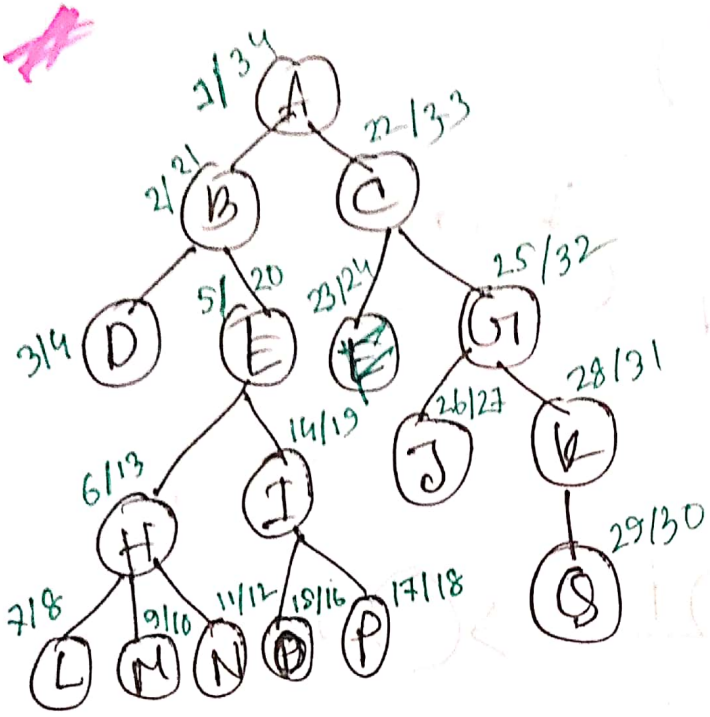


finishing time

A C G K S J F B E I P O H N M L D

topological sort.

lash {2341, 4234, 2839, 430, 22, 397, 3920}

0	3920
1	22
2	
3	2341
4	2839
5	430
6	4234
7	397

key	location
2341	3
4234	6
2839	4
430	3
22	1
397	5
3920	0

can Binary search used for linked list

yes, it is posible on the linked list if the list is ordered and we know the count of elements in the list

Binary search is usually fast and efficient for arrays Because accessing the middle index between two given indices is easy and fast (O(1))

But memory allocation for the singly linked list is dynamic and non-configuous. which makes finding the middle element difficult

TC: O(n)

Auxilary space O(1)

~~middle element of~~

if the list is 1→2→3→4→5
then output should be 3

if the list is 1→2→3→4→5→6
then the output should be 4

method

traverse the whole liked list and count the no. of nodes now traverse the list again till $\frac{count}{2}$ and return the node at $\frac{count}{2}$.

TC O(n) where n is no of nodes in linked list

Ex

5 → NULL
the middle element is [5]

4 → 5 → NULL
the middle element is [5]

3 → 4 → 5 → NULL
the middle element is [4]

2 → 3 → 4 → 5 → NULL
the middle element is [4]

Approach

(I) start node (set to head of list) and last node (set to NULL) initially.

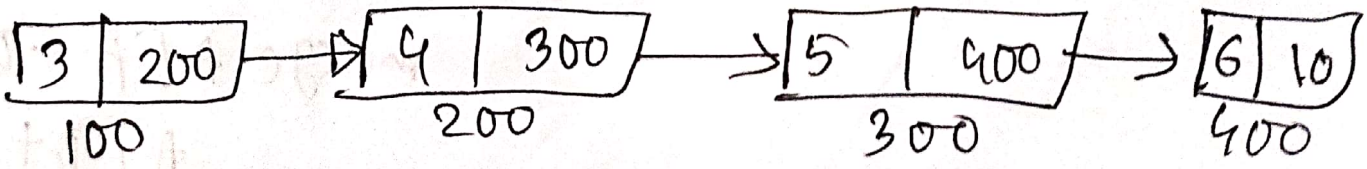
(II) middle is calculated.

(III) if middle matches the required value of search return it.

(IV) ese if middle data < value. move to upper half (setting start to middle's next).

(V) else go to lower half (seting last to middle)

(VI) the condition to come out is either the element found or either list is traversed. when entire list is traversed, last points to start



key = 4

Head = 100 = start
last = 10

middle = 300

middle->data(5) > key(4)

∴ last = 300 [update last]

middle = 200

middle->data(4) == key(4)

return middle.

