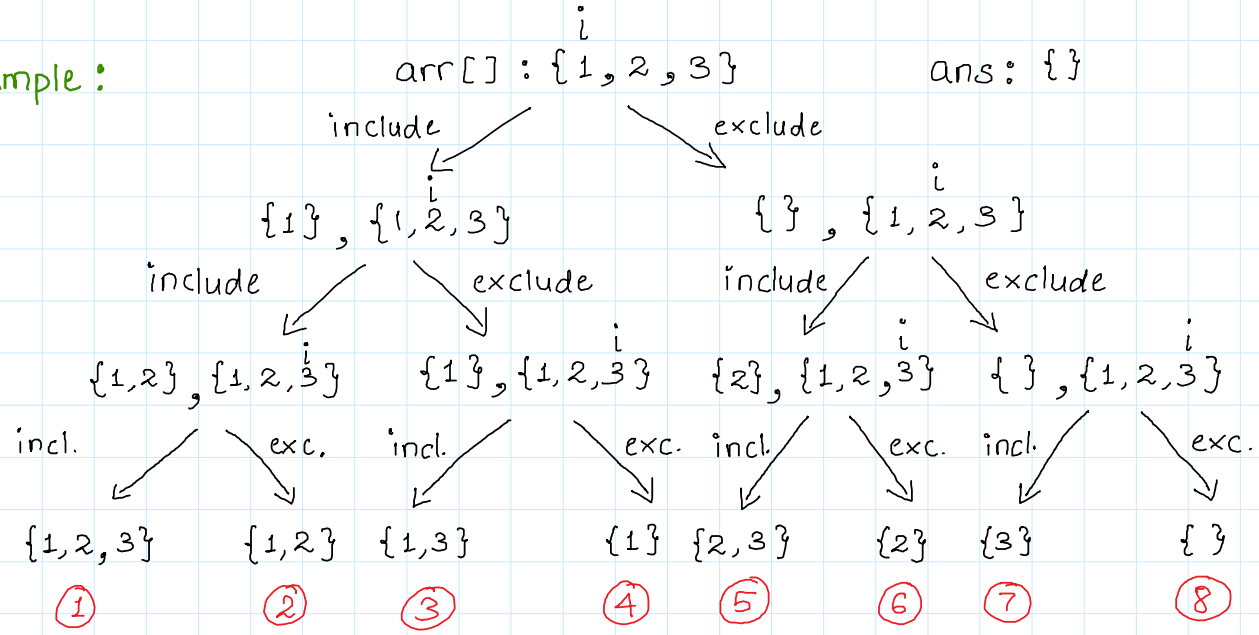
1. Generate all subsets (Power Set)

I/P : [1, 2, 3]
O/P : {{}, {1}, {2}, {3}, {1,2}, {2,3}, {1,3}, {1,2,3}}

Note - We can generate these subsets in any order as long as we cover all the subsets.

Approach : We can either include or exclude an element to generate a subset. Thus, we have 2 options for every element, either include it in the subset or exclude it.

Example :



What do you need ?

1. Array
2. Index
3. Answer array.

Code :

```
class Solution {
private:
    void solve(vector<int>& nums, vector<int> output, int i, vector<vector<int>> &ans) {
        if(i >= nums.size()) {
            ans.push_back(output);
            return;
        }
        // exclude this element and move on to the next index
        solve(nums, output, i+1, ans);
        // include this element
        output.push_back(nums[i]);
        solve(nums, output, i+1, ans);
    }
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> ans;
        vector<int> output;
        int ind = 0;
        solve(nums, output, ind, ans);
        return ans;
    }
};
```

Checking output array for each recursive call

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

void solve(vector<int>& nums, vector<int> output, int i, vector<vector<int>> &ans) {
    cout << "output: ";
    for(auto x: output) {
        cout << x << " ";
    }
    cout << endl;
    if (i == nums.size()) {
        ans.push_back(output);
        return;
    }
    // exclude this element and move on to the next index
    solve(nums, output, i + 1, ans);
    // include this element
    output.push_back(nums[i]);
    solve(nums, output, i + 1, ans);
}

vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> ans;
    vector<int> output;
    int ind = 0;
    solve(nums, output, ind, ans);
    return ans;
}
```

