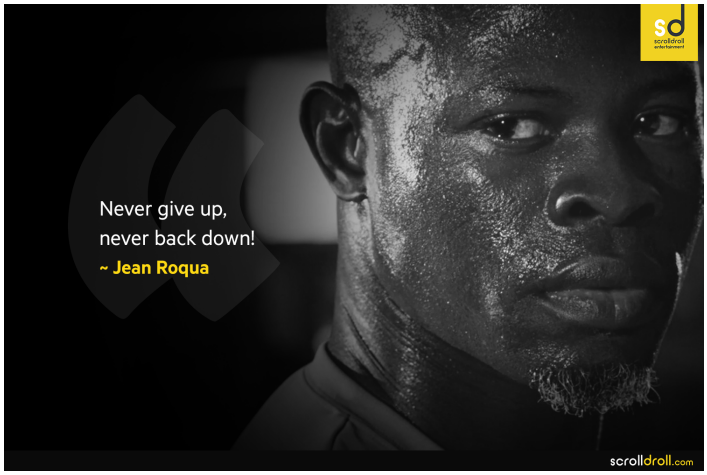
BACKTRACKING 1



Content

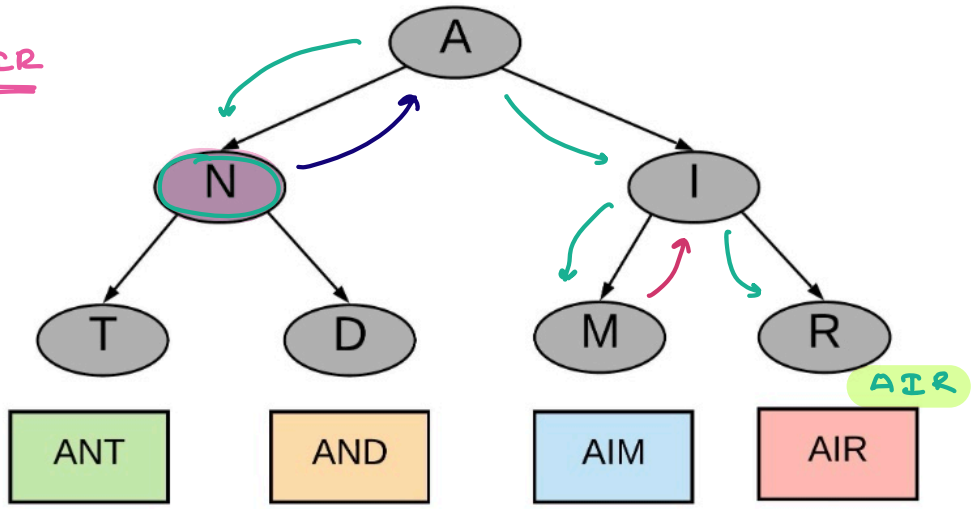
01. What backtracking means?
02. Print Valid paranthesis
03. Subsets
04. Permutation

Backtracking:

Backtracking is a **problem-solving algorithm** that incrementally builds a solution and abandons solutions (backtracks) as soon as it determines that the current path won't lead to a valid solution. It involves recursion, but with an additional step of undoing the last decision (backtracking) when a certain condition is not met.

Backtracking → Brute force Approach → Constraints → very small

Find AIR



Key Differences Between Recursion and Backtracking:

1. **Problem-Solving Approach:**
   - **Recursion:** Solves a problem by breaking it down into smaller subproblems until a base case is reached. It typically explores all possible subproblems without "undoing" any choices.
   - **Backtracking:** Builds the solution step by step and backtracks to try different options when a certain solution path doesn't work.
2. **Exploration of Solutions:**
   - **Recursion:** It goes deep into the recursive calls without evaluating whether a particular solution is valid along the way.
   - **Backtracking:** While it uses recursion, it continuously checks whether the current solution is valid and backtracks if it is not.
3. **Use Case:**
   - **Recursion:** Often used for problems like factorial calculation, Fibonacci sequence, divide-and-conquer algorithms like merge sort, and tree/graph traversals.
   - **Backtracking:** Useful for constraint satisfaction problems where you need to explore many possible configurations (e.g., N-Queens, Sudoku solving, maze solving, and generating permutations).

