

Naman Kumar Jaiswal

220687

Question 1: Ideal profits

Part a: We need to find the wealth of each company after m months. Since, there are no constraints on m , size and wealth, we need not care about storage data types.

Thinking Process: We need a time complexity which is $\log m$ for the m part. We also know that the main problem that worsens our time complexity is we need to do the distribution manoeuvre every year. This was the problem with iterative algorithm for Fibonacci Series problem as well. We solved it by using a $O(\log n)$ power function and using matrices to utilise matrix exponentiation to solve recursive relations.

Logic: We need to build a matrix of order $n \times n$, where n is the number of companies. We need to write a matrix that will capture the essence of distribution every year. The power exponentiation will happen yearly. Rest a factor 2^m is evident via yearly doubling.

2^{12} because of 12 months in a year. For k^{th} year. Prefix denotes company number. 1 is for root.

The root element will only experience halving of wealth every year.

$$\text{Relation: } {}_1A_k = {}_1A_{k-1} * (0.5) * 2^{12}$$

The leaf elements will experience no distribution and only wealth from parent.

$$\text{Relation: } {}_pA_k = {}_pA_{k-1} * 2^{12} + [p/2]A_{k-1} * 2^{12} * (0.25)$$

The rest elements will halve via distribution and gain wealth from parent.

$$\text{Relation: } {}_pA_k = {}_pA_{k-1} * 2^{12} * (0.5) + [p/2]A_{k-1} * 2^{12} * (0.25)$$

where $[x]$ is greatest integer function. $[p/2]$ is parent's company number of child element p .

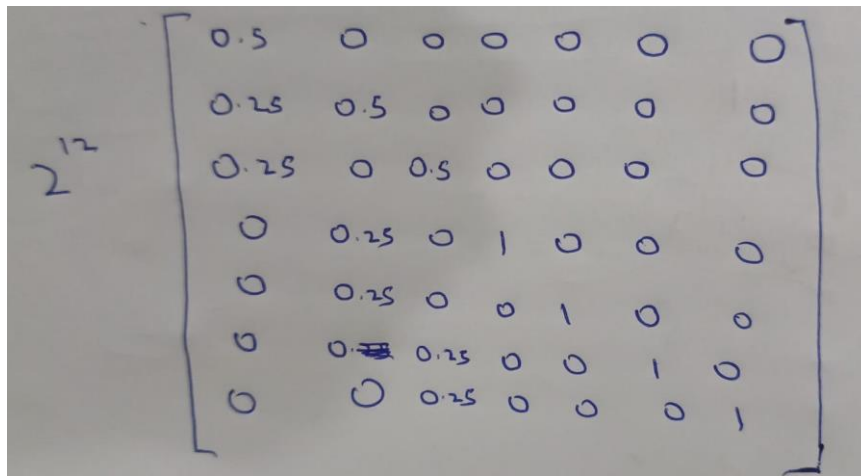
After applying this year-wise and adding $2^{m\%12}$ at end to constitute for the months that are not included in the yearly analysis.

Principle for Optimisation:

1. Representing the relations between company wealths in the form of a matrix. Using matrix multiplication to avoid keeping track of the companies every year. We will multiply the matrix multiple times and update the wealth at last. This contributes of n^3 part of complexity.
2. Use of divide and conquer power function to calculate matrix to the power $[m/12]$. This contributes to $\log m$ part of time complexity.

Matrix:

Example Matrix for $n = 7$:



A handwritten 7x7 matrix is shown. To the left of the matrix, the text 2^{12} is written. The matrix contains the following values:

0.5	0	0	0	0	0	0
0.25	0.5	0	0	0	0	0
0.25	0	0.5	0	0	0	0
0	0.25	0	1	0	0	0
0	0.25	0	0	1	0	0
0	0.25	0.25	0	0	1	0
0	0	0.25	0	0	0	1

Pseudocode:

//Matrix can be defined using stdlib library malloc function

//Matrix is to be implemented using arrays.

