SCALER

Backtracking

TABLE OF CONTENTS

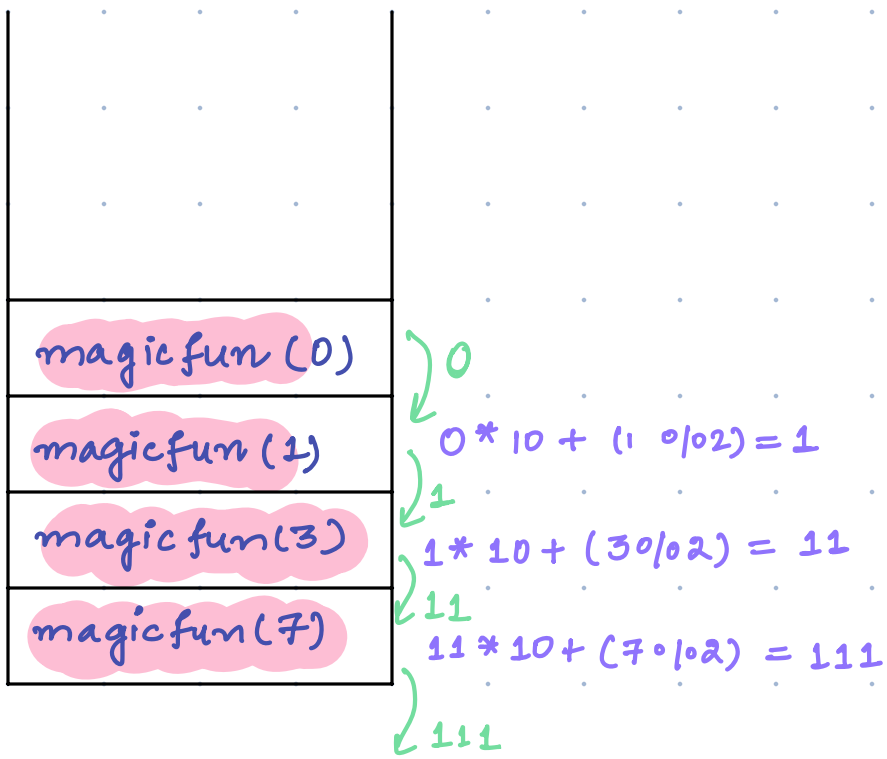
1. Dry run of some recursive functions
2. Backtracking Intro
3. Questions on Backtracking

Class starts at 7:05 IST

QUIZ

What is the output of below code for N = 7 ?

```
int magicfun( int N) {
    if ( N == 0)
        return 0;
    else
        return magicfun(N/2) * 10 + (N % 2);
}
```

