Efficient Algorithms and Data Structures

Ilaria Battiston *

Winter Semester 2019-2020

*All notes are collected with the aid of material provided by H. Racke. All images have been retrieved by slides present on the TUM Course Webpage.

Contents

1 Modeling issues

Modeling an algorithm is a complex task which involves measuring memory requirements, program size, power consumption but most of all computational time: comparisons, multiplications and hard disk accesses.

Measurements are taken according to theoretical analysis in a specific model of computation, giving asymptotic (lower) bounds typically focusing on the worst case.

Bounds are usually given by a function $f : \mathbb{N} : \mathbb{N}$ that maps input length to running time. The input length may be the size of the input in bits or the number of arguments, or just the number of arguments.

Performance can be measured calculating running time and storage space, or according to number of basic operations - easier but with less meaningful results.

1.1 Models of computation

The Touring machine is a simple model, allowing to alter the current memory location and useful to discuss computability. It is not a good model to develop efficient algorithms.

RAM has an input and an output tape, with infinite but countable number of registers holding indirectly addressed integers. Operations:

- Input operations (read);
- Output operations (write);
- Register-register transfers;
- Indirect addressing (loading content of the i-th register to the j-th);
- Branch based on comparisons;
- Jump to position x;
- Arithmetic instructions.

The cost model is uniform (every operation takes time 1) or logarithmic (depending on the content of the cells). The largest value stored in a bounded word model may not exceed 2^ω, where usually $\omega = \log_2 n$.

1.2 Complexity bounds and asymptotic notation

There are different types of complexity bounds:

- Best-case: $C_{bc}(n) := min\{C(x)|\,|x| = n\}$;
- Worst-case: $C_{wc}(n) := max\{C(x)|\,|x| = n\}$;
- Average (expected) case: $C_{avg}(n) := \frac{1}{|I_n|}\sum_{|x|=n} C(x)$
  $C_{avg}(n) := \sum_{x \in I_n} \mu(x)C(x)$.

I_n is a set of instances of length n.

There also are amortized complexity, the average cost of data structure operations over a worst case sequence, and randomized complexity (non-deterministic algorithms), worst case using random bits for a fixed input. Randomized complexity has cost $C(x)$ as expected cost $E[C(x)]$.

Running times are interesting for large values of n, without considering constant additive terms and exact numbers. A linear speed-up is always possible, but the complexity should be expressed by simple (positive, from $\mathbb{N}$ to $\mathbb{R}^+$) functions.

$$O(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\} \qquad g(x) \in O(f) : 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

$$\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\} \qquad g(x) \in \Omega(f) : 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$$

$$\Theta(f) = \Omega(f) \cap O(f) \qquad g(x) \in \Theta(f) : 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$$

$$o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\} \qquad g(x) \in o(f) : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

$$\omega(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\} \qquad g(x) \in \omega(f) : \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

It is important to note that a function belongs to a set, does not equal to it. The constant factors can be written or ignored.

Property of linearity of sum and multiplications apply to functions with the property $\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$ for O and Θ.

Asymptotic notations must not be used within induction proofs: for any constants $a, b, \log_a n = \Theta(\log_b n)$ and the base can be ignored. In general, $\log n = \log_2 n$.

Sometimes the input of an algorithm consists in several variables, which extend the general definition of asymptotic notation

2 Recurrences

Recurrences need to be solved to bring the expression from the number of comparisons to the closed form. This can be done in several ways:

1. Induction, proving or disproving the solution is correct by taking a wild guess of it (assuming n respects certain conditions, such as being a power of 2 to apply the logarithm);
2. Master Theorem, obtaining tight asymptotic bounds applicable to most recurrences;
3. Characteristic Polynomial, to solve linear homogeneous recurrences;
4. Generating Functions, a more general technique also for non linear relations;
5. Transformation of the Recurrence to a linear one.

