CSCI 570 : HW-03 (Spring 2021)

March 10, 2021

Question 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).

Answer

Firstly, lets define the Master Theorem.

$$T(n) = a.T(n/b) + f(n)$$
$$Let\ c = log_b(a), where\ a \geq 1\ and\ b > 1$$

Case 1: If $f(n) = O(n^{c-\varepsilon}),\ then\ T(n) = \theta(n^c)\ \ for\ \varepsilon > 0$

Case 2: If $f(n) = \theta(n^c log^k n), k \geq 0\ then\ T(n) = \theta(n^c log^{k+1} n)$

Case 3: If $f(n) = \Omega(n^{c+\varepsilon}),\ then\ T(n) = \theta(f(n))\ \ for\ \varepsilon > 0$

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

   $a = 4, b = 2, c = log_2(4) = 2$

   Since, $f(n) = n^2 log(n) = \theta(n^2 log n)$, $k = 1$, this satisfies condition of Case 2 in Master Theorem.

   So, $T(n) = \theta(n^2 log^2 n)$

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

   $a = 8, b = 6, c = log_6(8)$

   Since, $f(n) = n log(n) = O(n^{log_6 8})$, this satisfies condition of Case 1 in Master Theorem.

   So, $T(n) = \theta(n^{log_6 8})$

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

   $a = \sqrt{6006}, b = 2, c = log_2(\sqrt{6006})$

   Since, $f(n) = n^{\sqrt{6006}} \implies logf(n) = \sqrt{6006}log(n)$

   and, $log(n^c) = log_2(\sqrt{6006})log(n)$

   Using asymptotic analysis we can say, $\sqrt{6006} \cdot log(n) > log_2(\sqrt{6006}) \cdot log(n)$ and because log is monotonically increasing function we can also say that $f(n) = \Omega(n^c)$

   Applying Case 3 of Master Theorem, $T(n) = \theta(n^{\sqrt{6006}})$

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

   $a = 10, b = 2, c = log_2(10)$

   Since, $f(n) = 2^n = \Omega(n^{log_2(10)})$, Lets check if f(n) meets the regularity condition.

$$a \cdot f(n/b) \leq c \cdot f(n)$$
$$\implies 10 \cdot 2^{n/2} \leq c \cdot 2^n$$
$$\implies c \geq 10 \cdot 2^{-n/2}$$

   For sufficiently large n this inequality holds true, thus we can apply Case 3 of master theorem.

   So, $T(n) = \theta(2^n)$

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

   Lets assume $n = 2^m$

$$T(2^m) = 2 \cdot T(2^{m/2}) + log_2(2^m)$$

   If $S(m) = T(2^m)$, then

$$S(m) = 2 \cdot S(m/2) + m$$

   Here $a = 2, b = 2, c = log_2(2) = 1$

   So, $f(m) = m = \theta(m^c \cdot log^k m)$, where $k = 0$. This satisfies case 2 of Master's theorem.

$$S(m) = \theta(mlog(m))$$
$$\implies T(2^m) = \theta(mlogm)$$
$$\implies T(n) = \theta\big(log(n) \cdot log(logn)\big)$$

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

   $a = 1, b = 2, c = log_2(1) = 0$

   Since, $f(n) = (10 - n)$, it is not asymptotically positive. Hence we cannot apply master theorem here.

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

   Here, a is not a constant but a function of n, this is a limitation of Master Theorem and thus we cannot apply it here.

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

   $a = 2, b = 4, c = log_4(2) = 0.5$

   Since, $f(n) = n^{0.51} = \Omega(n^{0.5})$, this satisfies condition of Case 3 in Master Theorem.

   So, $T(n) = \theta(n^{0.51})$

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

   $a = 0.5, b = 2, c = log_2(0.5) = -1$

   Here, $a < 1$ which is a limitation of Master Theorem and thus we cannot apply it here.

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

    $a = 16, b = 4, c = log_4(16) = 2$

    Since, $f(n) = n! = \Omega(n^{2+\varepsilon})$ $\varepsilon > 0$, this satisfies condition of Case 3 in Master Theorem.

    So, $T(n) = \theta(n!)$

Question 2

Consider an array A of n numbers with the assurance that $n > 2, A_1 \geq A_2\ and A_n \geq A_{n1}$. 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.

