Backtracking -3

Question: Given an array, generate all the unique permutations

A = [1, 1, 1, 2, 2, 3, 3, 3, 3, 4] N = 10

Total Permutations = 10!

Unique Permutations = $\frac{10!}{3! \times 2! \times 4! \times 1!}$

[1, 1, 1, 2] → 4!

[1, 1, 2] → $\frac{4!}{3!} = 4$

$\frac{10!}{3!\,2!\,4!}$

unordered-map

A = [1, 1, 2] N = 3

$\frac{3!}{2! \cdot 1!} = \frac{3 \times 2}{2} = 3$

ordered-map / tree-map (Vectors as Keys)

[1, 1, 2] [1, 2, 1] [2, 1, 1]

Approach 1:

Generate all permutations and put them into a hashset.

Approach 2:

→ We don't want to even generate the duplicate permutations

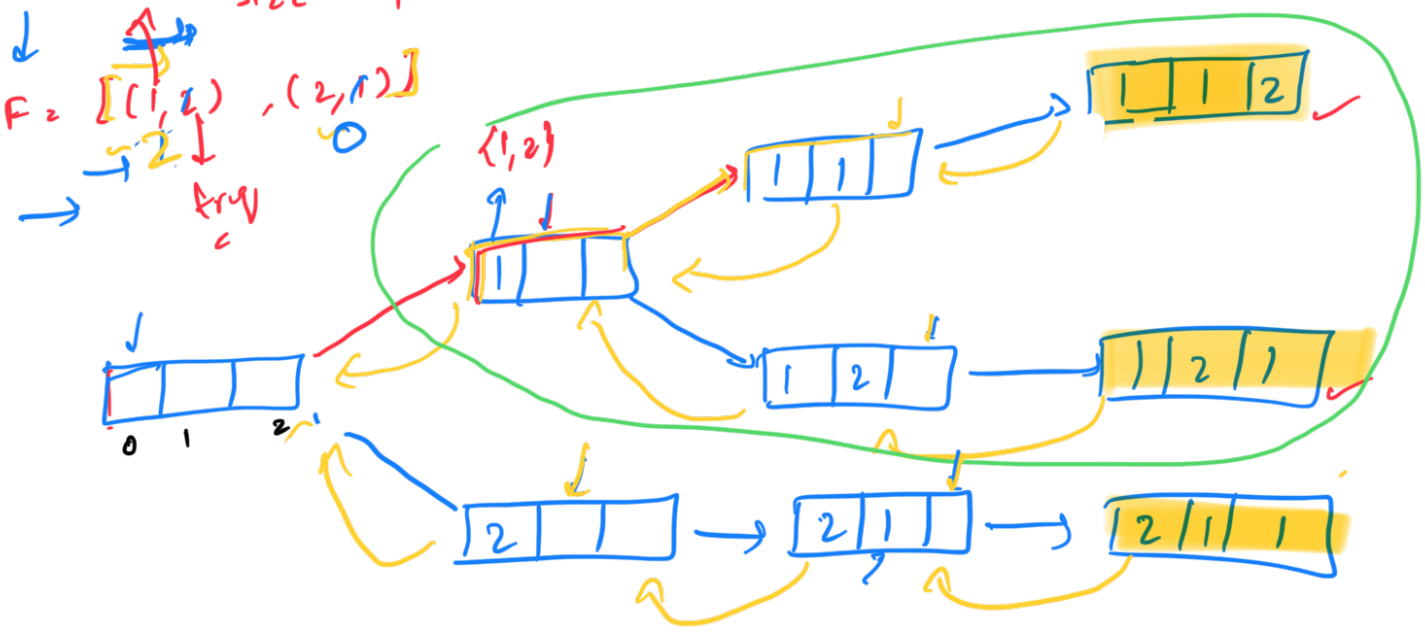
A = [1 1 2]

[(1, 2), (2, 1)]

F =

val freq val freq

ans =



Parameters

1) idx
2) Frequency Array (You can static. Array of size N)
3) temp

```
void generate (idx, F, temp) {
    if(idx == N) {
        ans.push(temp);
        return
    }

    for(i=0; i<F.size(); i++) {
        if(F[i].freq > 0) {

DO {            temp.push(F[i].val);
                F[i].freq--;
                generate(idx+1, F, temp);
                temp.pop();
UNDO {          F[i].freq++;
        }
    }
}
```

