Sorting Techniques

Criteria for Analysis

1) No. of comparisons
2) No. of Swaps
3) Adaptive
4) Stable
5) Extra Memory

Types of Sorts

- Comp. Based Sort
  - $O(n^2)$
    1) Bubble
    2) Insertion
    3) Selection
  - $O(n \log n)$
    4) Heap
    5) Merge
    6) Quick
    7) Tree
  - $O(n^{3/2})$
    8) Shell
- Index Based Sort
  - $O(n)$
    9) Count
    10) Bucket / Bin
    11) Radix

① Bubble Sort

n=5

	0	1	2	3	4
A	8	5	7	3	2

1st pass

8	5	5	5	5
5	8	7	7	7
7	7	8	2	2
3	3	2	8	3
2	2	3	3	8

Comp 4
Max Swap 4

2nd pass

5	5	5	5
7	7	3	3
2	3	7	2
3	2	2	7
8	8	8	8

Comp 3
Max Swap 3

3rd pass

5]	3	3
3]	5]	2
2	2]	5
7	7	7
8	8	8
No. of Comp	2	
Max Swaps	2	

4th pass

3]	2
2]	3
5	5
7	7
8	8
1	
1	

No. of passes : $(n-1)$ passes

No. of comp : $1+2+3+4\ldots n = \frac{(n)(n+1)}{2} = O(n^2)$

No. of Swaps : $1+2+3+4\ldots n = \frac{(n)(n+1)}{2} = O(n^2)$

Time complexity of any algorithm is given based on the number of comparisions ; so time complexity is $O(n^2)$

Q Why name given as Bubble Sort?

A Because heavier element settles at the bottom and the lighter elements trickles up like bubble.

* For 'k' passes, we get 'k' largest elements.

Pseudo Code ⇒

```
void Bubble sort (int A[], int n) {
    for (int i=0; i<n; i++) {
        for (int j=0; j<n-1-i; j++) {
            if (A[j] > A[j+1];
                swap (A[j], A[j+1]);
        }
    }
}
```

If the given list is already sorted, then we can introduce a flag variable in code and make the sort adaptive

Minimum TC → O(n) Adaptive ✓
Maximum TC → O(n²) Stable ✓

Stable

8 8 8 8
8 8 3 3
3 3 8 5
5 5 5 8

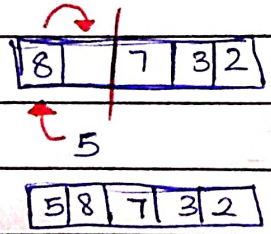
If the sorting algorithm is preserving the position of duplicate elements in the sorted list then it is called stable sorting algorithm

② Insertion Sort

A | 8 | 5 | 7 | 3 | 2 |

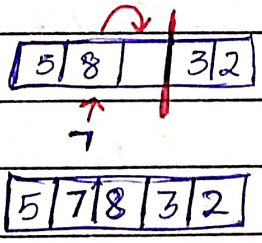
Assuming the first element to be sorted, then we take out the elements from the right hand side and insert it to the first position / left hand side.

1) Insert 5



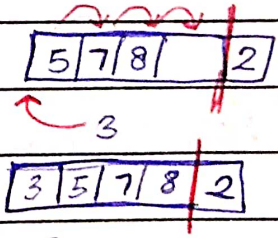
1 comp
1 swap

2) Insert 7



2 comp
2 swaps

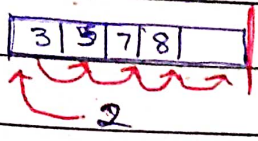
3) Insert 3



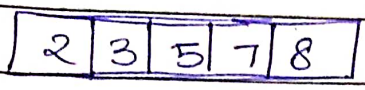
3 comp
3 swaps

4th pass

Insert 2



4 comp
4 swaps



No. of passes → $n-1$

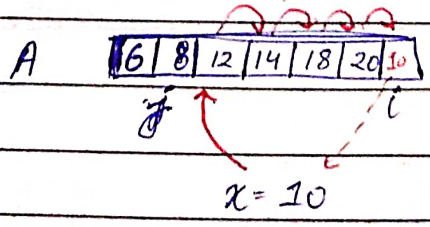
No. of comp → $1+2+3+4 \ldots n = \frac{(n)(n+1)}{2} = O(n^2)$

No. of swaps → $\frac{(n)(n+1)}{2} = O(n^2)$

* Intermediate results do not give any useful result
* Insertion Sort is designed for Linked List

Pseudo Code

```
for (i = 1; i<n; i++)          // Assuming first ele. is sorted
{
    j = i-1;                   // Placing 'j' an index before 'i'
    x = A[i];
    while (j>-1 && A[j]>x)
    {
        A[j+1] = A[j];
        j--;
    }
    A[j+1] = x;
}
```



