CSE548/AMS542 Fall 2013 Analysis of Algorithms

Homework 1

Himanshu Shah
ID : 109324380

Homework 1

Problem 1:

A) Ch-2 Problem 4 [KT]:

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)).

$(a) g_1(x) = 2^{\sqrt{\log n}}$

$(b) g_2(x) = 2^n$

$(c) g_4(x) = n^{4/3}$

$(d) g_3(x) = n(\log n)^3$

$(e) g_5(x) = n^{\log n}$

$(f) g_6(x) = 2^{2^n}$

$(g) g_7(x) = 2^{n^2}$

Solution:

It can be easily noted that $g_7(x) = O(g_6(x))$ and $g_2(x) = O(g_7(x))$. Also on taking the value of n=4, 16 and 64 the following order of growth can be seen:

$g_1(x) = O(g_4(x))$

$g_4(x) = O(g_3(x))$

$g_3(x) = O(g_5(x))$

$g_5(x) = O(g_2(x))$

Hence the growth of the functions in increasing order is :

$g_1(x), g_4(x), g_3(x), g_5(x), g_2(x), g_7(x) \text{ and } g_6(x)$. With $g_1(x)$ being the lowest and $g_6(x)$ being the largest

B) Ch-2 Problem 5 [KT]:

Assume you have functions f and g such that f(n) is O(g(n)). For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(i) $\log_2 f(n)$ **is** $O(\log_2 g(n))$

Solution:

True

- This is evident from the fact that Log is an increasing function. Hence if $x > y$ then $\log_2(x)$ will be greater than $\log_2(y)$.

f(n) is O(g(n)) so, let f(n) = x^2 and g(n) = x^3.

Now $x^2 \leq cx^3$
Applying $\log_2$ on both sides
$\log_2(x^2) \leq c\log_2(x^3)$
Hence, proved.
(ii) $2^{f(n)}$ is $O(2^{g(n)})$
-True
Proof by Induction:
For f(n) = O(g(n)) let $2^{f(n)}$ is $O(2^{g(n)})$
So, if f(n) = n^2 and g(n) = n^3.

$2^{n^2} \leq c\,2^{n^3}$

Now lets prove this for $f(n+1)$ and $g(n+1)$

L.H.S $= f(n+1)$
$= 2^{(n+1)^2}$
$= 2^{x^2+2x+1}$
$= 2^{x^2} 2^2 2$
$\leq C\,2^{x^3+3x^2+3x+1}$ because $2^{n^2} \leq c2^{n^3}$ for $n \geq 1$
$\leq C\,2^{(x+1)^3}$
Hence $2^{f(n)}$ is $O(2^{g(n)})$.

(iii) $f(n)^2$ is $O(g(n)^2)$
- True
- Here we will prove using the property of Big-Oh function.
According to the rule of Big-Oh multiplication by a function,
$O(f(n))O(g(n)) = O(f(n)g(n))$

So, as $f(n)$ is $O(g(n))$
if we multiple $f(n)$ with $f(n)$ we get a
$f(n)^2 = O(g(n)g(n))$

$f(n)^2 = O(g(n)^2)$

C) Ch-2 Problem 8 [KT]:

Solution a)
This problem can be solved using the following strategy:
Try dropping the jar at every even number of rugs starting from level L = 2 and then increasing to $L_{new} = L_{current} + 2$, until, the jar breaks.
Go one level lower $L_{new} = L_{current} - 1$, if the jar breaks, its highest safe rug level is $L_{current} - 1$.
Else it is $L_{current}$.

Example suppose a jar has the safe level of 6. Then, as per the strategy, one should try dropping it at level 2 , 4 ,6 and then 8. The jar will break on 8. So try level 7, and it will break again and hence proving that 6 is its highest safe rug level.

Solution b)
Solution for question b is similar to the first one. Infact the solution a is a special case of solution b where number of jars =2.
For this problem, I am considering k number of jars that can be used at maximum and $f_k(x)$ is the number of times an attempt is made. Here, n is the total number of rugs in the ladder.

The strategy is as follows.

Step-1 Based on k, test the strength of the jar at m * k level starting m from 1 and incrementing it by one until the jar breaks.

Step-2 Test the strength at one level less than the previous height and see if it breaks. continue until it stops breaking.

This level is the highest safest level for the jar.

> This algorithm will grow asymptotically lower than as the value of k will increase.

For k=3, n=10 and the highest safest level =7, this algorithm will work as:
> First attempt : 3
> Second attempt : 6
> Third attempt : 9 - breaks
Fourth attempt : 8 - breaks
Fifth attempt : 7 - does not break - SOLUTION Found

For k=4, n=10 and the highest safest level =7, this algorithm will work as:
> First attempt : 4
> Second attempt : 8 - breaks
Third attempt : 7 - does not break - SOLUTION Found

Problem 2. Prove or disprove (i.e., give counter examples) for the following claims. f (n), g(n) are non- negative functions.

