SORTING

- **Sort in C++** → sort is a general purpose library function used for sorting elements
  → used in containers which allow random access.

Introsort
- quicksort
- Heapsort
- Insertion sort

- **sort in array**

```
sort(arr, arr+n)
```

first address — address of element just after last element

- By default it sorts in increasing order
- * we can pass 3 parameters to specify the ordering in sorting

↓

```
sort(a, a+n, greater<int>)
```

Reverses the order i.e descending order.

- **sort in vector**

```
sort(v.begin(), v.end());
```

Program where we specify our own order :-

```
#include <iostream>
#include <algorithm>
using namespace std;

struct Point {
    int x, y;
};
bool myCmp(Point p1, Point p2) {
    return (p1.x < p2.x);
}
int main(){
    Point arr[] = {{3,10},{2,8},{5,4}}
    sort(arr, arr+n, myCmp);
}
```

Stability in sorting Algorithms

If two elements have the same value then they should appear in the same order in the output sorted array as they were in input array

arr → {9, 2, 2, 8, 4}

O/P → {2, 2, 4, 8, 9} ⟺ {2, 2, 4, 8, 9}

Stable — not - stable

Or we can visualise as

arr → {("Anil", 50), ("Ayan", 80), ("Pujush", 50), ("Ramesh"; 80)}.

↓

sort on basis of Marks

{("Anil", 50), ("Pujush", 50), ("Ayan", 80), ("Ramesh", 80))

Stable

{("Pujush", 50), ("Anil", 50, - - -)}.

unstable

- stability is important and significant only when the objects have multiple fields.

Stable sortings → Bubble sort, Insertion Sort, Merge sort

Unstable → Selection sort, Quick sort, Heapsort.

Insertion sort

- best performance when ~~n~~ input array size. is small.
- Hybrid algorithms - tim sort and introsort uses insertion sort

Algorithm

- start with 2nd element and maintain elements from 0 to i-1 are sorted.
- on each ~~e~~ iteration we look for the correct positions of i^{th} element and ~~swap~~ ~~it~~ ~~with~~ put it there by shifting all elements one position ahead.

50 20 40 60 10

- start with $i = 1$

~~20 is greater than 5.~~ $20 < 50$

hence 50 50 40 60 10 key = 20

20 50 40 60 10

- at $i = 2$

40 is placed between 20 and 50.

- at $i = 3$.

60 is not required to change its position

- at $i = 4$

10 will be put in front.

```
void isort (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; }.
```

storing the value of a[i] if it needs to be shifted

check if elements at left are greater

Time complexity

(a) **Best case**

elements are sorted, it will never go into the inner loop. → O(n)

(b) **Worst case**

array is reverse sorted → it will always go into the innerloop.

no of movements → $\frac{n(n-1)}{2}$ → $O(n^2)$.

Merge Sort

1. Divide and conquer algorithm
2. Stable algorithm
3. O(nlogn) time and O(n) extra space.
4. well suited for linked lists, works in O(1) Aux space.
5. For arrays → quicksort performs better.
6. well suited for external sorting
   ↳ bring in parts of array and then sort these parts.

Python uses a variation of merge sort called Tim sort.

Merge Two Sorted Arrays.

I/P a[] → {10, 15, 20, 40}
b[] → {5, 6, 6, 10, 15}.

O/P → {5, 6, 6, 10, 10, 15, 15, 20, 40}.

① **Naive solution**

- O(n+m) extra space
- O(n+m(log(n+m)) time complexity

copy all the elements in a separate array → sort it

Effective solution $O(m+n)$

If have two points $p_1 \to$ (for array 1) p_2 (array 2) and if we move these pointers as per cases.

- If $a[p_1] > b[p_2]$

  It means we need to print the smaller one i.e $b[p_2]$ and move the p_2 ahead (p_2++)

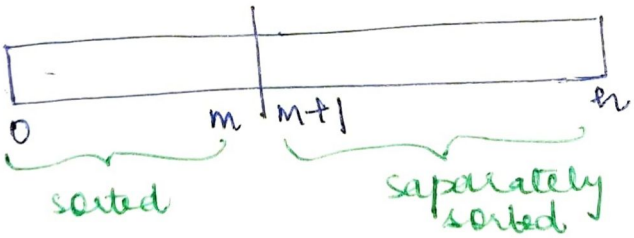
```
void merge (int a[], int b[], int m, int n) {
    int p1 = 0
    int p2 = 0
    while ( p1 < n1 && p2 < n2) {
      if ( a[p1] < b[p2]) {
            cout << b[p2];
            p2++;
      } else {
            cout << a[p1];
            p1++;
      }
    }
    while(p1< n1) { cout << a[p1++] ; }
    while (p2<n2) { cout << b[p2++] ; }
}
```

moving the pointer p_1 & p_2 as per cases stated

Merge Function of ~~per~~ merge sort

I/P a[] → {10, 15, 20, 40, 8, 11, 15, 22, 25}.

l = 0, h = 8, m = 3.



- elements from l to m are sorted
- elements from m+1 to h are sorted

- copy elements from 0 → m in a separate array 1
- copy elements from m+1 to h in a different array 2
- ~~call the sort the~~ to merge two sorted arrays by the algo done above

- size of a1[] → m - l + 1
- size of a2[] → r - m.

$l \le m < h$

```
void merge (int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int left[n1], right[n2];
```

copying elements in left array

```
    for (int i=0; i< n1 ; i++) {
        left[i] = arr[l + i];
    }
```

→ l is not always 0.

copying in right array

```
    for (int j=0 ; j<n2 ; j++) {
        right[j] = arr[m+1+j];
    }
```

merging logic & moving of ptrs

```
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]){
            arr[k++] = left[i++];
        }
        else {
            arr[k++] = right[j++];
        }
```

→ equals to ensures the ~~complexity~~ stability of merge sort

copying remaining elements

```
    while (i < n1)
        arr[k++] = left[i++];
    while (j < n2)
        arr[k++] = right[j++];
}
```

Merge Sort Algorithm

- check if array has minimum 2 elements by $(r > l)$
- find mid point of input array.
- Recursively sort the array - the two halfs.
- merge function sorts the two sorted array.

