CSCI 570 - Analysis of Algorithms

Assignment 3

Harshitha Kurra

---

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 positive and non-decreasing function of n and for small constants c independent of n, T(c) is also a constant independent of n. Note that some of these recurrences might be a little challenging to think about at first. Each question has 4 points. For each question, you need to explain how the Master Theorem is applied (2 points) and state your answer (2 points).**

   A. $T(n) = 4T(n/2) + n^2 log n$

   B. $T(n) = 8T(n/6) + n log n$

   C. $T(n) = \sqrt{6006}T(n/2) + n^{\sqrt{6006}}$

   D. $T(n) = 10T(n/2) + 2^n$

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

   F. $T(n) = T(n/2) - n + 10$

   G. $T(n) = 2^n T(n/2) + n$

   H. $T(n) = 2T(n/4) + n^{0.51}$

   I. $T(n) = 0.5T(n/2) + 1/n$

   J. $T(n) = 16T(n/4) + n!$

Solution:

A. $T(n) = 4T(n/2) + n^2 log n$

here, c = $log_b a$ = $log_2 4$ = 2

and $f(n) = n^2 log n$ which is equivalent to case 2: $n^c log^k n$ so we apply case 2 here.

Therefore, $T(n) = \Theta(n^2 log^2 n)$

B. $T(n) = 8T(n/6) + nlogn$

here c = $log_b a = log_6 8$ = 1.16 which is greater than f(n) so we apply case 1.

Therefore, $T(n) = \Theta(n^c)$ where c = $log_6 8$ = 1.16

$$T(n) = \Theta(n^{1.16})$$

C. $T(n) = \sqrt{6006}T(n/2) + n^{\sqrt{6006}}$

here c = $log_b a = log_2 \sqrt{6006}$ and $n^{log_2 \sqrt{6006}}$ < f(n) i.e; $n^{\sqrt{6006}}$ so we apply case 3.

Therefore, $T(n) = \Theta(n^{\sqrt{6006}})$

D. $T(n) = 10T(n/2) + 2^n$

here c = $log_b a = log_2 10$. $n^{log_2 10}$ is less than f(n) i.e; 2^n so we apply case-3.

Therefore, $T(n) = \Theta(2^n)$

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

here, let $m = logn$

$n = 2^m$

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

Let $S(m) = T(2^m)$

so, $S(m) = 2S(m/2) + m$ which is now in the format that masters theorem can be applied.

c = $log_b a$ = $log_2 2$ = 1. So we have n^c = f(n) here i.e; $m^1 = m$. We apply case 2

Therefore, $S(m) = \Theta(mlogm)$. Converting $S(m)$ to $T(n)$ by replacing m with logn. So, $T(n) = \Theta(logn(log(logn))$

F. $T(n) = T(n/2) - n + 10$

writing it in masters theorem format, $T(n) = T(n/2) + (10 - n)$. Here, our function $f(n) = 10 - n$ is monotonically decreasing. So masters theorem can't be applied.

G. $T(n) = 2^n T(n/2) + n$

According to the masters theorem, both a, b should be constants. Here, a = 2^n. Which is not constant. So, masters theorem can't be applied.

H. $T(n) = 2T(n/4) + n^{0.51}$

c = $log_b a$ = $log_4 2$ = 1/2 = 0.50. here n^c < f(n) i.e; $n^{0.50} < n^{0.51}$ so we apply case-3

Therefore, $T(n) = \Theta(n^{0.51})$

I. $T(n) = 0.5T(n/2) + 1/n$

Masters Theorem can't be applied here as our a I.e; 0.5 is less than 1. According to the theorem, both a, b should be greater than 1.
So, no solution.

J. $T(n) = 16T(n/4) + n!$

c = $log_b a$ = $log_4 16$ = 2. Here, n^c < f(n) i.e; $n^2 < n!$ so we apply case-3

Therefore, $T(n) = \Theta(n!)$

---

2. Consider an array A of n numbers with the assurance that n > 2, $A_1 \geq A_2$ and $A_n \geq A_{n-1}$. An index i is said to be a local minimum of the array A if it satisfies $1<i<n$, $A_{i-1} \geq A_i$ and $A_{i+1} \geq A_i$.

A. Prove that there always exists a local minimum for A.

B. Design an algorithm to compute a local minimum of A.
Your algorithm is allowed to make at most O(logn) pairwise comparisons between elements of A.

