DP and Bitmasking

Introduction

Bitmasking is when bits are used as a mask. To be more elaborate, when we use a collection of bits and in that collection the value of the bits and the different combinations that can be formed, are used to denote something relevant in a problem.

For example, in the problem to find the maximum sum of subset array bitmasking can be applied, where each subset could be represented as a combination of bits.

Suppose there is an array **Arr = [2, 5, 4, 3, 1]**. Now the subsets can be represented as :

0 0 0 0 0

0 0 0 0 1

0 0 0 1 0

0 0 0 1 1

.

.

.

1 1 1 1 1

Here, The length of the mask is equal to the length of the given array, and if the **i**$^{\textbf{th}}$ bit is set, that means the **i**$^{\textbf{th}}$ element of the array is included in the subset.

For an array of size N there can be $\mathbf{2^N}$ masks.

Solving subset sum using bitmasking:

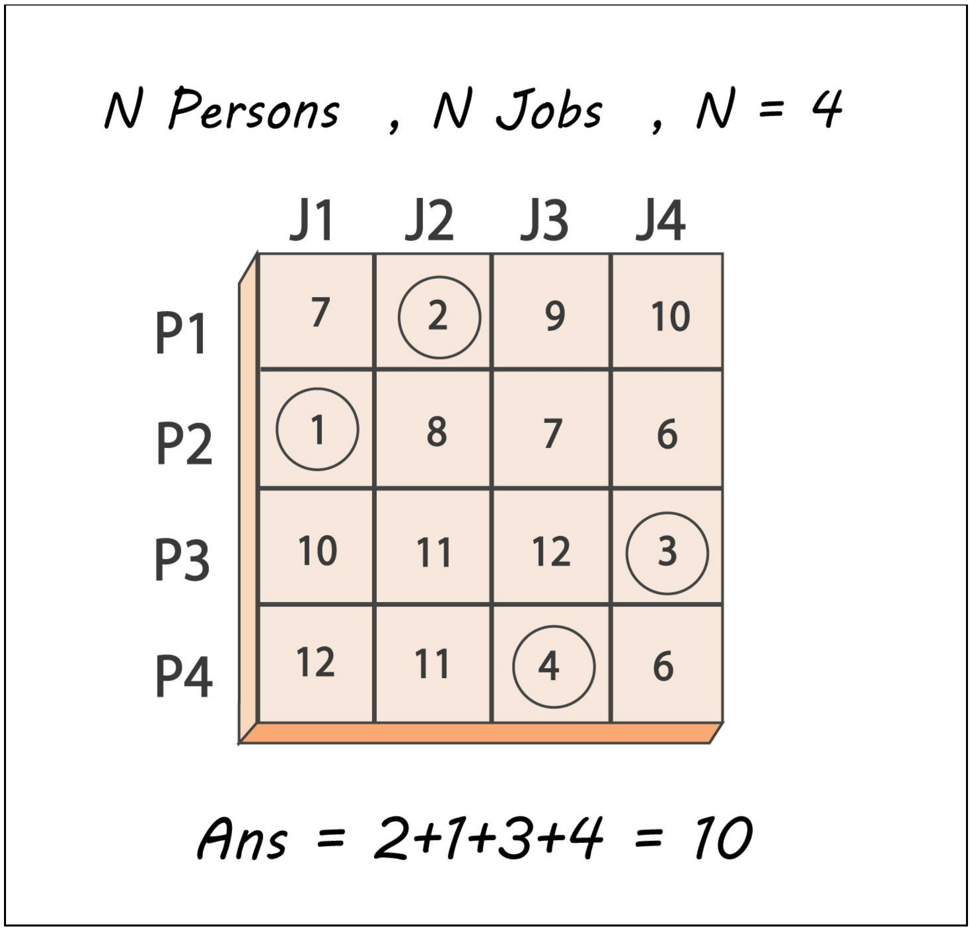
```
for mask = 0 to $2^N$-1 {
    sum = 0
    If the bits are set in the mask , then the corresponding  element of the array to be added in the sum. // O(N)
    If sum becomes equal to the given K, then countSubSet ++
}
```

The time complexity of the solution using bitmasking will be **O(N*2^N)**.

