Lecture 31

Recursion

→ function that calls itself.

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$\Rightarrow n! = n \times (n-1)!$

$\Rightarrow F(n) = n \times F(n-1)$

→ recursive relation

→ factorial_recursion.cpp

BASE CASE

→ Stopping condition for the function

Power of 2

Base Case $= 2^1 = 2$

$\Rightarrow 2^5 = 2 \times 2^4$

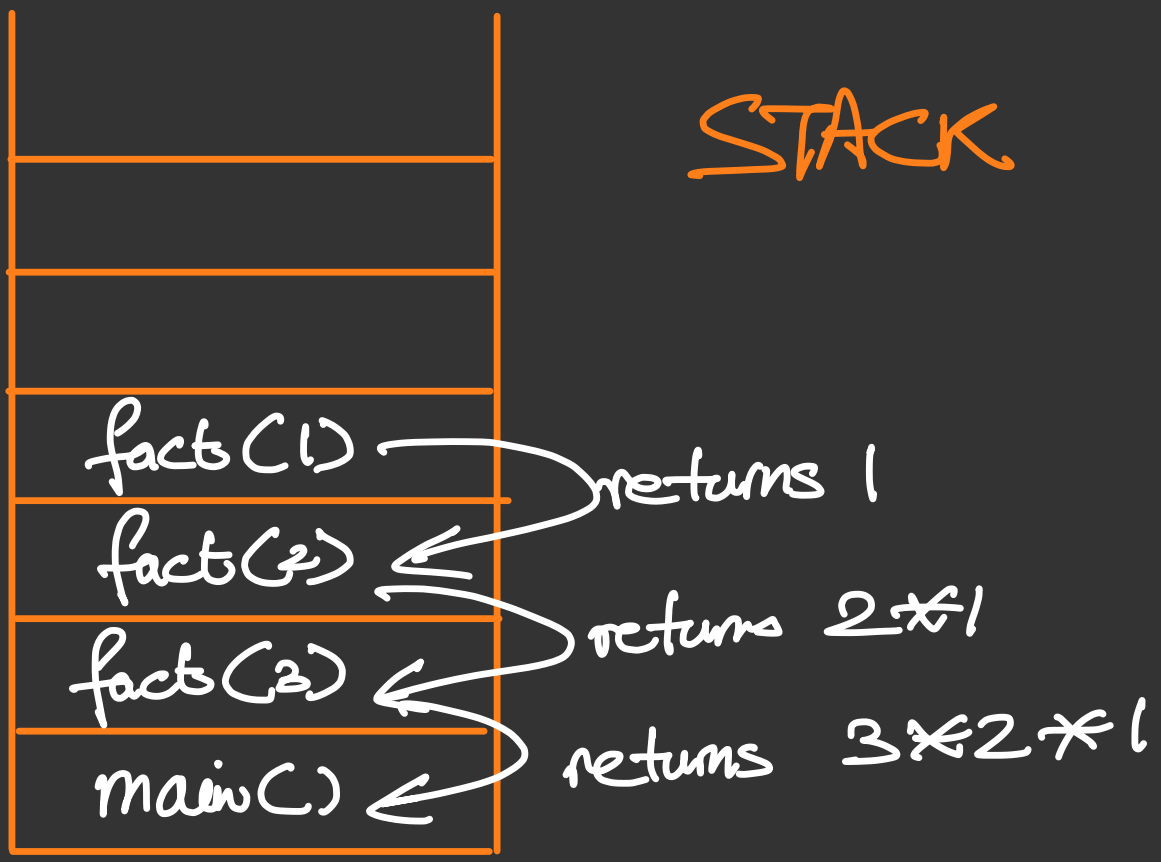
$\Rightarrow f(n) = 2 * f(n-1)$

$\longrightarrow$ powerOf2_recursion.cpp

Thus, for Recursion we need

1. Base Case (stopping condition)
2. Recursive Relation

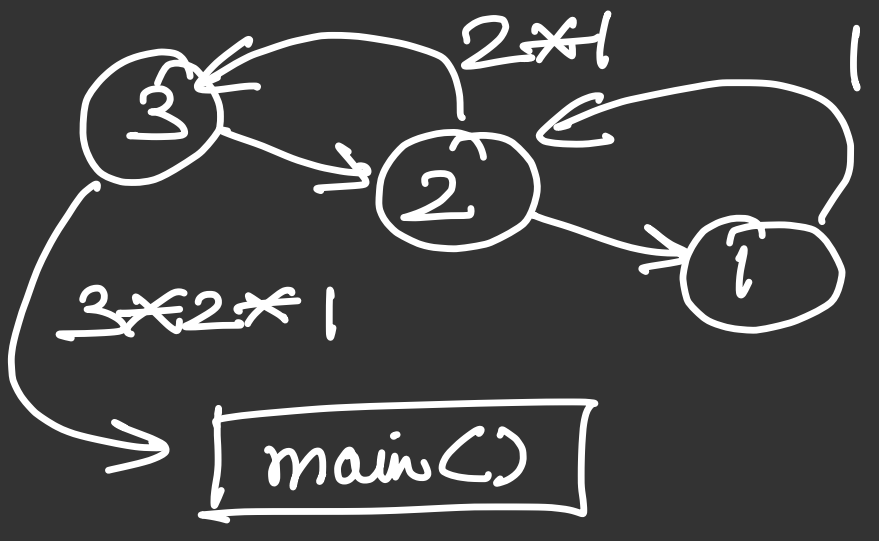
☆ If no base case $\Rightarrow$ segmentation faults.



☆ If no base case → segmentation faults.

As stack memory gets filled.

Recursion Tree



Types ⟶

1. Tail Recursion

```
func()
{
    → Base Case
    → Processing
    → Recursive Relation
}
```

2. Head Recursion

```
func()
{
    → Base Case
    → Recursive Relation
    → Processing
}
```

Print Counting

i/p → 5

o/p → 5 4 3 2 1

→ printCounting — recursion.cpp.

⚠ If we puts

printCount(n-1);
cout << n << " "; } in this order

then

o/p → 1 2 3 4 5

After recursion part done processing of statements below happen, and first printCount(1) gets free, then others.

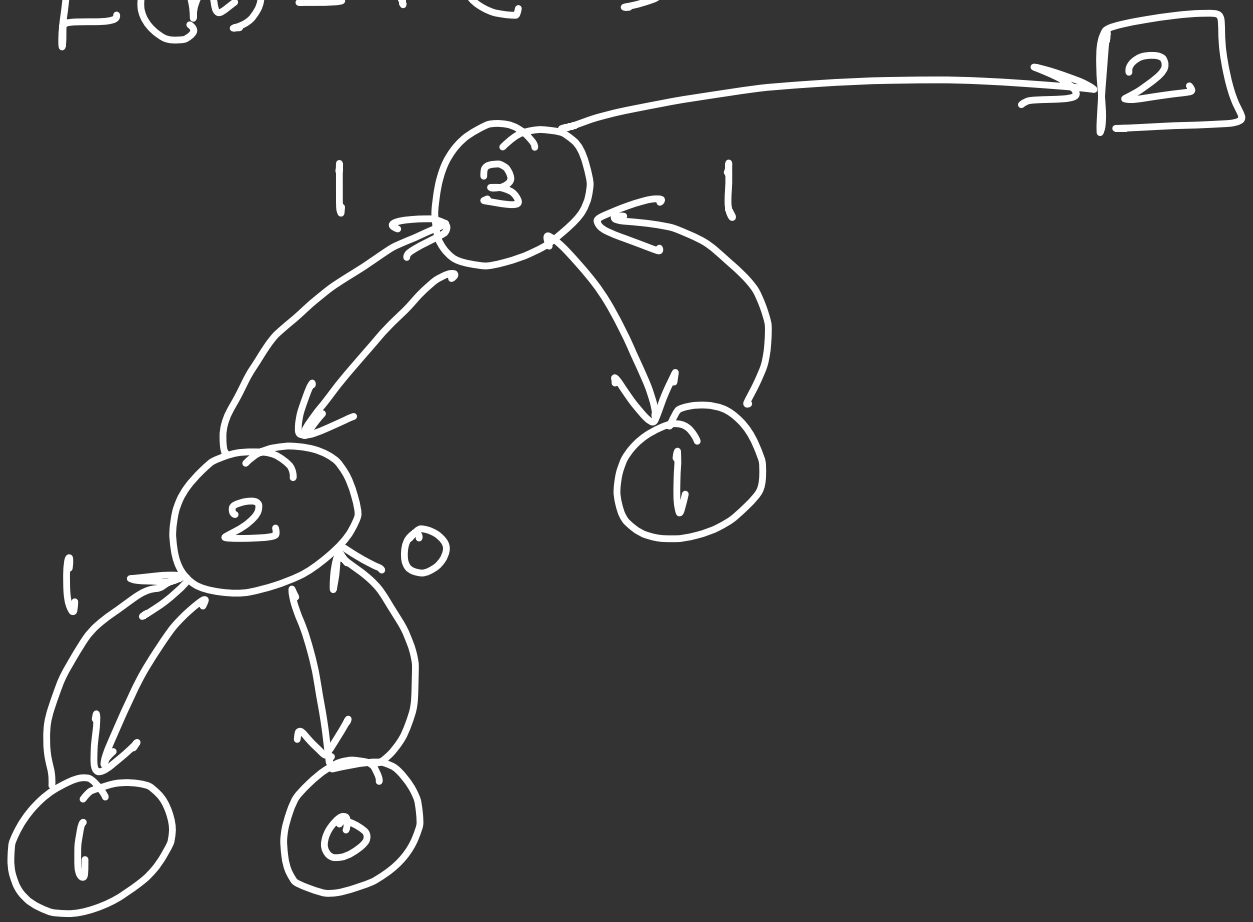
Lecture 32

☆ If the function is void, just `return;` also works!

