BACKTRACKING

Agenda → Decipher fundamental concepts of Backtracking

→ How it differs from recursion,

→ How to visualize it.

→ Problem solving

Q => Given a string of all unique characters, find all the permutation of string. (Recursively)

Ex "abc"

abc
acb
cab
cba
bac
bca

→ Every character gets a chance to get added to the prefix

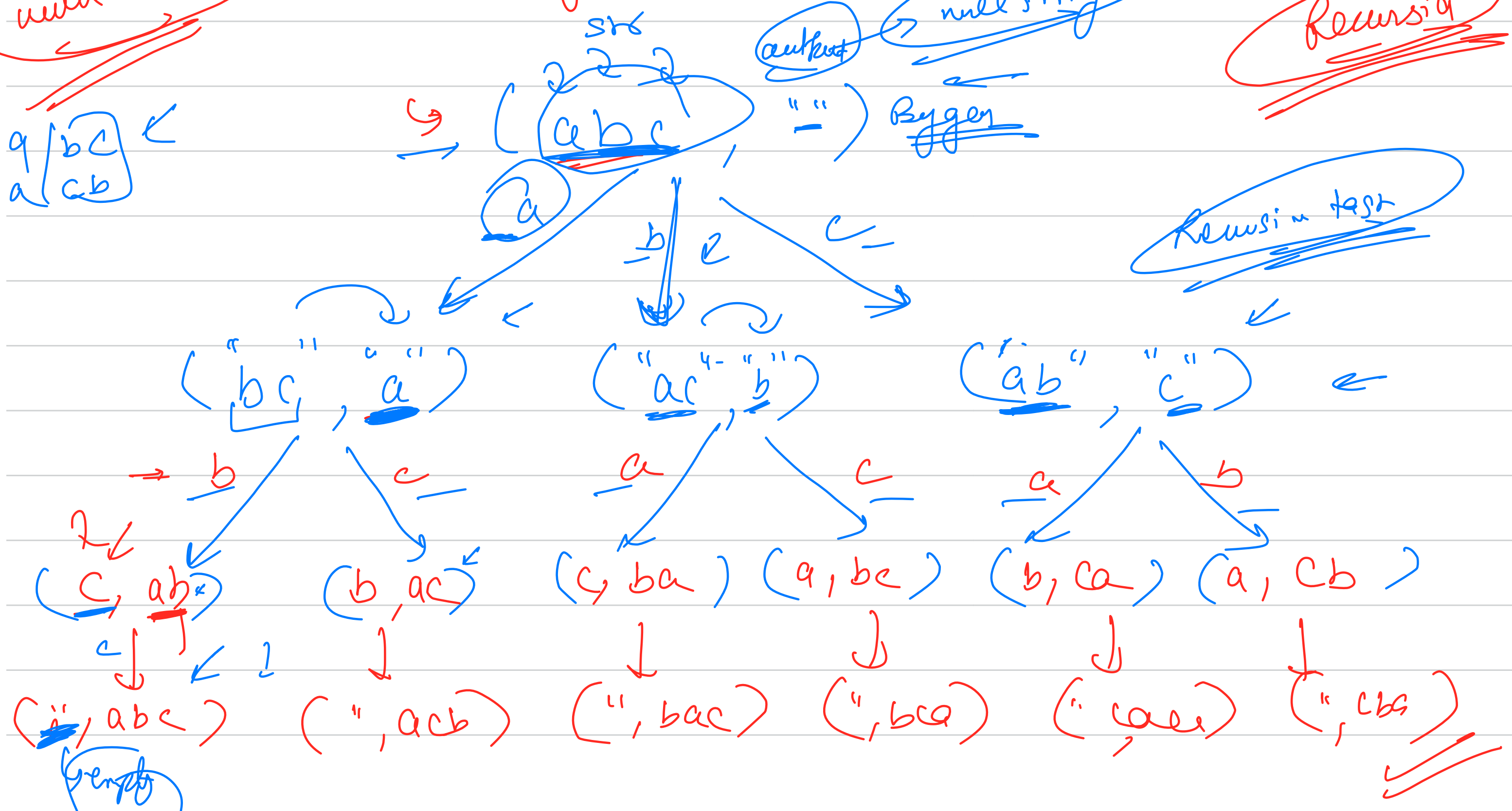
Try just with recursion

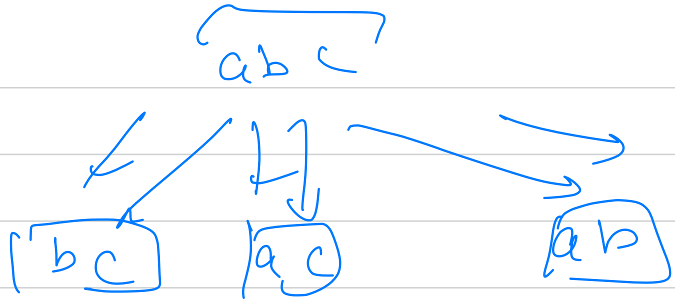
→ Base Case

→ Recursive Intuition

→ Self work

Recursion





$$\text{str} \Rightarrow \underbrace{s_1\, s_2\, s_3 \,-\,}\overset{s_k}{-} \; \overbrace{-\,-\,-\,-\; s_j}$$

$$\Rightarrow [\text{left}] + [\text{char}]^{s_k} + [\text{right}]$$

$$\rightarrow \underbrace{[s_1 - s_{k-1}] + [s_{k+1} - s_j]}$$ ✓

$s \rightarrow abc$

$\downarrow$

ac

$\underbrace{s.substr(0, 1)}_{a} + \underbrace{s.substr(2)}_{c}$

ac

s ← abcd

$$\text{abcd} \xrightarrow{a} \text{bcd} \qquad \text{abcd} \xrightarrow{b} \text{acd} \qquad \text{abcd} \xrightarrow{c} \text{abd} \qquad \text{abcd} \xrightarrow{d} \text{abc}$$

bcd   acd   abd   abc

s.substr(1) + s.substr(2)

$\Downarrow$ $\Downarrow$

a ⟷ cd

acd

s.substr(2) + s.substr(3)