Answer

Part (a)

Lets suppose for the purpose of contradiction that there exists no local minimum in A.

So after the element A_1, all the value A_i (i>1) should be monotonically decreasing ($A_i \geq A_{i+1}$), otherwise if any one of the value A_{i+1} is bigger or equal to A_i, then it will create local minimum.

But it was assured to us that last value in A (A_n) is greater than or equal to second last number A_{n-1}, this will make A_{n-1} a local minimum. **This is a contradiction, and we can say that there always exists a local minimum for A.**

Part (b)

We can compute the local minimum using Divide and Conquer strategy, below are the steps of the algorithm.

Step 1: If array size is 3 then return the middle element as local minimum.

Step 2: Find the middle element, compare it with the both the adjacent elements. If middle element is smaller or equal to both the adjacent elements then return middle element as local minimum.

Step 3: Otherwise, atleast one of the adjacent element of middle should be smaller or equal to middle element. Repeat from step 1, with half of the array containing one of the smaller element.

This algorithm guarantees that we will find a local minimum because in each step we are generating a smaller version of original problem as $A[mid] \geq A[mid+1]\ or\ A[mid-1] \leq A[mid]$.

Time Complexity : The above algorithm divides array in half and perform constant (2) number of comparison on each division. The algorithm can be shown by following recurrence relation.

$$T(n) = T(n/2) + \theta(1)$$

Using Master Theorem.

$a = 1, b = 2, c = log_2(1) = 0$

This satisfies Case 2 of Master Theorem, $fn) = 1 = \theta(n^0 log^k n)$ where $k = 0$

So **Time Complexity is** $\theta(logn)$

Question 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.

Answer

Subproblem

Lets define subproblem OPT[i,j] which gives minimized cost where i is the last city visited by Marco and j be the last city visited by Polo in their respective sets. Lets assume i<j without the loss of generality.

Recurrence Relation

Lets consider different possibilities to calculate OPT[i,j]

Case 1 (i < j-1):

If Polo's last visited city is $j-1$, then it is obvious that Marco's last city is before $j-1$. So j city could only be visited by Polo otherwise both Marco and Polo will be ahead of i^{th} city which will make definition of OPT[i,j] invalid. To find $OPT[i,j]$ we will just add $T_{j-1,j}$ to $OPT[i,j-1]$.

Case 2 (i = j-1)

If Marco's last visited city is i, so here again there are two possibilities. First possibility is Marco visited all cities from 1 to i and Polo has only 1 city in its set that is j, we can calculate time for this using a prefix array in constant time. Second possibility is Polo visited j^{th} city from some city which is before i^{th} city. Here we have to find option with minimum time, which also can be done in constant time if we just keep minimum value ($MN[j-1]$) maintained while calculating OPT[i,j-1] for every i < j-1.

$$Prefix[j] = \begin{cases} 0 & \text{if } j = 1 \\ Prefix[j-1] + T_{j-1,j}, & \text{otherwise} \end{cases}$$

$$OPT[i,j] = \begin{cases} OPT[i,j-1] + T_{j-1,j} & \text{if } i < j-1 \\ min(\ MN[j-1]\ ,\ Prefix[j-1]\ ), & \text{if } i = j-1 \end{cases}$$

$$MN[j] = min_{k=1}^{j-1}(\ OPT[k,j] + T_{k,j+1}\ )$$

Base Case: $OPT[1][2] = 0, MN[2] = 0 + T_{1,3}$. We assuming $i < j$ so we don't need values where $i \geq j$.

Pseudo Code

```
int T[n][n];  //given distance between every pair of cities

int OPT[n-1][n]; //keeps track of minimum cost
int MN[n];
int Prefix[n]; //Prefix[i] : time taken by a person to travel from city 1 to i

Prefix[1] = 0
for i in 2...n:
    Prefix[i] = Prefix[i-1] + T[i-1][i]

OPT[1][2] = 0 //for just 2 cities cost is 0
MN[2] = 0 + T[1,3]

for j in 3...n{
    for i in 1...j{
        if i < j-1{
            OPT[i][j] = OPT[i][j-1] + T[j-1][j]
        } else{
            OPT[i][j] = min( MN[j-1], Prefix[i] )
        }

        if j < n{
            MN[j] = min( MN[j], OPT[i][j] + T[i][j+1] )
        }else{
            MN[j] = min( MN[j], OPT[i][j] )
        }
    }
}

minimum_time = infinity  //to compute minimum time to travel.
for i in 1...n-1{
    minimum_time = min( minimum_time, OPT[i][n] )
}
```

Time Complexity : Calculating prefix will take $O(n)$ time, while filling every value of MN and OPT will take $O(n^2)$. So overall complexity for this algorithm is $O(n^2)$.