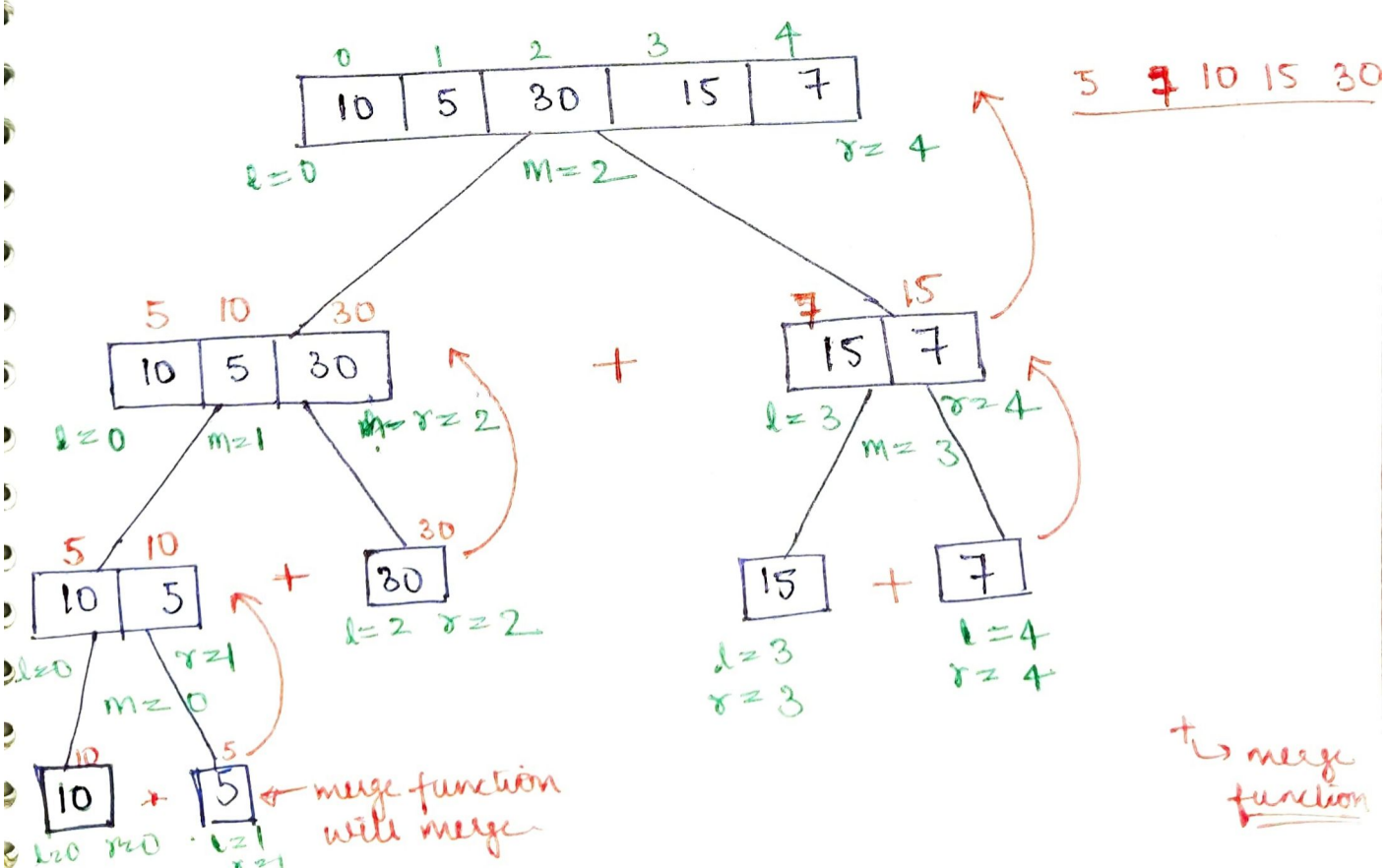
```
void mergesort (int arr[], int l, int r) {
    if (r > l) {
        int m = l + (r-l)/2;
        mergesort (arr, l, m);
        mergesort (arr, m+1, r);
        merge (arr, l, m, r);
    }
}
```

similar to $(l+r)/2$ did this in this way to avoid overflow

sorts left part of array

sorts right part of array

merge the two sorted halves.



Time complexity of Merge sort :-

$O(n) \times O(n \log n)$

- $O(n)$ → merge function
- $O(n \log n)$ → no of recursive calls.

Auxiliary space → $O(n)$

Intersection of two sorted Arrays

I/P a[] → {2, 5, 8, 13, 15}
b[] → {1, 3, 8, 15, 18, 20, 25}

* we can have duplicates also

O/P → {8, 15}.

naive solution

Traverse through both of the array and check if there is a common member.

Effective solution

$O(m+n)$ time comp, $O(1)$ Aux space

- maintain two pointers and check if

a[i] > b[j] → move j by 1 step
a[i] < b[j] → move i by 1 step
a[i] = b[j] → i++, j++, (we got the common element)

```
void findInt(int a[], int b[]){
    int i=0, j=0;
    // n1 = size of a
    // n2 = size of b
  while(i<n1 && j<n2){
    if(a[i] > b[j]){ j++; }
    if(a[i] < b[j]){ i++; }
    else if(a[i] == b[j]){
        i++; j++;
        cout << a[i];
    }
  }
}
```

* TO check for duplicates

→ else if(i>0 && a[i-1] == a[i]){ i++; }

Union of two Sorted Arrays

I/P a[] → {3, 8, 10}
b[] → {2, 8, 9, 10, 15}

O/P → 2, 3, 8, 9, 10, 15

Naive Solution

Sort the ~~arr~~ merged array using an extra array of (m+n) size and then traverse through the whole array & print distict elements only

T.C → O((n+m) log(n+m))
A.S → O(n+m)

Efficient Method

- we have the following conditions

① b[j] < a[i] → j++, print b[j]
② a[i] < b[j] → i++, print a[i]
③ a[i-1] == a[i] → i++
④ b[j-1] == b[j] → j++
⑤ b[j] == a[i] = i++, j++, print a[i].

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

Count inversion in a Array

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

@ in simple words when a greater element appear first in an array

I/P → {2, 4, 1, 3, 5}
O/P → 3 (4,1) (4,3) (2,1)

I/P → {10, 20, 30, 40}
O/P → 0 (sorted in increasing order)

I/P → {40, 30, 20, 10}
O/P → 6 sorted in reverse order
no of inversions = $\frac{n(n-1)}{2}$
$4(3)/2 = 6$

Naive solution

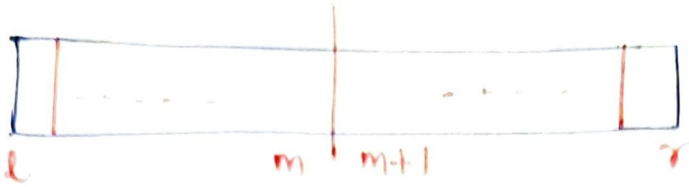
$O(n^2)$ approach → run two loops and check for elements in the right of an element & count if the smaller element is appearing in the right

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

Efficient solution

O(nlogn)

idea is to use merge sort



- possibilities for inversion pair (x, y) is $x > y$
  (a) x & y both lies in left half
  (b) x & y both lies in right half
  (c) x lies in left half & y lies in right half

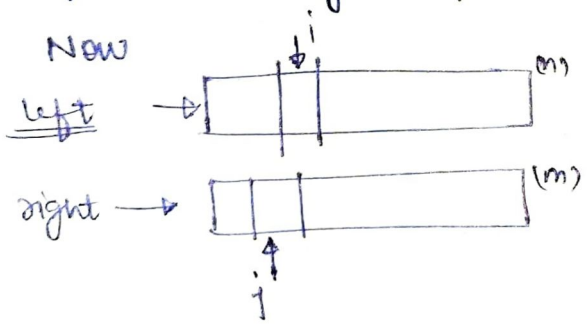
- we have not included the fourth possibility x in right and y in left because as array is sorted if x lies in right then it will not be inversion.

