Muhammed ÖZKAN 151044084

CSE-321 Introduction to Algorithm Design

Fall 2020 Take Home Final

1.

In order to build a dynamic programming solution, we must separate the problem into smaller subproblems. We call each subproblem a state. By combining the answers of subproblems, we can reach the answer to the full problem.

Let's think about the subproblems of our current problem. The full problem requires finding the answer to the full sequence. For a subproblem, we'll find the answer to a specific range from the original sequence.

Recursive formula:

$$LP[i \dots j] = \begin{cases} 1 & \text{if } i = j \\ \text{LP}[i+1 \dots j-1] + 2 & \text{if } i = j \\ \max(\text{LP}[i+1 \dots j], \text{LP}[i \dots j-1]) & \text{if } X[i] \neq X[j] \end{cases}$$

```
algorithm  longestPalindrome(String, i, j, table)
    if (i greater than j) then
        return 0

    if (i equal j) then
        return 1

    key ← (i, j)
    if (key not include table) then
        if (String[i] equal String[j]) then
            table[key] ← longestPalindrome(String, i + 1, j - 1, table) + 2
        else then
            table[key] ← max(longestPalindrome(String, i, j - 1, table),
                             longestPalindrome(String, i + 1, j, table))

    return table[key]
```

This algorithm explanation:

-Maintain a boolean table[n][n] that is filled in bottom up manner.

-The value of table[i][j] is true, if the substring is palindrome, otherwise false.

-To calculate table[i][j], check the value of table[i+1][j-1], if the value is true and string[i] is same as string[j], then we make table[i][j] true.

-Otherwise, the value of table[i][j] is made false.

-We have to fill table previously for substring of length = 1 and length = 2 because as we are finding , if table[i+1][j-1] is true or false , so in case of

(i) length == 1 , lets say i=2 , j=2 and i+1,j-1 doesn't lies between [i , j]

(ii) length == 2 ,lets say i=2 , j=3 and i+1,j-1 again doesn't lies between [i , j].

We use the LCS algorithm (Longest Common Subsequence) to find the longest common subsequence among the original sequence and reversed sequence. Here original LCS and reverse LCS are a function that returns the longest common subsequence between the pair of strings, now the answer from LCS will be the longest palindromic subsequence.

When we look at the algorithm structure, although it looks recursive, we see that two nested loop structures work and these loops depend on the value of j. where j is the string length given. This approach reduces multiple runtime complexities compared to native recursion. The complexity of this solution is O (N * N), where N is the length of the text. **The time complexity of O (n^2)**

Program output according to test input:

Input:

String = "ABDDDDEEEFFFCBCDBDCBBCDDEEEFFF"

Output:

The Longest palindrome length = 15

//FFFCBCDBDCBCFFF

2.

We can use two different solutions for this problem. The first one is to sort the given array in quick sort first and then to print the smallest element in the given range, namely the 0th index. This will take **O(nlogn)** time. The second way is to sort the given ranges one by one in quick sort and print the 0th index as a result. If we follow the second way, it will take **O(nlogn)** time to place. However, if we apply the first way, this time we will sort at first, so the numbers in the range will change so the result will be different. Therefore, I will show you two differ methods below using quick sorting .

I first explain the algorithm of fast sorting since it uses fast sorting in two methods.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Let's analyze quick sort. After the do the partitioning, we recursively call our algorithm on two partitioned lists. Regarding to Hoare Partition algorithm, Since i is between 0 to n − 1, average value of T(i) is:

$$E(T(i)) = \frac{1}{n}\sum_{j=0}^{n-1} T(j)$$

$$E\big(T(n-i)\big) = E\big(T(i)\big) \therefore T(n) = \frac{2}{n}\left(\sum_{j=0}^{n-1} T(j)\right) + cn$$

$$nT(n) = 2\left(\sum_{j=0}^{n-1} T(j)\right) + n^2 \qquad Multiply\ by\ n$$

$$(n-1)T(n-1) = 2\left(\sum_{j=0}^{n-1} T(j)\right) + (n-1)^2 \qquad put\ n\ -\ 1\ for\ n$$

$$nT(n) = (n+1)T(n-1) + 2cn$$

Solving this recurrence relation:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{j=3}^{n+1}\frac{1}{j}$$

$$\sum_{j=3}^{n+1}\frac{1}{j} \to \ln n + \gamma \therefore \frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\ln n + 2c\gamma$$