Fibonacci Series

0, 1, 1, 2, 3, 5, 8, 13

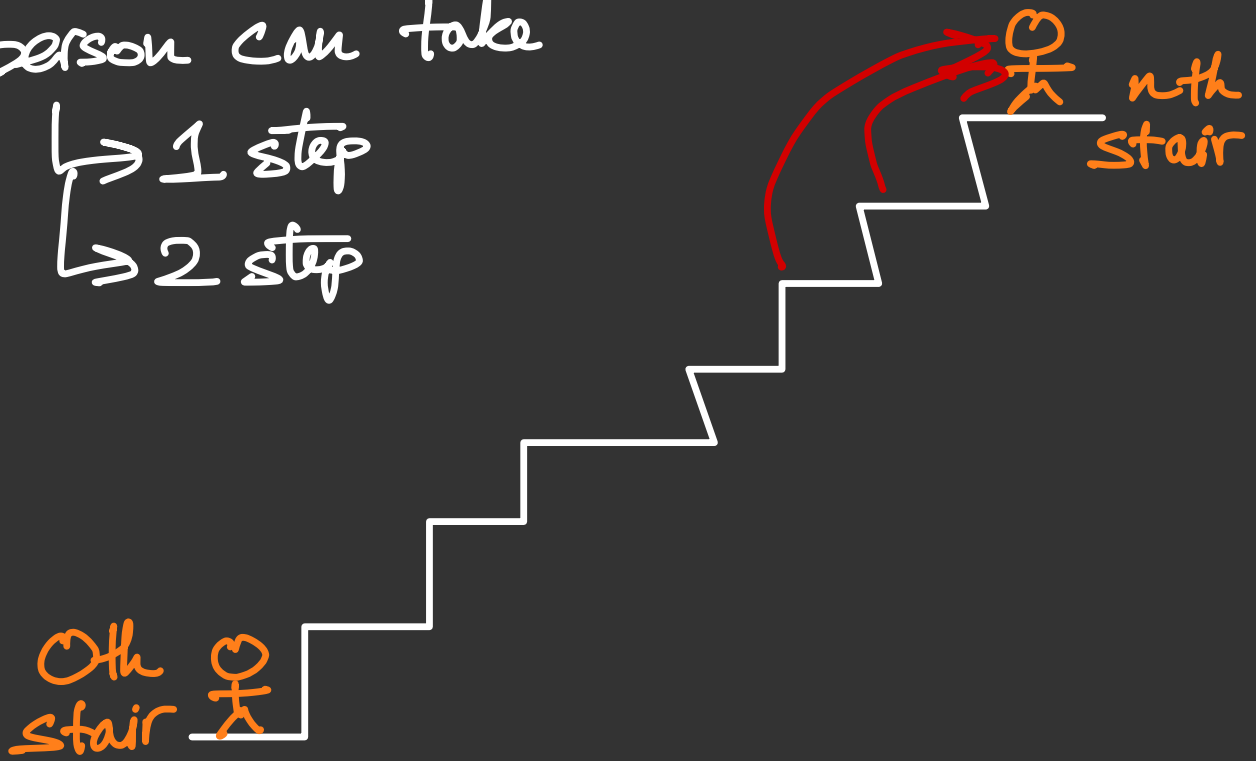
$$F(n) = F(n-1) + F(n-2)$$



→ fibonnaci_recursion.cpp

Count ways to reach the Nth Stairs

* person can take
  - 1 step
  - 2 step



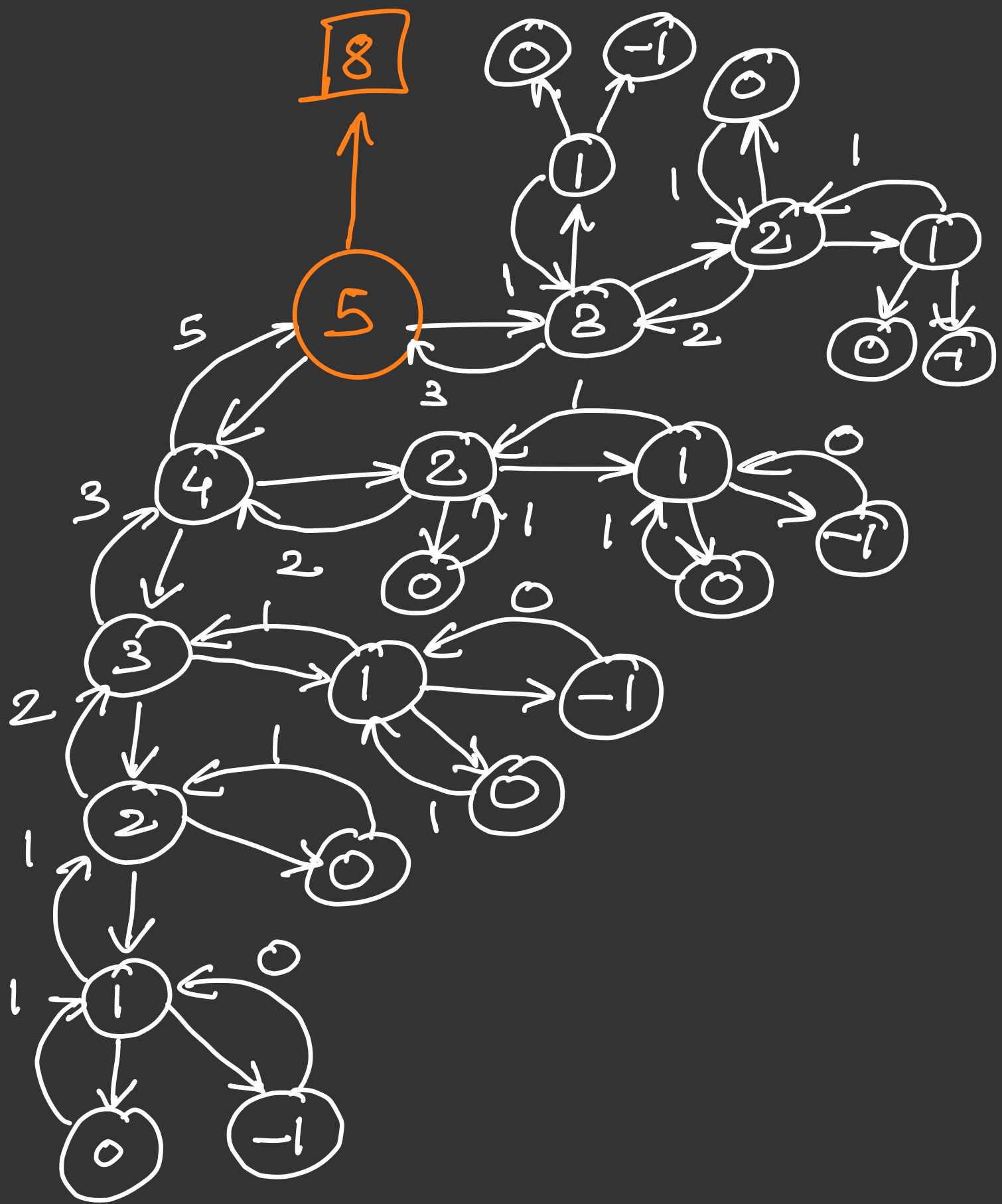
$$F(n) = F(n-1) + F(n-2)$$

So, to reach nth stair he will either come from $(n-1)^{th}$ stair or $(n-2)^{th}$ stair.

In short fibonacci series.

⟶ climb_stairs.cpp

Recursion Tree



Say Digits

i/p ⟶ 412

o/p ⟶ FOUR ONE TWO

⊠ Printing happens after recursion and after base case has been reached because digit extraction happens right to left and we want to print digits left to right.

⟶ sayDigits_recursion.cpp

Lecture 33

Array + Recursion

Is the Array Sorted or not?

My Approach

↳ I went from right to left in the array.

But we can go left to right also by passing (arr+1) address and size-1.

⟶ isArraySorted_recursio.cpp

Sum of the Array Elements

⟶ arraySum_recursion.cpp

Linear Search

* Used the other approach

arr[0] == key

↳ *(arr+0) == key

Passing (arr+1)

↳ *(arr+1+0) == key

⟶ linearSearch_recursion.cpp

Binary Search

```
s = 0
e = L-1
mid = s + (e-s)/2;
if (arr[mid] == key)
    return true;
else if (arr[mid] > key)
    e = mid - 1;
else
    s = mid + 1;
```

binarySearch(int arr[], int size, key)

If not found and Base condition also not true

↳ return (binarySearch(arr+s, e+1, key));

Did a binarySearch2 to return the index of the element

→ binarySearch_recursion.cpp

Lecture 34

Reverse a string

→ swap(str[i], str[j]);

→ i, j

→ Also, we need to pass string through reference variable

Initially Pass (str, 0, str.length() - 1);

→ reverseString_recursion.cpp

Check Palindrome with Recursion

☆ Did it using just one pointer parameter

→ checkPalindrome_recursion.cpp

Calculating Power using Recursion

b is even $= (a^{b/2})(a^{b/2})$

b is odd $= a(a^{b/2})(a^{b/2})$

Example:

$2^9 = 2 \times (2^4)(2^4)$

$\rightarrow 2^4 = 2^2 \times 2^2$

$\rightarrow 2^2 = 2 \times 2$

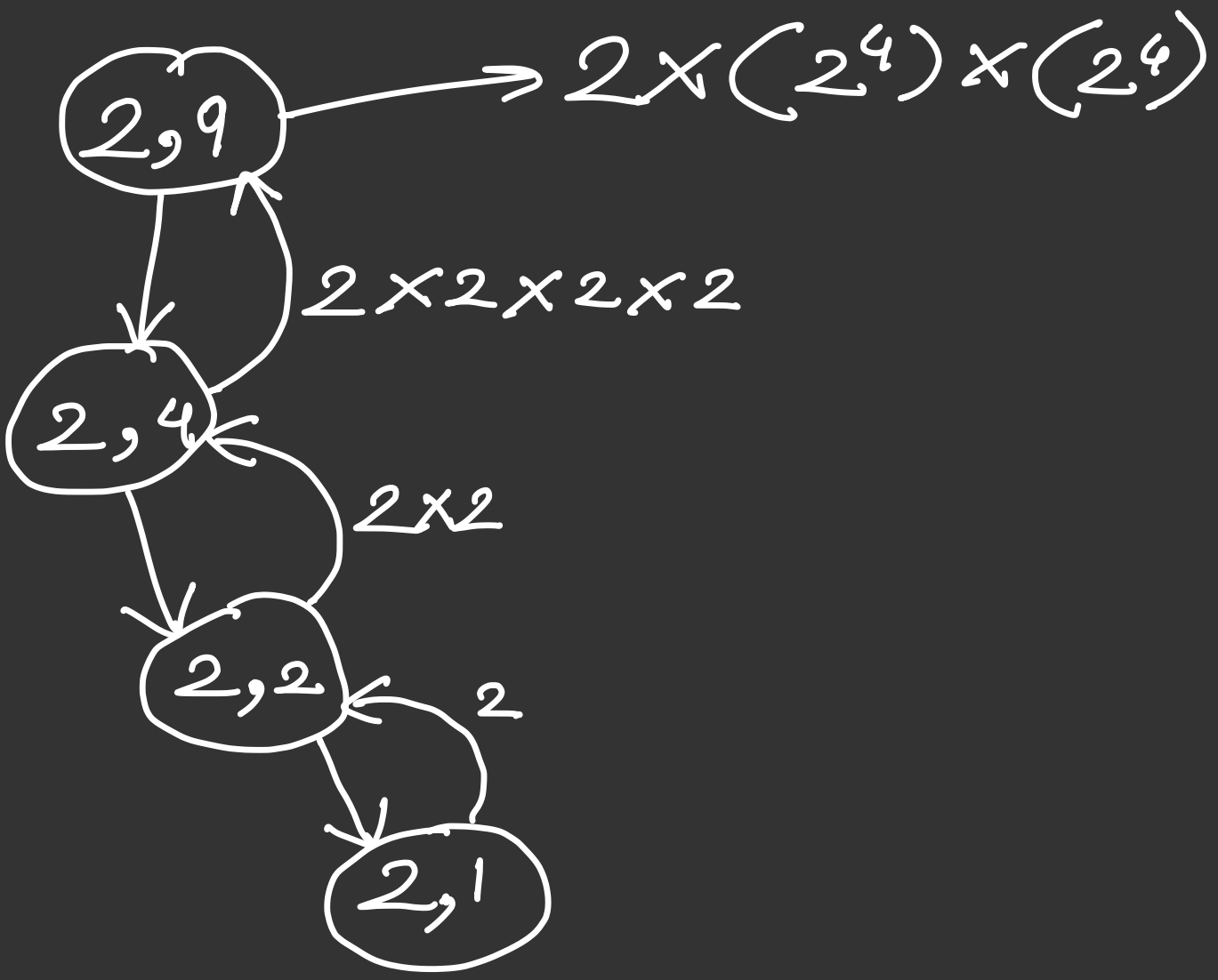
$\rightarrow 2^1 = 2 \times 2^0 \times 2^0$

$\rightarrow 1$

⚠ we have to go till $b == 1$ || $b == 0$ and then return a value, AFTER getting the value we multiply and then RETURN finally.

