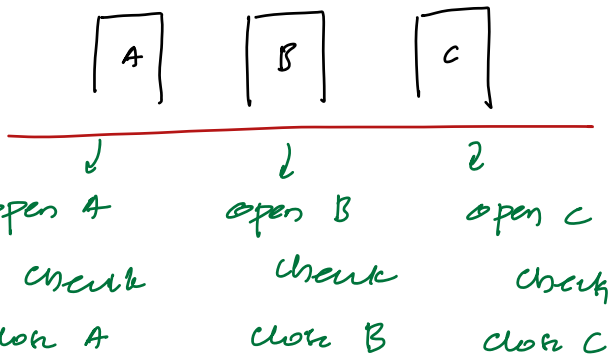
Today's Content:

- ✓ → Back tracking basics
- ✓ → generating all N digit numbers formed by 1 & 2
- ✓ → Number of Subsets with given sum
- ✓ → Print all Subsets with given sum
- ✓ → Generate all Permutations

Back tracking Basics:

→ turn open a box
→ close



→ generate all permutations / Combinations / Solutions with recursion Back tracking

Q) Print/Generate all N digit numbers using 2 digits {1, 2}

N=1 :
1
2

Idea1 : → Using bit manipulation technique we can generate them. {In Doubts Section}

N=2 :
1 1
1 2
2 1
2 2

Idea2 :

N=4

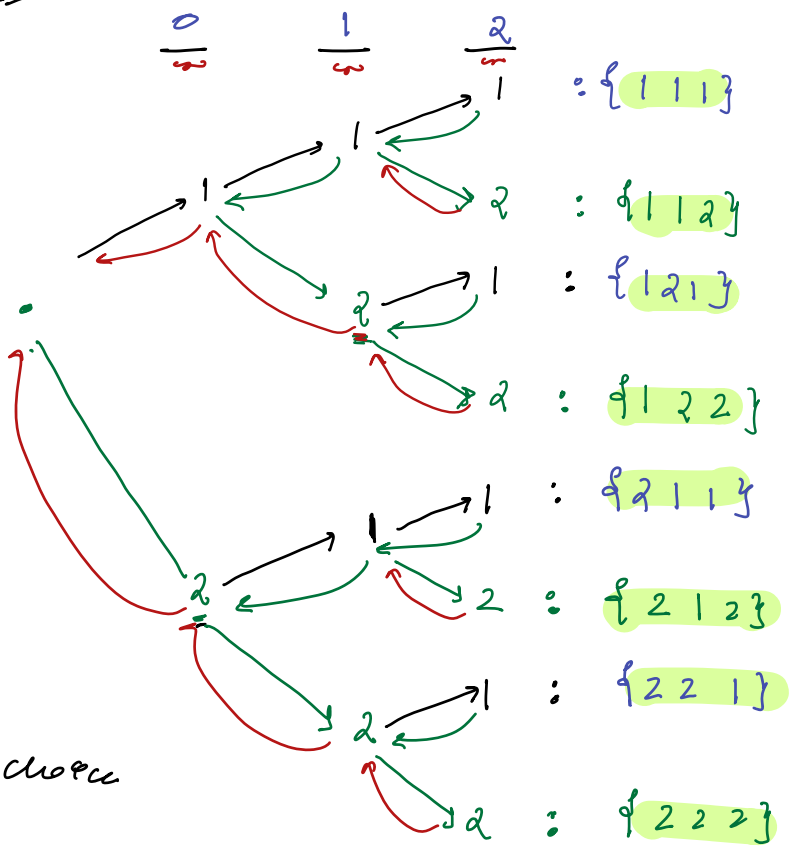
0 1 2 3
— — — —

1 { 3 positions left → {1, 2} }

2 { 3 position left → {1, 2} }

N=3 :
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2

N=3 → 3 digit number



→ parameters :

I, N, ar[N]