Hence, $\mathbf{T(n)} \in \mathbf{O(nlog}\,\boldsymbol{n}\,\mathbf{)}$

```
algorithm findMinIntervalMethod1(Array,i,j)
    temp←Array[i-1:j-1]
    quick_sort(temp,0,len(temp)-1)
    return temp[0]


algorithm findMinIntervalMethod2(Array,i,j)
    quick_sort(Array,0,len(Array)-1)
    temp←Array[i-1:j-1]
    return temp[0]
```

As can be seen from the program outputs below, we obtained two different results for the same input. However, if we want to operate with more than one interval using the first of these, our time complexity will be **O (m * nlogn).** (number of m intervals). If we use the second method and reach the results at simultaneously and in **O (nlogn)** time, the result will not match the result we want in the first list because the values of the intervals will change.

Method1

Interval [1,9] 1

[7, 3, 9, 10, 5, 4, 1, 15, 17, 32, 5, 7, 0]

Method2

Interval [1,9] 0

[0, 1, 3, 4, 5, 5, 7, 7, 9, 10, 15, 17, 32]

3.

For the solution of this problem, the recursive formula below will create many similar sub-problems, so we apply an approach like the following algorithm for dynamic programming solution.

```
algorithm billboardMaximize(billboard, revenue, n, limit)
    //base value
    table ← array filling with revenue value
    i ← 1

    loop (i less than n) do
        //first two recursion case
        value ← max(table[i - 1], table[i])

        //picking ith billboard, third recursion case
        j ← 0
        loop (j less than i) do
            if (billboard[j] less than (billboard[i] - limit)) then
                value ← max(value, table[j] + revenue[i])
            j ← j + 1
        table[i] ← value
        i ← i + 1

    return table[n - 1]
```

This algorithm explanation:

The working system of this algorithm is as follows. It fill table with billboard revenues. Then It evaluate each possible place from the first point where the billboard can be placed to the end point. Every time It go to a new point, It re-evaluate the previous places so that It can get the maximum income. This every comparison is recording in table. Comparisons take maximum income into consideration. This process continues until the places where the billboards can be placed are finished.

Since there will be many overlapping sub-problems solving recursion, we converted the following recursion to dynamic programming.

Recursive formula:

K(0)=0

if b_i is not picked, K(i)=K(i-1)

if billboard placement starts from b_i K(i)=r[i]

if we place b_i then need to pace b_j where d[i]>d[j]-t such that revenue maximizes

$$K(i) = max\begin{Bmatrix} K(i-1) & if\ b_i\ is\ not\ picked \\ r[i] & if\ billboard\ placement\ starts\ from\ b_i \\ r[i] + K(j)\ where\ j < i\ and\ d[j] < d[i] - t & if\ we\ place\ b_i\ then\ need\ to\ pace\ b_j \end{Bmatrix}$$

If we calculate **the time complexity of** our algorithm, we see that there is a nested loop. These loops work as follows

When i=1, the inner loop will run once (for j==0).
When i=2, the inner loop will run twice. (for j==0 and j==1).
When i=3, the inner loop will run three times. (for j==0 and j==1 and j==2).
This algorithm will thus increment sum the following number of times:

$$\sum_{x=1}^{n} x - 1 = 0 + 1 + 2 + 3 + 4 \ldots + \mathrm{n} - 1 = \frac{\mathrm{n}(\mathrm{n}-1)}{2}$$

We can see by inspection that the sum is a "triangular number". Distributing n into the rest of the numerator, we arrive at $\frac{n^2-n}{2}$, of which the fastest-growing term is n^2 hence complexity is

$$\boldsymbol{\theta(n^2) \equiv O(n^2)\ and\ \Omega(n^2)}$$

Program output according to test input:

Input:

billboard = [6, 7, 12, 13, 14]

revenue = [5, 6, 5, 3, 1]

limit = 4

Output:

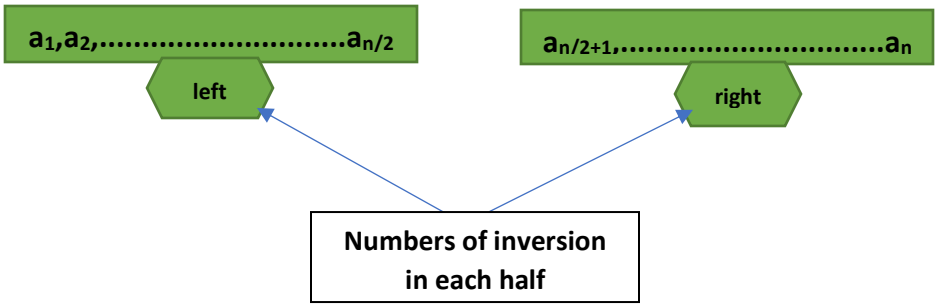
11

4. .

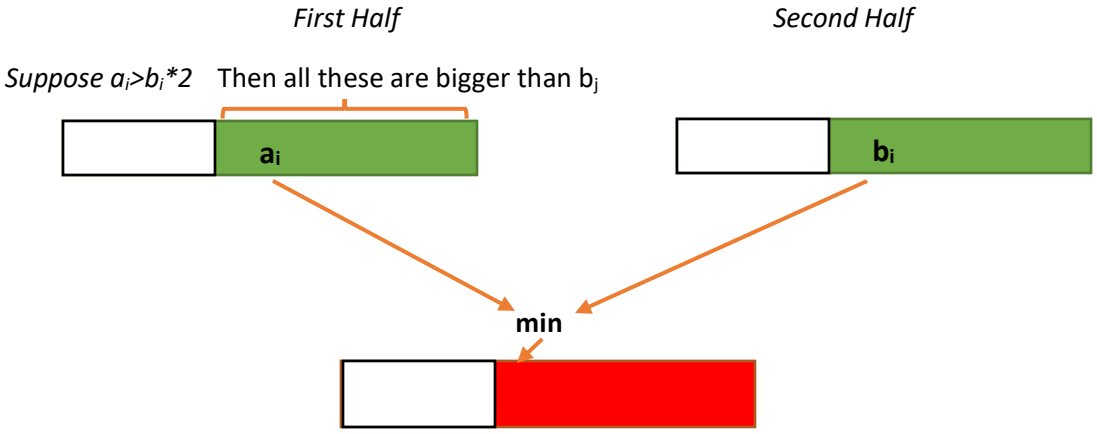
5.

The inversions that need to be counted during the merge step. Therefore, to get a number of inversions, that needs to be added a number of inversions in the left subarray, right subarray, and merge().Therefore, we will design our algorithm in a structure similar to merge sort.