Example : 2^9

$a=2, b=9$



f(a,b) → b is odd
→ a * f(a,b/2) * f(a,b/2)

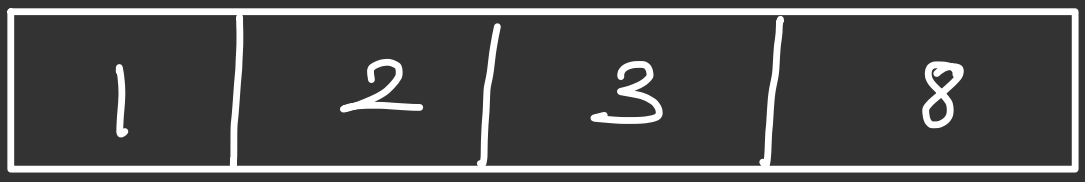
→ b is even
→ f(a,b/2) * f(a,b/2)

→ power_with_recursion.cpp

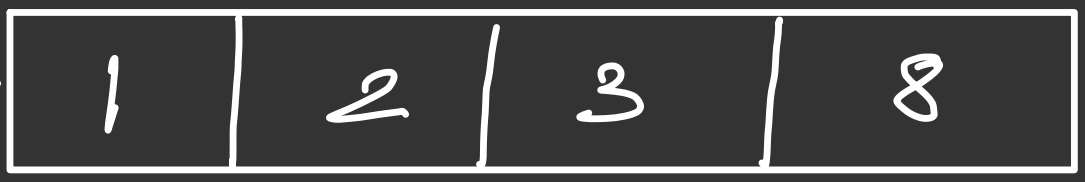
Bubble Sort

1-

2-



3-



Round i $\longrightarrow$ i^{th} largest

$\longrightarrow$ $n-i^{th}$ index

$\longrightarrow$ bubbleSort_recursion.cpp

Selection Sort

→ selectionSort_recursion.cpp

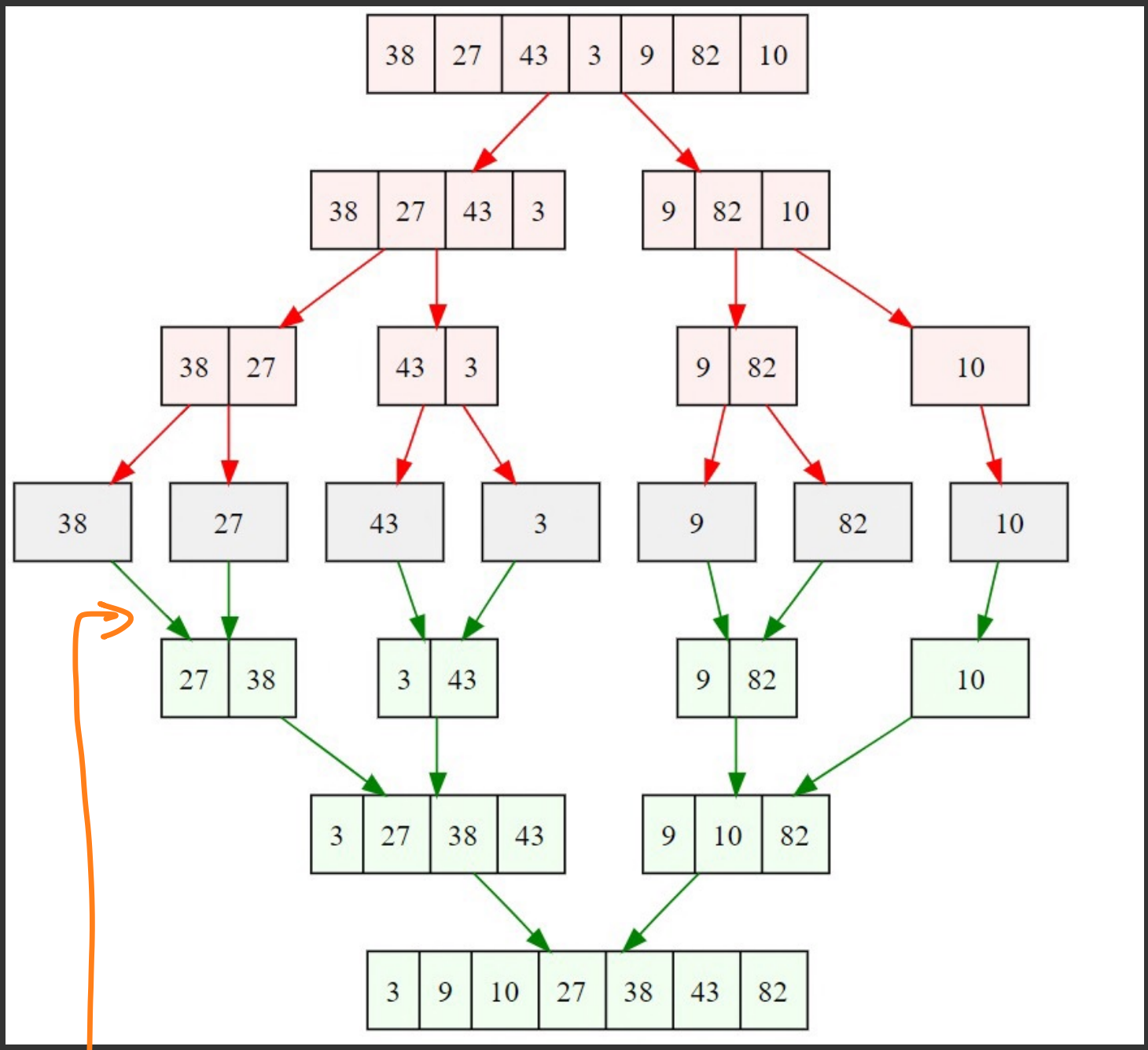
Insertion Sort

→ insertionSort_recursion.cpp

* Left out the recursion part for now.

Lecture 35
Merge Sort

i/p array →



Now sorting happens

Steps to remember ⟶

1. Check for (s >= e)
2. Calculate Mid
   - ↳ Send left in recursion
   - ↳ Send right in recursion
3. Merge function
   - ↳ Calculate Mid
   - ↳ Divide into two dynamic arrays, left and right Part
   - ⟶ Merge the two sorted Arrays, ie two subparts are already Sorted.
   - ⟶ Delete dynamic Arrays.

⚠ Fastest among all sorting algorithms

⚠ As soon as `s>=e`, the previous segment of array goes into the merge(arr, s, e)

⟶ mergeSort_using_recursion.cpp

Complexity

Time Complexity ⟶ $O(n \log n)$

Auxillary space ⟶ $O(n)$

Applications

⟶ Sorting large data sets

Advantages

- Stable Algorithm
- Gaurantee worst case performance.

Drawbacks

- Space Complexity, to store the merged sub arrays.
- Not in-place algorithm
- Not always optimal for smaller datasets.

Inversion Count

→ how far the array is from being sorted. And if the array is already sorted, inversion count should be equal to 0.

* Try this with Merge Sort as well, NOT DONE YET.

→ when it breaks down in the pairs

→ inversion_Count.cpp

Lecture 36

Quick Sort

→ placing each element at its correct position

1st step

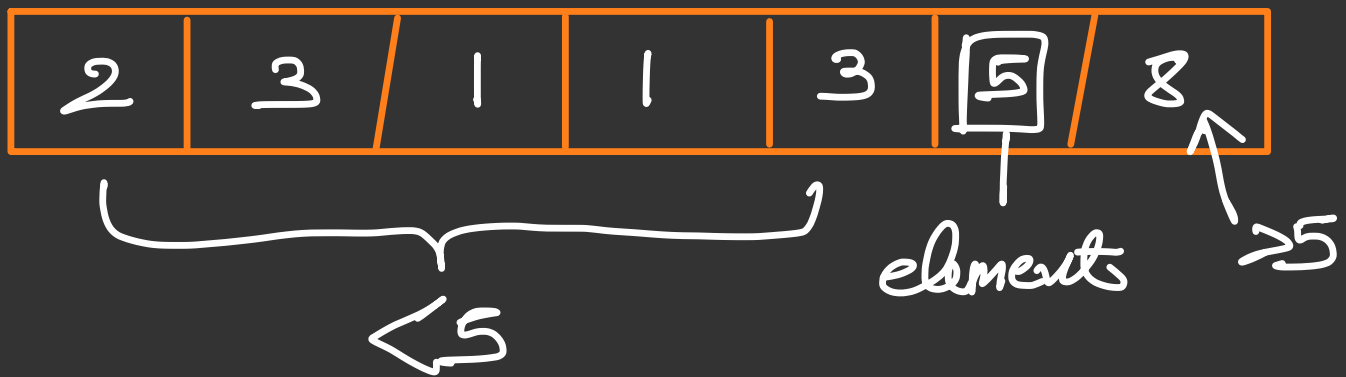
→ right place

↳ elements

Quick Sort

↳ partition

↳ sort using recursion



Code ⟶

```
void QuickSort(int arr[], int s, int e)
{
    // base case
    if (s >= e)
        return;

    // partition
    int p = partition(arr, s, e);

    // recursive call
    QuickSort(arr, s, p-1);
    QuickSort(arr, p+1, e);
}
```

☆ Practice writing code and Dry Run.

