

W① Insertion Sort: Insertion sort is a simple and intuitive sorting algorithm.

It works the same way we arrange cards in our hands. we take one card at a time & insert it into its correct position among the already sorted card.

Pseudo code:

INSERTION-SORT (A)

1. for $j = 2$ to $A.length$

2. $key = A[j]$

3. $i = j - 1$

4. while $i > 0$ and $A[i] > key$

5. $A[i+1] = A[i]$

6. $i = i - 1$

7. $A[i+1] = key$

Subject:

Date:

Simulation:

let an array with size $n=7$

$A =$

7	4	5	2	0	1	3
---	---	---	---	---	---	---

1 2 3 4 5 6 7

Step-01:-

let 7 is sorted part $\rightarrow A[1] = 7$

7	4	5	2	0	1	3
---	---	---	---	---	---	---

↑ i unsorted
sorted

Now, $A[2] = 4$ which is $key = 4$

compare with $A[1] = 7 \rightarrow 7 > 4$

so shift 7 right side and insert

$key = 4$ at position 1

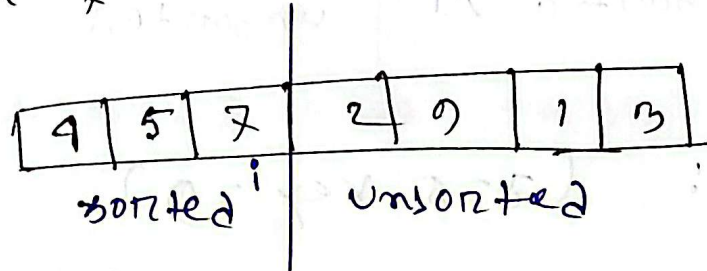
4	7	5	2	0	1	3
---	---	---	---	---	---	---

↑ i sorted unsorted

Subject:.....

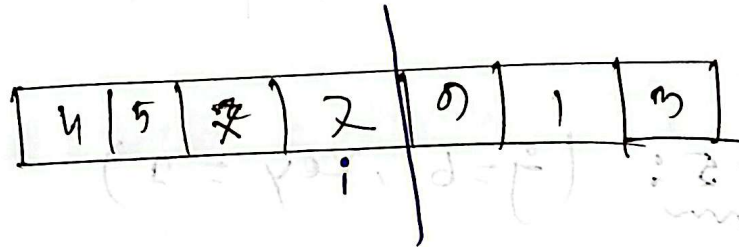
Date:.....

Step-02 :- Now, $A[3] = 5$ and $key = 5$
compare with $A[2] = 7 \rightarrow 7 > key$,
shift 7



Step-03 :- Now, ($j = 4$, $key = 2$)

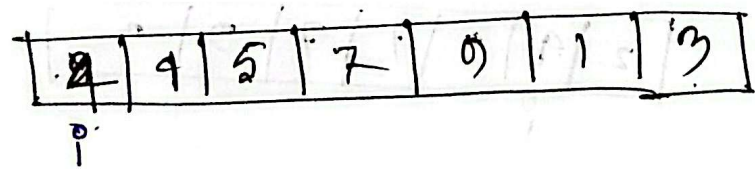
$\rightarrow i = 3$
 $\rightarrow$ compare $A[3] = 7 > 2 \rightarrow$ shift



here, $i = 2 \rightarrow A[2] = 5 > 2 \rightarrow$ shifted



And, $i = 1 \rightarrow A[1] = 4 > 2 \rightarrow$ shifted



Subject:.....

Date:.....

$i=0$ (stop) $\rightarrow$ insert 2 at $A[1]$

1	2	3	4	5	6	7
2	4	5	7	0	1	3
sorted $\uparrow i$				unsorted		

Step-04: ($j=5$, $key=0$)

$i=4$, compare $A[4]=7$ with $0 \rightarrow 7 < 0$ (stop)

No shift needed.

1	2	3	4	5	6	7
2	4	5	7	0	1	3
sorted				$\uparrow$ 6 1	unsorted	

Step-05: ($j=6$, $key=1$)

$i=5 \rightarrow A[5]=0 > 1 \rightarrow$ shifted

2	4	5	7	0	0	3
				$\uparrow$ i		

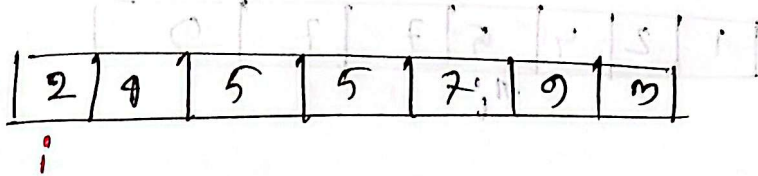
$i=4 \rightarrow A[4]=7 > 1 \rightarrow$ shifted

2	4	5	7	7	0	3
				$\uparrow$ i		

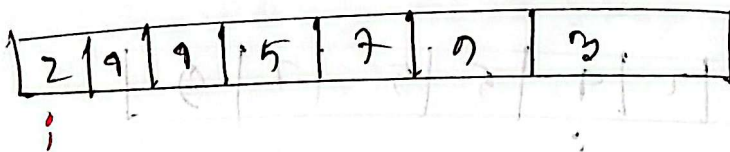
Subject:

Date:

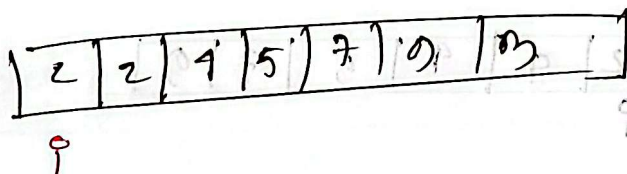
$i=3 \rightarrow A[3] = 5 > 2 \rightarrow \text{shifted}$



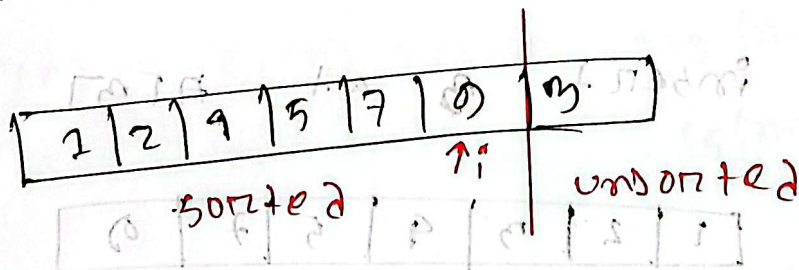
$i=2 \rightarrow A[2] = 4 > 2 \rightarrow \text{shifted}$



$i=1 \rightarrow A[1] = 2 > 2 \rightarrow \text{shifted}$

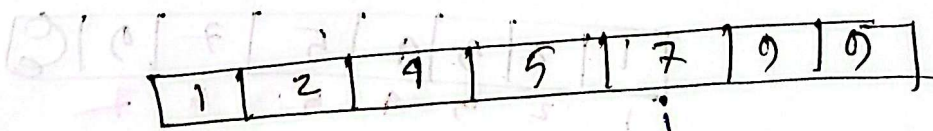


$i=0 \rightarrow A[0] = \text{invalid} \rightarrow 2 \text{ at } A[1]$



Step-06: ($j=7$, $key=13$)

$i=6$, $A[6] = 9 > 13 \rightarrow \text{shifted}$



Subject:.....

Date:.....

$i=5$, $A[5] = 7 > 3 \rightarrow$ shifted

1	2	4	5	7	7	9
---	---	---	---	---	---	---

$i=4$, $A[4] = 5 > 3 \rightarrow$ shifted

1	2	4	5	5	7	9
---	---	---	---	---	---	---

$i=3$, $A[3] = 4 > 3 \rightarrow$ shifted

1	2	4	4	5	7	9
---	---	---	---	---	---	---

$i=2$, $A[2] = 2 < 3$ (stop)

insert 3 at $A[3]$

1	2	3	4	5	7	9
---	---	---	---	---	---	---

Final sorted Array :

1	2	3	4	5	7	9
1	2	3	4	5	6	7

Subject:

Date:

W complexity Analysis:

	cost	time
INSETION-SORT (A)		
1. For $j = 2$ to $A.length$	c_1	n
2. $key = A[j]$	c_2	$n-1$
3. $i = j-1$	c_3	$n-1$
4. while $i > 0$ & $A[i] > key$	c_4	$\sum_{j=2}^n t_j$
5. $A[i+1] = A[i]$	c_5	$\sum_{j=2}^n (t_j - 1)$
6. $i = i-1$	c_6	$\sum_{j=2}^n 1$
7. $A[i+1] = key$	c_7	$n-1$

For Best case (Already sorted)

Every while fails immediately $t_j = 1$

$$\text{So, } T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7(n-1)$$

This is $O(n) \rightarrow$ linear time.

$$\therefore T_c = O(n)$$

Subject:.....

Date:.....

For Worst case (reverse sort):

Every while runs maximum $\rightarrow i=j$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \quad \text{and} \quad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

So,

$$T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4 \times \left[\frac{n(n+1)}{2} - 1 \right] + (c_5 + c_6) \left[\frac{n(n-1)}{2} \right] + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + (c_1 + c_2 - c_8) n - (c_2 + c_3 + c_4 + c_7)$$

this simplifies to

$$T(n) = an^2 + bn + c$$

thus T.C. = $O(n^2)$ $\rightarrow$ Quadratic time

Subject:.....

Date:.....

Average case: (random input)

On average, half the elements in subarray are checked $\rightarrow i \approx n/2$.

So, summations $\approx$ half of worst case still

$O(n^2)$.

$\therefore T.C. = O(n^2)$.

Best case: $O(n)$

Average case: $O(n^2)$

Worst case: $O(n^2)$

Space complexity:

Insertion sort has $O(1)$ space complexity because it sorts the array in place, using a constant array of extra memory (for variable like `key` and `i`).

(2)

Merge Sort: Merge sort is a divide-and-conquer sorting algorithm.

It works by repeatedly dividing the input array into two halves, sorting each half, and then merging the two sorted halves into one final sorted array.

→ uses recursion

→ stable sorting algorithm

→ always run in $O(n \log n)$ time.

Pseudocode:

MERGE-SORT (A, l, r)

1. if $l < r$

2. $\quad mid = \lceil (l + r) / 2 \rceil$

3. $\quad$ MERGE-SORT (A, l, mid)

4. $\quad$ MERGE-SORT ($A, mid+1, r$)

5. $\quad$ MERGE (A, l, mid, r)

6. MERGE (A, l, mid, r)

7. $\quad n_1 = mid - l + 1$

Subject:.....

Date:.....

```
8.      n2 = r - mid
9.      create arrays L[n1], R[n2]
10.     for i = 1 to n1
11.         L[i] = A[i-1]
12.     for j = 1 to n2
13.         R[j] = A[mid+j]
14.     i = 1
15.     j = 1
16.     k = left
17.     while i ≤ n1 and j ≤ n2
18.         if L[i] ≤ R[j]
19.             A[k] = L[i]
20.             i = i + 1
21.         else, A[k] = R[j]
22.             j = j + 1
23.             k = k + 1
24.         while i ≤ n1
25.             A[k] = L[i]
26.             i = i + 1
27.             k = k + 1
```

28. while $j \leq n/2$

29. $A[k] = A[j]$

30. $j = j + 1$

31. $k = k + 1$

32. END while.

✓ Simulation:

Let an array $A = [38 | 27 | 43 | 3 | 0 | 80]$

here, $left = A[1] = 38$

$right = A[6] = 80$

$mid = \left\lceil \frac{1+6}{2} \right\rceil = 3$

Step-01: First split the array into two subarray.

left half:

38	27	43
----	----	----

right half:

3	0	80
---	---	----

Subject:.....

Date:.....

STEP - 02: Recursively sort left half:-

Now divide

38	27	43
----	----	----

 into two halves.

38

 and

27	43
----	----

here 38 is single element, which is sorted.

Now sort

27	43
----	----

 :-

split into two halves

27

43

merge 27 and 43 with compare 27 and

43. New array

27	43
----	----

Subject:.....

Date:.....

→ Now merge

38
0

 and

27	43
1	2

compare 38 and 27 → pick 27

Array

27		
0	1	2

compare 38 and 43 → pick 38

Array

27	38	
0	1	2

copying remaining 43. Final merged array

27	38	43
0	1	2

Step - 03: Recursively sort right subarray

3	0	80
3	4	5

→ split into

3
3

 and

0	80
4	5

Subject:

Date:

3 is already sorted place it back

Now split right array

0

80

compare 0 and 80, which are already sorted.

Now merge 3, 0, 80

3 | 0 | 80

So, sorted left array and right array back in the main array.

27 | 38 | 43
0 1 2

3 | 0 | 80
3 4 5

Now compare 27 and 3, 0 we get

3 | 0 | 27 | | |
0 1 2 3 4 5

And compare 38 and 80 then 43 and 80

then we get 3 | 0 | 27 | 38 | 43 | 80
0 1 2 3 4 5

Subject:.....

Date:.....