// Product Function returns matrix product of two matrices M1, M2

Product (M1, M2, order) {

array [order, order] M = 0;

// new matrix M is defined to store the answer

// initialised to zero

for (i=0; i<order; i++) {

for (j = 0; j<order; j++) {

for (k = 0; k< order; k++) {M[i][j] += M1[i][k] * M2[k][j];}}

return M;

```
// Mpower (Matrix, power, N) returns matrix raised to the power
//Power Function
Mpower (Matrix, power, N) {
    if (power == 1) {return Matrix;}           //base case
    M1 = Mpower (Matrix, Power / 2, N);
    M2 = Product (M1, M1);
    if (n % 2) {return Product (M2, Matrix);}
    return M2;
```

```
//Main Function
// defining the primary matrix
// n, m has usual meaning
array [n, n] M;
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
    if (i == j and i < n/2) M[i][j] = 1 << 11;           //Note-- 1
    if (i == j) M[i][j] = 1 << 12;                       //Note-- 2
    if (j = (i-1)/2) M[i][j] = 1 << 10;                  //Note-- 3
    else (M[i][j] = 0)}}
}
```

```
// Matrix Exponentiation
// dividing m into years and months
years = m / 12; m = m % 12;
```

```
//Inorder Traversal of Binary Tree
array [n] arr; static q = 0;
//For a Binary Tree root
```

```
Inorder (root) {
    if (root == NULL) return;
    Inorder (root -> left);
    arr[q++] = root -> value;
    Inorder (root -> right);}

// Inorder Traversal requires O(n) time as we go through each node once

// Let Companies_in_order = array after Inorder Traversal of BST T
```

// refer to Answer 3

M = Mpower (M, years, n);

//updating final wealth

array [n] Final_Wealth; S = 0;

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {S = 0;

S+= Companies_in_order[j] * M[i][j]}

Final_Wealth = S;}}

Note:

1. Root Case
2. Leaf Case
3. Inheritance Case

Part b: Time Complexity Analysis

Let us look at time complexities of all the steps one by one:

1) Building matrix M $\Rightarrow O(n^2)$

2) Matrix Exponentiation

a. Matrix Multiplication $\Rightarrow O(n^3)$

b. Matrix Power $\Rightarrow O(\log m)$

Step a occurs $\log m$ times

$\Rightarrow O(n^3 \log m)$

3) Inorder Traversal $\Rightarrow O(n)$

4) Final Wealth Updating $\Rightarrow O(n^2)$

Final Complexity: $O(n^3 \log m)$

Part c:

We are to find the wealth of a single company after m months in $O(n)$ time.

Thinking Process: The only source of change of wealth of the company is doubling of revenue after every month.

Logic: Wealth after m months $= 2 * \text{Wealth after } m-1 \text{ months} = 2^2 * \text{Wealth after } m-2 \text{ months} = 2^m * \text{Initial Wealth}$.

Principle for Optimisation:

1. Computing 2^m will require $O(\log m)$ time at best using the divide and conquer power function as taught in the lectures. We can reduce it to $O(1)$ using bit operations.

Pseudocode:

```
// m and initial wealth w be given as input
Power = 1 << m;
print (Power * w);
```

Naman Kumar Jaiswal

220687

Question 2: Moody Friends

Part a:

We need to find the minimum cost room allocation for which the sum of room capacities is more than equal to P .

Thinking Process: Since we have to find a solution in $O(n)$, in most cases what we can do at max is linear traversal any number of times. Since, possible room sequences are consecutive and have the sum of capacities more than P . We can use two iterators which go in the array at once.

Logic: The pair (i, j) defines the room sequence $arr[i, i+1, i+2, \dots, j]$. What makes this algorithm to not go to $O(n^2)$ or $O(n \log n)$ despite two independent (i, j) is the fact that we make sure j or i does not traverse the array once again. Consider that i and j might be independent but they are bound by a thread. If j is ahead of i , this thread will pull i ahead when it feels that the cases with the fixed i and j increasing are surely not going to be the answer. We are able to consider all the possible cases without considering all the n^2 cases by considering only the cases which have a potential to be the answer.

We are going to allow the variable `min_rooms` to keep track of the minimum cost divided the per room cost.

Principle for Optimisation:

1. Not computing all possible values of total room cost. Only checking for costs that are near to the minimum costs.
2. Possible rooms are consecutive and have the sum of capacities more than P .

Pseudocode:

```
// i and j are going to be the iterators and the array arr is given
// both are initialised to 0
min_rooms = n; Capacity_total = 0;

for (i = 0; i < n; i++) {
    while (j < n and Capacity_total < P) {                //to keep a thread like structure
        Capacity_total += arr[j]; j++;}
    min_rooms = (j - i < min_rooms)? j - 1: min_rooms;    //storing minimum cost
    Capacity_total -= arr[i];
}
print (min_rooms * C);
```

Part c:

Complexity Analysis:

The crux of the algorithm is that (i, j) are going to traverse the array only once despite having two loops. This simply means $O(n)$ time complexity.

