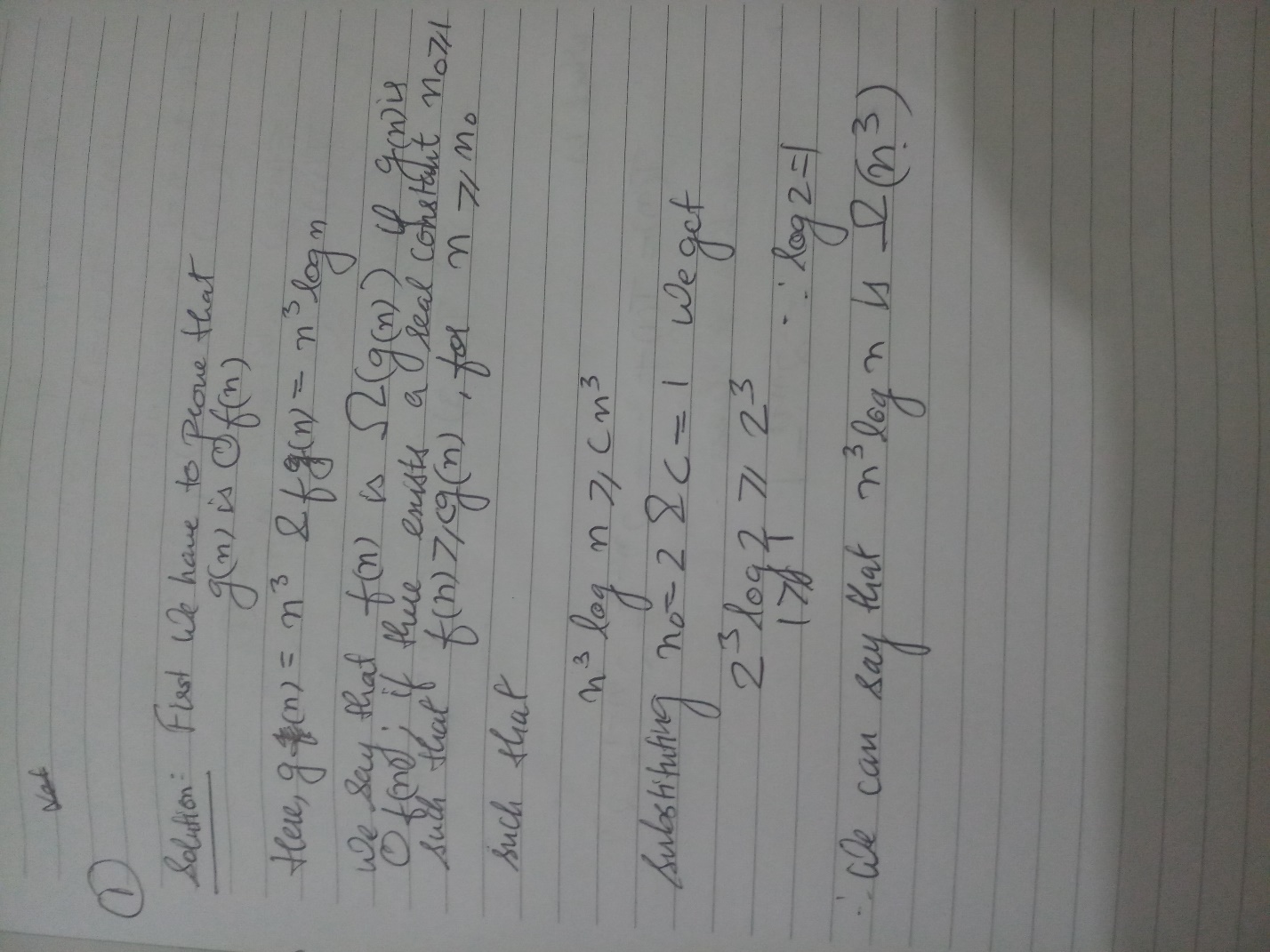
Midterm Exam Solution CS 600 Fall 2018

100 Points

Answer the following questions in this document or a word document and email it in Moodle to me and Ruiyang Sun.

1. (5 Points) Using the very definition of Big-omega notation, prove that n3 logn is Ω(n3). You must use the definition and finding the constants in the definition to receive credit.