At every step : We have 2 choices

→ 2 function calls :

```
void generate(int ar[], int N, int pos)
{
    if(pos == N) {
        iterate & print arr[]      // TC: O(N)
        return;
    }
    ar[pos] = 1;
    generate(ar, N, pos+1)
    ar[pos] = 2;
    generate(ar, N, pos+1)
}
// generate(ar, N, 0)
```

pos → your cursor index position

Recursion Relation :→ { } ← can verify & give you recursive relation

$T(N) = T(N-1) + T(N-1) + 1$

$T(N) = 2T(N-1) + N$

$T(0) = N$

{ at base condition we just print and return }

TC :

N digits :

How many N digit → 2^N

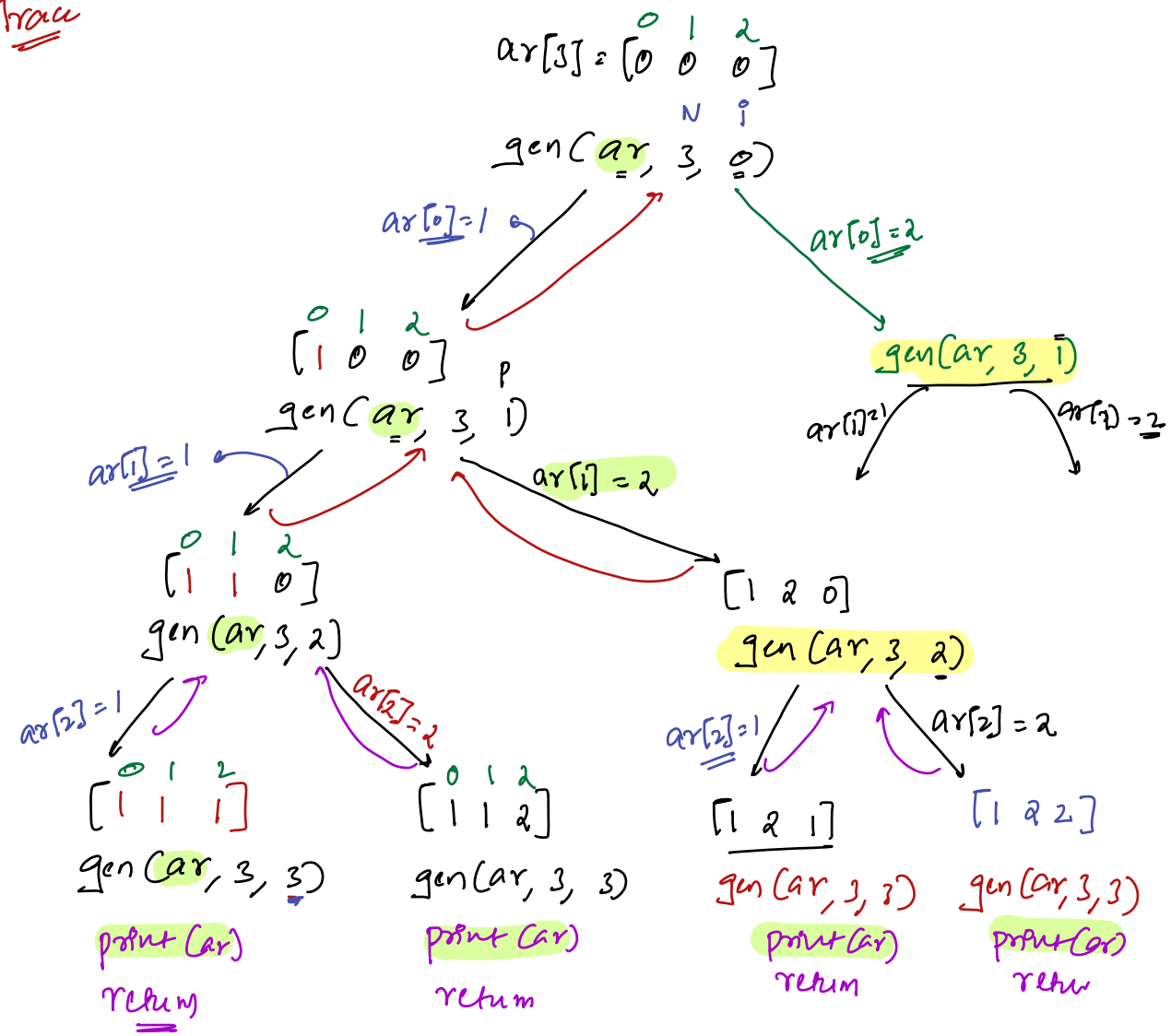
TC : $2^N * \{N \text{ time}\}$

TC : $N * 2^N$

SC : $O(N + N)$ ← array we are using; ↳ pass array by reference

Stack Space

Trace



1, 1 1

1 1 2