Solution:

We prove the existence of a local minimum by induction. For n = 3, we have $A[1] \geq A[2]$ and $A[3] \geq A[2]$ by the premise of the question and therefore A[2] is a local minimum. Let A[1 : n] admit a local minimum for n = k. We shall show that A[1 : k + 1] also admits a local minimum. If we assume that $A[3] \geq A[2]$ then A[2] is a local minimum for A[1 : k + 1] since $A[1] \geq A[2]$ by premise of the question. So let us assume $A[2] > A[3]$. In this case, the k length array $A[2 : k + 1] = A'[1 : k]$ satisfies the induction hypothesis ($A'[1] = A[2] \geq A[3] = A'[2]$ and $A'[k - 1] = A[k] \leq A[k + 1] = A'[k]$) and hence admits a local minimum. This is also a local minimum for A[1 : k + 1]. Hence, proof by induction is complete.

Algorithm:

Consider the following algorithm with array A of length n as the input, and return value as a local minimum element.

(a) If n = 3, return A[2].

(b) (b) If n>3,

i. k ← ⌊n/2⌋
ii. If $A[k] \leq A[k-1]$ and $A[k] \leq A[k+1]$ then returnA[k]

iii. If A[k] > A[k – 1] then call the algorithm recursively on A[1 : k], else call the algorithm on A[k : n].

Complexity: If the running time of the algorithm on an input of size n is T(n), then it involves a constant number of comparisons and assignments and a recursive function call on either A[1 : ⌊n/2⌋] (size = ⌊n/2⌋) or A[⌊n/2⌋ : n] (size = n – ⌊n/2⌋ = ⌈n/2⌉). Therefore, T (n) ≤ T (⌈n/2⌉) + Θ(1).

Invoking Masters Theorem gives $T(n) = O(logn)$

Proof of Correctness: We employ induction. For n = 3 it is clear that step (a) returns a local minimum using the premise of the question that $A[1] \geq A[2]$ and $A[3] \geq A[2]$. Let us assume that the algorithm correctly finds a local minimum for all n ≤ m and consider an input of size m + 1. Then k = ⌊(m + 1)/2⌋.

If step (b)(ii) returns, then a local minimum is found by definition and the algorithm gives the correct output. Otherwise, step (b)(iii) is executed since one of $A[k] > A[k-1]$ or $A[k] > A[k+1]$ must be true for step (b)(ii) to not return.

If $A[k] > A[k-1]$, then A[1 : k] must admit a local minimum by the first part of the question and the given algorithm can find it correctly if k ≤ m, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if ⌊(m + 1)/2⌋ ≤ m which is true for all m ≥ 1.

Similarly, if $A[k] > A[k+1]$, then A[k : m+1] must admit a local minimum by the first part of the question and the given algorithm can find it correctly if m – k + 2 ≤ m, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if k ≥ 2 or equivalently ⌊(m + 1)/2⌋ ≥ 2 which holds for all m ≥ 3.

Therefore, the algorithm gives the correct output for inputs of size m + 1 and the proof is complete by induction.

---

3. There are n cities where for each i < j, there is a road from city i to city j which takes $T_{i,j}$ time to travel. Two travelers Marco and Polo want to pass through all the cities, in the shortest time possible. In the other words, we want to divide the cities into two sets and assign one set to Marco and assign the other one to Polo such that the sum of the travel time by them is minimized. You can assume they start their travel from the city with smallest index from their set, and they travel cities in the ascending order of index (from smallest index to

largest). The time complexity of your algorithm should be O(n^2). Prove your algorithm finds the best answer.

Solution:

Sub Problem and Recurrence Relation:

Let OPT[i] be the optimal solution of Marco and polo traveling through cities where Marco and polo indexing is given as i, I+1.

Let X[i] be the time taken to travel consecutively without any breaks from city 1 to I

```
PS[1] = 0;

for(i=2;i<=n;i++) {

        PS[i] = PS[i-1] + T[i-1][i];  //where PS[ ] is the prefix sum of each city from city 1

}
```

Which gives for example, S[4] = T(1,2) + T(2,3) + T(3,4) and so on for the later.

Equation: $OPT[i] = min(OPT[j] + T[i, j+1] + PS[j] - PS[i+1])$ where j = (i+1) to (n-1)

One person goes from i to j+1 then the other person covers i+1 to j as the person who covered i to j+1 can't move backwards.