- Now we can solve this problem using the concept of merge function in merge sort.

During merging of array we face two situation

① left[i] < right[j] ⟶ this is not the cond^n for inversion

② left[i] > right[j] ⟶ condition for inversion.



If element in right array is ~~greater~~ smaller than left mat means it is smaller than all the remaining (n-i) elements of left array. Hence count will be increased by

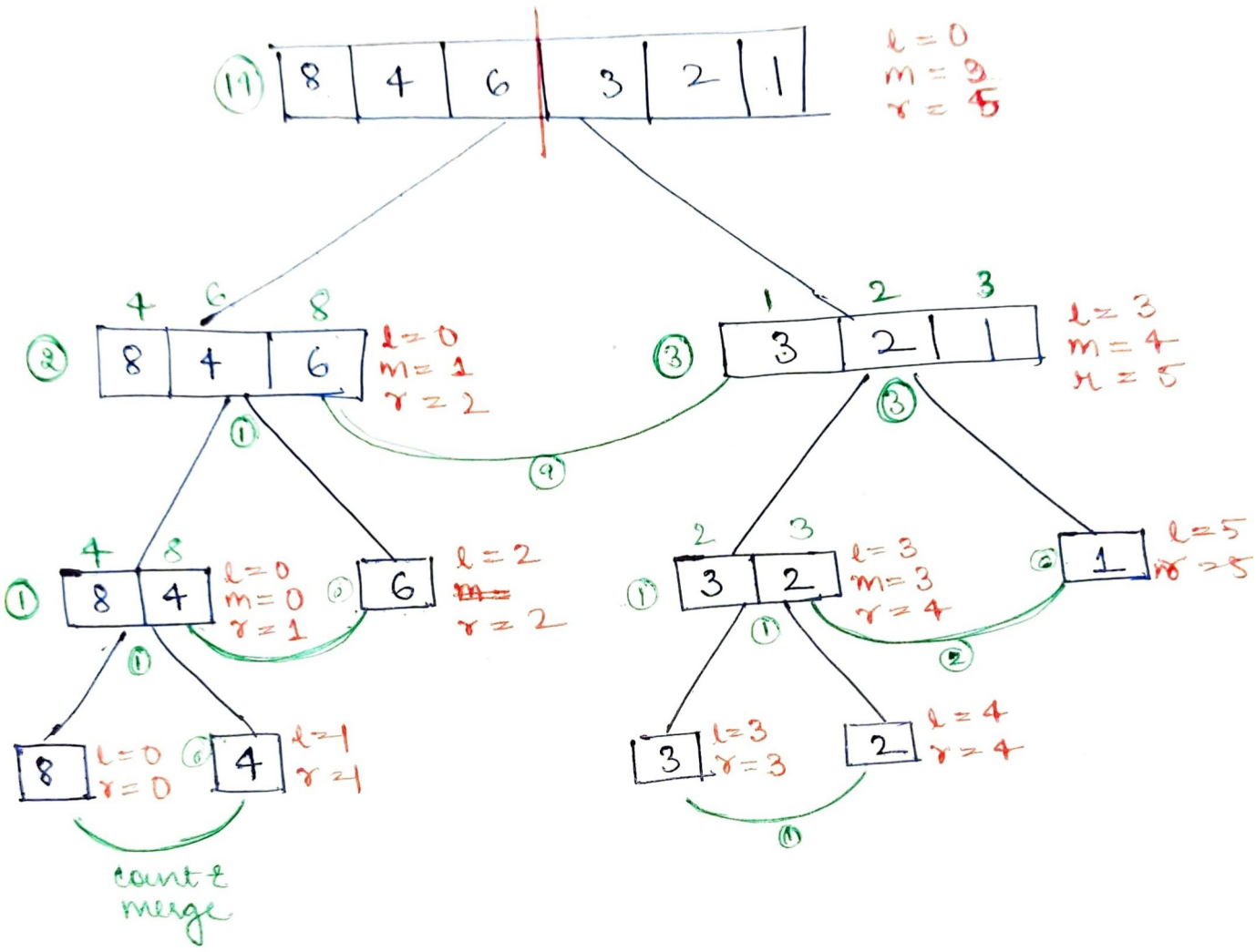
res += (n-i)

Algorithm

```
int countInv(int arr[], int l, int r) {
    int res = 0;
    if (l<r) {
        int m = l+(r-l)/2;
        res += countInv(arr, l, m);
        res += countInv(arr, m+1, r);
        res += countandMerge(arr, l, m, r)
    }
    return res;
}
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 j=0; j<n2; j++){
        right[j] = arr[m+j+1];
    }
    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 if{ right
        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;
}
```

Ex → 8, 4, 6, 3, 2, 1



Time complexity → O(nlogn) Auxiliary space - O(n)

Partition a given Array

I/P → arr - {3, 8, 6, 12, 10, 7} p = 5 (Index of element around which we have to partition)

O/P →

{3, 6, (7), 8, 10, 12}

or

{6, 3, (7), 8, 10, 12}

or

{6, 3, (7), 10, 8, 12}

- If the eq' element is equal to arr[p] it should come before it
- In short in a unsorted array, arr[p] should be at its correct position i.e at the same position as it would appear if array is sorted.

Naive solution

```
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];
}.
```

add cond^n ~~int~~ i ≠ p

index++;
temp[index] = arr[p];

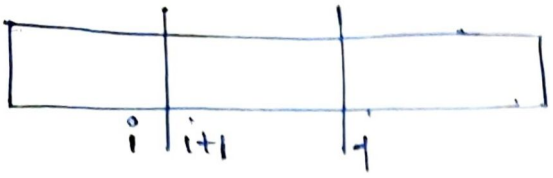
time complexity → O(n)
Auxiliary space → O(n).

Ex→ 2 3 8 4 5 6 9 2

p = 3

Lomuto Partition

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



at any point 'j' of the loop running from 0 → n-1 we maintain two things

① elements before i is smaller.
② elements after i is greater than the pivot

~~If we~~ here 'i' is variable we can change the size of window having smaller elements

so if arr[j] > arr[pivot] → we do nothing as we want that element to be at right of ~~array or~~ (i)

if arr[j] ≤ arr[pivot] → we swap the element with the element at i+1th position and then increase the size of window.