Complexity analysis:

$$\text{mergesort}(l, r) \longrightarrow T(n)$$

$$\text{if } (l < r) \longrightarrow 1$$

$$\text{mid} = (l + r) / 2 \longrightarrow 1$$

$$\text{mergesort}(l, \text{mid}) \longrightarrow T(n/2)$$

$$\text{mergesort}(\text{mid} + 1, r) \longrightarrow T(n/2)$$

$$\text{merge}(l, \text{mid}, r) \longrightarrow n$$

$$\text{Total, } T(n) = 2T(n/2) + n$$

Now, Recurrence relation,

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

$$2T(n/2) + n \quad \text{where } n=2^k$$

for $k > 1$

Subject:

Date:

these $n/2$ element then,

$$T(n) = 2 \left[2T(n/4) + n/2 \right] + n$$

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

$$= 2^k T(n/2^k) + kn$$

It can be ^{divided} maximum until there is one element left. So, then

$$\frac{n}{2^k} = 1$$

$$\text{or, } 2^k = n$$

$$\text{or, } k = \log_2 n$$

$$\therefore T(n) = n \cdot T(1) + n \cdot \log_2 n$$

$\therefore$ the time complexity is $O(n \log n)$

this are ~~best~~ all case.

Subject:.....

Date:.....

Best case:

$$T.C. = O(n \log n)$$

Average case:

$$T.C. = O(n \log n)$$

Worst case:

$$T.C. = O(n \log n)$$

Space complexity:

Merge sort has $O(n)$ space complexity because it needs an auxiliary array of size n to temporarily hold elements while merging.

③

Assignment - 02

Subject:

Date:

✓ Quick Sort: Quick sort, like merge sort, applies the divide and conquer paradigm introduced.

Divide: Partition (rearrange) the array $A[p \dots r]$ into two (possibly empty) subarray. One is $A[q+1 \dots r]$ and another is $A[p \dots q-1]$

Conquer: Sort the two subarray $A[p \dots q-1]$ and $A[q+1 \dots r]$ by recursive call to quick sort.

Combine: Because the subarray already sorted, no work is needed combine them. The entire array $A[p \dots r]$ is now sorted.

Subject:

Date:

✓ Pseudocode:

QUICKSORT (A, P, r)

1. if $P < r$
2. $q = \text{partition}(A, P, r)$
3. $\text{QUICKSORT}(A, P, q-1)$
4. $\text{QUICKSORT}(A, q+1, r)$

To sort an array, the entire array A, the initial quicksort (A, 1 --- A.length)

✓ Partition the Array:

PARTITION (A, P, r)

1. $x = A[r]$
2. $i = P - 1$
3. for $j = P$ to $r - 1$
4. if $A[j] \leq x$
5. $i = i + 1$

Date :.....

7. $\leftarrow$ exchange $A[i+1]$ with $A[i]$

8. $\pi e + 4\pi n$ $i+2$ F 0 0

Simulation:

let an array =

2	8	7	1	3	5	6	4
---	---	---	---	---	---	---	---

let $i=0, j=7$ and $\text{pivot} = \text{mid} = 3$ (7)

where $\min d = A \left[\frac{0+7}{2} \right] = A[3] = 2$

Step-01:

Diagram illustrating a binary tree structure (likely a B-tree or B+ tree) with 8 slots. The slots contain values: 2, 8, 7, 1, 3, 5, 6, 4. Above the slots are indices: 0, 1, 2, 3, 4, 5, 6, 7. Below the slots, an arrow points to the slot containing 1, labeled "Pivot".

move i to right until $arr[i] \geq pivot$
 $arr[i] < pivot$

move i to right until $a[i] \leq \text{pivot}$
move j to left until $a[j] \geq \text{pivot}$

Subject:.....

Date:.....

here, $i=0 \rightarrow arr[0] = 2 > 1$ (stop i)

$j=7 \rightarrow arr[7] = 4 > 2 \rightarrow j=6$

0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4
i			pivot			j	

$arr[6] = 6 > 1 \rightarrow j=5$

$arr[5] = 5 > 1 \rightarrow j=4$

0	1	2	3	4	5	6	7
2	8	7	1	3	5	6	4
i			pivot	j	j		

$arr[4] = 3 > 1 \rightarrow j=3$

$arr[3] = 1 < 1$ (stop j)

$i=0, j=3 \rightarrow$ swap $arr[0]$ and $arr[3]$

0	1	2	3	4	5	6	7
1	8	7	2	3	5	6	4
i		j					

Now, $i=1, j=2 \rightarrow$ continue.

Subject:

Date:

step-02:

$i=1$, $arr[1] = 8 > 1$ (stop)

$j=2$, $arr[2] = 7 > 1 \rightarrow j=1$

now, $i=1$, $j=1 \rightarrow$ they met

partition ends, return pivot index = 2

After first partition we get the

array is

1	8	7	2	3	5	6	4
0	1	2	3	4	5	6	7

left =

1
0

 $\rightarrow$ already sorted

right =

8	7	2	3	5	6	4
1	2	3	4	5	6	7

 $\rightarrow$ unsorted

Now second partition:

8	7	2	3	5	6	4
1	2	3	4	5	6	7

↑
pivot

$pivot = arr[3] = 3$

Subject:.....

Date:.....

Now, $i=1, j=7$

move $i \rightarrow \text{arr}[i] = 8 > 3$ (stop)

move $j \rightarrow \text{arr}[j] = 4 < 3 \rightarrow j=6$

$\text{arr}[6] = 6 > 3 \rightarrow j=5$

$\text{arr}[5] = 5 > 3 \rightarrow j=4$

$\text{arr}[4] = 3 < 3$ (stop)

i				j				j
	$\downarrow$			$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$
1	8	7	2	3	5	6	4	
0	1	2	3	4	5	6	7	
				Pivot				

Now, swap $\text{arr}[i]$ and $\text{arr}[4]$

i				j				
	$\downarrow$			$\downarrow$				
1	3	7	2	8	5	6	4	
0	1	2	3	4	5	6	7	

Subject:.....

Date:.....

Continue partition: —

$$j = 2 \rightarrow arr[2] = 7 > 3 \text{ (stop)}$$

$$i = 3 \rightarrow arr[3] = 2 \leq 3 \text{ (stop)}$$

swap $arr[2]$, $arr[3]$

0	1	2	3	4	5	6	7
1	3	2	7	8	5	6	4

↑
i
↑
j

now $i = 3$, $j = 2 \rightarrow \text{stop}$

Next partition:

3	4	5	6	7
7	8	5	6	4

↑
pivot

$$\text{pivot} = arr[5] = 5$$

$$i = 3 \rightarrow arr[3] = 7 > 5 \text{ (stop)}$$

$$j = 7 \rightarrow arr[7] = 4 < 5 \text{ (stop)}$$

swap $arr[3]$ and $arr[7]$

0	1	2	3	4	5	6	7
1	2	3	4	8	5	6	7

pivot

j

Subject:.....

Date:.....

Now, $i = 4, j = 6$

$arr[4] = 8 > 5$ (stop)

$arr[6] = 6 > 5 \rightarrow j = 5$

$arr[5] = 5 \leq 5$ (stop)

swap $arr[4], arr[5]$

0	1	2	3	4	5	6	7
1	2	3	4	5	8	6	7
			j	i			

Next partition:

$pivot = A[6] = 6$

1	2	3	4	5	8	6	7
				i	pivot	j	

$i = 5, j = 7$

$arr[5] = 8 > 6$ (stop)

$arr[7] = 6 \leq 6$ (stop)

swap them the we get the sorted array.

Subject:.....

Date:.....

W complexity analysis:

QuickSort(A, p, r)

if (p < r)

→ $O(1)$

q = partition(A, p, r) → $O(n)$ for subarray n

QuickSort(A, p, q-1) → $T(k)$ left side has k elements

QuickSort(A, q+1, r) → $T(n-k-1)$

partition(A, p, r)

· pivot = A[r]

→ $O(1)$

i = p-1

→ $O(1)$

for j = p to r-1

→ $O(n)$

if A[j] ≤ pivot → $O(1)$

i = i+1

→ $O(1)$

swap A[i], A[j]

→ $O(1)$

swap A[i+1], A[r]

→ $O(1)$

return i+1

→ $O(1)$

Subject:.....

Date:.....

Now, partition on a subarray of size n takes $O(n)$ (for-loop)

Quick Sort cost on size n with pivot ending at position k gives recurrence

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

where, k is the number of elements in the left subarray.

Worst case analysis:

If pivot always produces partition of size 0 and $n-1$ (already sorted array with last element pivot) then $k=0$ each time

Recurrence:

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

Subject:.....

Date:.....

Now,

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

$$= T(n-2) + T(n-1) + cn$$

.....

$$= T(1) + c(2+3+\dots+n)$$

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

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

$$T(n) = O(n^2)$$

Best case analysis: (perfectly balanced partition)

If pivot always splits array in half ($k \approx n/2$),

recurrence becomes :-

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

Subject:.....

Date:.....

Apply master's theorem we get

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

$$\text{then } n^{\log_b a} = n$$

then we find,

$$\therefore T(n) = O(n \log n)$$

Average case :

when pivot is equally likely fall at

any position $i(0 \leq i \leq n-1)$ then recurrence

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

Because the left & right subarray are symmetric :-

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

Subject:.....

Date:.....

$$\text{or, } n T(n) = 2 \sum_{i=0}^{n-1} T(i) + O(n^2)$$

Now, subtract ~~from~~ ^{for} $n-1$

$$n T(n) - (n-1) T(n-1) = 2 T(n-1) + O(n)$$

$$\text{or, } T(n) = \frac{n+1}{n} T(n-1) + O(1)$$

now unfolding this recurrence relation

we get,

$$\frac{T(n)}{T(n+1)} = T(1)/2 + O\left(\sum_{i=2}^n \frac{1}{i}\right)$$

$$\therefore T(n) = O(n \log n)$$

Best case : $O(n \log n)$

Average case: $O(n \log n)$

Worst case : $O(n^2)$

Subject:.....

Date:.....

Space complexity :

Worst case : $O(n) \rightarrow$ when partition are unbalanced (recursion tree height n)

Best case : $O(\log n) \rightarrow$ when partition are balanced (recursion tree height $\log n$)

④

Subject:.....

Date:.....

Bubble Sort

Bubble sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order. The largest element "bubbles up" to the last position after each pass.

Pseudocode:

BUBBLE-SORT(A):

1. $n = \text{length}(A)$
2. for $i = 0$ to $n - 1$
3. swapped = false
4. for $j = 0$ to $n - i - 2$
5. if $A[j] > A[j+1]$
6. swap $A[j]$ and $A[j+1]$
7. swapped = true
8. if swapped == false
9. break

Date :.....

Simulation:

let an unsorted array =

22	14	12	18	19
0	1	2	3	4

Iteration - 01 :- (i=0)

compare $j = 20 \rightarrow 22$ & $14 \rightarrow 22 > 14$

- swap.

22 14 12 18 40

14 22 12 18 10

14 12 22 18 10

74 $(12 \times 10^3) = 122 \text{ 90}$

$14 \leq 12 + 18 = 30$
 $22 \leq 6 + 9 + 20$
 sorted

largest element 22 bubbled to last
elem. position.

Subject:

Date:

iteration-02 : (i=2)

14 12 18 00 22



12 14 18 00 22



12 14 18 00 22



12 14 18 00 22

sorted

iteration-03 : (i=2)

(0-0) ← 12 14 0 18 22

(i-0) ← 12 14 0 18 22



0 ←

12 14 0 18 22



12 0 14 18 22

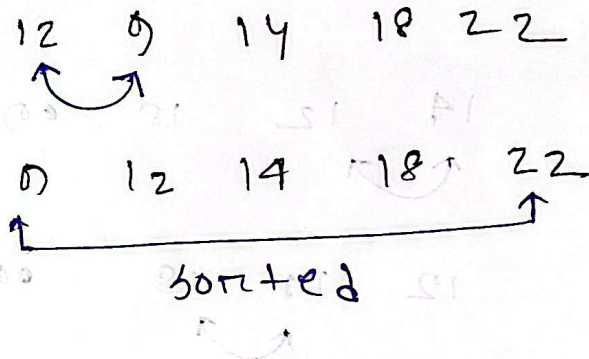
sorted

so the array is sorted

Subject:.....

Date:.....

iteration - 04: (i=1) : 50 - not sorted



Now the array is sorted

Complexity analysis:-

Bubble-Sort (A, n) : 50 - not sorted

for i=1 to n-1 $\rightarrow (n-1)$

for j=1 to n-i $\rightarrow (n-i)$

if $A[j] > A[j+1]$ $\rightarrow 1$

swap $A[j]$ and $A[j+1]$ $\rightarrow 1$

Subject:.....

Date:.....

Worst - case: Total number of comparison

are,

$$T(n) = n-1 + n-2 + \dots + 3 + 2 + 1$$

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

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

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

→ quadratic function.

$$\therefore T(n) = O(n^2)$$

So, for worst case occurs when the array is in reverse order.

Average case:

On average, about half swaps will occur
so complexity must be $O(n^2)$

Subject:.....

Date:.....

Best case:

Occurs when the array is already sorted.

Inner loops run only once, so only $n-1$ comparison and 0 swaps.

$\therefore$ complexity : $O(n)$

There For, $\leftarrow$

Best case : $O(n)$

Average case : $O(n^2)$

Worst case : $O(n^2)$

Space complexity:

Bubble sort is in-place, it only uses a constant amount of extra memory (for variables like i , j , swapped).

$\therefore$ complexity : $O(1)$

5

Subject:

Date:

Counting Sort:

Counting sort is a non-comparison based sorting algorithm. It works efficiently when the elements are integers within a known, small range.

Note:

- count the frequency of each element
- use the count to place elements directly into the sorted array.

This is stable sorting technique and time complexity depends on the range of numbers.

Subject:.....

Date:.....

