Sorting

Overview of Sorting Algorithms:

① Binary Array ⇒ partition of quickSort (Lomuto, Hoare, Naive)

② Array with three values (0, 1, 2)

③ Array of size n and small ranged value ⇒ Counting Sort
↓
O(k) extra space
O(n+k) Time

④ Array of size n and range is of size n^2 (or) n^3 (or) closer ⇒ Radix Sort
O(n) Time
O(n) Extra

⑤ Array of uniformly distributed data over a range ⇒ Bucket Sort

⑥ when memory writes are costly ⇒ Selection Sort, Cycle Sort ⇒ optimal

⑦ when adjacent swaps are allowed ⇒ Bubble Sort, Cocktail Sort ⇒ optimal

⑧ when array size is small ⇒ Insertion Sort

⑨ when available extra memory is less ⇒ Shell Sort
↓
TC = O(n log²n)
(or)
O(n * (log n)²)

General purpose Sorting Algorithms:

merge sort → divide & conquer
Heap Sort
Quick Sort

Hybrid Algorithms (used in Libraries)

Tim Sort ⇒ mix of (Insertion Sort + merge Sort)
IntroSort ⇒ mix of (Quick + Heap + Insertion)

stable-sort

Sort in C++ STL

Sort is mainly used for data stored in containers which allow random access ⇒ Arrays, vectors, deque

```
#include<iostream>
#include<algorithm>
using namespace std;
int main()
{
    int arr[] = {10,20, 5, 7};
    sort (arr, arr+n);
    for (int x : arr)
        cout << x << " ";
    sort (arr, arr+n, greater<int>());
    cout << endl;
    for (int x: arr)
        cout << x << " ";
}
```

o/p:

5	7	10	20
20	10	7	5

for arrays

for vectors

```
vector <int> V = {5,7,20,10};
sort (v.begin(), v.end());
sort (v.begin(), v.end(),
      greater<int>());
```

⇒ {5,7,10,20}

[20, 10, 7, 5]

```
#include<iostream>
#include<algorithm>
using namespace std;
struct point{
  int x,y;
};
bool mycomp(point p1, point p2)
{ return (p1.x < p2.x); }
int main()
{ point arr[] = {{3,10},{2,8},{5,4}}
  sort(arr, arr+n, mycomp);
  for(auto i:arr)
  cout<<i.x<<" "<<i.y<<endl;
}
```

O/P: 2 8
3 10
5 4

⇒ worst case and Average case = O(nlogn)
⇒ uses Intro Sort (Hybrid of Quick Sort, Heap Sort and Insertion Sort)

Stability in Sorting Algorithms

arr = [("Anil", 50), ("Ayan", 80), ("piyush", 50), ("Ramesh", 80)]

⇒ [("Anil", 50), ("piyush", 50), ("Ayan", 80), ("Ramesh", 80)] ⇒ Stable

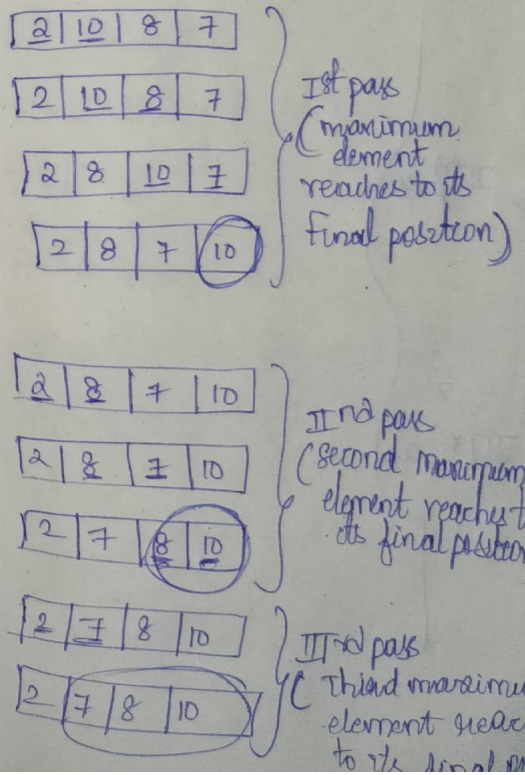
⇒ [("piyush", 50), ("Anil", 50), ("Ayan", 80), ("Ramesh", 80)]

⇒ Unstable

example stable Sorts: Bubble Sort, Insertion Sort, Merge Sort, - - -

example unstable Sort: Selection Sort, Quick Sort, Heap Sort -

Bubble Sort



```
void bubbleSort(arr, n)
{
  for (i=0; i<n-1; i++)
    for (j=0; j<n-1; j++)
      if (arr[j] > arr[j+1])
        swap(arr[j], arr[j+1]);
}
```

optimisation ⟺

```
for (i=0; i<n-1; i++)
  for (j=0; j<n-i-1; j++)
    if (arr[j] > arr[j+1])
      swap(arr[j], arr[j+1]);
```

n=4

i=0

j=0	10	8	20	5
j=1	8	10	20	5
j=2	8	10	20	5
	8	10	5	20

I st pass

i=1

j=0	8	10	5	20
j=1	8	10	5	20
	8	5	10	20

II nd pass

i=2

j=0	8	5	10	20
	5	8	10	20

III rd pass

Tc

$(n-1)+(n-2)+\dots+2+1$

$\Rightarrow \frac{n*(n-1)}{2}$

$\Rightarrow \boxed{\theta(n^2)}$

Optimized Bubble Sort :

```
void bubbleSort(arr, n)
{
  for (i=0; i<n-1; i++)
  {
    swapped = false;
    for (j=0; j<n-i-1; j++)
    {
      if (arr[j] > arr[j+1])
      {
        swap(arr[j], arr[j+1])
        swapped = true;
      }
    }
    if (swapped == false)
      break;
  }
}
```

i=0

j=0	3	5	10	20	40
j=1	3	5	10	20	40
j=2	3	5	10	20	40
j=3	3	5	10	20	40

Stability :

6 8 6 6

6 8 6 6

6 6 8 6

6 6 6 8

I st pass

6 6 6 8

6 6 6 8

6 6 6 8

II nd pass

Merge two Sorted Arrays (without using extra space)

Gap Method ⟹ Incremental (shell sort)

TC = O((n+m) log(n+m))

AS = O(1)

Selection Sort

writing data to I/O

Memory writes is a costly ops in terms of EEP rom. In EEP Rom if we do more writes age of memory reduces So we prefer selection sort.

10	5	8	20	2	18

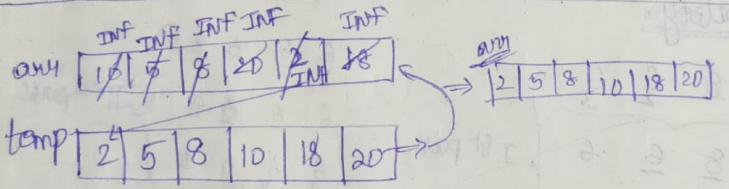
↳ $\theta(n^2)$ Algorithm

↳ Does less memory writes compared to Quick Sort, merge Sort, Insertion Sort etc But cycle Sort is optimal in terms of memory writes.

↳ Basic Idea for Heap Sort.

↳ Not stable

↳ In-place (doesn't require extra space)



Naive Implementation:

```
void selectionSort(arr,n)
{ int temp[n];
  for(int i=0; i<n; i++)
  { int min_ind=0;
    for(int j=1; j<n; j++)
    { if(arr[j]<arr[min_ind])
      { min_ind=j;
      }
    }
    temp[i]=arr[min_ind];
    arr[min_ind]=INF;
  }
  for(int i=0; i<n; i++)
  { arr[i]=temp[i];
  }
```

In-place Implementation

10	5	8	20	2	18
0	1	2	3	4	5

i=0 : min_ind=4

2	5	8	20	10	18

i=1 : min_ind=1

2	5	8	20	10	18

i=2 : min_ind=2

2	5	8	20	10	18

i=3 : min_ind=4

2	5	8	10	20	18

i=4 : min_ind=5

2	5	8	10	18	20

i=5 : min_ind=5

2	5	8	10	18	20

```
void selectSort(arr,n)
{
  for(i=0; i<n+; i++)
  { min_ind=i;
    for(int j=i+1; j<n; j++)
    { if(arr[j]<arr[min_ind])
        min_ind=j;
    }
    swap(arr[min_ind], arr[i]);
  }
}
```

Time Complexity

$(n-1)+(n-2)+\cdots+2+1$

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

AS = $\theta(1)$

Why it is not stable?

90	80	90	25
0	1	2	3

i=0 ; min_ind = 3

25	80	90	90

[the first 90 moves after the second 90]

i=1 ; min_ind = 1

25	80	90	90

i=2 ; min_ind=2

25	80	90	90

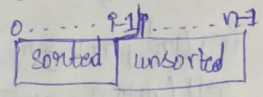
i=3 ; min_ind=3

25	80	90	90

Insertion Sort

- $O(n^2)$ worst case
- In-place and stable
- Used in practice for small arrays (Tim Sort & IntroSort)
- $O(n)$ in Best Case

Idea of Insertion Sort:



[20, 5, 40, 60, 10, 30] i=1

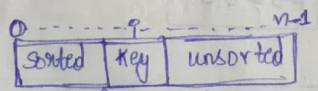
sorted [5, 20, 40, 60, 10, 30] i=2

[5, 20, 40, 60, 10, 30] i=3

[5, 20, 40, 60, 10, 30] i=4

[5, 10, 20, 40, 60, 30] i=5

[5, 10, 20, 30, 40, 60]



i goes from 1 to n-1

[20, 5, 40, 60, 10, 30] i=1, j=0

[5, 20, 40, 60, 10, 30] i=2

[5, 20, 40, 60, 10, 30] i=3

[5, 20, 40, 60, 10, 30] i=4, j=3,2,1

[5, 10, 20, 40, 60, 30] i=5, j=4,3,2

[5, 10, 20, 30, 40, 60]

```
Void insertionSort(int arr[], int n)
{
  for (int i=1; i<n; i++)
  {
    int key = arr[i];
    int j = i-1;
    while (j>=0 && arr[j]>key)
    {
      arr[j+1] = arr[j];
      j--;
    }
    arr[j+1] = key;
  }
}
```

to make this algorithm stable we don't consider equal to (=> sign here

Time Complexity:

Best case: Already sorted

[10, 20, 30, 50] $\Rightarrow \Theta(n)$

In General: $O(n^2)$

Worst case: Reverse Sorted

[50, 30, 20, 10]

[30, 50, 20, 10]

[20, 30, 50, 10]

$1+2+3 \cdots +(n-1)$

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

$\Rightarrow \Theta(n^2)$

Merge Sort

① Divide and conquer Algorithm
(Divide, conquer and merge)

② Stable Algorithm

③ $\Theta(n \log n)$ time and $O(n)$ Aux space.

