PW SKILLS

Backtracking (Part 1)

Backtracking

Recursion

↳ merge sort

↳ quick sort

↳ Programming paradigms

→ Brute force

→ DP

→ Greedy

→ DNC

in brute force we explore all the possible outcomes doesn't matter whether it leads to an answer or not.

we prune irrelevant calls.

→ we trace the path for an answer, & then revert it for other possibilities

Q. You are given a string consisting of small alphabets. Consider no repetition in the characters.

Print all possible permutations of the given string.

permutations ↪ arrangements.

(Ordering of permutations doesn't matter)

Ex → "abc" → abc

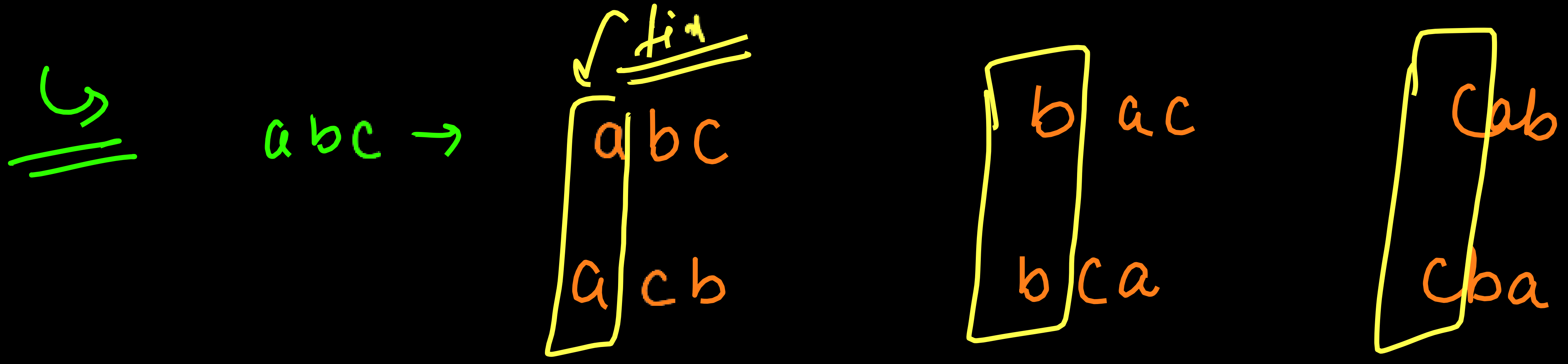
acb

bac

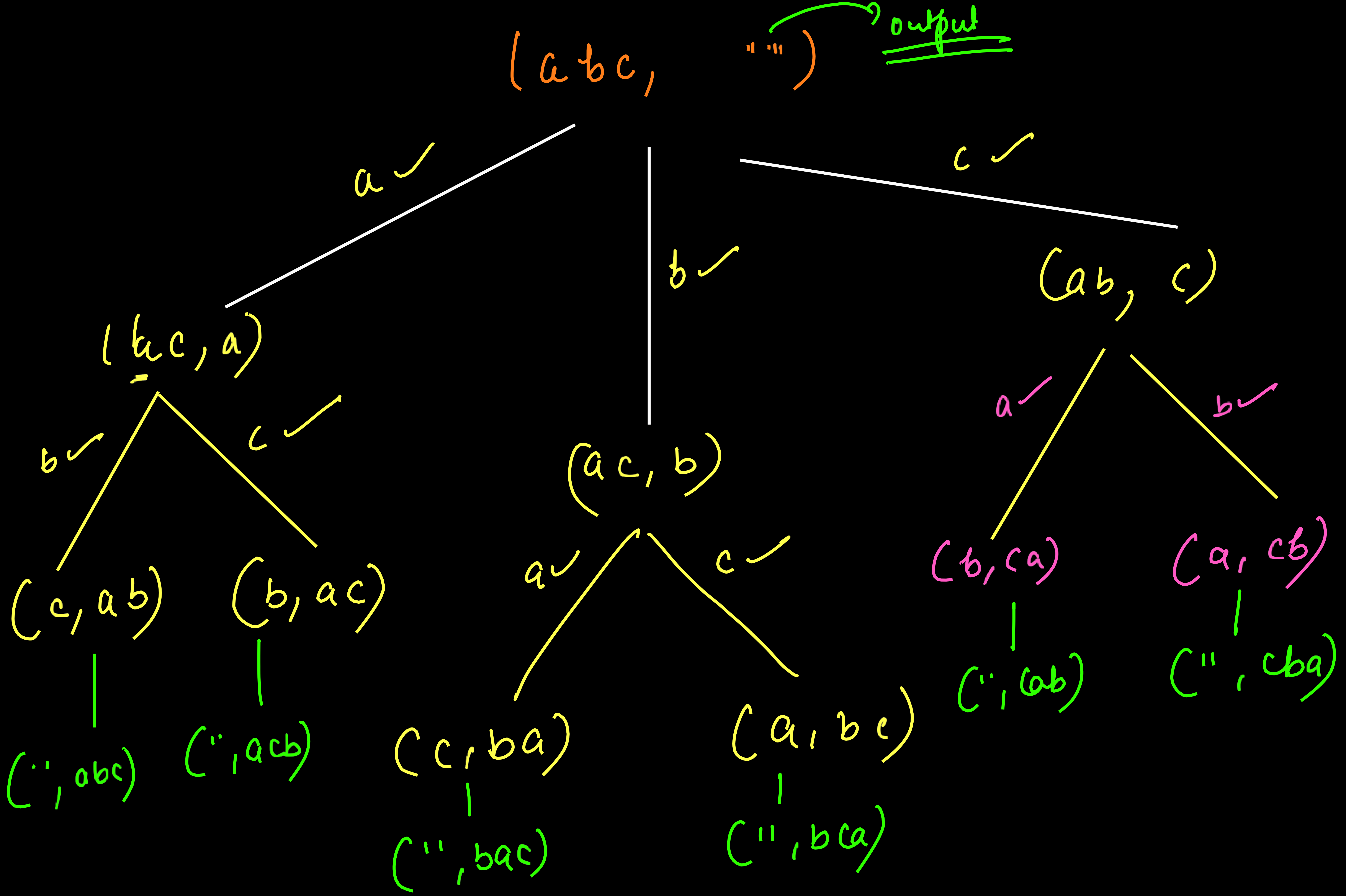
bca

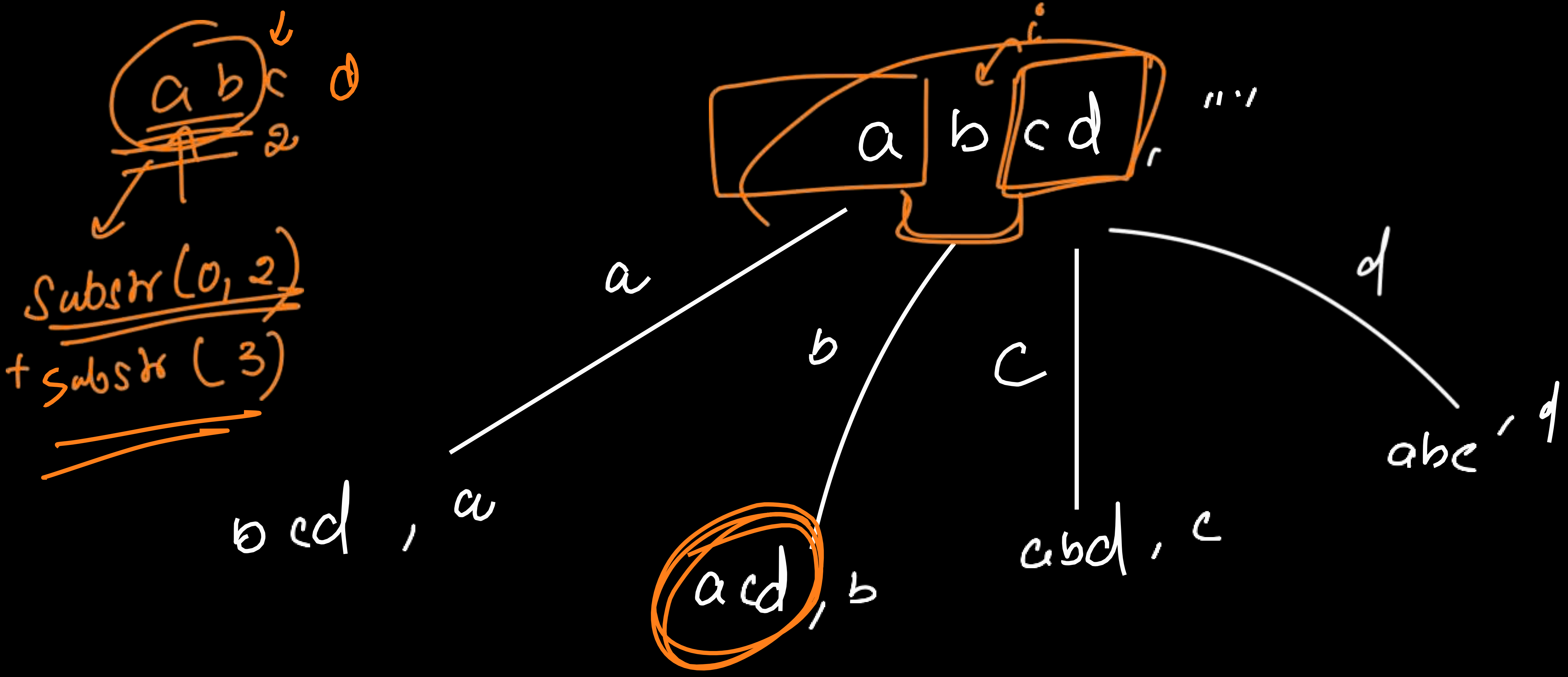
cab

cba

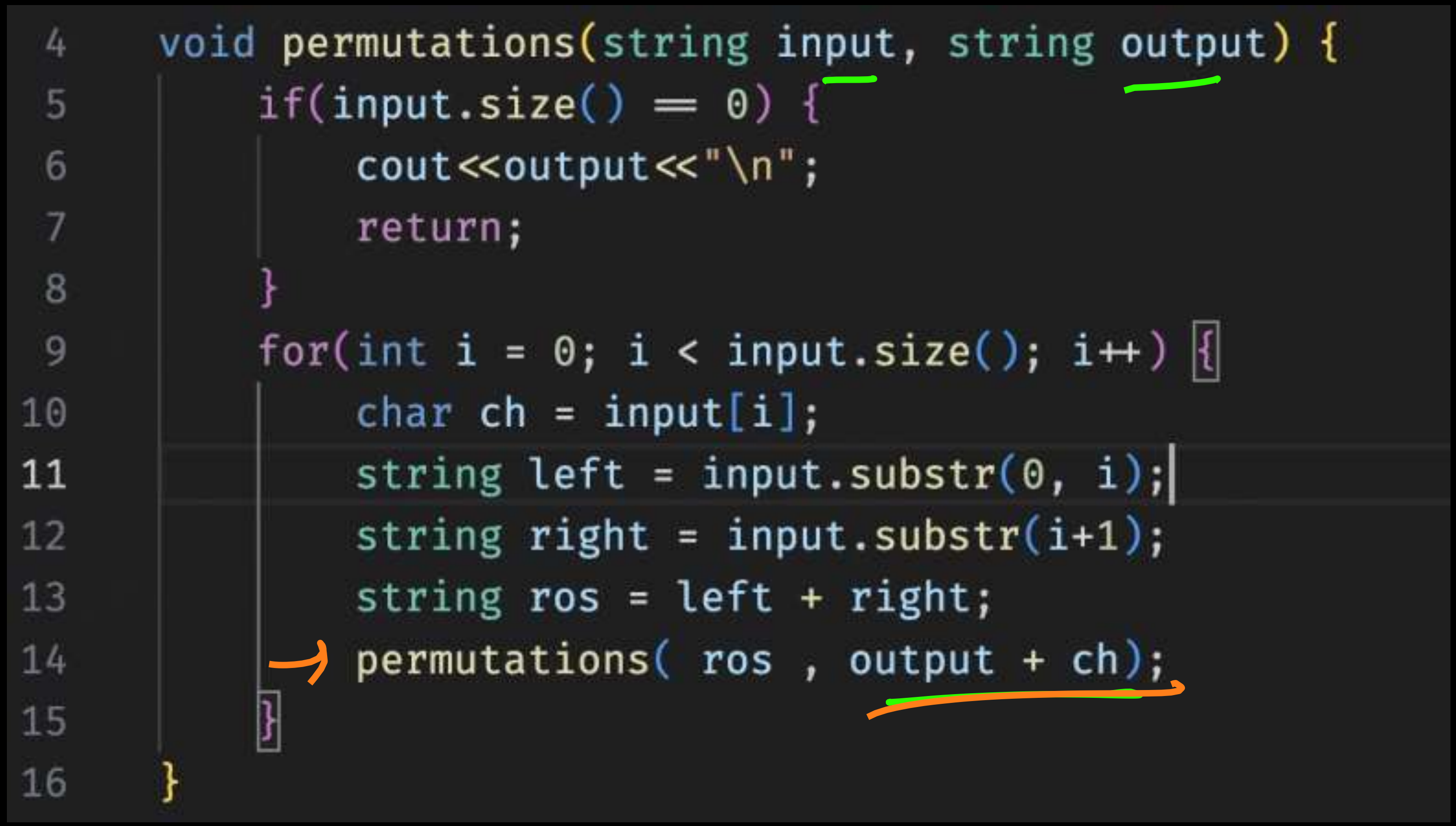


Every character is given a chance to become a suffix of a group of permutations


(abc, "")
output
a ✓
b ✓
c ✓
(bc, a)
(ac, b)
(ab, c)
b ✓
c ✓
a ✓
c ✓
a ✓
b ✓
(c, ab)
(b, ac)
(c, ba)
(a, bc)
(b, ca)
(a, cb)
('', abc)
('', acb)
('', bac)
('', bca)
('', cab)
('', cba)



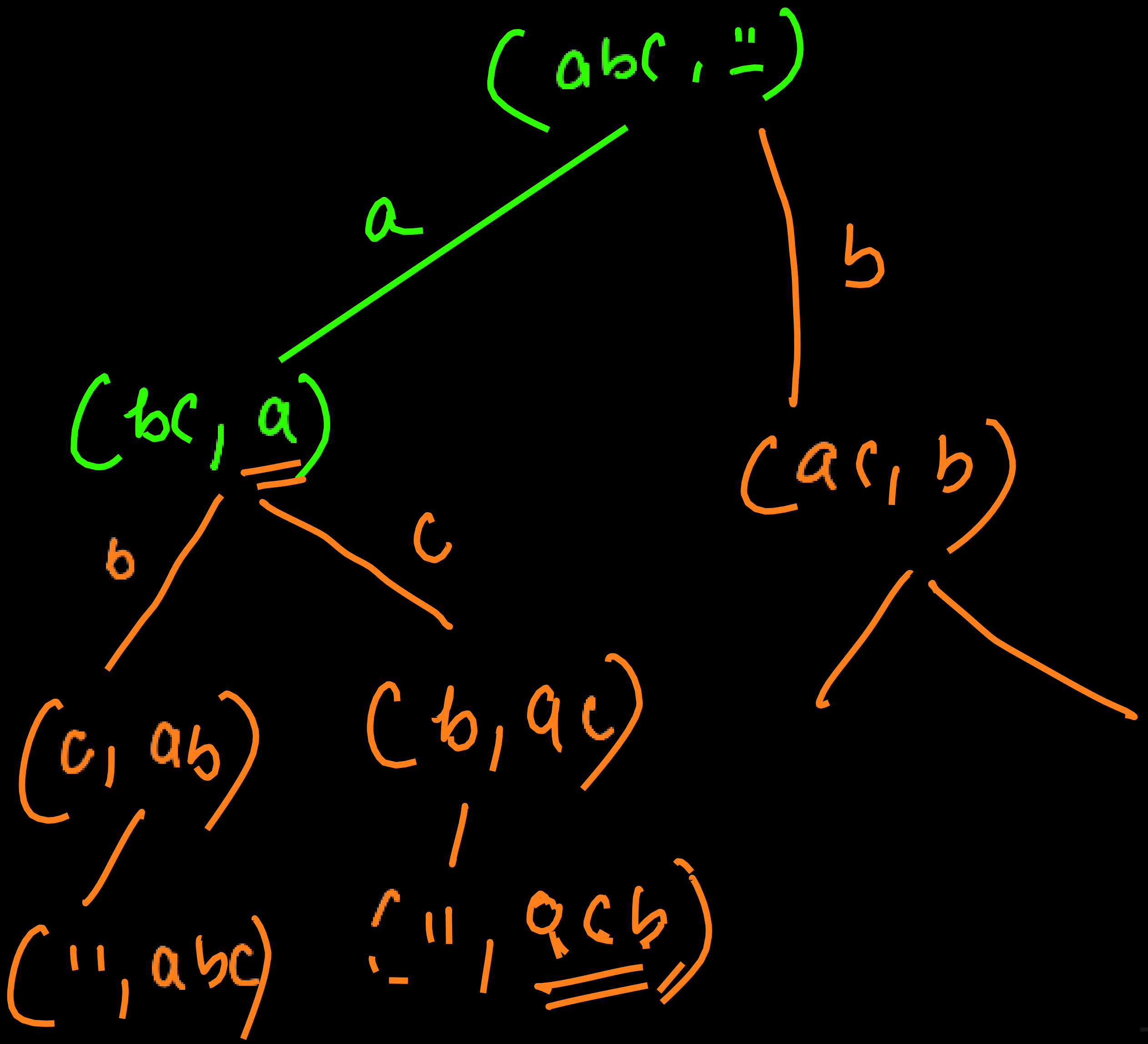
substr (0, 2)

