ESO207 (Data Structures and Algorithms): Lecture Notes Set 1

Shashank K Mehta

1 Random Access Machine

The computers of today can be simplified into the following model. These have a control unit, a memory where locations have address 0 to N, and a data-path where data manipulation is performed. Initially the program (a sequence of instructions) is placed in the memory, relevant data is placed in the memory (it is ensured that two remain disjoint), and the control unit is initialized with the memory address of the first instruction (by setting the address in the PC register.)

The control unit reads the address of the next instruction in PC, brings it from the memory, reads it and brings all the data referred in this instruction, sets up the datapath to perform the necessary manipulations and finally it restores the results back into the memory in the location mentioned by the instruction. In this cycle the PC is also suitably updated. Computer repeats this infinitely.

It is important to remember that accessing (reading/writing) any data item takes a fixed amount of time. Similarly each data manipulation (arithmetic, logic operation) has a fixed time requirement.

2 The Concept of Problem Size

A problem for which a program is written is a family of problem instances. For example a program that computes the sum of two numbers, say $a + b$, has one instance for each pair of values of a and b. Now consider two instances: $7 + 5$ and $690704374596869504 + 6045821019039569076$. Do you think both instances will take the same amount of time? In some way the latter instance appears to be "larger". So we need to define a notion of the "size" of the instances. We might define the number of digits in the larger of the two numbers as the "size" of the instance. In general the "size" represents the data size which is crucial in deciding the time it takes perform the computation.

Formally, the size is taken to be the number of bits required to store the input data. But informally we will simplify this.

3 Complexity

3.1 Measure of Time and Space

We can determine the time it takes to execute a program in a given computer. But there are variety of computers and each type of machine has different capability. The execution time on each machine is likely to be different. Thus how to define the time of a program or algorithm independent of an underlying machine? Each machine has a datapath which performs basic manipulation of the data. The datapath has some things called "functional units" which perform arithmetic and logic computations. Each functional takes a fixed number of time cycles (called clock). The computers are have different clock speed so even if two computers may finish a job in the same number of clocks but actual time may vary, depending the clock speed. To avoid this problem we take the time of the slowest functional unit to be one unit of time. So all functional units take at most 1 unit of time.

We may measure the time of an algorithm in terms of number of functional unit operations. But that too has a problem. Some computers may have a simple instruction set, i.e., very few functional unit performing very basic operations. Others may have dedicated functional units to perform complex operations. For example one computer can perform only add, subtract, divide, and multiplication. The second computer may do all these operations and may also compute square-root of a number. The first computer will compute square-root using an algorithm.

To avoid this problem we will assume that only elementary arithmetic and logic operations can be performed by the computer and they take unit time.

Informally we will simply consider the number of arithmetic/logic operations as time measure and the number of pieces of data as the space measure. Let $T(I)$ denote the time taken by an instance I and $S(I)$ denote the space taken by that instance.

3.2 Variations and Worst Case Time/Space

Generally the time measure not only varies with size, for example $GCD(4, 15)$ versus $GCD(4295783859676, 589695940302)$, but it also varies with in the instances of the same size, for example $GCD(4, 15)$ versus $GCD(4, 16)$. It takes two iterations for $GCD(4, 15)$ but four for $GCD(4, 16)$. To deal with this we will consider the worst case time/space required among the instances of a given size. And we will assume that $T()$ and $S()$ are functions of only the size.

Even when we consider functions $T(size)$ and $S(size)$ we will find that these are not smooth functions. Imagine that your task is to find the prime factors of a given number. Clearly a number with large number of prime factors will take more time compared to one with fewer factors. And you can have a very large prime number and a very small composite number with many prime factors.

If you draw $T(size)$ against $size$, then you will see an erratic curve. So we only attempt to estimate the worst case times by finding a smooth curve $f(size)$

(a polynomial or e^{size} or $\log(size)$) such that $T(size) \leq f(size)$. We try to find f such that it binds T as tightly as possible to give as good an estimate as possible.

There is another approach to estimate the time performance of an algorithm. Here we take the average of all the time performance of the instances of the same size. Usually we take the average with equal weight for each instance. Thus $T(size)$ denote this average. Once again we want to find a smooth function $f()$ such that $T(size) \leq f(size)$.

