

SORTING

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

sort in array

sort(arr, arr+n)

first address

address of element just after last element -

sort in vector

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

- 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.

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

$$\text{avr} \rightarrow \left\{ \frac{1}{9}, \frac{2}{2}, \frac{3}{2}, \frac{4}{8}, \frac{5}{4} \right\}$$

$$O/P \rightarrow \{ \begin{matrix} 2 & 3 & 5 & 4 & 1 \\ 2, & 2, & 4, & 8, & 9 \end{matrix} \} \Leftrightarrow \{ \begin{matrix} 3 & 2 & 5 & 4 & 1 \\ 2, & 2, & 4, & 8, & 9 \end{matrix} \}$$

stable not-stable

Or we can visualise as

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

↓
selected on basis of
Marks

$\{("Anil", 50), ("Pujsh", 50),$
 $("Ayan", 80), ("Ramesh", 80)\}$ $\{("Pujsh", 50), ("Anil", 50)$
 $\dots\}$.
stable unstable.

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

Stable sorting → Bubble Sort, Insertion Sort,
Merge Sort

unstable \rightarrow Selection sort, Quick sort, Heap sort -

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 iteration we look for the correct position of ith 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 arr 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. $\rightarrow O(n)$

(b) Worst case

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

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

Merge Sort

- ① Divide and conquer algorithm
- ② Stable algorithm
- ③ $O(n \log n)$ time and $O(n)$ extra space.
- ④ well suited for linked lists, works in $O(1)$ aux space.
- ⑤ for arrays \rightarrow 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 $a[] \rightarrow \{10, 15, 20, 40\}$
 $b[] \rightarrow \{5, 6, 6, 10, 15\}$.

O/P $\rightarrow \{5, 6, 6, 10, 10, 15, 15, 20, 40\}$.

① Naive solution

$\approx O(n+m)$ extra space

$\approx O(n+m(\log(n+m)))$ time complexity

copy all the elements in a separate array + sort it

Effective solution

$O(m+n)$

If have two points $p_1 \rightarrow l$ 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 arr, int b[], int m, int n) {

 int $p_1 = 0$

 int $p_2 = 0$

 while ($p_1 < m \wedge p_2 < n$) {

 if ($a[p_1] < b[p_2]$) {

 cout << $b[p_2]$;

p_2++ ;

 } else {

 cout << $a[p_1]$;

p_1++ ;

 }

}

 while ($p_1 < m$) { cout << $a[p_1++]$; }

 while ($p_2 < n$) { cout << $b[p_2++]$; }

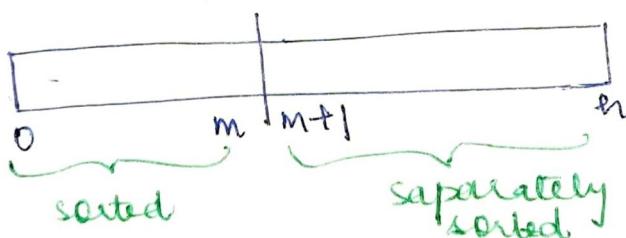
}

} moving the
pointers $p_1 + p_2$
as per cases
stated

Merge Function of par merge sort

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

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



- 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 set to merge two sorted arrays by the algo done above

• size of arr $\rightarrow m-l+1$

• size of arr $\rightarrow n-m$.

$$l \leq m < h$$

void merge (int arr[], int l, int m, int r) {

 int n₁ = m - l + 1;

 int n₂ = r - m;

 int left[n₁], right[n₂];

 } for (int i=0; i<n₁; i++) {

 left[i] = arr[l+i]; \rightarrow l is not always 0.

copying elements in left array

 } for (int j=0; j<n₂; j++) {

 right[j] = arr[m+1+j];

copying in right array

 while (i < n₁ & j < n₂) {

 if (left[i] < right[j]) {

equals to ensures the stability of merge sort

 arr[k++] = left[i++];

 } else {

 arr[k++] = right[j++];

 }.

merging logic & moving of pts

 while (i < n₁)

 arr[k++] = left[i++];

 while (j < n₂)

 arr[k++] = right[j++];

