Backtracking

n-Queens problem

Q1.

Click submit

Q2.

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

// Function to check if placing a queen at board[row][col] is safe
bool isSafe(vector<string>& board, int row, int col, int n) {
    // Check this row on the left side
    for (int i = 0; i < col; i++)
        if (board[row][i] == 'Q')
            return false;

    // Check upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q')
            return false;

    // Check lower diagonal on the left side
    for (int i = row, j = col; i < n && j >= 0; i++, j--)
        if (board[i][j] == 'Q')
            return false;

    return true;
}

// Recursive function to solve N-Queens problem
void solveNQueens(int col, int n, vector<string>& board, vector<vector<string>>& solutions)
{
    // If all queens are placed
    if (col == n) {
        solutions.push_back(board);
        return;
    }

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            // Place this queen in board[i][col]
            board[i][col] = 'Q';
            // Recur to place the rest of the queens
            solveNQueens(col + 1, n, board, solutions);

            // If placing queen in board[i][col] doesn't lead to a solution, then backtrack
            board[i][col] = '.';
        }
    }
}

vector<vector<string>> n_queens(int n) {
    vector<vector<string>> solutions;
    vector<string> board(n, string(n, '.'));
    solveNQueens(0, n, board, solutions);
    return solutions;
}

int main() {
    int n;
    cin >> n;
    vector<vector<string>> solutions = n_queens(n);
    sort(solutions.begin(), solutions.end());

    for (const auto& solution : solutions) {
        for (const auto& row : solution) {
            cout << row << "\n";
        }
        cout << "\n";
    }