* But ওটা হবে না কারণ inefficient
TC $O(N * \log N)$

Marge sort

Mergesort (A, p, r) // sort A[p...r] via mergesort:

if p < r

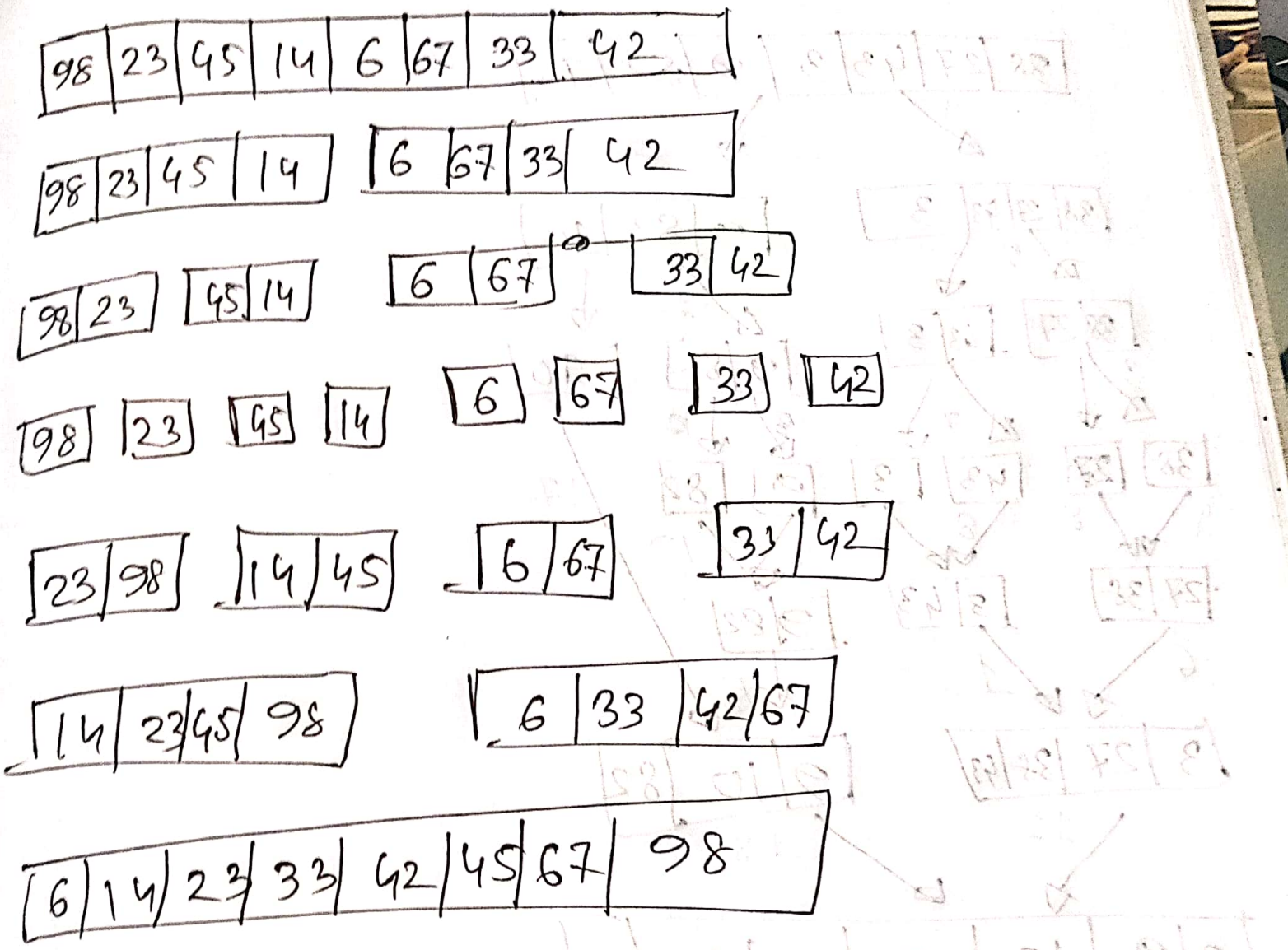
then $q = \frac{p + r}{2}$ // divide

Mergesort (A, p, q) // conqure

Mergesort (A, q+1, r) // conqure

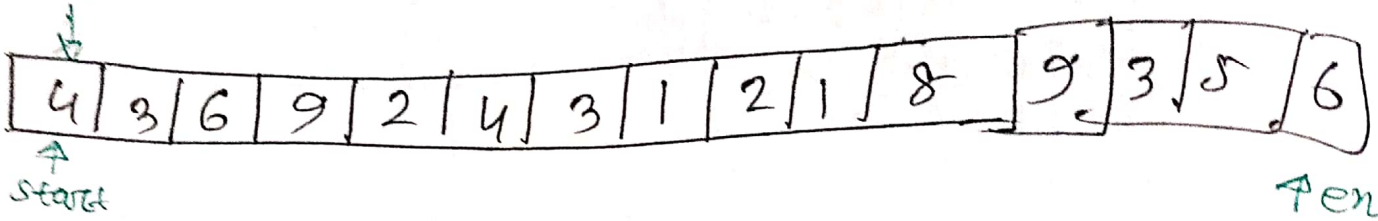
