[Algorithm]

Asymptotic Notation

O() - ("Big O")

As discussed in the previous lecture, often we are most concerned about the *worst case* behavior of an algorithm. Thus we define an *asymptotic upper bound* (although not necessarily tight) for a function *f(n)* denoted O() as follows: Given two functions *f(n)* and *g(n)*

$$O(g(n)) = \{ f(n) : \exists\ c_2, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0 \}$$

Thus O() describes how bad our algorithm can grow, i.e. is no worse than. Note: Often it is enough to find a loose upper bound to show that a given algorithm is better than (or at least no worse than) another.

Ω() - ("Big Omega")

We can similarly define an *asymptotic lower bound* (again not necessarily tight and often not particularly useful) for a function *f(n)* denoted Ω() as follows: Given two functions *f(n)* and *g(n)*

$$\Omega(g(n)) = \{ f(n) : \exists\ c_1, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \quad \forall n \geq n_0 \}$$

Thus Ω() describes how good our algorithm can be, i.e. is no better than. For this bound to be meaningful, it is often best to find a tight lower bound.

Θ() - ("Big Theta")

Finally we can define an *asymptotically tight bound* for a function *f(n)* denoted Θ() as follows: Given two functions *f(n)* and *g(n)* both *asymptotically non-negative*

$$\Theta(g(n)) = \{ f(n) : \exists\ c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0 \}$$

This notation indicates that *g(n)* is both a *lower bound* (best case) and *upper bound* (worst case) for *f(n)*, thus the algorithm has a consistent growth independent of the particular input set.

Big oh notation:

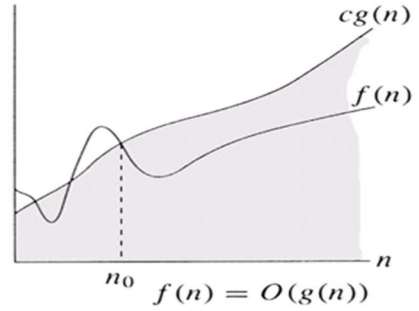
The big oh notation is denoted by ‘O’. it is the method of representing the upper bound of algorithm’s running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Let f(n) and g(n) be non-negative functions. Let n_o and constant c are two integers such that n_o denotes some value of input and $n>n_o$.

Similarly c is some constant such that c>0. We can write

$$F(n) \leq c * g(n)$$

F(n) is the big oh of g(n). it is also denoted as f(n) ϵ O(g(n)). In other words f(n) is less than g(n) if g(n) is multiple of some constant c.



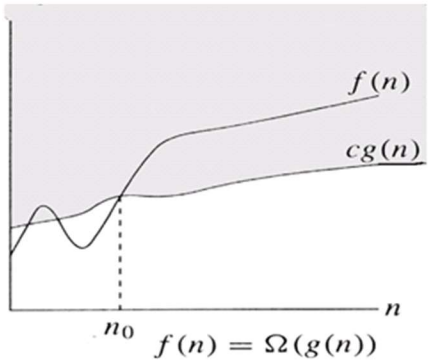
Omega notation:

The omega notation is denoted by ‘Ω’. It is the method of representing the lower bound of algorithm’s running time. Using omega notation we can give shortest amount of time taken by the algorithm to complete.

The function f(n) is said to be in Ω(g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$$F(n) \geq c * g(n) \qquad \text{for all } n \geq n_o$$

F(n) is the omega of g(n). it is also denoted as f(n) ϵ Ω(g(n)). Graphically we can represent it as:



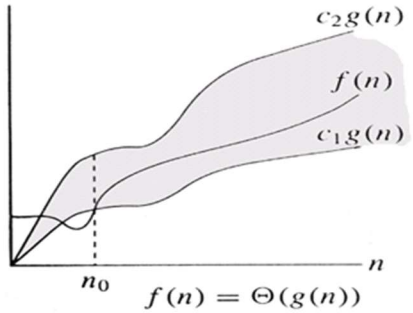
Theta notation:

The theta notation is denoted by ‘θ’. By this method the running time is between lower and upper bound.

Let f(n) and g(n) be non-negative functions. There are two positive constants namely c_1 and c_2. Such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

It is also denoted as f(n) ϵ θ(g(n)).

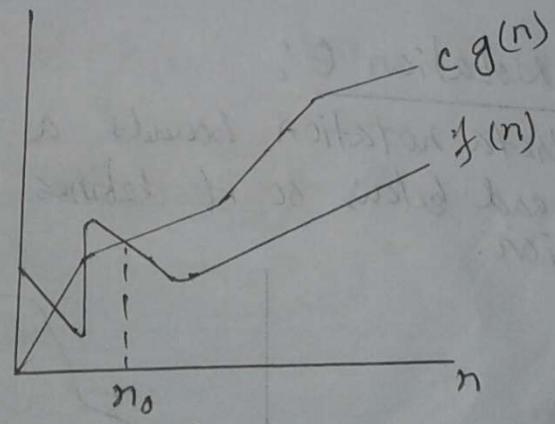


Asymptotic Notation

Upper Bound → O (Big O)
Lower Bound → Ω (Big Omega)
Tight Bound → Θ (Big Theta)

Big O Notation:

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.



$f(n) = O(g(n))$

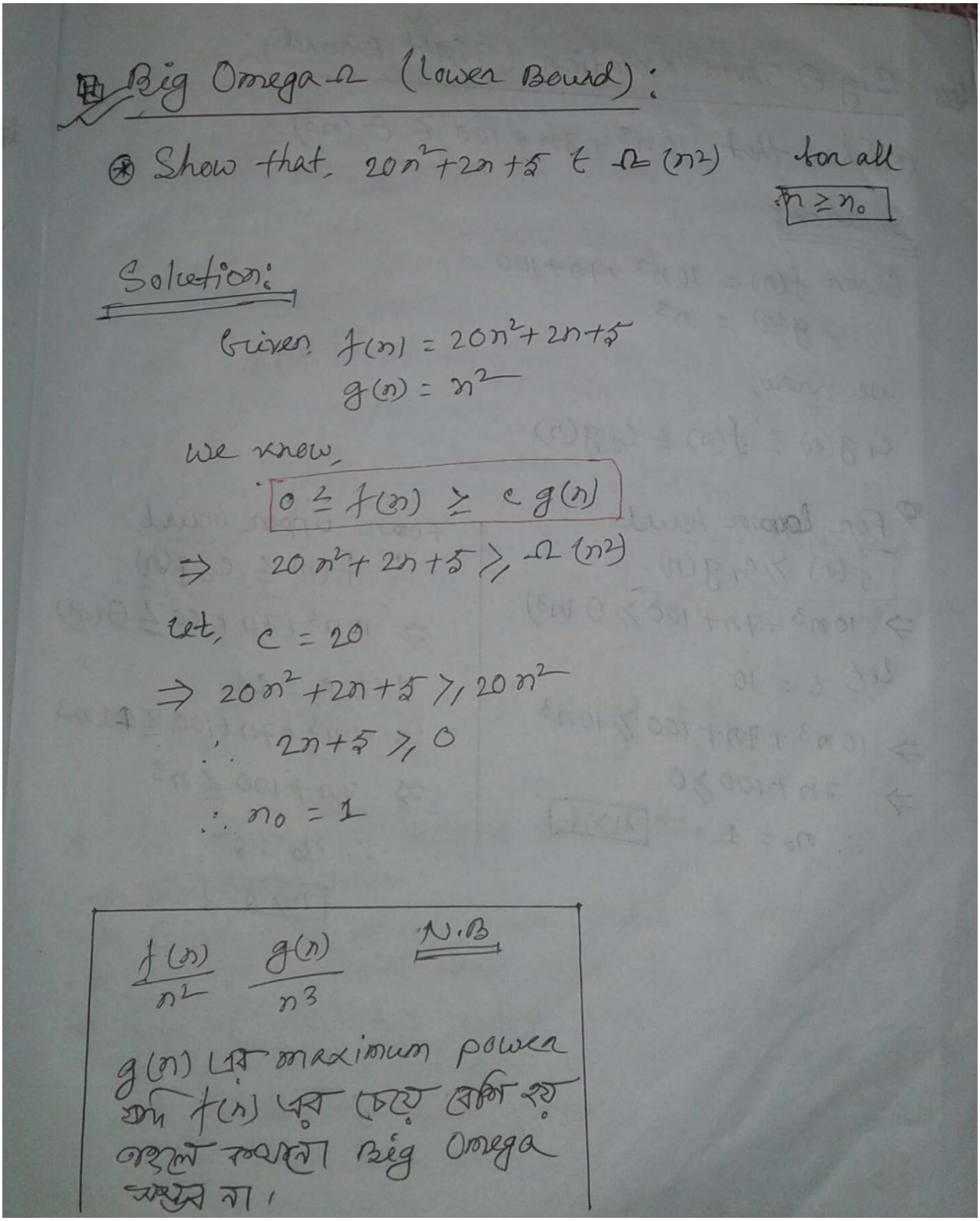
→ $g(n)$ is an asymptotic upper bound for $f(n)$.

Formula:

$c \cdot g(n) > f(n) > 0$

OR, $f(n) \leq c \cdot g(n)$

Example: Asymptotic Notation – (Big O , Big Omega , Big Theta)



Big Omega Ω (Lower Bound):

⊛ Show that, $20n^2+2n+5 \in \Omega(n^2)$ for all $n \geq n_0$

Solution:

Given, $f(n) = 20n^2+2n+5$

$g(n) = n^2$

we know,

$$0 \leq f(n) \geq c\,g(n)$$

$\Rightarrow 20n^2+2n+5 \geq \Omega(n^2)$

let, $c = 20$

$\Rightarrow 20n^2+2n+5 \geq 20n^2$

$\therefore 2n+5 \geq 0$

$\therefore n_0 = 1$

N.B

$\frac{f(n)}{n^2}$ $\frac{g(n)}{n^3}$

$g(n)$ এর maximum power যদি $f(n)$ এর চেয়ে বেশি হয় তাহলে কখনো Big Omega সম্ভব না।

Big Θ Theta Notation (Tight Bound):

⊛ Show that, $10n^3 + 7n + 100 \in \Theta(n^3)$

Solution:

Given, $f(n) = 10n^3 + 7n + 100$

$g(n) = n^3$

We know,

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

For lower bound,

$f(n) \geq c_1 g(n)$

$\Rightarrow 10n^3 + 7n + 100 \geq \Theta(n^3)$

Let, $c = 10$

$\Rightarrow 10n^3 + 7n + 100 \geq 10n^3$

$\Rightarrow 7n + 100 \geq 0$

$\therefore n_0 = 1$

$n > 1$

For upper bound,

$f(n) \leq c_2 g(n)$

$\Rightarrow 10n^3 + 7n + 100 \leq \Theta(n^3)$

Let, $c = 11$

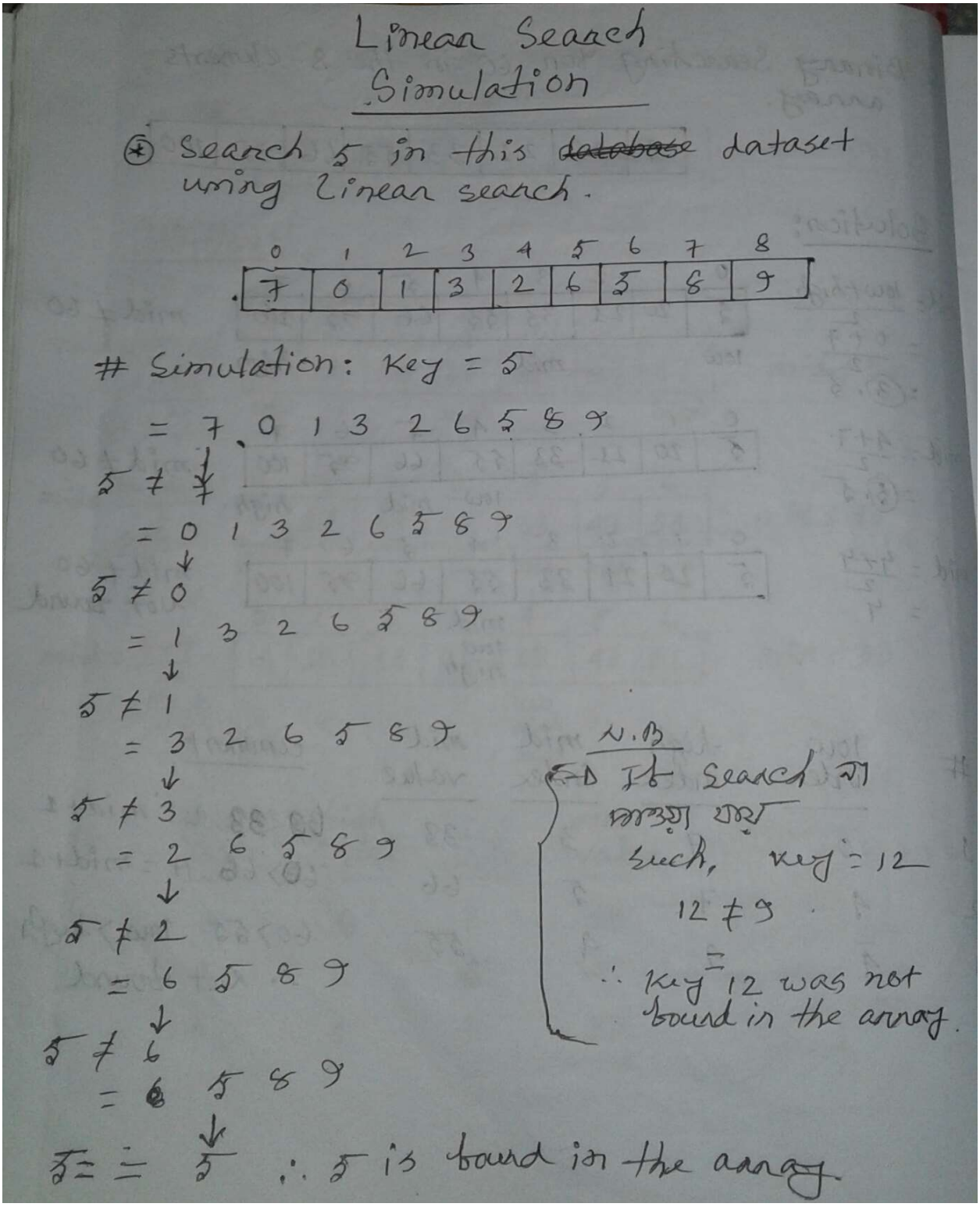
$\Rightarrow 10n^3 + 7n + 100 \leq 11n^3$

$\Rightarrow 7n + 100 \leq n^3$

$\therefore n_0 = 5$

$n > 5$

Linear Search



Binary Search

Binary Search – Algorithm

⇒ Formula:

$$\frac{mid + high}{2}$$

→ Mid-Point value is not the searching value & searching value > Mid-Points value.

So, $R = Mid\text{-}Point + 1$

Then, ignore all the values of the left side of new L.

→ Mid-Point value is not the searching value & searching < Mid-Points value

So, $R = Mid\text{-}Point - 1$

Then, ignore all the values of the right side of new R

Prerequisite of Binary Search:

① The middle element is looked to check if it is greater than or less than the value to be searched.

② Accordingly, search is done to either half of the given list.

⊛ Binary searching for 33 in the 10-element array.

6	13	14	25	33	43	51.

Solution:

$mid = \frac{low+hi}{2} = \frac{0+6}{2} = 3$

0	1	2	3	4	5	6
6	13	14	25	33	43	51

low mid high

mid ≠ 33

$mid = \frac{low+high}{2} = \frac{4+6}{2} = 5$

0	1	2	3	4	5	6
6	13	14	25	33	43	51

low mid high

mid ≠ 33

$mid = \frac{4+4}{2} = 4$

0	1	2	3	4	5	6
6	13	14	25	33	43	51

low high mid

mid = 33

#	low index	high index	mid index	mid value	comment
1	0	6	3	25	33 > 25 l = mid+1
2	4	6	5	43	33 < 43 h = mid+1
3	4	4	4	33	Found

⊛ Binary Searching for 60 in the 8-elements array.

5	20	21	33	55	66	95	100

Solution:

$mid = \frac{low + high}{2}$

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

$= ③\ 5$

0	1	2	3	4	5	6	7
5	20	21	33	55	66	95	100

low mid high

mid ≠ 60

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

$= ⑤\ 5$

0	1	2	3	4	5	6	7
5	20	21	33	55	66	95	100

low mid high

mid ≠ 60

$mid = \frac{4+4}{2}$

$= 4$

0	1	2	3	4	5	6	7
5	20	21	33	55	66	95	100

mid
low
high

mid ≠ 60

∴ Not found

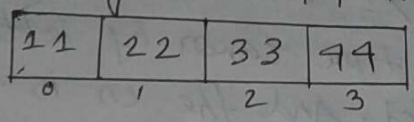
#	low index	high index	mid index	mid value	comment
1	0	7	3	33	60 > 33 L = mid + 1
2	4	7	5	66	60 < 66 H = mid - 1
3	4	4	4	55	60 > 55 low > high
4	5	4	—	—	∴ Not found

N.B

→ If low > high, Then not found

Binary Search Algorithm

(*) Binary searching for 44 in the 4 element array.



Solution:

#	low index	high index	mid index	mid value	comment
1	0	3	1	22	44 > 22 L = mid+1
2	2	3	2	33	44 > 33 L = mid+1
3	4	4	4	44	Found

Time Complexity of Binary Search

Observing the above process we can write

$\log_2 n + 1 = c$

for instance if $n = 2$, then

$\log_2 2 + 1 = c$

$1 + 1 = c$

$2 = c$

Similarly if $n = 8$, then

$\log_2 n + 1 = c$

$\log_2 8 + 1 = c$

$3 + 1 = c$

$4 = c$

Thus, we can write that, average case time complexity of binary search is $O(\log_2 n)$.

Merge Sort

Merge Sort: A merger sort algorithm splits an unsorted collection into two halves, it sorts the two halves and then merge then together from one, sorted collection--all of this, using recursion.

OR, A sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence. Run time is Θ(n log n).

1. **Divide** the problem into a number of subproblems
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, then simply solve the subproblems in a straightfoward manner.
3. **Combine** the solutions to the subproblems into the solution for the origional problem.

Example : Merge Sort

Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

Analysis : Merge Sort

1. **Divide** : O(1)
2. **Conquer** : 2T(n/2)
3. **Combine** : O(n)
4. T(n) = O(1) if n <= c, else aT(n/b) + D(n) + C(n).