W Pseudocode:

COUNTING-SORT (A, B, K)

1. let $c[0..K]$ be a new array

2. for $i = 0$ to K

3. $c[i] = 0$

4. for $j = 1$ to $A.length$

5. $c[A[j]] = c[A[j]] + 1$

6. for $i = 1$ to K

7. $c[i] = c[i] + c[i-1]$

8. for $j = A.length$ down to 1

9. $B[c[A[j]]-1] = A[j]$

10. $c[A[j]] = c[A[j]] - 1$

11. return B

Subject:.....

Date:.....

✓ simulation :

let an ~~an~~ unsorted array, $A = \{1, 2, 4, 3, 0, 2, 1, 7, 1, 4, 3, 0\}$

where size, $n = 12$.

Now we should solving step-by-step for sorting using counting sort technique given below :

Initial Array : -

$j \rightarrow$	0	1	2	3	4	5	6	7	8	9	10	11
$A =$	1	2	4	3	0	2	1	2	1	4	3	0

① find maximum value : -

Scan $A \rightarrow \max = 7$

let a count array for counting frequency of each elements, where count array size = $\max + 1 = 8$ (indices $0 \dots 7$)

Subject:.....

Date:.....

② Initialize Count Array

$i \rightarrow 0, 1, 2, 3, 4, 5, 6, 7$
count =

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

③ Frequency count (loop through A left to right)

code: (For this part only)

for ($i=0$; $i < n$; $i++$)

{
 value = A[i];
 count[value]++;
}

Now, increment count[value] for each element.

$i \rightarrow 0, 1, 2, 3, 4, 5, 6, 7$
 $\rightarrow i=0, \text{value}=1 \Rightarrow \text{count} =$

0	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

$i=1, \text{value}=2 \Rightarrow \text{count} =$

0	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

$i=2, \text{value}=4 \Rightarrow \text{count} =$

0	1	1	0	2	0	0	0
---	---	---	---	---	---	---	---

$i=3, \text{value}=3 \Rightarrow \text{count} =$

0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

$i=4, \text{value}=0 \Rightarrow \text{count} =$

2	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

Subject:.....

Date:.....

$i=5$, value = 2, count =

0	1	2	3	4	5	6	7
1	2	2	1	1	0	0	0

$i=6$, value = 1, count =

0	1	2	3	4	5	6	7
2	2	2	2	1	0	0	0

$i=7$, value = 2, count =

0	1	2	3	4	5	6	7
1	2	2	1	1	0	0	1

$i=8$, value = 1, count =

0	1	2	3	4	5	6	7
1	3	2	1	1	0	0	1

$i=9$, value = 4, count =

0	1	2	3	4	5	6	7
1	3	2	1	2	0	0	1

$i=10$, value = 2, count =

0	1	2	3	4	5	6	7
1	2	2	2	2	0	0	1

$i=11$, value = 0, count =

0	1	2	3	4	5	6	7
2	3	2	2	2	0	0	1

Final Frequency counts :-

0	1	2	3	4	5	6	7
2	3	2	2	2	0	0	1

here we can see that, count array
 0 appears 2 times, 2 appears 2 times,
 1 appears 3 times, 4 appears 2 time
 and 7 appears one time.

Subject:.....

Date:.....

4) cumulative counts : (prefix sum)

convert count so that $\text{count}[v] = \text{numbers of elements} \leq v$.

i →	0	1	2	3	4	5	6	7
	2	3	2	2	2	0	6	1

code:

```

for (int i = 1; i <= max; i++) {
    count[i] += count[i-1];
}

```

now →

$i=1, \text{count}[1] += \text{count}[0] \rightarrow 3+2=5$

i →	0	1	2	3	4	5	6	7
	2	5	2	2	2	0	0	1

$i=2, \text{count}[2] += 2 \times \text{count}[1] \rightarrow 2+5=7$

	0	1	2	3	4	5	6	7
	2	5	7	2	2	0	0	1

$i=3, \text{count}[3] += \text{count}[2] \rightarrow 2+7=9$

	0	1	2	3	4	5	6	7
	2	5	7	9	2	0	0	1

Subject:

Date:

$$i = 4, \text{count}[4] + = \text{count}[3] \rightarrow 2 + 9 = 11$$

0	1	2	3	4	5	6	7
2	5	7	0	11	0	0	1

$$i = 5, \text{count}[5] + = \text{count}[4] \rightarrow 0 + 11 = 11$$

0	1	2	3	4	5	6	7
2	5	7	0	11	11	0	1

$$i = 6, \text{count}[6] + = \text{count}[5] \rightarrow 0 + 11 = 11$$

0	1	2	3	4	5	6	7
2	5	7	0	11	11	11	1

$$i = 7, \text{count}[7] + = \text{count}[6] \rightarrow 1 + 11 = 12$$

0	1	2	3	4	5	6	7
2	5	7	0	11	11	11	12

$\therefore$ cumulative

count array

0	1	2	3	4	5	6	7
2	5	7	0	11	11	11	12

there are 2 elements ≤ 0 , 5 elements ≤ 4 ,
7 elements ≤ 7 ... 12 elements ≤ 7 .

Create output array using malloc which is initially all indices are blank and size 12.

```
for (int i = 1; i >= 0; i--) {  
    cout << "count[" << A[i] << "] = " << A[i] << endl;
```

→ Now,

$$i=11, \quad A\tau_{11} = 0$$

-- count[0] = 2 $\rightarrow$ 1 (decrement by 1)

place at position 1 of output array

$$\text{output}[1] = 0$$

count

0	1	2	3	4	5	6	7
2	5	7	9	11	11	11	12

[illegible]

Subject:.....

Date:.....

$i = 10$, $A[10] = 3$, $--count[3] = 0 \rightarrow 8$ (decrement 1)

place at position 8: $output[8] = 3$

	0	1	2	3	4	5	6	7	
count	1	5	7	8	11	11	11	12	

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

$i = 9$, $A[9] = 4$

$--count[4] = 11 \rightarrow 10$

$output[10] = 4$

	0	1	2	3	4	5	6	7	
count	1	5	7	8	10	11	11	12	

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

$i = 8$, $A[8] = 1$,

$--count[1] = 5 \rightarrow 4$

$output[4] = 1$

	0	1	2	3	4	5	6	7	
count	1	4	7	8	10	11	11	12	

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

Subject:.....

Date:.....

$$i=7, A[7]=7$$

$$-- \text{count}[7] = 12 \rightarrow 11$$

$$\text{output}[11] = 7$$

0 1 2 3 4 5 6 7

count

1	4	7	8	10	11	11	11
---	---	---	---	----	----	----	----

output

	0			1				3		4	7
--	---	--	--	---	--	--	--	---	--	---	---

$$i=8, A[8]=1$$

$$-- \text{count}[1] = 4 \rightarrow 3$$

$$\text{output}[3] = 1$$

count

1	3	7	8	10	11	11	11
---	---	---	---	----	----	----	----

output

	0		1	1				3		4	7
--	---	--	---	---	--	--	--	---	--	---	---

$$i=5, A[5]=2$$

$$-- \text{count}[2] = 7 \rightarrow 6$$

$$\text{output}[6] = 2$$

count

1	3	6	8	10	11	11	11
---	---	---	---	----	----	----	----

output

	0		1	1		2		3		4	7
--	---	--	---	---	--	---	--	---	--	---	---

Subject:

Date:

$$i = 4, A[4] = 0$$

$$--count[0] = 1 \rightarrow 0$$

$$output[0] = 0$$

count

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

output

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

$$i = 3, A[3] = 2$$

$$--count[2] = 8 \rightarrow 7$$

$$output[7] = 2$$

count

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

output

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

$$i = 2, A[2] = 4$$

$$--count[4] = 10 \rightarrow 9$$

$$output[9] = 4$$

count

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

output

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

Subject:.....

Date:.....

$i = 1, A[1] = 2$

--count[2] = 6 $\rightarrow$ 5

output[5] = 2

count	0	1	2	3	4	5	6	7
	0	2	5	7	9	11	11	11

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

$i = 0, A[0] = 1$

--count[1] = 3 $\rightarrow$ 2

output[2] = 1

count	0	1	2	3	4	5	6	7
	0	2	5	7	9	11	11	11

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

Final output array

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

Subject:.....

Date:.....

✓ complexity Analysis:

let, $\rightarrow n =$ number of elements in the array

$\rightarrow k =$ maximum value in the array (range 0 to k)

Now we get,

COUNTING-SORT (A, n)

$max = A[0] \rightarrow O(1)$

for $i = 1$ to $n-1 \rightarrow O(n)$

if $A[i] > max \rightarrow O(n)$

$max = A[i] \rightarrow O(n)$

$count[0 \dots max] = 0 \rightarrow O(k)$

for i to $n-1 \rightarrow O(n)$

$count[A[i]] = count[A[i]] + 1 \rightarrow O(n)$

for i to $max \rightarrow O(k)$

$count[i] += count[i-1] \rightarrow O(k)$

$output[0 \dots n-1] \rightarrow O(n)$

for $i = n-1$ down to $0 \rightarrow O(n)$

$output[-count[A[i]]] = A[i] \rightarrow O(n)$

for i to $n-1 \rightarrow O(n)$

$A[i] = output[i] \rightarrow O(1)$

Subject:.....

Date:.....

Now,

$$T(n, k) = O(n) + O(k) + O(n) + \dots + O(n)$$

which simplifies to $O(n+k)$ for all case.

Time complexity: $O(n+k)$

for all cases $\rightarrow O(n+k)$

Space complexity: $O(n+k)$

sum of count array (size k) and the output array (size n).

for all cases $\rightarrow O(n+k)$

6

Subject:.....

Date:.....

Radix Sort :

Radix sort is a non-comparative, integer based sorting algorithm, that sorts numbers digit by digit, starting from least significant digit or (LSD) most significant digit (MSD).

- modified version of counting sort.
- It's also stable sorting algorithm.

Pseudocode:

RADIX-SORT (A, n)

1. $m \leftarrow \text{Get-Max}(A, n)$
2. $\text{exp} \leftarrow 1$
3. while $m/\text{exp} > 0$
4. COUNT-SORT-BY-DIGIT (A, n, exp)
5. $\text{exp} \leftarrow \text{exp} * 10$
6. $m \leftarrow \text{Get-Max}(A, n)$
7. $\text{max} \leftarrow A[0]$
8. for $i = 0$ to $n-1$
9. if $A[i] > \text{max}$
 $\text{max} \leftarrow A[i]$

Subject:.....

Date:.....

11. return max

12. COUNT-SORT-BY-DEGIT (A, n, exp)

13. output [0 .. n-1]

14. count [0 .. 9] $\leftarrow$ 0

15. for $i = 0$ to $n-1$

16. index = $(A[i] / \text{exp}) \% 10$

17. count [index] = count [index] + 1

18. for $i = 1$ to 9

19. count[i] = count[i] + count[i-1]

20. for $i = n-1$ down to 0

21. index = $(A[i] / \text{exp}) \% 10$

22. output [count [index] .. count [index+1] - 1] = ~~arr~~ A[i]

23. for $i = 0$ to $n-1$

24. A[i] = output[i]

Subject:.....

Date:.....

W simulation:

Let an array, which is unsorted,

$$A = \{ 07, 57, 609, 208, 724 \}$$

Now we'll use radix sorting technique to sort them.

Now,

0	1	2	3	4	5
07	57	609	208	724	125

here, $n = 6$ and base = 10

Step-01: sort by unit digit (exp=1)

→ Units digits of each element

Elements

unit

07	→ 7
57	→ 7
609	→ 9
208	→ 8
724	→ 4
125	→ 5

step-02:

let a count array which size to 10,
and push into frequency cumulative count.

0	1	2	3	4	5
007	057	600	208	734	125
7	7	9	8	4	5

count[10] = {0}
in count array all
elements are initially
set to zero.

→ count frequency

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

→ initially all zero

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

2

→ Final frequency count array

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

code:

```
for (i=0; i<n; i++)
{
    count[(A[i]/10)%10]++;
}
```

Subject:.....

Date:.....

Step-03: store count of cumulative counts.
convert count so that, $\text{count}[\text{value}] = \text{number of elements} \leq v$.

code:

```
for (int i = 0; i < 10; i++)  
    count[i] += count[i-1];
```

count =

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

Step-04: Find output array using traverse from right to left for stability.

Now finding output array same as using

counting sort.

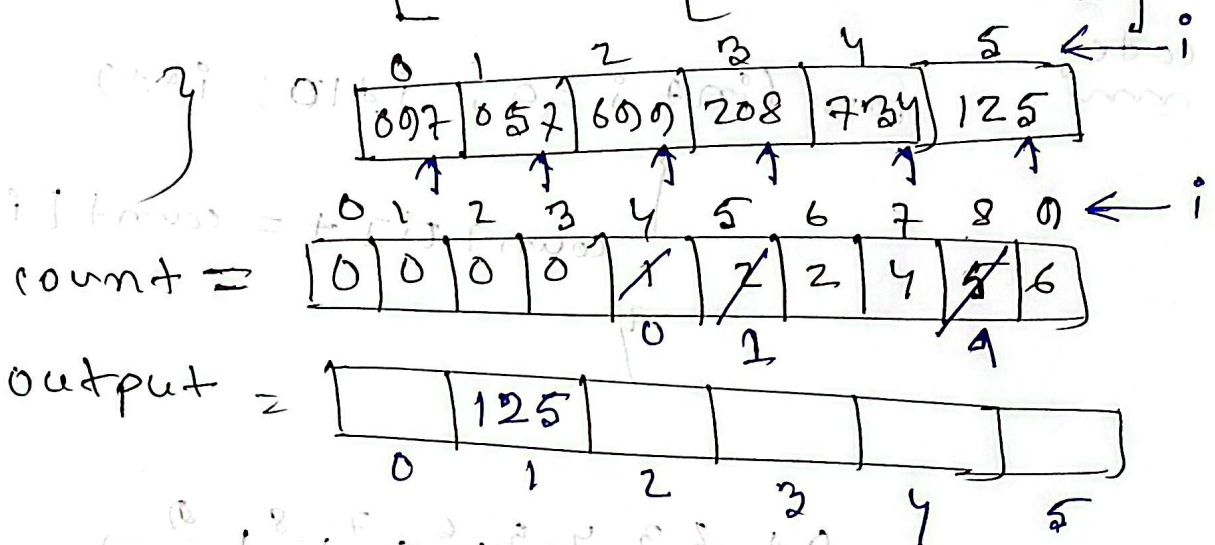
Subject:.....