Analysis

A | 2 | 5 | 8 | 10 | 12 |

No. of comp → n-1 = O(n)
No. of swap → O(1)

So, it is adaptive ✓

A | 3 | 5 | 8 | 10 | 12 | 5 |
i

x = 5

A | 3 | 5 | 8 | 10 | 12 | | ⇒ | 3 | 5 | 5 | 8 | 10 | 12 |
x = 5, i

So, it is stable ✓

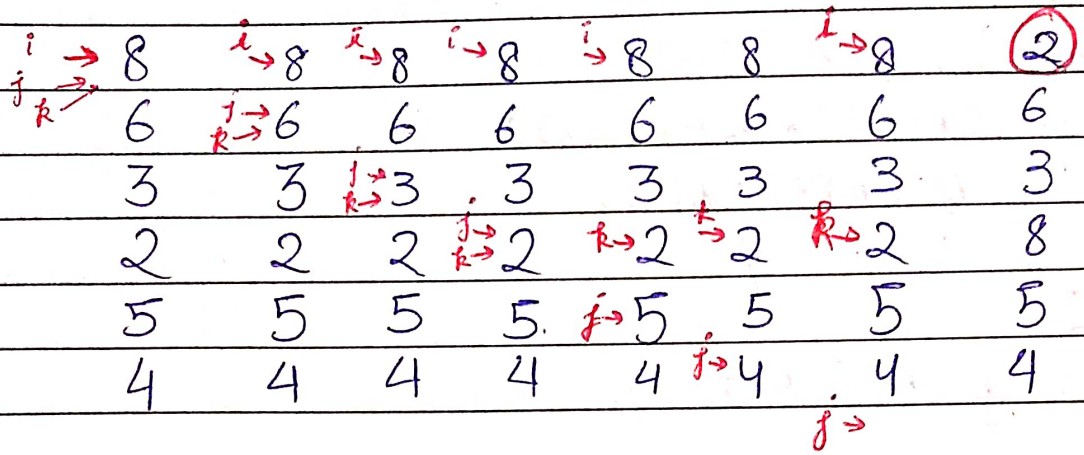
Comparing Bubble and Insertion

	Bubble Sort	Insertion Sort	
Min Comp	O(n)	O(n)	Ascending
Max Comp	$O(n^2)$	$O(n^2)$	Random / Desc
Min Swap	O(1)	O(1)	Ascending
Max Swap	$O(n^2)$	$O(n^2)$	Descending
Adaptive	✓	✓	
Stable	✓	✓	
LinkedList	No	Yes	
K passes	Yes	No	

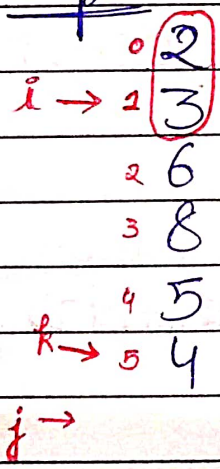
③ Selection Sort (To select a position)

A | 8 | 6 | 3 | 2 | 5 | 4 |

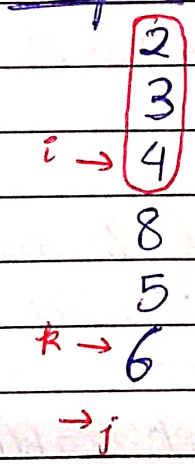
I st pass



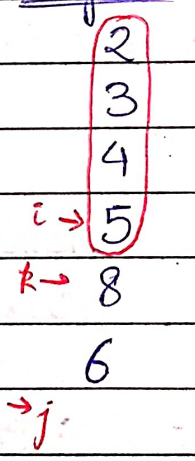
2nd pass



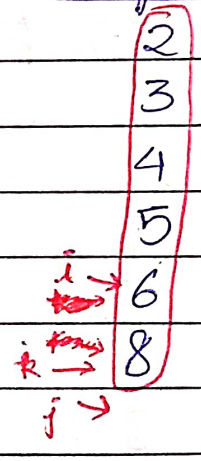
3rd pass



4th pass



5th pass



No. of comp	4	3	2	1
Swaps	1	1	1	1

No. of comp = $1+2+3 \ldots + n-1 = \frac{(n)(n-1)}{2} = O(n^2)$

No. of Swaps = $n-1 = O(n)$

★ Only Algorithm swapping in $O(n)$ [Taken multiple pointers]

★ Good for lesser number of swaps

* Just like bubble sort, these also 'k' intermediate steps give useful results (k-smallest elements)

Pseudo Code

```
int i, j, k;
for (int i=0; i< n-1; i++) {
    for (j=k=i; i<n; j++) {
        if (A[j] < A[k])
            k = j;
    }
    swap (A[i], A[k]);
}
```