Partition Code ⟶

pivot elements

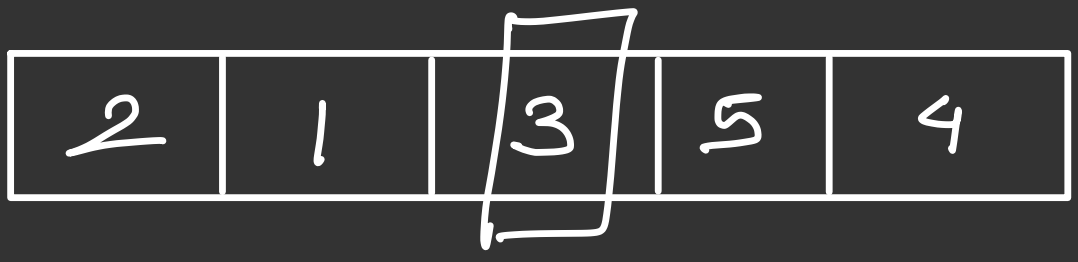
⟶ take a pivot elements

⟶ count all elements < pivot

⟶ pivot ⟶ s+ counts

⟶ <3 and >3

i ⟵ ⟶ j



till i==j

→ Now call Quick Sort for

→ quickSort_recursion.cpp

→ In-Place Sorting algorithm

Space Complexity → O(n)

Time Complexity → O(n log n)

In worst case
↳ O(n²)

avg case and best case
↳ O(n log n)

Read → Why Quick Sort is preferred over MergeSort for sorting Arrays?

Lecture 37

Subsets – LeetCode

// 78

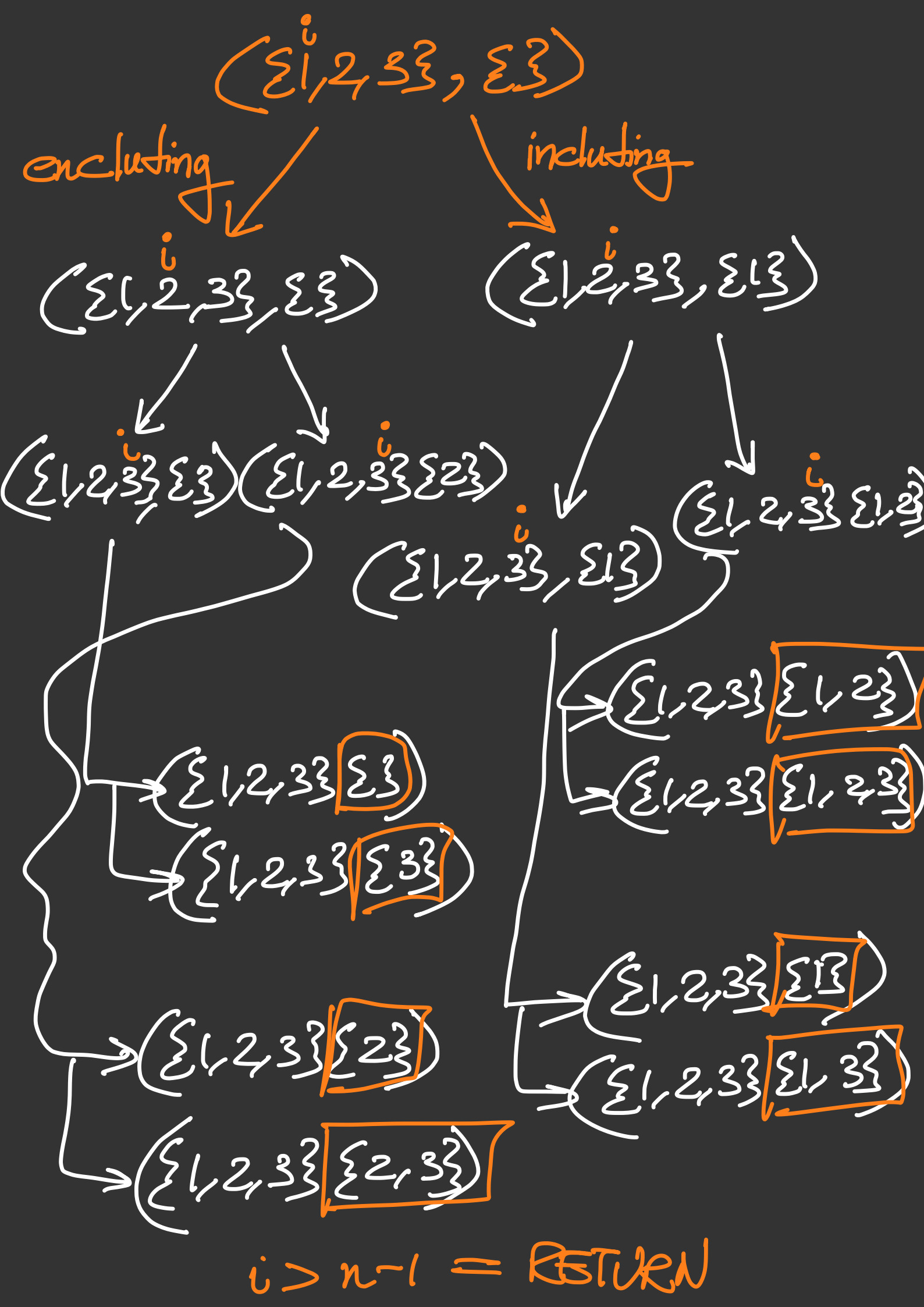
Power Sets ⟶ Set of All Subsets

i/p ⟶ [1, 2, 3]

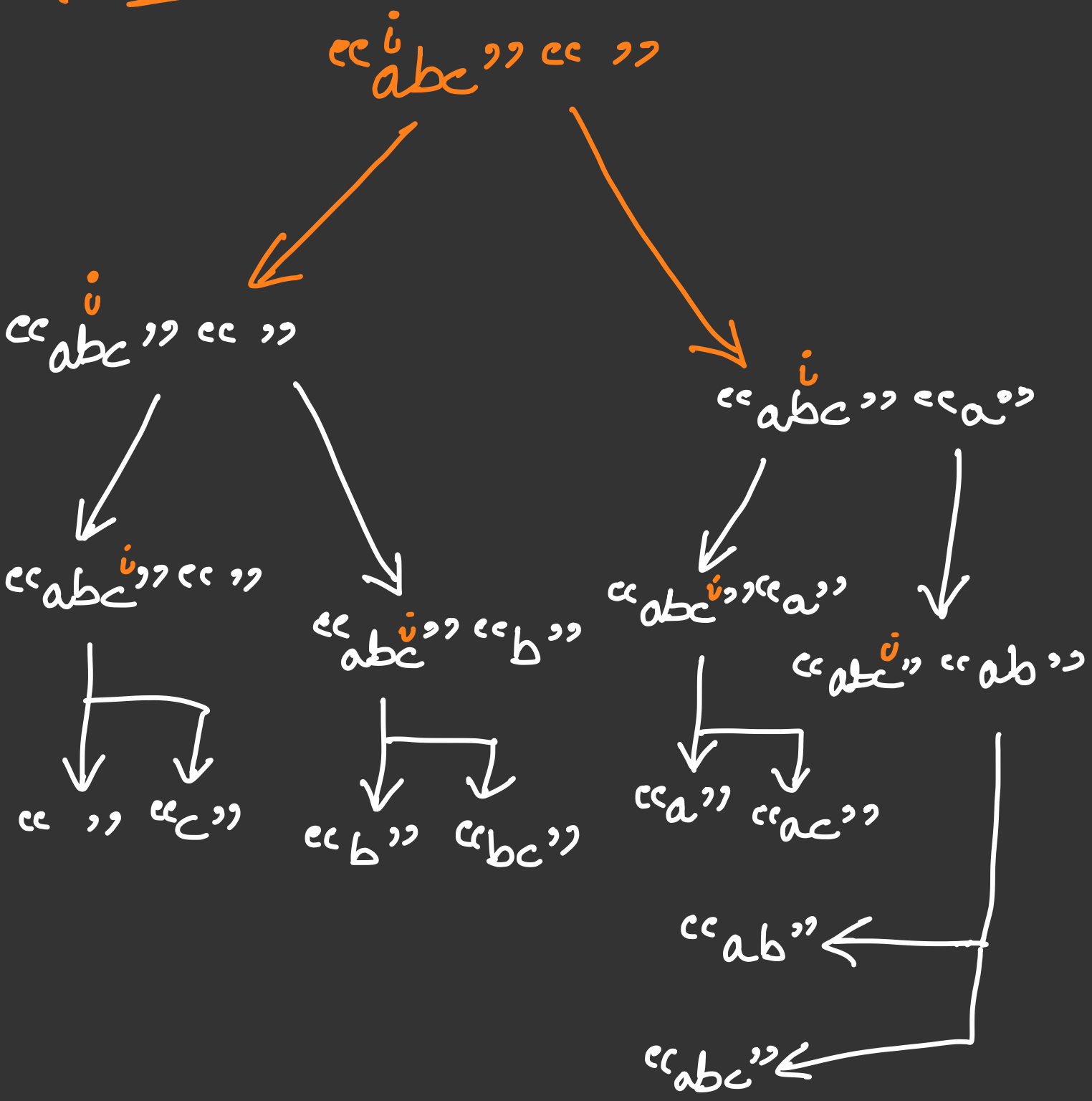
o/p ⟶ { {}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3} }

Total = $2^n = 2^3 =$ (8)