copying remaining elements

}

Thy

Merge Sort Algorithm

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

void mergesort (int arr[], int l, int r) {

if ($r > l$){

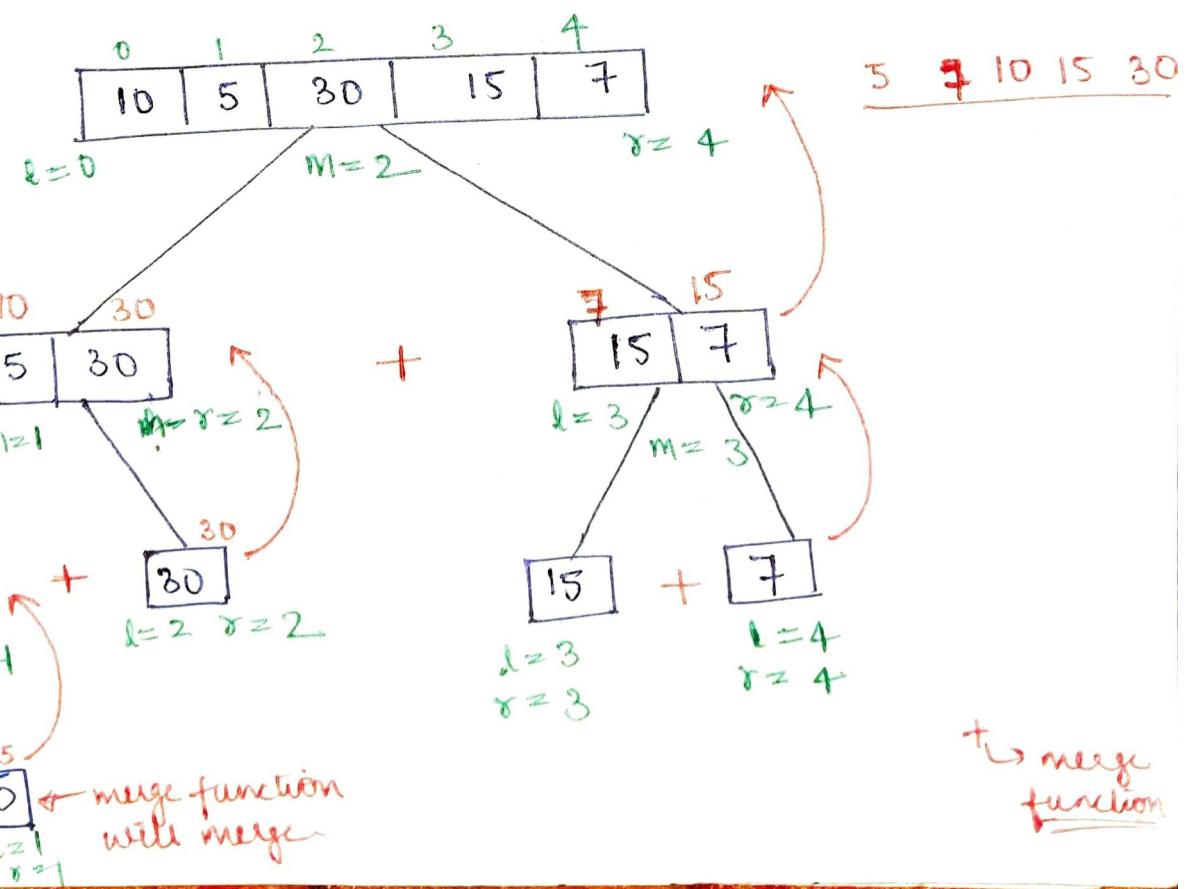
int m = $l + (r-l)/2$; → similar to
 $(l+r)/2$
mergesort (arr, l, m); did this in
this way to
avoid
overflow
mergesort (arr, m+1, r);
merge (arr, l, m, r);

}

merge the
two sorted
halves.

sets left part
of array

sets right part
of array



Time complexity of Merge sort :-

$$\Theta(n) \times \Theta(n \log n)$$

↗
merge
function ↗
no of recursive
calls.

Auxiliary space $\rightarrow \Theta(n)$

Intersection of two sorted Arrays

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

* we can have
duplicates
also

O/P $\rightarrow \{8, 15\}$.

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

$a[i] > b[j] \rightarrow$ move j by 1 step

$a[i] < b[j] \rightarrow$ move i by 1 step

$a[i] = b[j] \rightarrow i++, j++$, (we get the common
element)

void findInt(int a[], int b[]){

int $i = 0, j = 0;$

// n_1 = size of a
// n_2 = size of b

while ($i < n_1$ && $j < n_2$) {

 if ($a[i] > b[j]$) { $j++;$ }

* TO check for
duplicates

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

 else if ($a[i] == b[j]$) {

$i++;$ $j++;$

 cout << a[i];

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

}

}

Union of two sorted Arrays

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

O/P $\rightarrow 2, 3, 8, 9, 10, 15$

Naive Solution

Set the merged array using an extra array of $(m+n)$ size and then traverse through the whole array & print distinct elements only.

T.C $\rightarrow \Theta(n+m) \log(n+m)$

A.S $\rightarrow \Theta(n+m)$

Efficient Method

- we have the following conditions

- ① $b[j] < a[i] \rightarrow j++, \text{print } b[j]$
- ② $a[i] < b[j] \rightarrow i++, \text{print } a[i]$
- ③ $a[i-1] == a[i] \rightarrow i++$
- ④ $b[j-1] == b[j] \rightarrow j++$
- ⑤ $b[j] == a[i] = i++, j++, \text{print } a[i]$.

```
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++; j++}
        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]) {i++}
                  cout << a[i]; i++}
    while (j<n) {if (j==0 || b[j]==b[j-1]) {j++}
                  cout << b[j]; j++}
}
```

Count inversion in a Array

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

- ④ in simple words when a greater element appears first in an array

I/P $\rightarrow \{2, 4, 1, 3, 5\}$

O/P $\rightarrow 3 \quad (4,1) (4,3) (2,1)$

I/P $\rightarrow \{10, 20, 30, 40\}$

O/P $\rightarrow 0$ (sorted in increasing order)

I/P $\rightarrow \{40, 30, 20, 10\}$

O/P $\rightarrow 6$ sorted in reverse order

$$\text{no of inversions} = \frac{n(n-1)}{2}$$

$$4(3)/2 = 6.$$

Main solution

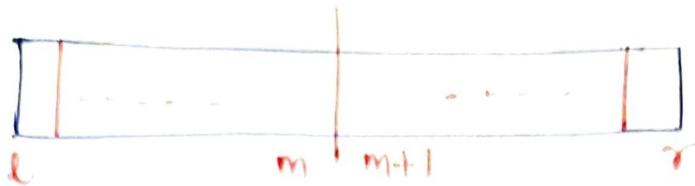
$O(n^2)$ approach \rightarrow run two loops and check for elements in the right of an element & count if the smaller element is appearing in the right

```
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[i] > arr[j])
                res++;
        }
    }
    return res;
}
```