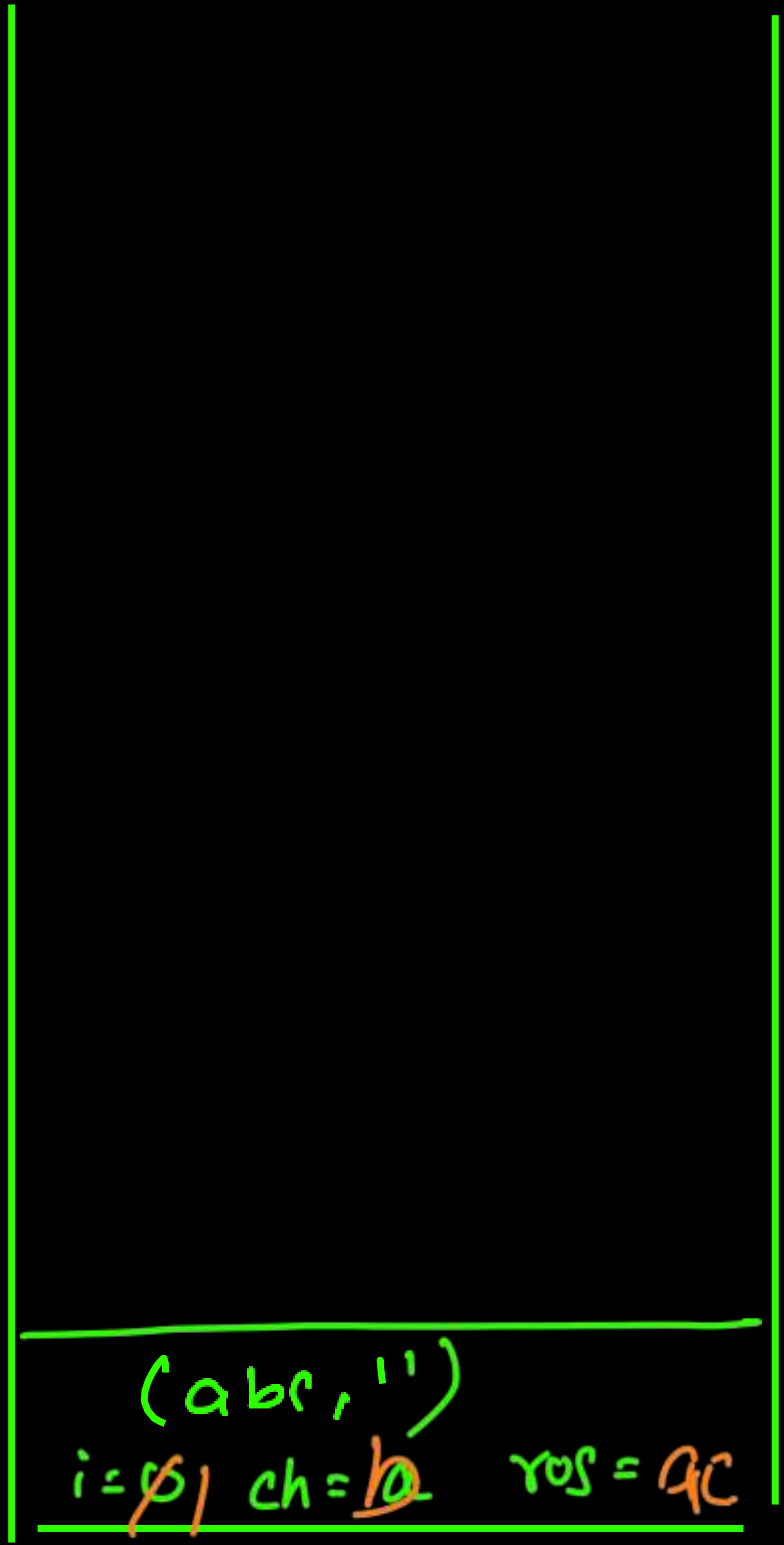


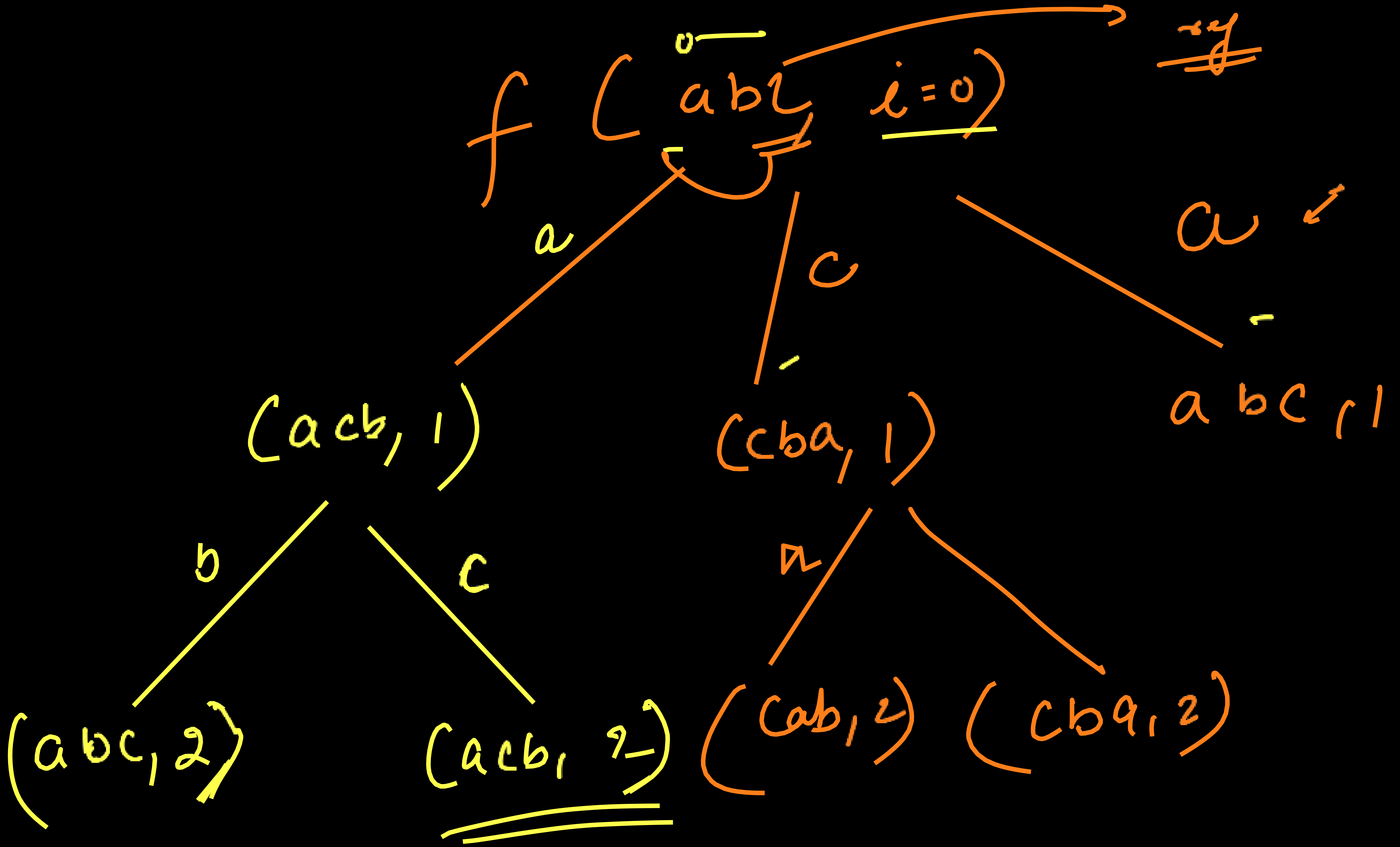
```
void permutations(string input, string output) {
    if(input.size() == 0) {
        cout<<output<<"\n";
        return;
    }
    for(int i = 0; i < input.size(); i++) {
        char ch = input[i];
        string left = input.substr(0, i);
        string right = input.substr(i+1);
        string ros = left + right;
        permutations( ros , output + ch);
    }
}
```

