Homework 5

Ques 1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that $T(\cdot)$ 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) $T(n) = 10T(n/2) + 2^n$
(e) $T(n) = 2T(\sqrt{n}) + \log_2 n$

Ans. If
$T(n) = aT(n/b) + f(n)$ is satisfied with
$f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$,
then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

(a) $f(n) = n^2 \log n$ and,
$n^{\log_2 4} = n^2 = n^{\log_b a}$
Applying generalized Master's theorem;
$T(n) = \Theta(n^2 \log^2 n)$

(b) $n^{\log_b a} = n^{\log_6 8}$ and
$f(n) = n\log n = O(n^{\log_6 8 - \varepsilon})$ for any $0 < \varepsilon < \log_6 8 - 1$.
Thus, invoking master's theorem gives $T(n) = \Theta(n^{\log_6 a}) = \Theta(n^{\log_6 8})$

(c)

$n^{\log_b a} = n^{\log_2 \sqrt{6000}} = n^{0.5 \log_2 6000} = O(n^{0.5 \log_2 8192}) = O(n^{13/2})$

also, $f(n) = n^{\sqrt{6000}} = \Omega(n^{70}) = \Omega(n^{(13/2)+\varepsilon})$

for any $0 < \varepsilon < 63.5$.

Hence, $T(n) = \Theta(f(n)) = n^{\sqrt{6000}}$

(d) $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10 + \varepsilon})$ ~~for any~~ for any $\varepsilon > 0$.

$\therefore$ Master's Theorem implies that $T(n) = \Theta(f(n)) = \Theta(2^n)$.

(e) using change of variables: $n = 2^m$ to get $T(2^m) = 2T(2^{m/2}) + m$.

then, $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$.

$S(\cdot)$ is a positive function due to the monotonicity of the increasing map $x \to 2^x$ and the positivity of $T(\cdot)$. All conditions for applicability of Master's Theorem are satisfied and using the generalized version gives

$S(m) = \Theta(m \log m) = \Theta(\log_2 n \log \log_2 n)$,

for large enough n so that the expression is positive.

Quest. Solve Kleinberg and Tardos, Chapter 5, Exercise 3

Suppose you are consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account.
It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computation determines whether they are equivalent.
Their question is the following: among the collection of n cards, is there a set of more that $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Ans. Say, no. of cards $= n$

- If more than $n/2$ cards belong to a single user, we call the user a majority user.
- Divide the cards in 2 halves - $n/2$ and $n/2$.
- For each half - check if exists a majority user and if it exists, find a card corresponding to the majority user as a representative.
- After solving problem for 2 halves, combine them to solve the problem for the whole set, finding global majority user.

Majority user is present in	Half 1	Half 2	
	①	①	→ whole set does not have a majority user.
	② ✓	② ✓	→ both have same majority user, then it is global majority user.
	③ ✓	③	→ If majority users are different or if one of them has a majority user, check if any of these users is a global majority user
	④	④ ✓	

can be done linearly by comparision.

(Compare representative card,

of the majority user with every other card in the whole set, counting the number of cards that belong to the same majority user.

$T(n) \rightarrow$ no. of comparisons (invocations to the equivalence tester) of the resulting algorithm, then

$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

$$\Rightarrow T(n) = O(n \log n)$$

Ques 3. Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, . . . , well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustration a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the ith line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is uppermost at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say L_i is visible if there is some x-coordinate at which it is uppermost - Intuitively, some portion of it can be seen if you look down from "$y = \infty$." Give ~~a~~ an algorithm that takes n lines as input and in $O(n \log n)$ time return all of the one that are visible. Figure 5.10 gives an example

Ans. Let $L = \{L1, L2, .., Ln\}$ be the sequence of lines sorted in increasing order of slope. From now on, when we say sort a set of lines, it is in increasing order of slope. Divide the et of lines in half and solve recursively. When the set has only one line, return the line as visible.

Recursively compute $L_{Bslash} = \{Li, Li_2, \ldots Li_l\}$ the sorted sequence of visible lines of the $\{L1, L2, \ldots L[\frac{n}{2}]\}$. In addition we compute the set of points $A = \{a1, a2, .. am-1\}$ where aj is the intersection of Lij and $Lij+1$.

Likewise compute $L_{slash} = \{Lk1, Lk2, .. Lkr\}$, the sorted sequence of visible lines of the set $\{L[\frac{n}{2}]+1, \ldots Ln\}$. In addition compute the set of points $B = \{b1, b2, \ldots br-1\}$, when bj is the intersection of Lkj and $Lkj+1$.

By construction $\{a1, a2, \ldots, am\}$ and $\{b1, b2, \ldots, br\}$ are in increasing order of x-coordinate since if two visible lines intersect, the visible part of the line with smaller slope is to the left.