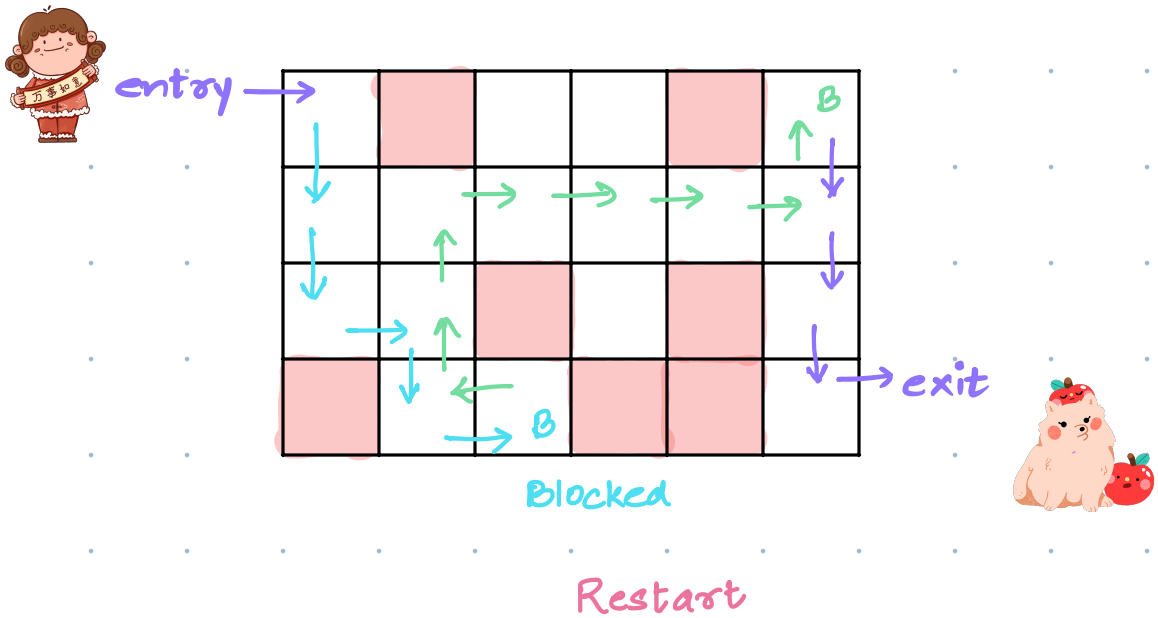


N → N/2 → N/4 - - - - - 0

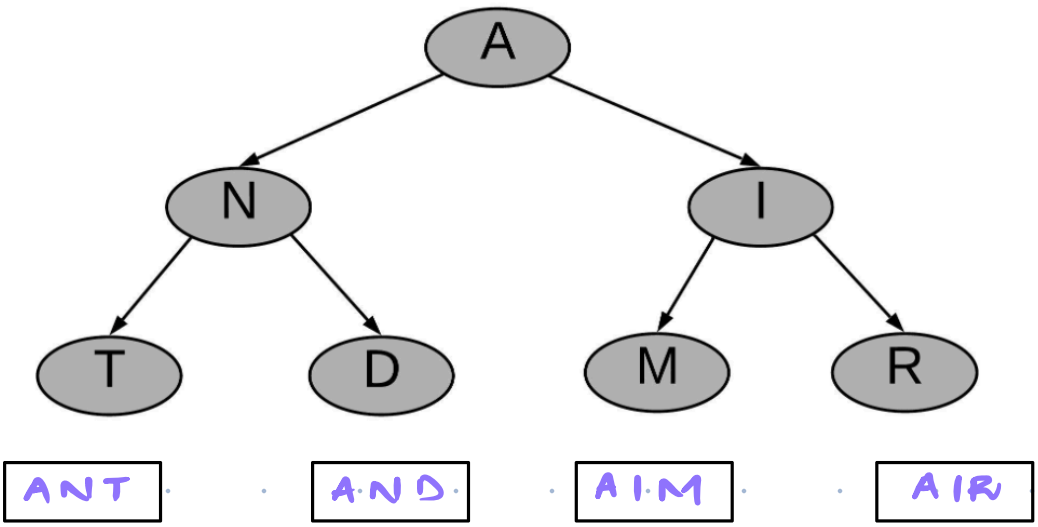
T.C : O(log N)

ans : 111

Backtracking

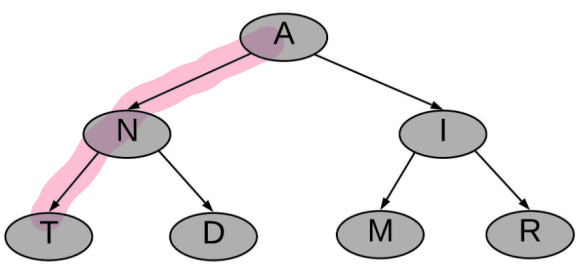


Given a set of words represented in the form of a tree, the tree is formed such that every branch ends in a word. Our task is to find out if a given word is present in the tree or not.