This is very similar to the example given in the lecture on time complexity.

Proof of Correctness:

Since we are confirmed to get an answer, (it might be $C * n$), we can claim that the answer room subsequence must be having a starting point I and an ending point.

i is confirmed to be equal to I at some point. When $i = I - 1$, i will only increase when Total capacity becomes just greater than P. And then i becomes I while capacity either decreases below P or not. If capacity is just above P, we have our answer. For the other case, when capacity just crosses, we get our answer.

It can be clearly shown for case 1, if we don't take the final element, capacity falls below P, and taking another element is ridiculous.

For case 2, we are off the P threshold and the first value at which we get the threshold cleared is confirmed our answer.

Naman Kumar Jaiswal

220687

Question 3: BST Universe

Part a: We need to find the nodes which have been swapped in the BST.

Thinking Process: There are two cases possible. Either the nodes swapped are consecutive in the In Order Traversal of the BST or they are a few elements apart.

Comparing it to the sorted array of the Inorder traversal, we can find the swapped Nodes.

Logic: For a given array we can go from right to left and compare the value to the last value to find exactly where the kink is. In case of consecutive elements swapped, there will be one kink, else there will be two.

Principle for Optimisation:

Inorder Traversal of a BST is always in ascending order.

Pseudocode:

```
//Inorder Traversal of Binary Tree
```

```
array [n] arr; static q = 0;
```

```
//For a Binary Tree root
```

```
Inorder (root) {  
    if (root == NULL) return;  
    Inorder (root -> left);  
    arr[q++] = root -> value;  
    Inorder (root -> right);}
```



```
/* define empty array of Size 4. Now, traverse the array arr and whenever encounter a kink append the element and its next element into the array. The swapped nodes are the first and last element of the new array. */
```

```
array N [4]; static q1 = 0;
```

```
for (i = 0; i < n-1; i++) {if (arr[i] > arr[i+1]) {N[q1++] = arr[i]; N[q1++] = arr[i+1];
```

```
print (arr [0], arr [q1-1]);
```

```
/* Now, for the task of common ancestry. we can use binary search in the binary tree and store the encountered nodes in an array.
```

```
For example, if 2 and 7 are the two swapped nodes, we can use binary search for 2 in the swapped Tree to traverse the ancestors of 7 and Search for 7 to get the ancestry of 2. Now we can compare the ancestry to get the common ancestors. */
```

```
//Binary Search in a BST
```

```
static q = 0;
```

```
BinarySearchBST (root, target, arr) {
```

```
    if (root == null) return null;
```

```
    if (root->value == target) return root;
```

```
    arr[q++] = root->value;
```

```
    if (target < root->value) {return BinarySearchBST (root->left, target);}
```

```
    return BinarySearchBST (root->right, target);}
```

```
//common ancestry
```

```
array A1 [height (BST)], A2 [height (BST)];
```

```
BinarySearchBST (root, arr [0], A1);
```

```
BinarySearchBST (root, arr [q1 - 1], A2);
```

```
for (int i = 0; i < height (BST); i++) {if (A1[i] == A2[i]) print(A1[i]);
```

```
    else break;}
```

Complexity Analysis:

Inorder Traversal requires $O(n)$ time as we go through each node once. Now, traversal in array arr to find kinks will require $O(n)$ time as well.

And two Binary Search Queries will take $O(\log n)$ time each.

Overall, it will take $O(n)$ time.

Part B:

Logic: For a given Inorder Traversal array we can go from right to left and compare the value to the corresponding element in its sorted counterpart.

Principle for Optimisation:

Use of Count sort for $O(n)$ Sorting.

Pseudocode:

```
// For the given BST, traverse through the tree and store the Inorder traversal in an array arr
/* Now, check whether  $G > n \log(n)$ .
If it is we can use the quicksort method to sort the array.
Else, we can use Count Sort. */
```

```
// For Count Sort
CountingSort(arr) {
    max_val = G; min_val = 0;
    Array Count [max_val - min_val + 1] = 0;           //all initialised to 0

    // Count the occurrences of each value in the input array
    for (i = 0; i < (max_val - min_val + 1); i++) {count[arr[i] - min_val] ++;}
    // Reconstruct the sorted array from the count array
    Array sorted_arr [length(arr)];
    index = 0;
    for (i from 0 to max_val - min_val) {while count[i] > 0:
        sorted_arr[index] = i + min_val
        index += 1
        count[i] -= 1}
    return sorted_arr;

// Sorting in  $n \log n$  is allowed to be assumed
/* Now Traverse the array arr and whenever encounter a difference with respect to the sorted counterpart, store it. At the end you will get all the rearranged Nodes. */
for (int i=0; i < n; i++) {if (arr[i] != sorted_arr[i]) {print(arr[i]);}}
```

