Theoretical Assignment 1

Havi Bohra (210429)

4 September 2023

Ans1(a)

Approach:

- Matrix Exponentiation Method: We will utilise the matrix exponentiation method to find the final wealth of the companies after "m" months.
- Transition Matrix Definition: A transition matrix of size "n x n" will be defined to facilitate the calculation of company wealth after one year.
- Total Wealth after m / 12 Years: Initially, we will calculate the total wealth of the companies after "m / 12" years through the following steps:
    - Raise the transition matrix to the power of "m / 12."
    - Multiply this result by the array representing the initial wealth distribution among the companies.
- Remaining Months: For the remaining months, we will take the result obtained in the previous step and multiply it by "2^(m % 12)" to determine the final wealth. This step accounts for the exponential growth of wealth, where the wealth doubles each month.

```
// Function to multiply two matrices A and B of size n x n
vector<vector<int>> matrixMultiply(vector<vector<int>>& A, vector<vector<int>>& B, int n) {
   // Initialize a matrix to store the result
   vector<vector<int>> result(n,vector<int>(n, 0));

   // Perform matrix multiplication
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         for (int k = 0; k < n; ++k) {
            result[i][j] += A[i][k] * B[k][j];
         }
      }
   }

   return result;
}

// Function for matrix exponentiation
vector<vector<int>> matrixExpo(const vector<vector<int>>& mat, int n, int p) {
   if (p == 1) {
      return mat;
   }
```

```
    vector<vector<int>> result(n, vector<int>(n, 0));
    vector<vector<int>> temp(n, vector<int>(n, 0));

    temp = matrixExpo(mat, n, p / 2);
    result = matrixMultiply(temp, temp, n);

    if (p % 2 != 0) {
        temp = matrixMultiply(mat, result, n);
        result = temp;
    }

    return result;
}

// Function to calculate final wealth of the companies after m months
vector<int> calculateWealth(int n, int m, const vector<int>& initial_wealth) {
    int a = 1<<12;
    int b = 1<<11;
    int c = 1<<10;

    // Create and populate the transition matrix
    vector<vector<int>> transition_matrix(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        if (i == 0) {
            transition_matrix[i][i] = b;
        } else if (i * 2 >= n) {
            transition_matrix[i][i] = a;
            transition_matrix[i][i/2] = c;
        } else {
            transition_matrix[i][i] = b;
            transition_matrix[i][i/2] = c;
        }
    }

    int years = m / 12;
    int remaining_months = m % 12;

    // Calculate the final matrix using matrix exponentiation
    vector<vector<int>> final_matrix = matrixExpo(transition_matrix,n,years);
    for (int i = 1; i < years; ++i) {
        final_matrix = matrixMultiply(final_matrix, transition_matrix, n);
    }

    // Calculate final wealth after m months
    vector<int> final_wealth(n, 0);
    for (int i = 0; i < n; ++i) {
        int temp = 0;
        for (int j = 0; j < n; ++j) {
            temp += transition_matrix[i][j] * initial_wealth[j];
        }
        // temp stores the wealth after 'm/12' years
        final_wealth[i] = temp * (1<<remaining_months);
    }

    return final_wealth;
}
```

calculateWealth will give us the required vector of final wealth of companies.

Ans 1(b)

Time Complexity Analysis:

- Initialising the `transition_matrix` takes O(n^2) time.
- The `matrixExpo` function, which includes matrix multiplication, has a time complexity of O(n^3) due to the matrix multiplication part. However, since you're applying matrix exponentiation with a power of m/12, it takes O(log(m/12)) = O(log(m)) iterations. Each iteration involves matrix multiplication, so the total time complexity for matrix exponentiation is O(n^3 * log(m)).
- Calculating the final values of `final_wealth` takes O(n^2) time.

Hence, the overall time complexity is O(n^3 * log(m)).

Correctness:

Let's analyse the correctness of the algorithm through mathematical induction.

Assertion P(i): After i months, `final_wealth` stores the correct wealth of each company.

Base Case P(0): After 0 months, `final_wealth` is equal to `initial_wealth`, which is true. Hence, P(0) is true.

Inductive Step:

Assuming P(i-1) to be true.

Case 1 (i-1 month is not the last month of the year):

Let `wealth[i][k]` denote the wealth of the k-th company after the i-th month. Therefore, from the algorithm, `wealth[i][k] = 2 * wealth[i-1][k]`.

