Agenda:

1. What is backtracking
2. Intro to Subset
3. Return all subset array in an array.
4. Return all permutation in an array

Quiz 1:

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

SC: O(log N)

N=7, magicfun(3)*10+1 → 111 (11)

N=3, magicfun(1)*10+1 (1)

N=1, magicfun(0)*10+1 (0)

N=0,

Quiz 2: T.C. $N \to N/2 \to N/4 \to N/8 \cdots 0$

$O(\log_2 N)$

[First value printed]

Quiz3:

```
void fun(char s[], int x){
    print(s);
    char temp;
    if(x < s.length/2){
        temp = s[x];
        s[x] = s[s.length - x - 1];
        s[s.length - x - 1] = temp;
        fun(s, x+1);
    }
}
```

SCROLL, 0

Output

SCROLL

LCROLS

LLROCS

LLORCS

0 1 2 3 4 5
SCROLL, 0.

0 < 6/2 → True

swap(0, 5)

↓

0 1 2 3 4 5
(LCROLS, 1)

1 < 6/2 → True

swap(1, 4)

↓

0 1 2 3 4 5
(LLROCS, 2)

2 < 6/2

swap(2, 3)

↓

(LLORCS, 3)

3 < 6/2 → False

Quiz 4: TC. $(N/2) \longrightarrow O(N)$

( )

Ques. Print all valid parenthesis of length 2N for a given value of N.

↳ Equal no. of opening & closing brackets

N = 1 ( ) ,) (

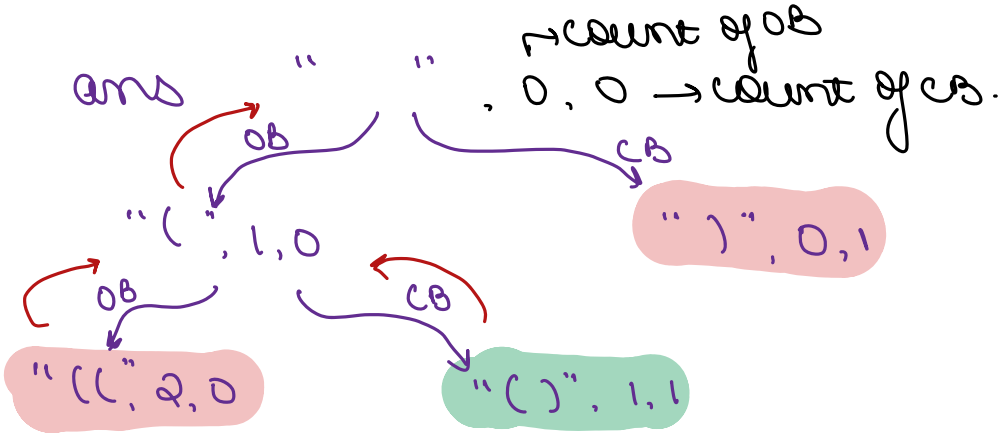
N = 2 ( )( ) , (( )) ,)) ((,)) ( ) ,))))

N = 3 ( )( )( ) , (( )) , ( )( ) , ( ) (( ) , (( ))( )

