

Recursion 2



Gyanu Mayank

17th February, 2023



Recursion 2

Agenda

1. Problems Based on Recursion

Problems

1. Program to print all permutations of the given string

Given a string S, the task is to write a program to print all permutations of a given string

Example

Input: S = "ABC"

Output: "ABC", "ACB", "BAC", "BCA", "CBA", "CAB"

Approach-1 Using Backtracking

Backtracking is an algorithmic strategy for recursively solving problems by attempting to develop a solution gradually, one step at a time, and

About Bosscoder !

BossCoder Academy is a platform for ambitious engineers who want to upskill and achieve great heights in their careers.

Our world-class program provides the learners with:

✓ Structured curriculum designed by experts.

✓ Covers Data Structures & Algorithms, System Design & Project Development.

✓ Live Interactive Classes from Top PBC engineers.

✓ 1:1 Mentorship

✓ Quick Doubt Support

✓ Advanced Placement Support

Want to upskill to achieve your dream of working at

in time.

We'll define a function `generatePermutationsHelper(Str, l, r)`. This function will generate the permutations of the substring starting from index "l" and ending at index "r". Calling the above function, `generatePermutationsHelper(Str, l, r)`.

If "l" is equal to "r", a new permutation is found. Insert this string in the "ans" list. Else, continue to iterate on the string from "l" to "r". Let "i" denote the current index. Swap `Str[l]` and `Str[i]` to fix the "ith" character on the index "l".

Call `generatePermutationsHelper(Str, l + 1, r)` to get the permutation of the rest of the characters. Now, backtrack and swap `Str[l]` and `Str[i]` again. In the end, we'll have the list "ans" having all the permutations of the given string. If we want the permutations in lexicographically increasing order, we have to sort the list.

Implementation

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

void generatePermutationsHelper(string &str
{

    if (l == r)
    {
        ans.push_back(str);
        return;
    }
```

[Learn now we can help.](#)

Complete Guide

to **Upskill yourself** for



Download PDF

1543+ downloads.

```

        swap(str[l], str[i]);
        generatePermutationsHelper(str, l + 1);
        swap(str[l], str[i]);
    }
}

int main()
{
    vector<string> ans;
    string str;
    cin >> str;

    int l = 0;
    int r = str.size() - 1;

    if (str.length() == 0)
    {
        cout << "No Permutations Possible!!"
    }
    else
        generatePermutationsHelper(str, l,
    sort(ans.begin(), ans.end());
    for (int i = 0; i < ans.size(); i++)
    {
        cout << ans[i] << endl;
    }
    return 0;
}

```

Approach-2 Avoid Repetition Using Backtracking

Create a recursive function and pass the input string and a string that stores the permutation

print the permutation. Otherwise, run a loop from $i = 0$ to N : Consider $S[i]$, to be a part of the permutation. Remove this from the current string and append it to the end of the permutation. Call the recursive function with the current string which does not contain $S[i]$ and the current permutation.

Implementation

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

void permute(string s, string answer)
{
    if (s.length() == 0) {
        cout << answer << endl;
        return;
    }
    for (int i = 0; i < s.length(); i++) {
        char ch = s[i];
        string left_substr = s.substr(0, i);
        string right_substr = s.substr(i + 1, s.length() - i - 1);
        string rest = left_substr + right_substr;
        permute(rest, answer + ch);
    }
}

int main()
{
    string s;
    cin>>s;
    string ans = "";
    if(s.length()==0)
    {
```

```

else
    permute(s, ans);
return 0;
}

```

Time Complexity: $O(N * N!)$ i.e. there are $N!$ permutations and it requires $O(N)$ time to print a permutation.

Space Complexity: $O(|S|)$

2. [All Permutation of given Array](#)

Given an array arr of distinct integers, print all permutations of Array.

