SORTING

Sort in C++ STL

- The STL sort function in C++ sorts data structures which allows random access to elements.

Ex – Arrays, vectors, deque.

(we can get any random element in constant time in the above data structures).

(ex) ① `int arr[] = {10, 20, 5, 7};`

`sort(arr, arr+n);` // Ascending order.

`arr, arr+n` → iterators.

`sort(arr, arr+n, greater<int>);`

// Descending order.

②
```
vector<int> v{5, 7, 20, 10};
sort(v.begin(), v.end());
sort(v.begin(), v.end(), greater<int>);
```

- We can also make our own sorting function depending upon our requirements, by changing the third parameter of the sort function.

Example

```
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);
    for (auto i : arr)
        cout << i.x << " " << i.y << endl;
}
```

O/P :
```
2 8
3 10
5 4
```

Time complexity : worst and average cases $O(n \log n)$.

→ Uses IntroSort (Hybrid of QuickSort, HeapSort and insertion sort).

Stability in Sorting Algorithm

(ex) arr[] = {("Anil", 50), ("Ayan", 80), ("Piyush", 50), ("Ramesh", 80)}

- The given array is sorted in lexicographical order of the names.
- our task is to sort the array in increasing order of the marks (second element).
  Now we clearly see there are repeating elements in the array in terms of marks.
- So, a stable sorting algorithm will be one which sorts in increasing order of marks as well as keeping the lexicographical order of names when even the marks are same.

So, we would get

{("Anil", 50), ("Piyush", 50), ("Ayan", 80), ("Ramesh", 80)}

↳ stable sort.

Example of stable sorts :-

→ Bubble sort
→ Insertion sort
→ Merge sort

Examples of unstable sorts :-

→ Selection sort
→ Quick sort
→ Heap sort.

Different Sorting Algorithms :-

① Bubble Sort

- It always compares adjacent elements.

(ex)

2, 10, 8, 7
2, 10, 8, 7
2, 8, 10, 7
2, 8, 7, 10

→ 1st pass (maximum element, i.e 10, reaches the final position)

2, 8, 7, 10
2, 8, 7, 10
2, 7, 8, 10

→ 2nd pass (second maximum element, i.e 8, reaches it's final position)

2, 7, 8, 10
2, 7, 8, 10

→ 3rd pass (third largest element, i.e 7, reaches final position)

(*) For 'n' numbers in an array, we would require (n-1) passes to sort the array using bubble sort technique.

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

No. of operations = $(n-1) + (n-2) + \cdots + 2 + 1$

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

∴ Time complexity : $\Theta(n^2)$

(Optimisation)

```
void bubbleSort( int arr[], int n){
    for( int i=0; i< n-1; i++){
        bool swapped = false;
        for( int 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;
    }
}
```

Time complexity : $O(n^2)$

⑤ Selection Sort

- $O(n^2)$ Algorithm.
- Does less memory writes compared to QuickSort, MergeSort, Insertion Sort, etc. But cycle sort is optimal in terms of memory writes.
- Basic idea for HeapSort.
- Not stable
- In-place algorithm (i.e doesn't require extra space)

(*) The basic idea of this algorithm is to keep on finding the minimum elements and then putting them in increasing order.

(Ex) {10, 5, 8, 20, 2, 18}

↓

	TEMP
{10, 5, 8, 20, INF, 18}	{2, - - - - - -}
{10, INF, 8, 20, INF, 18}	{2, 5, - - - -}
{10, INF, INF, 20, INF, 18}	{2, 5, 8, - - -}
{INF, INF, INF, 20, INF, 18}	{2, 5, 8, 10 - -}
{INF, INF, INF, 20, INF, INF}	{2, 5, 8, 10, 18, -}
{INF, INF, INF, INF, INF}	{2, 5, 8, 10, 18, 20}

(naive implementation)

```
void selectSort( int arr[], int 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] = INT_MAX;
    }
    for( int i=0; i<n; i++)
        arr[i]= temp[i];
}
```

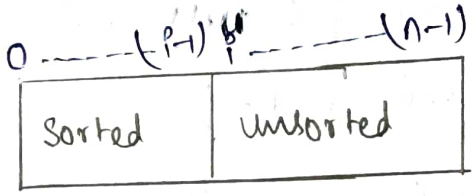
(optimisation)

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

③ Insertion Sort

- $O(n^2)$ worst case
- In-place and stable
- Used in practice for small arrays.
  (Timsort and Introsort)
- $O(n)$ in the best case.

IDEA



ex

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

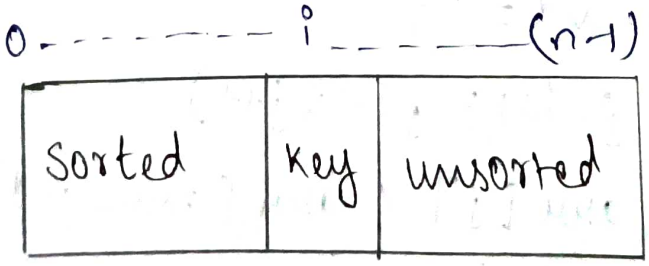
[5, 20, 40, 60, 10, 30]

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

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

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

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



i goes from (1 to n-1)

```
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;
    }
}
```

④ Merge Sort

- Divide and conquer algorithm.
  (Divide, conquer and merge).
- stable algorithm.
- $O(n \log n)$ Time and $O(n)$ Auxiliary space.
- well suited for linked lists. works in $O(1)$ auxiliary space
- used in external sorting
- In general for arrays, QuickSort outperforms merge sort.

8) Merge Two Sorted arrays-

(ex)

I/P: a[] = {10, 15, 20}

b[] = {5, 6, 6, 15}

O/P: 5 6 6 10 15 15 20

I/P: a[] = {1, 1, 2}

b[] = {3}

O/P: 1 1 2 3

(Naive solution)

```
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] << " ";
}
```

Time complexity: $O((m+n) * \log(m+n))$

Space complexity: $O(m+n)$

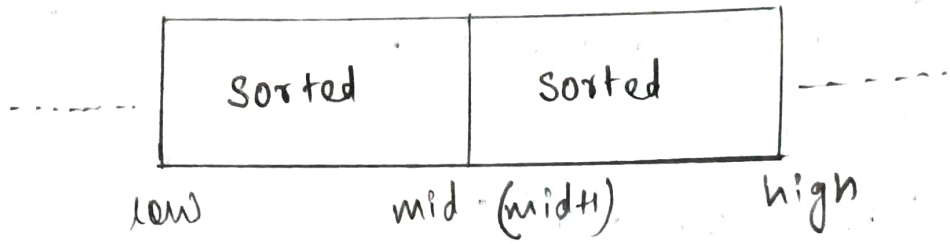
(Efficient approach)
(Traversing both arrays at same time)

```
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++;
    }
}
```

Time complexity : $O(m+n)$

Space complexity : $O(1)$

8) Merge Function of Merge Sort



we are given three values → low, mid and high.

From [low, mid] → array is sorted

From [mid+1, high] → array is sorted.

we need to sort the whole array.