Example: Search a number N in an array of n number. Observe that it may take any thing between 1 comparison to n comparisons, depending on the position of the target number. So the worst case T is N but the average case value is $N/2$.

3.3 Dealing with Non-Smooth Functions

In measuring the time taken by an algorithm for a problem we notice that often the time is not a smooth function of the size. We expect that the function may not even be monotonic at "microscopic" scale although it is always monotonically increasing at "macroscopic" scale. To understand the performance of the algorithm we prefer to see how fact does it increase with the size. Thus we always like to find a smooth function which binds the erratic function from above or below.

Let $F(x)$ be a given (erratic) function and $g(x)$ is a smooth function. Then $F(x) = O(g(x))$ means that there exists a constant c and a value x_0 in the domain such that $F(x) \leq c.g(x)$ for all $x \geq x_0$. Similarly $F(x) = \Omega(g(x))$ means that there exists a constant c and a value x_0 in the domain such that $F(x) \geq c.g(x)$ for all $x \geq x_0$. Finally, by $F(x) = \Theta(g(x))$ we mean that there exists a constants c_1, c_2 and a value x_0 in the domain such that $c_1 g(x) \leq F(x) \leq c_2.g(x)$ for all $x \geq x_0$. Equivalently it means $F(x) = \Omega(g(x))$ as well as $F(x) = O(g(x))$.

3.4 Asymptotic Complexity

We actually further simplify the estimate of time/space cost. We are mostly interested in the *rate* at which the (worst case / average case) cost increases with the size. So we do not differentiate between a function F and $c.F$ where c is a constant. Both have the same shape, i.e., same rate of growth. Therefore we find a smooth function $g(s)$ such that $T(s) = O(g(s))$, where s denotes the size and $T(s)$ denotes the maximum time taken by the algorithm for any instance of size s.

We do not discuss the lowerbound of an algorithm. Instead, we are always interested in the lowerbound for a problem. We say that a problem has time complexity $\Omega(g(s))$ to indicate that the worst case time complexity of any algorithm will take at least $c.g(s)$ time no matter which algorithm is used, including those algorithms which are not yet discovered.

In case an algorithm has time complexity $O(g(s))$ and the lower bound for this **problem** is $\Omega(g(s))$, then we express this fact by the notation $\Theta(g(s))$.

3.5 Where Time Complexity is Not a Deciding Factor

1. If the program is to be used on small size instances, then it may not be necessary to design a very efficient algorithm which only shows improvement asymptotically.
2. Some times the constant c is so large that in practice program with worse complexity do better.
3. Too complex algorithms may not be desirable as one needs to maintain the software.

4 Analysis of Some Simple Algorithms and Introduction to Pseudo Language

4.1 Bubble Sort

Problem: Given a set of n numbers, order them in a sequence such that no number is less than its predecessor, i.e., sort them in the non-decreasing order.

Algorithm:

Input: Let input be in an array A. So $n = length(A)$.

For $i = 0$ to $n - 2$ do

For $j = i$ to 0 do

If $A[j] > A[j + 1]$

Then $\{temp := A[j]; A[j] := A[j + 1]; A[j + 1] := temp\}$

Return A.

Time complexity Analysis: Left as an exercise.

4.2 Factorial

recursive and non-recursive: Left as an exercise.

4.3 GCD$(a_0.a_1)$

GCD(a_0, a_1) /* where $a_0 \geq a_1$ */ $x := a_0$;

$y := a_1$;

while $(y > 0)$ do

$temp := y$;

$y := remainder(x \div y)$;

$x := temp$;

return x;

4.4 Analysis

The GCD algorithm takes two inputs. Assume that a_0 and a_1 where $a_1 \leq a_0$. So the input size is $s = \log_2 a_0 + \log_2 a_1$. To analyze this algorithm we will attempt to find the $T(a_0) = max_{a_1}\{Time(GCD(a_0, a_1))\}$. This will leave us with only one parameter to keep under consideration. Then we will attempt to find a smooth function $g(a_0)$ such that $T(a_0) \leq g(a_0)$. Then we will state that the worst case time complexity of the algorithm is $O(g(a_0))$. If we want to state the time complexity in terms of s, then we modify the statement as follows. Note that $s \geq \log_2 a_0$. So $g(2^s) \geq g(a_0)$ because g is an increasing function. So we can state that the time complexity is $O(g(2^s))$.

