Array

Continous memory

20	21	22	23	24	25	26	27
0	1	2	3	4	5	6	7

Positions →

Properties → Random Access

We can access element randomly

When to use array ?

→ Searching

Python has concept of dynamic array

→ No need to define size

Address of 1d array

2	4	6	8	10	12	14
0	1	2	3	4	5	6

index →

datatype ? → int → 4 bytes → Size

1000 → memory address → Base Address

Now what is address of arr[6] ?

formula → Base address + (ind - lower bound) * size

1000 + (6 - 0) * 4 = 1024

2-d Array

Compose of rows and columns*

Columns ↓

Rows	0	1	2
0	2	4	6
1	8	10	12
2	14	17	(20)

matrix

(20) → arr[2][2]

We can store element in two ways → Row / Column

Address of 2d array

	0	1	2	3	4
0	10	20	30	40	50
1	60	70	80	90	100
2	110	120	130	140	150
3	160	170	180	190	(200)

(200) → arr[3][4]

formula → Base add + [(i - lb) * Nc + (j - lb)] * size → Row major form

Base add + [(j - lb) * Nr + (i - lb)] * size → Column major form

Linear Search

20	40	60	80	100	120

→ Array n = 6 x = 80 → target

Steps →
- Traversing
- Matching element with target
- If match then return index i
- Otherwise return -1

worst case

Time Complexity → O(n)
↓
Traversing through all the elements

Best case

Time Complexity → O(1)
↓
element present at initial position

Binary Search

→ divide & conquer

i=0 j=7

2	4	6	8	10	12	14	16

→ Sorted array n = 8 x = 14 → target

Small problem → Single element

Big Problem → More than one element

Steps →
- find mid value → i + (j-i)//2 → 3
- Mid element == x → return mid index
- Mid element < x → Binary(arr, mid+1, j, x) → Recursion
  - or
  - i = mid+1
- Mid element > x → Binary(arr, i, mid-1, x)
  - or
  - j = mid-1
- return -1

Search 2d matrix

Row wise sorted → left to right

First integer of each row > last integer of previous row

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

3 x 4

m → rows 0 → 3

n → column → 4

target = 3

Brute force approach

traverse through all the rows and column

time complexity ⟶ O(m x n)

Optimized approach

Arrange above matrix in row major form

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

⟶ mid value at index 5

Now we

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

row = mid / nc = 5/4 = 1*

column = mid / nc = 5 / 4 = 1*

Now we know that mid > target ⟶ 11 > 3

left = 0

right = mid - 1

1	3	5	7	10

(0+4) // 2 = 2 row = 2//4 = 0 column = 2 / 4 = 2

Now the time complexity ⟶ O(log mn)

Ternary Search*

1	2	3	4	5	6	7	8	9	10

mid1 mid2

⟶ sorted array

Steps ⟶

mid1 = l + (r-l) // 3 mid2 = r - (r-l) // 3 ⟶ dividing array in three equal parts

mid1 = 0 + 9 // 3 = 3 mid2 = 9 - 9 // 3 = 6

target = 5 ⟶ Searching element

arr[mid1] == x ⟶ 9 ≠ 5 ⟶ false

⟶ if true ⟶ return mid1

arr[mid2] == x ⟶ return mid2

x < arr[mid1] ⟶ 5 < 4 ⟶ false

right = mid1 - 1

x > arr[mid2] ⟶ 5 < 7 ⟶ true

left = mid2 + 1

else

mid1 , mid2 - 1

Sorting Algorithm

20	10	12	17	27	42	39

Sorting algorithm

- Comparison based
- Non-comparison based

Comparison within elements

- Selection Sort
- Bubble Sort
- Insertion Sort
- Quick Sort
- merge sort
- heap sort

→ divide & Conquer

Stable and non-stable Sorting

Stable	non stable
Relative Order Should be maintained	Quick Sort
dict → (4,2) (5,7) (4,3) (6,8) (7,10)	Selection Sort
↳ Hashing data structure	heap Sort

After Sorting (4,3)

(4,2) (5,7) (6,8) (7,10) → Relative order maintained → stable sorting

OR

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

merge sort → application of d & c

Inplace and outplace sorting algorithm

↳ not using any extra place

Comparison - Bubble Sort

70	20	50	30	90	5	15

point to be noted:

if n = 7 → than we need n-1 steps to sort array

Comparison ?

