Brute Force

3

Brute force is a straightforward approach to solving a problem usually directly based on the problem statement and definitions of the concepts involved.

1] Selection Sort

Algorithm Selection Sort (A[0...n-1])

// Sorts a given array by selection sort

// Input: An array A[0...n-1] of orderable elements

// Output: Array A[0...n-1] sorted in ascending order.

```
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Analysis

Input size: n

Basic operation: A[j] < A[min]

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

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

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

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

$\therefore C(n) = \Theta(n^2)$

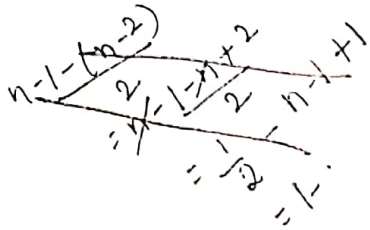
2) Bubble Sort

Algorithm BubbleSort (A[0...n-1])
// Sorts a given array by bubble sort
// Input: An array A[0..n-1] of orderable elements
// Output: Array A[0...n-1] sorted in ascending order

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j] swap A[j] and A[j+1]
```

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

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



OR

Algorithm Bubblesort (a[], n)

Purpose :- Arrange the numbers in ascending order.

Input :- An array A[0...n-1] of orderable elements

Output :- Array A[0..n-1] sorted in ascending order.

```
for j ← 1 to n-1 do
    for i ← 0 to n-j-1 do
        if (a[i] > a[i+1])
```

```
            temp = a[i]
            a[i] = a[i+1]
            a[i+1] = temp
        end if
    end for
end for.
```

Input parameter = n.

Basic operation = a[i] >= a[i+1]

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

$$= \sum_{j=1}^{n-1} n-j-1-0+1 = \sum_{j=1}^{n-1} n-j = (n-1)+(n-2)+\dots+3$$

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

$C(n) \in \Theta(n^2)$.

Sequential Search

Algorithm Sequential Search

// Implements sequential search with a search Key ~~as a sentinel~~.

// Input :- An array A of n elements and a search key K.

// Output :- The index of the first element in A[0..n-

whose value is equal to K or -1 if no such element is found.

```
A[n] ← K        // Insert search key at end of the list
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i   // Search successful
else return -1      // Search unsuccessful because item
                    // found in nth position.
```

Time efficiency = O(n) in worst case.

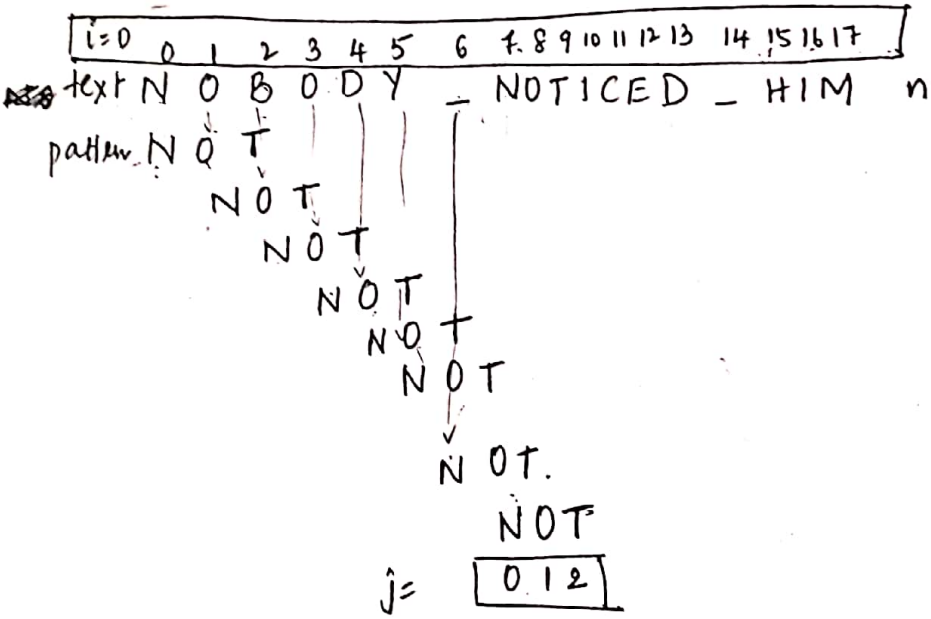
Brute force String Matching

Definition:- Given a string called text with n characters and another string called pattern with m characters where $m \leq n$. It is required to search for the string pattern in the string text. If the search is successful return the position of the first occurence of pattern string in the text string. Otherwise return -1

Algorithm BruteForceStringMatch (T[0...n-1] P[0...m-1])

// Implements brute-force string matching
// Input : An array T[0..n-1] of n characters representing a te
// an array P[0..n-1] of m characters representing a p
// Output: The index of the first character in the text
// that starts a matching substring or -1 if the
// search is unsuccessful.

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



18 n
- 3 m
15

Comparisons

text [7] with pattern [0] ie text [7+0] – pat
text [8] with pattern [1] text [7+1] – pat
text [9] with pattern [2] text [7+2] – pat

where $j <= 2$
$j < 3$
$j < m$ ($m = 3$ is length of pattern string)

Sliding the pattern string towards right is nothing but incrementing the index i of text string by 1.

Best case analysis

The best case occurs if the pattern string to be searched is present in the beginning of the text string. If m is the length of the pattern string, the no. of comparisons required is m.

$$\therefore f(n) = \Omega(m)$$

Worst case analysis

If the pattern string is present at the end of the text string or if the pattern string is not present in the text string.

Step 1:- Parameter are m & n

Step 2:- Basic operation = pattern[j] = text[i+j].

$$f(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

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

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

$$= mn - m^2 + m$$

$$= \in O(mn)$$

Exhaustive Search

The brute-force approach used to solve combinatorial problems is called exhaustive search. The solution to the combinatorial problems consumes too much time

This searching technique generate all the possible solutions by ~~the~~ satisfying the constraints given in the problem. Finally the desired solution which maximi the profit is selected.

The goal of exhaustive search method is to search all possible solution and obtain an optimal solution.

1) Traveling Salesman problem
2) Knapsack Problem
3) Assignment Problem.

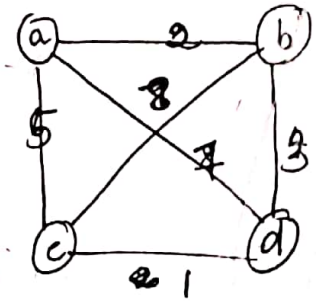
1) Travelling Salesman Problem

Given n cities, a salesperson starts at a specified city visit all n-1 cities only once and return to the city from where he has started. The objective of this problem is to find a route through the cities that minimizes the cost.

- vertex represent various cities
- weights associated with edges represent distance between two cities

The problem can be stated as the problem of finding the shortest Hamiltonian circuit

Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.