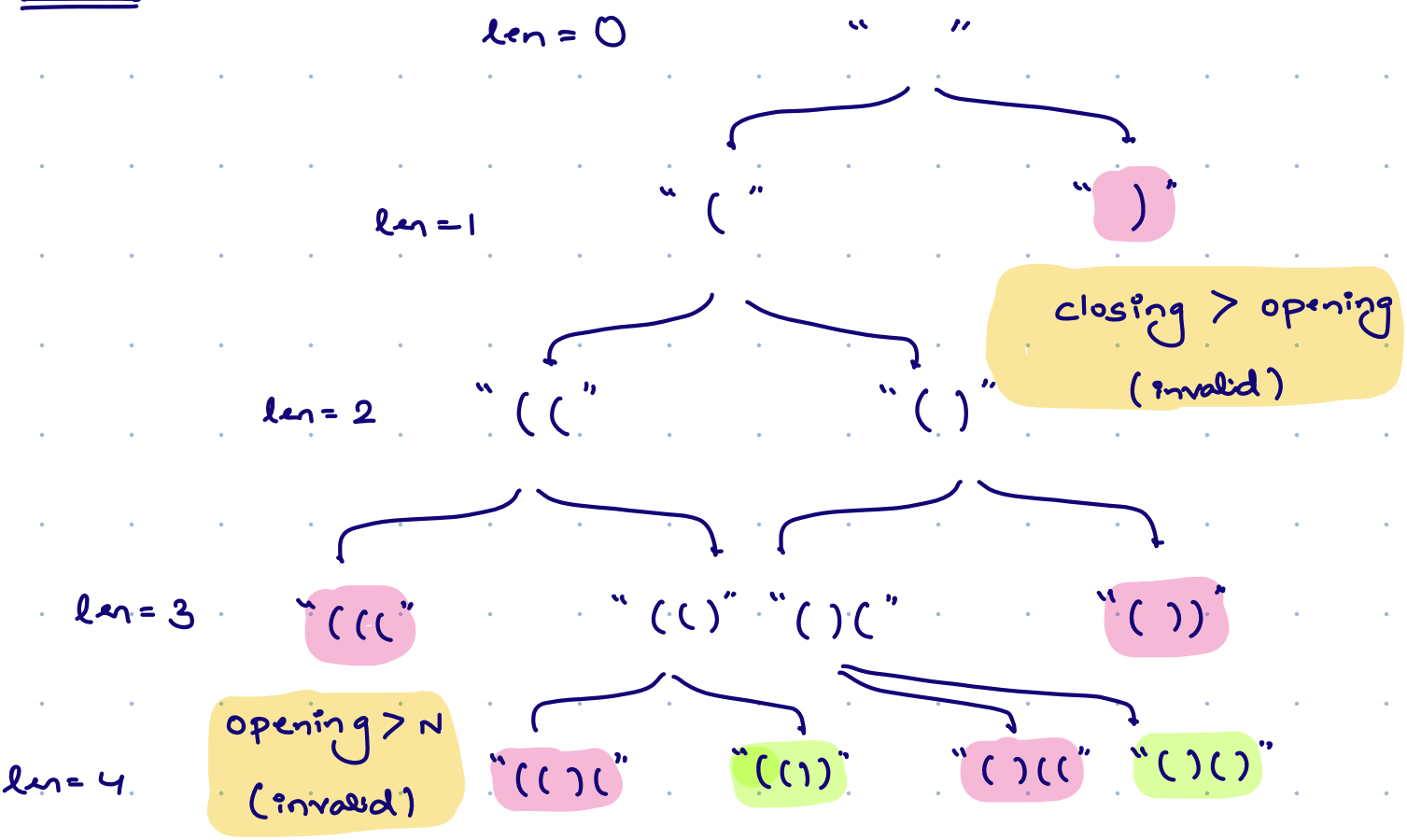
01. Given an integer N, Write a function to generate all valid combination of size 2N.