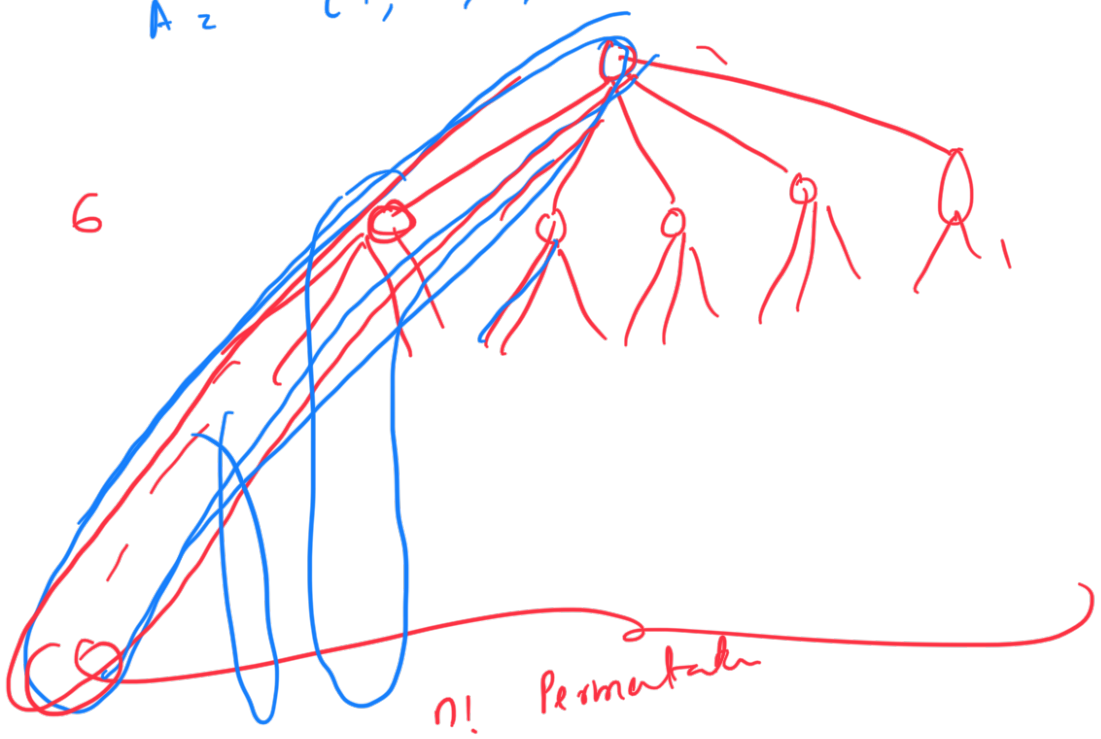
Function Call :

T.C

O(F.size()) => O(n)

}

A = [1, 2, 3, 4, 5, 6]



Total Permutations = $(n!)^6$

No. of nodes in the Path = n

T.C per node = n.

Upper bound : $O(n^2 \cdot n!)$ → Upperbound.

T.C : $O\left(\frac{n!}{r_1! \cdot r_2! \dots r_k!} \times n^2\right)$

frequencies of repeating elements

Doubts Session

Bookmarks

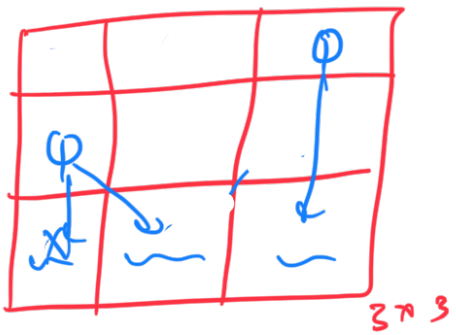
Question : N Queens

... in a N×N board s.t

Arrange N queens such that no 2 queens can kill each other. Print all possible boards.