$\Downarrow$ $\Downarrow$

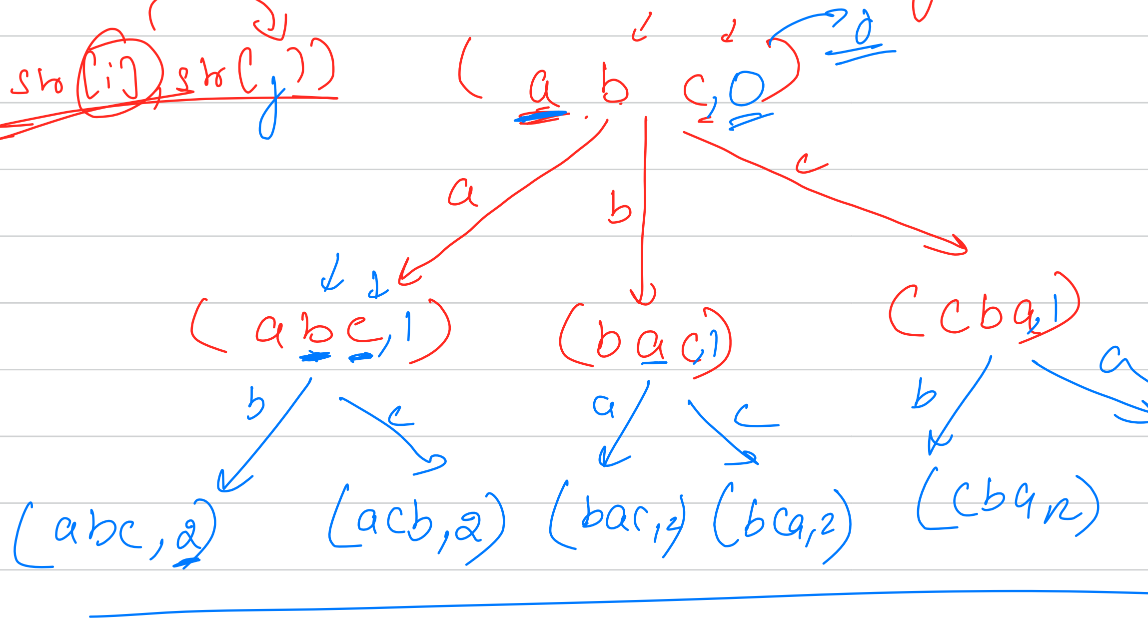
ab ⟷ d

$\Downarrow$

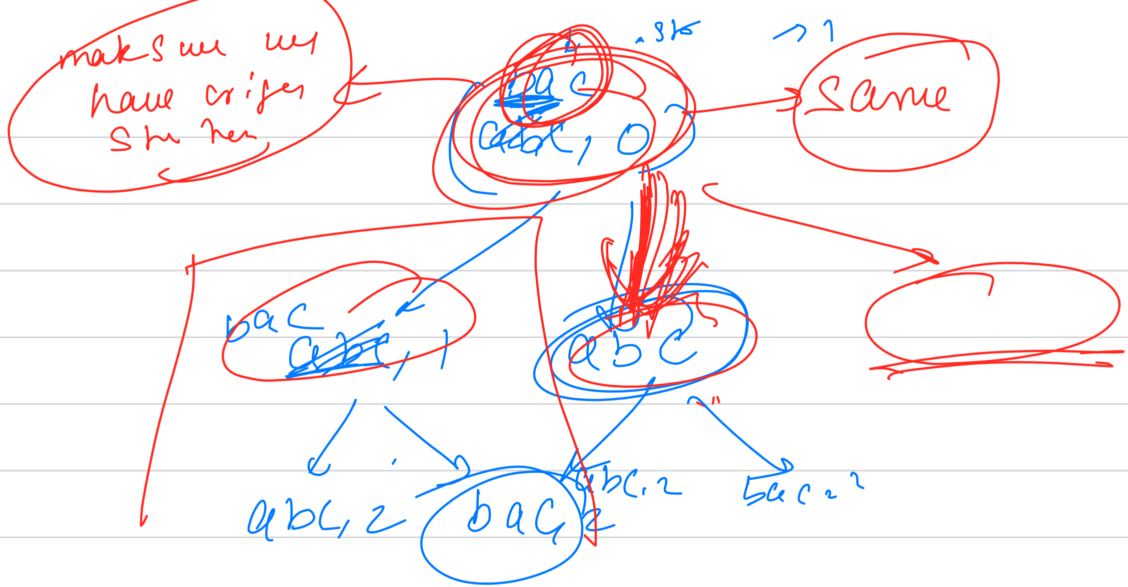
