University of Trieste

International School for Advanced Studies

The Abdus Salam International Centre for Theoretical Physics

Algorithmic Design

Lectures Notes

Author:
Marco Sciorilli

Gennaio 2021

Abstract

This document contains my notes on the course of Algorithmic Design held by Prof. Alberto Casagrande for the Master Degree in Data Science and Scientific Computing at Trieste University in the year 2020/2021. As they are a work in progress, every correction and suggestion is welcomed. Please, write me at: marco.sciorilli@gmail.com .

Contents

Chapter 1

Fundamentals

1.1 Algorithm and Computational Model

Definition 1. *Algorithm: a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time.*

The output should be consistent with the given definition of the problem. In some specific times, there could be actually an algorithm that goes on indefinitely. But that case is not studied in this course.

Definition 2. *Computational model: a mathematical tool to perform computations.*

A function described by an algorithm is calculable(i.e. I can give an high level set of instructions to turn the input of the function into the output of the function, I can provide and algorithm for it).
A function is implementable in a computational model is computable (i.e. Given a formal model, it provides a set of instructions which are fixed, making me able to write a program that turn the input of the function in the output of the function using only that proposed rules).
Notion of calculable and computable are related? Algorithms would not guarantee implementability! An interesting example is the "Halting problem": we want to write a program that takes as input any other program and it establish whether that ends in a finite amount of time or not.

Example 1. *Halting Problem: Let h be the function that establish whether any program p eventually ends its execution ($\downarrow$) on an input i or runs forever ($\uparrow$)*

$$h(p,i) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } p(i) \textit{ never ends} \\ 1 & \textit{otherwise} \end{cases} \tag{1.1}$$

It is not possible to implement h.

Proof. For any computable function $f(a, b)$ we can define the function

$$g_f(, i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i,i) = 0 \text{(if it ends)} \\ \uparrow & \textit{otherwise} \text{(it goes on forever)} \end{cases} \tag{1.2}$$

If we assume that f is computable, so is g_f, cause the first part can be implemented using a conditional statement, while the second can be implemented using a "while" forever loop. If I can evaluate the former, I can evaluate the latter. Let's call G_f the program implementing the function g_f.
Can f be the Halting function? No

- If $f(G_f, G_f) = 0 \Rightarrow g_f(G_f) = 0$ and $h(G_f, G_f) = 1$: a.k.a. if G_f applied on G_f gives 0, then g_f ends, so the halting function return 1.
- If $f(G_f, G_f) \neq 0 \Rightarrow g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$: a.k.a. if G_f applied on G_f is not 0, then g_f goes on forever, so the halting function return 0.

So, any computable function must be different from h, because any f with a given result must be different from its associated h. □

Luckily, there is a thesis that gives a correlation between computable and calculable.

Thesis 1. *Church-Turing Thesis: Every effectively calculable function is a computable function.*

So, if we are able to describe an algorithm for a function f, then f can be formally computed. This has some direct consequences

- All the "reasonable" computation models are equivalent.(i.e. we are able to write a program in every different programming language)
- We can avoid "hard-to-be-programmed" models.(i.e. we can avoid models which are not easy to be handled)

Example 2. *Random-Access Machine: an easy computational model.*
It allows to

- *Variables to store data (no types)*
- *Arrays*
- *Integer and floating point constatns*
- *Algebraic functions*
- *Assignments*
- *Pointers (no pointer arithmetic)*
- *Conditional and loop statements*
- *Procedure definitions and recursion*
- *Simple "reasonable" functions*

This is the computational model used in this course, not a real machine. Now lets look at an example of an algorithm, assuming to work on this computational model (so thanks to the Church-Turing Thesis every conclusion on this machinery will work on real life machine)

Algorithm 1: Algorithm example: find the maximum in an array

```
Input: An array A of numbers $< a_1, ..., a_2 >$
Output: The maximum among $a_1, ..., a_2$

def find_max(A)
    max_value← A[1]
    for i← 2..|A| do
        if A[i]>max_value then
            max_value← A[i]
        end
    end
    return  max_value
enddef
```

Ram is not a real hardware: any variable in a ram algorithm can store any possible number. It doesn't have memory hierarchy, and no difference in instruction execution time (for example, same time for all the operations).

