Columbia University
W4231: Analysis of Algorithms
Professor Luca Trevisan

Handout 1
September 7, 1999

# Notes on Models of Computation and Lower Bounds

The following notes complement the material of Chapter 1 of CLR.

## 1 Introduction

In this course we will describe several algorithms that efficiently solve basic computational problems. We will see techniques to *design* such algorithms, to prove their *correctness* and to analyze their *efficiency*.

The algorithms will be described using a pseudo-code that will be a mixture of English sentences and of constructs akin to those of C and Pascal. Though pseudo-code is a very informal way of describing an algorithm, it can be used so that the description is unambiguous and also so that it is apparent what is the number of elementary steps that are executed by a given algorithm on a given input. Our goal will be to show that for any given $n$, the running time of the algorithm for instances of size $n$ is bounded by some slowly growing function of $n$. More on pseudo-code in Section 1.1 of CLR.

In addition to proving efficiency and correctness, one would also like to prove that a given algorithm is *optimal*, namely that the algorithm solves the problem in $T(n)$ time and every other algorithm for the same problem must take time $\Omega(T(n))$ in the worst case. In order to prove the optimality of an algorithm, it is necessary to show a *lower bound* on the time needed by any possible algorithm to solve the problem. A lower bound is interesting in its own even if it does not prove optimality (e.g. if one shows that a problem requires $\Omega(n \log n)$ time, while the fastest known algorithm takes time $O(n^2)$), since it clarifies what are the margins for improvements of the current best algorithms. Usually, a lower bound also uncovers interesting "structural" aspects of the problem being considered.

In order to prove lower bounds, one needs a clear mathematical characterization of what can be done within a given time by an algorithm. Since it lacks a formal specification, pseudo-code is an inappropriate model for this kind of task. Even if it is possible to give a complete formal specification of the C language, it is still too strong a model to hope for simple lower bound arguments. The Random Access Machine (RAM) is a computational model that is (essentially) equivalent to C from the point of view of the time complexity of algorithms, up to $O()$ notation, and has the advantage of admitting a relatively simple mathematical formalization. It is, however, still too strong for lower bound arguments. We will describe below a very simple and abstract model of computation: the *decision tree*. This model has a simple structure, can be described in full mathematical rigor, allows to prove lower bounds, and one does not want to even think of using it to *specify* an algorithm. Luckily, one can show *simulation* results between such models and the generic model provided by the C language. Such simulation results can be seen as stating the existence of *compilers* that given a C problem return a RAM (and, under certain conditions, a decision tree) that does the same operations, with a constant slowdown. Thanks to the simulation results, a lower bound in one of these simplified models implies a lower bound in the C language model.

## 2 Models of Computation

A model of computation is a simplified abstraction of a computer that runs a program. The specification of a model of computation includes a description of what is an "instance" of the model (i.e. a particular machine/program), how the input is presented, how the output is obtained, and how the instance of the model computes its output given its input. Additionally, the model should provide a notion of an "elementary step" during the computation. The running time of an algorithm in the model is the number of elementary steps needed to complete an implementation of the algorithm. Likewise, the model should provide an abstraction of the notion of "memory usage".

### 2.1 Random Access Machines

**Description.** A Random Access Machine (abbreviated RAM) is an over-simplified version of a computer. It is provided with an input device, an output device, a memory unit, and a separate storage unit for the program. Each cell in the memory can hold an arbitrary integer. The program is a sequence of assembly-like instructions. Each instruction in the program has a unique label (for example we can assume that instructions are numbered by consecutive integers). The instruction set contains instructions to read an integer from the input device and store it into a memory location, to fetch an integer from the memory and write it to the output device, to sum the content of two memory locations, to do conditional branching, etc.

A discussion of possible choices for the instruction sets is in Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Programs*. Section 1.2.

**Simulations.** A C program that runs in $T(n)$ instructions, has a RAM implementation that runs in $O(T(n))$ time. Conversely, a RAM program can be simulated in C with a constant slowdown, provided that all the integers being used fit into a basic C data-type (say, a `long`). Under this assumption, the two models are equivalent. If a RAM uses huge numbers, for which it is unrealistic to assume that addition and other operations can be performed in unit time, the log-RAM model is more appropriate. The log-RAM model is the same as the RAM model, except that all operations involving integers stored in the memory take time $\log v$, instead of 1, where $v$ is the value of the integers. A log-RAM can be simulated on a real computer with constant slow-down without any further assumption.

The choice of which model is more realistic depends on the particular application. We will typically consider the standard RAM model, and we will use the log-RAM only for algebraic computations were really huge numbers are being used.

Since we will (almost) always express the running time of algorithms using $O()$ notation, the RAM model gives the same asymptotic bounds of a model based on pseudo-code or the C language. The RAM model has the advantage of being mathematically well-defined, and the disadvantage of being difficult to program.

