CSCI 570 – Homework 3
Due: March 7th, 2021

Name: Rohan Mahendra Chaudhari
USC ID: 6675-5653-85
Email ID: rmchaudh@usc.edu

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:

$T(n) = a.T(n/b) + f(n)$

Case 1: if $f(n) = O(n^{c-E})$ then $T(n) = \Theta(n^c)$, $E>0$

Case 2: if $f(n) = \Theta(n^c \log^k n)$, $k>=0$ then $T(n) = \Theta(n^c \log^{k+1} n)$

Case 3: if $f(n) = \Omega(n^{c+E})$ then $T(n) = \Theta(f(n))$, $E>0$

a. Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying case 2 of the generalized Masters theorem, $T(n) = \Theta(n^2 \log^2 n)$.

b. Observe that $n^{\log_b a} = n^{\log_6 8} = n^{1.1605}$ and $f(n) = n\log n = O(n^{1.1605})$.Thus, invoking case 1 of the Master's Theorem gives $T(n) = \Theta(n^{\log_6 8}) = \Theta(n^{1.1605})$.

c. We have $n^{\log_b a} = n^{\log_2 \sqrt{6006}} = n^{6.2761}$ and $f(n) = n^{\sqrt{6006}} = n^{77.4984}$ Thus, from case 3 of the Master's Theorem $T(n) = \Theta(f(n)) = \Theta(n^{\sqrt{6006}})$.

d. We have $n^{\log_b a} = n^{\log_2 10} = n^{3.322}$ and $f(n) = 2^n$. Therefore from case 3 of the Masters theorem implies $T(n) = \Theta(f(n)) = \Theta(2^n)$

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 $S(\cdot)$ 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 Masters Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $m^{\log_b a} = m$. We express the solution in terms of T(n) by

   $T(n) = T(2^m) =S(m) = \Theta(m \log m) = \Theta(\log_2^n \log \log_2^n)$, for large enough n so that the growth expression above is positive.

f. Master theorem cannot be applied to $T(n)=T(n/2)-n+10$, since here f(n) is not positive which is a requirement for the master theorem to be applicable.

g. Master theorem cannot be applied to $T(n) = 2^n T(n/2) + n$, since here $a = 2^n$ which is not a constant and a must be a constant for the master theorem to be applicable.

h. $T(n) = 2T(n/4) + n^{0.51}$: This recurrence would fall under case 3.

   $a = 2$, $b = 4$, $f(n) = n^{0.51}$
   $c = \log_b^a = \log_4^2 = 0.50$
   Thus, $n^{0.51} = \Omega (n^{0.50})$
   Which implies $\Theta(n^{0.51})$

i. Master theorem cannot be applied to $T(n) = 0.5T(n/2) + 1/n$, since here $a < 1$. For the master theorem to be applicable $a >= 1$.

j. $T(n) = 16T(n/4) + n!$: This recurrence would fall under case 3.
   $a = 16$, $b = 4$, $f(n) = n!$
   $c = \log_b^a = \log_4^{16} = 2$

Thus, $n^2 = \Omega(n!)$
Which implies that $T(n) = \Theta(n!)$

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:

   The following algorithm solves this problem in $O(\log n)$ time:

```
LocalMin(A[1..n]) :
    if n<100
        find the smallest element in A by brute force
    m←⌊n/2⌋
    if A[m] < A[m + 1] :
        return LocalMin(A[1..m + 1])
    else:
        return LocalMin(A[m..n]))
```

   If n is less than 100, then a brute-force search runs in $O(1)$ time. There's nothing special about 100 here; any other constant will do.

   Otherwise, if $A[n/2] < A[n/2 + 1]$, the subarray $A[1..n/2 + 1]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray. **– (1)**

   Finally, if $A[n/2] > A[n/2 + 1]$, the subarray $A[n/2..n]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray. **– (2)**

   The running time satisfies the recurrence $T(n) \leq T(\lceil n/2 \rceil + 1) + O(1)$. Except for the +1 and the ceiling in the recursive argument, which we can ignore, this is the binary search recurrence, whose solution is $T(n) = O(\log n)$.

   Alternatively, we can observe that $\lceil n/2 \rceil + 1 < 2n/3$ when $n \geq 100$, and therefore $T(n) \leq T(2n/3) + O(1)$, which implies $T(n) = O(\log_{3/2} n)$ $= O(\log n)$.