Merge (A, p, q, r) // combine:

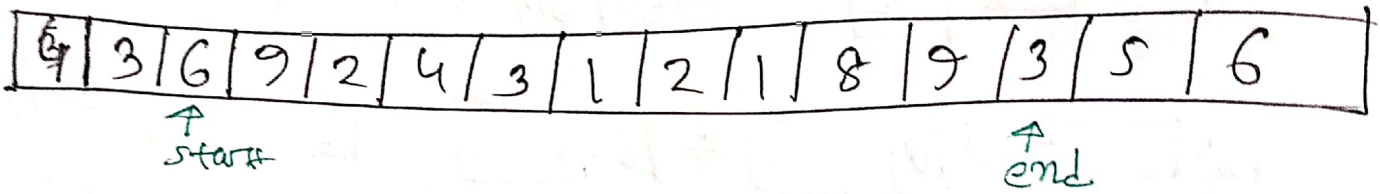
merge A[p...q] with A[q+1, ...r]

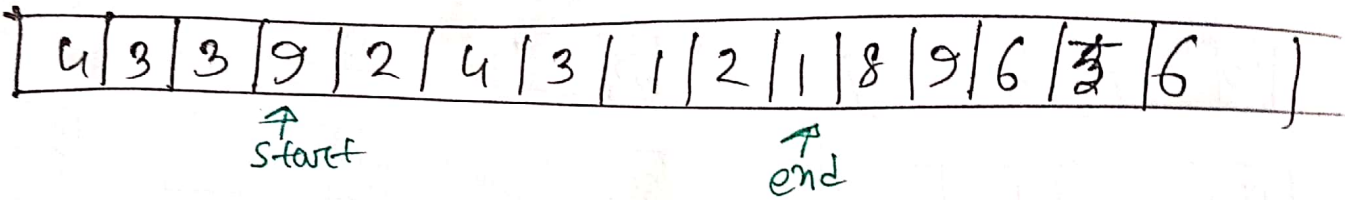

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
33
42
23 98
14 45
6 67
33 42
14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98