1 2 1

1 2 2

Print/Generate all N digit numbers using {1, 2, 3, 4, 5}

→ generate(ar, 3, pos)

```
void generate(int ar[], int N, int pos)
{
    if(pos == N) { Iterate & print arr[]; return; }   TC: O(N)

    ar[pos] = 1;
    generate(ar, N, pos+1)

    ar[pos] = 2;
    generate(ar, N, pos+1)

    ar[pos] = 3
    generate(ar, N, pos+1)

    ar[pos] = 4
    generate(ar, N, pos+1)

    ar[pos] = 5
    generate(ar, N, pos+1)
}
```

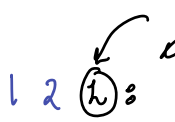
```
for(i = 1; i <= 5; i++){
    ar[pos] = i;
    generate(ar, N, pos+1)
}
```

TC:

5	5	5	5	5

→ How many N digits {1 2 3 4 5}

→ $5^N * N$ → TC: $(5^N * N)$



1 2 (1): 1 2 1

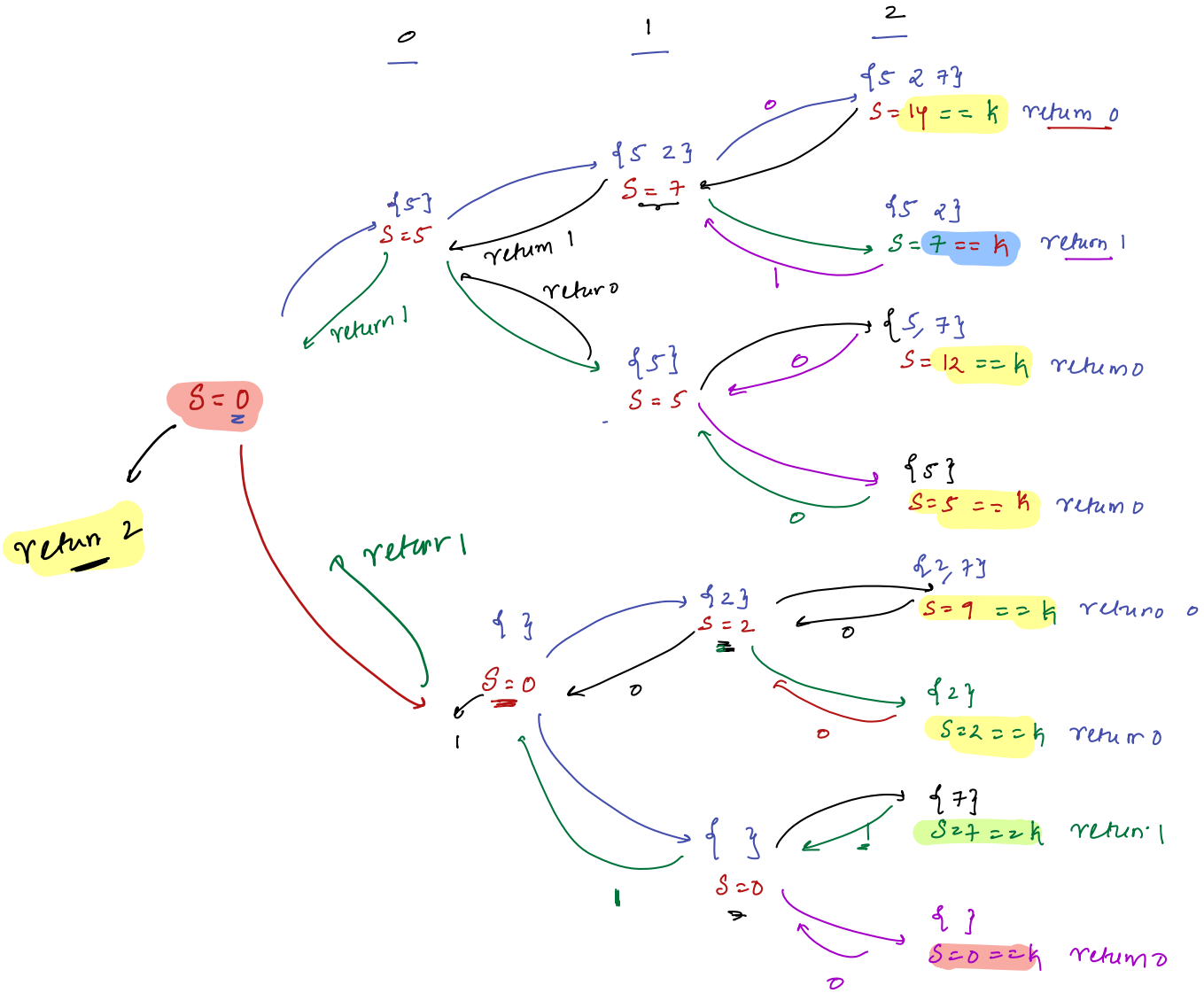
20) Given N array elements, Count no: of subsets with sum == k

