HW4

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) $\mathrm{T(n)} = 4T(n/2) + n^2 logn$

(b) $\mathrm{T(n)} = 8\mathrm{T(n/6)} + \mathrm{nlogn}$

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

(d) $\mathrm{T(n)} = 10\mathrm{T(n/2)} + 2^n$

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

Solution:

a) Observe that f(n) = n2log n and $n$log$b$ a = $n$log2 4 = n2, so applying the generalized Master's theorem, T (n) = Θ(n2 log 2n).

b) Observe that $n$log$b$ a = $n$log6 8 and f(n) = nlogn = O($n$log6 8−ε) for any 0 < ε < log6 8 − 1. Thus, invoking Master's Theorem gives T(n) = Θ($n$log$b$ a) = Θ($n$log6 8).

c) We have $n$log$b$ a = $n$log2 √6000 = n0.5 log2 6006 = O(n0.5 log2 8192) = n13/2 and 2 f(n) = n√6006 = Ω(n70) = Ω(n13 + ε) for any 0 < ε < 63.5. Thus, from Master's Theorem T(n) = Θ(f(n)) = Θ(n√6006).

d) We have $n$log$b$ a = $n$log2 10 and f(n) = 2n = Ω($n$log2 10+ ε) for any ε > 0.
Therefore, Master's Theorem implies T(n) = Θ(f(n)) = Θ(2n).

e) Use the change of variables n = 2m to get T(2m) = 2T(2m/2) + m. Next, denoting S(m) = T (2m) 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 ⟼ 2x and the positivity of T(·). All conditions for applicability of Master's Theorem are satisfied and using the generalized version gives S(m) = Θ(m log m) on observing that f(m) = m and $m$log$b$ a = m. We express the solution in terms of T(n) by T(n) = T(2m) = S(m) = Θ(m log$m$) = Θ(log$n$loglog$n$), with log having base 2 for large enough n so that the growth expression above is positive.

Problem 2

Suppose you're 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 computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than 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.

Solution:

We give two solutions for this problem. The first solution is a divide and conquer algorithm, which is easier to think of. The second solution is a clever linear time algorithm.

Via divide and conquer: Let e1,…, en denote the equivalence classes of the cards: cards i and j are equivalent if ei = ej. What we are looking for is a value x so that more than n/2 of the indices have ei = X.

Divide the set of cards into two roughly equal piles: a set of n/2 cards and a second set for the remaining n/2 cards. We will recursively run the algorithm on the two sides, and will assume that if the algorithm finds an equivalence class containing more than half of the cards, then it returns a sample card in the equivalence class.

Note that if there are more than n/2 cards that are equivalent in the whole set, say have equivalence class , than at least one of the two sides will have more than half the cards also equivalent to x. So at least one of the two recursive calls will return a card that has equivalence class x.

The reverse of this statement is not true: there can be a majority of equivalent cards in one side, without that equivalence class having more than n/2 cards overall (as it was only a majority on one side. So it a majority card is returned on either side we must test this card against all other cards.

If S =1 return the one card
If S =2

test if the two cards are equivalent
return either card if they are equivalent
Let S1 be the set of the first n/2, cards
Let S2 be the set of the remaining cards
Call the algorithm recursively for S1.
If a card is returned
then test this against all other cards
If no card with majority equivalence has yet been found
then call the algorithm recursively for S,.
If a card is returned
then test this against all other cards
Return a card from the majority equivalence class if one is found

The correctness of the algorithm follows from the observation above: that if there is a majority equivalence class, than this must be a majority equivalence class for at least one of the two sides.

To analyze the running time, let I(n) denote the maximum number of tests the algorithm does for any set of n cards. The algorithm has two recursive calls, and does at most 2n tests outside of the recursive calls. So we get the following recurrence (assuming n is divisible by 2): $T(n) < 2T(n/2) + 2n$.

Problem 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 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 illustrate a basic speedup that can be achieved. You are given n non vertical lines in the plane, labeled L1, . . . , Ln, with the i'th line specified by the equation y = aix + bi. We will make the assumption that no three of the lines all meet at a single point. We say line Li is uppermost at a given x-coordinate x0 if its y-coordinate at x0 is greater than the y-coordinates of all the other lines at x0:aix0 + bi>ajx0 + bj for all j=i. We say line Li 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 = ∞."