DP with Bitmasking and Minimum Cost for Jobs

Let us consider a problem in which we are given **N Persons** and **N Jobs**. There is a **N*N matrix** in which the row number gives the person and the column number gives the job. There are values assigned to the cells of the matrix that define how much it will cost to assign the job to a person. There can be only one job for one person and our **goal is to minimize the overall cost**.

Example:



N Persons , N Jobs , N = 4

$$
\begin{array}{c|cccc}
 & J1 & J2 & J3 & J4 \\
\hline
P1 & 7 & 2 & 9 & 10 \\
P2 & 1 & 8 & 7 & 6 \\
P3 & 10 & 11 & 12 & 3 \\
P4 & 12 & 11 & 4 & 6
\end{array}
$$

Ans = 2+1+3+4 = 10

Output: 10

Explanation: In the above example, if **Job 1** is assigned to **person 2**, **job 2** is assigned to **person 1**, **job 3** is assigned to **person 4** and **job 4** is assigned to **person 3**, then we get the minimum cost which is **10**.

Now the combinations of jobs and persons can be represented using bitmasking

For example,

- **1 0 0 0 means that P1 is assigned Job 1**
    - 1 1 0 0 means that P1 is assigned Job 1 and P2 is assigned Job 2
    - 1 0 1 0 means that P1 is assigned Job 1 and P2 is assigned Job 3
    - 1 0 0 1 means that P1 is assigned Job 1 and P2 is assigned Job 4

- **0 1 0 0 means that P1 is assigned Job 2**
    - Here, 1 1 0 0 means that P1 is assigned Job 2 and P2 is assigned Job 1
    - 0 1 1 0 means that P1 is assigned Job 2 and P2 is assigned Job 3
    - 0 1 0 1 means that P1 is assigned Job 2 and P2 is assigned Job 4

- **And so on ...**

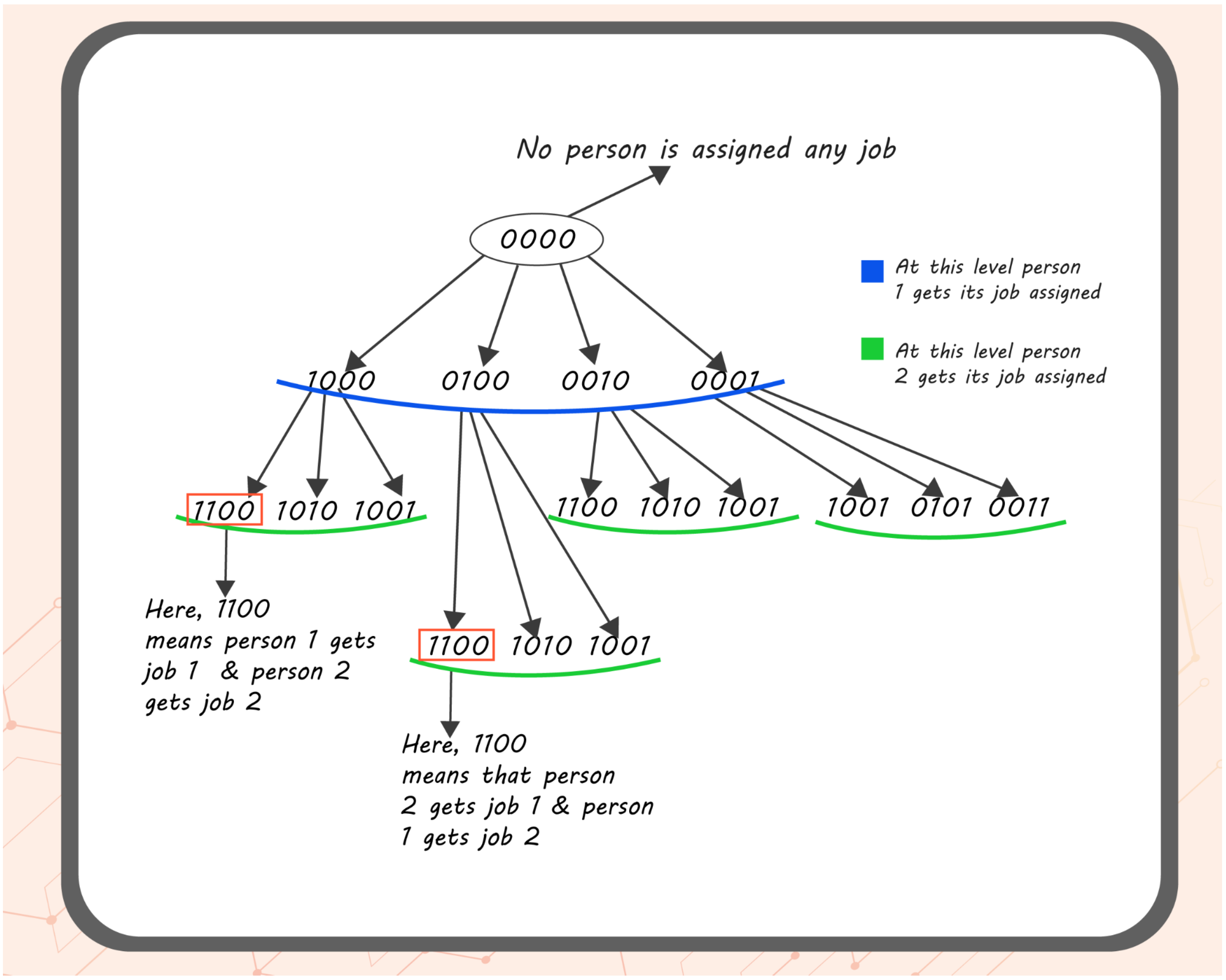
0000 means that no job is assigned to any person.