Merge Sort Simulation

38, 27, 43, 3, 9, 82, 10

Ascending : 3, 9, 10, 27, 38, 43, 82

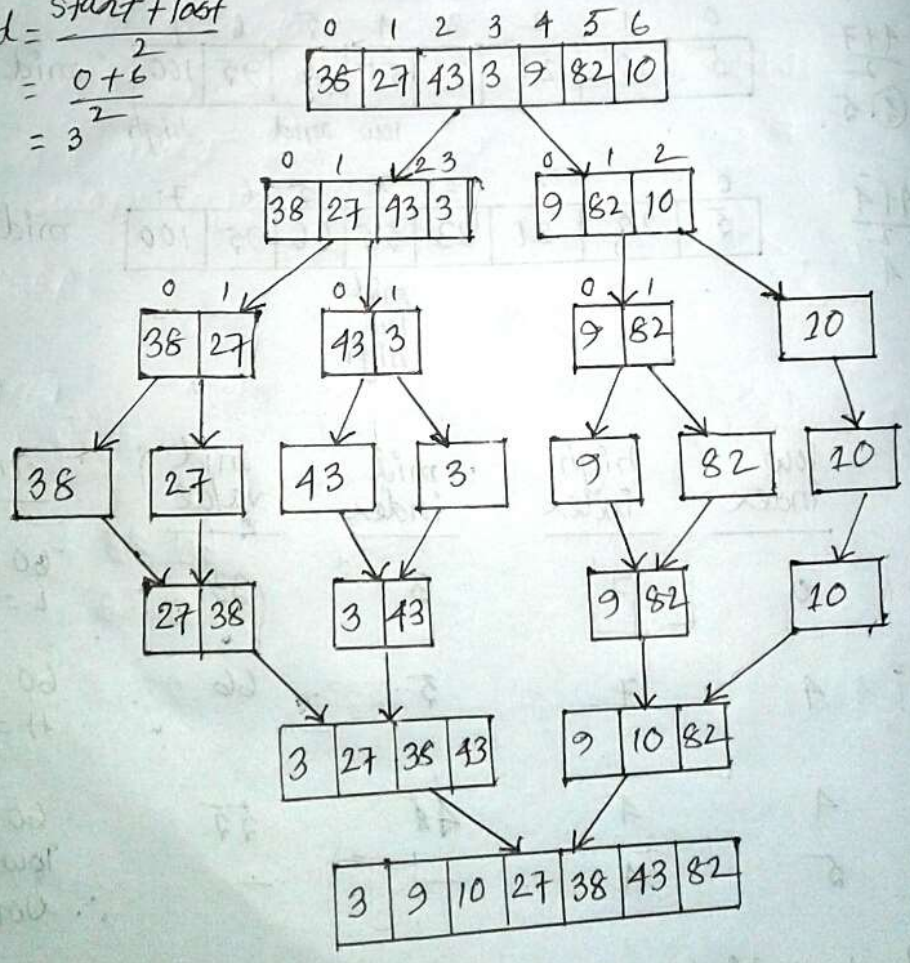
Descending : 82, 43, 38, 27, 10, 9, 8

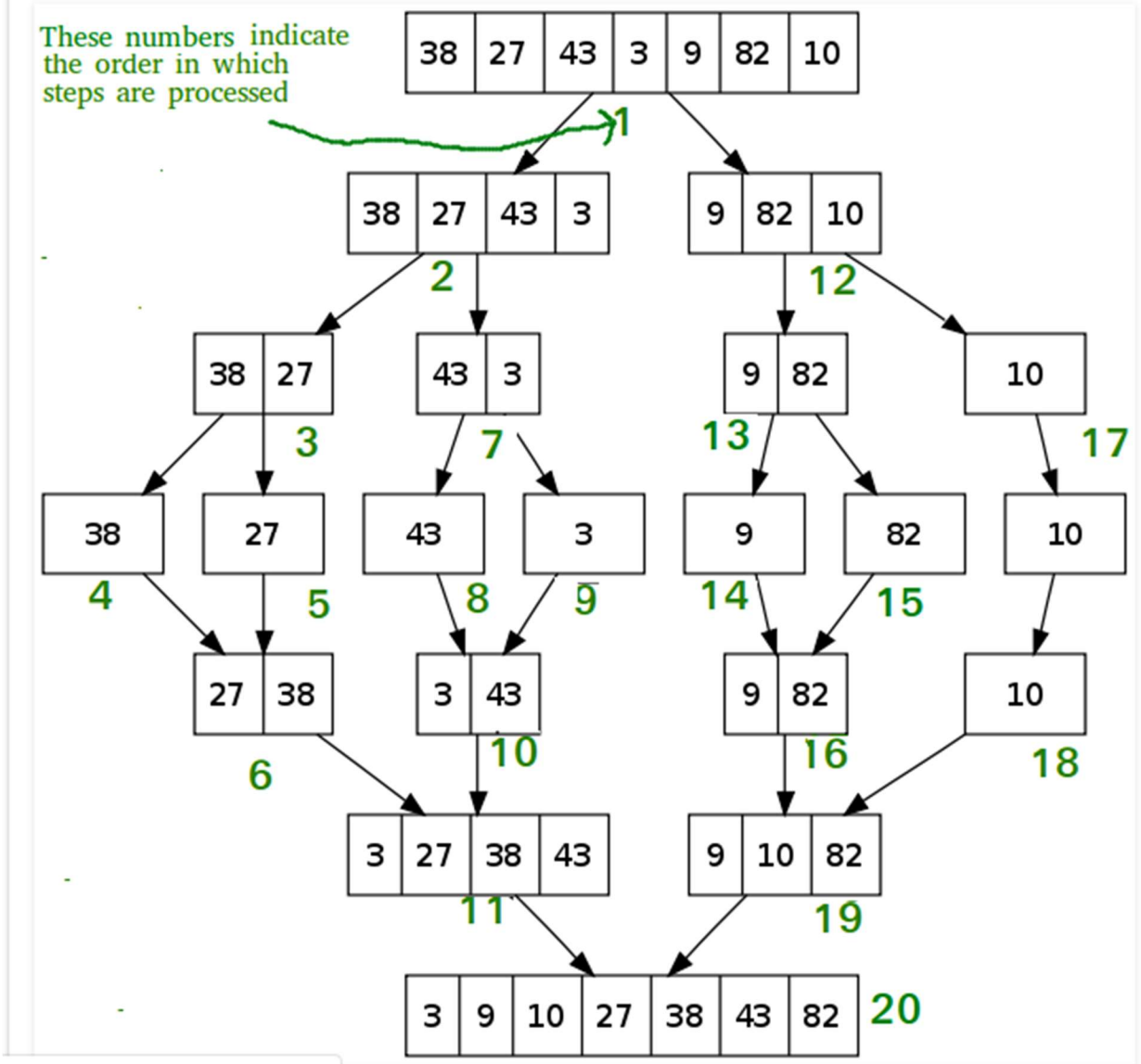
Ascending

$mid = \frac{start + last}{2}$

$= \frac{0+6}{2}$

$= 3$




These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort Simulation

38, 27, 43, 3, 9, 82, 10

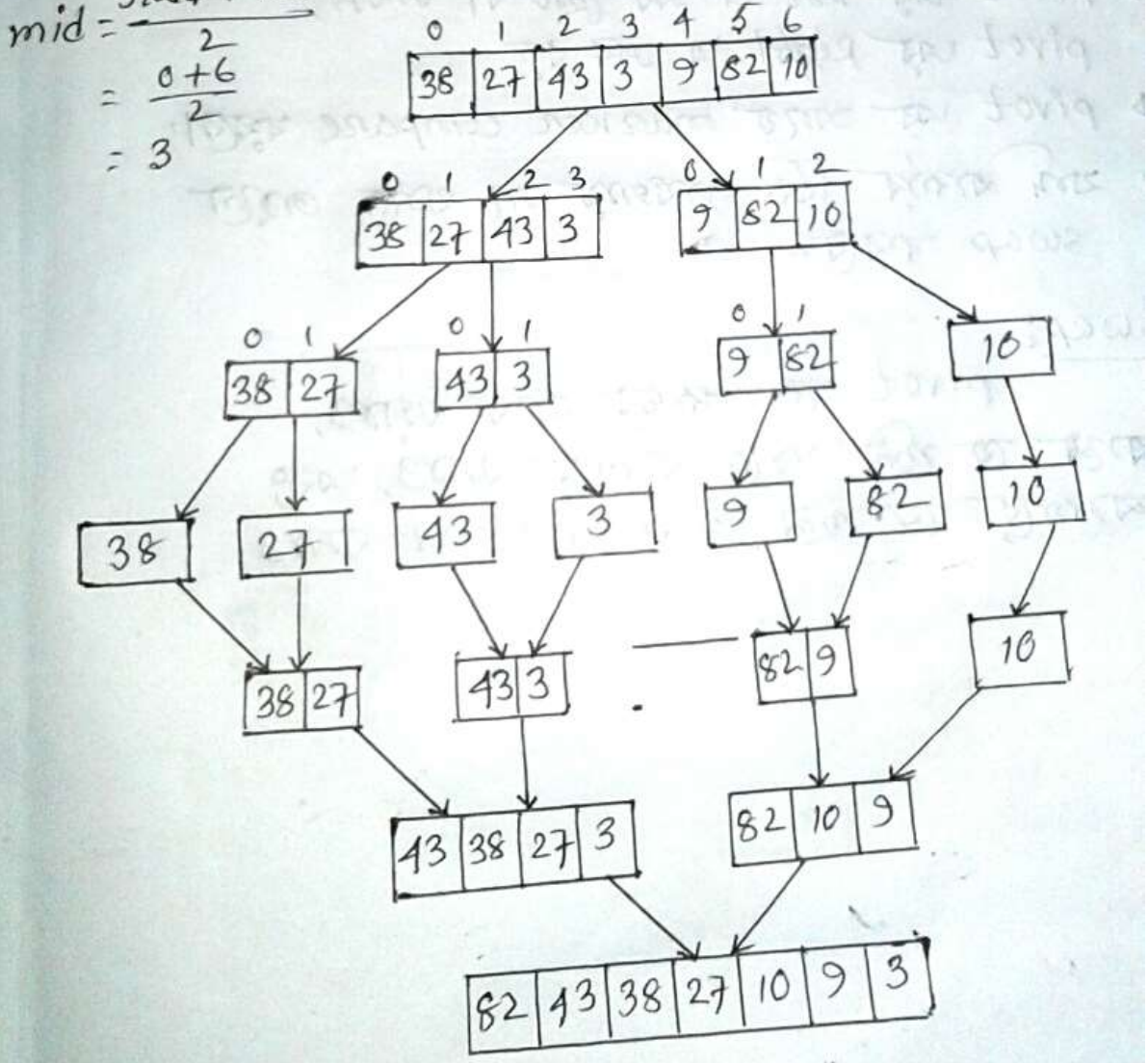
Descending : 82, 43, 38, 27, 10, 9, 8

Descending

$mid = \frac{start + last}{2}$

$= \frac{0+6}{2}$

$= 3$



Time complexity of Merge Sort

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

where, $n > 1$ and $T(1) = 0$

$T(n/2)$ is the time taken by left or right sublist to get sorted. And the cn is the time taken for combining two sublists.

We can obtain the time complexity of merge sort using two methods.

1. Master theorem
2. Substitution method

Using master theorem :

Let the recurrence relation for merge sort is

$$T(n) = T(n/2) + T(n/2) + cn \quad \text{---} \quad (1)$$

$$T(1) = 0 \quad \text{---} \quad (2)$$

As per master theorem

$T(n) = O(n^d \log n)$ if $a = b$

As in equation (1),

$a = 2$, $b = 2$ and $f(n) = cn$. i.e. n^d and $d = 1$

and

$a = b^d$

$2 = 2^1$

This case give us

$T(n) = O(n \log_2 n)$

Hence the average and worst case of time complexity of merge sort is $O(n \log_2 n)$.