Example

Input: $n=3$, arr = [1, 2, 3]

Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Approach

We have given the nums array, so we will declare an ans vector of vector that will store all the permutations and also declare a data structure. Declare a map and initialize it to zero and call the recursive function

Base condition: When the data structure's size is equal to n (size of nums array) then it is a permutation and stores that permutation in our ans, then returns it. Run a for loop starting from 0 to $\text{nums.size()} - 1$. Check if the frequency of i is unmarked, if it is unmarked then it means it has not been picked and then we pick. And make sure it is marked as picked. Call the recursion with the parameters to pick the other elements when we come back from the recursion make sure you throw that

Implementation

```
#include<bits/stdc++.h>

using namespace std;
class Solution {
private:
    void recurPermute(vector < int > & ds,
        if (ds.size() == nums.size()) {
            ans.push_back(ds);
            return;
        }
        for (int i = 0; i < nums.size(); i++)
            if (!freq[i]) {
                ds.push_back(nums[i]);
                freq[i] = 1;
                recurPermute(ds, nums, ans, freq)
                freq[i] = 0;
                ds.pop_back();
            }
    }
}

public:
    vector < vector < int >> permute(vector
        vector < vector < int >> ans;
        vector < int > ds;
        int freq[nums.size()];
        for (int i = 0; i < nums.size(); i++)
            recurPermute(ds, nums, ans, freq);
        return ans;
    }
};

int main() {
```

```

cin>>n;
vector<int> v(n);
for(int i=0;i<n;i++) cin>>v[i];
vector < vector < int >> sum = obj.permut
cout << "All Permutations are " << endl;
for (int i = 0; i < sum.size(); i++) {
    for (int j = 0; j < sum[i].size(); j++)
        cout << sum[i][j] << " ";
    cout << endl;
}
}

```

Optimised Approach

We have given the nums array, so we will declare an ans vector of vector that will store all the permutations. Call a recursive function that starts with zero, nums array, and ans vector. Declare a map and initialize it to zero and call the recursive function base condition: Whenever the index reaches the end take the nums array and put it in ans vector and return. Recursion: Go from index to $n - 1$ and swap. Once the swap has been done call recursion for the next state. After coming back from the recursion make sure you re-swap it because, for the next element, the swap will not take place.

```
#include<bits/stdc++.h>
```

```

using namespace std;
class Solution {
private:
    void recurPermute(int index, vector < i

```

```

        return;
    }
    for (int i = index; i < nums.size();
        swap(nums[index], nums[i]);
        recurPermute(index + 1, nums, ans);
        swap(nums[index], nums[i]);
    }
}
public:
    vector < vector < int >> permute(vector
        vector < vector < int >> ans;
        recurPermute(0, nums, ans);
        return ans;
    }
};

```

```

int main() {
    Solution obj;
    int n;
    cin>>n;
    vector<int> v(n);
    for(int i=0;i<n;i++) cin>>v[i];
    vector < vector < int >> sum = obj.permut
    cout << "All Permutations are" << endl;
    for (int i = 0; i < sum.size(); i++) {
        for (int j = 0; j < sum[i].size(); j++)
            cout << sum[i][j] << " ";
        cout << endl;
    }
}

```

Time Complexity: $O(N! \times N)$

Space Complexity: $O(1)$

(**candidates**) and a target number (**target**), find all unique combinations in **candidates** where the candidate numbers sum to **target**. Each number in **candidates** may only be used **once** in the combination.

Note: The solution set must not contain duplicate combinations.

Example

Input: n=7 candidates = [10,1,2,7,6,1,5], target = 8

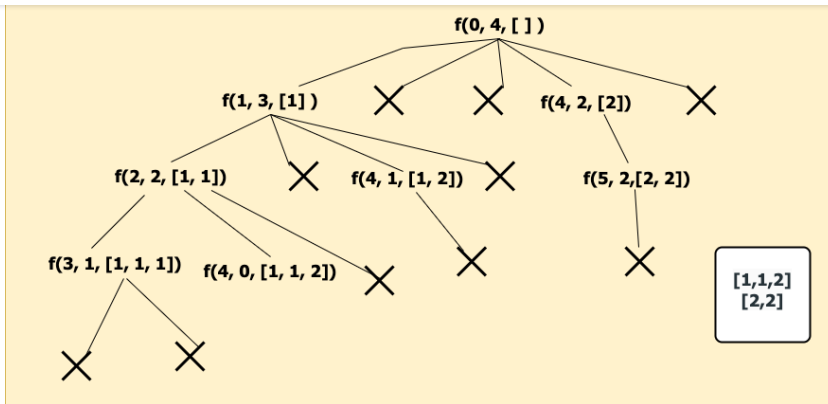
Output: [[1,1,6], [1,2,5], [1,7], [2,6]]