N = 1 → "( )" ")("

N = 2 → "( )( )" "(( ))" (((( ))( ))((

N = 3 → ( )( )(( )( )) (( )) (( ))( ) ( ))( ) . . .

N = 2



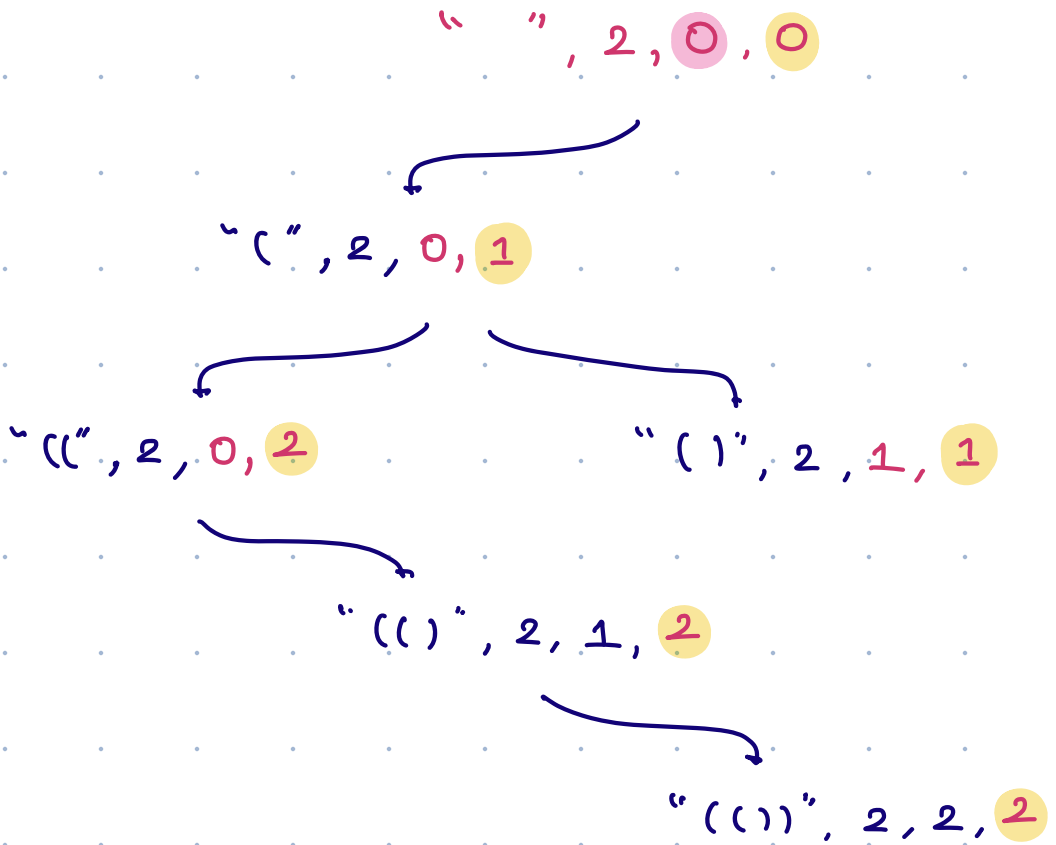
```
void solve ( str , N , closing , opening)
{
    if (str.length == 2*N){
        print (str);
        return;
    }

    if (opening < N)
        solve ( str + "(" , N , closing, opening+1);

    if ( closing < opening){
        solve ( str + ")" , N , closing+1 , opening);
    }
}
```

TC: $O(2^n)$

SC: $O(2*n)$



Subarray, subsequence & subset

A[] = {1, 2, 3, 4, 5}
0 1 2 3 4

Subarray → Continous part of an array.

[2, 3, 4]
[2]
[1 2 3]
[ ] → Not a subarray

Subsequence → Formed by deleting 0 or more elements from the array.

A[] = {1, 2, 3, 4, 5} → {1, 3}
0 1 2 3 4 {1, 5}
{1, 2, 4, 5}
{ }
{1, 2, 3}

Subset → subset is nothing but any possibble combination of any part of given array.