1.2 Time Complexity

We want to asses the efficiency of the algorithm. Execution time is not an effective way to do so, because is not applicable in many cases. Counting the number of operations could be a good idea, but it depends on the size of the input.

Definition 3. *Scalability: effectiveness of a system in handling input growth.*

It means that the system has to keep the execution time low even if the input is big. We want to estimate the relation between the input size and the execution time, we use a function to do so.
The important part of this function is the asymptotic behavior (its exponent). Constants are not relevant. A theorem called "linear time speedup theorem", support this idea, as it asses that for every Touring Machine, it is possible to built an equivalent one scaled by a constant.

Big * notation

These are ways in which one can collect all the functions that behave in the same way.

Definition 4. *Big-O notation:*

$$O(f) \overset{\text{def}}{=} \{g | \exists c > 0 \ \exists n_0 > 0 \ s.t. \ m \geq n_0 \ \Rightarrow g(m) \leq cf(m)\} \tag{1.3}$$

It give the upper bound.

Example 3. *Examples of functions' O(n) appartenance:*

- $n \in O(n)$
- $2n \in O(n)$
- $500n \in O(n)$
- $2n + 7 \in O(n)$
- $n \in O(n^2)$
- $n \in O(2n) = O(n)$

Definition 5. *Big-*Ω *notation:*

$$\Omega(f) \overset{\text{def}}{=} \{g | \exists c > 0 \ \exists n_0 > 0 \ s.t. \ m \geq n_0 \ \Rightarrow cf(m) \leq g(m)\} \tag{1.4}$$

It gives a lower bound.

Example 4. *Examples of functions'* $\Omega(n)$ *appartenance:*

- $n \in \Omega(n)$
- $2n \in \Omega(n)$
- $500n \in \Omega(n)$
- $2n + 7 \in \Omega(n)$
- $n \notin \Omega(n^2)$
- $n \in \Omega(2n) = \Omega(n)$

Properties of Big-O notation, for any $c_1, c_2 \in \mathbb{N}$ and for any $k \in \mathbb{Z}$:

- $f(n) \in O(f(n))$
- $O(f(n)) = O(c_1 f(n) + k)$
- if $c_1 \geq c_2$ then $O(f(n)^{c_1} + kf(n)^{c_2}) = O(f(n)^{c_1})$
- $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$
- if $h(n) \subseteq O(f(n))$ and $h'(n) \in O(g(n))$, then
  - $h(n) + h'(n) \in O(g(n) + f(n))$
  - $h(n) \times h'(n) \in O(g(n) \times f(n))$

Definition 6. *Big-*Θ *notation:*

$$\Theta(f(n)) \overset{\text{def}}{=} \{g(n) | \exists c_1, c_2 > 0 \ \exists n_0 > 0 \ s.t. \ m \geq n_0 \ \Rightarrow c_1 f(m) \leq g(m) \leq c_2 f(m)\} \tag{1.5}$$

Theorem 1.

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n)) \tag{1.6}$$

1.3 Cost Criteria

We are interested in how many time instructions are executed, not in the time of the single instruction.
Uniform cost criterion states that time cost is:

- 1 for any Boolean and algebraic expression evaluation.
- 0 for assignments and control instructions.

Any expression, no matter how difficult it is, cost the same. The aim is not to evaluate the execution time, but how many time an instruction is executed.

Example 5.

