Transform and Conquer

Presorting

Q2.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int N;
        cin >> N;
        vector<int> A(N);
        for (int i = 0; i < N; ++i) {
            cin >> A[i];
        }

        unordered_set<int> seen;
        bool isUnique = true;

        for (int num : A) {
            if (seen.find(num) != seen.end()) {
                isUnique = false;
                break;
            }
            seen.insert(num);
        }

        if (isUnique) {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

Q4.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int N;
```

```
        cin >> N;
        vector<int> A(N);
        for (int i = 0; i < N; ++i) {
            cin >> A[i];
        }

        bool isUnique = true;
        for (int i = 0; i < N; ++i) {
            for (int j = i + 1; j < N; ++j) {
                if (A[i] == A[j]) {
                    isUnique = false;
                    break;
                }
            }
            if (!isUnique) break;
        }

        if (isUnique) {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

Q5.
O(N^2) and O(N log N)
Q6.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int N;
        cin >> N;
        vector<int> arr(N);
        for (int i = 0; i < N; i++) {
            cin >> arr[i];
        }
        sort(arr.begin(), arr.end());
        int maxCount = 0, mode = arr[0];
        int count = 1;
        for (int i = 1; i < N; i++) {
            if (arr[i] == arr[i - 1]) {
                count++;
```

```
            } else {
                if (count > maxCount) {
                    maxCount = count;
                    mode = arr[i - 1];
                }
                count = 1;
            }
        }
        if (count > maxCount) {
            maxCount = count;
            mode = arr[N - 1];
        }
        cout << mode << endl;
    }
    return 0;
}
```

Q7.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false); // Disable synchronization for faster I/O
    cin.tie(NULL); // Untie cin and cout

    int T;
    cin >> T;
    while (T--) {
        int N, K, L;
        cin >> N >> K >> L;
        vector<int> A(N);
        for (int i = 0; i < N; ++i) {
            cin >> A[i];
        }

        // Sort the array in descending order
        sort(A.rbegin(), A.rend());

        long long totalDeliciousness = 0;
        // Calculate the positions you will be picking
        for (int i = L - 1; i < N; i += K) {
            totalDeliciousness += A[i];
        }

        cout << totalDeliciousness << endl;
    }

    return 0;
}
```

Balanced Binary Search Tees

Q3.
1
2
4
Q4.

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int val;
    Node* left;
    Node* right;

    Node() {
        val = 0;
        left = NULL;
        right = NULL;
    }
};

bool Search(Node* root, int x) {
    if(root == NULL) {
        return false;
    }
    if(root -> val == x) {
        return true;
    }
    if(root -> val < x) {
        return Search(root -> right, x);
    }
    return Search(root -> left, x);
}

void Run(Node *root) {
    cout << Search(root, 35) << "\n";
    cout << Search(root, 25) << "\n";
    cout << Search(root, 1) << "\n";
    cout << Search(root, 20) << "\n";
}

int main() {
    Node* root = new Node;
    root -> val = 10;

    Node* child1 = new Node;
    child1 -> val = 5;
```

```
    Node* child2 = new Node;
    child2 -> val = 30;

    root -> left = child1;
    root -> right = child2;

    Node *child3 = new Node;
    child3 -> val = 40;

    Node *child4 = new Node;
    child4 -> val = 25;

    child2 -> left = child4;
    child2 -> right = child3;

    Node *child5 = new Node;
    child5 -> val = 35;

    child3 -> left = child5;

    Node* child6 = new Node;
    child6 -> val = 1;
    child1 -> left = child6;

    Run(root);

    return 0;
}
```

Q5.

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int val;
    Node* left;
    Node* right;

    Node() {
        val = 0;
        left = NULL;
        right = NULL;
    }
};

Node* Insert(Node* root, int x) {
    if(root == NULL) {
        // At this place a node with value x should exist, but it doesn't
        Node* temp = new Node();
        temp -> val = x;
```

```
        return temp; // Return the newly created node
    }

    if(root -> val == x) {
        // Value already exists
        // No insertion, return the root
        return root;
    }

    if(root -> val < x) {
        // Insert the node into the right subtree
        root -> right = Insert(root -> right, x);

        return root;
    }

     // Insert the node into the left subtree
    root -> left = Insert(root -> left, x);
    return root;
}

bool Search(Node* root, int x) {
    if(root == NULL) { // If the node does not exist then return false
        return false;
    }
    if(root -> val == x) {
        return true;
    }
    if(root -> val < x) {
        return Search(root -> right, x);
    }
    return Search(root -> left, x);
}


int main() {
    int n; cin >> n;
    int a[n];
    for(int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // Creating an empty tree
    Node* root = NULL;

    for(int i = 0; i < n; i++) {
        root = Insert(root, a[i]);
    }
```

```
    for(int i = 0; i < n; i++) {
        cout << Search(root, a[i]) << "\n";
    }

    return 0;
}
```

AVL trees

Q2.
1
3

Q5.

