Homework 4

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

   In some cases, we shall need to invoke the Master Theorem with one generalization as described next. If the recurrence $T(n) = aT(n/b) + f(n)$ 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) Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying the generalized Master's theorem, $T(n) = \Theta(n^2 \log^2 n)$.

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

   (c) We have $n^{\log_b a} = n^{\log_2 \sqrt{6000}} = n^{0.5 \log_2 6000} = O(n^{0.5 \log_2 8192}) = O(n^{13/2})$. Further, $f(n) = n^{\sqrt{6000}} = \Omega(n^{70}) = \Omega(n^{(13/2)+\epsilon})$ for any $0 < \epsilon < 63.5$. Also, $a.f(n/b) \leq c.f(n)$ for $\sqrt{6000}/2^{\sqrt{6000}} < c < 1$ for all large n. Thus, from Master's theorem, $T(n) = \Theta(f(n)) = n^{\sqrt{6000}}$.

   (d) We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10 + \epsilon})$ for any $\epsilon > 0$. Also, we should have $a.f(n/b) \leq c.f(n)$. Replacing the values we have $10.2^{n/2}/2^n \leq c$. As n is large and $10.2^{n/2}/2^n$ decreases when n increases, we can say that $a.f(n/b) \leq c.f(n)$ for $10/\sqrt{2}^7 < c < 1$ for all large n. Therefore, Master's Theorem implies that $T(n) = \Theta(f(n)) = \Theta(2^n)$.

   (e) Use the change of variables: $n = 2^m$ to get $T(2^m) = 2T(2^{m/2}) + m$. Next, denoting $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$. Note that S(.) is a positive function due to the monotonicity of the increasing map $x \to 2^X$ and the positivity of T(.). All conditions for applicability of Master's Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $m^{\log_2 2} = m$. We express the solution in terms of T(n) by $T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log_2 n \log\log_2 n)$, for large enough n so that the expression is positive.

   Rubric (15 pts):

   - 3 pts: For each sub-part
     - 1 pt: For the correct answer
     - 2 pts: For the correct description

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

   In a set of cards, if more than half of the cards belong to a single user, we call the user a majority user.

   Divide the set of cards into two roughly two equal halves, (that is one half is of size $\lfloor \frac{n}{2} \rfloor$ and the other, of size $\lceil \frac{n}{2} \rceil$). For each half, recursively solve the following problem, "decide if there exists a majority user and if she exists, find a card corresponding to her (as a representative)".

   Once we have solved the problem for the two halves, we can combine them to solve the problem for the whole set, i.e. finding the global majority user. We can do that as follows.

   If neither half has a majority user, then the whole set clearly does not have a majority user.

If both the halves have the same majority user, then that user is the global majority user. We can pick either one of the output cards returned by the halves as a representative for the whole set.

If the majority users are different, or if only one of them has a majority user, we need to check if any of these users is a global majority user. We can do this in a linear manner by comparing the 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.

If $T(n)$ denotes the number of comparisons (invocations to the equivalence tester) of the resulting divide and conquer algorithm, then

$$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n \log n)$$

Rubric (10 pts):

- 7 pts: For describing the algorithm clearly
- 3 pts: For recurrence relation and time complexity computation

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

We first sort the lines in order of increasing slopes, and then use a divide-and-conquer approach by dividing the sequence of lines in half and solving recursively. The base case of divide-and-conquer happens if n <= 3, for which we can easily find the visible lines in constant time. (The first and third lines will always be visible; the second will be visible if and only if it meets the first line to the left of where the third line meets the first line.)

Now let's recursively compute the sequence of visible lines for the first half L_1, ..., L_m (m=ceiling(n/2)) - say they are L = $\{L_{i1}, ..., L_{ip}\}$ in order of increasing slope. We also compute the sequence of intersection points a_1, ... a_{p-1} where a_k is the intersection of line L_{ik} with L_{ik+1}. The intersection points a_1,..., a_p have increasing x-coordinates; since if two lines are both visible, the part of the line with the smaller slope that is uppermost lies to the left of the line with the larger slope that is uppermost.
Similarly, we repeat the same process for the other half L_{m+1}, ..., L_n and compute the sequence of visible lines L' = $\{L_{j1}, ..., L_{jq}\}$ in ascending order of slopes, along with the sequence of intersection points b_1, ..., b_{q-1} where b_k is the intersection of L_{jk} and L_{jk+1}).