Approach

Before starting the recursive call make sure to sort the elements because the ans should contain the combinations in sorted order and should not be repeated. Initially, We start with the index 0, At index 0 we have n – 1 way to **pick the first element of our subsequence.**

Check if the current index value can be added to our ds. If yes add it to the ds and move the index by 1. while moving the index skip the consecutive repeated elements because they will form duplicate sequences. Reduce the target by arr[i], and call the recursive call for f(idx + 1, Target – 1, ds,ans) after the call make sure to pop the element from the ds. (By seeing the example recursive You will understand).if(arr[i] > target) then terminate the recursive call because there is no use to check as the array is sorted in the next recursive call the index will be moving by 1 all the elements to its **right will be in increasing order.**

Base Condition: Whenever the target value is zero add the ds to the ans return



If we observe the recursive call for $f(2, 2, [1, 1])$ when it is returning the ds doesn't include 1 so make sure to remove it from ds after the call.

Implementation

```
#include<bits/stdc++.h>
```

```
using namespace std;
void findCombination(int ind, int target, vector<int> &ds, vector<vector<int>> &ans) {
    if (target == 0) {
        ans.push_back(ds);
        return;
    }
    for (int i = ind; i < arr.size(); i++) {
        if (i > ind && arr[i] == arr[i - 1]) continue;
        if (arr[i] > target) break;
        ds.push_back(arr[i]);
        findCombination(i + 1, target - arr[i], ds, ans);
        ds.pop_back();
    }
}

vector<vector<int>> combinationSum2(vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
    vector<vector<int>> ans;
    vector<int> ds;
```

```

}

int main() {
    int n;
    cin>>n;
    int target;
    cin>>target;
    vector<int>v;
    for(int i=0;i<n;i++) cin>>v[i];
    vector < vector < int >> comb = combinati
    cout << "[ ";
    for (int i = 0; i < comb.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < comb[i].size(); j++)
            cout << comb[i][j] << " ";
        }
        cout << "]";
    }
    cout << " ]";
}

```

Time Complexity: $O(2^n \cdot k)$

Space Complexity: $O(k \cdot x)$

4. Combination Sum

Given an array of distinct integers **candidates** and a target integer **target**, return a list of all unique combinations of **candidates** where the chosen numbers sum to **target**. You may return the combinations in any order. The same number may be chosen from **candidates** an unlimited number of times. Two combinations are unique if the frequency of at

Input: n=4, candidates = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

Approach

Initially, the index will be 0, the target as given and the data structure(vector or list) will be empty now there are 2 options, to pick or not pick the current index element. If you pick the element, again coming back at the same index as multiple occurrences of the same element is possible so the target reduces to $\text{target} - \text{arr}[\text{index}]$ (where $\text{target} - \text{arr}[\text{index}] \geq 0$) and also inserts the current element into the data structure. If you decide not to pick the current element, move on to the next index and the target value stays as it is. Also, the current element is not inserted into the data structure. While backtracking makes sure to pop the last element as shown in the recursion tree below. Keep on repeating this process while indexing $<$ size of the array for a particular recursion call. You can also stop the recursion when the target value is 0, but here a generalized version without adding too many conditions is considered. Using this approach, we can get all the combinations. Base condition: If $\text{index} == \text{size of array}$ and $\text{target} == 0$ include the combination in our answer

