Bubble Sort

After every pass, we get heaviest element at the end of the array & second least at second last position & so on.

5 9 8 2 1

1.1 5 9 8 2 1

1.2 5 8 9 2 1

1.3 5 8 2 9 1

1.4 5 8 2 1 9

5 8 2 1 9

2.1 5 8 2 1 9

2.2 5 2 8 1 9

2.3 5 2 1 8 9

5 2 1 8 9

3.1 2 5 1 8 9

3.2 2 1 5 8 9

2 1 5 8 9

4.1 1 2 5 8 9

Sorted

5 9 8 2 1

1.1 5 9 8 2 1

1.2 5 8 9 2 1

1.3 5 8 2 9 1

1.4 5 8 2 1 9

4 comparisons

(n-1) comp

5 8 2 1 9

2.1 5 8 2 1 9

2.2 5 2 8 1 9

2.3 5 2 1 8 9

3 comparisons

(n-2)

5 2 1 8 9

3.1 2 5 1 8 9

3.2 2 1 5 8 9

2 comparison

(n-3)

2 1 5 8 9

4.1 1 2 5 8 9

Sorted

1 comp

(n-4)

Total comp $= (n-1) + (n-2) + (n-3) \dots + 1$

$= \frac{n(n-1)}{2} = O(n^2)$ comparisons

$TC = O(N^2)$

```java
public static void bubbleSort(int[] arr) {
  //write your code here

  for(int i=0;i<arr.length - 1;i++) {
      for(int j=0;j<arr.length-i-1;j++) {
          if(isSmaller(arr,j+1,j)) {
              swap(arr,j+1,j);
          }
      }
  }
}
```

The main summary of Bubble Sort is that the heaviest element (largest) will be at its sorted position after the first iteration.

```java
public static void bubbleSort(int[] arr) {
  //write your code here
  boolean flag = false;
  for(int i=0;i<arr.length - 1;i++) {
      for(int j=0;j<arr.length-i-1;j++) {
          if(isSmaller(arr,j+1,j)) {
              swap(arr,j+1,j);
              flag = true;
          }
      }

      if(flag == false) {
          //Array is already sorted hence no swap occured
          break;
      }
  }
}
```

So, the largest elements keep on moving at the last of the array.

Best Case : Array Already Sorted

Best Case Time : $O(n)$ if this code is used else $O(n^2)$ only.

Selection Sort

At i^{th} iteration, do linear search & find i^{th} smallest element to put it at i^{th} index.

5 9 8 1 2

1.1 5 9 8 1 2

1.2 5 9 8 1 2

1.3 5 9 8 1 2

1.4 5 9 8 1 2

1 9 8 5 2

1 9 8 5 2

2.1 1 9 8 5 2

2.2 1 9 8 5 2

2.3 1 9 8 5 2

1 2 8 5 9

1 2 8 5 9

3.1 1 2 8 5 9

3.2 1 2 8 5 9

1 2 5 8 9

1 2 5 8 9

4.1 1 2 5 8 9

$\boxed{1\ 2\ 5\ 8\ 9}$

$\Downarrow$

Sorted

Selection Sort Code

```java
public static void selectionSort(int[] arr) {
  //write your code here
  for(int i=0;i<arr.length-1;i++) {
      int minIndex = i;
      for(int j = i + 1;j<arr.length;j++) {
          if(isSmaller(arr,j,minIndex)) {
              minIndex = j;
          }
      }
      swap(arr,i,minIndex);
  }
}
```

If array is already sorted, then also TC = $O(N^2)$

i.e. the Best Case will always be $O(N^2)$ (No improvisation can be done)

Bubble Sort is worst than selection sort because of high no of swaps.

Data Structure Operations Cheat Sheet

Data Structure Name	Average Case Time Complexity				Worst Case Time Complexity				Space Complexity
	Accessing n^{th} element	Search	Insertion	Deletion	Accessing n^{th} element	Search	Insertion	Deletion	Worst Case
Arrays	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stacks	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queues	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Binary Trees	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Balanced Binary Search Trees	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$
Hash Tables	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Note: For best case operations, the time complexities are $O(1)$.

Sorting Algorithms Cheat Sheet

Sorting Algorithm Name	Time Complexity			Space Complexity	Is Stable?	Sorting Class Type	Remarks
	Best Case	Average Case	Worst Case	Worst Case			
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Not a preferred sorting algorithm.
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	In the best case (already sorted), every insert requires constant time
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Comparison	Even a perfectly sorted array requires scanning the entire array
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Yes	Comparison	On arrays, it requires $O(n)$ space; and on linked lists, it requires constant space
Heap Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	No	Comparison	By using input array as storage for the heap, it is possible to achieve constant space
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	No	Comparison	Randomly picking a pivot value can help avoid worst case scenarios such as a perfectly sorted array.
Tree Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(n)$	Yes	Comparison	Performing inorder traversal on the balanced binary search tree.
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	Yes	Linear	Where k is the range of the non-negative key values.
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$	Yes	Linear	Bucket sort is stable, if the underlying sorting algorithm is stable.
Radix Sort	$O(dn)$	$O(dn)$	$O(dn)$	$O(d+n)$	Yes	Linear	Radix sort is stable, if the underlying sorting algorithm is stable.