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

h = 6

j=0

(initially window is at i=0-1)

at j=0 now arr[j] (10) < arr[h] (60)

~~Spa~~ swap (i+1) [-1+1] → 0th position with jth (0)th element.

now the window size is increased by 1 i.e 0

now j=1 arr[j] = 80 → do nothing

j = 2: arr[j] = 30

swap (i+1)th i.e 1st with 2nd. → {10, 30, 80 ... }.

```
void Partition (int arr[], int l, int h) {
    int pivot = arr[h];  ——→ // always last element
    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);
}
```

Ex → {10, 80, 30, 90, 40, 50, 70}

i = -1 j = 0

swap i+1 with j | i++

i = 1 j = 1 → do nothing → {10, 80, 30, 90, 40, 50, 70}

i = 1 j = 2 → swap → {10, 30, 80, 90, 40, 50, 70}

⋮

i = 2 j = 5 → {10, 30, 40, 90, 80, 50, 70}.

↓

{10, 30, 90, 50, 80, 90, 70}

i = 3 j = 6 = h

↳ we will not do anything
we will swap (i+1) with h.

{10, 30, 90, 50, 70, 90, 80}

← smaller → greater

corner cases

① {70, 60, 80, 40, 30}. → {30, 60, 80, 40, 70}.

i = -1 j = 0

② {30, 40, 20, 50, 80}.

i = -1 j = 1 → {30, 40, 20, 50, 80}

i = 1 j = 2 → {30, 40, 20, 50, 80}

There will be 1 swap at same position and final array would be

{30, 40, 20, 50, 80} // no change

- Lumoto partition Assumes that the pivot element is the last element. If we are given a pivot index other than n-1, we will swap the p^{th} element with $n-1^{th}$ index and then apply the standard lumoto p

Time complexity → O(n) (and only 1 traversal req.)
Aux. Space → O(1)

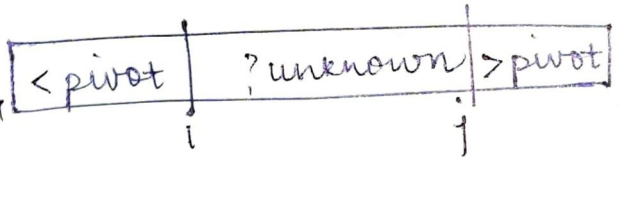
Hoare Partition

l → low
h → high

pivot element is the first element arr[l]

we have two windows

① elements on the left of i are less than pivot
② elements on the right of j are greater than pivot



initially
i = l-1
j = h+1
both are out of bound of array

i starts from low-1 & then i++

j starts from high+1 & then j--

we break the loop whenever i & j cross each other or i > j.

We move from both side and from left we search for element greater or than pivot and stop there & from left right we look for element smaller than pivot & stop there. And then swap (a[i], a[j])

```
int hPartition (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]);
    }
}
```

~~Hoare~~ Hoare's Partition does' not ensure that pivot is at it's correct position. i.e

{5, 3, 8, 4, 2, 7, 1, 10} P=0

↓

O/P → { 1, 3, 2, 4, 8, 7, 5, 10 }.

Smaller than a[0] (5)

greater than or equal to a[0] (5)

corner cases

(a) pivot is Largest

{12, 10, 5, 9} → {9, 10, 5, 12}

i = 0

{9, 10, 5, 12}

(b) all the elements are same

{5, 5, 5, 5} → {5, 5, 5, 5}

Not stable as equal elements are also swapped.

Quick sort

(a) divide and conquer algorithm

(b) the key part is partitioning → (Hoare, Lomuto, Naive)

Not stable — stable but auxiliary space

(c) worst case complexity → $O(n^2)$.

(d) Tail Recursive, cache friendly

- In quicksort the auxiliary space required is O(1) whereas it is O(n) in merge sort where we need extra array for merging.

Quicksort Algorithm

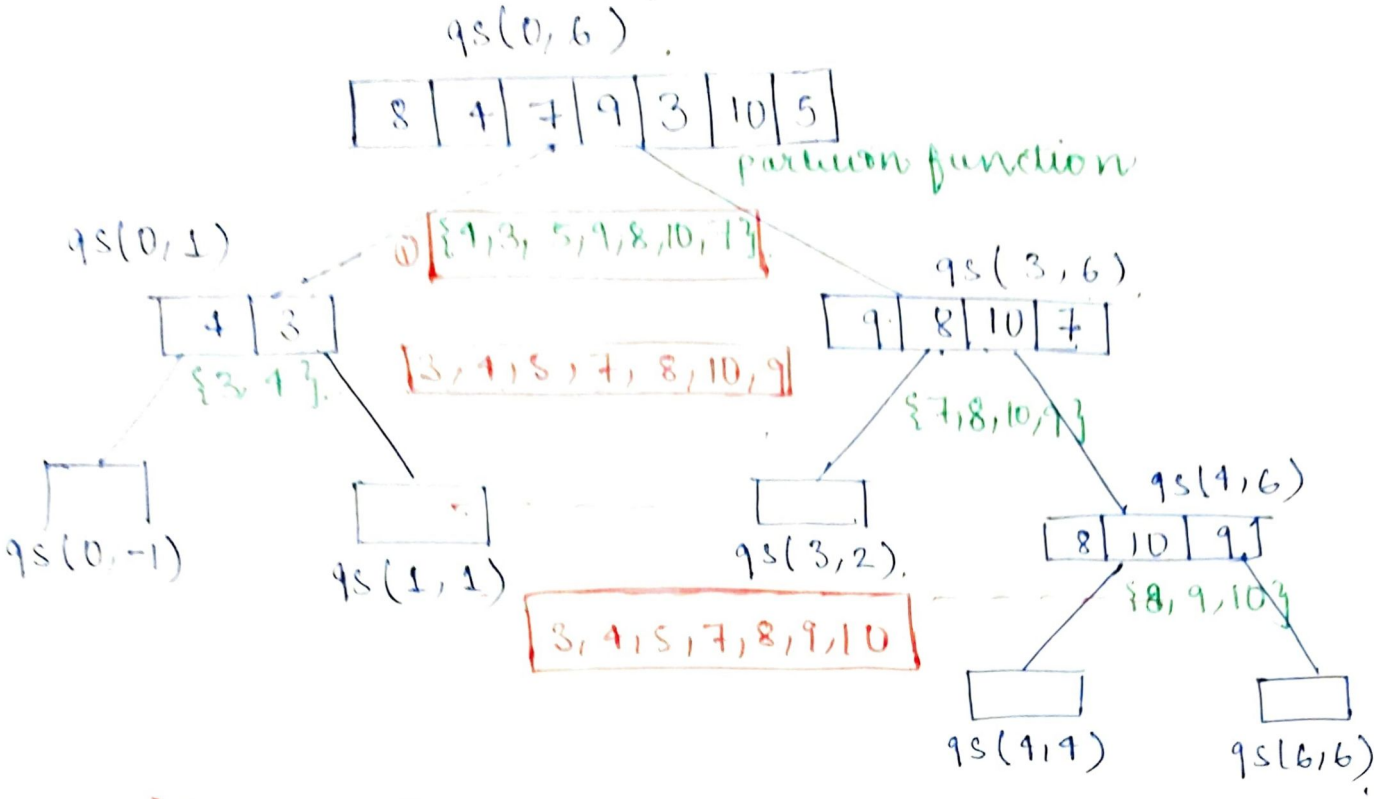
- By using Lomuto partition, it puts the last element at its correct position and then return that position
- we need to call quicksort again for left of pivot index (placed correctly). and for the right also.