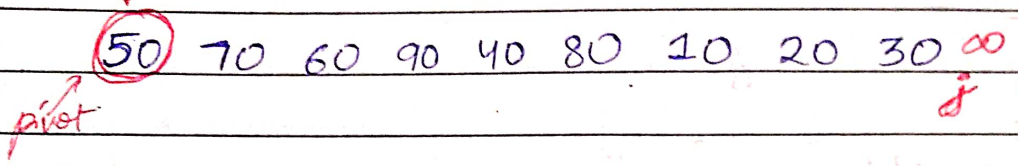
Adaptive ✗ (If the list is sorted or not sorted, it will take $O(n^2)$ time)
Stable ✗

8 ← i	2	Positions of 8 and 8 have been changed (in the sense, order is not preserved)
5	5	∴ It is not stable.
3	3	
8	8	
4	4	
2 ← k	8	
7	7	

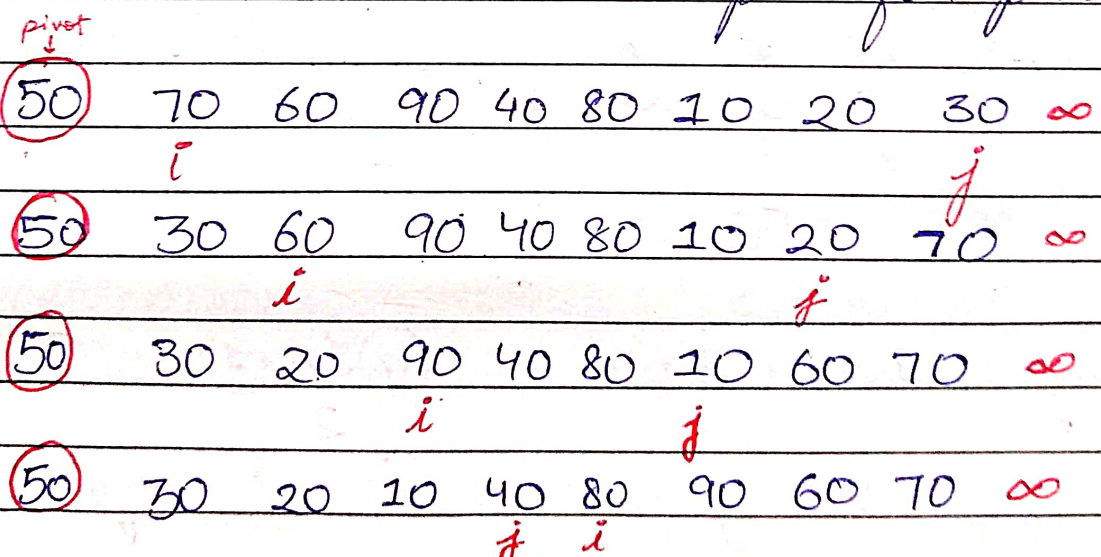
④ Quick Sort

It works on the idea that if an element is in a sorted position, if all the elements before it are smaller and after it are greater.

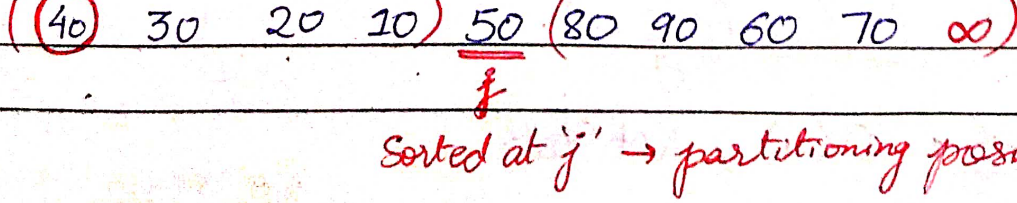
'Quick' here does not mean it is the fastest sorting technique



* 'i' will look for elements greater than pivot (incase no greater element is found, then we will add ∞ to the end)
* 'j' will look for element less than the pivot (or equal to) so that it terminates when it reaches the pivot in worst case
* We will swap the 'i' and 'j' elements if the conditions are met to make suitable place for pivot.