abc

acb





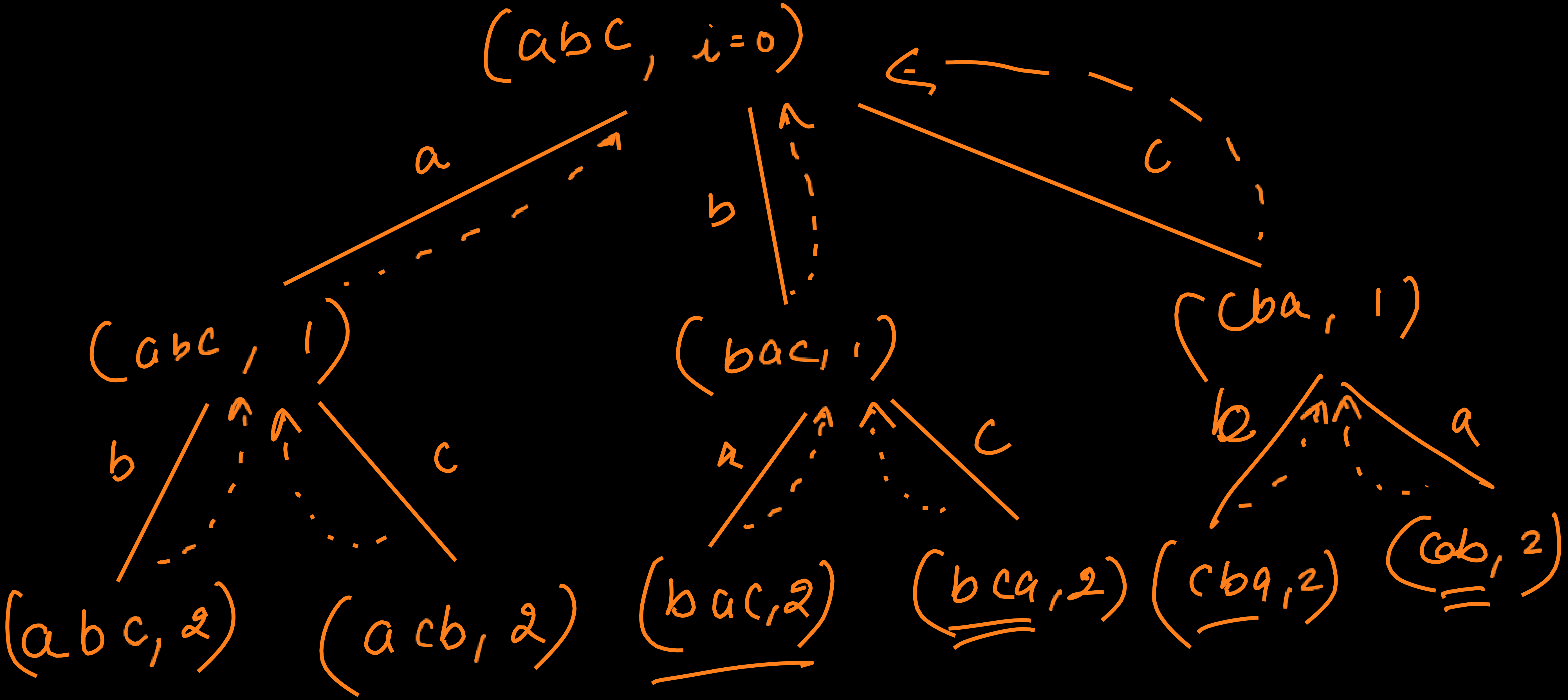

f(abc, i=0)
xy
a
c
a
(acb, 1)
(cba, 1)
abc, 1
b
c
a
(abc, 2)
(acb, 2)
(cab, 2)
(cba, 2)