Idea: Start generating string starting from 0 till len = 2N

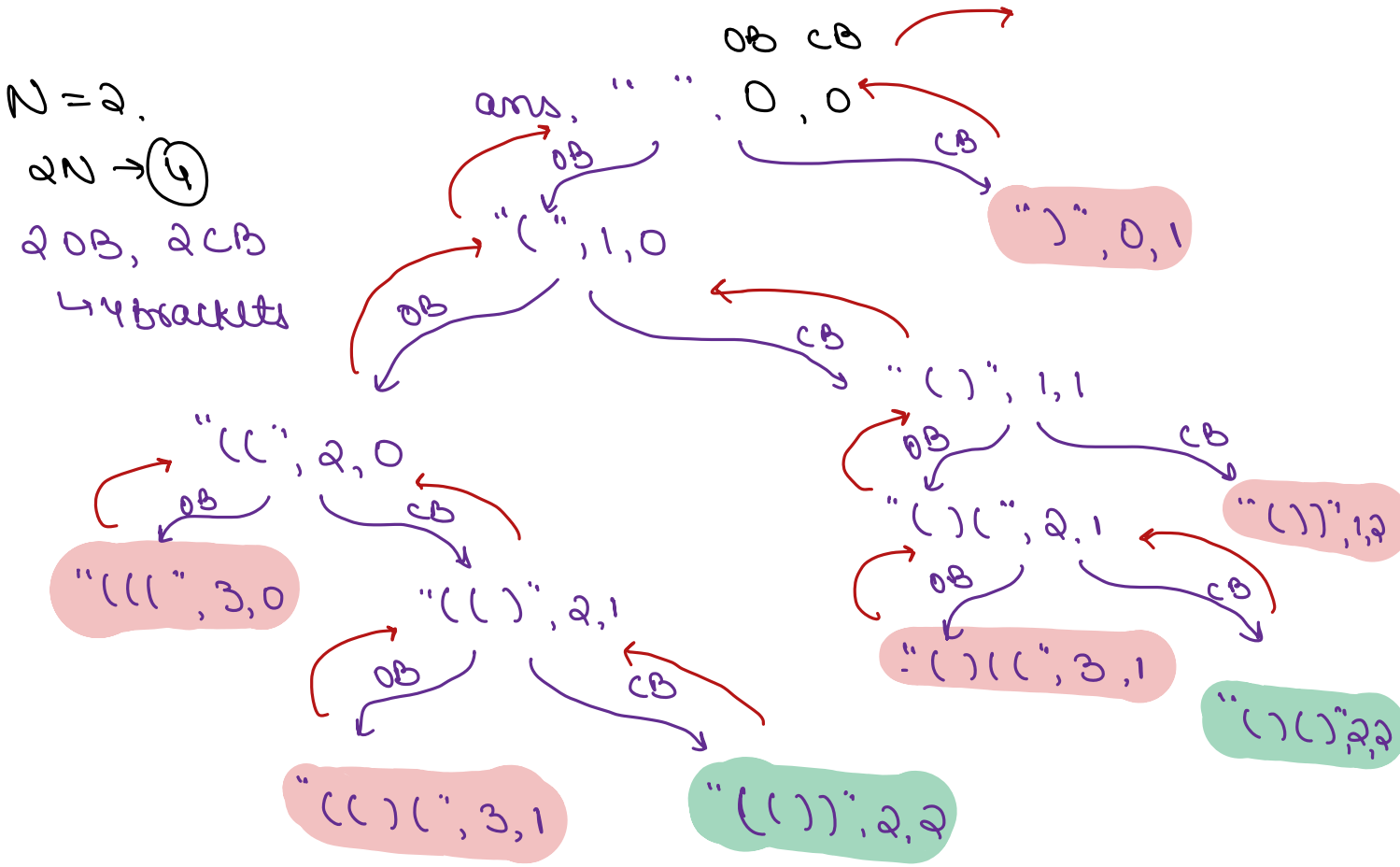
N = 1,

2N



Invalid

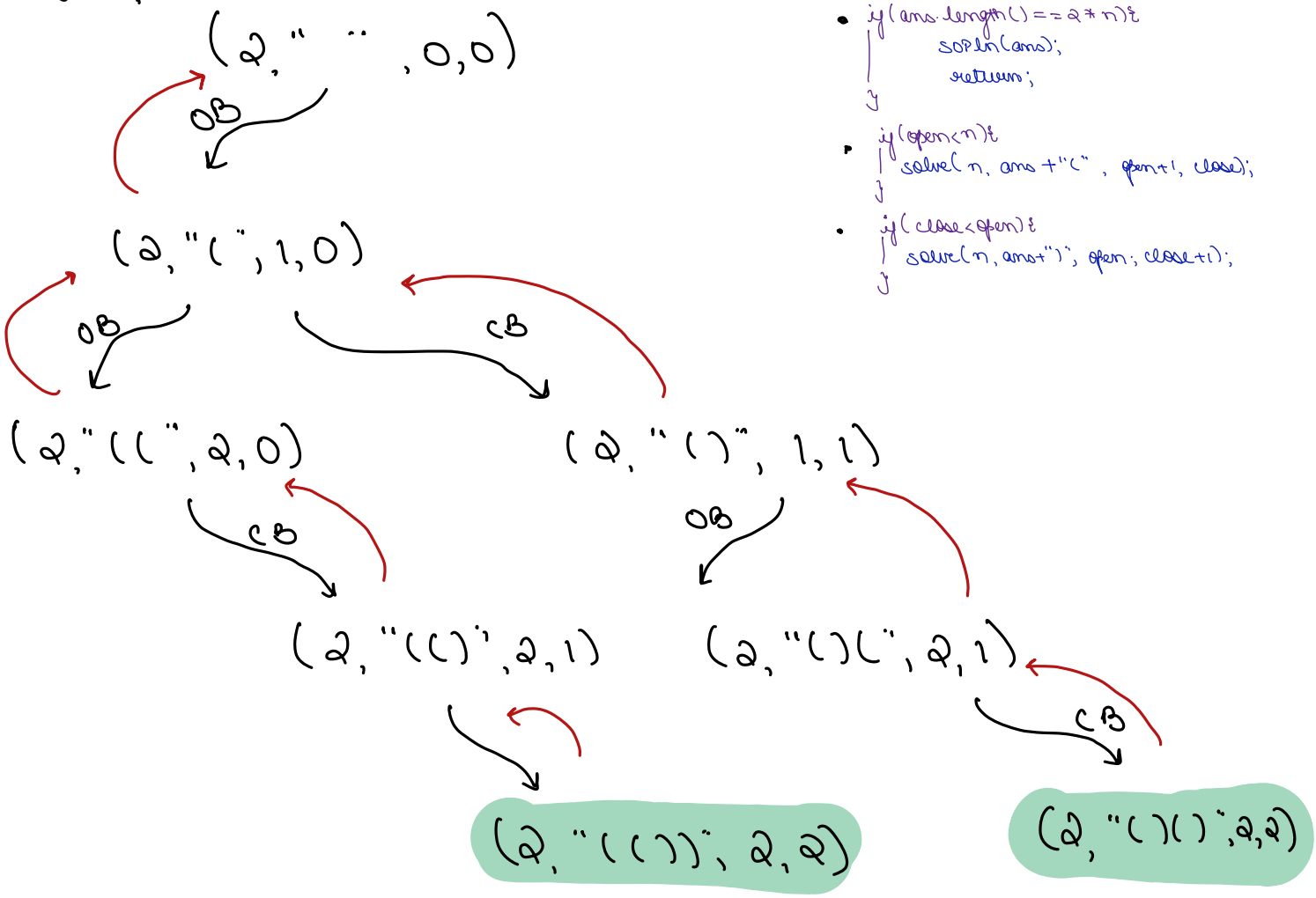
1. opening > N
2. closing > opening



Code:

```
void solve(int n, string ans, int open, int close){
    if(ans.length() == 2*n){
        sopln(ans);
        return;
    }
    if(open<n){