* Once 'j' becomes greater than 'i' then interchange 'j' with pivot element



Pseudocode

```
int Pivot = A[l];
int i = l, j = h;                    // l → first element, h → ∞
do {
    do {i++;} while (A[i] ≤ pivot);  // i → find greater ele
    do {j--;} while (A[j] ≥ pivot);  // j → find lower ele
    if (i<j)
        swap (A[i], A[j]);
} while (i<j);
Swap (A[l], A[j]);                   // Once j>i, swap A[l] with A[j]
return j;
```

l					h		
10	20	30	40	50	∞	n	Sorted List
j	i						
	l				h		
	20	30	40	50	∞	n-1	
	j	i					
		l			h		
		30	40	50	∞	n-2	
		j	i		h		
			l	i	h		
			40	50	∞	2	
			j	i			
				50	∞	1	

No. of comp: $1 + 2 + 3 + \dots n = \frac{(n)(n+1)}{2} = O(n^2)$

Best Case: If partitioning is in middle $O(n \log n)$

Worst Case: If partitioning is done at any one end $O(n^2)$

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

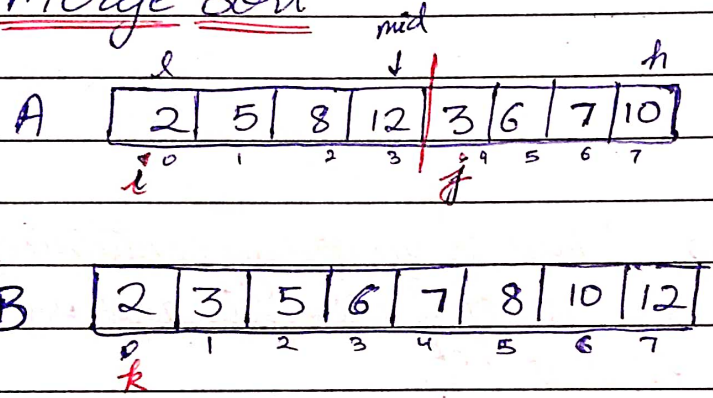
Space Complexity: $O(n)$, Not Stable

Randomize Quick Sort : Randomly selecting any element as pivot

If partioning in middle → $O(n \log n)$
If pastioning at the end → $O(n^2)$

Selection Sort → Selecting an position, finding an element
Quick Sort → Selecting an element, finding a position
↳ Aka → Selection Exchange sort; partition exchange sort

⑤ Merge Sort



From Single to Single

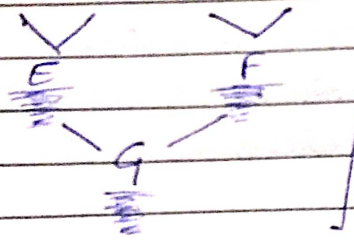
```
int i, j, k, B[h+1];
i = l; j = mid+1; k = l;
while (i<=mid && j<=h)
{   if (A[i] < A[j])
        B[k++] = A[i++];
    else
        B[k++] = A[j++];
}
for (; i<=mid; i++)
    B[k++] = A[i];
for (; j<=h; j++)
    B[k++] = A[j];
for (i=l; i<=h; i++)
    A[i] = B[i];
```

From 2 Array to a single array

```
int i, j, k;
i = j = k = 0;
while (i<m && j<n)
{
    if (A[i] < B[j])
        C[k++] = A[i++];
    else
        C[k++] = B[j++];
}
for (; i<m; i++)
    C[k++] = A[i];
for (; j<n; j++)
    C[k++] = A[j];
```

M-way Merging

A	B	C	D
2	3	5	8
5	6	9	16
15	18	12	20



→ 2-way Merging (General way of doing merge sort)

M-way Merging
⇒ 4-way Merging (When elements of all 4 lists are compared)

Iterative Merge Sort

A [8 | 3 | 7 | 4 | 9 | 2 | 6 | 5]
0 1 2 3 4 5 6 7

1st pass: 3 8 | 4 7 | 2 9 | 5 6

2nd pass: 3 4 7 8 | 2 5 6 9

3rd pass: 2 3 4 5 6 7 8 9

⇒ There is an array containing one list of 8 elements and we have to sort it.

⇒ To apply merge sort, we change the statement, there is an array containing 8 lists. Each element is a list itself

It is done for (log n) times ⇒ (looks like a flipped tree)
We have 8 elements, so $\log_2 8$ ⇒ 3 passes

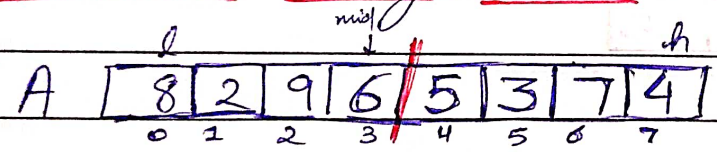
TC → O(n log n)
Stable ✓

SC → n + n + log n → O(n)
(Extra Array + Stack space)

Pseudo Code

```
int p, i, l, mid, h;
for (p=2; p<=n; p=p*2)
{
    for (i=0; i+p-1<n; i=i+p)
    {
        l=i;
        h=i+p-1;
        mid = [(l+h)/2];
        merge(A, l, mid, h);
    }
}
if (p/2<n)
    merge (A, 0, p/2, n-1);
```