(ex)

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}

(low <= mid < high)

IDEA (Same as last question)

```
void merge ( int a[], int low, int mid, int high){
    int n1 = mid - low+1;
    int n2 = high - mid;
    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+1+i];
    int i=0, j=0, k=low;
    while ( i<n1 && j<n2){
        if ( left[i] <= right[j]){
            a[k] = left[i];
            i++;
            k++;
        }
        else {
            a[k] = right[j];
            k++;
            j++;
        }
    }
    while (i<n1){
        a[k] = left[i]; i++; k++;
    }
    while ( j<n2) {
        a[k] = right[j]; j++; k++;
    }
}
```

Time : $\Theta(n)$ Aux. Space : $\Theta(n)$

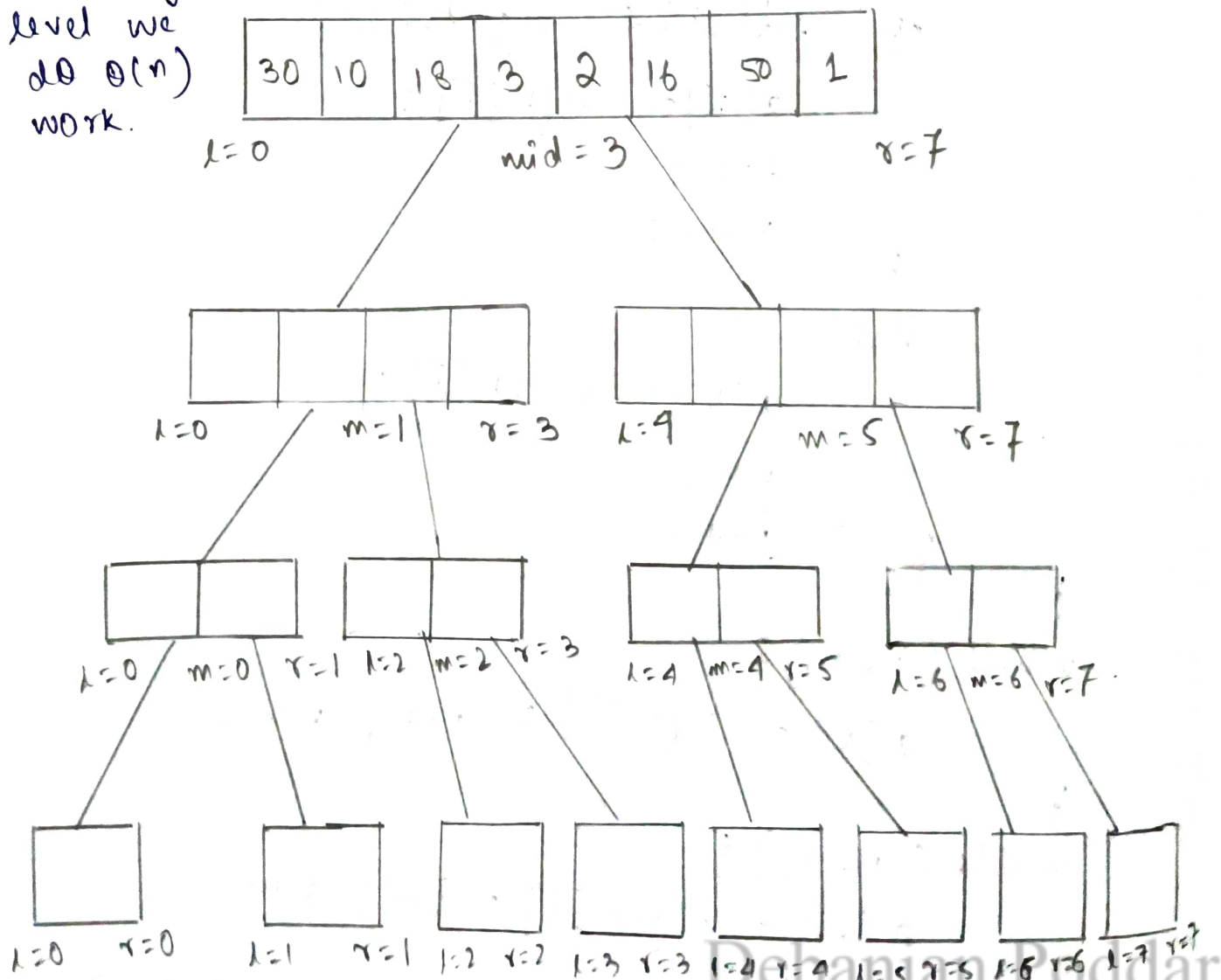
Merge Sort algorithm

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

↓

// This merge function is same as the last question.

At every level we do $\Theta(n)$ work.



Total work = work on each level × no. of levels.

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

$= \Theta(n \log n)$

∴ Time complexity: $\Theta(n \log n)$.

Auxiliary space: $\Theta(n)$

{ The space is $\Theta(n)$ instead of $\Theta(n \log n)$ because after every merging, space is deallocated }

Q) Intersection of two sorted array.

(ex) 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)

Traverse the array 'a[]', and correspondingly check for the value in array b[].

Time complexity: $O(n \times m)$

```
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 approach)

IDEA → using the merge function of merge sort.

```
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;
        }                                   // Time : O(m+n)
        if ( a[i] < b[j])
            i++;
        else if ( a[i]> b[j])
            j++;
        else {
            cout << a[i]<<" "; i++; j++;
    } } }
```

Q) Union of two sorted arrays -

(ex) 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 approach)

```
void union ( 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++)
        if ( i==0 || c[i] != c[i-1])
            cout << c[i] << " ";
}
```

Time complexity: $O((m+n)\log(m+n))$

Auxiliary space: $O(m+n)$

(Efficient approach)

IDEA → using the idea of merge function from merge sort algorithm.

```
void union( 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++;
    }
}
```

Time complexity: $O(m+n)$

Space complexity: $O(1)$

8) Count Inversions in an Array.

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

(ex) 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

I/P: { 40, 30, 20, 10}

O/P: 6 (40,30)
(40,20)
(40,10)
(30,20)
(30,10)
(20,10)

If array is sorted in decreasing order then number of inversions is equal to $\frac{n(n-1)}{2}$

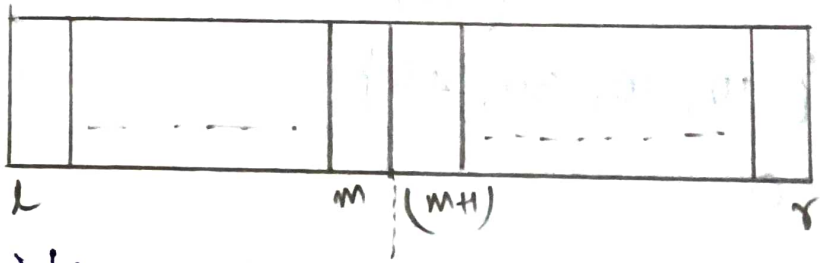
(naive approach)

```
int countInversions(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;
}
```

Time complexity: $O(n^2)$

(Efficient approach)

IDEA → using Merge sort concept.



$m = (l+r)/2$

- Every inversion (x, y) where $x > y$ has possibilities —
  (a) Both x and y are in left half
  (b) Both x and y are in right half
  (c) x is in left half and y is in right half.

```
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 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;
}
```

Time complexity : $O(n \log n)$
Auxiliary space : $O(n)$

8) Partition of a given array -

Given an index p.
Our goal is to make the array such that all the elements before a[p] become smaller (<=) and all elements after a[p] become larger than a[p]

(ex) I/P : arr[] = { 3, 8, 6, 12, 10, 7 }
p = 5 // Index of last element

O/P : arr[] = { 3, 6, 7, 8, 12, 10 }

OR

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

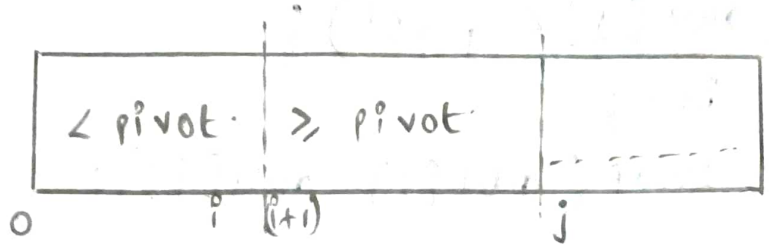
(Naive approach)

```
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];
}
```

Lomuto Partition

(for this we consider pivot as the last element)

IDEA



while traversing the array we make sure that the elements from $(0 \rightarrow i)$ are smaller than pivot and from $((i+1) \rightarrow j)$, elements are greater than pivot (considering current index as j). (for better understanding consider dry run of an example).

i → keeps track of window of smaller element.

while traversing we already know that the $(i+1)^{th}$ element is greater than the pivot element.

so, if we find (arr[j] < pivot),

then we increase size of the elements smaller than pivot (window size), i.e. we increase i (i.e it becomes (i+1) which contains a greater element than pivot), and then simply swap arr[j] with arr[i].