Date:.....

code:

```
for (i = 5-1; i >= 0; i--)
```

output [-- count [(A[i] / exp) % 10]] = A[i];



~~i = 9, A[9] = 6~~

count[4] = output[4]

meanwhile, index = 125

For, 734 → 4 then decrement frequency of 4 from count array then add which data into output corresponding index

Date :.....

output =

0	1	2	3	4	5
734	125	097	057	208	690

step - 05:

Sort by unit digit (exp = 10)

count =

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

(cumulative sum) count =

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

4

Now, right to left shift

Now, $i = 5, val = 600, d = 0 \rightarrow --count[0] = 5 \rightarrow output[5] = 600$
 $i = 6, val = 208, d = 0 \rightarrow count[0] = 0 \rightarrow output[0] = 208$

$i = 2, \text{val} = 600, d = 0 \rightarrow \text{count}[0] = 0 \rightarrow \text{output}[0] = 208$
 $i = 4, \text{val} = 208, d = 0 \rightarrow \text{count}[5] = 3 \rightarrow \text{output}[3] = 057$

$i = 2, val = 057, d = 5 \rightarrow \text{count}[5] = 3 \rightarrow \text{output}[3] = 057$

$i = 2, val = 07, d = 0 \rightarrow \text{count}[0] = 4 \rightarrow \text{output}[4] = 07$

Subject:.....

Date:.....

$i=1, val=125, d=2 \rightarrow \text{count} + A[2] = 1$

output $A[2] = 125$

$i=0, val=734, d=3 \rightarrow \text{count} + A[3] = 2$

output $A[3] = 734$

output :

0	1	2	3	4	5
208	125	734	57	07	600

copy back to main array

A =

0	1	2	3	4	5
208	125	734	57	07	600

STEP-06: (exp = 100)

A =

0	1	2	3	4	5
208	125	734	057	007	600

 $\leftarrow i$

count =

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

(sum) count =

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

 $\begin{matrix} 10 & 20 & 40 & 40 & 40 & 50 & 60 & 60 & 60 \end{matrix}$

Subject:.....

Date:.....

Now right to left shift.

$i = 5, val = 600, d = 6 \rightarrow --count[6] = 4$

$output[4] = 600$

$i = 4, val = 007, d = 0 \rightarrow --count[0] = 1$

$output[1] = 07$

$i = 3, val = 057, d = 0 \rightarrow --count[0] = 0$

$output[0] = 57$

$i = 2, val = 724, d = 7 \rightarrow --count[7] = 5$

$output[5] = 724$

$i = 1, val = 125, d = 1 \rightarrow --count[1] = 2$

$output[2] = 125$

$i = 0, val = 208, d = 2 \rightarrow --count[2] = 2$

$output[3] = 208$

$output =$

0	1	2	3	4	5
57	07	125	208	600	724

Subject:.....

Date:.....

copy back to the main Array

$A =$

57	07	125	208	600	734
----	----	-----	-----	-----	-----

012345

Final sorted array

57	07	125	208	600	734
----	----	-----	-----	-----	-----

012345

Complexity analysis :

Let n = number of elements

k = number of decimal digit in the largest number ($\log_{10}(\text{max value})$)

b = base (10 here, treated as constant)

needed time,

→ Get-Max $\rightarrow O(n)$

→ count-sort-by-digit $\rightarrow O((n+b) \times d)$

→ per digit (k times) $\rightarrow O(k \cdot (n+b))$
 $= O(k \cdot n)$

Subject:.....

Date:.....

$\therefore$ time complexity = $O(k \cdot n)$

Space:

$\rightarrow \text{count}[b] \rightarrow O(b) \approx O(1)$

$\rightarrow \text{output}[n] \rightarrow O(n)$

$\therefore O(n+b) \approx O(n)$

Only needed counting extra array.

$\therefore$ space complexity: $O(n)$

(7)

Bucket Sort

Bucket sort is a distribution-based sorting algorithm that :-

→ splits the input range into a fixed number of buckets.

→ places each element into its appropriate bucket according to its value.

→ sorts each bucket individually (often with insertion sort)

→ concatenates all buckets form the final sorted list.

It works best when the data is uniformly distributed within a known, bounded range.

Subject:

Date:

W Pseudocode:

BUCKET-SORT ($A[1..n]$)

1. let $B[0..n-1]$ be a new bucket

2. $n = A.length$

3. for $i = 0$ to $n-1$

4. make $B[i]$ an empty list

5. for $i = 1$ to n

6. insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

7. for $i = 0$ to $n-1$

8. sort list $B[i]$ with insertion sort

9. concatenate the lists $B[0], B[1], \dots$

$B[n-1]$ together in order.

Let an unsorted array A , where we would apply bucket sort for the sorted array.

→ create n empty buckets and follow

- iii) concatenate all sorted buckets.

step-01: Create an array of size 10

where each slot represent $\frac{1}{5}$ a bucket.

	0	1	2	3	4	5	6	7	8	9
A =	0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68

[illegible]

Subject:.....

Date:.....

Step-02: Inserting array elements into respective buckets.

→ index for each value = $\text{floor}(\text{value} \times 10)$

value	bucket index
0.78	B[7]
0.17	B[1]
0.30	B[3]
0.26	B[2]
0.72	B[7]
0.94	B[9]
0.21	B[2]
0.12	B[1]
0.23	B[2]
0.68	B[6]

∴ bucket array

B

is

now

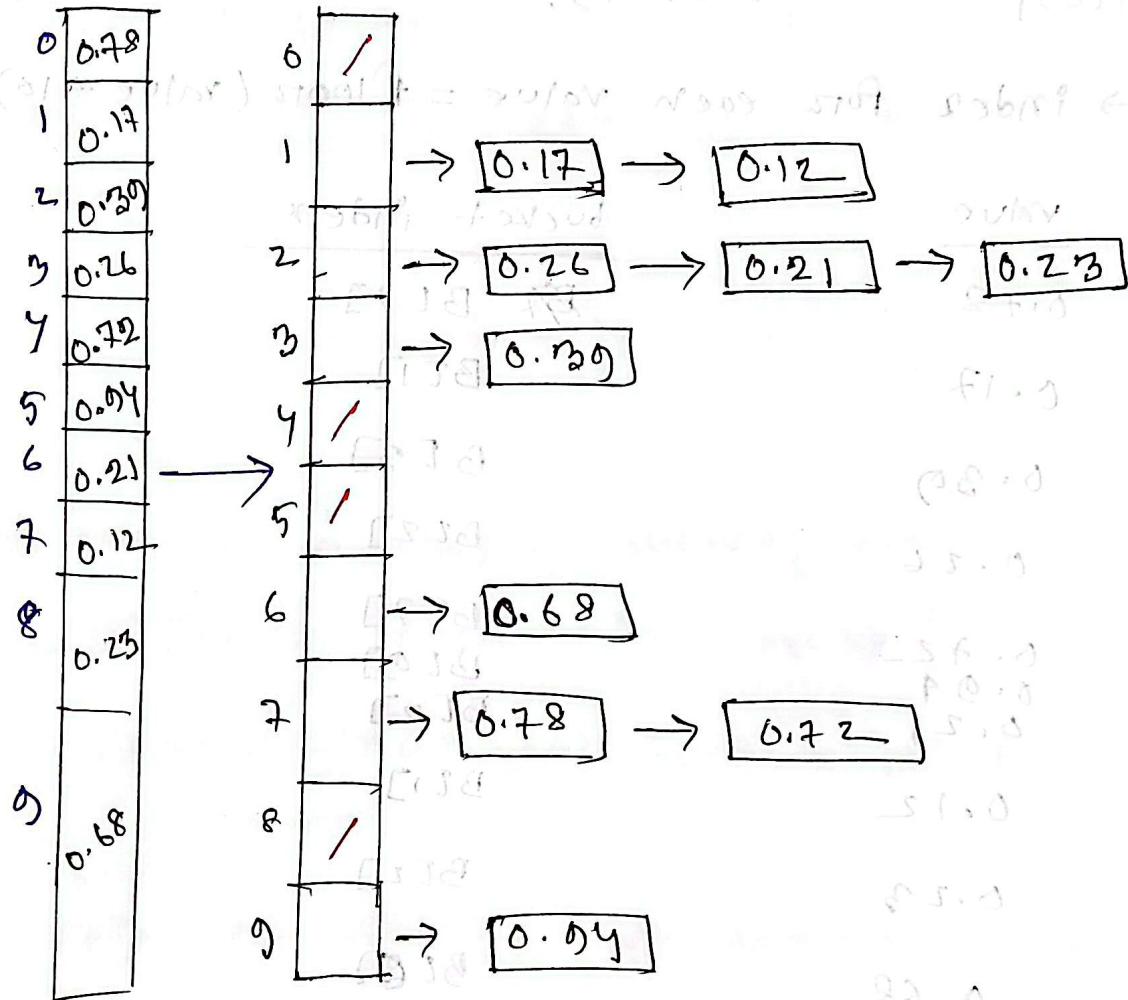
B[7] =



Subject:.....

Date:.....

Bucket array now, : 50-90%



AT → input array
BT → Bucket array

Note: For multiple elements on same bucket, linked list data structure will be used.

Subject:.....

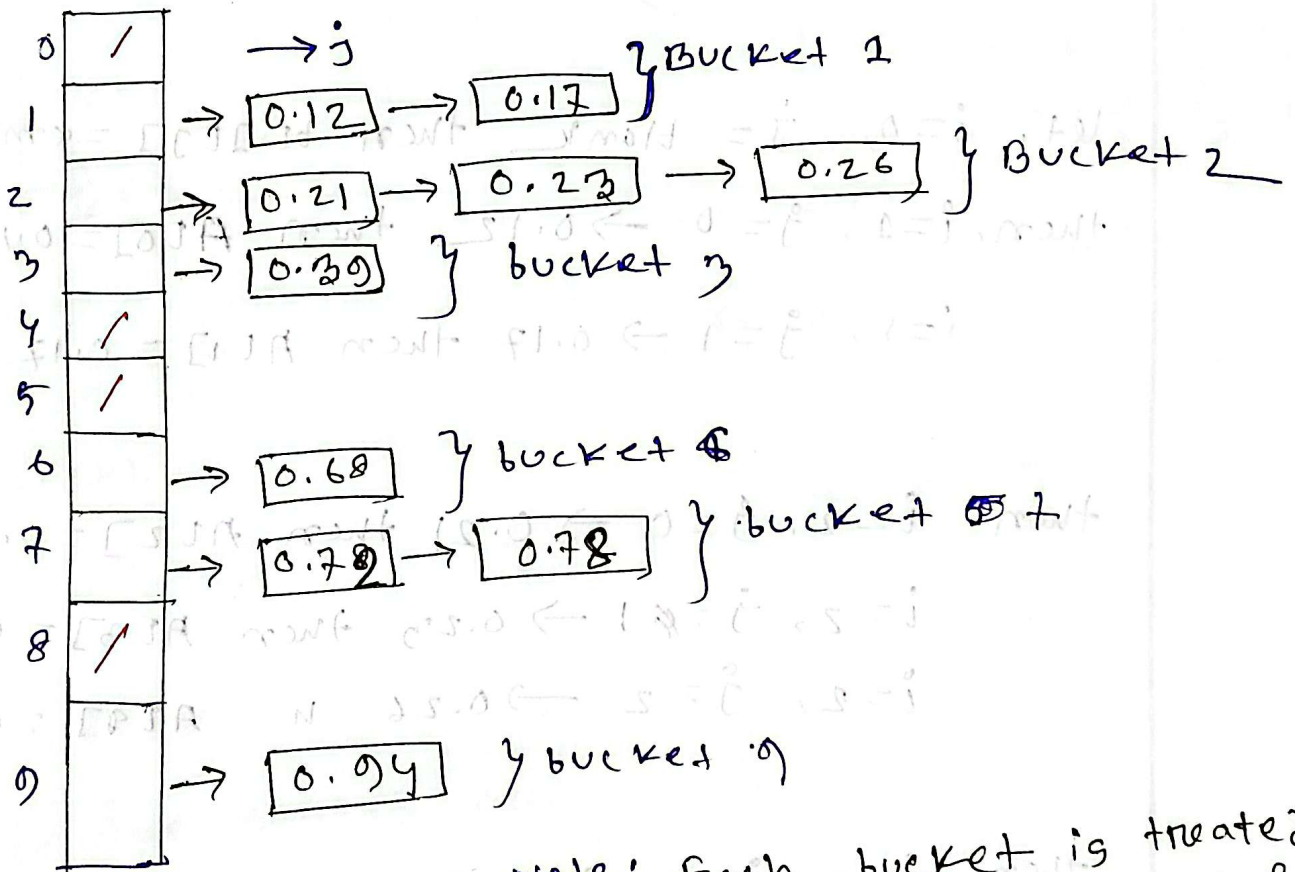
Date:.....

