ESO207 (Data Structures and Algorithms)
Lectures 1 to 3

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, relevent 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 (arithmatic, 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 bytes (8-bit set) required to store the 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 arithmatic and logic computations. It is possible that some machines may have complex functional units and some may only do simple functions.

Formally we measure the time of computation by the number of functional-unit operations performed by the computer. The space is measured by the largest number of bytes used at any point in time of the computation.

Informally we will simply consider the number of arithmatic/logic operations as time measure and the number of pieces of data as the space measure. Let $g_t(I)$ and $g_s(I)$ denote the time and space measure of instance I respectively.

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 g_t and g_s are functions of only the size.

Even when we consider functions $g_t(size)$ and $g_m(size)$ 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 $g_t(size)$ 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 $g(size) \leq f(size)$. We try to find f such that it binds g_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 $g_t(size)$ denote this average. Once again we want to find a smooth function $f()$ such that $g_y(size) \leq f(size)$.

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 g_t is N but the average case value is $N/2$.

4 Asymptotic Complexity

We actually further simplify the context of time/space cost. We are mostly inteested in the *rate* at which the 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. What we need is a function $f(size)$ such that for some constants n_0 amd c, $c.f(size) \geq g_t(size)$ for all $size \geq n_0$. If this holds for some function f, then we say that the *time complexity* of the algorithm is $O(f(size))$ (i.e., big-O of f). Clearly big-O described the performance of an algorithm.

Similarly if we have a function $h(size)$ such that for some constants n_0', c', $c'h(size) \leq P_t(size)$ for all $size \geq n_0'$, then we say that the problem is $\Omega(h(size))$. Here $P_t(size)$ is the minimum time required to solve any instance of size $size$, no matter what algorithm is used. Observe that $\Omega()$ describes the hardness of a problem. One must guarantee that this is a minimum required time no matter which algorithm is used (known or yet to be invented).

In case an algorithm is $O(f(size))$ and the lower bound for this problem is $\Omega(f(size))$, then we express this fact by the notation $\Theta(f(size))$.

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.

6 Analysis of Some Simple Algorithms and Introduction to Pseudo Language

6.1 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-degreasing 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.

6.2 Factorial

recursive and non-recursive: Left as an exercise.

6.3 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;

The algorithm computes as follows, where q_i denote the respective quotients.

$a_2 = a_0 - q_1 a_1$
$a_3 = a_1 - q_2 a_2$
...
$a_n = a_{n-2} - q_{n-1} a_{n-1}$
$a_{n+1} = a_{n-1} - q_n a_n = 0$

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 relationss 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. If the algorithm requires n-iterations then the number of basic cmputations will be $const.n$. Equivalently if the algorithm requires n iterations, the its time complexity will be $O(n)$. Now our goal is to express this time complexity in terms of a_0. We have deduced above that if $a_0 \leq F_{n+2}$, then the time complexity will be $O(n)$, for all n. So we want to find a non-decreasing function g such that $n \leq g(F_{n+2})$. Then the complexity will be $O(g(a_0))$. Here we are using a_0 as the size parameter. If we want to use $\log_2 a_0$, then we must modify g accordingly.

6.3.1 Finding a smooth function g

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

Proof W ■e will show this claim by induction.

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

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 $F_{i-2}/F_{i-1} \leq 1/2$. Thus we have $F_i/F_{i-1} \leq 1 + 1/2 = 1.5$.

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 $\log_{1.5}$.

So we conclude that the time complexity is $O(\log_{1.5} a_0)$.

6.3.2 The Complexity is Tight

We know that $a_0 = F_{n+2}$ requires exactly n iterations. We want to show the existance of a constant α such that $\alpha.\log_{1.5} F_{n+2} \leq n$ for all $n \geq n_0$ for some fixed n_0.

This time we use induction to find α, if it exists.

Suppose for some α, $\alpha.\log_{1.5} F_{n+1} \leq n - 1$ and we want to show that $\alpha.\log_{1.5} F_{n+2} \leq n$. So $\alpha.\log_{1.5}(F_{n+2}/2) \leq \alpha.\log_{1.5} F_{n+1} \leq n - 1$. Simplifying the expression gives $\alpha.\log_{1.5} F_{n+2} \leq n + (\alpha.\log_{1.5} 2 - 1)$. So the desired condition holds if $0 < \alpha \leq 1/\log_{1.5} 2$. Note that $(1/\log_{1.5} 2).\log_{1.5} F_3 = 1$. So $\alpha = 1/\log_{1.5} 2$ satisfies the required inequality, i.e., $(1/\log_{1.5} 2).\log_{1.5} F_{n+2} \leq n \leq \log_{1.5} F_{n+2}$ for all $n \geq 1$. This estabilshes that complexity $O(\log_{1.5} a_0)$ indeed tight.