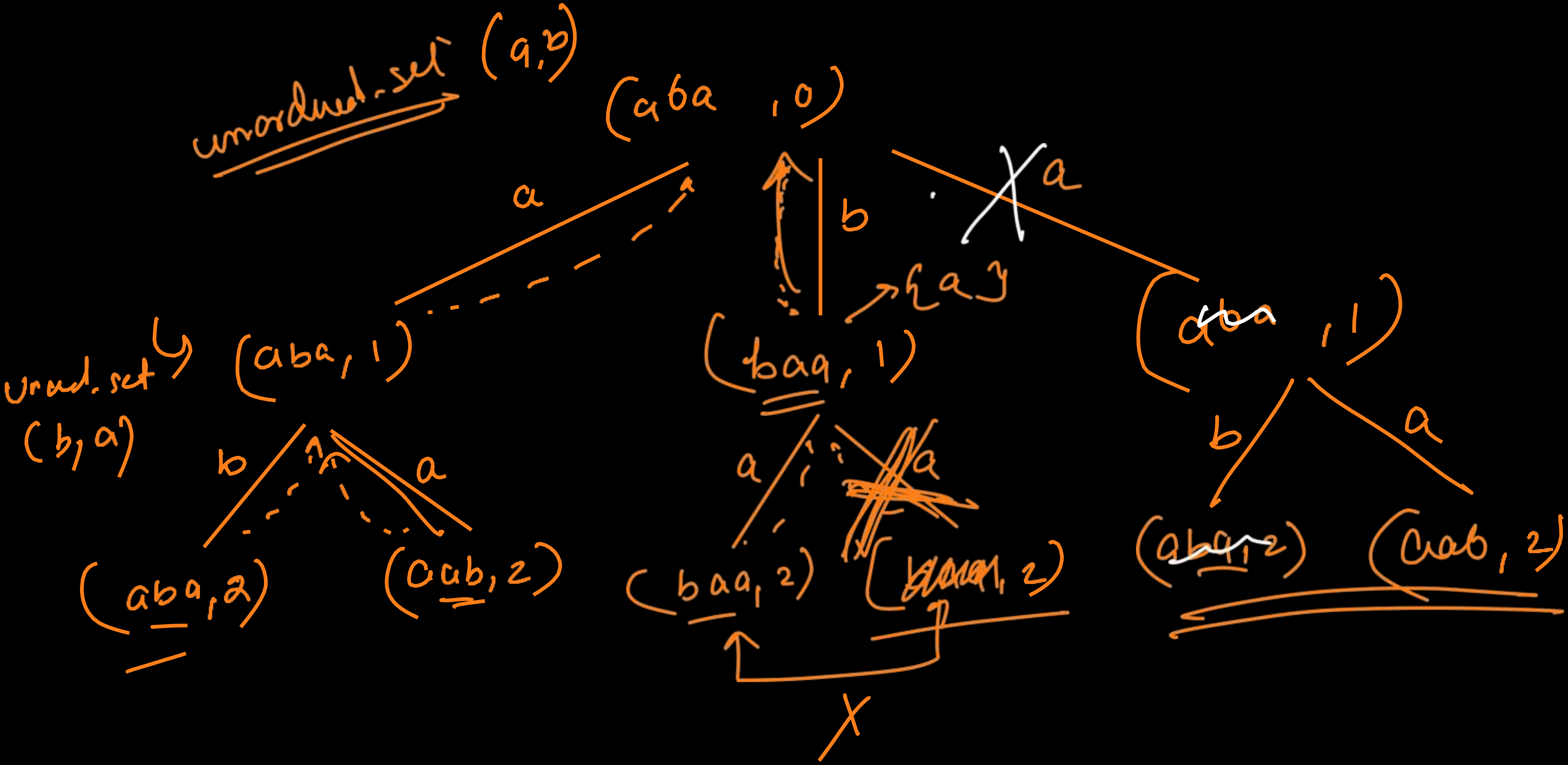
$f\,(str, i)$

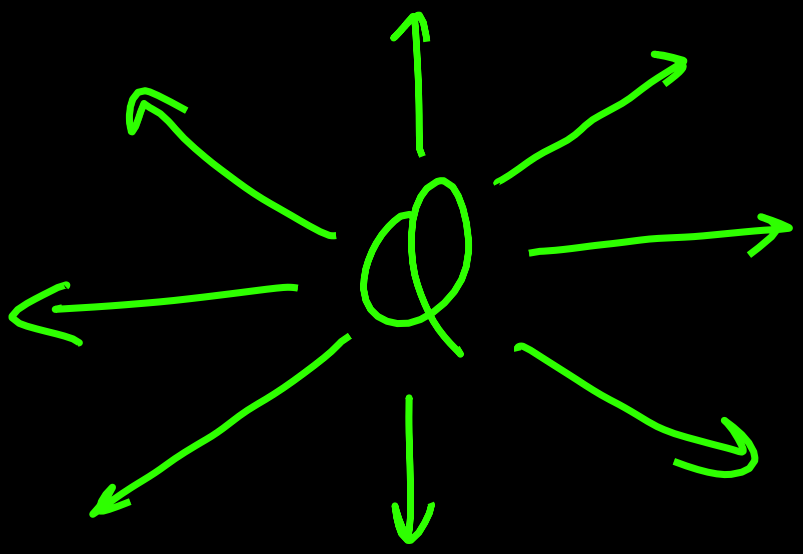
↓

generates perm of string

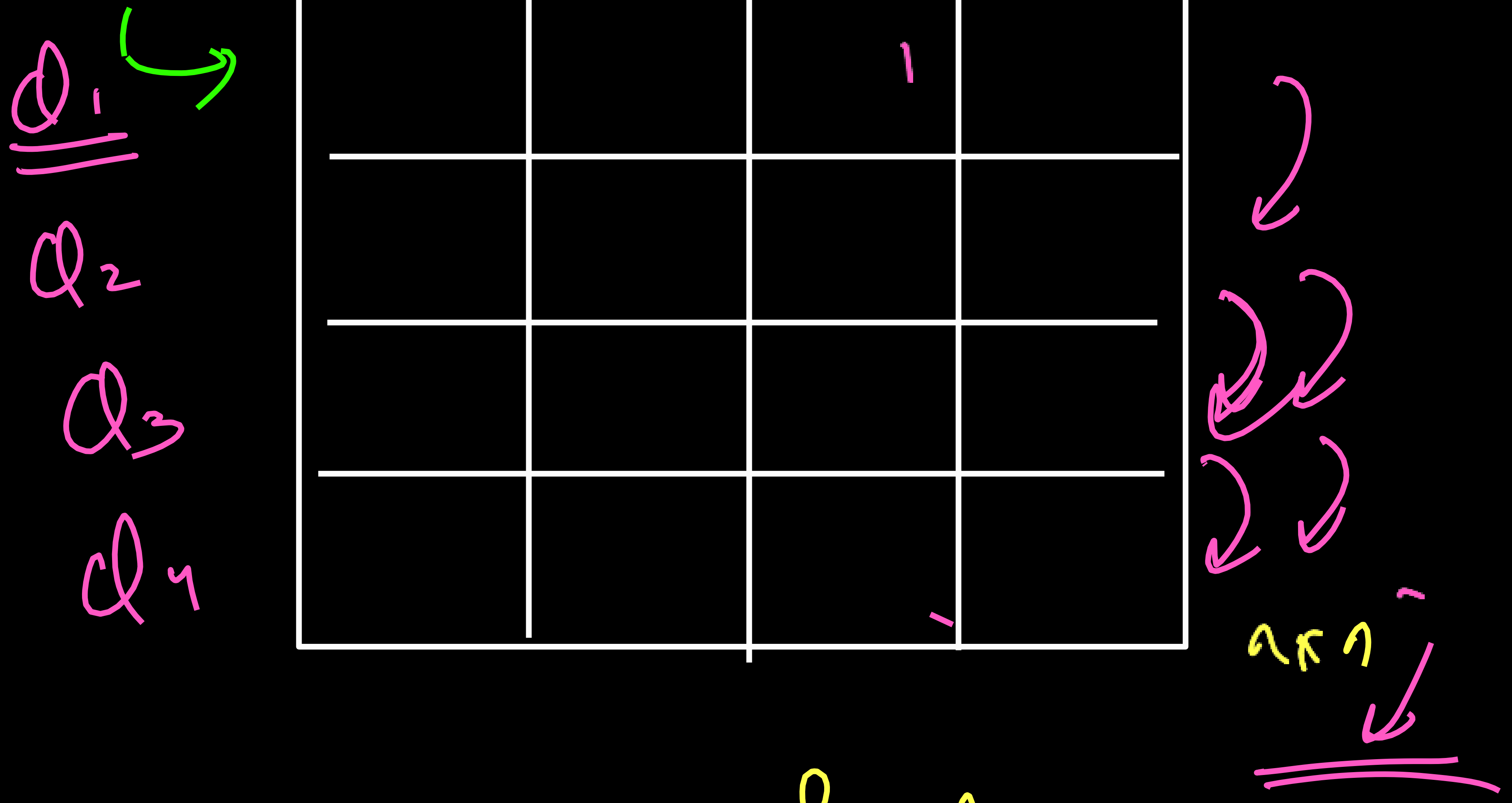
$str\,[i, n-1]$


(abc, i=0)
a
b
c
(abc, 1)
(bac, 1)
(cba, 1)
b
c
a
c
b
a
(abc, 2)
(acb, 2)
(bac, 2)
(bca, 2)
(cba, 2)
(cab, 2)


unordered_set
(a,b)
(aba ,0)
a
b
a
{a}
unord.set
(b,a)
(aba, 1)
(baa, 1)
(aba ,1)
b
a
a
a
b
a
(aba,2)
(aab,2)
(baa,2)
(baa,2)
(aba,2)
(aab, 2)


Q

n = 4

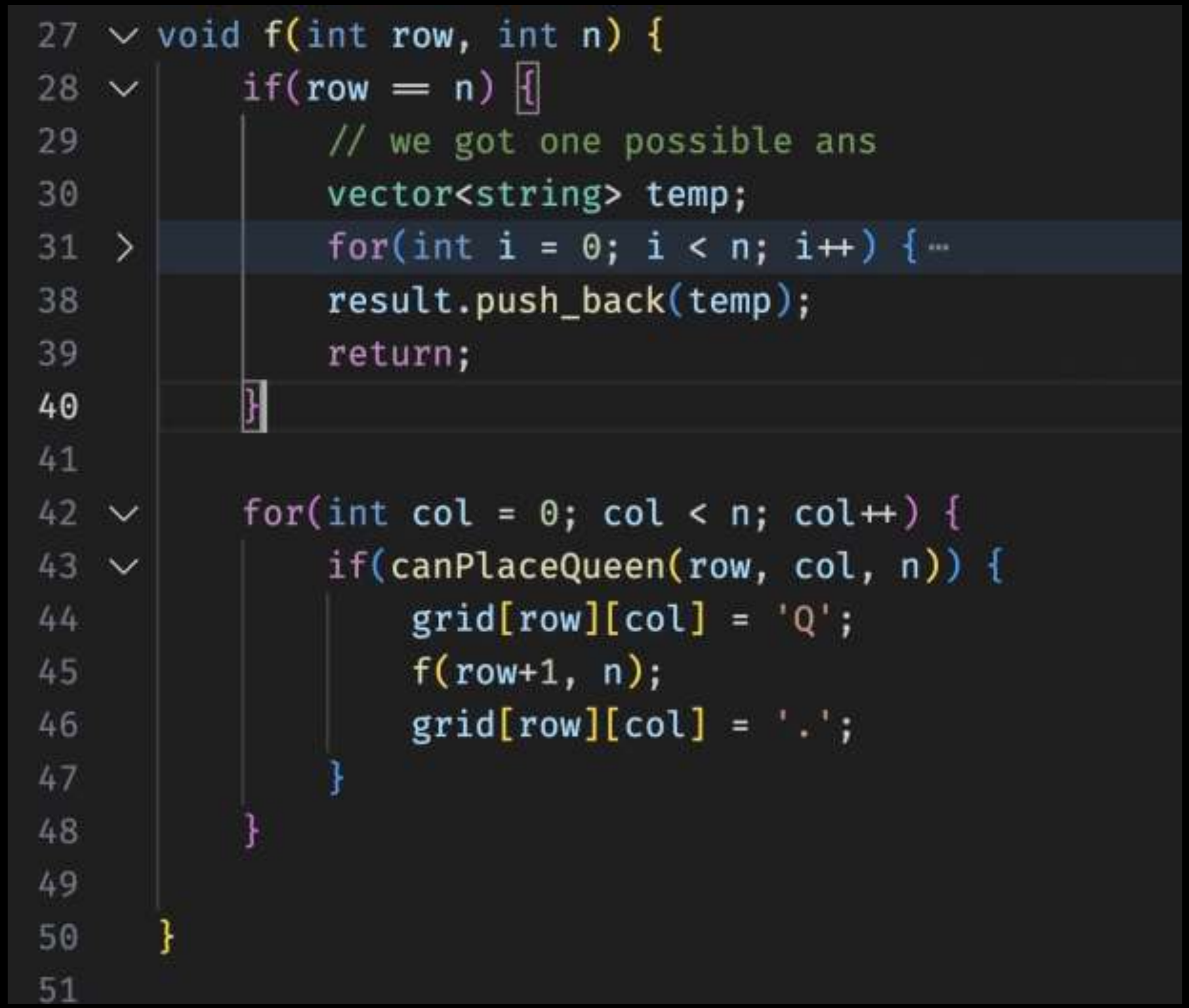


→ prune the call that doesn't lead to a valid ans.

→ if you change the state revert it as well.

every row has one & only one queen.

we have n rows, & n queens, i.e. every row needs to accomodate exactly 1 queen.