abd

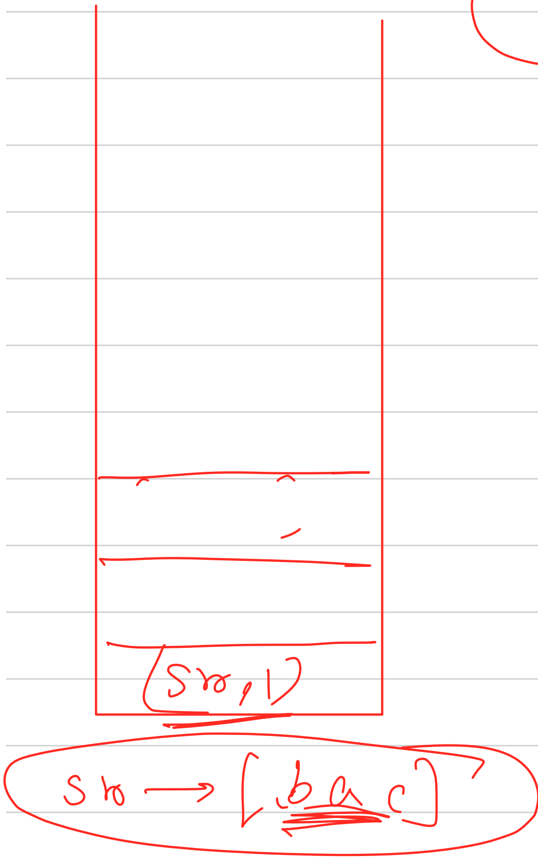
issue → but we need an overhead of calc substring