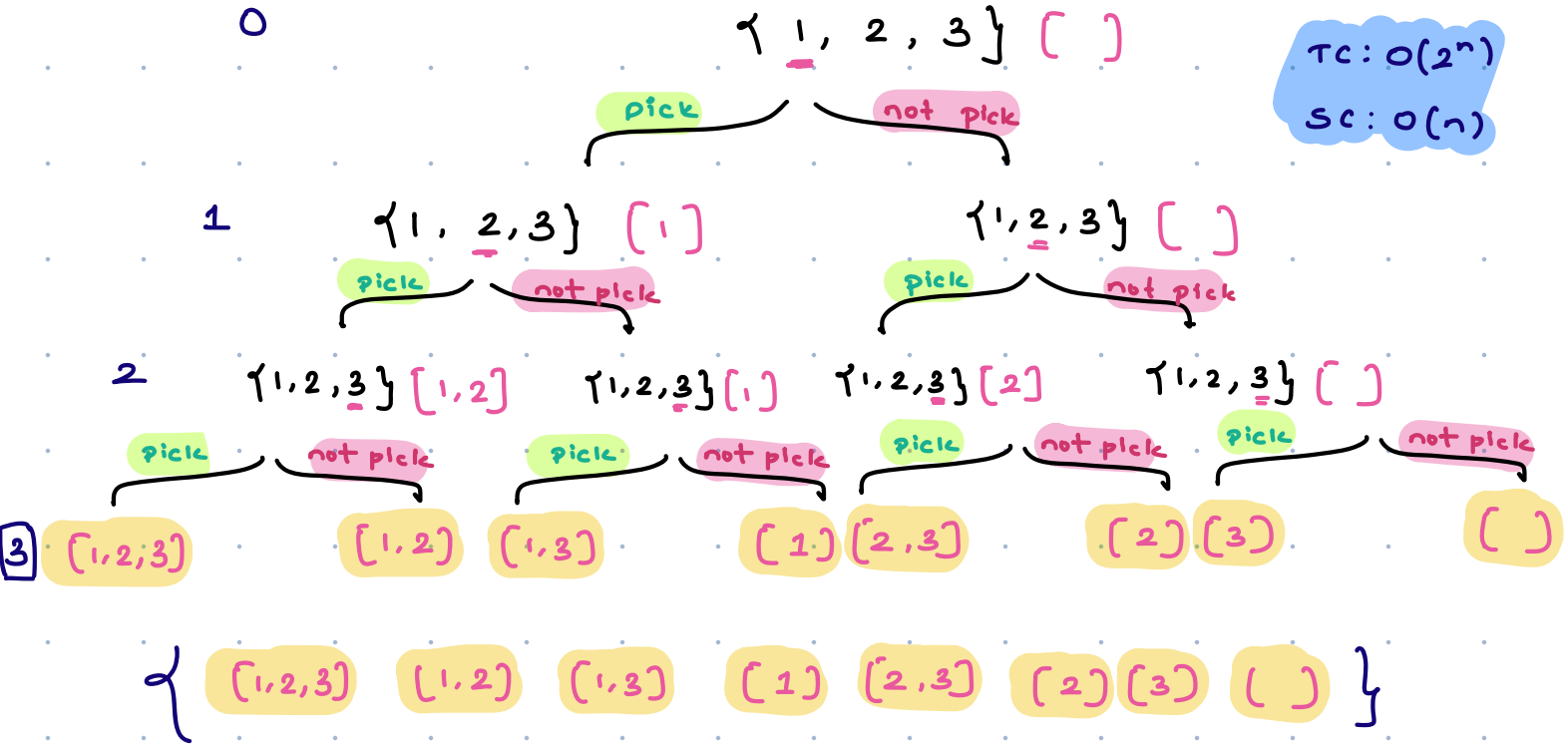
{1, 2, 3}
{1, 3, 2}
{2, 3, 1} } {1, 2, 3}

{2, 1, 3}

{3, 2, 1}

* **Subset**

Given an integer A[] with distinct elements, Generate all subsets using Recursion