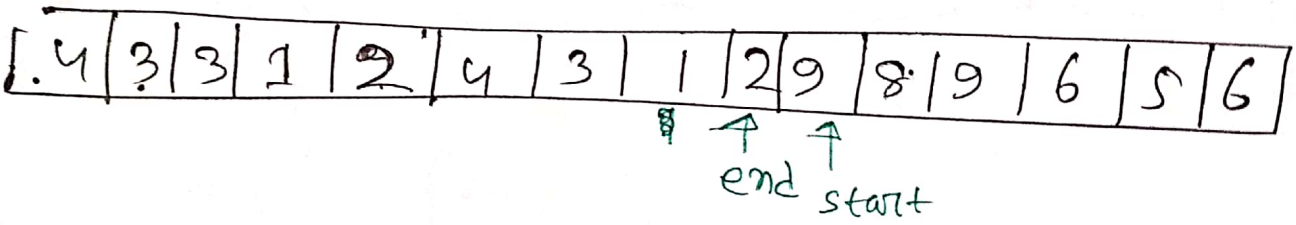
lowerbound

quicksort

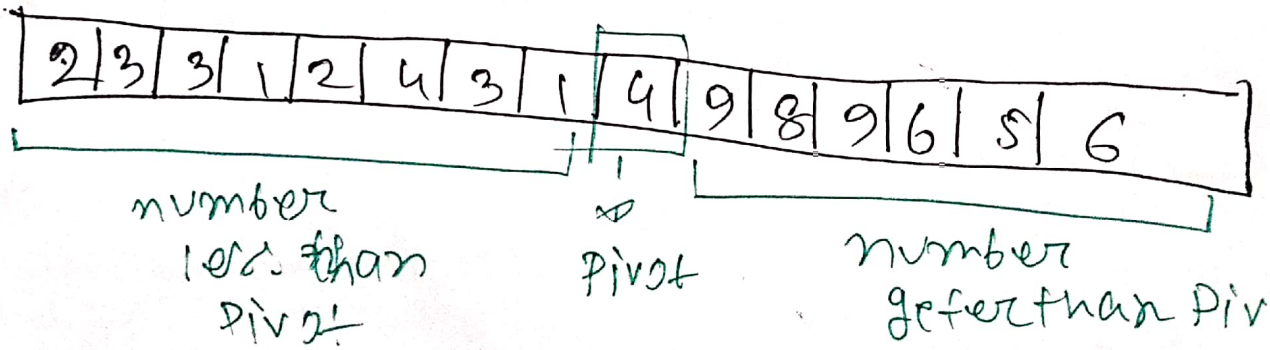




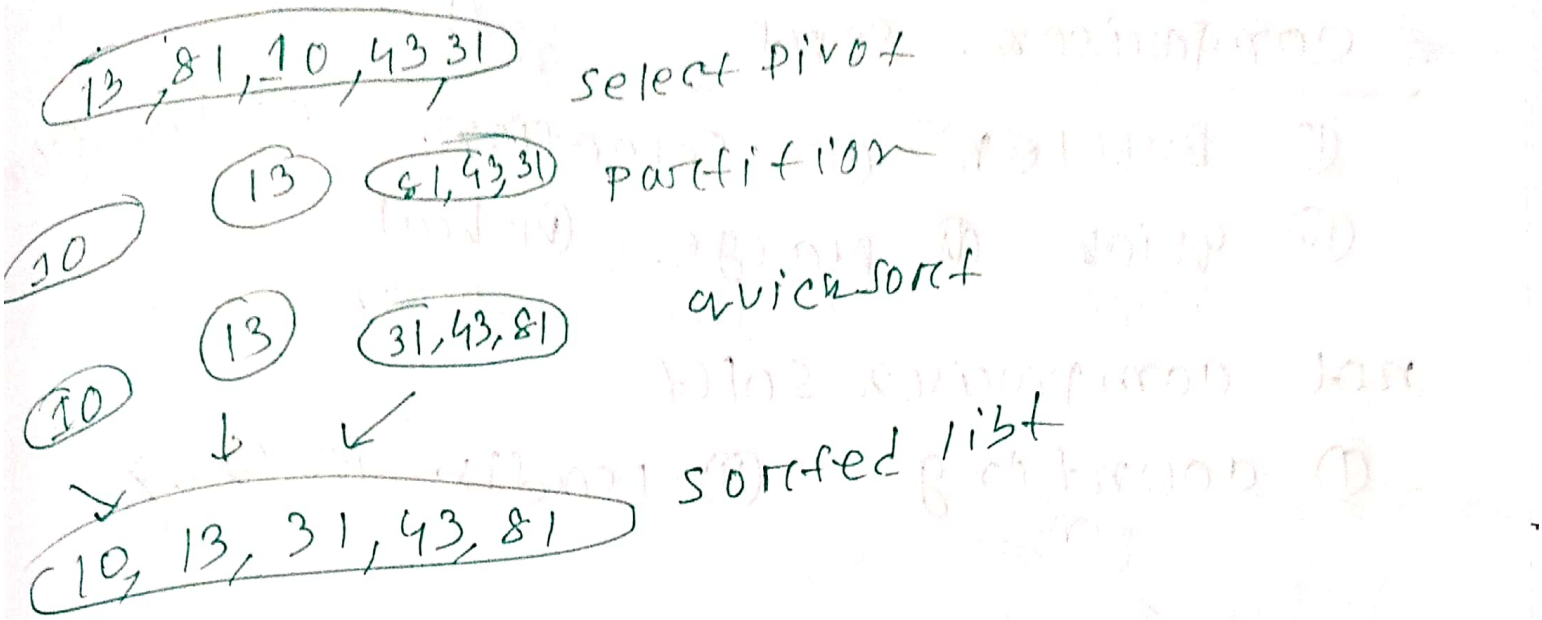




end < start
swap [end] and [lower bound]



select



The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter

comparison sort — n^2 (best + worst)

