Algorithms and computational models

Definition: An **algorithm** is a sequence of well defined steps that transforms a set of inputs into a set of outputs, in a finite amount of time.

The definition is vague: what is step? What is input/output?

Definition: A **computational model** is a mathematical tool to perform computations.

Namely, a computational model is a machinery to perform computations, so itself depends on the operations we are able to perform.

Ideally we want to compute (evaluate) functions, we have the following:

- a function described by an algorithm is **calculable**;
- a function *implementable* in a computational model (i.e. that can be turned into a program for a specific computational model) is **computable**.

The problem is that we cannot be sure that whenever we a function described by an algorithm (calculable) we can encode it into a program (computable), i.e. calulability does not imply computability (algorithms would not guarantee implementability).

Halting problem

Consider the **Halting problem**: we have a function **h** that etsblish wheter any program **p** ends its execution or runs forever on an input **i**:

$$h(p,i) = \begin{matrix} 0 \; if \; p(i) \; never \; ends \\ 1 \qquad\qquad\qquad otherwise \end{matrix}$$

Goal: implement **h**.

Take a computable function **f(a,b)**. Define a function g_f evaluating **f**:

$$g_f(i) = \begin{matrix} 0 \;\; if \;\; f(i,i) = 0 \\ \uparrow \qquad otherwise \end{matrix}$$

Since **f** is computable so it is g_f, then define a G_f as the program implementing g_f.

Problem: can the Halting function **h** be put in place of **f**?

There are two cases to consider:

1. $f(G_f, G_f) = 0$ then $g_f(G_f) = 0$ and by definition $\mathrm{h}(G_f, G_f) = 1$;
2. $f(G_f, G_f) \neq 0$ then $g_f(G_f) \uparrow$ and by definition $\mathrm{h}(G_f, G_f) = 0$;

For all computable functions **f**, **h**$\neq$**f** hence **h** is not computable.

Church-Turing thesis

Problem: we want to map algorithms into programs.

Church-Turing thesis: Every *effectively* calculable function is a computable function.

In simpler words: if the algorithm is not too complex, then it can be implemented, meaning that **calcualability implies computability**, if we have an algorithm for a certain function **f**, than **f** can be formally computed.

From Church Turing thesis also follow that all "reasonable" computational models are equivalent.

Random-Access Machine (RAM)

Before we can analyze an algorithm, we must have a model for the implementation technology that we will use, including a model for the resources of that technology and their costs.

RAM is a computational model, meant to model real hardware, but without some limitations of real hardware (for example there is no memory hierarchy, neither instructions execution time, every instruction takes the same time, nor finiteness).

The RAM model contains instructions commonly found in our computers: arithmetic (add, subtract, multiply, divide, floor, ceiling), data movement (copy, load, store), control ((un)conditional branch, subroutine call and return), data types are integers and floats.

In our context, algorithms are defined as programs on RAMs.

Time complexity

In order to measure the efficiency of an algorithm, we can use the **execution time**. The problem is that the execution time really depends on the input, moreover assuming 1 time unit per instruction is not realistic. Since algorithms are not programs, we cannot estimate the execution time of a program just by looking at the algorithm!

A possible solution is to measure efficiency of an algorithm using its **scalability** (rate/order of growth).

Definition: scalability is the effectiveness of a system in handling input growth.

In some sense we want to relate the input size of the algorithm with an estimate of the execution time. We are focusing on the growth of the execution time with respect to the growth of the input size. Hence we don't measure the execution time for a given input, but we estimate the relation between input size and execution time.

In doing this we look at the **asymptotic behaviour**, we abstract from the single instruction execution time (constants doesn't matter!).

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the *asymptotic efficiency* of the algorithm, that is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, i.e. as the size of the input increases without bound.

Linear Speedup Theorem: for any RAM we are able to build a RAM taking half of the time of the previous RAM.

This theorem gives the intuition behind the fact that constants do not matter.

More formally, we can use the **big-O notation** (asymtotic upper bound) to express this.

Definition:

$O(f(n)) \equiv \{g(n) | \exists c > 0 \ \exists n_0 > 0 \ m > n_0 \Rightarrow g(m) \leq c \cdot f(m))\}$.

n_0 in the formula is the input size.