Tours	Length
$a \to b \to c \to d \to a$	$l = 2+8+1+7 = 18$
$a \to b \to d \to c \to a$	$l = 2+3+1+5 = 11$ optimal
$a \to c \to b \to d \to a$	$l = 5+8+3+7 = 23$
$a \to c \to d \to b \to a$	$l = 5+1+3+2 = 11$ optimal
$a \to d \to b \to c \to a$	$l = 7+3+8+5 = 23$
$a \to d \to c \to b \to a$	$l = 7+1+8+2 = 18$.

Analysis

Let us see the various routes (permutations) that can be used by a traveling salesperson. where he visits each & every city exactly once and returns back to the city from where he he started.

Let us find out the possible route taken by the salesperson by assuming he has started from city a.

Let n denote no. of cities to visit

when $n = 2$, $a \to b \to a$, no. of route = 1 [ie (2-1)!]

when $n = 3$, $a \to b \to c \to a$
$a \to c \to b \to a$, no. of route = 2 [ie (3-1)!]

when $n = 4$,
$a \to b \to c \to d \to a$
$a \to b \to d \to c \to a$
$a \to c \to b \to d \to a$
$a \to c \to d \to b \to a$
$a \to d \to c \to b \to a$
$a \to d \to b \to c \to a$

6 routes = (4-1)!

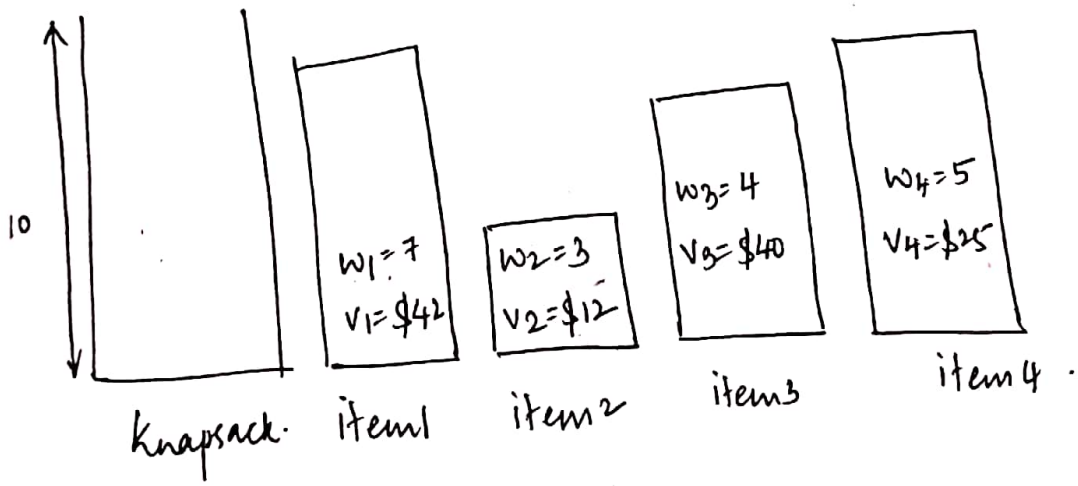
$\therefore f(n) = (n-1)!$

$\therefore$ Time complexity is $f(n) \in O(n!)$

Knapsack Problem

Given n items of known weights $w_1, \ldots w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

- The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets (ie ones with the total weight not exceeding the knapsack capacity) and finding a subset of the largest value among them.



Subset	Total weight	Total value
ϕ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	9	$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Efficiency $\Omega(2^n)$

Assignment Problem

There are n people who need to be assigned to execute n jobs, one person per job. The cost that would accrue if the ith person is assigned to the jth job is known as $C[i,j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.

q1

	Job1	Job2	Job3	Job4
Person1	9	2	7	8
Person2	6	4	3	7
Person3	5	8	1	8
Person4	7	6	9	4

q2

	Job1	Job2	Job3	Job4
Person1	10	3	8	9
Person2	7	5	4	8
Person3	6	9	2	9
Person4	8	7	10	5

<1,2,3,4> Cost = 10+5+2+5 = 22

<1,2,4,3> Cost = 10+5+9+10 = 34

<1,3,2,4> Cost = 10+4+9+5 = 28

<1,3,4,2> cost = 10+4+9+7 = 30

<1,4,2,3> Cost = 10+8+9+10 = 37

<1,4,3,2> cost = 10+8+2+7 = 27

<2,1,3,4> = 17

<2,1,4,3> = 29

<2,3,1,4> = 19

<2,3,4,1> = 25

<2,4,1,3> = 27

<2,4,3,1> = 21

<3,1,2,4> = 29

<3,1,4,2> = 31

<3,2,1,4> = 24

<3,2,4,1> = 30

<3,4,1,2> = 29

<3,4,2,1> = 33

<4,1,2,3> = 35

<4,1,3,2> = 25

<4,2,1,3> = 30

<4,2,3,1> = 24

<4,3,1,2> = 26

<4,3,2,1> = 30

The solution that takes least cost is,

<2,1,3,4> = 3+7+2+5 = 17.

Analysis The total no. of feasible solutions for $n=4$ will be 24 which is $4!$

$\therefore f(n) \in O(n!)$

Depth-first Search

→ Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited.

→ On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices a tie can be resolved arbitrarily.

→ The process continues until a dead end – a vertex with no adjacent unvisited vertices – is encountered.

→ At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

→ The algorithm eventually halts after backing up to the starting vertex, with the latter being a

→ By then all vertices have been visited
→ If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

→ Data structure used

We use stack to trace the operation of DFS.

1) we push a vertex onto the stack when the vertex is reached for the first time.
2) we pop a vertex off the stack when it becomes a dead end.

DFS traversal edges → Tree edge
DFS traversal edges → Back edge.

1] During traversal when a new unvisited vertex say v is reached for the first time from current vertex say u, then the edge (u,v) is called a <u>tree edge</u>. (represented by solid lines)

2] During traversal when an already visited vertex say v is reached from the current vertex u and if v is not the immediate

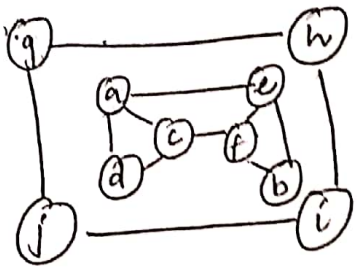
predecessor of u, then edge (u,v) is called back edge
(represented by dotted line)

Algorithm DFS(G)

// Implements a depth-first search traversal of given graph
// Input: Graph G = <V, E>
// Output: Graph G with its vertices marked with
// consecutive integers in the order they've been
// first encountered by the DFS traversal.
mark each vertex in V with 0 as a mark
of being "unvisited"
count ← 0

for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
// visits recursively all the unvisited vertices connected
// to vertex v by a path and numbers them in
// the order they are encountered via global variable
// count.
    count ← count+1 ; mark v with count
    for each vertex w in V adjacent to v do
        if w is marked with 0
            dfs(w).