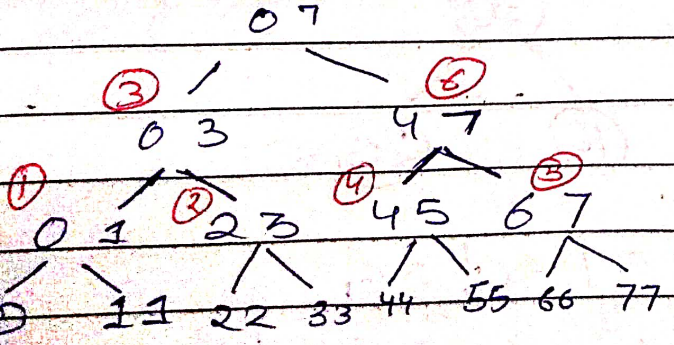
Recursive Merge Sort



28 69 35 47

26 89 3457

23456789




```
void MergeSort (int A[], int l, int h)
{
    if (l<h)
    {
        mid = [(l+h)/2]
        MergeSort (A, l, mid);
        MergeSort (A, mid+1, h);
        Merge (A, l, mid, h);
    }
}
```

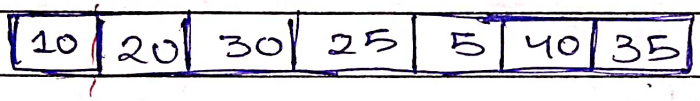
TC → O(n log n)

Done in Post Order

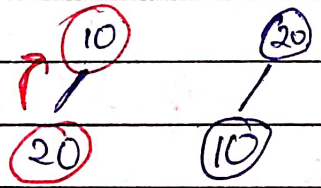
⑥ Heap Sort (Has to be a complete Binary tree)

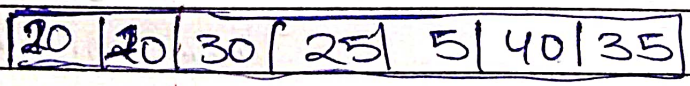
```
void Insert (int A[], int n)
{
    int temp, i=n;
    temp = A[n];
    while (i>1 && temp > A[i/2])
    {
        A[i] = A[i/2];
        i = i/2;
    }
    A[i] = temp;
}
```