Space Complexity : We are using a n x n table to store optimal value and n x 1 to keep track of minimum value for last iteration and to store prefix. So overall space complexity also be $O(n^2)$.

Question 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 an 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.

Answer

Subproblem

Lets define the sub-problem OPT[j, i] which gives the minimum exhaustion $\big($sum of max(0, $b_i - a_i$)$\big)$ for upto i^{th} day and where j lectures were done on $i^{th} day$. MN[k] gives the minimum exhaustion if atleast k lectures were done on previous day.

Recurrence Relation

If we have done j lectures on i^{th} day, then we can select sub-problem where we have done atleast $k - j$ lectures on $(i - 1)^{th}$ day and add exhaustion level if we done lectures greater than a_i. MN[k] will give the minimum exhaustion if atleast k lectures on $(i - 1)^{th}$ day. We can find minimum exhaustion by just using MN, but to backtrack and get optimal number of lectures we are required OPT.

$$OPT[j, i] = MN[k - j] + max(0, j - a_i)$$

$$MN[j] = \begin{cases} OPT[j,i] & \text{if } j+1 > k \\ min(OPT[j,i], OPT[j+1,i]) & \text{otherwise} \end{cases}$$

Base Case : $OPT[j,1] = MN[j] = max(0, j - a_1)$, where $0 \leq j \leq k$

We will then populate $OPT[j,i]$ for all days $1 \leq i \leq n$ and $0 \leq j \leq k$. Minimum exhaustion will be given by $min_{j=0}^{k} OPT[j,n]$..

To get optimal number of lectures we can backtrack starting from n to previous days and keeping the exhaustion level minimum, also keeping lectures at least k including both days. Example, if j lecture were selected on n^{th} day then we have to select atleast k-j lectures on $(n-1)^{th}$ day keeping exhaustion level minimum.

Psuedo Code

```
int OPT[k+1][n];
int MN[k+1][2]
int a[n];       //given Erica’s stamina and attention level on each day

for j in k...1{
    OPT[j][1] = MN[j][1] = max( 0, j-a[i] )
}

for i in 2...n{
    for j in k...1{
        OPT[j][i] = MN[k-j][1-i%2] + max(0, j-a[j])

        if j+1 > k{
            MN[j][i%2] = OPT[j][i]
        }else{
            MN[j][i%2] = min(OPT[j][i], MN[j+1][i%2])
        }
    }
}

min_penalty = infinity
for j in 1...k:
    min_penalty = min(min_penalty, OPT[j][n])
min_penalty  //minimum exhaustion that will incur to Erica

//to get lectures to be done on each day
lectures = []
arr = []
min_j = 0
for i in n...1{
    min_v = infinity
    for j in min_j, k:
        if OPT[j][i] < min_v:
            min_v = OPT[j][i]
            min_j = j
    lectures.push_front(min_j)
    min_j = k-min_j
}
```

```
lectures //it will give optimal number of lectures to be done on each day.
```

Time Complexity : $O(kn)$, Since we finding value of each value in OPT table using 2 for loops of size k and n respectively. To backtrack and get the optimal number of lectures it time complexity is also $O(kn)$.

Space Complexity : We are using a k x n table to store optimal value and k x 2 to keep track of minimum exhaustion for last day then space complexity is $O(kn + 2k) = O(kn)$.

Question 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.

Answer

Subproblem

Lets define the sub-problem OPT[i] which gives the maximum number of points chessman can get after any position after i including i.

Recurrence Relation

For a particular position i, maximum point chessman could get is a_i plus maximum points chessman could get after moving atleast a_i positions right. It is quite intuitive that maximum point we can get at position i is a_i plus the maximum points for some position after $a_i + i$.