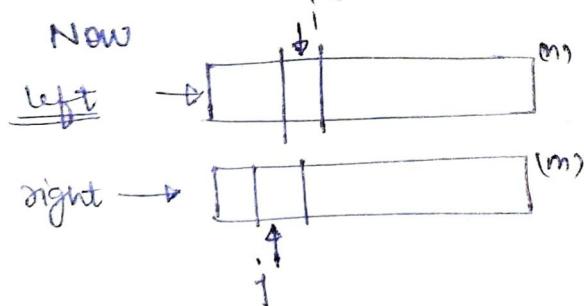
Efficient solution

$O(n \log n)$

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 situations:
 - ① $\text{left}[i] < \text{right}[j] \rightarrow$ this is not the cond" for inversion
 - ② $\text{left}[i] > \text{right}[j] \rightarrow$ condition for inversion.



If element in right array is ^{smaller} greater than left met means it is smaller than all the remaining $(n-i)$ elements of left array. Hence count will be increased by $\text{res} += (n-i)$

Algorithm

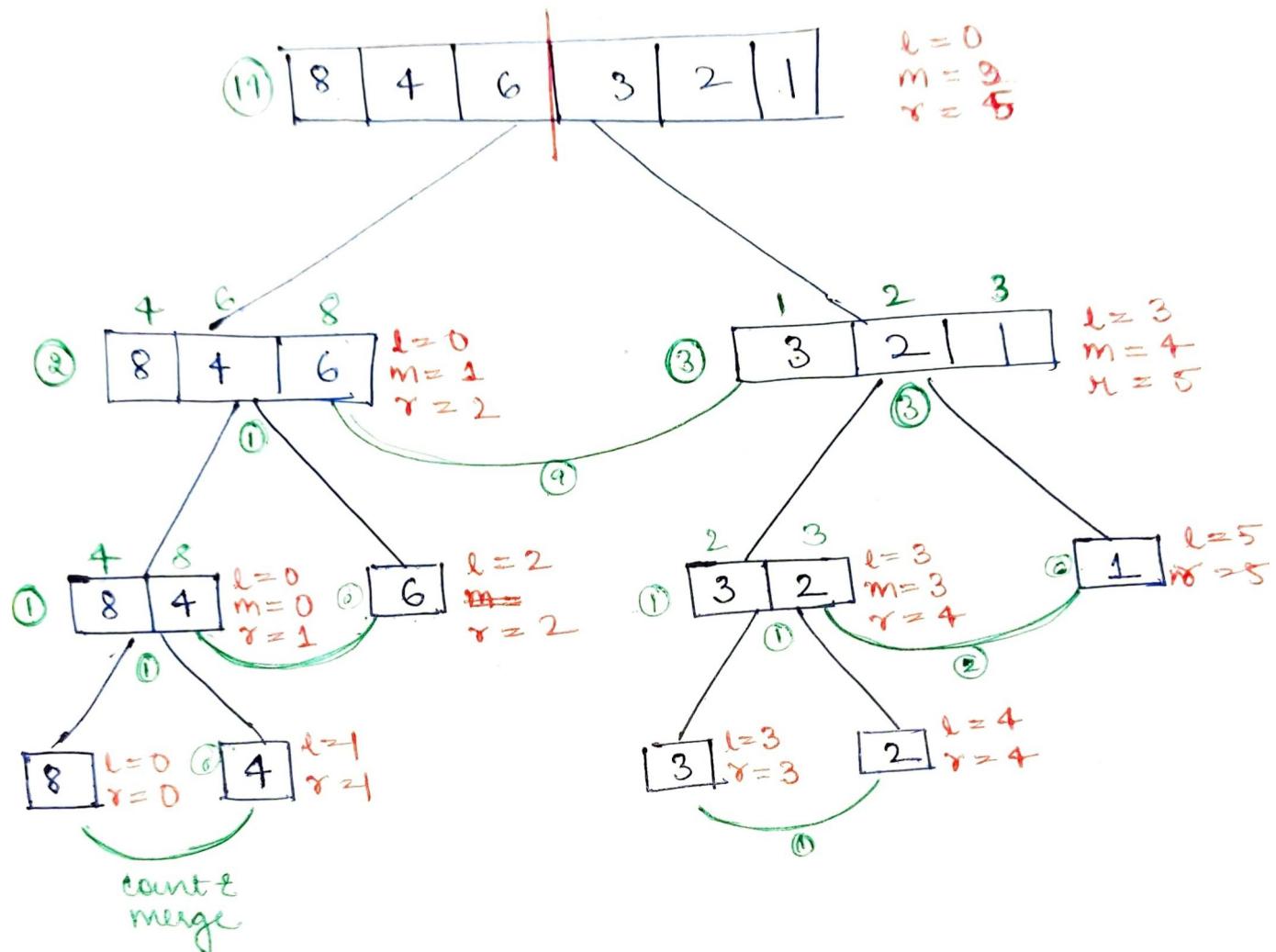
```
int countInv (int arr[], int l, int r) {
    int res = 0;
    if (l < r) {
        int m = l + (r - l) / 2;
        res += countInv (arr, l, m);
        res += countInv (arr, m + 1, r);
        res += countAndMerge (arr, l, m, r);
    }
    return res;
}
```

```
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 [j] < left [i]) {
            arr [k] = right [j];
            j++;
            res = res + (n1 - i);
        }
        k++;
    }
}
```

while ($i < n_1$) { $ans[k] = left[i]$; $i++$; $k++$; }
 while ($j < n_2$) { $ans[k] = right[j]$; $j++$; $k++$; }
 return ans ;

9

Ex \rightarrow 8, 4, 6, 3, 2, 1



Time complexity $\rightarrow O(n \log n)$ Auxiliary space - $O(n)$

Partition a Given Array

I/P \rightarrow arr = {3, 8, 6, 12, 10, 7} $p = 5$ (Index of element around which we have to partition)
 O/P \rightarrow {3, 6, 7, 8, 10, 12}
 or
 {6, 3, 7, 8, 10, 12}

{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 sorted array, arr[p] should be at its correct position i.e. at the same position as it would appear if array is sorted.

