Sri Dhanush Reddy Kondapalli
CWID: 10476150

CS 600 Advanced Algorithm Design and Implementation Midterm

1. (6 Points) Using the very definition of Big-Theta notation, prove that 2n+1 is θ(2n). You must use the definition and finding the constants in the definition to receive credit.

Sol: Given, $f(n) = 2^{n+1}$ and $g(n) = 2^n$

From the definition of Big-Oh, $f(n) <= cg(n)$ for every integer $n >= n_0$, $c > 0$ and integer constant $n_0 >= 1$

So substituting with the given functions,

$2^{n+1} <= 4*2^n$, $c = 4$, $n_0 = 1$

$\rightarrow 32 <= 64$

This satisfies the condition.

Hence, f(n) is O(g(n))

From the definition of Big-Omega, $f(n) >= cg(n)$ for every integer $n >= n_0$, $c > 0$ and integer constant $n_0 >= 1$

Now substituting our given functions,

$2^{n+1} >= 1*2^n$, $c = 1$, $n_0 = 1$

$\rightarrow 16 >= 9$

This satisfies the condition.

Hence, f(n) is Ω(g(n))

From the definition of Big-Theta, if f(n) is O(g(n)) and f(n) is Ω(g(n)) this implies that f(n) is Θ(g(n)).

Hence proved.

2. (7 Points) Given that T(n) = 1 if n=1 and T(n) = T(n-1)+ n otherwise; show, by induction, that T(n) = n(n+1)/2. Show all three steps of your induction explicitly.

Sol: Base: Given that, T(1) = 1

T(n) = T(n-1) + n

So T(2) = T(1) + 2 = 3

Assume: For n=k

T(k) = k(k+1)/2

Induction: For n=k+1

T(k+1) = T(k) + k + 1

= k(k+1)/2 + k + 1

= (k(k+1) + 2k + 2)/2

= (k2 + k + 2k + 2)/2

= ((k+1)(k+2))/2

= T(k+1)

Since both equations are satisfied, hence proved that T(n) = n(n+1)/2 by Induction.

3. (7 Points) Given the recurrence relation T(n) = 7 T(n/5) + 10n, for n>1; and T(1)=1. Find T(625).

Sol: In the given recurrence relation we have to evaluate for $n = 5^s$, where $s > 0$, since we have been given to find T(1)

Using the given formula,

Sri Dhanush Reddy Kondapalli
CWID: 10476150

$T(n) = 7T(\frac{n}{5}) + 10n$

$T(625) = 7*T(125) + 10*625$

$= 7*[7*T(25) + 10*125] + 10*625$

$= 7*[7*\{7*T(5) + 10*25\} + 10*125] + 10*625$

$= 7*[7*\{7*(7*T(1) + 10*5) + 10*25\} + 10*125] + 10*625$

So, in general to evaluate the relation for $n = 5^s$, so we can express in relation.

$T(5^s) = 7^s + \Sigma\{7^{(s-k-1)}*10*5^{(k+1)}\}$ where k can varies from 0 to (s-1), which is given s>0.

When we will calculate this equation then we found:

T(625) = 46801

4. (16 Points) Suppose that we implement a union-find structure by representing each set using a balanced search tree. Describe and analyze algorithms for each of the methods for a union-find structure so that every operation runs in at most O(logn) time in worst case.

Sol: Given that the union-find data structure is achieved by employing a balanced search tree to represent each set. Each element contained in the node carries a reference to the set name, which is the tree's root. A single set is a node that points to itself. A union-find data structure has three operations: makeSet(), union(), and find(). The algorithms for the aforementioned approaches are listed below.

Algorithm:

makeSet():

For every element 'a' do

a.root ← a

a.size = 1

This algorithm runs in O(1) time.

union(a, b):

If a.size < b.size then

    a.root ← b

    b.size = a.size + b.size

Else

    b.root ← a

    a.size = a.size + b.size

This union function runs in O(1) time.

find(a):

c ← a

While c.root p = c do

    c ← c.root d ← a

While d.parent p = d do e ← d

    d ← d.root e.root ← c

We only modify a node's parent pointer in the above find function when we union its set into a set that is at least as large as it. As a result, each time we perform a search function from a node to its parent, the size of the set rooted at that node must at least double. As a result, the longest series of parent pointers we can march over while doing a search is O(log n).

As a result, all methods of a union find data structure are designed to run in at most O(log n) time.

Sri Dhanush Reddy Kondapalli
CWID: 10476150

5. (16 Points). Recall Homework 2 Exercise 2.5.13, where you implemented a stack using two queues. Now consider implementing a queue using two stacks S1 and S2 where:

enqueue(o): pushes object o at the top of the stack S1

dequeue(): if S2 is empty then pop the entire contents of S1 pushing each element onto S2. Then pop from S2.

If S2 is not empty, pop from S2.

It is easy to see that this algorithm is correct. We are interested in its running time.

a) (4 Points) Show that the conventional worst case running time of a single dequeue is O(n).

b) (12 Points) Show that the amortized cost of a single dequeue is O(1). You must use Amortization Method to receive credit.