In place Algorithms

Insertion Sort

Consider the given array into 2 parts one is sorted & other unsorted. One by one pick an element from unsorted region & insert it in sorted region.

sorted ↙ unsorted

2 | 9 5 1 3

2 9 | 5 1 3

2 5 9 | 1 3

1 2 5 9 | 3

sorted ↙ unsorted

1.1 2 9 | 5 1 3

2.1 2 5 9 1 3

3.1 2 5 1 9 3

4.1 1 2 5 3 9

2.2 2 5 9 | 1 3

3.2 2 1 5 9 3

4.2 1 2 3 5 9

3.3 1 2 5 9 | 3

4.3 $\boxed{1\ 2\ 3\ 5\ 9}$

⇓

sorted

```
public static void insertionSort(int[] arr) {
  //write your code here
  for(int i=1;i<arr.length;i++) {
      for(int j=i;j>0;j--) {
          if(isGreater(arr,j-1,j)) {
              swap(arr,j-1,j);
          } else {
              break;
          }
      }
  }

}
```

$O(n^2)$

$BC = O(n)$

check yaha lagaenge.

Merge 2 Sorted Array (Two pointer technique)

(M) A: 2 3 6 7 9 10

(N) B: 1 4 5 7 10 12 13 14

(M+N) C =

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	3	4	5	6	7	7	9	10	10	12	13	14

Compare A[i] & B[j] and add the smaller element into C.

After one array is completed, copy all the elements of the second array as it is.

```java
public static int[] mergeTwoSortedArrays(int[] a, int[] b){
  //write your code here
    int[] c = new int[a.length + b.length];

    int i = 0, j = 0, k = 0;

    while(i < a.length && j < b.length) {

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

    while(j<b.length) {
        c[k] = b[j];
        j++;
        k++;
    }

    while(i<a.length) {
        c[k] = a[i];
        i++;
        k++;
    }
    return c;
```

A.length = n

B.length = m

TC = O(n+m)

Merge Sort (Divide & Conquer)