⟶ Subsets.cpp


excluding
including
i > n-1 = RETURN

Subsequences



→ Subsequences.cpp

* Both of the above questions can be done using BIT Manipulation

TRY

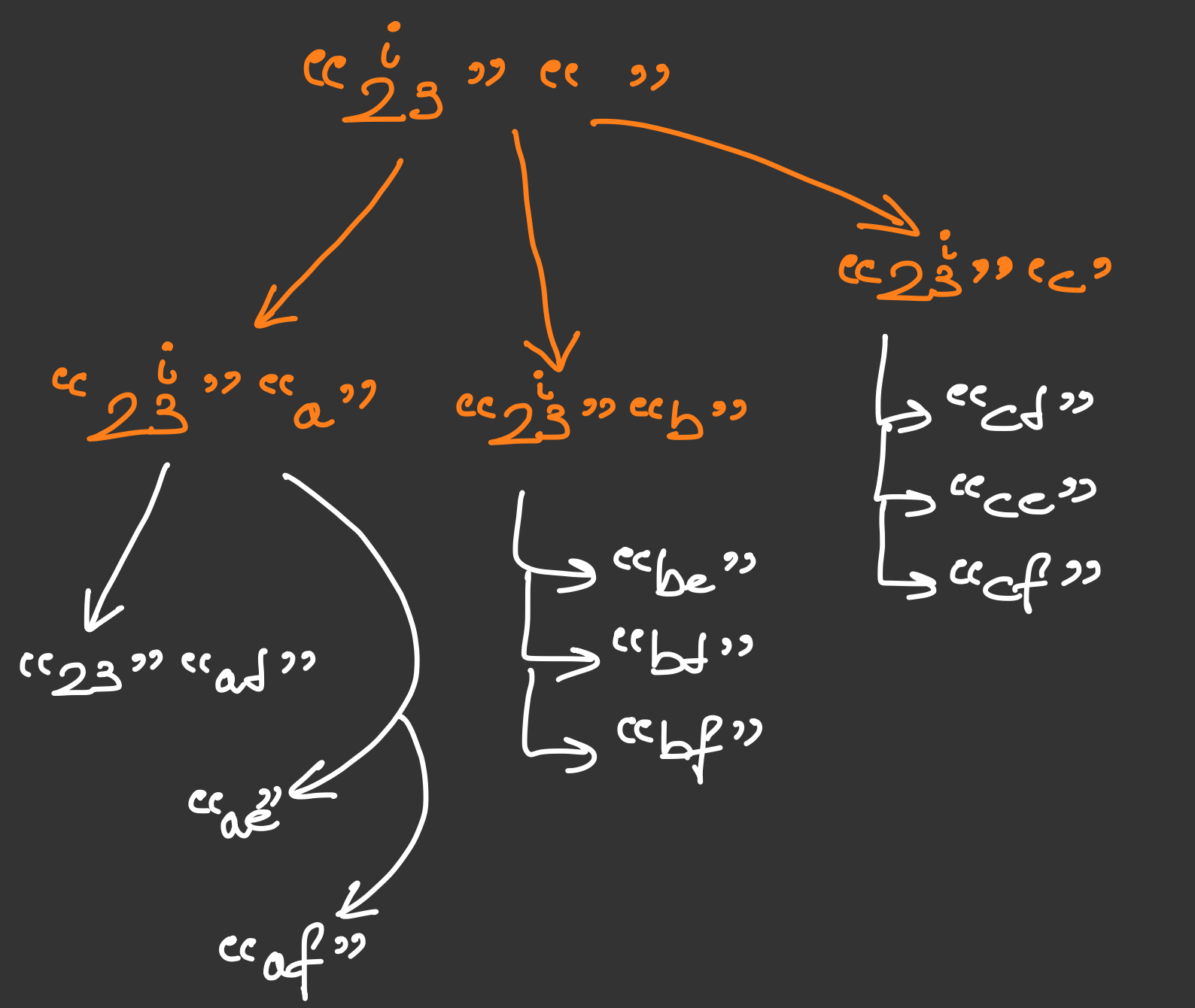
Lecture 38

LeetCode - 17

Letter Combinations of a Phone Number

2 - "abc"
3 - "def"
4 - "ghi"
5 - "jkl"
6 - "mno"
7 - "pqrs"
8 - "tuv"
9 - "wxyz"

→ letterCombination.cpp



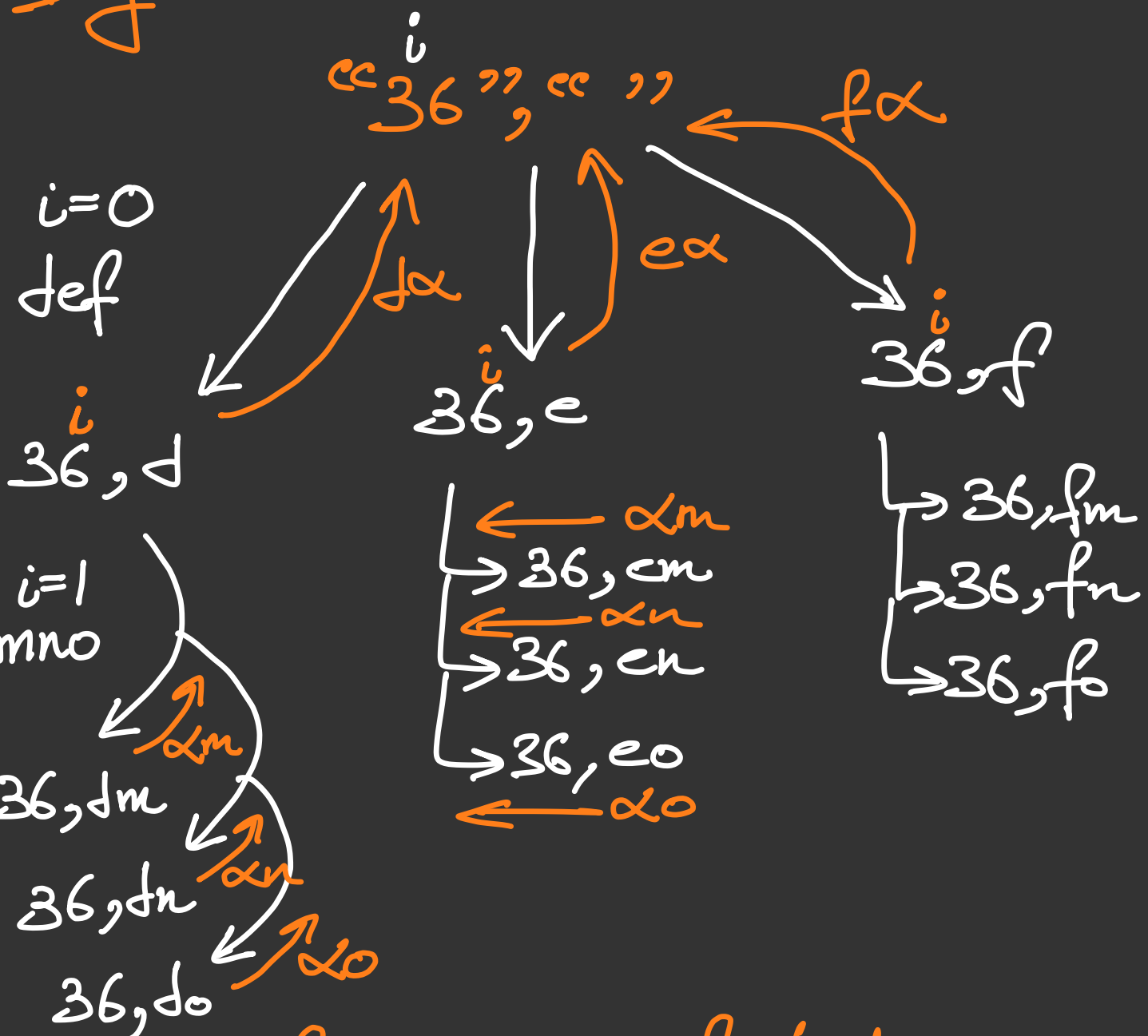
☆ outputs . pop_back ( ) ;

↳ because after going through all of a → ad → use return
→ ae
→ af

to "23", "" and the go for b, c and thus we need "" to be empty.

⟶ '2' − '0' = 2
↳ int

Dry Run ⟶



⟶ first whole of d happens
↳ back to original state
⟶ then e followed by f.

α ⟶ remove / pop

Steps ⟶

1. Declare vector<string> ans;
   Mapping of all strings
   An output string
   An index = 0 in the beginning
   i/p ⟶ str

2. Check if i >= str.length()
   ans.push_back(output);
   return;

3. Now we need the digit from str. `str[i] - '0'`

4. String from mapping using the digit.

⑤ A loop

↳ from string 0 to less than string.length()

everytime add character to the output string

⟶ Recursive call with index+1

⟶ output.pop_back(); Necessary

⑥ Return ans;

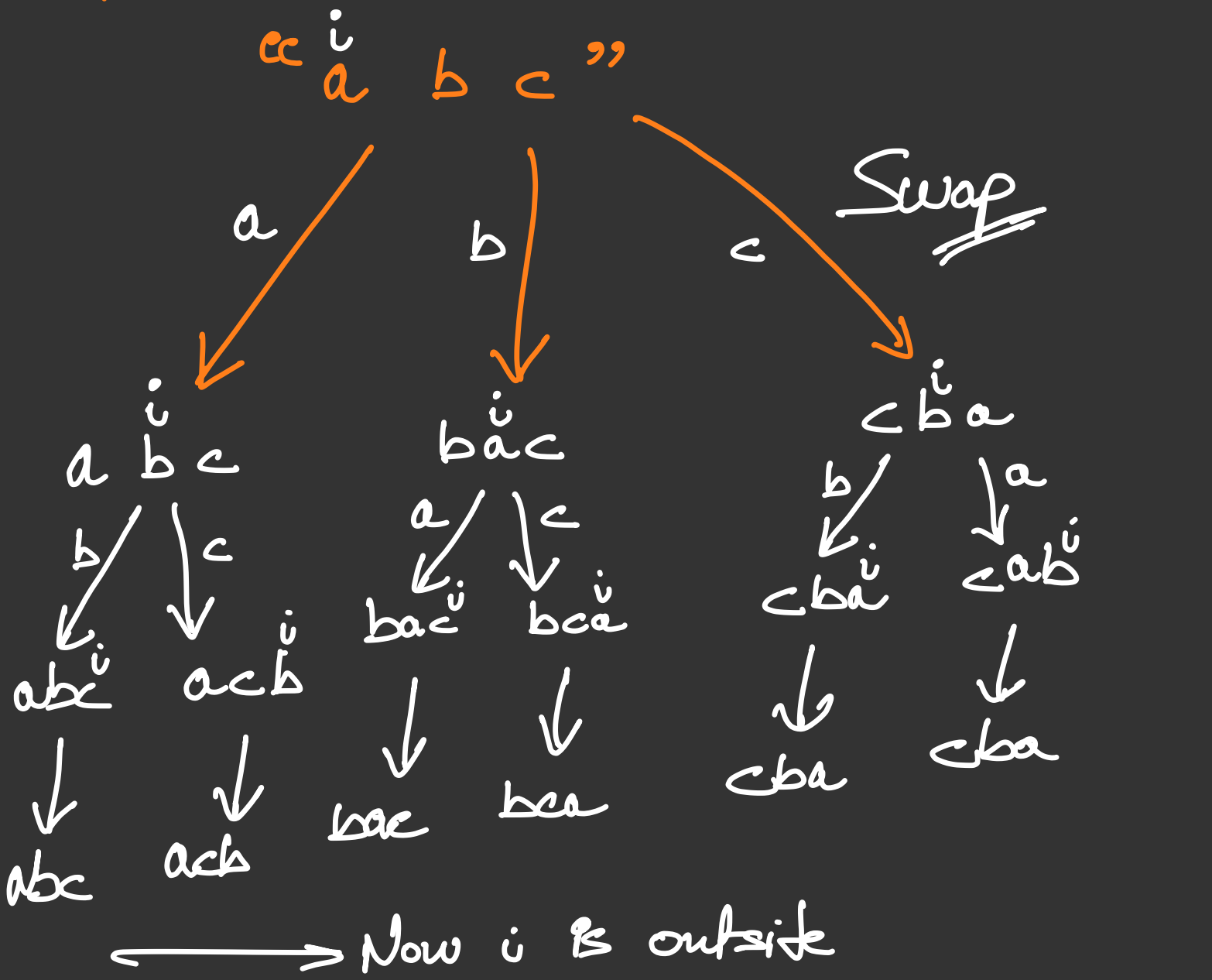
Lecture 39

Permutations of a string

i/p → "abc"

o/p → abc acb bac bca
cab cba

Approaches →



* We are swapping in this approach, and in the end returning the i/p variable itself, thus after returning we have to swap back, to get to the the original input for the next elements.