{8, 4, 7, 9, 3, 10, 5}

put 5 on correct position

{ 4 3 (5) _ _ _ }

call qs ← → call qs.




```
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);
    }
}

// partition function (lumuto)
```

Using Hoare's Partition

The only difference is in hoare's partition it

① partitions the array

In hoare's partition the pivot is 1st element of array and it doesn't fix the position of pivot it partitions the array in such a way that one element ~~from~~ before index i is less than pivot & element ~~for~~ after i is greaterthan or equal to pivot

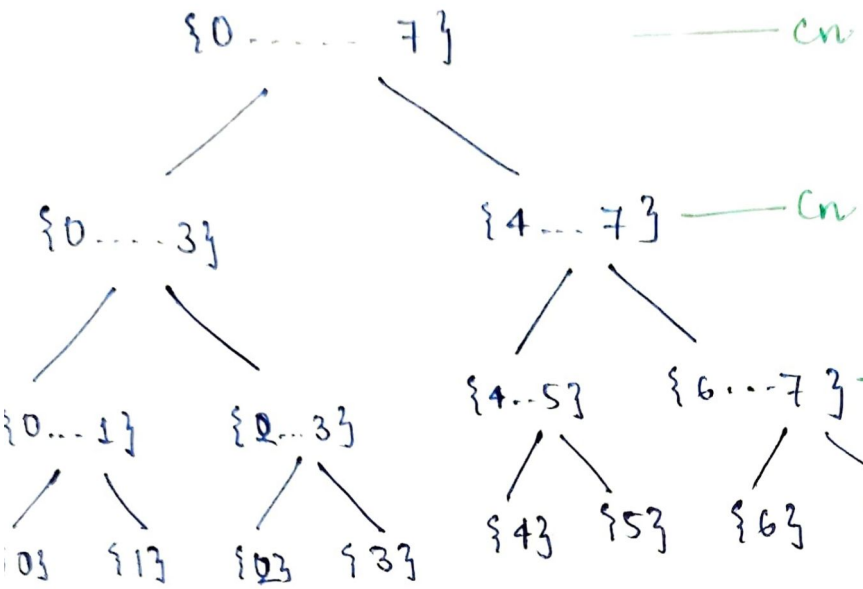
Hence the recursive calls are

```
qsort ( arr, l, p)
qsort ( arr, p+1, h).
```

Analysis of Quicksort

Best case [O(nlogn)]

- partition function divides the array into two equal halves. i.e (elements on both side are equal)



$Cn + Cn + Cn \cdots + O(n) \rightarrow Cn\log n + O(n)$

(logn times)

$= O(n\log n)$.

Algorithm

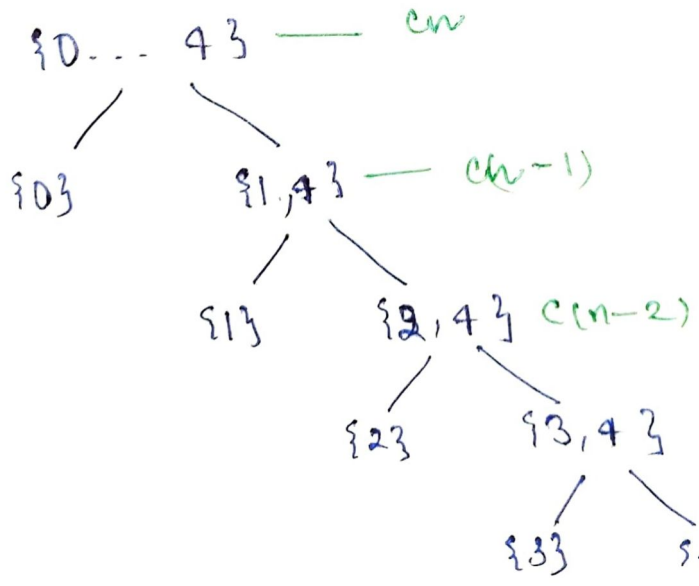
```
void qsort(int arr[],
           int l, int
{
  if (l < h) {
    int p = par(arr, l, h
    qsort(arr, l, p);
    qsort(arr, p+1, h)
  }
}
```

$T(n) = T(n-1) + O(n)$ — worst

$T(n) = 2T(n/2) + O(n)$ — best

Worst case

partition function divides the array in such a way that it have n-1 elements on one side and only 1 elements on other side



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

sum of natural nums.

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

for one element left n times

$O(n^2)$

Is quicksort Inplace?

There are two definitions of inplace algorithms.

(a) An algorithm is said to be inplace if it does not uses any extra space or the auxiliary space complexity is O(1).

By this definition quick sort is NOT inplace algorithm as it requires extra space for recursion call stack.

(b) an algorithm is said inplace if it does not copy input element to any other location

By this definition quicksort is inplace

auxiliary space Required

O(logn) → when in ~~worst~~ best case (Recursion call stack)

O(n) → ~~bst~~ worst case (Recursion call stacks).

- In input array is sorted both lomuto and hoare's partion algorithm will turn into ~~worst~~ worst case as they will have 1 element on one side and (n-1) elements on another side

That why we don't want to hardcode the choice of pivot

We generate a random pivot index

```
int p= random (l, h);
```

In hoare's partition

```
swap (arr[p], arr[l])
```

In lomuto partition

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

Tail call Elimination

As quicksort is tail recursive, we can apply tail call elimination as:

```
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;
    }
}
```

* can be used for optimisation of space req for recursive calls.

K^th Smallest Element (Quickselect Algorithm)

I/P → {10, 5, 30, 12} k=2

O/P → 10

Naive Approach

Sort the array first and then return arr[k-1].

time comp → O(nlogn).