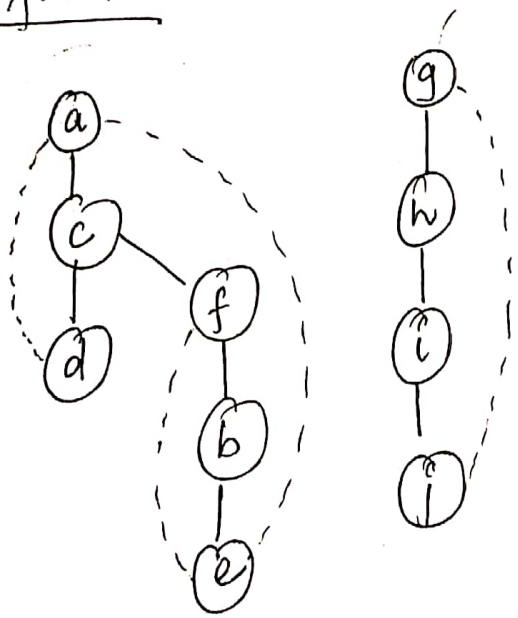
Stack	v=adj(s[top])	Nodes visited S	pop(stack)	o/p
		a		
a_1	–	a, c	–	a-c
a_1	c	a, c, d	–	c-d
a_1, c_2	d	a, c, d	$d_{3,1}$	–
a_1, c_2, d_3	–	a, c, d, f	–	c-f
a_1, c_2, f_4	f	a, c, d, f, b	–	f-b
a_1, c_2, f_4, b_5	b	a, c, d, f, b, e	$e_{6,2}$	b-e
a_1, c_2, f_4, b_5, e_6	e	a, b, c, d, e, f	$b_{5,3}$	–
a_1, c_2, f_4, b_5	–	a, b, c, d, e, f	$f_{4,4}$	–
a_1, c_2, f_4	–	a, b, c, d, e, f	$c_{2,5}$	–
a_1, c_2	–	a, b, c, d, e, f	$a_{1,6}$	–
a_1	–	a, b, c, d, e, f		–
Stack empty. So take next vertex which is not ma				
g_7	h	a, b, c, d, e, f, g, h	–	g-h
g_7, h_8	i	a, b, c, d, e, f, g, h, i	–	h-i
g_7, h_8, i_9	j	a, b, c, d, e, f, g, h, i, j	–	i-j
g_7, h_8, i_9, j_{10}	–	a, b, c, d, e, f, g, h, i, j	$j_{10,7}$	–
g_7, h_8, i_9	–	"	$i_{9,8}$	–
g_7, h_8	–	"	$h_{8,9}$	–
g_7	–	"	$g_{7,10}$	

Traversal stack

	$e_{6,2}$	
	$b_{5,3}$	$j_{10,7}$
$d_{3,1}$	$f_{4,4}$	$i_{9,8}$
$c_{2,5}$		$h_{8,9}$
$a_{1,6}$		$g_{7,10}$

In traversal stack, the first subscript number indicates the order in which a vertex was visited ie pushed onto the stack; the second one indicates the order in which it became a dead-end. ie popped off the stack.

DFS forest



BFS (Breadth First Search)

The Breadth First Search proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited ~~two~~ vertices two edges apart from it and so on, until all the vertices in the same connected component as the starting vertex are visited. If there ~~are~~ still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

BFS traversal edges
- Tree edge
- Cross edge

Tree edge :-

During traversal when a new unvisited vertex say v is reached for the first time from a current vertex say u, then edge (u, v) is called a tree edge. In the tree edge (u, v) the vertex u is the parent and vertex v is the child. The edges are represented using solid lines.

Cross edge

During traversal when an already visited vertex say v is reached from current vertex u and if v is not the immediate predecessor of u, then edge (u,v) is called cross edge. Represented by dotted lines.

Definations

Tree edge :- whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called **tree edge**

Cross edge

If an edge leading to a previously visited vertex other than its immediate predecessor (ie its parent in the tree) is encountered the edge is noted as a **cross edge**

: Data structure

- The queue which provides first in first out property is very useful in traversing the graph in BFS.
- When a vertex is reached for the first time, it is inserted into rear end of queue.
- When we get a dead end (ie the vertex is already explored), we delete a vertex from front of queue.

∴ The BFS yields only one ordering of vertices ie the order in which the vertices are inserted into queue is same as the order in which the vertices are deleted from queue.

Algorithm BFS(G)

// Implements a breadth-first search traversal of a given graph

// Input : Graph $G = \langle V, E \rangle$

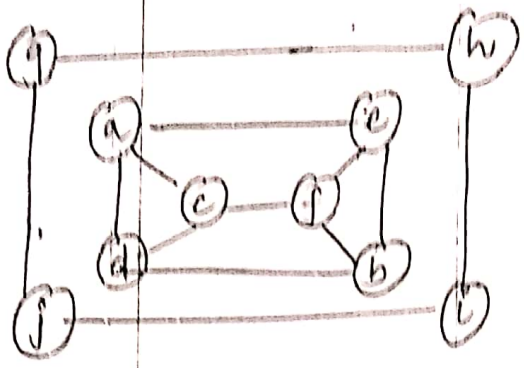
// Output : Graph G with its vertices marked with // consecutive integers in the order they have // been visited by BFS traversal

mark each vertex in V with 0 as a mark of being "unvisited"

contd...

```
for each vertex v in V do
    if v is marked with 0
        bfs(v)
```

```
bfs(v)
// visits all the unvisited vertices connected to
// vertex v by a path and assigns them the
// numbers in the order they are visited via
// global variable count
count ← count + 1; mark v with count & initialize
                   queue with v
while the queue is not empty do
    for each vertex w in V adjacent to front vertex do
        if w is marked with 0
            count ← count + 1; mark w with count
            add w to the queue
    remove the front vertex from the queue
```



Soln :-

Insert source vertex 'a' into Q and add a to S as shown in table.

	u = del(Q)	V = adj to u	Node visited S	Q	o/p T(u,v)
	–	–	a	a	–
Stage 1	a	c, d, e	a, c, d, e	c, d, e	a-c a-d a-e
~~Stage 2~~					
Stage 2	c	f	a, c, d, e, f	d, e, f	c-f
Stage 3	d	–	a, c, d, e, f	e, f	–
Stage 4	e	b	a, b, c, d, e, f	f, b	e-b
Stage 5	f	–	a, b, c, d, e, f	b	–
Stage 6	b	–	a, b, c, d, e, f	–	–

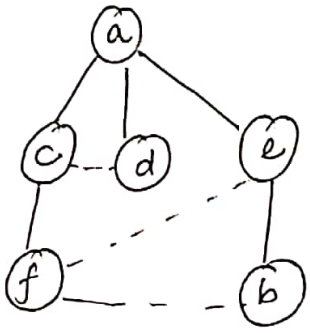
Note : Now queue is empty. But there are some vertices that are not visited.