```
int Partition(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);
}
```

DRY RUN

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

l=0, i=-1 pivot=70 h=6

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])

(i+1)th element is surely > pivot.

what if pivot is not the last element ??

- we just need to swap (arr[p], arr[h]), such that the pivot goes to the end. And then follow the same algorithm.

Hoare's partition

↳ works much better than Lomuto partition.

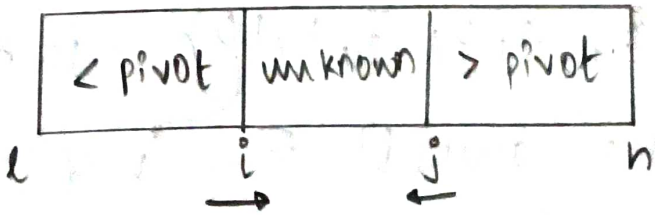
In this we consider first element as pivot element.

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

low=0

pivot = arr[l] = 5

we consider, i = l-1 and j = h+1



During traversal we make sure that, from (low to i), elements are less than pivot and from (j to high), elements are greater or equal to the pivot.

```
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]);
    }
}
```

DRY RUN

ex) i=-1 { 5, 3, 8, 4, 2, 7, 1, 10 } j=8

l=0 pivot = arr[l] = 5 h=7

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 j ≤ i now

- Here we see that the pivot element is not at it's correct place as we got in Lomuto partition. It is the main difference in the two partition method.
Here after the completion of the algorithm, it is ensured that, all the elements before j are <= pivot and all elements after j are >= pivot.

- In general Hoare's partition works much better than Lomuto partition as it does less no. of comparisons.

- Quicksort is a unstable algorithm because both Hoare's and Lomuto partition algorithms are unstable.
But the naive way of partitioning, though not effiient but it is a stable algorithm.

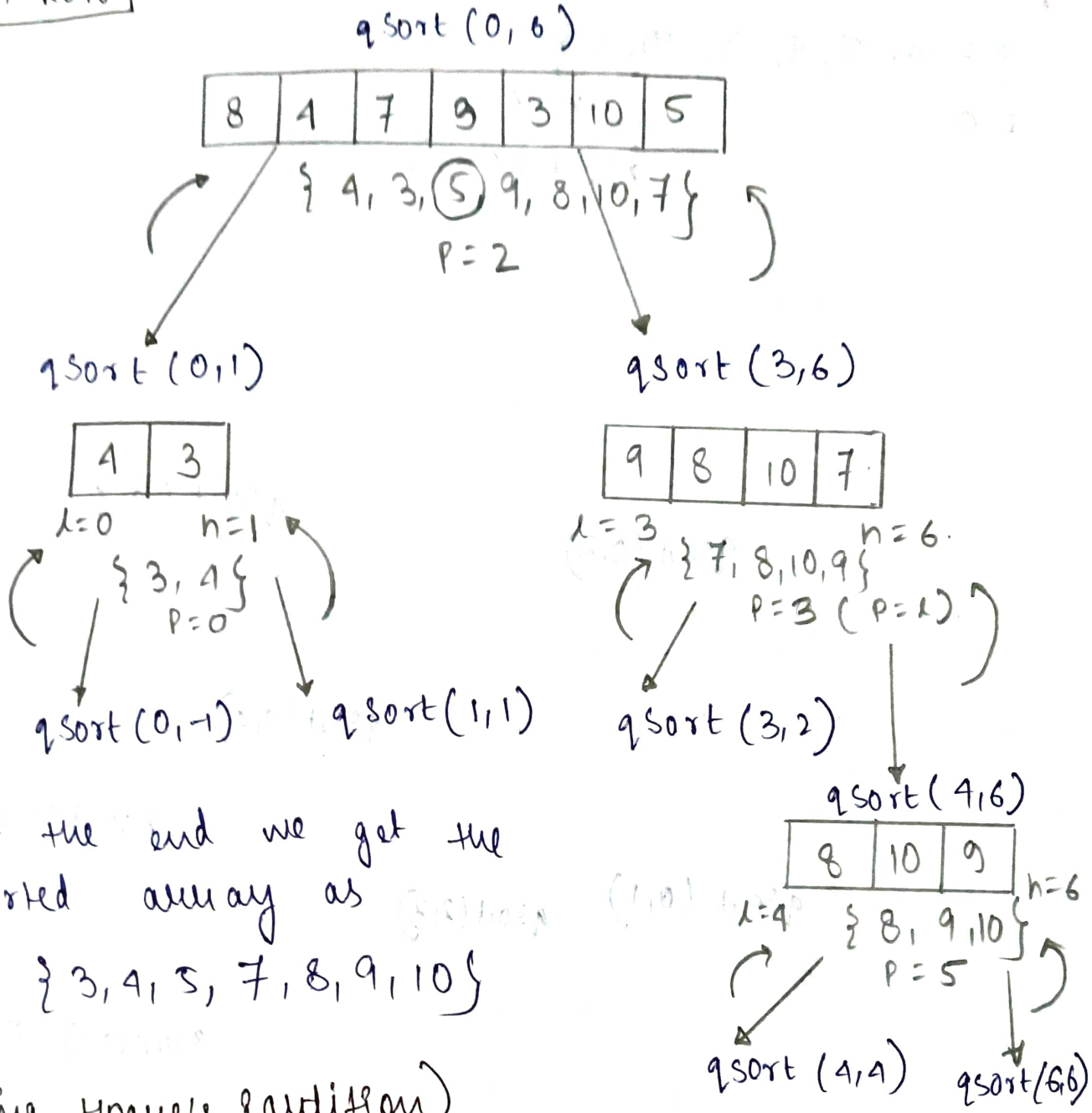
(5) Quick Sort

- Divide and conquer algorithm
- worst case time : $O(n^2)$
- Despite $O(n^2)$ worst case, it is considered fastest, because of the following reasons -
  - (a) In-place
  - (b) Cache friendly
  - (c) Average case is $O(n \log n)$
  - (d) Tail Recursive
- Partition is key function (Naive, Lomuto, Hoare).

(Using Lomuto Partition)

```
void qSort( int arr[], int l, int h) {
    if (l<h) {
        int p = partition( arr, l, h) ;
        // Partition function is the Lomuto
        // partition algorithm.
        // It returns index of pivot in final array
        qSort( arr, l, p-1) ;
        qSort( arr, p+1, h) ;
    }
}
```

DRY RUN



At the end we get the sorted array as

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

(Using Hoare's 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);
    }
}
```