Complexity Analysis:

Inorder Traversal $\Rightarrow O(n)$ time

Count Sort $\Rightarrow O(G + n)$ time // if $G < n \log n$

Quick Sort $\Rightarrow O(n \log(n))$ time

Final Traversal $\Rightarrow O(n)$ time

Total Complexity: $O(\min(G + n, n \log(n)))$

Naman Kumar Jaiswal

220687

Question 4: Helping Joker

Part a: We need to find the number k using the minimum number of reveals in the set of cards.

Thinking Process: We are guaranteed that k is not equal to 0 or n and all elements are distinct, so we are not going to care about the trivial cases.

Initially all the elements were in an increasing order but now there is exactly one kink where there is the drop in the value and then the values start to rise.

Logic: For a given array we can go from right to left and compare the value to the last value to find exactly where the kink is. The final answer is going to be $\text{number_of_elements} - \text{kink_index} - 1$.

Principle for Optimisation:

1. Use of binary search/divide and conquer to find the kink_index.
2. Use of fact: Answer = $\text{number_of_elements} - \text{kink_index} - 1$

Pseudocode:

left = 0; right = $n - 1$; // n is the number of elements

```
Kink_index (left, right, array) {  
    while (left <> right) {  
        mid = (left + right + 1)/2;           // Note-- 1  
        if (array[mid] > array[n-1]) {left = mid;} //Note-- 2  
        else {right = mid-1;}  
    }  
    return n - 1 - left;                       // single element  
}
```

Here, left and right are the boundaries of the array where our kink element possibly resides. We are halving that region in every iteration.

Note:

1. We defined mid to be half of left + right + 1 so as to make sure that the while loop does not get stuck in a two-element situation where mid again and again goes to left. This occurs because left goes to mid instead of mid + 1.
2. The reason why left goes to mid and not mid + 1 is that we are not searching. We can not find kink without comparing to the next element. So, the possibility of the mid element becoming the kink_index is still there.

Part b: Time Complexity Analysis

Since, every time the loop gets executed the array size containing the kink element gets approximately halved. The problem gets converted to another problem where the size becomes $n/2$.

We can write the recurse relation:

$$T(n) = c + T(n/2)$$

where $T(n)$ is the time for n^{th} Size array.

c is the computational time.

Solving this relation yields.

$$T(n) = c \log n.$$

Thus, the Complexity is $O(\log n)$.

Naman Kumar Jaiswal

220687

Question 5: One Piece Treasure

We need to find the number of pairs of $\{i, j\}$ such that the substring $\text{Str}[i, i+1, \dots, j]$ is a palindrome. We already have a black box which inputting this pair returns whether this substring is palindrome or not.

Thinking Process: Every Palindromic substring is going to have a centre element (or two centre elements in case of even length) and two end elements. Both these end elements are equidistant from the centroid of the substring. Now, let us call this distance the radius of palindrome for this substring.

Ex: In $\{1, 3, 6, 8, 4\}$, 6 is the centre while 2 is the radius.

We can clearly see that for a given centre_index and a given radius there is a unique substring corresponding to $\{\text{centre_index} - \text{radius}, \text{centre_index} + \text{radius}\}$.

Hence, instead of looking for substring starting index wise, we will count them centre wise.

Logic: For a given centre, we are going to use Binary Search to find the limiting radius for which the $\{\text{centre}, \text{radius}\}$ pair is palindromic. Treat this like a change of co-ordinate system from i, j to centre, radius.

First, we will count the odd length substrings and then the even length substrings.

Principle for Optimisation:

1. If $\{\text{centre}, \text{radius}\}$ is palindromic then $\{\text{centre}, \text{radius}-1\}$ is also palindromic. Then $\{\text{centre}, \text{radius}-2\}$ is also palindromic.
2. Hence, if we find the radius of palindrome for a specific centre. There will be $\text{radius} + 1$ palindromes with this centre corresponding to $r = 0, 1, 2, \dots, r_0$.

This principle will help in reducing the time complexity.

Pseudocode:

//Let the black box function for returning palindrome or not be oracle(i,j).