0	1	2	3	4	5	6	7	8	9	10
5	2	6	3	1	9	4	8	7	2	5

faith that both get sorted using MS

{1 2 3 5 6 9} {2 4 5 7 8}

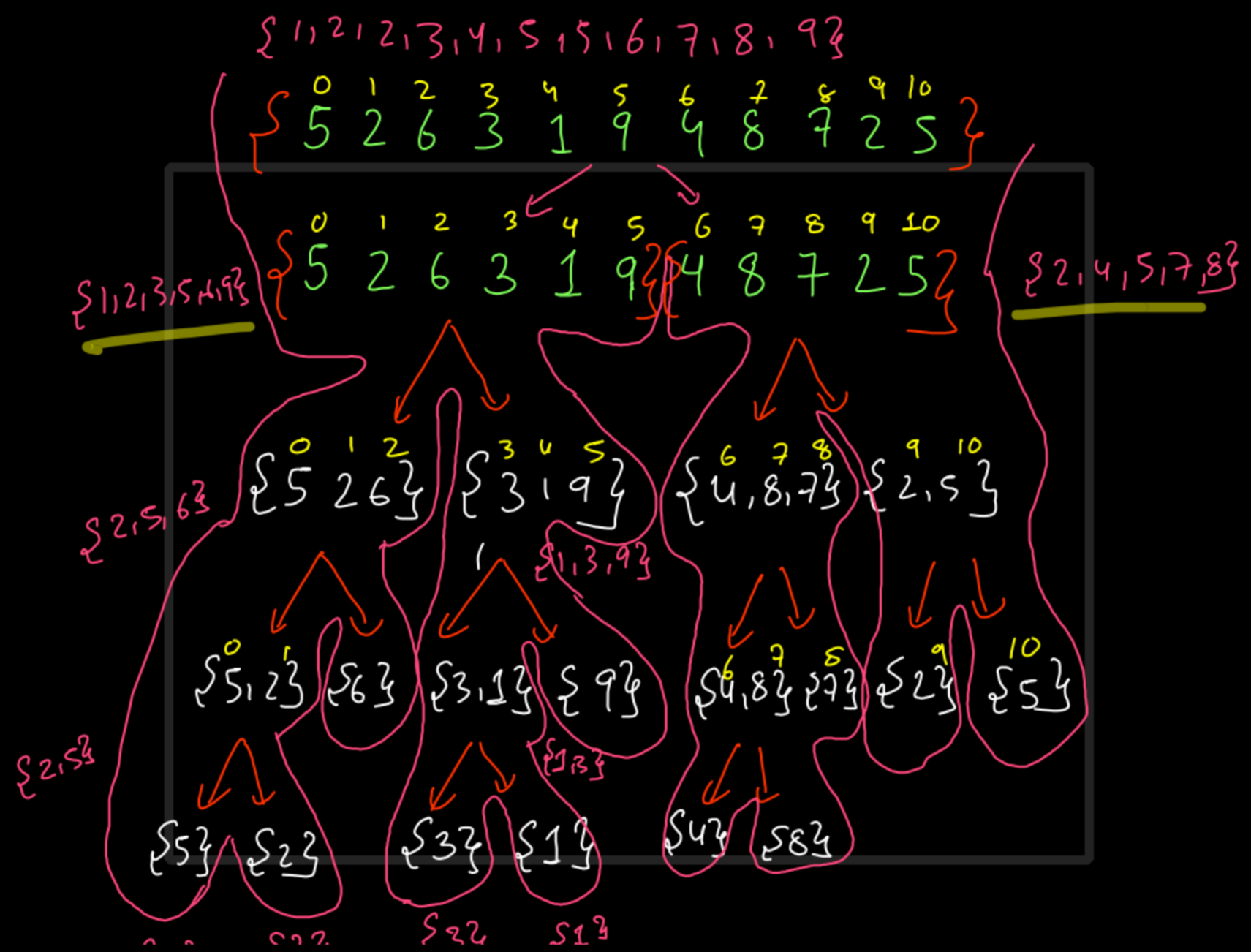
Merge them

{1 2 2 3 4 5 5 6 7 8 9}

Divide & Conquer: Divide big problem into 2 almost equally sized sub problems. (Recursion faith)

Now Conquer (Merge in this case)

Not Mandatory that conquer will be after divide only.


{1,2,2,3,4,5,5,6,7,8,9}
0 1 2 3 4 5 6 7 8 9 10
{5 2 6 3 1 9 4 8 7 2 5}
{1,2,3,5,4,9}
{5 2 6 3 1 9}{4 8 7 2 5}
{2,4,5,7,8}
{2,5,6}
{5 2 6}
{3 1 9}
{4,8,7}
{2,5}
{1,3,9}
{5,2}
{6}
{3,1}
{9}
{4,8}
{7}
{2}
{5}
{2,5}
{1,3}
{5}
{2}
{3}
{1}
{4}
{8}
{2}
{1}

```java
public static int[] mergeSort(int[] arr, int lo, int hi) {
  //write your code here

  if(lo == hi) {
      int[] baseArr = new int[1];
      baseArr[0] = arr[lo];
      return baseArr;
  }

  int mid = (lo + hi)/2;

  int[] leftSortedArray = mergeSort(arr,lo,mid);
  int[] rightSortedArray = mergeSort(arr,mid+1,hi);

  int[] sortedArray = mergeTwoSortedArrays(leftSortedArray,rightSortedArray);

  return sortedArray;
}
```

Calls = 2

Height = $\log_2 N$

$$(\text{calls})^{h} + (\text{pre}+\text{post}) * h = (2)^{\log_2 n} + (0 + \underbrace{N}_{\text{merge}}) * \log_2 N$$

$$= n + n \log_2 N$$

$$\boxed{O(\log_2 N)}$$

$$T(n) = 2\cancel{T(n/2)} + N$$

$$2\cancel{T(n/2)} = (2\cancel{T(n/4)} + N/2)\,2$$

$$2^2\cancel{T(n/4)} = (2\cancel{T(n/8)} + N/4)\,2^2$$

$$\vdots$$

$$2^{k-1}(\cancel{T(1)}) = (2T(0) + 1)\,2^{k-1}$$

$$T(n) = 2^k + N + \frac{N}{2} \times 2 + \frac{N}{2^2} \times 2^2 \ldots k \text{ times}$$

$$T(n) = 2^{\log_2 N} + N\log_2 N \quad (k = \log_2 N) = O(N\log_2 N)$$

① Extra/Auxi Space: $O(N)$ {at merging}

② Input/output : $O(N)$

③ Recursion call Stack Space: $O(\log_2 N)$

(Height of Stack/Tree)

Target Sum Pair

0	1	2	3	4
10	12	13	14	18

left

Right

```
//brute force
for(int i=0;i<nums.length;i++) {
    for(int j=i+1;j<nums.length;j++) {
        if(nums[i] + nums[j] == target) {
            return new int[]{i=1,j+1};
        }
    }
}

return null;
```

2 pointer solution

```
int lo = 0;
int hi = nums.length-1;

while(lo < hi) {
    if(nums[lo] + nums[hi] < target) {
        lo++;
    } else if(nums[lo] + nums[hi] > target) {
        hi--;
    } else {
        return new int[]{lo+1,hi+1};
    }
}

return null;
```

nums[left] + nums[right] > target

right--;

bada kuch zyada he
bada reh gaya

nums[left] + num[right] < target

left++;

chota kuch zyada he
chota leh gaya

$T(n) = T(n-1) + O(1)$

$\boxed{TC = O(N)}$

Partition an Array

pivot = 5

l r

	7	9	4	8	3	6	2	5
unexplored	✓	✓	✓	✓	✓	✓	✓	✓

≤ pivot

> pivot

Initial State

(all are unexplored)

l r

	7	9	4	8	3	6	2	5
unexplored		✓	✓	✓	✓	✓	✓	✓

(r to n)

≤ pivot

(0 to l-1)

> pivot ✓

(l to r-1)

```
if (arr[right] > pivot) r++;
```

l r

7 9 4 8 3 6 2 5

unexplored (r to n) ✓ ✓ ✓ ✓ ✓ ✓

≤ pivot (0 to l-1)