Now we need to show how to determine the visible lines from the combined set using L and L' and their corresponding intersection points in O(n). Note that at every point in the recursion p+q <= n, so it is enough to run it in O(p+q).

We know that L_{i1} and L_{jq} will be visible because they have the minimum and maximum slopes respectively among the combined set of lines. We merge the sorted lists $a_1,...,a_{p-1}$ and $b_1,...b_{q-1}$ into a single list $c_1,c_2,...,c_{p+q-2}$, ordered by increasing x-coordinates. This takes O(n) time. Now for each k, we consider the lines that are uppermost in L and L' at x-coordinate c_k. Let z be the smallest index for which the uppermost line in L' lies above the uppermost line in L at x-coordinate c_z. Let's call the two lines at this point L_{is} and L_{jt}, which belong to L and L' respectively. Let (x*,y*) be the interstection point of these two lines. We can therefore say that x* lies between the x-coordinates of c_{z-1} and c_z . This means that in the combined set, L_{is} is uppermost immediately to the left of x*, and L_{jt} is uppermost immediately to the right of x*. Therefore the sequence of visible lines will be L_{i1}, ..., L_{is}, L_{jt}, ...L_{jq}; and the sequence of intersection points is a_{i1}, ..., a_{is-1}, (x*,y*), b_{jt}, ..., b_{jq-1}.
Since this is what we need to return to the next level of the recursion, the algorithm is complete.

The combination step takes $\mathcal{O}(n)$ time. If $T(n)$ denotes the running time of the algorithm, then

$$T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}(n \log(n))$$

Rubric (10 pts):

- 7 pts: For describing the algorithm clearly
- 3 pts: For recurrence relation and time complexity computation

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 $\mathcal{O}(n)$ calls to the blackbox.

   If a is odd,

   $$x^a = x^{\lfloor \frac{a}{2} \rfloor} \times x^{\lfloor \frac{a}{2} \rfloor} \times x$$

   If a is even,

   $$x^a = x^{\lfloor \frac{a}{2} \rfloor} \times x^{\lfloor \frac{a}{2} \rfloor}$$

   In either case, given $x^{\lfloor \frac{a}{2} \rfloor}$ it takes at most three calls to the black-box to compute x^a. We have thus reduced the problem of computing x^a to computing $x^{\lfloor \frac{a}{2} \rfloor}$ (which is an identical problem with input of size one bit smaller). Let $T(n)$ denote the running time of the corresponding divide-conquer algorithm. Thus

   $$T(n) \leq T(n-1) + 3 \Rightarrow T(n) = \mathcal{O}(n)$$

   Remark: Such computations arise frequently in fields like cryptography. For instance, during the encryption of messages in the RSA cryptosystem, one has to make such a computation (m^e modulo a fixed number) where the exponent e is around 1024 bits. With the above algorithm, the number of multiplication calls to the blackbox is around 1024. If a naive method is used, it could take 2^{1024} multiplications which would be highly undesirable as this number far exceeds the number of atoms in the known universe!

   Rubric (10 pts):

   - 7 pts: For describing the algorithm clearly
   - 3 pts: For recurrence relation and time complexity computation

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 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 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).

   To make our proof straightforward, we first change the statement to another equivalent: 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. We use induction on n to prove the statement. The base case (n = 1) is trivial. Assume we now have proved the statement for all n < k. Now we focus on n = k. Regarding to the definition, there are two cases that b can be J-similar to a. Firstly, if a is equal to b, then apparently they have the same length. On the other hand, if the second case happens, by the induction either one of the following will be correct: len(a_1) = len(b_1), len(a_2) = len(b_2) or len(a_1) = len(b_2), len(a_2) = len(b_1) In both case we have len(a_1) + len(a_2) = len(b_1) + len(b_2), which means the length of b is also n.