Algorithm 2: Uniform Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 1
    for i← 1..n do // loop n time
        Z←Z*Z // costs 1
    end
    return Z
enddef
```

Cost functions is $T(n) = 1 + n * 1 \in \Theta(n)$

However, $test(n) = 2^{2^n}$, so the function in linear time, according to the uniform cost criterion, return a very large number, that could not be stored in a linear space.
Logarithmic Cost Criterion states that:

- $\max_{i\in[0,c]}(\log a_i)$ for any Boolean and algebraic expression involving $a_0, ..., a_c$ as operands.
- 0 for assignments and control instructions.

Example 6.

Algorithm 3: Logarithmic Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 2
    for i← 1..n do // loop n time
        Z←Z*Z // costs log Z
    end
    return Z
enddef
```

Cost functions is $T(n) = \sum_{i=1}^{n} \log 2^{2^i} = \sum_{i=1}^{n} 2^i = 2 * (2^n - 1) \in \Theta(2^n)$

Logarithmic cost better represents big-number algorithms. If instead the representation space is bounded (as for CPU arithmetic), uniform cost is enough. Where not otherwise declared, we will use uniform cost criterion.

1.4 Some Useful Notions

Definition 7. *Arrays: indexed collections of values which is fixed in length.*

Definition 8. *Single-Linked Lists: sequences of values supporting `head` (the first element) and `next` operations.*

Definition 9. *Double-Linked Lists: sequences of values supporting `head` (the first element), `next` and `previous` operations.*

Definition 10. *Queues: Collections of values ruled according to the FIFO policy (First In First Out:the first element inserted in the list, should be the first one to be taken out). They support `head`, `is_empty`,`insert_back`, `extract_head` operations.*

Definition 11. *Stacks: Collections of values ruled according to the LIFO policy (Last In First Out:the last element inserted in the list, should be the first one to be taken out). They support `head`, `is_empty`,`insert_back`, `extract_head` operations*

Definition 12. *Graphs: set of pairs (V,E) where:*

- *V is a set of nodes*
- *E is a set of edges*

If the edges are (un)directed, the graph itself is (un)directed.

Definition 13. *Path of length n between $a, b \in V$: a sequence $e_1, ..., e_n$ s.t.*

- *e_1 involves a*
- *e_n involves b*
- *e_i and e_{i+1} involve a common node n_i*

A Cycle is a node in which the fist node is also the last one.
A graph is connected if there is a path between every pairs of nodes.
A graph is acyclic if it does not contains cycles.
A tree is a connected and acyclic undirected graph. Tree are useful to organize data, and store it in a smart way. The `root` is the node from which all the tree originate.
The `depth` the length of the path going from the node considered to the `root`.
A `level` is a set of nodes having the same `depth`.
The parent of a node is a node one step closer to the root. The children of a node have the node as a parent. Two nodes are siblings if they have the same parent. Leaves are nodes without children, while internal nodes have children.
The height of a tree is the max depth among those of its leaves.
Every node of a n-ary tree can have up to n children. A n-ary tree is complete if the nodes in all the levels but the last one have n children.

Chapter 2

Strassen's Algorithm for Matrix Multiplication

2.1 Problem definition

Definition 14. *Row-Column Multiplication: let A be a $n \times n$ matrix and let B be a $m \times l$ matrix. $A \times B$ is a $n \times l$ matrix s.t.*

$$(A \times B)[i, j] = \sum_k A[i, k] * B[k, j] \tag{2.1}$$

Input: Two $n \times n$ matrices A and B (where n is a power of 2)
Output: The $n \times n$ matrix $A \times B$

2.2 Motivation

- In Deep Neural Network, both training and evaluation heavily rely on matrix multiplication.
- Good example to learn how to compute the complexity of an algorithm.

2.3 Naive Solution

Do exactly as you would do by hand.

Algorithm 4: Naive matrix multiplication

```
def naive_mult(C,A,B):
    for i← 1..rows(A) do
        for j ← 1...cols(B) do
            a← 0
            for k← 1..cols(A) do
                a← a + $A[i,k] * B[k,j]$
            end
            $C[i,j]$ ← a
        end
    end
    return C
enddef
```

The algorithm is composed by 3 nested loop with indexes in $[1, n]$. The inner block takes $O(1)$, so the overall execution complexity takes time $\Theta(n^3)$. Is it possible to find a better algorithm using recursion?