Step-03: Now sort the elements within each bucket using any stable sorting algorithm.

→ we are using here insertion sort

→ The elements within each bucket are

i ↓ now sorted.



Note: Each bucket is treated is a different linked list & sorted in ascending order

Bucket array

Step-04:code:

int index = 0

for (i=0; i < n; i++) {

for (j=0; j < b[i].size(); j++) {

A[index++] = b[i][j];

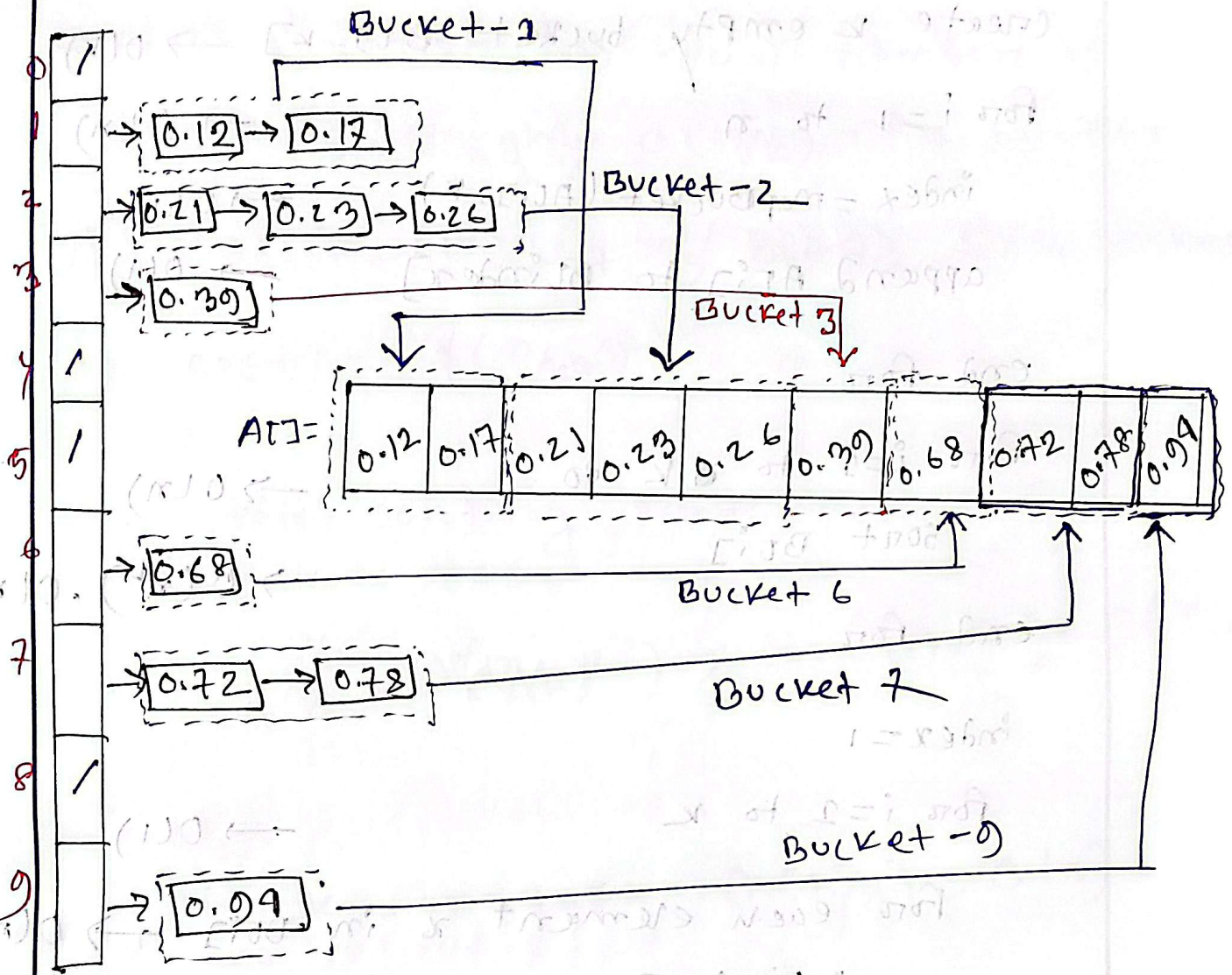
}

let, $i=0$, $j = \text{blank}$ then $b[i][j] = \text{empty}$ then, $i=1$, $j=0 \rightarrow 0.12$ then $A[0] = 0.12$ $i=1$, $j=1 \rightarrow 0.17$ then $A[1] = 0.17$ then $i=2$, $j=0 \rightarrow 0.21$ then $A[2] = 0.21$ $i=2$, $j=1 \rightarrow 0.23$ then $A[3] = 0.23$ $i=2$, $j=2 \rightarrow 0.26$ then $A[4] = 0.26$ then, $i=3$, $j=0 \rightarrow 0.39$ then $A[5] = 0.39$ $i=4$, $j = \text{blank}$ then empty $i=5$, same as $i=4$

similar way for all we get.

Subject:.....

Date:.....



So the sorted array is

0.12	0.17	0.21	0.23	0.26	0.30	0.68	0.72	0.78	0.94
0	1	2	3	4	5	6	7	8	9

Subject:.....

Date:.....

✓ complexity analysis :

create k empty bucket $B[1..k] \rightarrow O(k)$

for $i = 1$ to n $\rightarrow O(n)$

index = mapBucket($A[i], k$) $\rightarrow O(1)$

append $A[i]$ to $B[index]$ $\rightarrow O(1)$

end for

for $i = 1$ to k do $\rightarrow O(n)$

sort $B[i]$ $\rightarrow O(n^2), O(n^2/k)$

end for

index = 1

for $i = 1$ to k $\rightarrow O(1)$

for each element x in $B[i]$ $\rightarrow O(n)$

$A[index] = x$

$\rightarrow O(1)$

index = index + 1

end for $\rightarrow O(1)$

end for

return A $\rightarrow O(1)$

Subject:

Date:

Best case:

→ Each Bucket has equal number of elements, roughly $O(n/k)$ per bucket.

→ If insertion sort is used: each bucket cost = $O((n/k)^2)$

∴ total sorting ~~sort~~ cost:

$$\sum_{i=1}^k O((n/k)^2) = k \cdot O((n/k)^2) = O(n^2/k)$$

if $k = n$, then $O(n^2/n) = O(n)$

∴ ~~best~~ best case = $O(n)$

Total time (best):

Distribute = $O(n)$, sort bucket = $O(n)$

concatenate: $O(n)$

$$\therefore T_{\text{best}} = O(n+k)$$

Subject:.....

Date:.....

Average case :

→ Random distribution of elements into buckets

→ expected bucket size $= n/k$

→ cost per bucket $= (n/k)^2$

∴ total cost is :-

$$\sum_{i=1}^k (n/k)^2 = n^2/k$$

→ adding distribution and reconnection

$$O(n + k + n^2/k)$$

∴ Average time : $O(n + k + n^2/k)$

Subject:.....

Date:.....

Worst case :

→ All element go into one bucket

→ insertion sort = $O(n^2)$

→ Distribution $O(n)$, concatenation $O(n)$
negligible compared to $O(n^2)$

total time :

$$T_{\text{worst}} = \cancel{O(n \log n)} + O(n^2)$$

note: If merge or heap sort is used

per bucket :

$$T_{\text{worst}} = O(n \log n)$$

Subject:.....

Date:.....

✓ Space complexity:

→ Bucket headers → $\Theta(k)$

→ storage for elements in bucket

→ $\Theta(n)$

→ sorting within bucket → $O(k), O(n)$

Total auxiliary space = $\Theta(n+k)$.

Subject:.....

Date:.....

Some Problem solve

we prove that counting sort is stable :-

Solⁿ:

A sorting algorithm is stable if equal elements appear in the output in the same relative order as they appear in the input.

Formally, $A[i] \geq A[j]$ and $i < j$ in the input, then the sorted output, the element from $A[i]$ appears before the elements from $A[j]$.

ex: $a = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 2 & 1 & 2 & 0 & 3 & 1 \\ \hline \end{array}$ $\leftarrow i$

When we should apply counting sort here, then $a[5]$ must be ^{place} before $a[1]$ then it's called stable.

Subject:.....

Date:.....

Another example 2:

Let $A = \begin{bmatrix} 0 & 1 & 2 \\ 4a & 2 & 4b \end{bmatrix}$, where $4a, 4b$ are keys which are distinguishable elements.

maximum key $k=4 \rightarrow c[0 \dots 4]$

now, $c[4]$ initially $= 2 \rightarrow$ last position

means: —

$c \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow$ count array

$c \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 0 & 2 \end{bmatrix}$

$c \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 1 & 3 \end{bmatrix}$ ← i

output: $\begin{bmatrix} 2 & 4a & 4b \end{bmatrix}$

Note that $4a$ appears before $4b$ just as in input.

$\therefore$ The relative order of equal key is preserved $\rightarrow$ stable.

Subject:.....

Date:.....

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ -notation.

Solⁿ:

let, $f(n) = n^3/1000 - 100n^2 - 100n + 3$

As $n \rightarrow \infty$, the highest-degree term dominates the growth.

(1) $n^3/1000 \approx n^3$

where, lower level terms $-100n^2, -100n, 3$ are negligible compared to $n^3/1000$ for large n .

For Big-Theta:

if $f(n) = O(g(n))$ if there exist constants $c_1, c_2 > 0$

and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > n_0$$

Subject:.....

Date:.....

That why choose $\Theta(n^3)$

$$\therefore f(n) = \Theta(n^3)$$

✓ Rewrite Insertion-sort for nonincreasing sorted order.

Pseudocode:

INSERTION-SORT-DESCENDING(A)

1. for $j = 2$ to $A.length$

2. $key = A[j]$

3. $i = j - 1$

4. while $i > 0$ & $A[i] < key$

5. $A[i+1] = A[i]$

6. $i = i - 1$

7. $A[i+1] = key$

Subject:

Date:

The Big O Notation :-

→ way to describe the performance of an algorithm w.r.t to the input size.

→ In linear search, the number of comparison is directly proportional to the input size.

Linear search :

Numbers of items	Numbers of comparisons
100	100
1000	1000
10 million	10 million
n	n

Running time : $O(n)$

↑ No. of operation

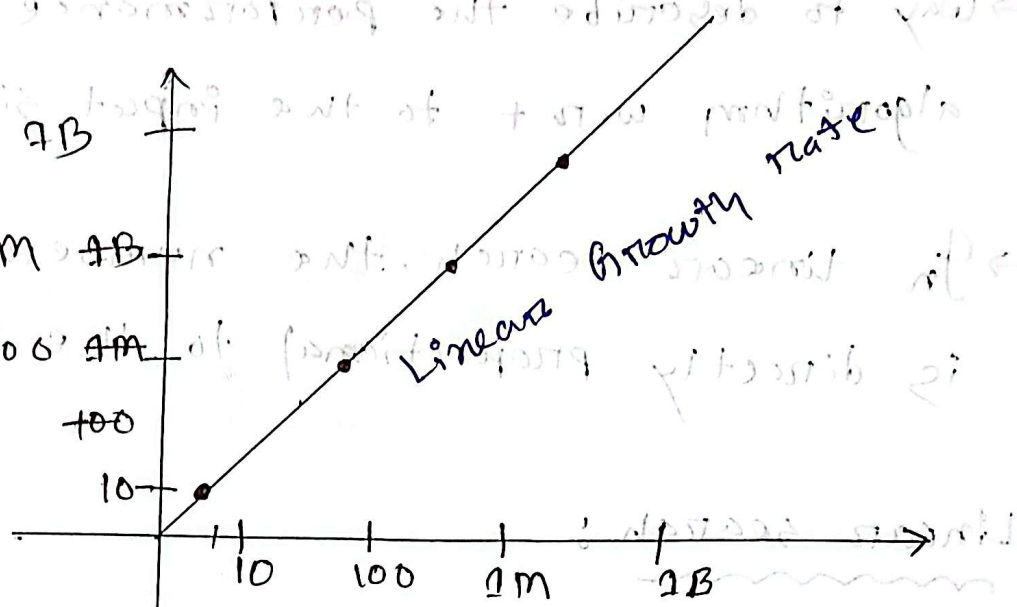
→ Big O notation doesn't tell speed in seconds

→ It helps in comparison the number of operation

Subject:.....

Date:.....

Growth:



Running time: $O(n)$

Binary search: In binary search, every time half of the list of the n is eliminated.