~~swap(str[i], str[j])~~



~~This is wrong~~





contents

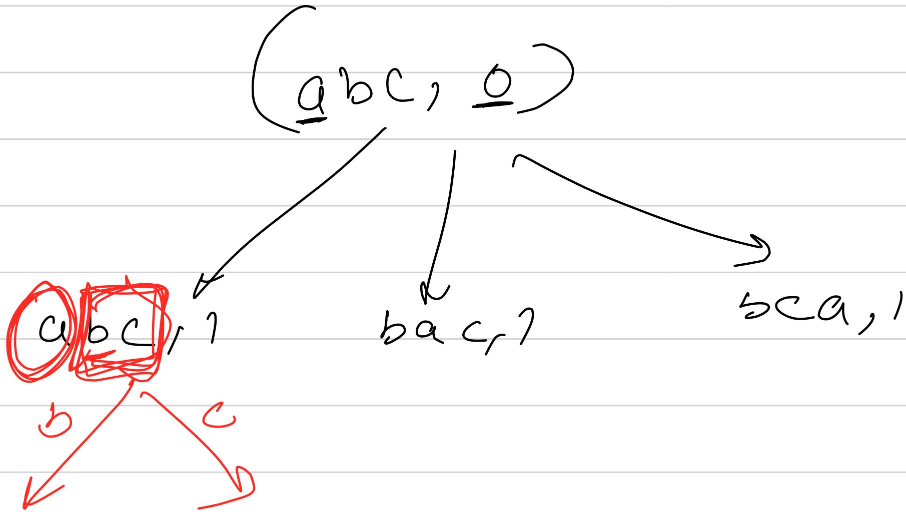
copy

abc

bac

```
void fun (str, j) {
    if (j == str.size()-1) {
        cout << str
        return;
    }
    for (i = 0; i < str.size ; i++) {
        swap (str[i], str[j]);
        fun (str, j+1);
    }
```

Due to the swapping, original string changed



str ← (abc)

Recursive task → Calculate all permutation of str.substr(i)

self work → attach every character to prefix

str.substr(0, i) + str.substr(i+1)

→ O(n)

extra work

Space

abc
acb

for subproblems

(abc, $j=0$)

"" → Output

$O(n)$

(abc, 1) (bac, 1) (cba, 1)

abc, 2 acb, 2 ( ) ( ) ( ) ( )

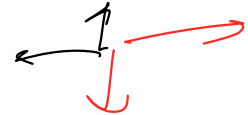
Backtracking → Backtracking is an algorithm that tries to find a solution for given parameters recursively where it builds up some candidate solution &