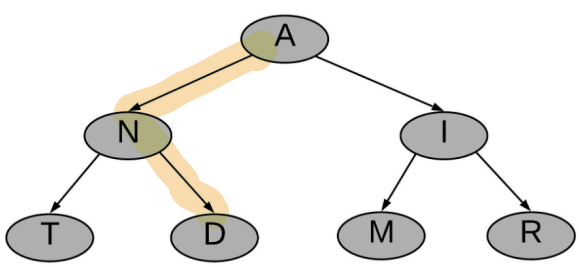


Word to be Searched : AIM

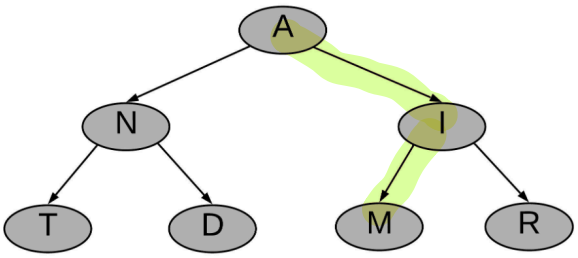
Approach 1 :



ANT ! = AIM



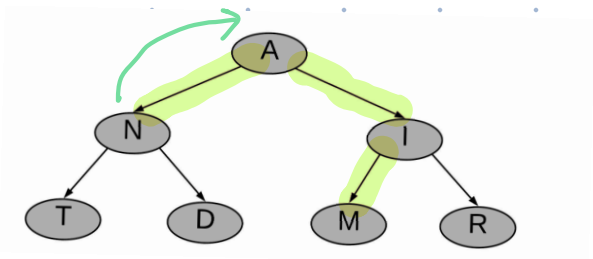
AND ! = AIM



Checking all branches till we get our desired word.

AIM == AIM

Backtracking approach :



AIM

AI != AN

What is backtracking?

Algorithmic Technique of exploring all possibilities using Recursion.

In Backtracking, we begin by choosing an option & explore all possibility until we reach a state where we cann't go any further so now be backtrack & go to next option.

we repeat this step until we get the desired result.

PRUNING : Reducing the search space by discarding the branches that are not worth exploring.

General pattern for Backtracking code :-

```
Solve( ) {
    Base Case

    Option 1
        Recur      or { DO Recur Undo }

    Option 2
        Recur      or { DO Recur Undo }
        |
        |
        |
}
```

Valid Parentheses

Question - Given an integer A, write a function to generate all valid parentheses of length 2*A.

( )

Traversing L → R

c <= #o

Overall

c == # o == A

A = 1 ( )