Eg1: [10 2 7 6 1 5]
0 1 2 3 4 5

k=8 → { {2, 6}, {2, 1, 5}, {7, 1} }

Idea: generate all subsets Sums == k
↳ using bit manipulation
TC: { $2^N * N$ } Technique
↳ Iterative

Eg1:
ar [5 2 7], k=7
0 1 2



parameters

target sum

actual sum

ar[N], N, k, pos, sum

Function calls?

2 function calls

pick pt n leave pt

```
int AllSums(int ar[], int N, int k, int pos, int sum){
    if(pos == N) { if(sum == k) return 1; else return 0 }

    // pick it:
    sum = sum + ar[pos]
    x = AllSums(ar, N, k, pos+1, sum)

    // leave it:
    sum = sum - ar[pos] // Subtract what we added
    y = AllSums(ar, N, k, pos+1, sum)

    return x+y
}
```

$T(N) = 2T(N-1) + 1$

$T(N) = 2^N$

SC: $O(N)$ → only recursive stack

10:52 pm

→ Ex:

0	1	2	3	4
7	0	0	-3	3

{7}

{7 0}

{7 0}

{7 0 0}

Subsets == 7

→ (All balanced parantheses)

→ (Rat in maze)

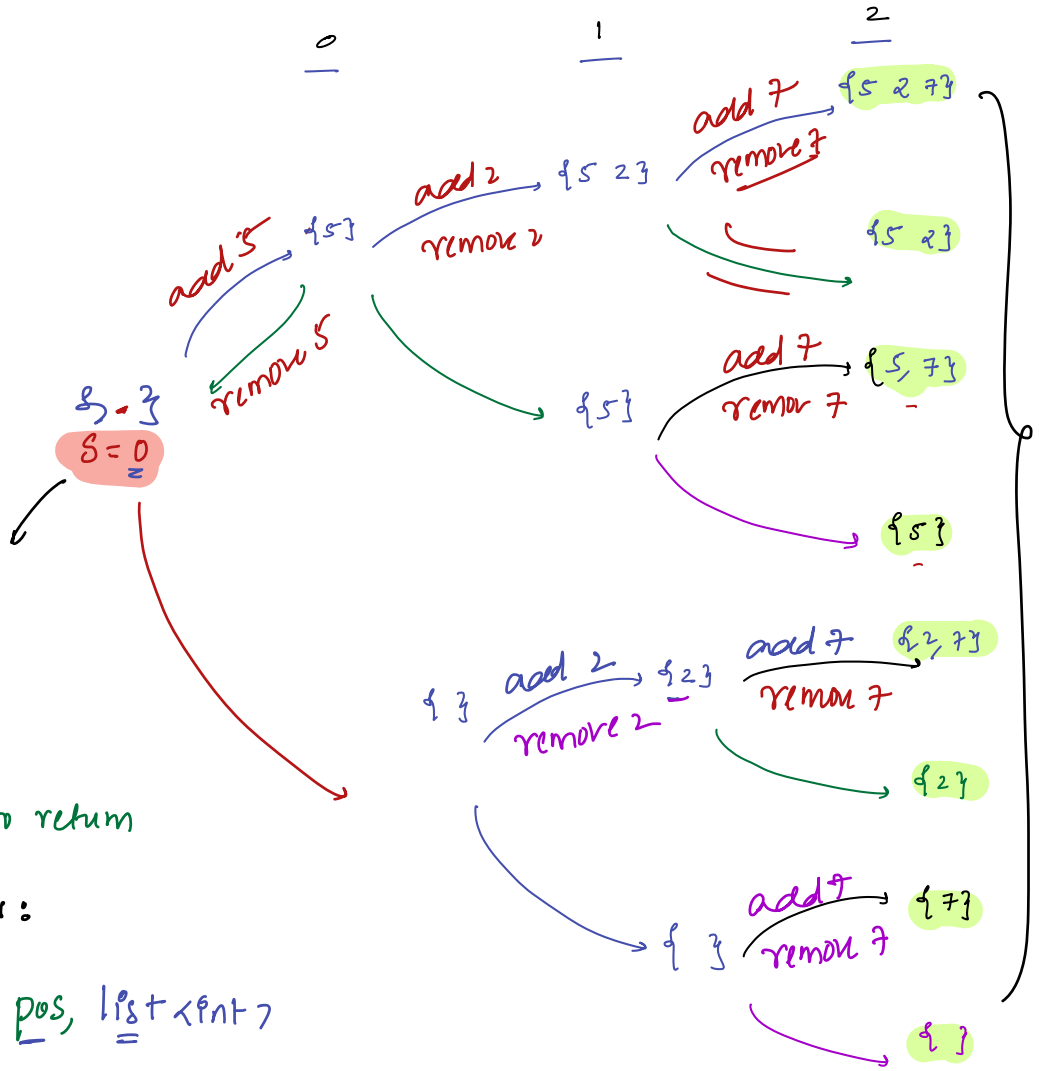
→ (N Queen)

→ (Sudoku)

→ (palindrom partitioning)

→ (graph colouring)

30): print all Subsets



→ no need to return

parameters:

ar[] N, pos, list<int>

function calls:

2

```
printallSubsets( int ar[], int N, int pos, list<int> d ) {
    if (pos == N) { print list d
                    return }

    // pick it
    d.add( ar[pos] )
    printallSubsets (ar, N, pos+1, d)
    d.pop( ) → remove last element
    printallSubsets (ar, N, pos+1, d)
}
```

pass by reference

Base ra

func( )

choice

func( )

another

func( )

// given N distinct elements, print all permutations ?

N = 2 : 8 9 → arrangement of data

8 9

9 8

$\frac{3}{=} \times \frac{2}{=} \times \frac{1}{=}$ → 6 choice → 3!