1. Insert 10

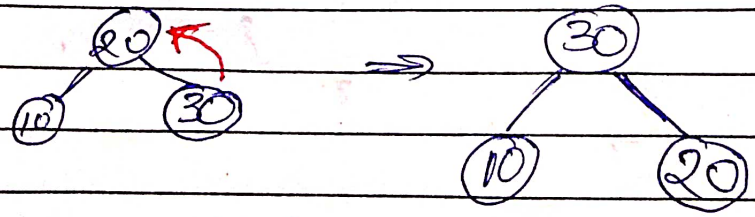


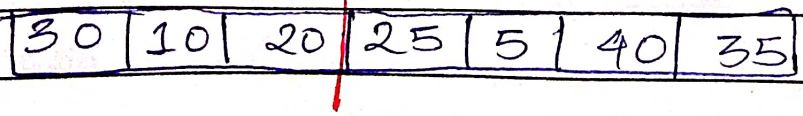
2. Insert 20



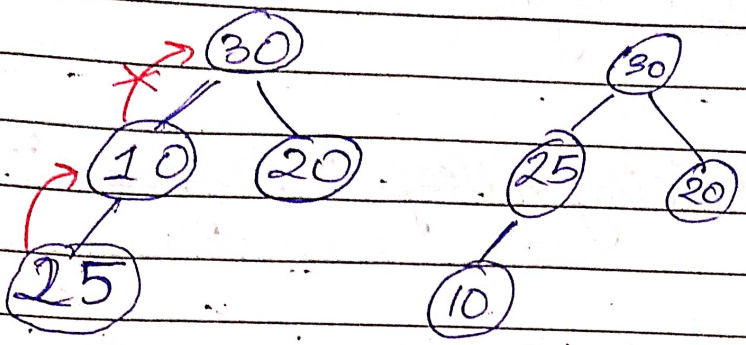


3. Insert 30



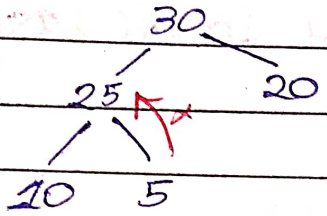


4. Insert 25



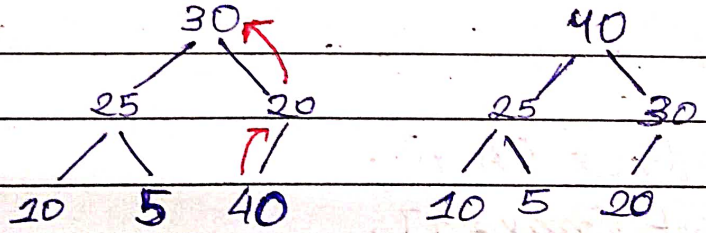
A	30	25	20	10	5	40	35
	1	2	3	4	5	6	7

5. Insert 5



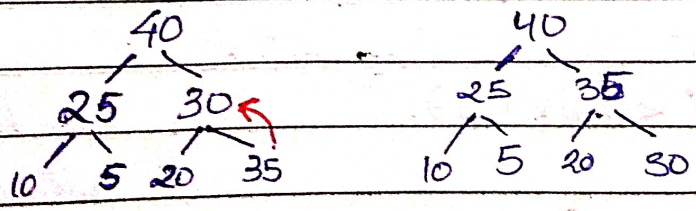
A	30	25	20	10	5	40	35

6. Insert 40



A	40	25	30	10	5	20	35

7. Insert 35



A	40	25	35	10	5	20	30

Creation of Heap → O(n log n)
SC → O(1)

Bounded condition for Deleting in Heap ⇒ Only delete the large element

Deletion of Heap → O(n log n)

⑦ Count Sort

* Fast, but consumes space
* Size of 'Count' array must be equal to the largest element from the given array.
* TC → O(N) and SC → O(N)

Pseudo Code

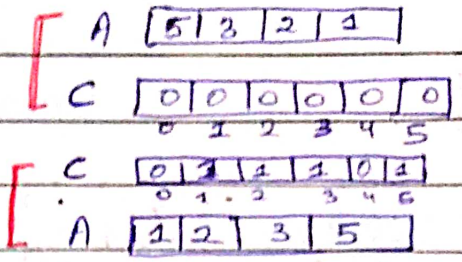
```
void CountSort (int A[], int n)
{
    int max, i;
    int *C;
    max = findmax (A, n);
    C = new int [max+1];
    for (int i=0; i<max+1; i++)
        C[i] = 0;
    for (int i=0; i<n; i++)
        C [A[i]]++;
    i=0, j=0;
    while (i < max+1)
    {
        if (C[i]>0)
            A[j++]=i; C[i]--;
```

// Initialize Count Array

// Add the count of nums to count array

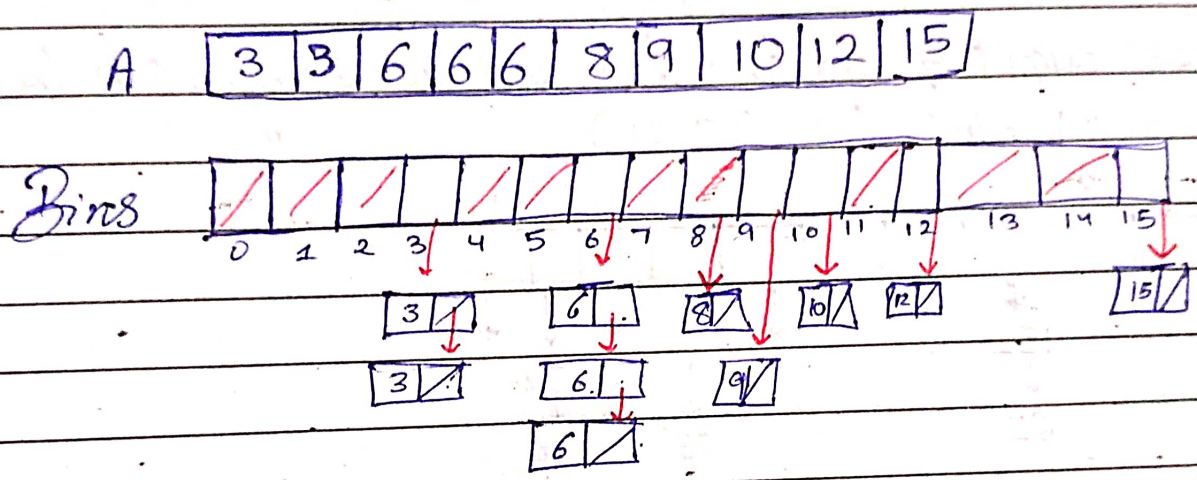
// Re-fill the given array with count array

```
else
    i++;
}
}
```



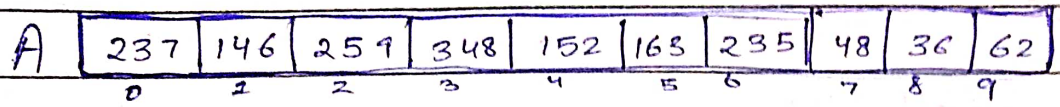
⑧ Bucket / Bin Sort

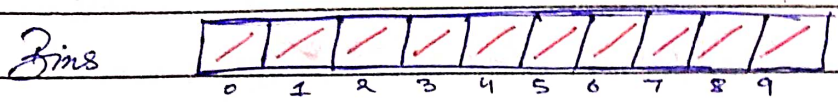
* Array of Linked List
* TC → O(n), SC → O(n), Stable → ✓