Main 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];
            set index++;
        }
    }
}
```

add cond
~~i ≠ p~~

```
for (int i = l; i ≤ h; i++)
    if (arr[i] > arr[p]) {
        temp[index] = arr[i];
        set index++;
    }
}
```

```
for (int i = l; i ≤ h; i++)
    arr[i] = temp[i-l];
```

}

time complexity → O(n)

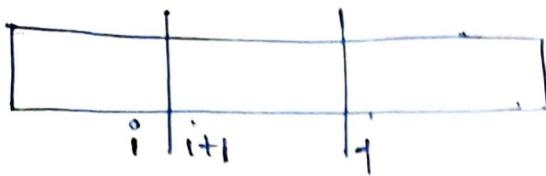
Auxiliary space → O(n).

Ex → 2 3 8 4 5 6 9 2

p = 3

Lamuto Partition

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



at any point j of the loop running from $0 \rightarrow n-1$ we maintain two things

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

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

so if $\text{arr}[j] > \text{arr}[\text{pivot}] \rightarrow$ we do nothing as we want that element to be at right of ~~array or~~(i)

if $\text{arr}[j] \leq \text{arr}[\text{pivot}] \rightarrow$ we swap the element with the element at i^{th} position

and then increase the size of window.

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

\uparrow
 $j=0$
 (initially window
 is at $i=0-1$)

$$h = 6$$

at $j=0$ now $\text{arr}[j] (10) < \text{arr}[h] (60)$

so swap $(+1) [-1+1] \rightarrow 0^{th}$ position with $j^{th} (0)^{th}$ element.

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

now $j=1$ $\text{arr}[j] = 80 \rightarrow$ do nothing

$j=2$ $\text{arr}[j] = 30$

swap $(+1)^{th}$ i.e. 1st with 2nd $\rightarrow \{10, 30, 80 \dots\}$

```

void lPartition(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}

$i = -1$ $j = 0$
 \swarrow swap $i+1$ with j $i++$

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

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

;

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

↓

{10, 30, 40, 50, 80, 90, 70}

$i = 3$ $j = 6 = h$

\hookrightarrow we will not do anything
we will swap $(i+1)$ with h .

{10, 30, 40, 50, 70, 90, 80}

← smaller

→ greater

Corner cases

① $\{70, 60, 80, 40, 30\} \rightarrow \{30, 60, 80, 40, 70\}$.



② $\{30, 40, 20, 50, 80\}$.

$i = -1 \quad j = 1 \rightarrow \{30, 40, 20, 50, 80\}$.

$i = 1 \quad j = 2 \rightarrow \{30, 40, 20, 50, 80\}$

There will be 4 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^m element with $n-1^m$ index and then apply the standard lumoto-p

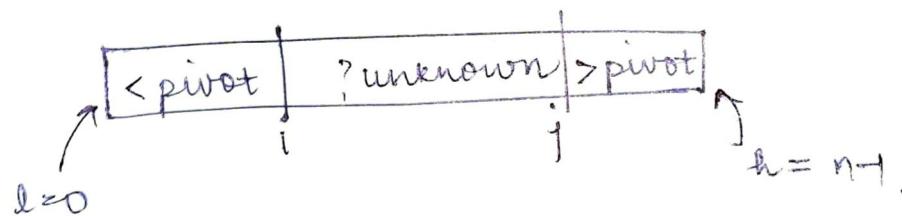
Time complexity $\rightarrow O(n)$ (and only 1 traversal reqd).
Aux. Space $\rightarrow O(1)$

Hoare Partition

pivot element is the first element arr[1]

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



i starts from low-1 & then $i++$

j starts from high+1 & then $j--$

we break the loop whenever $i \geq j$ or $i > j$.