(I) Bubble (n^2) (II) selection (III) insertion (n^2)

(IV) quick $n \log n$ (V) merge $n \log n$ (VI) heap $n \log n$ (TC)

non comparison sort

(I) counting $O(n)$ (II) radix BUCKET

Bubble sort : (I) stable

(II) in place sort

(III) Tc (n^2)

But $O(n)$ when array is already sorted.

selection sort : (I) Tc (n^2)

(II) not stable (but we can made)

(III) in place

we can make stable by changing the key comparison operation

Insertion Sort (I) n^2 TC (worst + avg)

(II) space $O(1)$

(III) $O(n)$ when elements are already sorted. $O(n^2)$ we elements are reverse order.

(IV) in place

(V) yes stable

(VI) is used when number of element are small and. it is also useful when input array is almost sorted

counting sort → difference / range.

(I) TC $O(n+k)$ / $O(n)$

↓ number of element

k ~~=~~ $O(n)$ | k, $O(n^2)$ হলে খারাপ না

(II) space $O(n)$

(III) not stable but we can make

(IV) stable

(V) comparison based না,

Q1) Illustrate the operation of counting sort:

$A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$

A[11] =

6	0	2	0	1	3	4	6	1	3	2	
1	2	3	4	5	6	7	8	9	10	11	

C[6] =

0	1	2	3	4	5	6

```
for(int i=0; i<=11; i++)
    C[A[i]]++;
```

C[6] =

2	2	2	2	1	0	2
0	1	2	3	4	5	6

```
for(i=1; i<=6; i++
    C[i] = C[i] + C[i-1]
```

C[6] =

2	4	6	8	8	9	11
0	1	2	3	4	5	6

~~for (i=1 to 11)~~

for (i = 11, i >= 1 ; i--) {

~~b E e E A~~

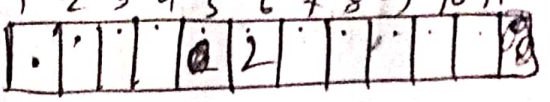
c [A[i]] -- ;

b[c[a[i]]] = a[i] ;

}

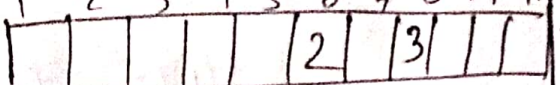
for i = 11 ;