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int val;
    Node* left;
    Node* right;
    int height; // Store height to get balance nature

    Node() {
        val = 0;
        height = 1; // Every node has height = 1 initially
        left = NULL;
        right = NULL;
    }
};

int get_height(Node* node) {
    if(node == NULL) {
        return 0; // Height of a NULL node is 0
    }
    return node -> height;
}

int calculate_height(Node* node) {
    return 1 + max(get_height(node -> left), get_height(node -> right));
}

Node* right_rotate(Node* node) {
    Node* temp = node -> left;
    Node* temp2 = node -> left -> right;

    // Rotate the nodes
```

```
    temp -> right = node;
    node -> left = temp2;

    // Recalculate height
    node -> height = calculate_height(node);
    temp -> height = calculate_height(temp);

    return temp;
}

Node* left_rotate(Node* node) {
    Node* temp = node -> right;
    Node* temp2 = node -> right -> left;

    // Rotate the nodes
    temp -> left = node;
    node -> right = temp2;

    // Recalculate height
    node -> height = calculate_height(node);
    temp -> height = calculate_height(temp);

    return temp;
}

int Height_difference(Node* node) {
    return get_height(node -> left) - get_height(node -> right);
}

Node* Insert(Node* root, int x) {
    if(root == NULL) {
        // At this place a node with value x should exist, but it doesn't
        Node* temp = new Node;
        temp -> val = x;

        return temp; // Return the newly created node
    }
    if(root -> val < x) {
        // Insert the node into the right subtree
        root -> right = Insert(root -> right, x);
    }
    if(root -> val > x) {
         // Insert the node into the left subtree
        root -> left = Insert(root -> left, x);
    }

    int height_difference = Height_difference(root);
```

```
    // Left Left case
    if(height_difference > 1 && x < root -> left -> val) {
        return right_rotate(root);
    }

    // Left Right case
    if(height_difference > 1 && x > root -> left -> val) {
        root -> left = left_rotate(root -> left);  // Convert into Left Left case
        return right_rotate(root);
    }

    // Right Right case
    if(height_difference < -1 && x > root -> right -> val) {
        return left_rotate(root);
    }

    // Right Left case
    if(height_difference < -1 && x < root -> right -> val) {
        root -> right = left_rotate(root -> right);  // Convert into Right Right case
        return right_rotate(root);
    }

    // Return if the tree is already balanced
    return root;
}

bool Search(Node* root, int x) {
    if(root == NULL) { // If the node does not exist then return false
        return false;
    }
    if(root -> val == x) {
        return true;
    }
    if(root -> val < x) {
        return Search(root -> right, x);
    }
    return Search(root -> left, x);
}

int main() {
    int q; cin >> q;
    Node* root = NULL;

    for(int i = 0; i < q; i++) {
        int type, id;
        cin >> type >> id;

        if(type == 1) {
```

```
            root = Insert(root, id);
        } else {
            if(Search(root, id)) {
                cout << "YES" << "\n";
            } else {
                cout << "NO" << "\n";
            }
        }
    }

    return 0;
}
```

Q6.

```
#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false); // Faster I/O
    cin.tie(NULL); // Untie cin from cout

    int Q;
    cin >> Q;

    unordered_map<int, string> name_map;

    for (int i = 0; i < Q; ++i) {
        int query_type;
        cin >> query_type;

        if (query_type == 1) {
            int id;
            string name;
            cin >> id >> name;
            name_map[id] = name;
        } else if (query_type == 2) {
            int id;
            cin >> id;
            if (name_map.find(id) != name_map.end()) {
                cout << name_map[id] << endl;
            } else {
                cout << -1 << endl;
            }
        }
    }

    return 0;
```

```
}
```

Heap and Heap Sort

Q2.
1
4
Q6.

```
#include<bits/stdc++.h>
using namespace std;

struct MinHeap{
    int* arr;
    int last_index;

    MinHeap(int n) {
        arr = new int[n + 1];
        last_index = 1;
    }

    void insert(int value) {
        int idx = last_index;
        arr[last_index++] = value;
        while(idx > 1) {
            if(arr[idx / 2] > arr[idx]) {
                swap(arr[idx / 2], arr[idx]);
                idx = idx / 2 ;
            }
            else {
                break;
            }
        }
    }

    int top() {
        if(last_index == 1) {
            return -1;
        }
        return arr[1];
    }

    void pop() {
        if(last_index == 1) {
            cout << "Heap Empty, cannot pop\n";
        }
```

```
        arr[1] = arr[--last_index];
        int idx = 1;
        while(idx <= last_index) {
            int left = 2 * idx;
            int right =  2 * idx + 1;
            int smallest = idx;
            if(left <= last_index && arr[left] < arr[smallest])
                smallest = left;
            if(right <= last_index && arr[right] < arr[smallest])
                smallest = right;
            if(smallest == idx) {
                break;
            }
            swap(arr[smallest], arr[idx]);
            idx = smallest;
        }
    }
};

// Replace '_' to solve the problem
void Heapsort(int arr[], int n) {
    MinHeap hp(n);
    for(int i = 0; i < n; i++) {
        hp.insert(arr[i]);
    }
    for(int i = 0; i < n; i++) {
        arr[i] = hp.top();
        hp.pop();
    }
}

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

    Heapsort(arr, n);

    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n";
}
```

Q8.

```cpp
#include <bits/stdc++.h>
using namespace std;

int minMergeCost(vector<int> &A) {
    priority_queue<int, vector<int>, greater<int>> minHeap(A.begin(), A.end());
    int totalCost = 0;

    while (minHeap.size() > 1) {
        int first = minHeap.top();
        minHeap.pop();
        int second = minHeap.top();
        minHeap.pop();

        int mergedLength = first + second;
        totalCost += mergedLength;

        minHeap.push(mergedLength);
    }

    return totalCost;
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        int N;
        cin >> N;
        vector<int> A(N);
        for (int i = 0; i < N; ++i) {
            cin >> A[i];
        }
        cout << minMergeCost(A) << endl;
    }
    return 0;
}
```