Thus with the given boundary conditions of the original problem and **(1)** & **(2)**, an array must always contain at least 1 local minimum. Also since the algorithm is based on binary search where we compare middle element with its neighbours.
If middle element is not greater than any of its neighbours, then we return it. If the middle element is greater than its left neighbour, then there is **always** a local minima in left half.
If the middle element is greater than its right neighbour, then there is **always** a local minima in right half.

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 travellers 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:

   The basic idea is to split the array into 2 disjoint sets, solve the algo on the two sets, and then merge the solutions in order to find the path with minimum cost that passes through all the cities as per the premise of the problem. For the cut, we can find length of the array such that,

   a. If length is odd, split the array into 2 sets containing (n+1)/2 and (n-(n+1)/2) elements.
   b. If length is even, split the array into 2 sets containing (n/2) elements each.

   To merge the solutions we join the two arrays by making an edge swap.

   To choose which edge swap to make, we consider all pairs of edges of the recursive solutions consisting of one node ‘i’ from the left and one node ‘j’ from the right and determine which pair minimizes the increase in the following cost $T_{i,j}$.

Here is the pseudocode for the algorithm:

```
fun algo(P):
    case(|P|)
            of 0,1 => raise TooSmall
              | 2 => {(P[0],P[1]),(P[1],P[0])}
              | n => let
                   val ($P_l$,$P_r$) = splitLongest(P)
                   //splits the input into 2 halves – left and right
                   val (L,R) = (algo($P_l$) || algo($P_r$))
                   val (c,(i,j)) = $\text{minVal}_{\text{first}}\{T_{i,j} : i \in L, j \in R\}$
               in
                   SwapEdges(append(L,R),i,j)
              end
```

The function swapEdges(N,i,j) finds the nodes i and j in N and swaps the endpoints (there are two ways to swap, so the cheaper is picked).

Now let's analyse the cost of this algorithm in terms of work. We have

$$W(n) = 2W(n/2) + O(n^2)$$

$f(n) = O(n^2)$
$a = 2, b = 2$
$c = \log_2 2 = 1$
Thus, $f(n) = \Omega(n^c)$
$n^2 = \Omega(n)$ (Case 3) – only internal states
Thus, time complexity is $O(n^2)$

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 ith 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 analyse it's complexity.
Hint : k is O(n)

Solution:
Let a be an array of max number of lectures Erica can study on a specified day before getting exhausted.
Let k be the min number of lectures Erica must study in 2 days.
Let b be the number of lectures Erica studies in a particular day.
Let dp[b,k] be the max number of lectures Erica can study to reach atleast ‘k’ in 2 days using ‘b’ [0<=b<=a] number of lectures per day where ‘a’ is the maximum number of lectures she can learn per day before getting exhausted. [1<=a<=k]

Recurrence for dp[b,k]
```
If($b_i$ > $a_i$):
    sum = sum + max(0,($b_i$ – $a_i$)) //penalty
else:
    dp[$b_i$,k] = (dp[$b_i$-1,k] or dp[$b_i$-1,k-$b_i$])
```

Base case:
dp[b,0] = 0

Pseudo code:
```
lecture(a,b,k):
    //Here, a = array of max number of lectures Erica can study per day
    n = len(b)
    days = 2 //every 2 days
    cnt = 0
    sum = 0
    penalty = INT_MAX
    for i in range(1, n + 1):
            if(cnt < days):
                    if($b_i$ > $a_i$):
                            sum = sum + max(0,($b_i$ – $a_i$))
                    else:
                            dp[$b_i$,k] = (dp[$b_i$ – 1] or dp[$b_i$ – 1,k-$b_i$])
                    cnt = cnt + 1
            else:
                    sum = 0
                    cnt = 0
                    if(sum < penalty):
                            penalty = sum
                            res = dp[n,k]
    return res
```

Time complexity:
O(n.1) //Here, n is the size of the array we loop over and 1 is the work at each position visited.

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 analyse its complexity.

   Solution:

   Initialize an array dp[] where dp[i'] will store the maximum answer to reach position i' from an arbitrary position i.

   Recurrence:
   We can define the iterative structure to fill the table by using the recurrence relation of the recursive solution.
   $dp[i'] = a[i'] + max(dp[i' + a_i], dp[i' + a_i + 1], …., dp[i' + a_i + dp[i'] + 1]$
   where, dp[i'] stores the maximum answer to reach point i' from i.

   After filling the table, our final solution gets stored at the last Index of the array i.e. return dp[N-1].