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

array

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 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]);  
    }  
}
```

3.

~~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

$$\{12, 10, 5, 9\} \rightarrow \{9, 10, 5, 12\}$$

$i=0$ ↑ ↑
 $\{9, 10, 5, \underline{12}\}$

(b) all the elements are same

$$\{5, 5, 5, 5, 5\} \rightarrow \{5, \cancel{5}, \cancel{5}, \cancel{5}, 5\}$$

↑
 $\rightarrow \{\underline{5}, \underline{5}, \underline{5}, \underline{5}\}$

Not stable as equal elements are also swapped.

Quick sort

(a) divide and conquer algorithm

Not stable

stable
but
auxiliary
space

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

(c) worst case complexity $\rightarrow O(n^2)$.

(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 returns that position
- we need to call quicksort again for left of pivot index (placed correctly), and for the right also.

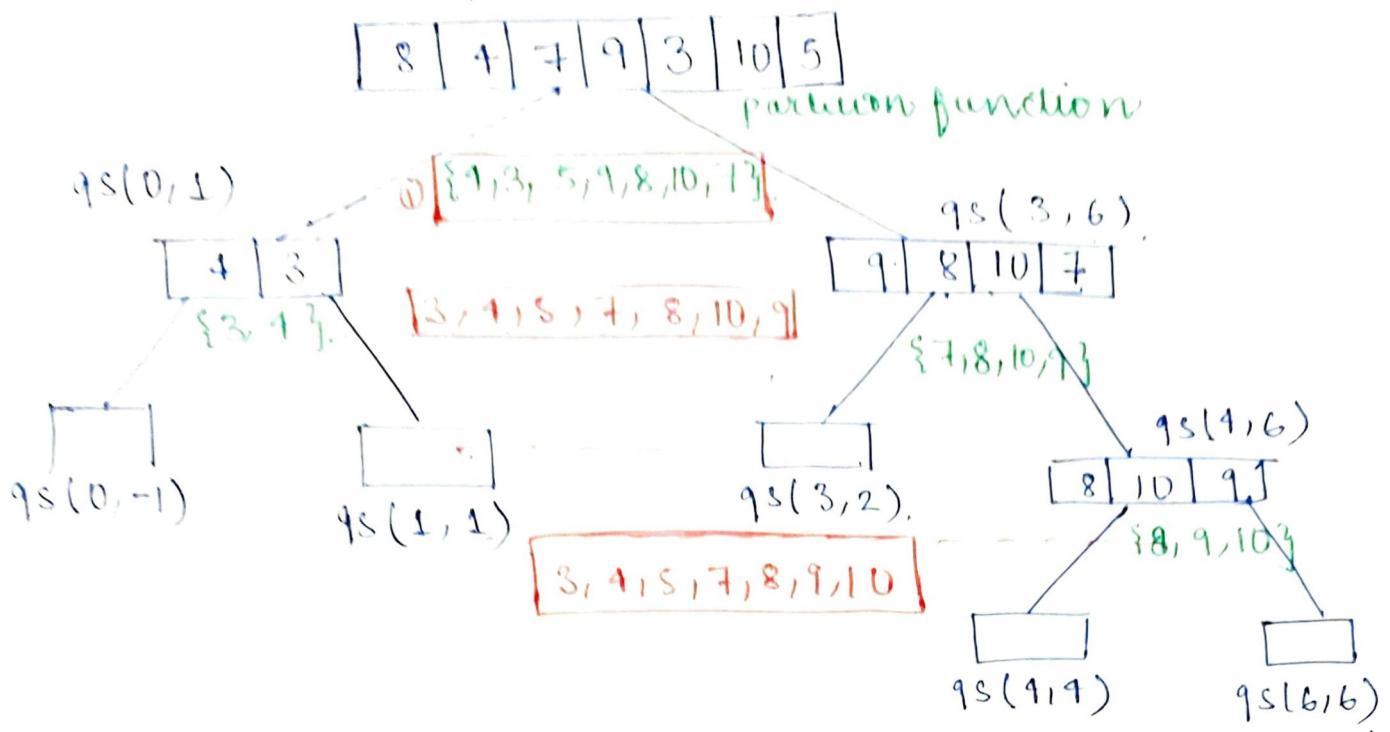
$$\{8, 4, 7, 9, 3, 10, 5\}$$

put 5 on correct position

$$\{8 \cancel{4} \underline{3} \cancel{5} _ _ _ _ _ _ \}$$

call qs call qs

qs(0, 6)



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 (lemento)

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 the element before index i is less than pivot & element after i is greater than or equal to pivot.

Hence the recursive calls are

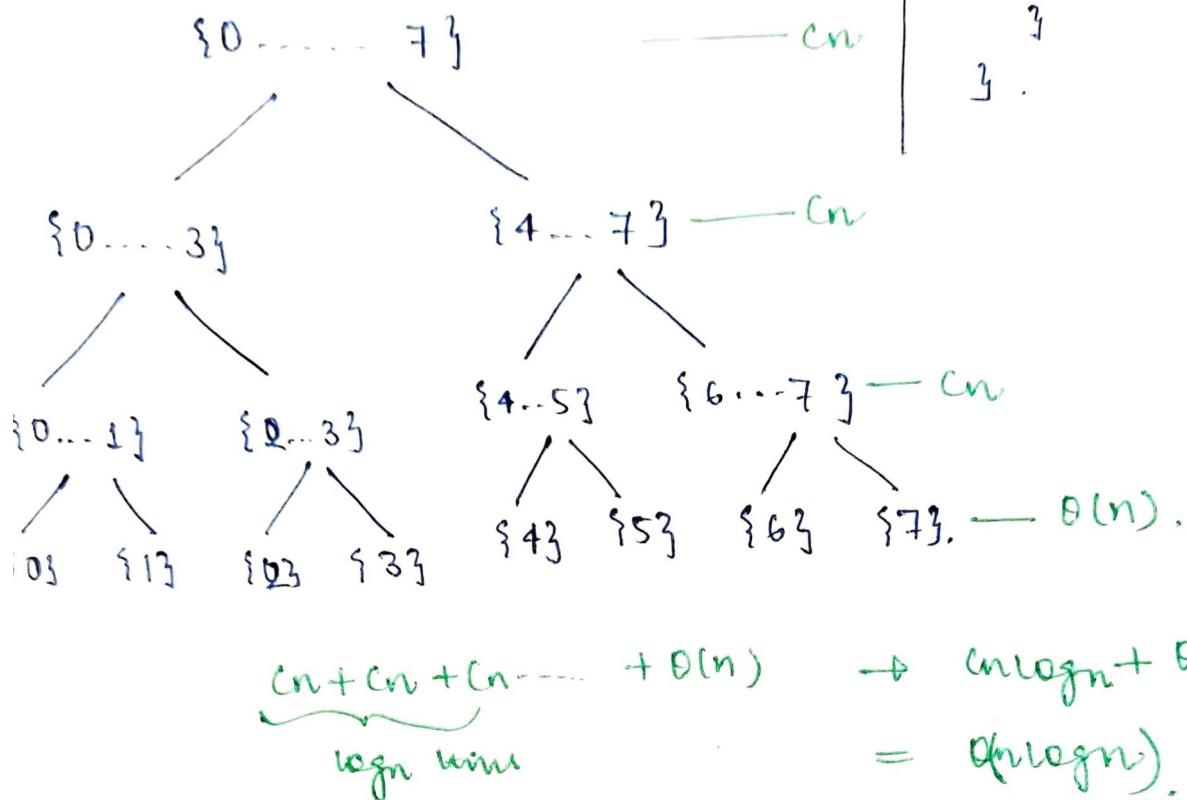
$\text{qsort}(\text{arr}, \text{l}, \text{p})$

$\text{qsort}(\text{arr}, \text{p}+1, \text{h})$.

Analysis of Quicksort

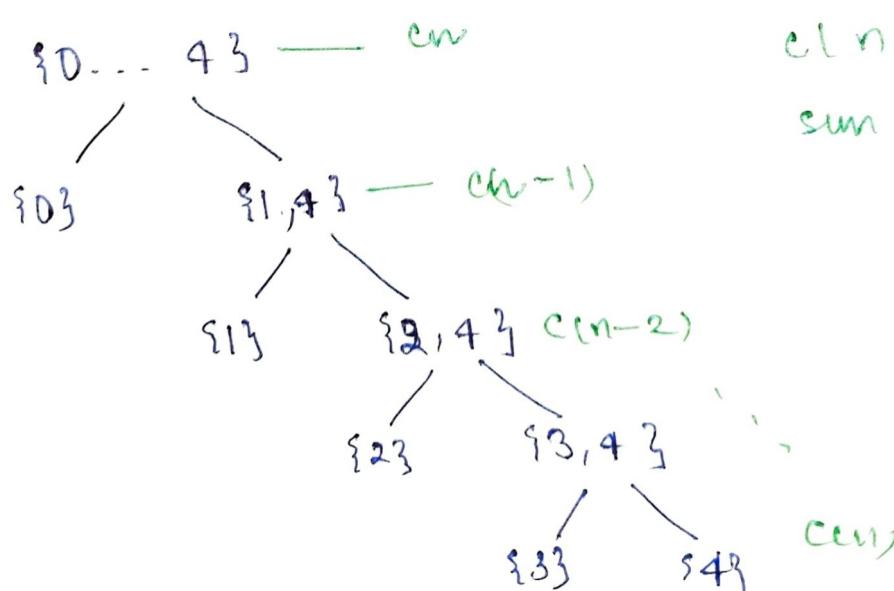
Best case [$\Theta(n \log n)$]

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



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



Algorithm

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

$$T(n) = T(n-1) + \Theta(n) - \text{worst}$$

$$T(n) = 2T(n/2) + \Theta(n) - \text{best}$$

$C(n-1) + C(n-2) + \dots + C1 + C0$
sum of natural numbers.

$\Theta(n^2) + \Theta(n)$
for one element left in line

$\Theta(n^2)$

Is quicksort Inplace?

There are two definitions of inplace algorithms.

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

By this definition, quicksort 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

$O(\log n)$ → when in ^{best} worst case
(Recursion call stack)

$\Theta(n)$ → ~~worst~~ best case
(Recursion call stacks).

In input array is sorted both Lomuto and Hoare's partition algorithm will turn into ~~worst~~ best case as they will have 1 element on one side and $(n-1)$ elements on another side.

That's 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;
```

```
y     goto Begin;
```

```
y
```