        solve(n, ans+"(", open+1, close);
    }
    if(close<open){
        solve(n, ans+")", open, close+1);
    }
}
```

N = 2.

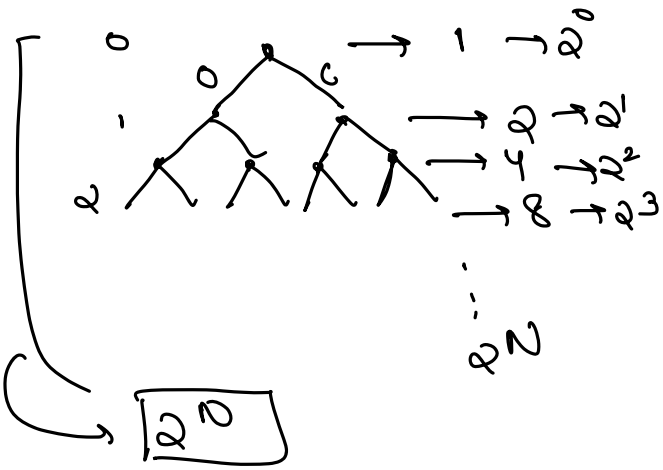



```
if(ans.length() == 2*n){
    sop(ans);
    return;
}
if(open<n){
    solve(n, ans+"(", open+1, close);
}
if(close<open){
    solve(n, ans+")", open, close+1);
}
```