④ well suited for Linked Lists. works in $O(1)$ Aux space

⑤ used in external sorting.

⑥ In general for arrays, Quicksort outperforms it

merge two Sorted Arrays:

Naive solution: T.C = $O((m+n) * \log(m+n))$, A.S = $\Theta(m+n)$

a[] = [10, 15, 20, 20], m = 4

b[] = [1, 12], n = 2

After I st loop: c[] = [10, 15, 20, 20, -, -]

After II nd loop: c[] = [10, 15, 20, 20, 1, 12]

After Sorting: c[] = [1, 10, 12, 15, 20, 20]

```
void merge (int a[], int b[], int m, int n)
{ int c[m+n];
  for(int i=0; i<m; i++) c[i] = a[i];
  for(int i=0; i<n; i++) c[m+i] = b[i];
  sort (c, c+m+n);
  for (int i=0; i<(m+n); i++)
    cout<<c[i]<<" ";
}
```

Efficient Solution: $\Theta(m+n)$ Time

a[] = [| |] (i)

b[] = [| |] (j)

Initially i=0, j=0

```
if (a[i]<=b[j]) { ? }
else { ? }
```

this is tells about stability

```
if (a[i]<=b[j]) { print(a[i]); i++; }
else { print(b[j]); j++; }
```

a[] = [10, 20, 35]
b[] = [5, 50, 50]

i=0, j=0
5 → j=1
10 → i=1
20 → i=2
35 → i=3

T.C = $\Theta(m+n)$
A.S = $O(1)$

```
void merge (int a[], int b[], int m, int n)
{ int i=0, j=0;
  while (i<m && j<n)
  { if (a[i]<=b[j])
    { cout<<a[i]<<" "; i++; }
    else
    { cout<<b[j]<<" "; j++; }
  }
  while (i<m)
  { cout<<a[i]<<" "; i++; }
  while (j<n)
  { cout<<b[j]<<" "; j++; }
}
```

a = [10, 20, 50] m=3
b = [5, 50, 50] n=3
Initially: i=0, j=0
I st iteration: print(5), j=1
II nd iteration: print(10), i=1
III rd iteration: print(20), i=2
IV th iteration: print(50), i=3
Next while loop:
I st iteration: print(50), j=2
II nd " : print(50), j=3

Merge function of merge sort:

I/P: a[] = [10, 15, 20, 11, 30]

low = 0
mid = 2
high = 4

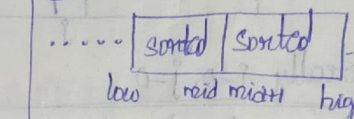
O/P: a[] = [10, 11, 15, 20, 30]

I/P: a[] = [5, 8, 12, 14, 7]

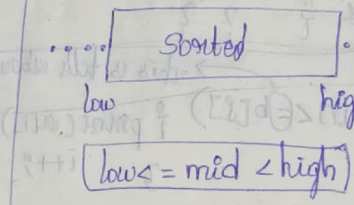
low = 0
mid = 3
high = 4

O/P: a[] = [5, 7, 8, 12, 14]

I/P:



O/P:



a[] = [10, 15, 20, 40, 8, 11, 55]

low = 0
mid = 3
high = 6

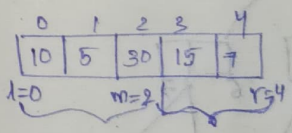
left[] = [10, 15, 20, 40]
right[] = [8, 11, 55]
a[] = [8, 10, 11, 15, 20, 40, 55]

```
void merge(int a[], int low, int mid, int high)
{
    int n1 = mid-low+1, n2 = high-mid;          // setting up auxiliary arrays
    int left[n1], right[n2];
    for (int i=0; i<n1; i++) { left[i] = a[low+i]; }
    for (int i=0; i<n2; i++) { right[i] = a[mid+i+1]; }

    int i=0, j=0, k=low;
    while (i<n1 && j<n2)                        // => standard merge logic
    { if (left[i] <= right[j]) { a[k] = left[i]; i++; k++; }
      else { a[k] = right[j]; j++; k++; }
    }
    while (i<n1) { a[k] = left[i]; i++; k++; }
    while (j<n2) { a[k] = right[j]; j++; k++; }
}
```

T.C = θ(n)
A.S = θ(n)

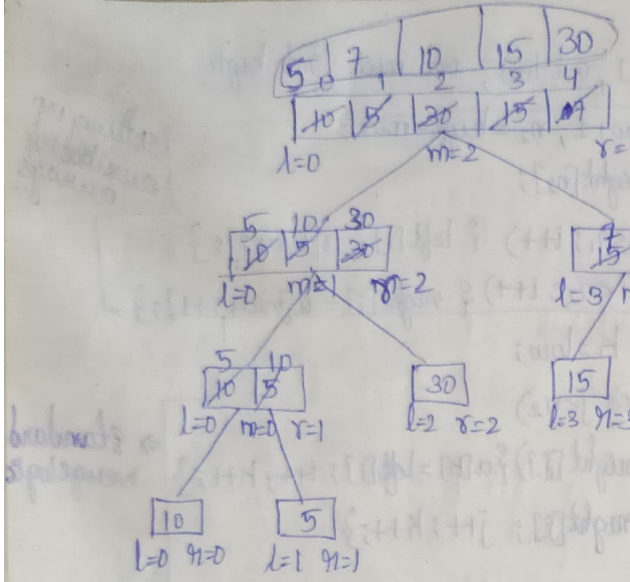
Merge Sort Algorithm



5 10 30 7 15

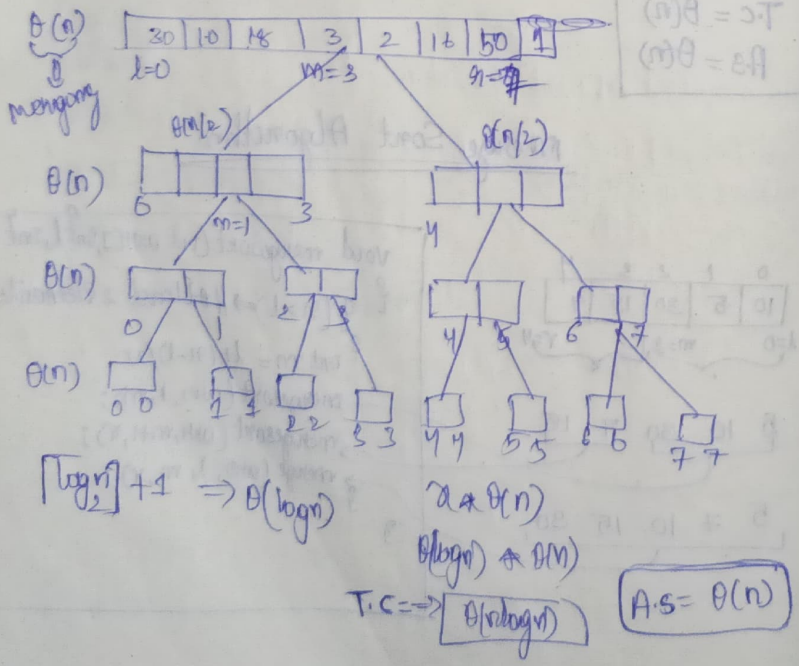
5 7 10 15 30

```
void mergeSort(int arr[], int l, int r)
{
    if (r > l) // Atleast 2 elements
    { int m = l + (r-l)/2;
      mergesort(arr, l, m);
      mergesort(arr, m+1, r);
      merge(arr, l, m, r);
    }
}
```



merge sort Analysis:

n is not power of two merge sort Analysis:



$\lceil \log_2 n \rceil + 1 \Rightarrow \theta(\log n)$

$\theta(\log n) * \theta(n)$

$\theta(n \log n) * \theta(n)$

T.C ⇒ $\theta(n \log n)$

A.S = $\theta(n)$

Intersection of two Sorted Arrays

I/p: a[] = [3, 5, 10, 10, 10, 15, 15, 20]

b[] = [5, 10, 10, 15, 30]

o/p: 5, 10, 15

I/p: a[] = [1, 1, 3, 3, 3]

b[] = [1, 1, 1, 1, 3, 5, 7]

o/p: 1 3

Naive Solution: O(n x m)

a[] = [1, 20, 20, 40, 60]

b[] = [2, 20, 20, 20]

i=0 : j=0,1,2,3

i=1 : j=0,1

print(20)