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

K^{th} Smallest Element (Quicksort Algorithm).

I/P $\rightarrow \{10, 5, 30, 12\}$ $R=2$

O/P $\rightarrow 10$

Naive Approach

Sort the array first and then return $\text{arr}[k-1]$.

time comp $\rightarrow O(n \log n)$.

* modifies original array.

Optimised Approach

We can use iterative partition. If the index returned by partition function is $(k-1)$ then we will return $\text{arr}[p]$ or $\text{arr}[k-1]$.

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

worst case comp $\rightarrow O(n^2)$

Average case comp $\rightarrow O(n)$. (Much better than naive approach)

```

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

```

Chocolate Distribution Problem

I/P $\rightarrow \{7, 3, 2, 4, 9, 12, 56\}$ m = 3

O/P $\rightarrow 3$

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 $\rightarrow \{7, 3, 2, 4, 9, 12, 56\}$

if we select \rightarrow	<u>7, 3, 4</u> \rightarrow 4	diff
	<u>4 9 12</u> \rightarrow 8	
	<u>2 4 9</u> \rightarrow 7	
	<u>9 12 56</u> \rightarrow 47	
	<u>3 2 4</u> \rightarrow 2	min
<u>this combⁿ</u>		
<u>is selected</u>		

• we can sort the array and can make a window of m elements and see the relative difference

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

$$m = 3$$

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

$$\text{diff} = 7 - 3 = 4$$

$$\text{res} = 4$$

$\{2, 3, 4, \underline{7}, 9, 12, 56\}$ $\text{diff} = 9 - 7 = 2$ $\text{res} = 2$

$\{2, 3, 4, \underline{7, 9}, 12, 56\}$ $\text{diff} = 9 - 4 = 5$ $\text{res} = 2$

$\{2, 3, 4, \underline{7, 9, 12}, 56\}$ $\text{diff} = 12 - 7 = 5$ $\text{res} = 2$

$\{2, 3, 4, \underline{7, 9, 12, 56}\}$ $\text{diff} = 47$ $\text{res} = 2$.

$$\boxed{\text{res} = \min(\text{res}, \text{diff})}$$

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

Naive Approach

Make a temporary array of size n , and then initially copy elements 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 ($\text{arr}[i] > \text{arr}[p]$).