Binary search:

items

comparisons

10

4

100

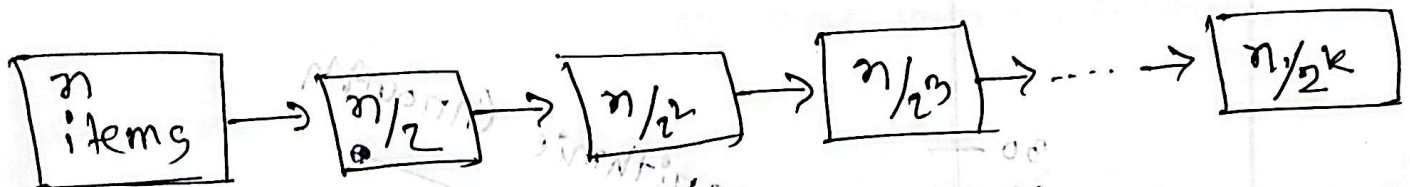
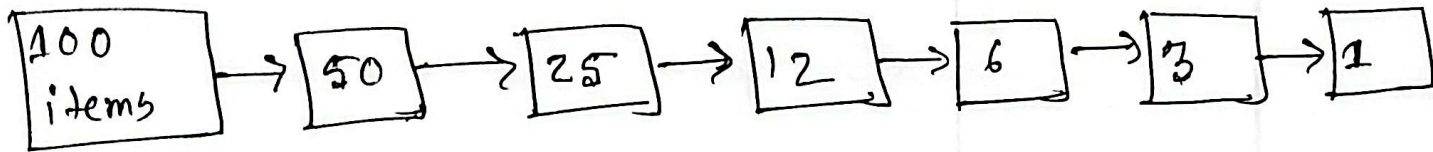
7

1 billion

30

Subject:.....

Date:.....



here,

$$\frac{n}{2^k} = 1$$

$$\Rightarrow n = 2^k$$

$$\Rightarrow \log_2 2^k = \log_2 n$$

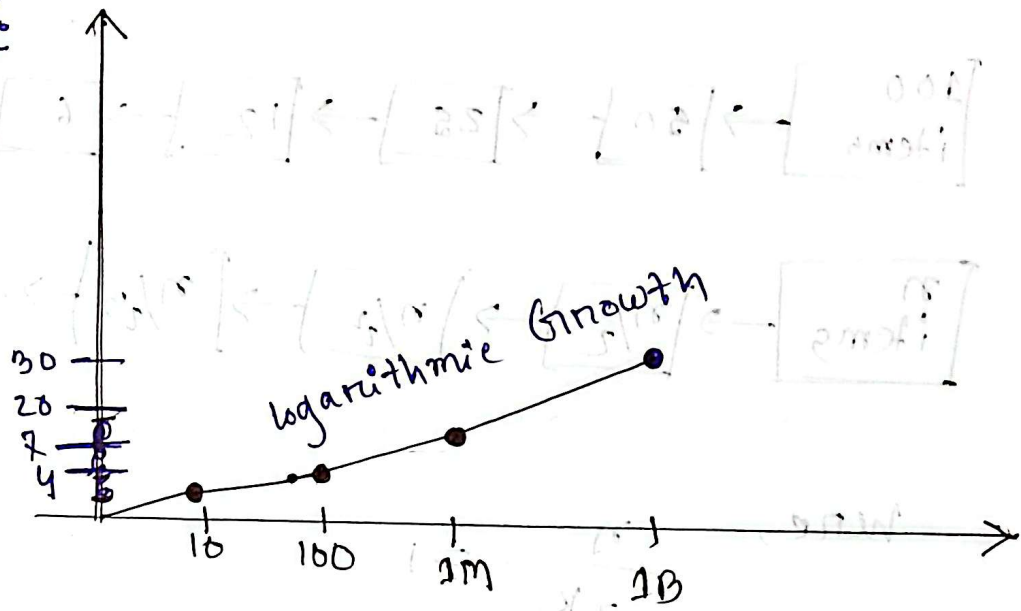
$$k = \log_2 n$$

Running time : $O(\log n)$

Subject:.....

Date:.....

Growth:



Running time: $O(\log n)$

no. of items	no. of comparisons
10	4
100	7
1m	20
1B	30

Subject:.....

Date:.....

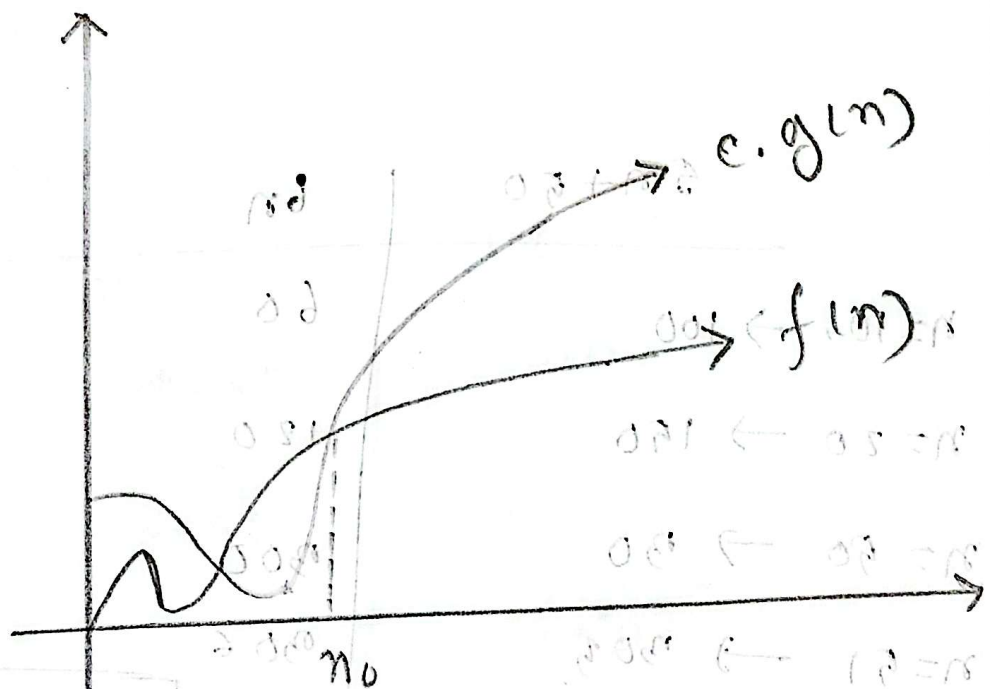
Comparison Growth Rate comparison of Functions.

Assuming $f(n)$ and $g(n)$ are non-negative functions.

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values of

n where $n \geq n_0$ and c and n_0 are constants.

where $g(n)$ is the tight upper bound of $f(n)$.



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

Problem - 01 :-

Assume that $f(n) = 5n + 50$ & $g(n) = n$.

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

Soln:- According to the definition of Big O

notations:

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values

of n where $n > n_0$ and c and n_0 are constants.

let assume $c = 6$

$5n + 50$	$6n$
$n = 10 \rightarrow 100$	60
$n = 20 \rightarrow 150$	120
$n = 50 \rightarrow 300$	300
$n = 51 \rightarrow 305$	306

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

$$5n + 50 = O(n)$$

for $c = 6$ and $n > 50$

Ans: = (11)

Subject:.....

Date:.....

Problem-02 :- Assume that, $f(n) = 5n + 50$ and $g(n) = \log_{10} n$. Is $g(n)$ upper bound of $f(n)$?

Soln:

We would need positive constants c and n_0 such that,
 $5n + 50 \leq c \log_{10}(n)$ for all $n > n_0$.

Let, $f(n)$ is linear and $g(n)$ is logarithmic which grows very slowly.

Assume, $c = 1000$

$5n + 50$	$1000 \log_{10} n$
$n = 10$ 100	1000
$n = 10^2$ 550	2000
$n = 10^3$ 5050	3000
$n = 10^4$ 50050	4000

$\therefore g(n)$ is not upper bound of $f(n)$.

Ans)

Subject:

Date:

Problem-03: Find the upper bound for
 $f(n) = 3n + 8$

Solⁿ:

Step-01: Identify the dominant term.

	$3n$	8
$n=10$	30	8
$n=10^2$	300	8
$n=10^3$	3000	8

So, $3n$ is the dominant term.

Step-02: choose $g(n)$ according to the dominant term

like, $n, n \log n, n^2, n^3, \dots$
[$g(n)$]

Subject:.....

Date:.....

Step-03: Apply the Big O notation.

to find the growth rate of the function

According to Big O Notation we assume

~~c=1~~

c = 4

$3n + 8 \leq 4n$

		$f(n)$	$g(n)$
		$3n + 8$	$4n$
01	$n = 7$	29	28
01	$n = 8$	32	32
01	$n = 10$	38	40
01	$n = 10^2$	308	400

here, clearly we can see that

$c \cdot g(n) \geq f(n)$

therefore, $3n + 8 \leq 4n \rightarrow \text{true}$

therefore $3n + 8 = O(n)$ for $c = 4, n_0 = 8$

$\therefore$ upper bound = $O(n)$

Subject:.....

Date:.....

Problem - 04:

Find the upper bound of $f(n) = n^2 + 10$

Solⁿ:

dominant term,

	n^2	10
$n=10$	100	10
$n=100$	10000	10
$n=1000$	1000000	10

choose $g(n)$ according to dominant term

$f(n) \leq n^2, n^2, 2^n, \dots$

$g(n)$

According to the O notation,

Assume $c = 2$ then

Subject:.....

Date:.....

	$n^2 + 10$	$2n^2$
$n=1$	11	2
$n=2$	14	8
$n=3$	19	18
$n_0 \rightarrow n=4$	26	32

Now, $n^2 + 10 \leq 2n^2$ is true

$\therefore n^2 + 10 = O(n^2)$ where $c=2, n_0=4$

Problem-05: Find out the upper bound

for $f(n) = n^4 + 100n^2 + 35$.

Soln:

Dominant term: n^4 ,

Assuming $g(n) = n^4$

According to the definition of Big O notation

$f(n) = O(g(n))$ iff $f(n) \leq cg(n)$

for all values of n where n_1, n_0 and c and n_0 are constant.

Subject:.....

Date:.....

Assuming $c = 2$

Now, $n^4 + 100n^2 + 35 \leq 2n^4$?

	$n^4 + 100n^2 + 35$	$2n^4$
$n=1$	136	2
$n=10$	20035	20000
$\rightarrow n=15 (n_0)$	73160	101250

$\therefore n^4 + 100n^2 + 35 \leq 2n^4$ is true.

$n^4 + 100n^2 + 35 \in O(n^4)$ where

$c = 2$ and $n_0 = 15$

Subject:

Date:

Problem: 06:-

Find out upper bound for $f(n) = 2^n + 3n^3$

Soln:

Dominant term: 2^n

Assuming $g(n) = 2^n$

According to Big O notation, Assuming $c=2$

$$\text{Now, } 2^n + 3n^3 \leq 2^{n+1} \quad ?$$

here	$2^n + 3n^3$	2^{n+1}
		2048
$n=10$	4024	
		4096
$n=11$	6041	
		8192
$n=14 (n_0)$	24616	
		32768

So, $2^n + 3n^3 \leq 2^{n+1}$ is true.

$\therefore 2^n + 3n^3 = O(2^n)$ for $c=2$ and $n_0=14$.

Subject:.....

Date:.....

Problem - 07:

Find the upper bound if $f(n) = 300$.

Soln:

Dominant : 300 Assuming $g(n) = 1$

According to Big O notation

Assume $c = 300$

then, $300 \leq 300$ is true.

$\therefore 300 = O(1)$ for $c = 300$ and $n_0 = 1$

Subject:

Date:

Common Big-O runtimes:

- (i) $O(\log n)$, also known as log time, Ex: Binary search
- (ii) $O(n)$, also known as linear time, Ex: Linear search
- (iii) $O(n \log n)$, also known as log time, Ex: quicksort
- (iv) $O(n^2)$, also known as polynomial time, Ex: selection sort
- (v) $O(n!)$, also known as exponential time, Ex: traveling salesman

Graph:

n	$\log n$	n	$n \log n$	n^2	$n!$
10	1 ms	10 ms	10 ms	100 ms	3628800 ms
20	2.3 ms	20 ms	26 ms	400 ms	2.43×10^{18} ms
100	2 ms	100 ms	200 ms	10000 ms	0.33×10^{157} ms
					↓ problem
					6 mins
					7.7×10^7 years
					2.95×10^{147} years

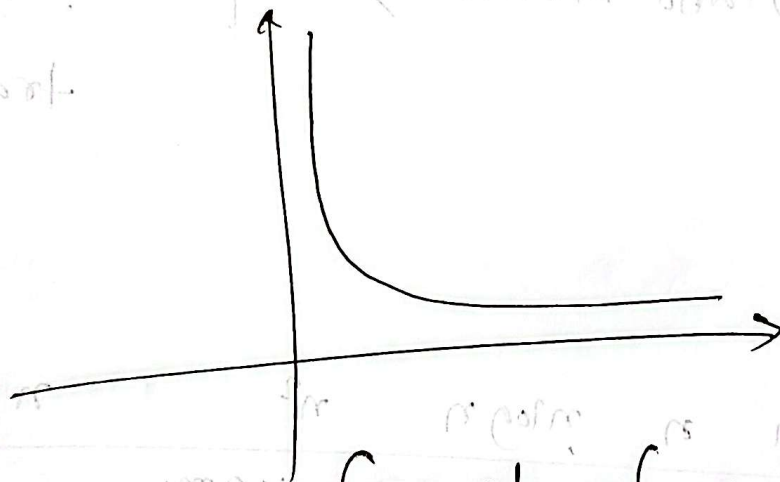
Subject:.....

Date:.....

Decrement Function:

A function in which the denominator is bigger than the numerator is called decrement function.