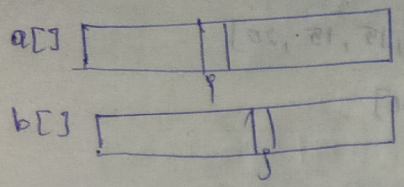
i=2 :

i=3 : j=0,1,2,3

i=4 : j=0,1,2,3

```
void intersection(int a[], int b[],
                  int m, int n)
{
  for (int i=0; i<m; i++)
  {
    if (i>0 && a[i]==a[i-1])
      continue;
    for (int j=0; j<n; j++)
    {
      if (a[i]==b[j])
      {
        cout << a[i] << " ";
        break;
      }
    }
  }
}
```

Efficient Solution: $\Theta(m+n) \Rightarrow$ T.C




```
if (i>0 && a[i]==a[i-1]){ i++; continue; }
if (a[i]<b[j]) { i++; }
if (a[i]>b[j]) { j++; }
if (a[i]==b[j]) { print(a[i]); i++; j++; }
```

a[] = [10, 20, 80, 40, 60]
b[] = [2, 20, 20, 20]
Initially: i=0, j=0
I st iteration: j=1
II nd iteration: i=1
III rd " : print(20) i=2 j=2
IV th " : i=3
V th : j=3
VI th : j=4

```
void intersection(int a[], int b[], int m, int n)
{ int i=0, j=0;
  while(i<m && j<n)
  { if(i>0 && a[i]==a[i-1])
    { i++; continue; }
    if(a[i]<b[j]) { i++; }
    else if(a[i]>b[j]) { j++; }
    else
    { cout<<a[i]<<" ";
      i++;
    } j++;
  }
}
```

Union of two Sorted Arrays

I/p: a[] = [3, 5, 8]
b[] = [2, 8, 9, 10, 15]

O/p: [2, 3, 5, 8, 9, 10, 15]

I/p: a[] = [2, 3, 3, 3, 4, 4]
b[] = [4, 4]

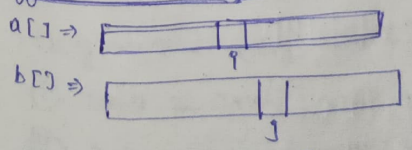
O/p: 2 3 4

Naive: T.C = $O((m+n)\log(m+n))$

```
void printUnion(int a[], int b[], int m, int n)
{ int c[m+n];
  for(i=0; i<m; i++)
    c[i] = a[i];
  for(j=0; j<n; j++)
    c[m+j] = b[j];
  Sort(c, c+m+n);
  for(int i=0; i<m+n; i++)
  { if(i==0 || c[i] != c[i-1])
      cout<<c[i]<<" ";
  }
}
```

a[] = [10, 20, 20], m=3
b[] = [5, 20, 40, 40], n=4
After I st loop: c[] = [10, 20, 20, -, -, -, -]
After II nd loop: c[] = [10, 20, 20, 5, 20, 40, 40]
After sorting: c[] = [5, 10, 20, 20, 20, 40, 40]
III rd loop: 5 10 20 40

Efficient solution: T.C = $\Theta(m+n)$



a[] = [2, 3, 20, 20]
b[] = [3, 20, 40]

~~2, 3, 10~~

2 3 10 20

```
if(i>0 && a[i]==a[i-1]){ i++; continue; }
if(j>0 && b[j]==b[j-1]){ j++; continue; }
if(a[i]<b[j]) { print(a[i]); i++; }
if(a[i]>b[j]) { print(b[j]); j++; }
if(a[i]==b[j]) { print(a[i]); i++; j++; }
```

```
void printUnion(int a[], int b[], int m, int n)
{
  int i=0, j=0;
  while (i<m && j<n)
  {
    if(i>0 && a[i]==a[i-1]){ i++; continue; }
    if(j>0 && b[j]==b[j-1]){ j++; continue; }
    if(a[i]<b[j]) { cout<<a[i]<<" "; i++;}
    else if (a[i]>b[j]) { cout<<b[j]<<" "; j++;}
    else { cout<<a[i]<<" "; i++; j++;}
  }
  while (i<m)
  { if(i==0 || a[i]!=a[i-1]) { cout<<a[i]<<" ";} i++;}
  while (j<n)
  { if(j==0 || b[j]!=b[j-1]) { cout<<b[j]<<" ";} j++;}
}
```

efficient solution

T.C = O(m+n)
A.S = O(1)

a[] = [2, 10, 20, 20]
b[] = [3, 20, 40]
i=0, j=0
1st Iteration: print(2), i=1
2nd " " : print(3), j=1
3rd " " : print(10), i=2
4th " " : print(20), i=3, j=2
5th " " : i=4
Next while loop: while (i<m)
1st Iteration: 40, j=3

Count Inversions in an Array

A pair (arr[i], arr[j]) forms an inversion when (i<j) and arr[i] > arr[j]

I/p: [2, 4, 1, 3, 5]
o/p: 3 (2,1) (4,1) (4,3)

I/p: [10, 20, 30, 40]
o/p: 0

I/p: [40, 30, 20, 10] ⇒ (40,30) (30,20) (40,20) (30,10) (40,10) (20,10)
o/p: 6

when array is sorted in decreasing order
(n-1)+(n-2)+(n-3)- - - +1
n(n-1)/2 Inversions

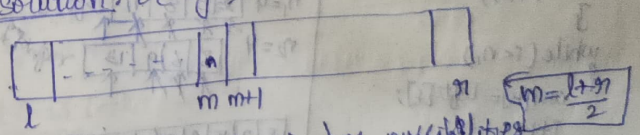
Naive solution:

T.C = O(n²)
A.S = O(1)

```
int countInversions(int arr[], int n)
{
  int res=0;
  for(int i=0; i<n; i++)
    for(int j=i+1; j<n; j++)
      if(a[j]<a[i])
        res++;
  return res;
}
```

[2, 4, 1, 3, 5]
i=0 : res=1
i=1 : res=3
i=2 : res=3
i=4 : res=3

Efficient solution: O(nlogn) time



m = (l+r)/2

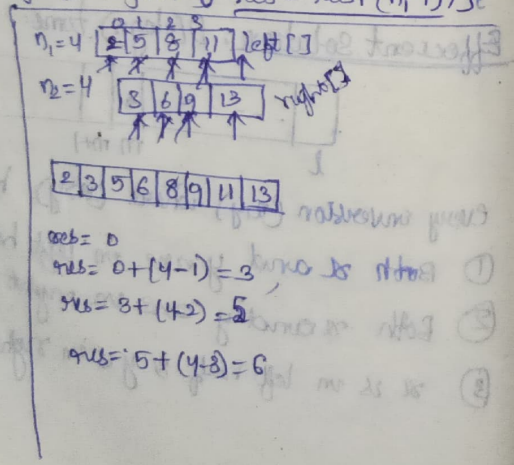
Every inversion (x,y) where (x>y) has possibilities
① Both x and y are in left half
② Both x and y are in right half
③ x is in left half & y is in right half

T.C = O(nlogn)

```
int countInv(int arr[], int l, int r)
{ int res=0;
  if(l<r)
  { int m = (r+l)/2;
    res += countInv(arr, l, m);
    res += countInv(arr, m+1, r);
    res += countAndMerge(arr, l, m, r);
  }
  return res;
}
```

T.C = O(nlogn)
A.S = O(n)

```
int CountAndMerge(int arr[], int l, int m, int r)
{
  int n1 = m-l+1, n2 = r-m;
  int left[n1], right[n2];
  for(int i=0; i<n1; i++) { left[i] = arr[l+i]; }
  for(int i=0; i<n2; i++) { right[i] = arr[m+1+i]; }
  int res=0, i=0, j=0, k=l;
  while (i<n1 && j<n2)
  {
    if(left[i] <= right[j]) { arr[k] = left[i]; i++; }
    else { arr[k] = right[j]; j++; res = res + (n1-i); }
    k++;
  }
  while(i<n1)
  { arr[k] = left[i];
    i++;
    k++;
  }
  while (j<n2)
  { arr[k] = right[j];
    j++;
    k++;
  }
  return res;
}
```



partition of a Given Array (stable)

Naive partition

I/P: arr[] = [3, 8, 6, 12, 10, 7]
p=5 // index of the last element

O/P: arr[] = [3, 6, 7, 8, 12, 10]
(OR)
[6, 3, 7, 12, 8, 10]
(OR)
Permutations!

Naive: arr[] = [5, 13, 6, 9, 12, 11, 8]
pivot p is given
l (low) = 0
h (high) = 6
p = 6
O/P: temp[] = [5, 6, 8, 9, 13, 12, 11]
arr[] = [5, 6, 8, 9, 13, 12, 11]

```
Void partition(int arr[], int l, int h, int p)
{ int temp[h-l+1], index=0;
  for(int i=l; i<=h; i++)
    if(arr[i] <= arr[p])
    { temp[index] = arr[i];
      index++;
    }
  for(int i=l; i<=h; i++)
    if(arr[i] > arr[p])
    { temp[index] = arr[i];
      index++;
    }
  for(int i=l; i<=h; i++)
    arr[i] = temp[i-l];
}
```

T.C = O(n)
A.S = O(n)

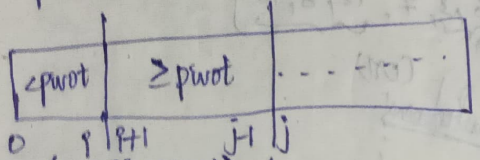
(Lomuto partition)

pivot is always a last element in the array

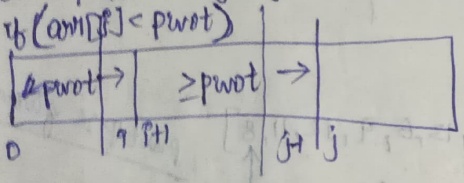
arr[] = [10, 80, 30, 90, 40, 50, 70]

l=0, h=6

pivot = 70



if (arr[j] < pivot)




```
int lpartition(int arr[], int l, int h)
{ int pivot = arr[h]; // Always last
  int i = l-1;
  for (int j=l; j<=h-1; j++)
  {
    if(arr[j] < pivot)
    {
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i+1], arr[h]);
  return (i+1);
}
```