Case 2 (i-1 month is the last month of the year):

```
wealth[i][k] = wealth[i-1][k] / 2 + wealth[i][k/2] / 4
```

Hence, P[i] is true. Thus, we've shown by induction that the algorithm correctly calculates the wealth of each company after i months.

Ans 1(c)

When wealth accumulates solely at the root node without any further distribution, it follows an exponential growth pattern. In this scenario, if we denote the initial wealth as "W" and it doubles each month, we can calculate the wealth after "m" months using the formula:

Wealth after m months = W * 2^m

A number can be multiplied by 2 using the left shift operator in constant time.

Hence, the final answer is W<<m.

Ans 2 (a)

Approach:

- Initialization: Start with two pointers, `start` and `end`, to define the current subarray of chosen rooms. Initialize a variable `length` to track the number of elements in the subarray.
- Capacity Check: While the current capacity of the selected rooms is less than the desired capacity `P` and the `end` pointer remains within the array bounds:
    - Increment the `end` pointer to expand the current capacity of the selected rooms.
- Capacity Met: When the current capacity of the chosen rooms becomes greater than or equal to the desired capacity `P`:
    - Increment the `start` pointer to reduce the current capacity of selected rooms.
    - Update the `length` to be the minimum of the current `length` and its previously stored value.
- Iteration: Continue iterating through the array, repeating steps 2 and 3, until either the `end` pointer reaches the end of the array or the `start` pointer goes out of bounds.
- Final Check: After processing the entire array, check if the `length` remains at its initial value (infinity). If it does, it indicates that no valid subarray meeting the capacity constraint `P` was found, and there's no valid answer. Otherwise, calculate the minimum cost as `length * C`, where `C` represents the cost per room.

This approach efficiently determines the minimum cost of selecting a subarray of rooms while considering a capacity constraint, and it handles cases where no valid subarray can be found.

n = Number of rooms
C = Cost of booking of each room
P = Minimum sum of capacity
Array 'capacity' = Array of capacities of n rooms

findMinCost(n, C, P, capacity): This function returns the required minimum cost for contiguous subarray of rooms with Sum of capacities >= P

Pseudocode:

```
int findMinCost(int n, int C, int P, vector<int>& capacity) {
    int start = 0;
    int end = 0;
    int curr_capacity = capacity[0];
    int length = numeric_limits<int>::max(); // Initialize length to infinity

    while (end < n && start < n) {
        if (curr_capacity < P) {
            end++;
            if (end < n) {
                curr_capacity += capacity[end];
            }
        } else {
            length = min(length, end - start + 1);
            curr_capacity -= capacity[start];
            start++;
        }
    }

    if (length == numeric_limits<int>::max()) {
        return -1; // No valid subarray found
    }

    int cost = length * C;
    return cost;
}
```

Ans 2(b)

Approach:

1 Binary Search: We utilise binary search to find the longest subarray where the greatest common divisor (GCD) of its elements is greater than or equal to the desired value `P`.

2 Initialization: We initialise the search interval with a `start` and `end` pointer, where `start` points to the beginning of the array, and `end` points to the end of the array.

3 Search for Longer Subarray: In each iteration of the binary search, we calculate the `mid` point of the search interval and check if we can find a subarray of length `mid` with GCD greater than or equal to `P`.

-> If such a subarray exists, we move the `start` pointer to `mid + 1` to search for an even longer subarray.

-> If such a subarray does not exist, we move the `end` pointer to `mid - 1` to search for a shorter subarray.

4 Repeat Binary Search: We repeat the binary search until the `start` and `end` pointers meet or cross. At this point, we have found the maximum length subarray with GCD greater than or equal to `K`.

This approach efficiently finds the desired subarray length by iteratively narrowing down the search interval using binary search, allowing us to optimise the calculation of the longest subarray with the required GCD condition.