Formally O(f(n)) is a set, but thanks to some properties we can work with it as it was a number.

Properties:

substantially we consider only leading term of the function (lower order terms are insignificant for big n) and we ignore leading term's constant coefficient.

$\forall c_1, c_2 \in \mathbb{N}$ *and* $\forall k \in \mathbb{Z}$ we have:

- $f(n) \in O(f(N))$
- $O(f(n)) = O(c_1 \cdot f(n) + k)$
- $if \ c_1 \geq c_2 \Rightarrow O(f(N)^{c_1} + k \cdot f(n)^{c_2}) = O(f(n)^{c_1})$

i.e. what is important in the term having the higher power, we can ignore the rest. * $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ * $if h(n) \in O(f(n)) an f h'(n) \in O(g(n))$, then: * $h(n) + h'(n) \in O(f(n) + g(n))$ * $h(n) \cdot h'(n) \in O(g(n) \cdot f(n))$

With the big-O notation we have an upper bound for our function, however we need also a lower bound. For this purpose we can use the $\mathbf{big - \Omega notation}$ (asymptotic lower bound).

Definition:

$\Omega(f(n)) \equiv \{g(n) | \exists c > 0 \ \exists n_0 > 0 \ m > n_0 \Rightarrow g(m) \geq c \cdot f(m))\}$.

In order to bound from both sides, we can use the $\mathbf{big - \Theta notation}$.

Definition:

$\Theta(f(n)) \equiv \{g(n) | \exists c > 0 \exists n_0 > 0 \ m > n_0 \Rightarrow c_1 \cdot f(m) \leq g(m) \leq c_2 \cdot f(m)\}$

It holds the following:

Theorem:

$f(n) \in \Theta(g(m)) \Leftrightarrow f(n) \in O(g(n)) \cap \Omega(g(n))$

Remark: the overall cost of the algorithm is the sum of the costs of the instructions.

Abstract Data Types

Arrays: indexed homogeneous collection of values fixed in length (to change the length we need to rebuilt from scratch the arry).

Single-Linked List: sequences of values supporting *head* and *next* operations (we can increase the size of the data structure by by adding elements).

Double-Linked Lists: sequences of values supporting *head*, *next* and *previous* operations (we can move backward and forward selecting *previous* or *next* operation respectively).

Queues: collections of values ruled according to the FIFO (First In First Out) policy. They support *head*, *is_empty*, *insert_back* and *extract_head* operations.

Stacks: collection of values ruled according to the LIFO (Last In First Out) policy. They support *top*, *is_empty*, *insert_top* and *extract_top* operations.

Graph Theory

Graphs are pairs (V,E) where V is a set of **nodes** and E is a set of **edges**.

A graph is (un)directed if the edges are (un)directed. In directed graphs, edges are arrows, in this case the starting node is the *source*, the arrival node is the *destination*.

A **path** is a sequence of continuous edges moving from one node to another one. A **cycle** is a path in which initial (source) and final (destination) node coincide. A graph is **acyclic** if it does not contain any cycle. It is **connnected** if there is a path between every pair of nodes. A **tree** is a connected and acyclic undirected graph.

Take a tree in the graph theory sense, in order to use it as an abstract data type, we need to organize it in hierarchical levels. First of all we need to decide which node is the *root* of the graph (our tree), then we define the **depth** of a node as the distance from the root, where distance is defined as the shortest path, in the graph theory sense. A **level** is a set of nodes having the same depth.

Since we have defined a root, it makes sense to define the **parent of a node** as the node one step closer to the root, the **children of a node** as the node having the current node as its parent, a **leaf** is a node without children, while **internal nodes** are nodes having children. Two node are **siblings** if they have the same parent. The **height** of a tree is the maximum depth among those of

its leaves. A tree is **n-ary** if every node can have up to n children, a n-ary tree is **complete** if every node except the leaves have n children.

Matrix multiplication

Consider the **row-column multiplication** of a n x m matrix A and a m x l matrix B. The result is a n x l matrix such that: $(A \times B)[i,j] = \sum_k A[i,k] \cdot B[k,j]$

Consider, for sake of simplicity, just n x n square matrices.