$(a)\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Let $f(n) = n^2$ and $g(n) = \log_2 n$
So max($f(n), g(n)$) = n^2
We need to prove $c_1(n^2 + \log_2 n) \leq n^2 \leq c_2(n^2 + \log_2 n)$
We can prove that
$n^2 \leq c_2(n^2 + \log_2 n)$
for $c_2 > 0$ and $n \geq 1$
however for any $c_1 > 0$ and $n \geq 1$
$c_1(n^2 + \log_2 n) \leq n^2$ does not hold.
So this is false.

$(b)O(f(n)) \cap \omega(f(n)) = \emptyset$

Consider $f(n)$ is $O(g(n))$ and $f(n)$ is $\omega(g(n))$
hence, $f(n) \leq c_1 g(n)$ .....(i)
and $c_2 g(n) \geq f(n)$ .....(ii)

Thus,
$c_2 g(n) \leq f(n) \leq c_1 g(n)$

This implies that $O(f(n)) \cap \omega(f(n)) = f(n)$ and not $\emptyset$

Example, $f(n) = 6n^2$
then,
$c_2 n^2 \leq f(n) \leq c_1 n^2$
where $c_1 = 7$ and $c_2 = 3$

Thus, proved.

$(c)(n + a)^b = \Theta(n^b)$, a, b are positive integers

Consider a=1, and b=3

so $(n + a)^b = (n + 1)^3$
$= n^3 + 3n^2 + 3n + 1$
$= O(n^3)$
$\leq c_1(n^3)$ for c_1 greater than 0.
but $n^3 + 3n^2 + 3n + 1$ cannot be $\geq$ for $c_2(n^3)$ with c_2 greater than 0.
Hence proved.

$(d) f(n) = O(f(n)^2)$
Proof by induction:
Let $f(n) = n$ so $n \leq c n^2$ for $c > 0$

Lets prove the same for $f(n + 1)$
LHS $= (n + 1)$
$\leq c n^2 + 1$
$\leq c n^2 + 2n + 1$
for n > 1.
Hence $f(n) = O(f(n)^2)$.

$(e) f(n) = O(g(n))$ implies that $2^{f(n)} = O(2^{g(n)})$
For f(n) = O(g(n)) let $2^{f(n)}$ is $O(2^{g(n)})$
So, if $f(n) = n^2$ and $g(n) = n^3$.

$2^{n^2} \leq c 2^{n^3}$

Now lets prove this for $f(n + 1)$ and $g(n + 1)$

L.H.S $= f(n + 1)$
$= 2^{(n+1)^2}$
$= 2^{x^2+2x+1}$
$= 2^{x^2} 2^2 2$
$\leq C 2^{x^3+3x^2+3x+1}$ because $2^{n^2} \leq c 2^{n^3}$ for $n \geq 1$
$\leq C 2^{(x+1)^3}$
Hence $2^{f(n)}$ is $O(2^{g(n)})$.

Problem 3:
Problem regarding the previous large element question.

I would like to state two algorithms, one has the running time of $O(n^2)$ and the other one has a running time of $O(n)$.

Algo 1:

For each element a_j in a series $a_1, a_2, a_3, \ldots\ldots, a_{j-1}, a_j$ we perform a linear search from a_{j-1} to a_1 for finding a number greater than a_j, and we would stop as soon as we have got one.

Now the time complexity of such an algorithm will be :
$=$ The total number of elements X maximum work done for each element
$= n(n-1)$.
$= O(n^2)$.

Algo 2:
I would like to refer to the previous large element notation as PLE from here onwards.

In this method, we will be maintaining a stack.
For all the elements in the sequence, we will start visiting from the right hand side and move towards the left hand side one step at a time.
Let the current element being visited be X . We will perform the following operation for each element we traverse:
Step 1 - IF stack is empty, initialize the stack - Push(X)
Step -2 Else compare the element with the top of the stack:

IF Top of stack > X
Push (X)
Else
Pop the element from the stack and set the PLE of the element popped.
go to Step 2.

Step -3 After all the elements are traversed, and pop all the elements from the stack and for each element set PLE as 0.

Time complexity of this algorithm is going to be $O(n)$ because we are traversing each element in the sequence only once.