> pivot (l to r-1) ✓ ✓

```
if (arr[right] ≤ pivot) {
    swap(arr[l], arr[r]);
    l++; r++;
}
```

l r

4 7

~~7~~ 9 ~~4~~ 8 3 6 2 5

unexplored (r to n) ✓ ✓ ✓ ✓ ✓ ✓

≤ pivot (0 to l-1) ✓

> pivot (l to r-1) ✓ ✓

l r

4 3 2 5 9 7 8

~~7~~ ~~9~~ ~~4~~ ~~8~~ ~~3~~ 6 ~~2~~ ~~5~~

unexplored
(r to n)

≤ pivot ✓ ✓ ✓ ✓
(0 to l-1)

> pivot ✓ ✓ ✓ ✓
(l to r-1)

(Relative order changed)

Partitioned Array = 4 3 2 5 9 6 7 8

left = first ele of greater region

right = first ele of unexplored region

```
public static void partition(int[] arr, int pivot){
  //write your code here
  int left = 0;
  int right = 0;

  while(right < arr.length) {
      if(arr[right] > pivot) {
          right++;
      } else {
          swap(arr,right,left);
          left++;
          right++;
      }
  }
}
```

Time Complexity = O(N)

Sort 01

Same as the partition of array. Just make pivot = 1 or 0 and partition accordingly.

```
public static void sort01(int[] arr){
  //write your code here
  int left = 0;
  int right = 0;

  for(int i=0;i<arr.length;i++) {
    if(arr[i] > 0) {
      right++;
    } else {
      int temp = arr[left];
      swap(arr,right,left);
      left++;right++;
    }
  }

}
```

Variations

Partition Variations

Partition itself ⇒ $\leq x$, $> x$ or $> x$, $\leq x$ or $< x$, $\geq x$ or $\geq x$, $< x$

Sort 01 variations ⇒

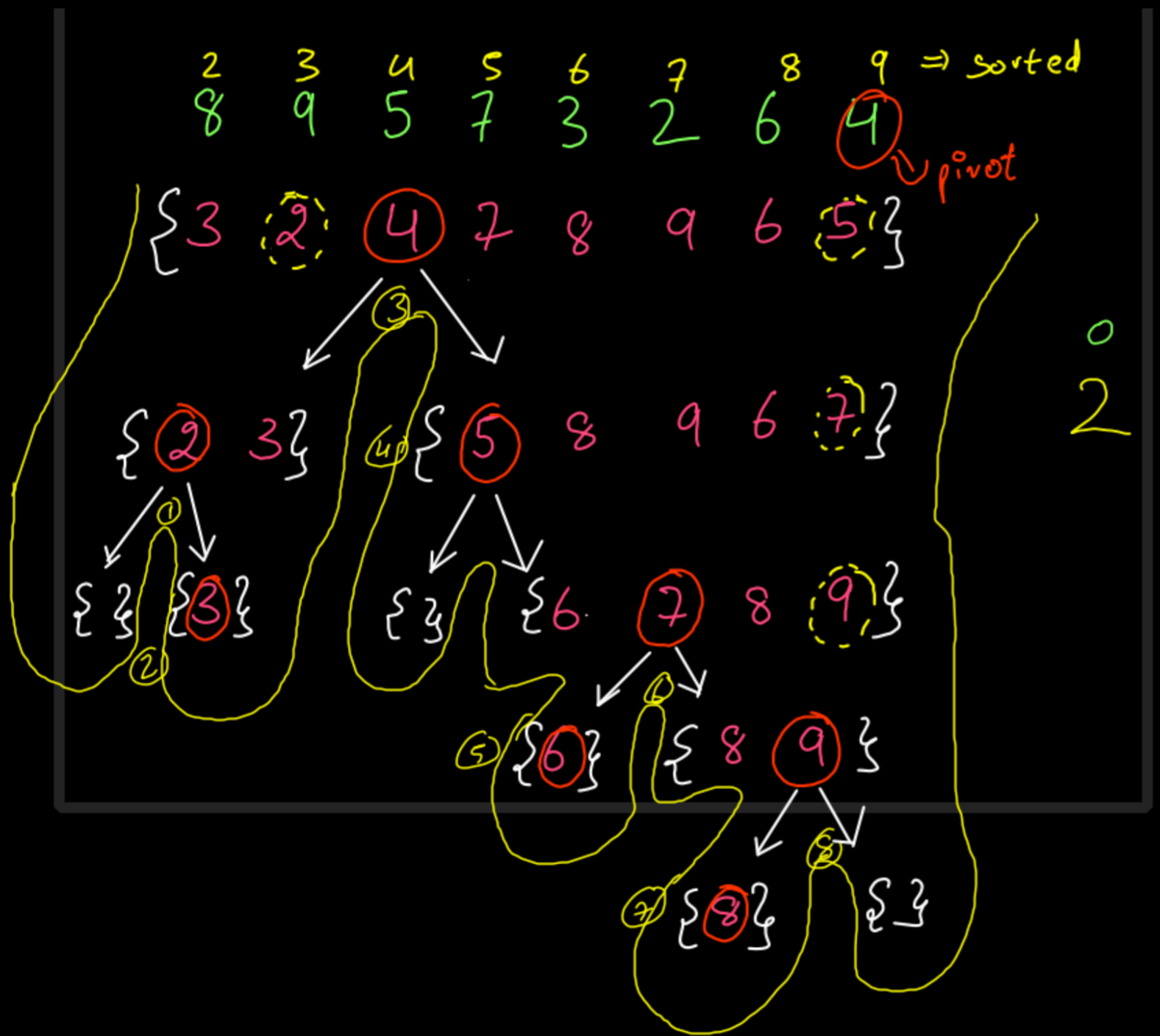
1. 0's and 1's
2. Reds & Blues
3. -ves & +ves
4. 0's & non-zeroes { move 0's to right / left }
5. Odds & Evens