TODO : N=3

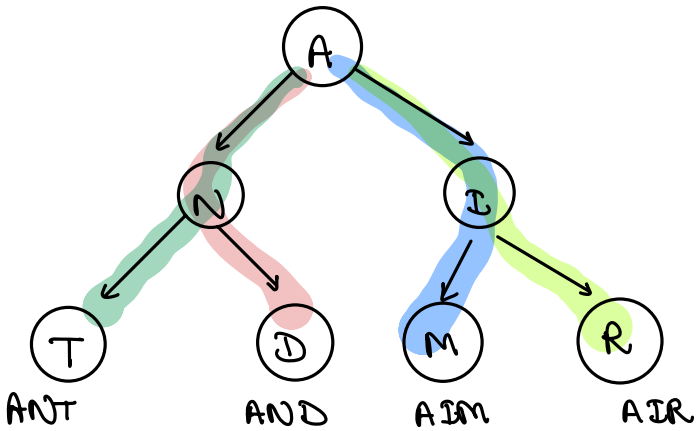
TC: $O(2^N)$

SC: $O(N)$



Backtracking :

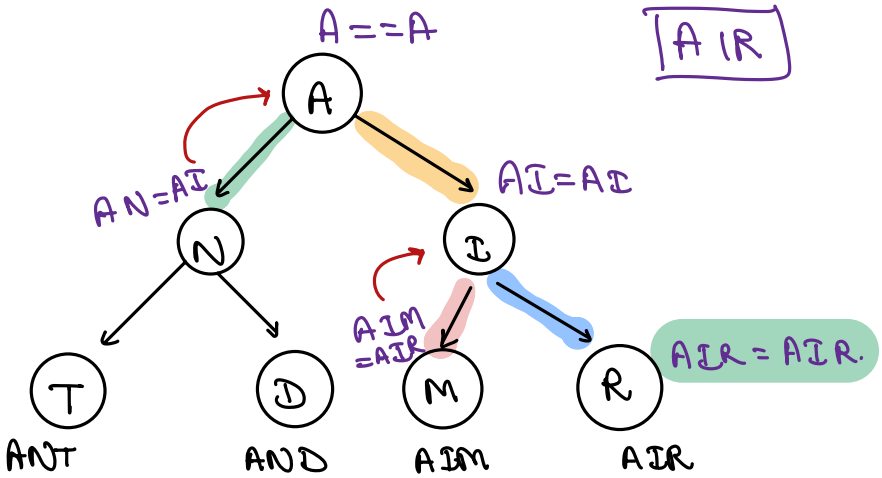
Backtracking is like trying different paths. When you hit a dead end, you backtrack to the last choice and try a different route.



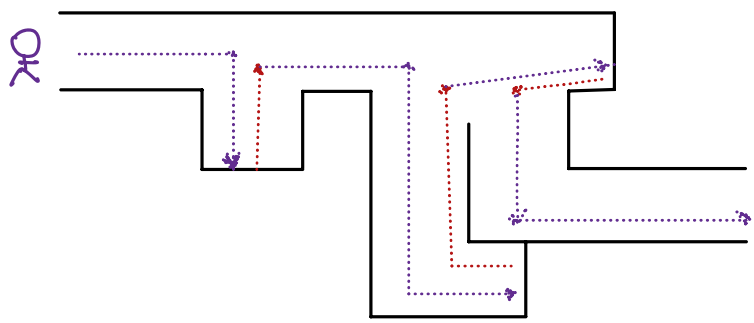
find word "AIR"

Efficient way. : Backtracking

↳ Backtracking optimises the search, avoiding unnecessary exploration



Maze:



Idea: While generating all solutions, choose path but if we cannot go any further, backtrack and take another path.

Subsets

Subarray: Continuous part of array.

Subset: Any possible combination of the original array is a subset.