T.C = O(n)
A.S = O(1)

arr[] = [10, 80, 30, 90, 40, 50, 70]

pivot loop

low=0, high=6, pivot=70

i=-1, j=0

[10, 80, 30, 90, 40, 50, 70]

i=0, j=1

[10, 80, 30, 90, 40, 50, 70]

i=0, j=2

[10, 30, 80, 90, 40, 50, 70]

i=1, j=3

[10, 30, 80, 90, 40, 50, 70]

i=1, j=4

[10, 30, 40, 90, 80, 50, 70]

i=2, j=5

[10, 30, 40, 50, 80, 90, 70]

i=3 j=6

we stop here because j=h

we now do:

swap(arr[i+1], arr[h])

return (i+1)

[10, 30, 40, 50, 70, 90, 80]

Corner cases:

① [70, 60, 80, 40, 30] pivot = 30

i=-1 >30

array all elements are greater than pivot

when j = h+1

swap(a[i+1], a[h])

⇓

[30, 60, 80, 40, 70]

② [30, 40, 20, 50, 80]

i=-1 j=0

⇓

[30, 40, 20, 50, 80]

How to Handle the cases when pivot is not the last element?

[50, 30, 20, 10, 5, 11]

p=2

swap(arr[p], arr[h])

⇓

[50, 30, 11, 10, 5, 20]

Hoare's partition (Not stable)

⇓

pivot is always first element

l=0 h=7

[5, 3, 8, 4, 2, 7, 1, 10]

pivot = arr[l] = 5

i = l-1 = -1

j = h+1 = 8

<pivot	unknown?	>pivot

l → i ← j h

```
int partition(int arr[], int l, int h)
{
  int pivot = arr[l];
  int i = l-1, j = h+1;
  while(true)
  {
    do{ i++; }while(arr[i]<pivot);
    do{ j--; }while(arr[j]>pivot);
    if(i>=j) return j;
    swap(arr[i], arr[j]);
  }
}
```

i=-1, j=8

[5, 3, 8, 4, 2, 7, 1, 10]

i=0, j=6

[1, 3, 8, 4, 2, 7, 5, 10]

i=2, j=4

[1, 3, 2, 4, 8, 7, 5, 10]

i=4, j=3

we return j as i≥j now

Corner cases:

pivot is smallest: [5, 10, 9, 12]

l=0 h=3

i=-1, j=4

i=0, j=0 ⇒ [5, 10, 9, 12]

pivot is largest:

[12, 10, 5, 9]

l=0 h=3

i=-1, j=4

[12, 10, 5, 9]

i=0, j=3

[9, 10, 5, 12]

i=3, j=2

All elements are same: (unstable)

l=0 h=3

[5, 5, 5, 5]

i=-1, j=4

[5, 5, 5, 5]

i=0, j=3

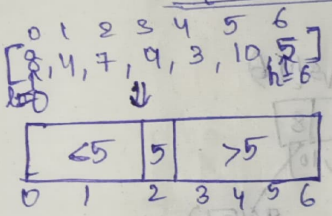
[5, 5, 5, 5]

i=1, j=2

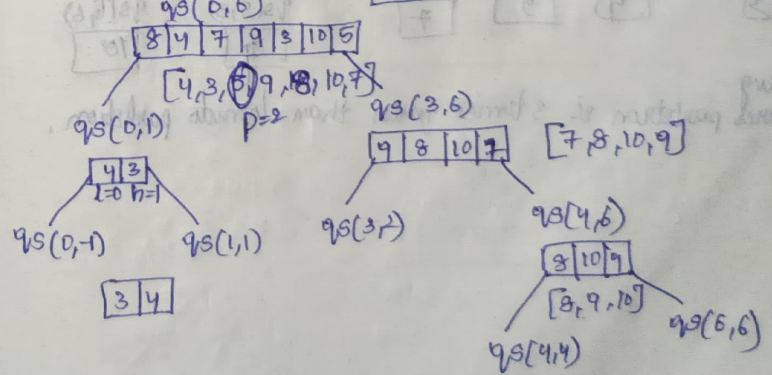
Quick Sort

1) Divide and conquer algorithm
2) worst case time : $O(n^2)$
3) Despite $O(n^2)$ worst case, it is considered faster, because of the following reasons.
   a) Inplace
   b) cache friendly.
   c) Average case is $O(n \log n)$
   d) Tail recursive
4) partition is key function (Naive, Lomuto, Hoare) — stable.

Quick sort algorithm: (using lomuto partition)

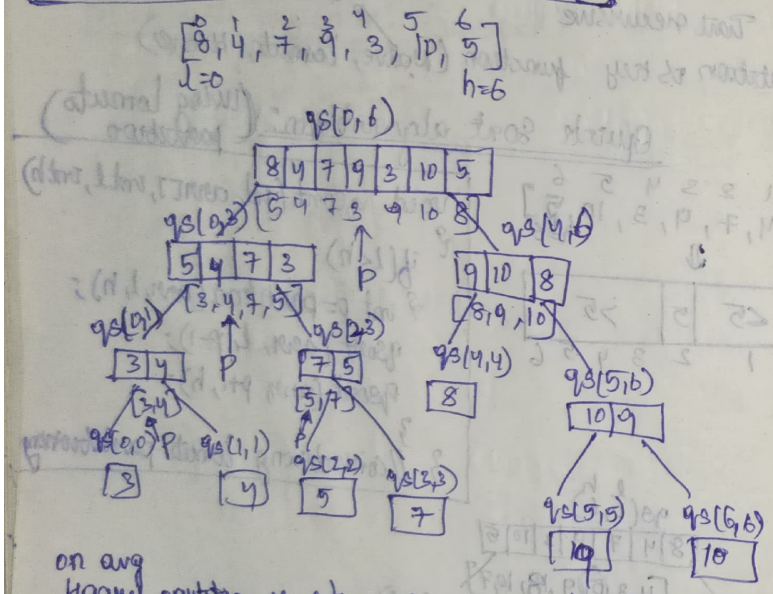



```
void qsort(int arr[], int l, int h)
{
  if(l<h)
  { int p=partition(arr, l, h);
    qsort(arr, l, p-1);
    qsort(arr, p+1, h);
  }
} // considering lomuto partition
```



QuickSort Using Hoare partition

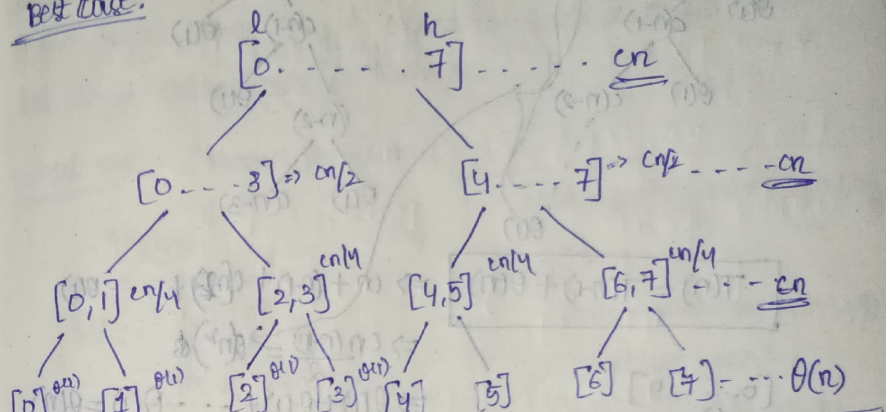
```
void qsort(int arr[], int l, int h)
{
  if(l<h)
  { int p = partition(arr, l, h);
    qsort(arr, l, p);
    qsort(arr, p+1, h);
  }
} // considering Hoares partitioning.
```



on avg
Hoares partition is 3 times faster than Lomuto partition.

Analysis of QuickSort

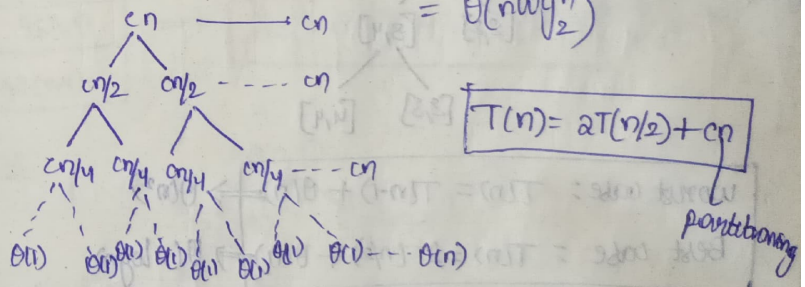
Best case:



$cn + cn + cn + \theta(n)$

$cn \times \log_2 n + \theta(n)$

$= \theta(n \log_2 n)$



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

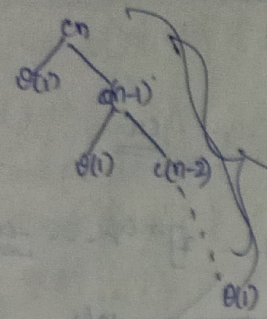
partitioning

$cn + cn + cn \ldots cn + \theta(n)$

$cn * \log_2 n + \theta(n)$

$= \boxed{\theta(n \log n)}$

worst case:

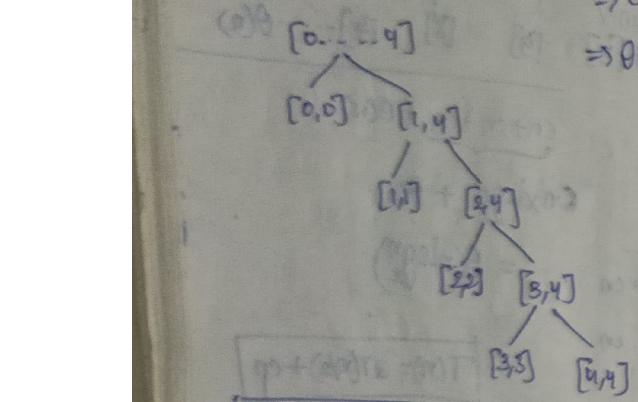


$T(n) = T(n-1) + \theta(n)$ $\Rightarrow cn + c(n-1) + c(n-2) + \cdots \theta(1)$

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

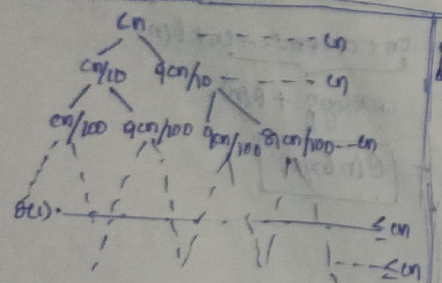
$\Rightarrow \theta(1) + \theta(1) + \cdots \theta(1) = \theta(n)$

$\Rightarrow \theta(n^2) + \theta(n) = \theta(n^2)$



Worst case: $T(n) = T(n-1) + \theta(n) \Rightarrow \theta(n^2)$

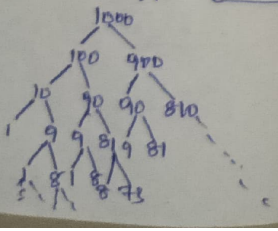
Best case: $T(n) = 2T(n/2) + \theta(n) \Rightarrow \theta(n \log n)$



Deepest Level $\Rightarrow \log_{10/9} n$

$cn + cn + \cdots$

$cn * \log_{10/9} n = O(n \log n)$

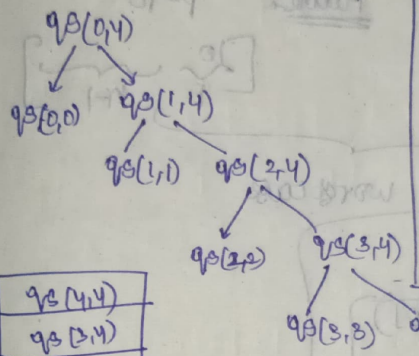


Space Analysis of QuickSort:

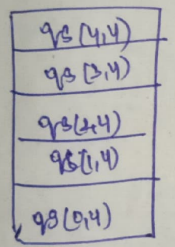
Inplace Algorithm:

Here input array is not copied to another extra array. but it uses extra space for recursion call stacks.

worst case: (Honors particular)

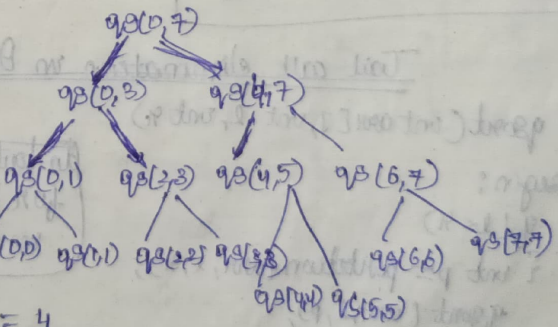


```
void qsort(int arr[], int l, int h)
{
  if(l<h)
  {
    int p = partition(arr, l, h);
    quicksort(arr, l, p);
    quicksort(arr, p+1, h);
  }
}
```



$\Rightarrow$ A.S worst case: $\theta(n)$ → No of elements in the array

Best case:



$\log_2^3 + 1 = 4$

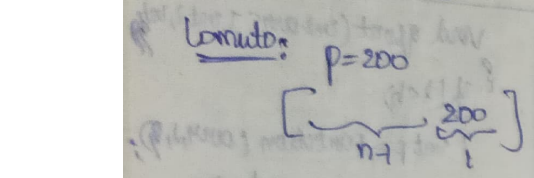
Best case: $\theta(\log n)$

choice of pivot and worst case of quick sort:

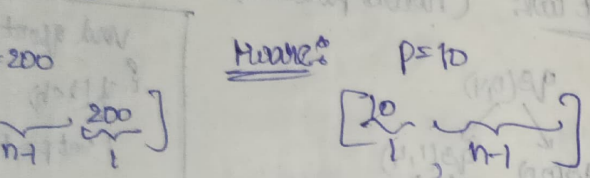
If the array is sorted in ↑/↓ with the partitioning schemes (Lomuto / Hoares) they will turn into worst case

[10, 50, 100, 200]

Lomuto: p=200



Hoare: p=10



worst case

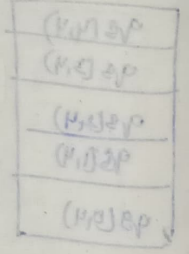
p = random(l, r)

Swap (arr[l], arr[p])

~~Lomuto~~ Hoare

Swap (arr[r], arr[h])

Lomuto



Tail call elimination in Quick sort:

```
void qsort(int arr[], int l, int r)
{ Begin:
    if(l<r)
    { int p = partition(arr, l, r);
      qsort(arr, l, p);
      l = p+1;
      goto Begin;
    }
}
```

do tail call elimination for longer part to modify extra space

Kth smallest element

I/P: arr = [10, 5, 30, 12] k = 2

O/P: 10

I/P: arr = [50, 20, 5, 10, 8] k = 4

O/P: 20

k ≤ size of the array

T.C = O(n log n)

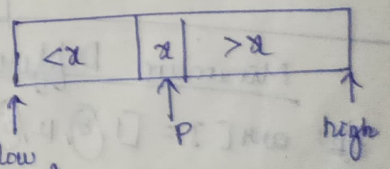
A.S is decided on modifications required for array

Naive solution:

```
int kthSmallest(int arr[], int n, int k)
{ sort(arr, arr+n);
  return arr[k-1];
}
```

[10, 3, 5, 20]
k = 2
After sorting:
[3, 5, 10, 20]
arr[k-1] = 5

Idea for optimized solution:



① partition the array around a pivot

p = partition(arr, low, high)

② If p == k-1 p=1, k=2
↳ return arr[p]

else if (p > k-1) p=3, k=2
↳ high = p-1

else (p < k-1) p=2, k=4
↳ low = p+1

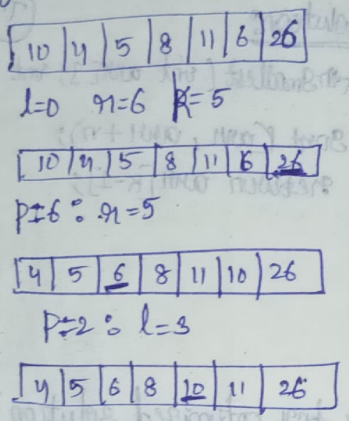
T.C = Same as quick sort.

```
int kthSmallest(int arr[], int n, int k)
{
    int l=0, r=n-1;
    while (l<=r)
    {
        int p = partition(arr, l, r);
        if (p == k-1)
            return p;
        else if (p > k-1)
            r = p-1;
        else
            l = p+1;
    }
    return -1;
}
```

→ Lomuto

T.C: worst case: $O(n^2)$
Best & Avg: $O(n \log n)$

we always pick the last element of current subarray as pivot



Minimum Difference in an Array:

I/p: arr[] = [1, 8, 4, 5, 18]
O/p: 3

I/p: arr[] = [8, 15]
O/p: 7

I/p: arr[] = [8, -1, 0, 3]
O/p: 1

I/p: arr[] = [10]
O/p: INF.

```
int getMinDiff(int arr[], int n)
{
    int res = INF;
    for (int i=1; i<n; i++)
        for (int j=0; j<i; j++)
            res = min(res, abs(arr[i]-arr[j]));
    return res;
}
```

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

arr[] = [5, 3, 8]
res = INF
i=1: j=0
res = min(INF, abs(3-5))
= 2
i=2: j=0
res = min(2, abs(8-5))
= 2
j=1
res = min(2, abs(8-3))
= 2

$a_0, a_1, a_2 \ldots a_i \ldots a_j \ldots a_{n-1}$

Efficient Solution: [10, 3, 20, 12]

1. Sort the array
   [3, 10, 12, 20]
2. Compute differences b/w the adjacent elements
   10-3 = 7
   12-10 = 2
   20-12 = 8
3. And return the minimum Difference.

T.C = same as quick sort.

```
int kthSmallest(int arr[], int n, int k)
{
    int l=0, r=n-1;
    while (l<=r)
    {
        int p = partition(arr, l, r);
        if (p == k-1)
            return p;
        else if (p > k-1)
            r = p-1;
        else
            l = p+1;
    }
    return -1;
}
```

T.C: worst case: $O(n^2)$
Best & Avg: $O(n \log n)$

comute

we always pick the last element of current subarray as pivot

10	4	5	8	11	6	26

l=0 r=6 k=5

10	4	5	8	11	6	26

p=6 : r=5

4	5	6	8	11	10	26

p=2 : l=3

4	5	6	8	10	11	26

p=4 : return 10

Minimum Difference in an Array:

I/p: arr[] = [1, 8, 12, 5, 18]
o/p: 3

I/p: arr[] = [8, 15]
o/p: 7

I/p: arr[] = [8, -1, 0, 3]
o/p: 1

I/p: arr[] = [10]
o/p: INF.

```
int getMinDiff(int arr[], int n)
{ int res = INF;
  for (int i=1; i<n; i++)
    for (int j=0; j<i; j++)
      res = min(res, abs(arr[i] - arr[j]));
  return res;
}
```

T.C = $\theta(n^2)$

$a_0, a_1, a_2 \ldots a_j \ldots a_i \ldots a_{n-1}$

arr[] = [5, 3, 8]
res = INF
i=1: j=0
res = min(INF, abs(3-5))
= 2

i=2: j=0
res = min(2, abs(8-5))
= 2

j=1
res = min(2, abs(8-3))
= 2

Efficient solution: [10, 3, 20, 12]

1. Sort the array
   [3, 10, 12, 20]
2. compute differences b/w the adjacent elements
   10-3 = 7
   12-10 = 2
   20-12 = 8
3. And return the minimum Difference.