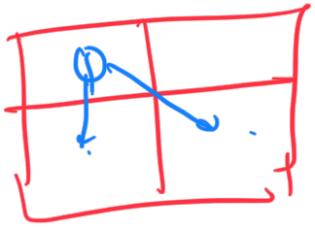
Moves : Any no. of steps :
1) Horizontally
2) Vertically
3) Diagonally

N = 3



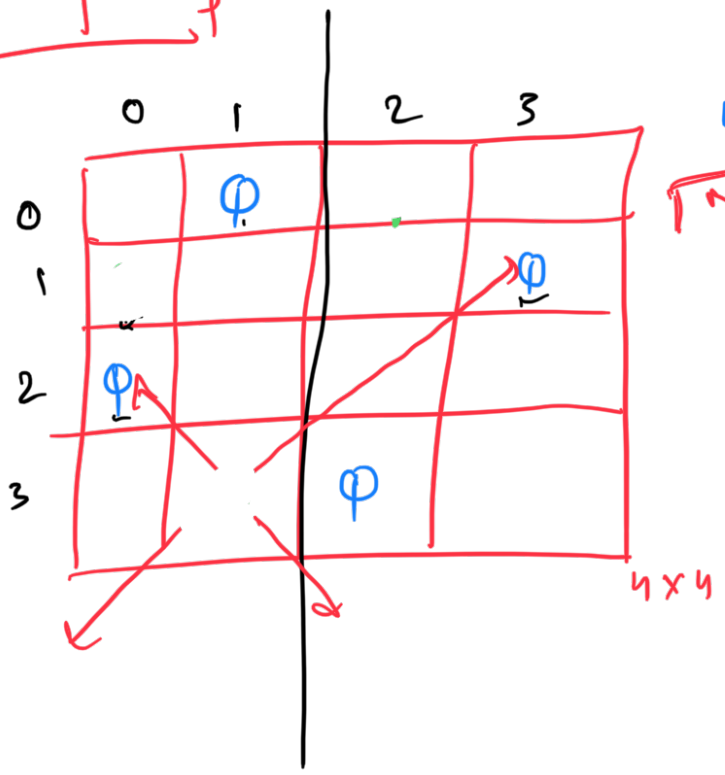
Place 3 queens
Not Possible

N = 2



X

N = 4



4 Queens
N Queens
4 rows
4 columns

Observations

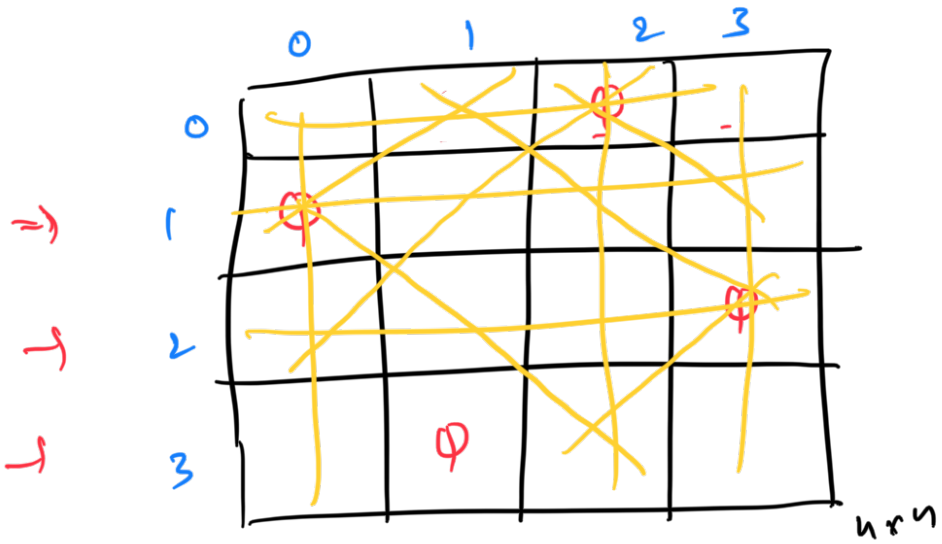
each rows

1) Only 1 queen in each column

2) Only 1 queen at max in any row

3) Only 1 queen in each diagonal.

4 queens



→ Tracking eliminated cells gets tricky

Parameters:

row ⇒ Row Num

N ⇒

board[][]

```
void nQueens(row, N, board) {
  if(row == N) {
    Print board;
    return;
  }

  for(col = 0; col < N; col++) {
    if(possible to place at (row, col)) {
      board[row][col] = 1;
      nQueens(row+1, N, board);
      board[row][col] = 0;
    }
  }
```

DO
RECURSION
UNDO

CONDITIONS:

Conditions to place a Queen in i^{th} row

↳ No Queen should be present in that row.