Merge Sort – Analysis

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

$\boxed{T(n) = 2T(n/2) + n}$ — ①

substitute $n/2$ in place of n in eqn ①

$T(n/2) = \boxed{2T(n/4) + n/2}$ — ②

substitute eqn ② in eqn ①

$T(n) = 2\{2T(n/4) + n/2\} + n$

$\boxed{T(n) = 2^2 T(n/2^2) + 2n}$ — ③

substitute $n/4$ in place n in eqn ①

$\therefore T(n/4) = 2T(n/8) + n/4$ — ④

$T(n) = 2^2\{2T(n/8) + n/4\} + 2n$

$T(n) = 2^3 T(n/2^3) + 3n$

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

$\vdots$

$T(n) = 2^i T(n/2^i) + in$

$T(1) = 1$

Let $n/2^i = 1 \Rightarrow \boxed{n = 2^i}$

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

$\boxed{\log_2 n = i}$

$T(n) = nT(1) + n\log_2 n$

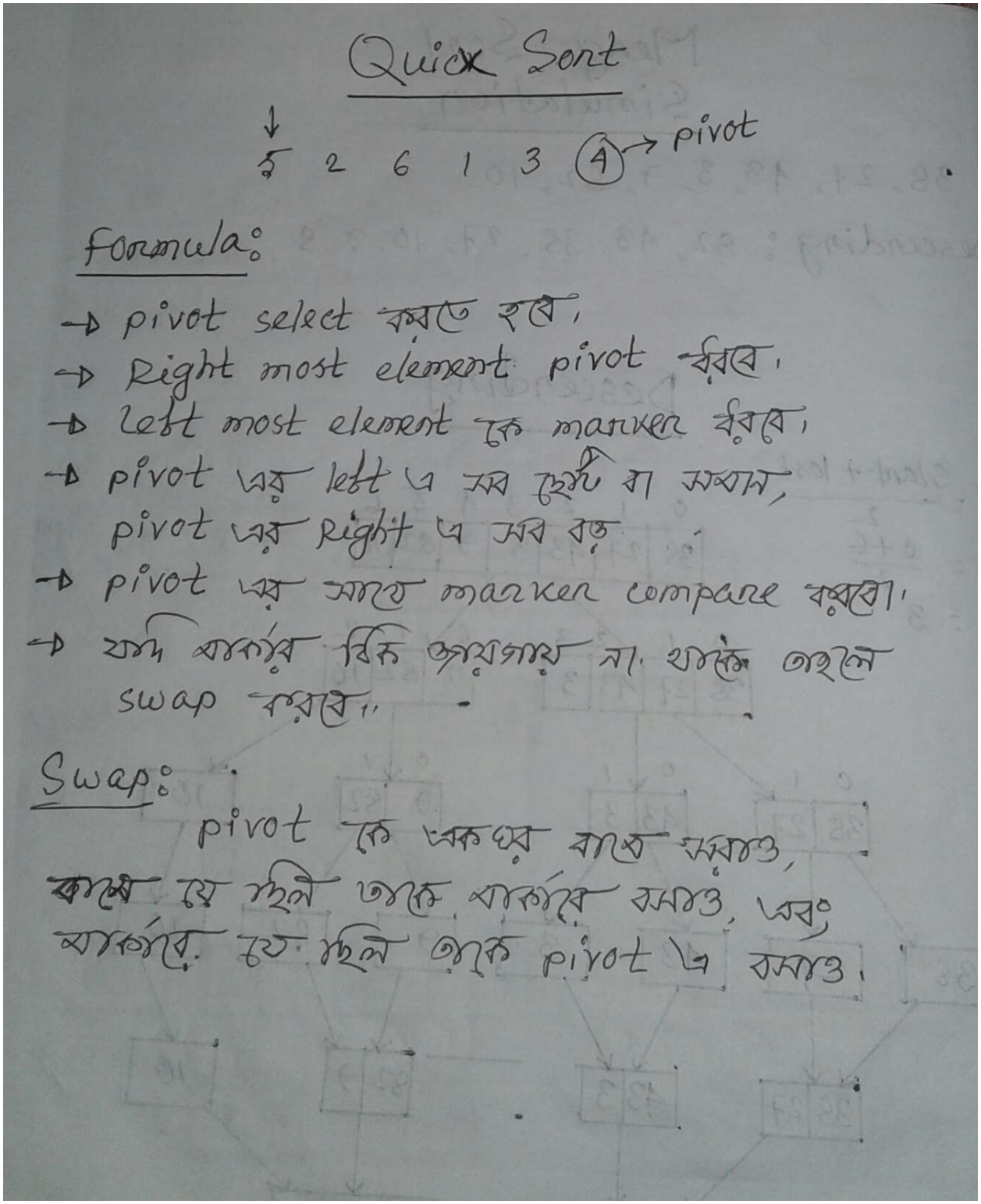
$\boxed{T(n) = n + n\log_2 n}$

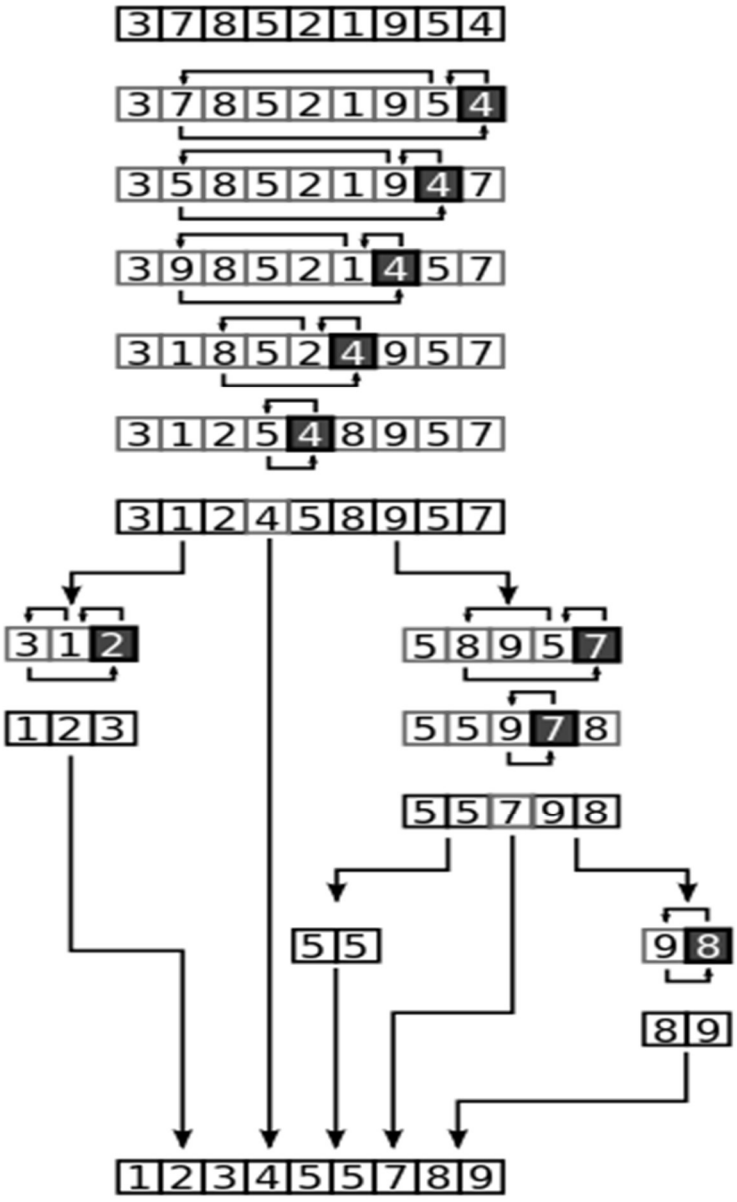
$= O(n \log_2 n)$

Merge Sort: Code

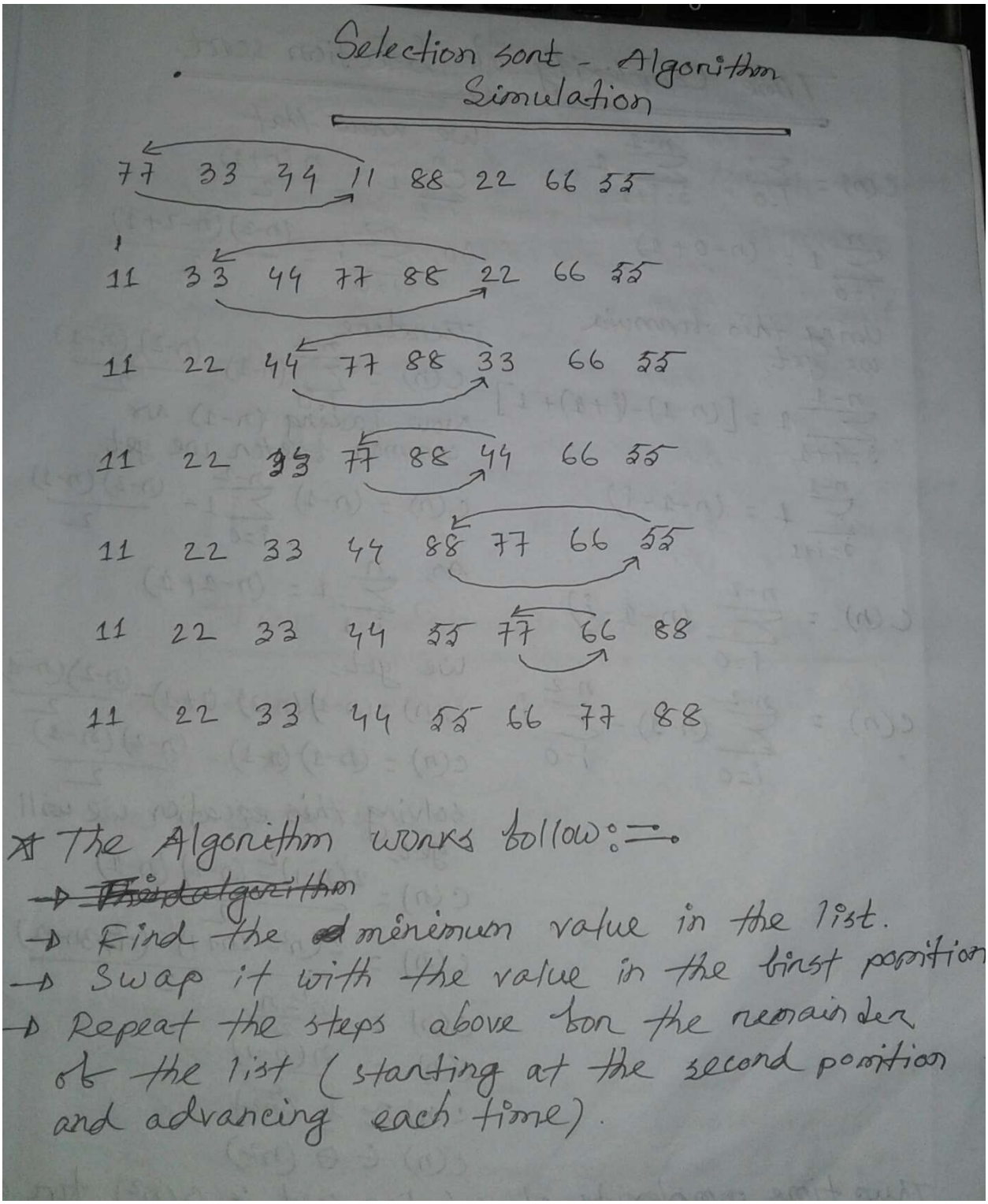
```
void mergesort (int a[], int low, int high)
{
    int mid;
    if (low < high) {
        mid=(low+high)/2;
        mergesort(a, low, mid);
        mergesort(a, mid+1, high);
        merge(a, low, mid, mid+1, high);
    }
}
```

Quick Sort

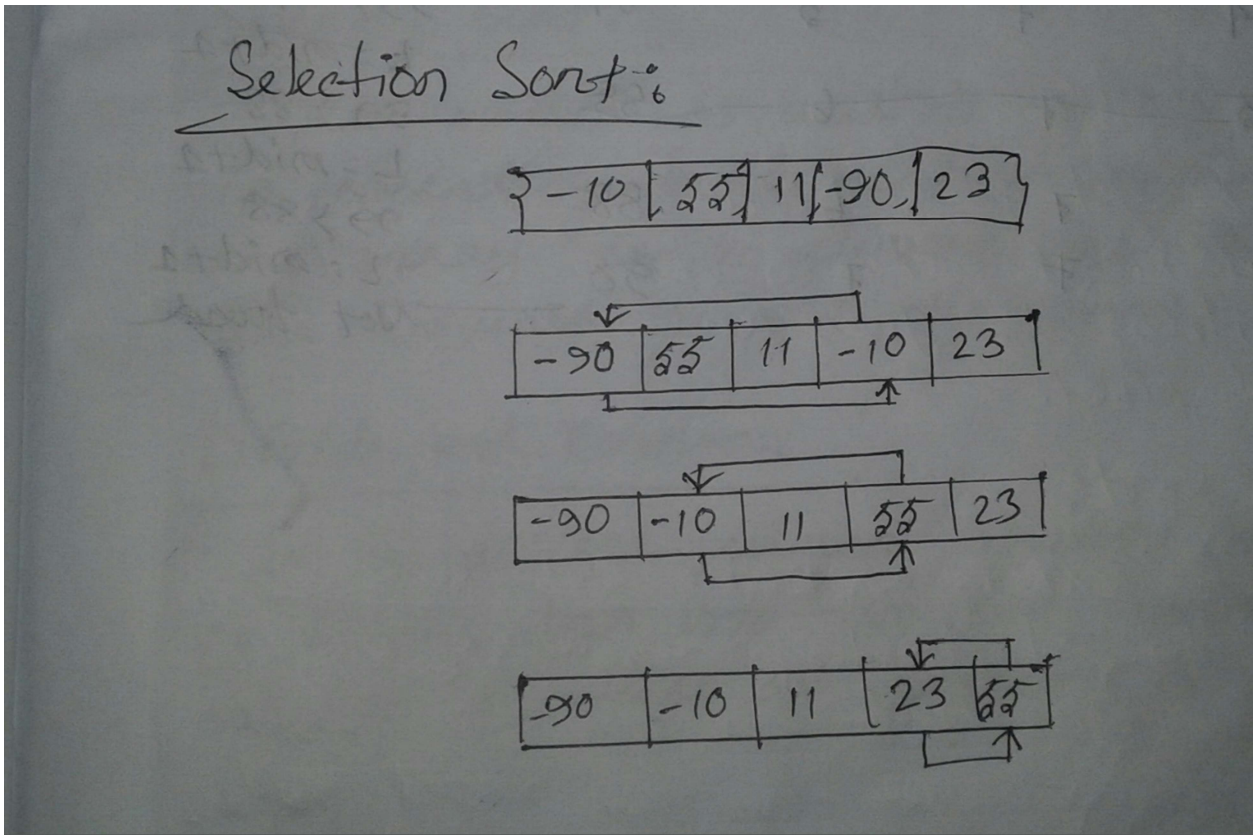

Quick Sort
5 2 6 1 3 4
pivot
Formula:
→ pivot select করতে হবে,
→ Right most element pivot ধরবে.
→ Left most element কে marker ধরবে.
→ pivot এর left এ সব ছোট বা সমান,
pivot এর Right এ সব বড়
→ pivot এর সাথে marker compare করবো.
→ যদি মার্কার ঠিক জায়গায় না থাকে তাহলে
swap করবে.
Swap:
pivot কে একবার বামে সরাও,
কাছে যে মহিলা তাকে মার্কারে বসাও, এবং
মার্কারে যে মহিলা তাকে pivot এ বসাও.


