ but $f \notin \Theta(g)$
- $f \in \Omega(g) \Leftrightarrow g \in \mathcal{O}(f)$
- $f \in \omega(g) \Leftrightarrow g \in o(f)$

Example .1.

$$\begin{aligned} n^2 &= \mathcal{O}(n^2) \\ &= o(n^3) \\ &= \Theta\left(\frac{n(n-1)}{2}\right) \\ &= \Omega(n \log(n)) \end{aligned}$$

Part I
Sorting algorithms

We will start with illustrating some ideas on sorting algorithms. The sorting problem may be defined by its input and output. Hence our goal is to find some algorithms such that:

Input : an array $T = \begin{pmatrix} T(1) & \dots & T(n) \end{pmatrix}$.
Output : the array is sorted in increasing order.

1 Selection Sort

First, we consider a naïve algorithm: **Selection Sort**. This first algorithm can be formulated as follows:

- Find the smallest entry of T.
- Put it in the first position.
- Repeat this procedure iteratively: Find the smallest entry of the rest, put it in the second position, etc

What about the time complexity of this algorithm?

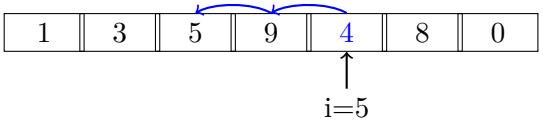
$$\begin{aligned} \text{Time} &= \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

Remark I.1. The complexity of this algorithm is $\Theta(n^2)$ for each instance. We have Worst case = Best case = Average case time complexity = $\Theta(n^2)$.

2 Insertion Sort

- **For** $i = 2$ to n % $T(1), \ldots, T(i-1)$ are already sorted = "Loop-invariant".
  Shift $T(i)$ to the left by recursive swaps until well paced.

Example I.2. We can consider a small exemple, the first four elements of the array being sorted:



Regarding the time complexity, we have:

- in worst case, at every step we swap until the beginning of the array: worst-case complexity = $\Theta(1) + \ldots + \Theta(n) = \Theta(n^2)$. This case corresponds to an instance sorted in decreasing order.
- Best-case corresponds to an array already sorted: $\Theta(n)$
- Average-case: instances come with uniform probability distribution over all possible orders (= $n!$ orders). Then:

$$\begin{aligned} \mathbb{E}[Time] &= \mathbb{E}[\sum_{i=2}^{n} \Theta(t_i)] && \text{With } t_i \text{ the number of swaps needed for } T(i). \\ &= \Theta\left(\sum_{i=2}^{n} \mathbb{E}[t_i]\right) && \text{By linearity of } \mathbb{E}. \\ &= \Theta\left(\sum_{i=2}^{n} \frac{i}{2}\right) \\ &= \Theta(n^2) \end{aligned}$$

In *some* applications, the relevant probability distribution is not uniform but biased towards almost-sorted instances.
$\Rightarrow$ Average-case can be $o(n^2)$.

3 Quick Sort

This third algorithm is based on the idea of "Divide-and-Conquer" which solves the sorting problem in the following manner:

- Split into small sorting problems.
- Recombine the outputs, which gives the sortd instance.

In particular, in this case:

- $Pivot := T(1)$.
- $T_{low} := [T(i) : T(i) \leq Pivot]$.
- $T_{high} := [T(i) : T(i) > Pivot]$.
- $\left.\begin{array}{ll} \rightarrow & \text{Quick Sort } T_{low}. \\ \rightarrow & \text{Quick Sort } T_{high}. \end{array}\right\} \Rightarrow T := [T_{low}; T_{high}]$.
  (The pivot being in T_{low}).

About the complexity of this algorithm:

- Worst case: when instance is already sorted !
  In this case: $\left.\begin{array}{ll} T_{low} & = 1 \text{ entry.} \\ T_{high} & = n-1 \text{ entries.} \end{array}\right] \Rightarrow$ maximally unbalanced subproblems. Time $= \Theta(n^2)$
  Indeed: construct $T_{low}/T_{right} = \Theta(n)$, hence we have the worst-case recurrence equation:

$$\begin{aligned} t_n &= t_1 + t_{n-1} + \Theta(n) \\ &= t_1 + (t_1 + t_{n-2} + \Theta(n-1)) + \Theta(n) \\ &= t_1 + (t_1 + (t_1 + t_{n-3} + \Theta(n-2)) + \Theta(n-1)) + \Theta(n) \\ &= \dots \\ &= n \cdot t_1 + \Theta(n^2) = \Theta(n^2) \end{aligned}$$

- Average-case complexity:

$$\begin{aligned} t_n &= t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n) \\ &= 2 \cdot t_{\frac{n}{2}} + \Theta(n) \\ &= 2\left(2 \cdot t_{\frac{n}{4}} + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= 2\left(2 \cdot \left(2 \cdot t_{\frac{n}{8}} + \Theta(\frac{n}{4})\right) + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= \dots \\ &= 2^{\log_2(n)} \cdot t_1 + \Theta(n \log_2(n)) \\ &= \Theta(n \log_2(n)) \end{aligned} \tag{1}$$

Remark I.3. We will denote $\log_2$ by log, since $\log_a(n) \in \Theta(\log_b(n))$, $\forall a, b > 1$.

Remark I.4. In $t_n = t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n)$,

$$\begin{aligned} \mathbb{E}[t_n] &= \mathbb{E}[t_{|T_{low}|}] + \mathbb{E}[t_{|T_{high}|}] + \Theta(n) \text{ would be more correct.} \\ &\cong t_{\mathbb{E}[|T_{low}|]} + t_{\mathbb{E}[|T_{high}|]} + \Theta(n) : \text{ in fact OK.} \end{aligned}$$

Now, what if you don't know that your instances are uniformly distributed? This leads to the next algorithm.

4 Randomized Quick Sort

Here, the first step of the algorithm is to shuffle entries randomly. It is equivalent to picking a random entry as pivot, instead of $T(1)$.

Now on any instance, Randomized Quick Sort runs in expected time $\Theta(n \log(n)) \Rightarrow$ Worst-case expected time of this random algorithm is also $\Theta(n \log(n))$.

One advantage of randomization: Robin Hood effect, i.e. you take from rich instances and give to the poor. Now, everybody can be unlucky from time to time but lucky most of the time.

Can we reach worst-case complexity $\mathcal{O}(n \log(n))$ deterministically?
The answer is yes, as we will see in the next section (Merge Sort).

5 Merge Sort

This algorithm is also based on Divide-and-Conquer approach.

- $T_{left} := [T(1), \dots, T(\lfloor \frac{n}{2} \rfloor)]$.
- $T_{right} := [T(\lfloor \frac{n}{2} \rfloor + 1), \dots, T(n)]$.
- $\left.\begin{array}{ll} \rightarrow & \text{Merge Sort } T_{left}. \\ \rightarrow & \text{Merge Sort } T_{right}. \end{array}\right\} \Rightarrow T := Merge\,(T_{left}, T_{right})$.

Time of merge is $\Theta(n)$, so we have the recurrence equation:

$$\begin{aligned} t_n &= 2 \cdot t_{\lceil \frac{n}{2} \rceil} + \Theta(n) \\ &= \Theta(n \log(n)) \text{ (see equation (??))} \end{aligned}$$

Which one to use?

- Random Quick Sort:
  - $\ominus$ You need to generate randomness: not so easy for a computer!
  - $\ominus$ You can be unlucky.
  - $\oplus$ Hidden constants are smaller if implemented efficiently.
  - $\oplus$ can be implemented "in place": on T itself $\rightarrow$ no need for extra memory for intermediate variables.