n = Number of rooms
K = Minimum GCD of sub-array
capacity = Array of capacities of n rooms
findMaxSubarray(n, K, capacity): Returns the maximum number of contiguous rooms having GCD greater than K
isAvailable(n, K , capacity, mid): Returns true if a sub-array of length mid with GCD greater than or equal to K is present in the array. Else returns false.
Pseudocode:

```
bool isAvailable(int n, int P, vector<int>& capacity, int mid) {
   bool found = false;
   vector<int> Forward_gcd(n);
   vector<int> Backward_gcd(n);
   int curr_gcd = capacity[0];

   // Calculate Forward GCD of all elements in segments of length mid
   for (int i = 0; i < n; ++i) {
      if (i % mid == 0) {
         curr_gcd = capacity[i];
      }
      curr_gcd = GCD(curr_gcd, capacity[i]);
      Forward_gcd[i] = curr_gcd;
   }

   curr_gcd = capacity[n - 1];

   // Calculate Backward GCD of all elements in segments of length mid
   for (int i = n - 1; i >= 0; --i) {
      if ((i + 1) % mid == 0) {
         curr_gcd = capacity[i];
      }
      curr_gcd = GCD(curr_gcd, capacity[i]);
      Backward_gcd[i] = curr_gcd;
   }

   for (int i = 0; i <= n - mid; ++i) {
      int gcd = GCD(Backward_gcd[i], Forward_gcd[i + mid - 1]);
      if (gcd >= P) {
            found = true;
            break;
        }
    }

    return found;
}

// Function to find the maximum length subarray with GCD greater than or equal to P
int findMaxSubarray(int n, int P, vector<int>& capacity) {
    int start = 1;
    int end = n;
    int maxLength = -1;

    while (start <= end) {
        int mid = (start + end) / 2;
        bool possible = isPossible(n, P, capacity, mid);

        if (possible) {
            maxLength = mid;
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }

    return maxLength;
}
```

Ans 2(c)

Proof of Correctness of Algorithm in 2(a):

Proof of Correctness by Induction:

We will use mathematical induction to prove the correctness of the algorithm by inducting over the position of the `end` pointer in the array.

- Base Case (end = 0): When the `end` pointer is at position 0, if the first element of the array is greater than or equal to `P`, the `length` variable will be set to 1. Otherwise, it will remain at infinity.

Assertion:

- P[i]: When the `end` pointer is at position i, the `length` variable stores the size of the smallest subarray among all possible subarrays up to index i. (A possible subarray is one having a sum of capacities greater than or equal to `P`).

To Prove:

- The above assertion holds true for (i + 1).

Inductive Step:

The `end` pointer only moves to the next index if the sum of the current subarray becomes less than `P`. When the `end` pointer moves to (i+1) index, there are two possible cases:

Case 1: After adding the new element, the sum of the subarray remains lower than `P`. Since the sum is less than `P`, moving the `start` pointer will only decrease the sum. Hence, we cannot find any subarray of smaller size than the previously stored `length`. In this case, the `length` stores the size of the smallest possible subarray.

Case 2: After adding the new element, the sum becomes greater than or equal to `P`. In this case, we check if the size of the current subarray is smaller than the `length` variable. If it is, we update the `length` variable to the smaller value. Then, we increment the `start` pointer. Within this case, there are further two subcases:

- Case 2_a: After incrementing the `start` pointer, the sum is still greater than or equal to `P`. We again check if the size of the current subarray is smaller than the `length`. If it is, we update the `length` to a smaller value. We continue repeating these steps until the sum becomes less than `P`.
- Case 2_b: After incrementing the `start` pointer, the sum becomes less than `P`. In this case, we update the `end` pointer to the next index.

Since the `end` pointer iterates through all the indices, the `length` variable will indeed store the size of the smallest possible subarray. Therefore, by mathematical induction, we have shown that the algorithm correctly determines the smallest subarray satisfying the capacity constraint `P`.

Time Complexity Analysis of Algorithm in 2(a):

The Algorithm is structured around a single while loop, and each step within the loop takes constant time, O(1). Therefore, the overall time complexity is primarily determined by the number of iterations in this loop.

In each iteration, we either increment the `start` or `end` pointer. It's important to note that each pointer traverses the entire array exactly once during the execution of the loop. Therefore, the loop will run a total of 2n times in the worst case, accounting for both `start` and `end` pointers visiting each element of the array once.

As a result, the time complexity of the Algorithm is O(n).

Ans 3 (a)

Approach:

1. Correcting a Binary Search Tree:

In a Binary Search Tree (BST), the inorder traversal should yield a sorted, ascending order of elements. However, if two elements are swapped, this order is disrupted. There are two possible scenarios:

Case 1: When the swapped elements are adjacent in the inorder traversal, we will encounter a decreasing pair once.

Case 2: In other cases, there will be two instances of decreasing pairs.

2. Identifying the Swapped Nodes:

To identify the swapped elements, we maintain three pointers: `first`, `middle`, and `last`. The first decrement encountered updates `first` and `middle`. If a second decrement occurs, we update `last`.

3. Swapping the Nodes:

If `last` remains `NULL`, it implies that the swapped elements are adjacent. In this case, we swap `first` and `middle`. Otherwise, we swap `first` and `last`.