1) Iterate over the i^{th} row and check → O(n)

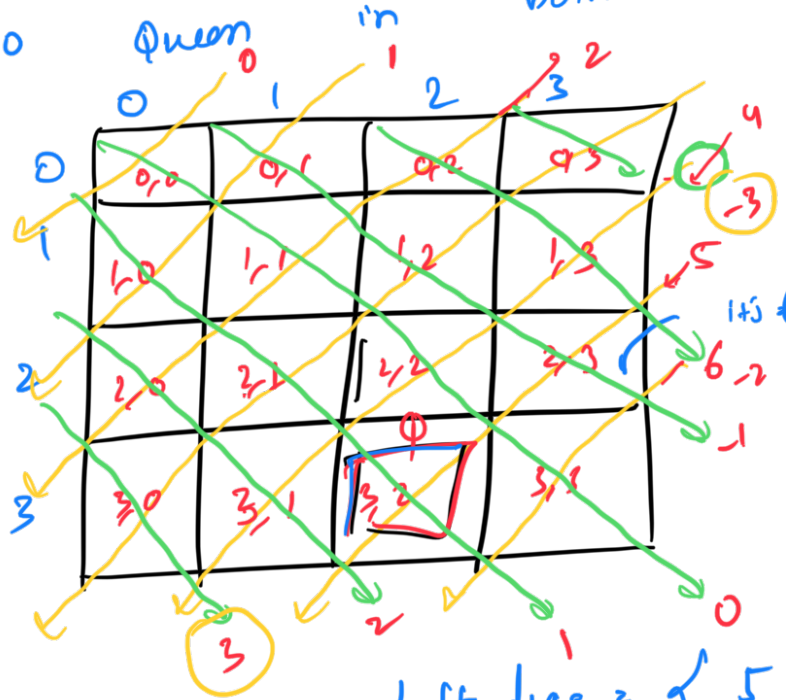
2) Hashset (Array of size n) (i) arr[i]

→ Place a Queen in j^{th} column

1) Iterate the col O(n)

2) Hashset

→ No Queen in both the diagonals



Left_diag = { 5 }

right_diag = { }

Lecture Notes

(row, N, board) {

void

```
for(col = 0; col < N; col++) {
    if( row not in row_set &&
        col not in col_set &&
        (row+col) not in left_diag_set &&
        (row-col) not in right_diag_set) {

        board[row][col] = 1;
        row_set.insert(row);
        col_set.insert(col);
        leftdiag_set.insert(row+col);
        rightdiag_set.insert(row-col);
        nQueens(row+1, N, board);
        board[i][j] = 0;
        // Remove from all the sets
    }
}
```