B[11] =



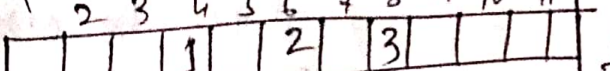
for i = 10

B[11] =

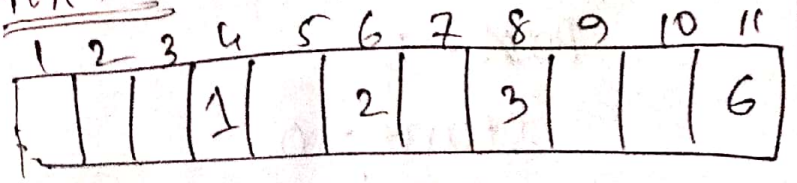


for i = 9

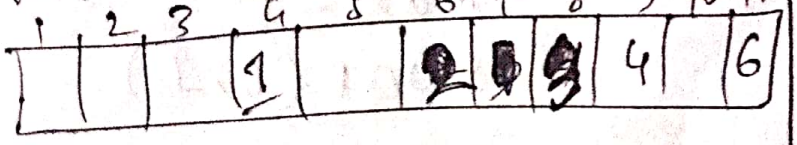
B[11] =



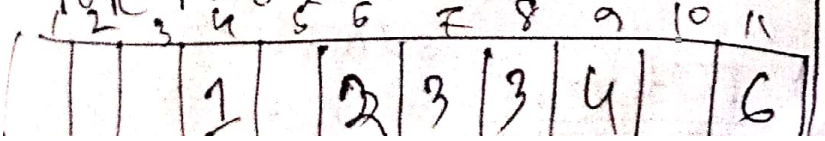
for i = 8



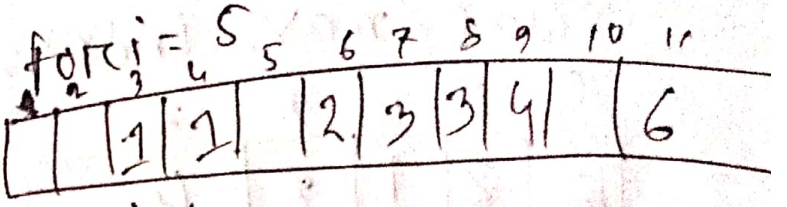
for i = 7



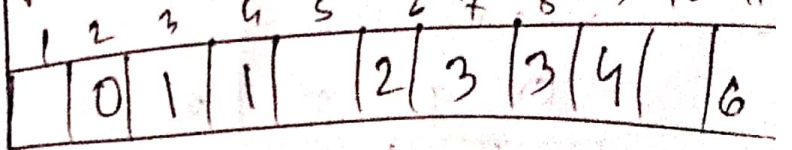
for i = 6



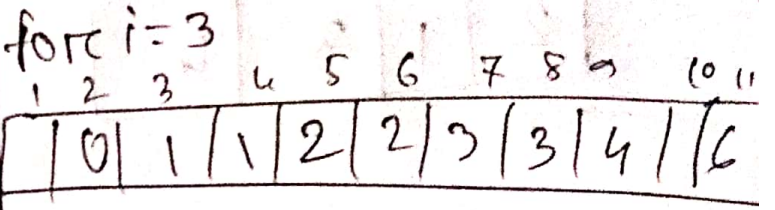
for i = 5



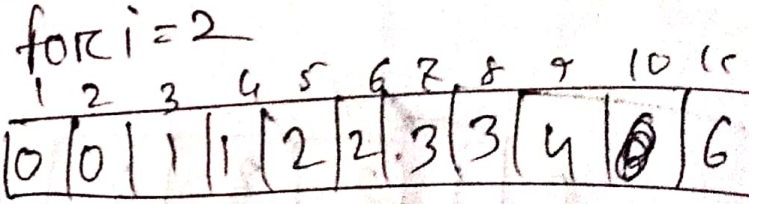
for i = 4



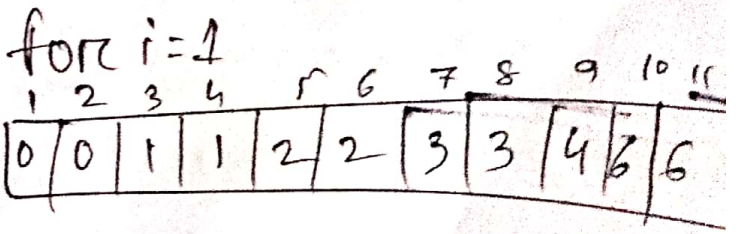
for i = 3



for i = 2

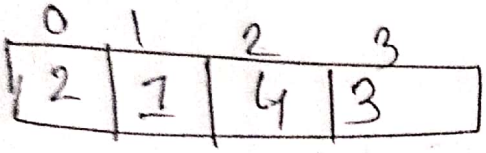


for i = 1



2020

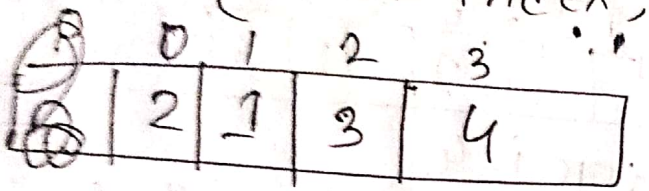
1) b) selection sort



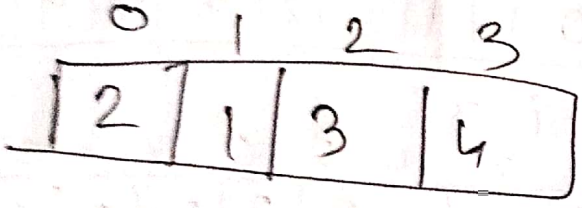
start = 0
end = 3
start < end

max_index = 2

swap(max_index, end)

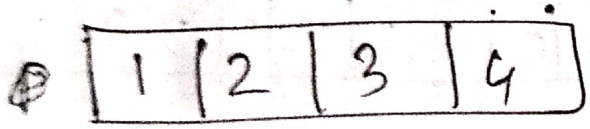


Pass-1



Pass-2

start = 0
end = 2
start < end
max = 2
swap(2, 2)



Pass-3

start = 0
end = 1
max = 0
swap(0, 1)

Pass-4

start = 0
end = 0

return

while analyzing TC of an algorithm why do we need to perform worst case analysis?