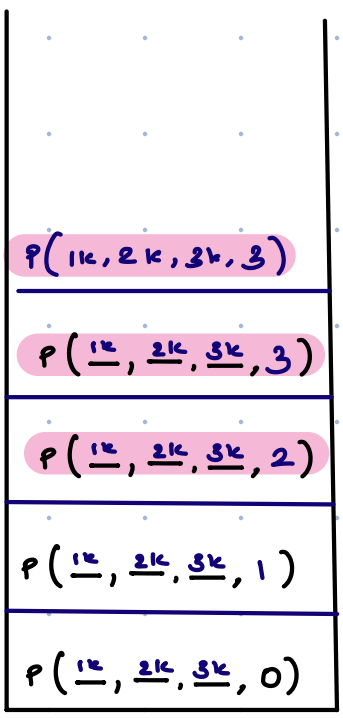
{ [1,2,3] [1,2] [1,3] [1] [2,3] [2] [3] [ ] }

```
solve () {
    AL<I> ans = new AL<>();
    AL<AL<I>> finalans = new AL<>();
    subset ( A, ans, finalans, 0 )
    return finalans
}
```

```
void subset ( int[] A, ans, finalans, idx ){
    if (idx == A.length)
    {
        finalans.add ( new ArrayList<>(ans));
        return;
    }

    ans.add (A[idx])
    subset ( A, ans, finalans, idx +1);      // pick
    ans.remove ( ans.size()-1);

    subset ( A, ans, finalans, idx+1);      //not pick
}
```



10:30 → 10:40 pm

finalans = { {1,2,3} {1,2}, }

ans = {1, 2}

Note → What if we have duplicate elements?

* Total permutations of string with unique characters

⇒ N!

Problem

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.

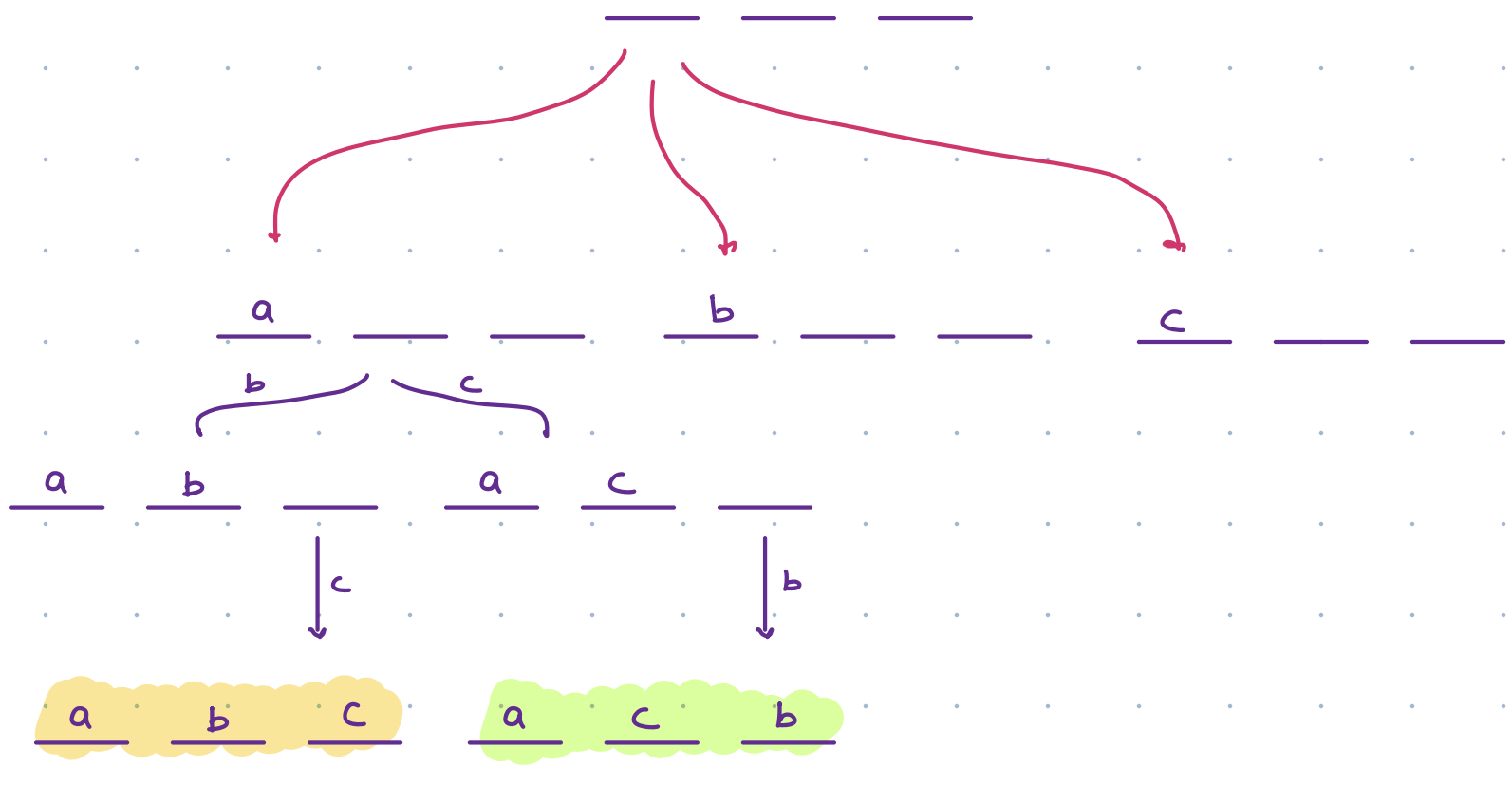
Example:

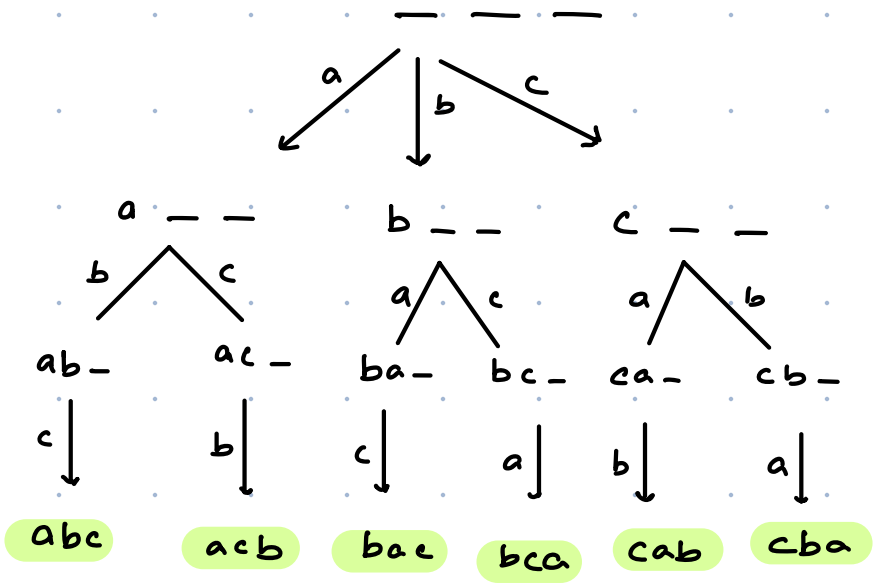
A = Push-ups
B = Squats
C = Burpees
D = Planks

} 4!

str = " a b c " ⟶

a b c
a c b
b a c
b c a
c a b
c b a






```
permute ( A[], idx, ans[], vis[]){
    if (idx == n){
        print (ans[]);
        return ;
    }

    for ( i=0; i<N; i++){
        if (vis[i] == false){
            vis[i] = true ;
            ans[idx] = A[i];
            permute (A, idx+1, ans, vis);
            vis[i] = false
            ans[idx] = 0     // remove whatever you have stored at ans[idx]
        }
    }
}
```

Verify once

TC: $O(N * N!)$

SC: $O(N)$

A[] = { a b c }
       0 1 2

vis[] = { T f T }