(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 );
        do { j--; } while ( arr[j] >= 0 );
        if ( i >= j )
            return;
        swap( arr[i], arr[j] );
    }
}
```

for even odd ($\text{arr}[i] \neq 0$)
for ($0 \neq 1s$)
 $\text{arr}[i] = 0$

Sort an Array with three types of Elements

Versions of problem

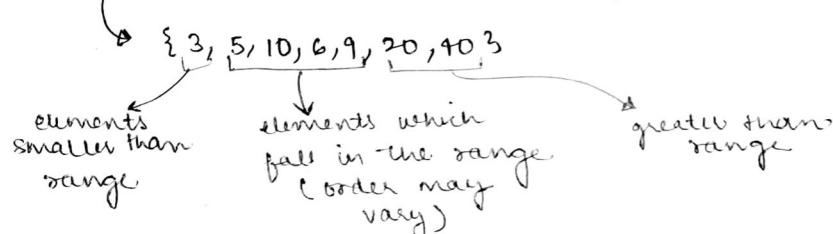
- sort arrays of 0's, 1's and 2's.
- Three way partitioning when multiple occurrence of pivot may exist

$$\{2, 3, 3, 4, 2, 5, 6, 3\} \rightarrow \begin{cases} 2, 2 \\ 3, 3 \\ 4, 5, 6 \end{cases}$$

Type 1 Type 2 Type 3
 $< 3 = 3 > 3$

- Partition around a range.

$$\{10, 5, 6, 3, 20, 9, 40\} \text{ range } [5, 10]$$



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

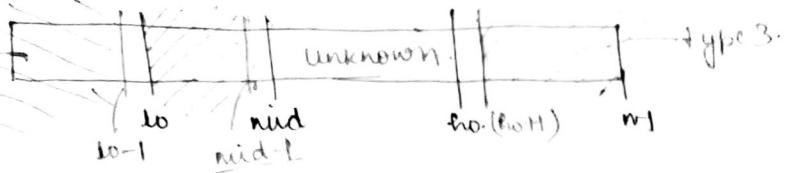
DUTCH National flag Algorithm / 3 way partition

- we will be maintaining 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 $\cancel{mid+1}$ belong to type 2

③ hi → elements from ($hi+1$) to $arr[n]$ 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

$a(lo)$ 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 3 4 5
{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

Merge overlapping intervals

I/P $\rightarrow \{1, 3\}, \{2, 4\}, \{5, 7\}, \{6, 8\}$.

O/P $\rightarrow \{1, 4\}, \{5, 8\}$.

I/P $\rightarrow \{7, 9\}, \{6, 10\}, \{4, 5\}, \{1, 3\}, \{2, 4\}$.

O/P $\rightarrow \{1, 5\}, \{6, 10\}$.

Time complexity
 $O(n)$.

Auxiliary space
 $O(1)$

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];

③ check if two intervals overlap;

① method 1

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

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

$i_1 = \{10, 20\}$ > larger of start $\rightarrow 100$
 $i_2 = \{100, 200\}$ > 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

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

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

How to merge two intervals

$i_3 \leftarrow$ new interval

$i_3.start = \min(i_1.start, i_2.start)$

$i_3.end = \max(i_1.end, i_2.end)$

Naive Approach $O(n^3)$

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

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

$i=0 > \text{No}$ $i=0 > \text{Yes, they merge.}$
 $i=1$

$\rightarrow \{\{5, 10\}, \{2, 3\}, \{1, 7\}\}$,

$i=0 > \text{No Yes} \rightarrow \{\{1, 10\}, \{2, 3\}, \{X, X\}\}$.

$i+1 / i=1 > \text{No Yes,} \rightarrow \boxed{\{1, 10\}}$

$O(n^2)$ 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

cor $\rightarrow x_0, x_1, x_2, x_3, x_4, \dots, x_i, x_i$

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

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

because x_i will be merged with m_j only if the start time of x_i lies between m_j and m_{j+1}

elaborating further

m_j ka start time hamisha chota hogi ki se.
so x_i ka start time hamisha m_j se bda hoga.
agari wo start time m_j se chota hua to
merging hogi verna nhi.

$i.i.start \geq mij.i.start$

$mij.i.end < mij.i.start$ (because they are non overlapping).

Implementation

```
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++) {
```

intervals are merged and we are updating the result interval only so far it is not merged.

```
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 {
```

Not merged yet increment res & put arr[i] in that place.