   Base:
   dp[0] = 0

   Pseudo Code
   //Here, s is the arbitrary start index and n is the size of array a

```
maxpoints(a,n,s):
    [n] dp = {INT_MAX}
    dp[0] = 0
    for i in range(s,n):
            for j = i+a[i] to max(i+a[i] + 1, n):
                    dp[j] = a[i] + max[dp[j],1+dp[i]]
    return dp[n-1]
```

Time Complexity:
$O(n^2)$

The problem can also be solved by the greedy approach in O(n) time complexity.

Pseudo Code:
```
maxpoints(a,n,s):
    prev = s
    current = s
    point = a[s]
    for i in range(s,n):
        if(a[i]>a[previous]):
            point = point + a[current]
            previous = current
        current = max(current,i+a[i])
    return point
```

Time Complexity:
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 a1,a2 of the same length, and cut b in the same way, then one of following is correct:
        (a) a1 is J-similar to b1, and a2 is J-similar to b2.
        (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. 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:
   Given 2 string a and b, we must first check the length of both the strings.

1. If length is odd, we cannot split the 2 strings into 2 equal halves, thus we just check if a and b are similar strings (equal in length and if the strings are same). If yes, then we say the 2 strings are J similar.

2. If length is even, we must split the strings into 2 halves and check if they are J-similar by using the above mentioned premise.
   The algorithm splits the 2 strings into 2 halves and then calls itself for the 2 halves. We store the 4 halves calculated in variables a1,a2,b1,b2. Then we check if (a1 is J-similar to b1 and if a2 is J-similar to b2) or if (a2 is J-similar or b1 and a1 is J-similar to b2). To check if 2 strings are J-similar we first check if they are of the same length. Finally, the algorithm merges the 2 sorted halves in order to obtain the rearranged strings. If the 2 strings are J-similar the projected strings for both a and b after rearranging/merging would be also similar.

   ```
   Algo(s,l,r)
   If r>l:
         1. Find middle point to divide the string into 2 halves:
               Middle m = l + (r-l)/2
         2. Call Algo for first half if the current strings are J-similar :
               Call Algo(s,l,m)
         3. Call Algo for second half if the current strings are J-
            similar:
               Call Algo(s,m+1,r)
         4. Merge the 2 halves sorted in step 2 and 3 after checking if
            they are J-similar
               Call merge(s,l,m,r)
   ```

   Here, the merge() function is used for merging the 2 halves and the Algo() function is used for splitting the strings into 2 halves and checking if they are J-similar.

   The algorithm is recursive and thus the time complexity can be expressed as,
   $T(n) = 2T(n/2) + \Theta(n)$
   Thus, according to master theorem
   $a = 2$, $b= 2$, $f(n) = n$
   $c = \log_b^a = \log_2^2 = 1$
   Thus, $n = \Theta(n)$
   Therefore according to case 2,
   Time complexity is $\Theta(n \log n)$

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 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 pˆ, specifically, $\Sigma_{a \le i \le b}$ pˆi. He also wants you to do it real fast, so make sure your algorithm runs less than O(b) time. Explain your algorithm and analyse its complexity.

   Solution:

   Let us denote an input array P (ex: [0,1,1,0]) and reverse of this as R (ex: [1,0,0,1]). Now, let us 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 and so on
   Step 1: We find the mid of the current array.
   Step 2: We compare the mid to the value of ‘a’, if the value of mid < a, then we must skip the split and move on to generate the next inverted array until mid >=a.
   Step 3: Once the value of mid >= a, we must then check if len(array)=b. If no, then we must continue to concatenate arrays until the length(array) >= b.
   Step 3: Once the len(array)>=b then we must find the mid and recursively call the function to add all elements on the left of mid until low reaches index ‘a’ and then recursively call the function again to add all the elements on the right of mid until high reaches index ‘b’.

   Recurrence equation:
   2T(b/2) + O(b-a)
   Here, we consider b since we only create the array until b.

Pseudo code:

```
def add(a,b,p):
    if(a == b):
        return p[a]
    mid = (a+b)/2
    if(mid >=a):
        if(len(p) >= b):
            return add(a,mid,p)+add(mid+1,b,p)
        else:
            p = concatenation(p)
    else:
        p = concatenation(p)
```

Time complexity:

$2T(b/2) + O(b-a)$

$f(n) = (b-a)$

$c = \log_2 2 = 1$

Thus, $b = \Omega(b-a)$

Therefore, by the case 3 of master theorem,

$T(n) = \Theta(b)$