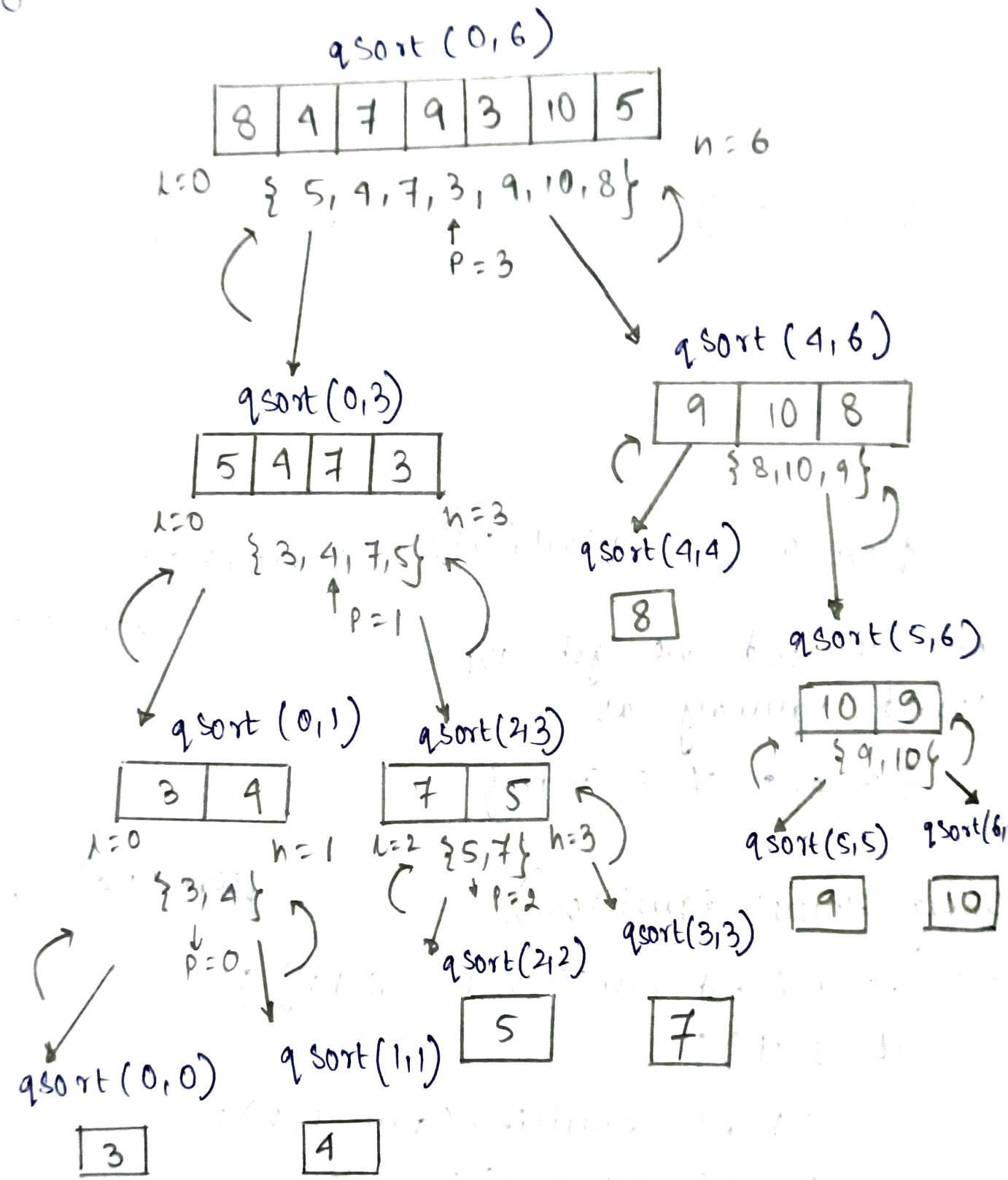
It is not (p-1) because here it is not guaranteed that the pivot will be at correct position.

DRY RUN

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

l = 0

n = 6

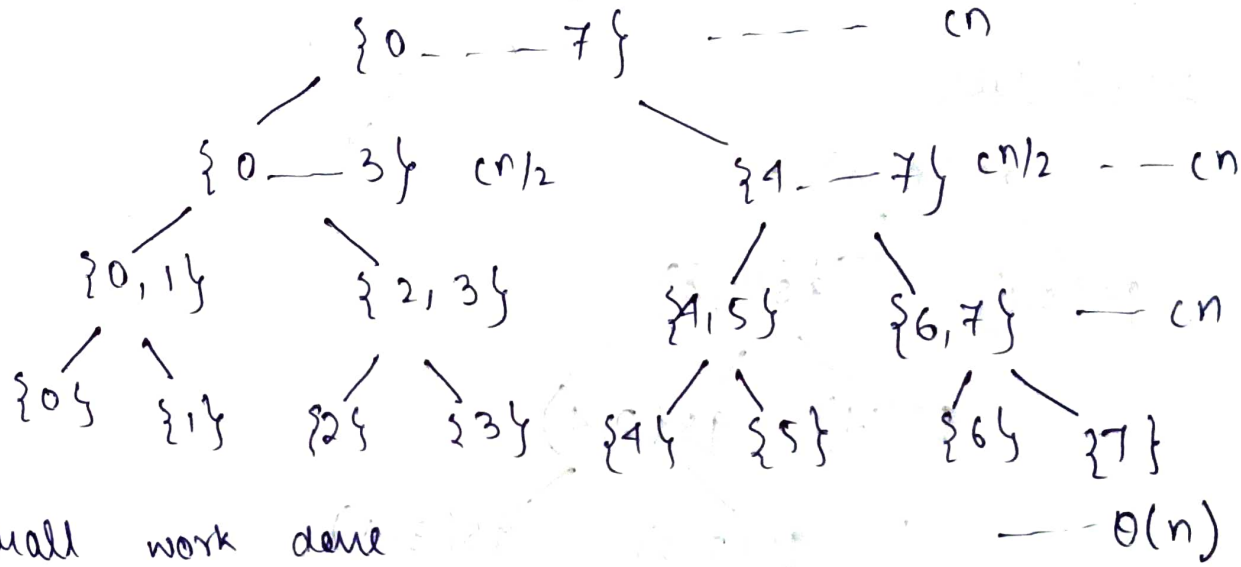


Analysis of quick sort

① **Best case**

When the partition divides the array of equal length.

(ex) $\{\overset{l}{0} \text{---} \overset{h}{7}\}$



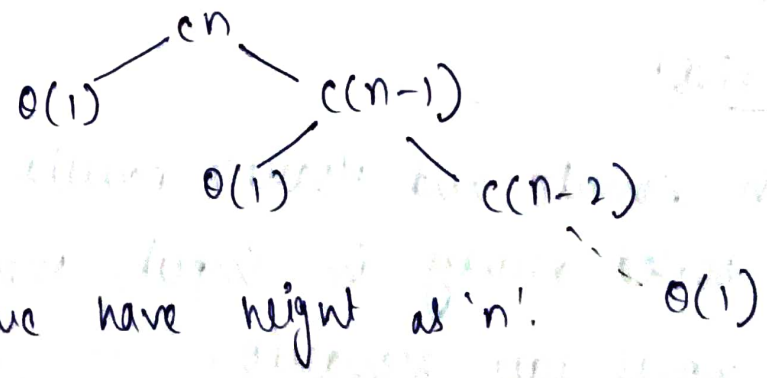
Overall work done — $\Theta(n)$

$= (cn + cn + cn \text{---} cn) + \Theta(n)$

$= cn(\log_2 n) + \Theta(n) = \Theta(n \log n)$

② **Worst case**

When the partition happens such that 1 element is at one part and rest $(n-1)$ elements on other part.



In this we have height as 'n'.