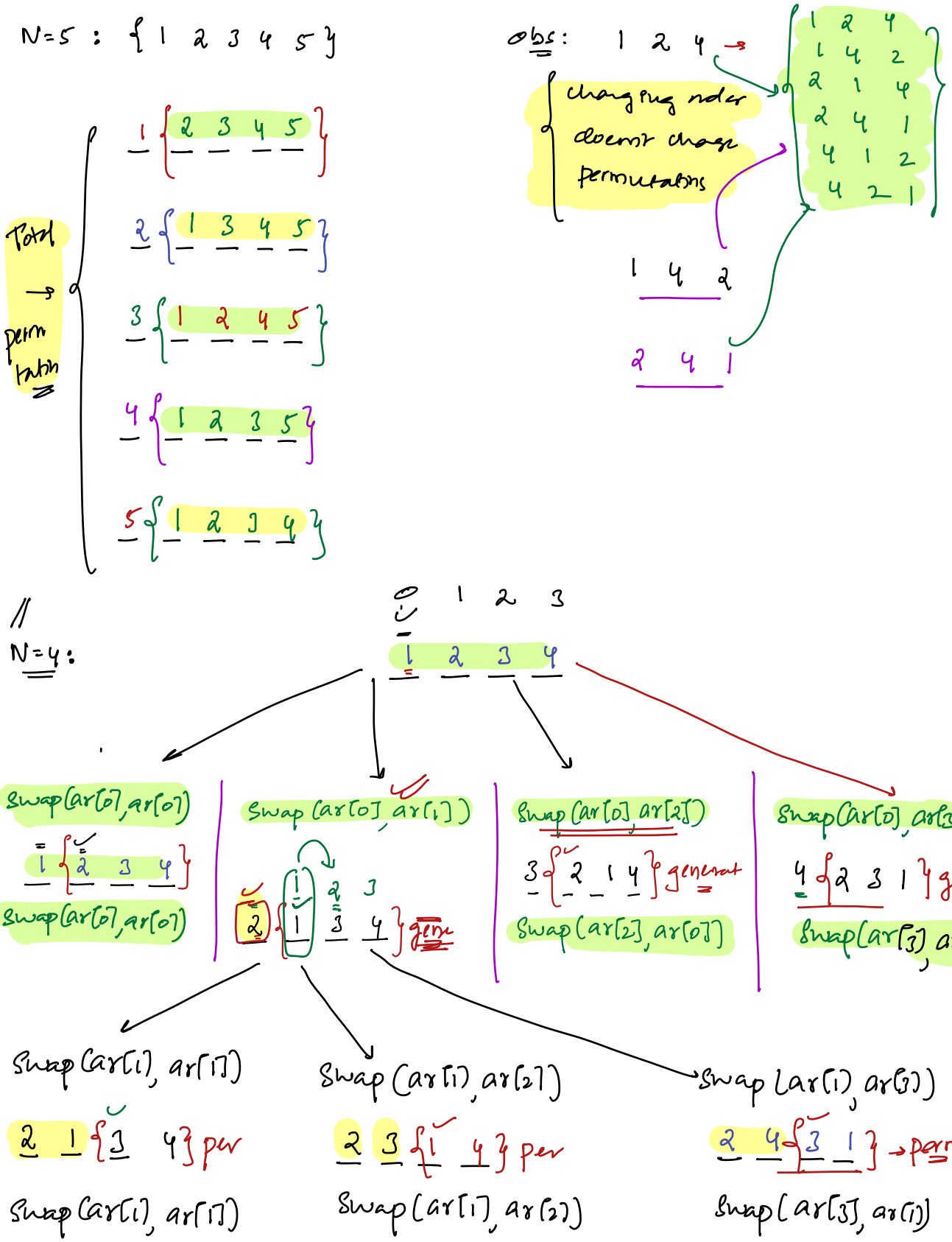
$\{ \begin{matrix} a \\ b \\ c \end{matrix} \} \rightarrow \{2\}$

N = 3 :

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

4 3 2 1 → 4! → 24

N N-1 N-2 — — 1 → N! permuta


N=5 : { 1 2 3 4 5 }
Total → perm tation
1 { 2 3 4 5 }
2 { 1 3 4 5 }
3 { 1 2 4 5 }
4 { 1 2 3 5 }
5 { 1 2 3 4 }
obs: 1 2 4 →
changing order doesn't change permutations
1 2 4
1 4 2
2 1 4
2 4 1
4 1 2
4 2 1
1 4 2
2 4 1
//
N=4:
i
0 1 2 3
1 2 3 4
swap(ar[0], ar[0])
1 { 2 3 4 }
swap(ar[0], ar[0])
swap(ar[0], ar[1])
2 { 1 3 4 } gen
swap(ar[0], ar[2])
3 { 2 1 4 } generat
swap(ar[2], ar[0])
swap(ar[0], ar[3])
4 { 2 3 1 } gener
swap(ar[3], ar[0])
swap(ar[1], ar[1])
2 1 { 3 4 } per
swap(ar[1], ar[1])
swap(ar[1], ar[2])
2 3 { 1 4 } per
swap(ar[1], ar[2])
swap(ar[1], ar[3])
2 4 { 3 1 } → perm
swap(ar[3], ar[1])