Efficient Solution:

```
int getMinDiff(int arr[], int n)
{
    sort(arr, arr+n);          ⟹ O(nlogn)
    int res = INF; // INT_MAX
    for(int i=1; i<n; i++)
        res = min(res, a[i]-a[i-1]);   ⟹ θ(n)
    return res;
}
```

arr[] = [10, 8, 1, 4] Tc = O(nlogn)

After sorting:

arr[] = [1, 4, 8, 10]

i=1 : res = min(INF, 4-1)
= 3

i=2 : res = min(3, 8-4)
= 3

i=3 : res = min(3, 10-8)
= 2

Chocolate distribution problem:- (Fair distribution)

Ip: arr[] = [7, 3, 2, 4, 9, 12, 56]
m = 3

Op: a // we pick 2, 3, 4 ⟹ (4-2) = 2
(min chocolates does a child gets & max chocolates does a child gets) ⟹ diff remain.

Ip: arr[] = [3, 4, 1, 9, 56, 7, 9, 12]
m = 5

o/p: 6 // we pick 3, 4, 7, 9, 9.

Idea for the solution:

[7, 3, 2, 4, 9, 12, 56]
m = 3

① Consider every item as minimum one by one.

[7, 9, 12] ⟹ 5

[3, 4, 7] ⟹ 4

[3, 3, 4] ⟹ 2

[4, 7, 9] ⟹ 5

[9, 12, 56] ⟹ 47

② minimum of all the values is 2

```
int minDiff(int arr[], int n, int m)
{
    if(m>n) return -1;
    sort(arr, arr+n);
    int res = arr[m-1] - arr[0];
    for(int i=1; (i+m-1)<n; i++)
        res = min(res, arr[i+m-1] - arr[i]);
    return res;
}
```

T.C = O(nlogn)

a0, a1, a2 ..... ai ..... a(i+m-1) an-1

arr[] = [7, 3, 1, 8, 9, 12, 56]
m = 3;
After sorting
arr[] = [1, 3, 7, 8, 9, 12, 56]
res = 7-1 = 6
i=1 : res = 5
i=2 : res = 2
i=3 : - - -
i=4 : - - -

Sort an array with Two Types

① Segregate positive and negative

I/p: arr[] = [15, -3, -2, 18]

O/p: arr[] = [-3, -2, 15, 18]

② Segregate even and Odd

I/p: arr[] = [15, 14, 13, 12]

O/p: arr[] = [14, 12, 15, 13]

③ Sort a Binary Array

I/p: arr[] = [0, 1, 1, 1, 1, 0]

O/p: arr[] = [0, 0, 1, 1, 1]

Naive solution:

```
Void segregatePosNeg(int arr[], int n)
{ int temp[n], i=0;
  for (int j=0; j<n; j++)
      if (arr[j]<0)
      { temp[i] = arr[j];
        i++;
      }
  for (int j=0; j<n; j++)
      if (arr[j]>=0)
      { temp[i] = arr[j];
        i++;
      }
  for (int i=0; i<n; i++)
      arr[i] = temp[i];
}
```

arr[] = [13, -12, 18, -10]

After I st loop:
temp[] = [-12, -10, _, _]

After II nd loop:
temp[] = [-12, -10, 13, 18]

After III rd loop:
arr[] = [-12, -10, 13, 18]

T.C = Θ(n)
A.S = Θ(n)
} three traversals

Efficient solution:

Idea: This problem is mainly a variation of partition of quickSort
↳ Hoares (or) Lomuto partition can solve this in Θ(n) time and Θ(1) Aux-space

```
Void segPosNeg(int arr[], int n)
{ int i=-1, j=n;
  while (true)
  { do
    { i++;
    } while (arr[i]<0);

    do
    { j--;
    } while (arr[j]>=0);

    if (i>=j) return;

    swap(arr[i], arr[j]);
  }
}
```

arr[] = [-12, 18, -10, 15]
i=-1, j=4

1st Iteration: i=1, j=2
arr[] = [-12, -10, 18, 15]

IInd Iteration: i=2, j=1

Time: Θ(n)
A.S = Θ(1)
} Single traversal

Sort an array with three types

① Sort an array of 0's, 1's and 2's

I/p: arr[] = [0, 1, 0, 2, 1, 2]

O/p: arr[] = [0, 0, 1, 1, 2, 2]

② three way partitioning

I/p: [2, 1, 2, 20, 10, 20, 1] pivot = 2

O/p: [1, 1, 2, 2, 20, 10, 20]

③ partition around a range.

I/p: [10, 5, 6, 3, 20, 9, 40], range = [5, 10]

O/p: [3, 5, 6, 9, 10, 20, 40]

Naive Solution:

```
Void sort (int arr[], int n)
{
  int temp[n], i=0;
  for(int j=0; j<n; j++)
    if (arr[j]==0){ temp[i]=arr[j]; i++;}
  for(int j=0;j<n;j++)
    if (arr[j]==1) { temp[i]=arr[j]; i++;}
  for(int j=0;j<n;j++)
    if (arr[j]==2) {temp[i]=arr[j]; i++;}
  for (int j=0; j<n;j++)
    arr[j]=temp[j];
}
```

T.C = θ(n) ⇒ four traversals
A.S = θ(n)

arr[] = [0, 1, 2, 0, 1]

After first loop: temp[] = [0, 0, —, —, —]

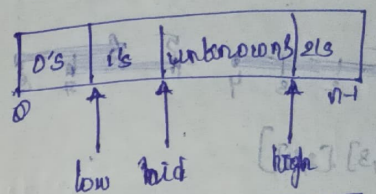
After second loop: temp[] = [0, 0, 1, 1, —]

After third loop: temp[] = [0, 0, 1, 1, 2]

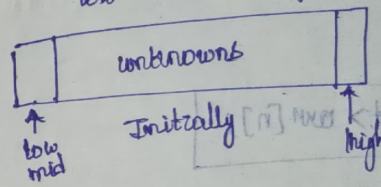
After fourth loop: arr[] = [0, 0, 1, 1, 2]

Efficient Solution:

(Dutch National Flag Algorithm)



T.C = θ(n)
A.S = θ(1)
one traversal




```
void sort (int arr[], int n)
{ int low=0, high=n-1, mid=0;
  while(mid<=high)
  { if(arr[mid]==0)
    { swap(arr[low], arr[mid]);
      low++;
      mid++;
    }
    else if (arr[mid]==1)
      mid++;
    else
    { swap(arr[mid], arr[high]);
      high--;
    }
  }
}
```

[0, 1, 2, 1, 1, 2] low=0, mid=0, hi=5

[0, 1, 2, 1, 1, 2] low=1, mid=1, hi=5

[0, 1, 2, 1, 1, 2] low=1, mid=2, hi=5

[0, 1, 2, 1, 1, 2] low=1, mid=2, high=4

[0, 1, 1, 1, 2, 2] low=1, mid=2, high=3

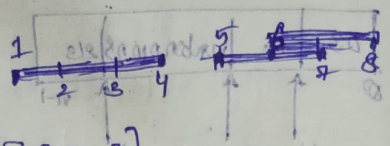
[0, 1, 1, 1, 2, 2] low=1, mid=3, high=3

mid=4, hi=3

Merge overlapping Intervals

I/p: [1,3], [2,4], [5,7], [6,8]

O/p: [[1,4], [5,8]]



I/p: [[7,9], [6,10], [4,5], [1,3] [2,4]]

O/p: [[1,5], [6,10]]

pair<int, int> arr[n];

```
struct Interval{
    int start;
    int end;
};
Interval arr[];
```

How to check if two intervals overlap?