Sol: A stack is a container for things that are added and deleted using the last-in-first-out (LIFO) principle. Only two operations are permitted in pushdown stacks: pushing an item into the stack and popping an item out of the stack. A queue is a container for objects (a linear collection) that are added and deleted in accordance with the first-in-first-out (FIFO) principle. Only two operations are permitted in the queue: enqueue and dequeue. Enqueue means to add an item to the rear of the queue, whereas dequeue means to remove the item from the front.

a) Enqueue(x) takes O(1) everytime because it is just insertion of an element in the array. Dequeue(x) takes O(n) sometimes. When the S2 is empty, we've to pop all the elements in S1 onto S2. It takes O(n) time.
Here, contents S1 are placed on S2 in reverse order.
Therefore, enqueue(x) = O(1) & dequeue(x) = O(n).

b) Given two stacks are empty i.e S1 = 0, S2 = 0
If S2 not equal to 0 pop (x, S2)
If S2 is equal to 0 pop all item form S1 and push to S2.
Pop S2(return)
Amortized cost :- given a sequence of n operations the amortized cost is cost (n, operations)/n.
We can use aggregate method
O(1) = S1 = n(X,S2)/n not equal to zero
O(1) = S2 = n(x, S2)/n equal to zero pop (S1) push(2).
S1 = S2 = 0.
Stack operation
Push(s,x),O(1)
Pop(s) O(1)
Multipop (s,k), min(s, k)
While not stack = empty(s) and k > 0
Do pop (s)
K = k - 1

After amortization analysis, the cost of a single enqueue and dequeue operation is found to be O(1). Thus, the amortized cost of a single dequeue is O(1).

Sri Dhanush Reddy Kondapalli
CWID: 10476150

6. (16 Points) Consider an n by n matrix M whose elements are 0's and 1's such that in any row, all the 1's come before any 0's in that row. Assuming A is already in memory, describe an efficient algorithm for finding the row of M that contain the most of 1's. What is the running time of the algorithm?

Sol: Considering a matrix M of order n x n, whose elements contains only 0's or 1's where 1's come only before 0's in a row of that matrix. To find the row with maximum number of 1's we can traverse the matrix as:

- We start with 1st element in the first row, if it is 0 then traverse to next row in same column. Else check the next element in same row.
- In the next row, check if element to the right of current position is 0, if yes then ignore the row.
- Else, we move to the right, until we find a non-zero value. And update our max_row_index to the current row.
- For this algorithm we get the worst case complexity as O(n+n), since we will cover max 2*n cells.

If the matrix contains all zeroes then it is considered as best case and the complexity of it is O(n).

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \rightarrow \text{Row index with maximum 1's}$$

7. (16 Points) You are given two sequences A and B of n numbers each, possibly containing duplicates. Describe an efficient algorithm for determining if A and B contain the same set of numbers, possibly in different orders. What is the running time of this algorithm?

Sol: We are given two sequences of n elements, A and B, which may contain duplicates.
We can't compare two sets of sequences if they're in random order because it's inefficient. So, first and foremost, we will sort both sequences in ascending order. To sort both sequences in O(n log(n)) time, we can use quick-sort or Merge-Sort. Now, starting from the first position in both sequences, we begin comparing DISTINCT elements in both sequences while skipping duplicates in each sequence. If we come across an element that is present in A but not in B, we can conclude that A and B contain different sets of elements.

Step-by-step algorithm:
The algorithm's runtime is O(n log(n)).
It takes O(n log(n)) time to sort the sequences. We are simply traversing the elements of the sequences of length n after sorting, which will take O(n) time.
Hence Overall running time of this method is O(n log(n)) + O(n) = O(n log(n)).

8. (16 Points) Let A and B be two sequences of n integers each, in the range [1, n4]. Given an integer x, describe an O(n)–time algorithm for determining if there is an integer a in A and an integer b in B such that x=a+b.

Sol: We are given two sequences A and B, each containing n integers in the range [1, n^4] and the integer x such that integer a exists in A and integer b exists in B and x = a + b. To determine whether the above condition is possible, we must devise an O(n) running time algorithm. The radix sort algorithm can be used to implement this. To begin, we must replace all elements in sequence A that are equal to a with (x – a). As a result of the

equation, (x-a) equals b. Radix sort is used for these elements in A with all of the elements in sequence B, with the integers in the sequences represented as 4 tuples of numbers ranging from 1 to n. Because we changed the 'a' element in A to x-a, it equals b. So, if there is a duplicate such that an element from A and an element from B are in the same tuple, the algorithm returns true because an in A and b in B exist such that x = a + b. The radix sort has a running time of O(d (n + N)), which is equivalent to O(n). After sorting, we must look for duplicates in the sorted sequence. The worst-case time complexity is O(n). As a result, the total time taken is T(n) = 2n, implying that the total running time is of the order O(n).

9. (16 Points). Suppose you are given an instance of the fractional knapsack problem in which all the items have the same weight. Describe an algorithm and provide a pseudo code for this fractional knapsack problem in O(n) time.

Sol: In one instance of the fractional knapsack problem, all objects have the same weight w.

- Assume the Knapsack size is W, which equals n*w because all objects are the same weight.
- We must choose the top $\lfloor W / w \rfloor$ items based on their benefits, followed by a subset of things with benefit rank $\lfloor W / w \rfloor$.
- The linear time selection procedure may be used to locate the item with the benefit rank of $\lfloor W / w \rfloor$ in O(n) time.
- We can traverse the components using the selection to discover things with benefits bigger than this item, and then we can determine the percentage of this item that should be considered in knapsack.
- The sorting algorithm is not used here to sort n elements. Thus, it will not take time O(n log n).
- The selection algorithm runs in O(n) time which is called recursively in the complete algorithm.
- if-else comparisons are applied, these comparisons take O(1) time.
- As a result, all of these actions require linear time, and we created an algorithm that runs in O(n) time.