4. Finding Common Ancestors:

To find the common ancestors of the swapped nodes, we need to store the path from the root to each of these nodes in separate arrays. Then, by traversing these arrays, we can identify the common elements, which represent the common ancestors of the nodes.

- `recoverTree(root)`: This function identifies and swaps the nodes that need correction to restore the original BST.
- `inorderAndFindElements(root)`: This function performs an inorder traversal while updating the `first`, `middle`, and `last` pointers, which are essential in the `recoverTree` function.
- `findCommonAncestors(root, node_1, node_2)`: This function finds the paths from the root to both `node_1` and `node_2`, storing the common ancestors in a separate array.

- `findPath(root, node, path)`: This function searches the BST for a specific node while recording all encountered nodes in the `path` array.

In summary, these functions work together to identify and correct the swapped elements in a BST while also determining the common ancestors of the nodes.

Pseudocode:

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function to swap two values
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// Variables to store the values of swapped nodes
int node_1 = 0;
int node_2 = 0;

// Function to recover the BST
void recoverTree(TreeNode* root) {
    TreeNode* first = NULL;
    TreeNode* middle = NULL;
    TreeNode* last = NULL;
    TreeNode* prev = NULL;

    // Helper function to find swapped elements and update first, middle, and last pointers
    function<void(TreeNode*)> inorderAndFindElements = [&](TreeNode* root) {
        if (!root) return;

        inorderAndFindElements(root->left);

        if (prev != NULL && root->val < prev->val) {
            if (first == NULL) {
                first = prev;
                middle = root;
            } else {
                last = root;
            }
        }
        prev = root;

        inorderAndFindElements(root->right);
    };

    inorderAndFindElements(root);

    // Swapping the nodes based on the identified cases
    if (first && last) {
        swap(first->val, last->val);
        node_1 = first->val;
        node_2 = last->val;
    } else if (first && middle) {
        swap(first->val, middle->val);
        node_1 = first->val;
        node_2 = middle->val;
    }
}

// Function to find the path from root to a node and store it in the path array
bool findPath(TreeNode* root, int node, vector<int>& path) {
    if (!root) return false;

    path.push_back(root->val);

    if (root->val == node) return true;

    if (node < root->val && findPath(root->left, node, path)) return true;
    if (node > root->val && findPath(root->right, node, path)) return true;

    path.pop_back();
    return false;
}

// Function to find common ancestors of two nodes
vector<int> findCommonAncestors(TreeNode* root, int node1, int node2) {
    vector<int> path1, path2;
    vector<int> commonAncestors;

    findPath(root, node1, path1);
    findPath(root, node2, path2);

    int n = min(path1.size(), path2.size());

    for (int i = 0; i < n; ++i) {
        if (path1[i] == path2[i]) {
            commonAncestors.push_back(path1[i]);
        } else {
            break;
        }
    }