So we can call recursion with total four possibilities :

1. 1 0 0 0 : assigning P1 with Job 1
2. 0 1 0 0 : assigning P1 with Job 2
3. 0 0 1 0 : assigning P1 with Job 3
4. 0 0 0 1 : assigning P1 with Job 4

And further calling recursions on the above 4 possibilities.

Look at the image below for a better understanding


No person is assigned any job
0000
At this level person 1 gets its job assigned
At this level person 2 gets its job assigned
1000 0100 0010 0001
1100 1010 1001
1100 1010 1001
1001 0101 0011
Here, 1100 means person 1 gets job 1 & person 2 gets job 2
1100 1010 1001
Here, 1100 means that person 2 gets job 1 & person 1 gets job 2

If we look carefully at the masks we notice that some of them are the same, for example, **1 1 0 0**, However they may represent different answers. This is because of the order of the recursion call.

The first 1 1 0 0 means that Person 1 is assigned Job 1 and Person 2 is assigned Job 2, whereas the second 1 1 0 0 means that Person 1 is assigned Job 2 and Person 2 is assigned Job 1.

This basic approach is not very efficient as there are a lot of repetitions happening, so we use memoization for a more optimized solution.

Code using memoization:

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

// parameters of the function are:
//  1. cost matrix
//  2. number of persons and jobs
//  3. person which has to be assigned a job
//  4. mask (initially zero)
//  5. dp array
int minCost(int cost[4][4], int n, int p, int mask, int*dp){

    if(p>=n){
        return 0;
    }

    if(dp[mask] !=INT_MAX){
        return dp[mask];
    }

    int minimum = INT_MAX;
    for(int j=0; j<n; j++){
        // check if the bit is set in the mask or not
        if(!(mask&(1<<j))){
            // if the bit is not set then call minCost for the next person
            int ans = minCost(cost, n, p+1, mask | (1<<j), dp) +
cost[p][j];
            if(ans < minimum){
                minimum = ans;
            }
        }
    }
    dp[mask] = minimum;
    return minimum;
}