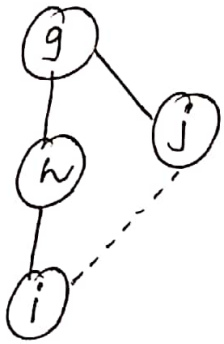
Initial step

—	—	a, b, c, d, e, f, g	g	—
g	h, j	a, b, c, d, e, f, g, h, j	h, j	g-h, g-j
h	i	a, b, c, d, e, f, g, h, i, j	j, i	h-i
j	—	a, b, c, d, e, f, g, h, i, j	i	—
i	—	a, b, c, d, e, f, g, h, i, j	—	—

ordering

a, c2 d3 e4 f5 b6
g7 h8 j9 i10

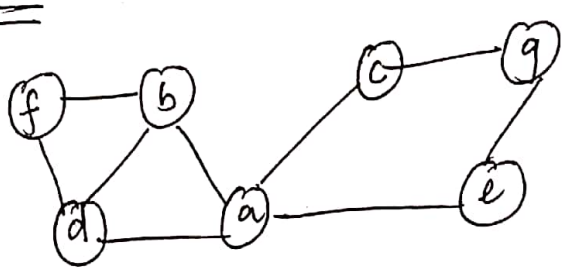




	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
edge types (undirected graphs)	tree & back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	minimum-edge paths, connectivity, acyclicity
Efficiency for adjacent matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
efficiency for adjacent lists	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$

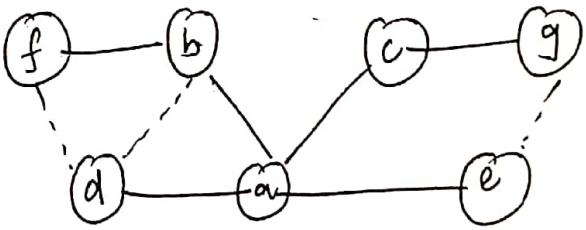
Eg

BFS



ordering

$a_1 b_2 c_3 d_4 e_5 f_6 g_7$.



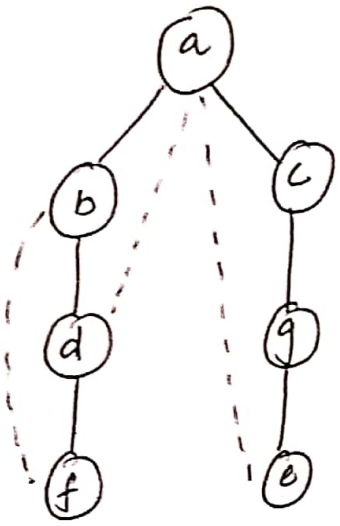
DFS

$f_{4,1}$ $e_{7,4}$

$d_{3,2}$ $g_{6,5}$

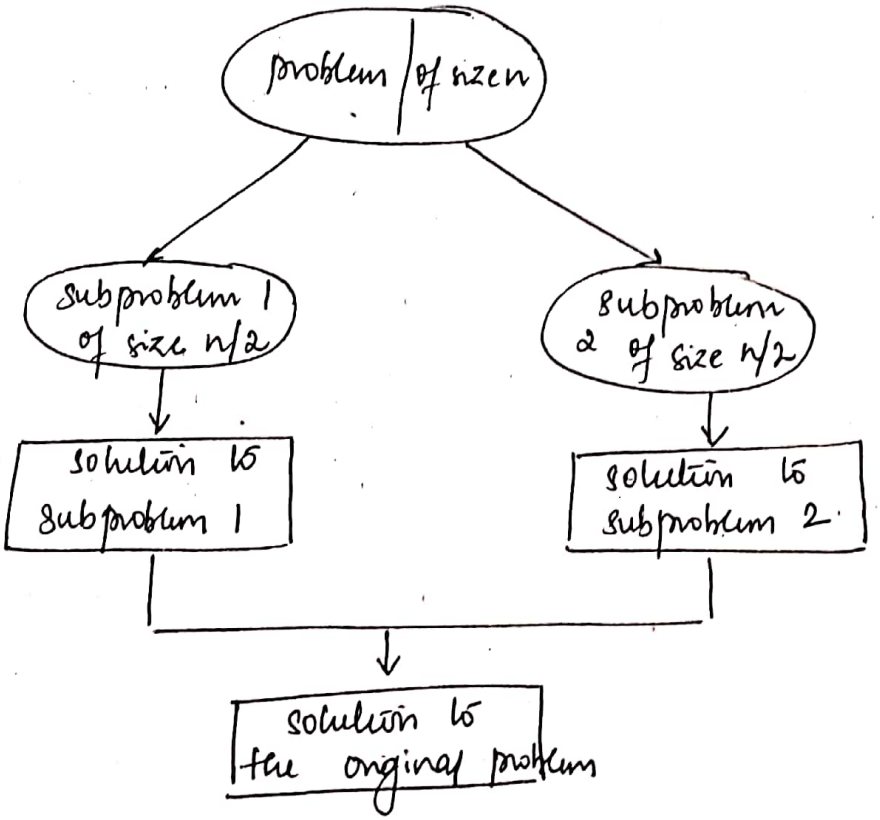
$b_{2,3}$ $c_{5,6}$

$a_{1,7}$



Divide & Conquer (4)

Divide and conquer is a top-down technique for designing algorithms that consist of dividing the problem into smaller sub problems. The solutions of all smaller problems are then combined to get a solution for the original problem.



In most typical case of divide-and-conquer a problem instance of size n is divided into two instances of size n/2. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here a and b are constants; a≥1 and b>1)

Master theorem

The generalized relation for divide & conquer method is :-

$$T(n) = aT(n/b) + f(n) \quad \text{---(1)}$$

and the time complexities can be obtained using following relation :-

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases} \quad \text{---(2)}$$

for sum of n numbers, recurrence relation is,

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2)+T(n/2)+1 & \text{otherwise} \end{cases}$$

$$f(n) = n^d = 1 = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2)+1 & \text{otherwise} \end{cases}$$

n^d n^d

$\therefore T(n) = 2T(n/2) + 1$

where $a = 2$, $b = 2$ & $f(n) = 1 = n^0$

$= n^d$

$\therefore d = 0$

since $a > b^d$ $\therefore T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_2 2})$

$= \theta(n)$

Mergesort

Mergesort sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Algorithm Mergesort ($A[0..n-1]$)

// Sorts array $A[0..n-1]$ by recursive mergesort.
// Input: An array $A[0..n-1]$ of orderable elements
// Output: Array $A[0..n-1]$ sorted in nondecreasing
// order.

if $n > 1$

copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$

Mergesort ($B[0..\lfloor n/2 \rfloor - 1]$)

Mergesort ($C[0..\lceil n/2 \rceil - 1]$)

Merge (B, C, A)