$(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 \longrightarrow \frac{n(n-1)}{2}$

Sum of n natural number $\frac{n(n+1)}{2}$

Time Complexity → $O(n^2)$

Steps →

70 > 20

> 20 70

70 > 50

> 20 50 70

70 > 30

50 > 30

> 20 30 50 70

70 > 90

> 20 30 50 70 90

90 > 5 50 > 5 20 > 5

70 > 5 30 > 5

> 5 20 30 50 70 90

90 > 15 50 > 15 30 > 15 5 < 15

70 > 15 90 > 15 20 > 15

> 5 15 20 30 50 70 90

Comparison - Selection Sort

50	38	45	79	19	27	29

Concept → Take minimum index

Steps ① min → 0 ② min = 1 ③ min = 2 ④ min = 3 ⑤ min = 4 ⑥ min = 5

50 > 38 → Change min → 1

38 < 45, 38 < 79 → 38 < 50, 38 > 27 → Change min → 5

38 > 19 → Change min → 4

19, 38, 45, 79, 50, 27, 29

38 < 45, 38 < 79 → Change min → 5

38 > 29 → Change min → 6

19, 27, 29, 79, 50, 38, 45

45 < 79, 45 < 50 → 45 > 38 → Change min → 5

50 > 38 → Change min → 5

19, 27, 29, 38, 50, 79, 45

79 > 50 → Change min → 4

50 < 79, 50 > 45 → Change min → 6

19, 27, 38, 45, 79, 50

79 > 50 → 19, 27, 29, 38, 45, 50, 79 → final output

Note → At every pass → one swap required

Time Complexity → # Comparison → $\frac{n(n-1)}{2} = O(n^2)$

Swap → $(n-1)^{th}$ Swap → $O(n)$

Time Complexity → $O(n^2)$

Comparison - Insertion sort

75	90	100	95	85	50	100	110	7

Steps →

75 90

75 90 100

75 90 100 95 → 95 00

100 > 95

75 90 95 100 85

100 > 85, 95 > 85, 90 > 85

75 85 90 95 100 50

100 > 50, 95 > 50, 90 > 50, 85 > 50, 75 > 50

50 75 85 90 95 100 110 7

100 > 7, 95 > 7, 85 > 7, 75 > 7, 50 > 7

7 50 75 85 90 95 100 110 → final output

Time Complexity → # Comparison → $1 + 2 + 3 + \dots + n-1$

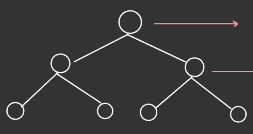
$\frac{n(n-1)}{2} = O(n^2)$

Swaps → $O(n^2)$

Overall time Complexity → $O(n^2)$

Heap data structure*

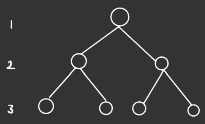
→ Complete binary tree



Binary tree ⟶ # Child nodes 0 1 2

Almost node 2

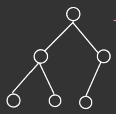
Full binary tree ⟶ Every node has 2 child apart from leaf nodes



Complete binary tree ⟶ After completion of first level then only move towards the next level

After completion of left side node then only go for right side node


Not CBT


Valid CBT

Properties of full binary tree

> nodes ⟶ n level ⟶ K

K=3 , n=?

$n = 2^K - 1 \longrightarrow 2^3 - 1 = 7$

Now n = 63 , K = ?

$n = 2^K - 1$

$2^K = n + 1$

$K = \log_2(n+1) \longrightarrow \log_2(63+1) \longrightarrow \log_2 2^6$

$K = 6$

> $\frac{n}{2}$ ⟶ # Number of leaf nodes

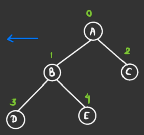
let assume n = 7

$\frac{7}{2} \longrightarrow 4$ leaf node

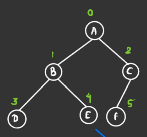
$7 - 4 \longrightarrow$ non leaf node

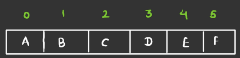
Heap — minheap, maxheap (P tree)

left index = 2*i+1

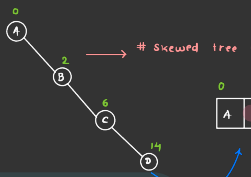


Right index = 2*i+2





Arranging tree in array



Skewed tree not valid CBT

Arranging tree in array

lots of wastage of space

less wastage of space ⟶ Almost Complete binary tree

Minheap *

parent node < child node

Root node ⟶ Contains smallest element

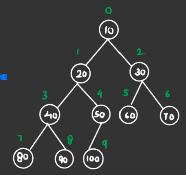
Maxheap *

parent node > child node

Root node ⟶ Contains largest element

Insertion in minheap

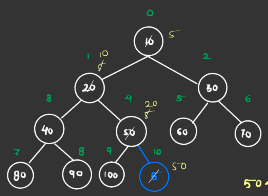
Complete binary tree



Arrange in array

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

let suppose want to insert 5 in minheap



According condition not valid minheap

50 < 5 ⟶ false

swap

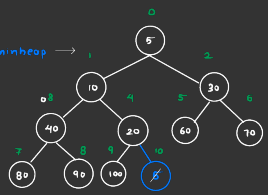
20 < 5 ⟶ false

swap

10 < 5 ⟶ false

swap

After inserting minheap ⟶



$n = 2^k - 1$ ⟶ Complete binary tree

$n+1 = 2^k$

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

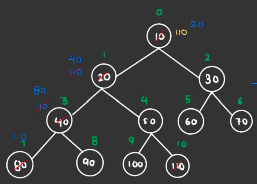
$k = \log_2(n+1)$ ⟶ # levels

Comparison in minheap ⟶ $\log n$

Swap ⟶ $\log n$

Time Complexity ⟶ $O(\log n)$ ⟶ for insertion

Deletion in min or maxheap



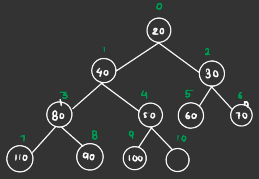
Valid minheap

parent < child

delete ⟶ 10

deletion always start from min index

and last element 110 will place at 0th position



Minheap after deletion

Comparison

110 > 20 and 30 ⟶ but will take smallest one

swap

110 > 40 and 50

swap

110 > 80 and 90

swap

deletion 1st time ⟶ 1st smallest element

deletion 2nd time ⟶ 2nd smallest element

deletion nth time ⟶ nth smallest element

Heapsort ⟶ Comparison based Sorting

⟶ # increasing order sorted array

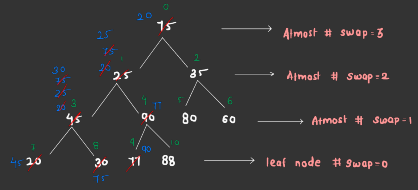
maxheap ⟶ decreasing order array

Overall time Complexity ⟶ O(n logn) ⟶ HeapSort

Build Heap

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

If build minheap ⟶ At leaf node swapping required



Almost # swap = 3

Almost # swap = 2

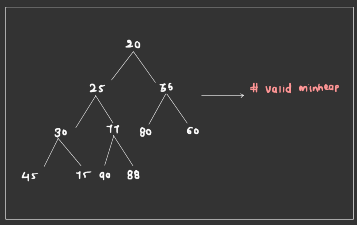
Almost # swap = 1

leaf node # swap = 0

Steps

Create Complete binary tree

Swap element for creating min or max heap.



valid minheap

Time Complexity ⟶ O(n)

Comparison Heapsort

1) Build heap ⟶ minheap or maxheap O(n)

2) delete all the elements and store deleted element O(nlogn)

Overall time Complexity ⟶ O(n) + O(nlogn)

O(nlogn)

Practice Question

[1, 1, 1, 2, 2, 3] ⟶ K=2 ⟶ Top 2 frequent element?

1 ⟶ 3 # frequencies

2 ⟶ 2

3 ⟶ 1

key, value

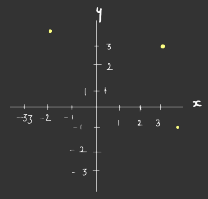
Excepted output ⟶ [1, 2]

Counter in python ⟶ Count the frequency

Coding ⟶

heapq.nlargest(K, Count.keys())

Points ⟶ [3,3 , [5,-1 , -2,4] K = 2 ⟶ K closest points] # Output ⟶ [3,3 , -2,4]



origin ⟶ (0,0)

distance measure ⟶ Euclidean distance

$P_1(x_1,y_1)$ —— $P_2(x_2,y_2)$

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

In our question we have measure from origin ⟶ $(x_1,y_1) = 0$

$\sqrt{x_2^2 - y_2^2}$

Create minheap ⟶ (distance, point)
→ # final output

Applying ⟶ delete operation on minheap ⟶ # K times
→ Top K minimum distances

Recursion

, functions calls itself directly or indirectly

Example ⟶ Factorial

5! = 5*4*3*2*1 = 120

$n! = n * (n-1)!$ ⟶ # formula

Base Condition ⟶ Code will terminate at condition.

fact(n): # pseudo code

if n=0 or n=1 ⟶ Return 1 # Small problem (Base Condition)

else n>1 ⟶ Recursive Call ⟶ Return n * fact(n-1)

Return result

Recurrence Relation

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

$T(n) = T(n-1) + c$ ⟶ # Substitution method.

put ⟶ n = n-1

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

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

$= T(n-3) + 3c$

↓ # K times ⟶ n-K=1 K = n-1

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

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

$= T(1) + nc - c$

$T(n) = O(n)$ ⟶ # Time Complexity

Fibonacci Series

0 1 1 2 3 5 8 13 21 34 ⟶ # series

0 1 2 3 4 5 6 7 8 9 n=10

pseudo code

if n<=1 ⟶ return n # Base condition

else ⟶ return fib(n-1) + fib(n-2)

Count number of ways to reach upstairs

One or two steps at a time

steps ⟶ n=1 ⟶ only one way to reach upstair

n=2 ⟶ Two ways -# (1,1) (2)

n=3 ⟶ Three ways # (1,2), (1,1,1), (2,1)

n=4 ⟶ five ways # (1,1,1,1), (2,2), (1,2,1), (2,1,1), (1,1,2)

⟶ # fibonacci Series

pseudo Code

Helper method
- if n<=1 ⟶ Return n
- else ⟶ Return Cnw(n-1) + Cnw(n-2)

Another function for counting ways to reach

Return helper(S+1)

final Output

when n ⟶ very large ⟶ we can optimized time complexity using dp

Time Complexity ⟶ $O(2^n)$ # Exponential time complexity

Divide & Conquer

Big problems ⟶ divide and conquer

Small problems ⟶ Return Solution

Recursion ⟶ function calling itself

Recursive tree

Recurrence relation

1) Substitution

2) Recursive

3) Master theorem

Pseudo Code

divideAndconquer(arr, P, Q)

if small ⟶ arr, P, Q

Return Solution

else:

mid = divide(arr, P, Q)

Concept of Recursion ⟶ Conquering part

Return Combine(Recursion) ⟶ combine part ⟶ not always required

P ⟶ lower index = 0

Q ⟶ Higher index = len(arr) - 1

Recurrence relation for above pseudo code :

$T(n) = 2T(n/2) + C \longrightarrow T(n) = aT(n/b) + f(n)$ # Master theorem

Binary Search Recurrence Relation

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

QuickSort Recurrence Relation

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

Maxima and Minima

70	50	45	10	12	15	75	29	37	57

→ Array

0 1 2 3 4 5 6 7 8 9

Brute force Approach

max =

min =

Best case → $(n-1)-1 = O(n)$

- Only one comparison
- Iterating through all element

Best case → $(n-1)\ 2 = O(n)$

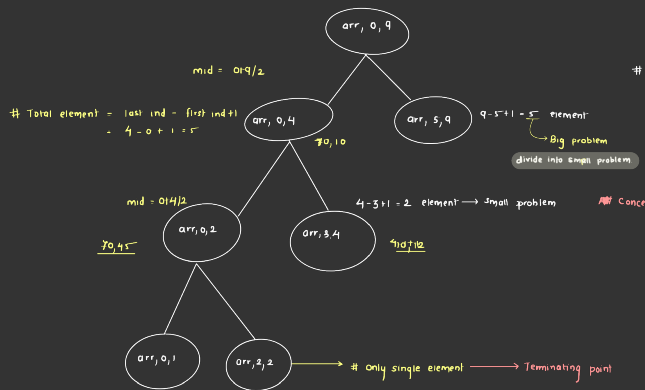
Average case → $\frac{n-1}{2} \cdot 1 + \frac{n-1}{2}\ 2$

Optimized above solution using → divide and conquer

Small problem → n == 1 or n == 2 # Single Comparison

Return n # max and minima

Big problem → Recursive Tree



Stack → data structure algorithm → last in first out

Responsible to store function call whenever → Recursion happens

Concept → if i == j → starting index and ending index both are equal

- Small problem → # Single element
- if i == j-1 → a[i] < a[j]
  - # two elements

→ Both Small problem

Psuedo Code

```
findminAndmax(arr,i,j):  ——> i,j # Starting and ending index

    if i==j ——> single element

        min,max = arr[i]

    elif i==j-1 ——> two element

        if arr[i] < arr[j]:

            min = arr[i]

            max = arr[j]

        else:

            min = arr[j]

            max = arr[i]

    else:

        mid = int((j-i)/2        # divide part

    maxL,minL = findmaxAndmin(arr,i, mid)
                                          ——> # Recursion OR conquer
    maxR, minR = findmaxAndmin(arr, mid+1, j)

        if maxL < maxR

            max = maxR
                                          ——> # Combine part
        else:

            max = maxL

        if minL < minR:

            min = minL
                                          ——> # Combining Part
        else:

            min = minR
```

Recurrence Relation ——> $T(n) = 2T\left(\frac{n}{2}\right) + C$

Master theorem ——> a=2, b=2, k=1, p=0

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

$\log_b^a > K$

$O(n^{\log_b^a}) = \theta(n)$ # Time Complexity

finding of power of an element

a ⟶ constant 2

n ⟶ number 10 ⟶ # Output ⟶ a^n ⟶ 2^{10} = 1024

Condition ⟶ $n \geq 1$

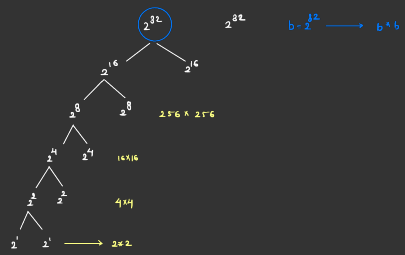
Application of divide and conquer

Small problem ⟶ Return solution directly n=1

Big problem ⟶ divide and conquer approach n>1

n = 64 a = 2 ⟶ 2^{64}

$a^n \longrightarrow a^{n/2} * a^{n/2}$



Recursive tree approach ⟶

Pseudo Code

```
if n==1:  ⟶ Small
    Return a
else:  ⟶ Big Problem
    mid = n//2
```

code

```
findPower(a,n)
    If n==1:  ⟶ Return a   # Small problem
    else:  # Big problem
        mid = n//2
        result = findPower(a,mid)
        if n%2==0 ⟶ Even
            Return result
        else
            Return result * a
```

if n ⟶ Even

2^{64} ⟶ will get correct answer

what if n ⟶ odd

2^{11} ⟶ $2^{10} * a$ ⟶ will get Correct answer

Time Complexity or Recurrence Relation

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

Applying masters theorem

$a=1, b=2, k=0, P=0$

$\log_b^a = \log_2^1 = 0$ ⟶ k=0 ⟶ Case 2

$P > -1$ ⟶ P=0

$O(\log n)$ ⟶ Time Complexity

Binary Search

> Sorted array

2	4	8	12	20	25	60	70
0	1	2	3	4	5	6	7

logic and pseudo code

Small problem ⟶ Single element

x = 50

if arr[i] == x ⟶ Return i

Otherwise Return -1

x = 50 ⟶ Target element

Big problem ⟶ elements more than 1 or 2

mid = (i+j)//2 = (0+7)//2 = 3

if arr[mid] == x ⟶ Return mid

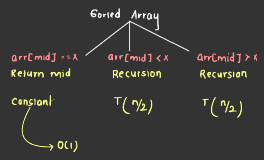
elif arr[mid] < x :

will go to ⟶ Right side of array ⟶ Recursion

else:

will go to ⟶ left side of array ⟶ Recursion

Recurrence Relation and time complexity



$$T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + c & n>1 \end{cases}$$ ⟶ # Recurrence Recurrence

Master's theorem

$\log_b^a = \log_2^1 = \text{constant}$ $\quad f(n) = O(n^k \log^p(c))$, $k=0$, $p=0$

$\log_b^a = k$ ⟶ case 2

$\theta(c \log n)$ ⟶ $O(\log n)$ # time complexity

Two pointer approach

Sorted array ⟶

0	1	2	3	4	5	6	⟶ Index
20	40	50	75	120	145	200	

target = 90

Output ⟶ arr[i] + arr[j] = target

arr[1] + arr[2] = 40 + 50 = 90

Pseudo Code for two pointer approach

a = 20, target - b = 90 - 20 = 70 ⟶ present or not ? in array

Searching are their elements

Linear Search ⟶ $O(n^2)$ ⟶ Brute force Approach

Binary Search ⟶ Optimized Approach ⟶ $O(n \log n)$

logic

0	1	2	3	4	5	6	
20	40	50	75	120	145	200	target = 90
i						j	

code

for i = 0 to n-1

if arr[i] + arr[j] == target ⟶ Return i, j

elif arr[i] + arr[j] > target

20 + 200 > 90

j = j-1 ⟶ decreement of j from 200 to 145

else

i = i+1 ⟶ increment of i

Return -1, -1

⟶ No such kind of pairs

Merge Sort

→ Combining task

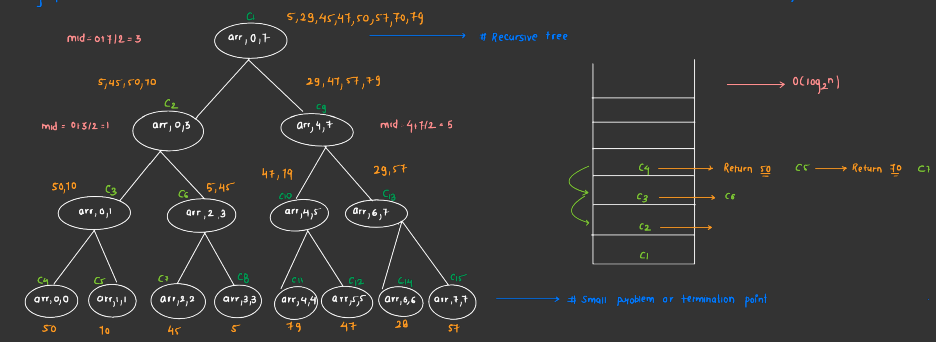
0	1	2	3	4	5	6	7
50	70	45	5	79	47	29	52

→ n=8

Small problem → Single element in array

Big problem → n>1

if l==r → #Single element



TOP to Down → divide & conquer → Small problem

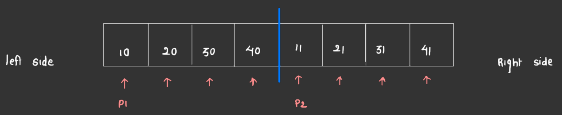
Down to TOP → Combining → Merge Procedure → # Sorted Array

Time Complexity of merge procedure

→ depends on two factors

→ number of moves

→ number of comparison



Merge Procedure

10	11	20	21	30	31	40	41

10<11

20>11

20<21 → # Comparisons → n-1 comparison

30>21 # worst case scenario

31>30

40>31

41>40

Best Case Scenario

30 > 5

30 > 10 → # no more comparison

no of comparison → min(m,n)

min(2,4) = 2 times comparison

moves → n+m # moving all the items

Time complexity for moves → O(m+n)

Overall time complexity merge sort → O(N)

Space complexity → O(N)

Extra Space → Outplace algorithm

Psuedo code for mergesort

```
mergeSort(arr, i, j)
    if i == j  → # small problem
        Return arr
    else
        mid = i + (j - i) // 2
        mergeSort(arr, i, mid)
        mergeSort(arr, mid+1, j)      → # Conquer part
        mergeProcedure(arr, i, mid, mid+1, j)  → # Combine sorted array
    Return arr
```