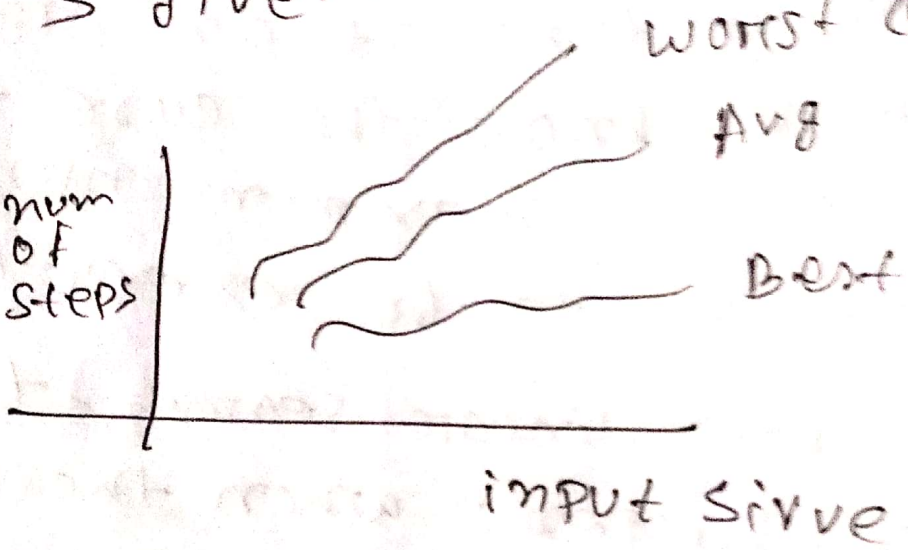
⇒ In the worst case-analysis, we calculate the upper bound on the running time of an algorithm.

- we must know the case that causes a maximum number of operations to be executed.

Ex for linear search, the worst case happens when the element to be searched (x) is not present in the array.

since we can seldom guarantee precisely how an algorithm will be exercised, usually we cannot obtain an avg-case measurement that is likely to be accurate. The worst case gives us an upper bound of performance.

Analyzing an algorithm's worst case guarantees that it will never perform worse than what we determine, no matter which input is given



what is the best TC that we can achieve by using comparison based sorting algorithms.

⇒ The comparison based sorting elements of an array are compared with each other to fing the sorted array

Best case TC n when array is already sorted → insertion

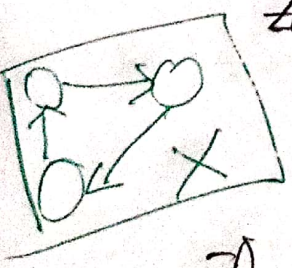
worst case n^2 whe array is reverse sorted

But in maximum time the TC is $n\log n$. we can not get better this time.

What does topological order do? what are the properties of a graph to have a topological order?

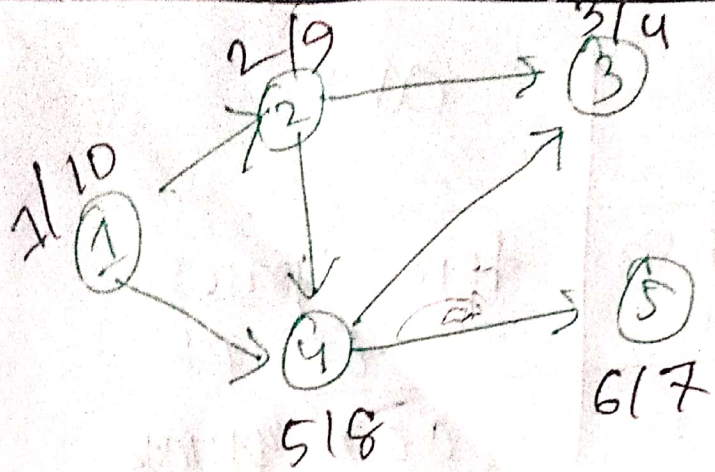
=> In computer science, a topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

=> Properties:



Graph should be directed and Acyclic.