   The algorithm: we rearrange both of the two strings by some certain movements, then we prove they will project to the same string if and only if they are J-similar to each other. How to design this movement? To begin with, it's obvious that for any string a of even length, once we split it into two halves (a_1,a_2), $a' = a_2a_1$ should be also J-similar to a. Based on this, we can design a divide and conquer approach to rearrange a, b into their lexicographically minimum equivalents. Specifically, we design a function J-sort(a) as follows:

   function J-sort(a) {
   if len(a) % 2 == 1:
   return a # Unable to cut strings of odd length
   a_1, a_2 = a[:len(a)/2], a[len(a)/2:] # Cut string to half
   a_{1m} = J-sort(a_1)

a_{2m} = J-sort(a_2)

if $a_{1m} < a_{2m}$: # lexicographical order

return $a_{1m} + a_{2m}$ # concatenate

else:

return $a_{2m} + a_{1m}$

}

It's not hard to conclude that J-sort(a) has the lowest lexicographical order among all the strings that is J-similar to a (Just use induction like above). Because of this (and the same thing to b), we can further prove that a is J- similar to b if and only if J-sort(a) is equal to J-sort(b). Let T (n) be the complexity of calculating J-sort(a) with respect to length n. Based on the previous analysis, we have T (n) = 2 · T (n/2) + O(n). The Master Theorem tells us T(n) = O(nlogn). Aside from that, checking the equivalence between two strings costs only O(n) time, which is negligible.

Rubric (15 pt):

- 5 points for proving the statement correctly
- 7 points for designing the Divide and Conquer Algorithm
- 3 points for computing the Time Complexity correctly.

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.

a) First check whether the middle element is a Fixed Point or not. If it is, then return it; otherwise check whether the index of the middle element is greater than the value at the index. If the index is greater, then Fixed Point(s) lies on the right side of the middle point (obviously only if there is a Fixed Point). Else the Fixed Point(s) lies on the left side. If subproblem size equals one and Fixed is not found, return -1.

b) a=1, b=2, f(n) = O(1). $n^{log_b a} = n^{log_2 1} = n^0 = O(1) \implies$ Case 2: T(n) = Θ(log n)

c) Say a fixed point is 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 integers. Hence, for all other elements j above i, we must have arr[j] > j. And for all elements j below i, we must have arr[j] < j.

Rubric (12 pts):

- -3 if the incorrect half of array is pruned
- -1 if Master Theorem case is incorrect
- -1 if n**(log a base b) is incorrectly computed

7. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a **Dynamic Programming** algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and recursively compute the maximum points we can get for the other piece. It is also possible to recursively attempt to obtain the maximum value for both pieces after the cut, but that requires adding an extra check to see if the value obtained by selling a rod of this length is more than recursively cutting it up.

The bottom-up pseudo-code to obtain the maximum amount of money is:

function Bottom-Up-Cut-Rod(p, n) {

Let $r[0, ..., n]$ be a new array

$r[0] = 0$

for $j = 1$ to n
$q = -\infty$
for $i = 1$ to j
$q = max(q, p[i] + r[j - i])$
end for
$r[j] = q$
end for
return r[n]
}

The time complexity of this algorithm is $\theta(n^2)$ because of the double nested for loop.

Rubric (10 pts):

- 5 points for a correct dynamic programming solution
- 3 points if the solution runs in $\theta(n^2)$
- 2 points for providing analysis of runtime complexity

8. From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have infinitely many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size w_i and value v_i. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

Similar to what is taught in the lecture, let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$. We find the recurrence relation of $OPT(k, w)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

- We include another item of type k and solve the sub-problem $OPT(k, w - w_k)$.
- We do not include any item of type k and move to consider next type of item this solving the sub-problem $OPT(k - 1, w)$.

Therefore, we have

$$OPT(k, w) = \max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}.$$

Moreover, we have the initial condition $OPT(0, 0) = 0$.

Pseudocode: {
for $k = 1$ to n
for $w = 0$ to W
if $w \geq w_k$
$OPT(k, w) = \max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}$
else
$OPT(k, w) = OPT(k - 1, w)$
end if
end for
end for
return $OPT(n, W)$
}

Another Solution: https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed/

Rubric (10 pts):

- 5 points for a correct dynamic programming solution
- 3 points if the solution runs in $\theta(nW)$
- 2 points for providing analysis of runtime complexity