Quick Sort

Merging was possible only on 2 sorted arrays. Hence merging was done in postorder. Partition is done when the array is not sorted hence partition will be done in preorder.

focus on 2 why?

1. Why pivot on last element?
2. Stability?



0	1	2	3	4	5	6	7
2	3	4	5	6	7	8	9

5 3 2 6 4 8 7 9 ⇒ partitioned array

this element should be at 6's left.

Aisa isliye ho rahe hai kyuki agar pivot k baad koi chota element tha array me to vo pivot k right me he rah jaega. Because jab partition hui aur hum pivot par aae to vo less than equal to vala case hai, so pivot arr[left] se swap hoga. aur left aur right dono aage bad gaye. Ab agar pivot se chota kuch aaya to vo arr[left] se swap hoga lekin left to already pivot ko cross kar chuka hai. To swap hone k baad bhi vo pivot k right me he rahega

pivot se chota no

all this is when we use the current algo of partition.

If we use some other method, things might be different.

The partition that we are using is Lomuto Partition scheme and all the previous observations are in accordance with that.

In lomuto partition, both indices start from 0 & rest we know from the procedure.

The other way is Hoare's Partition where left starts from 0 and right from last index & they move towards each other.

Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal. But like Lomuto's partition scheme, Hoare partitioning also causes Quicksort to degrade to $O(n^2)$ when the input array is already sorted; it also doesn't produce a stable sort.

```java
public static void quickSort(int[] arr, int lo, int hi) {
  // partition the array first and then sort them

  if(lo > hi) {
      return;
  }

  int pivot = arr[hi];
  int partitionIndex = partition(arr,pivot,lo,hi);

  quickSort(arr,lo,partitionIndex - 1);
  quickSort(arr,partitionIndex + 1,hi);
}
```

QuickSort is decrease & conquer because subproblems are not of same size.

if partition is not done at arr[right]

⇒ element is not at its sorted pos

⇒ left ≤ & right > is not true

⇒ recursive calls cannot be made.

T/O → O(N)

Aux sp → O(1)

Rec Space → O(N)

(worst case)

```java
public static void quickSort(int[] arr, int lo, int hi) {
  // partition the array first and then sort them

  if(lo > hi) {
      return;
  }

  int pivot = arr[hi];
  int partitionIndex = partition(arr,pivot,lo,hi);

  quickSort(arr,lo,partitionIndex - 1);
  quickSort(arr,partitionIndex + 1,hi);
}
```

$(\text{calls})^{\text{hight}} + (\text{pre} + \text{post}) * \text{hight}$

Worst Case

$(1)^{N} + (N+0) * N = O(N^2)$

only 1 call because second call will be base case.

Avg Case & Best Case

$(2)^{\log_2 N} + (N+0) * \log_2 N = O(N \log_2 N)$

```java
public static void quickSort(int[] arr, int lo, int hi) {
  // partition the array first and then sort them

  if(lo > hi) {
      return;
  }

  int pivot = arr[hi];
  int partitionIndex = partition(arr,pivot,lo,hi);

  quickSort(arr,lo,partitionIndex - 1);
  quickSort(arr,partitionIndex + 1,hi);
}
```

Worst Case: $T(n) = T(n-1) + O(N)$

$T(n-1) \Downarrow$ Rec call

$O(N) \Downarrow$ Partition

$$\left.\begin{aligned} T(n) &= T(\cancel{n-1}) + n \\ T(\cancel{n-1}) &= T(\cancel{n-2}) + n \\ &\vdots \\ T(\cancel{1}) &= T(0) + n \end{aligned}\right\} n \text{ times}$$

$T(n) = T(0) + n^2 \Rightarrow \boxed{T(n) = O(n^2)}$

$T(0) \Downarrow$ only one ele or 0 ele

```java
public static void quickSort(int[] arr, int lo, int hi) {
  // partition the array first and then sort them

  if(lo > hi) {
      return;
  }

  int pivot = arr[hi];
  int partitionIndex = partition(arr,pivot,lo,hi);

  quickSort(arr,lo,partitionIndex - 1);
  quickSort(arr,partitionIndex + 1,hi);
}
```

Best and Average Case

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

$$T(n) = \cancel{2T(n/2)} + N$$

$$2\cancel{T(n/2)} = (2\cancel{T(n/4)} + N/2) \times 2$$

$$2^2 T(\cancel{n/4}) = (2\cancel{T(n/8)} + N/4)\, 2^2$$

$$\vdots$$

$$2^{k-1}\cancel{T(n/2^{k-1})} = (2T(n/2^k) + N/2^{k-1})\, 2^{k-1}$$

$$T(n) = 2^k T(n/2^k) + k * N$$

We assume that $T(1) = 1$ i.e one element takes 1 unit of time

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$k = \log_2 N$$

$$\Rightarrow T(n) = 2^{\log_2 N}\, T(n/2^{\log_2 N}) + \log_2 N * N$$

$$T(n) = N * T(1) + N \log_2(N) = \boxed{O(N \log_2 N)}$$

Randomised QuickSort

We are not reducing the WC time i.e. $O(N^2)$, rather we are reducing the no of times worst case hits.