```
3
1 2 3
```

Output.txt

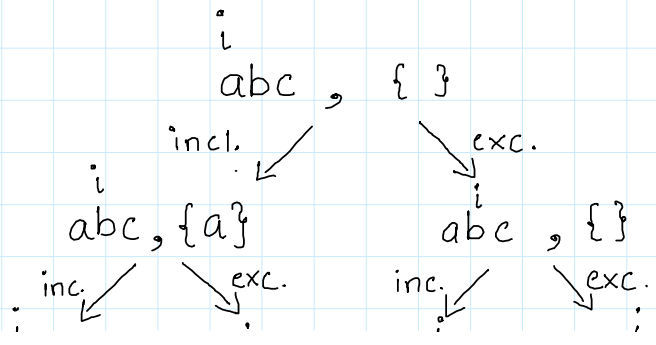
```
output:
output:
output:
output:
output: 3
output: 2
output: 2
output: 2 3
output: 1
output: 1
output: 1
output: 1 3
output: 1 2
output: 1 2
output: 1 2 3

3
2
2 3
1
1 3
1 2
1 2 3
```

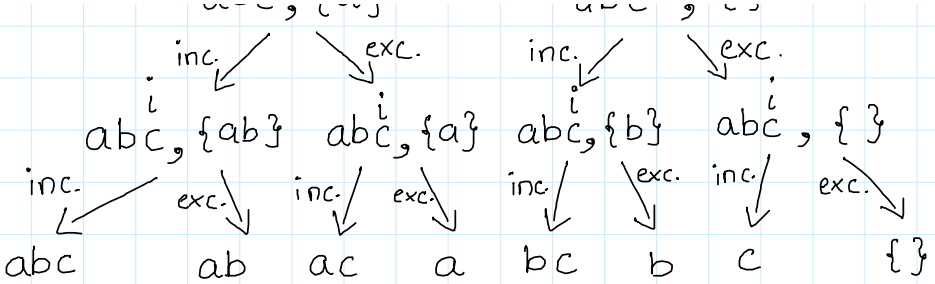
2. Subsequences of String. (Slight Change)

I/P : "abc"
O/P : {"", "a", "b", "c", "ab", "ac", "bc", "abc"}

Same Approach.

Recursion Tree :





Code :

```
void solve(vector<string>& ans, string str, string output, int i) {
    //base case
    if(i>=str.length()) {
        if(output.length()>0)
            ans.push_back(output);
        return;
    }

    //exclude
    solve(ans, str, output, i+1);
    //include
    output.push_back(str[i]);
    solve(ans, str, output, i+1);
}

vector<string> subsequences(string str){

    vector<string> ans;
    string output = "";
    solve(ans,str,output,0);
    return ans;

}
```

Note : In the question, we have to exclude the empty substring / subsequence.

Homework : Solve the above questions using bits.

Concept : If we have a string like "abc", then we know that 3-bit numbers 000, 001, ..., 111 represent all $2^3 = 8$ subsequences.

"abc" :

000	⇒	""
001	⇒	"c"
010	⇒	"b"
011	⇒	"bc"
100	⇒	"a"
101	⇒	"ac"

We 'mask' the binary number on the string.

Ex: "abc"
011

1 1 0 ⟹ "ab"

1 1 1 ⟹ "abc"

"bc"

So, for a string of length n, we will take an n-bit binary number and iterate through all the 2^n ranging from $\underbrace{000\ldots00}_{n}$ to $\underbrace{111\ldots11}_{n}$.

Code :

```
string maskedString(string str, int n) {
    string ans;
    int ind = str.length()-1;
    for(int j=0;j<str.length();j++) {
        if((1<<j) & n) {
            ans.push_back(str[ind]);
        }
        ind--;
    }
    return ans;
}

vector<string> subsequences(string str){

    vector<string> ans;
    for(int i=0;i<(1<<str.length());i++) {
        string temp = maskedString(str, i);
        if(!temp.empty())
            ans.push_back(temp);
    }
    return ans;
}
```

```
1 abc
```

Output.txt

```
1 c
2 b
3 cb
4 a
5 ca
6 ba
7 cba
8
```