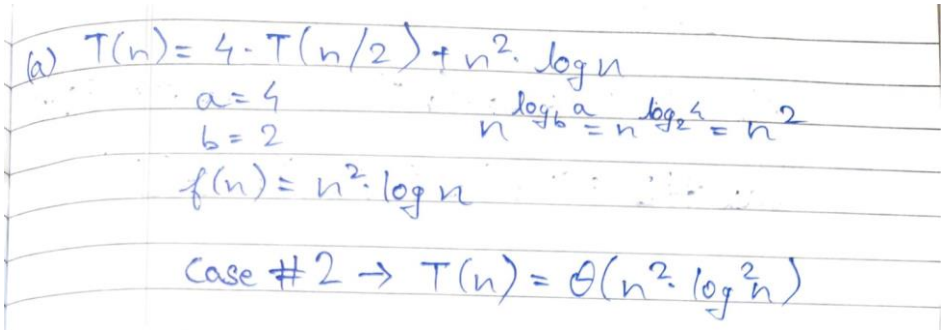
Anash Mehta
USC ID: 6498-8280-04

CSCI 570 - Fall 2022 - HW 5

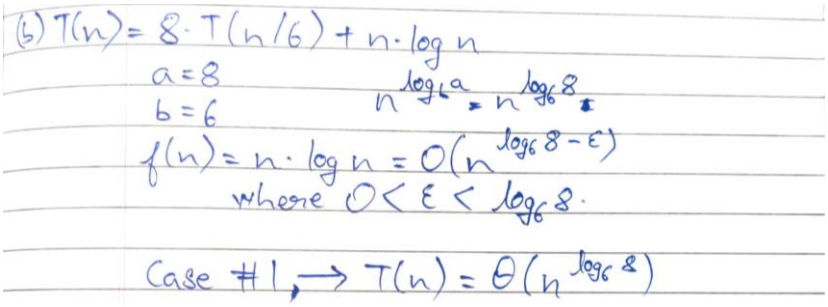
Problem 1

Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T (·) represents the running time of an algorithm, i.e. T(n) is a positive and non-decreasing function of n. For each part below, briefly describe the steps along with the final answer.

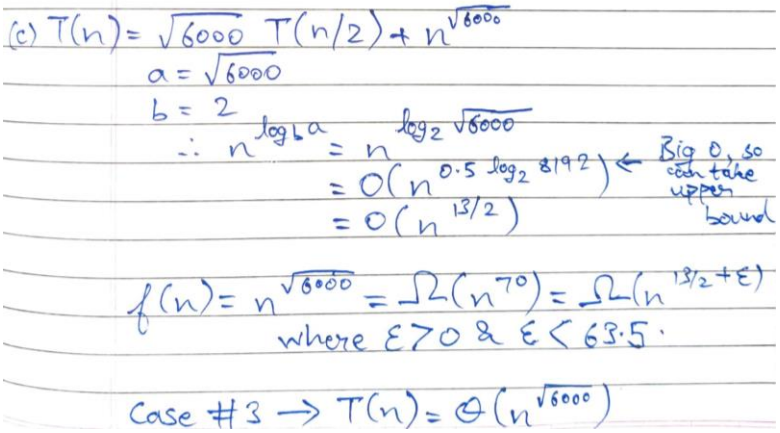
(a) $T(n) = 4T(n/2) + n^2 \log n$



(b) $T(n) = 8T(n/6) + n\log n$



(c) $T(n) = \sqrt{6000}\, T(n/2) + n^{\sqrt{6000}}$



(d) $T(n) = 10T(n/2) + 2^n$

(d) $T(n) = 10\,T(n/2) + 2^n$

$a = 10$, $b = 2$

$n^{\log_a b} = n^{\log_2 10}$

$f(n) = 2^n = \Omega(n^{\log_2 10 + \varepsilon})$ where $\varepsilon > 0$

Case #3 $\rightarrow T(n) = \Theta(f(n)) = \Theta(2^n)$

(e) $T(n) = 2T(\sqrt{n}) + \log_2 n$

(e) $T(n) = 2 \cdot T(\sqrt{n}) + \log_2 n$

Let $n = 2^m$

$\therefore T(2^m) = 2 \cdot T(2^{m/2}) + m$

Now, let $A(m) = 2 \cdot A(m/2) + m$

$a = 2$, $b = 2$

$m^{\log_b a} = m^{\log_2 2} = m^1$

$f(m) = m$

Case #2 $\rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$2^m = n$. Thus, $m = \log_2 n$

$\therefore A(m) = \Theta(m \cdot \log m)$

$= \Theta(\log n \cdot \log(\log n))$

Problem 2

Solve Kleinberg and Tardos, Chapter 5, Exercise 3.