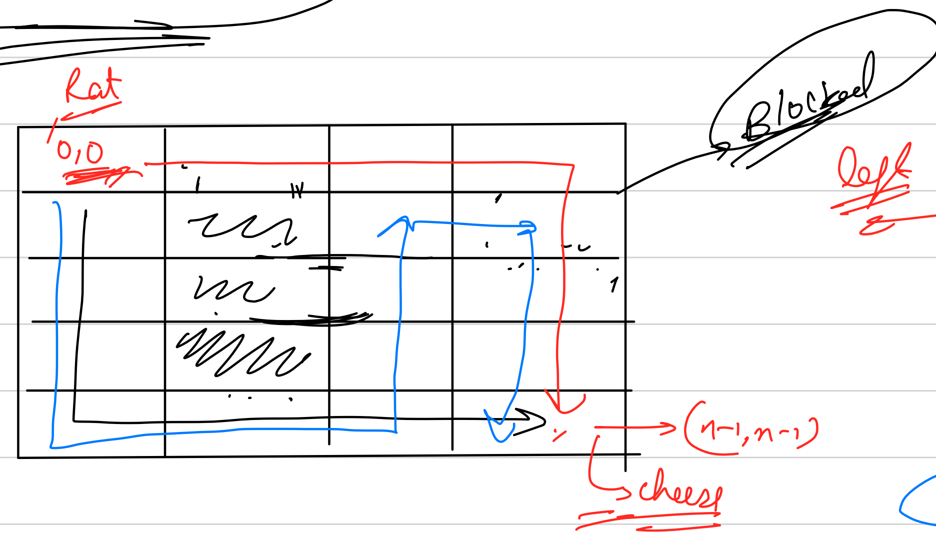
prune

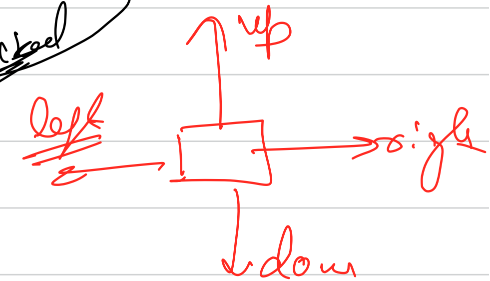
abandons the ones which don't fulfill it.

It ensures, that whenever we are explore a new path for the solution, we have the original problem in the original state.

Q:- Rat in a Maze



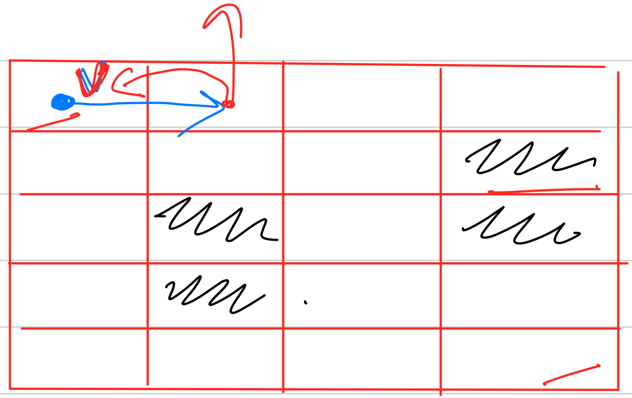


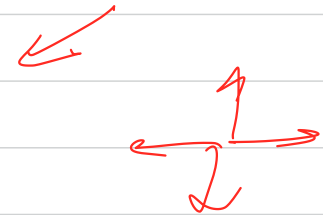


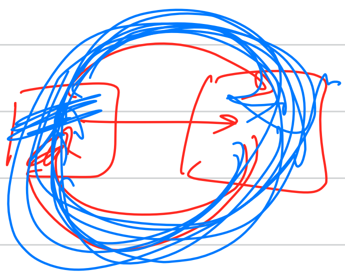
No. of ways

(0,0) → (n-1, n-1)

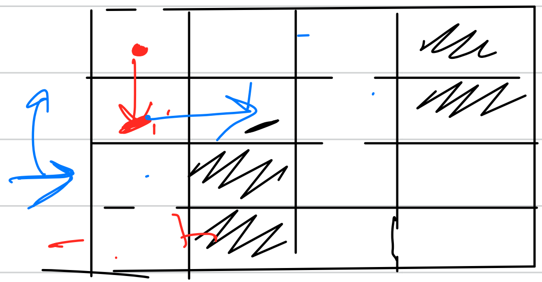
cell → visited







visited arr



UR UR DDDR

1, 2

(1,1)