2.4 Divide-and-Conquer Strategy

We try to use recursion by splitting the matrix into 4 quadrants, where

$$C_{ij} = (A_{i1} \times B_{1j}) + (A_{i2} \times B_{2j}) \tag{2.2}$$

We can than use the standard technique on the blocks. Let's find the complexity of this new matrix.
We can find the complexity required to find a quadrant by finding the complexity of the matrix product between matrices a quarter of the original size, plus the cost of the sum of the resulting matrices. Defining some symbols:

- $+$ is the element-wise matrix sum (with time complexity $\Theta(n^2)$
- $\times$ is the usual row-column moltiplication
- A_{ik} and B_{kj} are $\frac{n}{2} \times \frac{n}{2}$ matrices

So it is possible to write the equation

$$T(n) = \begin{cases} 1 & n = 1 \\ 4(2 \cdot T(\frac{n}{2}) + \Theta(\frac{n^2}{4})) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2) & otherwise \end{cases} \tag{2.3}$$

To solve it we can use a recursion tree. Each node is labeled by the cost of that specific call. In particular since we are saying that the cost at each step of the recursion is quadratic, we may select one specific constant such that the cost we are dealing with belong to $\Theta(n^2)$.

The first call cost $c \cdot n^2$. After the first call the matrix is split in four parts, and for each of the four new call we need to go through recursive call, so we have four child node of cost $c \cdot (\frac{n^2}{2})$ and we go on splitting in new children nodes until $n = 1$.
The height of the tree is $\log_2 n$. So

$$\begin{aligned}
T(n) =& c \cdot n^2 + 2 \cdot c \cdot \frac{4}{2^2} \cdot n^2 + ... \\
&= \sum_{i=0}^{\log_2 n} 8^i \cdot c \cdot \frac{n}{n^i}^2 = \sum_{i=0}^{\log_2 n} (\frac{8}{4})^i \cdot c \cdot n \\
&= c \cdot n^2 \cdot \frac{(2^{(\log_2 n+1)} - 1)}{2-1} \\
&= c \cdot n^2 \cdot (2 \cdot n - 1) \\
&= 2 \cdot c \cdot n^3 - c \cdot n^2 \in O(n^3)
\end{aligned}$$

Which is the same as the naive algorithm. It would be nice to reduce the recursive calls. It is possible to do so.

2.5 Strassen's Algorithm

This algorithm consists of doing a bunch of sums that ends up in 10 different matrices, than perform on them 7 matrix multiplications, and from them, doing other sums, we are able to end up with the quadrants of the final matrix. Strassen achieve better results by doing more sums. The recursive equation associated with Strassen's algorithm

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7 \cdot T(\frac{n}{2}) + \Theta(n^2) & otherwise \end{cases} \tag{2.4}$$

The recursion tree this time is So the final time complexity is

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} 7^i \cdot c \cdot \frac{n}{n^i}^2 \\
&= c \cdot n^2 \cdot \sum_{i=0}^{\log_2 n} (\frac{7}{4})^i \\
&= c \cdot n^2 \cdot \frac{((\frac{7}{4})^{\log_2 n+1} - 1)}{\frac{7}{4} - 1} \\
&= c' \cdot n^2 \cdot ((\frac{7}{4})^{\log_2 n+1} - 1) \quad \text{for} \quad c' = \frac{4}{3}c \\
&= c'' \cdot n^2 \cdot (\frac{7}{4})^{\log_2 n} - \frac{4}{7} \\
&= c'' \cdot 4^{\log_2 n} \cdot (\frac{7}{4})^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot 7^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot n^{\log_2 7} - c' \cdot n^2 \in \Theta(n^{\log_2 7})
\end{aligned}$$

So we are able to reduce the complexity to

$$O(n^{log_2 7}) \subseteq O(n^3) \tag{2.5}$$

Chapter 3

Matrix Chain Multiplication Problem