(Smaller of end values) and check in other interval

$i_1 = [5,10]$
$i_2 = [1,7]$

larger of start values check that value doesn't lie in the second interval (other)
means overlap

max(5,1) = 5, it lies b/w i_2 so overlap

$i_1 = [10,20]$ $i_2 = [5,15]$

$i_1 = [10,20]$ $i_2 = [100,200]$

Naive solution: O(n³)

[5,10] [2,7]

start = min(i1.start, i2.start);
end = max(i1.end, i2.end);

overlaps

[[5,10], [2,3], [6,8], [1,7]]

if an interval overlaps with other interval we merge the interval and delete the other interval.

merge interval

[[5,10], [2,3]]

[[1,10], [2,3]]

[1,10]

Tc = n² for to every other interval for an interval
deleting an interval from the list → n
Sc = O(n²) * O(n) = O(n³)

Efficient solution:

sort by start time in increasing order

sort by end time in decreasing order (on)

I/p: sorted on increased order of start time: $x_1, x_2, x_3 \dots x_{i-1}, x_i$

O/p: $m_1, m_2 \dots m_{j-1}, m_j$

only merge with m_j

merged from x_1 to x_{i-1}

Non overlapping Intervals

$m_{j-1} = [5,10]$
$m_j = [11,20]$
$x_i = [15,30]$

merging m_j = [11,30]

x_i.start >= m_j.start
m_{j-1}.end < m_j.start

x_i.start > m_{j-1}.end

Efficient solution: T.C: O(nlogn) A.S: O(1)

```
struct Interval{
  int st, end;
};
bool mycmp(Interval a, Interval b)
{ return a.st < b.st; }
void mergeIntervals(Interval arr[], int n)
{
  sort(arr, arr+n, mycmp);
  int res = 0;
  for (int i=1; i<n; i++)
  {
    if (arr[res].end >= arr[i].st)
    {
      arr[res].end = max(arr[res].end, arr[i].end);
      arr[res].st = min(arr[res].st, arr[i].st);
    }
    else
    {
      res++;
      arr[res] = arr[i];
    }
  }
  for (int i=0; i<=res; i++)
    cout << arr[i].st << " " << arr[i].end << endl;
}
```

arr[] = [[5,10],[3,5],[18,30],[2,7]]

after sorting ⇒ [[2,7],[3,15],[5,10],[18,30]]

res = 0

arr[] = [[2,5],[3,15],[5,10],[18,30]]

arr[] = [[2,15],[18,30]]

Meeting Maximum Guests (Most How many guests at most)

$0 \le arr[], dep[] \le 2359$

I/P: arr[] = [9:00, 9:40]
dep[] = [10:00, 10:30]

O/P: 2 (9:40 – 10:00)

I/P: arr[] = [800, 700, 600, 500]
dep[] = [840, 820, 830, 530]

O/P: 3 (8:00 – 8:20)

I/P: arr[] = [9:00, 9:40, 9:50, 11:00, 15:00, 18:00]
dep[] = [9:10, 12:00, 11:20, 11:30, 19:00, 20:00]

O/P: 3 (11:00 – 11:20)

Efficient solution: T.C = O(nlogn), A.S = O(1) ⇒ O(1)

```
int maxGuest(int arr[], int dep[], int n)
{
  sort(arr, arr+n);
  sort(dep, dep+n);
  int i=1, j=0, res=1, curr=1;
  while(i<n && j<n)
  {
    if(arr[i] <= dep[j])
    {
      curr++; i++;
    }
    else
    {
      curr--; j++;
    }
    res = max(res, curr);
  }
  return res;
}
```

O(n+n)

arr[] = [900, 600, 700]
dep[] = [1000, 800, 730]

after sorting

600	A	1
700	A	2
730	D	1
800	D	0
900	A	1
1000	D	0

After sorting: arr[] = [600, 700, 900]
dep[] = [730, 800, 1000]

Initially: i=1, j=0, curr=1, res=1

arr[i] <= dep[j]
i=2, j=0, curr=2, res=2

arr[i] > dep[j]
i=2, j=1, curr=1, res=2

arr[i] > dep[j]
i=2, j=2, curr=0, res=2

arr[i] <= dep[j]
i=3, j=2, curr=1, res=2

cycle sort

① A worst case $O(n^2)$ sorting algorithm.
② Does minimum memory writes and can be useful for cases where memory write is costly
③ Inplace and not stable.
④ Useful to solve questions like find minimum swaps required to sort an array

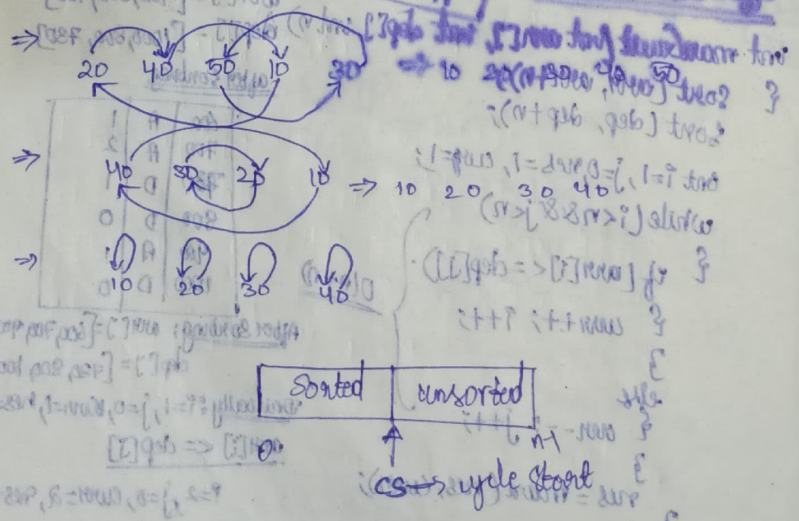
→ Count how many elements are smaller than the current element for every element go to that count position & fix that number. & take item as that position no.

item=20
20 40 50 10 30
10 20 40

item=40
item=10
item=20

10 20 50 40 30
item=50 30 50
item=30
item=50

10 20 30 40 50




```
void cycleSortDistinct(int arr[], int n)
{
    for(int cs=0; cs<n-1; cs++)
    {
        int item = arr[cs];
        int pos = cs;
        for(int i=cs+1; i<n; i++)
            if(arr[i] < item)
                pos++;
        swap(item, arr[pos]);
        while(pos != cs)
        {
            pos = cs;
            for(int i=cs+1; i<n; i++)
                if(arr[i] < item)
                    pos++;
            swap(item, arr[pos]);
        }
    }
}
```

exercise: do cycle sort for duplicates
② Modify cycle sort so that it counts min swaps required to sort an array

cs=0
20 40 50 10 30
item=20 pos=0 → swapped.

pos=1
item=40
20 20 50 40 30

pos=3
20 20 50 40 30
item=10
pos=0

10 20 50 40 30
item=20, cs=1, pos=1

10 20 50 40 30
cs=2, pos=2, item=50
pos=4

10 20 50 40 50
cs=2, item=30
pos=2

item=50
10 20 30 40 50
cs=3
cs=4

T.C = O(nlogn) ⇒ Best, Avg, Worst

Heap sort:

⇒ It is based on selection sort. In selection sort for finding min we use linear search instead of this we use heap data structure

ip: arr[] = [10, 15, 50, 4, 20]

op: arr[] = [4, 10, 15, 20, 50]

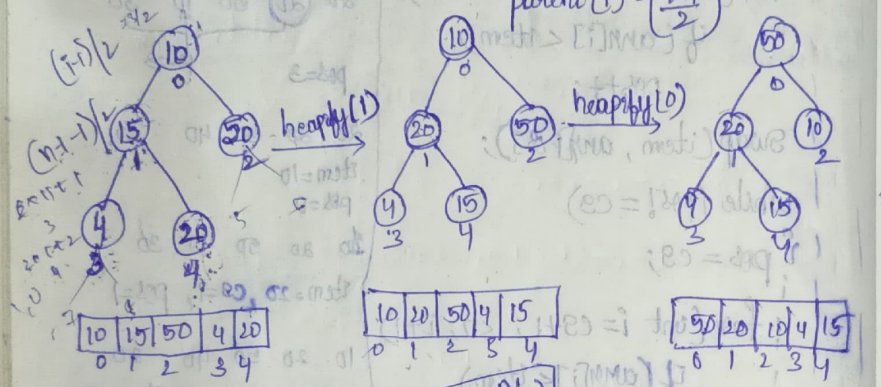
max Heap ⇒ sort in ↑ order

Min Heap ⇒ sort in ↓ order

step-1: Build a max Heap

parent(i) = (i-1)/2

(i-1)/2 (n-1-1)/2

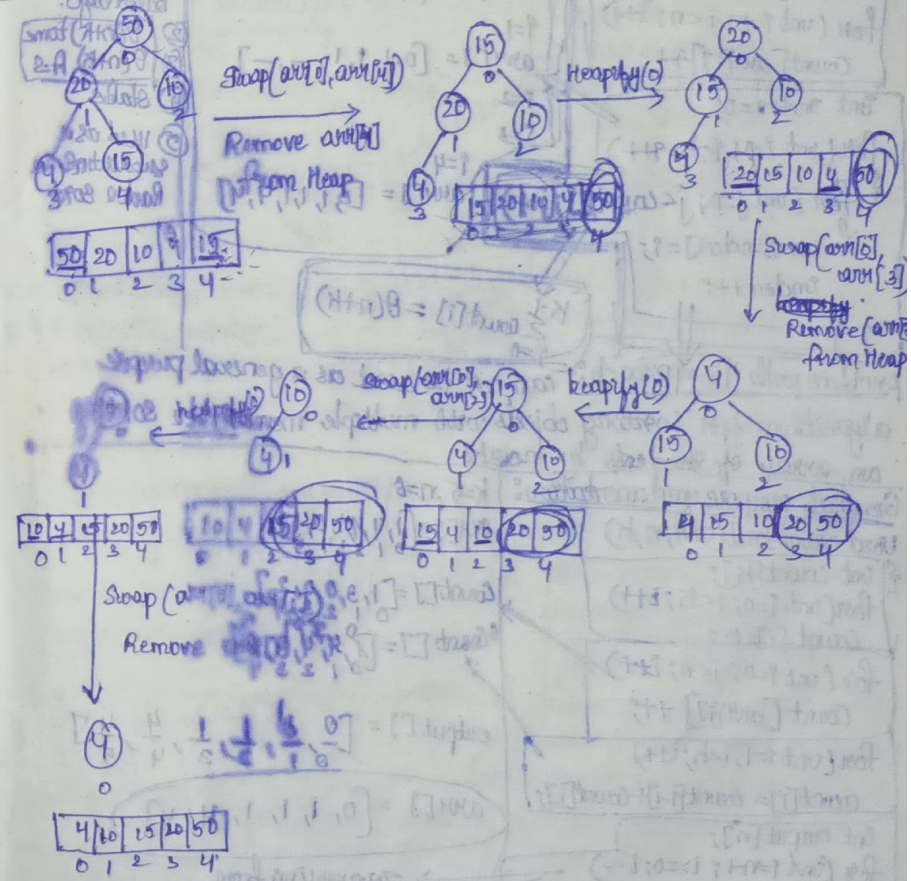



```
void buildHeap(int arr[], int n) ⇒ TC = O(n)
{
  for(int i = (n-2)/2; i >= 0; i--)   // parent of last node = (n-1-1)/2 = (n-2)/2
    maxHeapify(arr, n, i);   // initially start with internal node
}
Void maxHeapify(int arr[], int n, int i)
{ int largest = i; left = 2*i+1, right = 2*i+2;
  if(left < n && arr[left] > arr[largest]) largest = left;
  if(right < n && arr[right] > arr[largest]) largest = right;
  if(largest != i)   // 1 → (4 != 1)
  { swap(arr[largest], arr[i]);
    maxHeapify(arr, n, largest); // call maxHeapify for swapped child.
  }
}
```

step-2: Repeatedly Swap root with last node, reduce heap size by one and heapify.

```
void HeapSort(int arr[], int n)
{ buildHeap(arr, n);
  for(int i = n-1; i >= 1; i--)
  { swap(arr[0], arr[i]);
    maxheapify(arr, i, 0);
  }
}
```