Answer:
If there are a set of cards which belong to more than half the users, we say that the card is a majority card.

We can divide the global set of cards and then divide it into two and solve it recursively. We will find for a majority card and if it exists, return the card.

Once we solve the halves, we come at the combine step and do the following:

i. If no halves contain the majority card, then it is trivial that there is no majority card at the upper level.
ii. If both halves have the same majority card, it is obvious that the card is the majority card and return any one of the cards that is associated with the user.
iii. If both return different majority cards, we will check with all of the other cards in the whole set. This process can be done in linear time.

If T(n) denotes the number of comparisons in the equivalence tester, the algorithm will be:

$$T(n) = 2.\ T(floor(n/2)) + \Theta(n) \rightarrow T(n) = \Theta(n \log n)$$

Problem 3

Solve Kleinberg and Tardos, Chapter 5, Exercise 5.
Answer:

We will first begin by sorting the sequence of lines $L = \{L_1, L_2, \ldots, L_n\}$ in an increasing order of slope. We take the slopes because lines can extend infinitely. We will then divide the lines in half and solve it recursively. When there is one line left in the set, we return the line as visible.

We compute 2 arrays – $L_{Bslash} = \{L_{i1}, L_{i2,} \ldots, L_{im}\}$, the sorted array of visible lines in the set $\{L_1, L_2, \ldots, L_{n/2}\}$. Compute the set of points $A = \{a_1, a_2, \ldots, a_{m-1}\}$ where a_j is the intersection of L_{ik} and L_{ik+1}.

Likewise compute $L_{slash} = \{L_{j1}, L_{j2,} \ldots, L_{jr}\}$, the sorted array of visible lines in the set $\{L_{n/2+1}, \ldots, L_n\}$. Compute the set of points $B = \{b_1, b_2, \ldots, b_{r-1}\}$ where b_j is the intersection of L_{jk} and L_{jk+1}.

We can observe that arrays A and B are in increasing order of x-coordinate since if the two lines intersect, the visible part of line is to the left.

We will now go through how the two parts' solutions are merged. To obtain the list for the combined set of lines, we must merge the two recursively generated sorted lists. When viewed from above, the collection of lines that are visible essentially creates a boundary.

Finding the intersection of the two parts' boundaries is the logical next step. The boundary for the entire set, or the set of visible lines for the entire set, can then be directly determined using this. We scan the two recursively calculated sorted lists to find the first case where a line from the first half is below a line from the second half in order to find the intersection point.

More specifically, we need to merge the two sorted lists A and B. Let $L_{up}(j)$ be the uppermost line in L_{Bslash} and $L'_{up}(j)$ the uppermost line in L_{slash}. Let x be the smallest index at which L'_{up} is above L_{up}.

Let s and t be the indices such that $L_{up}(x) = L_{is}$ and define $L'_{up}(x) = L_{jt}$
Let (a, b) be the intersection of $L_{up}(x)$ and $L'_{up}(x)$. This implies that $L_{up}(x)$ is visible immediately to the left of a and $L'_{up}(x)$ to the right. Hence the sorted set of visible lines of L is $L_{i1}, L_{i2}, \ldots, L_{is-1}, L_{is}, L_{jt}, L_{jt+1}, \ldots, L_r$.

The combination step takes $\Theta(n)$ time since the step will go through each of the lines. If $T(n)$ denotes the time of running the algorithm, then:

$$T(n) <= 2 \,.\, T(floor(n/2)) + \Theta(n)$$
$$-> T(n) = \Theta(n \log n)$$

Problem 4

Assume that you have a blackbox that can multiply two integers. Describe an algorithm that when given an n-bit positive integer a and an integer x, computes x^a with at most $O(n)$ calls to the blackbox.