Overall work

$= n[n + (n-1) + (n-2) + \text{---} 1] + \Theta(n)$

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

Best case : $T(n) = 2T(n/2) + \Theta(n)$

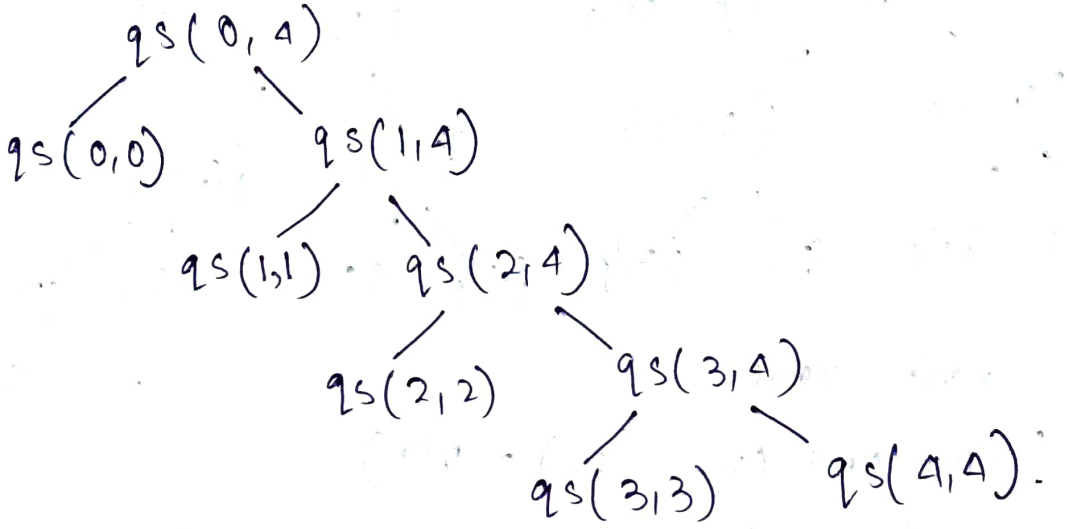
Worst case : $T(n) = T(n-1) + \Theta(n)$

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

Space complexity of quick sort

Auxiliary space is required for recursion call stack.

worst case



At most, n recursion call stack will be there at a a time, so auxiliary space is $\Theta(n)$.

Best case

$\Theta(\log n)$ is the maximum space required.

Choice of pivot

In both Lomuto and Hoare's partition, if we give a sorted array in input, then it will be the worst case scenario.

Both will partition with 1 element on one side and $(n-1)$ on other side.

due to this reason, most programming languages use a random pivot and place it suitably at the first or last position of array and then use the standard algorithms.

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;
        }
}
```

8) K^th Smallest element

(ex) I/P: {10, 5, 30, 12}

K = 2

O/P: 10

I/P: {30, 20, 5, 10, 8}

K = 4

O/P: 20

(Array is unsorted).

(naive approach)

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

Time complexity: O(nlogn)

(Efficient approach) [QUICK SELECT ALGORITHM]

"[USING LOMUTO PARTITION]"

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

Time complexity:

→ Best case → O(n)

→ Worst case → $O(n^2)$

→ Average case → O(nlogn)

8) chocolate distribution problem.

Given an array in which the elements depicts the number of chocolates in a packet. Also given 'm' = no. of children.

Task is to distribute the packets among children such that -

(i) Every children gets exactly 1 packet.

(ii) we have to minimize the difference between the minimum chocolate and maximum chocolate received by the children.

(ex) I/P: {7, 3, 2, 4, 9, 12, 56}

m = 3

O/P: 2

I/P: {3, 4, 1, 9, 56, 7, 9, 12}

m = 5

O/P: 6

```
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;
}
```

↳ Basically we compare the difference of min & max element of all the possible set of size m

Time complexity: O(n log n)

8) Sort an array with two types

Different forms of the question :—

① segregate positive and negative

I/P : arr[] = { 15, -3, -2, 18 }

O/P : arr[] = { -3, -2, 15, 18 }

order doesn't matter

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, 0 }

O/P : arr[] = { 0, 0, 1, 1, 1 }

(Naive approach) (considering ①)

```
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 j=0; j<n; j++)
        arr[j] = temp[j];
}
```

Time : O(n) (Three traversals)

(Efficient solution)

IDEA : This problem is mainly a variation of partition() of QuickSort.

→ Hoare or Lomuto partition can solve this in O(n) time and O(1) auxiliary 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]);
    }
}
```

// Time complexity : O(n)
(single traversal)
Auxiliary space : O(1)

(ex) arr[] = {-12, 18, -10, 15}

i = -1 , j = 4

1st iteration : i = 1
j = 2

arr[] = {-12, -10, 18, 15}
↑ ↑

2nd iteration : i = 2
j = 1

return;

8) Sort an array with three types

Different forms of the question :-

① Sort an array of 0s, 1s and 2s.

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} → > pivot

< pivot → order doesn't matter.

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} (order doesn't matter)

3 → elements less than 5

6, 9 → elements in range 5 to 10

20, 40 → elements greater than 10

(Naive approach)
(considering ①)

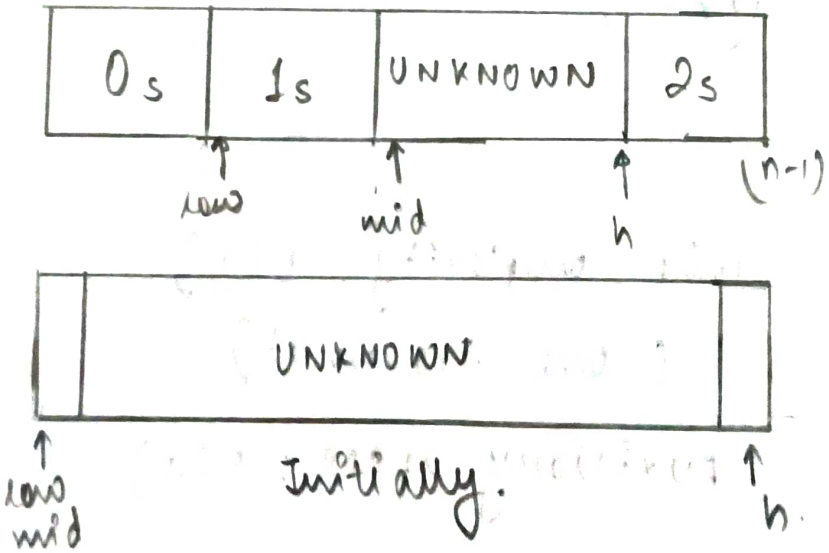
```
void sort ( int arr[], int n){
    int temp[n] , i=0 ;
    for ( int j=0 ; j<n ; j++)
        if ( arr[j]==0) {
            temp [i] = arr[j];
            i++ ;
        }
```

```
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];
}
```

Time complexity: $O(n)$
(four traversal).

(Efficient approach)

DUTCH NATIONAL FLAG ALGORITHM



This algorithm is a modification of Hoare's partition algorithm.

The idea is to bring all 1s, (the pivot element in this case) together, and then according to the partition algorithm all 0s will come before the 1s and all 2s will go after 1s.

```
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--;
        }
    }
}
```

Time complexity: O(n)
(one traversal)

Auxiliary space: O(1)

Q) Minimum difference in an Array.

(ex) 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

(Naive approach)

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

Time complexity: $O(n^2)$

(Efficient approach)

IDEA

{ 10, 3, 20, 12 }

① Sort the array

{ 3, 10, 12, 20 }

② compute differences of adjacent elements of the sorted array.

10 - 3 = 7

12 - 10 = 2

20 - 12 = 8

(3) Return the minimum difference.

```
int getMinDiff(int arr[], int n){
    sort(arr, arr+n);
    int res = INT_MAX;
    for(int i=1; i<n; i++)
        res = min(res, arr[i] - arr[i-1]);
    return res;
}
```

Time complexity: O(n log n)

Q) Merge overlapping Intervals

(ex) 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} }

→ How to check if two intervals overlap??

$i_1 = \{5, 10\}$

$i_2 = \{1, 7\}$

one way of checking is that we take the largest of the first element and check whether it exists in the other interval or not.

