ESO207 Theoretical Assignment 1

September 5, 2023

1 Ideal profits

Name : Shreyash Kumar
Roll no : 211009

1.1 (a)

We observe that wealth of a node depends only on itself or its parent node. We make a matrix , wherein the i^{th} row depicts the wealth of the $(i+1)^{th}$ node after y years

$$\begin{array}{c} \boldsymbol{W_{y+1}} \\ \textbf{N x 1} \\ \begin{vmatrix} W_{y+1}[0] \\ W_{y+1}[1] \\ W_{y+1}[2] \\ W_{y+1}[3] \\ . \\ . \\ W_{y+1}[n-1] \end{vmatrix} \end{array} \qquad\qquad \begin{array}{c} \boldsymbol{W_{y}} \\ \textbf{N x 1} \\ \begin{vmatrix} W_{y}[0] \\ W_{y}[1] \\ W_{y}[2] \\ W_{y}[3] \\ . \\ . \\ W_{y}[n-1] \end{vmatrix} \end{array}$$

We can set up a relation between the two matrices.

$$\begin{array}{c} \boldsymbol{W_{y+1}} \\ \text{N x 1} \\ \begin{vmatrix} W_{y+1}[0] \\ W_{y+1}[1] \\ W_{y+1}[2] \\ W_{y+1}[3] \\ . \\ . \\ W_{y+1}[n-1] \end{vmatrix} \end{array} = \begin{array}{c} \boldsymbol{M} \\ \text{N x N} \\ \begin{bmatrix} 2^{11} & 0 & 0 & 0 & . & . & 0 \\ 2^{10} & 2^{11} & 0 & 0 & . & . & 0 \\ 2^{10} & 0 & 2^{11} & 0 & . & . & 0 \\ 0 & 2^{10} & 0 & 2^{11} & . & . & 0 \\ . & . & . & . & . & . & 0 \\ . & . & . & . & . & . & 0 \\ 0 & 0 & 0 & 0 & . & . & 2^{12} \end{bmatrix} \end{array} \times \begin{array}{c} \boldsymbol{W_{y}} \\ \text{N x1} \\ \begin{vmatrix} W_{y}[0] \\ W_{y}[1] \\ W_{y}[2] \\ W_{y}[3] \\ . \\ . \\ W_{y}[n-1] \end{vmatrix} \end{array}$$

*leaf nodes have 2^{12} since they don't share their wealth
We now carry out matrix exponentiation to calculate the wealth of respective nodes after the given time.

Pseudocode :

```
int mult_mat(int a[ ], int b[ ])
{
    int ans = 0
    for i from 0 to n-1
        for j from 0 to n-1
            for k from 0 to n-1
                 ans += a[i][k] * b[k][j]
     return ans;
}

int pow(M,y)
{
    if(y == 0) return 1;
    int temp = pow(M,y/2)
    temp = mult_mat(temp,temp)
    if(y is ODD) temp = mult_mat(temp,y)
    return temp
}
```

1.2 (b)

Time complexity :

Running time of the algorithm is $(N^3 logm + N)$. Therefore, time complexity of the algorithm is $O(N^3 Log(m))$.

Correctness :

$$W_o = M_o x W_o$$
$$W_o = W_o$$

Assume W_y is correctly given as $M^y * W_o$, then using the discussed observation we can say,

$$\text{Wy+1 = M x Wy}$$

Replacing W_y with $M_y * W_o$, we get

$$W_{y+1} = M_{y+1} * W_o$$

Hence proved.

1.3 (c)

Since we now only have a single node, the final wealth will be 2^m times the inital wealth after m months, which can be calculated using the left shift operator :

$$W_f = W_o << m$$

2 Moody Friends

Name : Shreyash Kumar
Roll no : 211009

2.1 (a)

The given question is equivalent to finding the minimum size sub-array whose sum is greater than or equal to P, where capacity of each element(room) of the array is variable. We need to find the minimum number of contiguous rooms, the sum of the capacity of which meets or exceeds the required value P.
We approach the problem using two pointers, maintaining a window of variable size. We report the minimum-sized window among all the valid windows.

Pseudocode:

```
int min_cost(int rooms[ ],int n, int P,int C)
{

    int lo = 0,hi = 0,capacity = 0,min_len = 1e8;

    while(hi < n)
    {
        capacity += rooms[hi++];
        //we add the room's capacity to the total capacity, while also increasing the window size for the next iteration

        while(capacity >= P)
        {
            min_len = min(min_len, hi-lo);
            capacity -= rooms[lo++];
            //we decrease the size of the window for optimal cost while making sure the window is still valid
        }
    }
    return min_len*C;
}
```

2.2 (b)

We need to find the length of the maximum contiguous subsequence such that GCD of the subsequence is >= k.