2 1 3 5 4 6 8 7 → pivot {wc} ⇒ nearly/almost sorted array

⇒ Swap 7 with any random element

2 1 3 5 7 6 8 4 ⇓ pivot

{this can also hit wc}

Now not on sorted array.

Worst case: $O(N^2)$

Best/Avg: $O(N \log N)$

can be anything

{2 1 3} 4 {7 6 8 5}

⇓

again will cause WC so swap with any random element.

Merge Sort

Best, Avg, Worst } $O(n \log n)$ Time

Extra space : → $O(n)$

Inplace ✗ { Merge 2 Sorted Array }

Stability ✓

+

Prefered for Non-Sequential Data (LL)

Quick Sort

Best, Avg ⇒ $\Theta(n \log n)$

Worst case ⇒ $O(n^2)$

Extra space : → $O(1)$ → Inplace ✓

Stable ✗

Prefered for Arrays

Stability

$\boxed{4^a}\ 3^b\ 5^c\ 1^d\ \boxed{4^e}\ 5^f\ 1^g$

after sorting also, 4^a should come before 4^e

Why is Merge Sort Stable?

```
public static int[] mergeTwoSortedArrays(int[] a, int[] b){
  //write your code here
    int[] c = new int[a.length + b.length];

    int i = 0, j = 0, k = 0;

    while(i < a.length && j < b.length) {

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

    while(j<b.length) {
        c[k] = b[j];
        j++;
        k++;
    }

    while(i<a.length) {
        c[k] = a[i];
        i++;
        k++;
    }
    return c;
```

↳ in case of equal ele, ele of first array is added to the merged array.

Why Qs is not Stable?

⇒ Because partitioning is not stable.

Quick Select (k^{th} Smallest Element)

Kind of a mixture of Binary Search (Divide & Conq) & Quicksort Partition.

(K = 6)

6th smallest

8 9 5 7 1 2 6 4 3

1 2 3 (7 8 9 6 4 5)

⇓

4 5 (9 6 7 8)

⇓

(6 7 8 9

⇓

(6) 7

⇓

6th Smallest Element

Quick Select Code

```java
public static int quickSelect(int[] arr, int lo, int hi, int k) {
  //write your code here
  if(lo == hi) {
      return arr[lo];
  }

  int pivot = arr[hi];
  int partitionIndex = partition(arr,pivot,lo,hi);

  if(partitionIndex == k) {
      return arr[k];
  } else if(partitionIndex < k) {
      return quickSelect(arr,partitionIndex + 1, hi,k);
  }

  return quickSelect(arr,lo,partitionIndex - 1,k);
}
```

⇝ k-1 is not used as we have passed k-1 in main func when k is inputed.

Quick Select Time Complexity

$$T(n) = T(n/2) + O(N)$$

$$T(n) = \cancel{T(n/2)} + N$$

$$\cancel{T(n/2)} = \cancel{T(n/4)} + \frac{N}{2}$$

$$\cancel{T(n/4)} = \cancel{T(n/8)} + N/4$$

$$\vdots$$

$$\cancel{T(n/2^{k-1})} = T(n/2^k) + N/2^{k-1}$$

$$T(n) = T(n/2^k) + N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} \cdots \frac{N}{2^{k-1}}$$

Assumption: $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$k = \log_2 N$

$$T(n) = \cancel{T(1)}^{\,0} + N\left\{1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots \frac{1}{2^{k-1}}\right\}$$

$$T(n) = N\left\{\underbrace{1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots \frac{1}{2^{k-1}} + \frac{1}{2^k}} - \frac{1}{2^k}\right\}$$

$$\text{GPsum} = \frac{a[1-r^n]}{1-r} = \frac{1\left[1-\left(\frac{1}{2}\right)^k\right]}{\frac{1}{2}} = 2\left[1 - \frac{1}{2^{\log_2 n}}\right]$$

$$\text{GPSum} = 2\left[1 - \frac{1}{n}\right] = 2 - \frac{2}{n}$$

$$T(n) = N\left\{2 - \frac{2}{N} - \frac{1}{2^{\log_2 n}}\right\} = N\left\{2 - \frac{2}{N} - \frac{1}{N}\right\}$$

$$T(n) = N\left\{2 - \frac{3}{N}\right\} = 2N - 3 \approx \boxed{O(N)}$$

In calculating the Time Complexity of Quick Select, our formula gives wrong answer.