    return 0;
}
```

Hamiltonian Circuit Problem

Q2.
Any two adjacent vertices in the path must in adjacent in the graph.
No vertex should be visited more than once (except the starting vertex).
Starting and ending vertex should be the same.
Q3.

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

// Check if 'next' vertex can be added after vertex 'v'
bool check(int v, int next, vector<int> &circuit, vector<vector<bool>> &mat) {
    // Ensure the next vertex is adjacent to the current vertex
    if (!mat[v][next]) {
        return false;
    }
    // Ensure the next vertex has not been visited
    for (int vertex : circuit) {
        if (vertex == next) {
            return false;
        }
    }
    return true;
}

void backtrack(int v, int n, vector<int> &circuit, vector<vector<int>> &ans,
vector<vector<bool>> &mat) {
    if (circuit.size() == n) { // Circuit is completed
        if (mat[circuit[0]][v]) { // Check if cycle is completed, starting should be adjacent to
ending
            circuit.push_back(circuit[0]);
            ans.push_back(circuit);
            circuit.pop_back();
        }
        return;
    }

    for (int i = 1; i <= n; i++) {
        // Calling the check function
        if (!check(v, i, circuit, mat))
            continue;

        // If not visited and is adjacent, add it to our candidate solution
        circuit.push_back(i);
        backtrack(i, n, circuit, ans, mat);
        circuit.pop_back();
    }
}

// Number of vertices and adjacency matrix
vector<vector<int>> hamiltonian_circuit(int n, vector<vector<bool>> &mat) {
    vector<int> circuit; // Initially empty circuit
    vector<vector<int>> ans; // To store all the circuits

    for (int i = 1; i <= n; i++) { // Fix the starting vertex
        circuit.push_back(i); // Add i to circuit
        backtrack(i, n, circuit, ans, mat);
        circuit.pop_back(); // Remove i from circuit
    }

    return ans;
}
```

```
int main() {
    int n;
    cin >> n;
    int m;
    cin >> m;

    // Adjacency matrix
    vector<vector<bool>> mat(n + 1, vector<bool> (n + 1, false));
    while (m--) {
        int a, b;
        cin >> a >> b;
        mat[a][b] = 1;
        mat[b][a] = 1;
    }

    vector<vector<int>> cycles = hamiltonian_circuit(n, mat);
    sort(cycles.begin(), cycles.end());

    for (auto &x : cycles) {
        for (auto &y : x) {
            cout << y << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Q4.

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

// Check if 'next' vertex can be added after vertex 'v'
bool check(int v, int next, vector<int> &circuit, vector<vector<bool>> &mat) {
    // Ensure the next vertex is adjacent to the current vertex
    if (!mat[v][next]) {
        return false;
    }
    // Ensure the next vertex has not been visited
    for (int vertex : circuit) {
        if (vertex == next) {
            return false;
        }
    }
    return true;
}
```

```
void backtrack(int v, int n, vector<int> &circuit, vector<vector<int>> &ans,
vector<vector<bool>> &mat) {
    if (circuit.size() == n) { // Circuit is completed
        if (mat[circuit[0]][v]) { // Check if cycle is completed, starting should be adjacent to
ending
            circuit.push_back(circuit[0]);
            ans.push_back(circuit);
            circuit.pop_back();
        }
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (!check(v, i, circuit, mat))
            continue;

        // If not visited and is adjacent, add it to our candidate solution
        circuit.push_back(i);
        backtrack(i, n, circuit, ans, mat);
        circuit.pop_back();
    }
}

// Number of vertices and adjacency matrix
vector<vector<int>> hamiltonian_circuit(int n, vector<vector<bool>> &mat) {
    vector<int> circuit; // Initially empty circuit
    vector<vector<int>> ans; // To store all the circuits

    for (int i = 1; i <= n; i++) { // Fix the starting vertex
        circuit.push_back(i); // Add i to circuit
        backtrack(i, n, circuit, ans, mat);
        circuit.pop_back(); // Remove i from circuit
    }

    return ans;
}

int main() {
    int n, m;
    cin >> n >> m;

    // Adjacency matrix
    vector<vector<bool>> mat(n + 1, vector<bool>(n + 1, false));
    while (m--) {
        int a, b;
        cin >> a >> b;
        mat[a][b] = true;
        mat[b][a] = true;
    }

    vector<vector<int>> cycles = hamiltonian_circuit(n, mat);
    sort(cycles.begin(), cycles.end());

    for (auto &x : cycles) {
        for (auto &y : x) {
            cout << y << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Subset Sum Problem

Q2.
The sum of integers in the subset must be X.

Q3.

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

void backtrack(int idx, vector<int> &a, int x, vector<int> &subset, vector<vector<int>> &ans) {
    if (idx == a.size()) {
        // Check if the sum of the subset is equal to x
        int sum = accumulate(subset.begin(), subset.end(), 0);
        if (sum == x) {
            ans.push_back(subset);
        }
        return;
    }

    // Don't take the i-th integer
    backtrack(idx + 1, a, x, subset, ans);

    // Take the i-th integer
    subset.push_back(a[idx]);
    backtrack(idx + 1, a, x, subset, ans);
    subset.pop_back();
}

vector<vector<int>> subset_sum(vector<int> a, int x) {
    vector<int> subset; // Creating an empty subset
    vector<vector<int>> ans; // To store all the subsets

    backtrack(0, a, x, subset, ans);
    return ans;
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (auto &elem : a) cin >> elem;

    vector<vector<int>> subsets = subset_sum(a, x);
    for (auto &subset : subsets) sort(subset.begin(), subset.end());
    sort(subsets.begin(), subsets.end());

    for (auto &subset : subsets) {
        for (auto &elem : subset) {
            cout << elem << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Q4.

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

void backtrack(int idx, vector<int> &a, int x, vector<int> &subset, vector<vector<int>> &ans) {
    if (idx == a.size()) {
        int sum = accumulate(subset.begin(), subset.end(), 0);
        if (sum == x) {
            ans.push_back(subset);
        }
        return;
    }

    // Don't include the current element in the subset
    backtrack(idx + 1, a, x, subset, ans);

    // Include the current element in the subset
    subset.push_back(a[idx]);
    backtrack(idx + 1, a, x, subset, ans);
    subset.pop_back();
}

vector<vector<int>> subset_sum(vector<int> a, int x) {
    vector<int> subset; // Creating an empty subset
    vector<vector<int>> ans; // To store all the subsets

    backtrack(0, a, x, subset, ans);

    return ans;
}

int main() {
    int n, x;
    cin >> n >> x;
    vector<int> a(n);
    for (auto &elem : a) cin >> elem;

    vector<vector<int>> subsets = subset_sum(a, x);
    for (auto &subset : subsets) sort(subset.begin(), subset.end());
    sort(subsets.begin(), subsets.end());

    for (auto &subset : subsets) {
        for (auto &elem : subset) {
            cout << elem << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

Q6.

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

// Function to generate permutations
void backtrack(vector<int>& nums, vector<vector<int>>& result, vector<int>& current,
vector<bool>& used) {
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }
    for (int i = 0; i < nums.size(); i++) {
        if (used[i] || (i > 0 && nums[i] == nums[i - 1] && !used[i - 1])) {
            continue;
        }
        used[i] = true;
        current.push_back(nums[i]);
        backtrack(nums, result, current, used);
        used[i] = false;
        current.pop_back();
    }
}

// Function to find unique permutations
vector<vector<int>> uniquePermutations(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    vector<vector<int>> result;
    vector<int> current;
    vector<bool> used(nums.size(), false);
    backtrack(nums, result, current, used);
    return result;
}

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];
        }

        vector<vector<int>> permutations = uniquePermutations(A);

        cout << permutations.size() << endl;
        for (const auto& perm : permutations) {
            for (int num : perm) {
                cout << num << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```

Q7.

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

// Function to generate valid parentheses
void backtrack(int open, int close, string &current, vector<string> &result, int n) {
    if (current.size() == 2 * n) {
        result.push_back(current);
        return;
    }
    if (open < n) {
        current.push_back('(');
        backtrack(open + 1, close, current, result, n);
        current.pop_back();
    }
    if (close < open) {
        current.push_back(')');
        backtrack(open, close + 1, current, result, n);
        current.pop_back();
    }
}

// Function to find all valid parentheses strings
vector<string> generateParenthesis(int n) {
    vector<string> result;
    string current;
    backtrack(0, 0, current, result, n);
    sort(result.begin(), result.end()); // Ensure lexicographical order
    return result;
}

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

        vector<string> validParentheses = generateParenthesis(N);

        cout << validParentheses.size() << endl;
        for (const auto& s : validParentheses) {
            cout << s << endl;
        }
    }
    return 0;
}
```