- We apply binary search on the maximum length which ranges between 1 and n, taking logn time.
- We check every possible index if it can be the starting point of the subarray. This takes O(n) time. So we just have to find the GCD of subarrays in O(1) time, such that our overall algorithm becomes O(nlogn).
- We use range-minima query to solve the given problem, making a $n * logn$ matrix A, such that A[i][j] gives the GCD of all the elements from i to i + 2^i. (j < logn)

Forming the matrix takes extra space of nlogn and also takes nlong time. Therefore the overall time complexity of our algorithm is still **O(nlogn)**

2.3 (c)

Time complexity :

The main loop runs through the array once, with both the left and right pointers moving from the beginning to the end of the array. Each element is processed once and only once. Therefore, the loop's time complexity is **O(n)**.

Proof of correctness :

We assume that the algorithm correctly identifies the minimum length of a contiguous subarray whose sum is at least k at the end of each iteration up to index i.

For i = 0, we get the intialized value for minimum length, meaning we haven't encountered any such subarray yet, which is consisten.

Now, we need to prove that after the $(i+1)^{th}$ iteration, the algorithm still correctly identifies the minimum length of a contiguous subarray whose sum is at least "k" up to index (i+1).

During the $(i+1)^{th}$ iteration, the algorithm:

- Adds the element at index right to currentSum.
- Checks if currentSum is greater than or equal to "k."
- If it is, it enters a while loop that attempts to minimize the subarray length by moving the left pointer to the right while keeping the sum at least "k."
- Updates minLength if the current subarray length is smaller than the previously stored minLength.

Therefore, based on our assumption and the conditions checked during the $(i+1)^{th}$ iteration, the algorithm correctly identifies the minimum length of a contiguous subarray whose sum is at least "k" up to index (i+1).

3 BST universe

Name : Shreyash Kumar
Roll no : 211009

3.1 (a)

> We make use of the fact that the *Inorder traversal* (Left-Node-Right) of a BST is a sorted array. Since two of the nodes have been swapped, there will be two anomalies in the sorted array.
> To find the swapped values, we traverse the array. The first element that we encounter such that arr[i] > arr[i+1] is one of the elements that was swapped.
> To find the second element, we may traverse the array again in reverse order, and the first element that we encounter such that arr[i] < arr[i-1] will be our swapped value.

Now we proceed to find the common ancestors of the swapped values. We know that in a BST, the left child is smaller than the root and the right child is greater than the root.
We start from the root node as the current node. We encounter the following cases during our traversal :

- Current node is smaller than both the swapped nodes : move to the right child
- Current node is greater than both the swapped nodes : move to the left child
- Current node value lies between the values of the swapped nodes or Current node value is equal to the value of one of the swapped nodes: **least common ancestor** encountered

We can use a data structure to store all the common ancestors starting from the root to the LCA in the aforementioned traversal.

3.2 (b)

We take different approaches for different values of G and n :

- G > nlog(n) :
  We again invoke the fact that the inorder traversal of a BST gives a sorted array of node values.
  We copy the inorder traversal of the BST in another array and sort it, taking O(nlog(n)) time. All the anomalies encountered while comparing the two arrays will give us the list of all the swapped nodes.

  Pseudocode :

```
int arr[n], ideal[n], count = 0;
void inorder(root,int arr[ ]) {
if(root == NULL) return;
    inorder(root− >left);
    arr.push(root− >val);
    inorder(root− >right);
}
int main(){
    inorder(root,arr);
    ideal[n] < − arr[n];
    sort(ideal[n]);
    for(i = 1 − > n-1) {
        if(arr[i] is NOT EQUAL TO ideal[i]) count++;
    }
    return count;
}
```

- $G < n\log(n)$:
  We use a visited array for keeping track of all the node values of the BST and initialize it with 0.
  We store the inorder traversal of the BST in another array.
  Note that size of the visited array is G, while that of the traversal array is n.
  Now we keep two pointers, one at the start of each array. For any node value in the visited array, if vis[value] == 1, then our current pointer in the traversal array must point to a node having the same value if the given BST is ideal. Since our BST is not ideal, an inequality in the above condition can only mean a swapped node. In this manner, we can find out all the swapped nodes.

  Pseudocode :

```
int arr[n], val[G], count = 0;
void inorder(root,int arr[ ]) {
if(root == NULL) return;
    inorder(root− >left);
    arr.push(root− >val);
    inorder(root− >right);
}
int main()
{
    int ans;
    int ptr1 = 0, ptr2 = 0;
    while(ptr1 < G)
    {
        if(val[ptr1] == 1 and arr[ptr2] ! = val[ptr1])
        {
            ans.push(val[ptr1])
            ptr2++
        }
        else if(val[ptr1] == 1 and arr[ptr2] == val[ptr1])
            ptr2++

        ptr1++
    }
    print(ans)
}
```

Thus, we see that the problem can be solved in **min(G + n, nlog(n))** time

4 Helping Joker

Name : Shreyash Kumar
Roll no : 211009

4.1 (a)

The array that we obtain after the Master has shifted the top k cards to the bottom has a distribution similar to the one shown in Figure 1.