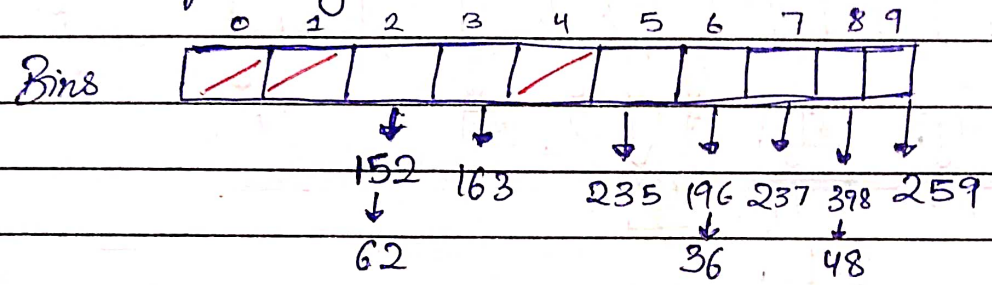
⑨ Radix Sort

* Based on modulo operation of the elements
* 9 bins created to handle the elements

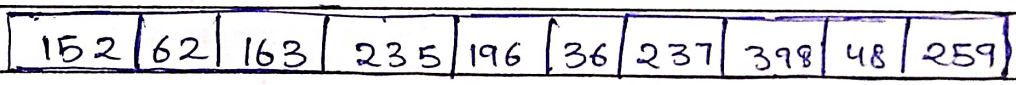




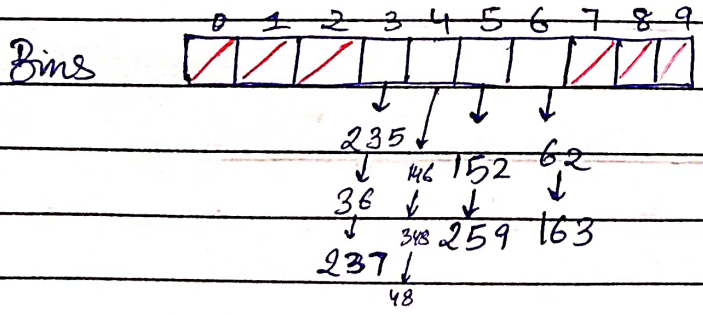
① Check last digit of the numbers and then drop it in the corresponding bin (A[i] % 10)



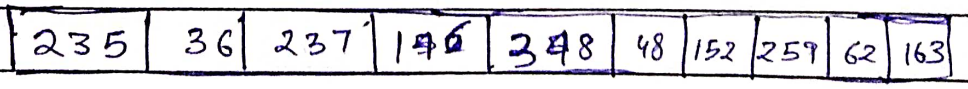
Putting back to array ⇒ 1st pass



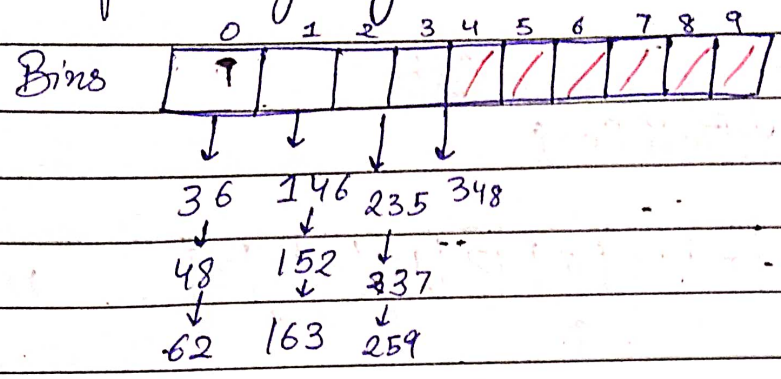
② Checking the second digit of numbers [(A/10) % 10]



Putting back to Array ⇒ 2nd pass



⑤ Check the first digit of numbers



Putting back to Array] 3rd pass

36	48	62	146	152	163	235	237	259	348

TC → O(n)

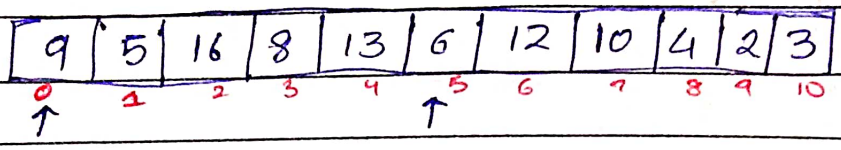
SC → O(n)