Base Case: $OPT[n-1] = 0$. Which implies both the persons are at n-1,n.

Algorithm:

```
int function(int time[ ][ ],int S[ ]) {

opt[n];
opt[n-1] = 0;

for(i=n-2;i>=1;i--) {

        for(j=i+1;j<n;j++) {

                OPT[i] = min(OPT[j] + T[i, j+1] + s[j] - s[i+1] //for the situation discussed above.

        }

}
```

//for below for loop: There exits a situation where person starts from first city1 to i-1 position before jumping. For one person travels from 1 to i-1 and jumps from i-1 to i+1

for(i=2,i<n-2;i++) {

return min($opt[i] + PS[i-1] - PS[1]$);

}

}

Time Complexity: O(n^2)

---

4. **Erica is an undergraduate student at USC, and she is preparing for a very important exam. There are n days left for her to review all the lectures. To make sure she can finish all the materials, in every two consecutive days she must go through at least k lectures. For example, if k = 5 and she learned 2 lectures yesterday, then she must learn at least 3 today. Also, Erica's attention and stamina for each day is limited, so for i'th day if Erica learns more than a_i lectures, she will be exhausted that day. You are asked to help her to make a plan. Design a Dynamic Programming algorithm that output the lectures she should learn each day (lets say b_i), so that she can finish the lectures and being as less exhausted as possible(Specifically, minimize the sum of max(0,$b_i - a_i$)). Explain your algorithm and analyze it's complexity.**

Solution:

Sub Problem and Recurrence Relation:

Let $OPT[i]$ be the number of lectures covered before Erica can get exhausted.

Assume indexing starts from 0.

$$OPT[i] = max(a[i], k - OPT[i-1])$$

Base case:

$$OPT[0] = a[0]$$

Algorithm(Pseudo Code):

function(int a[ ],int k) {

opt[0] = a[0];

for(i=i;i<n;i++){

$$OPT[i] = max(a[i], k - OPT[i-1])$$

}

return OPT;

Time Complexity:

Since in the above algorithm, there is one for loop which runs for n, then the time complexity will be **O(n).**

---

5. **Due to the pandemic, You decide to stay at home and play a new board game alone. The game consists an array a of n positive integers and a chessman. To begin with, you should put your character in an arbitrary position. In each steps, you gain a_i points, then move your chessman at least a_i positions to the right (that is, $i' >= i + a_i$). The game ends when your chessman moves out of the array. Design an algorithm that cost at most O(n) time to find the maximum points you can get. Explain your algorithm and analyze its complexity.**

Solution:

Recurrence Relation:

Let OPT(i) be the maximum points we get. So out recurrence relation now will be the following:

if $a_i + 1 \leq n$ then $OPT(i) = max(OPT(i+1), OPT(i+a_i) + a_i)$

Else $OPT(i) = max(OPT[i+1], OPT[i])$

Base Case:

$OPT(n) = a_n$

Algorithm:

We will be using recursive dynamic programming to solve this.

```
int a[];

Int OPT[n];
int n;
int fun(int i)
{
for(int i=0;i<n;i++)
        {
              OPT[i] = a[i];
        }
        for(int i = n-2; i >=0; I++){
          if (a[i] + i <= n-1){
                    OPT[i]=max(a[I]+OPT(i+a[i]),OPT(i+1));
          }
          else{
                    OPT[i] = max(OPT(i), OPT(i+1));
          }

        return OPT[0];
}
```

Time Complexity: We go through all the elements of the array, so the complexity is O(n)

---

6. Joseph recently received some strings as his birthday gift from Chris. He is interested in the similarity between those strings. He thinks that two strings a and b are considered J-similar to each other in one of these two cases:

1. a is equal to b.
2. he can cut a into two substrings a_1,a_2 of the same length, and cut b in

the same way, then one of following is correct:
(a) a_1 is J-similar to b_1, and a_2 is J-similar to b_2.

(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.
He ask you to help him sort this out. Please prove that only strings having the same length can be J-similar to each other, then design an algorithm to determine if two strings are J-similar within O(nlogn) time (where n is the length of strings).

Solution:

Proof:

By proof of contradiction, let us assume the strings are of different length are J-Similar to each other. Case 1 fails as **a** can not be equal to **b** in any-case given the length is different for both. Considering case 2, if length of a < length of b then length of a_1 < length of b_1 and length of a_2 < length of b_2.