DO {

UNDO {

1D and 2D Indices of Matrix

Row-Major

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

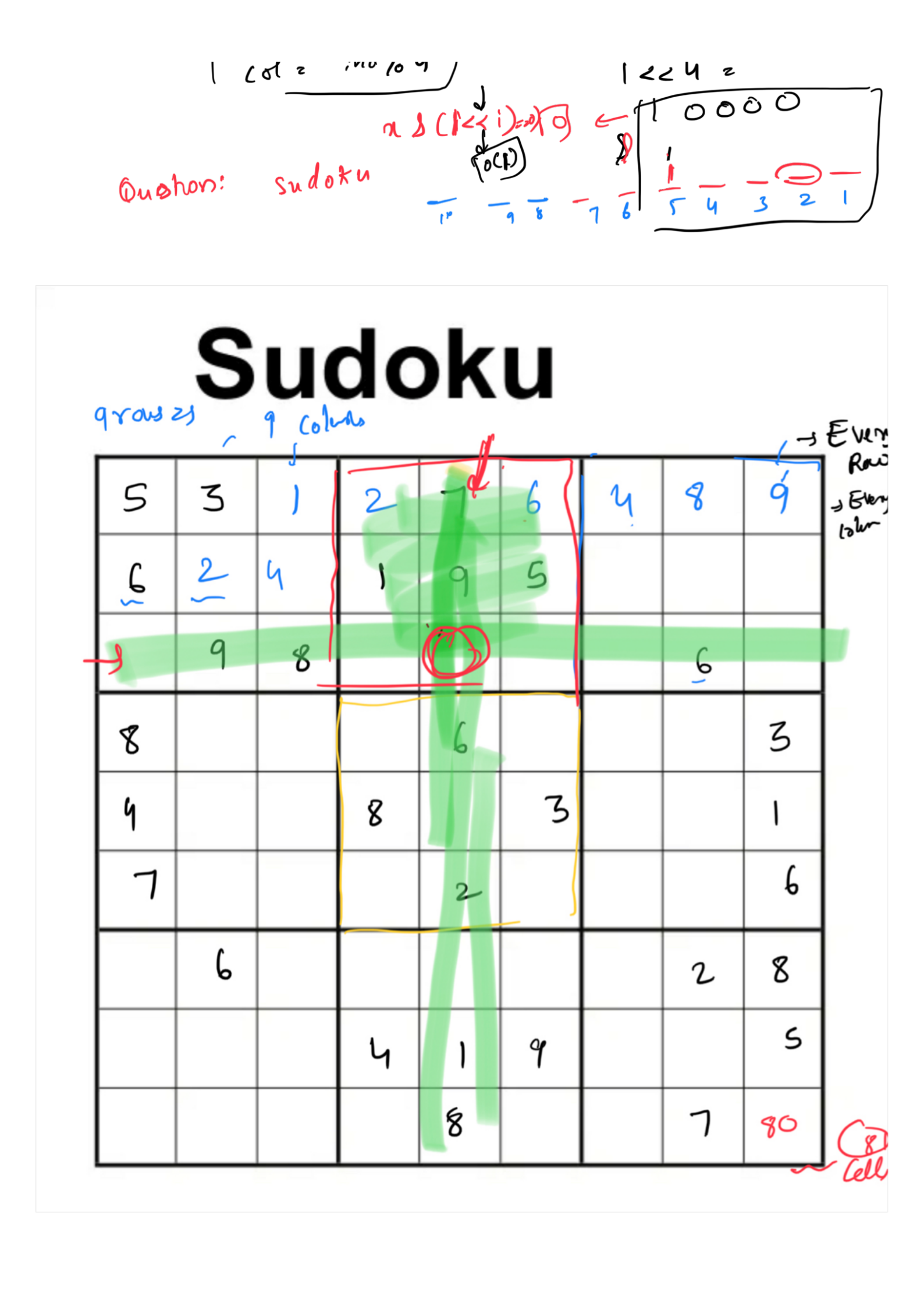
4 x 4

10 => (2,2)

13 => (3,1)

row = ind / 4

ind % 4


Sudoku
9 rows
9 columns
Every Row
Every Column
Cell

Parameters:

ind : 1D index of 2 matrix

board: 2D matrix

```
validSudoku(ind, board){
    if(ind == 9*9) {                    // N^2
        print(board);
    }
    row = ind / 9;                      // N
    col = ind % 9;                      // N
    if(board[row][col] != 0){
        validSudoku(ind+1, board);
        return;
    }
    for(K=1; K<=9; k++){                // (N)
        if(K is possible at (row, col)){
            board[row][col] = K;        // DO
            validSudoku(ind+1, board)
            board[row][col] = 0;        // UNDO
        }
    }
}
```

Approach 1:

→ check the row O(n)

→ check the column O(n)

→ check the block. O(n)

Approach 2:

Maintain { 9 hashsets for 9 rows O(1)

9 " " 9 column O(1)