4.4.1 Computing Bound for $max_{a_1}\{GCD(a_0, a_1)\}$

The algorithm computes as follows, where q_i denote the respective quotients and these are non-negative integers.
$a_2 = a_0 - q_1 a_1$
$a_3 = a_1 - q_2 a_2$
...
$a_n = a_{n-2} - q_{n-1} a_{n-1}$
$a_{n+1} = a_{n-1} - q_n a_n = 0$

This reduction takes n steps/iterations to compute the $gcd = a_n$.

So
$a_{n-1} = q_n a_n + a_{n+1}$
$a_{n-2} = q_{n-1} a_{n-1} + a_n$
$a_{n-3} = q_{n-2} a_{n-2} + a_{n-1}$
...
$a_{i-1} = q_i a_i + a + i + 1$
...
$a_0 = q_1 a_1 + a_2$

The question we want to answer is: what is the smallest a_0 which will lead to n iterations?

From the above relations we see that the smallest value of a_0 is possible when each $q_i = 1$. Besides we also see that the smallest value of a_{n-1} can be 2. For this case let the resulting value of a_i be denoted by b_i.

So we see that $b_{n+1} = 0, b_n = 1 = F_2, b_{n-1} = 2 = F_3, b_{n-2} = b_n + b_{n-1} = F_4, \ldots, b_0 = F_{n+2}$, where F_i is the i-th Fibonacci number. Thus we conclude that $a_0 = F_{n+2}$ is the smallest number for which this algorithm requires n iterations. Define a step function $Step : \mathbb{Z} \to \mathbb{Z}$ given by $Step(a) = n$ if $F_{n+2} \leq a < F_{n+3}$. Then we know that $GCD(a, b)$, where $b \leq a$, will require at most $Step(a)$ iterations.

A step function is not a smooth function. So we want to determine a function g such that $Step(a) \leq g(a)$ for all a. Thus we require that (i) $g(a) \geq F_a$ and

(ii) $g()$ is an non-decreasing function.

4.4.2 Finding a smooth function g

Claim 1 $1.5 \leq F_i/F_{i-1}$ *for all* $i \geq 3$.

Proof We will show this claim by induction.

Base case: $F_3/F_2 = 2/1 = 2 > 1.5$.

Induction step: For any $i > 3$, $F_i/F_{i-1} = (F_{i-1} + F_{i-2})/F_{i-1} = 1 + F_{i-2}/F_{i-1}$. Observe that F_j is an increasing sequence. So $F_{i-1} = F_{i-2} + F_{i-3} \leq 2F_{i-2}$ or $F_{i-2}/F_{i-1} \geq 1/2$. Thus we have $F_i/F_{i-1} \geq 1 + 1/2 = 1.5$. ■

From this claim we see that $F_{n+2} = (F_{n+2}/F_{n+1})(F_{n+1}/F_n)) \ldots (F_3/F_2).F_2 \geq (1.5)^n$. Hence $n \leq \log_{1.5} F_{n+2}$. Hence a possible g function is $\log_{1.5}$, i.e., $g(a) \geq \log_{1.5} a \forall a$.

4.4.3 The Complexity

The deduction given above leads to the conclusion that the time taken by $GCD(a_0, a_1)$ is at most $c.\log_{1.5} a_0$ for some constant c. So we can state that the time complexity of this algorithm is $O(\log_{1.5} a_0)$.

The problem has two structural parameters, a_0 and a_1. The parameters are integers so their sizes is $\log_2 a_0$ and $\log_2 a_1$. So the size of the instance is $s = \log_2 a_0 + \log_2 a_1$. If we want to state the time complexity in terms of the size, then observe that $s \geq \log_2 a_0 \geq (1/c').\log_{1.5} a_0$, where $c' = log_{1.5}2$. Also, g is a non-decreasing function. So the number of iterations will be at most $c'.s$. So the time complexity is $O(s)$.