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

// Merges two sorted arrays into one sorted array.
// Input: Arrays $B[0..p-1]$ and $C[0..q-1]$) both sorted
// Output:- Sorted array $A[0..p+q-1]$ of the elements

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$

```
while i<p and j<q do
    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]
```

Analysis

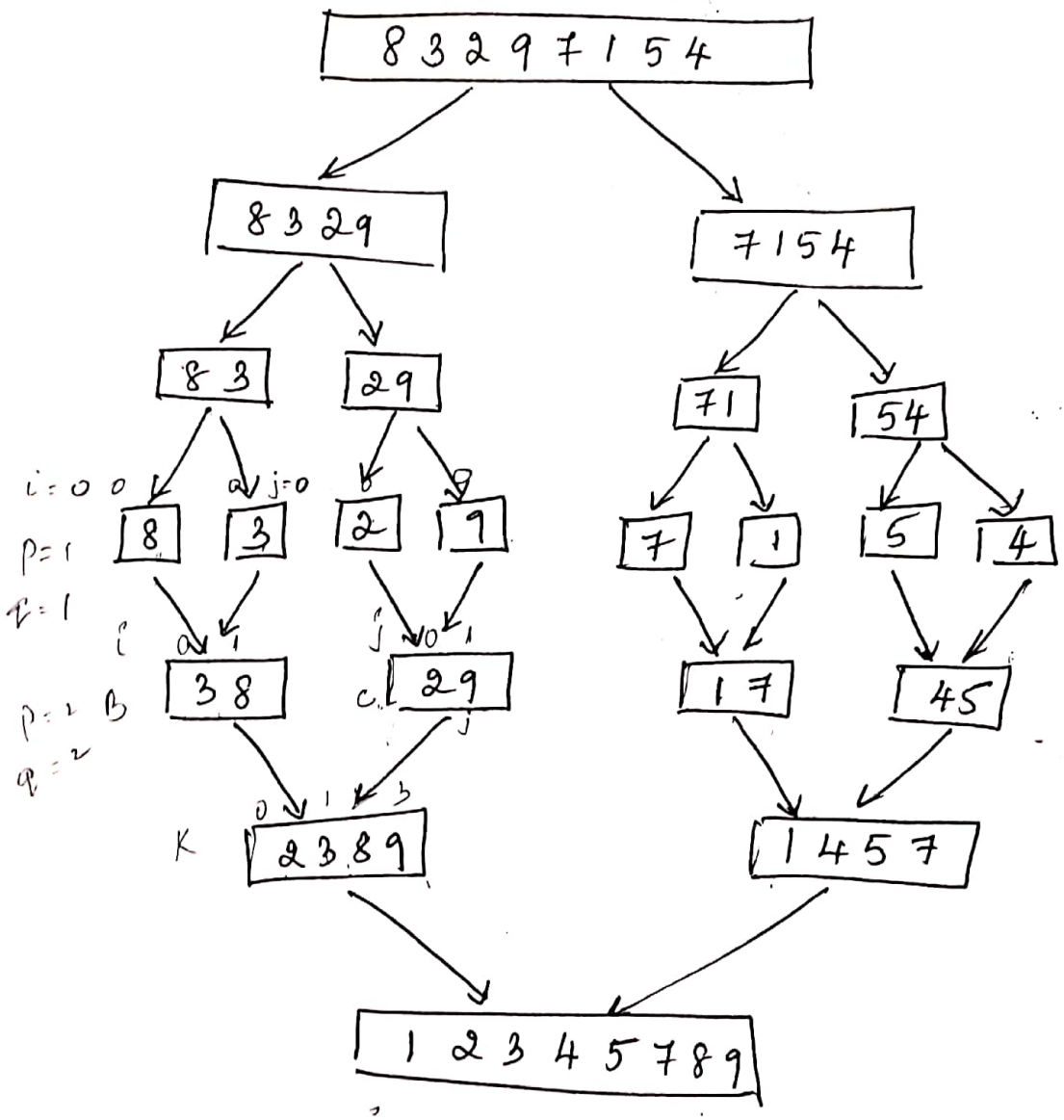
The recurrence relation for no. of key comparisons C(n) is:-

$C(n) = 2C(n/2) + C_{merge}(n)$ for $n>1$ $\quad C(1)=0$.

$C(n) = C(n/2) + C(n/2) + C_{merge}(n)$ for $n>1$ $\quad C(1)=0$.

- $C(n/2)$: Time required to sort left part of the array
- $C(n/2)$: Time required to sort right part of the array.

$C_{merge}(n)$, the no. of key comparisons performed during the merging stage.



At each step, exactly one comparison is made, after which total no. of elements in the two arrays still needed to be processed is reduced by one.