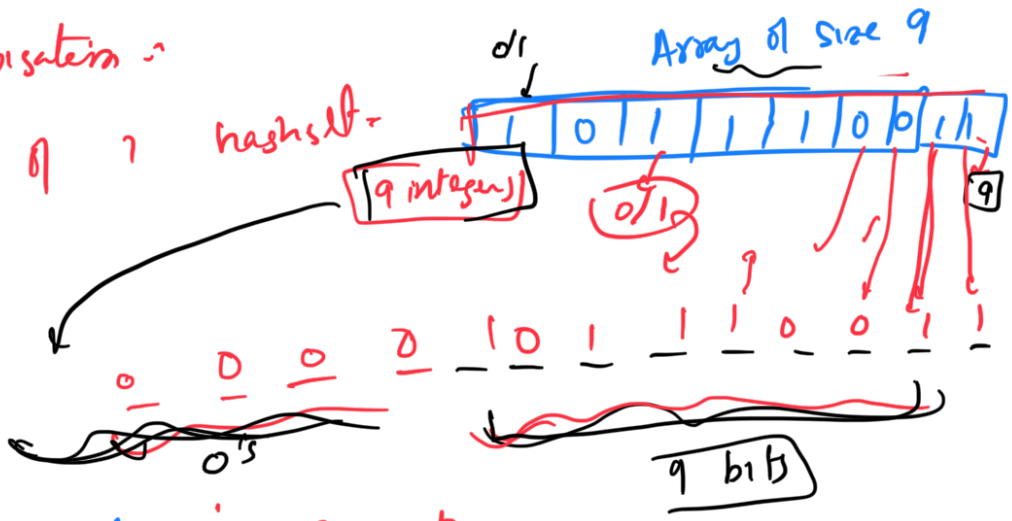
9 " " 9 blocks O(1)

(27×9) integers

S.C : 27 hashsets

Space Optimisation :

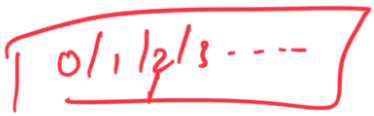
size of 1 hashset



Integer =>

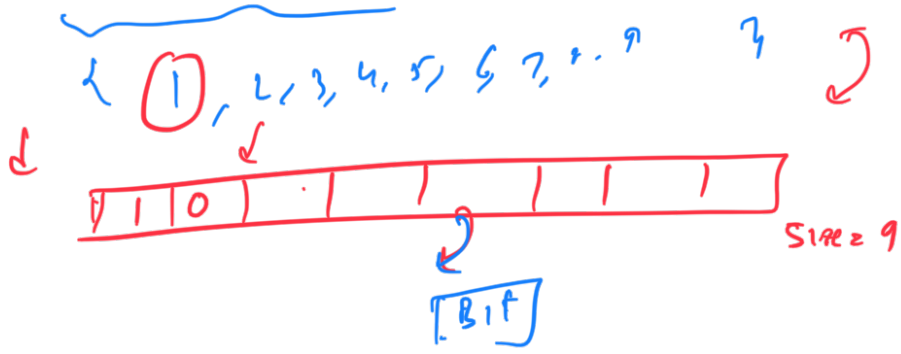
New space : 27 integers

Way better than



hashset { 1, 3, 5, 7 }

Progress Array



Question : Given N, print all the valid parenthesis

N = 1 → 1

( )

N = 2 → 2

( )( )

(( ))

2N

Ans : Length

( )( )

N = 3

( ) ( ) ( )
(( )) ( )
((( )))
( ) (( ))
(( ) ( ))

} 5
Length = 6

How to check if a string is balanced.

s = ((( )))

count = 1 2 3 2 1 0

s = ( ))) ((

count = 0 1 0 -1

Invalid

Approach 1:

Generate all strings of length 2N

N = 3

( ) ( ) ( ) ( ) ( ) ( )

'(' , ')'