- ↳ Set can be obtained by deleting 0 or more elements from the array.
- ↳ Empty set is a valid set
- ↳ Order does not matter here.

arr: {3, 2, 1, 9, 6, 8}

Subsets:

{1, 6, 8} —Subset→ {*, *, 1, *, 6, 8} ✓

{9, 6, 3} —Subset→ {3, *, *, 9, 6, *} ✓

{2, 9, 8} —Subset→ {*, 2, *, 9, *, 8} ✓

Every subarray ⟶ subset

vice versa not valid

Subsequence

Obtained by deleting elements from an array without changing order.

eg: arr: [1, 2, 3, 4, 5]

	Subarray	subset	subsequence
{1, 2, 3, 4}	✓	✓	✓
{2, 3, 5}	✗	✓	✓
{4, 1, 2}	✗	✓	✗
{2, 3}	✓	✓	✓
{1, 3, 5}	✗	✓	✓

Note:
1. A subset may or may not be a subsequence
2. A subsequence is always a subset
3. A subarray is also a subsequence

Q → Given an array with distinct integers.
Print all subsets using recursion.
↳ order does not matter (all possible combinations)

eg. arr = {1, 2, 3}

Subsets: [1], [1,2], [1,2,3],
[2], [2,3], [3], [1,3], [ ]

arr = {6, 10}

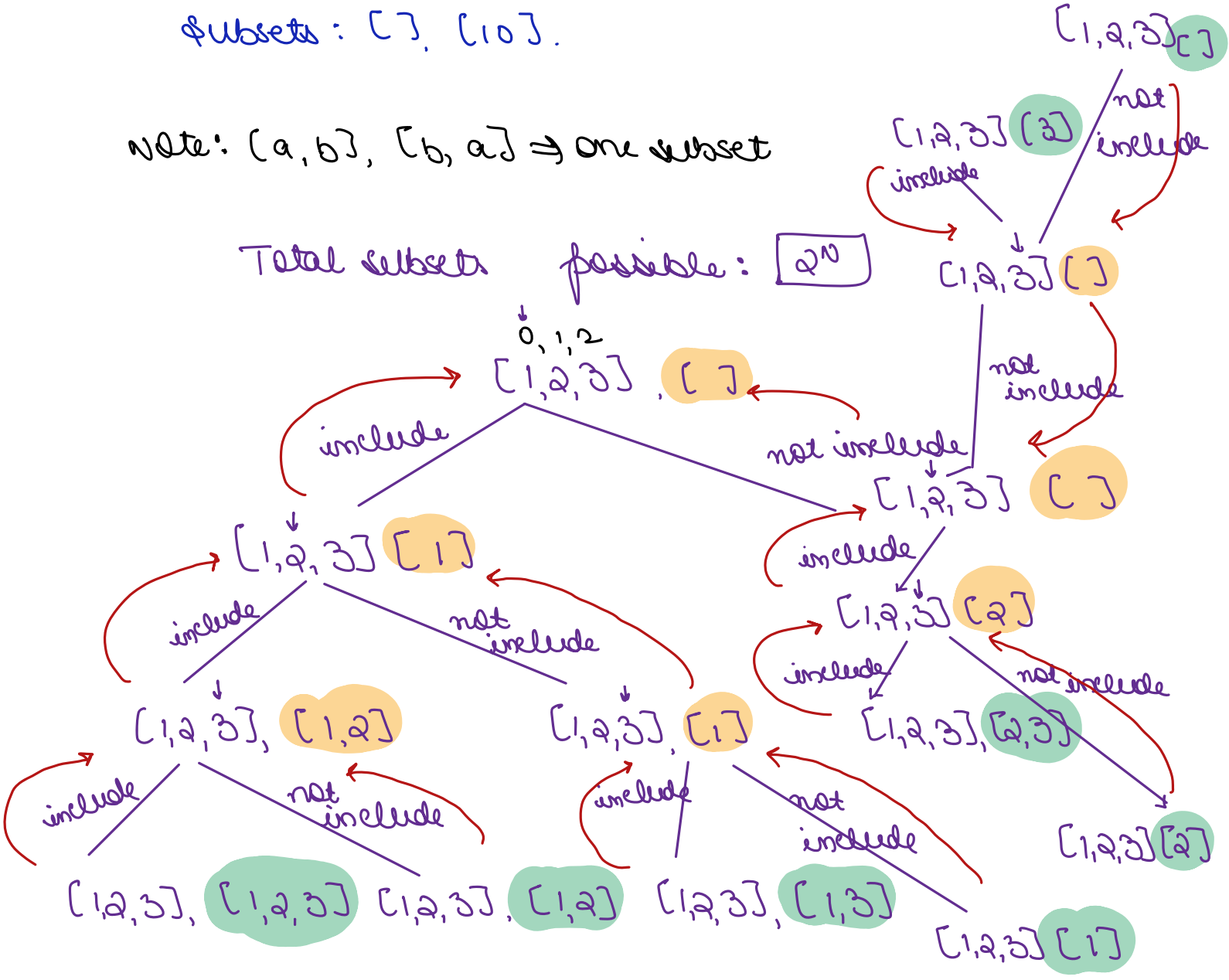
subsets: [6], [6,10], [10], [ ]

arr = {10}

subsets: [ ], [10].