$\therefore C_{merge}(n) = n-1$

$$C_{worst}(n) = 2C_{worst}(n/2) + n-1 \quad \text{for } n>1$$

$$C_{worst}(1) = 0.$$

$a = 2$, $b = 2$, $f(n) = n$

$= n^d$

$\therefore d = 1$, $\underline{a = b^d}$

$$T(n) = \theta(n^d \log_b n)$$

$$= \theta(n^1 \log_2 n)$$

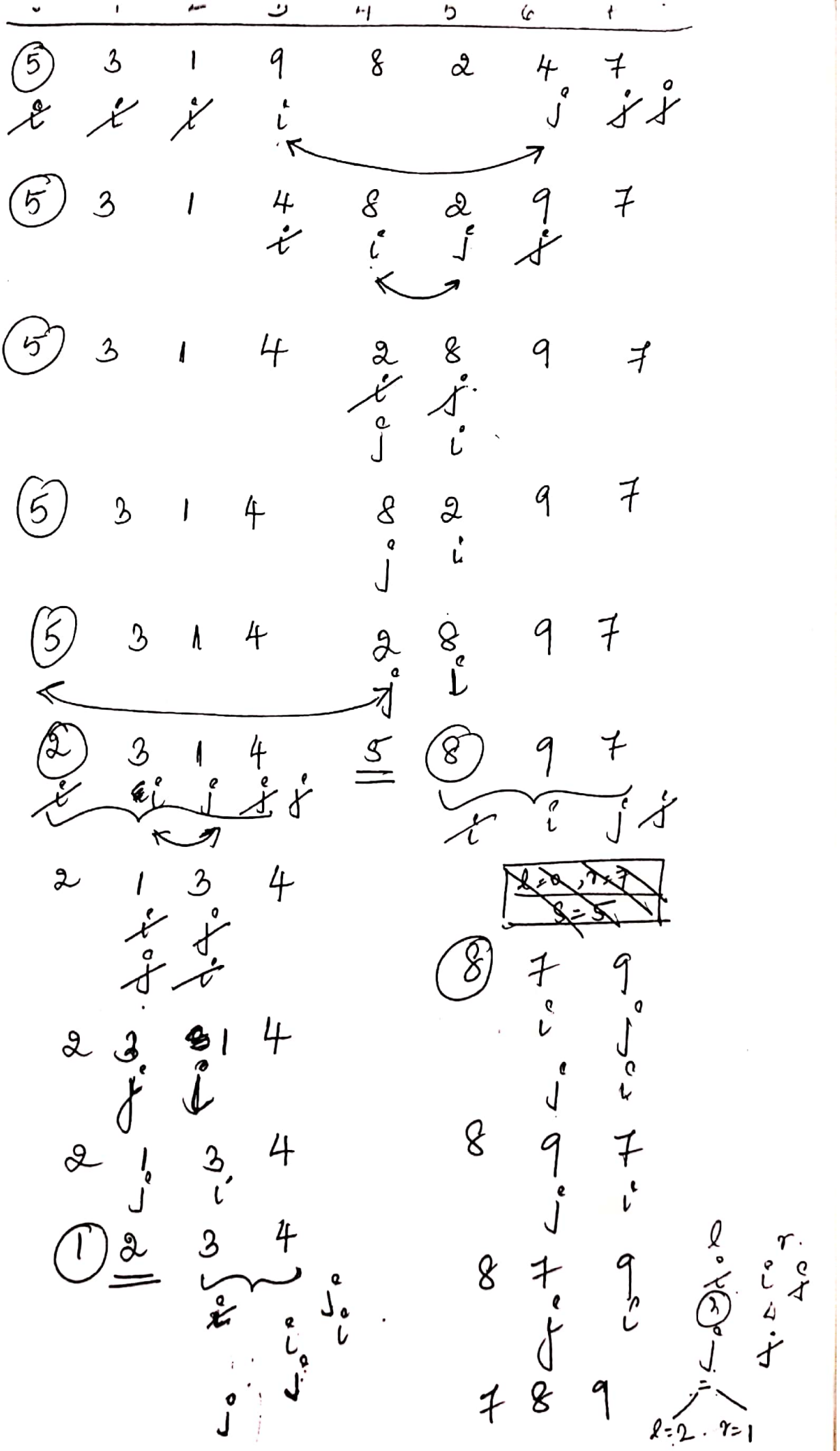
$$\boxed{T(n) = \theta(n \log_2 n)}$$

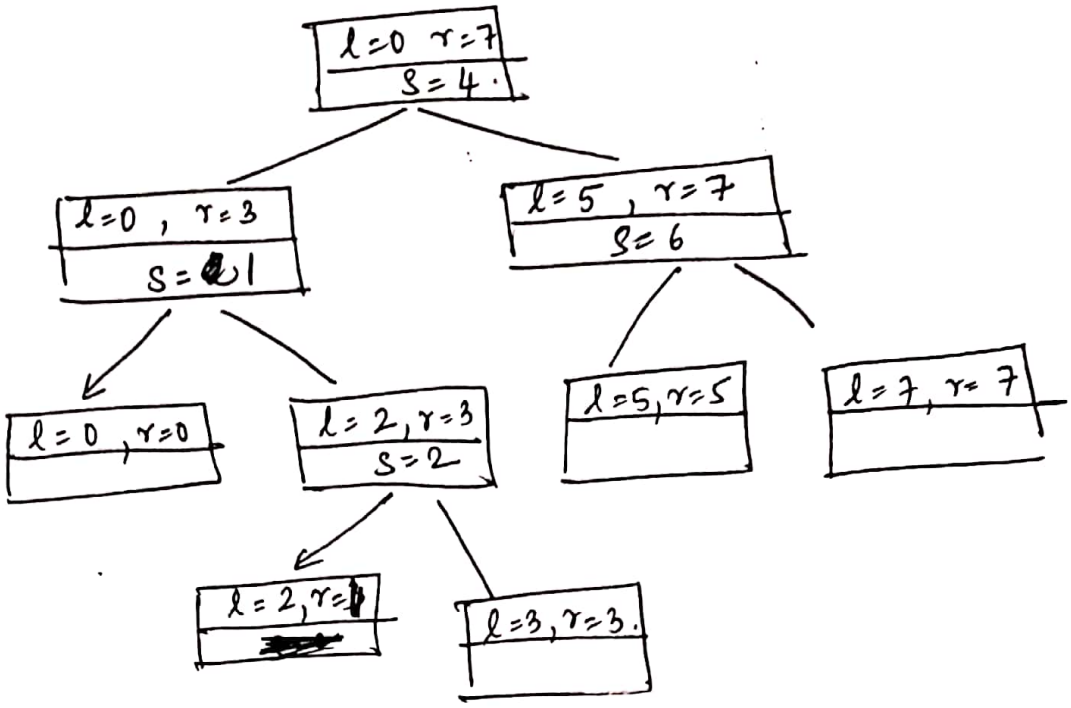
Quicksort

- Quicksort is another important sorting algorithm that is based on divide and conquer approach. Unlike mergesort which divides its input's elements according to their position in the array, quicksort divides them according to their value.
- It rearranges elements of a given array $A[0..n-1]$ to achieve its partition, a situation where all elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \geq A[s]}$$

- After partition has been achieved $A[s]$ will be in its final position in the sorted array and we can continue sorting the two subarrays of the elements preceding & following $A[s]$ independently.





Algorithm Quicksort(A[l...r])

// Sorts a subarray by quicksort

// Input: A subarray A[l...r] of A[0...n-1], defined by its left and right indices l and r.

// Output: subarray A[l..r] sorted in nondecreasing order

```
if l < r
    s ← Partition(A[l...r]) // s is a split position
    Quicksort(A[l...s-1])
    Quicksort(A[s+1..r])
```

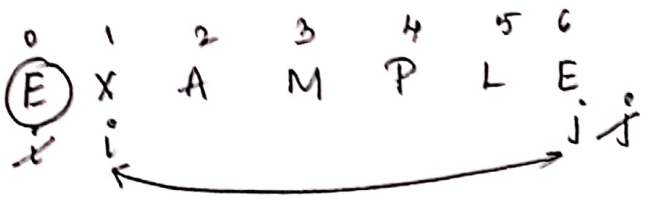
Algorithm Partition(A[l...r])

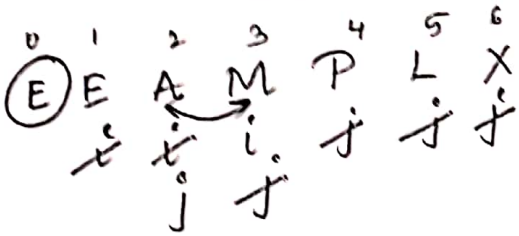
// Partitions a subarray by using its first element as its pivot

// Input: A subarray A[l...r] of A[0...n-1] defined by its left & right indices l and r (l<r)

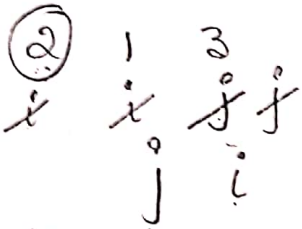
// Output: A partition of A[l...r] with the split position returned as this function's value.