2^{2n}

T.C: $O(2n \cdot 2^{2n})$

Observations

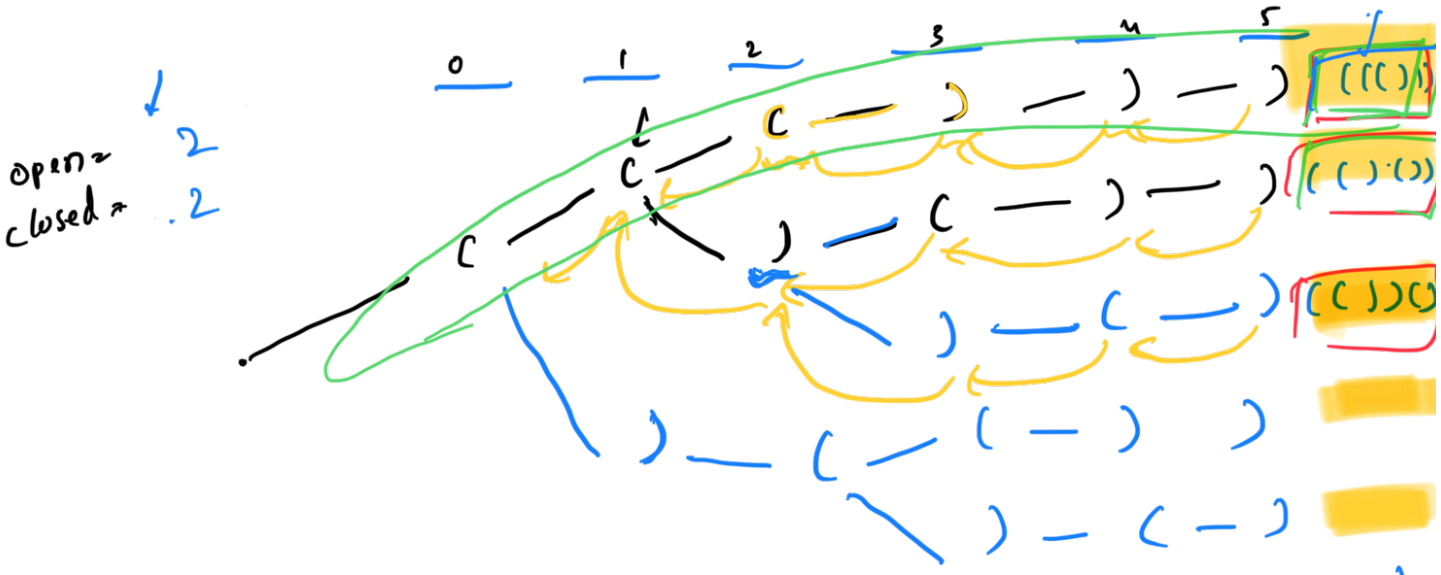
1) open ≥ closed
2) open ≤ n

Strategy

1) if (open < N) we can place open bracket

2) if open > closed, we can place a closed

N = 3




```
void printAll ( ind, n, open, closed, temp ) {
    1) if( ind == 2N ) ans.push(temp);
    2) if( closed == N)

O(1)

    if (open < N) {
        temp.push ( '(' );
        printAll ( ind+1, n, open+1, closed, temp)
        temp.pop();
    }

    if( open > closed) {
        temp.push ( ')' );
        printAll (ind+1, open, closed+1,
                                    temp);
        temp.pop();

    }   }
```

T.C: O(# balanced parenthesis x 2n)

T.C: O(# balanced parenthesis x n)

$N = 3$

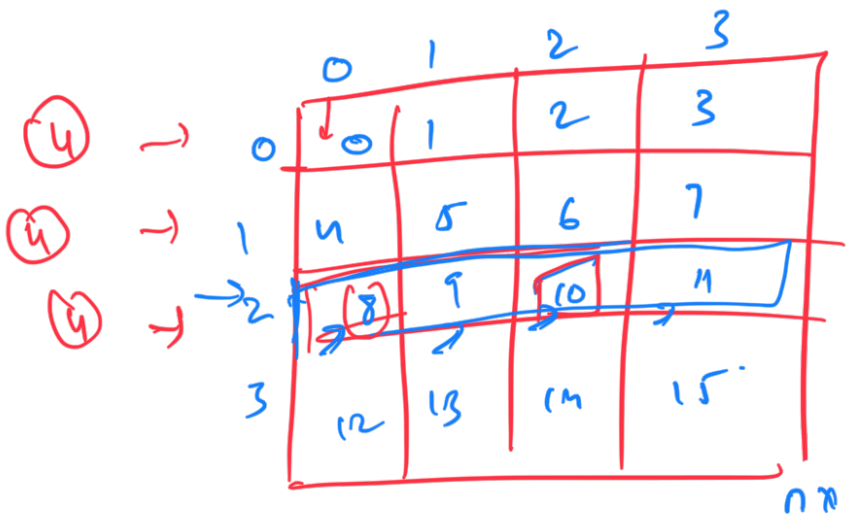
$^{2n}C_n$

$n \neq 1$

$n = 3$

$\frac{6C_3}{4} = \frac{6 \times 5 \times 4}{3 \times 2} \Big/ 4$

$\frac{5 \times 4}{4} = 5$



cols = 4

$\frac{10}{4} = 2$

$\frac{10}{4} = 2$

10%4

10%4

8

n x m

rows, cols

i/m

i/n or i/m ?

row

((( )))

→ ( )) (( )

+1 -1

Return False.