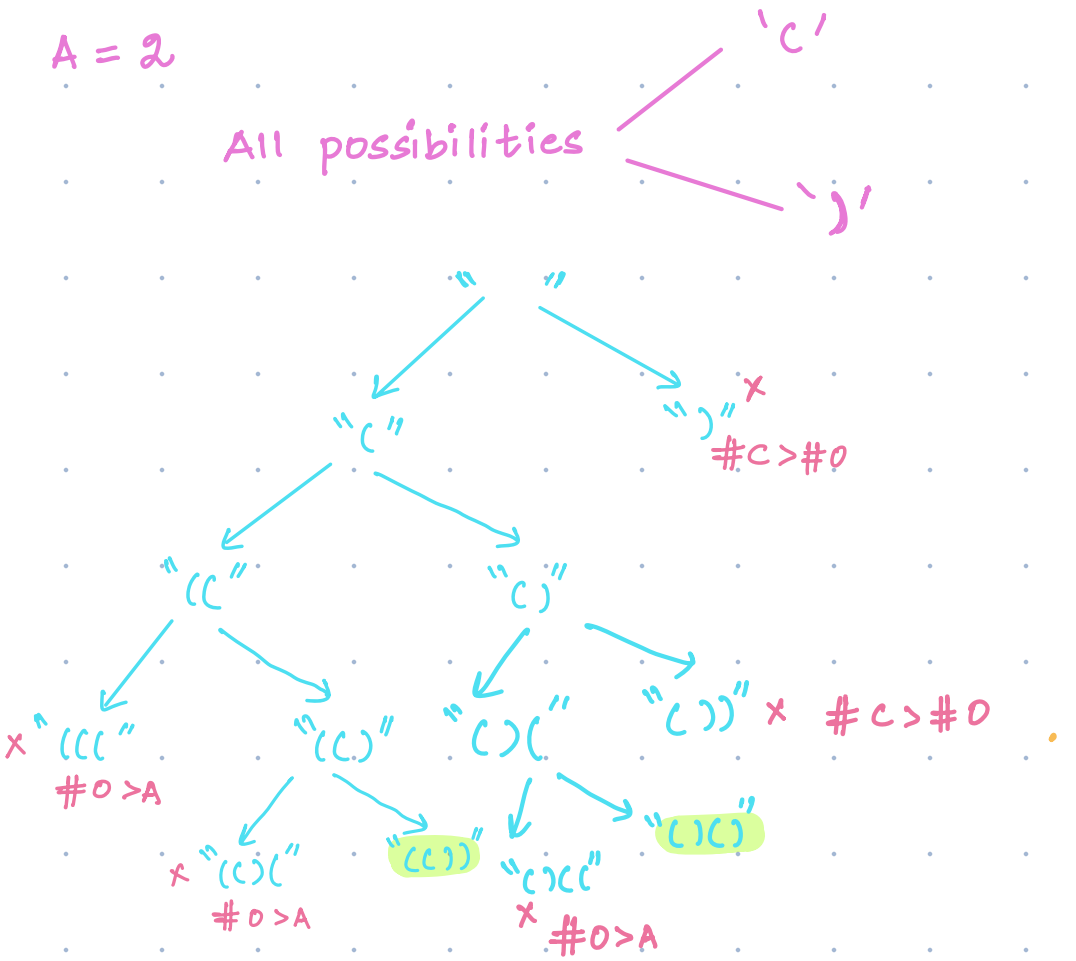
A = 2 (( )) ✓

( ) ( ) ✓

) ( ) (✗

)) ((✗

Approach :



We need to keep track of #o & #c.

solve (A , open_count , close_count)

Conditions for adding "(" and ")"

When open_count < A

Add '(' & recur.

When close_count < open_count

Add ')' & recur.

Base case

str. length == 2A

CODE :