3 7 8 5 2 1 9 5 4
3 7 8 5 2 1 9 5 4
3 5 8 5 2 1 9 4 7
3 9 8 5 2 1 4 5 7
3 1 8 5 2 4 9 5 7
3 1 2 5 4 8 9 5 7
3 1 2 4 5 8 9 5 7
3 1 2
1 2 3
5 8 9 5 7
5 5 9 7 8
5 5 7 9 8
5 5
9 8
8 9
1 2 3 4 5 5 7 8 9

Selection Sort



Selection Sort:



Time Complexity of Selection sort

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

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

Using this formula we get,

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

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

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

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

We know that

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

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

Therefore,

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

Now taking $(n-1)$ as common factor we get,

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

As, $\sum_{i=1}^{n} 1 = (n-1+1)$

We get,

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

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

Solving this equation we will get,

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

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

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

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

$$C(n) \approx \frac{1}{2}(n^2)$$

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

Thus time complexity of selection sort is $\Theta(n^2)$ for all input But total number of key swaps is only $\Theta(n)$.

Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The subarray which is sorted.
- Remaining subarray which is unsorted.

Graph:

A graph is a collection of vertices or nodes connected by a collection of edges. Formally graph Gr is a finite set of (V, E) where V is a set of vertices and E is a set of edges.

Directed Graph:

A directed graph Gr consists of a finite set V - called the vertices or nodes and E set of ordered pairs, called the edges of Gr

Undirected Graph:

A directed graph Gr consists of a finite set V - set of vertices and E - a set of edges of Gr.

Linear Search:

A linear Search scans one items at a time, without jumping to any item.

Prequisite:

① The worst case complexity is O(n), sometimes known as O(n) search.

② Time ~~Search~~ taken to search elements keep increasing as the number of elements are increased

Quick Sort:

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

Merge Sort:

Merge sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

Time Complexity:

The space complexity can be defined as amount of computer time required by an algorithm to run to completion.

The time complexity dependent on following factors:

1. System load
2. Number of other programs running
3. Instruction set used
4. Speed of underlying hardware

The time complexity is therefore given in terms of frequency count. Frequency count is a count denoting number of times of execution of statement.

Growth of Function:

Asymptotic notation is a shorthand way to represent the time complexity.

Various notations such as Ω, θ, O used are called asymptotic notations.

→ Greedy method is used for obtaining optimum solution.

Best case Analysis:

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity.

Worst case Analysis:

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity.

Average Case Analysis:

The time complexity that we get from certain set of inputs is an average same.
Therefore, corresponding input such a time complexity is called average case time complexity.

Vertices: Interconnected objects in a graph are called vertices. vertices are also known as nodes.

Edges: Edge are the links that connect the vertices.

Directed Graph: In a directed graph, edges have direction.

Undirected Graph: In a undirected graph, edges have no direction.

Algorithm:

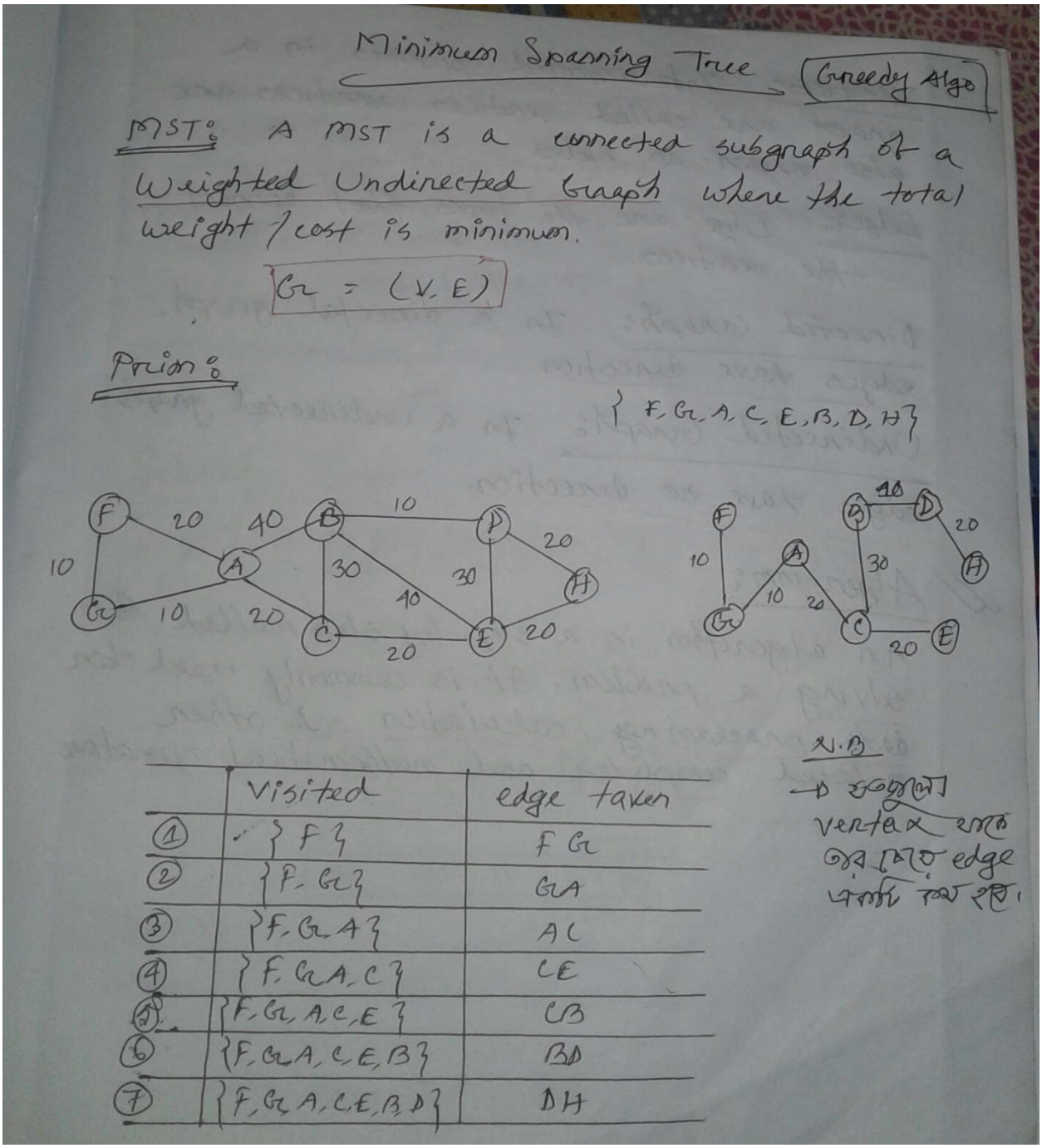
An algorithm is a step by step method of solving a problem. It is commonly used for data processing, calculation and other related computer and mathematical operations

Complexity Summary

Sorting Algorithm	Best Case	Average Case	Worst Case
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Seketion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
Tree Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
Shell Sort	$O(n \log_2 n)$	$O(n^{4/3})$	$O(n \log^2 n)$
Heap Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Bin Sort		$O(n)$	$O(n)$

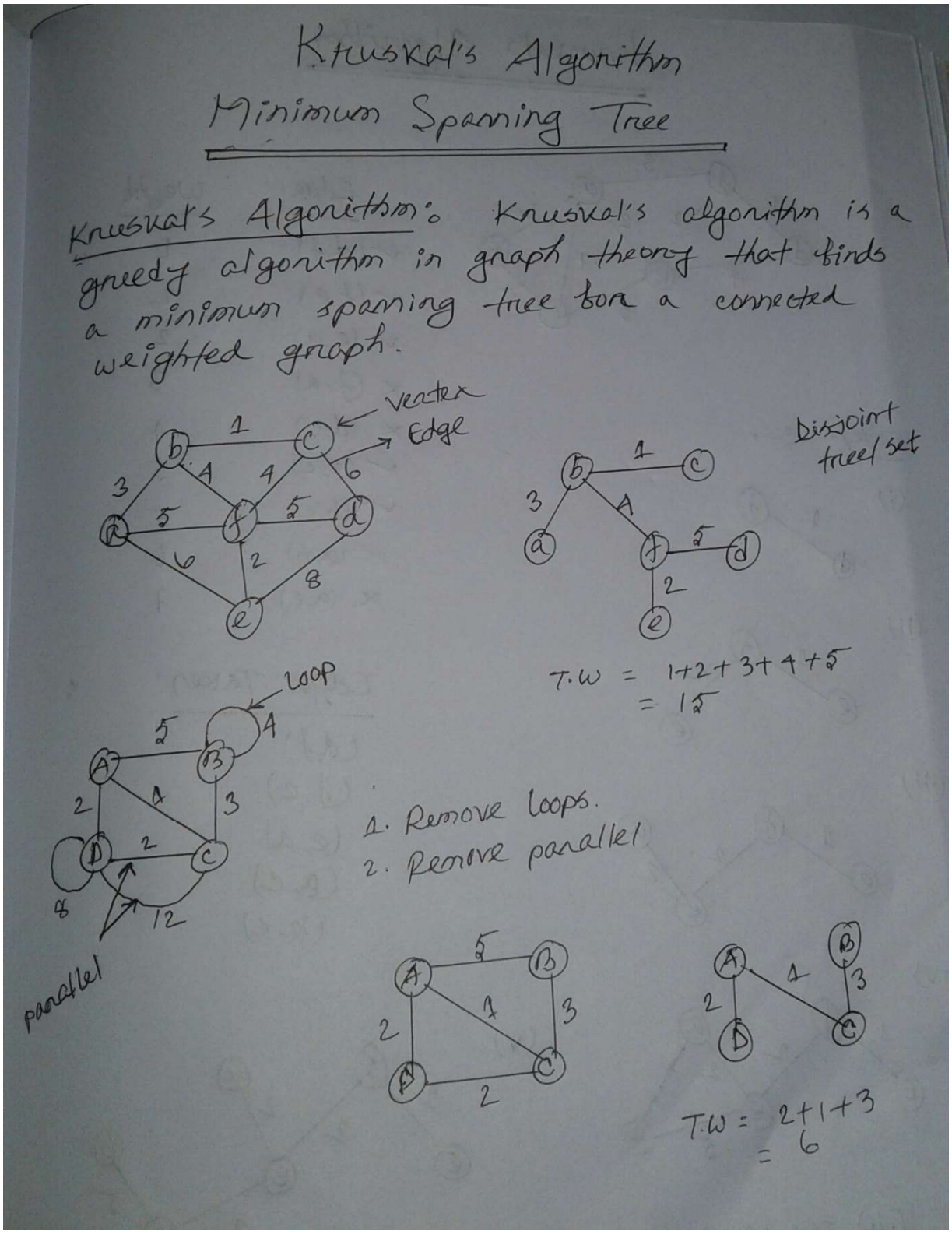
Minimum Spanning Tree (MST)

Prims

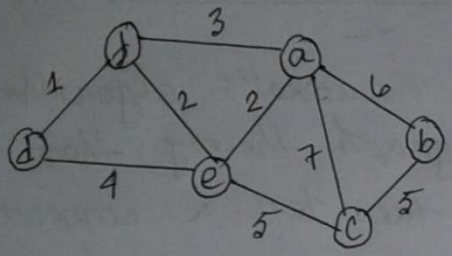


	Visited	edge taken
①	{F}	F G
②	{F, G}	G A
③	{F, G, A}	A C
④	{F, G, A, C}	C E
⑤	{F, G, A, C, E}	C B
⑥	{F, G, A, C, E, B}	B D
⑦	{F, G, A, C, E, B, D}	D H

Kruskal's


Kruskal's Algorithm
Minimum Spanning Tree
Kruskal's Algorithm: Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.
vertex
Edge
Disjoint tree/set
T.W = 1+2+3+4+5
= 15
Loop
parallel
1. Remove loops.
2. Remove parallel
T.W = 2+1+3
= 6

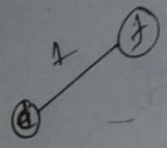
Kruskal's Algorithm



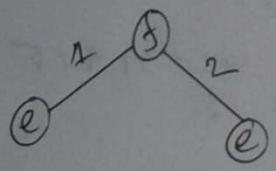
Edge	Weight
✓ (d,f)	1
✓ (f,e)	2
✓ (e,a)	2
× (f,a)	3
× (d,e)	4
✓ (e,c)	5
✓ (b,c)	5
× (a,b)	6
× (a,c)	7