To find out OPT[i] we can use following recurrence relation.

$$OPT[i] = max \begin{cases} \begin{cases} a_i & \text{if } i + a_i > n \\ a_i + OPT[i + a_i], & \text{otherwise} \end{cases} \\ \\ OPT[i+1] \end{cases}$$

Base Case : $OPT[n] = a_n$

Psuedo Code

```
int OPT[n];
int a[n];       //points chessman will get at position i
```

```
OPT[n] = a[n]
for i in n-1...1{
    if i+a[i] > n {
            OPT[i] = max(a[i], OPT[i+1] )
    }
    else{
            OPT[i] = max( a[i]+OPT[i+a[i]], OPT[i+1] )
    }
}

OPT[1]   //maximum points you can get for some arbitrary position
```

Runtime Complexity

Time Complexity : Since we are doing a single linear pass over all n positions, it will be linear $O(n)$

Space Complexity : We are only storing single values per position so, space complexity will also be linear $O(n)$.

Question 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).

Answer

Firstly lets prove that only two strings of same length can be J-similar to each other.

For two strings to be J-similar they have to be equal or their halves (a_1, a_2 for a) and (b_1, b_2 for b) have to be J-similar using following rule.

$$(a_1 \text{ J-similar } b_1) \; and \; (a_2 \text{ J-similar } b_2)$$

or

$$(a_2 \text{ J-similar } b_1)\ and\ (a_1 \text{ J-similar } b_2)$$

We can see first case where $a = b$, is not possible because both the strings are of different length. Similarly it is easy to conclude that their halves will also be different length $|a_1|\ or\ |a_2| \neq |b_1|\ or\ |b_2|$. And for strings to be J-similar their both halves have to be J-similar to each other in any order. Here equality condition will be unsatisfied because of not all of them are of same length.

Thus, for 2 strings to be J-similar they have to of equal length.

Algorithm :

We will use the following divide and conquer algorithm to find if the strings (a and b) are J-similar.

```
Function J-Similar( A, B )
    STEP 1 : If A and B are of different length return False
    STEP 2 : A', B' = Split-and-Rearrange(A, B)
    STEP 3 : If A'==B': return True else return False


Function Split-and-Rearrange( A, B )
    STEP 4 : IF if A and B are of odd length then return A, B
    STEP 5 : Else divide A and B into 2 halves a1, a2 and b1, b2 respectively
    STEP 6 : a3, b3 = Split-and-Rearrange(a1, b1)
    STEP 7 : a4, b4 = Split-and-Rearrange(a2, b2)
    STEP 8 : a', b' = Rearrange( a3, a4, b3, b4 )  [O(2n)]
    STEP 9 : return a', b'


Function Rearrange( a3, a4, b3, b4 ):
    STEP 10 : if a3 > a4 then a' = a4a3 else a3a4. [O(n)]
    STEP 11 : if b3 > b4 then b' = b4b3 else b3b4. [O(n)]
    STEP 12 : return a', b'
```

Here, Step 1, 4, 5, 9, 12 takes constant time. In step 6 and 7 we are making 2 recursive calls and dividing the original problem into 2 halves. In step 10, 11 which is our merging step, we are comparing 2 string and rearranging them alphabetically which will take linear time O(2n).

Using this, we can define the time-complexity of algorithm using following recurrence relation:

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

$a = 2, b = 2, c = log_2(2) = 1$

Here, we can see $f(n) = n = \theta(n^1 log^k n)$ where $k = 0$

This satisfies Case 2 of Master Theorem, and we can define time complexity of above algorithm as $T(n) = \theta(n^c log^{k+1} n) = \theta(nlogn)$

Question 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 ($n \ll a \ll b, \ll$ means much less than) Design an algorithm to calculate the sum of elements between a and b of the generated infinite array p, specifically, $\sum_{a \le i \le b} \hat{p}_i$. 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.

Answer

We will use the Divide-And-Conquer strategy to get the sum of all elements before some index i. We can use this strategy twice to calculate sum before a and b, then take the difference to get the sum of elements between a and b.