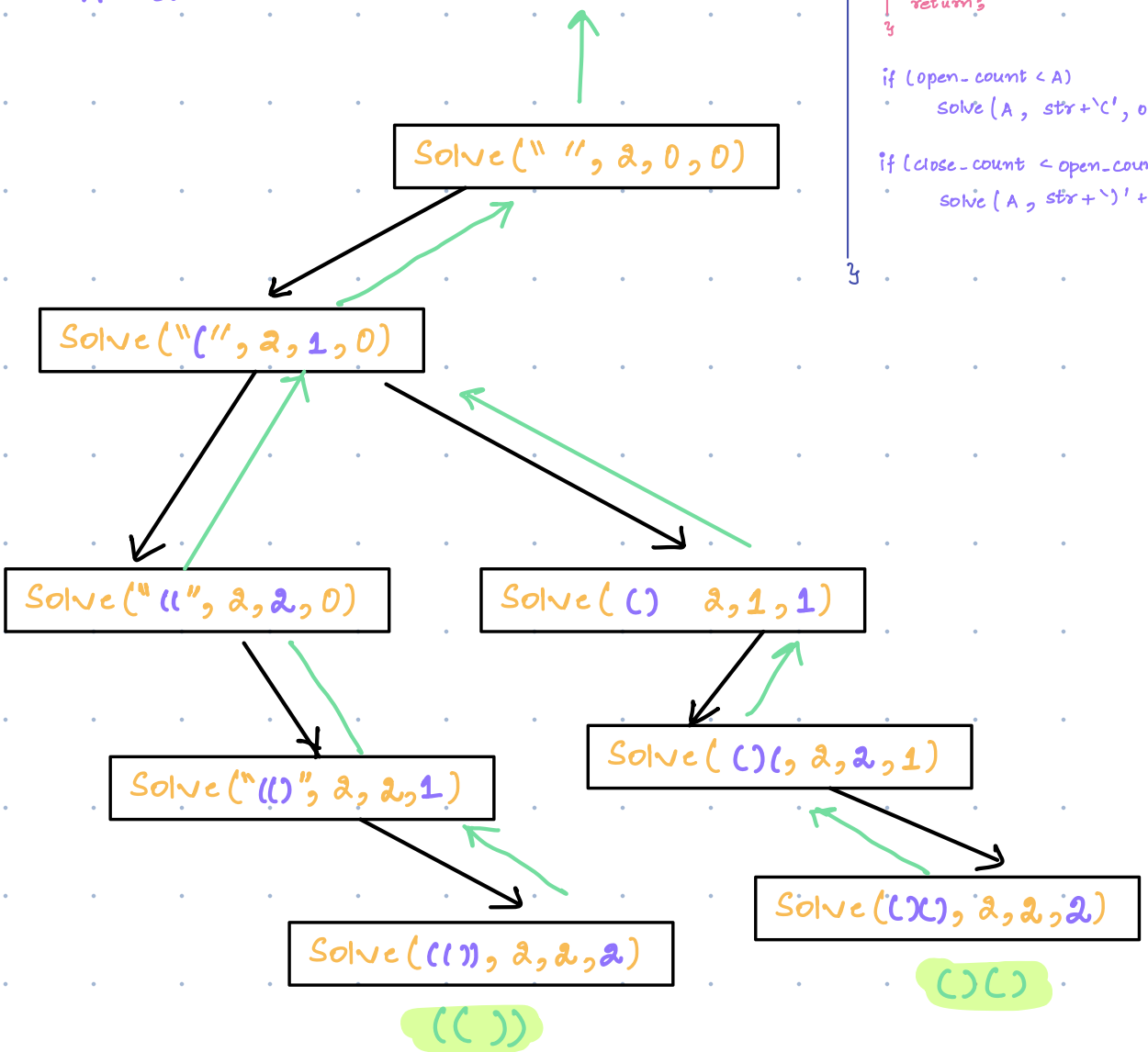
```
solve( A , str, open_count, close_count) {

    // Base case
    if (str.length == 2A) {
        print (str);
        return;
    }

    if (open_count < A)
        solve (A , str + '(', open_count + 1 , close_count);

    if (close_count < open_count)
        solve (A , str + ')' + open_count, close_count + 1);

}
```

T.C $\leq O(2^A)$

S.C $= O(2A) = O(A)$

Dry run

A = 2




```
solve(str, A , open_count, close_count) {
    // Base case
    if (str.length == 2A) {
        print (str);
        return;
    }

    if (open_count < A)
        solve(A, str+'(', open_count+1, close_count);

    if (close_count < open_count)
        solve(A, str+')' + open_count, close_count+1);
}
```

Break till 8:30

Subarray, Subsequence, Subsets

Subarray : Continous part of an array.

Subsequence, Subsets } Non-continuous part of array.

Subsequence	Subsets
• Relative order of elements is maintained.	• Order not maintained.

[5 1]
[1 5] } These are not considered different subsets

1	2	3	4	5

	Subarray ?	Subsequence ?	Subset ?
[1 3 4]	✗	✓	✓
[1 5 3]	✗	✗	✓
[1 2 3]	✓	✓	✓
[ ]	✗	✓	✓
[1 2 3 4 5]	✓	✓	✓
[2 5]	✗	✓	✓
[4]	✓	✓	✓
[5 1]	✗	✗	✓
[4 3 2]	✗	✗	✓

- Subarray is a subsequence but not vice-versa
- Subsequence is a subset but not vice-versa.
- Subarray is a subset but not vice-versa.