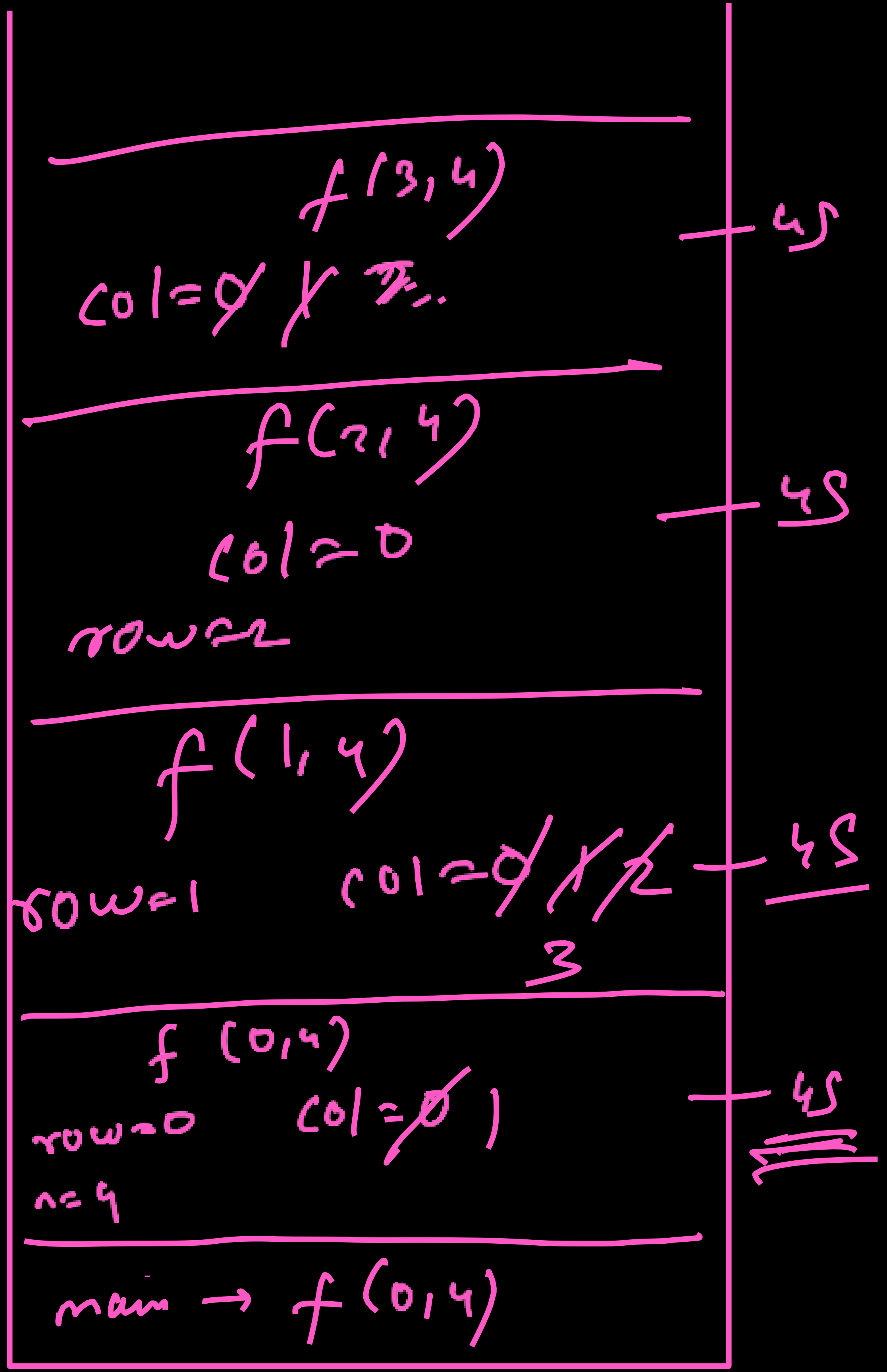


```
void f(int row, int n) {
    if(row == n) {
        // we got one possible ans
        vector<string> temp;
        for(int i = 0; i < n; i++) {⋯
        result.push_back(temp);
        return;
    }

    for(int col = 0; col < n; col++) {
        if(canPlaceQueen(row, col, n)) {
            grid[row][col] = 'Q';
            f(row+1, n);
            grid[row][col] = '.';
        }
    }

}
```

.	Q	.	.
.	.	.	Q
Q	.	.	.
.	.	.	.