//Function for finding radius of palindrome

//Binary Search

//Note-- 1

```
L_Binary_Search(centre) {  
    left = 0; right = centre;           //Note-- 2  
    Last_Palindrome_Index = centre;    //Note-- 3  
    while(left <> right) {  
        mid = (left + right)/2;  
        if (oracle (mid, 2*centre - mid)) {  
            right = mid - 1;  
            Last_Palindrome_Index = mid;}  
        else {left = mid + 1;}}  
    return centre - Last_Palindrome_Index + 1;}
```

```
R_Binary_Search(centre) {  
    left = centre; right = n-1;  
    Last_Palindrome_Index = centre;  
    while (left <> right) {  
        mid = (left + right)/2;  
        if (oracle (2* centre - mid, mid)) {  
            left = mid + 1;  
            Last_Palindrome_Index = mid;}  
        else {right = mid - 1;}}  
    return centre - Last_Palindrome_Index + 1;}
```

//Note-- 4

```
L_Binary_Search_even(centre) {  
    left = 0; right = centre;  
    Last_Palindrome_Index = centre;  
    while(left<> right) {  
        mid = (left + right)/2;  
        if (oracle (mid, 2*centre - mid + 1)) {  
            right = mid - 1;  
            Last_Palindrome_Index = mid;}  
        else {left = mid + 1;}}  
    return centre - Last_Palindrome_Index;}
```

```
R_Binary_Search_even(centre) {  
    left = centre; right = n-1;  
    Last_Palindrome_Index = centre;  
    while (left <> right) {  
        mid = (left + right)/2;  
        if (oracle (2* centre - mid - 1, mid)) {  
            left = mid + 1;  
            Last_Palindrome_Index = mid;}  
        else {right = mid - 1;}}  
    return centre - Last_Palindrome_Index;}
```

//Now for the main function

//Note-- 5

// Let Arr be the input array

```
Num_Palindromes = 0;                                //Count Variable  
for (i = 0; i < n/2; i++) {Num_Palindromes += L_Binary_Search(i);}  
for (i = n/2; i < n; i++) {Num_Palindromes += R_Binary_Search(i);}  
for (i = 0; i < n/2; i++) {Num_Palindromes += L_Binary_Search_even(i);}  
for (i = n/2+1; i < n; i++) {    Num_Palindromes += R_Binary_Search_even(i);}
```


Note:

1. There are 4 functions defined that count the number of palindromic sub sequences in each of the 4 cases. L_Binary_Search and L_Binary_Search_even count the sub sequences as centre being in the first half of the string. Same for L_Binary_Search and R_Binary_Search_even. The even counter parts cater the case for even length palindromes.
2. L_Binary_Search will fix left bound as 0 and right bound as the centre. Then find the limiting case when the radius is equal to the radius of palindrome for that centre. As we get a true value, we shift the right bound and as we get a false value, we shift the right bound.
3. We need to find the last found radius for which the pair (c, r) is palindromic. Which is closest to the limiting radius as we proceed along the iterations of binary search.
4. The difference between the even and the odd counterpart is that of oracle function and return value. Here, centre in L correspond to the substrings with centre elements (centre, centre+1) and centre in R correspond to the substrings with centre elements (centre -1, centre). Also, radius = 0 here means 0 possible substrings as opposed to 1 in odd substrings.
5. For the whole array, we traverse twice, once for odd palindrome and once for even palindrome. For each element we find the number of corresponding palindromes. We keep track of whether to apply the left or right counterpart.

Number of Queries:

For a single traversal for $i = 0$ to $n/2$ we will require at max $\log(i)$ queries and from $i = n/2$ to $n-1$ $\log(n-i)$ queries. This means for using the traversal 2 times, we need $2 * [\log 1 + \log 2 + \log 3 + \dots \log n/2 + \log n/2 + \log (n/2 - 1) \dots + \log 1]$ queries.

$$\text{Sum} = 4 * \log(\text{factorial}(n/2))$$

Now, using stirling's approximation

$$\ln n! = \ln 1 + \ln 2 + \ln 3 + \ln 4 + \dots + \ln n$$
$$\ln n! = \sum_{i=1}^n \ln n_i = \int_1^n \ln n \, dn$$
$$\ln n! = n \ln n - n + 1$$

$$\begin{aligned}\text{Sum} &= 4 * [n/2 * \log(n/2) - n/2 + 1] \\ &= 2n * \log(n) - 2n * \log 2 - 2n + 4\end{aligned}$$

So Approximately Number of Queries is $O(n * \log n)$.

$$\text{Number of Queries} < n * \log^2(n)$$