(ex) $\{5,10\}$ $\quad \max(1,5) = 5$

$\{1,7\}$ $\quad$ and $5 \in \{1,7\}$

so they overlap.

(Naive approach)

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

In this approach we traverse the array with nested loop (comparing each interval with other intervals. If we see they overlap, then we merge them with the current interval and delete the interval which we merged.

Here, $\{5,10\}$ and $\{6,8\}$ overlap, so we get

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

$\{\{1,10\}, \{2,3\}\} \Rightarrow \{\{1,10\}\}$

Time complexity : $O(n^2) \times O(n) = O(n^3)$

$O(n^2)$ ↓ for traversal

$O(n)$ ↳ for deletion of interval

for merging of intervals,

we get

start = min (i_1.start, i_2.start)

end = max (i_1.end, i_2.end).

(Efficient approach)

Sort by start time in increasing order.

Sorted in increasing order of start time : $x_1, x_2, x_3, \ldots x_{i-1}, x_i, \ldots$

Merged from (x_1 to x_{i-1}) : $m_1, m_2 \ldots m_{j-1}, m_j$

(ex)

$m_{j-1} = \{5, 10\}$

$m_j = \{11, 20\}$

$x_i = \{15, 30\}$

considering current index 'i'

clearly,

$x_i.start \geq m_j.start$ — (1)

$m_{j-1}.end < m_j.start$ — (2)

(as because they are not merged)

From (1) and (2), it can be concluded that

$$x_i.start > m_{j-1}.end \quad — (3)$$

Hence this gives the assurance that we just need to check the x_i^{th} interval with last merged interval (i.e m_j) ONLY and not the merged intervals before m_j.

If condition (3) would have failed, then only, m_j and m_{j-1} would have gotten merged before x_i.

```
struct Interval {
    int st, end;
};
void mergeIntervals( Interval arr[], int n) {
    sort ( arr, arr+n, myComp);
    // myComp is a comparator which sorts in
    // increasing order of start value of interval.
    // we have to write the myComp comparator.
    int res=0;
    // Intervals from 0 to res are already merged.
    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 << "\n"
}
```

8) Meeting Maximum guests

The arrival and departure time of different guests are given who are going to attend a party. You have to tell the maximum number of guests that you can meet.

(ex) I/P: arr[] = {900, 940}
dep[] = {1000, 1030}

$0 \leq arr[i], dep[i] \leq 2359$

O/P: 2 (940 → 1000, we would be able to meet both the guests)

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

O/P: 3 (From 800 → 840, first 3 guests would be there)

I/P: arr[] = {900, 940, 950, 1100, 1500, 1800}
dep[] = {910, 1200, 1120, 1130, 1900, 2000}

O/P: 3 (From 1100 — 1120)

(Efficient approach)

arr[] = {900, 600, 700}

dep[] = {1000, 800, 730}

we sort both the arrays.

arr[] = {600, 700, 900}

dep[] = {730, 800, 1000}

Time	state	guests
600	A	1
700	A	2
730	D	1
800	D	0
900	A	1
1000	D	0

we use the merge function to implement the above table.

```
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;
    // we start with i=1, as the arrival of first
    //guest is always considered.
    while ( i<n && j<n){
        if ( arr[i] <= dep[j]) {
            curr++;
            i++;
        }
        else {
            curr--;
            j++;
        }
        res = max (res, curr);
    }
    return res;
}
```

time complexity: O(nlogn)

Miscellaneous sorting techniques

① Cycle Sort

→ A worst case $O(n^2)$ sorting algorithm.

→ Does minimum memory writes and can be useful for cases where memory write is costly.

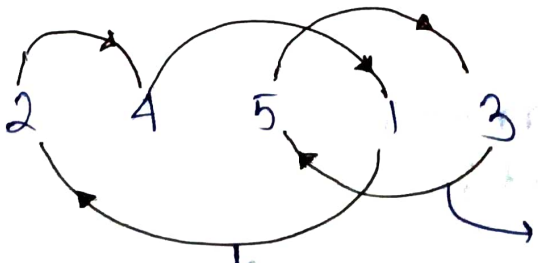
→ In-place and not stable algorithm.

→ Useful to solve questions like - find minimum number of swaps required to sort an array.

IDEA

It is based on the idea that array to be sorted can be divided into cycles. Cycles can be visualized as a graph. There are n nodes and and edge directed from node i to node j if the element at i^{th} index must be placed at the j^{th} index in the sorted array.

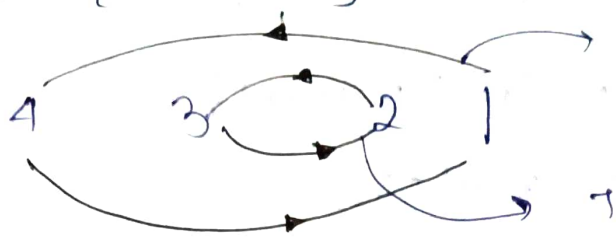
arr[] = {2, 4, 5, 1, 3}



This is a cycle

This is another cycle

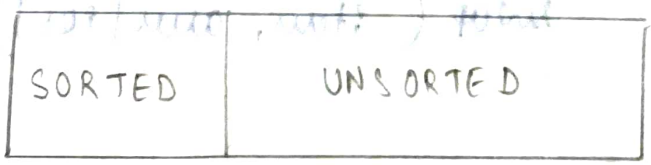
arr[] = { 4, 3, 2, 1 }



This is a cycle

This is another cycle.

We one by one consider all cycles. We first consider the cycle that includes first element. We find correct position of first element, place it at it's correct position, say j. We consider old value of arr[j] and find it's correct position, we keep doing this till all elements of current cycle are placed at correct position, i.e we don't come back of starting point.



cs: (cycle start) is the current index

Elements from 0 to (cs-1) are already sorted as they must have been part of previous cycles.

From cs to (n-1), some elements may be sorted due to previous cycles.

pos: tells how many elements are smaller than arr[cs]

```
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]);
        }
    }
}
```

ex

cycle 1

20 40 50 10 30

pos = 0
item = 20

pos = 1
item = 40

20 20 50 10 30
pos = 3

pos = 3

20 (20) 50 (40) 30

item = 10

pos = 0

(10) (20) 50 (40) 30

item = 2

pos = 0 = cs (end of cycle (1))

cycle 2

(10) (20) 50 (40) 30

cs = 1, pos = 1, item = 20

(10) (20) 50 (40) 30

cs = 2, pos = 2, item = 50

pos = 4

(10) (20) 50 (40) (50)

cs = 2, item = 30

pos = 2

(10) (20) (30) (40) (50) item = 50.

cs = 3

cs = 4 ... loop ends.

EXERCISE

Q1) Modify the cycle sort function so that it can take duplicate elements also in the input array.

Q2) write a code which tells us the minimum number of swaps required to sort an array.