Implementation

```
#include<bits/stdc++.h>
```

```
using namespace std;  
class Solution {
```

```

        if (ind == arr.size()) {
            if (target == 0) {
                ans.push_back(ds);
            }
            return;
        }

        if (arr[ind] <= target) {
            ds.push_back(arr[ind]);
            findCombination(ind, target - arr[ind], ans, ds);
            ds.pop_back();
        }

        findCombination(ind + 1, target, arr, ans, ds);
    }
public:
    vector < vector < int >> combinationSum(int target, vector<int> arr) {
        vector < vector < int >> ans;
        vector < int > ds;
        findCombination(0, target, candidates, arr, ans, ds);
        return ans;
    }
};

int main() {
    Solution obj;
    int n;
    cin>>n;
    int target;
    cin>>target;
    vector<int>v(n);
    for(int i=0;i<n;i++) cin>>v[i];
    vector < vector < int >> ans = obj.combinationSum(target, v);
    cout << "Combinations are: " << endl;
    for (int i = 0; i < ans.size(); i++) {

```

```

    cout << endl;
}
}

```

Time Complexity: $O(2^t * k)$ where t is the target, k is the average length

Space Complexity: $O(k * x)$, k is the average length and x is the no. of combinations

5. Number of ways to reach from source to destination with obstacles

Given a maze with obstacles, count the number of paths to reach the rightmost-bottommost cell from the topmost-leftmost cell. A cell in the given maze has a value of -1 if it is a blockage or dead-end, else 0.

From a given cell, we are allowed to move to cells $(i+1, j)$ and $(i, j+1)$ only.

Examples

Input: $\text{maze}[R][C] = \{\{0, 0, 0, 0\}, \{0, -1, 0, 0\}, \{-1, 0, 0, 0\}, \{0, 0, 0, 0\}\};$

Output: 4

Approach

Modify the given $\text{grid}[][]$ so that $\text{grid}[i][j]$ contains the count of paths to reach (i, j) from $(0, 0)$ if (i, j) is not a blockage, else $\text{grid}[i][j]$ remains -1. If the current cell is a blockage Do not change. If we can reach $\text{maze}[i][j]$ from $\text{maze}[i-1][j]$ then increment the count. If we can reach $\text{maze}[i][j]$ from $\text{maze}[i][j-1]$ then increment count.

Implementation

```
# define C 3

int countPaths(int maze[][C],int R)
{
    if (maze[0][0]==-1)
        return 0;

    for (int i=0; i<R; i++)
    {
        if (maze[i][0] == 0)
            maze[i][0] = 1;
        else
            break;
    }

    for (int i=1; i<C; i++)
    {
        if (maze[0][i] == 0)
            maze[0][i] = 1;
        else
            break;
    }

    for (int i=1; i<R; i++)
    {
        for (int j=1; j<C; j++)
        {
            if (maze[i][j] == -1)
                continue;
            if (maze[i-1][j] > 0)
                maze[i][j] = (maze[i][j] + maze[i-1][j]);

            if (maze[i][j-1] > 0)
                maze[i][j] = (maze[i][j] + maze[i][j-1]);
        }
    }
}
```

```
return (maze[R-1][C-1] > 0) ? maze[R-1][C-1]  
{
```

```
int main()  
{  
    int R;  
    cin>>R;  
    int maze[R][3];  
    for(int i=0;i<R;i++){  
        for(int j=0;j<3;j++){  
            cin>>maze[i][j];  
        }  
    }  
    cout << countPaths(maze,R);  
    return 0;  
}
```

Time Complexity : $O(R \times C)$

Space Complexity : $O(1)$

.



"BossCoder had everything I was looking for"

Udit Sharma
Software Developer

BossCoder had everything I was looking for

Date: 12th May, 2023

[Read](#)



"BossCoder helped me clear my basics of DSA and System Design"

Rachit Arora
Head of Engineering


BossCoder helped me clear my basics of DSA and System Design

Date: 5th May, 2023

[Read](#)

[View All blogs](#)



Helping ambitious learners upskill themselves & shift to top product






Who are we

About us
Blog
Attend a FREE Event
Privacy Policy
Terms & Condition
Pricing and Refund Policy

Contact Us

Email: ask@bosscoderacademy.com

Follow us on

 LinkedIn
 Youtube
 Instagram
 Telegram
 Reviews on Quora



UPSKILL WITH US

Recursion
all about
it.