row

4x4



Backtracking

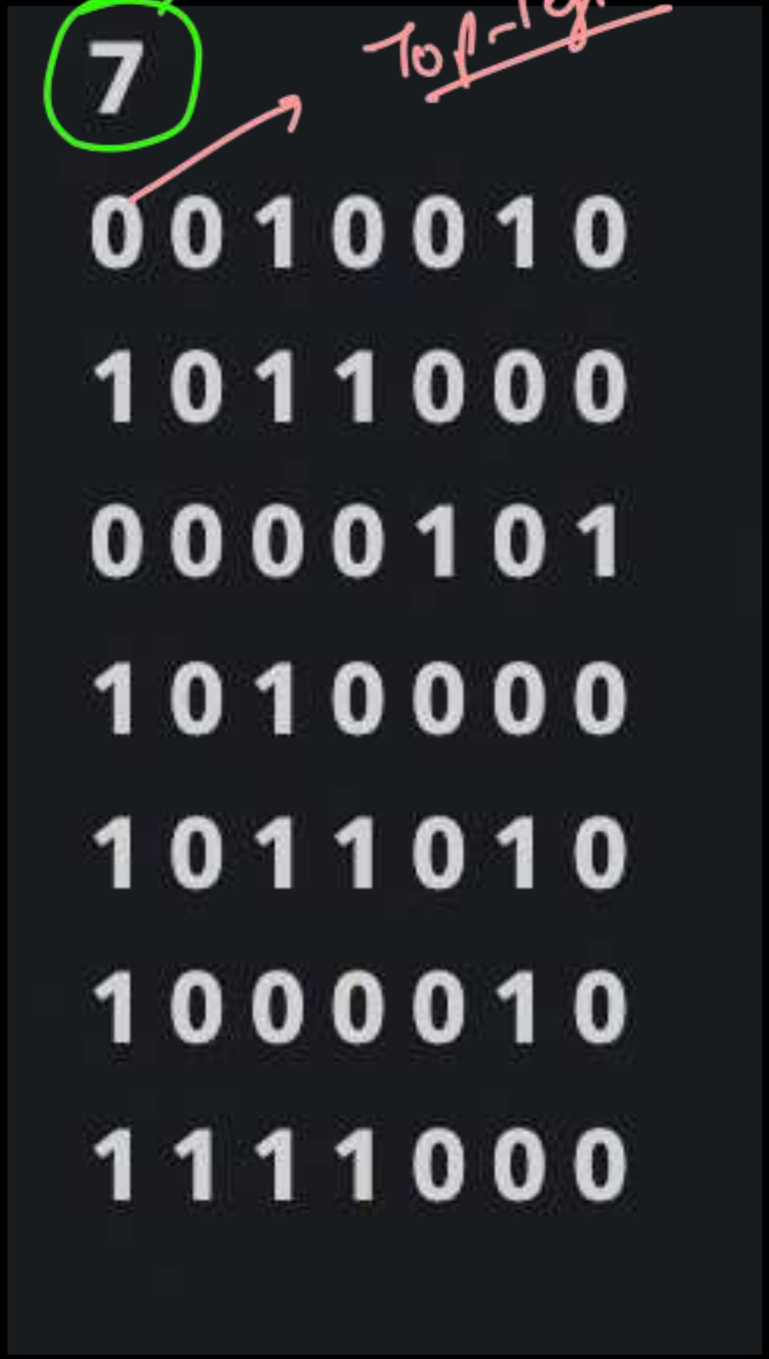
Rat In A Maze

0 → open cells

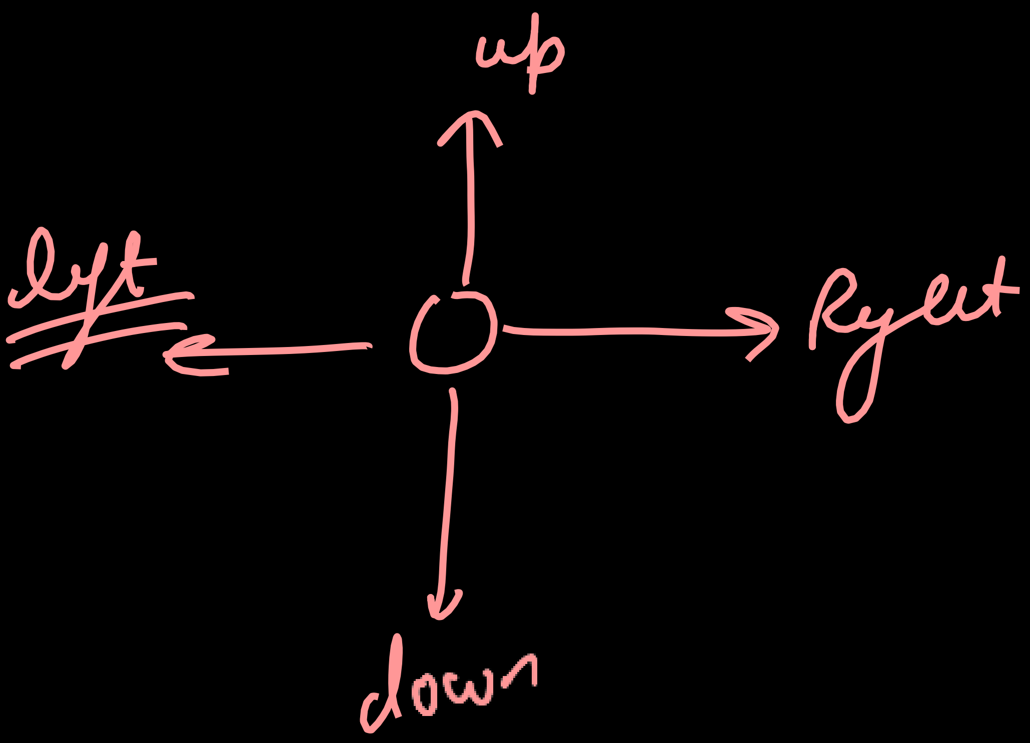
1 → blocked cells

n

Top-left

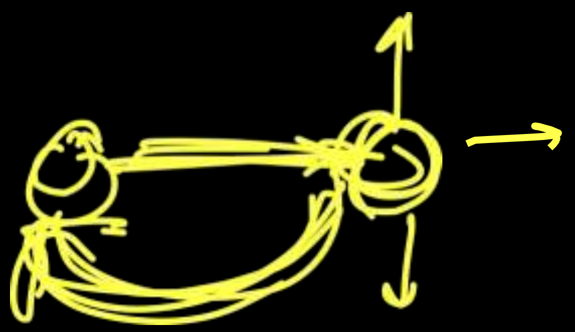


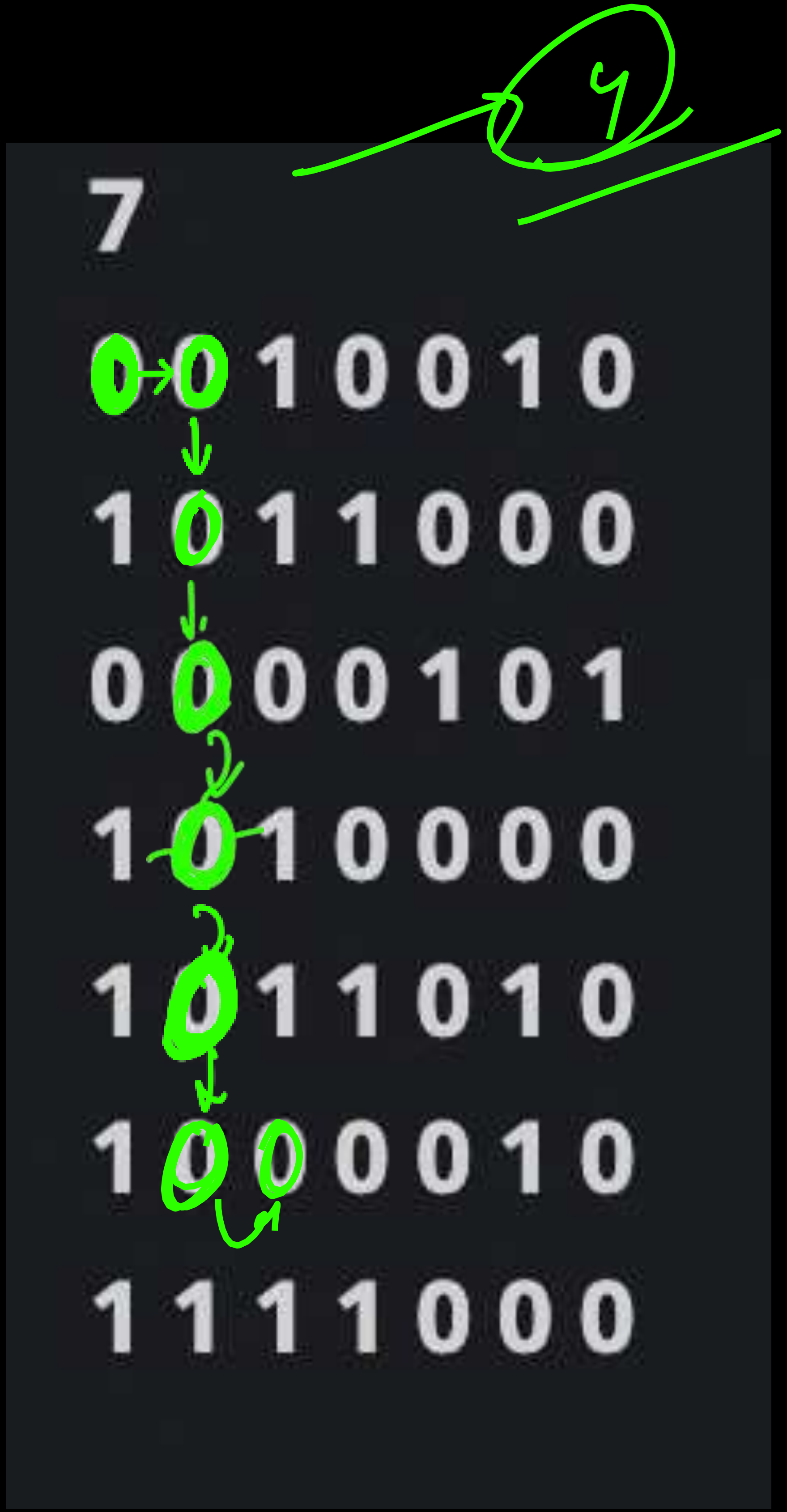
```
7
0010010
1011000
0000101
1010000
1011010
1000010
1111000
```



Start from (0,0)

In how many ways the rat can reach the bottom Right.



