Romil V. Shah

20008692

CS / CPE 600

Mid Term Exam

Submission Date: 10/23/2022

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.

Definition: Let g and f be the function from the set of natural numbers to itself. The function f is said to be θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1 \* g(n) ≤ f(n) ≤ c2 \* g(n) for all n ≥ n0.

For f(n), choose a positive integer n0, c1, c2 → 0 ≤ c1 g(n) ≤ f (n) ≤ c2 g(n)

f(n) = 2n+1, g(n) = 2n

0 ≤ c1 ∗2n ≤ 2n+1 ≤ c2 ∗2n

Choose c1 =1, c2 =3

We get 0 ≤ 2n ≤ 2 ∗ 2n ≤ 3 ∗ 2n which is true.

So, 2n+1 is θ(2n)

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.

n = 1, T(n) = 1

T(n) = T(n – 1) + n = T(n – 2) + (n – 1) + n = T(n – 2) + 2n – 1 = T(n – 3) + 3n – 3 = …….

Therefore, T(n) = T(n – k) + (n – (k – 1)) + (n – (k – 2)) + … + n

Let n - k = 1, k = n – 1

T(n) = T(1) + (n – (n – 1 – 1) + (n – (n – 1 – 2)) + …….

T(n) = 1 + 2 + 3 + …….

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

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.

Here, T(1) = 1.

The recurrence relation is T(n) = 7 T(n/5) + 10n

So,

T(5) = 7 T(5/5) + 10(5) = 7 T(1) + 50 = 57

T(25) = 7 T(25/5) + 10(25) = 7 T(5) + 250 = 649

T(125) = 7 T(125/5) + 10(125) = 7 T(25) + 1250 = 5793

T(625) = 7 T(625/5) + 10(625) = 7 T(125) + 6250 = 46801

So, 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(log n) time in worst case.

Sol.

A union-find algorithm is an algorithm that performs 2 operations on a data structure:

1. Find: Determine which subset an element is in.

This is useful to determine if 2 elements are in same subset.

1. Union: Join 2 subsets into a single subset.

In Union operation, we first need to check if the 2 subsets belong to the same set or not. If not, then, union operation cannot be performed.

In Balanced Search Tree, operation find, insert, and remove each take O(log n) time.

1. In find(parent, x) algorithm, we input parent of the Balanced Search Tree, and element x. Then, we use Balanced Search Tree find operation to find x in the structure, for which the run time is O(log n).

2. in union algorithm, we simply use update() algorithm from Balanced Search Tree and update the lement. which will take O(log n) time.

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.

Pop() all elements of S1 to S2, costs O(n) time. Thus, the conventional worst case running time of a single dequeue is O(n).

b.

Let’s assume dequeue operation costs 2 cyber dollars. First, we copy elements from S1 to S2 which costs 1$. Then, the extra 1$ can offset the cost of the copy operation.

The average of n dequeue operation cost O(1), so, the cost of a single dequeue operation is O(1).

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.

We take the following matrix as an example:

1 1 1 0 0 0 0

1 0 0 0 0 0 0

1 1 1 1 0 0 0

1 1 0 0 0 0 0

1 1 1 1 1 0 0

1 1 0 0 0 0 0

1 1 1 1 0 0 0

1. Traverse the first row of the matrix from left to right, keep increasing the count of 1 until we meet the first 0. We set the global max 1 count equal to the current count of 1.

2. Traverse next row, but this time we don’t scan from the beginning of the second row. We start scan from the column where the pervious row hit its first zero (since all the 1’s come before any 0’s in row).

3. If we have more 1’s count, we update the global count.

4. Repeat the process above, in the end return the global count.

The Time complexity is O(n + m).

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.

Algorithm:

Input: 2 sequences A and B of n elements.

Output: If A and B contain the same elements, otherwise 0.

We can solve this using a count-array:

Step 1. Create a count array of size 2n with all elements as 0.