If we implement the algorithm mimicking the definition (naive solution), we end up with a complexity $\Theta(n^3)$.

Divide-and-Conquer strategy

Many useful algorithm are *recursive* in structure, and typically they follow a *divide-and-conquer* approach: break the problem into sever subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and combine these solutions to get a solution to the original problem. Those three steps (divide, conquer, combine) are involved at each level of the recursion.

Idea is to split matrices A and B in four blocks each:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

so that $C_{i,j} = (A_{i,1} \times B_{1,j}) + (A_{i,2} \times B_{2,j})$.

This is the basis to write a recursive algorithm, since matrices $A_{i,j}$ and $B_{i,j}$ can be further decomposed in blocks, until single elements.

In the formula above, the + is the element-wise matrix sum, with time complexity $\Theta(n^2)$, for n x n matrices. $A_{i,k}$ and $B_{k,j}$ are $\frac{n}{2} \times \frac{n}{2}$ matrices.

In the end the recursive algorithm requires:

- 8 multiplication between $\frac{n}{2} \times \frac{n}{2}$ matrices;
- 4 sums between $\frac{n}{2} \times \frac{n}{2}$ matrices.

(this numbers comes from the fact that initially we splitted the matrices in 4 blocks).

When an algorithm contains a recursive call to itself, we can describe (often) its running time by a *recurrence equation*, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. Solving the recurrence equation provides bounds on the performance of the algorithm.

Hence if $T_M(n)$ is the complexity of the recursive algorithm for input square matrices of dimension n, we have the following recursive equation:

$$T_M(n) = 8 \cdot T_M(\frac{n}{2}) + 4 \cdot \Theta(n^2) = 8 \cdot T_M(\frac{n}{2}) + \Theta(n^2)$$

(cost of recursive call + cost of the sum).

In order to solve this equation we replace $\Theta(n^2)$ with $c \cdot n^2$ and we write the recursion tree. Since at every step we half the dimension of the matrix, this tree has depth $log_2 n$.

At the very beginning, the 1^{st} call, the sum will cost $c \cdot n^2$. After that matrices are split in 4 blocks of dimension $\frac{n}{2}$ hence we need to solve 8 block-block multiplication, consequently the sums will cost $8 \cdot c \cdot (\frac{n}{2})^2$. This will be repeated until we arrive to element-wise multiplication (1x1 blocks), in that case sum will cost c. In general, for the i-th recursive call, the cost of a sum is $c \cdot (\frac{n}{2^i})^2$ hence the complexity $8^i \cdot c \cdot (\frac{n}{2^i})^2 = [since\ 8 = 2^3] = 2^i \cdot (c \cdot n^2)$ with $i = 0...log_2 n$

The overall complexity is obtained by summing the complexity at each level (recursion-tree method). Hence $T_M = c \cdot n^2 \cdot (2^{1+log_2 n} - 1) = c \cdot n^2 \cdot (2 \cdot n - 1) \in \Theta(n^3)$ (which is exactly the complexity of the naive algorithm).

The reason why the complexity doesn't decrease is that there are too many recursive calls in the divide-et-conquer algorithm.

Strassen algorithm

Goal is to reduce the complexity of the definition itself! Instead of having 8 recursive calls we will just have 7 (intuition is that $8 = 2^3, 7 < 2^3$ and this is very relevant from the algorithmic point of view). In the have we will have more sums than divide-and-conquer, but less recursive calls.

The complexity equation for this case is $T_M(n) = 7 \cdot T_M(\frac{n}{2} + \Theta(n^2))$ the depth of the recursion tree is again $log_2 n$. Following the same reasoning as before, we have that at the i-th step the complexity is $7^i \cdot c \cdot (\frac{n}{2^i})^2 = (\frac{7}{4})^i \cdot c \cdot n^2$ with $i = 0...log_2 n$.

Hence $T_M(n) = \ ... \ \in \Theta(n^{log_2 7})$ which is more than quadratic, so the asymptotic complexity is better than in the naive algorithm (it is a smarter solution than the definition itself!).

However this algorithm is not in-place (every step we require non-constant amount of additional memory), so we need to take care in handling the memory throughout our implementation.