Subsets

Question - Given an array with distinct integers. Print all the subsets using recursion.

{ }
{ a }
{ b }
{ c }
{ a b }
{ a c }
{ b c }
{ a b c }

Observations:

Each element has 2 options
- Include in set
- Exclude in set

_ _ _
a _ _
_ b _
_ _ c
a b _
a _ c
_ b c
a b c

2 x 2 x 2 = 8

n elements

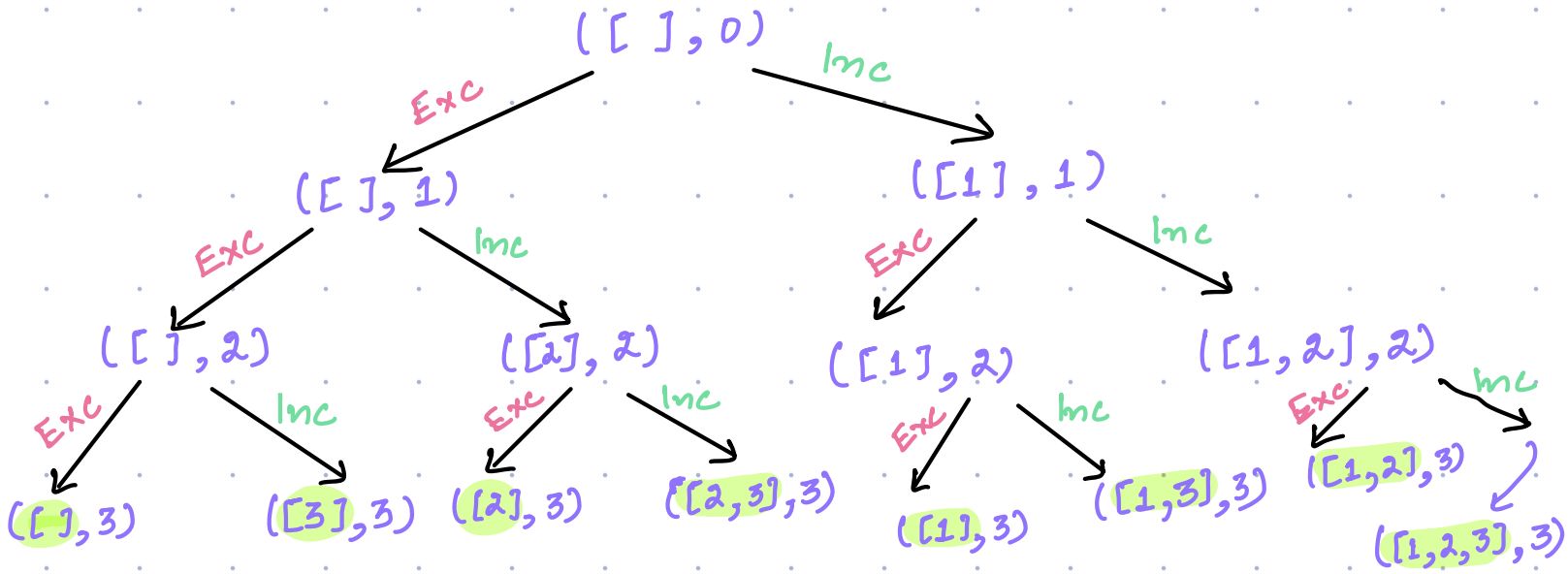
2 x 2 x 2 - - - 2 = 2^N

Total Subsets : 2^N.