note: [a, b], [b, a] ⇒ one subset

Total subsets possible: 2^N



Idea:

↳ 2D list

list<ans>

TC: O(2^N)

SC: O(N)

```
void subsets(list A, list<> curset, int idx){

    if(idx == A.length){
        ans.add(curset);
        return;
    }

    // for every ele, we have 2 choices
    // choice 1 → keep it in curset

    curset.add(A[idx]);
    subsets(A, curset, idx+1);

    // choice 2 → don't keep in curset
    curset.remove(A[idx]);
    subsets(A, curset, idx+1);
}
```

[[]]

[2,6,9], [], 0

Dry Run.

```
if(idx==A.length){
    ans.add(curr);
    return;
}
```

idx=0 curr[2]

```
curr.add(A[idx]);
subset(A, curr, idx+1);
curr.remove(A[idx]);
subset(A, curr, idx+1);
```

idx=1

```
if(idx==A.length){
    ans.add(curr);
    return;
}
```

curr[2,]

```
curr.add(A[idx]);
subset(A, curr, idx+1);
curr.remove(A[idx]);
subset(A, curr, idx+1);
```

idx=2

```
if(idx==A.length){
    ans.add(curr);
    return;
}
```

curr[2,6,]

```
curr.add(A[idx]);
subset(A, curr, idx+1);
curr.remove(A[idx]);
subset(A, curr, idx+1);
```

idx=2

```
if(idx==A.length){
    ans.add(curr);
    return;
}
```

curr[2,9]

```
curr.add(A[idx]);
subset(A, curr, idx+1)
curr.remove(A[idx]
subset(A, curr, idx+
```

idx = 3

```
if(idx == A.length){
    ans.add(curset);
    return;
}

curset.add(A[idx]);
subset(A, curset, idx+1);
curset.remove(A[idx]);
subset(A, curset, idx+1);
```

idx = 3

```
if(idx == A.length){
    ans.add(curset);
    return;
}

curset.add(A[idx]);
subset(A, curset, idx+1);
curset.remove(A[idx]);
subset(A, curset, idx+1);
```

ans: [[2,6,9], [2,6]]

TODO complete day Rem

Backtracking → changes you have done while going down in recursion, undo that changes

a, b, c → length 3

abc
acb
bac
bca
cab
cba

→ 6

3! ⇒ 6

N!

ab
ab
ba

2! ⇒ 2

Permutations:

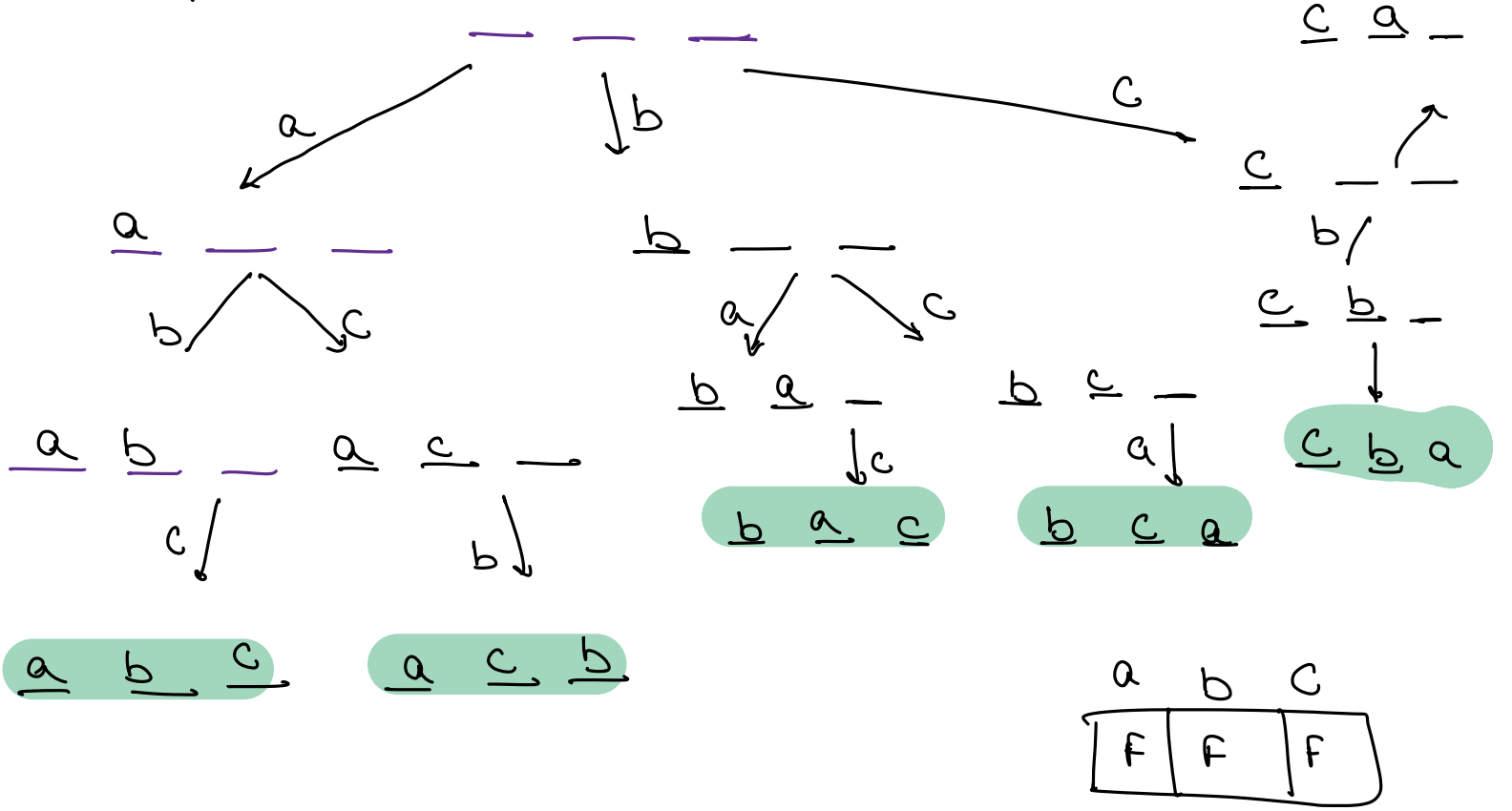
Given a character array with distinct elements.
Print all permutations of it without modifying it.

i/p: [a, b, c]

Permutations, abc
acb
bac
bca
cab
cba

N=3 → 6 permutations
3!

Approach:



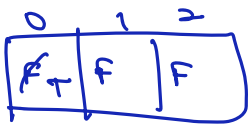
a	b	c
F	F	F

Code

```
void permutations(char[] ar, ans[N], idx, boolean visited[]){
    if(idx == ar.length){
        print(ans);
        return;
    }

    for(int i=0; i<N; i++){ //all possibilities
        if(visited[i] == false){
            visited[i] = true;
            ans[idx] = ar[i];
            permutations(ar, ans, idx+1, visited[]);
            visited[i] = false; //undo changes.
        }
    }
```

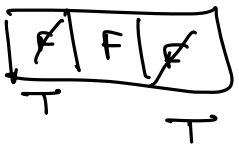
↳ backtracking



TC: $O(2^N)$

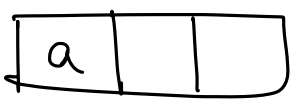
SC: $O(N)$

idx = 0,




```
if(idx == ar.length){
    print(ans);
    return;
}
for(int i=0; i<n; i++){ //all possibilities   i = 0
    if(visited[i] == false){
        visited[i] = true;
        ans[i] = ar[i];
        permutations(ar, ans, idx+1, visited[]);
        visited[i] = false; //undo changes.
    }
}
```