Step :-

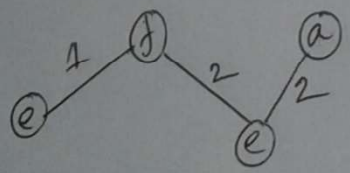
(i)



(ii)



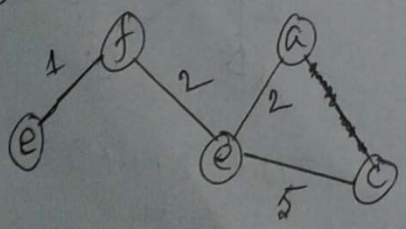
(iii)



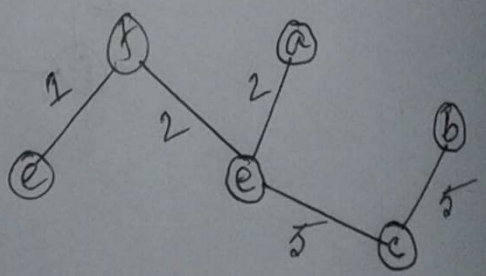
Edge Taken

(d,f)
(f,e)
(e,a)
(a,c)
(b,c)

(iv)



(v)



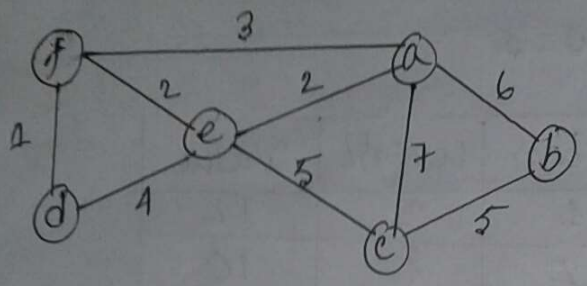
T.W = 1+2+2+5+5 = 15

Kruskal's Algorithm – MST

[Greedy]

[edge নিয়ে
বসিয়া করবো]

[vertices থেকে
Edge একটি
করে হবে]



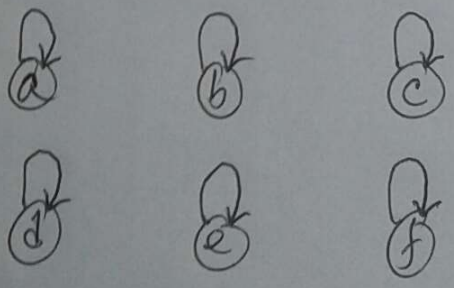
Edge	df	be	de	fa	ae	ab	ac	ec	bc
weight	1	2	4	3	2	6	7	5	5

Sort the edges

✓ ✓ ✓ × × ✓ ✓ × ×

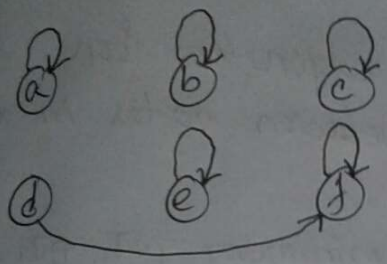
Edge	df	fe	ae	fa	de	ec	bc	ab	ac
weight	1	2	2	3	4	5	5	6	7

Disjoint Set:



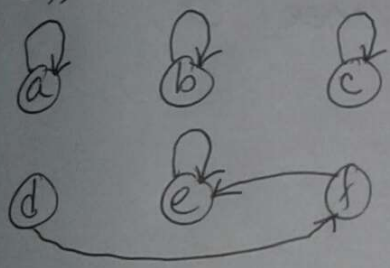
edge taken

df.



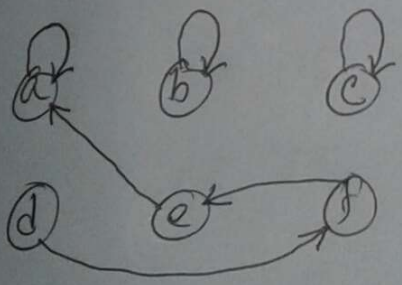
edge taken

df, fe



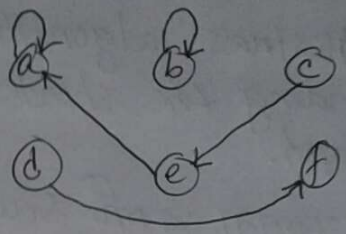
edge taken

df, fe, ae



edge taken

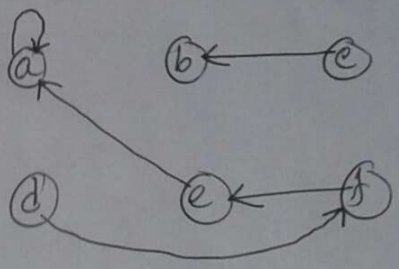
df, fe, ae, ec.



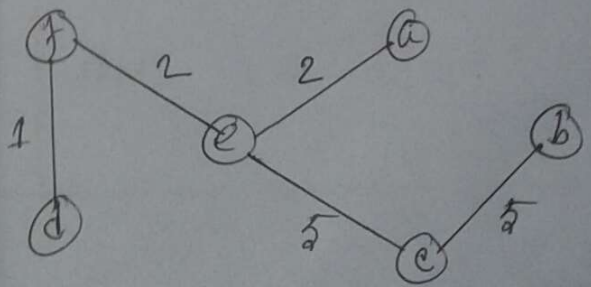
cycle হওয়া যাবে না

edge taken

df, fe, ae, ec, bc



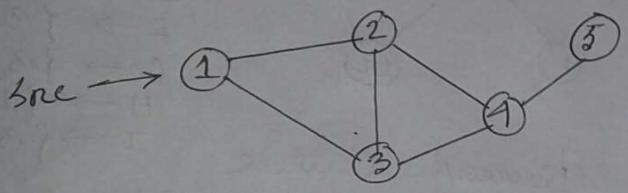
Now



Breadth First Search - (BFS)

Breadth First Search (BFS)

* It is a powerful technique of systematically traversing the edges of a given graph such every edge is traversed once and also each vertex is visited at least once.



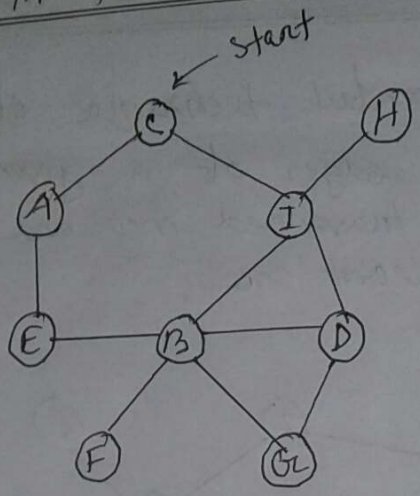
N.B
→ Source বলে কিছু থাকে না। Like Prims এর মতো

Visited	Current	Queue
{1}	1	[1]
{1,2,3}	1	[1,2,3]
{1,2,3}	2	[2,3]
{1,2,3,4}	2	[2,3,4]
{1,2,3,4}	3	[3,4]
{1,2,3,4,5}	4	[4,5]
{1,2,3,4,5}	5	[5]
{1,2,3,4,5}	- -	[-]

Adjacency List

1 → {2, 3}
2 → {1, 3, 4}
3 → {1, 2, 4}
4 → {2, 3, 5}
5 → {2, 3, 5}

Breadth first Search (BFS)



Adjacency List

A → { C, E }
B → { E, I, G, F, D }
C → { A, I }
D → { B, G, I }
E → { A, B }
F → { B }
G → { B, D }
H → { I }
I → { B, C, D, H }

Visited	Current	Queue
{ C }	C	[C]
{ C, A, I }	C	[C, A, I]
{ C, A, I }	A	[A, I]
{ C, A, I, E }	A	[A, I, E]
{ C, A, I, E }	I	[I, E]
{ C, A, I, E, B, D, H }	I	[I, E, B, D, H]
{ C, A, I, E, B, D, H }	E	[E, B, D, H]
{ C, A, I, E, B, D, H }	B	[B, D, H]
{ C, A, I, E, B, D, H, G, F }	B	[D, H, G, F]
{ C, A, I, E, B, D, H, G, F }	D	[H, G, F]
{ C, A, I, E, B, D, H, G, F }	H	[G, F]
{ C, A, I, E, B, D, H, G, F }	F	[F]
{ C, A, I, E, B, D, H, G, F }	G	[G]
{ C, A, I, E, B, D, H, G, F }	-	-

Sub: SAT SUN MON TUE WED THU FRI

DATE: / /

BFS:-

Visited	current	Queue
{c}	C	[c]
{c, A, I}	c	[c, A, I]
{c, A, I}	A	[A, I]
{c, A, I, E}	A	[A, I, E]
{c, A, I, E}	I	[I, E]
{c, A, I, E, B, D, H}	I	[I, E, B, D, H]
{c, A, I, E, B, D, H}	E	[E, B, D, H]
{c, A, I, E, B, D, H}	B	[B, D, H]
{c, A, I, E, B, D, H, F, G}	B	[B, D, H, F, G]
{c, A, I, E, B, D, H, F, G}	D	[D, H, F, G]
{c, A, I, E, B, D, H, F, G}	H	[H, F, G]
{c, A, I, E, B, D, H, F, G}	F	[F, G]
{c, A, I, E, B, D, H, F, G}	G	[G]
{c, A, I, E, B, D, H, F, G}	—	—

06

Sub: DATE: / /

06-12-2

Adjacency list:

A → { C, E }

B → { D, E, F, G, I }

C → { A, I }

D → { B, G, I }

E → { A, B }

F → { B }

G → { B, D }

H → { I }

I → { B, C, D, H }

AAA

Adjacency matrix :

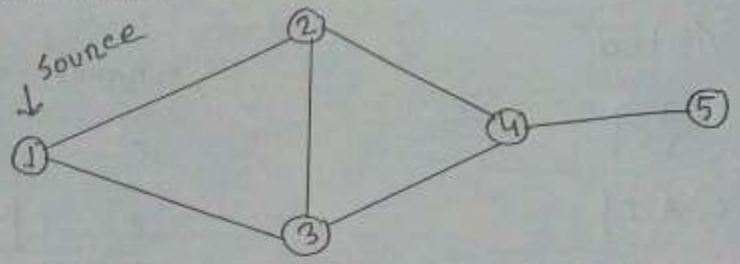
	A	B	C	D	E	F	G	H	I
A	0	0	1	0	1	0	0	0	0
B	0	0	0	1	1	1	1	0	1
C	1	0	0	0	0	0	0	0	1
D	0	1	0	0	0	0	1	0	1
E	1	1	0	0	0	0	0	0	0
F	0	1	0	0	0	0	0	0	0
G	0	1	0	1	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1
I	0	1	1	1	0	0	0	1	0

Sub:

SAT SUN MON TUE WED THU FRI

DATE / /

#



Visited	current	Queue
{1}	1	[1]
{1, 2, 3}	1	[1, 2, 3]
{1, 2, 3}	2	[2, 3]
{1, 2, 3, 4}	2	[2, 3, 4]
{1, 2, 3, 4}	3	[3, 4]
{1, 2, 3, 4}	4	[4]
{1, 2, 3, 4, 5}	4	[4, 5]
{1, 2, 3, 4, 5}	5	[5]
{1, 2, 3, 4, 5}	–	–