## 2.2 Decision trees and lower bounds

A decision tree is a totally different kind of model of computation. Firstly, it is not possible to express an *algorithm* in the decision tree model, but only the *behavior of an algorithm for inputs of a fixed length.* So, in order to fully describe an algorithm, one has to specify an *infinite family* of decision trees, one for every input length. The decision tree is of interest only for proving lower bounds. See also Section 9.1 of CLR.

**Description of the model.** A decision tree for an algorithm on inputs of length $n$ is a binary tree. The input is assumed to be a sequence of $n$ integers, represented using the formal variables $x_1, \ldots, x_n$. Each internal node (i.e. each node that is not a leaf) is labeled by some Boolean expression involving the inputs. Typically, an internal node is labeled by a comparison between two inputs $x_i$ and $x_j$, such as e.g. $(x_4 \geq x_9)$, but more involved expressions are admissible. The leaves of the tree are labeled by descriptions of possible outputs. Such descriptions may involve some of the inputs.

Consider for example a decision tree for the problem of finding (and outputting) the minimum of the tree input integers $x_1, x_2, x_3$. The root node makes the comparison $(x_1 < x_2)$.

- If the answer is YES, one goes to a node that makes the comparison $(x_1 < x_3)$:
  - if the answer to $(x_1 < x_3)$ is YES then one goes to a leaf that outputs $x_1$;
  - if the answer to $(x_1 < x_3)$ is NO then one goes to a leaf that outputs $x_3$;
- If the answer to $(x_1 < x_2)$ is NO, one goes to a node that makes the comparison $(x_2 < x_3)$:
  - if the answer to $(x_2 < x_3)$ is YES then one goes to a leaf that outputs $x_2$;
  - if the answer to $(x_2 < x_3)$ is NO then one goes to a leaf that outputs $x_3$;

This should make it clear that one better not use decision trees to specify algorithms. The following paragraphs describe how to use this model to prove lower bounds.

**Computations and running time.** A *computation* in a decision tree works as follows. Given an input $a_1, \ldots, a_n$, one descends in the tree from the root to a leaf, one level at a time. At any given time, one looks at the expression in the current node and evaluates it by substituting $a_i$ in every occurrence of $x_i$ for $i = 1, \ldots, n$. Then one moves to the left child if the expression is true, or to the right child if the expression is false. When a leaf is reached, the label of the leaf is given as output. Therefore the "time" needed to run such a computation is the length of the root-leaf path that has been followed. The worst-case running time is the depth of the tree. More precisely, the worst-case time is the length of a longest root-leaf path that can be descended, where the maximum is taken over all the possible inputs. The latter quantity maybe smaller than the depth of the tree, since the tree may contain some root-leaf paths that are never followed, no matter what the input, because inconsistent expressions should be both true in order for the path to be followed. In any case, the depth of the tree is an upper bound on its worst-case "running-time".

Recall that the depth of a binary tree is at least the logarithm of the number of leaves. This observation yields a lower bound for sorting: the decision tree has $n!$ leaves and therefore must have depth at least $\log_2 n! = \Omega(n \log n)$ (see CLR Section 9.1).

**Simulations.** If we restrict ourselves to RAM algorithm that access their input by only evaluating Boolean expressions (e.g. by making comparisons), then any such RAM algorithm running in time $T(n)$ yields a family of decision trees of depth $\leq T(n)$. This can be proved by "unfolding" the computation of the algorithm and by making a branch in the tree every time the algorithm evaluates a Boolean expression involving the input. The number of branches (the depth of the tree) is upper bounded by the total number of steps of the algorithm. Therefore, a lower bound in the decision tree model implies a lower bound that holds for all the RAM algorithms that only access the input to evaluate Boolean expressions. If the input is a sequence of items from a data type that allows *only* access by Boolean functions, and if the data type is implemented using data hiding so that an algorithm cannot access the input in any other way, then the restriction is done without loss of generality. In particular, if we are looking for RAM algorithms for sorting, and we restrict to algorithms that access the input vector only by making comparisons, then $\Omega(n \log n)$ is a lower bound for such algorithms. In principle, when sorting integers, we can access the input in more sophisticated ways than doing comparisons, and better bound are achievable. For "universal" sorting algorithms, that work with any data type that provides a "comparison" operator, the $\Omega(n \log n)$ lower bound cannot be improved (and it is achievable using e.g. mergersort).

Simulating decision trees by means of a RAM is more problematic. Even if a problem admits decision trees of small depth, there may be no efficient algorithm that finds such trees (or trees of comparable depth). Indeed, one can come up with undecidable problems[1] that admit small-depth decision trees. Since we want to use the decision tree model only in order to prove lower bounds, it is enough to have only one type of simulation.

[1] i.e. problems that do not admit algorithms at all, no matter how inefficient.