* Modifies original array.

Optimised Approach

We can use lomuto partition, if the index returned by partition function is (k-1) ~~with~~ we will return arr[p] or arr[k-1].

If p < k-1 it means that the elements would occur at right part of array. and we would call only ~~left~~ right part, and if p > k-1 we would call left half.

worst case comp → O(n²)

Average case comp → O(n). (Much better than naive approach)

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

Chocolate Distribution Problem

I/P → {7, 3, 2, 4, 9, 12, 56} m = 3

O/P → 3

we need to distribute chocolate to children where.

(a) ~~packet[i]~~ in packet contain arr[i] chocolates

(b) we can give only 1 packet to one child

(c) distribution must be in such a way that the difference between the min. and max. chocolates distributed must be minimum.

Ex → {7, 3, 2, 4, 9, 12, 56}

if we select →

	diff
7, 3, 4 →	4
4 9 12 →	8
2 4 7 →	7
7 12 56 →	47
3 2 4 →	2 ← min

this comb^n is selected

- We can sort the array and can make a window of m elements and see the relative difference

Ex:- {7, 3, 2, 4, 9, 12, 56}

m = 3

↓

{2, 3, 4, 7, 9, 12, 56}

diff = 4 − 2 = 2

res = 2

↓

{2, 3, 4, 7, 9, 12, 56} diff = 7 − 3 = 4 res = 2

↓

{2, 3, 4, 7, 9, 12, 56} diff = 9 − 4 = 5 res = 2

↓

{2, 3, 4, 7, 9, 12, 56} diff = 12 − 7 = 5 res = 2

↓

{2, 3, 4, 7, 9, 12, 56} diff = 47 res = 2.

res = min (res, diff)

```
int minDifference (int arr[], int n, int m) {
    if (m > n) return -1
    sort (arr, arr+n);
    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;
}.
```

Sort an array with two types of elements

- Segregate -ve and +ve
- Segregate even & odd
- Sort a binary array.

} more versions of same question

~~Hint~~

Naive Approach

Make a temporary array of ~~zero~~ size n, and the initially copy elements ~~not~~ satisfying condition 1 to it and then again ~~traverse~~ the array and store elements satisfying condition 2 after that index i (upto i elements of type 1 are stored)

* O(n) approach
* Require two traversal of array
* O(n) auxiliary space

Effective Approach

Use partition function of quick sort and put necessary conditions instead of (arr[i] > arr[p]).

(a) sorting (segregating) negative and positive elements

```
void segregate ( int arr[], int n) {
    int i = -1, j = n;
    while (true) {
        do { i++; } while ( arr[i] < 0);      → for(0 & 1s)
        do { j--; } while ( arr[j] >= 0);     arr[i] == 0
        if (i >= j)
            return;
        swap( arr[i], arr[j]);
    }
}
```

→ for even odd (arr[i]%2==0)

Sort an Array with three types of Elements

Versions of problem

- Sort arrays of 0's, 1's and 2s.
- Three way partitioning when multiple occurence of pivot may exist

  {2, 3, 3, 4, 2, 5, 6, 3} → {2, 2, 3, 3, 4, 5, 6}
  Type 1 (<3), Type 2 (=3), Type 3 (>3).

- Partition around a range.

  {10, 5, 6, 3, 20, 9, 40} range – [5, 10]

  → {3, 5, 10, 6, 9, 20, 40}

  elements smaller than range | elements which fall in the range (order may vary) | greater than range

Naive solution

make a temporary ~~#~~ array and then push element in order of their conditions.

- push elements of type 1
- push elements of type 2
- push elements of type 3
- copy temp to arr and return arr.

* Efficient Approach

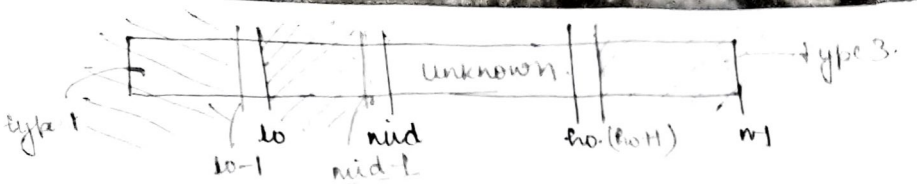
DUTCH National flag Algorithm / 3 way partition

- we will be maintaing three indexes

① lo → all the elements upto lo-1 will be of Type 1 i.e a[lo-1] belongs to category 1 but a[lo] won't

② mid → elements from lo to ~~high~~ (mid-1) belong to type 2

③ hi → elements from (hi+1) to high belong to ③



we can have 3 cases (we always ~~make~~ keep a check on a(mid)

① elements belong to type -1

then swap a(lo) with a(mid).

- a(lo) belong to category 2
- a(mid) belong to category 1
- on swapping the window of both lo & mid is increased
  . lo++, mid++

② element belong to category 2

do mid++ only

③ element belong to category 3

a(hi) belong to unknown.
a(mid) belong to type 3
swap (both)
now hi-- (window of type 3 is increased by 1)
and do nothing with mid as this element needs to be checked.

```
void segregate (int arr[], int n) {
    int lo = 0, hi = n-1, mid = 0
    while (mid <= hi) {
        switch (arr[mid]) {
            case 0: swap (arr[lo], arr[mid]);
                    lo++; mid++;
                    break;
            case 1: mid++;
                    break;
            case 2: swap (arr[mid], arr[hi]);
                    hi--;
                    break;
        }
    }
}
```

{0, 1, 2, 1, 1, 2}
lo = 0 mid = 0 hi = 5

{0, 1, 2, 1, 1, 2}
lo = 1 mid = 1, hi = 5

{0, 1, 2, 1, 1, 2}
lo = 1 mid = 2 hi = 5

{0, 1, 2, 1, 1, 2}
lo = 1 mid = 2 hi = 4

{0, 1, 1, 1, 2, 2}
lo = 1 mid = 2 hi = 3

{0, 1, 1, 1, 2, 2}
lo = 1 mid = 3 hi = 3

{0, 1, 1, 1, 2, 2}
lo = 1 mid = 4 hi = 3

loop break | mid > right

Time complexity
O(n)

Auxiliary space
O(1)

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}}.

we can consider input array as.

① array of pairs

```
pair<int, int> arr[n];
```

② structure or class

```
class Interval {
    int start, end;
};

Interval arr[n];
```

① check if two intervals do overlap;

① method 1

take larger of start values and check if it lies in other interval.

$i_1 = \{5, 10\}$
$i_2 = \{1, 7\}$ > larger of start → 5, check if 5 lies in i_2 → Yes.