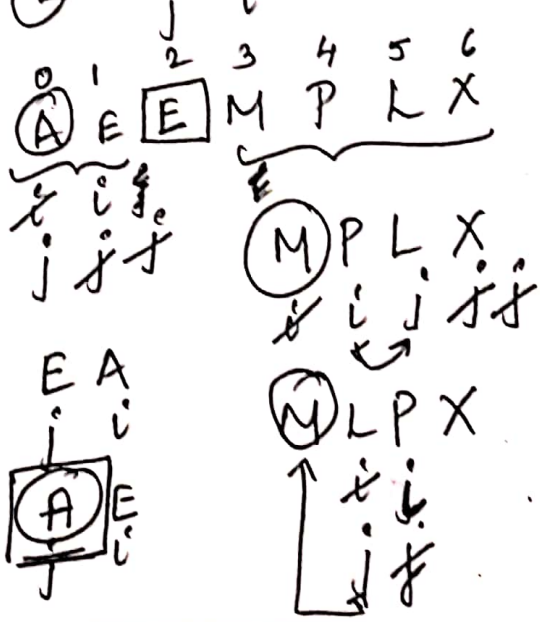
```
p ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j-1 until A[j] ≤ p
    swap (A[i], A[j])
until i ≥ j
swap (A[i], A[j]) //undo last swap when i ≥ j
swap (A[l], A[j])
return j
```



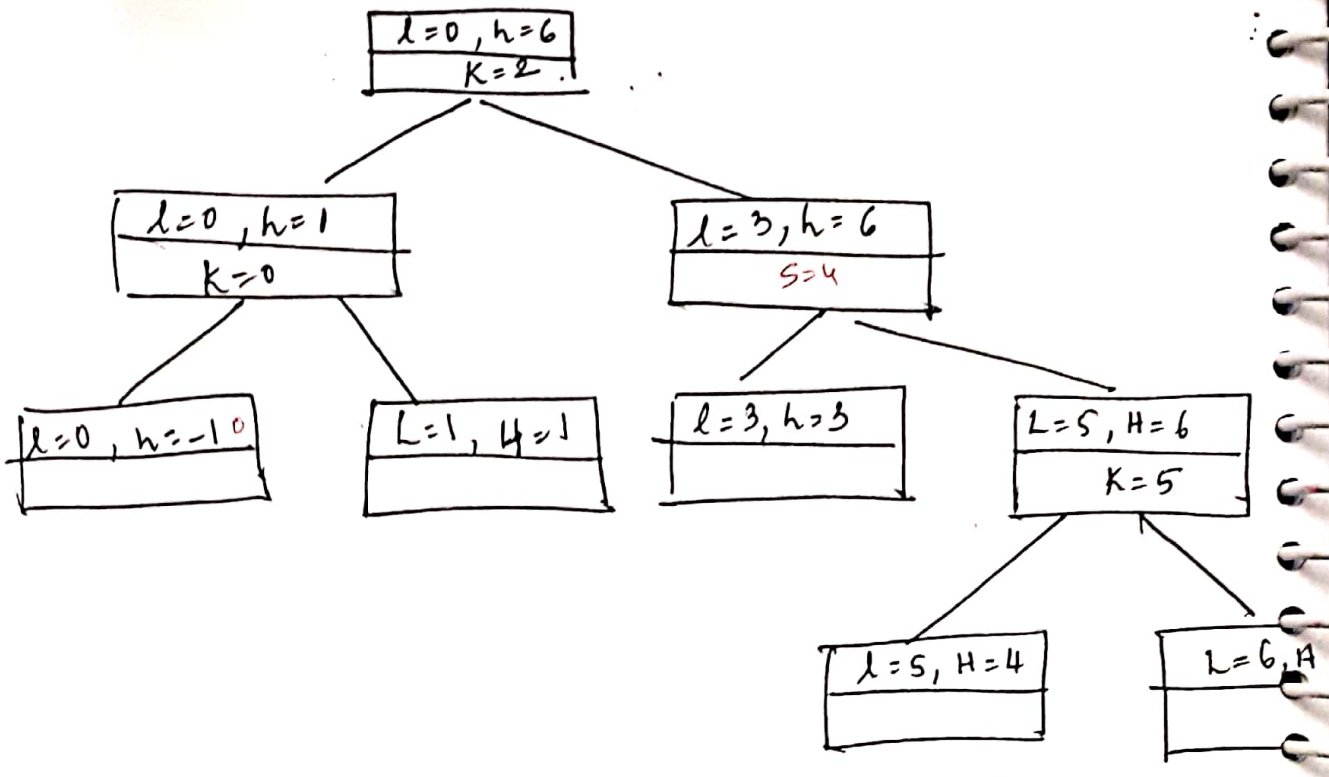


L M P X





n+1



Analysis of Quicksort

Best case time efficiency

This case occurs, if the key element is present exactly at the center dividing the array into two equal parts

$$\therefore T(n) = \begin{cases} 0 & \text{if } n=1 \quad \text{(sorting not req} \\ T(n/2) + T(n/2) + n & \text{otherwise} \end{cases}$$

- $T(n/2)$ → Time required to sort the left part of the array
- $T(n/2)$ → Time required to sort the right part of the array.
- n → Time required to partition n elements

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Using Master's theorem

$a = 2$, $b = 2$, and $f(n) = n = n^1 = n^d$

$\therefore$ we get $a = 2$, $b^d = 2^1 = 2$

$\therefore a = b^d$

$$\therefore T(n) = \theta(n^d \log_b n) = \theta(n^1 \log_2 n)$$

Worst case efficiency

The worst case occurs when at each invocation of the procedure, the current array is partitioned into two sub arrays with one of them being empty. This situation occurs if all the elements are arranged in ascending order or descending order.

eg [22, 33, 44, 55]

22 [33, 44, 55]

22 33 [44 55]

22 33 44 [55]

In general if the array has n elements, after partitioning n-1 elements will be there towards right and 0 elements will be there towards left of the key element.

∴ The recurrence relation is,

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(0)+T(n-1)+n & \text{otherwise} \end{cases}$$

$\therefore T(n) = T(0) + T(n-1) + n$

$= T(n-1) + n$

$= T(n-2) + (n-1) + n$ → By replacing n by n-1

$= T(n-3) + (n-2) + (n-1) + n$ → By repla... n-1 by n...

$= \cancel{T(n-1)}$ $0+1+2+\ldots+(n-2)+(n-1)+n$

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

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

Average case

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} (n+1) + C_{avg}(s) + C_{avg}(n-1)$$ for

$C_{avg}(0) = 0$ & $C_{avg}(1) = 0$

Multiplication of Large Integers

Some cryptology and modern applications require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer.

Multiplication of large numbers use divide & conquer technique

Eg 23 and 14.