    return commonAncestors;
}
```

Ans 3(b)

Approach:

1. Utilising Inorder Traversal:

To ensure that the elements are in sorted order, we leverage the property of Binary Search Trees (BST) where an inorder traversal naturally yields elements in ascending order. Thus, we perform an inorder traversal of the BST to gather the values of its nodes and store them in an array.

2. Sorting the Array:

Following the collection of values, the next step involves sorting them. We accomplish this by creating a new array and arranging the collected values in ascending order within this array.

3. Identifying Misplaced Elements:

With two arrays at our disposal – one containing the original values and the other with the sorted values – we proceed to pinpoint elements that are not positioned correctly. This task is achieved by comparing the elements in the two arrays.

4. Sorting the Array - Two Scenarios:

We encounter two scenarios for sorting the array:

Case 1 ($G < n * \log(n)$):

- In instances where G (the range of values) is smaller than n times the logarithm of n, we deploy Counting Sort. This sorting method exhibits linear time complexity.
- The process begins by creating an auxiliary array, known as `count`, sized to accommodate G unique elements. This array serves the purpose of counting the occurrences of each unique element in the input array. Remarkably, this

phase only processes each element once, resulting in a time complexity of O(n).

- Additionally, we craft a `position` array that keeps track of the positions of elements in the sorted sequence. This phase involves iterating through the `count` array, which bears a size of G. Consequently, the time complexity for this step is O(G).
- Concluding this approach, we generate an output array matching the size of the input array. Beginning from the end of the input array and proceeding backward, we locate each element's rightful place in the sorted order using the `position` array. Subsequently, we decrement the count for that particular element and correctly position it in the output array. Yet again, this phase processes each element precisely once and thereby incurs a time complexity of O(n).

To summarise, the overall time complexity of this sorting algorithm is determined by the sum of the time complexities associated with these three phases: O(n) + O(G) + O(n). Consequently, the overall time complexity is O(n+G).

Case 2 (G > n * log(n)):

- In scenarios where G significantly surpasses n times the logarithm of n, we resort to a sorting method that has a time complexity of n * log(n).

Thus, the final time complexity is the minimum between G + n and n * log(n), ensuring efficiency regardless of the range of values G.

Pseudocode:

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Function for inorder traversal of BST
void inorder(TreeNode* root, vector<int>& arr) {
    if (!root) return;

    inorder(root->left, arr);
    arr.push_back(root->val);
    inorder(root->right, arr);
}

// Case 1: G > n * log(n)
vector<int> findSwappedNodes_1(TreeNode* root, int n) {
    vector<int> arr, arr_sorted, rearranged;
    int num = 0;

    inorder(root, arr);
    arr_sorted = arr; // Copy the original array for sorting

    sort(arr_sorted.begin(), arr_sorted.end());

    for (int i = 0; i < n; ++i) {
        if (arr[i] != arr_sorted[i]) {
            rearranged.push_back(arr_sorted[i]);
            num++;
        }
    }

    return rearranged;
}

// Case 2: G < n * log(n)
vector<int> findSwappedNodes(TreeNode* root, int n, int G) {
    vector<int> arr, arr_sorted, rearranged;
    vector<int> count(G + 1, 0); // Array to store count of elements
    vector<int> position(G + 1, 0); // Array to store position of elements if occurred in sorted
order

    inorder(root, arr);

    for (int i = 0; i < n; ++i) {
        count[arr[i]]++;
    }

    position[0] = count[0];

    for (int i = 1; i <= G; ++i) {
        position[i] = count[i] + position[i - 1];
    }

    arr_sorted.resize(n);

    for (int i = n - 1; i >= 0; --i) {
        int element = arr[i];
        int pos = position[element] - 1;
        arr_sorted[pos] = element;
    }

    int num = 0;

    for (int i = 0; i < n; ++i) {
        if (arr[i] != arr_sorted[i]) {
            rearranged.push_back(arr_sorted[i]);
            num++;
        }
    }

    return rearranged;
}
```

4 Helping Joker

(A) First query the top card and then the last one. Now use binary search to get an index where change (decrease in number on card) occurs and call it a pivot. We know that all elements from pivot to last are smaller than all elements before pivot.

Pseudo code :-

```
if(first < last) return 0; // i.e. no shuffle
int lo=1,hi=n;
int pivot = (lo+hi)/2;

While(lo < hi){
        If(query(pivot) >= first) lo = pivot +1;
        Else hi=pivot;
        pivot= (lo+hi)/2;
}
Return n-pivot+1; // k= n-pivot+1
```

- If current card value is greater than the first, then the pivot is definitely ahead of it; otherwise, the current element can be a pivot, or there can be an element before it that is the pivot.

-*query()* tells the number on given indexed card from top

(B) Time complexity - O(log(n)): Assume that the *query()* works in O(1). In each step of while loop, it discards half the search space, so it will stop after at O(log(n)) steps and rest steps works in O(1).

5 One Piece Treasure

We are given that *substring(i,j)* tells in O(1) that sub-string from index i to j is palindrome or not.

Fact: A palindrome is symmetric about its middle and if we remove first and last element of a palindrome then it also remains a palindrome.

Approach: Calculating biggest odd sized palindrome if current index is middle element using binary search and similarly even sized palindrome if current index is just left to symmetry line using binary search.

pseudocode:

```
int findpalin(string &s){
        int n=s.size();

        int odd=0,even=0;
        for(int i=0;i<n;i++){
                odd+=(maxoddpal(s,i)+1);
                even+=maxevenpal(s,i);
        }
        return odd+even;
}
```

-*maxoddpal(s,i)* makes queries using *substring(,)*, for biggest odd sized palindrome centred about index i using binary search upon possible half size of palindrome

-I added 1 together in odd because a string of length 1 is trivially a palindrome.

-*maxevenpal(s,i)* makes queries using *substring(,)*, for biggest even sized palindrome centred about index i using binary search upon possible half size of palindrome

- Directly added maximum possible size of palindrome because that many palindromes will be centred from that index.

Time Complexity: O(N*log(N)), as *findpalin(s)* runs a loop of n steps each having two function calls of binary search so O(log (N)) in each step, so overall O(N*log (N)), not that this is using assumption that that *substring(i,j)* takes O(1).