**Solution:**



1. (5 Points) Given that T(n) = 1 if n=0 and T(n) = T(n-1)+ 2n otherwise; show, by induction, that T(n) = 2n+1 -1. Show all three steps of your induction explicitly.

**Solution:**

Step 1(Putting n=0): We can see that when we put n=0 T(n)=1 which is clearly given in the question

Step 2(Putting n=1): Now we substitute the value of n=1 and we get, T(1)= T(0)+21 which gives us T(1)=21+1=22-1.

Therefore,

T(2)=T(1)+22

T(2)=2+22+1

T(2)=2(1+2)+1=7=23-1

Similarly,

T(3)=15=24-1

Thus we proved that the rule stands for Stands for n=0 and n=1. We assume that the rule would also stand for k which is,

T(k)=2k+1-1.

Step 3(Putting n=k+1): Substituting n=k+1 we get,

T(k+1)=T(k+1-1)+2k+1 which gives us

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

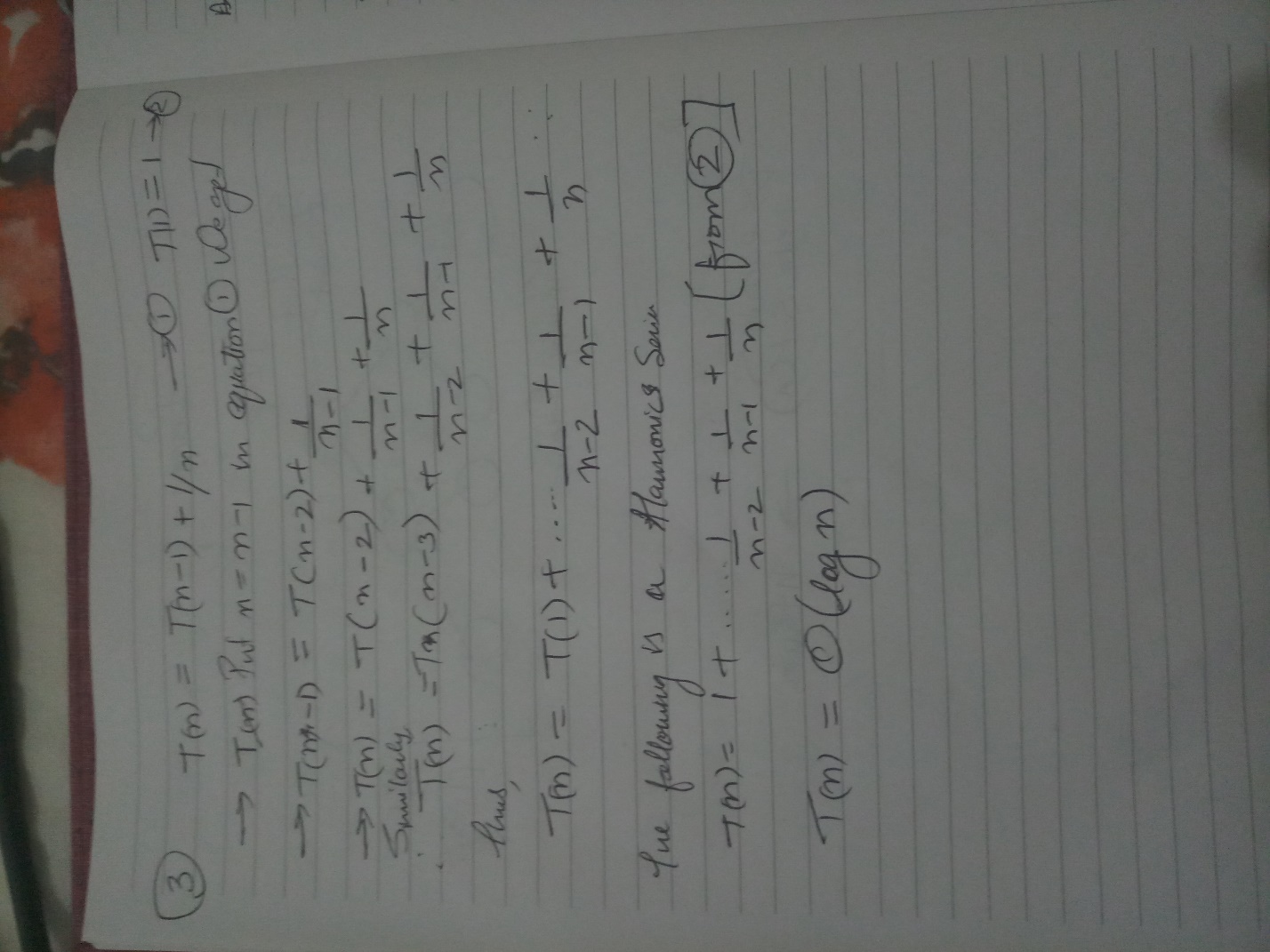
Thus it is also true for n=k+1 which finally be true for n as well which gives us

T(n)=2n+1-1.