Algorithm

Suppose P be the array he received and $\bar{P}$ be its inversion where both are of length n. For purpose of understanding lets consider the part of infinte array S that is generated using $P \ and \ \bar{P}$.

$$S = P\ \bar{P}\ \bar{P}P\ \bar{P}PP\bar{P}\ \bar{P}PP\bar{P}P\bar{P}\bar{P}P$$

Here, a space denote where the array is getting doubled after appending inverted of previous array. To make the array double, we will consider the previous array as not inverted and just append its inversion at the end, we will call this appended part inverted.

Now, whenever we divide the non-inverted array in 2 halves, left part we will be non-inverted and right part will be call inverted. Similarly, if we divide the inverted array in 2 halves, right part will be inverted and left part will be non-inverted. Example, in S which is of length $(16 \cdot n)$, $S[1 : 8 \cdot n]$ is inversion of $S[8 \cdot n + 1 : 16 \cdot n]$. Also $S[8 \cdot n + 1 : 12 \cdot n]$ is inversion of $S[12 \cdot n + 1 : 16 \cdot n]$.

We will use this property to develop an D & C algorithm to calculate sum of elements before some index i. Firstly lets define the a formula which will give index of element just ahead of i and is generated from last operation. Lets call this index u.

$$u = n \cdot 2^{\lceil log(\lceil i/n \rceil) \rceil}$$

We will recursively divide the the index u (generated from i) into 2 parts using middle index (calculated using low and high). If i lies in the left of middle index we will discard the right sub array and return sum as 0. if i lies in the right of middle index we can compute of sum of left sub-array using (mid-low)/2. because it is obvious that the length of left sub-array is some permutation containing equal number of P and $\bar{P}$, so this permutation will have equal number of 1's and 0's and its sum will be half of length of sub-array. If the length any sub-array is equal to n then we will return sum of elements of P or $\bar{P}$ depending whether it is not inverted or inverted respectively. If i lies inside the sub array of length n, then we will just return sum of initial $i\%n$ elements of P or $\bar{P}$ depending whether it is not inverted or inverted.

To find sum of elements between index a and b. We could just subtract sum of elements before b with sum of elements before a.

Pseudo Code

We will use the following Divide and Conquer algorithm to find sum of elements between a and b.

```
P  <- given array
P' <- inversion of P
n  <- length of P

Function Upper-Bound( i )
    STEP 1 : return n*2^( ceil( log2(ceil( i/n )) ) )


Function Recursive-Sum(i, low, high, not_inverted )
    STEP 2: Calculate mid = (low+high)//2
    STEP 3: If high-low+1 == n then
              if i is between low and high then
                return sum(P[:i%n]) if not_inverted else return sum(P'[:i%n])
              else
                return sum(P) if if not_inverted else return sum(P')
    STEP 4: If i < low then return 0
    STEP 5: If i <= mid then
              return Recursive-Sum(i, low, mid, not_inverted)
    STEP 6: Otherwise
              return (mid-low+1)//2
                        + Recursive-Sum(i, mid+1, high, !not_inverted)

Function Sum_Between_a_and_b( a, b ):
    STEP 7: sum_a = Recursive-Sum(a, 1, Upper-Bound(a), True)
    STEP 8: sum_b = Recursive-Sum(b, 1, Upper-Bound(b), True)
    STEP 9: return sum_b - sum_a
```

Time Complexity

Here, In step 2 and 4 we are doing the constant amount of work. In step 3, we are calculating sum of P or $\bar{P}$ of size n, so it will be O(n). In step 5 or 6, we are dividing the array into single sub-problems. So using this we can define the following recurrence relation for Recursive-Sum function.

$$T(k) = T(k/2) + O(n)$$

Since given index k $(a\ or\ b) \gg n$. So we can write $T(k) = T(k/2) + O(1)$.

Using Masters Theorem, $a = 2, b = 2$ and $c = log_2 1 = 0$. Here, $f(k) = 1 = O(k^0 log^0 k)$, which satisfies Case 2 of Masters Theorem.

Thus, $T(k) = \theta\big(log(k)\big)$.

Here we are doing computing Recursive_Sum twice (for a and b). So time complexity is $\theta(log(a) + log(b))$. But here $b \gg a$, so **Time Complexity is** $\boldsymbol{\theta\big(log(b)\big)}$**.**