Ex: $\frac{1}{n}, \frac{n}{n^2}, \frac{n}{2n}, \frac{n^2}{3n}$



Graph of a decrement function

Problem: Arrange the following function in the decreasing order slowest to fastest.

$$\frac{100}{n}, \frac{n}{2n}, \frac{n}{n^2}, \frac{n^2}{3n}, \frac{n^3}{3n}$$

Date :.....

we get, after simply simplify the fraction,

$$\frac{100}{n}, \frac{100}{2n}, \frac{100}{n}, \frac{n^2}{3n}, \frac{n^3}{3n}$$

Here, $n < 2^n < 3^n$ then come,

$$\frac{100}{n}, \frac{1}{n}, \frac{n}{2^n}, \frac{n^2}{3^n}, \frac{n^3}{4^n}$$

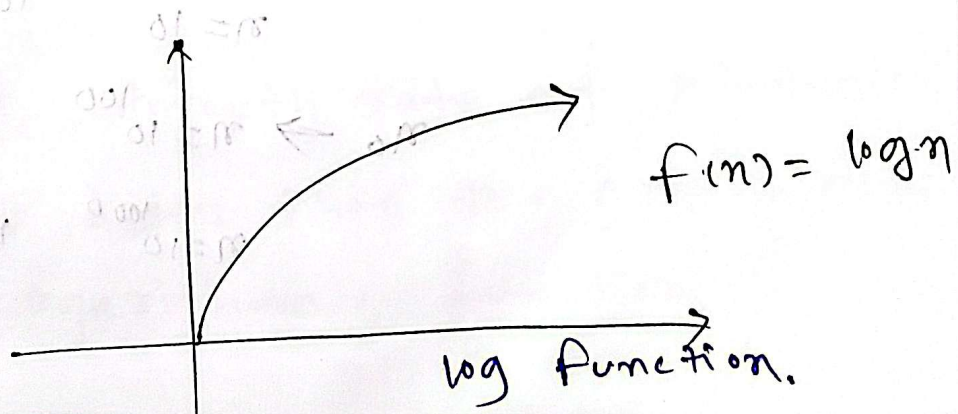
slowest

Fasteq

↳ Logarithmic Function:

A function that grows positively, but with slower growth rate.

example: $\log n$, $(\log n)^{10}$, $\log \log n$, $\log \log \log n$, etc



Subject:.....

Date:.....

Note: logarithmic function is asymptotically bigger than the constant function.

Problem: $f(n) = 100$, $g(n) = \log_{10} n$. is $f(n) = O(g(n))$.

Solⁿ:

According to definition of Big-O notation:-

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for all values of n where $n > n_0$ and c and n_0 are constants.

Assume $c = 1$

$$100 \leq \log_{10} n ?$$

	100	$\log_{10} n$
$n = 10$	100	1
$n_0 \rightarrow n = 10^{100}$	100	100
$n = 10^{1000}$	100	1000

Subject:.....

Date:.....

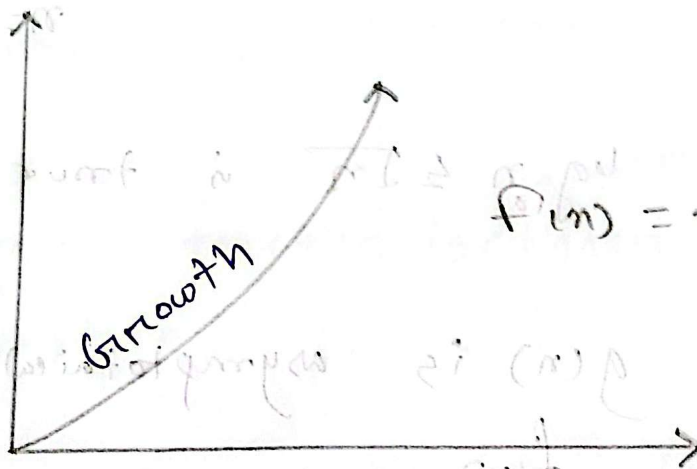
$\therefore 100 \leq \log_{10} n$ is true

therefore $g(n)$ is asymptotically bigger than $f(n)$

Polynomial Function:

A function whose growth rate increases polynomially.

Example: $n^{0.1}$, $\sqrt{n}$, n^2 , $n \log n$, n^k etc.



Graph of the polynomial function.

Note: The growth rate of polynomial function is bigger than the growth rate of logarithmic function.

Subject:.....

Date:.....

Problem:

$$f(n) = \log_{10} n \text{ and } g(n) = \sqrt{n}$$

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

Solⁿ:

let assume, $\epsilon = 1$

$$\log_{10} n \leq \sqrt{n} \quad ?$$

	$\log_{10} n$	$\sqrt{n}$
$n_0 \rightarrow n=10$	1	3.16
$n=100$	2	10
$n=1000$	3	100

$\therefore \log_{10} n \leq \sqrt{n}$ is true.

$\therefore g(n)$ is asymptotically bigger than $f(n)$.

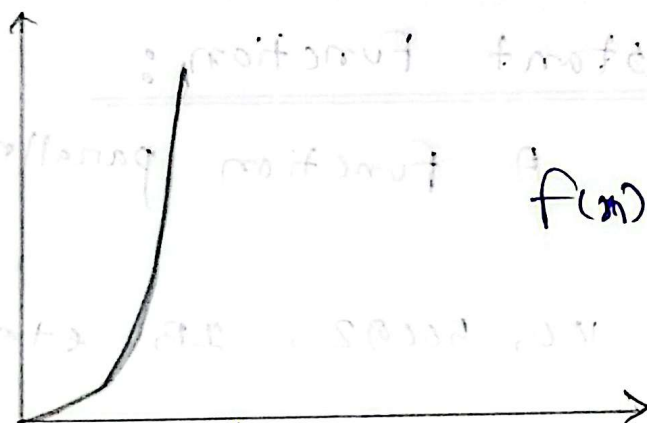
Subject:

Date:

Exponential Function:

A function whose growth rate increases rapidly.

Ex: $2^n, 3^n, n!, n^n, a^n$ etc.



Graph of the exponential function.

Note: Exponential function is ^{asymptotically} bigger than the polynomial function.

problem: $f(n) = n^2$ and $g(n) = 2^n$, is $f(n) = O(g(n))$?

Soln:

let assume $c=1$

$n^2 \leq 2^n$ is ?

$n_0 \rightarrow n=16$
 $n=20$

n^2	2^n
100	1024
400	1048576

Subject:.....

Date:.....

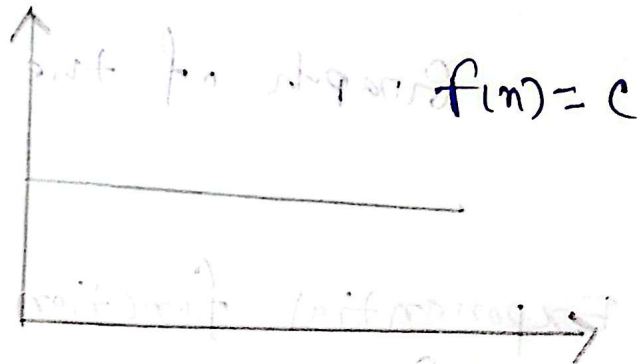
Now, $n^2 \leq 2^n$ is true.

therefore, $g(n)$ is asymptotically bigger than $f(n)$.

✓ Constant Function :

$f(n) = c$ A function parallel to x -axis.

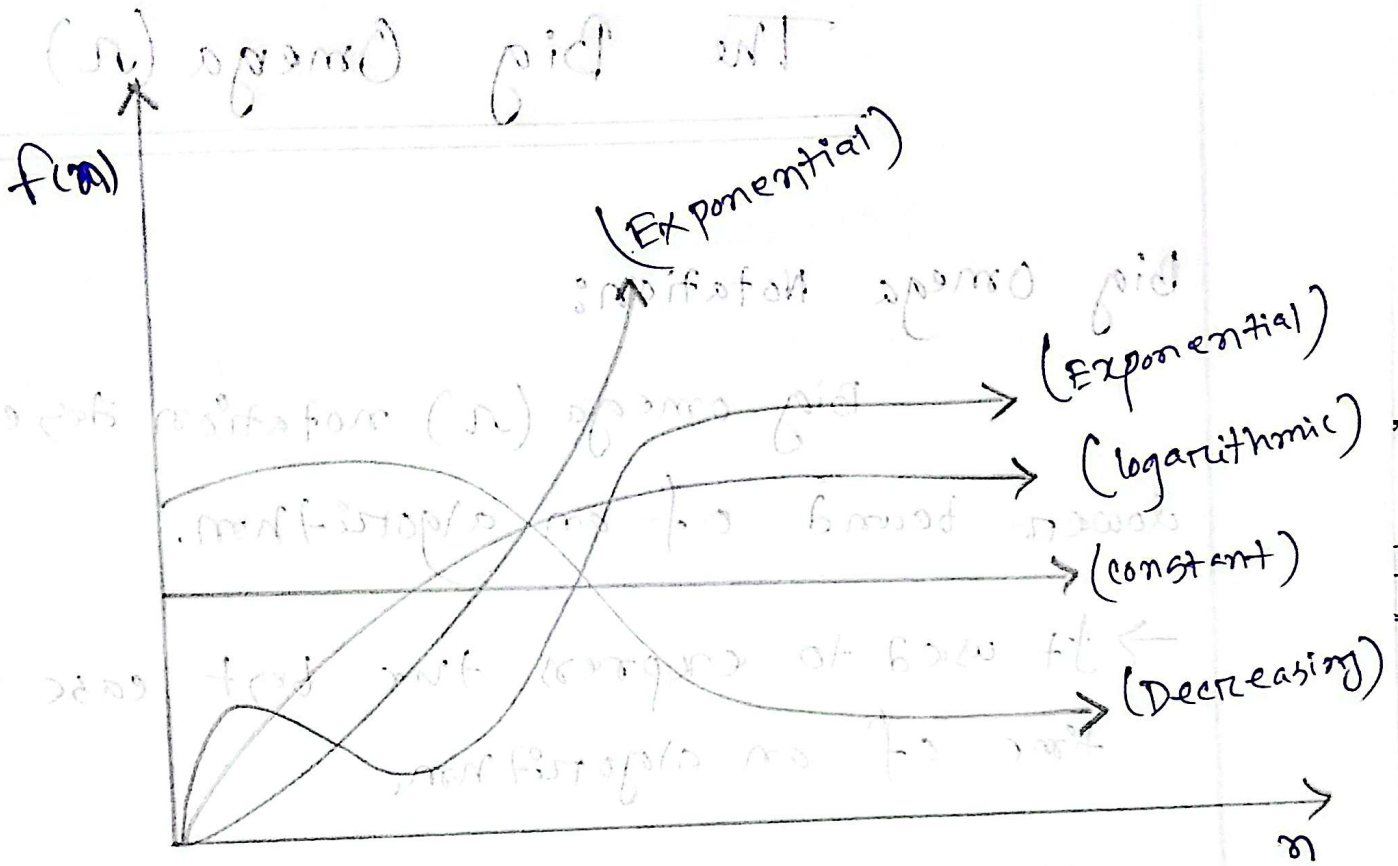
Ex: 100, 50002, 2B etc.



Graph of constant function.

Subject:.....

Date:.....



Decreasing function < constant function < logarithmic function < polynomial function < Exponential function.

The Big Omega (Ω) Notation

Big Omega Notation:

Big omega (Ω) notation describe the lower bound of an algorithm.

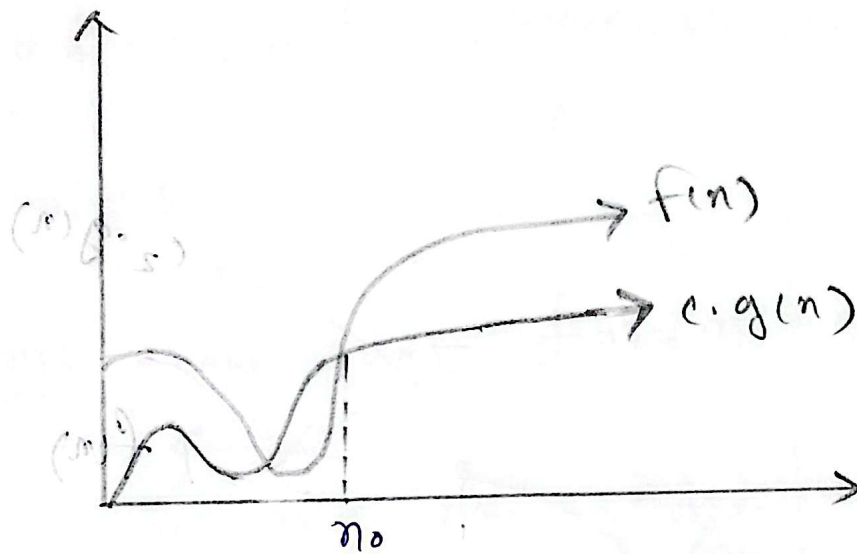
→ It used to express the best case running time of an algorithm.

Definition: Assuming $f(n)$ and $g(n)$ are ~~non-negative~~ non-negative functions. $f(n) = \Omega(g(n))$ iff $f(n) \leq c \cdot g(n)$ for some $c > 0$ and for all values of n where $n > n_0$ and c and n_0 are constants.

here ~~$g(n)$~~ $g(n)$ is the asymptotic tight lower bound of $f(n)$.

Subject:.....

Date:.....