$23 = 2\times10^1 + 3\times10^0$

$14 = 1\times10^1 + 4\times10^0$

$23*14 = (2*10^1 + 3*10^0) * (1*10^1 + 4*10^0)$.

$= (2*1)10^2 + (2*4 + 3*1)10^1 + (3*4)10^0$

$= \underline{\underline{322}}$ → But uses same technique as the pen-and-pencil algorithm.

We can compute the middle term with just one digit multiplication by taking advantage of the products $2*1$ and $3*4$ that need to be computed anyway.

$2*4 + 3*1 = (2+3)*(1+4) - 2*1 - 3*4$

Thus in general for any pair of two digit integers $a = a_1 a_0$ and $b = b_1 b_0$ their product c can be computed by the formula.

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0$$

where

$c_2 = a_1 * b_1$ is the product of their first digits

$c_0 = a_0 * b_0$ is the product of their second digits

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 & c_0

Consider the following example

$a = 1234$ and $b = 5678$.

we split the given numbers equally into two parts.

$a_1 = 12$ and $a_0 = 34$ ——— (1)

$b_1 = 56$ and $b_0 = 78$.

$\therefore a = 10^2 * a_1 + 10^0 * a_0$ ——— (2).

$a = 10^2 * 12 + 10^0 * 34 = 1234$

Similarly

$b = 10^2 * b_1 + 10^0 * b_0$ ——— (3)

$= 10^2 * 56 + 10^0 * 78 = 5678$ ——— (4)

Now the two numbers 1234 & 5678 can be multiplied using ② & ③ as shown below:-

$$a*b = (10^2 a_1 + 10^0 a_0)(10^2 b_1 + 10^0 b_0)$$
$$= 10^4 a_1 b_1 + 10^2 (a_1 b_0 + b_1 a_0) + 10^0 a_0 b_0 \quad \text{—— (4)}$$

Substituting the values

$$1234 * 5678 = \underline{7006652}$$

In above equation we have four multiplication which has to be reduced to three multiplication

Let us consider,

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 \quad \text{— (5)}$$

substituting (5) in (4)

$$\therefore a * b = 10^4 a_1 b_1 + 10^2 [(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0] + 10^0 a_0 b_0$$

$$= 10^4 a_1 b_1 + 10^2 [(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0] + a_0 b_0 \quad \text{—— (6)}$$

In above equation we have three multiplication

→ $a_1 b_1$

→ $a_0 b_0$

→ $(a_1 + a_0)(b_1 + b_0)$

In general

$$a*b = (a_1 10^{n/2} + a_0)(b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$
$$c_1 10^n + c_1 10^{n/2} + c_0$$

Where

$c_2 = a_1 * b_1$ is product of their first halves

$c_0 = a_0 * b_0$ is product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is product of the sum of the a's halves & the sum of b's halves minus the sum of c_2 & c_0.

Analysis

Two n-digit numbers requires 3 multiplication of n/2 digit numbers and recurrence relation is given by:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 3T(n/2) & \text{otherwise} \end{cases}$$

$T(n) = 3.\ T\left(\frac{n}{2}\right)$ → Replacing n by n/2 we have $T\left(\frac{n}{2}\right) = 3T\left(\frac{n}{2^2}\right)$

$= 3.3\left(\frac{n}{2^2}\right)$

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

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

$\vdots$

$= 3^i\left(\frac{n}{2^i}\right)$ • substitute $n = 2^i$

$= 3^i \cdot 1 = 3^i$

$\therefore \quad T(n) = 3^i$

From $2^i = n$, we have taking log on both sid

$$\log 2^i = \log n$$

$$i \log_2 2 = \log_2 n$$

$$i = \log_2 n$$

$$\therefore T(n) = 3^{\log_2 n}$$

$$= n^{\log_2 3}$$

$$= n^{1.585}$$

Note $a^{\log_2 b} = b^{\log_2 a}$

OR

$M(n) = 3M(n/2)$ for $n > 1$ $\quad M(1) = 1$

Solving it by backward substitution for $n = 2^k$ yields

$$M(2^k) = 3M(2^{k-1})$$

$$= 3[3M(2^{k-2})]$$

$$= 3^2 M(2^{k-2})$$

$$= \ldots$$

$$= 3^i M(2^{k-i})$$

$$= \ldots = 3^k M(2^{k-k})$$

$$= 3^k.$$

$n = 2^k \Rightarrow k = \log_2 n$ $\Rightarrow$ $\boxed{a^{\log_b c} = c^{\log_b a}}$

$$M(n) = 3^{\log_2 n}$$

$$= n^{\log_2 3} \approx n^{1.585}$$

Strassen's Matrix Multiplication

We can apply divide-and-conquer approach to find the product C of two 2-by-2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

This is accomplished by using the following formulas:-

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where,

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 - multiplications

18 - additions / subtractions.

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix}$$

$a_{00} = 1$ $\quad b_{00} = 8$

$a_{01} = 2$ $\quad b_{01} = 7$

$a_{10} = 5$ $\quad b_{10} = 1$

$a_{11} = 6$ $\quad b_{11} = 2$

$m_1 = (1+6)(8+2) = 70$

$m_2 = (5+6)\,8 = 88$

$m_3 = 1(7-2) = 5$

$m_4 = 6 * (1-8) = -42$

$m_5 = (1+2) * 2 = 6$

$m_6 = (5-1) * (8+7) = 60$

$m_7 = (2-6) * (1+2) = -12$

$$C = \left[\begin{array}{c|c} 70 - 42 - 6 - 12 & 5+6 \\ \hline 88 - 42 & 70 - 5 - 88 + 60 \end{array}\right]$$

$$= \begin{bmatrix} 10 & 11 \\ 46 & 47 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 8 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 1 & 7 \end{bmatrix}.$$

Apply Strassen algorithm to compute

$$\left[\begin{array}{cc|cc} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ \hline 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{array}\right] * \left[\begin{array}{cc|cc} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ \hline 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{array}\right]$$

A_{00} A_{01} A_{10} A_{11} $\quad$ B_{00} B_{01} B_{10} B_{11}

$$\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} - \begin{bmatrix}$$

Time efficiency

$M(n) = 7M(n/2)$ for $n > 1$

$M(1) = 1$

Since $n = 2^k$.

$M(2^k) = 7M(2^{k-1})$

$= 7[7M(2^{k-2})]$

$= 7^2 M(2^{k-2}) = \ldots 7^i M(2^{k-i})$

$= 7^k M(2^{k-k})$

$= 7^k$.

Since $n = 2^k$

$k = \log_2 n$

$M(n) = 7^{\log_2 n}$

$= 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx n^{2.807}$

Note: This algorithm works fine if size of the matrix is always a power of 2.

If the size of the matrix is not a multiple of 2, add sufficient zeros both in rows & columns so that size of the matrices are always multiples of 2.