$i_1 = \{10, 20\}$
$i_2 = \{100, 200\}$ > larger of start → 100, check if 100 lies between 10 to 20 — NO

② method - 2

take smaller of end values and check if it lies in other interval

$i_1 = \{5, 10\}$
$i_2 = \{1, 7\}$ > smaller value → 7, check if 7 lies in i_1 — Yes

$i_1 = \{10, 20\}$
$i_2 = \{100, 200\}$ > smaller → 20, 20 lies in i_2 — NO

How to merge two intervals

i_3 ← new interval

$i_3.start = min(i_1.start, i_2.start)$
$i_3.end = max(i_1.start, i_2.end)$

Naive Approach $O(n^3)$

Run to two loops and check if two intervals merge, if yes merge them.

{{5,10}, {2,3}, {6,8}, {1,7}}.

$i = 0$, $j = 1$ > NO
$i = 0$, $j = 2$ > Yes, they merge.

→ {{5,10}, {2,3}, {1,7}}

$i = 0$, $j = 3$ > Yes → {{1,10}, {2,3}, X, X}

i++ / $i = 1$, $j = 3$ > Yes, → {{1,10}}

$O(n^2)$ for two loops and $O(n)$ for deleting a pair hence $O(n^3)$ solution

Efficient Approach $O(n \log n)$

we sort the intervals in increasing order of their start-times

and then

sorted → $x_0, x_1, x_2, x_3, x_4 \dots x_{i-1}, x_i$

merged → $(m_0, m_1, m_2 \dots m_{j-1}, m_j)$ x_i

claim is if x_i is going to be merge with any interval, it would by m_j only.

- because x_i will be merged with m_j only if the start time of x_i lies between m_j

↳ elaborating further

m_j ka start time hamesha chota hoga x_i se. so x_i ka start time hamesha m_j se bada hoga. agar wo start time $m_j.end$ se chota hua to merging hogi varna nahi.

$x_i.start \geq m_j.start$

$m_{j+1}.end < m_j.start$ (because they are non overlapping).

Implementation

```
struct Interval {
    int st, end
}
```

```
bool myComp(Interval i1, Interval i2){
    return (i2.start > i1.start);
}
void mergeIntervals(Interval arr[], int n){
    sort(arr, arr+n, myComp);
    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 = max(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;
    }
}
```

intervals are merged and we are updating the result interval only so res is not incremented

Not merged we increament the res & put a[i] in that place.

res=0

~~arr[] → {{5,10}, {3,15}, {18,30}, {2,7}}~~

i=0

~~merging~~

~~{{5,}~~

arr[] → {{5,10}, {3,15}, {18,30}, {2,7}}

sorting → {{2,7}, {3,15}, {5,10}, {18,30}}.

res=0 i=1

merging occurs → {{2,15}, {3,15}, {5,10}, {18,30}}

res=0 i=1

merging occurs → {{2,15}, {3,15}, {5,10}, {18,30}}

res=0 i=2

NO merging → {{2,15}, {18,30}, {5,10}, {18,30}}

res=1

Result is increamented and our return would be arr[res].

Meeting the Maximum guests

- we create a timeline of events by sorting both arrival and departure time.
- If the next element in the timeline is arrival time we increament the count and if it is departure time we decreament the count.
- I/P arr[] → {900, 600, 700}
  dep[] → {1000, 800, 730}.

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

We can merge it in another array and then implement the approach or we can have two pointer and apply them.

```
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 // Assuming one guest has already arrived but not departed
    while (i<n && j<n){
        if (arr[i] <= dep[j]){
            curr++;
            i++;
        }else{
            curr--;
            j++;
        }
        res = max (res, curr);
    }
    return res;
}
```

Time comp → O(nlogn)

Cycle Sort

- $O(n^2)$ worst case algorithm
- minimum memory writes and can be used for cases where memory writes are costly
- Ex:- minimum swaps needed to sort an array
- In-place and not stable algorithm

Ex:- [20, 40, 50, 10, 30]

10 20 40

[10 20 50 40 30]

30 50

[10 20 30 40 50]

we run again and check which item is not at correct place

item = 50

4 items all <

item = 30

2 elements are <.

item = 20

no of elements < 20 = 1

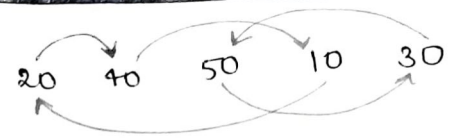
so 20 must be at index = 1

item = 40

3 items are smaller than 40

item = 10 | item = 20

0 items | repeated



Ex-2 {40, 30, 20, 10}

10 40

cycle formed

{10, 30, 20, 40}

20 30

cycle formed

[10 20 30 40]

item = 40
pos = 3

item = 10
pos = 0

item = 30
pos = 2

item = 20
pos = 1

→ Array is sorted
→ every item is at its correct place

Implementation (when elements are Distinct)

- we start by index = 0 and maintain a variable cs which means that elements from 0 to cs-1 are sorted.
- They may be elements from cs to n-1 which are sorted because of loop, but 0 to cs-1 are 100% sorted.
- Now we will check for ~~loop~~ cycle and ~~move to 0~~ run loop uptill ps != cs

pos = the position where an element (item should be present).

~~10 20 50 40 30~~

20 40 50 10 30

pos = 0

item = 20

we check how many elements on right of 20 are greater than 20 because elements on left are sorted already and then increament the position by that amount.

```
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] < arr[cs])
                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]);
        }
    }
}
```

Q// write algorithms for duplicates

Q// minimum swaps to sort an array

Counting sort

- It is used when we know that the input array has elements upto range k.

I/P → {1, 4, 4, 1, 0, 1} k → {4} or precisely k = 5

O/P → {0, 1, 1, 1, 4, 4} [0, 5)

Naive Approach. O(n+k).

Make an array named count and then store the frequency of elements of arr in that array.

i.e. arr → {1, 4, 4, 1, 0, 1}

count → { (1) (3) 0 0 (2) }
0 1 2 3 4

Now run a loop to the elements of count and overwrite arr by the no of times count[i].

{0, 1, 1, 1, 4, 4}.

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

Efficient Approach.

arr[] → {1, 4, 4, 1, 0, 1}

count → {1, 3, 0, 0, 2}

count → {1, 4, 4, 4, 6}

This number actually tells how many elements are smaller than or equal to the element index

Ex:- 6 elements are smaller or equal to 5

Now

building output array output → $\frac{0}{0}$ $\frac{1}{1}$ $\frac{1}{2}$ $\frac{1}{3}$ $\frac{4}{4}$ $\frac{4}{5}$

we actually traverse from right to left to maintain stability in algorithm

i = n-1 = 5 arr[i] = 1

count[arr[i]] → 4.

i.e the position of 1 should by 4th.

out[4-1] = out[3] = 1.

and reduce one occurence, hence

count[arr[i]]--;

i = 4 arr[i] = 0

count[arr[i]] = 1

i.e position of 0 must be 1st

out[1-1] = out[0] = 0.

similarly it can be implemented for others also

output → {0, 1, 1, 1, 4, 4}.

copy output array to arr and return

```
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];
}
```

Important Points

- Not a comparision based algorithm
- $\theta(n+k)$ time
- $\theta(n+k)$ aux space
- stable
- used as a subroutine in radix sort
- only useful when k is really small.