int main(){
    int cost [4][4] = {{10,2,6,5}, {1,15,12,8}, {7,8,9,3}, {15,13,4,10}};

    int* dp = new int[1<<4];
    for(int i=0; i<(1<<4); i++){
        dp[i] = INT_MAX;
    }

    cout << minCost(cost, 4, 0, 0, dp)<<endl;
    cout<<dp[0];
    return 0;
}
```

Now, try to code the iterative solution on your own.

Dilemma

Problem Statement: Abhishek, a blind man recently bought N binary strings all of equal length. A binary string only contains '0's and '1's . The strings are numbered from 1 to N and all are distinct, but Abhishek can only differentiate between these strings by touching them. In one touch Abhishek can identify one character at a position of a string from the set. Find the minimum number of touches T Abhishek has to make so that he learns that all strings are different .

Explanation:

Let's take an example to understand what exactly touching a string means,

String1 = "1 1 1 0 1 0"

String2 = "1 0 0 1 0 0"

We start iterating the two strings from index 0. Since, String1[0] == Sring2[0], therefore, we move on to the next index and increase the touch count from 0 to 1.

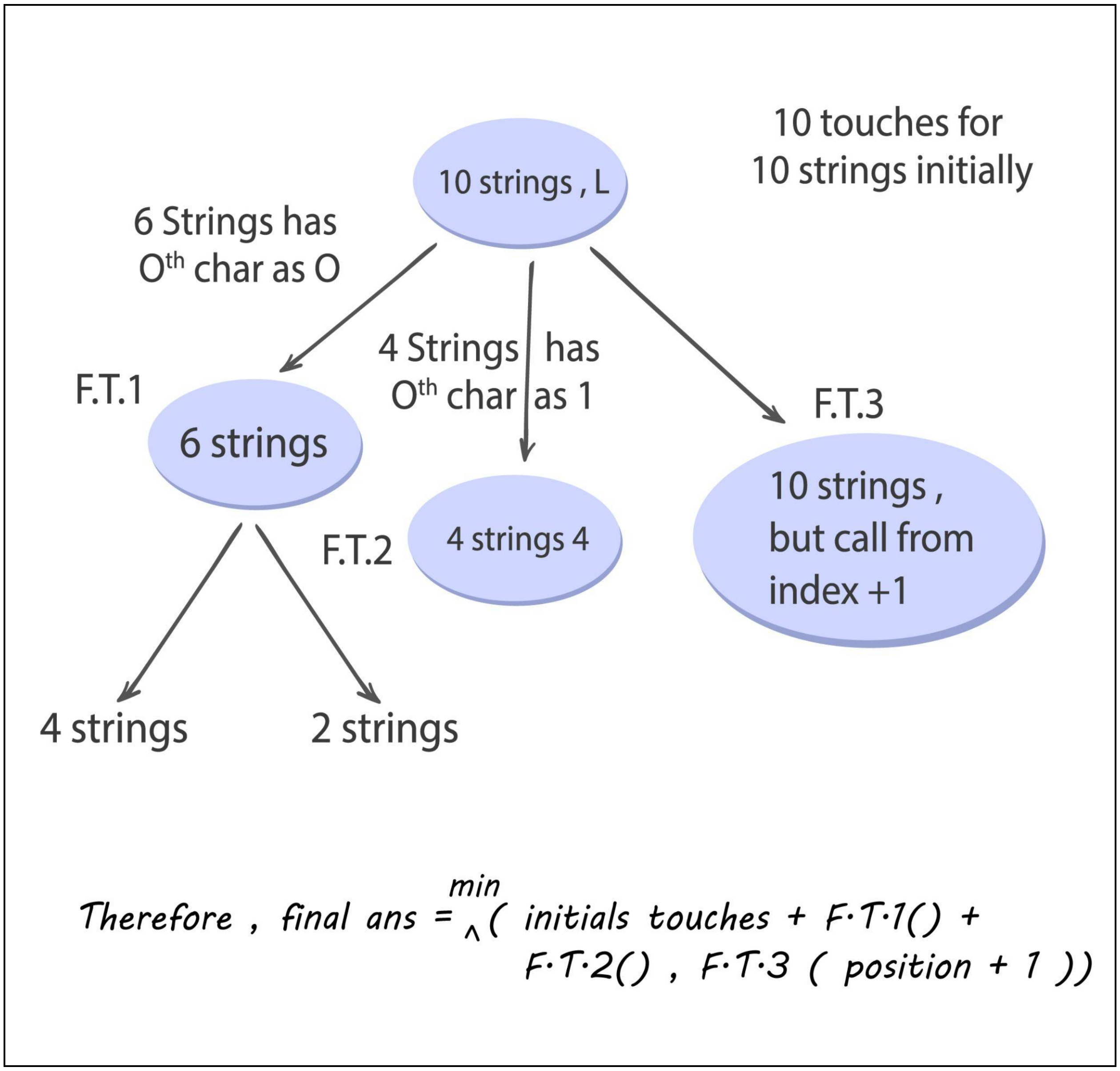
Now, at index 1, String1[1] != String2[1], therefore we know that the two strings are different and we increase the touch count from 1 to 2 for iteration over index 1.

Therefore, total touches = 2.

Let's take a look at the approach for this problem:

- Suppose there are 10 Strings of length L.
- At the 0th index there will be a total of 10 touches for 10 strings.
- Now, let's say that 6 strings have 0th character as 0 and 4 strings have 0th character as 1. And let's call the final answers to these two recursive calls as **FT1 and FT2.**
- Similarly there can be more recursive calls.
- Another case would be that we don't check the 0th character initially to avoid those 10 initial touches and call recursion from index 1 rather than 0 on all 10 strings. Let's call the final answer of this recursion call to be FT3.