=> Every DAG (Directed acyclic graph) have atleast one topological ordering.



topological order: $1 \to 2 \to 4 \to 5 \to 3$

or

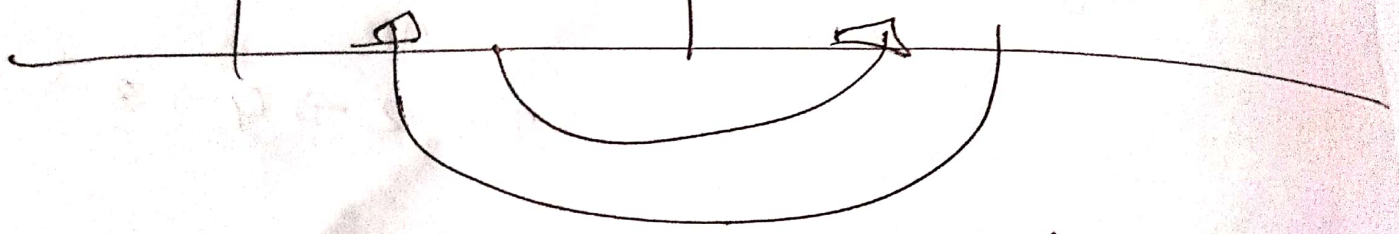
$1 \to 2 \to 4 \to 3 \to 5$

difference between adjecency matrix and list

geeks for geeks

traversing	matrix	list
storage space	$O\|V^2\|$	$O(V+E)$ traversing
adding vertex	$O\|V^2\|$	$O(1)$
adding edge	$O(1)$	$O(1)$

removing edge	O(1)	O(V)
use	for dense graph	for sparse graph
finding edge	requires more time to find	can quick find if an edge exinst



* we use Adjecency list.

limitation : the determining whether there is an edge from vetex i to vertex j may take as many as n steps.

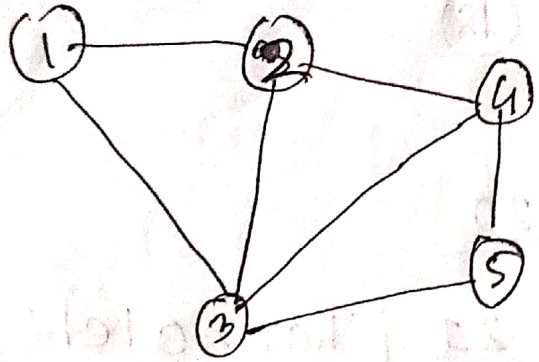
~~to~~ ~~limi~~ ~~to~~

limitations show when we need to check if two nodes have a direct edge or not.

walk / trail path
distinguish with example

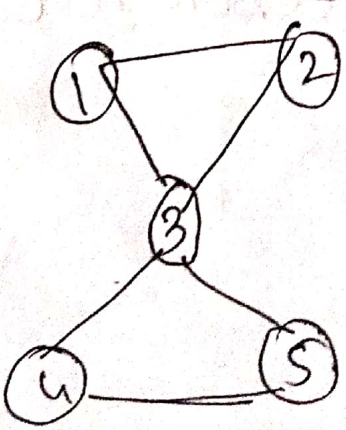
walk

a sequence of edges and vertices in a network, where vertices and edges can be repeated



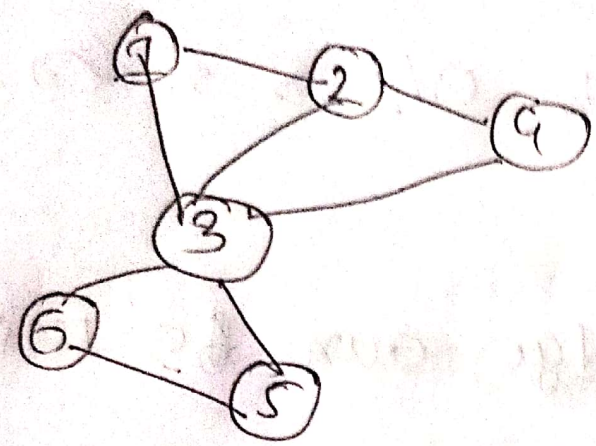
walk: $1 \to 2 \to 4 \to 5 \to 3 \to 4 \to 2 \to 1 \to 3$

trail: ~~vertex~~ a walk where vertex can be repeted but edge cannot repeted.



$1 \to \underline{3} \to 4 \to 5 \to \underline{3} \to 2$

Path : a trail that connect all vertices. Vertex neither no edge can repeated



$5 \to 6 \to 3 \to 4 \to 2 \to 1$