Sub: SAT SUN MON TUE WED THU FRI

DATE / /

06-12-

Visited	current	Queue
{1}	1	[1]
{1, 5, 4, 6}	1	[5, 4, 6]
{1, 5, 4, 6}	5	[5, 4, 6]
{1, 5, 4, 6, 2}	5	[5, 4, 6, 2]
{1, 5, 4, 6, 2}	4	[4, 6, 2]
{1, 5, 4, 6, 2, 3}	4	[4, 6, 2, 3]
{1, 5, 4, 6, 2, 3}	6	[6, 2, 3]
{1, 5, 4, 6, 2, 3, 7}	6	[6, 2, 3, 7]
{1, 5, 4, 6, 2, 3, 7}	2	[2, 3, 7]
{1, 5, 4, 6, 2, 3, 7}	3	[3, 7]
{1, 5, 4, 6, 2, 3, 7, 8}	3	[3, 7, 8]
{1, 5, 4, 6, 2, 3, 7, 8}	7	[7, 8]
{1, 5, 4, 6, 2, 3, 7, 8}	8	[8]
{1, 5, 4, 6, 2, 3, 7, 8}	—	—

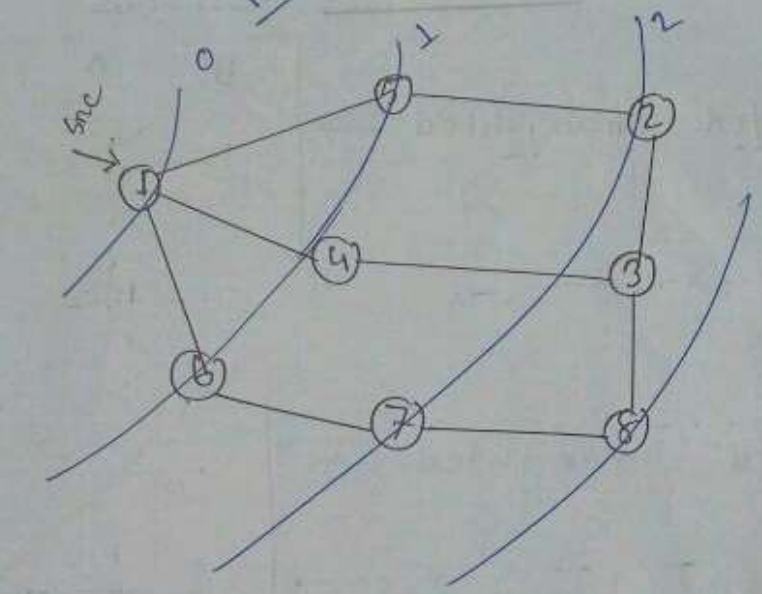
Sub:

DATE: / /

06-12-201

BFS

nodes / Level / Breadth



order {1, 5, 4, 6, 2, 3, 7, 8}

1 } level 0 / parents

5, 4, 6 } level 1 / child

2, 3, 7 } level 2 /

8 } level 3 /

Sub: SAT SUN MON TUE WED THU FRI

DATE / /

06-12-

Visited	current	Queue
{1}	1	[1]
{1, 5, 4, 6}	1	[5, 4, 6]
{1, 5, 4, 6}	5	[5, 4, 6]
{1, 5, 4, 6, 2}	5	[5, 4, 6, 2]
{1, 5, 4, 6, 2}	4	[4, 6, 2]
{1, 5, 4, 6, 2, 3}	4	[4, 6, 2, 3]
{1, 5, 4, 6, 2, 3}	6	[6, 2, 3]
{1, 5, 4, 6, 2, 3, 7}	6	[6, 2, 3, 7]
{1, 5, 4, 6, 2, 3, 7}	2	[2, 3, 7]
{1, 5, 4, 6, 2, 3, 7}	3	[3, 7]
{1, 5, 4, 6, 2, 3, 7, 8}	3	[3, 7, 8]
{1, 5, 4, 6, 2, 3, 7, 8}	7	[7, 8]
{1, 5, 4, 6, 2, 3, 7, 8}	8	[8]
{1, 5, 4, 6, 2, 3, 7, 8}	—	—

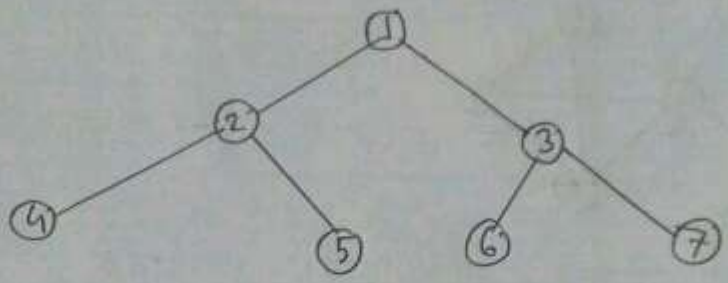
Sub:

SAT SUN MON TUE WED THU FRI

DATE: / /

06-12

BFS and DFS



Level-order = BFS : 1, 2, 3, 4, 5, 6, 7

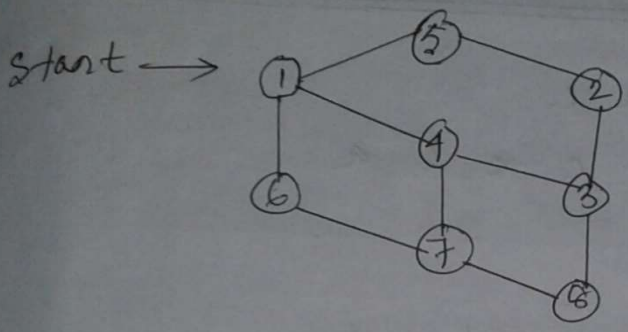
pre-order = DFS : 1, 2, 4, 5, 3, 6, 7

↓

Root → Left → Right

Depth First Search - (DFS)

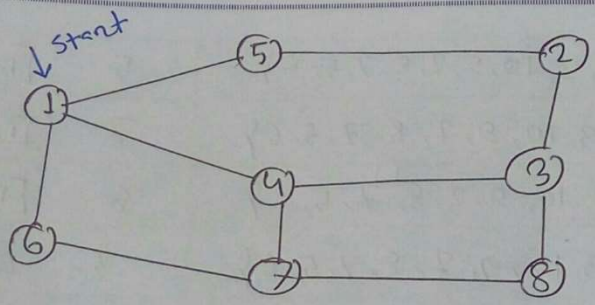
Depth First Search (DFS)



Visited	Current	Stack
{1}	1	[1]
{1}	1	[5, 4, 6]
{1, 6}	6	[5, 4, 7]
{1, 6, 7}	7	[5, 4, 8, 4]
{1, 6, 7, 4}	4	[5, 4, 8, 3]
{1, 6, 7, 4, 3}	3	[5, 4, 8, 2, 8]
{1, 6, 7, 4, 3, 8}	8	[5, 4, 8, 2]
{1, 6, 7, 4, 3, 8, 2}	2	[5, 4, 8, 5]
{1, 6, 7, 4, 3, 8, 2, 5}	5	[5, 4, 8]
{1, 6, 7, 4, 3, 8, 2, 5}	8	[5, 4]
{1, 6, 7, 4, 3, 8, 2, 5}	4	[5]
{1, 6, 7, 4, 3, 8, 2, 5}	5	[-]

DATE: / /

#



DFS

Visited	current	Stack	
{1}	1	[1]	
{1, 6}	6	[1, 6]	
{1, 6, 7}	7	[1, 6, 7]	
{1, 6, 7, 4}	4	[1, 6, 7, 4]	
{1, 6, 7, 4, 3}	3	[1, 6, 7, 4, 3]	
{1, 6, 7, 4, 3, 8}	8	[1, 6, 7, 4, 3, 8]	
{1, 6, 7, 4, 3, 8, 6}	3	[1, 6, 7, 4, 3]	pop, 8
{1, 6, 7, 4, 3, 8, 2}	2	[1, 6, 7, 4, 3, 2]	
{1, 6, 7, 4, 3, 8, 2, 5}	5	[1, 6, 7, 4, 3, 2, 5]	
{1, 6, 7, 4, 3, 8, 2, 5}	2	[1, 6, 7, 4, 3, 2]	pop, 5
{1, 6, 7, 4, 3, 8, 2, 5}	3	[1, 6, 7, 4, 3]	pop, 2
{1, 6, 7, 4, 3, 8, 2, 5}	4	[1, 6, 7, 4]	pop, 3
{1, 6, 7, 4, 3, 8, 2, 5}	7	[1, 6, 7]	pop, 4
{1, 6, 7, 4, 3, 8, 2, 5}	6	[1, 6]	pop, 7
{1, 6, 7, 4, 3, 8, 2, 5}	1	[1]	pop, 6

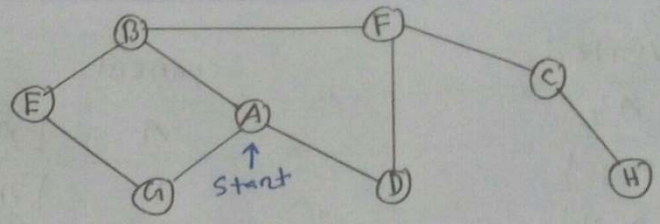
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	5	[1, 4, 3, 2, 8, 7, 5]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	7	[1, 4, 3, 2, 8, 7]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	8	[1, 4, 3, 2, 8]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	2	[1, 4, 3, 2]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	3	[1, 4, 3]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	4	[1, 4]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	1	[1]
{1, 4, 3, 10, 9, 2, 8, 7, 5, 6}	—	—

Sub:

SAT SUN MON TUE WED THU FRI

DATE: / /

Ex

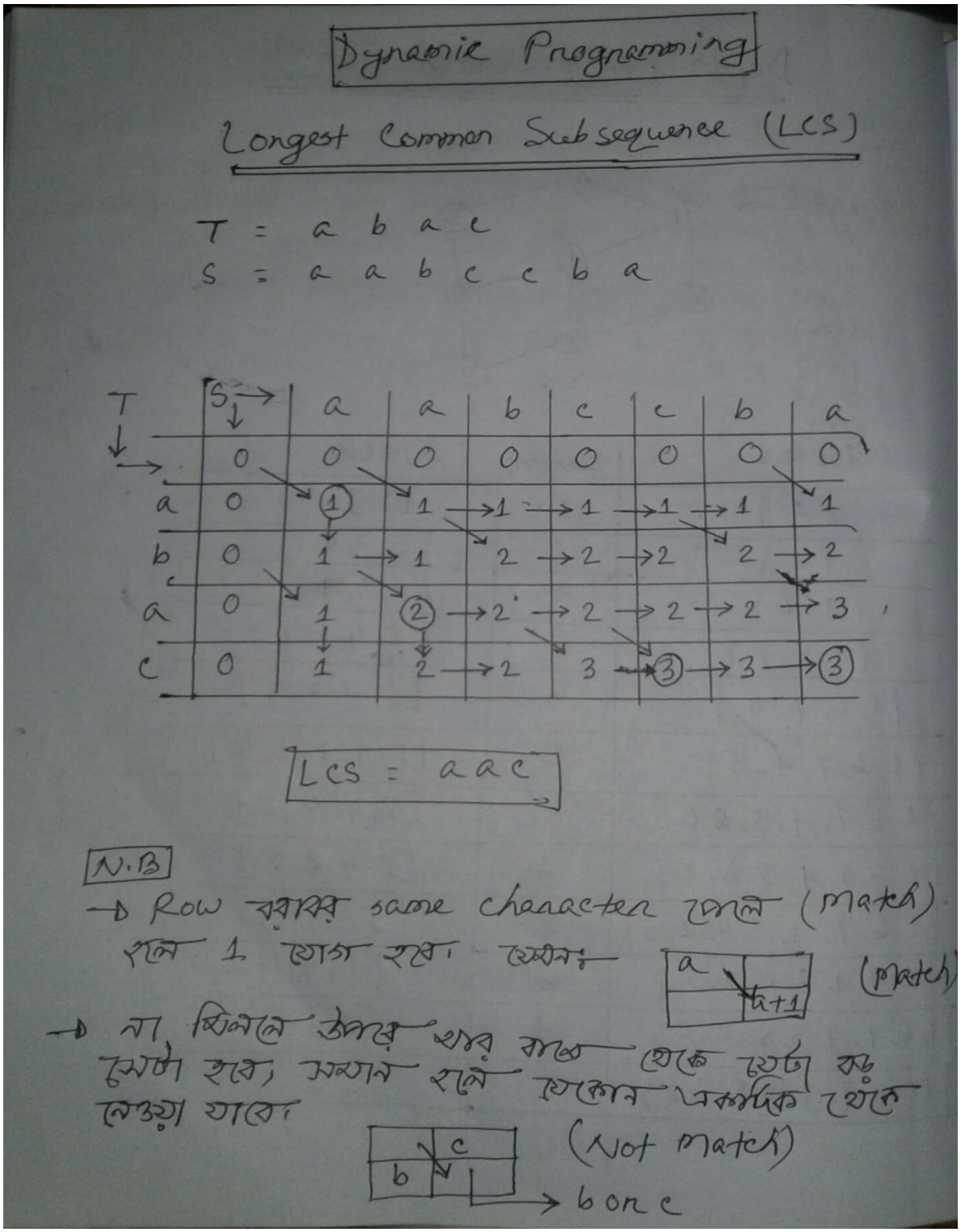


DFS

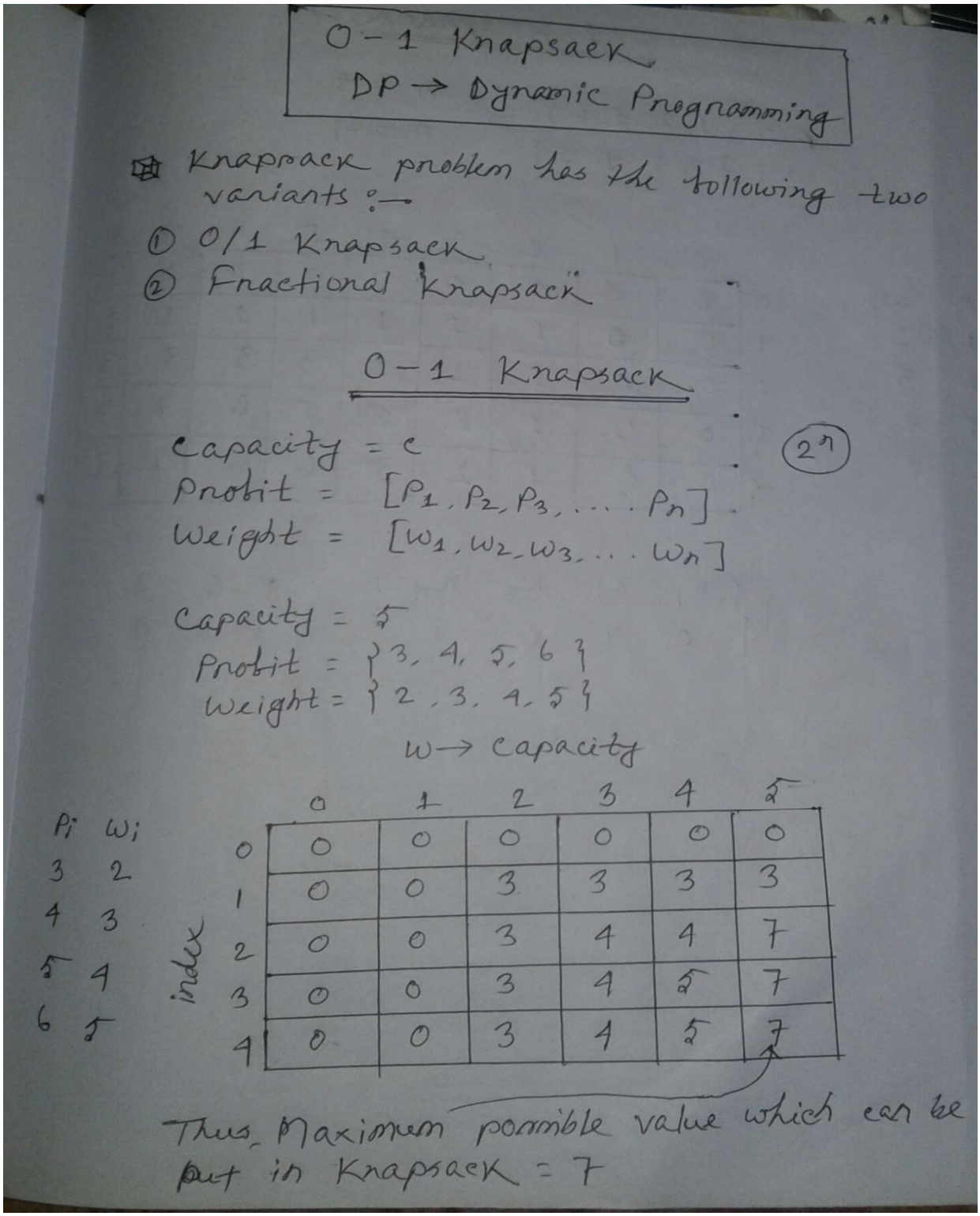
Visited	current	stack	
{A}	A	[A]	
{A, B}	B	[A, B]	
{A, B, E}	E	[A, B, E]	
{A, B, E, G}	G	[A, B, E, G]	
{A, B, E, G}	E	[A, B, E]	POP, G
{A, B, E, G}	B	[A, B]	" E
{A, B, E, G, F}	F	[A, B, F]	
{A, B, E, G, F, C}	C	[A, B, F, C]	
{A, B, E, G, F, C, H}	H	[A, B, F, C, H]	
{A, B, E, G, F, C, H}	C	[A, B, F, C]	" H
{A, B, E, G, F, C, H}	F	[A, B, F]	" C
{A, B, E, G, F, C, H, D}	D	[A, B, F, D]	
{A, B, E, G, F, C, H, D}	F	[A, B, F]	" D
{A, B, E, G, F, C, H, D}	B	[A, B]	" F
{A, B, E, G, F, C, H, D}	A	[A]	" B
{A, B, E, G, F, C, H, D}	—	—	" A

Dynamic Programming

Longest Common Sebsequence - (LCS)



(0 / 1) Knapsack - DP



0/1 Knapsack

$P = \{1, 2, 5, 6\}$ $\quad m = 8$

$w = \{2, 3, 4, 5\}$ $\quad n = 4$

P_i	w_i		0	1	2	3	4	5	6	7	8
		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
6	5	4	0	0	1	2	5	6	6	7	(8)

x_1	x_2	x_3	x_4
0	1	0	1

$8 - 6 = 2$

$2 - 2 = 0$

0/1 Knapsack

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

Example:

$P(i, w) = \max \{v_i + P(i-1, w-w_i), P(i-1, w)\}$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

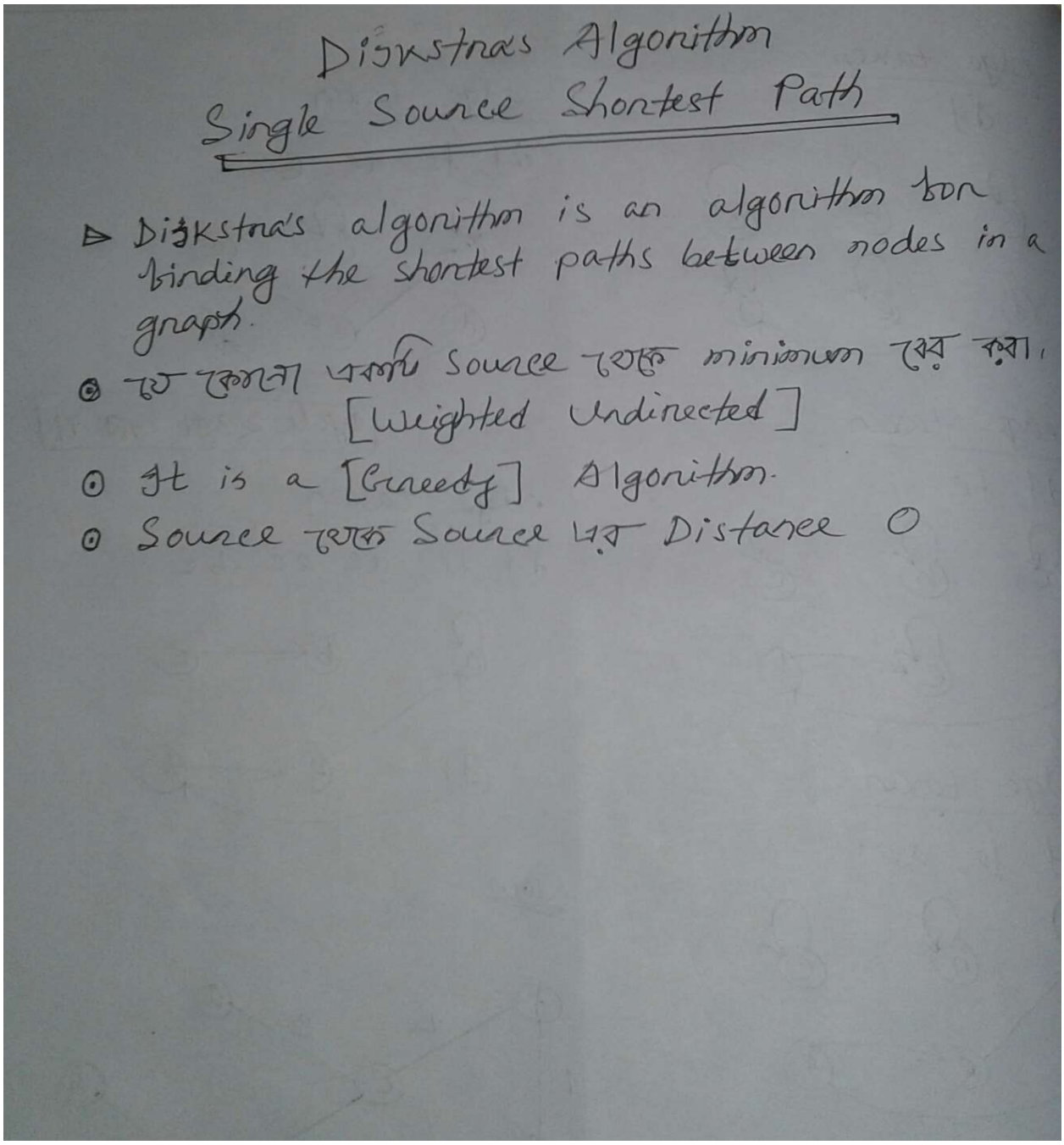
P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Dijkstra's

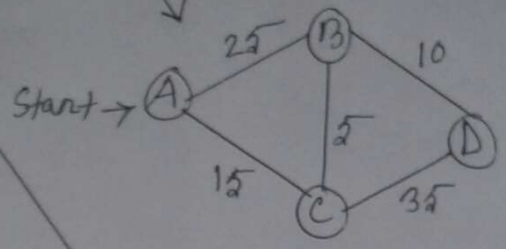
Single Source Shortest Path



Relaxation:

condition true ⟹ Replace

if $(d[v] > d[u] + w(u,v) : d[v] = d[u] + w(u,v)$



A	B	C	D	U	V = adj (U)	Queue
0	∞	∞	∞	A	B, C	B, C
0	25	15	∞	B	A, D, C	C, D
0	25	15	35	C	A, B, D	D, B
0	20	15	35	D	B, C	
0	20	15	30	B	A, D, C	D
0	20	15	30	D	B, C	