Give an algorithm that takes n lines as input and in O(n log n) time returns all of the ones that are visible. Figure 5.10 gives an example.

Solution:

The problem you've described is a classic computational geometry problem and is usually

solved using a sweep line algorithm. This technique involves sorting the line segments and "sweeping" across them from left to right, processing each segment as it's encountered.

Here is a high-level description of how this algorithm works:

1. First, for each line `Li = aix + bi`, find the two x-coordinates where it intersects with its adjacent lines in sorted order of slopes. We'll call these intersection points `left` and `right`. Note that the `left` of `Li` is the intersection with `Li-1` (the line with the next lower slope) and `right` is the intersection with `Li+1` (the line with the next higher slope).

2. Create an event for each intersection point and sort these events. There will be 2n events in total. Each event consists of an x-coordinate and a line. There are two types of events: start events (when we first encounter a line) and end events (when a line is no longer visible).

3. Initialize an empty binary search tree (BST) data structure, where each node corresponds to a line segment, and the BST is ordered by the y-coordinate of the lines at the current sweep line location.

4. Now start the "sweep" by processing the events from left to right:

- If it's a start event for line `Li`, insert `Li` into the BST. After the insertion, check if `Li` is covered by its neighboring line in the BST. If it is, then it will never be visible and we can remove it. If `Li` covers its neighbors, remove those lines.

- If it's an end event, that means we are done processing line `Li`. Remove `Li` from the BST. If `Li` was covering any lines, those lines may now become visible.

5. The lines remaining in the BST are the visible lines.

This algorithm works in O(n log n) time. Sorting the events takes O(n log n) time and we handle each event in O(log n) time (for the BST operations), so the total time complexity is O(n log n).

A precise implementation of this algorithm requires careful handling of the edge cases and numerical precision issues. Moreover, you need a data structure that allows for efficient query and removal of elements, so using a balanced BST or a library that offers log-time insertion, deletion, and retrieval is a good idea.

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 xa with at most O(n) calls to the blackbox.

Solution:
This problem is a classic example of exponentiation by squaring, which can be done using a simple loop or recursion. The idea here is to use the binary representation of the exponent 'a' to calculate 'x^a' in a fast way. Here's how to do it iteratively:

1. Initialize the result as 1.
2. While 'a' is greater than 0:
   - If the least significant bit of 'a' is set (i.e., 'a' is odd), multiply the result with 'x' using the black box.
   - Square 'x' using the black box and shift 'a' one bit to the right (equivalent to integer division by 2).
3. Return the result.

Let's analyze the time complexity. Each iteration involves at most 2 calls to the blackbox (one for multiplying the result with 'x' and one for squaring 'x') and one right shift operation, which are all constant time operations. Since 'a' is an n-bit number and we're right shifting 'a' at each step, there will be at most n iterations. Thus, the total number of calls to the blackbox is at most 2n, which is O(n) as required.

<u>Problem 5</u>
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 a1 and a2 of the same length, and divide b in the same way, then one of following holds: (a) a1 is J-similar to b1, and a2 is J-similar to b2 or (b) a2 is J-similar to b1, and a1 is J-similar to b2. 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).

Solution: Let's take two strings a and b of different lengths. Suppose a has an even length and b has a odd length. In case 1 a will never be equal to b since they both have different lengths and In case 2 we cannot divide b into substrings of equal lengths since b has an odd length. Thus a and b are not J-similar and therefore only strings having the same length can be J-similar to each other.

We can solve this problem with a divide and conquer approach. Here's the recursive solution to it:

```python
def Jsim(a, b):
        n = len(a)
        if n != len(b): return False
```

```
    if a == b: return True
    else :
        return (Jsim(a[:n/2], b[:n/2]) AND (Jsim(a[n/2:], b[n/2:]) OR (Jsim(a[:n/2], b[n/2:]) AND (Jsim(a[n/2:], b[:n/2])
```

For complexity analysis, each divide step cuts the string size in half, hence we go log(n) levels deep in the recursion tree, and at each level, we have 4 recursive calls, leading to a time complexity of O(n log n). The work done at each node (string comparison and slicing) is proportional to the length of the string, hence overall time complexity remains 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.*

Solution:
(a) Using the Divide and Conquer technique here we can divide the problem set into two problem sets at n/2 index and check if the value at n/2 index in the original array is lesser than or greater than or equal to it's index in the full array. If it's lesser than that we recursively traverse down the right half of the original array and if it's greater then we traverse down the left half of the original array. If it's equal then we return True. We can keep track of the original index while recursively passing down the relative index value.