2.1 Master theorem

Let $a \geq 1$, $b \geq 1$ and $\epsilon > 0$ denote constants. Considering the recurrence

$$T(n) = aT\left(\frac{n}{b} = f(n)\right)$$

where the first term can be substituted with a constant c havig n sufficiently small, there are three possible cases:

$$f(n) = O(n^{\log_b(a)-\epsilon}) \implies T(n) = \Theta(n^{log_b a})$$

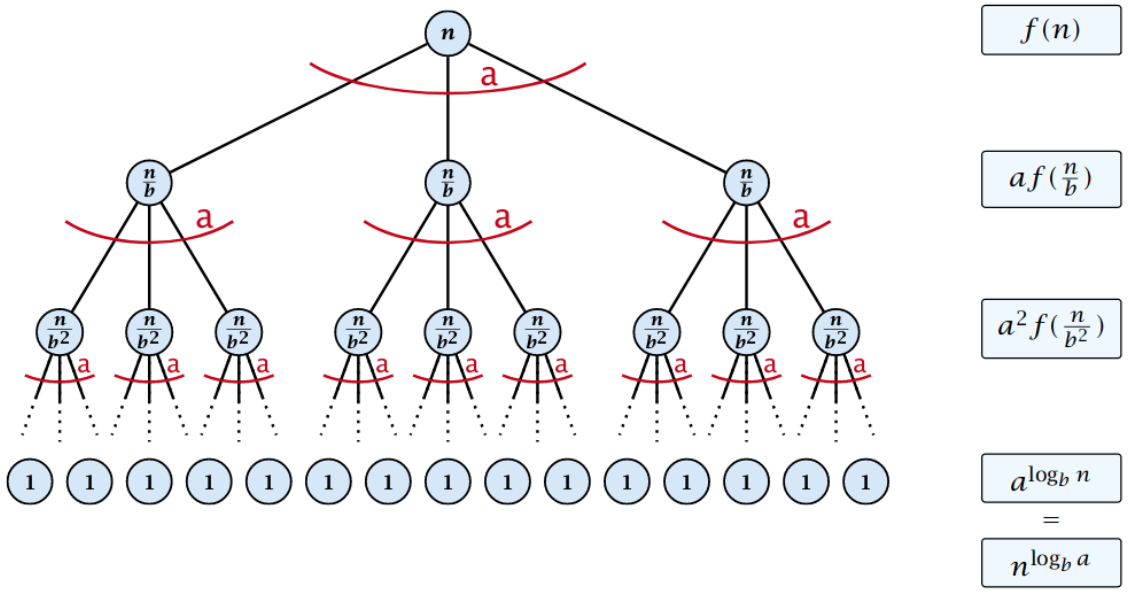
$$f(n) = \Theta(n^{\log_b(a)} \log^k n) \implies T(n) = \Theta(n^{\log_b a} \log k + 1n) \quad k \geq 0$$

$$f(n) = \Omega(n^{\log_b(a)-\epsilon}) \wedge af\left(\frac{n}{b}\right) \leq cf(n) \text{ for sufficiently large } n \text{ and } c < 1 \implies T(n) = \Theta(f(n))$$

This methods divides the problem in a sub-problems of the same size $^{n}/_{b}$ and then combines them. The master theorem can be applied looking at the function $f(n)$ and comparing it to $n^{\log_b a}$ to understand which case it falls into:

1. Strictly smaller;
2. Same growth, adding a logarithm factor with increasing exponent;
3. Bigger growth (usually not happening with good recursive algorithms);
4. None of those cases (theorem invalid).

The Master theorem can be proven using a recursion tree to visualize the running time of an algorithm, assuming n is in the case b^l with non-recursive case applied to problem of the size 1.



The tree stops when timing becomes $a^{log_b n}$, ignoring smaller sub-problems. From this can be derived the final equation, assuming $(b^{log_b a})^{log_b n} = a^{log_b n} = n^{log_b a} = (b^{log_b n})^{log_b a}$.

$f(n)$ (the cost of the algorithm at the first step) is compared to the three cases: if it's the first (smaller), the actual computation is performed at the lower levels. If functions have the same growth, the running time is approximately the same at every level - otherwise, the total time is dominated by the root (technically, not happening in practice).

This gives a formula which needs to be evaluated for each case: