UNIT - 3 ANALYSIS OF SORTING & SEARCHING ALGORITHMS

Brute Force Strategy - Selection Sort - bubble Sort - brute force string matching - Exhaustive Search - knapsack problem - assignment problem - travelling salesman problem - divide and conquer - general method - masters theorem - merge sort - Quick Sort - binary search.

Algorithm design techniques / strategy

Important to build, readable, scalable and efficient system.

Selecting a proper design technique is a complex but important task.

Few techniques are.

1. Brute force / Exhaustive search technique.
2. divide & conquer technique.
3. Greedy algorithm technique.
4. dynamic programming technique.
5. Branch & Bound technique.
6. Randomised alg
7. Backtracking technique.

Brute force technique

It is a straight forward approach to solve a problem directly based on the problem statement and definitions of the concepts involved.

It finds all the possibilities to find the satisfactory solution for the given problem.

It trys out all the possibilities till the satisfactory solution is found.

used to solve trivial and every problem.

Eg: linear search, some sorting algorithms, find the maximum element of a list, find out an element in a range i, j based on a condition, finding the password (or) PIN (or) OTP

Advantages

1. Simple strategy.
2. Idle for solving smaller & simpler problem

disadvantages

1. Consumes time — algorithm runs slowly
2. works well only for lesser no. of elements
3. inefficient for solving real time problems when the complexity is very high.

Algorithms using brute force technique

1. Linear / sequential search,
2. Selection sort.
3. bubble sort.
4. brute force string matching.

13/2/23

Selection sort

purpose

To sort the set of elements in an array of size n.

How?

The algorithm starts with an assumption of 2 partition - sorted & unsorted list.

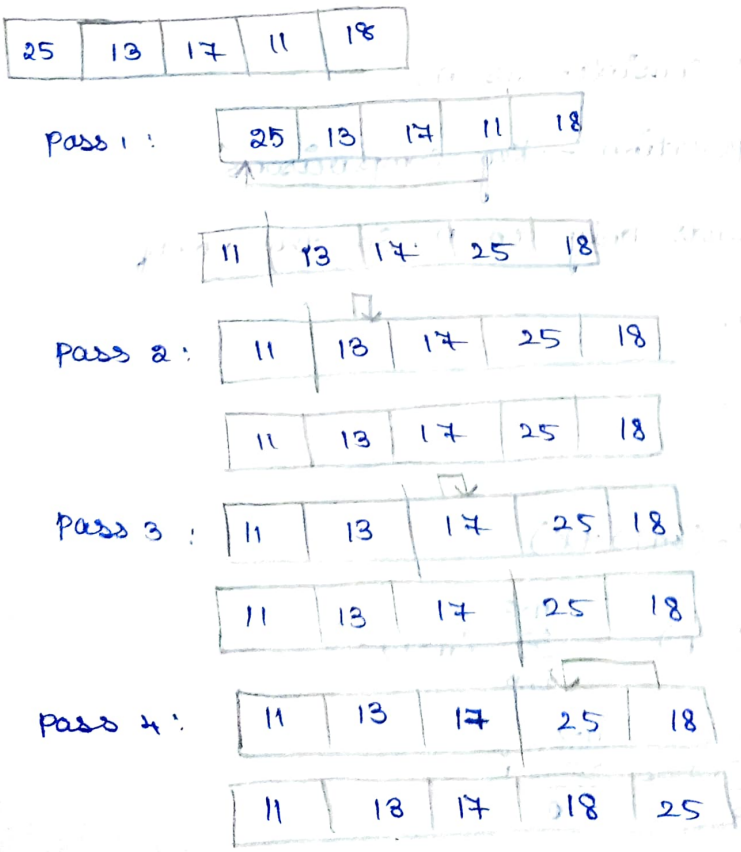
The algorithm finds the minimum element of the unsorted list and place it in the 1st position of the unsorted list.

After every pass, the size of sorted list increases & size of the unsorted list decreases by 1.

The process is repeated n-1 times. so the no. of passes = n-1

Example.

n = 5



No. of passes = 4 = n-1

Alg SelectionSort (A[0..n-1])

// Sorts n elements in ascending order using selection sort.

// I/P: An array A[0..n-1]

// O/P: A sorted array with n elements

```
for i = 0 to n-2 do
    min = i
    for j = i+1 to n-1 do
        if A[min] > A[j]
            min = j
    temp = A[min]
    A[min] = A[i]
    A[i] = temp
return A
```

Analysis

1. The parameter considered is n.
2. The basic operation - key comparisons
3. Analysis depends only on n. So the no. of comparisons = C(n).

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1-(i+1)+1)$$

$$= \sum_{i=0}^{n-2} (n-i-1) = \sum_{i=0}^{n-2} (n-1) - i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$= (n-1)(n-2-0+1) - \frac{(n-2)(n-2+1)}{2}$$

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

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

$$= (n-1)\left(\frac{n}{2}\right) = \frac{(n-1)n}{2} = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

The algorithm includes swaps. So no. of swaps = $\sum_{i=0}^{n-2} 1 = (n-1) \in \Theta(n^2)$

Bubble Sort

Purpose

To sort n elements in ascending or descending order.

Strategy

Every pair of adjacent elements are compared from 0th position to n-1th position

If there are not in order, they are exchanged or swaped. Thus during pass one, the largest element moves towards the n-1th position, in pass 2, the second largest element moves to in the position in the sorted order.

After n-1 passes the elements get sorted.

In each pass an element bubbles towards its position in the sorted list.

In ith pass,

$A_0, \ldots, A_j, A_{j+1}, \ldots, A_{n-i} \mid A_{n-i+1} \leq \ldots \leq A_{n-1}$

A_j and A_{j+1} compare

$A_0, \ldots, A_{n-i}$ — Unsorted list

$A_{n-i+1} \ldots A_{n-1}$ — sorted

After $n-1$ passes:

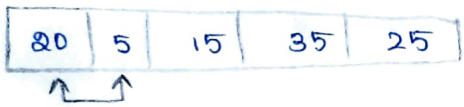
A_0 Unsorted list | $A_1 \leq \ldots \leq A_{n-1}$ Sorted list

Example.

$n = 5$

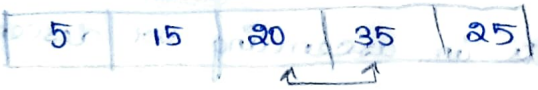
20, 5, 15, 35, 25

Pass 1



Swap

Swap

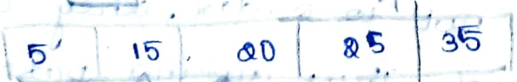


No swap

Swap

Pass 2

No swap



No swap

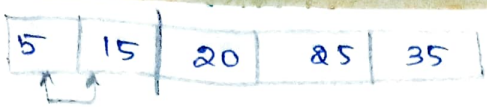
No swap

Pass 3

No swap

No swap

Pass 4



No swap

No. of passes = $n-1 = 5-1 = 4$

No. of comparison in each passes for $n=5$

Pass 1 = 4

Pass 2 = 3

3 = 2

4 = 1

Alg bubblesort (A[0..n-1])

// sorts n elements in ascending order using bubble sort.

// I/P : An array A[0..n-1]

// O/P : A sorted array with n elements

```
for i = 0 to n-2 do
    for j = 0 to n-2-i do
        if A[j] > A[j+1]
            temp = A[j]
            A[j] = A[j+1]
            A[j+1] = temp
return A
```

15/2/23

Analysis

1. parameter used is n - The no. of elements in the list.

2. basic operation - key comparisons.

3. The analysis depends on n only, so best, worst, average case need not be separately done.

4. The summation is set up based on n to count the no. of comparisons.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

5. To solve the summation

$$C(n) = \sum_{i=0}^{n-2} n-2-i+1$$

$$= \sum_{i=0}^{n-2} n-1-i$$

$$= \sum_{i=0}^{n-2} (n-1)-i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

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

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

$$= (n-1)\left[\frac{2n-2-n+2}{2}\right]$$

$$= (n-1)\left(\frac{n}{2}\right) = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

No. of comparisons $C(n) \in \Theta(n^2)$

another operation performed in the algorithm is swapping.

No. of swaps depend on the i/p provided.

If the list is in descending order, the no. of swaps will be maximum - one swap per iteration

A worst case i/p is a list in descending order.

$$\text{No. of swaps} = \frac{n(n-1)}{2} \in \Theta(n^2)$$

17/2/23

Brute force string matching (3m)

* n - character called text
* $m \leq n$ pattern

* The problem is to find whether a string are called pattern is present in another pattern is called text
* It is done by comparing every character of the pattern with that of text
* If the 1st character match proceed comparing the 2nd char & continue

when the mismatch occurs, move the pattern & aline with the next sub sequence char. of text & proceed comparing with the patterns 1st char again.

* The algorithm return the position from which the pattern exactly matches the text.
* If pattern is not present in the text the algorithm returns -1.
* let T be the text with n characters & let P be the pattern with m character where $m \leq n$

Algorithm

* T[0..n-1], P[0..m-1]

// I/P : (T[0..n-1], P[0..m-1])

// O/P : Index of 1st character or return (-1)

```
for i ← 0 to n-m do
    j ← 0
    while j<m and P[j] = T[i+j] do
        j ← j+1
    if j = m
        return i
return -1
```

Analysis

parameter - n, m

Basic operation - comparison

Best case

- The 1st m char of the text contains the pattern.

Eg text = NOBODY

Pattern = NOB

worst case

- if the m-1 characters of the pattern is matching with the text from 0 to n-1 char & last char of the text & pattern don't match.

The alg takes m comparison every times.

Hence $C_{worst}(n) = m(n-m+1)$

Eg text → a a a a a a a a

Pattern → a a a b.

text → $t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1}$

Pattern → $P_0 \ldots P_j \ldots P_{m-1}$

↓

size = m

this is status of aliring the pattern with the text in any ith iteration.

If $t_i \ldots t_{i+m-1}$ and $P_0 \ldots P_{m-1}$ match then the algorithm returns i.

22/2/23

Exhaustive search (ES)

A brute force approach to solve combinatorial problem.

Given a finite collection of objects and a

set of constraints, we can find and objects are a set of object that satisfies all the constraints.

This can be solved using exhaustive search technique.

It can also be used for optimized result.

optimisation has 2 categories

maximisation → the maximum value among all the possibilities is chosen as the best

minimisation

↓

The minimum value among all the possibilities is chosen as best.

Procedure to solve using exhaustive search

1. generate all the possible combinatorial objects using an appropriate algorithm.
2. Identify those thats satisfy the constraints.
3. find the best out of the identify elements - optimisation

Problems using E.S

1. Travelling salesman problem (TSP)
2. knapsack problem
3. Assignment problem.

1. Travelling salesman problem

The problem is to find the shortest tour through a set of n city, visiting each of the city only once & returing to the city where the tour was started.

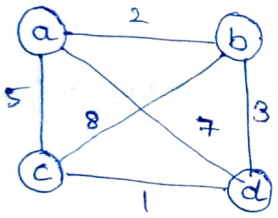
The procedure to solve TSP. Since this is based on a graph $G=\{V, E\}$ and the weighted graph, to solve TSP

1. generate all the possibilities of tours that contain all permutation of n-1 cities. and the 1st & last city same as source.

2. Compute the total length for each possibilities.

3. Find the shortest among them.

Eg



If source is A the possible paths are

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$

Paths	Path length
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$2+8+1+7 = 18$
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$2+3+1+5 = 11$
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$5+8+3+7 = 23$
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$5+1+3+2 = 11$
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$7+1+8+2 = 18$
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$7+3+8+5 = 23$

when source = a, $(n-1)!$ possibilities available similarly if we consider remaining $n-1$ vertices also, we get a total of $n(n-1)!$ possibilities available.

$n!$

Hence it is an exhaustive search technique

d must precede b → the no. of permutations possible =

$\frac{(n-1)!}{2}$

1/3/23

A hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly one.

TSP is also considered as the problem of finding the shortest hamiltonian circuit of the graph.

Assignment problem

There are n people who need to be assigned to execute n jobs, one person per job.

The cost that would incur if the i^{th} person is assigned to the j^{th} job. → assignment costs → $c[i, j]$ for each pair $i, j = 1, 2, \ldots, n$.

Problem → to find an assignment with the minimum total cost.

	Job 1	Job 2	Job 3	Job 4
person 1	9	2	7	8
person 2	6	4	3	7
person 3	5	8	1	8
person 3	7	6	9	4

Select one element in each row of the matrix so that all selected elements are in different columns & the total sum of the selected elements is the smallest possible.

Eg $<2, 3, 4, 1>$ person 1 is assigned job 2, person 2 - job 3, person 3 - job 4, person 4 - job 1.

Procedure to solve this problem

generate all the permutations of the set of n jobs.

Compute the total cost for each assignment.

Select the one with the smallest sum.

Solution.

Assignment	Assignment costs
<1, 2, 3, 4>	9 + 4 + 1 + 4 = 18
~~<2, 1, 4, 3>~~	~~2 + 6 + 8 + 9 = 25~~
<1, 2, 4, 3>	9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	9 + 7 + 1 + 6 = 23
<2, 1, 3, 4>	2 + 6 + 1 + 4 = 13
<2, 1, 4, 3>	2 + 6 + 8 + 9 = 25
<2, 3, 1, 4>	2 + 3 + 5 + 4 = 14
<2, 3, 4, 1>	2 + 3 + 8 + 7 = 20
<2, 4, 3, 1>	2 + 7 + 1 + 7 = 17
<2, 4, 1, 3>	2 + 7 + 5 + 9 = 23

No. of permutations that can be considered n!

So impractical when n increases.

An efficient algorithm for this problem → the hungarian method.

Knapsack problem

given n items with known weights $w_1, w_2 \dots w_n$

* values $v_1, v_2 \ldots v_n$ and a knapsack of capacity of w, find the most valuable subset of items that fit into the knapsack.

Procedure

generate all the subsets of the set of n items

compute the total weight of each subset

identify feasible subsets

identify the optimal subset with maximum value and total weight less than the knapsack capacity.

Find the solution for the given knapsack problem using exhaustive search technique.

Items (i)	weight (w_i)	value (v_i)
1	7	Rs. 42
2	3	Rs. 12
3	4	Rs. 40
4	5	Rs. 25

knapsack capacity, W = 10

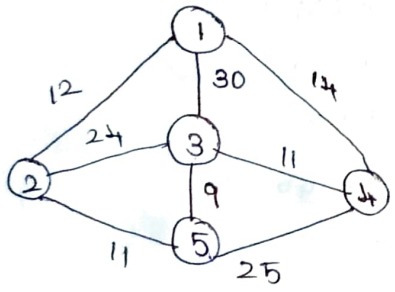
subset	Total weight	Total Value
ϕ	0	Rs. 0
{1}	7	Rs. 42
{2}	3	Rs. 12
{3}	4	Rs. 40
{4}	5	Rs. 25
{1,2}	10	Rs. 54
{1,3}	11	Rs. 82 → Not feasible
{1,4}	12	Rs. 67 → Not feasible
{2,4}	8	Rs. 37
{3,4}	9	Rs. 65
{1,2,3}	14	Rs. 94 → Not feasible
{1,2,4}	15	Rs. 79 → Not feasible

{2, 3, 4}	12	47 → not feasible.
{1, 2, 3, 4}	19	119 → not feasible
{2, 3}		

The no. of subsets when n = 4 (items), = 16 which is equal to 2^4.

Since the no. of subsets of an n-element set = 2^n, the exhaustive search leads to $C(n) \in \Omega(2^n)$.

1) Apply exhaustive search to solve TSP for the given graph



2) knapsack problem

items	weight	values
1	2	Rs. 5
2	1	Rs. 2
3	3	Rs. 4
4	5	Rs. 13
5	6	Rs. 14

knapsack capacity, W = 8

3) assignment problem

Jobs	machines I	II	III	IV
A	10	12	19	11
B	5	10	7	8
C	12	14	13	11
D	8	15	11	9

1) path — path length

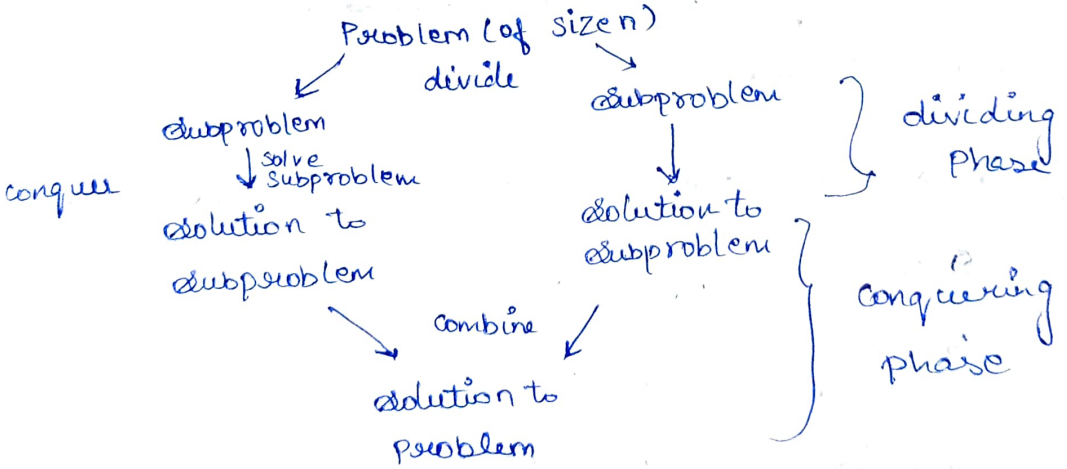
1 → 2 → 5 → 3 → 4 → 1
1 → 2 → 3 → 4 → 5 → 1
1 → 2 → 5 → 4 → 3 → 1
1 → 3 → 4 → 5 → 2 → 1
1 → 3 → 2 → 5 → 4 → 1
1 → 4 → 5 → 2 → 3 → 1
1 → 4 → 3 → 5 → 2 → 1
1 → 4 → 5 → 3 → 2 → 1
2 → 3 → 1 → 4 → 5 → 2
2 → 5 → 4 → 3

3/3/23

divide and conquer

General plan.

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved.
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.



Eg.

The problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$

If $n = 1$ return a_0;

if $n > 1$, divide the problem into 2 portions.

$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{n/2-1}) + (a_{n/2} + \ldots + a_{n-1})$.

For eg., if n=10, $(a_0 + \ldots + a_4) + (a_5 + \ldots + a_9)$.

0	1	2	3	4	5	6	7	8	9
88	44	11	22	99	66	77	55	33	99

Problem can be divided into >= 2 partitions

Assume: No. of partitions → b Each of size → n/b

No. of partitions to be solved → a

Hence: The running time of an algorithm using D & C technique is

$$T(n) = a\,T(n/b) + f(n)$$

where $f(n)$ → function that accounts for the time spent on dividing phase.

Master theorem

divide and Conquer recurrence can be be solved using master theorem.

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Analogous results hold for the O and Ω notations also too.

Eg $A(n) = 2A(n/2) + 1$

$a = 2 \quad b = 2 \quad f(n) = 1 = n^0 \Rightarrow d = 0$

$a < b^d \rightarrow 2 < 2^0 \rightarrow \times$

$a = b^d \rightarrow 2 = 2^0 \rightarrow \times$

$a > b^d \rightarrow 2 > 2^0 \rightarrow \checkmark$

$\therefore A(n) \in \Theta(n^{\log_b a})$

$A(n) \in \Theta(n^{\log_2 2})$

$\boxed{A(n) \in \Theta(n)}$

Few algorithms that can be solved using divide & conquer technique

1. Merge sort
2. Quick sort
3. Binary search

Merge sort

The elements to be sort is in an array of size n.

In dividing phase, the array is divided into 2 partitions of same size.

The partitions are recursively and repeatedly divided into 2 partitions everytime, till the partition size equals one.

In conquering phase, 2 adjacent portions are sorted & merged. This is done recursively till a single array is achieved.

Algorithm.

Alg : MergeSort(A[0..n-1])

// sorts a list of elements using merge sort, recursively.

// i/p : An array A[0..n-1] with n sortable elements

// o/p : An array with elements in sorted order.

if n>1

Copy A[0..$\frac{n}{2}$-1] to B[0..n/2-1]

Copy A[0..n/2-1] to C[0..n/2-1]

Mergesort(B[0..n/2-1])

Merge sort (C[0.. n/2 - 1])

Merge (B, C, A)

the alg. mergesort is recursive algorithm

It uses another algorithm called merge for merging 2 sorted arrays.

Alg: Merge (B[0..p-1], C[0..q-1], A[0..p+q-1])

// merges two sorted arrays

// i/p: Two sorted arrays B[0..p-1] & C[0..q-1]

// o/p: A single sorted array A[0..p+q-1]

```
i ← 0
j ← 0
k ← 0
while i < p and j < q
    if B[i] < C[j]
        A[k] ← B[i]
        i ← i+1
    else
        A[k] ← C[j]
        j ← j+1
    k ← k+1
if i = p
    copy C[j..q-1] to A[k..p+q-1]
else
    copy B[i..p-1] to A[k..p+q-1]
return A
```

The basic operation to be analysed is key comparison.

If C(n) is the no. of key comparison then

$$C(n) = 2C\left(\frac{n}{2}\right) + C_{merge}(n)$$

The no. of comparison in worst case

$C_{worst}(n) = n-1$ → during conquering phase. So

$C_{worst}(n) = 2C\left(\frac{n}{2}\right) + n - 1$

To solve this using master theorem

$a = 2 \quad b = 2$

$f(n) = n-1 \in \Theta(n) \Rightarrow n^d = n \Rightarrow d = 1$

$a < b^d \longrightarrow 2 < 2^1 \longrightarrow \times$

$a = b^d \longrightarrow 2 = 2^1 \longrightarrow \checkmark$

$a > b^d \longrightarrow 2 > 2^1 \longrightarrow \times$

so $\boxed{C(n) \in \Theta(n^d \log n)}$

8/3/23

Variations in MergeSort

1. Bottom-up approach consider every pair of elements in the array & sort them & merge them as the 1st step.

In the next step consider merging two different adjacent pairs.

Proceed in the same way till all the n elements are considered as a single merged sorted array.

2. Multiway merge.

The no. of partitions can be more than two.

Quick Sort

The dividing phase is an important one in which the elements are partition based on their values.

but in merge sort the partitioning is

done based on their position of the elements.

This is an important algo that uses divide & conquer technique.

The basic idea

Partition the array A[0..n-1] into 3 partition (or) sub arrays. such that A[0], ... ① elements ≤ A[s-1], A[s], A[s+1], ... A[n-1]. ② Elements ≥ pivot

pivot

pivot

The pivot element is moved to a splitting position s, all the elements towards its left or less than the pivot & all the elements towards its right or greater than the pivot.

s is called as splitting position.

Pivot the choice of pivot can vary — either a first element or last element, or the middle element of a sub array.

Once the partition can be done apply the same procedure to partition the sub array ① & ② recursively.

Finally, when no more partitioning is possible the dividing phase is over & the list is sorted — no separate conquering phase.

Algorithm

Alg: QuickSort (A[l..r])
// Sorts an array recursively using Quicksort
if l < r
s = partition // I/P: A sub array A[l..r] that was partition from the full array A[0..n-1].

// O/P: A sorted sub array.

if $l < h$

$s =$ partition $(A[l..h])$

Quicksort $(A[l..s-1])$

Quicksort $(A[s+1..h])$

Partitioning procedure

The subarray has to be divided into 3 partitions

1. elements < pivot
2. the pivot
3. elements > pivot

Partitioning starts by selecting the pivot - An elements with respect to which the array has to be partitioned

The simplest strategy is to chose the 1st element as pivot.

Use to marker i & j, start i with the 2nd element position & start j with the last element position.

If the element pointed by i is ≤ pivot move the pointer to next position. otherwise stop moving the pointer i

If the element pointed to by j ≥ pivot, move the pointer j to previous position, otherwise stop moving j.

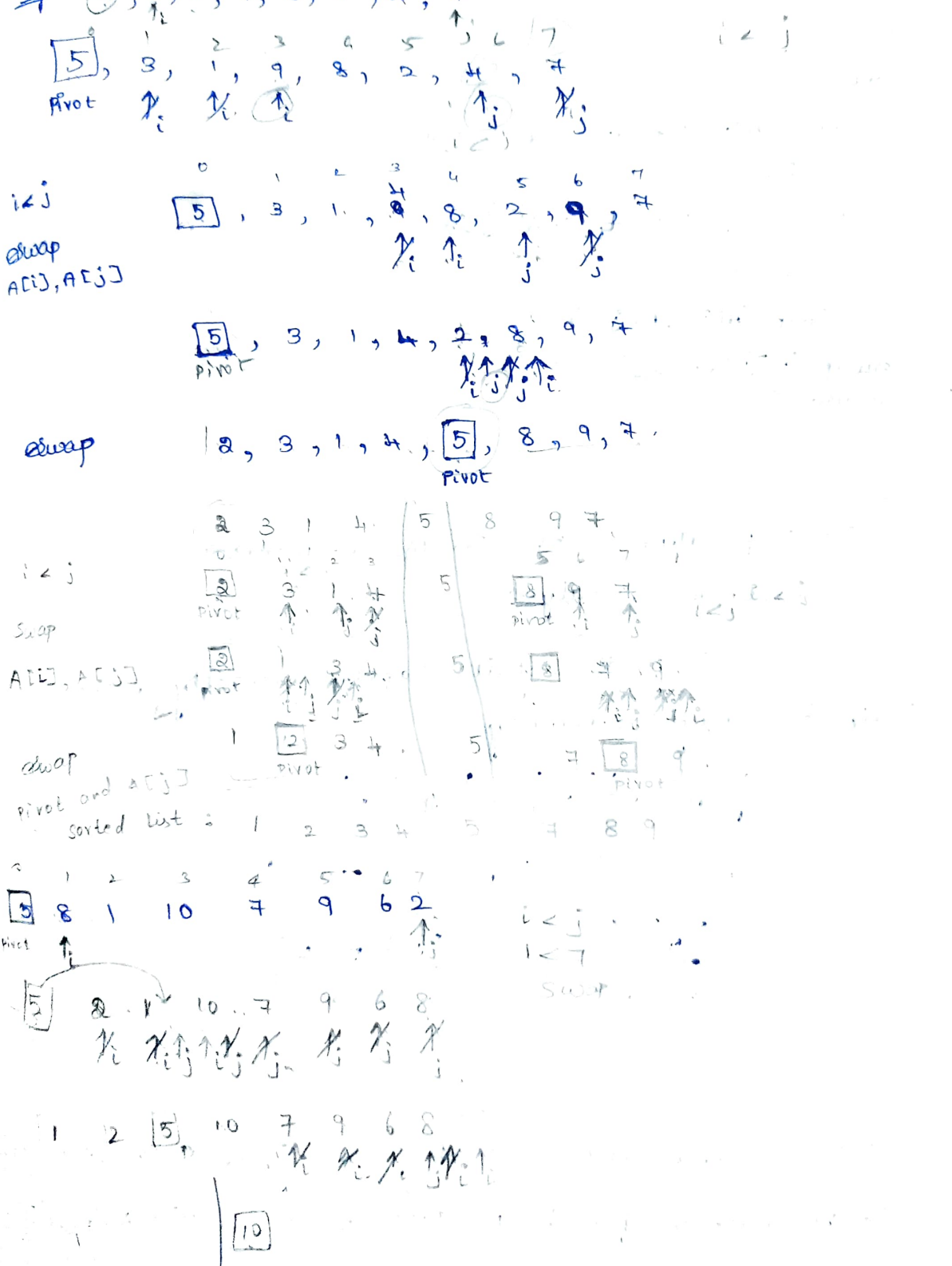
i & j can stop moving in 2 cases

i) $i < j$ → swap A[i] and A[j].

ii) $i \geq j$ → swap pivot and A[j].

The second case stops the iteration with 2 subarrays available.

Continue the partitioning in the same way on the subarrays obtained.



10/3/23

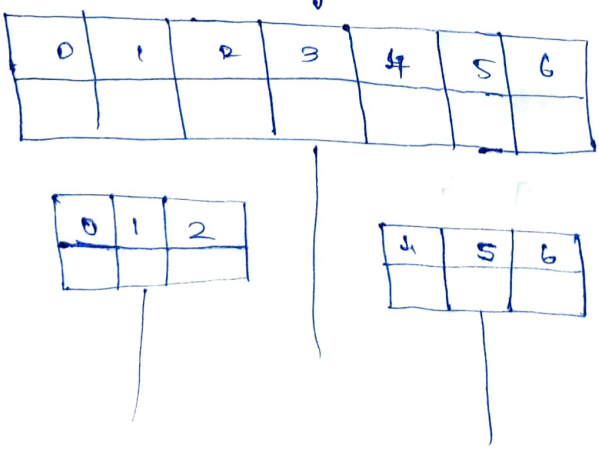
```
Alg : partition ( A[l..r] )
// Splits an array into two subarrays
// i/p : A[l..r]
// o/p : The splitting position is returned
pivot = A[l]
i = l+1
j = r
while i ≤ j
    while A[i] ≤ pivot and i < r
        i = i+1
    while A[j] ≥ pivot and j > l
        j = j-1
    swap (A[i], A[j])
swap (A[j], A[l])
return j
```

Analysis

Basic operation is the no. of comparisons.

1. Best case analysis

When the split happens exactly in the middle in every subarray.



The no. of comparisons in best case $C_{best}(n) = n+1 \approx n$

The recurrence for best case is $C_{best}(n) = 2C_{best}\left(\frac{n}{2}\right) + n$

Solving using master theorem.

Best $a = 2 \quad b = 2$

$f(n) = n \in \Theta(n) \Rightarrow n^d = n \Rightarrow d = 1$

$a < b^d \longrightarrow 2 < 2^1 \longrightarrow \times$

$a = b^d \longrightarrow 2 = 2^1 \longrightarrow \checkmark$

$a > b^d \longrightarrow 2 > 2^1 \longrightarrow \times$

$\therefore$ $C_{best}(n) \in \Theta(n \log n)$

17/3/23

2. worst case

If the partitions are skewed to one extreme.

Eg: If the list is in increasing order.

0	1	2	3	4	5	6	7
2	3	4	5	66	77	81	90

$1 + 8 = 9 = n+1 \longrightarrow 1^{st}$ iteration

$1 + 7 = 8 = n$

$= n-1$

$n-2$

$\vdots$

$3 \longrightarrow$ last iteration

Total no. of comparisons in worst case =

$(n+1) + n + (n-1) + (n-2) + \ldots + 3$.

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

$3 + \ldots + n + n + 1 = \frac{(n+1)(n+2)}{2} - 3$

$C_{worst}(n) \in \Theta(n^2)$

Binary search.

This algo is used to find whether a element k is present in a list or not.

The element must be in sorted order to implement this algorithm.

This algo is based on decreased by

Constant factor technique which is a variety of Divide & Conquer technique.

basic strategy

To search for a key k in an array of n elements, compare k with the middle element of the array.

If they match, stop the algorithm & return the middle position.

If k is less than the middle element apply the same process in the first half of the array.

If k > middle element apply the same process in the second half of the array.

A[0], A[1], ... A[m-1], A[m], A[m+1], ... A[n-1]

Search here if k < A[m] — k — Search here if k > A[m]

Eg

0	1	2	3	4	5	6	7	8
3	8	9	11	12	18	22	35	40

k = 22

First = 0 last = 8

$mid = \left\lfloor \frac{First + last}{2} \right\rfloor = \left\lfloor \frac{8}{2} \right\rfloor = 4$

k → A[mid] ⇒ 22 > 12

So search in 2nd half.

18	22	35	40
5	6	7	8

First = mid + 1
= 4 + 1
= 5

last = 8 mid = $\left\lfloor \frac{5+8}{2} \right\rfloor = \left\lfloor \frac{13}{2} \right\rfloor = \lfloor 6.5 \rfloor$
= 6

K → A[6] ⇒ 22 = 22
stop the search
K = 22 is in position 6.

K = 11

First = 0 last = 8
mid = $\left\lfloor \frac{\text{first} + \text{last}}{2} \right\rfloor = \left\lfloor \frac{8}{2} \right\rfloor = 4$

K → A[mid] = 11 < 12
do search in 1st half.

3	8	9	11
0	1	2	3

K → A[mid] = 11 > 8.
do search in 2nd half

9	11
2	3

first = 2 last = 3
mid = $\left\lfloor \frac{2+3}{2} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$

K → A[mid] = 11 > 9
So search in 2nd half.

11
3

first = 3 Last = 3
mid = $\left\lfloor \frac{3+3}{2} \right\rfloor = \left\lfloor \frac{6}{2} \right\rfloor = 3$
K → A[mid] ⇒ 11 = 11
K = 11 is in position 3

First = 0 Last = 4 - 1 = 3
mid = $\left\lfloor \frac{3}{2} \right\rfloor = \lfloor 1.5 \rfloor = 1$

4/3/23

59

0	1	2	3	4	5	6	7	8	9	10	11	12
8	11	15	17	30	32	58	59	60	72	75	83	90

K = 59

First = 0 Last = 12
mid = $\left\lfloor \frac{\text{First} + \text{last}}{2} \right\rfloor = \left\lfloor \frac{0+12}{2} \right\rfloor = 6$

K → A[mid] ⇒ 59 > 58
do search in 2nd half.

59	60	72	75	83	90
7	8	9	10	11	12

First = mid + 1
= 6 + 1
= 7

Last = 12
mid = $\left\lfloor \frac{7+12}{2} \right\rfloor = \left\lfloor \frac{19}{2} \right\rfloor = 9$

K → A[9] ⇒ 59 < 72
do search in 1st half.

59	60
7	8

First = mid - 2
= 9 - 2
= 7

last = mid - 1
9 - 1
last = 8
mid = $\left\lfloor \frac{7+8}{2} \right\rfloor = \left\lfloor \frac{15}{2} \right\rfloor = 7$

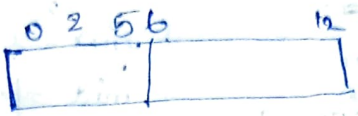
$k \longrightarrow A[7] \Rightarrow 59 = 59$

stop the search

so k

$k = 59$ is in position 7

② $k = 20$



First = 0 Last = 12

$mid = \left\lfloor \frac{12}{2} \right\rfloor = 6$

Compare

$k \xrightarrow{\text{and}} A[mid] \Rightarrow 20 < 58$

do search in 1st half.

0	1	2	3	4	5
8	11	15	17	30	32

First = 0 ~~mid~~ Last = 5

$mid = \left\lfloor \frac{5}{2} \right\rfloor = 2$

$k \longrightarrow A[mid] \Rightarrow 20 > 15$

do search in 2nd half.

17	30	32
3	4	5

first = 3 Last = 5

$mid = \left\lfloor \frac{8}{2} \right\rfloor = 4$

$k \rightarrow A[mid] \Rightarrow 20 < 30$

do search in 1st half.

17
3

First = 3 Last = 4 − 1 = 3

$mid = \left\lfloor \frac{3+3}{2} \right\rfloor = \left\lfloor \frac{6}{2} \right\rfloor = 3$

$\frac{3+3}{2} = \left\lfloor \frac{6}{2} \right\rfloor = 3$

~~$k = 20$ is not present~~

$k \rightarrow A[3] \Rightarrow 20 \neq 17$

$k = 20$ not in the list

The alg returns −1, since $k = 20$ is not found in the array.

Algorithm

Alg Binary search (A[0..n-1], K)

// Alg searches whether K is present in the array or not using binary search technique.

// i/p : Array A[0..n-1] with elements in sorted order and the key K.

// o/p : position of K, if K is present in the array; otherwise -1 if K is not present in the array.

```
first = 0
last = n-1
while first ≤ last
    mid = ⌊(first + last)/2⌋
    if K = A[mid]
        return mid
    else if K < A[mid]
        last = mid - 1
    else
        first = mid + 1
return -1
```

Analysis

The basic operation includes key comparison and few arithmetic calculation.

$$C(n) \in \Theta(\log n)$$

Since the array is divided into 2 halfs every time. and the search process is continued in one of the parts.