→ Backtracking logic (swapping back)

* We are using swapping in original string only to reduce space complexity.

Lecture 39.5

PRACTICE

- Merge Sort
- Quick Sort
- Subsets
- Combinations
- Permutations

Merge Sort

→ Split till s >= e
then return

→ Split
- mid
- two recursive calls
  - s, mid
  - mid+1, e

→ then merge
- Again split into two
  store in dynamic arrays
- they are sorted
- Merge two sorted arrays
  (one might be larger than other)

Mistakes

→ s >= e

→ length l = mid - s + 1

→ proper naming

→ len1

→ len2

Quick Sort ✓

Steps →

① get Position

→ place first element in its actual place by counting number of elements less than the element

→ Swap to its actual place

→ Now swap the elements left and right, making sure left ones are left than the element and right ones bigger than element

→ Return position

(2) Sort recursive call

(s, pos-1)

(pos+1, e)

Lecture 40

Rat in a Maze Problem

i/p →

	0	1	2	3
0	1	0	0	0
1	1	1	0	1
2	1	1	0	0
3	0	1	1	1

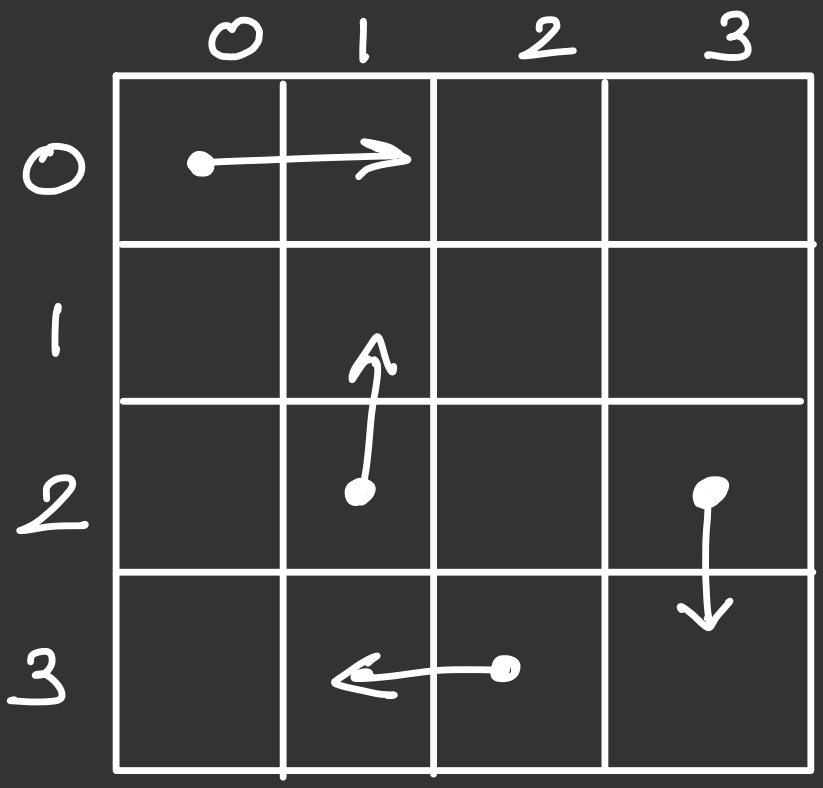
src → (0, 0)

destination → (n-1, n-1)

1 → open path (Allowed)

0 → blocked path (not allowed)

Approach →



Movements

Up → $(x-1, y)$

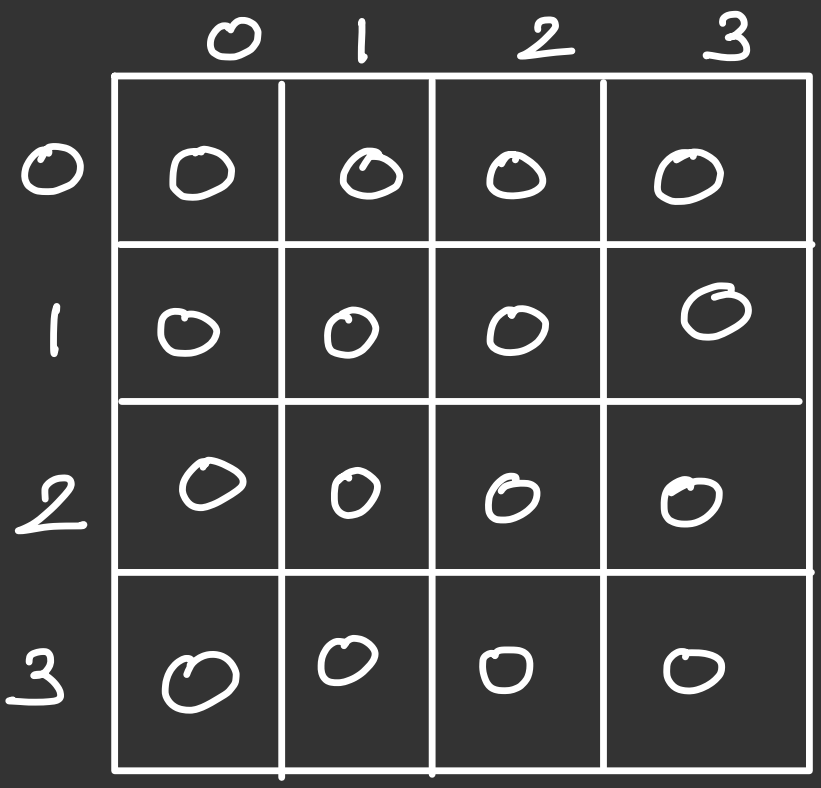
Down → $(x+1, y)$

Left → $(x, y-1)$

Right → $(x, y+1)$

src_x = 0
src_y = 0

* To keep a track of all the visited cells → Visited Array



(i, j) → (k, l)

- should be inside the matrix
- m[k][l] == 1
- v[k][l] == 0

If all condition true move forward and mark v[k][l] = 1 but when going backword, v[k][l] = 0, to find out other solutions.

key points ⟶

1. In the starting mark visited as 1, but while going back from this function call mark it again as 0.

2. pop back the path character from the string after recursive call.

(3) All four moments can be accessed so to get multiple solutions for the path, if it returns after all recursive calls from one movement, lets say down, it check for up, left, right also before returning.

→ rat_in_A_maze.cpp

↳ Look at comments in the code.

Lecture 41

Space and Time Complexity

Linear Search

- TC – O(n)
- SC – O(1)

Recursion

TC → time required as function of input f(n)

Factorial

```
int fact(int n)
{
    if(n==0)
        return;

    return (n * fact(n-1));
}
```

- `if(n==0) return;` → K_1
- `n` → K_2
- `fact(n-1)` → $T(n-1)$

$$f(n) = n * f(n-1)$$

$$T(n) = k_1 + k_2 + T(n-1)$$

$$k$$

$$T(n) = k + T(n-1)$$

$$T(n-1) = k + T(n-2)$$

$$T(n-2) = k + T(n-3)$$

$$\vdots$$

$$T(1) = k + T(0)$$

$$T(0) = k_1$$

$$T(n) = n * k + k_1$$

* ignore constants

$$T(n) = n$$

$$TC = O(n)$$

Binary Search

every time the array gets half

$F(n) = xyz + F(n/2)$

```
bool binarySearch(int *arr, int s, int e , int k ) {


    //base case

    //element not found
    if(s>e)
        return false;

    int mid = s + (e-s)/2;

    //element found
    if(arr[mid] == k)
        return true;

    if(arr[mid] < k) {
        return binarySearch(arr, mid+1, e, k);
    }
    else{
        return binarySearch(arr, s, mid-1, k);
    }
}
```

k_1

k_2

$T(n/2)$

$T(n) = k_1 + k_2 + T(n/2)$

$$T(n) = k + T(n/2)$$
$$T(n/2) = k + T(n/4)$$

$$\vdots$$

$$T(2) = k + T(1)$$
$$T(1) = k$$

$$T(n) = a \text{ times } k$$
$$= a * k$$

$$\frac{n}{2^a} = 1 \longrightarrow a = \log n$$

$$TC \longrightarrow O(\log n)$$

Merge Sort

```
void mergeSort(vector <int>& nums, int s, int e)
{
    if(s>=e)
        return;

    int mid = s + (e-s)/2;

    mergeSort(nums,s,mid);
    mergeSort(nums,mid+1,e);

    merge(nums,s,e);
}
```

k_1

k_2

$T(n/2)$

$T(n/2)$

→ copy $k_3 n$

→ merge $k_4 n$

$$T(n) = k_1 + k_2 + T(n/2) * 2 + k_3 n + k_4 n$$

$$T(n) = k + 2T(n/2) + (k_3 + k_4) n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + k_5 n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + nk$$

$$2 \times T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}k$$

$$= 4T\left(\frac{n}{4}\right) + nk$$

$$T(1) = k$$

$$T(n) = a * nk$$

$$a \longrightarrow \log n$$

$$T(n) = \log n * n$$

$$TC = O(n \log n)$$

Fibonnaci

Base Condition

Addition

$$T(n) = k_1 + k_2 + T(n-1) + T(n-2)$$
$$= k + T(n-1) + T(n-2)$$

$$T(n) = k + T(n-1) + T(n-2)$$
$$T(n-1) = k + T(n-2) + T(n-3)$$
$$T(n-2) = k + T(n-3) + T(n-4)$$
$$T(n-3) = k + T(n-4) + T(n-5)$$