Hence Proved,

1. (6 Points) Given T(n) = T(n-1) + 1/n, show that The Master Theorem is not applicable to this recurrence equation, hence show that T(n)=O(logn) using algebraic substitution given T(1) =1

**Solution:**



1. (14 Points) Design an O(n lg n)-time algorithm that, given an array A of n integers and another integer x, determines whether or not there exist two (not necessarily distinct) elements in A whose sum is exactly x.

**Solution:** One of the ways which we can solve the problem is sorting using merge or heap sort. We make use of two variables (say P1 and P2) to the find the element

. P1 would be at the index of the array and P2 would be at the end of the array. We can parse P1 towards P2 if the two values are small compared to the element, or vice-versa.

If P1==P2 then we return two elements in the array whose sum is “x” not found. The heap sort or merge sort algorithm has a time complexity of O (n log n). Thus the run time would be **O(n log n)**.

**Algorithm FindTHAT(X):**

**Input:** The element X.

**Ouput:** Return he two sub elements whose sum is equal to element X or return not found in the array A.

MergeSort(A) //Decreasing order

P1🡸0

P2🡸arraysize(A)-1

**while**(P1<P2) **do**

**if** (A[P1]+A[P2] == X) **do**

**return** P1 P2

**else if** (A[P1] + A[P2]< X) **do**

P1🡸P1+1

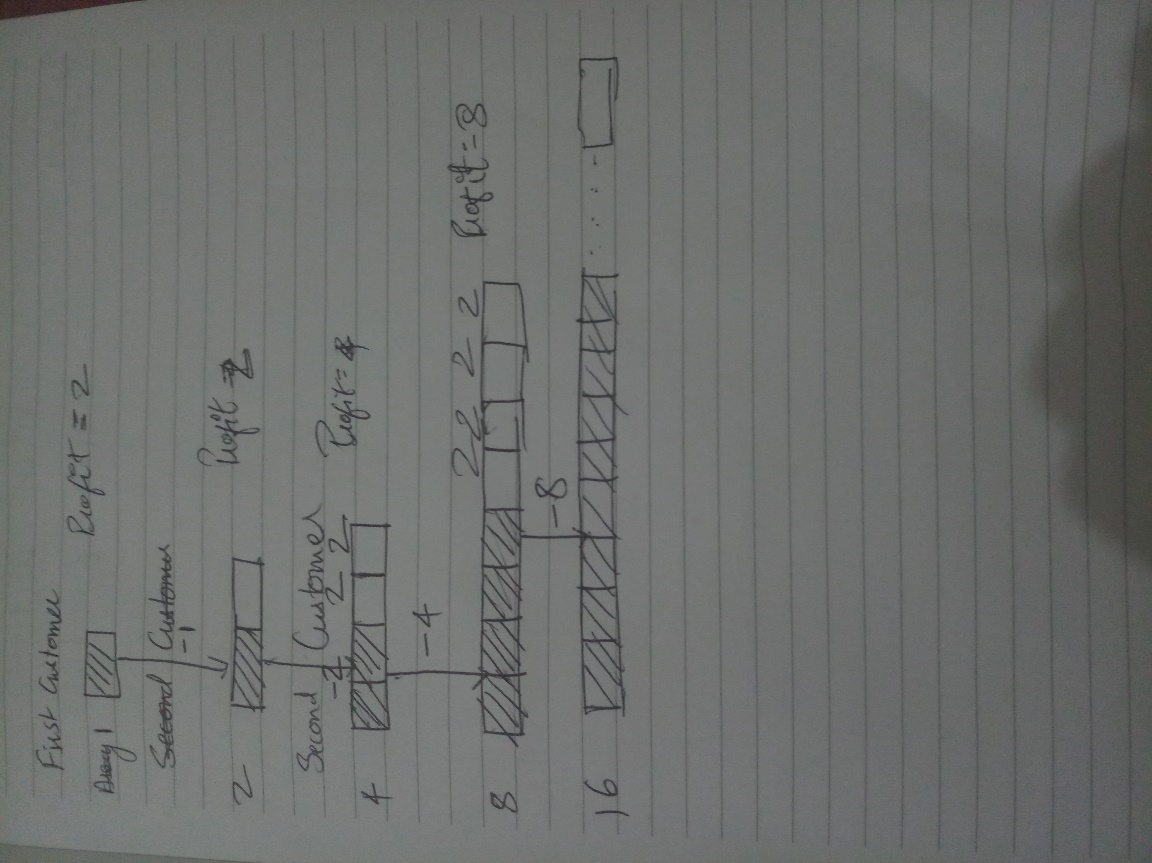
**Else do**

P2🡸P2-1

**Return** “not found in the array”

1. (14 Points) Recall the Extendable Array Implementation from Sections 1.4.1 and 1.4.2. Now cconsider an extendable table that supports both add and remove methods. Suppose we grow the underlying array implementing the table by doubling its capacity any time we need to increase the size of this array, and we shrink the underlying array by half any time the number of (actual) elements in the table dips below N/4, where N is the current capacity of the array. Use amortization method with cyber dollars to show that a sequence of n add and remove methods, starting from an array with capacity N = 1, takes O (n) time. You must use amortization Technique with cyber dollars to receive any credit.

**Solution:**  By concept of Amortization, we can solve this problem by charging 3 cyber dollars per customer for adding an element. This can be demonstrated by the following diagram. Let us assume that it takes 1 cyber dollar for the computer to perform operation for adding it extending it size. Therefore we get,­­­