0	1	2
1	2	3

(current_set, index)

Approach :



Base condition

index == N

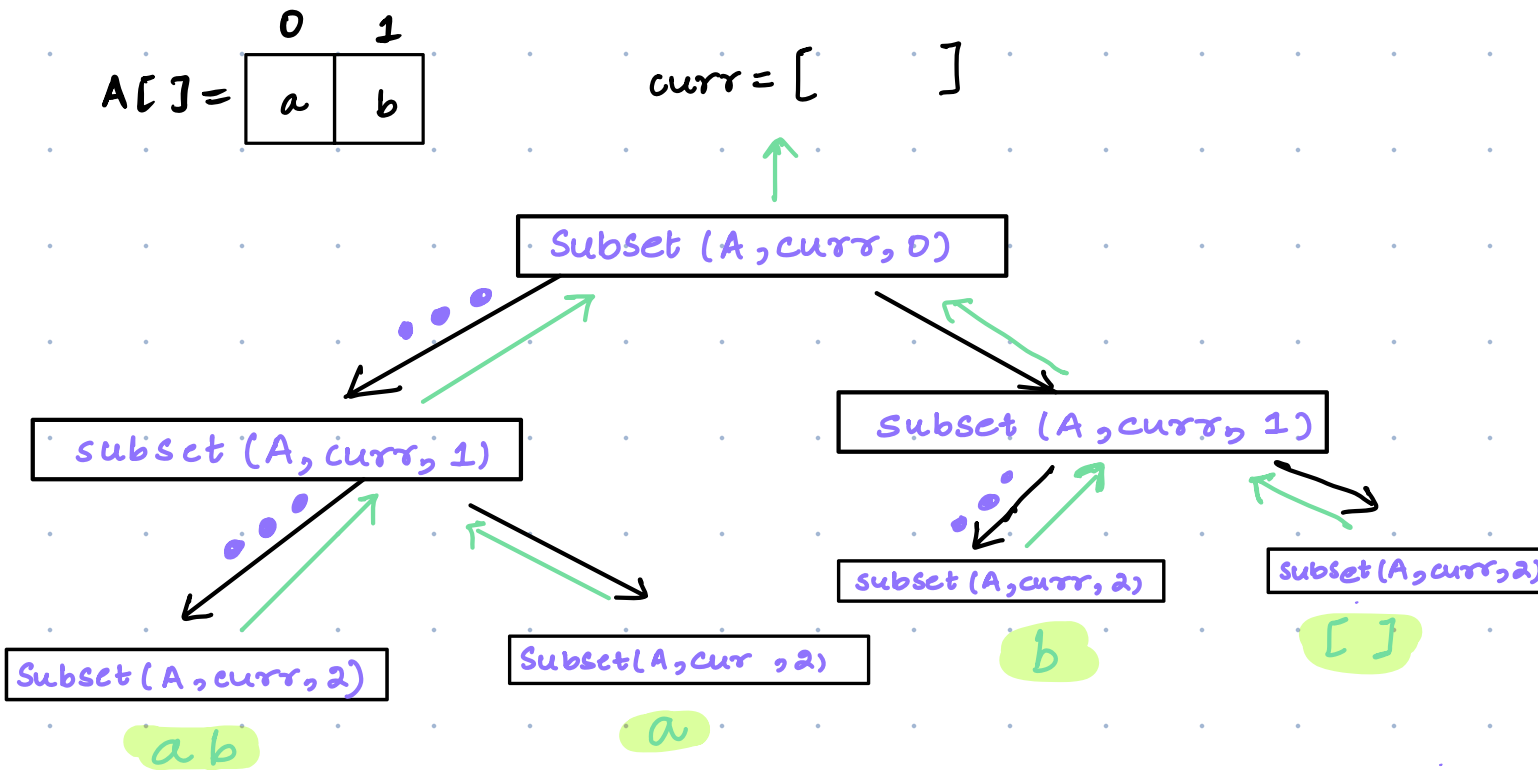
CODE

```
Subsets ( A[] , curr[] , index) {

    // Base Case
    if (index == A.size()) {
        print (curr[]);
        return ;
    }

    // Option 1 : Include element in current set.
        curr. add (A [index]);          } Do
        subset ( A[], curr[], index + 1) ; } Recur
        curr. remove (curr. length()-1); } Undo

    // Option 2 : Exclude element
        subset (A[], curr[], index + 1);

}
```