- So, our final answer will be **min(initial_touches + FT1 + FT2, FT3)**.



Applying bitmasking:

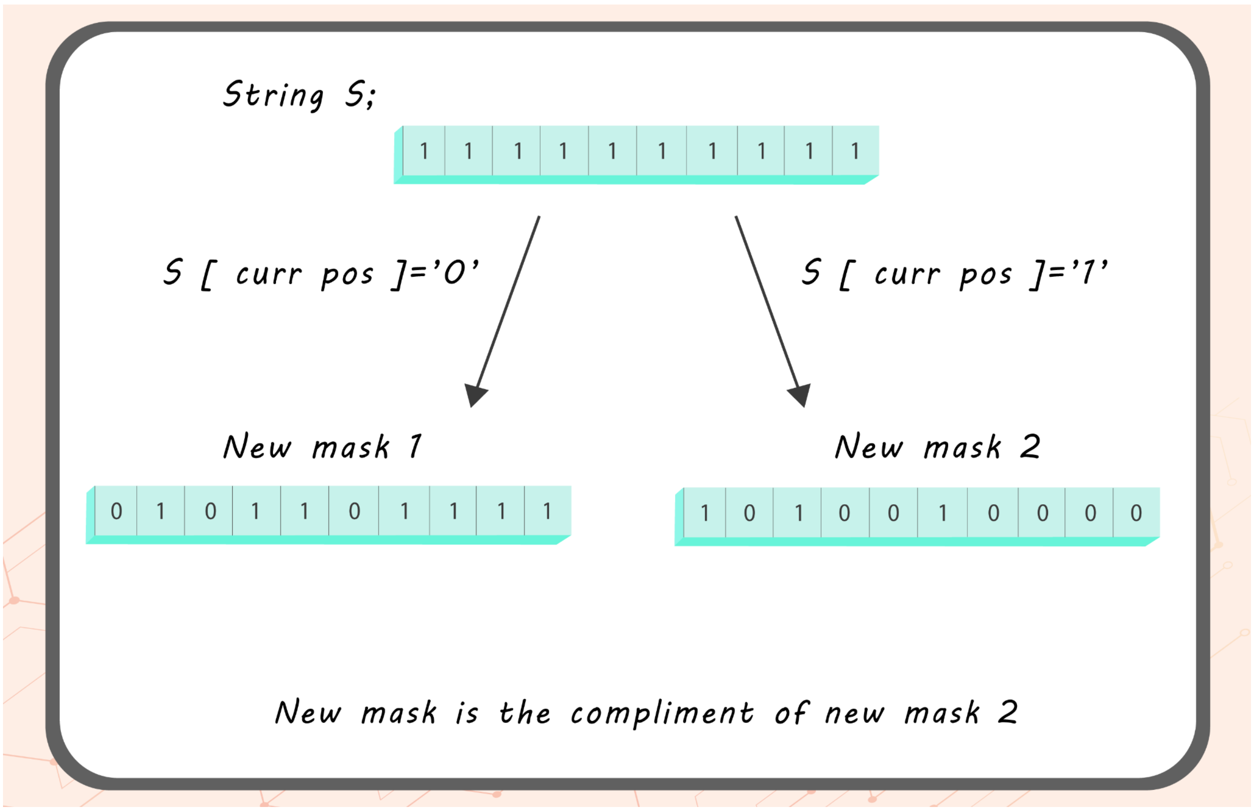
Now, we need to figure out a way with which we can keep the track of strings that we have distuinughed by touching so far.

For this, we will use the concept of BITMASKING.

The maximum number of strings that can be given in a test case according to the question constraints is 10. Thus, in this problem, we maintain a mask of 10 bits, where each bit would represent the state of the string. If ith bit is not set, it means that ith string has been successfully distinguished on the basis of current check(touch). Conversely, if the ith bit is set, it means that this string has not yet been successfully distinguished on the basis of touching the value at the current position, and we need to check(touch) the following positions to distinguish them.

You may check(touch) the string either from the 0th position or the (L - 1)th position, where **L** is the length of the strings. This means that, while calling recursion, you may begin the recursive calls from **pos = 0** and end them using base case **pos = L**, or you may begin recursive calls from **pos = L - 1** and end them using base case **pos = -1**.

Using the mask, we can easily find how many touches will be used to distinguish the strings. How? The number of set bits in the mask will give us the number of touches that are required to further distinguish the strings. At each level we will check the value of the string at the current position. Depending on whether it is 1 or 0, we will create 2 masks recursively, which would give us the number of touches required at that level.



Now, for the base cases:

1. If mask has only one set bit then return 0
2. If position < length then return INT_MAX because we will not be able to distinguish anything.

Code:

```
/*
 Time complexity: O(2^N * ( L ))
 Space complexity: O(L * (2 ^ N))
 where N is the number of strings;
 and L is the length of the string
*/
#include <bits/stdc++.h>
using namespace std;
int dp[105][1 << 11];  // acc to the constraints, you can have max 10
```

```
strings in a single test case, hence to be on safe side we shall use 10+1
as the size.
int find_touches(int pos, int mask, vector<string> &v) {
    if (!(mask & (mask - 1)) && mask) {
        return 0;
    }
    if (pos == -1 || mask == 0) {
        return 1000000;
    }
    if (dp[pos][mask]) {
        return dp[pos][mask];
    }
    int newmask1 = 0, newmask2 = 0, touches = 0;
    for (int i = 0; i < v.size(); i++) {
        if ((mask >> i) & 1) {
            touches++;
            if (v[i][pos] == '0') {
                newmask1 |= (1 << i);
            } else { newmask2 |= (1 << i);
            }
        }
    }
    return dp[pos][mask] = min(find_touches(pos - 1, newmask1, v) +
find_touches(pos - 1, newmask2, v) + touches, find_touches(pos - 1, mask,
v));
}
int solve(int n, vector<string> v) {
    return find_touches(v[0].size() - 1, (1 << n) - 1, v);
}
int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<string> v;
        memset(dp, 0, sizeof(dp));
        for (int i = 0; i < n; i++) {
            string s;
            cin >> s;
            v.push_back(s);
        }
        cout << solve(n, v) << endl;
    }
}
```