Recurrence Relation

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

By Master's theorem

a = 1 , b = 1 , K = 1 , P = 0

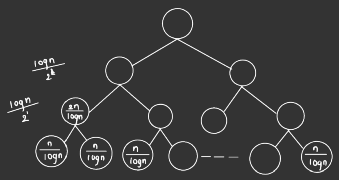
$\log_b^a = \log_2^2 = 1$ → Case no 2

$\log_b^a = K$

$\theta(n \log n)$ → # Time complexity

Problems Related to mergesort

1. log n Sorted subarray & each of subarray is of size n/logn



1 2 3 4 ... log n - 1 log n

$\frac{n}{\log n} \times \log n = n$ # At every layers number of element is n

until we reach at topmost layer → $\frac{\log n}{2^K} = 1$

$\log n = 2^K$

Taking $\log_2$ on both side

$\log_2(\log n) = K$ → # number of levels

Time complexity → O(n·K)

O(n log(log n))

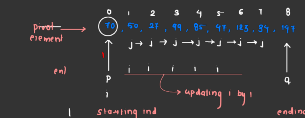
Quick Sort Algorithm

→ dividing → mid = (i+(j-1))//2

→ partition Algorithm

→ # other Application

How partition Algorithm works?



we can choose any pivot element

j ⟶ will take element greater than pivot element ⟶ it moves on

i ⟶ < pivot element

Pseudo code

i < pivot i = p

for j in i, j<n, j++

if arr[j] <= pivot:

i++

swaping of element will done

swap(arr(i), arr(p))

Quicksort(0,3) ⟶ # Recursive call

somehow smaller than pivot

Algorithm in 1st iteration ⟶ 34, 50, 27, 47, 70, 99, 123, 85, 197

pivot element reaches at own position

largest than pivot Quicksort(5,8) ⟶ # Recursive call

final output look like ⟶ 27, 34, 47, 50, 70, 85, 99, 123, 197

At every iteration pivot reaches its at own position

Pseudo code of partition algorithm

def partition(arr,p,q):

i = p j > pivot element ⟶ it move on

pivot = arr(p) i < pivot

for j in i+1 to n+1

if arr[j] <= pivot: 35, 50, 27, 47, 70, 99, 123, 85, 197

i++ Quicksort(p, m-1) m-p Quicksort(m+1, p)

Swap(arr(i), arr(j))

Swap(arr(i), arr(p))

Return i # index of mid element pivot ⟶ m

pseudo code of Quicksort Algorithm

Quicksort(arr,p,q)

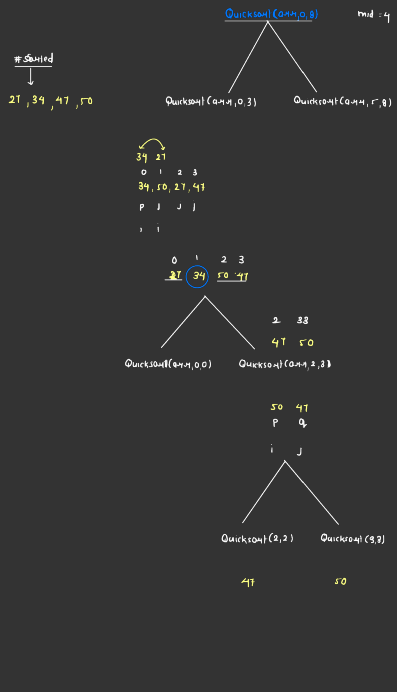
if p<q

mid = partition(arr,p,q)

Quicksort(arr, p, m-1) # left side Recursion ⟶ T(m-p)

Quicksort(arr, m+1, q) # Right side Recursion ⟶ T(q-m)

No need of combining



Recurrence relation

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

#Best case scenario

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

$$T(n) = 2T(n/2) + n$$ → # Same as mergesort

By master theorems

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

worst case scenario

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

$$T(n) = O(n^2)$$ # Time complexity

Quicksort
- Best or Average case $O(n \log n)$
  - # when array is highly unsorted
- Worst case $O(n^2)$
  - # Array is almost or fully sorted

Note

Inplace sorting algorithm

not stable algorithm (Quicksort, Heapsort) → sequence is not retain

Randomized Quicksort → # Another version of quicksort

Pivot element completely random

we can random function in order to select pivot element R

```
random_pivot = random(arr,p,q)
swaping(arr[random_pivot], arr(p))
return partition(arr,p,q)
```

→ # All the implementation remain same as quicksort

Why random selection of pivot is required ?

As we know worst case scenario time complexity → $O(n^2)$

→ This can become less in randomized

→ # Chances are there