```
res++;
```

```
arr[res] = arr[i];
```

```
}
```

```
for (int i = 0; i < res; i++) {
```

```
cout << arr[i].st << " " << arr[i].end << endl;
```

```
.
```

```
res = 0
```

```
↓
```

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

```
↑  
i = 0
```

~~merging~~

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



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 implemented and our order would be arr[res].

Meeting the Maximum Guests

- we create a timeline of events by setting both arrival and departure time.

- If the next element in the timeline is arrival time we increment the count and if it is departure time we decrement the count.

- I/P arr[] → {900, 600, 700}
 dep[] → {1000, 800, 730}.

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

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

```
int maxGuest (int arr[], int dep[], int n) {
```

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

```
    sort (dep, dep+n);
```

```
    int i=1, j=0, res=1, curv=1 // Assuming one guest has already arrived but not departed
```

```
    while (i < n && j < n) {
```

```
        if (arr[i] < dep[j]) {
```

```
            curv++;
```

```
            i++;
```

```
} else {
```

```
    curv--;
```

```
    j++;
```

```
4
```

```
    res = max (res, curv);
```

```
3
```

```
return res;
```

```
2
```

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]

item = 30

2 elements are <

item = 50

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

item = 50

4 items all <

Time comp +
 $O(n \log n)$



Ex-2

{40, 30, 20, 10}

10

cycle formed

{40, 30, 20, 40}

20

30

cycle formed

10 20 30 40

- 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 cycle and move to & run loop until pos >= 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 selected already and then increment the position by that amount.

item = 40
3 items are smaller than 40
item = 10 | item = 20
0 items | repeated

```

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 $\rightarrow \{1, 4, 4, 1, 0, 1\}$

O/P $\rightarrow \{0, 1, 1, 1, 4, 4\}$

$k \rightarrow \{4\}$, or precisely
 $k = 5$

[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 $\rightarrow \{1, 4, 4, 1, 0, 1\}$

count $\rightarrow \begin{matrix} \textcircled{1} & \textcircled{3} & \underline{\textcircled{0}} & \underline{\textcircled{0}} & \textcircled{2} \\ 0 & 1 & 2 & 3 & 4 \end{matrix}$

Now run a loop to the elements of count and overwrite arr by the no of times arr[i].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[] \rightarrow {1, 4, 4, 1, 0, 1}

count \rightarrow {1, 3, 0, 0, 2}

count \rightarrow {1, 4, 3, 4, 6}

Ex:- 6 elements
are smaller
or equal to
5

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

Now
building output array output \rightarrow 0 1 1 1 4 4
we actually traverse from right to left to
maintain stability in algorithm.

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

count[arr[i]] \rightarrow 4.

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

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

and reduce one occurrence, 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 \rightarrow {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]]--;
    }
}

```

Important Points

- Not a comparison 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 .
 $\max \text{ value} - \min \text{ value} = k$

↑
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 zeros.

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

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 → $x_n x_{n-1} \dots x_k x_{k+1} \dots x_1$

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

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 $m = \text{no. of digits in largest number.}$

- ③ maintain a variable $\text{exp} = 1$

on every iteration $\text{exp} * 10$

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

~~initially~~ when we want

1st digit $\text{exp} = 1$

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

2nd digit $\text{exp} = 10$

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

indirectly
 $\text{exp} = 10^{k-1}$

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];
```

O(n)

O(d * (n+b))

d = no. of digits

n = no. of elements

b = base

3

```
    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;
```

```

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

making { 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];
}

```

$\text{array}[] = \{319, 212, 618, 100, 50\}$

counting sort (arr, 6, 1)

counting sort (arr, 6, 10)

Counting Set (over, 6,100)

call counting sort where index = $(arr[i]/exp)^{1/15}$

Time comp $\rightarrow \Theta(n * (n + b))$

Aux comp $\rightarrow \underline{\mathcal{O}(n+b)}$

Bucket sort

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

Now, uniformly distributed means

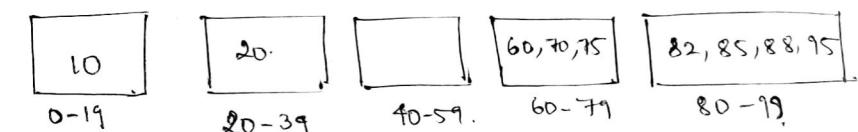
Ex: {20, 88, 10, 85, 75, 95, 18, 82, 60}

Range - 0 to 99

Step-
scatter



step 2
sort bucket



Step 3
Join
sorted
books

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

* This example is not uniformly distributed but bucket set work efficiently only when elements are uniformly distributed

- * we can use insertion sort if number of items in buckets are less

I/P $\rightarrow \{20, 80, 10, 85, 75, 99, 18\}$

$k = 5$

O/P $\rightarrow \{10, 18, 20, 75, 80, 85, 99\}$

void bucketsort(int arr[], int n, int k) {

```

    int max_val = arr[0];
    for (int i = 1; i < n; i++) {
        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[i].size(); j++) {
            arr[index] = bkt[i][j];
            index++;
        }
    }
}

```

Best case \rightarrow uniformly distributed data
 $O(n)$

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

Total range
 $= 0 - 99$
buckets = 5

Shell Sort

Variation of insertion sort, we initialise gap as $\frac{n}{2}$ and then decrease it to 1.

\Rightarrow 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

$\uparrow_i \rightarrow$ $\uparrow_j \rightarrow$

increment i and j till j reaches $n-1$.
and if swapping happens ie $(arr[i] > arr[j])$,
we have to compare it with previous index
~~then~~ ie $i - \text{gap}$

After one pass decrement gap $\rightarrow \text{gap}/2$ and repeat the same till gap reaches ≥ 1

23	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
7	9	5	23	29	15	19	31	

} Pass-1

gap \rightarrow gap/2 (2)

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

} Pass 2

gap \rightarrow gap/2 (1)

2	5	7	9	15	19	23	29	31
2	5	7	9	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] > arr[i]) {  
                break;  
            }  
            else {  
                swap(arr[i+gap], arr[i]);  
            }  
        }  
    }  
}.
```