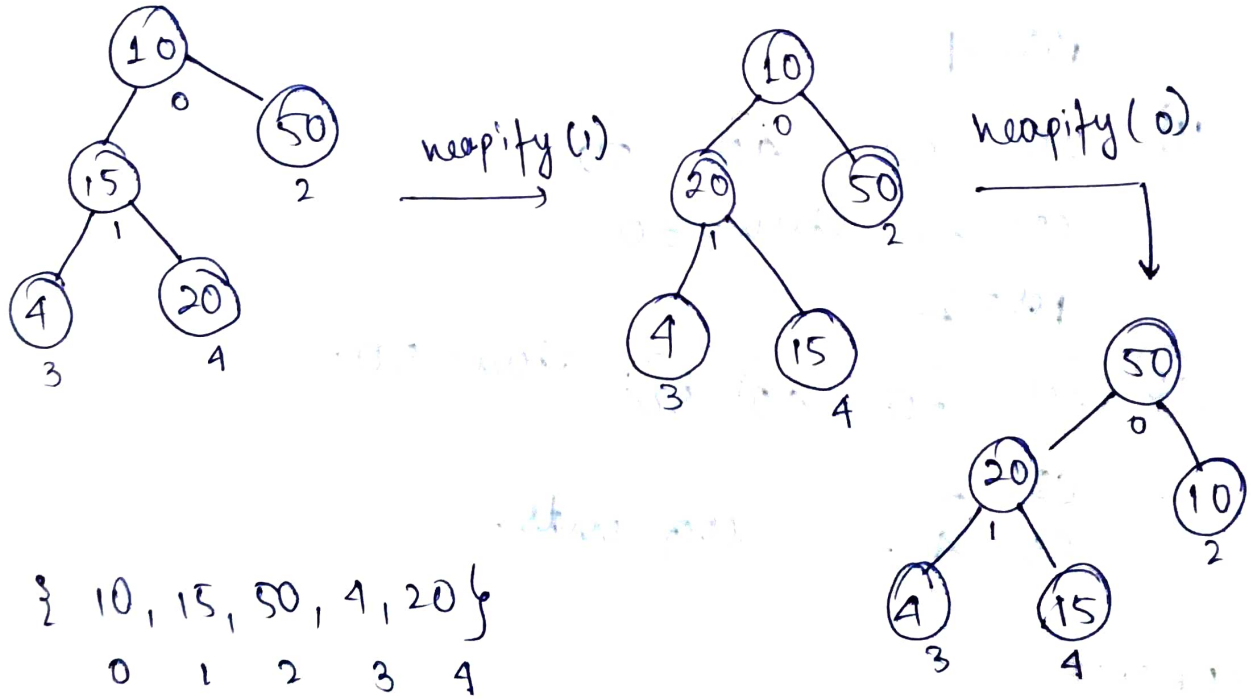
(2) Heap sort

(Optimization on selection sort)

- It uses max heap data structure to find the maximum element in the array
unlike in selection sort where we use linear search to find the maximum element, here we build a max Heap.

Step 1 : Build a max heap.

(ex) { 10, 15, 50, 4, 20 }
0 1 2 3 4



{ 10, 15, 50, 4, 20 }
0 1 2 3 4

{ 10, 20, 50, 4, 15 }
0 1 2 3 4

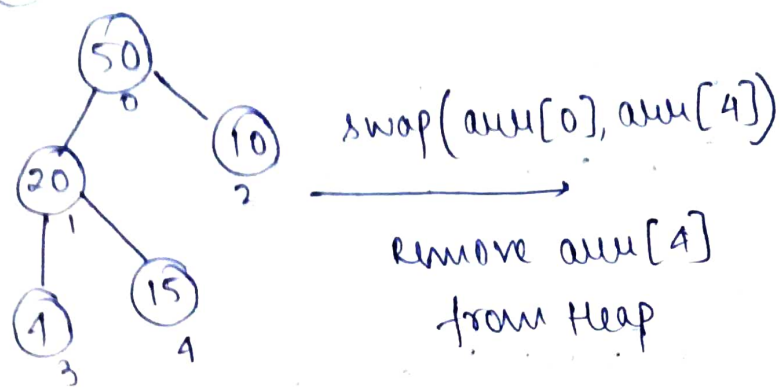
{ 50, 20, 10, 4, 15 }
0 1 2 3 4

```
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){
        swap( arr[largest], arr[i]);
        maxHeapify( arr, n, largest);
    }
}
void buildHeap(int arr[], int n){
    for( int i = (n-2)/2 ; i>=0; i--)
        maxHeapify(arr, n, i);
}
```

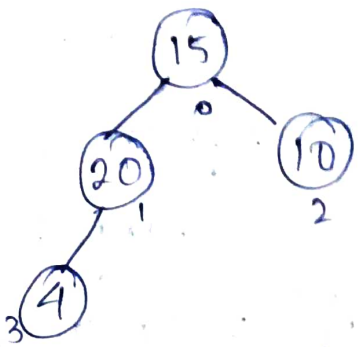
STEP 2 : Repeatedly swap root with last node, reduce heap size by one and heapify.

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

(8x)

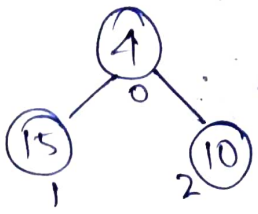


{50, 20, 10, 4, 15}

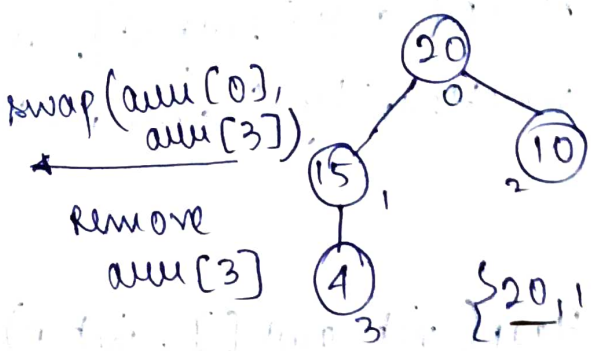


{15, 20, 10, 4, 50}

↓ Heapify(0)

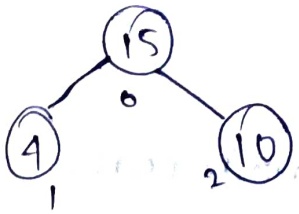


{4, 15, 10, 20, 50}

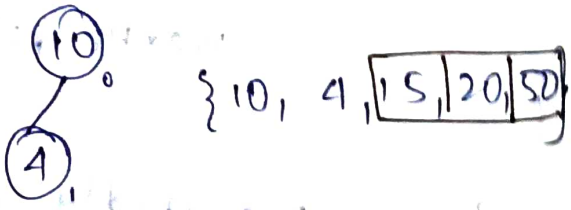


{20, 15, 10, 4, 50}

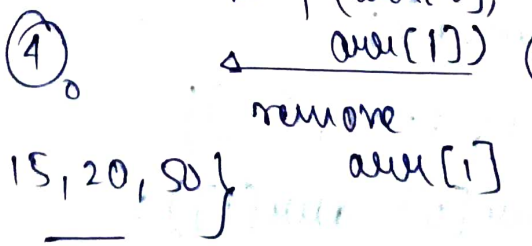
↓ Heapify(0)



{15, 4, 10, 20, 50}



↓ heapify(0)



{4, 10, 15, 20, 50}

Time complexity: O(nlogn)

(3) Counting sort

- It is not a comparison based algorithm.
- It takes $O(n+k)$ time to sort n elements in range 0 to $(k-1)$, where k is linear in comparison to n.