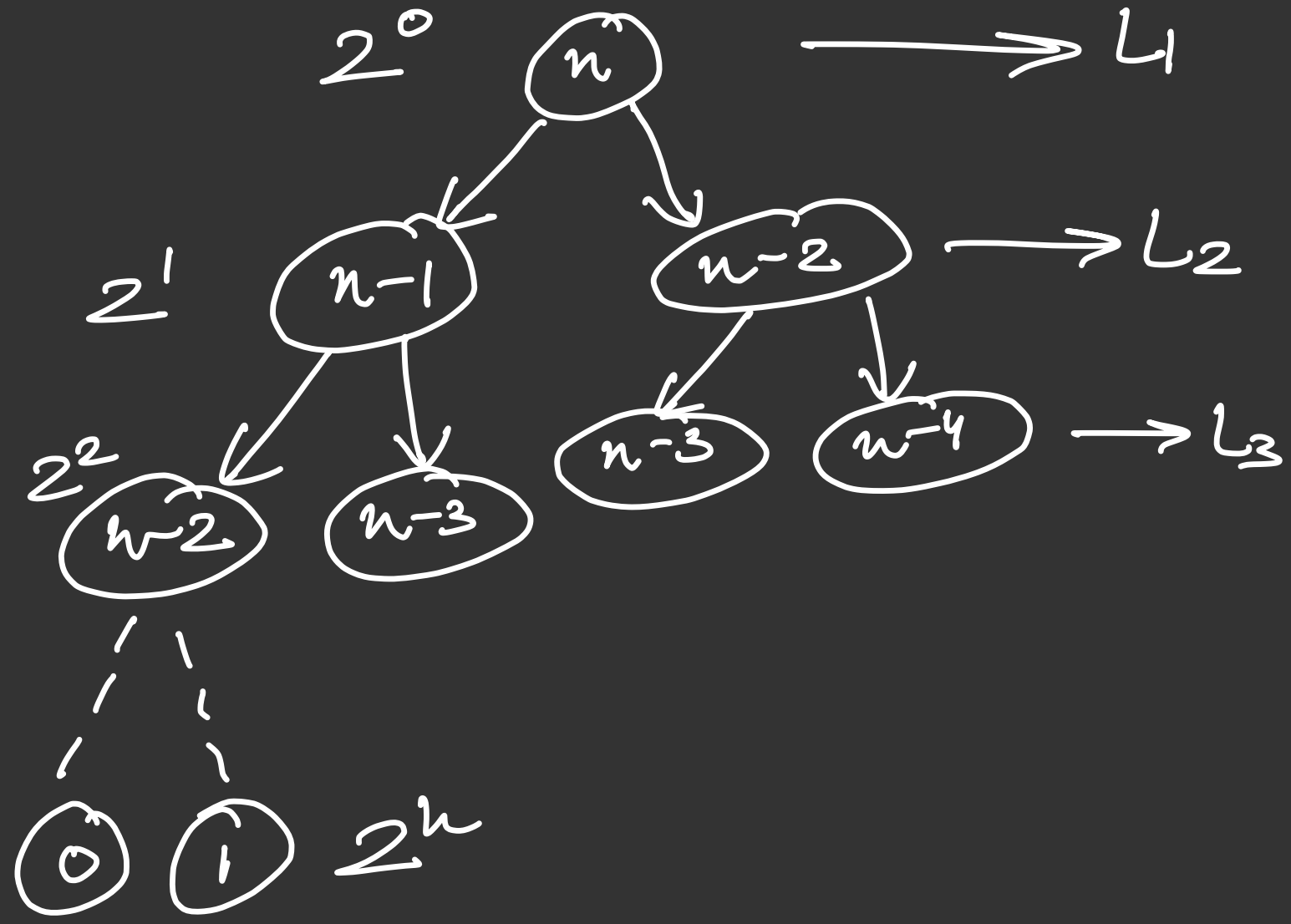
⋮

$$T(1) = k_1$$
$$T(0) = k_1$$

left

thus not a good way to solve

Recursion Tree



each node takes $2 \rightarrow$ k time

total time = k * total nodes

total nodes

$\hookrightarrow 2^0 + 2^1 + 2^2 - - - 2^n$

$= 2^{n+1} - 1$

$$TC = k(2^{n+1} - 1)$$
$$= k(2^{n+1}) - k$$
$$= 2^{n+1}$$
$$= 2^n \times 2$$
$$= 2^n$$
$$= O(2^n)$$

↳ exponential (bad complexity)