Q8.

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

// Function to check if a string is a palindrome
bool isPalindrome(const string &s, int start, int end) {
    while (start < end) {
        if (s[start] != s[end])
            return false;
        start++;
        end--;
    }
    return true;
}

// Backtracking function to find all palindrome partitions
void backtrack(int start, string &s, vector<string> &currentPartition, vector<vector<string>>
&allPartitions) {
    if (start >= s.size()) {
        allPartitions.push_back(currentPartition);
        return;
    }
    for (int end = start; end < s.size(); end++) {
        if (isPalindrome(s, start, end)) {
            currentPartition.push_back(s.substr(start, end - start + 1));
            backtrack(end + 1, s, currentPartition, allPartitions);
            currentPartition.pop_back();
        }
    }
}

// Function to find all unique palindrome partitions
vector<vector<string>> palindromePartitioning(string s) {
    vector<vector<string>> allPartitions;
    vector<string> currentPartition;
    backtrack(0, s, currentPartition, allPartitions);
    sort(allPartitions.begin(), allPartitions.end());
    return allPartitions;
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        string S;
        cin >> S;

        vector<vector<string>> partitions = palindromePartitioning(S);

        cout << partitions.size() << endl;
        for (const auto &partition : partitions) {
            for (const auto &substring : partition) {
                cout << substring << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```