Example : Consider a sequence of numbers : 5,14,11,10,13,12,15

So as per this algorithm:

Stack: ----	Element: 15	PUSH	
Stack: 15	Element: 12	PUSH	
Stack: 15,12	Element: 13	POP	12 --->13
Stack: 15	Element: 13	PUSH	
Stack: 15,13	Element: 10	PUSH	
Stack: 15,13,10	Element: 11	POP	10--->11
Stack: 15,13	Element: 11	PUSH	
Stack: 15,13,11	Element: 14	POP	11--->14

Stack: 15,13	Element: 14	POP	13--->14
Stack: 15	Element: 14	PUSH	
Stack: 15,14	Element: 5	PUSH	
Stack: 15,14,5	----------	POP	5 ---> 0
Stack: 15,14	----------	POP	14--->0
Stack: 15	----------	POP	15--->0

Problem 4:

Given n unsorted numbers, compute the maximum and minimum using $\lceil 3n/2 \rceil - 2$ comparisons.

Solution:

The algorithm to compute the maximum and minimum on n elements in $\lceil 3n/2 \rceil - 2$ comparisons can be done as follows :

Step -1 :
Compare each of the even numbered element with the consecutive element in the list and determine the one that is greater. Add the elements that are larger in a set and name it SET A and the ones that are smaller in another SET B.

Step -2 :
For each element in SET A find the maximum by performing a linear search.

Step - 3:
For each element in SET B find the minimum by performing a linear search.

At the end of the algorithm, we will be able to determine the maximum and minimum from a given set of n numbers.

Calculating the number of comparisons made in each step of the algorithm:

Step -1 : For a given set of n numbers in order to generate two sets for larger and smaller numbers, one needs to make $\lceil n/2 \rceil$ Comparisons.

Step -2 : In order to find the maximum from SET A using linear searching technique, one needs to perform, $\lceil n/2 \rceil - 1$ comparisons. This is because, the number of elements in the set is $\lceil n/2 \rceil$. So, the number of comparisons required is one less than the total number of elements in the set.

Step -3 : In order to find the minimum from SET B using linear searching technique, one needs to perform, $\lceil n/2 \rceil - 1$ comparisons. This is because, the number of elements in the set is $\lceil n/2 \rceil$. So, the number of comparisons required is one less than the total number of elements in the set.

Hence, total number of comparisons is $\lceil n/2 \rceil + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1$
$= 3\lceil n/2 \rceil - 2$
$= \lceil 3n/2 \rceil - 2$

Hence, proved.

Problem 5:
Page 107 Problem #4 :

Considering n specimens and collection of m judgments for the pairs that were not declared ambiguous, the algorithm for the problem can be written as:

Create two sets A and B:
For each pair of specimen (i,j) or (j,i) and its corresponding judgment, perform the following:
Start from node i, add the node to set A.

--If node j is not present in any of the two sets (i.e. if it isn't already traversed)
Judgment = "same" -- ADD node j -- to set A,
Otherwise -- ADD node j -- to set B.
Continue with pairs that contain either of i or j (Neighbors).

--Else
If the node is present in the correct set (The set determined from the above if condition), continue with the pairs that contain either of i or j (Neighbors).
else
break and mark the judgment to be inconsistent.

The judgments are consistent if the algorithm exits normally.

Running Time:
This algorithm is a slightly modified version of Breadth first Search and hence its running time is $O(|m| + |n|)$ where m is the number of judgments and n is the number of specimens.

Page 107 Problem #6 :

Considering Graph G=(V,E) and Tree T for the below proof.

In a BFS tree, the edges of the graph G that appear in tree are either connected in the same layer or to adjacent layer.

So based on this property the graph can contain two types of edges, either in the same or in adjacent layers.

But, as this tree is also a DFS tree, the property that for any nodes x and y in the tree, either x or y is an ancestor of the other prevails.

Hence only one type of edges are present and they are only between adjacent layers.

This proves the fact that there cannot be any other types of edges in T and hence no extra edges are present in G.

Hence, proved.

Page 107 Problem #10 :

We can apply a slight modification of the BFS Algorithm for this problem.
The central idea for the algorithm is to maintain a list of reachability information to every node from the starting node and use this information to calculate the number of possible shortest paths.

In order to find the number of shortest paths between node(i,j), we will start performing a BFS graph traversal from node i.
- We will assign the reachability of node i as 0.
- From node i, we will assign the reachability as 1 for all the neighbors of i.
- Similarly we will perform BFS on each of the neighbors of node i and assign their reachability as , 1 + reachability of previous node.
- We will perform this until all the nodes have been traversed.

Once all of this information is gathered, we need to traverse the graph from reverse direction, from node j and count the number of incoming edges that have the lowest reachability number.

We need to follow all such edges until node i, and increment the counter every time , if there are more than one edges having same lowest reachability which are incident on a node.

The running time for his algorithm is $O\ (m\ +\ n\ )$ because we are using primarily using BFS for most of the computation.