Question 10: Generate All Permutations of a String

Problem

Generate all permutations of a given string.

Algorithm 1: Backtracking

1. Create a recursive function that takes the current permutation state.
2. If the current permutation is complete (same length as input string), add it to result.
3. For each character in the string: a. If the character has already been used in current permutation, skip it. b. Include the character in current position and recurse for next position. c. Backtrack by removing the character before trying next possibility.

Program Implementation

cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void generatePermutationsHelper(string& s, int index, vector<string>& result) {
    if (index == s.length()) {
        result.push_back(s);
        return;
    }

    for (int i = index; i < s.length(); i++) {
        // Swap characters at positions index and i
        swap(s[index], s[i]);

        // Recurse for next position
        generatePermutationsHelper(s, index + 1, result);

        // Backtrack
        swap(s[index], s[i]);
    }
}

vector<string> generatePermutations(string s) {
    vector<string> result;
    generatePermutationsHelper(s, 0, result);
    return result;
}

int main() {
    string s = "abc";

    vector<string> permutations = generatePermutations(s);

    cout << "All permutations of \"" << s << "\":" << endl;
    for (const string& perm : permutations) {
        cout << perm << endl;
    }

    cout << "Total number of permutations: " << permutations.size() << endl;

    return 0;
}
```

Algorithm 2: Iterative Approach (Lexicographic Order)

1. Sort the string in ascending order.
2. Find permutations in lexicographical order: a. Find the largest index i such that s[i] < s[i+1]. b. Find the largest index j > i such that s[j] > s[i]. c. Swap s[i] and s[j]. d. Reverse the substring starting at s[i+1].
3. Repeat until no such i is found.

cpp

```cpp
vector<string> generatePermutationsIterative(string s) {
    vector<string> result;

    // Sort the string in ascending order
    sort(s.begin(), s.end());

    do {
        result.push_back(s);
    } while (next_permutation(s.begin(), s.end()));

    return result;
}

// Implementing next_permutation
bool nextPermutation(string& s) {
    int i = s.length() - 2;
    while (i >= 0 && s[i] >= s[i + 1]) {
        i--;
    }

    if (i < 0) return false;

    int j = s.length() - 1;
    while (s[j] <= s[i]) {
        j--;
    }

    swap(s[i], s[j]);
    reverse(s.begin() + i + 1, s.end());

    return true;
}
```

Time Complexity

- O(n * n!) where n is the length of the string

- We generate all n! permutations, and each permutation takes O(n) time to construct

Space Complexity

- O(n * n!) to store all the permutations
- O(n) call stack space for the recursive solution

Example Explanation

Let's trace through generating permutations for "abc":

1. Start with index = 0:
    - Swap a with a: "abc" -> Recurse for index 1
        - Swap b with b: "abc" -> Recurse for index 2
            - Swap c with c: "abc" -> Complete permutation: "abc"
        - Swap b with c: "acb" -> Recurse for index 2
            - Swap b with b: "acb" -> Complete permutation: "acb"
    - Swap a with b: "bac" -> Recurse for index 1
        - Swap a with a: "bac" -> Recurse for index 2
            - Swap c with c: "bac" -> Complete permutation: "bac"
        - Swap a with c: "bca" -> Recurse for index 2
            - Swap a with a: "bca" -> Complete permutation: "bca"
    - Swap a with c: "cba" -> Recurse for index 1
        - Swap b with b: "cba" -> Recurse for index 2
            - Swap a with a: "cba" -> Complete permutation: "cba"
        - Swap b with a: "cab" -> Recurse for index 2
            - Swap b with b: "cab" -> Complete permutation: "cab"

The algorithm generates all 6 permutations: "abc", "acb", "bac", "bca", "cba", "cab".