(ex) 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 approach)

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

We just count how many times each element occurs in the corresponding index in the count array

Time complexity: $O(n+k)$

→ Problems with the approach

cannot 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++)          ↷ Prefix sum
        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];
}
```

Here we are actually using the objects instead of using the indexes.

- For any index 'i', count[arr[i]] gives us the number of elements smaller than or equal to arr[i].

(ex) n = 4, k = 7

arr[] = { 5, 6, 5, 2 }

count[] = { 0, 0, 1, 0, 0, 2, 1 }

After prefix sum,

count[] = { 0, 0, 1, 1, 1, 3, 4 }

0 1 2 3 4 5 6

output[] = { _, _, _, _ }

i = 3

arr[3] = 2, count[arr[3]] = count[2] = 1

∴ 1 element is less than equal to arr[3] = 2

So, we place arr[3] = 2 at 1st place, i.e at 0th index.

{ 2, _, _, _ } ← output[]

count[] = { 0, 0, ⓪, 1, 1, 3, 4 }

i = 2

output[] = { 2, _, 5, _ }

count[] = { 0, 0, 0, 1, 1, ② 4 }

i = 1

output[] = { 2, _, 5, 6 }

count[] = { 0, 0, 0, 1, 1, 2, ③ }

i = 0

output[] = { 2, 5, 5, 6 }

count[] = { 0, 0, 0, 1, 1, ⓪, 3 }

Important points

(i) Not a comparison based algorithm

(ii) $\Theta(n+k)$ time

(iii) $\Theta(n+k)$ Auxiliary space

(iv) Stable

(v) Used as a sub routine in Radix sort.

Exercise:

Extend the given implementation to work for an arbitary range of size k.

(4)

Radix Sort

- Radix sort algorithm is an improvement in the counting sort algorithm.

Counting sort algorithm takes $O(n+k)$ time. It is feasible to be used only when order of k is linear or else it becomes worse than algorithms like merge sort, quick sort etc.

But radix sort algorithm does an optimization in such a way that, no matter what the order of k is, still it would be able to sort the array in linear time.

- (ex) { 319, 212, 6, 8, 100, 50}

STEP 1: Re-writing the numbers with leading zeros such that the length of each numbers are same.

we get,

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

STEP 2 : stable sort according to last digit (LSD)

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

STEP 3 : stable sort according to the middle digit

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

STEP 4 : stable sort according to the first digit (MSD)

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

(Implementation)

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

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

mx = 319

calls

countingSort (arr, 6, 1) n=6, exp=1

countingSort (arr, 6, 10) n=6, exp=10

countingSort (arr, 6, 100) n=6, exp=100

```
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=1; i<10; i++){
        count[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];
}
```

this gives us the different digits at different places (index)

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

count[10] = { 2, 0, 1, 0, 0, 0, 1, 0, 1, 1 }

(Prefix array) 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 }

Time complexity : $\boxed{\Theta(d * (n+b))}$

$\approx \Theta(d * n)$

- b can be ignored because it is already known that b = 10 as the digits vary from 0 to 9.

d → no. of digits in the largest number

n → length of array

Auxiliary space : $O(n+b)$

Interesting fact

There is a trade off between time and space complexity in radix sort. If we increase b (the base), then automatically d will decrease and hence time complexity will decrease but again the size of count[] array will increase which would increase auxiliary space required.

⑤ Bucket Sort

- consider the following situations –

① consider a situation where we have numbers uniformly distributed in range from 1 to 10^8.

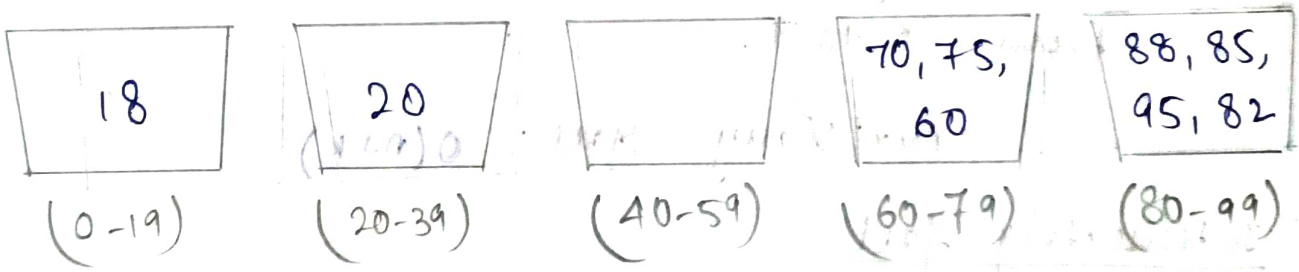
How do we sort efficiently

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

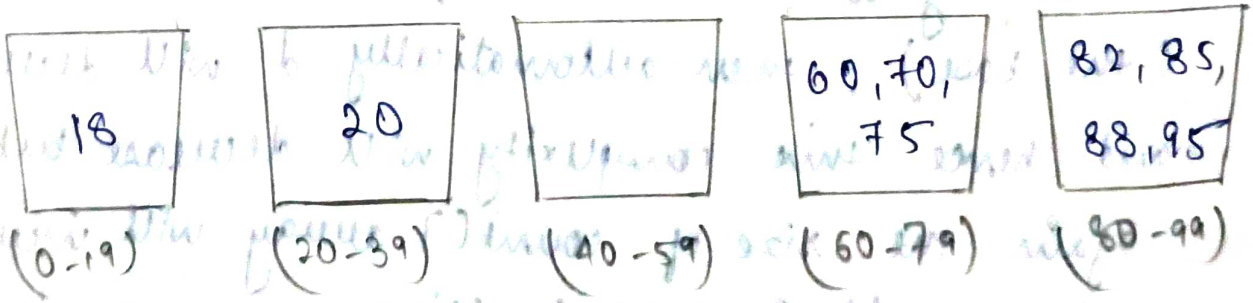
ex I/P: {20, 88, 70, 85, 75, 95, 18, 82, 60}

Range → 0 to 99.

STEP 1 : Scatter



STEP 2 : Sort buckets



STEP 3 : Join the sorted buckets

{18, 20, 60, 70, 75, 82, 85, 88, 95}

In general,

Let there are total n elements and k buckets in total (no. of buckets is taken by us). Then on average each bucket will have n/k elements.

(considering uniform distribution)

Now sorting each bucket will take O(1) time on average, because order of k is close to n.

For example if k = n/3, then no. of elements in each bucket will be n/k = 3 elements which can be sorted in close to constant time.

Hence this uniform distribution of elements is the key feature of this sorting algorithm.

(Ex) I/p : arr[] = {20, 80, 10, 85, 75, 99, 18}

k = 5

(For simplicity, we assume that no. of buckets will be given as input)

How to find total range?

We make the range from 0 to maximum element of the array.

In this case, range → (0 - 99)

∴ No. of elements in each 100/5 = 20 elements

So, the distribution is like,

(0-19), (20-39), (40-59), (60-79), (80-99)

```
void bucketSort( int arr[], int n, int k){
    int max_val = arr[0];
    for( int i=1; i<n; i++)
        max_val = max( 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];
}
```

(ex) {30, 40, 10, 80, 5, 12, 70}

k = 4

0	10	5	12
1	30	40	
2			
3	80	70	

(0-20)
(21-40)
(41-60)
(61-80)

sort buckets

0	5	10	12
1	30	40	
2			
3	70	80	

Join buckets

arr[] = { 5, 10, 12, 30, 40, 70, 80 }

Time complexity

Assuming $k \approx n$

Best case : Data is uniformly distributed.

$O(n)$

Worst case : All items go into a single bucket.
If we use insertion sort to sort the individual buckets, then $O(n^2)$.
If we use $O(n \log n)$ algorithm, then it will be $O(n \log n)$

Overview of Sorting Algorithm

① Binary array
(Application of partition function of quick sort)

② Array with three value.
(Application of partition function of quick sort).

③ Array of size n and small ranged value.
(counting sort)

④ Array of size n and range is of size n^2 or n^3 or closer. (Radix sort)

⑤ Array of uniformly distributed data over a range. (Bucket sort)

⑥ when memory writes are costly.
(Selection sort and cycle sort)

⑦ when adjacent swaps are allowed.
(Bubble sort and cocktail sort)

⑧ when array size is small.
(Insertion sort)

⑨ when available extra memory is less
(shell sort) ⟶ $O(n \log^2 n)$

→ General purpose sorting algorithms

- Merge Sort
- Heap sort
- Quick sort

→ Hybrid algorithms

- Tim Sort (Insertion + merge sort)
- Intro sort (Quick + Heap + insertion sort)

not stable.

- C++ has a function stable_sort(), if we require stability in the sorting of the input.