(1,0) → down

(0,0) → down

visited

(0,0) (1,0)

(1,1)

up

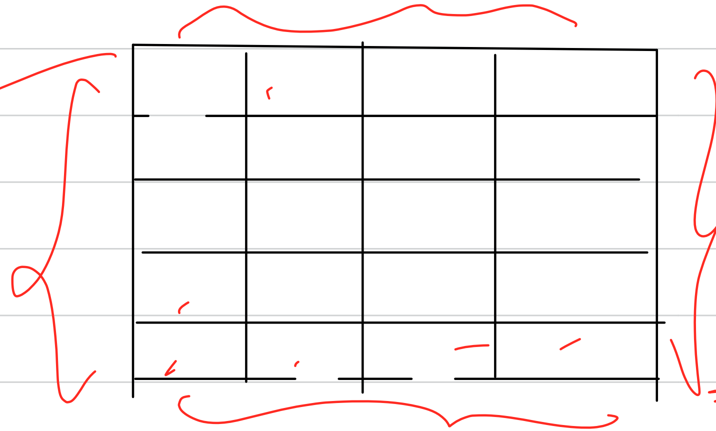
left

down

right

worst

2^n



$n \times n$

space we can go back

no blocked

$4 \times 4 \times 4 \times 4$ — — — — — — n^2 times

100sr 4^{n^2} ✗

100sr 3^{n^2} $\Rightarrow O\left(3^{n^2-(4n-4)} \times 2^{4n-4}\right)$

Ques 2

Q:- array → Calculate all possible subsets

[1, 2, 3]

[ ]

[1]

[2]

[3]

[1 2]

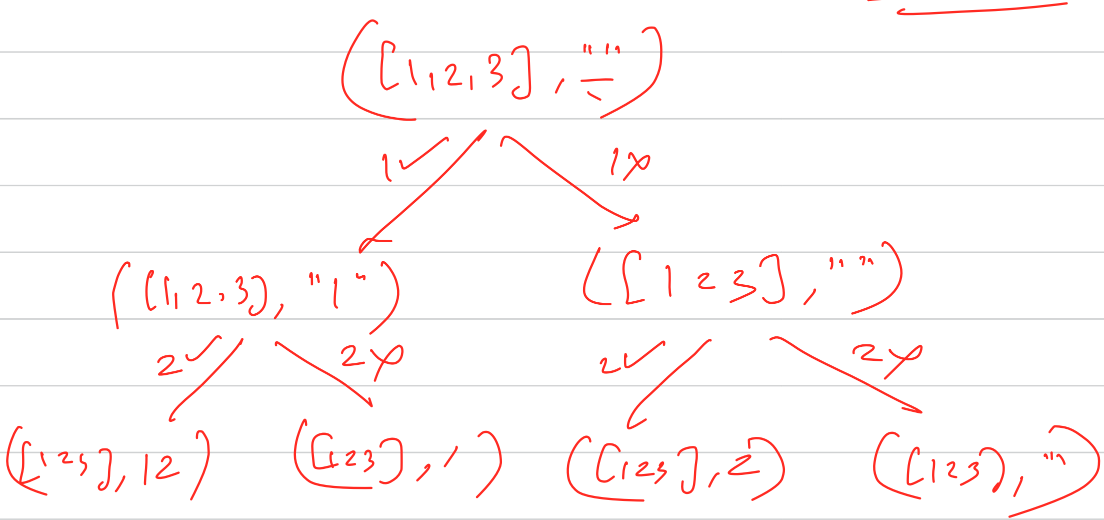
[1 3]

[2 3]