Dry run :

```
Subset ( - - - ) {
    Base Case

    Option1 : Include
        • Do
        • Recur
        • Undo
    Option 2 : Exclude
        Recur
}
```

```
Subsets ( A[] , curr[] , index) {
    //Base Case
    if (index == A.size()) {
        print (curr[]);
        return ;
    }

    // Option 1 : Include element in current set.
        curr. add (A[index]);          } Do
        subset ( A[], curr[], index + 1) ; } Recur
        curr. remove (curr. length()-1); } Undo

    // Option 2 : Exclude element
        subset (A[], curr[], index + 1);
}
```



T.C : $O(2^N)$

S.C : $O(N)$

A popular Fitness app FitBit, is looking to make workouts more exciting for its users. The app has noticed that people get bored when the same exercises are shown in the same order every time they work out. To mix things up, FitBit wants to show all the different ways the exercises can be arranged so that each workout feels new.

Your challenge is to write a program for FitBit that takes a string A as input, where each character in the string represents a different exercise. Your program should then find and display all possible arrangements of these exercises.

Example:
A = Push-ups
B = Squats
C = Burpees
D = Planks

Then different ways of doing the exercise includes -
ABCD
ACBD
ADBC
ADCB
.
.
etc...

} Permutations

QUIZ!

N = 3

No of ways to arrange 3 balls : _ _ _

3 × 2 × 1 = 6 = 3!

N (N-1) (N-2) — — — 1 = N! → Total no of permutations

Permutations

Question - Given a string with distinct elements. Print all permutations of it without modifying it.

[a b c]

N = 3

3! = 6

a b c

a c b

b a c

b c a

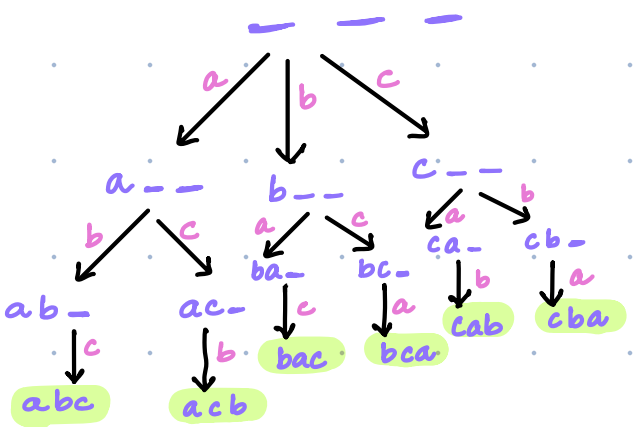
c a b

c b a

Approach :

N = 3

[a b c]



Base case :

all N spots filled.

CODE

```
void permutation (str, index, string ans, boolean [] visited) {
    // Base case
    if (index == str.length) {
        print (ans);
        return;
    }
    for (i = 0 ; i < str.length() ; i++) { // All Possibility
        if (visited[i] == false) { // Valid Possibility
            visited[i] = true ; } Do
            permutations (str, index + 1, ans + str[i], visited) ; } Recurse
            visited[i] = false ; } Undo
        }
    }
}
```

Dry Run :

```
void permutation (str, index, string ans, boolean[] visited) {
    // Base case
    if (index == str.length) {
        print (ans);
        return;
    }
    for (i = 0; i < str.length(); i++) {   // All Possibility
        if (visited[i] == false) {   // Valid Possibility
            visited[i] = true;   // Do
            permutations (str, index+1, ans+str[i], visited);   // Recur
            visited[i] = false;   // Undo
        }
    }
}
```

```
permutation ( - - - ) {
    Base case
    Option 1 : i = 0
        • Do
        • Recur
        • Undo
    Option 2 : i = 1
        • Do
        • Recur
        • Undo
        .
        .
        .
    i = N-1
}
```

A[ ]

0	1
a	b

N = 2

visited → v

0	1
F	F