→

Now, merge the two recursively computed sorted lists to get the list for the combined set of lines.

The set of visible lines essentially forms a boundary, when seen from above. The intuition here is to find the point where the boundaries for the two halves intersect.

This can then directly be used to find the boundary for the whole set; i.e. finding the set of visible lines for the whole set.

Then parse the two recursively-computed sorted lists to locate the first instance where a line from the first half is below a line from the second half. The intersection of these lines gives us the point
Merging can be done in $O(n)$ time

We need to merge 2 sorted list A and B.
Let $L_{up}(j)$ be the uppermost line in $L_{Bslash}(j)$ and I_{up} the uppermost line in L_{slash}.
Let l be the smallest index at which I_{up} is above L_{up}.
Let s and t be indices such that $L_{up}(l) = L_{is}$ and define $I_{up}(l) = L_{jt}$.
Let (a,b) be the intersection of $L_{up}(l)$ and $I_{up}(l)$. This implies that $L_{up}(l)$ is ~~not~~ visible immediately to the left of a and $I_{up}(l)$ to the right. Hence the sorted

- set of visible lines of L is $L_{i1}, L_{i2}, \ldots L_{i2-1}, L_{is}, L_{jt}, L_{jt+1}, \ldots L_r$.
- The combination step takes $O(n)$ time. If $T(n)$ denote the running time of the algorithm, then

$$T(n) \le 2T(\lceil \tfrac{n}{2} \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

Ques. 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.

Ans. If a is odd,

$$x^a = x^{[\frac{a}{2}]} \times x^{[\frac{a}{2}]} \times x$$

a is even, $\quad x^a = x^{[\frac{a}{2}]} \times x^{[\frac{a}{2}]}$

in either way, $x^{[\frac{a}{2}]}$ ~~it~~ takes at most 3 calls to compute. Now, the problem is reduced to computing $x^{[\frac{a}{2}]}$.

If $T(n)$ is the running time,

$$T(n) \le T(n-1) + 3 \Rightarrow T(n) = O(n)$$

→

Ques5. Consider two strings a and b and we are interested in a special type of similarity called the "J-similarity". Two strings a and b are 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 b in the same way, then one of following holds:

(a) a_1 is J-similar to b_1, and ~~b_2 is~~ a_2 is J-similar to b^2 or (b) a_2 is J-similar to ~~a~~ b_1, and b_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 $O(n \log n)$ time (where n is the length of strings).

Ans. Statement : For every string a of length n and String b, b will be J-similar to a only if the length of b is equal to n.

Base case ($n=1$).
Assume we have proved for all $n<k$.
Now, let us take $n=k$.
~~if a=b~~,
There can be 2 cases that b can be J-similar to a,
1) $a=b$, they have same length.
2) if ① $len(a_1) = len(b_1)$
$len(a_2) = len(b_2)$
② $len(a_1) = len(b_2)$
$len(a_2) = len(b_1)$
In above 2 cases,
$len(a_1) + len(a_2) = len(b_1) + len(b_2)$.
$\Rightarrow$ length of b is also n.

- J-sort(a) :

```
J-sort(a) :
    if len(a) % 2 == 1:
        return a # unable to cut strings of
                   odd length.
    a1, a2 = a[: len(a/2)/2], a[len(a)/2] # Cut string to half
    a1m = J-sort(a1)
    a2m = J-sort(a2)
    if a1m < a2m : # lexicographical order
        return a1m + a2m # Concatenate
    else:
        return a2m + a1m
```

$T(n)$ = complexity of calculating J-sort (a) with respect to length (n).

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

Masters theorem,

$T(n) = O(n \log n)$.

→ checking the equivalence b/w two strings costs only $O(n)$ time.

Ques 6.

Given an array of n distinct integer 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.

Ans.

(a)

```
def Fixed_point(arr):
    array_size = n
    index_mid = n/2  if n%2==0 else  n/2+1
if  array[index_mid] == Fixed_Point:
        return  array[index_mid]
else:
    if index_mid > array[index_mid]:
        Fixed_Point(array[n/2:])
    else if  index_mid < array[index_mid]:
        Fixed_Point(array[:n/2]
    ~~else~~

if n==1 and fixed point not found:
        return -1.
```

(b) $a=1$

$b=2$

$f(n) = O(1)$.

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

↳ Case 2 : $T(n) = \Theta(\log n)$.

(c) If fixed point found at index i.

Check indices i+1 and i-1. If we don't find fixed points at these two indices, then there cannot be any other fixed points since the array is sorted and all elements are distinct.

~~...~~ For all element $j > i$,

$$arr[j] > j$$

and $j < i$,

$$arr[j] < j.$$