$$TC = (\text{calls})^{\text{height}} + (\text{pre} + \text{post}) * \text{height}$$

$$TC = (1)^{\log_2 n} + (N + 0) * \log_2 N$$

$$TC = O(N \log_2 N) \Rightarrow \text{This is wrong.}$$

Sort 012 {Dutch National Flag Algo (DNF Sort)}

i j k

1 0 2 0 1 0 2 0 2 2 0 1 1 0 2

unexplored (j to k)

0's (0 to i-1)

1's (i to j-1)

2's (k to n)

```
public static void sort012(int[] arr){
  //write your code here
  int i = 0, j = 0, k = arr.length-1;

  // 0 to i-1 -> 0's region
  //  i to j-1 -> 1's region
  // j to k-1 -> unexplored
  // k to n-1 -> 2's region
  while(j <= k) {
      if(arr[j] == 0) {
          swap(arr,j,i);
          i++;
          j++;
      } else if(arr[j] == 1) {
          j++;
      } else {
          swap(arr,j,k);
          k--;
      }
  }
```

j++ isliye nahi kiya because hum 2 peeche bhej

vahe hain vo hume pata hai, lekin peeche se 0, 1 ya 2 kya aa raha hai vo hume nahi pata.

DNf Sort (Sort 012) Variations

1. Sort 0s, 1s & 2s
2. Sort R's, G's & B's.
3. Sort $a[i]\%3==0$, $a[i]\%3==1$, $a[i]\%3==2$
4. Dual pivot partitioning

   $p_1 < p_2$

   $< p_1, [p_1, p_2], > p_2$
5. Three way partitioning

   $< p, = p, > p$

   Improves QuickSort in case of duplicate elements.

5 4 2 3 4 2 3 4 1 5 2 1 3

(1 1 2 2 2)(3 3 3)(4 4 4 5 5)

Size of subarrays reducing because of middle region

Count Sort Without Stability

1st Iteration: find min & max in the array

min = 2

max = 9

> formulas for range
>
> $[l, r] = r - l + 1$
>
> $(l, r]$ or $[l, r) = r - l$
>
> $(l, r) = r - l - 1$

2nd Iteration: fill the freq array { farr = new int [max - min + 1] }

0	1	2	3	4	5	6	7
1	3	2	2	3	0	1	3
2	3	4	5	6	7	8	9

3rd Iteration: fill original array from frequency array

Code for Count Sort Without Stability

```java
public static void countSort(int[] arr, int min, int max) {
 //write your code here
  //Since max and min is already given we don't perform the first iteration

  int size = max - min + 1;
  int[] farr = new int[size];

  for(int i=0;i<arr.length;i++) {
      farr[arr[i] - min]++;
  }

  int j = 0;
  for(int i=0;i<farr.length;i++) {
      for(int k=0;k<farr[i];k++) {
          arr[j] = i + min;
          j++;
      }
  }
}
```

Count Sort (Range of elements is fixed & small)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	6	3	5	3	4	3	9	6	4	6	5	8	2	9

Count Sort With Stability

1st Iteration: find min & max in the array

min = 2

max = 9

2nd Iteration: fill the freq array { farr = new int [max - min + 1] }

0	1	2	3	4	5	6	7
1	3	2	2	3	0	1	3
2	3	4	5	6	7	8	9

3rd Iteration: Prefix Sum Array

0	1	2	3	4	5	6	7
1	4	6	8	11	11	12	15
2	3	4	5	6	7	8	9

prefixSum[5] = sum of freqs of all elements less than or equal to 5

$ps(2) = \overbrace{freq(2)}^{O(N^2)}$

$ps(3) = \underline{freq(2) + freq(3)} = \overbrace{ps(2) + freq(3)}^{O(N)}$

$ps(4) = \underline{freq(2) + freq(3)} + freq(4) = ps(3) + freq(4)$

⋮

$ps(9) = \underline{freq(2) + freq(3) + \cdots} freq(9) = ps(8) + freq(9)$

DP (Logic wise)

$\Rightarrow \boxed{ps[i] = ps[i-1] + freq[i]}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	6	3	5	3	4	3	9	6	4	6	5	8	2	9
2	3	3	3	4	4	5	5	6	6	6	8	9	9	9

⇒ lastIndex of an element $x = ps[x] - 1$

{Because ps tells us no of elements less than or equal to curr ele which indirectly tells us lastIndex of each ele. in a sorted array}

3rd Iteration: Prefix Sum Array

0	1	2	3	4	5	6	7
1	4	6	8	11	11	12	15
2	3	4	5	6	7	8	9

4th Iteration: Sort the Array

0	1	2	3	4	5	6	7
~~1~~	4	~~6~~	~~8~~	~~11~~	11	~~12~~	~~15~~
2	3	4	5	6	7	8	9

(Traversing the input array in reverse)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9′	6′	3′	5′	3″	4′	3‴	9″	6″	4″	6‴	5″	8	2	9‴
2	3′	3″	3‴	4′	4″	5′	5″	6′	6″	6‴	8	9′	9″	9‴

↑ This will be a seperate resultant array to avoid loss of information.