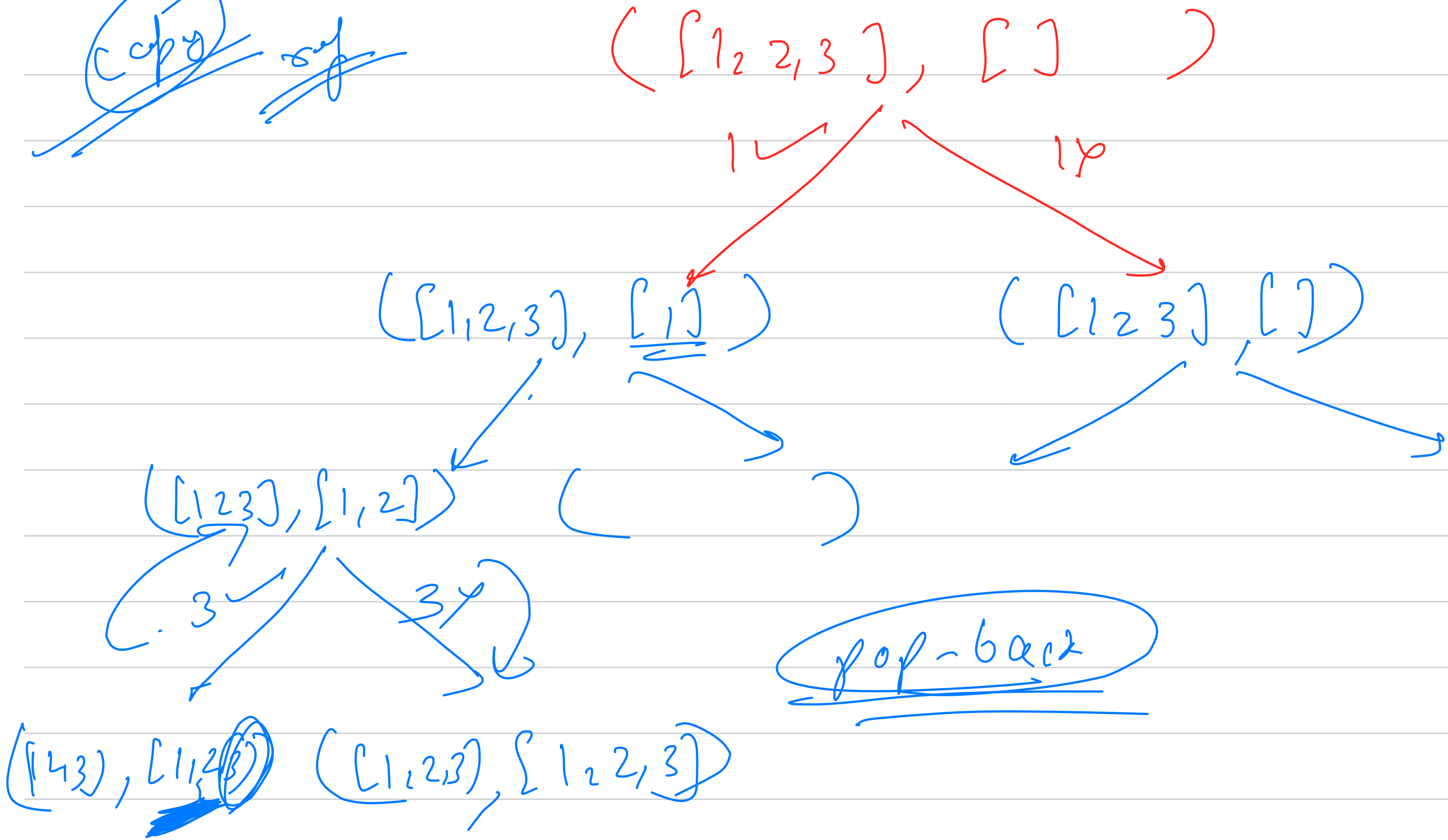
[1 2 3]

$[1, 2, 3] \longrightarrow 2^n$

every element has 2 choices → include
→ excludes



~~copy~~ ~~ref~~



pop-back

→ Base Case → (i==n)

→ Recursive task → Go and calc all subset of arr+1

→ Self work → once include arr[0]
once exclude arr[0]