⚠ Time Complexity MCQ's on CodeStudio

↳ Link in Bookmark under Contests

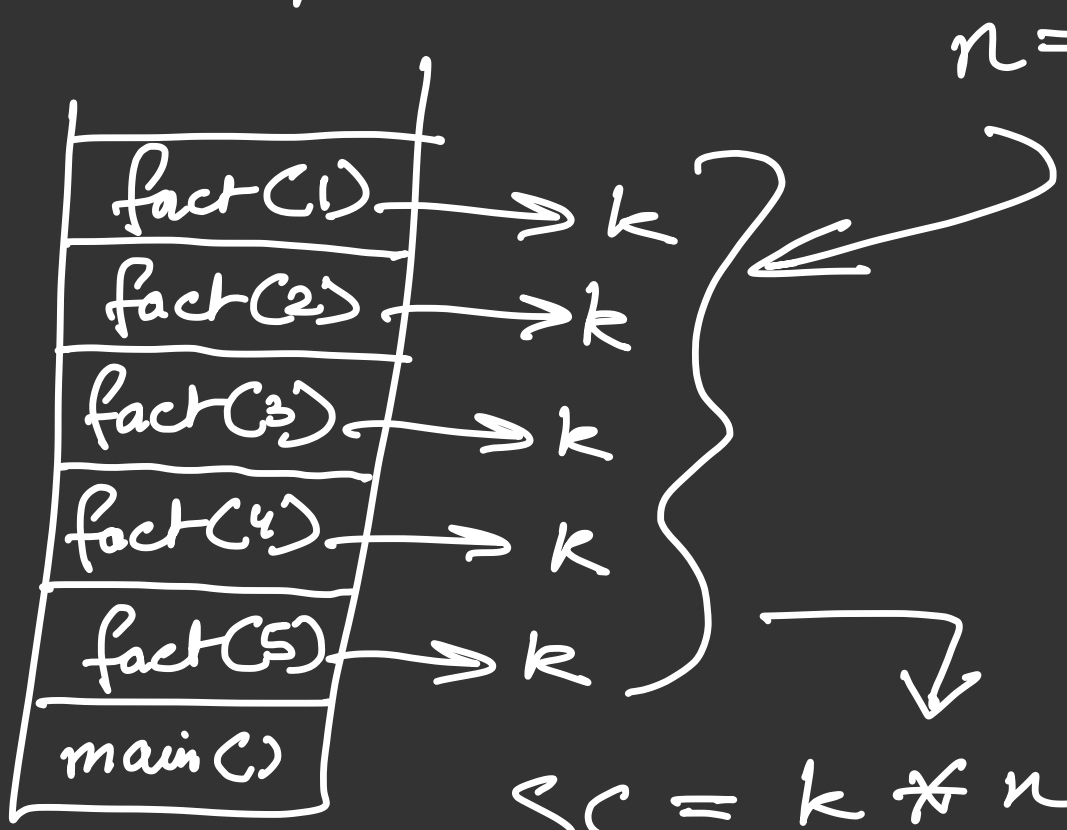
Space Complexity

→ space required as $f(n)$

→ maximum space required at any instance of time

Factorial

Assuming each call takes k space



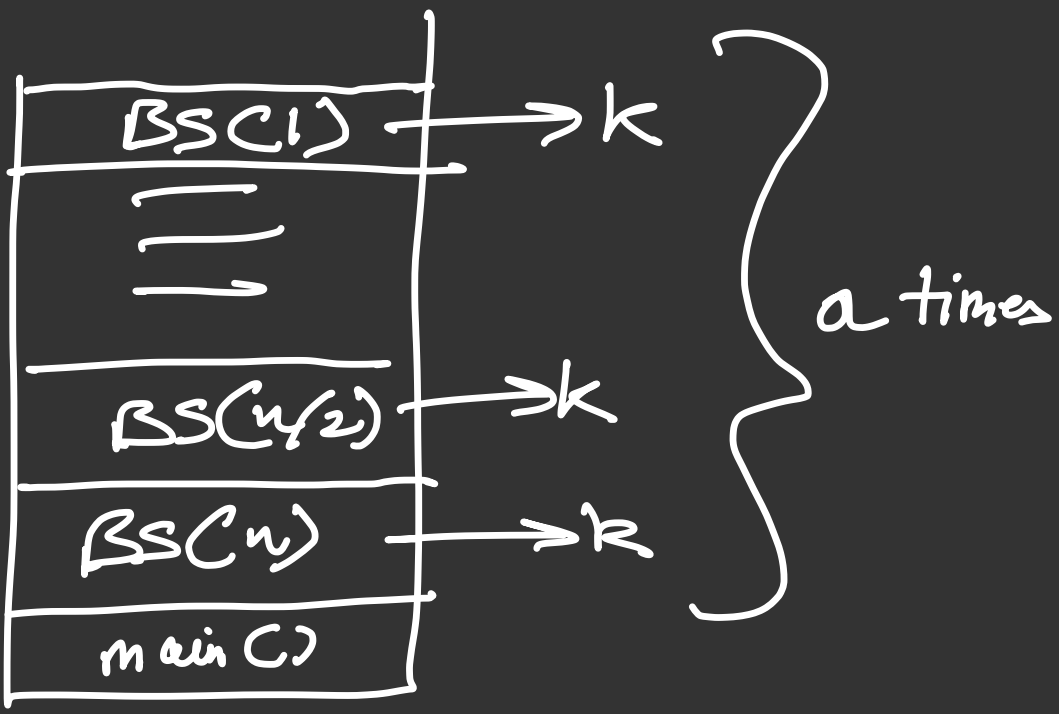
$SC = k * n$

$= O(n)$

Binary Search

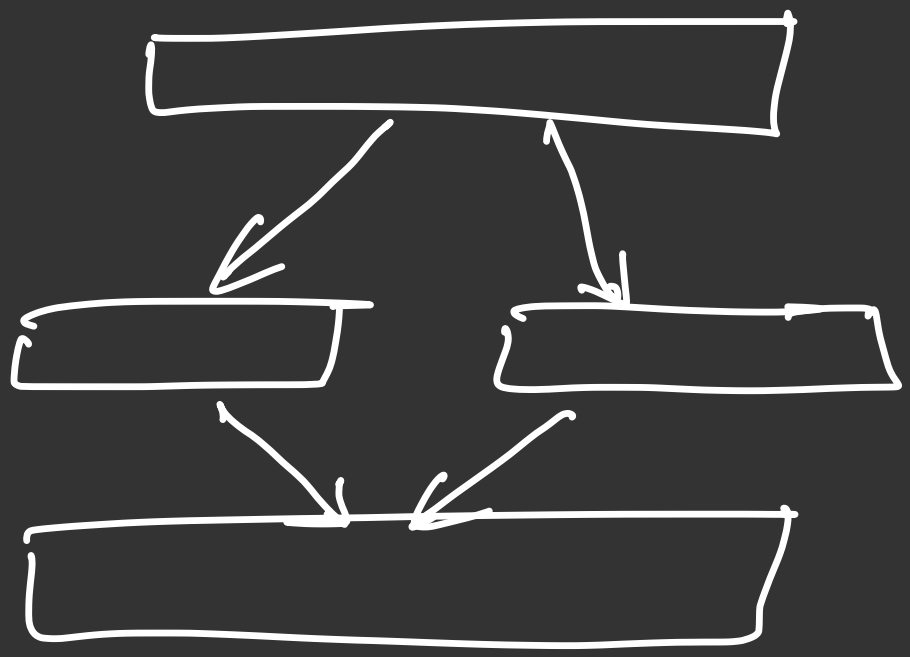
$a = \log n$

1

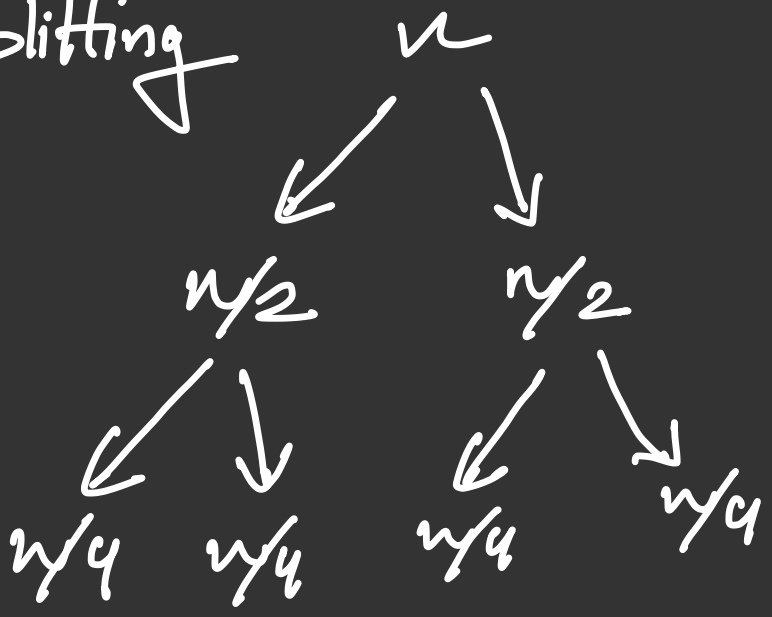


$$SC = k * \log n$$
$$= O(\log n)$$

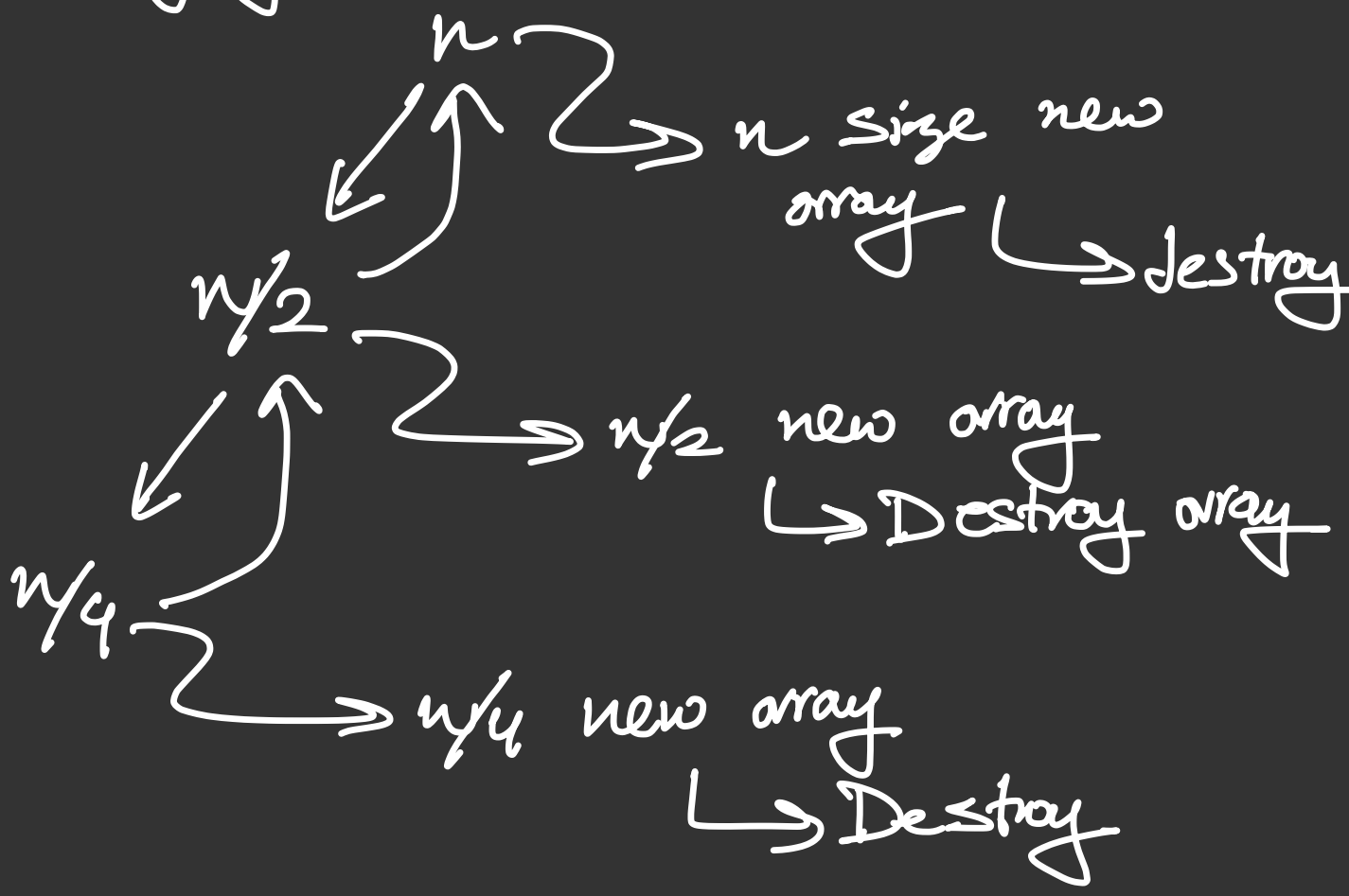
Merge Sort



① Splitting



(2) Merging



→ We consider the maximum

thus → n

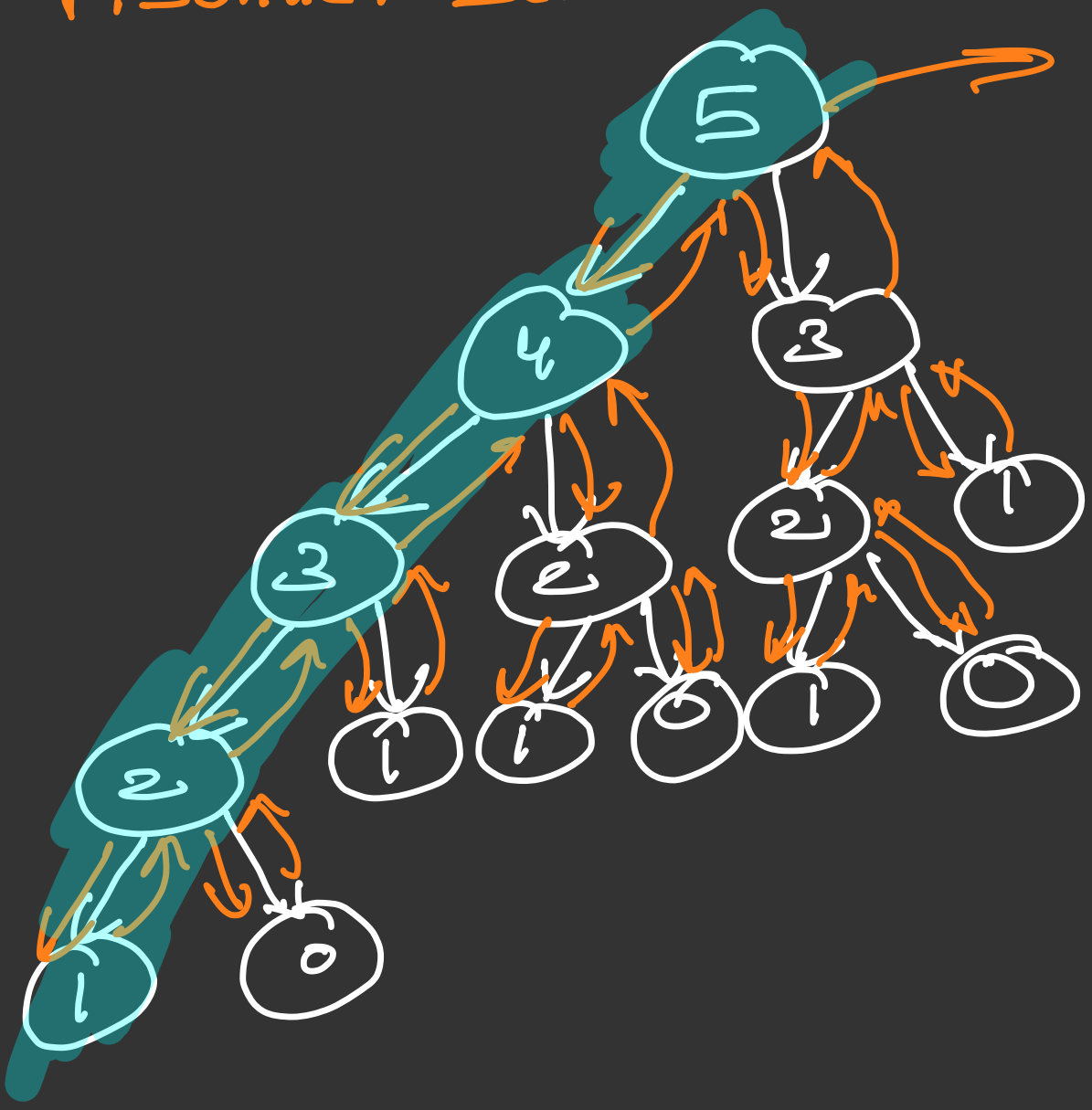
$SC = k \log n + n$

→ $O(n)$

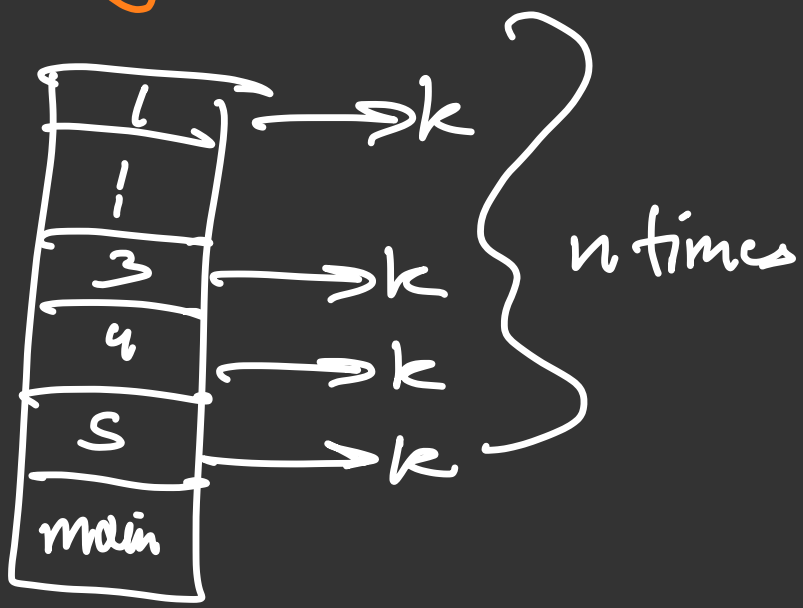
$n \gg \log n$

→ neglecting $\log n$

Fibonacci Series



Longest recursion depth = n



SC = n * k

↳ O(n)

* Solve the Code Studio Questions.