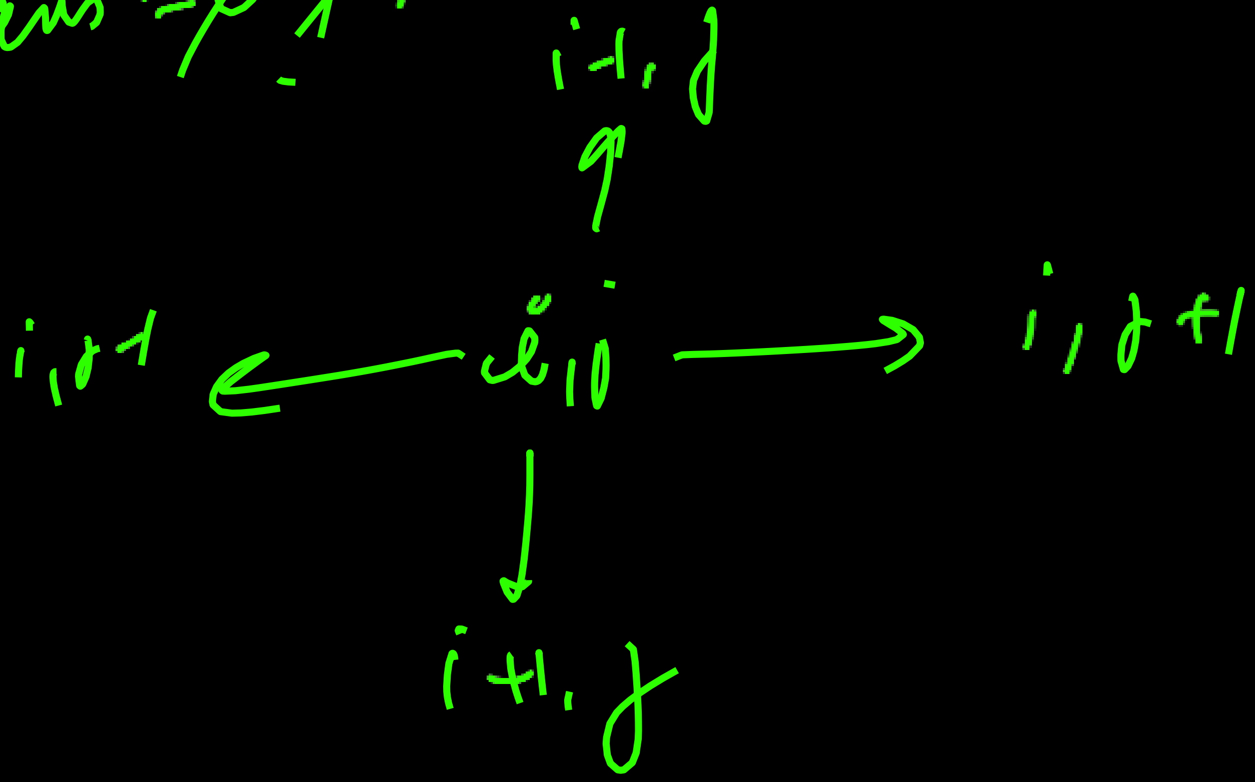


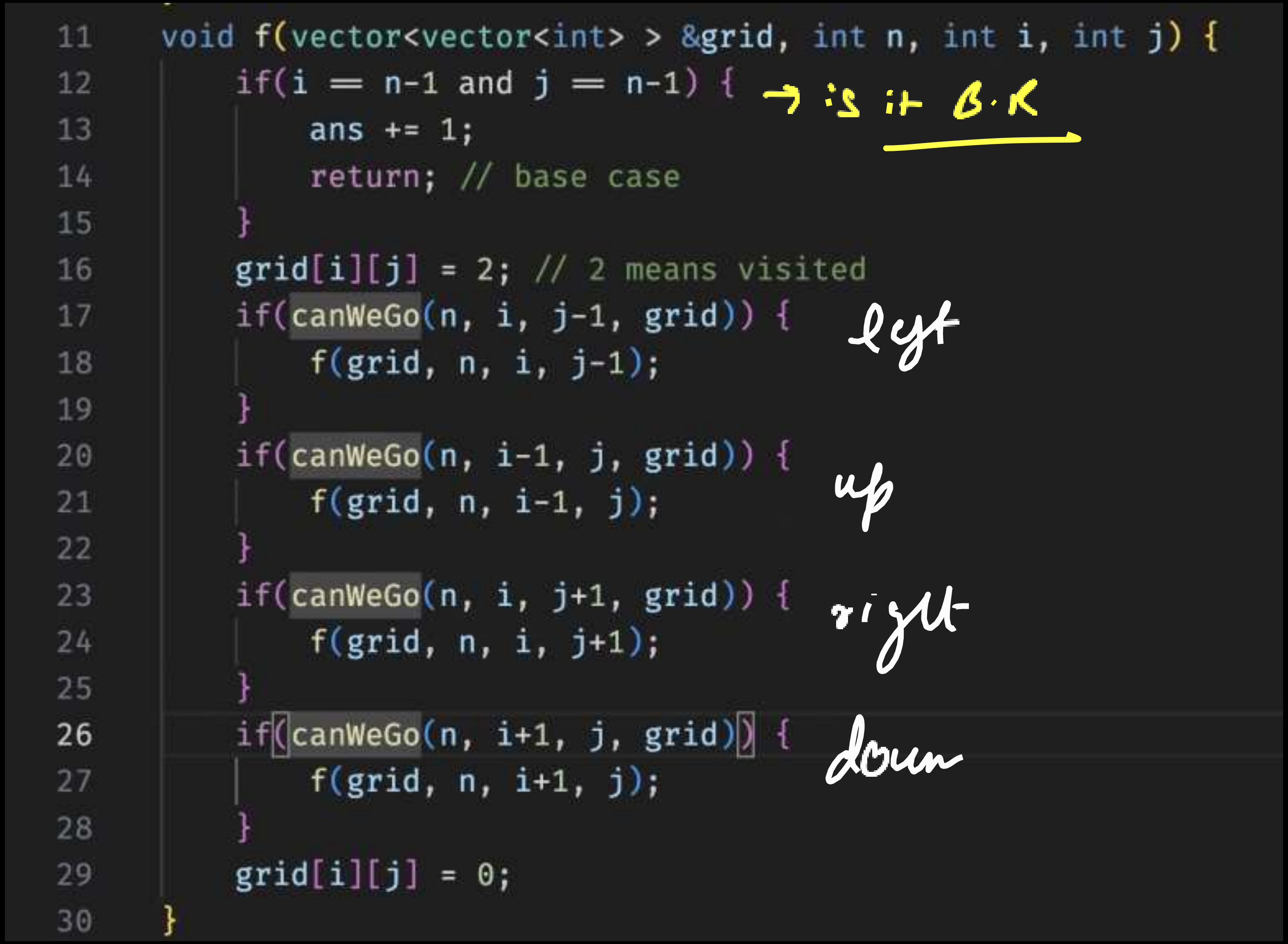
```
7

0010010
1011000
0000101
1010000
1011010
1000010
1111000
```

l, up, right, down

visited →

ans = ~~0~~ ~~1~~ 2 -





```
void f(vector<vector<int> > &grid, int n, int i, int j) {
    if(i == n-1 and j == n-1) {
        ans += 1;
        return; // base case
    }
    grid[i][j] = 2; // 2 means visited
    if(canWeGo(n, i, j-1, grid)) {
        f(grid, n, i, j-1);
    }
    if(canWeGo(n, i-1, j, grid)) {
        f(grid, n, i-1, j);
    }
    if(canWeGo(n, i, j+1, grid)) {
        f(grid, n, i, j+1);
    }
    if(canWeGo(n, i+1, j, grid)) {
        f(grid, n, i+1, j);
    }
    grid[i][j] = 0;
}
```

→ is it B·K

left

up

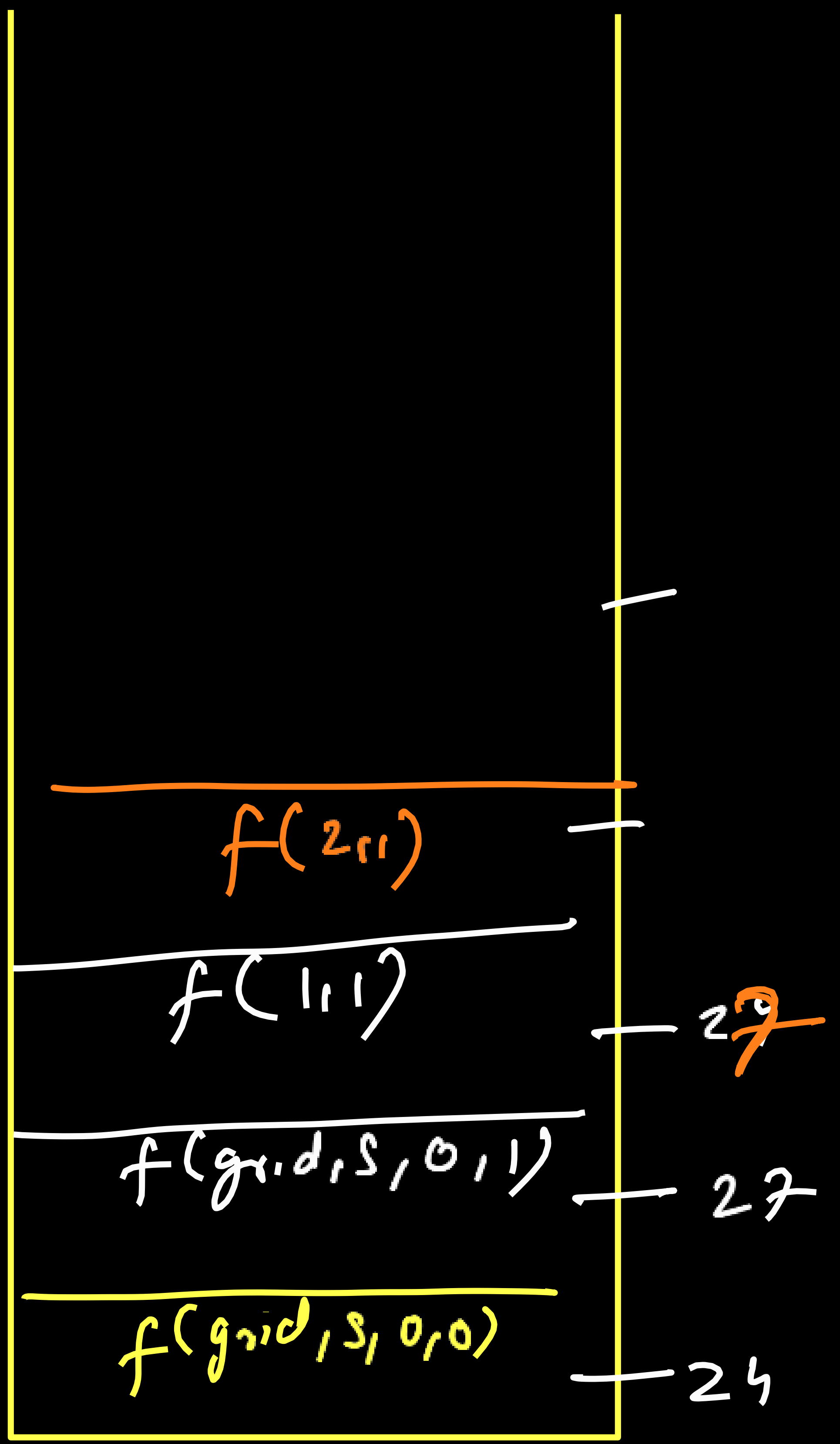
right

down

~~0~~2	2~~0~~	1	1	1
1	2~~0~~	0	0	1
1	0	1	0	1
1	0	0	0	1
1	1	1	0	0

n=5

ans = ~~0~~ 1



PW SKILLS

THANK YOU