Pseudocode:

parameters:

ar[], N, pos

functioncall: ——

We can write 2nd base condition instead of for loop

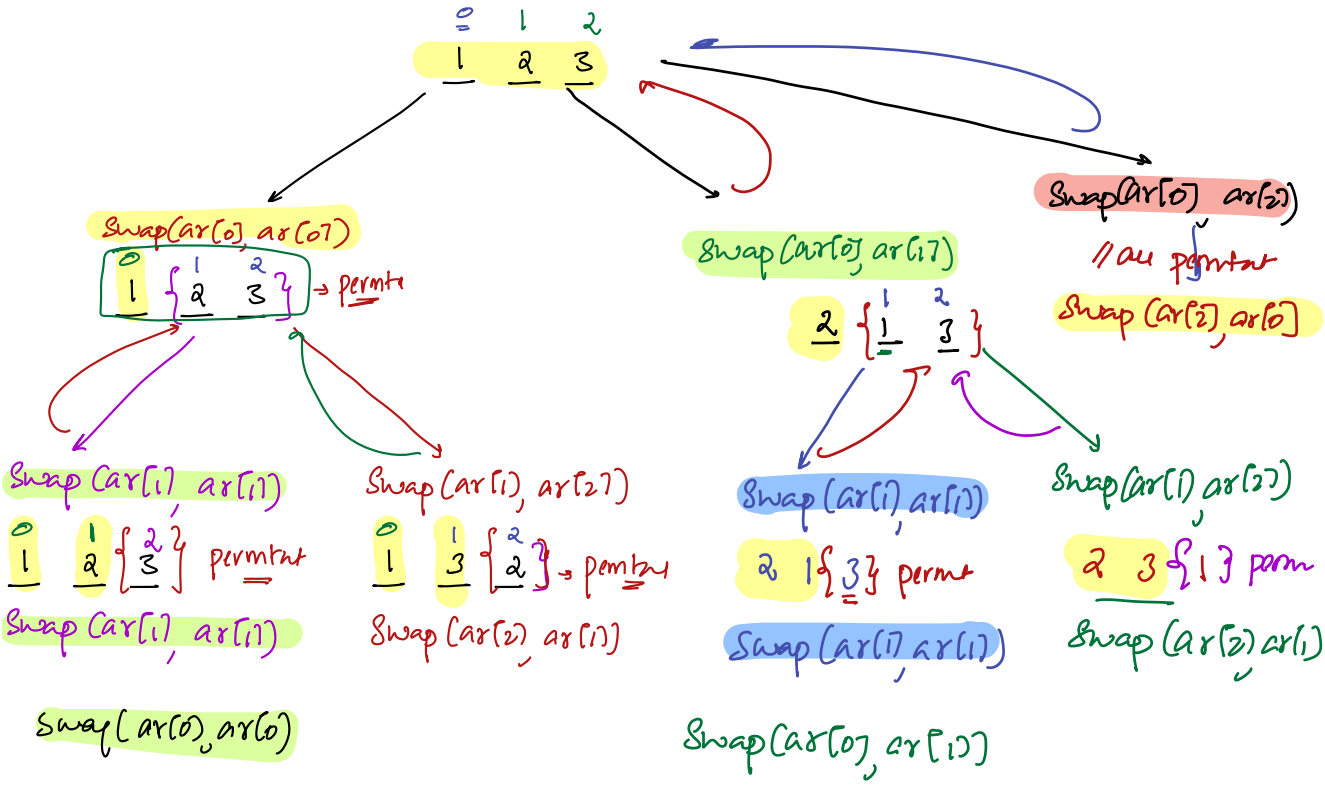
```
void AllPermutations(int ar[], int N, int pos)
{
    if(pos == N) { print(ar[]) return }
    if(pos == N-1) { print(ar[]) return }
    // choice at current pos
    for(i = pos; i < N; i++){
        swap(ar[i], ar[pos]) } at pos data is fixed
        AllPermutations(ar, N, pos+1) // all permutations
        swap(ar[i], ar[pos]) }
}
```

[0 ... pos-1] data is fixed

pos → fix data

TC: N! * {N} → to print each permutation

SC: Stack space: O(N)



→ 1 2 3

1 3 2

2 1 3

2 3 1

// duplicate permutations :

→ generate all permutations from an Set &
get only unique permutations.

friday → Trie
→ Heaps