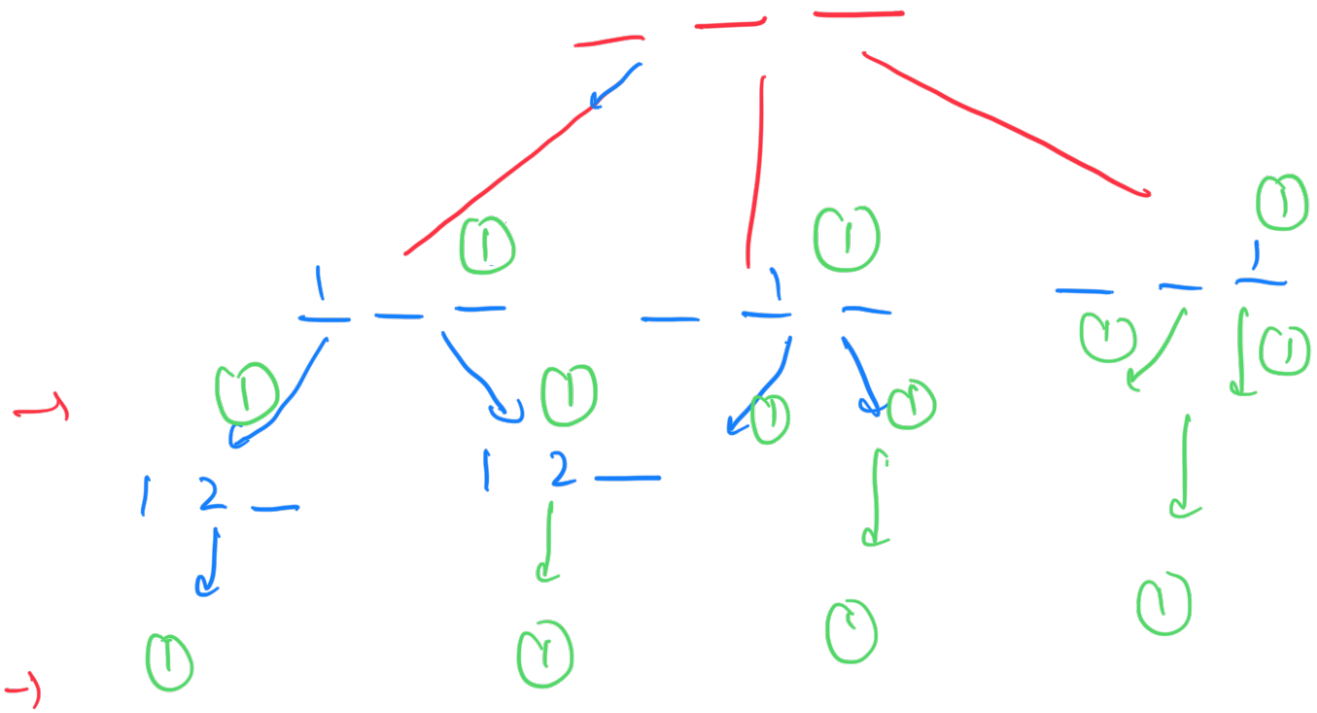
Function calls: $n + n(n-1) + n(n-1)(n-2) + \ldots n!$

$n! \times n$

Upperbound : $n! \times n \times n$ $\leq O(n^2 \cdot n!)$

$N = 3$

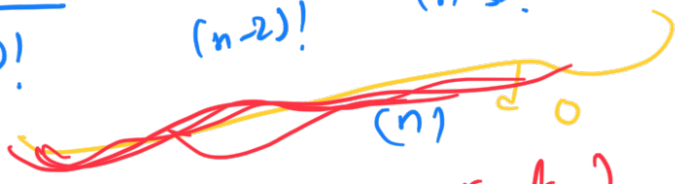
$A = [1, 2, 3]$



$3 + 3.2 + 3.2.1$

$n + n(n-1) + n(n-1)(n-2) + \ldots n!$

$$n! \left[\frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \frac{1}{(n-3)!} + \cdots \right]$$



T.C: $O($ # Function Calls $\times$ T.C Per Function $)$

$\downarrow$

$O(n)$

$(n! \times n)$

$O(n^2 \times n!)$