For Big-Omega (Ω)

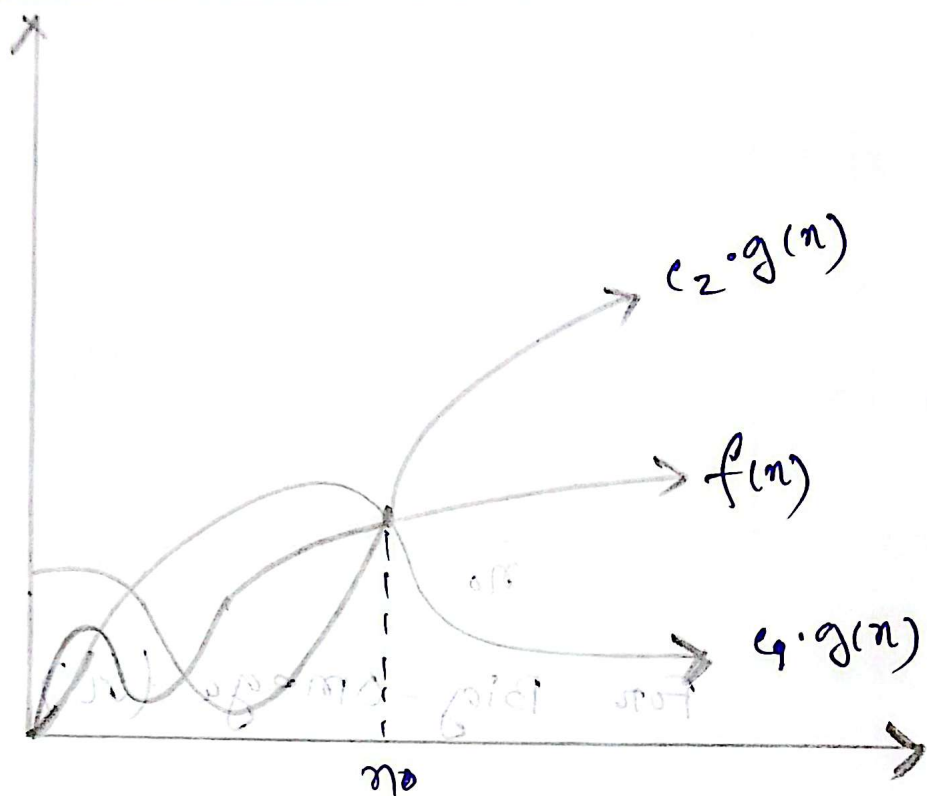
The Big Theta Notation

Definition: Assuming $f(n)$ and $g(n)$ are non-negative functions. $f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$

for all values of $n > n_0$ and c_1, c_2 are constants.

Subject:.....

Date:.....



For Big Theta (Θ)

Problem:- Find the lower bound

$$f(n) = 10n^2 + 5n$$

Soln:-

Dominant term: $10n^2$

$$\text{Assume } g(n) = n^2$$

According to the omega notation,

$$\text{Assume } c = 10$$

Subject:.....

Date:.....

$10n^2 + 5 > 10n^2$ is true for all $n > 1$

$\therefore 10n^2 + 5 = \Omega(n^2)$ for $c = 10$, & $n_0 = 1$

problem: show that $f(n) = n^3 + 3n^2 = \Theta(n^3)$

soln:

From Big theta notation:

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

Assuming $c_2 = 2$

$n^3 + 3n^2 \leq 2n^3$ is true for $n \geq 3$

✓ Priori analysis:

Estimation of time & memory space required by an algorithm before executing it on the system.

✓ posteriori analysis:

Calculation of time & ^{memory} space required by the algorithm after executing it on the system.

✓ cpu computations: refers to a task performed by the cpu or instruction execution by the cpu

✓ Main memory space: used to temporarily store data and instruction that cpu needs for quick access during program execution.

Subject:.....

Date:.....

Time complexity:

Estimation of total CPU computation required to execute an algorithm.

Time complexity = Total no. of CPU computations of an Algorithm

= sum of frequency count of each instruction of an algorithm.

↓
- No. of times an instruction is executed.

ex:

Algorithm { sum (a, n) → 1
 { sum = 0 → 1 unit
 for (i = 1; i ≤ n; i++) { → n
 sum = sum + a[i] → 2n
 } return sum → 1 unit

$$\therefore \underline{TC}: 1 + 1 + n + 2n + 1 = 3n + 3 = O(n)$$

Space complexity:

Estimation of main memory space required to execute an algorithm.

$$\text{Space complexity} = \begin{cases} \text{(i) Space required to store source code} \\ + \\ \text{(ii) Space required to simple variables} \\ + \\ \text{(iii) need for data structure used} \\ + \text{ use extra array} \\ \text{(iv) stack for recursive algorithm} \end{cases}$$

we are not considers (i) and (ii) factor.

we need (iii) & (iv) factor.

Ex:

Algo (A, n) {

sum = 0

for (i = 0; i <= n; i++)

sum += 2 * A[i]

return sum

∴ here space complexity = $O(1)$

Subject:.....

Date:.....

Time complexity analysis:

① `for (i = n; i >= 1; i = i/5)`
`printf("ABC")` $\rightarrow \log_5 n + 1$ times
 $= O(\log n)$

here, ~~i = 0~~ $i = n, \frac{n}{5}, \frac{n}{5^2}, \frac{n}{5^3} \dots \frac{n}{5^k}$

now, $\frac{n}{5^k} = 1$

$$\Rightarrow 5^k = n$$

$$\Rightarrow k = \log_5 n$$

$$TC: O(\log n)$$

② `for (i = 1; i <= n; i++)` $\rightarrow n$
`for (j = 1; j <= n; j++)` $\rightarrow n$

$$TC = O(n^2)$$

Subject:.....

Date:.....

$$\begin{aligned}
 \textcircled{iii} \quad & \text{for } (i=3; i \leq n; i=i^3) \rightarrow \log_3 \log_3 n \text{ times} \\
 & \text{for } (j=n; j \geq 1; j=j/2) \rightarrow \log_2 n \text{ } \downarrow \\
 & \text{for } (k=1; k \leq n; k=k+1) \rightarrow \log_2 n \\
 & \quad x = x + 1 \rightarrow 1
 \end{aligned}$$

$$\therefore T.C = \Theta(\log^4 n, \log \log n)$$

$$\begin{aligned}
 \textcircled{iv} \quad & \text{for } (i=n^2; i \geq 1; i=i/2) \rightarrow 2 \log_2 n^2 \\
 & \text{printf("Hello")} \rightarrow 1
 \end{aligned}$$

$$\text{here, } i=n^2, \frac{n^2}{2}, \frac{n^2}{2^2}, \dots, \frac{n^2}{2^k}$$

$$\text{Now, } \frac{n^2}{2^k} = 1$$

$$\Rightarrow 2^k = n^2$$

$$\Rightarrow k = \log_2 n^2 = 2 \log_2 n$$

$$\therefore k = 2 \log_2 n$$

Subject:.....

Date:.....

here, $2 \log_2 n + 2 = O(\log n)$

Problem: Find complexity

for ($i=n, j=0; i>0; i/=2, j+=i$);

~~Let~~ $val(j) = \Theta(\log n)$. Let $val(j)$ denote the value stored in the variable j after termination of the for loop.

Solⁿ: $val(j)$: value of j after the termination of the for loop.

iteration-1: $i=n, j=0$

iteration-2: $i=n/2, j=n/2$

iteration-3: $i=n/2^2, j=n/2 + n/2^2$

4 - 4 : $i=n/2^3, j=n/2 + n/2^2 + n/2^3$

(10) 3

Subject:.....

Date:.....

$$val(j) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \dots + \frac{n}{2^{k-1}}$$

Now, $\frac{n}{2^{k-1}} = 1$ (base case)

$(i = 1, \dots, k-1) : 2^i = n$ (inductive step)

$$\Rightarrow 2 = n$$

$\Rightarrow (k-1) \log_2 2 = \log_2 n$

$\Rightarrow k = \log_2 n + 1$

$$\therefore val(j) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \dots + \frac{n}{2^{\log_2 n}}$$

$$= n \left(1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^{\log_2 n}} \right)$$

$$= n \left(\frac{1 - \frac{1}{2^{\log_2 n + 1}}}{1 - \frac{1}{2}} \right)$$

$$= 2n - 1$$

$$= \Theta(n)$$

Subject:.....

Date:.....

Problem solve :

consider the problem of adding two n -bit binary integers, stored in two n -elements arrays A and B . The sum of two integers should be stored in binary form in an $(n+1)$ element array C .

State the problem formally and write pseudocode for adding the two integers.

Soln:

we are given two binary integers of length n , stored as arrays.

Input:

Two arrays $A[0 \dots n-1]$ and $B[0 \dots n-1]$ where each element is either 0 or 1.

The LSB is the rightmost index $n-1$

Subject:.....

Date:.....

Output:

- An array $[0 \dots n]$ of size $n+1$ to store the binary sum of A and B
- $[0]$ contains MSB and the rest hold the sum bits.

Idea:

→ add bit from right to left (LSB $\rightarrow$ MSB)

→ keep a carry that starts at 0.

→ For each position i from $n-1$ down to 0:

$$\rightarrow \text{sum} = A[i] + B[i] + \text{carry}$$

$$\rightarrow C[i+1] = \text{sum} \bmod 2$$

$$\rightarrow \text{carry} = \text{sum} \div 2$$

→ after finishing, store final carry into $C[0]$.

Pseudocode:

ADD-BINARY ($A[0 \dots n-1]$, $B[0 \dots n-1]$)

1. let $c[0 \dots n]$ be a new array

2. $carry = 0$

3. for $i = 0$ to $n-1$

4. $total = A[i] + B[i] + carry$

5. $c[i+1] = total \text{ mod } 2$

6. $carry = total \text{ div } 2$

7. end for

8. $c[0] = carry$

9. return c

example:

$A = [1, 0, 1, 1]$ (binary 1011 = 11)

$B = [1, 1, 0, 1]$ (binary 1101 = 13)

addition $c = [1, 1, 0, 0, 0] \rightarrow$ binary 11000 (24)

Subject:.....

Date:.....

Time complexity: ^{and space}

Time complexity $T(n) = \Theta(n)$

Space complexity $T(n) = O(n)$

Because additional memory space needed for carry bit array.

Problem: How can we modify almost any algorithm to have a good best-case running time.

→ We can usually give almost any algorithm an excellent best-case running time by adding a quick check for an already solved input before doing the heavy work.

Subject:.....

Date:.....

Key Idea: Add a pre-test at the start that recognizes when the input is "trivial"

(already ordered)

→ If the input passes this test, return immediately.

→ Otherwise run the normal algorithm.

Because the check can often be done in $O(1)$ or $O(n)$ time - much

cheaper than the worst-case work.

The best case becomes that checks cost.

Subject:.....

Date:.....

example:

Algorithm	Modification	Resulting Best Case
Sorting (merge sort, insertion sort)	Before sorting, scan once to see if the array is already sorted	Best case $O(n)$ or $O(n \log n)$
Searching	First test if the key equals the first / last elements	Best case $O(1)$
Graph algorithm	Quickly check if a graph has 0 or 1 vertex	Best case $O(1)$

General template:

MODIFIED - ALGORITHM (input):

1. if easy - case (input) then
2. return Easy - answers
3. else
4. run original algorithm

Subject:.....

Date:.....

Easy-easy is fire fast check.

Trade off:

- Best case time improves dramatically
- worst case time usually stays the same

So, to give almost any algorithm a good best-case running time.

Problem: Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ -notation, prove that

$$\max(f(n), g(n)) = \Theta(f(n) + g(n))$$

Soln: For function $p(n)$ and $q(n)$
 $p(n) = \Theta(q(n))$ means there exists
 n_0 such that for all $n > n_0$, $f(n) > 0$
 and $g(n) > 0$.

For functions $p(n)$ and $q(n)$, $p(n) = \Theta(q(n))$
 means there exists positive constants
 c_1, c_2 and n_0 such that.

$$c_1 q(n) \leq p(n) \leq c_2 q(n) \text{ for all } n > n_0$$

Here $p(n) = \max(f(n), g(n))$ and

$$q(n) = f(n) + g(n)$$

Subject:.....

Date:.....

Now, For upper bound, any real numbers a, b . $\max \{a, b\} \leq a + b$.

$$\max \{f(n), g(n)\} \leq f(n) + g(n).$$

Now, For lower bound, for any real numbers a, b , $\max \{a, b\} \geq \frac{1}{2}(a+b)$

thus, for all n , no

$$\therefore \max \{f(n), g(n)\} \geq \frac{1}{2}(f(n) + g(n))$$

we have now combine for all case of n

we get,

$$\frac{1}{2}(f(n) + g(n)) \leq \max \{f(n), g(n)\} \leq$$

$$(f(n) + g(n)).$$

$$\therefore \max \{f(n), g(n)\} = \Theta(f(n) + g(n))$$

(shown)

Problem: Show that for any real number constants a and b : where $b > 0$

$$\text{then, } (n+a)^b = \Theta(n^b)$$

Solⁿ:

$f(n) = \Theta(g(n))$ means there exists positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n > n_0$$

$$\Rightarrow f(n) = (n+a)^b$$

$$\Rightarrow g(n) = n^b$$

upper bound:

For large n ,

$$n+a \leq n+|a| \leq 2n$$

Because $b > 0$

$$(n+a)^b \leq (2n)^b = 2^b n^b$$

Subject:.....

Date:.....

lower bound :-

For $n > 2|a|$

$$n+a > n - |a| > \frac{n}{2}$$

$$\therefore (n+a)^b > (n/2)^b = 2^{-b} n^b$$

combine :

For all $n > n_0$ (choose $n_0 > 2|a|$)

$$2^{-b} n^b \leq (n+a)^b \leq 2^b n^b$$

$$\therefore (n+a)^b = \Theta(n^b)$$

(proved)