Stable → ✓

(10) Shell Sort

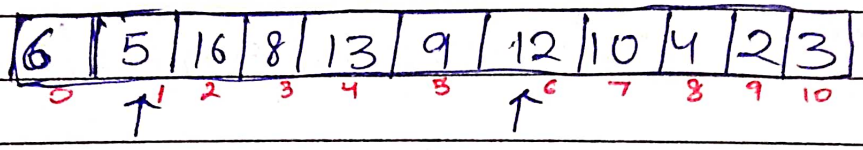
* ★ Used for sorting very large size lists
* ★ Extension of Insertion sort

$gap = \left\lfloor \frac{n}{2} \right\rfloor$ A

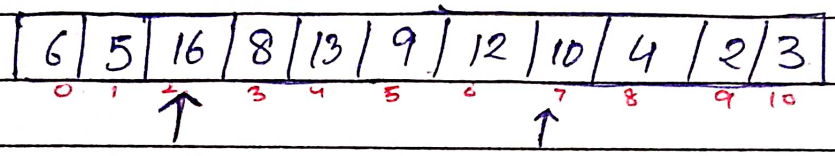


$n = 11$

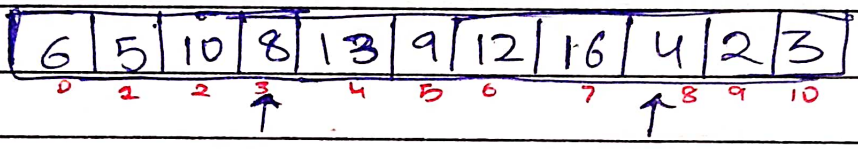
$gap = \left\lfloor \frac{n}{2} \right\rfloor$



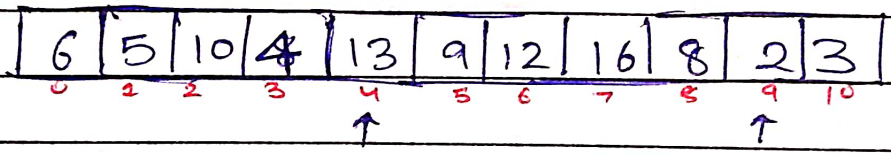
$gap = \left\lfloor \frac{n}{2} \right\rfloor$



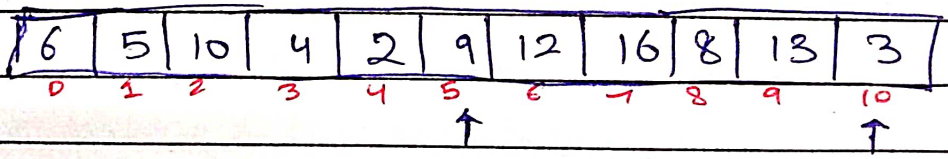
$gap = \left\lfloor \frac{n}{2} \right\rfloor$



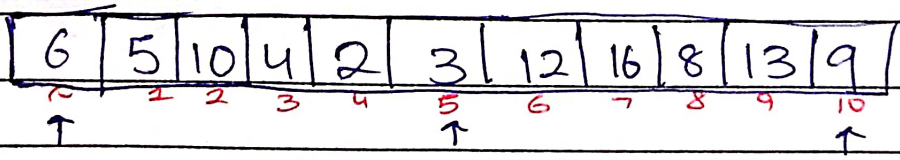
$gap = \left\lfloor \frac{n}{2} \right\rfloor$

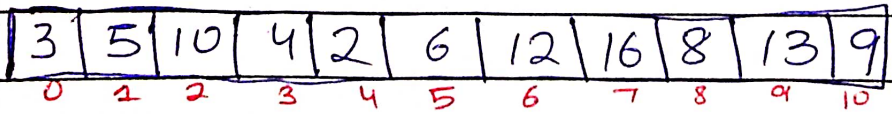


$gap = \left\lfloor \frac{n}{2} \right\rfloor$



$gap = \left\lfloor \frac{n}{2} \right\rfloor$





Performing @ $gap = \left\lfloor \frac{5}{2} \right\rfloor = 2$

Final after 2 pointer swaps

2	4	3	5	8	6	9	13	10	16	12

$gap = \left\lfloor \frac{2}{2} \right\rfloor = 1$

Final Array after 1 pointer

2	3	4	5	6	8	9	10	12	13	16

* Can consider taking prime numbers as well for gaps
* General TC → $O(n \log_2 n)$, SC → $O(1)$
  But incase taking prime numbers as gaps ⇒ $O(n^{3/2})$
* Adaptive ✓ (Following Insertion Sort)
* Stable ✗ (Change in every gap)