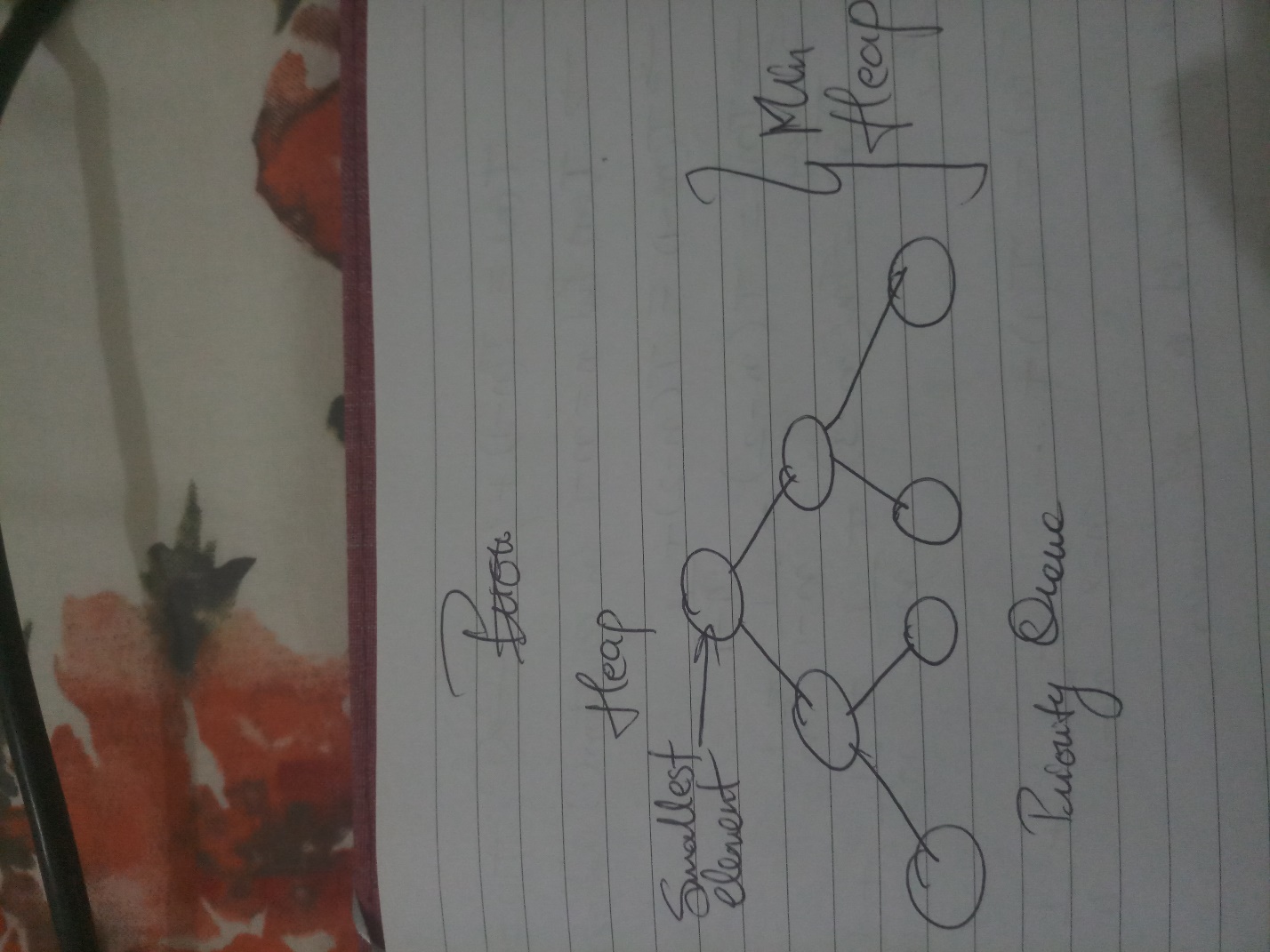


The profit in terms of cyber dollars k would be (3/4)\*N\*k >= (1/4)\*N would be done in such a way that we have to remove (3/4)\*N times to achieve (3/4)\*N\*k profits, Furthermore we have to copy the elements into the new array which takes (1/4)\*N elements while resizing. So if k=1, then and only then we have the power to spend K+1=2 cyber dollars for the whole removing method.

1. (14 Points) Develop an algorithm that computes the kth smallest element of a set of n distinct integers in O (n + k log n) time.

**Solution:** One way to solve this problem is by using a priority queue which is actually implemented by a minimum heap. Being a heap, the operation would cost O(n) time and we can pop up the smallest element is kth time to get a particular kth element. Since it is a heap it would spend log n time to adjust each and every time the smallest element popping in done. Thus the runtime for this algorithm would be

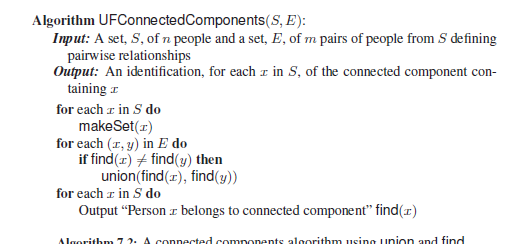
T(n)=O(n)+ O(klogn)= O(n+klogn) time. Where k is the element required to be search n is the set of integers.



1. (14 Points). Suppose a social network, N, contains n people, m edges, and c connected components. What is the exact number of times that each of the methods, makeSet, union, and find, are called in computing the connected components for N using Algorithm 7.2?