Answer:
If a is even:

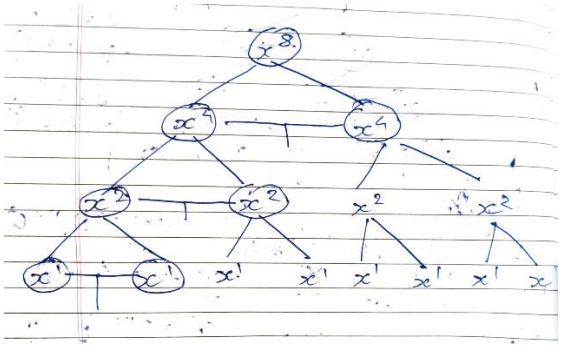
$$x^a = x^{a/2} * x^{a/2}$$

If a is odd:

$$x^a = x^{a/2} * x^{a/2} * x$$

The maximum number of calls thus is three times to the blackbox. We will not solve the right sub half of the problem since it is exactly same as the left half, so we can just store it in a variable. We assume that $a = 2^n$. Thus, $\log_2 a = n$. If a=8, then a=3.

Only the circled ones will be computed in the blackbox. For a=8, as you can see, the blackbox calculation happens 3 times i.e., $\Theta(n)$.



Problem 5

Consider two strings *a* and *b* and we are interested in a special type of similarity called the "J-similarity". Two strings a and bare considered J-similar to each other in one of the following two cases: Case 1) *a* is equal to *b*, or Case 2) If we divide *a* into two substrings a_1 and a_2 of the same length, and divide bin the same way, then one of following holds: (a) a_1 is J-similar to b_1, and a_2 is J-similar to b_2 or (b) a_2 is J-similar to b_1, and a_1 is J-similar to b_2. Caution: the second case is not applied to strings of odd length.
Prove that only strings having the same length can be J-similar to each other. Further, design an algorithm to determine if two strings are J-similar within *O*(n logn) time (where *n* is the length of strings).

Answer:
We first need to assume that if string *a* is of length *n*, string *b* also needs to be of length *n*. This is because even if we divide the strings *a* and *b* into a_1, b_1, a_2, b_2, respectively, and if length of a_1 = length of b_1 and length of a_1 = length of b_2, or length of a_1 = length of b_2 and length of a_2 = length of b_1, that means length of $a_1 + a_2$ = length of $b_1 + b_2$.

We will rearrange both of the strings and then only prove if they are J-similar to each other. For this, we must make sure that when we divide the strings, they also should be J-similar to the combined string. Thus, we design a function as follows:

```
function J-Sort(a) {
        if len(a) % 2 == 1 #odd
                return a
        a1, a2 = a[:len(a)/2], a[len(a)/2:]
        a1m = j-sort(a1)
        a2m = j-sort(a2)
        if a1m < a2m: #lexicographic ordering
                return a1m + a2m
        else
                return a2m + a1m
}
```

We observe that the output of J-sort algorithm returns the lowest lexicographic order among all of the strings. With this, we can prove that if J-sort(a) and J-sort(b) are similar, they are J-similar.

Let T(n) be the complexity.
The function is:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

This is case #2, hence according to Master's theorem, the complexity of T(n) is $O(n \log n)$.

Problem 6

Given an array of n distinct integers sorted in ascending order, we are interested in finding out if there is a Fixed Point in the array. Fixed Point in an array is an index *i* such that arr[i] is equal to *i*. Note that integers in the array can be negative Example: Input: arr[] = -10, -5, 0, 3, 7 Output: 3, since arr[3] is 3

a) Present an algorithm that returns a Fixed Point if there are any present in the array, else returns -1. Your algorithm should run in *O* (log n) in the worst case.

b) Use the Master Method to verify that your solutions to part a) runs in *O* (log n) time.

c) Let's say you have found a Fixed Point P. Provide an algorithm that determines whether P is a unique Fixed Point. Your algorithm should run in *O* (1) in the worst case

Answer:

(a) We will employ a solution similar to Binary Search because it utilizes O(log n) in the worst case. We begin by checking for the middle element and if it is a Fixed Point. If it is a fixed point, return it, else we will check if the middle element is smaller than the index of the middle value. If it is found to be true, we proceed to the right sub-half of the array n; else we check on the left half of the array n. If the array size is one and we cannot find the Fixed Point, return -1.

(b) a=2, b=1, f(n) = *O(1)* → We are just comparing it with the index element, that takes constant time.

$$n^{\log_2 1} = n^0 = O(1).$$

Follows Case #2 of Master's Theorem, hence it is *O(log n)*.

(c) If a fixed point is found at position *i*, we will check for positions *i+1* and *i-1*. If they have a Fixed point, return -1, else return 1, denoting that element at position *i* is the unique fixed point. We can have this algorithm because the array is sorted, elements are distinct, and there cannot be other fixed points. This algorithm will take *O(1).*