- Or - if length of a > length of b then length of a_1 > length of b_1 and length of a_2 > length of b_2.

Therefore, it fails in case 2 too. So, by contradiction, a and b should be of same length ignorer to be J-Similar.

Algorithm(Pseudo-Code):

```
If (length(a) ≠ length(b)):
        Return false
If length(a) = length(b):
        if(length is odd):
                if(a == b):
                        return true
                else:
                        return false
        if(length is even):
                - split the arrays a, b recursively
                - sort // sort (a1 & a2) and (b1 & b2). Not the individual characters, we
                        only sort blocks of strings.
                // for string adcb, a1 = ad and a2 = bc. a1 is already sorted. A2 will
                // become bc and now we merge them. The string a will now be "adbc"
                - merge them in alphabetical order (merge a1, a2 to a & b1, b2 to b)
                if ( a == b ):
                        return true
```

else:

return false

Complexity: Comparing a and b takes $O(n)$ and if the length is even, the steps mentioned above is merge sorting which takes $O(nlogn)$.

So the total complexity will be O(nlogn)

---

7. Chris recently received an array p as his birthday gift from Joseph, whose elements are either 0 or 1. He wants to use it to generate an infinite long super- array. Here is his strategy: each time, he inverts his array by bits, changing all 0 to 1 and all 1 to 0 to get another array, then concatenate the original array and the inverted array together. For example, if the original array is [0, 1, 1, 0], then the inverted array will be [1, 0, 0, 1] and the new array will be [0, 1, 1, 0, 1, 0, 0, 1]. He wonders what the array will look like after he repeat this many many times.

He ask you to help him sort this out. Given the original array p of length n and two indices a, b (n ≪ a ≪ b, ≪ means much less than) Design an algorithm to calculate the sum of elements between a and b of the generated infinite array pˆ, specifically, $\sum_{a \le i \le b}$ pˆ. He also wants you to do it real fast, so make sure your algorithm runs less than O(b) time. Explain your algorithm and analyze its complexity.

Solution:

Algorithm:

Let denote Input Array as P and reverse of this as R. Now, let's examine the pattern for different values of number of times the operations are performed.
N=1 - P. N=2- PR. N=3- PRRP. N=4 - PRRPRPPR. N=5 - PRRPRPPRRPPRPRRP.
From the pattern, we can see that every block of 4 elements(P or R) has 2 Ps and 2Rs.
Now We know the number of 0s and 1s in given array. So to calculate the sum of digits between a and b, we will be using divide & conquer.
We will be using an array of length b to store the length after each step m. For m>1, the length is $2^m * n$
Implies
Complexity: **O(logb)** here.

divide the subarray from index a to index b in sections as describe below :
Block B1 = Sum till index a-1 —> S_a
Block B2 = Sum till Index b —> S_b
Return the difference between B1, B2.

Algorithm:

1. Block B1:
length of a block pr is 2n, and the sum of elements in the block pr is n.

Find an index a' in arrLen such that it's value is the first index after a which is equal to 2^k*n, for some k < log b. We will be using binary search on the array. Complexity here is O(log(log b)), as the size of arrLen is log b.

s_a = arrLen[a'] /2

blockLen = 2n

Initialize s_b = 0

Total_blocks = (a' - a)/blockLen

Remaining = (a' - a) % blockLen

s_a = s_a – (Total_blocks * n)

Sum of remaining elements = Suml - (traversing through the first (remLen-1) elements of the block pr)

2. Block B2:
   This can be done similar to the previous step, except that now, we're looking for a index b' which is the last index before b which is equal to 2^k* m. The steps to do this have been given below.

length of a block pr is 2n, and the sum of elements in the block pr is n.

Find an index b' in arrLen such that it's value is the first index before b which is equal to 2^k*n, for some k < log b. This can be done using binary search on the array arrLen. The runtime complexity of this step is O(log(log b)), as the size of arrLen is log b.

s_b = arrLen[b'] /2

Initialize blockLen = 2n

Initialize $s_b = 0$

Total_blocks = (b-b')/blockLen

Remaining = (b-b') % blockLen

$s_b = s_b$ + (Total_blocks * n)

Sum of remaining elements = Suml + (traversing through the first (remLen) elements of the block pr)

3. Return $s_b - s_a$