Step 2. Traverse A and use the value of elements in A as index in count array to count the occurrences of number in A.

for i < -1 to n do

Count [A[i]] = count [A[i]] + 1

Step 3. Traverse B and use the value of element in B and index in count array, first if count is 0 then return false, else decrease count by 1

for j < -1 to n do

if count[B[j]] = 0

return false

else

count[B[j]] = count[B[j]] - 1

Step 4. If all elements in count array are zero at the end, then they have the same set, return true. otherwise return false

for k < -1 to 2n

if count[k] != 0

return false

Step 5. After the iteration is complete, it means all elements in count array are 0, then return true.

return true

This algorithm will take O(n) time, as it traverse array A, B, count array, the iteration will together take O(n + n +2n) which is O(n) time.

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 can use hash map like the count array in Q7 and do the following steps.

Step 1. Create a Hash Map H that stores all elements in A, use the value of element in A as index, set the value of element in H as 1.

for i < -1 to n do

h.put(A[i],1)

Step 2. Then, for each element b in B, if H[x – b] = 1, then it means that there are pairs a, b -> a + b = x, so, return true.

for b < -1 to do

if H[x – b] = 1

return true

Step 3. After the traverse of B, return false, since we cannot find pairs a, b -> a + b = x.

return false

This algorithm will take O(n) time, as hash map have a run time of O(1) and the worst case is going through all the elements in B, which takes O(n) time.

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.

Given the weights and values of N items, in the form of {value, weight} put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In Fractional Knapsack, we can break items for maximizing the total value of the knapsack, dort the input, which cannot be less than O(n) even in best case that too if you use bubble/insertion sort, which looks completely foolish because both sorting algorithms have O(n^2) average/worst case performance.

So, we use the weighted medians approach which will cost you O(n) as finding the weighted median will take O(n). The code for this approach is given below.

Weighted median approach for fractional knapsack:

We will work on value per unit of item in the following code.

The code will first find the middle value (mid of val per unit of items if given in sorted order) and place it in its correct position.

We will use the quicksort partition method for this.

Once we get the middle (call it mid) element, following two cases need to be taken into consideration:

1. When the sum of weight of all items present in the right side of mid is more than the value of W, we need to search our answer in the right side of mid.
2. Else, sum all the val present in the right side of mid (call it v\_left) and search for W-v\_left in the left side of mid (include mid as well).

Pseudo Code:

Algorithm partition(wght, val, start, end):

x = val[end] / wght[end]

i = start

for j in range(start,end):

if val[j]/wght[j] < x:

val[i],val[j] = val[j],val[i]

wght[i], wght[j] = wght[j],wght[i]

i+=1

val[i],val[end] = val[end],val[i]

wght[i], wght[end] = wght[end],wght[i]

return i

Algorithm helper\_find\_kth(wght, val, start, end, k):

ind = partition(wght, val, start, end)

if ind - start = = k - 1:

return ind

if ind - start > k-1:

return helper\_find\_kth(wght, val, start, ind - 1, k)

return helper\_find\_kth(wght, val, ind + 1, end, k – ind – 1)

Algorithm find\_kth(wght, val, k):

return helper\_find\_kth(wght, val, 0, len(wght) - 1, k)

Algorithm fractional\_knapsack(wght,val,w):

if w == 0 or len(wght) == 0:

return 0

if len(wght) == 1 and wght[0] > w:

return w\*(val[0]/wght[0])

mid = find\_kth(wght,val,len(wght)/2)

//get all the wght from mid-point to end of the array

w1 = reduce(lambda x,y: x+y,wght[mid+1:])

//get all the val from mid-point to end of the array

v1 = reduce(lambda x,y: x+y, val[mid+1:])

if(w1>w):

return fractional\_knapsack(wght[mid+1:],val[mid+1:],w)

return v1 + fractional\_knapsack(wght[:mid+1],val[:mid+1],w-w1)

So, the time complexity analysis will be:

T(n) = T(n/2) + O(n).

And we will get O(n) as a solution