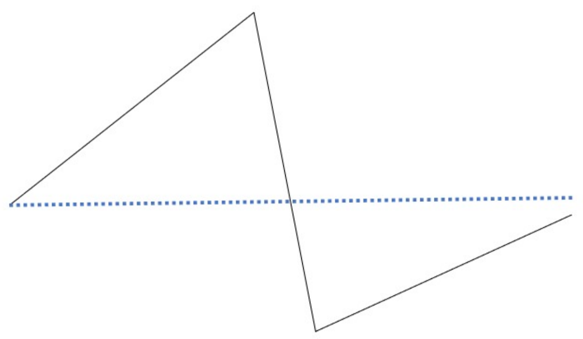

Figure 1: distribution of values on cards

We must note that the topmost card on the pile is always greater than the bottom-most card, as is shown by the dotted-line. We can classify the pile having two sections, the first containing cards having values $a_{k+1}, a_{k+2}, ..., an$ and the second containing cards having values $a_0, a_1, ...a_k$. The first section consists of n-k cards, while the second has k cards. Finding the size of either gives us the value of k.

Pseudocode:

```
int find_k(int query[ ], int n)
{
    int lo = 0, hi = n-1, mid;

    bool found = false;

    while(NOT found)

    {

        mid = (lo+hi)/2;

        if(hi - lo == 1) found = true;

        else if(query[mid] > query[lo]) lo = mid; //we are in the first section

        //it is sufficient to compare with query[lo] since query[lo] will always be greater than query[hi]
        else if(query[mid] < query[hi]) hi = mid; //we are in the second section

        //it is sufficient to compare with query[hi] since query[hi] will always be lesser than query[lo]
    }

    return (n-mid-1);
}
```

4.2 (b)

At every iteration of the while loop, the element is bound to be in either of the two halves, thereby assuring the updation and eventual termination of the loop.
The proposed algorithm is similar to binary search.
It follows the recurrence relation : T(N) = T(N/2) + c
T(N/2) = T(N/4) + c
T(N/4) = T(N/8) + c
T(N/2) = T(N/8) + c + c
Similarly, T(N) = T(n/2^k) + c + c + ...
T(N) = T(2^k/2^k) + c + ... + c (k times)
T(N) = T(1) + $c * 2^k$
As $2^k = n$, k = log(N), T(N) = c*log(N)

Therefore, we conclude that our algorithm has a running time complexity of **O(log(N))**

5 One Piece Treasure

Name : Shreyash Kumar
Roll no : 211009

5.1 (a)

We can find out whether a given substring is palindromic or not in O(1) time. So the brute force solution would be to check all possible substrings for the same, which would take O($n^2 * 1$) = O(n^2) time. But this is more than the number of queries we are allowed to make.

Observations :

- For an odd-sized palindrome centered at index i, if k is the length of the largest such palindrome, the total number of palindromes with center i is (k-1)/2 + (k-3)/2 + ... + 1.
- For an even-sized palindrome made by the indices i and i+1, we can similarly calculate the total palindromes
- We need to do this for all letters, which will take a running time of O(n). So, each operation must take less than $log^2(n)$ time.

Algorithm :

We first count odd-length palindromes by applying binary search on the answer. For index i, the minimum odd-sized palindrome will be of length 0 and maximum possible length would be min(i,n-i-1). W need to find the maximum length of k such that the oracle will return true for the substring S(i-k,i+k).
Then, we check whether i,i+1 can be the middle element of a palindrome. Similarly, we find the maximum possible k.

Time complexity :

Binary search takes 0(n*log(n)) time to execute. In the worst case, we end up applying binary searches for both odd and even length palindromes, requiring 2*n*log(n) queries.

Pseudocode :

```
int main()
{
    int sum = 0;
    //for odd-length palindromes
    for(i = 0 − > n-1)
    {
        int lo = 0, hi = min(i,n-i-1);
        int res;
        while(lo <= hi)
        {
            int mid = (lo + hi)/2;
            if(oracle(i-mid,i+mid))
            {
                res = mid;
                lo = mid + 1;
            }
            else hi = mid-1;
        }
        sum += res + 1;        //substring of length 1 is also a palindrome
        //for even-length substrings
        if(i < n-1 and oracle(i,i+1))
        {
        int lo = 0, hi = min(i,n-i-2);
        int res;
        while(lo <= hi)
        {
            int mid = (lo + hi)/2;
            if(oracle(i-mid,i+mid))
            {
                res = mid;
                lo = mid + 1;
            }
            else hi = mid-1;
        }
    sum += res;
    print(sum)
    }

}
```