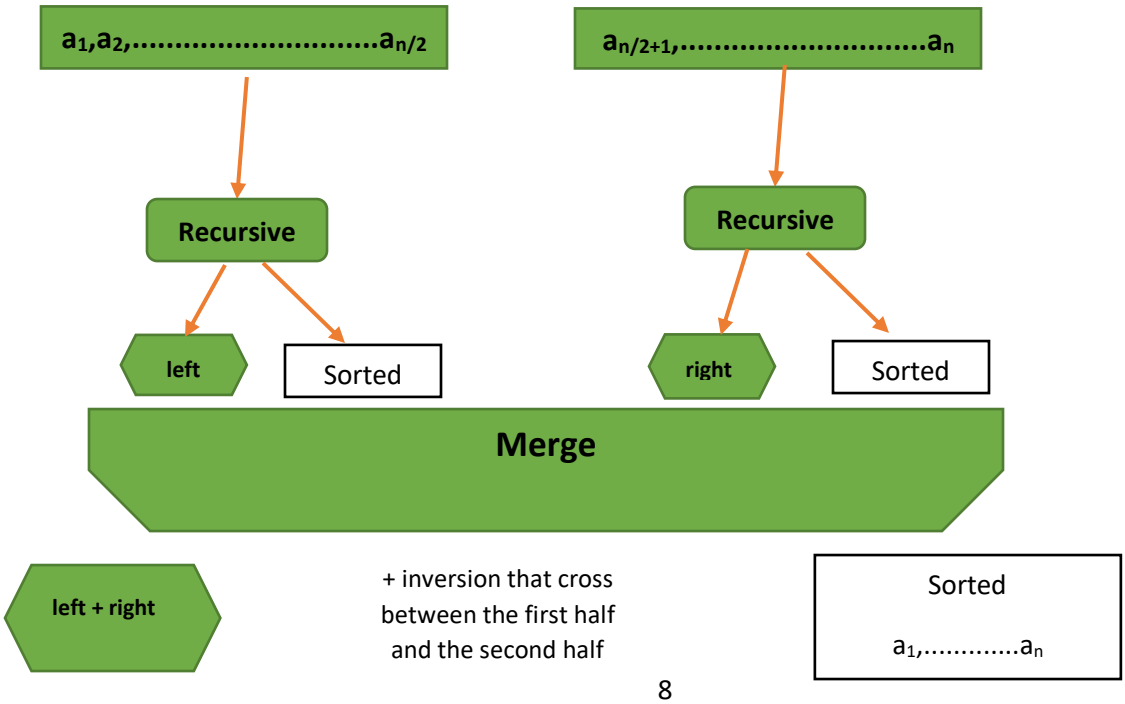
```
algorithm  getInversionCount(arr, temp_arr, left, right)

    inv_count ← 0

    if (left less than right) than
        mid ← ((left + right) / 2)
        inv_count ← inv_count + getInversionCount(arr, temp_arr,left, mid)
        inv_count ← inv_count + getInversionCount(arr, temp_arr,mid + 1, right)
        inv_count ← inv_count + merge(arr, temp_arr, left, mid, right)

    return inv_count


algorithm merge(arr, temp_arr, left, mid, right)
    i ← left
    j ← mid + 1
    k ← left
    inv_count ← 0

    loop ((i less than or equal mid) and (j less than or equal right)) do
        if (arr[i] less than or equal (arr[j]*2)) than
            temp_arr[k] ← arr[i]
            k ← k + 1
            i ← i + 1
        else
            temp_arr[k] ← arr[j]
            inv_count ← inv_count + (mid - i + 1)
            k ← k + 1
            j ← j + 1

    loop (i less than or equal mid) do
        temp_arr[k] ← arr[i]
        k ← k + 1
        i ← i + 1

    loop (j less than or equal right) do
        temp_arr[k] ← arr[j]
         k ← k + 1
         j ← j + 1

    temp ← left
    loop (temp less than right + 1) do
        arr[temp] ← temp_arr[temp]
        temp ← temp + 1

    return inv_count
```



In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j]*2, then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].



Result:



This algorithm explanation:

-The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.

-Create a function merge that counts the number of inversions when two halves of the array are merged, create two indices i and j, i is the index for first half and j is an index of the second half. if a[i] is greater than a[j]*2, then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].

-Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, number of inversion in the second half and the number of inversions by merging the two.

-The base case of recursion is when there is only one element in the given half.

-Return total number of inversions.

Time complexity:

Since our algorithm uses the structure of the merge sort algorithm, it is still **O (nlogn)** as we only added an additional function that does the job of counting the number of inversions. It seems that the added function does not add any additional complexity to the algorithm.

Program output according to test input:

Input:

array=[5, 6, 2, 3, 9] // [5,2] [6,2]

array=[11,5,2,0] // [11,5] [11,2] [11,0] [5,2] [5,0] [2,0]

Output:

Number of inversion(s) 2

Number of inversion(s) 6