**Solution:**

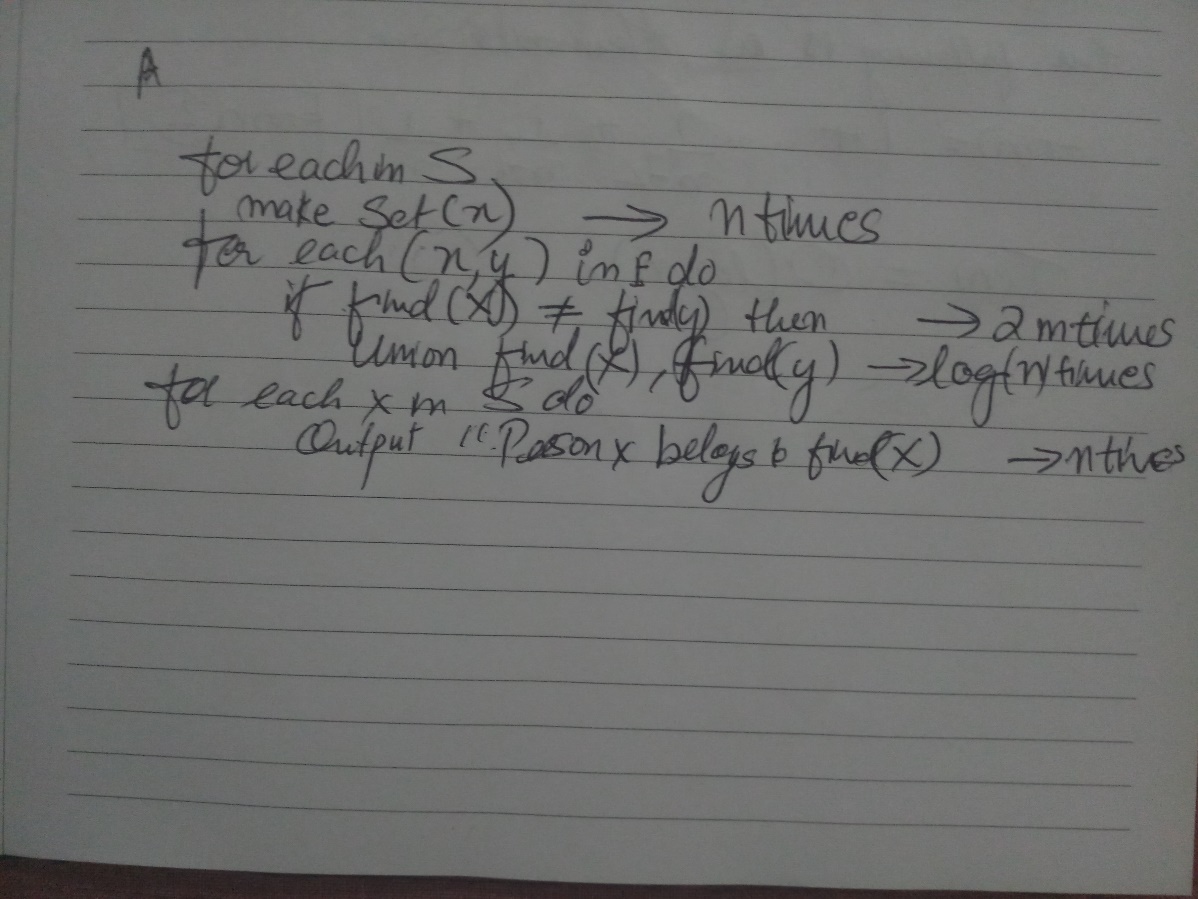
The algorithm 7.2 as given in the textbook



a. makeSet is done with **n times**.

b. union is done **log(n)** times.

c. Find consumes **2m+n** times.



1. (14 Points) Consider a single machine scheduling problem, where we are given a set, T, of tasks specified by their start times and finish times, as in the task scheduling problem, except now we have only one machine and we wish to maximize the number of tasks that this single machine performs. Design a greedy algorithm for this single machine scheduling problem.

**Solution:** Firstly, we must calculate the profit for each set of T tasks, which can be done by the following formula

Profit = Finish time – Start time

Now, we have to select the jobs which has the maximum profits and this way we will get the single machine to perform maximum tasks.

**Algorithm MaximazeMachine(M):**

**Input:** A set of T tasks that have their start time and end times.

**Output:** The maximum task that a single machine can perform within its given capacity.

For i🡨0 to T **do**

Profit[i]🡨Finishtime-Start time

//Sort profit in descending order

While(Machine!= itscapacity)

//Select the number tasks T which have maximum profits until machines capacity is filled

Return “The total tasks the machine can do”

The run time for this algorithm would depend upon which kind of sorting is done and parsing takes O (n) time. We take Merge Sort O (n log n) time. Thus the total run time would be **O (n logn)**.

1. (14 Points) Show that we can solve the telescope schedulingproblem in O(n) time even if the list of n observation requests is not given to us in sorted order, provided that start and finish times are given as integer indices in the range from 1 to n2

**Solution:**

According to the textbook, this approach can be solved using Dynamic programming by using the recursive definition



We substitute the value i=0 we get B0=0 as it is the boundary condition,

We can see from the formula that irrespective of what order we get, There is a function B pred(i) and B[i-1] which checks for the previous order. Which means that each of the n observations which we check would be current as well as previous one this guarantees about sub optimality condition. Sorted or not sorted either way the list is going to be parsed only once.Thus guaranteeing us **O(n)** runtime.