This solution would take O (log n) time since we are dividing the problem into two halves at each step and every other computation at each step takes constant time.

(b) The divide step = T(d) = O(1)
Combine step = T(c) = O(1)

$n^{log_b a} = 1$ where a = 1 and b = 2

Therefore according to Master Method, this falls under case 2 where f(n) = $\Theta(n^{log_b a})$ and thus T(n) = Θ(log n)

(c) We can find out if our Fixed point is a unique fixed point or not by doing the following two computations:
- If the value at index 0 is not 0, then there are no fixed points to the left of P.
- If the value at index n-1 is less than n-1, then there are no fixed points to the right of P.

These two computations will run in O(1) time in the worst case.

Problem 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 pi 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.

Solution:

Sure, let's devise a dynamic programming (DP) solution for this problem. It's essentially the Rod Cutting problem, a classic in the study of dynamic programming. Let's assume that the rod has an integer length of N units, and pi is the price you get for a rod of length i where 1 <= i <= N.

1. Initialization:
Create an array `dp` of size `N+1`. Each `dp[i]` represents the maximum value we can get for a rod of length `i`. Initialize `dp[0]` to 0 since a rod of length 0 has no value.

2. State Transition:
The main idea behind the dynamic programming solution is that we can cut the rod at various positions and compare the prices we get after each cut to find the maximum. For a rod of length `i`, we can cut it in `i` ways, where the cut is at position `j` (where `1 <= j <= i`). For each cut, we have two parts of lengths `j` and `i-j`. The price for a cut at position `j` can be either `p[j] + dp[i-j]` if we choose to cut the second part further, or `p[i]` if we don't cut the rod at all.

So `dp[i] = max(dp[i], max(p[j] + dp[i-j], p[i])) for each 1 <= j < i`.

3. Algorithm:
Here's a Python-style pseudocode representing the above idea:

```python
def rod_cutting(p, N):
    dp = [0] * (N + 1)  # Initialize dp array
    for i in range(1, N + 1):  # iterate for each length of rod
        for j in range(1, i + 1):  # iterate for each cut position
            dp[i] = max(dp[i], p[j] + dp[i - j])  # maximize the price
    return dp[N]  # maximum price for rod of length N
```

Please note that the price array `p` is 1-indexed for this pseudocode.
Also, the time complexity of this algorithm is `O(N^2)` due to the two nested loops. The space complexity is `O(N)` for the dp array.

Problem 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 wi and value vi. 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.

Solution:
In the knapsack problem with infinite supply of each type of item (also known as the unbounded knapsack problem), you can pick an item multiple times if the knapsack capacity allows.

This problem can also be solved using dynamic programming, where the main difference from the 0-1 knapsack problem is that you do not move on to the next item after considering the current item in the dynamic programming array, but stay on the current item and continue considering it until you can no longer put it in the knapsack.

Here's a Python-style pseudocode for the unbounded knapsack problem:

```python
def unbounded_knapsack(n, W, sizes, values):
    dp = [0] * (W + 1)  # Initialize dp array

    for w in range(1, W + 1):  # For every capacity
        for i in range(n):  # For every item
            if sizes[i] <= w:  # If the item can be put in the knapsack
                dp[w] = max(dp[w], dp[w - sizes[i]] + values[i])
    return dp[W]  # Maximum value for capacity W
```

Here, `n` is the number of types of items, `W` is the total capacity of the knapsack, `sizes` is an array containing the size of each item, and `values` is an array containing the value of each item.

The time complexity of this algorithm is O(nW) as there are two nested loops: one iterating over all items, and the other iterating over all capacities up to W. The space complexity is O(W) for the dp array.