In practice constants are more hidden in Heap Sort. So we prefer merge sort & quick sort.



for k values → n, 2n, ... : Counting sort (used when I/p elements in small ranges)
not for n log n → Radix sort

O(n+k) time to sort n elements in range from 0 to k-1

I/p: arr[] = [1, 4, 4, 1, 0, 1]
k = 5
O/p: arr[] = [0, 1, 1, 1, 4, 4]

I/p: arr[] = [2, 1, 8, 9, 4]
k = 10
O/p: arr[] = [1, 2, 4, 8, 9]

Naive Implementation

```
void countSort(arr, n, k)
{
    int count[k];                    θ(k)
    for(int i=0; i<k; i++)
        count[i] = 0;
    for(int i=0; i<n; i++)           θ(n)
        count[arr[i]]++;
    int index = 0;
    for(int i=0; i<k; i++)
    {
        for(int j=0; j<count[i]; j++)
        {
            arr[index] = i;
            index++;
        }
    }
}
```

arr[] = [1, 4, 4, 1, 0, 1]
k = 5
count[] = [1, 3, 0, 0, 2]
0 1 2 3 4

$\sum_{i=0}^{k-1} count[i] = \theta(n+k)$

i=0 arr[] = [0, -, -, -, -, -]
i=1 arr[] = [0, 1, 1, 1, -, -]
i=4 arr[] = [0, 1, 1, 1, 4, 4]

$T[n] = \theta(n+k)$

Drawbacks
1. Not a comparison based algo.
2. θ(n+k) time
3. θ(n+k) A.S
4. Stable
5. used as a subroutine in Radix sort

problem with this approach: can not be used as a general purpose algorithm for sorting objects with multiple members like sorting an array of students by marks

General purpose implementation:

```
void countSort(arr, n, k)
{
    int count[k];
    for(int i=0; i<k; i++)
        count[i] = 0;
    for(int i=0; i<n; i++)
        count[arr[i]]++;
    for(int i=1; i<k; i++)
        count[i] = count[i-1] + count[i];
    int output[n];
    for(int i=n-1; i>=0; i--)
    {
        output[count[arr[i]]-1] = arr[i];
        count[arr[i]]--;
    }
    for(int i=0; i<n; i++)
        arr[i] = output[i];
}
```

k=5, n=6
arr[] = [1, 4, 4, 1, 0, 1]
0 1 2 3 4 5
count[] = [1, 3, 0, 0, 2]
0 1 2 3 4
count[] = [1, 4, 4, 4, 6]
0 1 2 3 4

output[] = [0/0, 1/1, 1/2, 1/3, 4/4, 4/5]

arr[] = [0, 1, 1, 1, 4, 4]

traversing from right to make stable

eg: extend the given implementation to work for an arbitrary range of size m

Radix Sort

319, 212, 6, 8, 100, 50

→ Recording numbers with leading zeros
319, 212, 006, 008, 100, 050

→ Stable sort according the last digit (last significant digit)
100, 050, 212, 006, 008, 319

→ Stable sort according to the middle digit
100, 006, 008, 212, 319, 050

→ Stable sort according to the most significant digit.
006, 008, 050, 100, 212, 319

① Stable
②

```
void radixSort(int arr[], int n)
{
    int mx = arr[0];
    for (int i=1; i<n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    for (int exp=1; mx/exp>0; exp=exp*10)
        countingSort(arr, n, exp);
}
void countingSort(int arr[], int n, int exp)
{ int count[10], output[n];
  for (int i=0;i<10;i++){ count[i]=0; }
  for (int i=0; i<n; i++){ count[(arr[i]/exp)%10]++; }
  for (int i=0; i<10; i++)
  { count[i] = count[i] + count[i-1]; }
  for (i=n-1; i>=0; i--)
  { output[count[(arr[i]/exp)%10]-1] = arr[i];
    count[(arr[i]/exp)%10]--; }
  for (int i=0; i<n; i++){ arr[i]=output[i]; }
}
```

b → θ(n)

d * (n+b) → θ(d * (n+b))

b=10

k=b=10

0 to 9

TC = θ(d * (n+b)) ⇒ If we ↑ value of b, then our d will decrease so our TC ↓ & extra space ↑

SC = θ(n+b)

arr[] = [319, 212, 6, 8, 100, 50]
mx = 319
countingSort(arr, 6, 1)
countingSort(arr, 6, 10)
countingSort(arr, 6, 100)

countingSort(arr, 6, 1); index = (arr[i]/1)%10

count[10] = [2, 0, 1, 0, 0, 0, 1, 0, 1, 1]
count[10] = [2, 2, 3, 3, 3, 3, 4, 4, 5, 6]
arr[] = output[] = [100, 50, 212, 6, 8, 319]

countingSort(arr, 6, 10); index = (arr[i]/10)%10

count[10] = [3, 2, 0, 0, 0, 1, 0, 0, 0, 0]
count[10] = [3, 5, 5, 5, 5, 6, 6, 6, 6, 6]
arr[] = output[] = [100, 6, 8, 212, 319, 50]

countingSort(arr, 6, 100); index = (arr[i]/100)%10

count[10] = [3, 1, 1, 1, 0, 0, 0, 0, 0, 0]
count[10] = [3, 4, 5, 6, 6, 6, 6, 6, 6, 6]
arr[] = output[] = [6, 8, 50, 100, 212, 319]

Bucket Sort

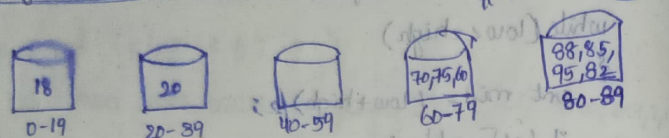
① Consider a situation where we have numbers uniformly distributed in range from 1 to 10^8. How do we sort efficiently.

② Consider another situation where we have floating point numbers uniformly distributed in range from 0.0 to 1.0

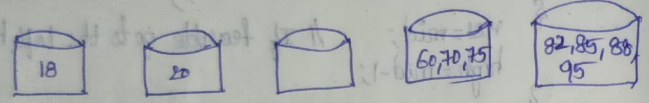
I/p: 20, 88, 70, 85, 75, 95, 18, 82, 60
Range: 0 to 99

n ⇒ elements
k ⇒ buckets
n/k

Step 1: Scatter



Step 2: Sort Buckets



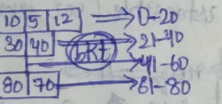
Step 3: Join Sorted Buckets ⇒ 18, 20, 60, 70, 75, 82, 85, 88

I/p: arr[] = [20, 80, 10, 85, 75, 99, 18]
K=5
o/p: arr[] = [10, 18, 20, 75, 80, 85, 99]

I/p: arr[] = [20, 80, 40, 30, 70]
K=4
o/p: arr[] = [20, 30, 40, 70, 80]

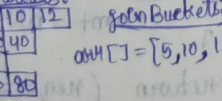
```
void BucketSort(int arr[], int n, int k)
{ int max_val = arr[0];                    // find max value
  for(int i=1; i<n; i++)
    max_val = max(max_val, arr[i]);
  max_val++;
  vector<int> bkt[k];                      // for buckets
  for (int i=0; i<n; i++)
  { int bi = (k*arr[i])/max_val;           // bi = bucket index
    bkt[bi].push_back(arr[i]);
  }
  for (int i=0; i<k; i++)                  // Sort Buckets
    sort(bkt[i].begin(), bkt[i].end());
  int index=0;                             // Join Buckets
  for (int i=0; i<k; i++)
    for (int j=0; j<bkt[i].size(); j++)
      arr[index++] = bkt[i][j];
}
```

[30, 40, 10, 80, 5, 12, 70]
k=4



max_val = 81
i=0: bi = 4*30/81 = 1
i=1: bi = 4*40/81 = 1
i=2: bi = 4*10/81 = 0
i=3: bi = 4*80/81 = 3 ⇒ If we donot ↑ max_val 4*80/80 = 4 which is out of bounds
i=4: bi = 4*5/81 = 0
i=5: bi = 4*12/81 = 0
i=6: bi = 4*70/81 = 3

Sort Buckets:



Join Buckets:
arr[] = [5, 10, 12, 30, 40, 70, 80]

Time Complexity:

Assuming k ≈ n

Best case: Data is uniformly Distributed
O(n) ⇒ use Insertion sort

Worst case: All items go into a single bucket
If we use Insertion sort to sort the individual buckets, then O(n²).
If we use O(n log n) algorithm to sort the individual buckets, then O(n log n).