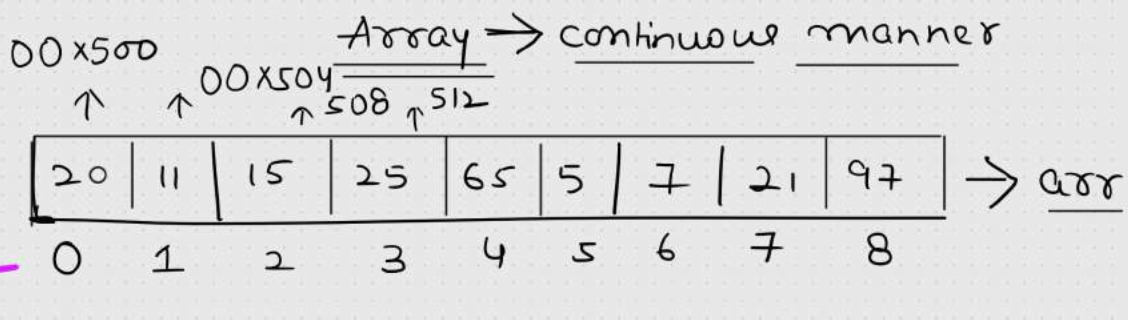
Data Structure

1) Linear DS → Array, Stack, Queue & many more
2) Non-Linear DS → Tree, Graphs



int → 4 Byte

↳ Searching *

Property :-

n = 9

Random access → arr[8]

↳ Print the element present at last position in an array

arr(n-1) →

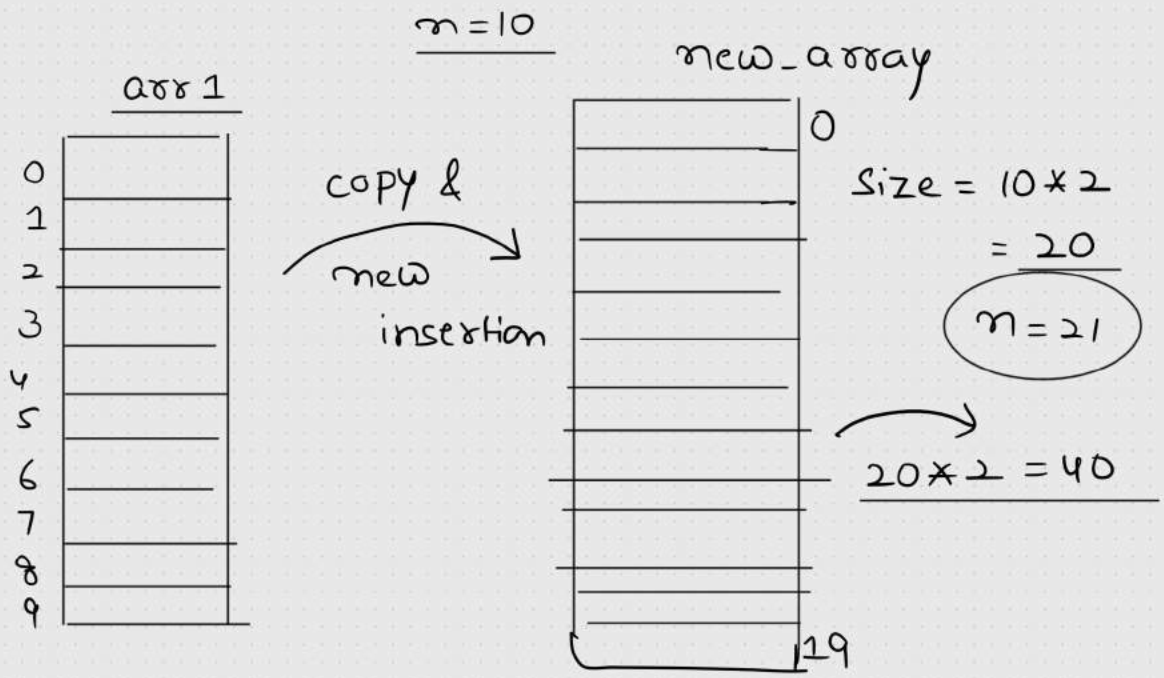
Python → Dynamic array

Array ↔ List → different types

Static array → n = 9 → define the size of an array prior

Dynamic array → no need to define the size of an array prior

n = 10 → Exception (C++, Java)



Abstract Data Types

↳ customized

- List DT → insert(), remove(), replace()
- Stack DT → push(), pop(), peek()
- Queue DT
  - ↳ enqueue, dequeue()

Array

1D Array → [2, 4, 6, 8, 12, 15, 20, 27, 37] → arr

1008

0 1 2 3 4 5 6 7 8 — index

↳ 1004

1000

↳ Base address

arr[6] ↳ 20

int → 4 Bytes

arr[6] → 1000 + (6 − 0) * 4

1000 + 24 = 1024

(contiguous storage of an array elements) (Index)

⟹ Memory address

arr(i) = Base address + (i − Lower Bound (By default → 0) of an index) * size of an element

Random access

2D Array → (Rows, columns)

↳ Matrix computation

{ i → Row index
J → column index

rows ↓ / columns →	0	1	2
0	2	4	6
1	8	10	12
2	14	17	20

[2,4,6], [8,10,12], (14,17,20) ↳ rows

[2,8,14], [4,10,17], [6,12,20] ↳ columns

arr[2][2]

arr[0][1]

1000

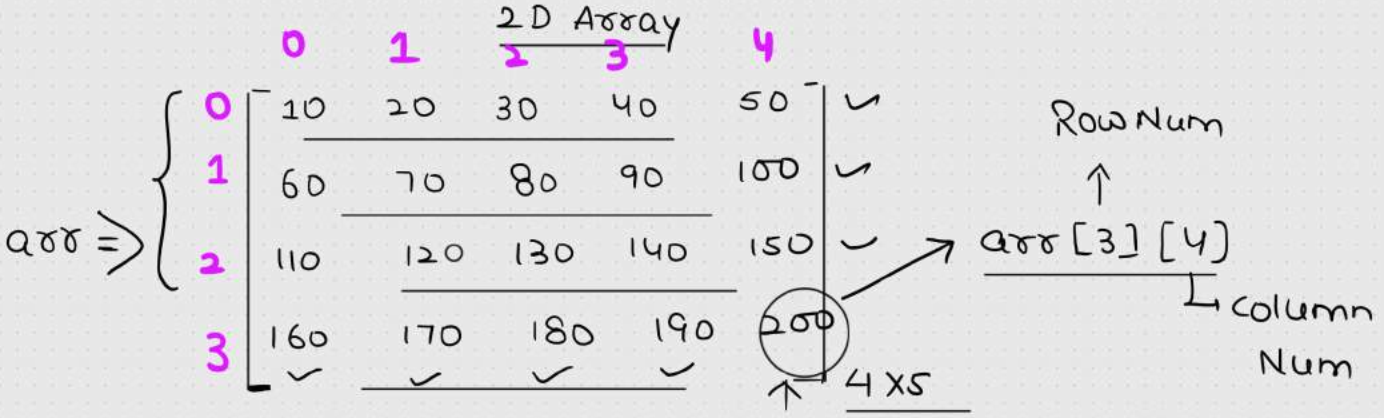
↳ Row major form

0	1	2	3	4	5	6	7	8
2	4	6	8	10	12	14	17	20

2 8 14 4 10 17 6 12 20 → CMF

1) Row major form → Row-wise

2) column major form → column-wise



arr[0][0] ⟶ Base address → 1000

Row major form

nr = 4
nc = 5

Elements need to Skip

Loc(arr[3][4]) ⇒ 1000 + [(3 − 0) * 5 + (4 − 0)] * 2

⇒ 1000 + (15 + 4) * 2

⇒ 1000 + 38

⇒ 1038

Row major form

Loc(arr[i][j]) ⇒ Base address + [(i − lBr) * nC + (j − lBc)] * size of each element

Column major form

Loc(arr[i][j]) ⇒ Base address + [(j − lBc) * nR + (i − lBr)] * size of each element

Searching → Linear Search & Binary Search

arr = [20, 40, 70, 10, 12, 11, 29, 75, 46]
0 1 2 3 4 5 6 7 8

n = 9

x = 47

Linear Search

x = 11

n = len(arr)

```
for i in range(n):        # 0 to n-1
    if arr[i] == x:
        return i
return -1
```

↳ Element is not present in an array

Time complexity
↳ Worst case scenario :- O(n)
↙ Element is present almost at the last index.

Best case scenario :- O(1) → Element is present at initial indexes.

Average case scenario :-

$$\frac{1+2+3+4+---+n}{n}$$

sum of n natural numbers

$$\frac{\frac{n(n+1)}{2}}{n} = O(n)$$

Space complexity → O(1)

Binary Search

↳ Divide & conquer

i = 0

J = 7

1) Sorted array

0	1	2	3	4	5	6	7
2	4	8	12	20	25	50	70

arr = [2, 4, 8, 12, 20, 25, 50, 70]

x = 50

small Problem → single element

x = 50 ⟷ [50] — only one condition

```
if arr(i) == x :
    return i
return -1
```

Big Problem → — overflow

mid ⇒ (i + J) // 2 = (0 + 7) // 2

= 3

✓

⇒ i + (J - i) // 2

(0 + 7) // 2

= 3

$x_2 = 4$, $x_1 = 50$, $x = 12$

0	1	2	3	4	5	6	7
2	4	8	12	20	25	50	70

→ Sorted

BinarySearch (arr, 0, 7, x) — BinarySearch (arr, i, J, x)

1. mid = 3

i <= J

2. arr(mid) == x :

return mid

Psuedocode

12 < 50 — Recursion

arr(mid) < x

BinarySearch(arr, mid+1, J, x)

OR

i = mid + 1

12 > 4

arr(mid) > x — Recursion

BinarySearch (arr, i, mid-1, x)

OR

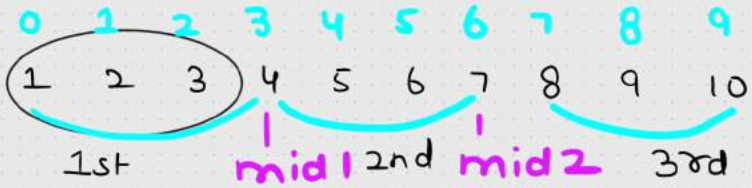
J = mid - 1

return -1

Ternary Search

↳ Sorted array

↳ Search space into three parts



$mid1 = l + (r-l)//3 = 0 + 9//3 = 3$ $\quad l=0,\ r=9$

$mid2 = r - (r-l)//3 = 9 - 9//3 = 6$

$x = 5$ ⟶ Searching element

$arr(mid1) == x$

↳ return mid1

$arr(mid2) == x$

↳ return mid2

$x = 2$

↱ 4

$x < arr(mid1)$

$r = mid1 - 1$ ⟹ 1st search space

$x = 9$

↱ 7

$x > arr(mid2)$

$l = mid2 + 1$ ⟹ 3rd search space

else:

$mid1+1,\ mid2-1$

↳ 2nd search space

Recurrence Relation (Binary Search)

Recursion → Sorted array

mid = i + (j - i)//2

i → starting index

j → ending index

BinarySearch(arr, i, j, x)

- arr[mid] == x → return mid → O(1) → c
- arr(mid) < x → Recursion → BinarySearch(arr, mid+1, j, x) → $T(n/2)$ (Right side of mid)
- arr(mid) > x → Recursion → BinarySearch(arr, i, mid-1, j) → $T(n/2)$ (Left side)

Binary Search

Recurrence Relation

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$T\left(\frac{n}{2}\right)$: Left side / Right side

$a = 1, b = 2$

$T(n) = T(n/2) + c$

$f(n) = \Theta(n^k \log^p c)$

$k = 0, p = 0$

Master's Theorem :-

$\log_b^a = \log_2^1 = 0$

$\log_b a = k \rightarrow$ case 2

$\Rightarrow \Theta(c \cdot \log n)$

$\Rightarrow \Theta(\log n)$

Substitution Method

$$T(n) = T\left(\frac{n}{2}\right) + c \quad \text{---- 1st}$$

$$T(n) = T\left(\frac{n}{2^2}\right) + c + c \quad \text{---- 2nd}$$

$$= T\left(\frac{n}{2^3}\right) + c + c + c \quad \text{--- 3rd}$$

$T(1) = 1$

k times $\left\{ \frac{n}{2^k} = 1 \Rightarrow n = 2^k \right.$

$k = \log_2 n$

$$\Rightarrow T\left(\frac{n}{2^k}\right) + c \cdot k$$

$$\Rightarrow T\left(\frac{n}{2^{\log_2 n}}\right) + c \cdot \log_2 n$$

$$\Rightarrow T\left(\frac{n}{n^{\log_2 1}}\right) + c \cdot \log_2 n$$

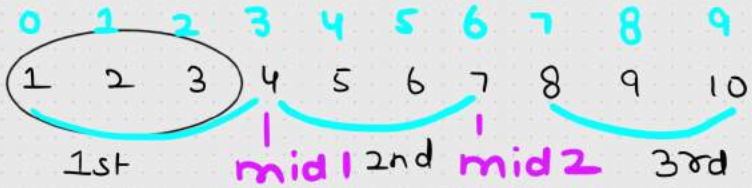
$$\Rightarrow 1 + c \cdot \log_2 n$$

$$\Rightarrow O(\log_2 n)$$

Ternary Search

↳ Sorted array

↳ Search space into three parts



$mid1 = l + (r-l)//3 = 0 + 9//3 = 3$ $\quad l=0,\ r=9$

$mid2 = r - (r-l)//3 = 9 - 9//3 = 6$

$x = 5$ ⟶ Searching element

$arr(mid1) == x$

↳ return mid1

$arr(mid2) == x$

↳ return mid2

$x = 2$

↱ 4

$x < arr(mid1)$

$r = mid1 - 1$ ⟹ 1st search space

$x = 9$

↱ 7

$x > arr(mid2)$

$l = mid2 + 1$ ⟹ 3rd search space

else:

$mid1+1,\ mid2-1$

↳ 2nd search space

Recurrence Relation (Ternary Search)

$$T(n) = T\left(\frac{n}{3}\right) + c \quad \text{——}$$

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/3) + c & n > 1 \end{cases}$$

$$T(n) = T\left(\frac{n}{3^2}\right) + c + c \quad \text{——}$$

$$T(n) = T\left(\frac{n}{3^3}\right) + c + c + c \quad \text{——}$$

k times $\Big\downarrow$ $\frac{n}{3^k} = 1$

$$\frac{n}{3^k} = 1$$

$$n = 3^k$$

$$k = \log_3 n$$

$$T(n) = T\left(\frac{n}{3^k}\right) + c \cdot k$$

$$= T\left(\frac{n}{3^{\log_3 n}}\right) + c \cdot \log_3 n$$

$$\Rightarrow T\left(\frac{n}{n^{\log_3 3}}\right) + c \cdot \log_3 n$$

$$\Rightarrow O(\log_3 n)$$

Search 2D Matrix

↳ Row-wise sorted from left to right

↳ first integer of each row > Last integer of the previous row

2D Matrix →

	0	1	2	3
0	1	3	5	7
1	10	11	16	20
2	23	30	34	60

3x4

m = 3, n = 4

m → # rows

n → # columns

target = 60

left = 0

right = m x n - 1

= 3 x 4 - 1

= 11

target = 3 → true

target = 22 → false

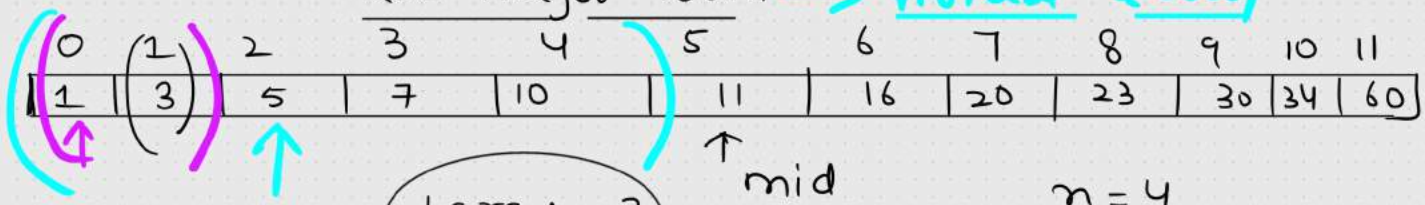
Brute force Approach →

O(m * n)

```
for i = 0 to m-1:          (m)
    for j = 0 to n-1:      (n times)
        if arr[i][j] == target:
            return True
return false
```

mid = (0 + 1)/2 = 0

Row-major form → Virtual array



target = 3

mid

n = 4

(0+4)//2 = 2

(Binary search)

O(log mn)

row = 2 // 4 = 0

col = 2 % 4 = 2

mid = 0 + (11-0)//2 = 5

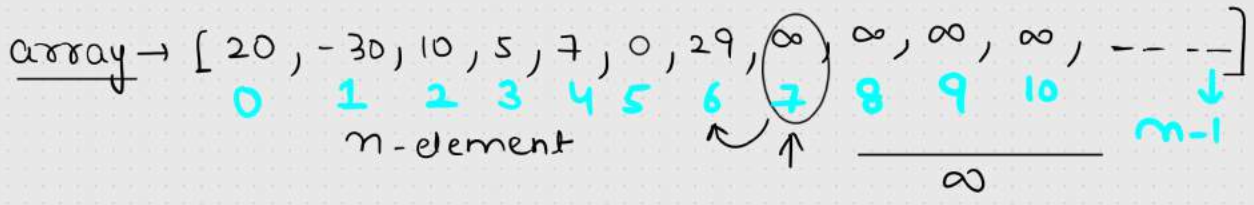
row ← 5 // 4 = 1

column ← 5 % 4 = 1

left = 0

right = mid - 1

Binary Search



Problem → Position of first infinite element

output → 7

Linear Search → for loop

if arr(i) == 'inf'

↳ return i

⇒ O(n)

Binary Search → mid = i + (j-i)//2

if arr[mid] != 'inf'

↳ Right side of the mid

if arr(mid) == 'inf'

Previous index element → != '∞'

↳ return mid

otherwise

↳ Left side of the mid

Note → Array (logically apply binary search)

↓↓

(Left side & right side movement)

Inplace & Outplace Sorting Algorithm

↳ Not using any extra space to sort an array

Outplace → using an extra space to sort the array.

↳ MergeSort

↳ Application of D&C.

Sorting Algorithm

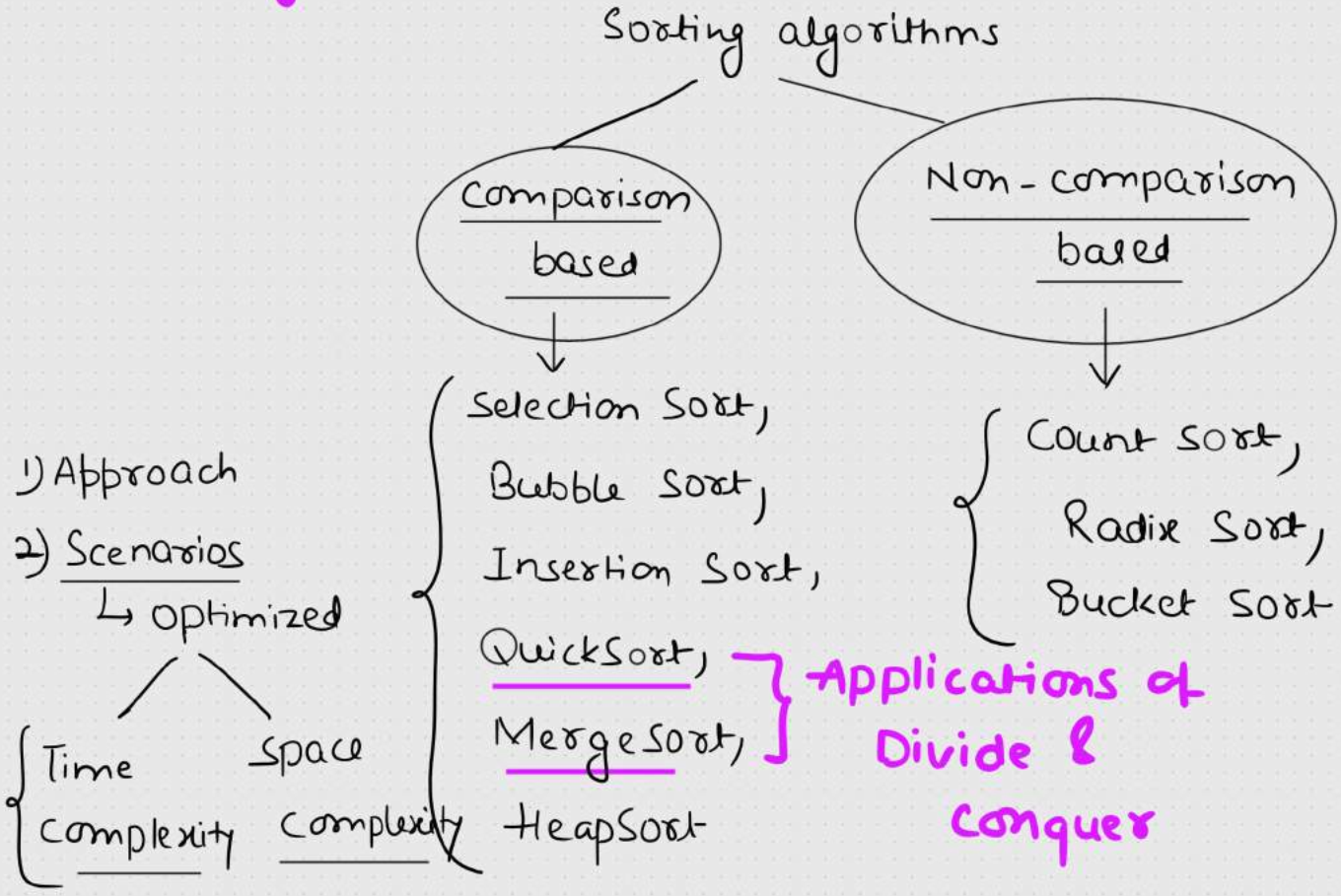
0	1	2	3	4	5	6
20	10	12	17	27	42	39

$n = 7$

↓ Sorting algorithm

0	1	2	3	4	5	6
10	12	17	20	27	39	42

→ Sorted Array



Stable vs Unstable Sorting Algorithms

Stable → Relative order should be maintained after sorting algorithms.

Dictionary → (4,2) (5,7) (4,3) (6,8) (7,10)
(Hashing Data Structure)

↓ Sorting algorithm

Output

(4,2) (4,3) (5,7) (6,8) (7,10)
OR
(4,3) (4,2) (5,7) (6,8) (7,10)

← Stable Sort

↳ Not stable sort

Stable sort

$[4^a, 5, 7, 3, 4^b, 6, 10] \longrightarrow [3, 4^a, 4^b, 5, 6, 7, 10]$
OR
$[3, 4^b, 4^a, 5, 6, 7, 10]$

↓

← Unstable sort

- Quick Sort,
- Selection Sort,
- Heap Sort

Bubble Sort (Comparison Based Sorting)

$\Rightarrow$ [70, 20, 50, 30, 90, 8, 15] $n = 7$

Approach

0	1	2	3	4	5	6	
20	70	70	70	5	90	90	70 > 50
i	50	30			15		70 < 90
	d						

Pass 1

0	1	2	3	4	5	6	
20,	50,	30,	70,	8,	15)	90	$(n-1)$
	30	50	5	70	70		
				15			

Pass 2

0	1	2	3	4	5	6	
20,	30,	50,	8,	15)	70,	90	$(n-2)$
		5	50	50			
			15				

Pass 3

0	1	2	3	4	5	6	
20,	30,	8,	15)	50,	70,	90	$(n-3)$
	5	30	30				
		15					

Pass 4

0	1	2	3	4	5	6
20,	8,	15)	30,	50,	70,	90
5	20	20				
	15					

Pass 5

0	1	2	3	4	5	6
5,	15)	20,	30,	50,	70,	90

Pass 6

0	1	2	3	4	5	6	
5,	15,	20,	30,	50,	70,	90	(1)

Points to Note :-

1) $n = 7 \longrightarrow$ Pass 6

$n \longrightarrow (n-1)$ Passes

2) # comparisons $\longrightarrow \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$

$\hookrightarrow (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$

Sum of n natural numbers

↳ $\frac{n(n+1)}{2}$

$\Rightarrow O(n^2)$

3) swaps
- Best case → 0
- Worst case → $\frac{n(n-1)}{2}$ ↳ $O(n^2)$

Bubble Sort → comparisons + swaps

→ $O(n^2) + O(n^2)$

Implementation → $O(n^2)$

Loops

↳ $O(n^2)$

Space complexity → $O(1)$ ⇒ constant

Implementation (Insertion Sort)

0	1	2	3	4
9	5	1	4	3
	J	i		

i = 1 to 4

i = 1

key = 5

J = 0

True

5 < 9

0	1
~~5~~	~~9~~ 9

while J >= 0 && key < arr(J)

True

arr(J+1) = arr(J)

J = -1 ⇐ J = J-1

arr(J+1) = key

0	1	2
~~5~~ 1	~~9~~ 5	~~1~~ 9

i = 2

key = 1

1 < 5

True

J = ~~1~~ 0 -1

1 < 9

True

while J >= 0 and key < arr(J)

0	1	2
1	5	9

arr(J+1) = arr(J)

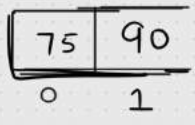
J = J - 1

arr(J+1) = key

Insertion Sort

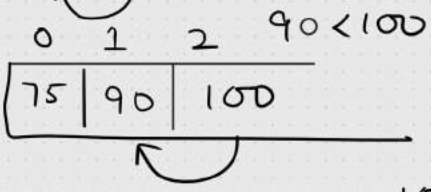
75, 90, 100, 95, 85, 50, 100, 110, 7

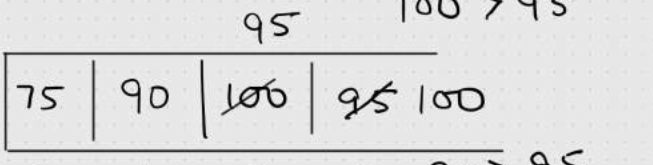
0 1 2 3 4 5 6 7 8



Logic Building

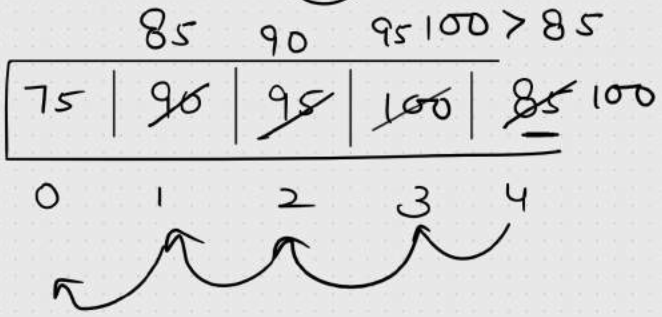
↳ Deck of cards

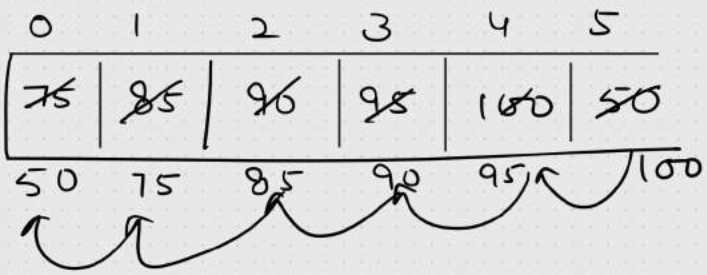




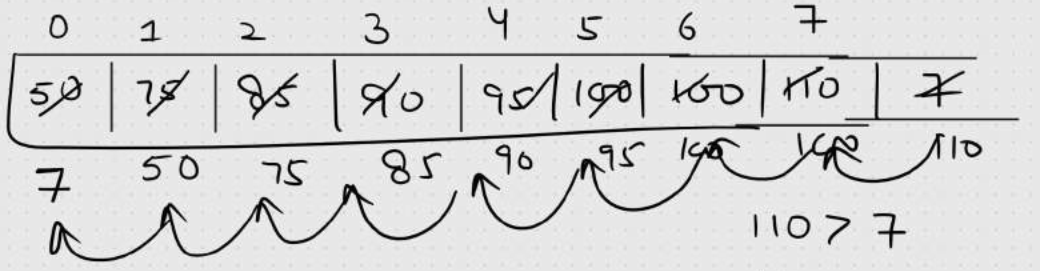
90 > 85

95 > 85





100 < 110



110 > 7

100 > 7

100 > 7

95 > 7

90 > 7

85 > 7

75 > 7

50 > 7

7, 50, 75, 85, 90, 95, 100, 100, 110

0 1 2 3 4 5 6 7 8

↳ Sorted array

Best case

Ascending order — $n-1$ comparisons, 0 swaps

10, 20, 30, 40, 50
0 1 2 3 4

$n = 5$

10 — No comparison

10 | 20 — $10 < 20$ ①

10 | 20 | 30 — $20 < 30$ ②

10 | 20 | 30 | 40 — $30 < 40$ ③

10 | 20 | 30 | 40 | 50 — $40 < 50$ ④

n — $(n-1)$ comparisons $= O(n)$, 0 swaps

$$T(n) = O(n)$$

Worst case

↳ Decending order $n = 5$

50, 40, 30, 20, 10

0 1 2 3 4

n elements

C	S
1	1
2	2
3	3
4	4
1	1
$n-1$	$n-1$

50

0

50 > 40

~~50~~ 40 | ~~40~~ 50

0 1

50 > 30

40 > 30

~~40~~ | ~~50~~ | ~~30~~

30 40 50

50 > 20

40 > 20

30 > 20

20 30 40 50

~~30~~ | ~~40~~ | ~~50~~ | ~~20~~

50 > 10

40 > 10

30 > 10

20 > 10

10 20 30 40 50

~~20~~ | ~~30~~ | ~~40~~ | ~~50~~ | ~~10~~

comparisons → $1+2+3+---+n-1$

$$\Rightarrow \frac{(n-1)\,n}{2} = O(n^2)$$

Swaps

↳ $1+2+3+\ldots+n-1$

$$\frac{(n-1)n}{2} = O(n^2)$$

$$T(n) = O(n^2)$$

Best case → Ascending order
(Almost sorted → ascending order)

INSERTION SORT ⟹

$\Downarrow$

$O(n)$

Worst case → Descending order

$\Downarrow$

$O(n^2)$

Selection Sort

array = [19, 38, 45, 79, 50, 27, 29]
0 1 2 3 4 5 6

min = 0 ~~1~~ 4
↳ index of the minimum value

Pass 1

[19, (27, 38, 45, 79, 50, 38, 29)]
0 (1 2 3 4 5 6)

min = ~~1~~ 5

Pass 2

[19, 27, (29, 45, 79, 50, 38, 45)]
0 1 (2 3 4 5 6)

min = ~~2~~ ~~5~~ 6

Pass 3

[19, 27, 29, (38, 79, 50, 38, 45)]
0 1 2 (3 4 5 6)

min = ~~3~~ ~~4~~ 5

Pass 4

19, 27, 29, 38, (45, 79, 50)
0 1 2 3 (4 5 6)

min = ~~4~~ 6

Pass 5

19, 27, 29, 38, 45, (50, 79)
0 1 2 3 4 (5 6)

min = ~~5~~ 6

Pass 6

0 1 2 3 4 5 6
19, 27, 29, 38, 45, 50, 79

Note

↳ At every pass, only one swap is required.

↳ $(n-1)$th Pass → 1 swap
|
1st Pass → 1 swap

swaps
$(n-1)$ swaps
↳ $O(n)$

↳ # comparisons
$(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)\,n}{2}$

$\Rightarrow O(n^2)$

Time complexity →
- comparisons → $O(n^2)$
- swaps → $O(n)$

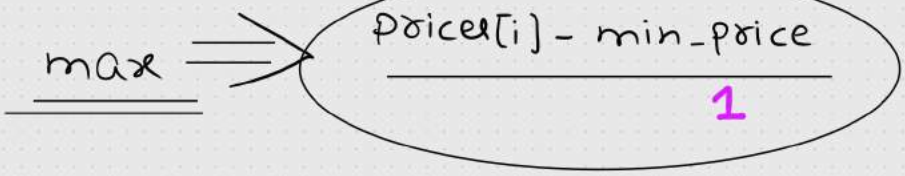
$\Rightarrow O(n^2)$

Best Time to Buy & Sell Stocks

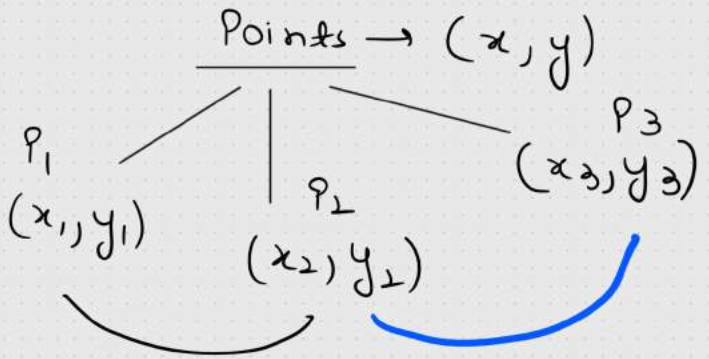
↳ prices = [7, 1, 5, 3, 6, 4, 15]

1 2 3 4 5 6 7

output → max_profit = 15 - 1 = 14



Collinear Points



(P_1, P_2, P_3)

↳ Collinear or not

$$m_1 = \frac{y_2 - y_1}{x_2 - x_1}$$

$$m_2 = \frac{y_3 - y_2}{x_3 - x_2}$$

Approach 1

$m_1 = m_2$ → Points are collinear

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{y_3 - y_2}{x_3 - x_2}$$

$$(y_2 - y_1)(x_3 - x_2) = (x_2 - x_1)(y_3 - y_2)$$

Approach 2

Area of triangle = 0

(collinear)

$$\Rightarrow \frac{1}{2}\left(x_1 * (y_2 - y_3) + x_2 (y_3 - y_1) + x_3 (y_1 - y_2)\right)$$

Majority Element

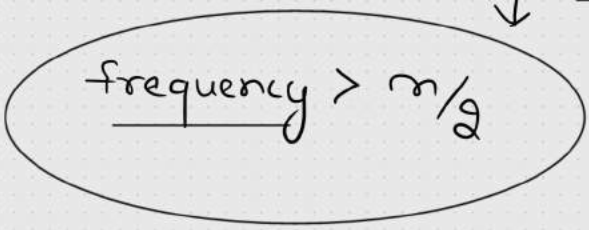
0 1 2 3 4 5 6

nums = [2, 2, 1, 1, 1, 2, 2]

Output = 2

n = 7

Majority Element



2 : 4 > 7/2 (3)

Approach 1 : Sort the array — nlogn

↳ O(nlogn) (1, 1, 1, 2, 2, 2, 2)

↳ (2)

Approach 2 : Hash Data Structure (Dictionary)

(key, value)

2 : 4 * → Output = 2

1 : 3

TC = O(n)

SC = O(n)

Hashtable

key	value
2	4
1	3

→ (2)

Approach 3: Boyer-Moore Voting Algorithm

Example 1

{ TC → O(n)
  SC → O(1)

2 ⟷ 4 > 7/2 (3)

↳ Yes

[2, 2, 1, 1, 1, 2, 2]

candidate = ~~None~~ ~~1~~ 2

count = ~~0 1 2 1 0 1 2 1 0~~ 1

num == candidate
↳ count += 1

count == 0
↳ candidate = num

else:
count -= 1

Example 2

n = 5

n/2 = 5/2 = 2

0 1 2 3 4
[2, 3, 4, 3, 3]

output = 3 ⟷ 3 > 2 (n/2)

candidate = ~~None~~ ~~2~~ ~~3~~ 3 ~~4~~

count = ~~0 1 0 1~~ 0 ~~1~~ 2 ~~0 1~~

3 ⟷ 3 > n/2

↳ Yes

Example 3

0 1 2 3 4

(2, 3, 7, 3, 4)

↑ ↑ ↑ ↑ ↑

$n = 5$

$n/2 = 5/2 = 2$

$3 \leftrightarrow 2 \not> n/2$

↳ No majority element

~~2~~ ~~3~~ ↑ ~~3~~ 4

candidate = ~~None~~

count = ~~0~~ ~~1~~ ~~0~~ ~~1~~

~~0~~ ~~1~~ ~~0~~

1 ~~0~~ ~~1~~

$4 \leftrightarrow 1 \not> n/2$

↳

No majority element

Sort colors

↳ Red, green, blue
↓ ↓ ↓
0 1 2

- [1, 1, 0, 2, 0, 2] ← Input
- [0, 0, 1, 1, 2, 2] ← output

1) MergeSort / QuickSort
   ↳ $\Theta(n \log n)$

Optimized approach

2) Two pointers
   - p_0 → to store 0's in extreme left
   - p_2 → to store 2's in extreme right

n → size of an array

$p_0 = 0$
$p_2 = 5$

0	1	2	3	4	5
2	0	2	1	1	0

↑ p_0, curr = 0

$p_2 = n - 1$

$\Theta(n)$

while curr <= p2:

```
if nums(curr) == 0:
    swap(nums(curr), nums(p0))
    p0 += 1
    curr += 1
```

```
elif nums(curr) == 2:
        swap(num(curr), nums(p2))
        p2 -= 1

else:
    cur += 1

return nums
```

Top k frequent Elements

[1, 1, 1, 2, 2, 3] → arr n = 6

Freq. 0 1 2 3 4 5

1	3 ←
2	2 ←
3	1
↓	↓
key	value

k = 2 → Top 2 frequent elements

[1,2] → Expected output

counter → to count the frequency of each element in an array

heapq.nlargest(k, count.keys())

⇒ arr = [1, 2, 3, 4] n = 4

k = 4

↳ {1, 2, 3, 4}

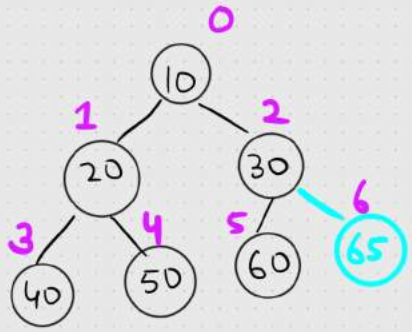
⇒ arr = [1, 1, 2, 3] n = 4

k = 4

↳ {1, 2, 3}

Deletion in Minheap

Insertion
- worst & average = $O(\log n)$ ⟷ n elements
- Best case = $O(1)$ = constant



$30 < 65 \Rightarrow$ valid Minheap

1 comparison & 0 swap

elements = $n * 2^n$ (Insertion)

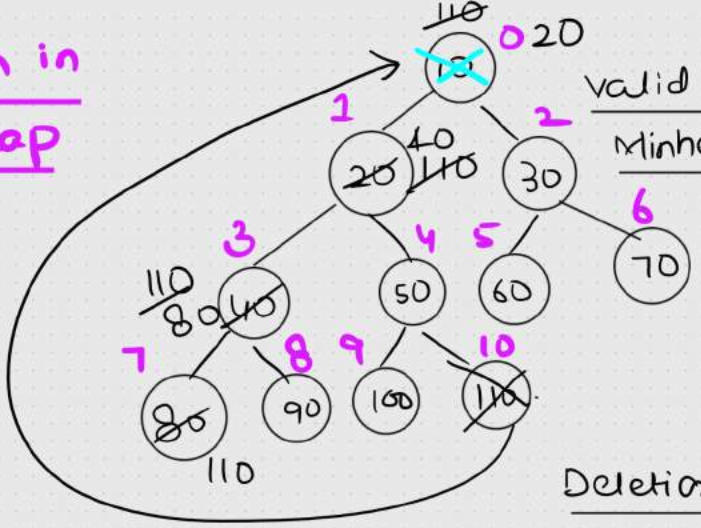
$= \log_2(n * 2^n)$

$= \log_2 n + \log_2 * 2^n$ ($n * \log_2 2$, $\log_2 2 = 1$)

$= \log_2 n + n$

$= O(n)$ ⟸

Deletion in minheap

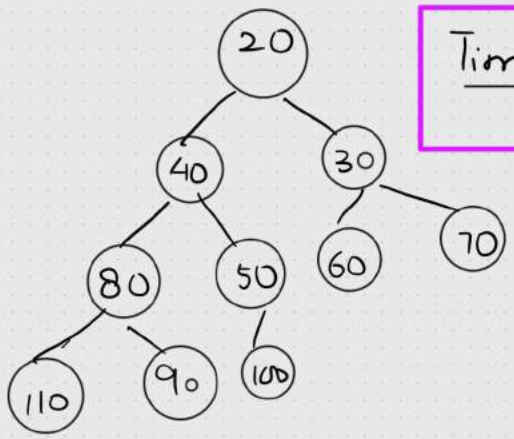


Valid Minheap { 1) CBT
2) Parent < child node data data

⇒ 10 Delete

Deletion → # comparisons
↳ 2 * logN
⇓
Root Node to Leaf Node } Left child & Right child

swaps → logN



Time complexity → O(logN)

comparison based sorting

Heapsort

Deletion - 1st time → 1st smallest (logN)
2nd time → 2nd smallest (logN)
3rd time → 3rd smallest (logN)
n time → nth smallest (logN)

Array ⇓ Sorted array ⇓ increasing order

elemew-

Overall time complexity = O(nlogn)

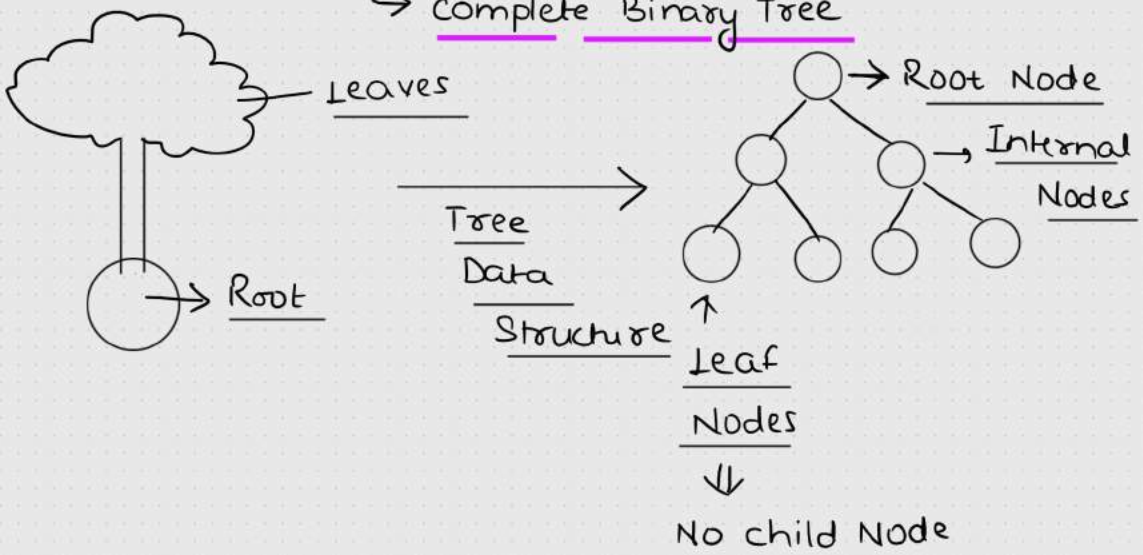
Maxheap → Sorted array
Deletion → Decreasing order

n number of times

Heapsort → $O(n \log n)$

Heap Data Structure

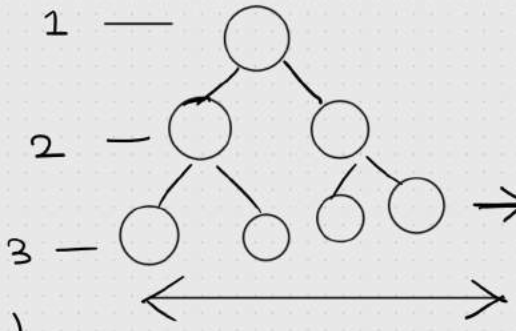
↳ complete Binary Tree



Binary Tree → # child Nodes → 0, 1, 2

↳ Atmost the node → 2 child nodes

full Binary Tree → Every node has 2 child nodes apart from the leaf nodes.

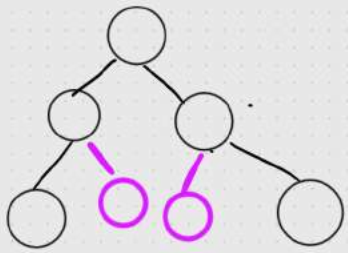


(CBT)

Complete binary tree → 1) After completion of first level, then only move towards filling of next level.

2) After completion of left side node, then only go for the completion of right side node.

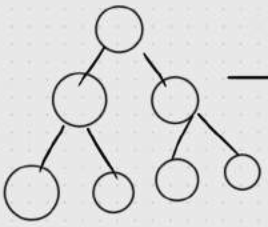
Not CBT ←



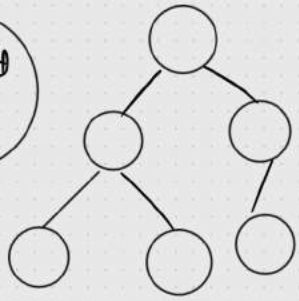
→ Not CBT

Full Binary Tree

↑



→ Valid CBT



→ valid CBT

↓

Almost complete Binary Tree

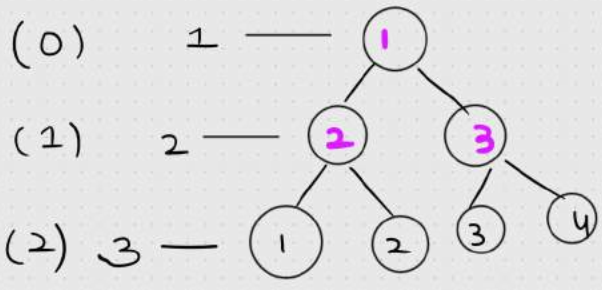
Almost complete Binary Tree

↳ always node → single child at the leaf node

CBT = FBT + ACBT

↓

flexible



nodes

↑

$n = 7$, $k = 3$

↓

Levels

Leaf Node = 4

Non-Leaf Node = 3

Properties of Full Binary Tree (Complete Binary Tree)

1. $n = 2^k - 1 = 2^3 - 1$

$n = 7$

$n = 2^k - 1$

$2^k = n + 1$

$k = \log_2 (n+1)$

↓

Levels

$n = 63$

$k = \log_2 (63+1)$

$k = \log_2 64$

$k = \log_2 2^6$

$k = 6 \log_2 2$

$k = 6$

2. $\lceil n/2 \rceil$ = Number of Leaf Nodes

$\lceil 7/2 \rceil = 4 =$ # Leaf Nodes

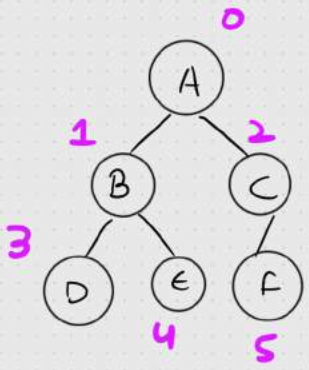
$\lfloor n/2 \rfloor$ = Number of non-leaf Nodes

$\lfloor 7/2 \rfloor = 3 =$ # Non-Leaf Nodes

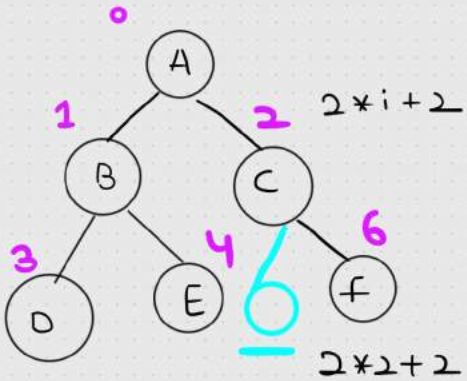
Heap → Complete Binary Tree

↳ Array ??

Heap → Minheap Tree, Maxheap Tree → Array for the storage purpose



2*i+1



very less wastage of space ↳ ACBT

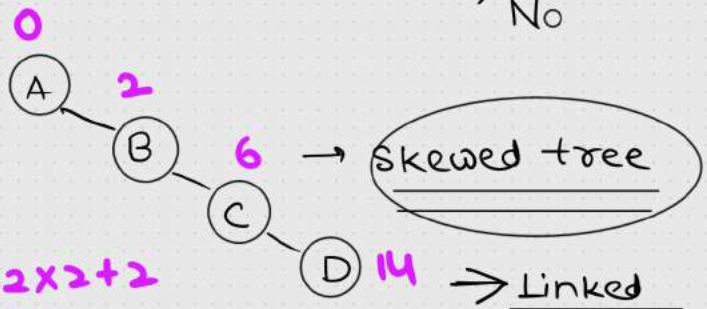
0	1	2	3	4	5
A	B	C	D	E	F

CBT → ??

↳ No

Left index → 2*i+1

Right index → 2*i+2



→ Skewed tree → Linked List

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A		B				C								D	

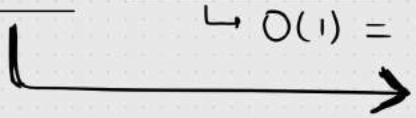
↳ lot of wastage of space

Minheap Tree

↳ Parent node data < child node data

Root Node → smallest element

↳ O(1) = constant



a(0)

Maxheap Tree

↳ Parent node data > child node data

Root Node → maximum/largest element

↳ O(1) = constant

a[0]

Heapsort Algorithm

1) Build heap → minheap or maxheap (Increasing or Decreasing
   ↳ O(n) ↓ minheap ↳ maxheap
   sorted fashion)

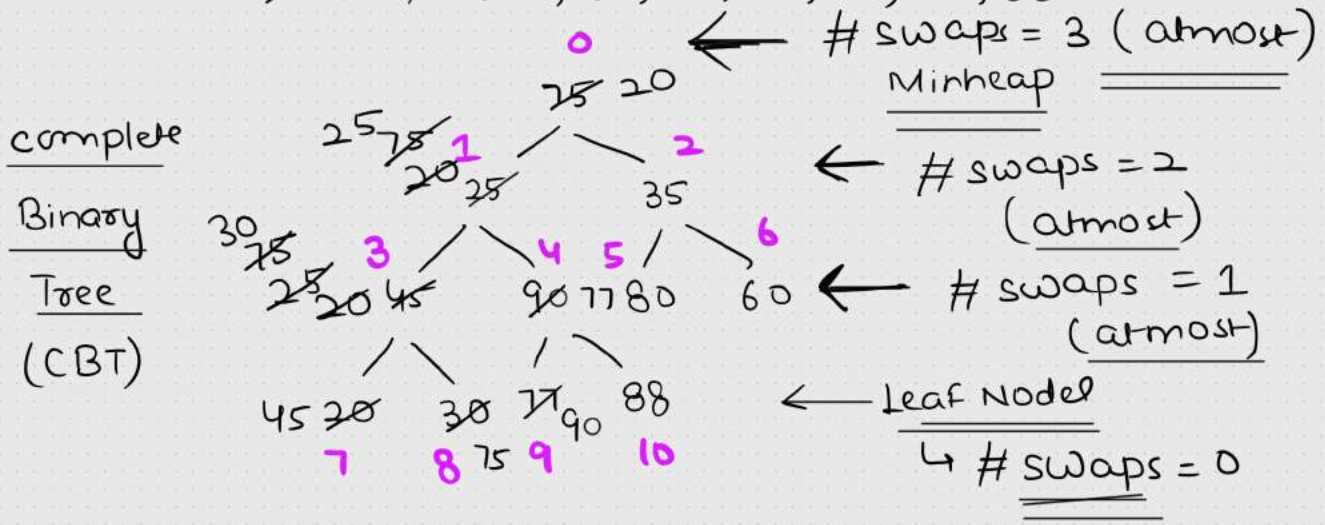
2) Delete all the elements step by step
   ↳ & store those deleted elements in any data structure
   ↳ O(n log n)

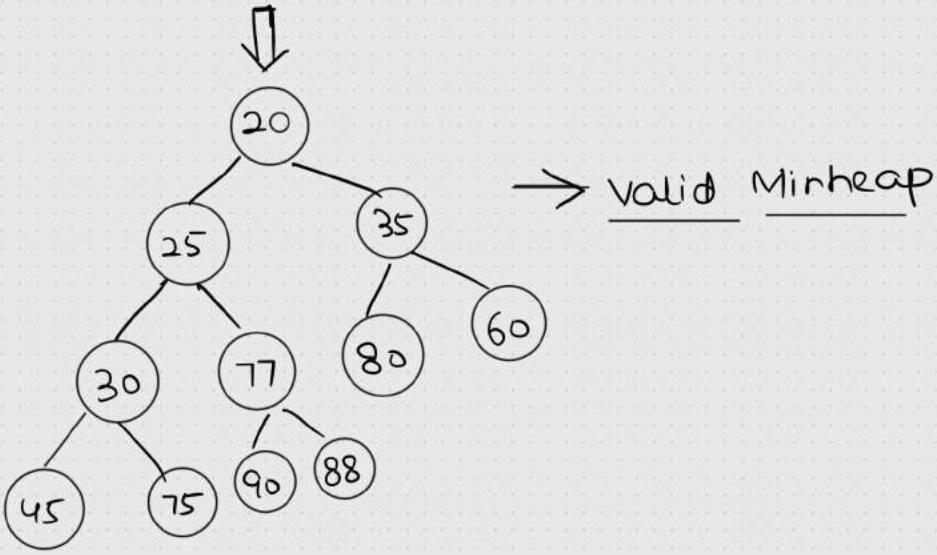
Time complexity = O(n) + O(n log n)
= O(n log n)

heapq → Python

Build Heap → O(n)

75, 25, 35, 45, 90, 80, 60, 20, 30, 77, 88



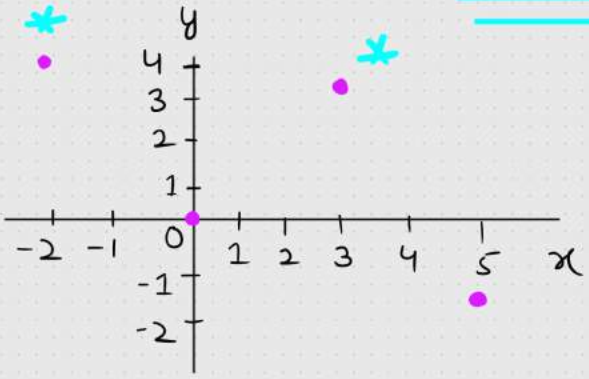


→ Valid Minheap

K closest Points

Points = [[3,3], [5,-1], [-2,4]] K = 2
(x y)

Output = [[3,3], (-2,4)] origin = (0,0)



Distance measure
⇓
Euclidean Distance

(0,0) → $P_1(x_1, y_1)$
(x_2, y_2) → P_2

Standard formula ⇒ $\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

$\Rightarrow \sqrt{x_2^2 + y_2^2}$

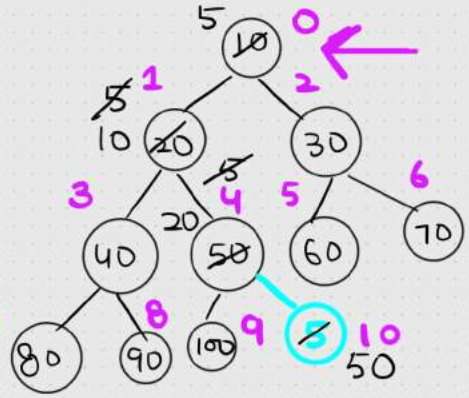
0 1

Approach ⇒
- Minheap → (Distance, Points)
  - ⇓ final output
- Deleting (Pop) ⇒ k times
  - ↳ Top k minimum distances
    - ⇓ Print points

Insertion in Minheap / Maxheap

$n = 10$

Complete Binary Tree



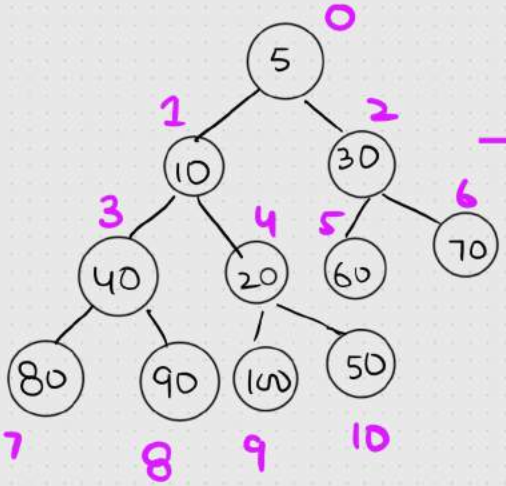
$x = 5$ → New data element

$50 < 5$ → Not true

$50 > 5$
$20 > 5$
$10 > 5$
→ 3 swaps

0	1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100	5

$\log N$



→ Valid minheap Tree

comparisons = $\log N$

swaps = $\log N$

0	1	2	3	4	5	6	7	8	9	10
5	10	30	40	20	60	70	80	90	100	50

$n = 2^k - 1$ → complete binary tree

$(n+1) = 2^k$

$k = O(\log n)$

$\log_2 (n+1) = \log_2 2^k$

$\log_2 (n+1) = k \log_2 2$

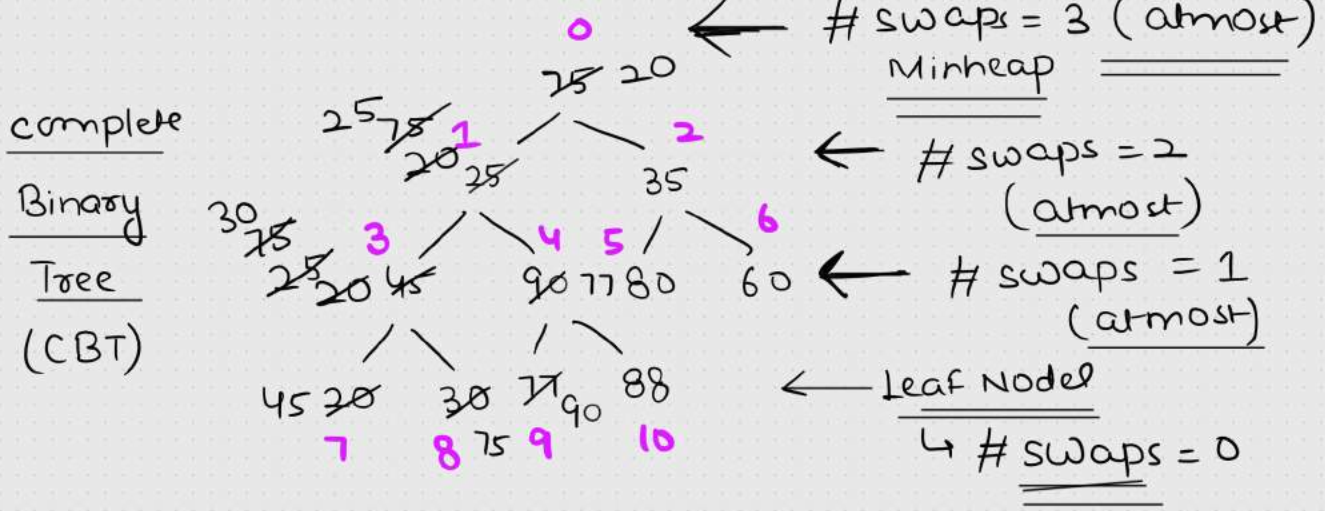
Time complexity = Number of comparisons + Number of swaps

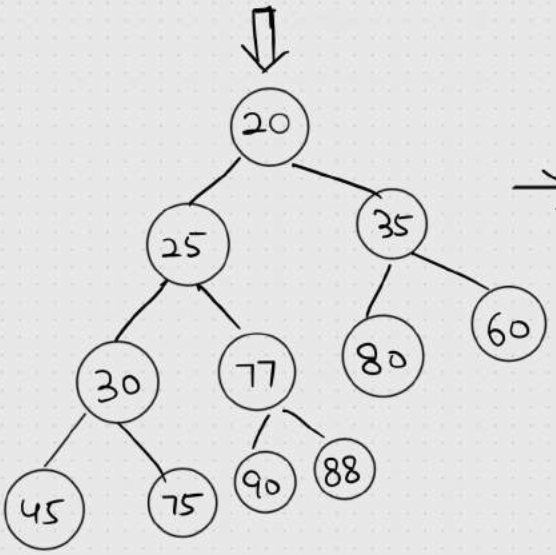
Insertion (Minheap / Maxheap)

$= O(\log N)$

Build Heap → O(n)

75, 25, 35, 45, 90, 80, 60, 20, 30, 77, 88

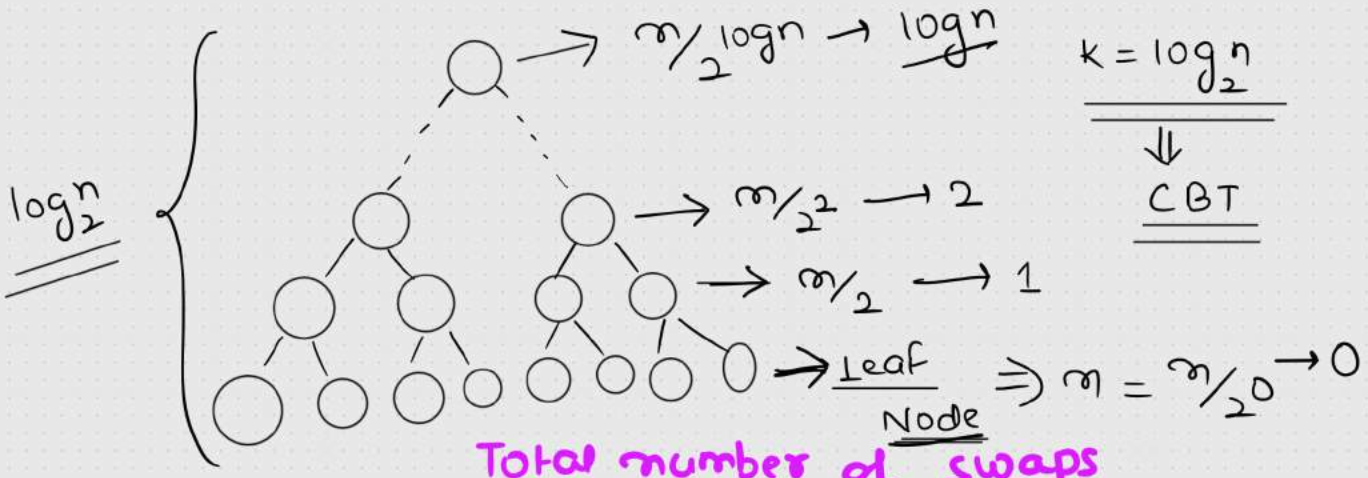




→ Valid Minheap

Heapify Method

Time complexity to build heap = O(n)



Total number of swaps

$$S = \frac{n}{2^0} * 0 + \frac{n}{2^1} * 1 + \frac{n}{2^2} * 2 + \dots + \frac{n}{2^{\log n}} * \log n$$

$$S = n\left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots + \frac{\log n}{2^{\log n}}\right) \quad \text{—①}$$

Divide ①/2 → GP series

$$\frac{S}{2} = n\left(\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \dots + \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n + 1}}\right) \quad \text{—②}$$

Subtraction of ② from ①

$$\frac{S}{2} = n\left(\left(\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n + 1}}\right)$$

↳ valid GP Series

$a = 1/2$ (first element)

$r = 1/2^2 \,/\, 1/2^1 = 1/2 < 1$ (common ratio)

(sum of GP series) $r < 1$

Sum of GP series $= \frac{a(1-r^n)}{1-r}$; $r < 1$

$$\frac{S}{2} = n\left(\left(\frac{\frac{1}{2}\left(1-\frac{1}{2^{\log n}}\right)}{1-\frac{1}{2}}\right) - \frac{\log n}{2^{\log n+1}}\right)$$

$$\frac{S}{2} = n\left(\left(\frac{2^{\log n}-1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n+1}}\right)$$

$2^{\log n+1} \Rightarrow 2^{\log n} \cdot 2$

$$2^{\log_2 n} = n^{\log_2 1} = n$$

$$\frac{S}{2} = n\left(\left(\frac{n-1}{n}\right) - \frac{\log n}{n*2}\right)$$

$$\frac{S}{2} = n\frac{(n-1)}{n} - n * \frac{\log n}{n*2}$$

$$\frac{S}{2} = n-1-\frac{\log n}{2}$$

$$S = 2n - 2 - \log n$$

$T(n) = O(n)$

Recursion

↳ function calls itself directly or indirectly

factorial

↳ $5! = 5 * 4 * 3 * 2 * 1 = 120$

Base case condition

↳ code will terminate

Small problem

$T(n)$ — `fact(n):`

$n = 0$ or $n = 1$

C — `if n == 0 or n == 1:`

↳ 1

`return 1`

Base case condition

`else:`

Recursion $n > 1$

`result = n * fact(n-1)` ↳ Recursive call

$n * T(n-1)$

`return result`

$n = 5$

`fact(5)` → 120

↳ $5 * fact(4)$ → 24

↳ $4 * fact(3)$ → 6

↳ $3 * fact(2)$ → 2

↓

$2 * fact(1)$

↓

1

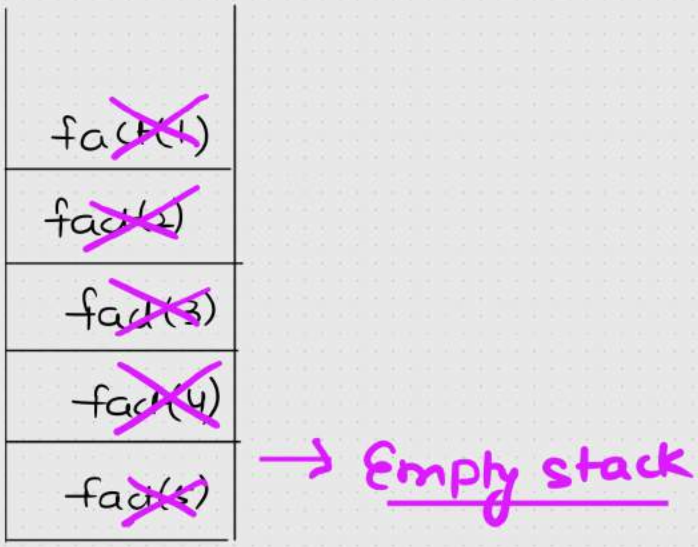
$1! = 1$

Code will stop

Stack → LIFO
(Last In first Out)

↓↓

to store function call



Recurrence Relation

(factorial of number)

$$T(n) = \begin{cases} 1 & n = 1 \\ n * T(n-1) & n > 1 \end{cases}$$

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

↳ Substitution method

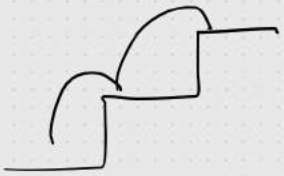
Count number of ways to reach upstairs

↳ 1 step at a time or 2 steps at a time

$n = 1 \longrightarrow$ 1 way

$n = 2 \longrightarrow$ 2 way

↳ (1,1), (2)



4

0 1 2 3 5 8 13 21 34

0 1 2 3 4 5 6 7 8

0 1 1 2 3 5 8

(8+1)

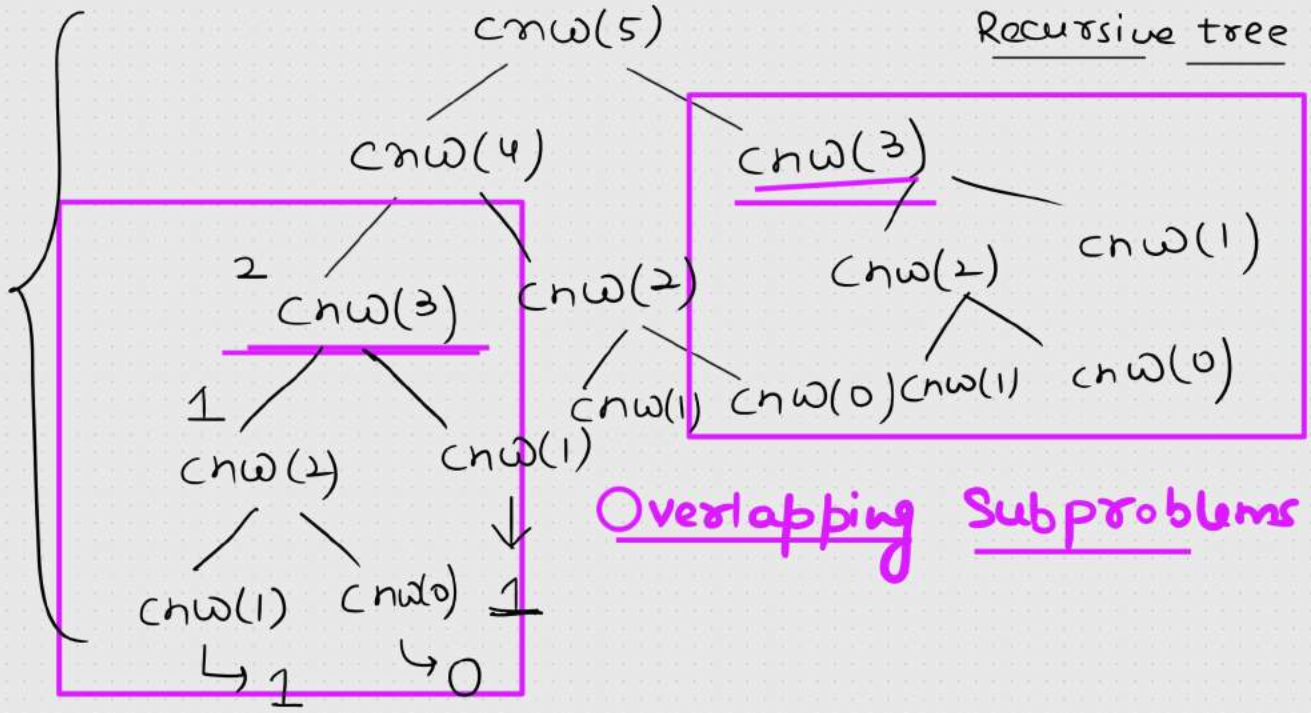
$n = 3 \longrightarrow$ (1,2), (1,1,1), (2,1) — 3 ways

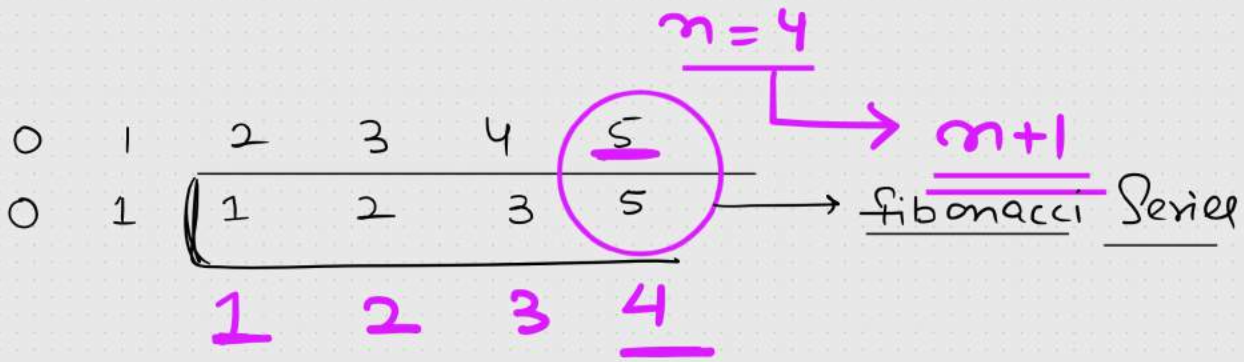
$n = 4$

(1,1,1,1)
(2,2)
(1,2,1)
(2,1,1)
(1,1,2)

5 ways

```
cnw(n)
{ if  n<=1
      ↳ return n
  else
      cnw(n-1) + cnw(n-2)
```





Time complexity

CBT → Complete Binary Tree

$2^n - 1$

$O(2^n)$ → Exponential Time complexity

$n \longrightarrow$ very very large

↳ Dynamic Programming

Divide & conquer

n is large ⟶ Big → Divide & conquer

n is small ⟶ small → return solution

↳ size of elements

Recursion → Calling the function itself directly or indirectly

↓

Recursive tree ⟶ Recurrence Relation

↳ 1) substitution
2) Recursive tree
3) Master's Theorem

← Time complexity

Psuedocode

arr → array, p → lower index (0), q → higher index (len(arr)-1)

T(n) ⟶ divideAndConquer(arr, p, q):

c ⟶ { if small(arr, p, q):
return solution

else:

Divide — ① ⌐

c ⟶ mid = Divide(arr, p, q) / $T(n/2)$

conquer — { b = divideAndConquer(arr, p, mid)
c = divideAndConquer(arr, mid+1, q)

↳ Recursion

conquer ② $T(n/2)$

return combine(b, c) ⟶ c

↳ combine

↳ ③

↳ optional

Recurrence Relation

$$T(n) = 2T(n/2) + c$$

$$T(n) = aT(n/b) + f(n)$$

Binary Search

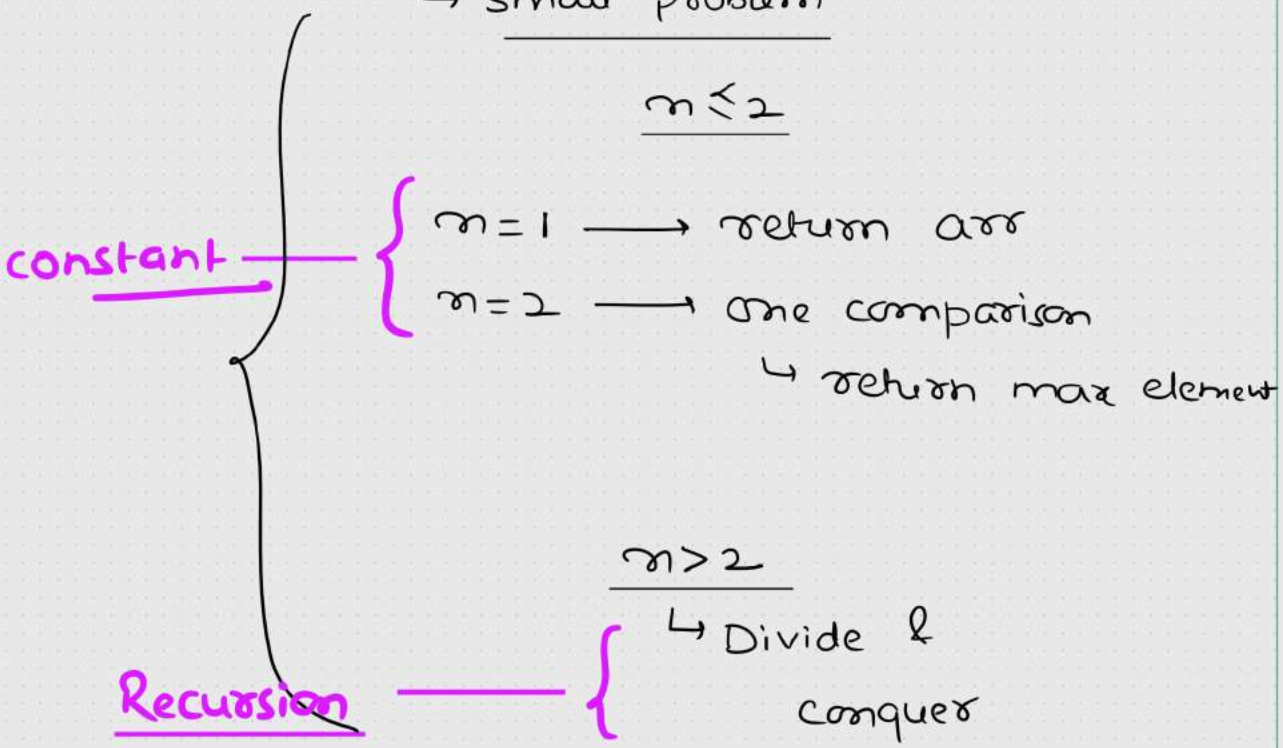
$$T(n) = T(n/2) + c$$

QuickSort

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

find max. element in an array

↳ small problem



Count of Number of Inversions

70, 50, 60, 10, 20, 30, 80, 15

0 1 2 3 4 5 6 7

↑ ↑ ↑ ↑

Inversion
↳ i < J → index
AND number
a(i) > a(J)

70 → 50, 60, 10, 20, 30, 15

50 → 10, 20, 30, 15

60 → 10, 20, 30, 15

10 → X

20 → 15

30 → 15

80 → 15

15 → X

Inversions = 17

1) Divide → mid = i + (J - i)/2 → O(1)

2) Conquer → $T(n/2) + T(n/2)$

3) combine → mergeProcedure → MergeSort
↳ n

Recurrence Relation

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

Using master's method / substitution method

$\Rightarrow O(n \log n)$

Recursive Tree

Divide & conquer approach

↳ Single element

↳ # inversion = 0

10 →

50 → 15, 20, 30

60 → 15, 20, 30

70 → 15, 20, 30

$\frac{0+7}{2} = 3$

20 → 15

30 → 15

C1

arr, 0, 7,
5+3+9 = 17

C2

arr, 0, 3
1+1+3 = 5

10, 50, 60, 70

50 → 10

70 → 10, 60

$\frac{0+3}{2} = 1$

C9

arr, 4, 7,
0+1+2 = 3

15, 20, 30, 80

$\frac{4+7}{2}$

15, 80 ⇒ 5

C3

arr, 0, 1,
0+0+1 = 1

50, 70

C6

arr, 2,
3,
0+0+1 = 1

10, 60

C10

arr, 4,
5,
0+0+0 = 0

20, 30

C13

arr, 6,
7, 0+0+1 = 1

C4

arr, 0,
0,
0

70

C5

arr, 1, 1
0

C7

arr, 2,
2 0

50

60

C8

arr, 3,
3 0

10

C11

arr, 4,
4 0

20

C12

arr, 5,
5 0

30

C15

arr, 7,
7 0

15

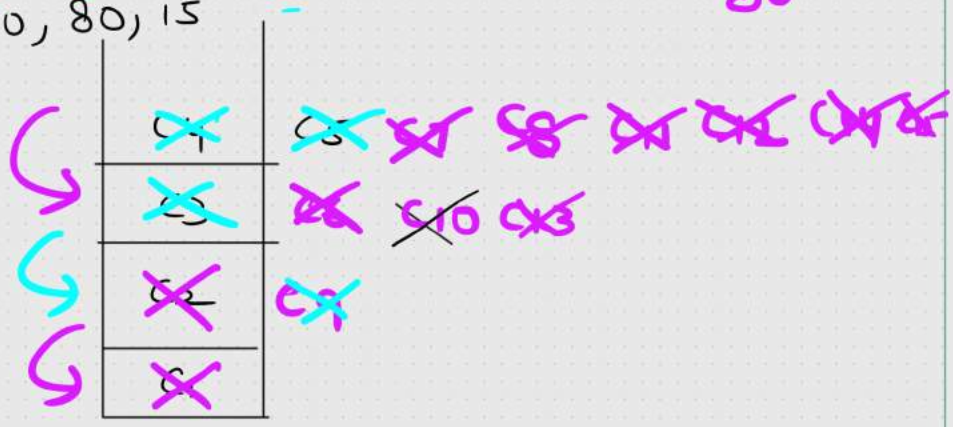
C14

arr, 6,
6 0

80

0 1 2 3 4 5 6 7

70, 50, 60, 10, 20, 30, 80, 15



MergeSort

90 - 95 %

↳ Implementation

5 % → Pending

Practice task :- Implementation of count of number of inversions

↳ github

↓↓

LinkedIn

↳ Tag "Priya Bhatia"

finding of maxima & minima in an array

arr = [70, 50, 45, 10, 12, 15, 75, 29, 37, 57]
0 1 2 3 4 5 6 7 8 9

Brute force approach

max = ~~0~~ 70

min = ~~50~~ ~~45~~ 10

{ max = 75
{ min = 10

Best case :- $(n-1) \cdot 1 = O(n)$

worst case :- $(n-1) \cdot 2 = O(n)$

average case :- $\frac{n-1}{2} \cdot 1 + \frac{n-1}{2} \cdot 2$

$\Rightarrow O(n)$

Divide & Conquer

small problem :- $n == 1$ or $n == 2$

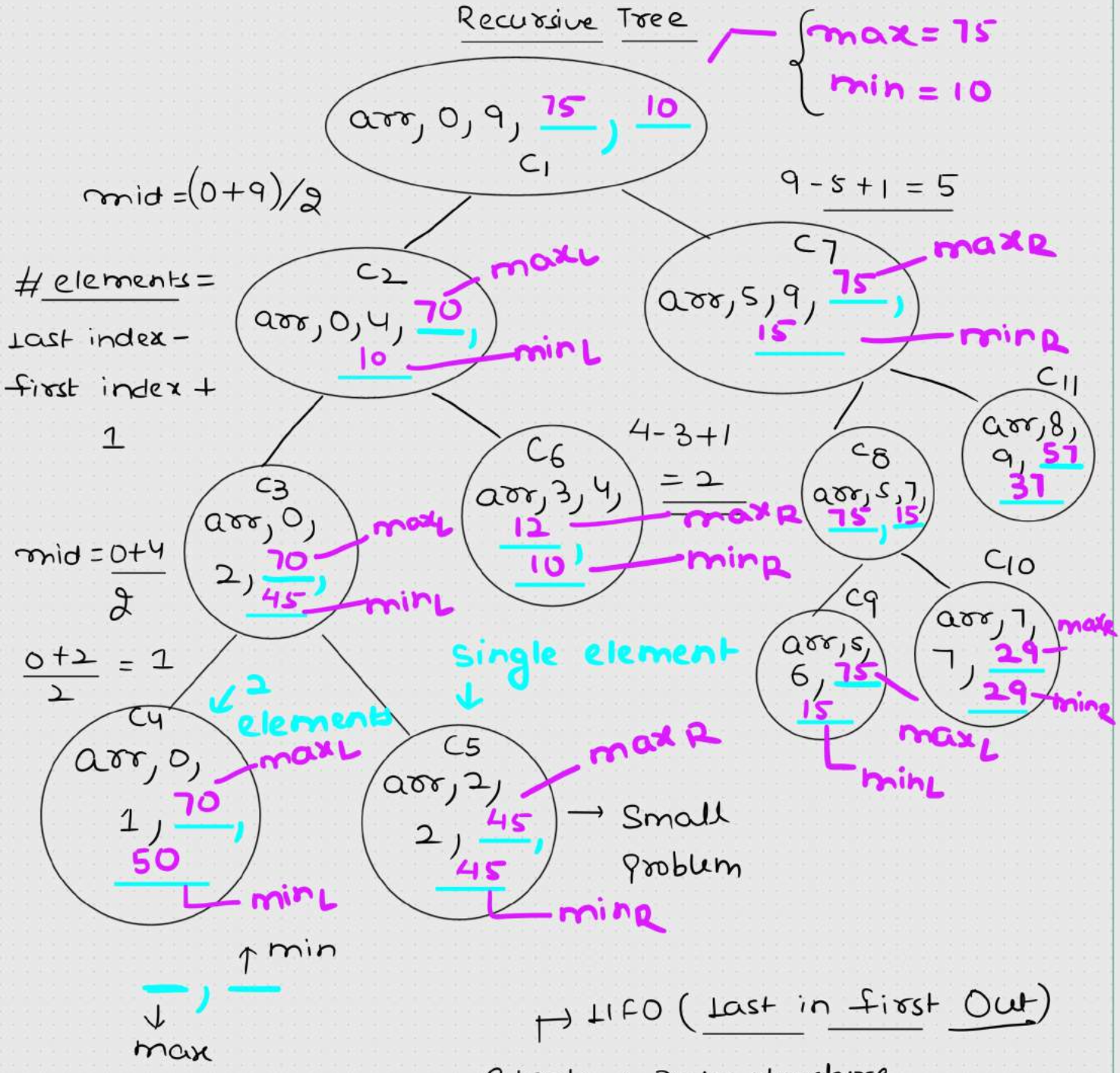
$n == 1$ ⇓ return arr { max, min

$n == 2$ ⇓ Single comparison

big problem :- $n > 2$

↳ Divide & conquer

Recursive Tree



↳ LIFO (Last in first Out)

Stack → Data structure

↳ store the function calls

Small problem

i == J

↳ single element in an array

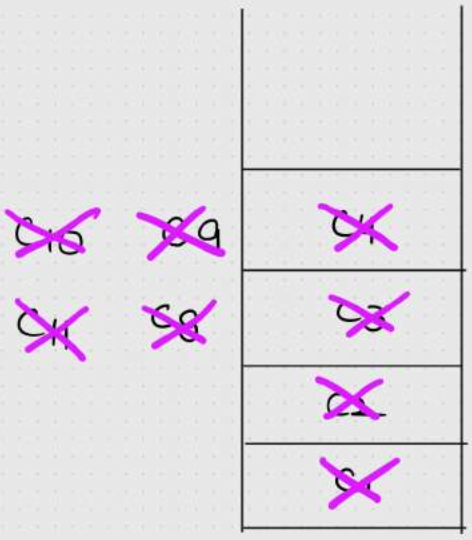
i → starting index

J → ending index

0 1-1

i == J-1

↳ two elements in an



4 levels = $\log_2 n = \log_2 10$

Stack space = $O(\log n)$ array

Psuedocode

i → starting index
J → ending index

T(n) ⇒ findminAndmax(arr, i, J):

Small problem ↓ c

if i == J: → Single element in an array
    min = arr(i)
    max = arr(i)

elif i == J-1: → two elements in an array
    if arr(i) < arr(J):
        min = arr(i)
        max = arr(J)
    else:
        min = arr(J)
        max = arr(i)

else: ⌐ Divide — c
    mid = i + (J-i)/2

conquer
    max_L, min_L = findmaxAndmin(arr, i, mid) — T(n/2)
    max_R, min_R = findmaxAndmin(arr, mid+1, J) — T(n/2)

combine → c
    if $max_L < max_R$:
        max = max_R
    else:
        max = max_L

```
if min_L < min_R :
    min = min_L
else :
    min = min_R
```

return (max, min)

Recurrence Relation

$$T(n) = 2T(n/2) + c$$

⟶ Substitution Method

↳ O(n)

Master's Theorem

$a = 2$ $\quad k = 0$

$b = 2$ $\quad p = 0$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a > k$$

↳ $\Theta(n^{\log_b a})$

$\Rightarrow \Theta(n^1)$

$\Rightarrow \Theta(n)$

finding of Power of an Element

$a \to 2$

$n \to 10$

↳ $n \geq 1$ ←

$n < 1$ ←

Output = $a^n = 2^{10} = 1024$

$\begin{cases} 2^1 = 2 \\ 4^1 = 4 \end{cases}$ $5^1 = 5$

Divide & conquer

Small problem ⟶ $n = 1$

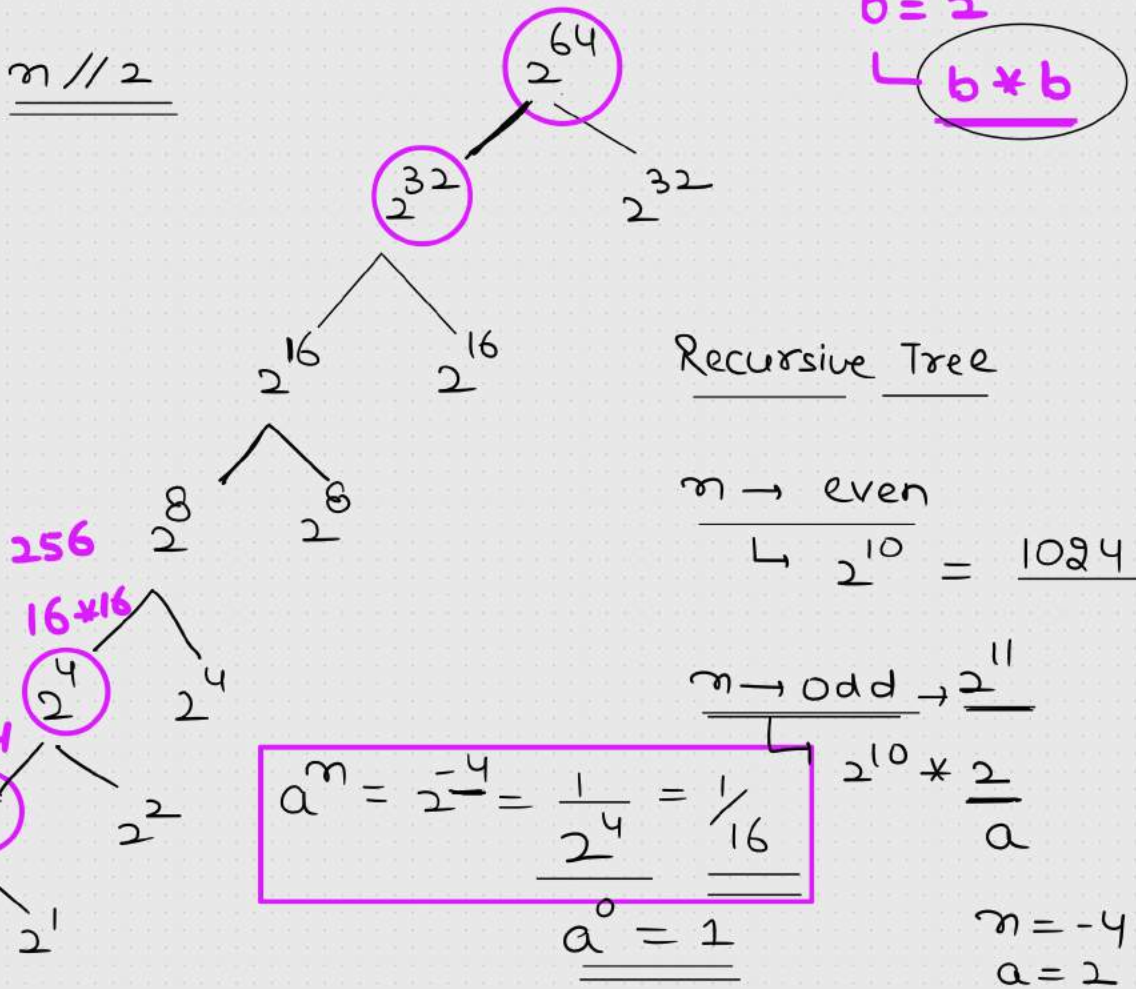
↳ return solution ↳ return a

big Problem ⟶ $n > 1$

↳ Divide & conquer $a^n = a^{n/2} * a^{n/2}$

$n = 64$ $a = 2$

$n // 2$



Recursive Tree

$n \to$ even

↳ $2^{10} = 1024$

$n \to$ odd $\to 2^{11}$

$2^{10} * 2$

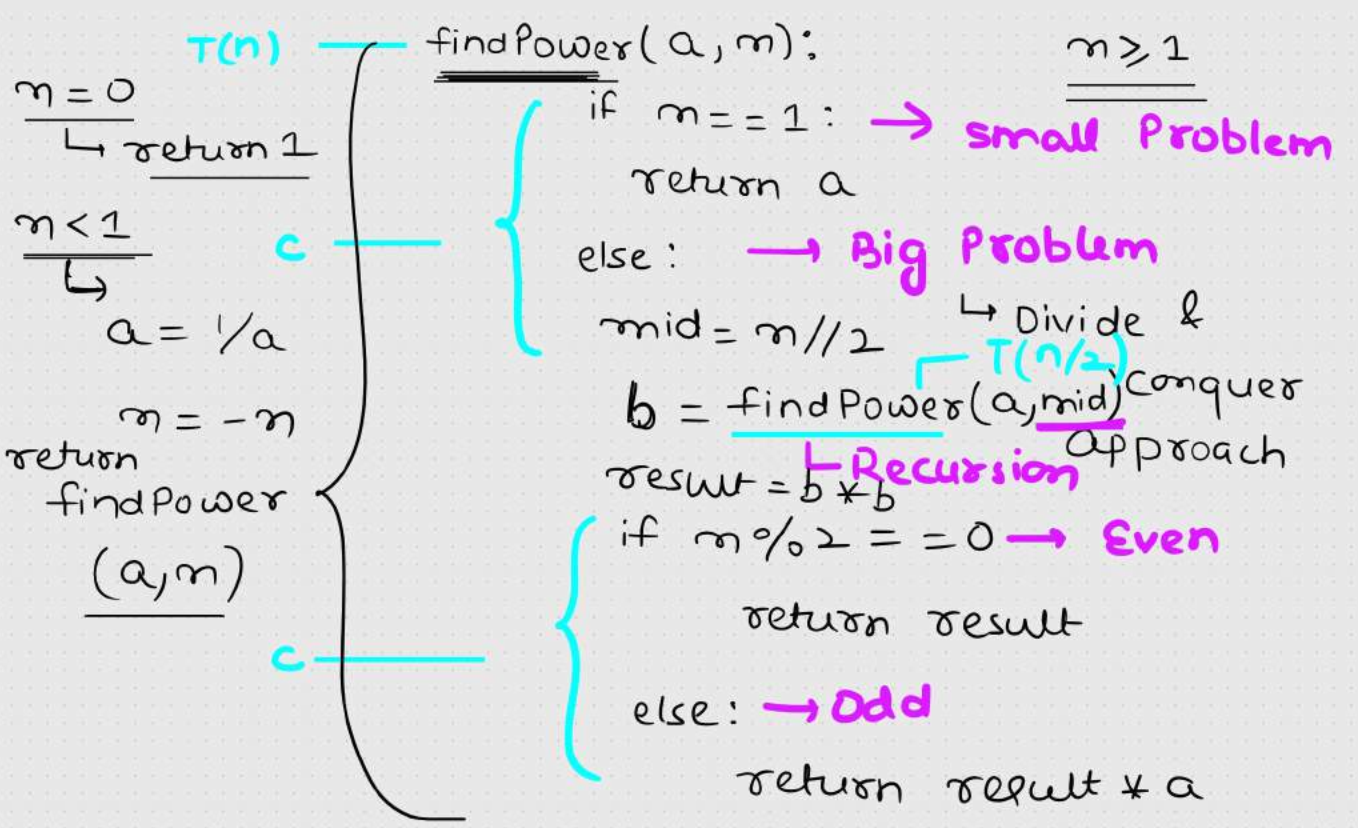
a

$a^n = 2^{-4} = \frac{1}{2^4} = \frac{1}{16}$

$a^0 = 1$

$n = -4$

$a = 2$



Recurrence Relation

↳ $T(n) = T(n/2) + c$

↳ Master's Theorem or Substitution Method

Master's Theorem

$a = 1 \qquad k = 0$

$b = 2 \qquad p = 0$

$$\log_b^a = \log_2^1 = 0$$

$\log_b^a = k = 0$ — case 2

$p > -1 \Rightarrow p = 0$

$\Theta(\log n)$

Substitution Method → Recursive Term

$$T(n) = T(n/2) + c$$

$$= T\left(n/2^2\right) + c + c$$

$$= T\left(\frac{n}{2^3}\right) + c + c + c$$

$$\frac{n}{2^k} = 1 \qquad \Big\downarrow \text{ k times}$$

$$k = \log_2 n$$

$$= T\left(\frac{n}{2^k}\right) + k \cdot c$$

$$= T\left(\frac{n}{2^{\log_2 n}}\right) + k \cdot c$$

$$= T\left(\frac{n}{n^{\log_2 2}}\right) + k \cdot c$$

$$= T(1) + k \cdot c$$

$$= 1 + \log_2 n \cdot c$$

$$\Rightarrow O(\log_2 n)$$

Mergesort
↳ Merge Procedure
↳ combining task

[50, 70, 45, 5, 79, 47, 29, 52]
0 1 2 3 4 5 6 7

$n = 8$

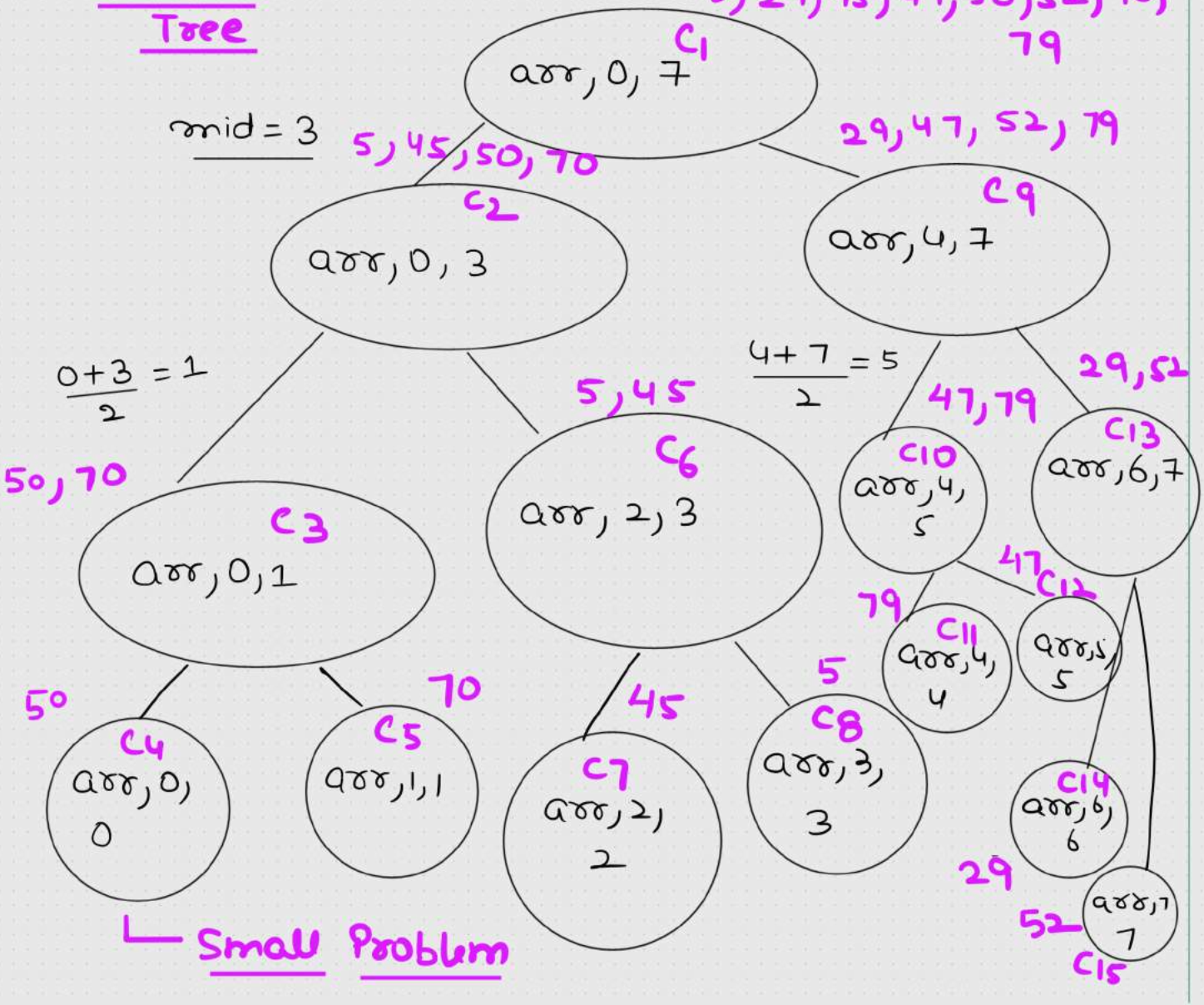
Small Problem
↳ single element
↳ sorted array

Big Problem $n > 1$

Sorted array ↑↑

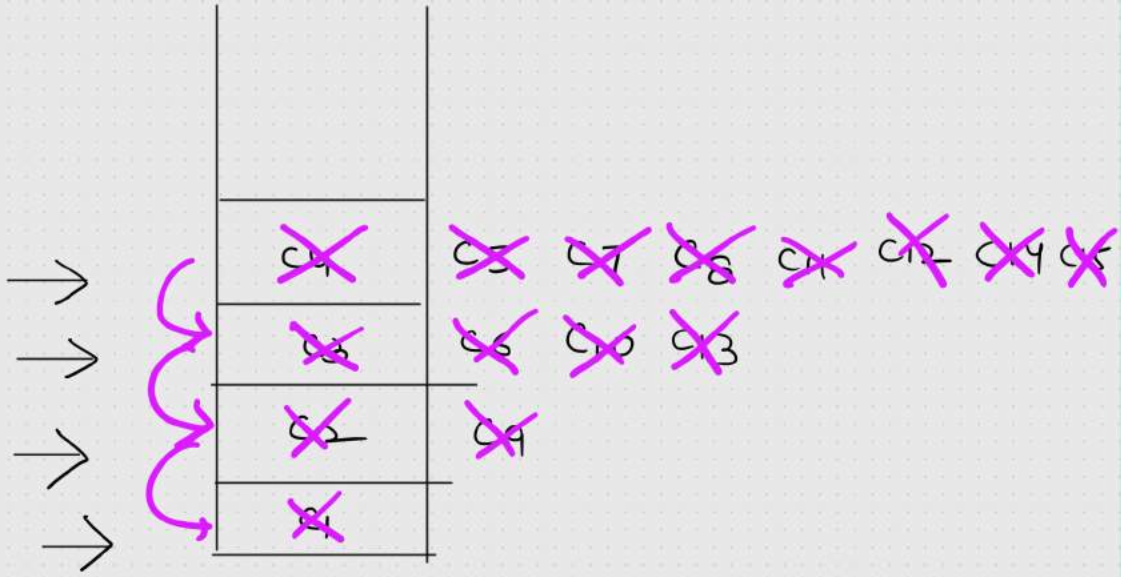
5, 29, 45, 47, 50, 52, 70, 79

Recursive Tree



$i == j \Rightarrow$ array contains only single element

Stack data structure $= O(\log_2 n)$



Top to Down ⟶ Divide & conquer

↳ Small problem

Down to Top ⟶ combine

↳ Merge Procedure

↓↓

fully sorted

subarray

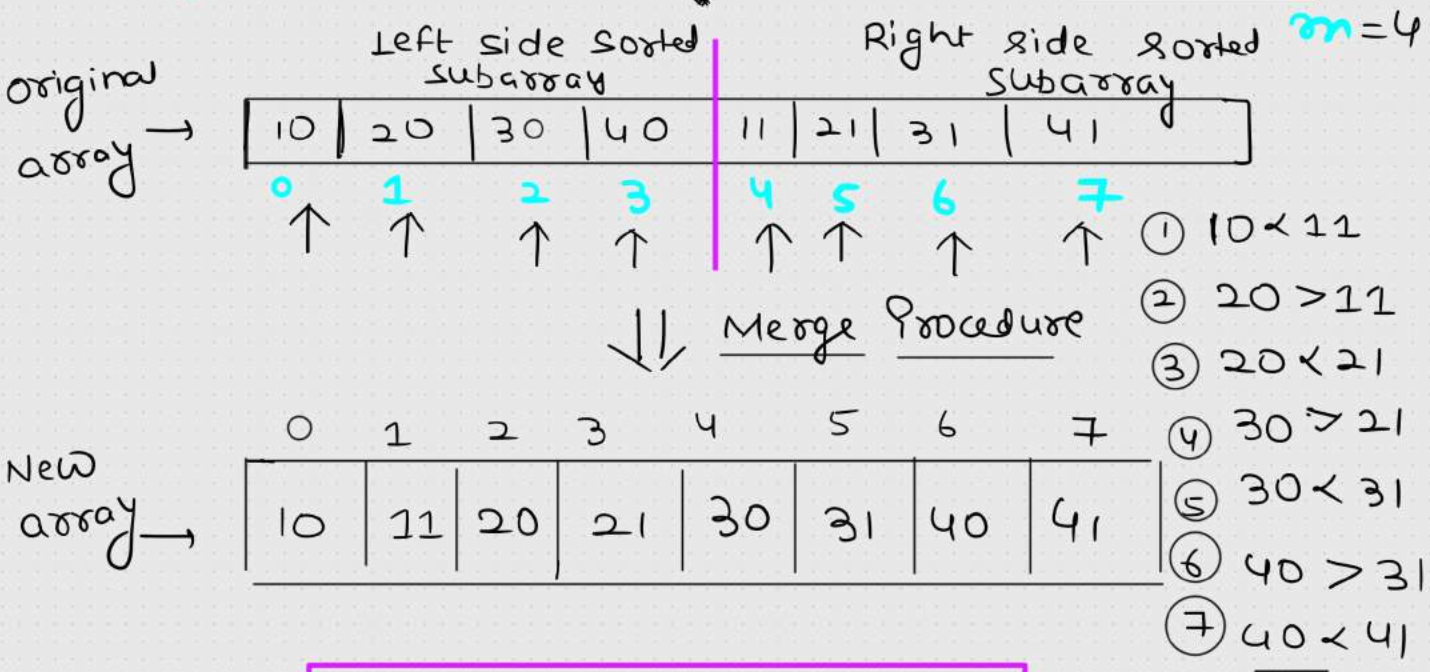
Worst case Scenario

Time complexity of merge procedure

↳ Number of moves

Number of comparisons

$n = 4$ $m = 4$

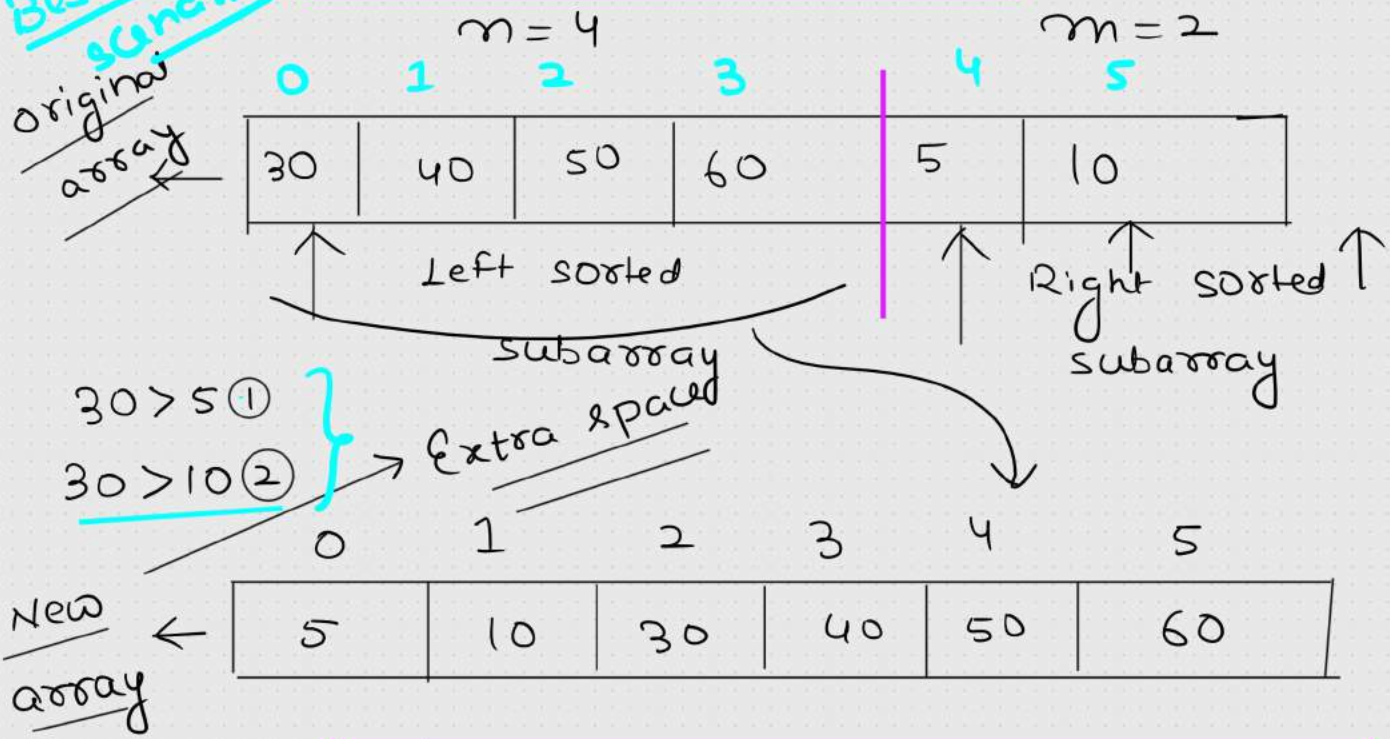


① $10 < 11$
② $20 > 11$
③ $20 < 21$
④ $30 > 21$
⑤ $30 < 31$
⑥ $40 > 31$
⑦ $40 < 41$

Worst case scenario

↳ $n + m - 1$

Best case scenario



Best case scenario

Comparisons = $\min(n, m)$

$= \min(2, 4) = 2$

moves = $m+n$ $\qquad$ $m+n = N$

Time complexity → $O(m+n)$

(Merge Procedure) → $O(N)$

⇓

moves + # comparisons

$(m+n) + (m+n-1)$

$\Rightarrow O(m+n)$

Space complexity → $O(N)$

⇓

extra space (new array)

⇓

Outplace sorting algorithm

$T(n) \Rightarrow$ mergeSort(arr, i, j): ↑ c

Pseudocode

if i == j:

return arr } small Problem

else:

c — Divide → mid = i + (j-i)/2

$2T(N/2)$ — conquer → $T(N/2)$ mergeSort(arr, i, mid)

$T(N/2)$ mergeSort(arr, mid+1, j)

$O(N)$ — Combine → mergeProcedure(arr, i, mid, mid+1, j)

return arr

Recurrence Relation

$T(n) = 2T(N/2) + N$

- Master's Theorem
- Substitution Method

$a = 2$ $k = 1$

$b = 2$ $p = 0$

Masters Theorem

$\log_b^a = \log_2^2 = 1$

$\log_b^a = k$

(1) (1)

$\Rightarrow \Theta(N \log N)$

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

$$= 2\left(2T(n/2^2) + n/2\right) + n$$

Substitution Method

$$\Rightarrow 2^2 T(n/2^2) + 2n$$

$$\Rightarrow 2^2\left(2T(n/2^3) + \frac{n}{2^2}\right) + 2n$$

$$\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$\frac{n}{2^k} = 1 \quad \Big\downarrow k$$

$$k = \log_2 n$$

$$2^k T\left(\frac{n}{2^k}\right) + k \cdot n$$

$$2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot n$$

$$n^{\log_2 2} T\left(\frac{n}{n^{\log_2 2}}\right) + n \log_2 n$$

$$n + n \log_2 n$$

$\Rightarrow O(m \log n)$

QuickSort Algorithm

↳ Dividing → mid = i + (j - i) // 2

→ Other application

QuickSort ↓

Partition Algorithm

* Partition Algorithm

	0	1	2	3	4	5	6	7	8
	34			47	~~34~~ 70	99		85	
Pivot ←	(70),	50,	27,	~~99~~,	~~85~~,	~~47~~,	123,	~~34~~,	147
Element	p	j →	j →	j →	j →	j →	j →	j →	q j
		i	i	i	i	i			

j → > pivot → moves on

i → < pivot i = p

```
for(j = i+1; j < n; j++)
    if arr[j] <= pivot :
        i = i+1
        swap(arr(i), arr(j))
swap(arr(i), arr(p))
```

smaller than Pivot Element ↑ | ↓ pivot | ↑ larger than pivot element

0	1	2	3	4	5	6	7	8
34,	50,	27,	47,	(70),	99,	123,	85,	147

n = 4

QuickSort(0, 3) — QS (p, n-1)

QuickSort(5, 8) — QS(n+1, q)

0	1	2	3	4	5	6	7	8	
27,	(34),	47,	50,	(70),	85,	99,	123,	147	→ final sorted array

Psuedocode of Partition Algo

```
Partition(arr, p, q):
    i = p
    pivot = arr(p)
    for j = i+1 to n-1:
        if arr(j) <= pivot:
            i = i+1
            swap(arr(i), arr(j))
    swap(arr(i), arr(p))
    return i
```

$O(n)$

$j \rightarrow >$ pivot
↳ move on
$i \rightarrow <$ pivot

swap(arr(i), arr(p)) → Pivot

Psuedocode QuickSort Algorithm

```
QuickSort(arr, p, q):
    if p < q:
        m = Partition(arr, p, q)
        QuickSort(arr, p, m-1)
        QuickSort(arr, m+1, q)
```

Divide

Left side Recursion ← QuickSort(arr, p, m-1) — T(m-p)

Right side Recursion ← QuickSort(arr, m+1, q) — T(q-m)

Conquer

* No need of combine part in QuickSort Algo

Recursive Tree

27, 34, 47, 50

QuickSort(arr, 0, 8) mid = 4

QuickSort(arr, 0, 3)

QuickSort(arr, 5, 8)

27 34 47 50
0 1 2 3

~~34~~, ~~50~~, ~~27~~, 47

P J → J → J

i i

m = 1

0 1 2 3

(27) (34) (50, 47)

i

QuickSort(arr, 0, 0)

27

27

(47, 50)

QuickSort(2, 3)

2 3
~~47~~ 50

~~50~~, ~~47~~

m = 3

P Q

i J

i

QuickSort(2, 2)

47

QuickSort (3, 3)

50

(27), 19, 49, 57, 29, 46, 37

0 1 2 3 4 5 6

19, 27, 49, 57, 29, 46, 37

i = 1 → m = 1

Recurrence Relation

QuickSort

$$T(n) = \begin{cases} 1 & n = 1 \\ T(m-p) + T(q-m) + n & n > 1 \end{cases}$$

Best case scenario

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

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

Master's Theorem or Substitution Method;

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

Worst case scenario

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

$$T(n) = O(n^2)$$

array is highly unsorted

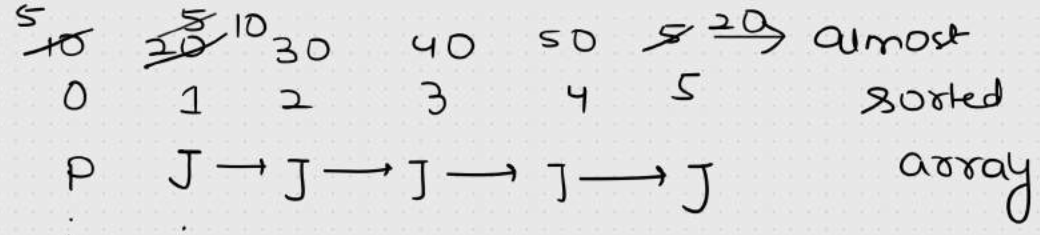
⇑

QuickSort

- Best/average case scenario → $O(n \log n)$
- Worst case scenario → $O(n^2)$

Insertion Sort → $O(n)$

array is almost/fully sorted



↳ worst case scenario

⇓

QuickSort

⇓

$O(n^2)$

Note:

↳ Inplace sorting algorithm (not using any extra space)

↳ Not a stable algorithm (QuickSort, HeapSort)

10^1 10^2 10^3 10^4 10^5

sorting

10^1 10^2 10^3 10^4 10^5

↳ sequence is not retained

↳ Real time scenario

(Actually the array is unsorted)

Randomized QuickSort

↳ Pivot Element

↳ Randomly

randomQuickSort(arr, p, q): ⟹ Worst case scenario

```
random_pivot = random(arr, p, q)
swap(arr[random-pivot], arr(p))
return partition(arr, p, q)
```

⇓ $O(n^2)$ ⇓ Less

QuickSort	Randomized QuickSort
Best case - O(nlogn)	Best case - O(nlogn)
Average - O(nlogn) case	Average case - O(nlogn)
Worst case - $O(n^2)$	Worst case - $O(n^2)$ ⇓ chances is quite less

Selection Procedure

↳ Kth smallest element in an array

0	1	2	3	4	5	6	7
40,	20,	70,	50,	39,	47,	65,	75

K = 2

Output = 39

Sorted array

p, m-2 → 1

m

k = 6 → (m, q) → q

m > k → ↳ left

1	2	3	4	5	6	7	8
20,	39,	40,	47,	50,	65,	70,	75
0	1	2	3	4	5	6	7

40 → Pivot

↳ Partition algo

↳ 2 → k → 3rd smallest element in an array

m = i+1 → ↳ Position

4 → 3

Position = index + 1

selectionProcedure(arr, p, q, k):

if p < q: ⇑ n

m = Partition(arr, p, q)

↳ Position (i+1)

Pseudocode of Selection Procedure

if m == k:

return arr(m-1)

elif m < k:

return SelectionProcedure(arr, m, q, k)

Right side T(q-m)

else:

return selectionProcedure(arr, p, m-2, k)

left side T(m-p)

Recurrence Relation

$T(n) = n + T(m-p)$

OR

$n + T(q-m)$

Best case scenario

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

$= O(n)$

Worst case scenario

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

$= O(n^2)$

$a = 1 \quad b = 2$

$\log_2^1 = 0, \; k = 1$

$\log_b^a < k$

Strassen's Matrix Multiplication

Matrix Multiplication → $O(n^3)$

↳ Three for loops

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\times5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix}$$

⇓

Divide & conquer approach

⇓

Recursion

⇓

Recurrence Relation

⇓

$$T(n) = 7\,T\left(\frac{n}{2}\right) + n^2$$

$$T(n) = O\left(n^{\log_2 7}\right)$$

$$T(n) = O\left(n^{2.81\cdots}\right)$$

Practice Question

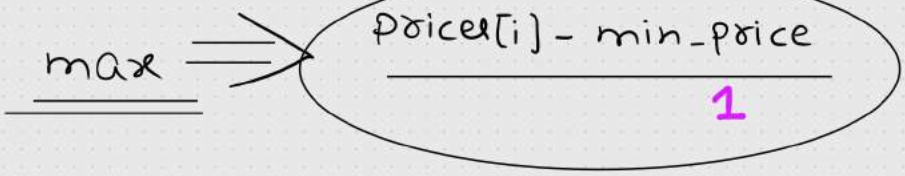
Strassen's Matrix Multiplication

Best Time to Buy & Sell Stocks

↳ prices = [7, 1, 5, 3, 6, 4, 15]

1 2 3 4 5 6 7

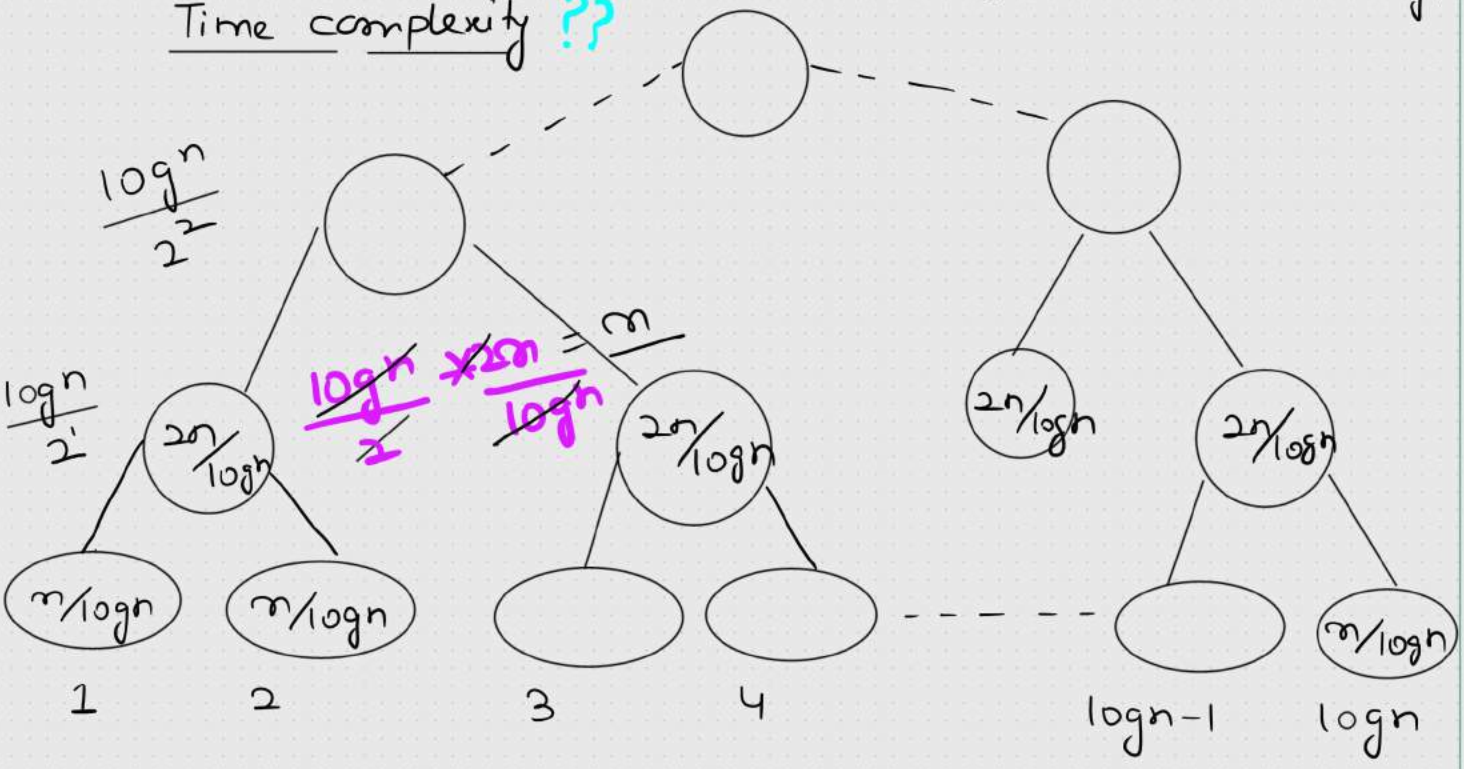
output → max_profit = 15 - 1 = 14



Problems Related Merge Procedure

1) logn sorted subarrays & each of subarray is of size $n/\log n$

Time complexity ??



$$\frac{n}{\log n} * \log n = n$$

$$\frac{\log n}{2^k} = 1$$

$$\log n = 2^k$$

$$\boxed{\log_2(\log n) = k}$$

Time complexity $\Rightarrow O(n \cdot k)$

$\Rightarrow O(n \log(\log n))$

10 ice-creams

1 ——— 2
2 ——— 2
3 ——— 2
4 ——— 2
5 ——— 2

$2+2+2+2+2=10$

$2*5=10$

1 ——— 1
2 ——— 3
3 ——— 2
4 ——— 7
5 ——— 8

$1+3+2+7+8$

2) nk sorted subarray & each of size n/k

Time complexity → $O(n^2 \cdot n)$

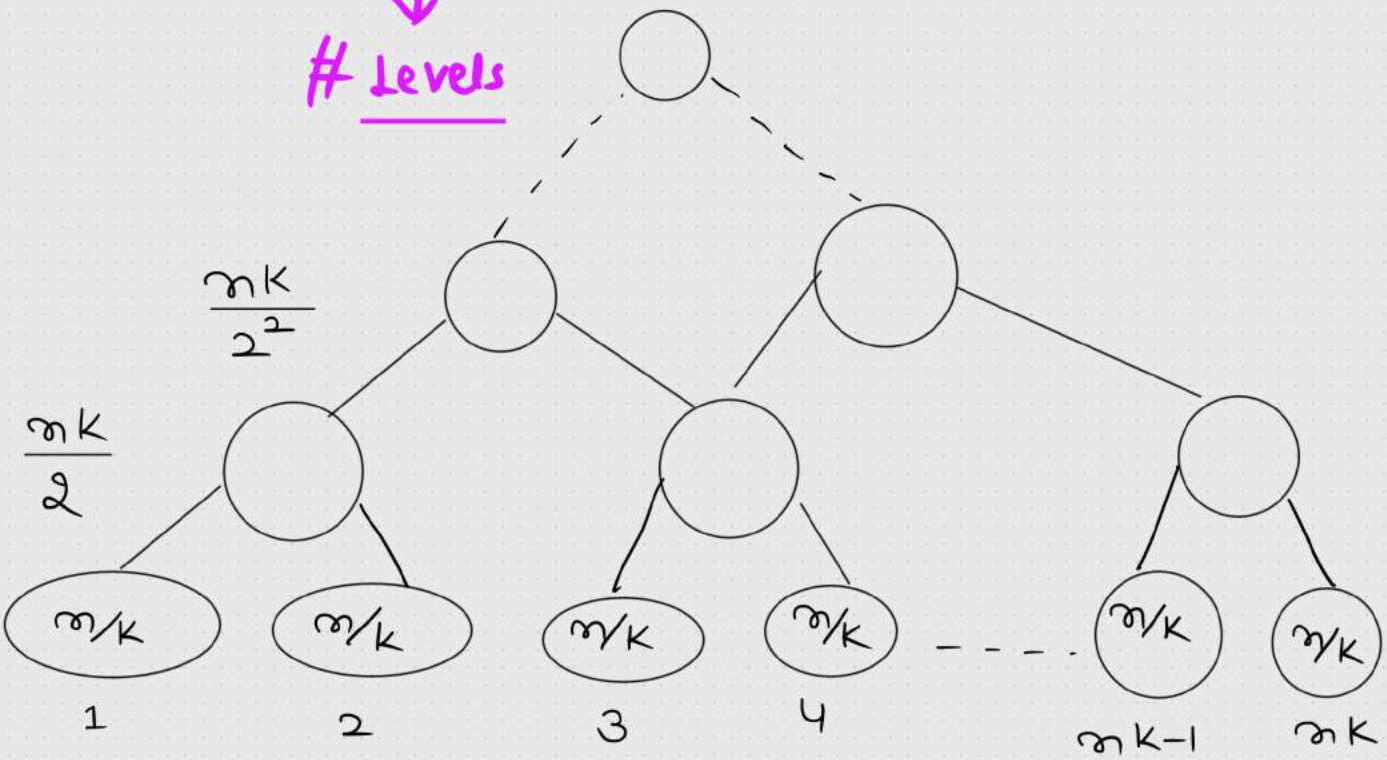
$\frac{nk}{2^n} = 1$ $\quad O(n^2 \log_2(nk))$

$nk = 2^n$

$n = \log_2(nk)$

↓

Levels



$n/k * nk = n^2$

Merge Procedure → $O(n)$

Problems on QuickSort

1) Sorting of n numbers → n/7 th smallest element is selected as pivot using $O(n^2)$ time complexity

⇓

worst case time complexity of quicksort

0	1	2	3	4	5	6
70,	50,	45,	79,	82,	65,	95
1	2	3	4	5	6	7

→ 3rd smallest element as a pivot

n/7 th smallest element ⇓ 3rd Position after applying Partition algo

3

50, 65, 70, ... n/7 Position

Pivot Element

n/7 - 1 | n/7 | n − n/7 chosen pivot

Recurrence Relation

$$T(n) = T\left(\frac{n}{7} - 1\right) + T\left(n - \frac{n}{7}\right) + n^2$$

Left subarray ⇓ n ⇓ Partition ⇓ Right subarray

$$T(n) \Rightarrow T\left(\frac{n}{7}\right) + T\left(\frac{6n}{7}\right) + n^2 \Rightarrow$$ Recursive Tree Approach

$$\Rightarrow O(n^2)$$

2) Sorting of n numbers → $n/10^{th}$ element selected as pivot using $O(\log n)$. (Not mentioning about smallest or largest)

⇓

worst case time complexity of quickSort

$$T(n) = T(n-1) + \log n + n$$

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

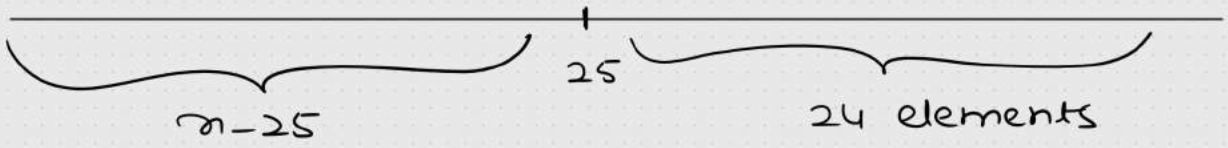
$$\boxed{T(n) = O(n^2)}$$

3) Sorting of n numbers $\Rightarrow$ 25th largest element is selected as pivot using $O(n^2)$

$\Downarrow$

worst case time

complexity of QuickSort



$$T(n) \Rightarrow T(n-25) + T(24) + n^2 + n$$

$$T(n) \Rightarrow T(n-25) + n^2$$

$$T(n) \Rightarrow O(n^3)$$

Linked List

↳ Linear Data Structure

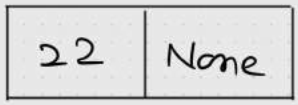
⇓

Not in a contiguous manner

Node — Data

Pointer → store the address of next node.

100

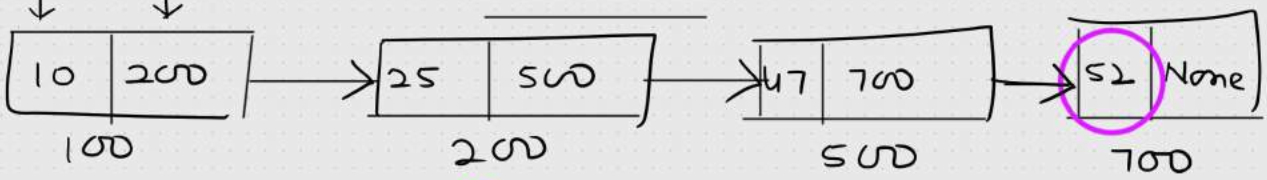


↑ Data ↑ Pointer

No concept of Random access

Linked List → more space

Data next



→ Linear Search O(n)

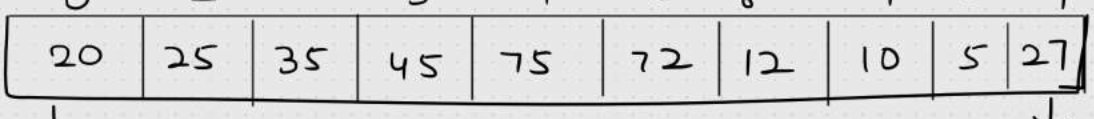
head → store the address of the first node

{ Random access is not allowed

arr[8]

10 elements → Array

4 Byte



1000 1036

1000 + (9 − 0) * 4

1036

Record of Students data

⇓

insertion & deletion (frequent)

→ Linked List

searching (Binary search)

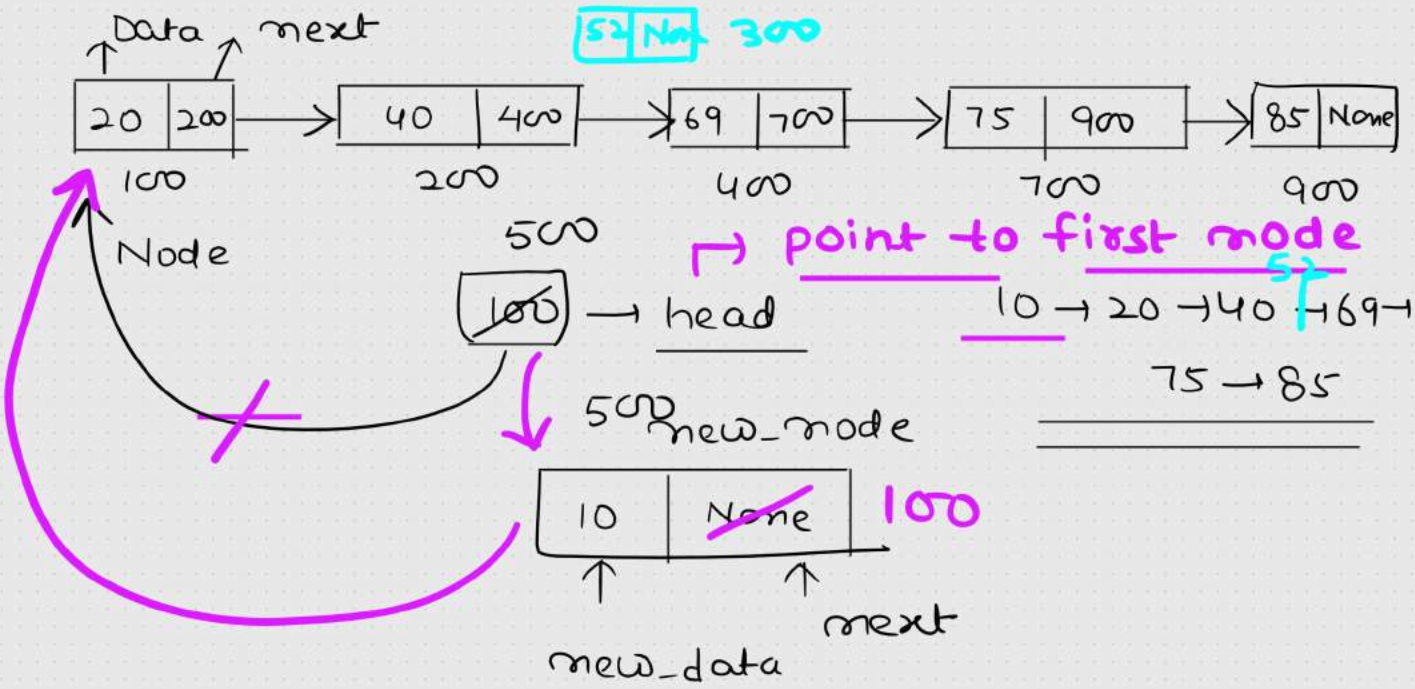
↳ O(log n)

→ Array

Random access

Scenario, we can chose what data structure to use

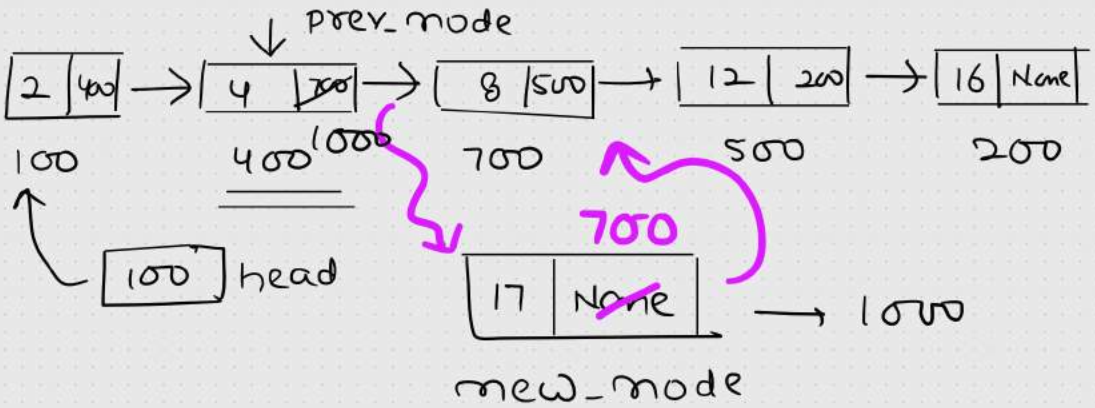
Insertion of node → front




```
new_node.next = self.head
self.head = new_node
```

O(1) → constant time complexity

Insertion after any node

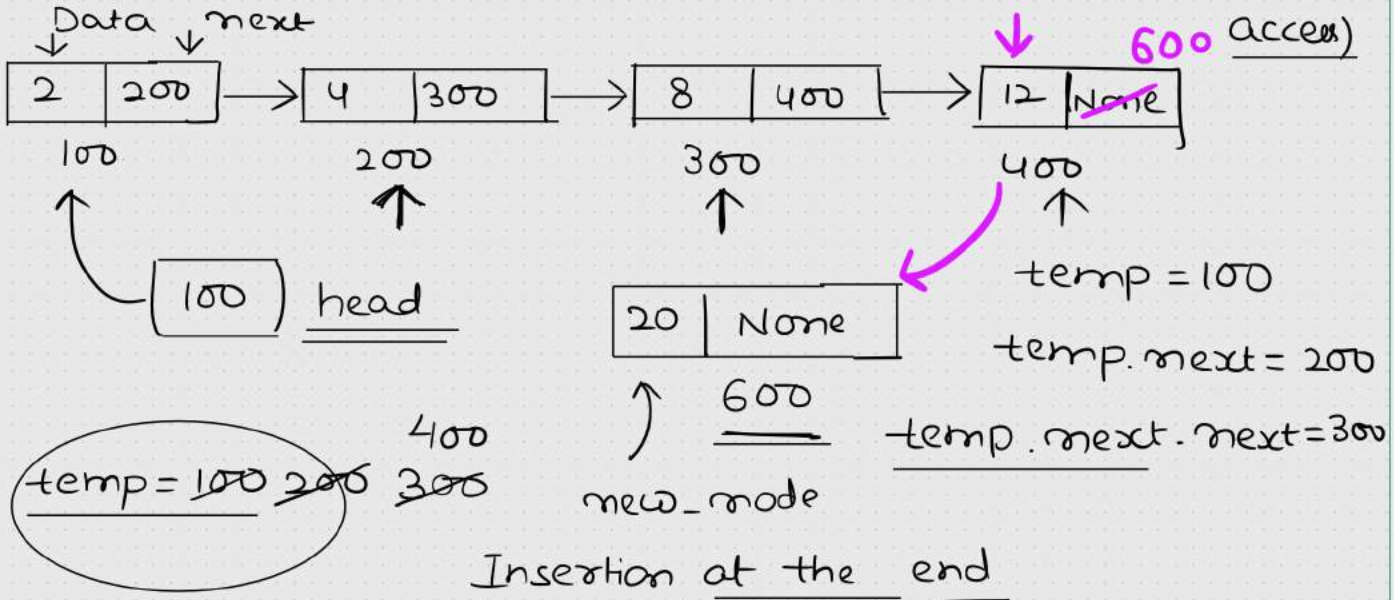


new_node.next = prev_node.next

prev_node.next = new_node

2 → 4 → 17 → 8 → 12 → 16

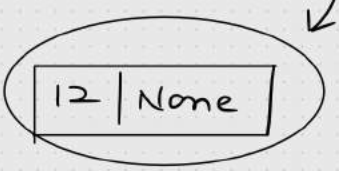
Insertion (End) Linked List (No Random Access)



Insertion at the end

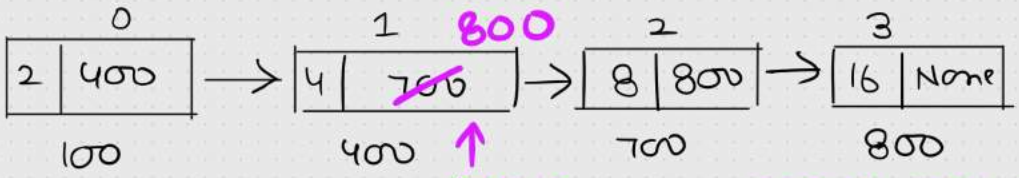
```
temp = self.head
while temp.next :
    temp = temp.next
temp.next = new_node
```

first insertion




```
self.head = new_node
return
```

Deletion in Linked List



temp

pos = 2

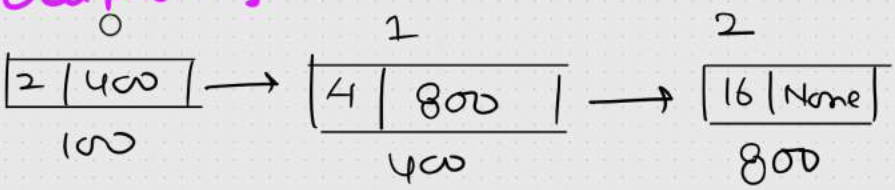
next_ptr = 800

next_ptr = temp.next.next

temp.next = None

temp.next = next_ptr

Expected output



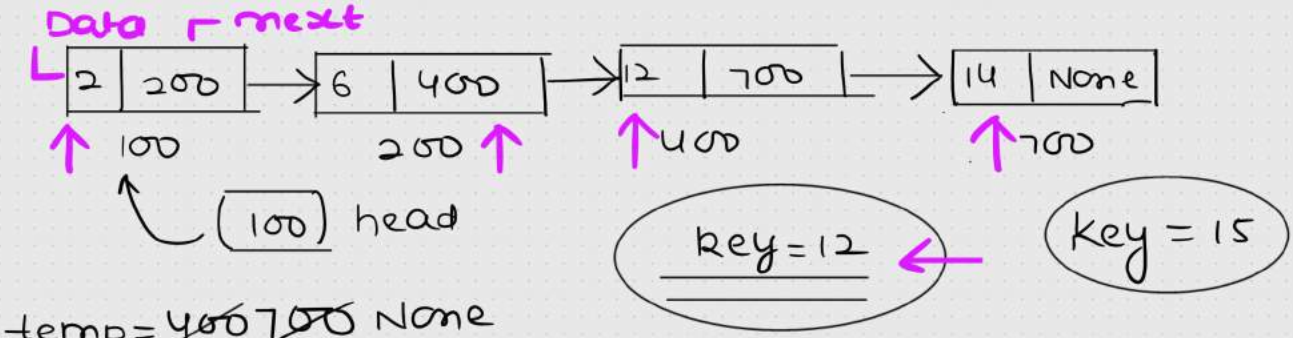
Base case condition

Linked list is empty

self.head is None:

return

Searching in Linked List



temp = ~~400~~ ~~700~~ None

Binary search (Random access not allowed)

↳ mid = i + (j - i) // 2

temp = ~~100~~ ~~200~~ 400

Linear search → O(n)

```
temp = self.head
while temp:
    if temp.data == key:
        return True
    temp = temp.next
return false
```

True

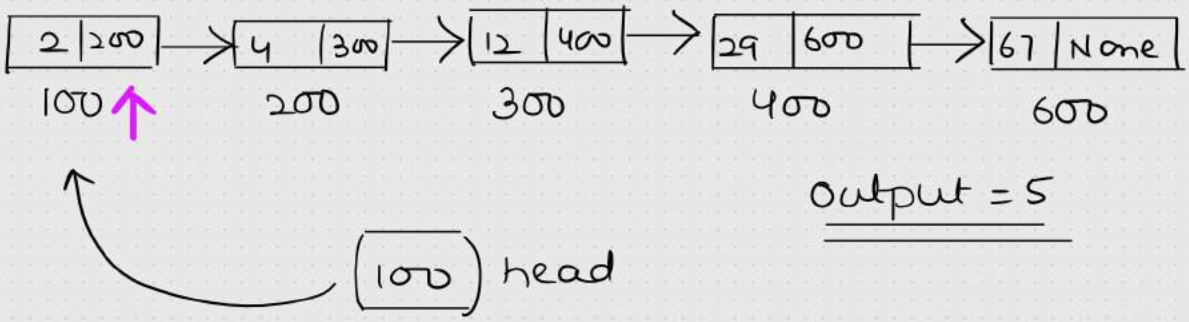
↳ (15 is not present inside the LL)

n → # nodes

Linked List → modified version of binary search

Skip list → Advance of DSA

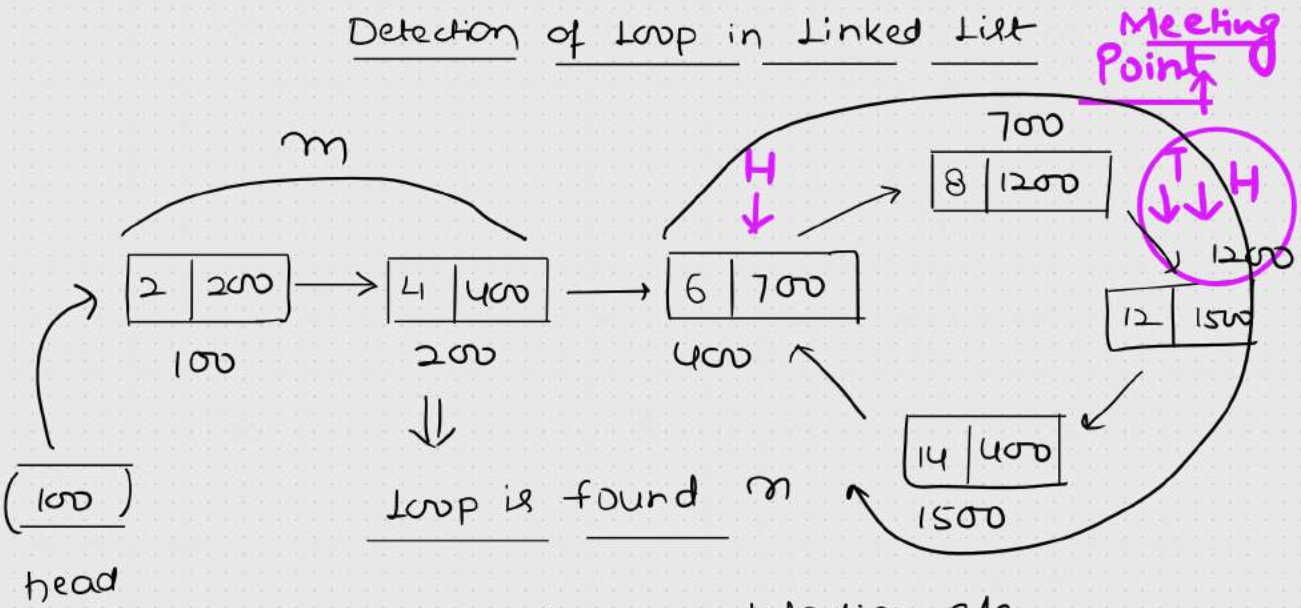
Count number of nodes



Output = 5

```
findNumNodes(self):
    temp = self.head
    count = 0
    while temp:
        count += 1
        temp = temp.next
    return count
```

Detection of Loop in Linked List



floyd's cycle detection algo

Hare - Tortoise

fast_ptr

- H → 2 node/iteration (fast & slow ptr)
- T → 1 node/iteration

slow_ptr

Meeting Point

```
if H==T:
    print("Loop is found")
if H==None:
    print("Loop is not found")
```

$$N = m + m + C * i \quad \text{---} (1)$$

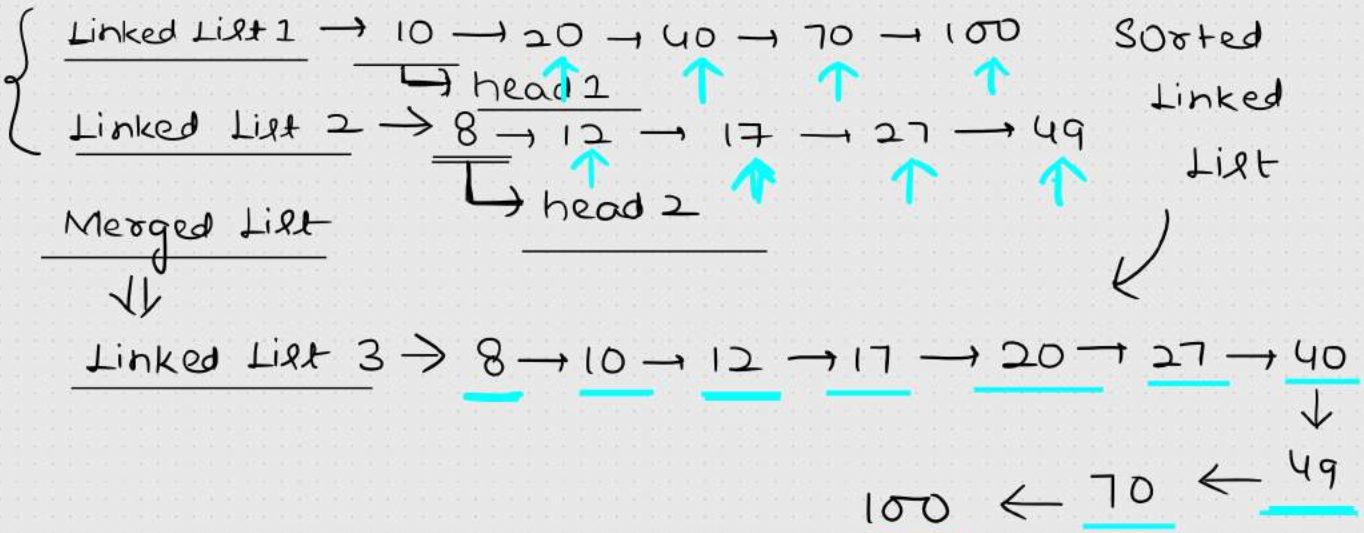
$$2N = m + m + C * J \quad \text{---} (2)$$

eq(2) − eq(1)

$$N = C(J - i)$$

↳ valid

Merge two sorted Linked list

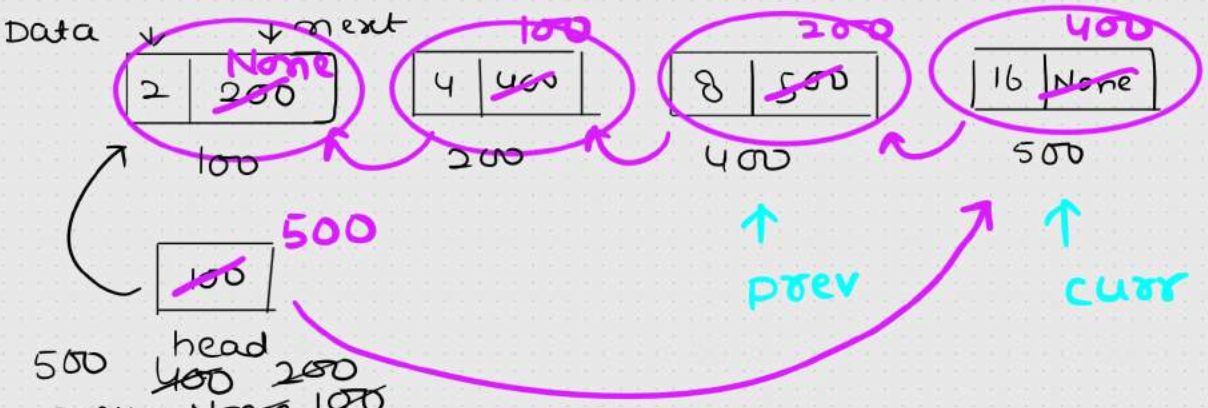



```
mergeLists(head1, head2):
    temp = None                          ← LL1 is empty
    if head1 is None:
        return head2
                                         ← LL2 is empty
    if head2 is None:
        return head1

    if head1.data <= head2.data
        temp = head1
        temp.next = mergeLists(head1.next, head2)
                                         ← Recursion
    else:
        temp = head2
        temp.next = mergeLists(head1, head2.next)

    return temp
```

Reversal of Linked List



500 400 200 100

prev = None None 500 400

curr = 100 200 400

next_ptr = 200 500 None

prev = None

next_ptr = None

curr = self.head

2 → 4 → 8 → 16

16 → 8 → 4 → 2

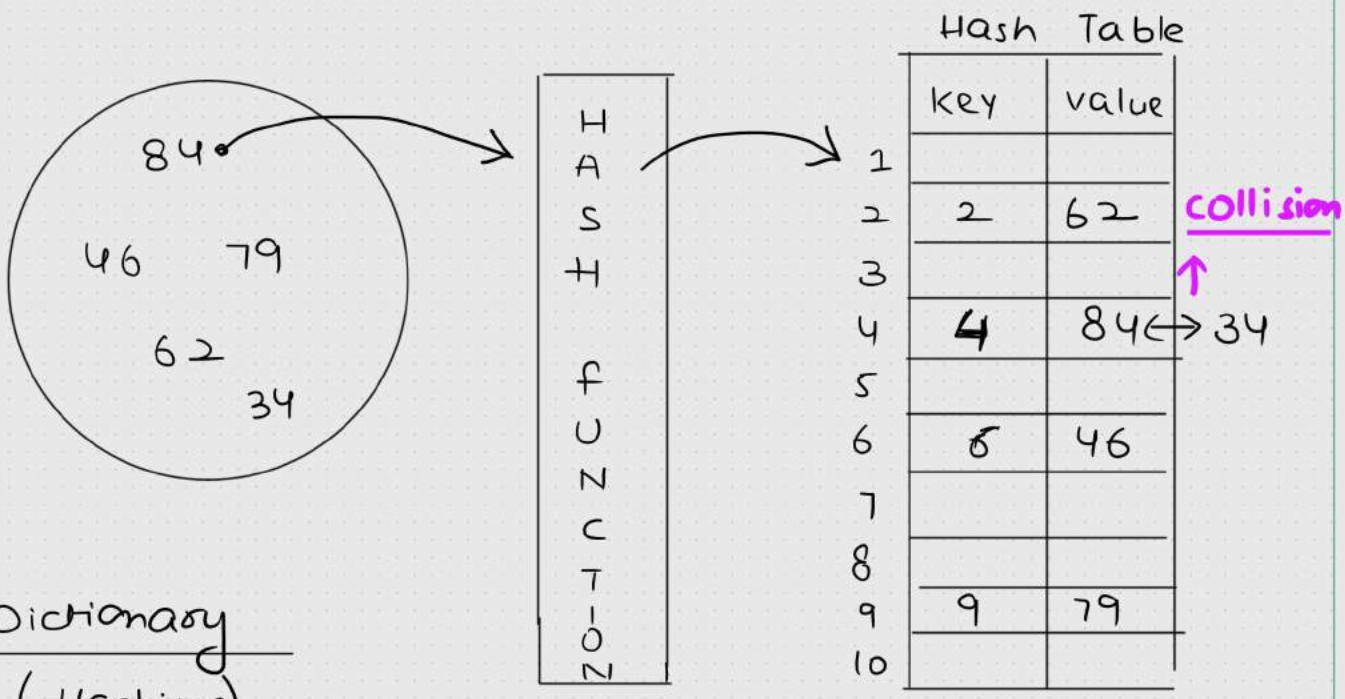
head

While curr :

- next_ptr = curr.next
- curr.next = prev
- prev = curr
- curr = next_ptr

self.head = prev

Different Types of Hash function



Dictionary (Hashing)

1) Division Modulo — $n \rightarrow$ size of the hash table

val % n

84 % 10 = 4

46 % 10 = 6

2) Mid-Square method

$(326)^2$ = 106276

n = 0 to 999

62 → key

3) Digit folding Method → three digits at a time.

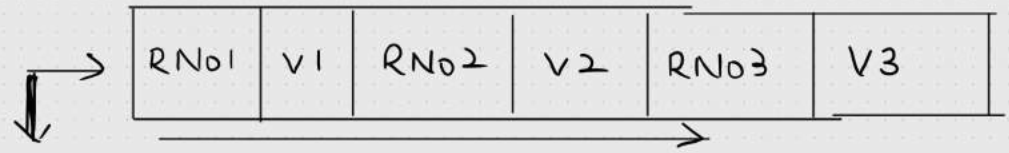
123456 24 → 123, 456, 24

Key = 123 + 456 + 24

= 603

- Less collisions
  - Open addressing
  - Chaining
- Uniform distribution of values
  - Hash Table

Hashing → Dictionary



Array / List in Python

Linear Search

Size of hash table

Sum % m = _____

ASCII code = sum

Roll no. → AI20MTECH14015 = 62

Hash function → key = 3

Hash Table → (key, value)

key	value
0	
1	
2	
3	62.0
4	
5	
6	

O(1) — Lookup

Hashing Data Structure

→ O(1) ⇒ constant

→ searching

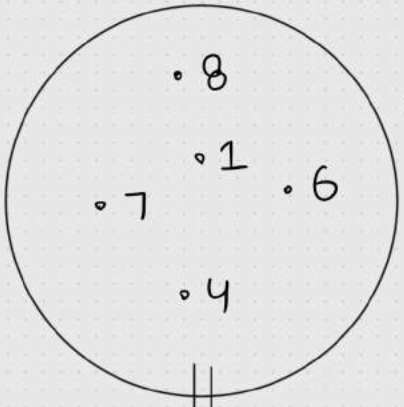
(key, value)

⇓

unique

Hash Table

Key	value
1	1
2	
3	
4	
5	
6	6
7	7
8	8



Direct addressing table

1000
2

Hash functions → Types

⇓

Division modulo Hash function

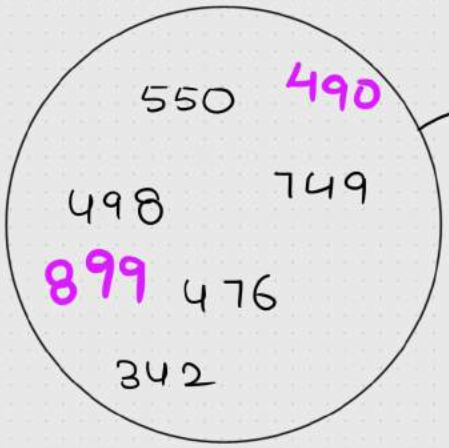
Key = Value % size of hash table

n = 10

collision

Hash table

Key	value
0	550
1	
2	342
3	
4	
5	
6	476
7	
8	498
9	749



Hash function

⇓

550 % 10 = 0

498 % 10 = 8

749 % 10 = 9

476 % 10 = 6

342 % 10 = 2

Collision

$490 \% 10 = 0$ (key)

$899 \% 10 = 9$ (key)

Collision Resolution Techniques

- open addressing
- chaining
  - Linked List

load factor

$\alpha = \frac{m}{n} \approx 0.75 \longrightarrow$ Rehashing

$n \rightarrow$ size of the hash table

$m \rightarrow$ # entries

↓ resize the hash table

$n \rightarrow 10$ → $n \rightarrow 10 * 2 = 20$

$m \longrightarrow 8$

$n \longrightarrow 10$

$\alpha = \frac{8}{10} = 0.8$

↓ Rehashing

Double Hashing → 2 Hash functions

$m = 10$

Values = 50, 75, 99, 20, 35, 88, 45, 23, 55, 67

→ $DH(v, i) = (hf_1(v) + i\,hf_2(v)) \% m$

$hf_2(v) = 1 + v \bmod (m-2)$

$hf_1(v) = v \% m$

$hf_1(v) = 50 \% 10 = 0$

$DH(50, 0) = (0 + 0) \% 10 = 0$

$hf_1(75) = 75 \% 10 = 5$

$DH(75, 0) = (5 + 0) \% 10 = 5$

$hf_1(99) = 99 \% 10 = 9$

$DH(99, 0) = (9 + 0) \% 10 = 9$

$hf_1(20) = 20 \% 10 = 0$

$DH(20, 0) = (0 + 0) \% 10 = 0$

$DH(20, 1) = (0 + 1 *$

$hf_2(20) = 1 + 20 \% 8$

$= 1 + 4$

$= 5$

$DH(20, 2) = (0 + 2 * 5) \% 10$

$= (0 + 10) \% 10$

$= 0$

$hf_1(20) = 0$

$hf_2(20) = 5$

Hash Table

Key	value
0	50
1	45
2	
3	35
4	
5	75
6	
7	
8	88
9	99

(0, 5)

$DH(20,3) = (0 + 3*5) \% 10$

$= 5$

$hf_1(35) = 35 \% 10 = 5$, $hf_2(35) = 1 + 35 \% 8$

$= 1 + 3$

$= 4$

$DH(35,0) = (5 + 0) \% 10 = 5$

$DH(35,1) = (5 + 1*4) \% 10 = 9$

$DH(35,2) = (5 + 2*4) \% 10$

$13 \% 10 = 3$

$hf_1(88) = 88 \% 10 = 8$

$DH(88,0) = (8 + 0) \% 10 = 8$

$hf_1(45) = 45 \% 10 = 5$, $hf_2(45) = 1 + 45 \% 8$

$= 1 + 5$

$= 6$

$DH(45,0) = (5 + 0) \% 10$

$= 5$

$DH(45,1) = (5 + 1*6) \% 10$

$= 11 \% 10 = 1$

(23, 55, 67) —— Hash Table

Note

1) Space is available → but still not able to insert all the elements
   ↳ inside hash table

2) Search, Delete, Insertion
   Best case → O(1)
   Worst case → O(n)

3) Not using any extra space outside the hash table.

Open Addressing

1) Linear Probing

m = Size of hashtable

m = 10

hf(value) = value % m

value = 50, 75, 99, 20, 35, 88, 45, 23, 55, 67

$hf(v) = v \% m$

$= 49 \% 10$

$m = 10 \quad = 9$

49

$LP(value, i) = (hf(value) + i) \% m$

Linear Probing

99

collision

$\Rightarrow 99 \% 10$

$= 9$

$50 \% 10 = 0$

$LP(50, 0) = (0 + 0) \% 10$

$= 0$

$75 \% 10 = 5$

$LP(75, 0) = (5 + 0) \% 10$

$= 5$

key	value	collision
0	50	20, 55
1	20	55
2	55	
3	23	
4	67	55
5	75	35, 45
6	35	55, 45
7	45	55
8	88	55
9	99	55

collision

collision

Hash Table

collision

NOTE: i = 0

$hf(value) = x$

$\% \; m = y$

$x = y$

$LP(99, 0) = 99 \% 10 = 9$

$(9 + 0) \% 10$

$= 9$

$LP(20,0) = hf(20) = 20 \% 10 = 0$

$(hf(value) + i) \% m$ → $(0+0) \% 10 = 0$

$LP(20,1) = (0+1) \% 10$

$= 1$

$LP(35,0) = hf(35) = 35 \% 10 = 5$

$(5+0) \% 10 = 5$

$LP(35,1) = (5+1) \% 10 = 6$

$LP(88,0) = hf(88) = 88 \% 10 = 8$

$(8+0) \% 10 = 8$

$LP(45,0) = hf(45) = 45 \% 10 = 5$

$(5+0) \% 10 = 5$

$LP(45,1) = (5+1) \% 10 = 6$

$LP(45,2) = (5+2) \% 10 = 7$

$LP(23,0) = hf(23) = 23 \% 10 = 3$

$(3+0) \% 10 = 3$

Collisions

LP(55, 0) = hf(55) = 55 % 10 = 5

(5 + 0) % 10 = 5

LP(55, 1) = (5+1) % 10 = 6

LP(55, 2) = (5+2) % 10 = 7

LP(55, 3) = (5+3) % 10 = 8

LP(55, 4) = (5+4) % 10 = 9

LP(55, 5) = (5+5) % 10 = 0

LP(55, 6) = (5+6) % 10 = 1

LP(55, 7) = (5+7) % 10 = 2

65 → hf(65) = 65 % 10 = 5

Primary clustering → Drawback of linear Probing

Worst case search time → O(n)

(Empty slot)

Quadratic Probing

$m = 10$

$hf(v) = v \% m$

$LP(v,i) = (hf(v) + i) \% m$

values = 50, 75, 99, 20, 35, 88, 45, 23, 55, 67

$QP(v,i) = (hf(v) + c_1 i + c_2 i^2) \% m$

↳ # collisions

$c_1 = c_2 = 1$ Best case → O(1)

$QP(50,0) = hf(50) = 50 \% 10 = 0$

↳ $(0 + 0 + 0) \% 10 = 0$

$QP(75,0) = hf(75) = 75 \% 10 = 5$

$(5 + 0 + 0) \% 10 = 5$

$QP(99,0) = hf(99) = 99 \% 10 = 9$

$(9 + 0 + 0) \% 10 = 9$

$QP(20,0) = hf(20) = 20 \% 10 = 0$

$(0 + 0 + 0) \% 10 = 0$

$QP(20,1) = (0 + 1 + 1) \% 10 = 2$

$QP(35,0) = hf(35) = 35 \% 10 = 5 \rightarrow (5 + 0 + 0) \% 10 = 5$

$QP(35,1) = (5 + 1 + 1) \% 10 = 7$

$QP(88,0) = hf(88) = 88 \% 10 = 8$

↳ $(8 + 0 + 0) \% 10 = 8$

Hash Table

key	value
0	50
1	45
2	20
3	23
4	
5	75
6	
7	35
8	88
9	99

$Q_P(45,0) = hf(45) = 45 \% 10 = 5$

$\hookrightarrow (5+0+0) \% 10 = 5$

$Q_P(45,1) = (5+1+1) \% 10 = 7$

$Q_P(45,2) = (5+2+4) \% 10 = 1$

$(hf(v) + c_1 i + c_2 i^2) \% m$

$Q_P(23,0) = hf(23) = 23 \% 10 = 3$

$\hookrightarrow (3+0+0) \% 10 = 3$

$Q_P(55,0) = hf(55) = 55 \% 10 = 5$

$\hookrightarrow (5+0+0) \% 10 = 5$ ✓

(5, 7, 1)

↳ already filled

$Q_P(55,1) = (5+1+1) \% 10$

$= 7$ ✓

cycle

$Q_P(55,2) = (5+2+4) \% 10$

$= 1$ ✓

$Q_P(55,3) = (5+3+9) \% 10$

$= 17 \% 10 = 7$ ✓

$Q_P(55,4) = (5+4+16) \% 10$

$= 5$ ✓

$QP(67, 0) = hf(67) = 7$ (7, 9, 3)

↳ $(7 + 0 + 0)\%10 = 7$

$QP(67, 1) = (7 + 1 + 1)\%10$

$= 9$

Worst case Search time ↳ O(n)

Cycle

$QP(67, 2) = (7 + 2 + 4)\%10$

$= 3$

$QP(67, 3) = (7 + 3 + 9)\%10$

$= 9$

$QP(67, 4) = (7 + 4 + 16)\%10$

$= 7$

$hf(77) = 7$ ↑ wastage of time

value = 77 → Same path followed (value = 67)

$hf(67) = 7$

Drawback → { Secondary clustering

Delete

↳ replace → $ symbol

↓

Rehashing → get rid of all $ symbols

Stack

↳ LIFO

↳ Last In first Out

OR

FILO

↳ first In Last Out

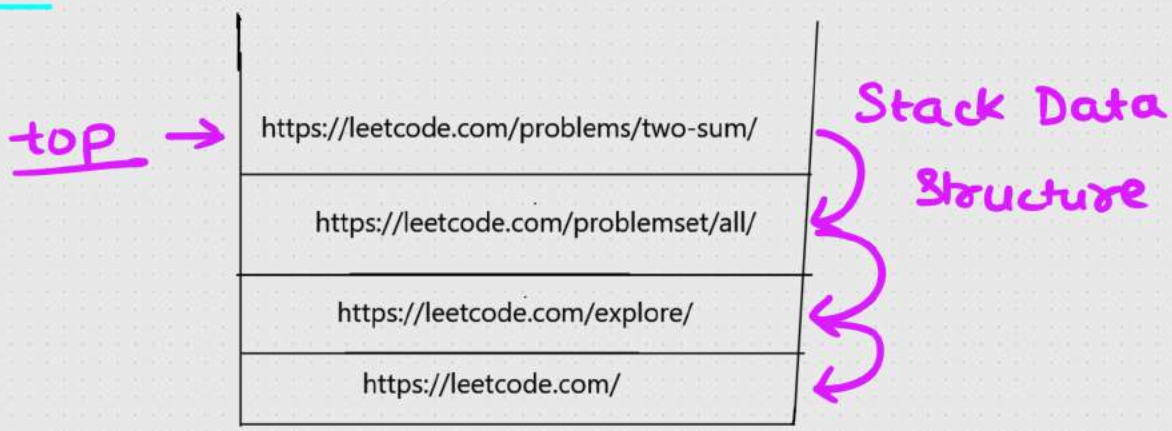
Abstract Data Type

⇓

1) Push → insert the elements inside the elements

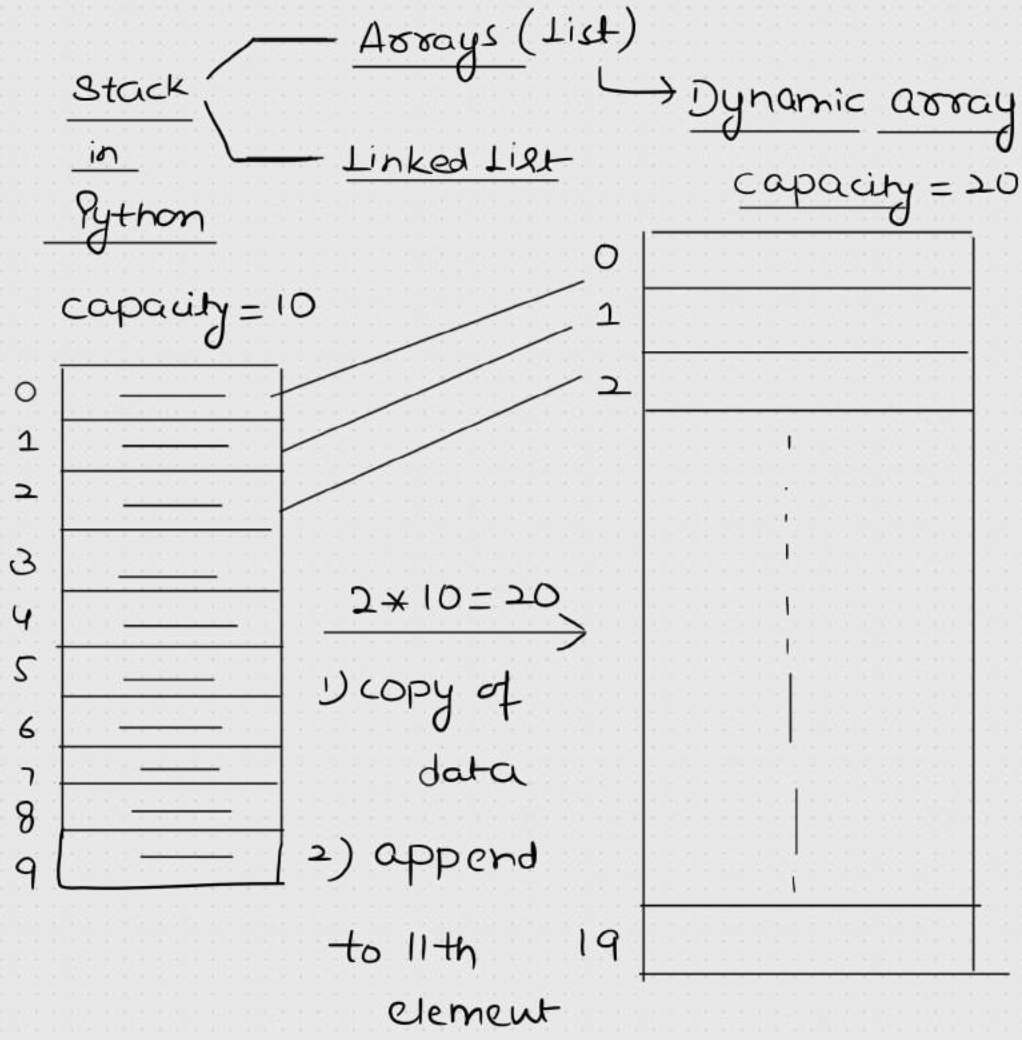
2) Pop → delete the elements from the stack

Push → top variable



undo → Ctrl+Z (LIFO)

function calling → Recursion



Push → Static array

O(1) ⇐ push(arr, data):

↓↓

constant time operation

if top == n:

↳ stack overflow

else:

top = top + 1

arr[top] = data

Pop

Stack

↳ LIFO

↳ Last In first Out

OR

FILO

↳ first In Last Out

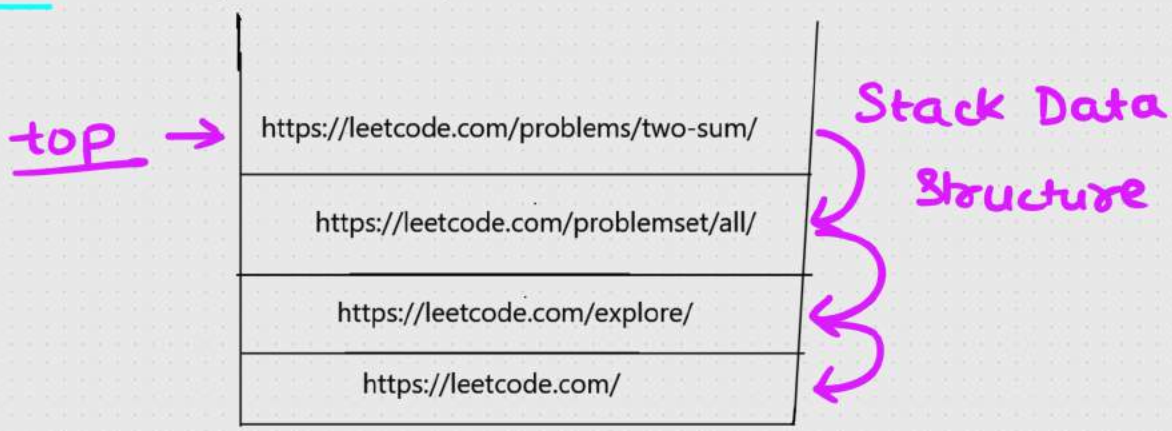
Abstract Data Type

⇓

1) Push → insert the elements inside the elements

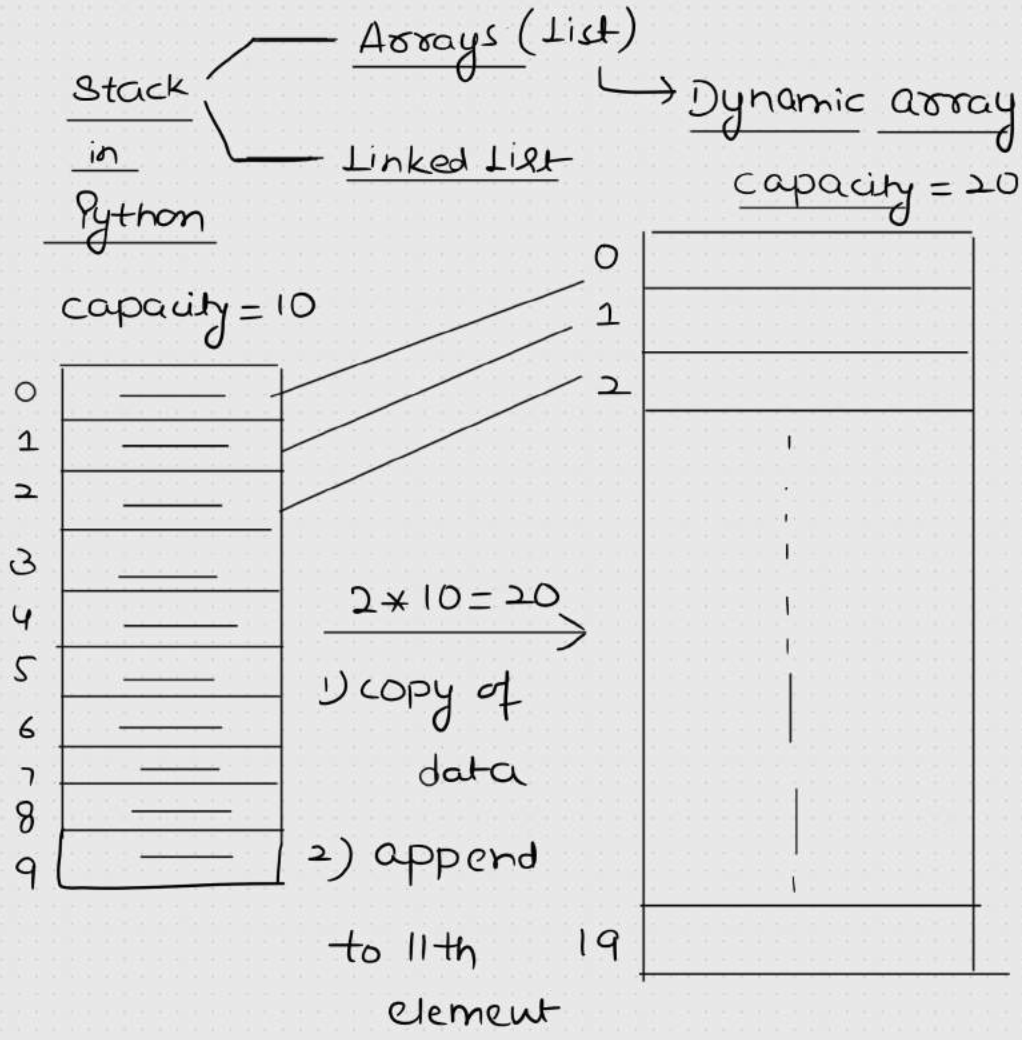
2) Pop → delete the elements from the stack

Push → top variable



undo → Ctrl+Z (LIFO)

function calling → Recursion



Push → Static array

O(1) ⇐ push(arr, data):

↓↓

constant time operation

if top == n:

↳ stack overflow

else:

top = top + 1

arr[top] = data

Pop

Psuedocode

```
pop():
    if top == -1:
        print("stack underflow")
    else:
        data = arr[top]
        top -= 1
    return data
```

stack is empty

O(1) ⟸
↳ constant

Stack — Push, Pop ⟹ O(1) (constant)

Python

Array (Lists)
↳ append
pop()
↳ pop

push

Linked List (circular deque) → Documentation
↳ append
pop()
↳ pop

push

peek → Similar to pop (Deletion of data not happened)

Valid Parenthesis

{ }, [ ], ( )

{ ([{ }]

String = '{ ( ) }'

0 1 2 3

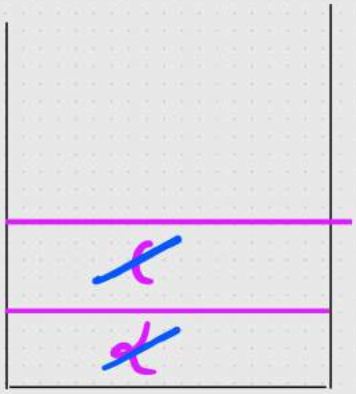
{, [, (

Stack → Last In first Out

1) for every open bracket, store char inside the stack.

char == ')'

dictionary['('] = ')'



Stack Data Structure

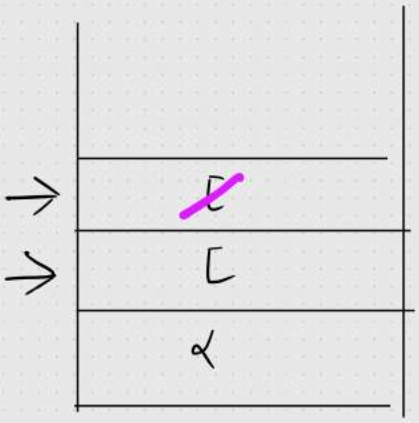
2) for every close bracket, pop the element from the stack.

3) valid Parenthesis

↳ Empty stack

dictionary[key] = value

d[[]] ⟶ false



Queue Data Structure

→ FIFO ⇒ First In first Out

↳ Queue in a Railway Station

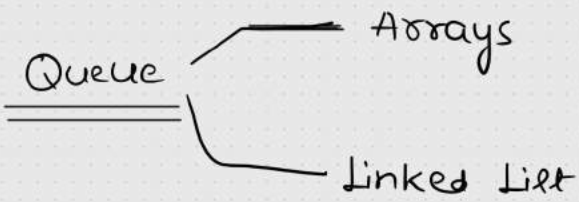
front → Deletion

Rear → Insertion

(Abstract Data Type)

ADT → 1) Enqueue → Insertion of element

2) Dequeue → Deletion of element



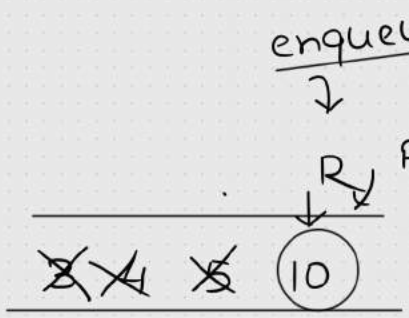
Psuedocode (Enqueue)

⇒ f = R = -1 → Queue is empty

```
Enqueue(Q, data):                    # Queue is fully filled
    if (R == n-1):
        print('Queue overflow')
                                     # Queue is empty
    elif (f == R == -1)
        f += 1
        R += 1
        Q[R] = data
    else:
        R += 1
        Q[R] = data
```

O(1) → constant

FIFO



O(1)

constant time complexity

```
Dequeue(Q):
    if (f = R = = -1)          # Queue is empty
        print('Queue underflow')
    elif f = = R:              # Queue single element
        f = R = -1
    else:
        data = Q[f]
        f = f + 1
    return data
```

Chaining → Linked list

values → 70, 49, 27, 62, 43, 10, 25, 34, 55

1000 2000 4000 5000 8000 9000 7000 1500 1100

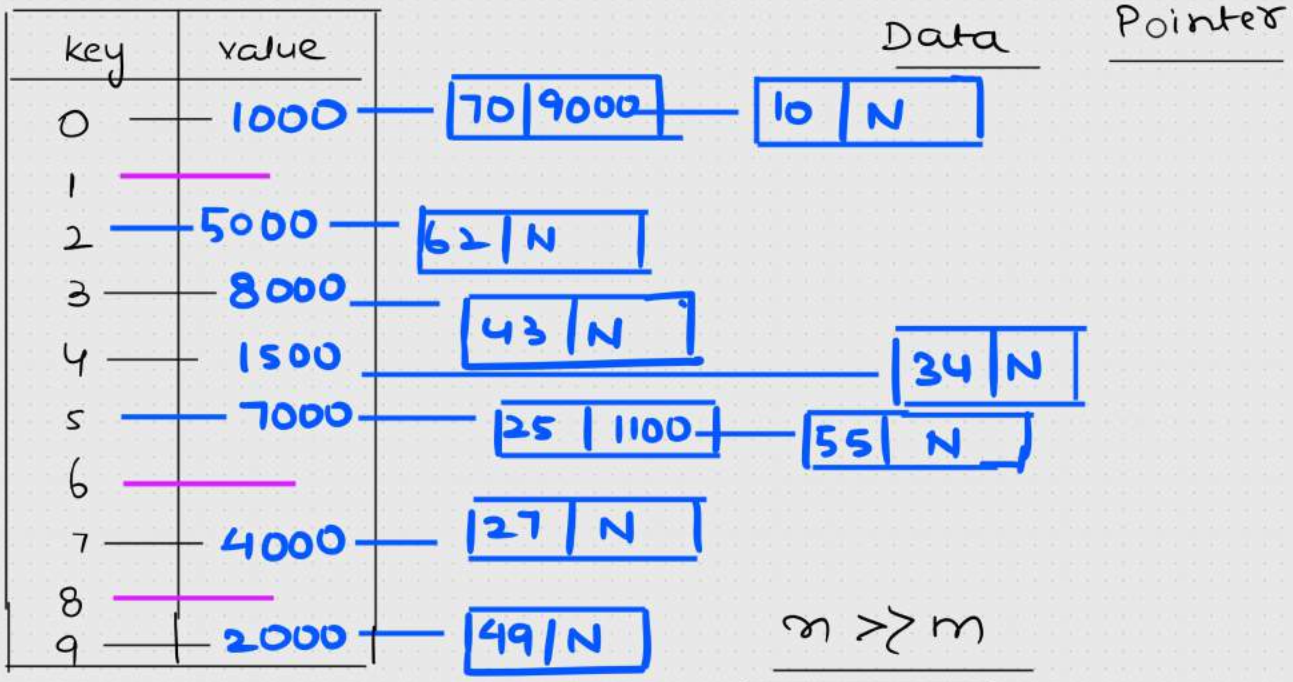
$m = 10$

$hf(v) = v \% m$

Linked list

↳ Node

Data | Pointer



$n >> m$

$hf(10) = 10 \% 10 = 0$

$hf(55) = 55 \% 10 = 5$

Advantage → Store as many data as we want

↳ Hash Table

Disadvantage → Extra space to store the data

↳ space is available inside hash table

Search → Worst case → O(N)

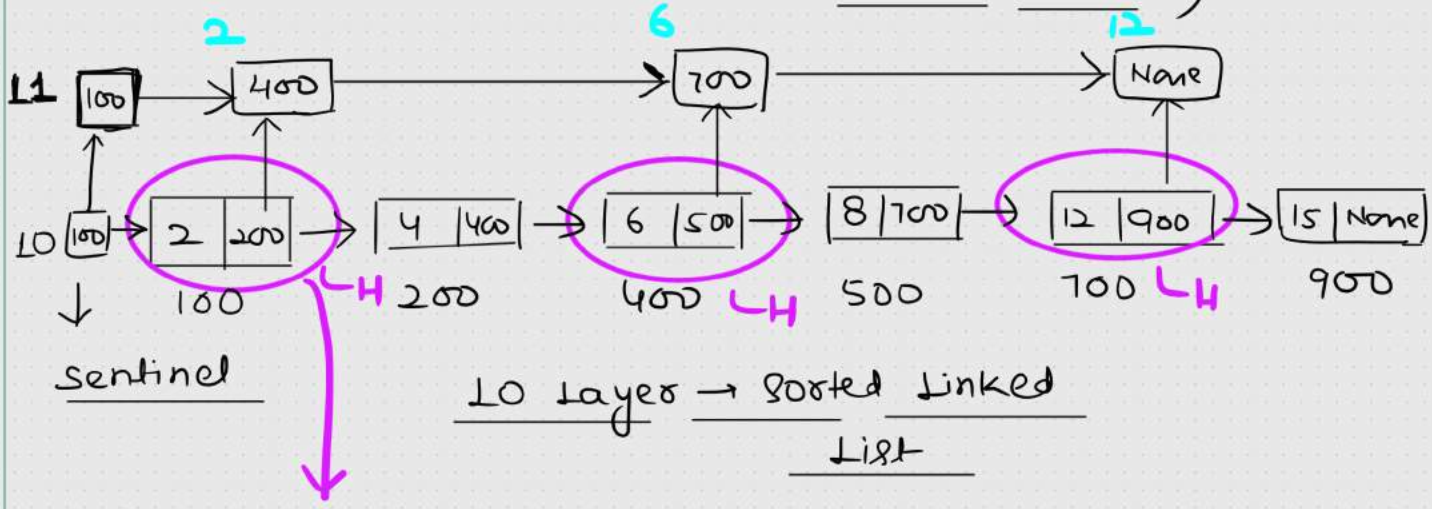
Deletion → best case → O(1)

Insertion → O(1)

Skip List

↳ Binary Search

Node — Data
Node — Next Pointer (Data inside the node sorted manner)



L0 layer → Sorted Linked List

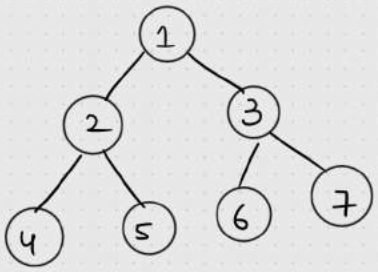
Randomization → extend node to upper layer

flip a coin
- P = 1/2 (Head)
- P = 1/2 (Tail)

(Unbiased) Random

↳ No extension of the node to upper layer

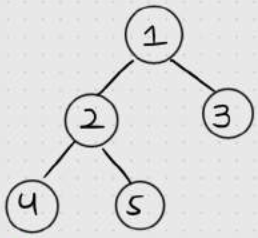
Tree Traversal Algorithms



1) Inorder
2) Preorder
3) Postorder

Recursion concept

Inorder
- Left side of the tree
- print root node
- Right side of the tree



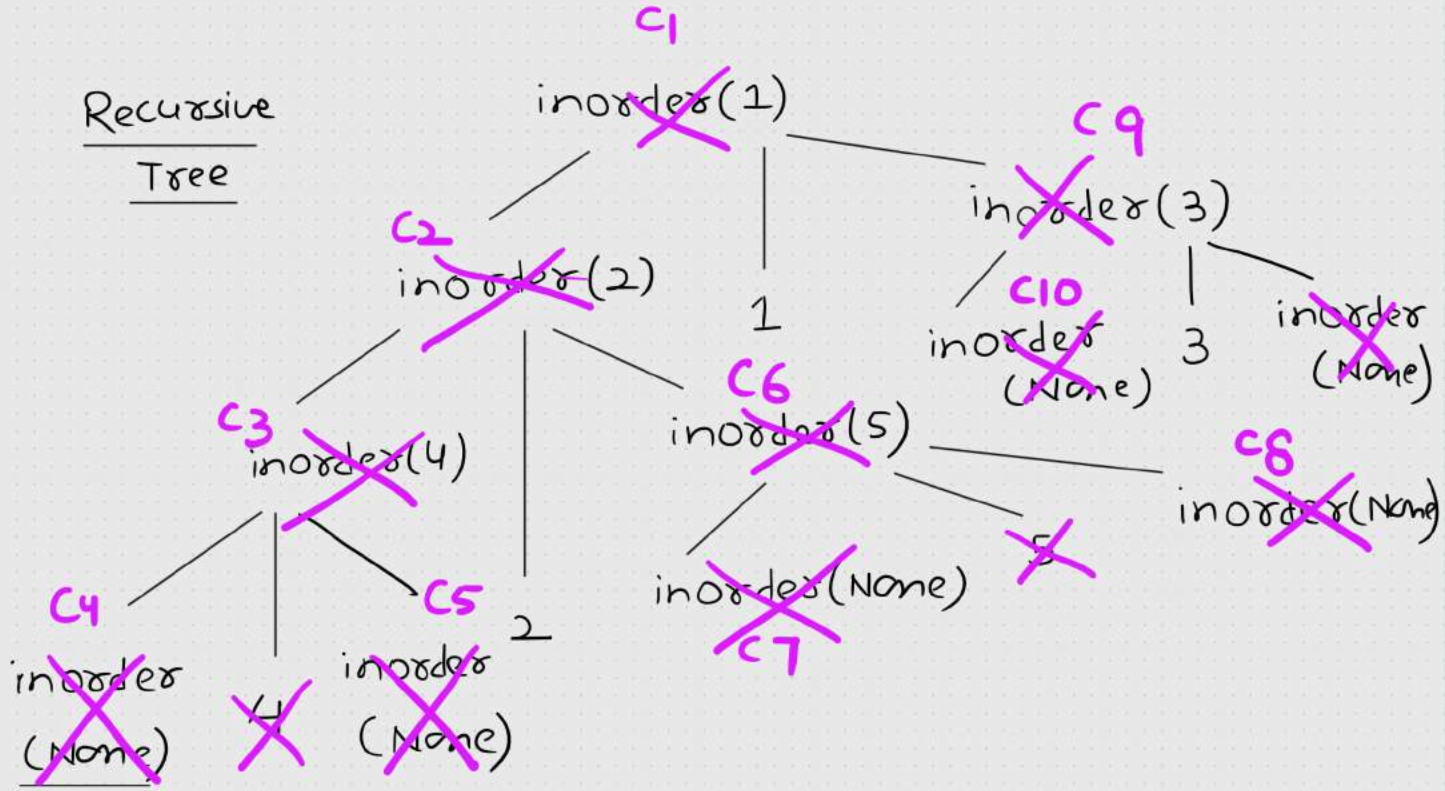
Root Node

Leaf Node

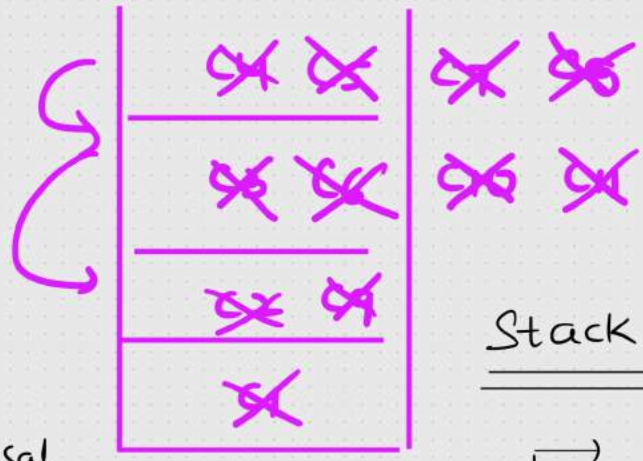
$T(n)$

```
inorder(root):
    if root:                      <- Recursion
T(n/2)  inorder(root.left)
 c      print(root.data)
T(n/2)  inorder(root.right)
```

Recursive Tree



4 2 5 1 3



Stack

for every traversal algorithm → Recurrence Relation (Best/average case scenario)

→ Balanced Tree

$$T(n) = 2T(n/2) + c$$

By master's theorem

$a = 2$ $\quad k = 0$

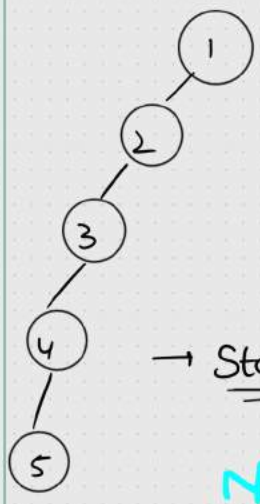
$b = 2$ $\quad p = 0$

$$\log_b^a = 1 > k$$

$\hookrightarrow O(n)$

Stack space

$\hookrightarrow \log_2 n$



worst case scenario → unbalanced tree or skewed tree

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

$\Rightarrow O(n)$

→ Stack Space

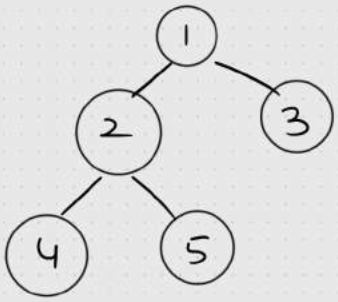
$\hookrightarrow O(n)$

Note: for every case, overall time complexity = $O(n)$

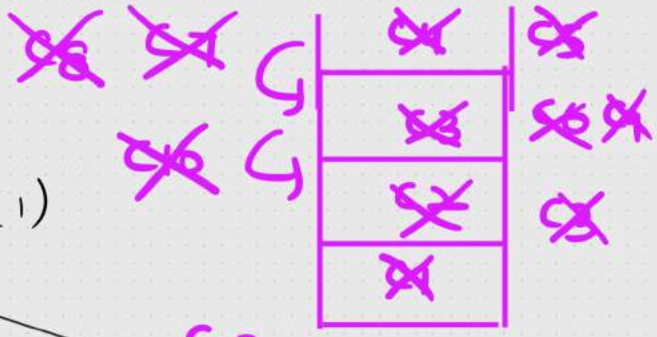
Preorder Traversal

```
preorder(root):
    if root:
        print(root.data)      — ①
        preorder(root.left)   — ②
        preorder(root.right)  — ③
```

↳ Recursion



1 2 4 5 3

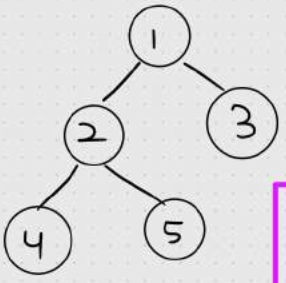


1 2 4 5 3

Recursive Tree

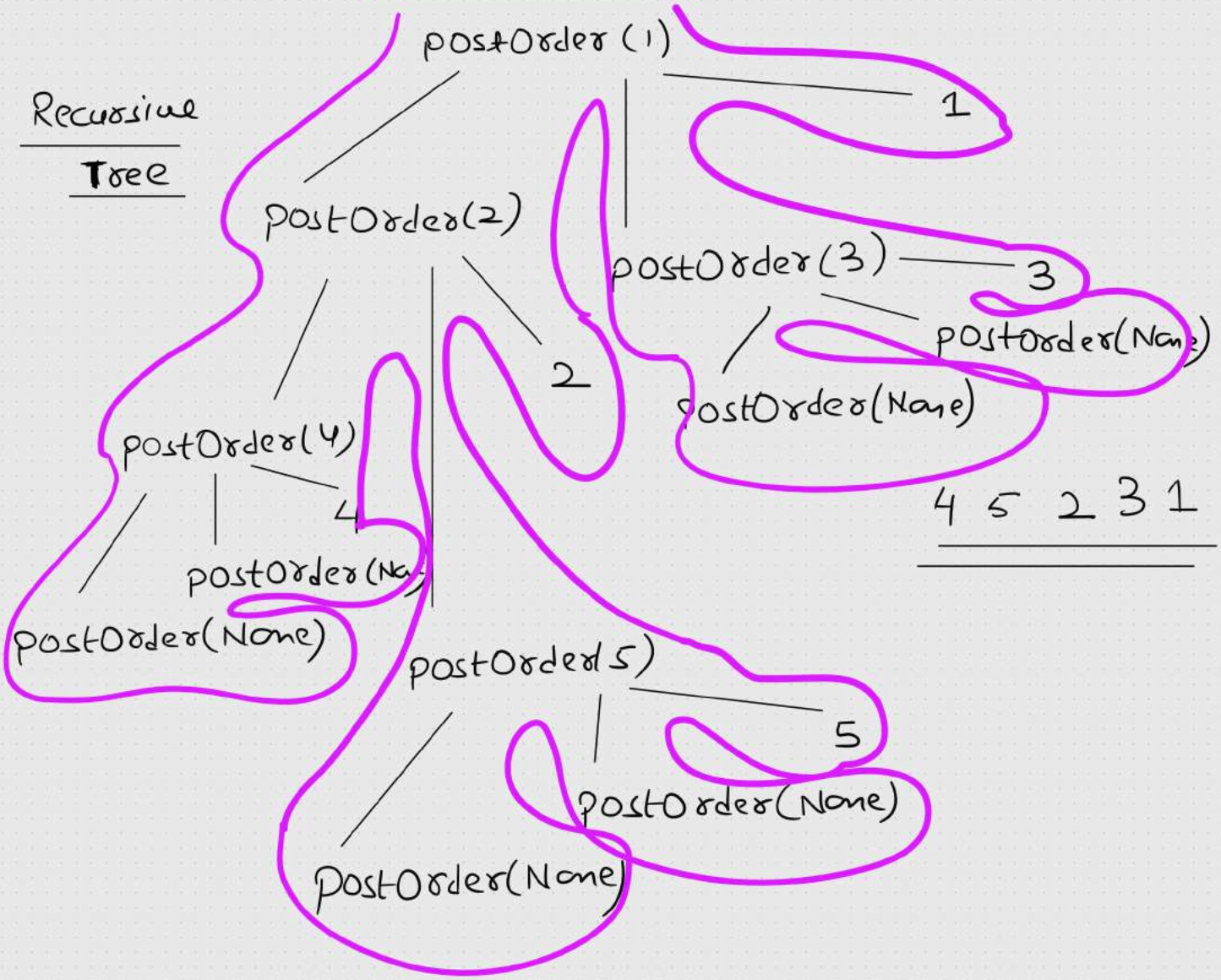
Postorder

```
postOrder(root):
    if root:
        postOrder(root.left)
        postOrder(root.right)
        print(root.data)
```



4 5 2 3 1

Recursive Tree



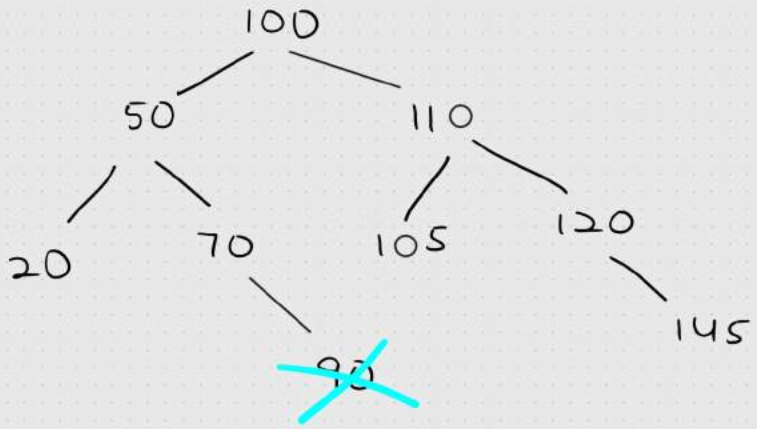
4 5 2 3 1

Note: Root Node of a Tree

- Inorder (Middle)
- Preorder (first)
- Postorder (Last)

Deletion in BST

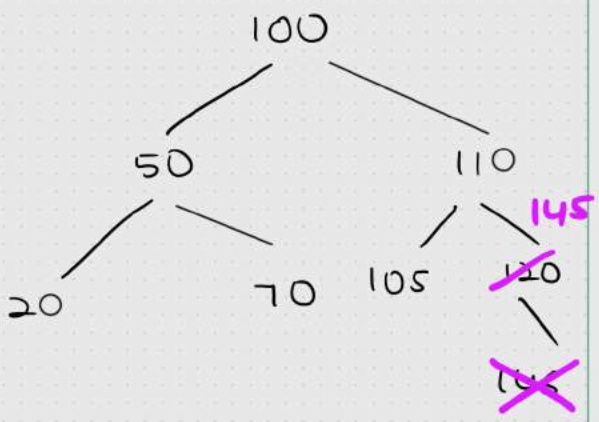
100, 110, 50, 70, 120, 90, 145, 105, 20



1) Delete a node → no child
(leaf node)

Delete (90)

1) Search → 90
2) Delete 90



2) Delete a node → one child

1) search → 120
2) Delete (120)

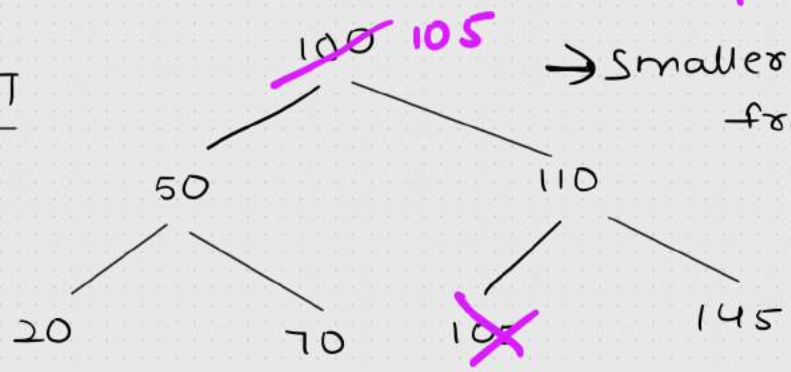
temp = 145

Inorder successor

↑

→ Smaller value from right subtree

Valid BST



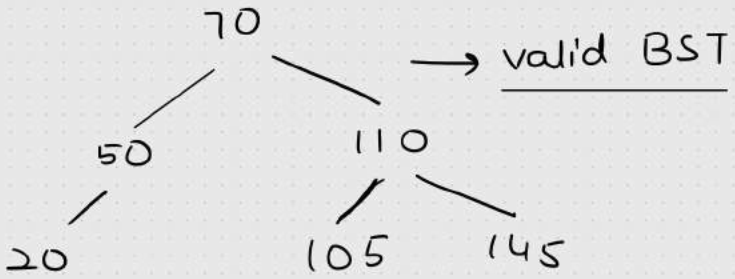
Inorder Predecessor

← → larger value in left subtree

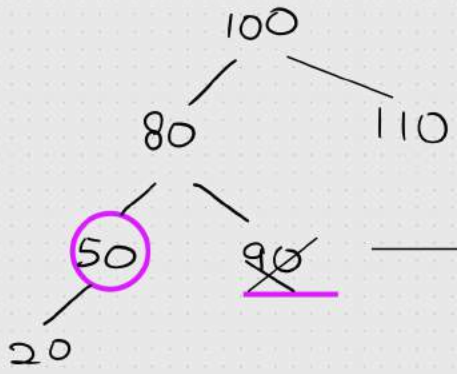
3) Delete a node → two child

Delete (100)

OR



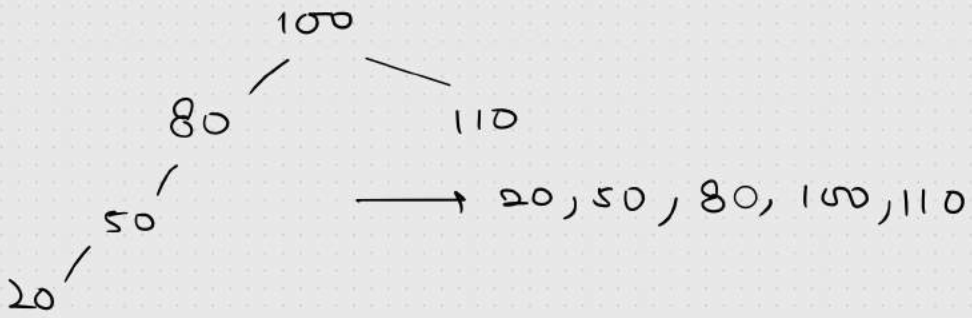
→ valid BST



Inorder Traversal

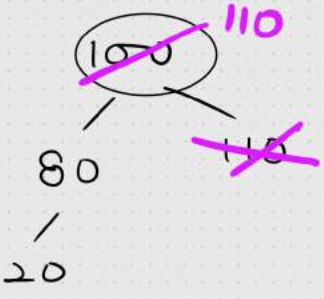
→ 20, 50, 80, 90, 100, 110

1) Delete (90) — No child



→ 20, 50, 80, 100, 110

2) Delete (50) — single child



→ 20, 80, 100, 110

3) Delete(100) – Two child

20,80,110

Interview Question (catalan Number)

$n \rightarrow$ # nodes in BST

Possible BST $\Rightarrow$??

$C_0 = C_1 = 1$

$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \cdots + C_{n-1} C_0$

$C_2 = C_0 C_1 + C_1 C_0$

$= 1 + 1 = 2$

$C_0 \; C_1 \; C_2 \; C_3$

$1, 1, 2, 5$

$C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0$

$= 1 * 2 + 1 + 2$

$= 5$

$n = 0$ ——— 1 way

$n = 1$ ——— 1 way

$n = 2$ ——— A — B OR B — A 2 ways

$A < B$

(A, B)

$n = 3$ ——— A — BC OR B (3) — A C 5 ways

(A, B, C)

$A < B < C$

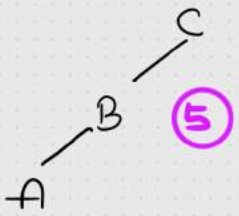
A — B — C (1)

A — C — B (2)

B — A — C (3)

B — C — A (4)

C — A — B (5)



n ⟶ Possible unique BST

⇓

Catalan Number

Interview Questions

PostOrder = D I H C E A B G F J

InOrder = D E H I C J F B A G

Inorder, Postorder ↔ unique binary tree

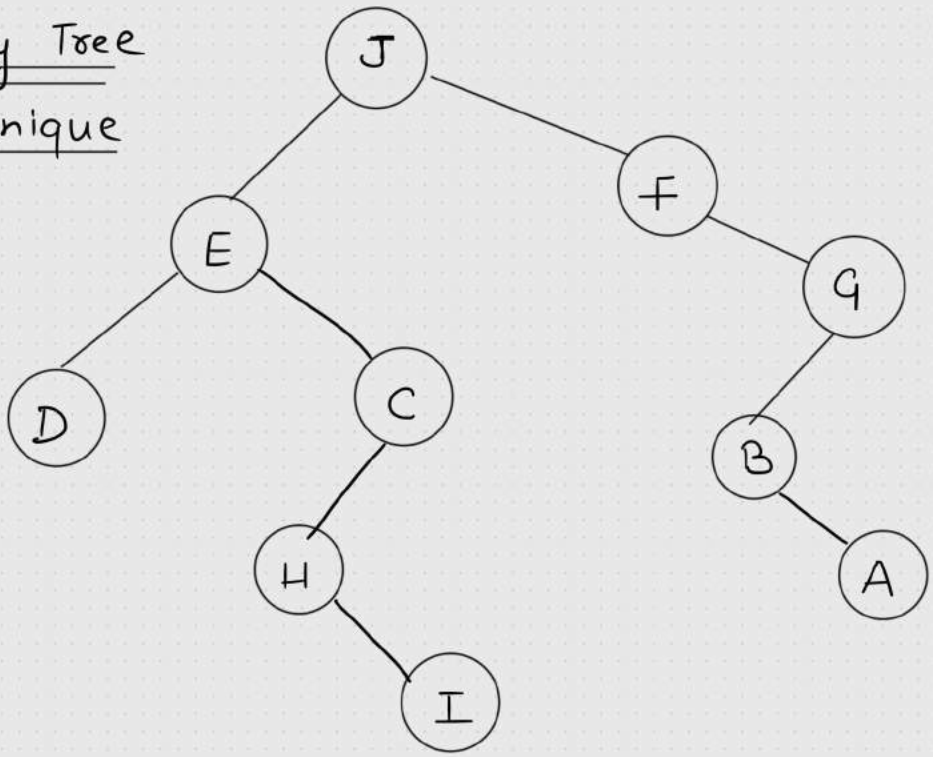
Inorder, Preorder ↔ unique binary tree

Postorder, Preorder ↔ multiple binary tree

Inorder
- Left subtree
- Root Data
- Right Subtree

Postorder
- Left Subtree
- Right Subtree
- Root Data

Binary Tree
↳ unique



Inorder: D E H I C J F B A G

Postorder :

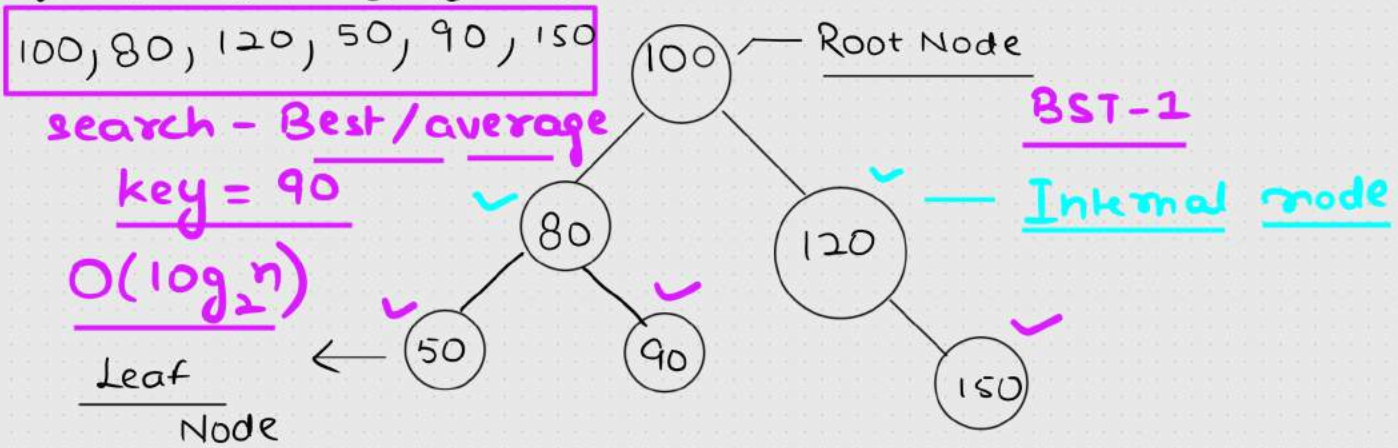
Preorder :

Practice Task :→ Time complexity required to create/ Obtain unique binary tree?

Binary Search Tree

1) Left node data < Parent node data
2) Right node data > Parent node data

100, 80, 120, 50, 90, 150



search - Best/average

key = 90

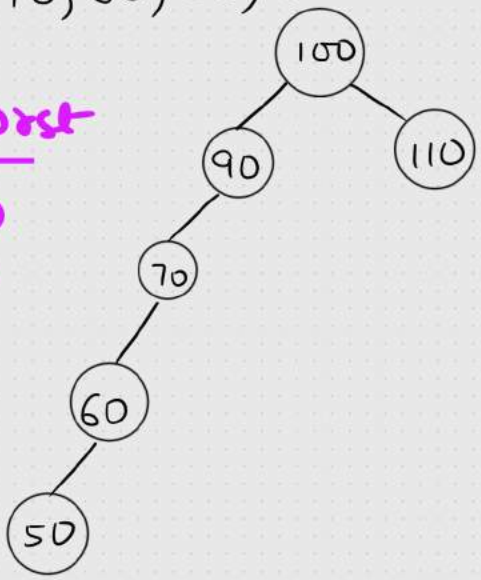
$O(\log_2 n)$

BST-1

BST

Worst case scenario → Skewed BST → Left skewed BST, Right skewed BST

100, 90, 70, 60, 50, 110



→ Left skewed Binary Search Tree

search - Worst

key = 50

O(n)

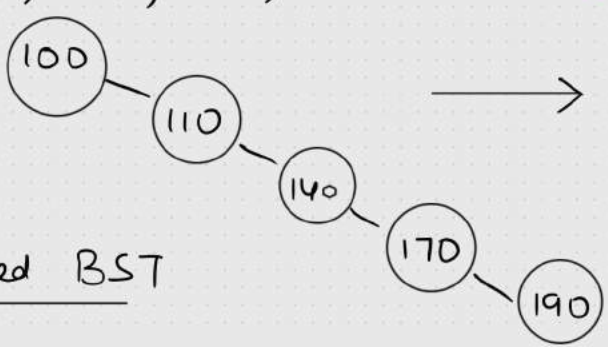
BST-2

Worst case Scenario

key = 210

O(n)

100, 110, 140, 170, 190, 210



→ Right skewed Binary Search Tree

BST-3

unbalanced BST

Note:

1. Balanced BST (Best/average) $\rightarrow O(\log_2 n)$
2. Unbalanced BST (Worst) $\longrightarrow O(n)$

* Skewness

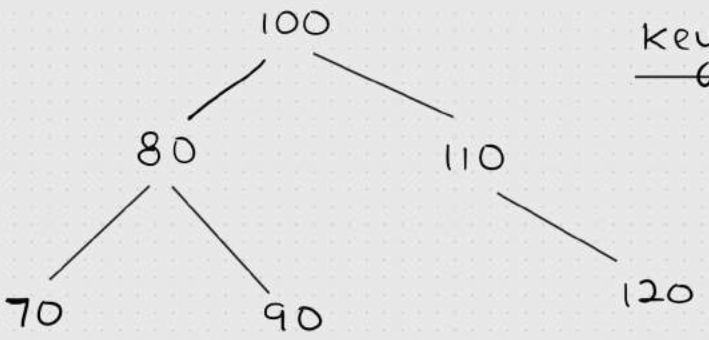
self balancing Tree

1. AVL Tree
2. Red Black Tree

Rules & Regulations

Searching in BST

100, 80, 110, 70, 90, 120



key = 80
↳ Present

key = 140
↳ Not Present

BST
- Balanced → searching
  (Best & Average case)
  ↳ O(log n)
- Unbalanced → Skewed BST
  (Worst case scenario)
  ↳ O(n)

Graph Data Structure

↳ Non-Linear Data Structure

1. Depth first Search
2. Breadth first Search

← Graph — Edge
Vertical / Node

↓ form a cycle

undirected or directed

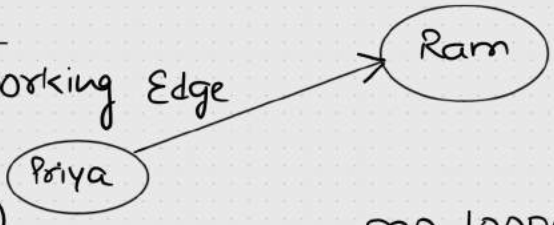
Tree → 1) Preorder
2) Postorder
3) Inorder

↳ Not form a cycle

Directed

Application

↳ Networking Edge (social media platform)



Two types — simple graph (no loops)
multigraph (cycle)

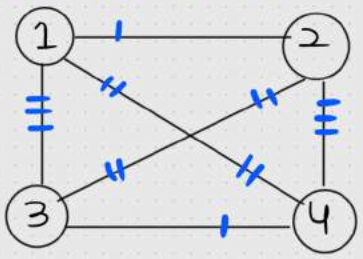
↓ loops are allowed

Degree

for each node

max degree ↕ complete graph

n = # node

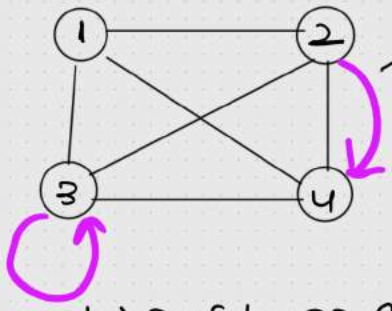


→ simple graph
→ undirected graph
→ Degree of all node = 3 (n-1)

↳ how many edges are connected to each node

Max Degree = n - 1

→ Min Degree = 0



→ Parallel edges are also allowed

→ Multigraph

↳ self loops are allowed

Degree(A) = 0

Null graph

A B Degree(B) = 0

C D Degree(D) = 0

Degree(C) = 0

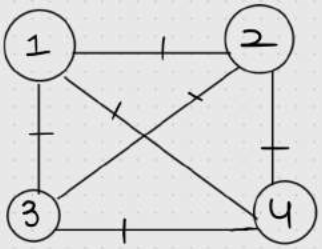
1) simple graph
   - null graph ↔ 0
   - complete graph ↔ n-1

Max Degree
↳ n-1

Min Degree → 0

2) multigraph

Properties of complete graph



n → # vertices or # nodes

Total number of edges = 6

sum of Degree

$$\Rightarrow \frac{n * n-1}{2}$$

$$\Rightarrow \frac{4 * 3}{2} = 6$$

n = 4

Sum of Degree = $4 * 3 = 12$

4 ↓ # nodes; 3 → Degree of each node

$= n * (n-1)$

Total num of edges $= \frac{\text{sum of degree}}{2}$

sum of degree = Total num of edges $* 2$

$= 2 * E$

Relationship b/w # edges & # vertices

complete graph →

$E = \frac{V(V-1)}{2}$ True

$2E = V^2 - V$

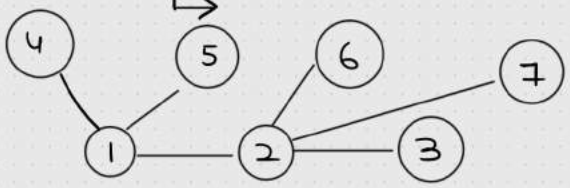
$E = O(V^2)$

By taking log on both sides;

$\log E = 2 \log V$

$\log E = O(\log V)$

Breadth first Traversal



1 2 4 5 3 6 7

FIFO (first In first Out)

Queue

Psuedocode — No Recursion

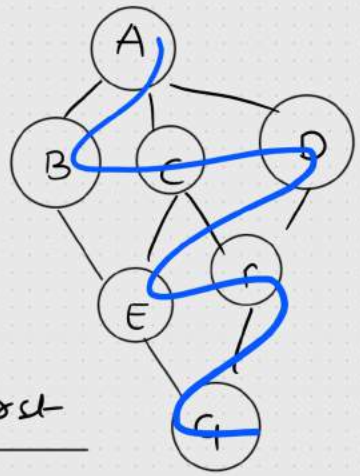
```
BFT(v):
    visited(v) = 1
    add(v, Q)
    while Q:                    // Not empty
        x = delete(Q)
        print(x)
        for all ω adj to x:     // 2E
            if ω — not visited:
                add(ω, Q)
```

Q - Queue

Time complexity → $O(V+E)$

Graph

A B C D E F G



Breadth first

Traversal

↳ Level order

Traversal

BFT(A)

~~A~~	

ω = BCD

~~B~~	~~C~~	~~D~~

ω = AE

~~E~~	~~D~~	~~E~~

ω = Ef

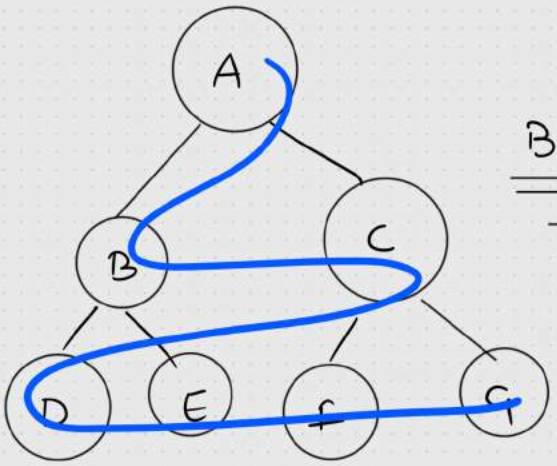
~~f~~	~~f~~	~~f~~

ω = ~~Af~~

ω = ~~BCG~~

~~G~~	~~G~~

ω = ~~CDG~~



BFT

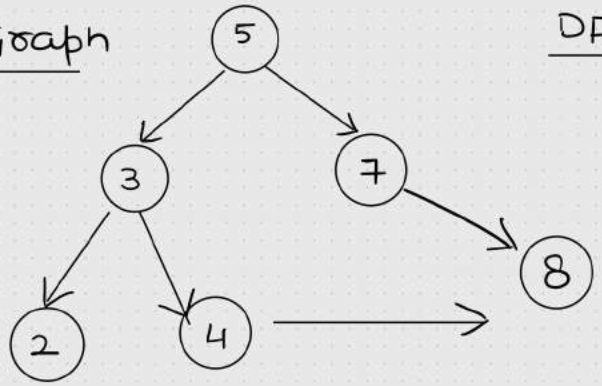
A B C D E F G

Level Order

Traversal

DFT Implementation → Recursion

Directed Graph



- 5 → 3, 7
- 3 → 2, 4
- 4 → 8
- 7 → 8
- 2 → —

DFT(5)

ω = 3, 7

DFT(3)

ω = 2, 4

DFT(2)

ω = X

DFT(7)

ω = 8

DFT(4)

ω = 8

DFT(8)

ω = X

5, 3, 2, 4, 8, 7

Psuedocode of DFT

```
DFT(v):
    visited(v) = 1
    print(v)
    for all ω adj v:
        if ω is not visited:
            DFT(ω)
```

0 → initialization
1 → vertex is visited

Adjacency List
↳ 2E

↳ Recursion

A	B	C	D	E	f	G
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~ 1	~~0~~ 1

visited

1 1 1 1 1

DFT(A)
ω = B C D

DFT(B)
ω = A E

DFT(E)
ω = B C G

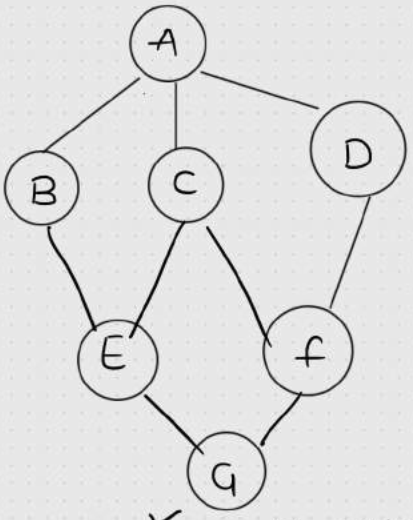
DFT(C)
ω = A E f

DFT(f)
ω = C D G

DFT(D)
ω = A f

DFT(G)
ω = E f

Graph



DFT →

DFT → A B E C f D G

Traversal

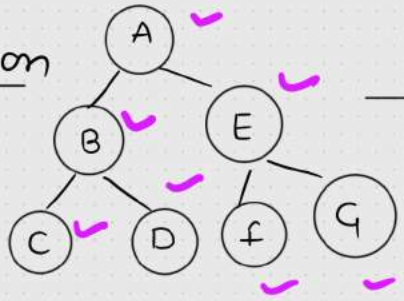
↳ other Results

Time complexity → O(V+E)

Recursion → function calls
↳ Stack Data Structure

Depth first Traversal

1) Exploration
2) visit

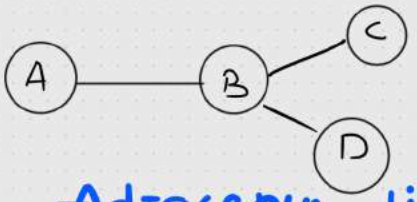


→ A B C D E f G

↳ Preorder traversal

↳ Root
Left
Right

Storage of Data (Graph)
- Adjacency List
- Adjacency Matrix *



Adjacency List (Linked List)

- A → (1) B
- B → (3) C, A, D
- C → (1) B
- D → (1) B

→ O(E)

sum of Degree = 2E
= 6 = 2 * 3

V = n

Adjacency Matrix

- 0 → No edge
- 1 → edge

(4 x 4) → (V x V)

	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	0
D	0	1	0	0

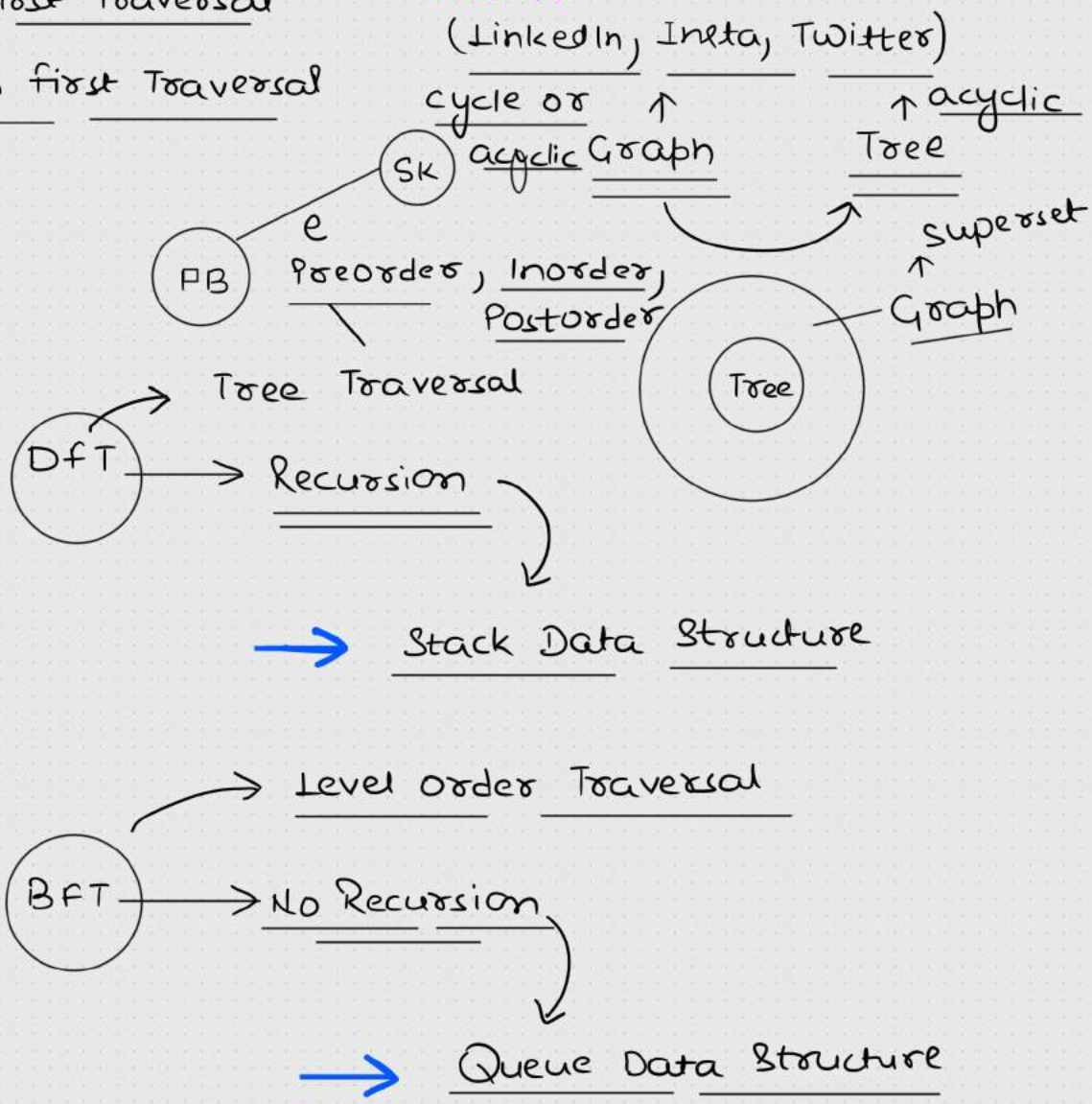
↳ $O(V^2) = O(n^2)$

Graph Traversal

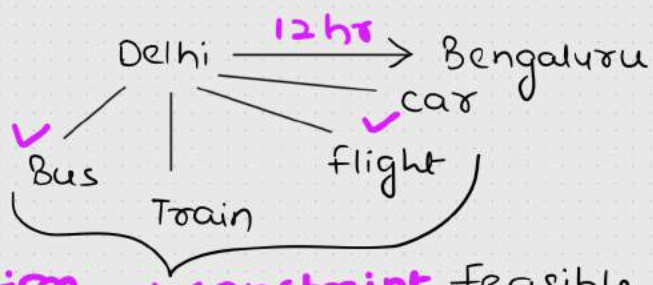
↳ Network

1) Depth first Traversal

2) Breadth first Traversal



Greedy Algorithms



Application 1) constraint feasible solutions

2) maximum profit

or

Minimum cost

↳ Optimization Problems

1) Greedy Approach

2) Dynamic Programming

1) fractional Knapsack

obj < Profit / weight

M = 37

min. cost

Bus

↳ optimized solution

(single optimized solution)

2) Job sequencing with Deadline

Job < Profit / Deadline

3) Huffman coding → Data compression technique

4) Optimal merge pattern

5) Minimum spanning tree < Kruskal / Prim's

6) Single source shortest Path

↳ Dijkstra's Algorithm

fractional Knapsack

	1	2	3	4	5	6	7
Profit	25	75	100	50	45	90	30
Weight	5	10	12	4	7	9	3

M = 37

n = 7

Profit maximum ← (objects profit)

constraint

> M — cannot pick those objects

Step 1

n	1	2	3	4	5	6	7
Profit/ weight	5	7.5	8.3	~~25~~ 12.5	6.4	10	10

— O(n)

Step 2

Sort Profit/Weight

n	4	6	7	3	2	5	1
Profit/ weight	~~25~~ 12.5	10	10	8.3	7.5	6.4	5

— O(nlogn)

Step -3

x (fraction)	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 9/10	0	0

M = 37

O(n)

Net Weight	Net Profit
37 - 4 = 33	50
33 - 9 = 24	50 + 90 = 140
24 - 3 = 21	140 + 30 = 170
21 - 12 = 9	170 + 100 = 270
9 - ~~10~~ * 9/10 = 0	270 + 9/10 * 75

Reducing weight & increasing Profit

= 337.5 → Net Profit

Time complexity

$$n + n\log n + n \approx O(n\log n)$$

fractional Knapsack

	1	2	3	4	5	6	7
Profit	25	75	100	50	45	90	30
Weight	5	10	12	4	7	9	3

M = 37

n = 7

Profit maximum ← (objects profit)

constraint

> M — cannot pick those objects

Step 1

n	1	2	3	4	5	6	7
Profit/ weight	5	7.5	8.3	25	6.4	10	10

— O(n)

Step 2

Sort Profit/Weight

n	4	6	7	3	2	5	1
Profit/ weight	25	10	10	8.3	7.5	6.4	5

— O(nlogn)

Step-3

x (fraction) ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~9/10~~ 0 0

M = 37

Net weight	Net Profit
37 - 4 = 33	50
33 - 9 = 24	50 + 90 = 140
24 - 3 = 21	140 + 30 = 170
21 - 12 = 9	170 + 100 = 270
9 - ~~10~~ * 9/~~10~~ = 0	270 + 9/10 * 75

= 337.5 → Net Profit

O(n)

Reducing weight & increasing Profit

Time complexity

$$n + n\log n + n \approx O(n\log n)$$

Huffman Coding

Input → character ↦ freq

- a = 5
- b = 9
- c = 12
- d = 13
- e = 16
- f = 45

zip
↳ Data compression

1 Byte — 8 bits

Pre-requisite
↳ Heap
Data structure

$5*8 + 9*8 + 12*8 + 13*8 + 16*8 + 45*8$

$\Rightarrow 40 + 72 + 96 + 104 + 128 + 360$

$\Rightarrow$ 800 bits

Psuedocode →
- More frequency → Lesser bit
- Low frequency → Larger bit

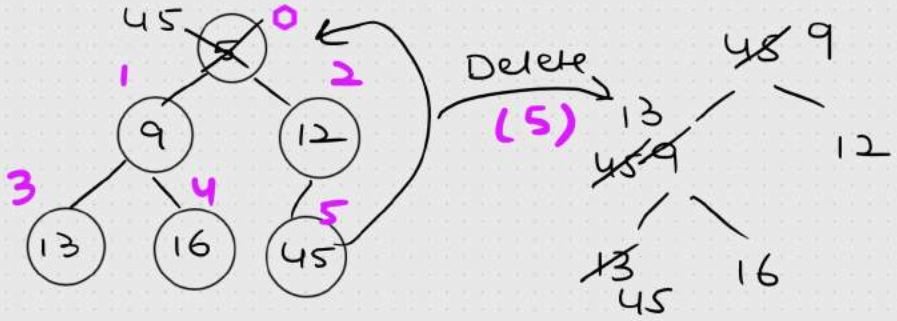
Huffman Tree

1) Build heap ——— O(n)
↳ Minheap
↳ Mathematical Derivation

2) Pop 2 times & insert the addition of two deleted elements → Minheap
— 2 Del
— 1 Insertion
$(n-1)\ 3 \log n$

Deletion ——— O(log n)
↳ Minheap
↳ smallest element

Insertion ——— O(log n)



(13, 45, 16, 13, 14)

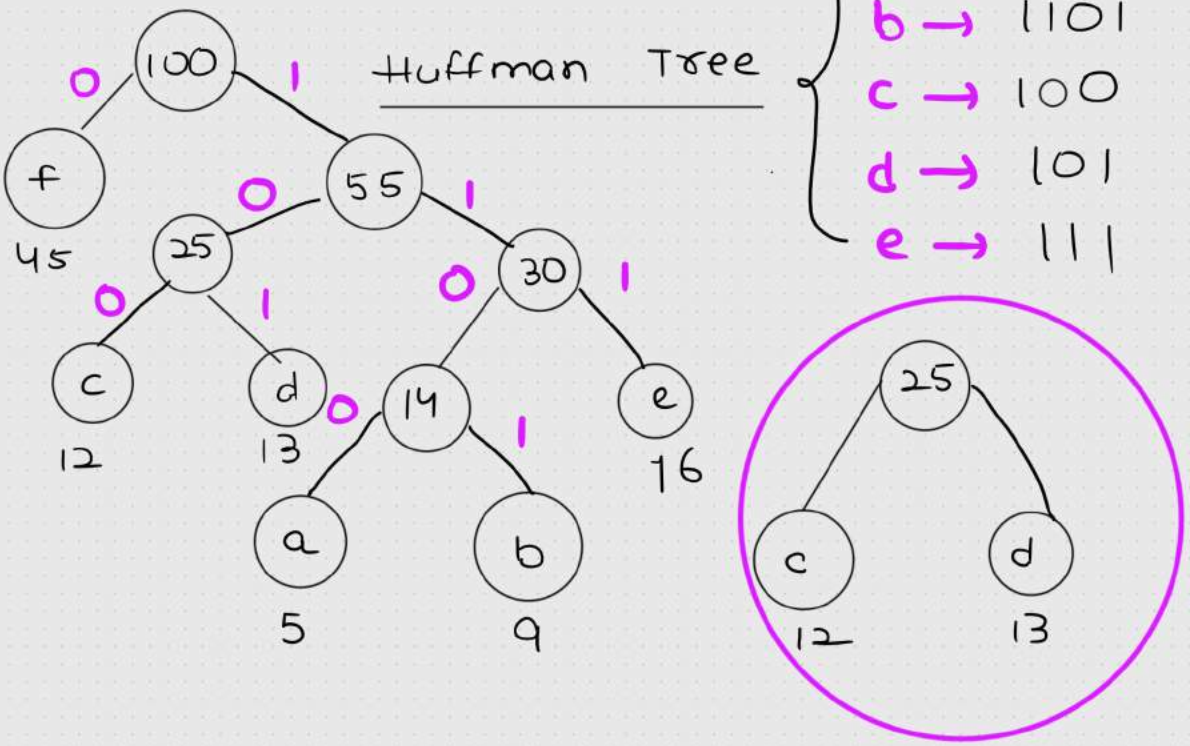
25, 45, 16, 14, 30, 55

(9) Delete

Output

f → 0
a → 1100
b → 1101
c → 100
d → 101
e → 111

Huffman Tree



Time complexity

$\Rightarrow n + (n-1) * 3 \log n$

$\Rightarrow O(n \log n)$

Python

heapq

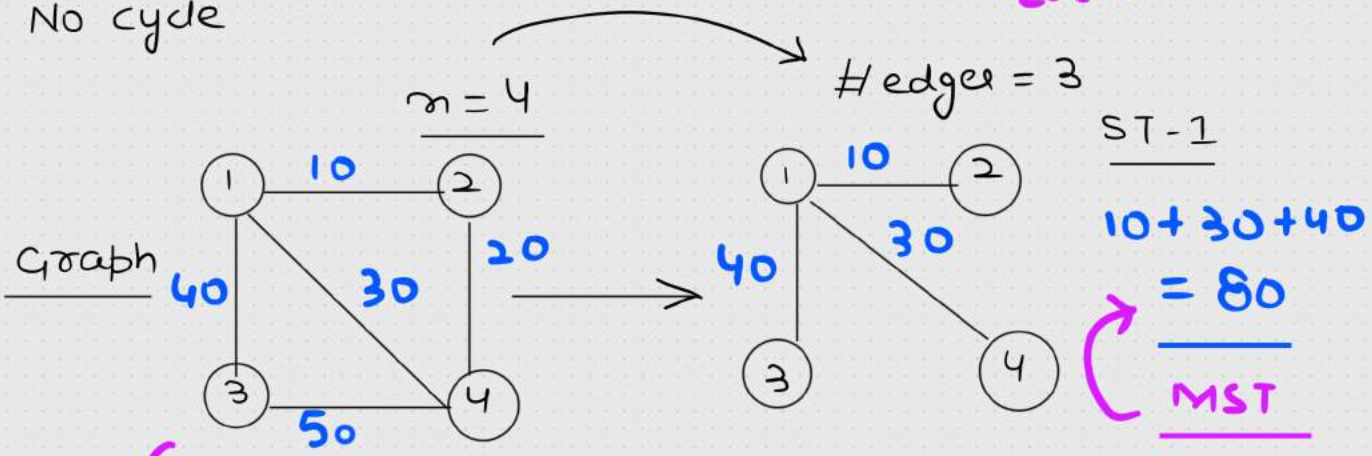
- heappush (Insertion)
- heappop (Deletion)

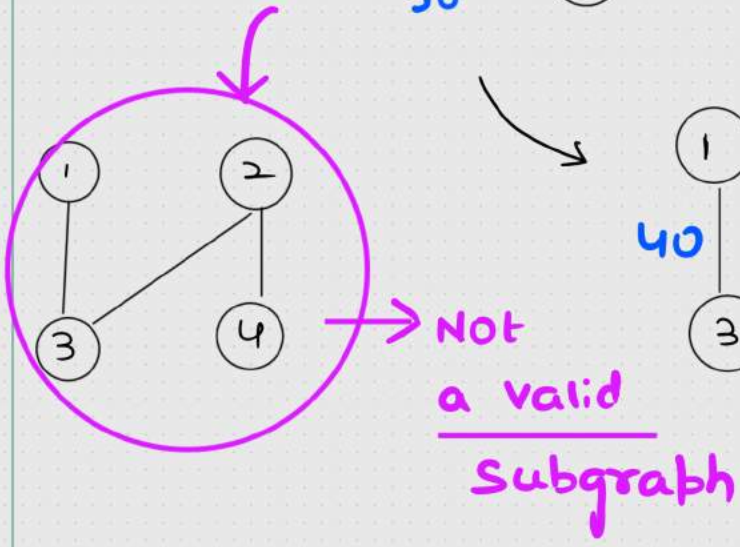
Spanning Tree

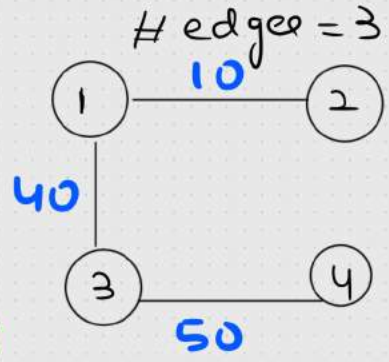
Subgraph of graph G → Minimum Spanning Tree

1) all the vertices → available
2) $(n-1)$ edges to connect
   ↳ n = # vertices
3) No cycle

Important properties of spanning tree







Properties

Complete graph → $ST(n) = n^{n-2}$

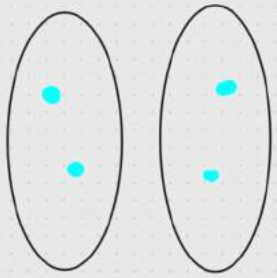
MST → Greedy Algorithms

(Minima) → **optimizations**

Algorithms
1) Kruskal Algorithm
2) Prim's Algorithm

⟹ Heap Data Structure

Applications → 1) Image Segmentation

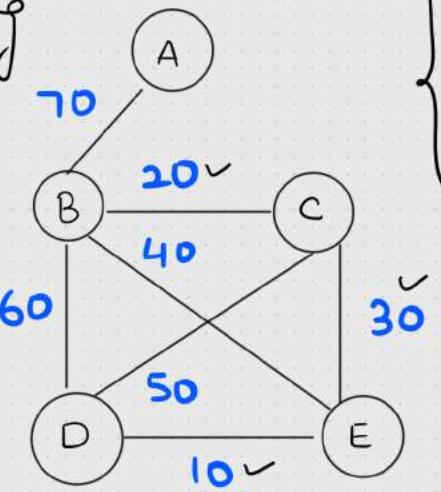


2) compute networks

Minimum Spanning Tree

Kruskal Algorithm

Logical understanding



AB — 70, CE — 30
BC — 20, BE — 40
BD — 60, CD — 50
DE — 10

cost = # edges

Minheap

(10, 50, 30, 60, 40, 20, 70)

Worst case scenario

1) Min. cost of an edge

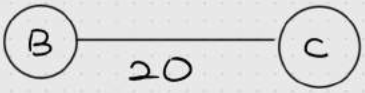
Build Minheap = O(E)

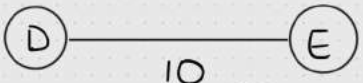
+

Delete the item

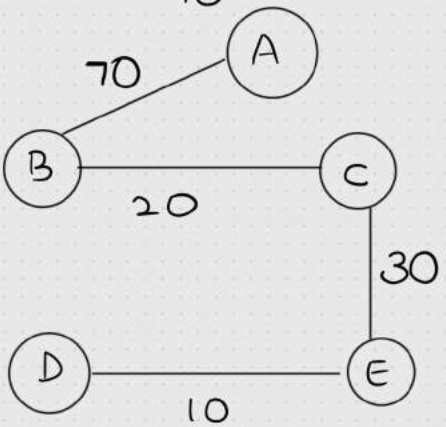
↳ minimum element

2





3



Worst case scenario → O(E log E)

Total cost (MST) =

70 + 20 + 30 + 10

⇒ 130

n = 5 ⟷ e = 4

Best case scenario :- $(V-1) \cdot \log E$

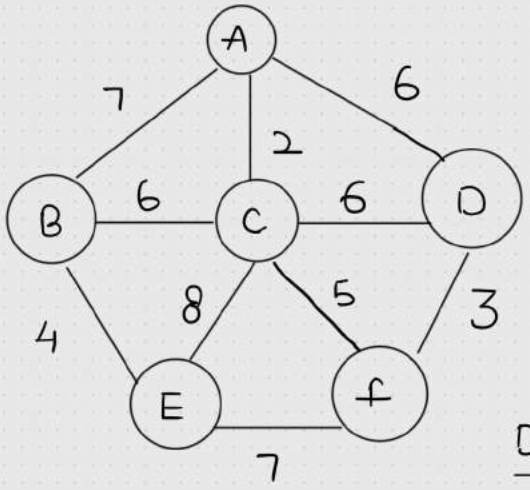
$O(V \log E)$

$O(V \log V)$

heapq → Documentation in Python

Prim's Algorithm → MST

↳ Adjacent vertex



Build minheap

↳ O(v)

Decrease key operation

↳ log E

A	B	C	D	E	f	
0	∞	∞	∞	∞	∞	
N	N	N	N	N	N	
	7	2	6	∞	∞	→ 3+ 3 log E
	A	A	A	N	N	
	6		6	8	5	→ 5+ 4 log E
	C		C	C	C	
	6		3	7		→
	C		f	f		
	6			7		→
	C			f		
				4		→
				B		

log V

log V

log V

log V

log V

2E

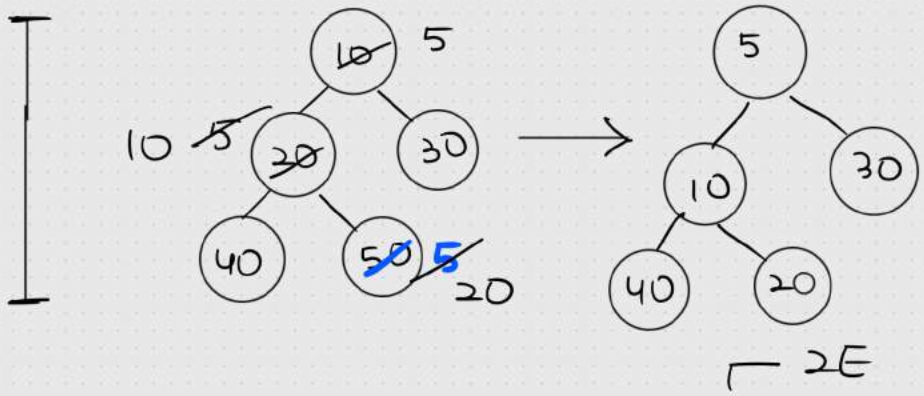
↑ adjacency list

adjacent + E log E

Time complexity → V + V log V + 2E + E log E

⇒ (V + E) log V

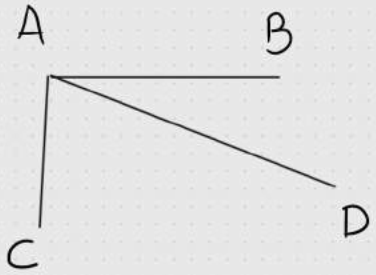
$O(\log n)$



$\vdash 2E$

Graph
- Adjacency List
- Adjacency Matrix

$\hookrightarrow V^2$

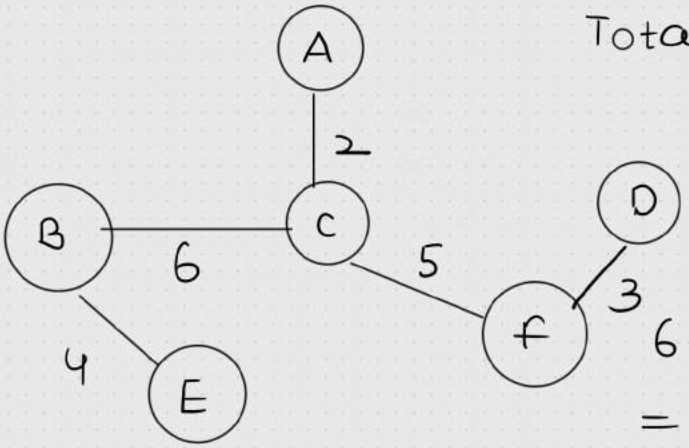


	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	0
D	1	0	0	0

VxV

4x4

Sum of degree $= 2E$



Total cost of MST using Prim's algo =

$6 + 2 + 5 + 3 + 4$

$= 20$

Adjacency Matrix

$$TC = V + V\log V + V^2 + E\log E$$

$$\boxed{TC = V^2 + E\log E}$$

- $V \rightarrow$ Build of minheap
- $V\log V \rightarrow$ Deletion of all the vertex
- $V^2/2E \rightarrow$ adjacent
- $E\log E \rightarrow$ Decrease key operation

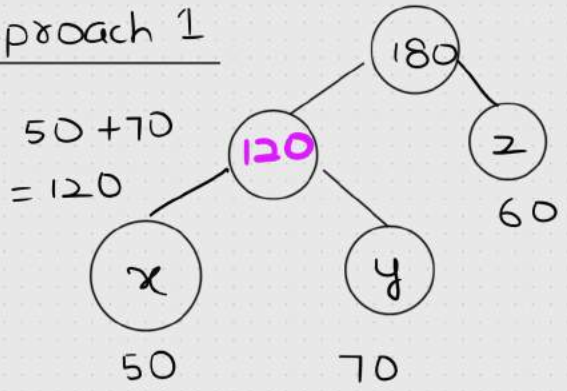
Optimal Merge Pattern

Problem Statement

$n = 3$
$x = 50$
$y = 70$
$z = 60$

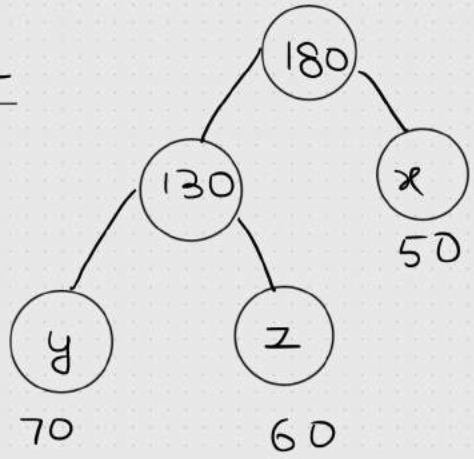
⇒ Objective :- Minimize the computational cost of merging all the given files.

Approach 1



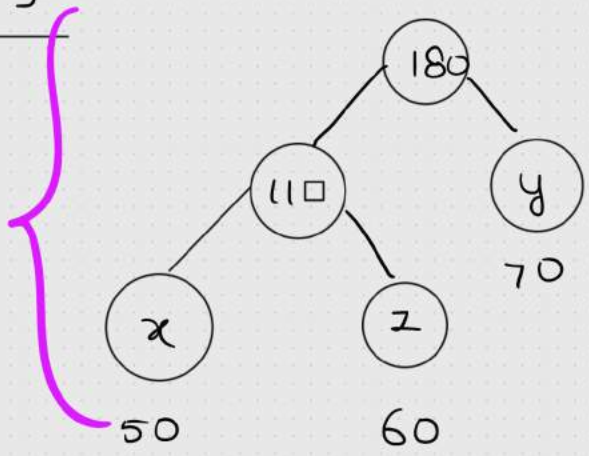
120 + 180 = 300

Approach 2



130 + 180 = 310

Approach 3



180 + 110 = 290

Approach 3

↳ Min. computational cost

Approach → Huffman coding

1) Build heap —— $O(n)$

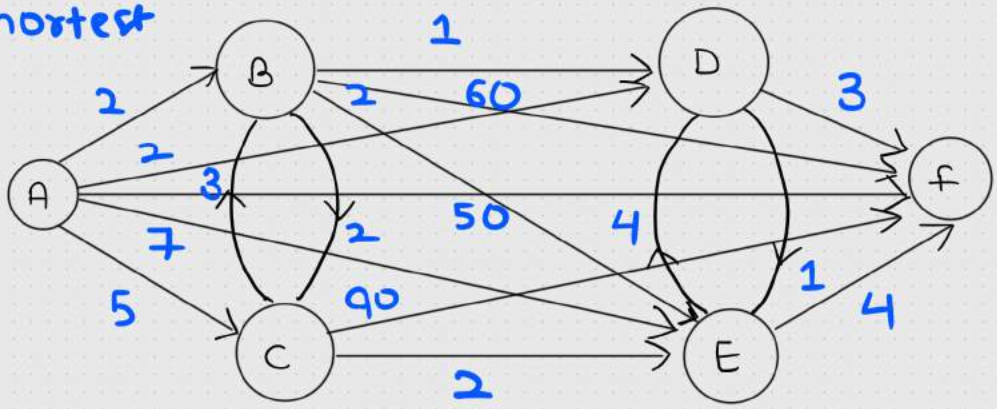
2) Pop 2 elements —— $(n-1)\,3\log n$

↳ addition of those 2 poped elements & insertion in the minheap

Time complexity $= O(n \log n)$

Dijkstra's Algorithm

4 single source shortest path



Source = A

Distances — **Degree**

A: {B:2, C:5, D:2, E:7, f:50} (5)

B: {C:2, D:1, E:2, f:60} (4)

C: {B:3, E:2, f:90} (3)

D: {E:1, f:3} (2)

E: {D:4, f:4} (2)

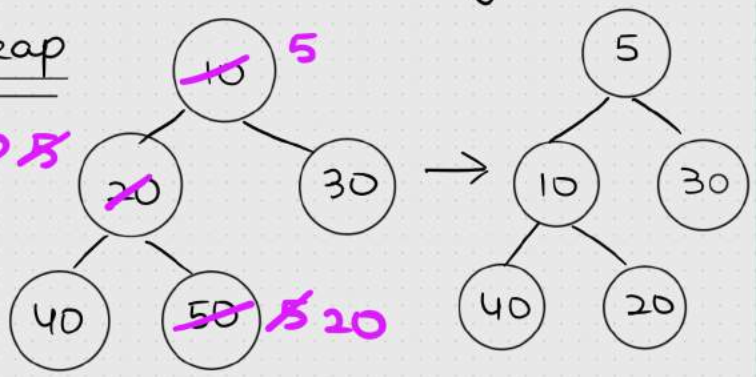
f: {}

+ve edge weights

$\Theta(\log n)$

Minheap

Decrease Key



Adjacency List

Dijkstra's Algorithm

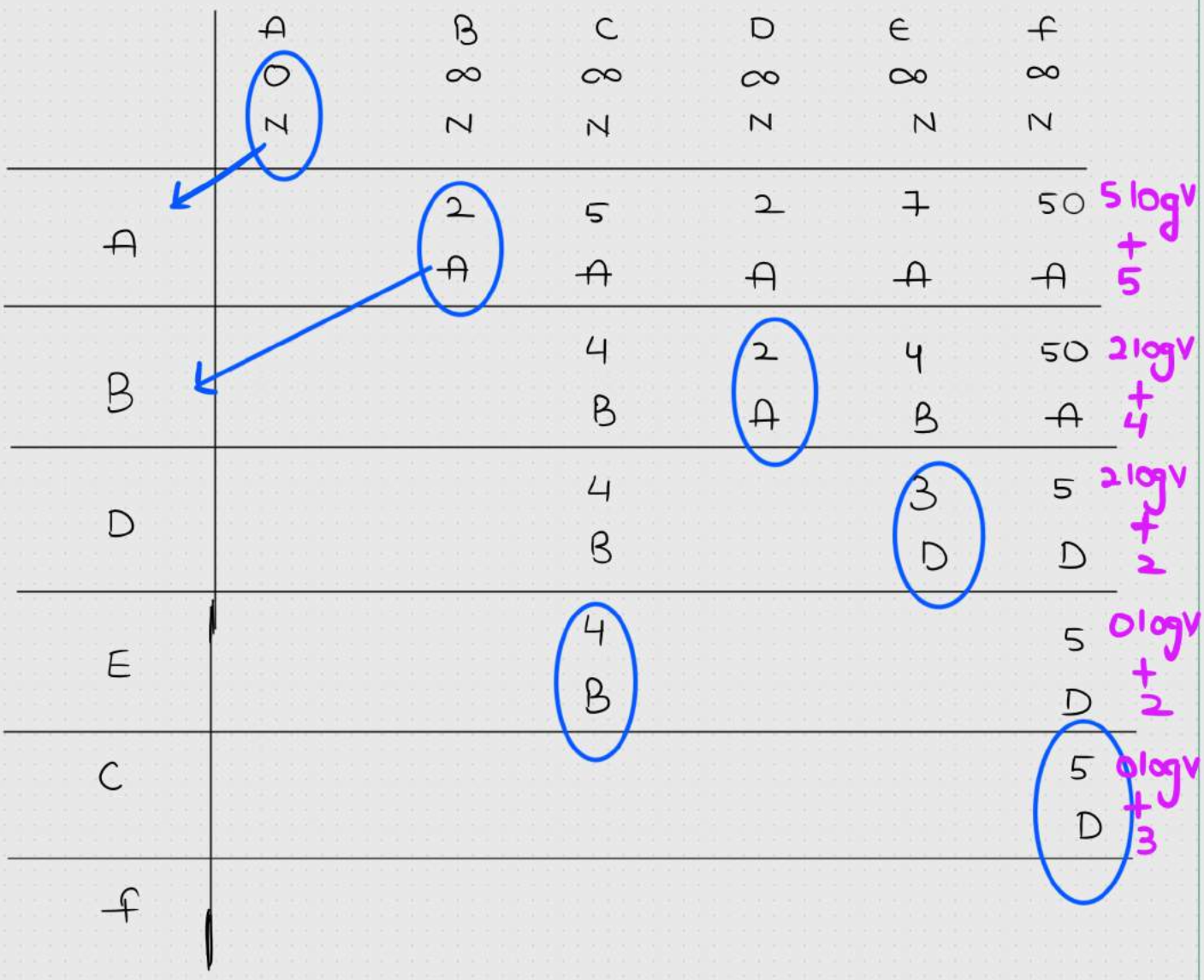
Time complexity ⇒ $V + V\log V + E\log V + 2E$

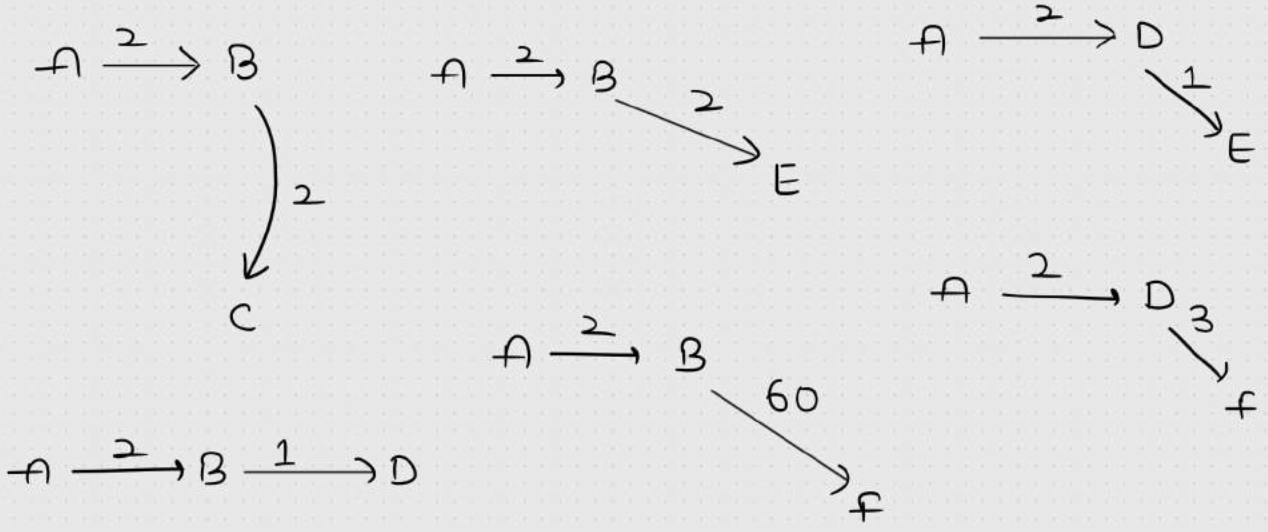
Build minheap — Pop — Decrease key — sum of degree

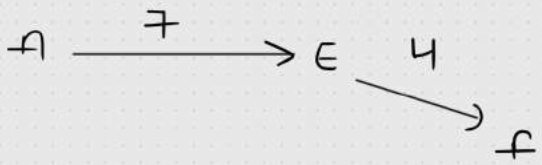
$\Rightarrow \Theta((V+E)\log V)$

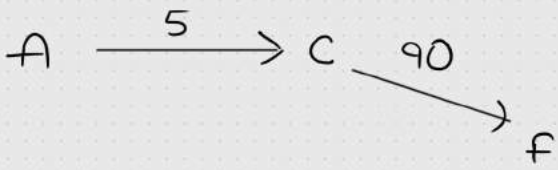
Pop (Minheap) = $\Theta(\log n)$ — Pop elements from minheap

Source = A


A B C D E f
0 ∞ ∞ ∞ ∞ ∞
N N N N N N
A
2 5 2 7 50
A A A A A
B
4 2 4 50
B A B A
D
4 3 5
B D D
E
4 5
B D
C
5
D
f
5 log V + 5
2 log V + 4
2 log V + 2
0 log V + 2
0 log V + 3


A 2 B 2 C
A 2 B 2 E
A 2 D 1 E
A 2 B 1 D
A 2 B 60 f
A 2 D 3 f





Time complexity Analysis

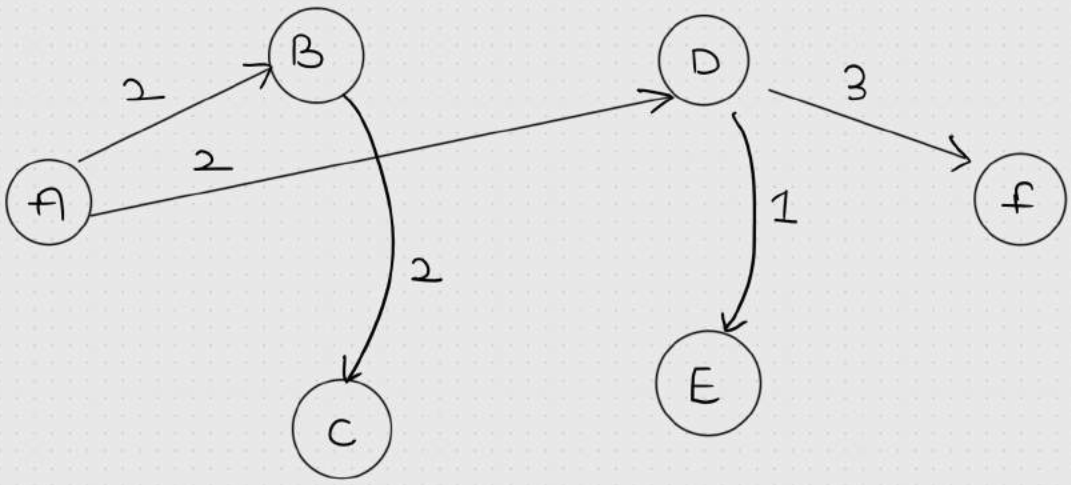
Adjacency Matrix → $V + V\log V + E\log V + V^2$ ← Degree

$\Rightarrow O(V^2)$

Result

A - B = 2	A - D = 2	A - F = 5
A - C = 4	A - E = 3	A - A = 0

final output



Limitation

↳ -ve edge weight

↳ Dijkstra's Algorithm is not working well

⇓

Dynamic Programming ← Explore all the possible path

↳ More time to give result

↳ Bellman ford Algorithm

Dynamic Programming

↳ Optimization problems
↳ (minima & maxima)

↳ Overlapping subproblems

T(n) ↑

1 2 3 4 5 **fib. series**
1, 1, 2, 3, 5, 8, - - - - - -

fib(n):

c { n==1 or n==2
result=1

Otherwise → n>2

T(n-1) + T(n-2) { result = fib(n-1) + fib(n-2)
return result

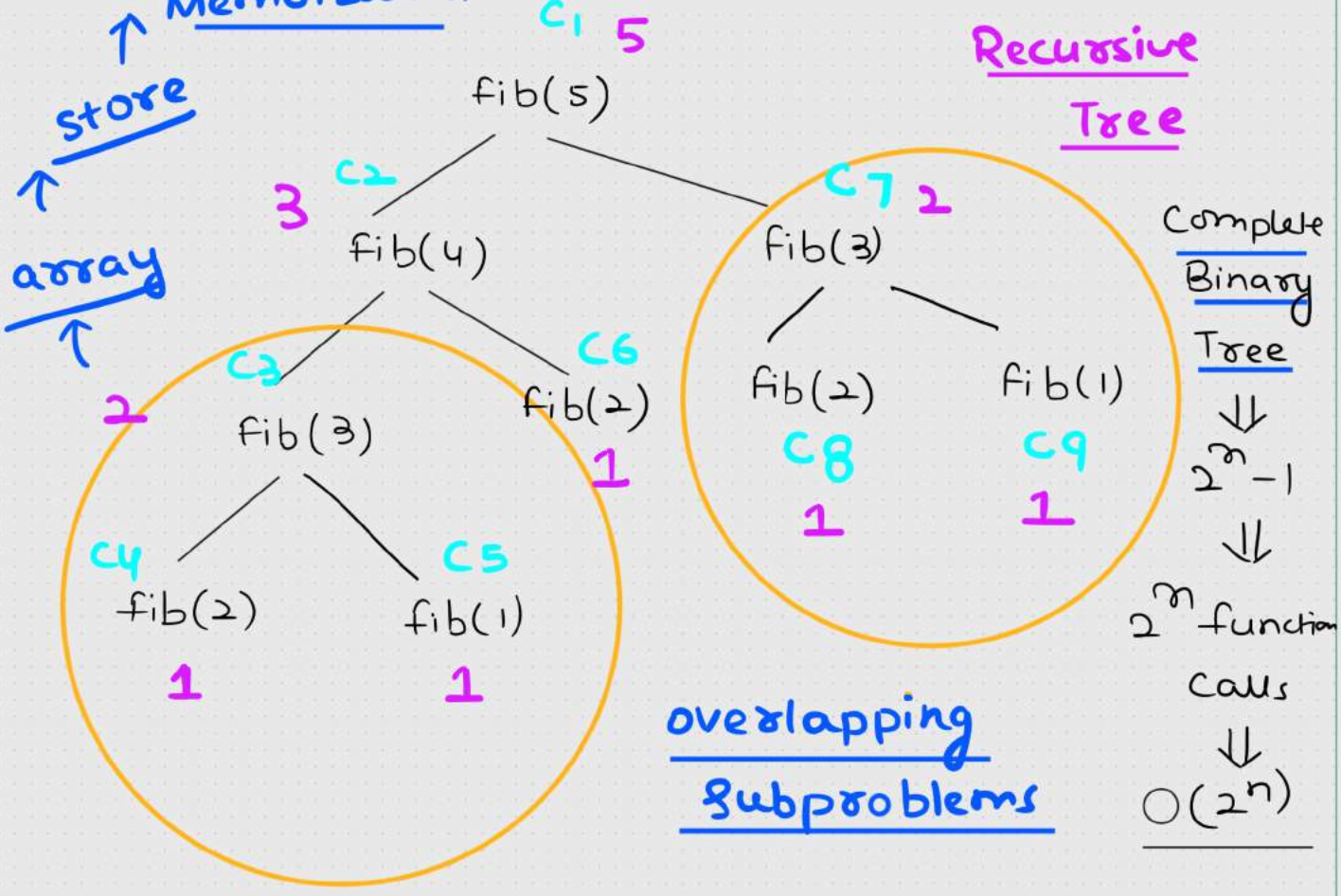
↳ Recursion

n = 4
↳
output = 3

↑ **Memoization**

store

array



Recursive Tree

Complete Binary Tree
⇓
$2^n - 1$
⇓
2^n function calls
⇓
$O(2^n)$

overlapping subproblems

Note: $k = 2^n - 1$ (CBT)

1) Recursion (store the results)
↳ overlapping subproblems
(Memoization)

Merge Sort
↳ No overlapping Subproblem
↳ Divide & conquer

↳ Top Down Approach

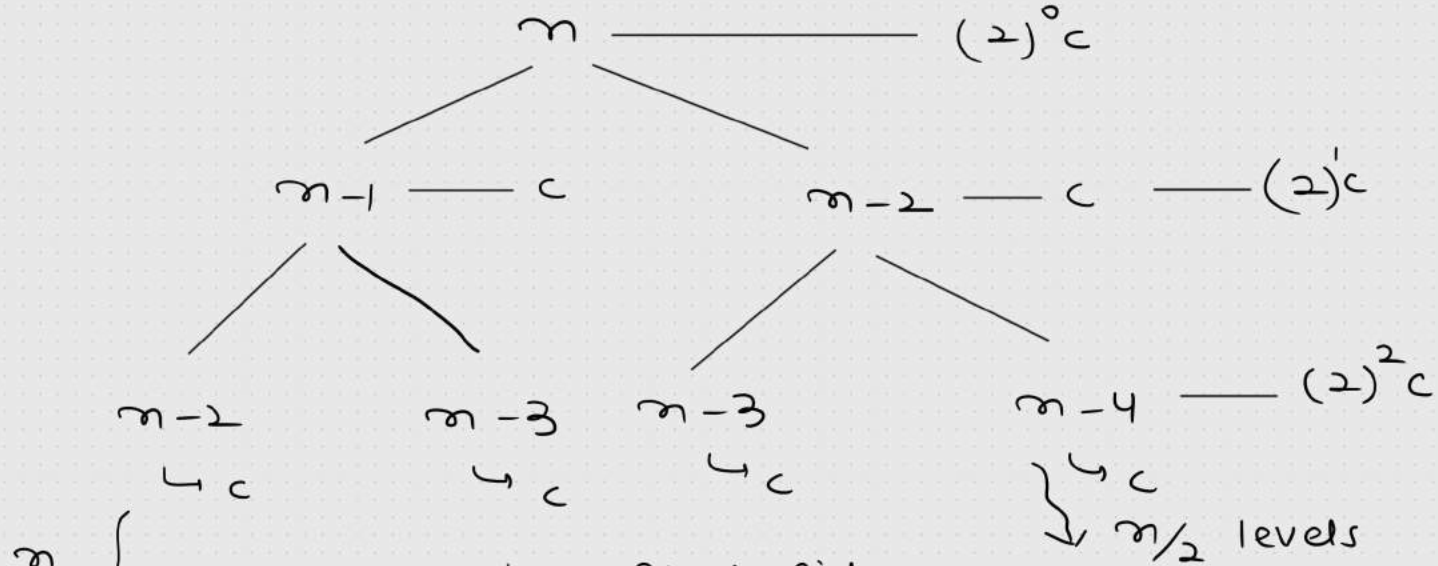
2) Tabulation (Bottom up Approach)
↳ Not Recursion

1	2	3	4	5	6
1	1	2	3	5	8

Recurrence Relation :- (Recursion)

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

Recursive Tree Approach



Left side

$n - k = 1$

$k = n - 1$

Right side

$n - 2k = 1$

$\frac{n-1}{2} = k$

$c(2^0 + 2^1 + 2^2 + \dots + 2^k)$ $k = n$ (max)

$c(2^0 + 2^1 + 2^2 + \dots + 2^n)$

⇓

GP series

$r = 2$ $r > 1$

$$S = \frac{a(r^n - 1)}{r - 1} = \frac{2^n - 1}{1}$$

$\Rightarrow O(2^n)$

↳ Exponential Time complexity

$O(n)$

fib(5)
↓
fib(4)
↓
fib(3)
↓
fib(2)
↓
fib(1)

(Recursion)
Exponential Time complexity
⇓
Linear Time complexity
(Dynamic Programming)

0-1 Knapsack

1 2 3 4

Profit = [1, 2, 5, 6]

weight = [2, 3, 4, 5]

M = 8

0 — not choosing

1 — choosing the object

1 2 3 4

[0/1 0/1 0/1 0/1]

$\Theta(2^n)$ ⇒ Exponential Time complexity

Tabulation approach

			w → 0	1	2	3	4	5	6	7	8
P	w										
0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
n 6	5	4	0	0	1	2	5	6	6	7	8

x_1	x_2	x_3	x_4
0	1	0	1

8 − 6 = 2

Important

w >= wt(i) — case (2)

Profit(i, w) = value(i, w) =

$$\begin{cases} \max\left(V(i-1, w),\ V(i-1, w - wt(i) + p(i))\right) \\ \max(5, 6) = 6 \end{cases}$$

$\omega < wt(i) \rightarrow$ case (1)

↳ $v(i-1, \omega)$

CNF Satisfiability → Base Problem

$x_i = \{x_1, x_2, x_3\}$

$\Rightarrow (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

↳ NP-Hard

CNF Satisfiability α 0/1 knapsack

↳ $O(2^n)$ → exponential

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

→ 2^3 combinations

n variables

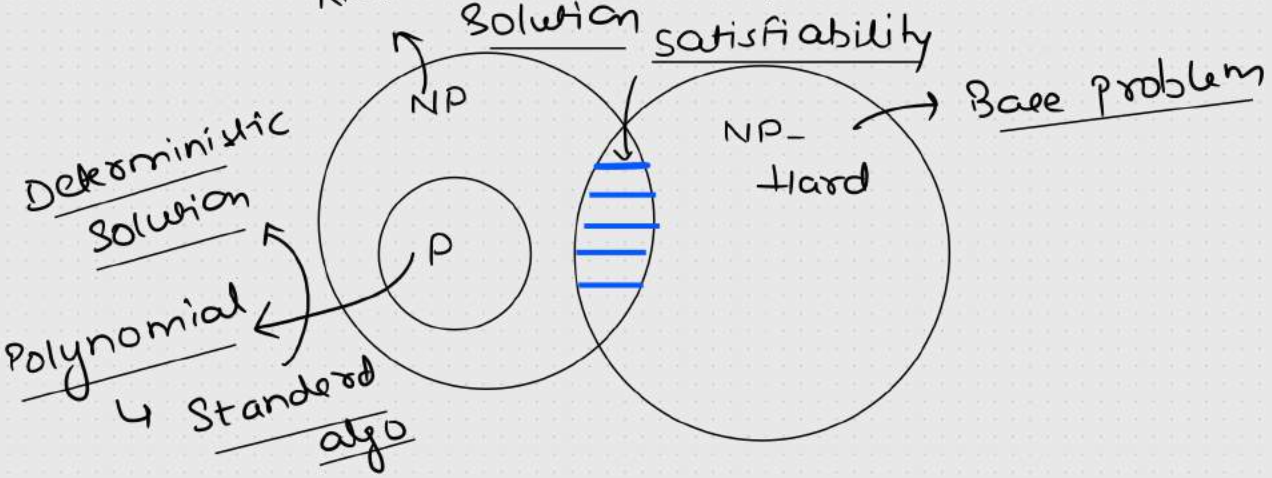
↳ 2^n combinations

2^n → Exponential

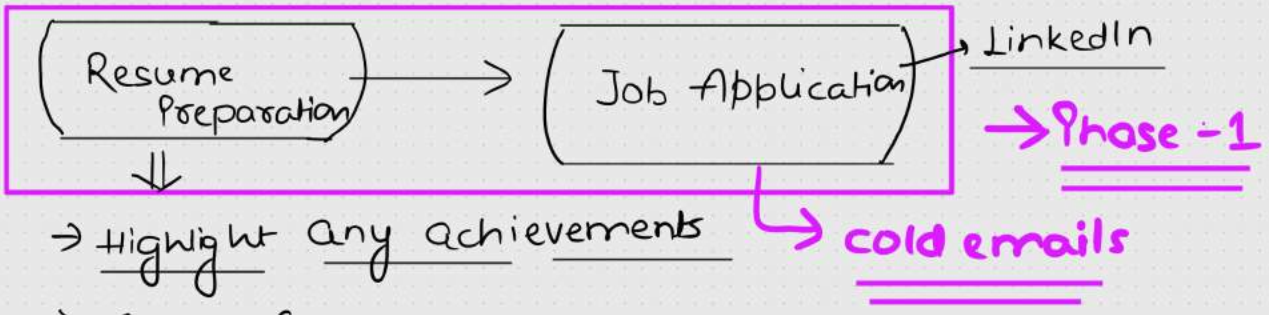
$2^3 = 8$

n	1	2	3
Profit	—	—	—
Weight	—	—	—
x_i	0/1	0/1	0/1

Non-determination → NP-Complete



FAANG Interviews Roadmap



→ Highlight any achievements

→ Overleaf

→ Research prior for the company's demand

→ fake information

Phase-2