// Extend it with the given range of k.

max value − min value = k

↑

This can be negative too

Radix Sort

Radix sort supports linear time even for larger range

- counting sort is used as a subroutine.

Ex → {319, 212, 6, 8, 100, 50}.

Re-write number with leading zeroes.

→ {319, 212, 006, 008, 100, 050}.

Stable sort according to last digit (least sig. digit)

{100, 050, 212, 006, 008, 319}

Stable sort according to middle digit

{100, 006, 008, 212, 319, 050}

Stable sort according to most significant digit

{006, 008, 050, 100, 212, 319}

Sorted Array

To get the k^{th} digit of a number

num → $x_n x_{n-1} \cdots x_k x_{k-1} \cdots x_1$

$$\left(\frac{num}{(10)^{k-1}}\right) \% 10$$

Ex → num = 2345

to get 2nd digit

$$\left(\frac{2345}{10^{2-1}}\right) \% 10 = \frac{2345}{10} \% 10 = 234 \% 10 = 4$$

Steps

① check for the largest number

② we have to run the loop upto m times where m = no. of digits in largest number.

③ maintain a variable exp = 1

on every iteration exp *= 10

exp is used to get the certain digit of a number.

initially when we want

1st digit exp = 1

to get digit → $\left(\frac{num}{exp}\right) \% 10$

2nd digit exp = 10

$= \left(\frac{num}{10}\right) \% 10$

indirectly

$exp = 10^{k-1}$

IMPLEMENTATION

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

void countingSort (int arr[], int n, int exp){
    int count[10], output[n];
    for(int i=0; i<n ; i++) count [i] = 0;
```

O(n)

O(d * (n+b))

d → no. of digits
n → number of elements
b → base

making count array

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

making cummulative array

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

```
for(int 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];
}
```

arr[] = {319, 212, 6, 8, 100, 50}

mx = 319

countingsort(arr, 6, 1)
countingsort(arr, 6, 10)
countingsort(arr, 6, 100)

call counting sort where index = (arr[i]/exp)%10

Time comp → $\theta(n*(n+b))$
Aux comp → $\theta(n+b)$

Bucket sort

- situation where numbers are uniformly distributed in range from 1 to 10^8.
- similar is the case with floating point numbers. Or range is 0.0 to 1.0

NOW, uniformly distributed means

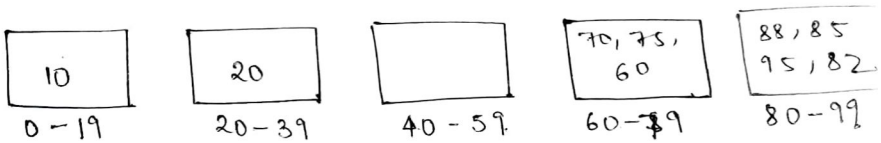
1000 numbers in range 0.0 to 1.0

- 100 in 0.0 to 0.1 (0.1 excluded)
- 100 in 0.1 to 0.2 (0.2 excluded)
- ⋮
- 100 in 0.9 to 1 (1 excluded)

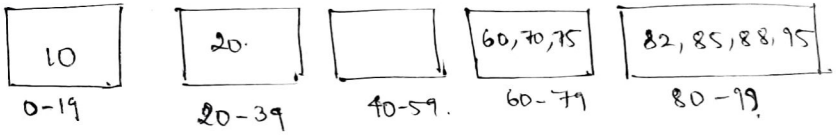
Ex:- {20, 88, 10, 85, 75, 95, 18, 82, 60}

Range – 0 to 99.

step 1 scatter



step 2 sort buckets



step 3 join sorted buckets

10, 20, 60, 70, 75, 82, 85, 88, 95

* this example is not uniformly distributed but bucket sort work efficiently only when elements are uniformly distributed
* we can use insertion sort if number of items in buckets are less

I/P → {20, 80, 10, 85, 75, 99, 18}
k = 5

Total range = 0 – 99
Buckets = 5

O/P → {10, 18, 20, 75, 80, 85, 99}

```
void bucketSort(int arr[], int n, int k) {
    int max_val = arr[0];
    for (int i = 1 → n) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    max_val++;
    vector<int> bkt[k];
    for (int i=0 ; i<n ; i++) {
        int bi = (k* arr[i]) / max_val;
        bkt[bi].push_back(arr[i]);
    }
    for (int i=0 ; i<k ; i++) {
        sort(bkt[i].begin(), bkt[i].end());
    }
    int index = 0;
    for (int i=0; i<k ; i++) {
        for (int j=0 ; j< bkt.size() ; j++){
            arr[index] = bkt[i][j];
            index++;
        }
    }
}
```

max_val is calculated

choosing the bucket & pushing elements

Best case → uniformly distributed data
O(n)

worst case → All items goes into single bucket & if we use insertion sort here we'll have $O(n^2)$ complexity

Shell sort

Variation of insertion sort, we initialise gap as $n/2$ and then decrease it to 1.

To use to bring smaller elements to left with less number of comparisons

0	1	2	3	4	5	6	7	8
23	29	15	19	31	7	9	5	2
↑i				↑j				

gap = ⌊n/2⌋ = 4

increament i and j till j reaches n+1.
and if swapping happens i.e (arr[i] > arr[j]) we have to compare it with previous index i.e i - gap

After one pass decreament gap → gap/2 and repeat the same till gap reaches → 1

29	(7)	15	19	31	(29)	9	5	2
23	7	(9)	19	31	29	(15)	5	2
23	7	9	(5)	31	29	15	(19)	2
23	7	9	5	(2)	29	15	19	31
(2)	7	9	5	(23)	29	15	19	31

Pass – 1

gap → gap/2 (2)

2	(5)	9	(7)	23	29	15	19	31
2	(5)							
2	5	9	7	(15)	29	(23)	19	31
2	5	9	7	15	(19)	23	(29)	31

Pass 2

gap → gap/2 (1)

2 5 (7) (9) 15 19 23 29 31 } Pass – 3

```
for(gap = n/2 ; gap >= 1 ; gap/2) {
    for (j= gap ; j< n ; j++) {
        for (i= j-gap ; i >= 0 ; i-gap) {
            if (arr[i+gap] > a[i]) {
                    break;
            }
            else {
                swap(a[i+gap], a[i])
            }
        }
    }
}
```