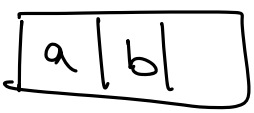

```
if(idx == ar.length){          i = 1
    print(ans);
    return;
}
for(int i=0; i<n; i++){ //all possibilities
    if(visited[i] == false){
        visited[i] = true;
        ans[i] = ar[i];
        permutations(ar, ans, idx+1, visited[]);
        visited[i] = false; //undo changes.
```



i = 2


```
if(idx == ar.length){
    print(ans);
    return;
}
for(int i=0; i<n; i++){ //all possibilities   i = 2.
    if(visited[i] == false){
        visited[i] = true;
        ans[i] = ar[i];
        permutations(ar, ans, idx+1, visited[]);
        visited[i] = false; //undo changes.
```

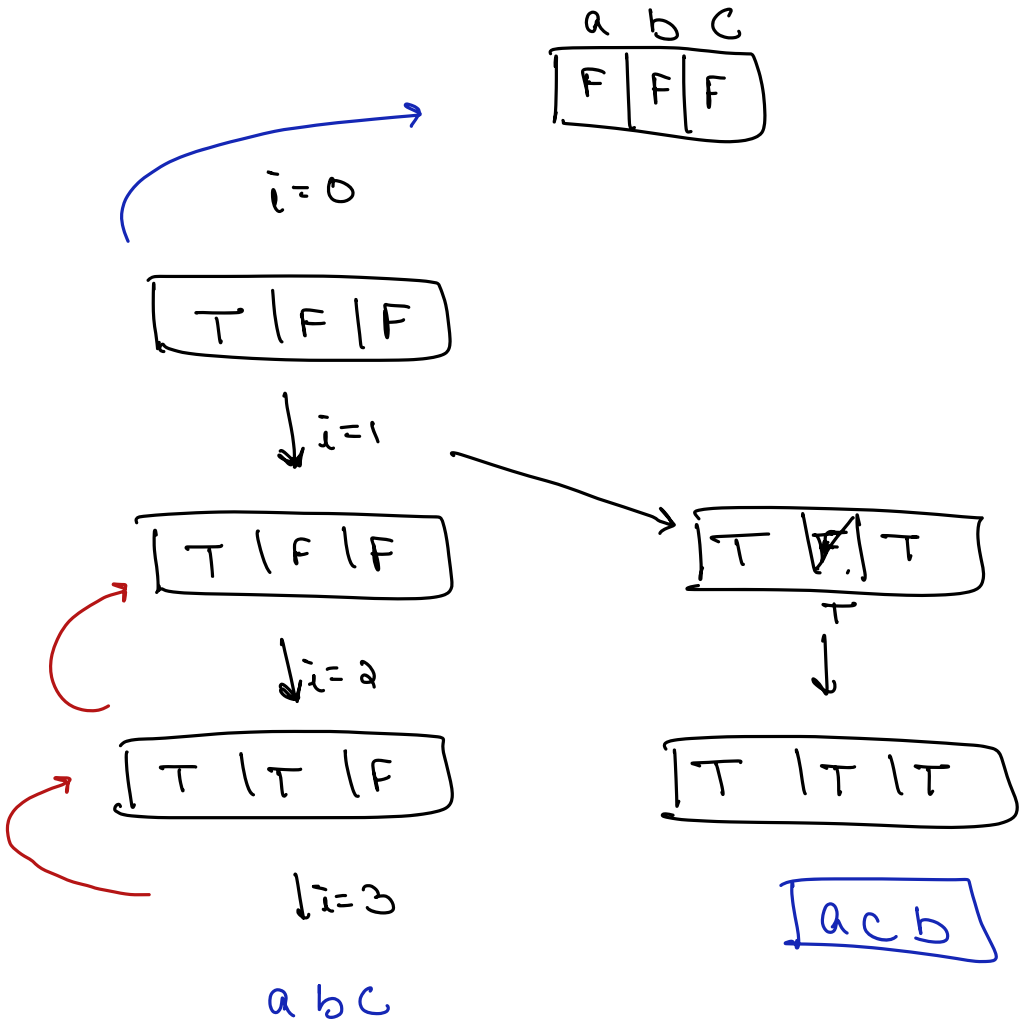
i = 3



i = 3

```
if(idx == ar.length){
    print(ans);
    return;
}
for(int i=0; i<n; i++){ //all possibilities
    if(visited[i]==false){
        visited[i]= true;
        ans[i] = ar[i];
        permutations(ar, ans, idx+1, visited[]);
        visited[i] = false; //undo changes.
```

a, b, c



a b,