Code for Count Sort with Stability

```java
public static void countSort(int[] arr, int min, int max) {
 //write your code here
  //Since max and min is already given we don't perform the first iteration

  int size = max - min + 1;
  int[] farr = new int[size];

  for(int i=0;i<arr.length;i++) {
      farr[arr[i] - min]++;
  }

  //now create the prefix sum array
  int[] prefixSum = new int[farr.length];
  prefixSum[0] = farr[0];

  for(int i=1;i<prefixSum.length;i++) {
      prefixSum[i] = prefixSum[i-1] + farr[i];
  }

  //Iterate from the back of the array
  int[] res = new int[arr.length];
  for(int i=arr.length-1;i>=0;i--) {
      prefixSum[arr[i] - min]--;
      int idx = prefixSum[arr[i] - min];
      res[idx] = arr[i];
  }

  for(int i=0;i<arr.length;i++) {
      arr[i] = res[i];
  }

}
```

$TC = O(N)$

Extra space: $O(Range)$

$\Downarrow$

max size of array is 10^6

Hence 10^6 will be the max Range size till when we can apply count sort.

Radix Sort (No restriction on Range)

213	982	213	37
97	213	718	64
718	123	123	97
123	443	37	123
37	683	443	213
443	64	64	375
982	375	375	443
64	97	982	683
375	37	683	718
683	718	97	982
H T O			

ones → stable

tens → stable

Hund's → stable

$(arr[i]/exp)\%10$

⇓

To extract a digit

Number of Digits

$1 \rightarrow 1$

$10 \rightarrow 2$

$100 \rightarrow 3$

$1000 \rightarrow 4$

$10000 \rightarrow 5$

$10^5 \rightarrow 6$

$10^6 \rightarrow 7$

++

$$\boxed{N \rightarrow (\log_{10} N + 1) \text{ digits}}$$

Time complexity = If the max no is max

we apply $\log_{10}$ Max iterations

& in every iteration; we apply count sort

$$\boxed{TC = O(N * \log_{10}(max))}$$

Radix Sort Code

```java
public static void countSort(int[] arr, int exp) {
  // write code here
  System.out.print("After sorting on " + exp + " place -> ");
  int size = 10;
  int[] farr = new int[size];

  //filling the freq array
  for(int i=0;i<arr.length;i++) {
      int idx = (arr[i]/exp)%10;
      farr[idx]++;
  }

  //now create the prefix sum array
  int[] prefixSum = new int[farr.length];
  prefixSum[0] = farr[0];

  for(int i=1;i<prefixSum.length;i++) {
      prefixSum[i] = prefixSum[i-1] + farr[i];
  }

  //Iterate from the back of the array
  int[] res = new int[arr.length];
  for(int i=arr.length-1;i>=0;i--) {
      prefixSum[(arr[i]/exp)%10]--;
      int idx = prefixSum[(arr[i]/exp)%10];
      res[idx] = arr[i];
  }

  for(int i=0;i<arr.length;i++) {
      arr[i] = res[i];
  }
  print(arr);
}
```

```java
public static void radixSort(int[] arr) {
  // write code here
  int max = Integer.MIN_VALUE;
  for(int val: arr) max = Math.max(max,val);

  int digits = (int)(Math.log10(max)) + 1;
  int maxPlaceValue = (int)Math.pow(10,digits-1);

  for(int i=1;i<=maxPlaceValue; i*=10) {
      countSort(arr,i);
  }
}
```

Binary Search

Time complexity: $O(\log_2 n)$

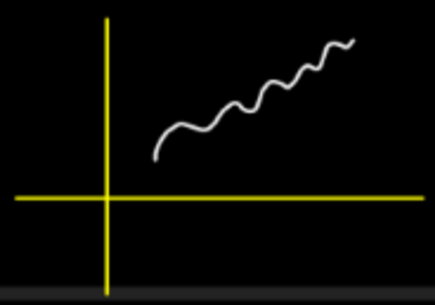
Constraint: Monotonic

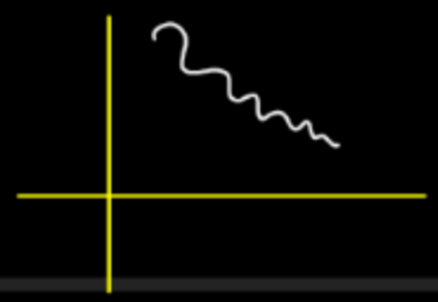
- Inc
  - Strictly Inc
- Dec
  - Strictly Dec
- Binary
  - 0s then 1s
  - 1s then 0s

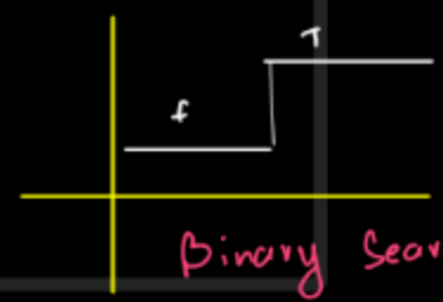
Modified BS

- Rotated Sorted Array
- Bitonic Array

- Infinite Array (Unbounded)
- Matrix
- Nearly Sorted Array {K Sorted Array}







Binary Search On Answer ★★ Very Imp concept

Apply Binary Search

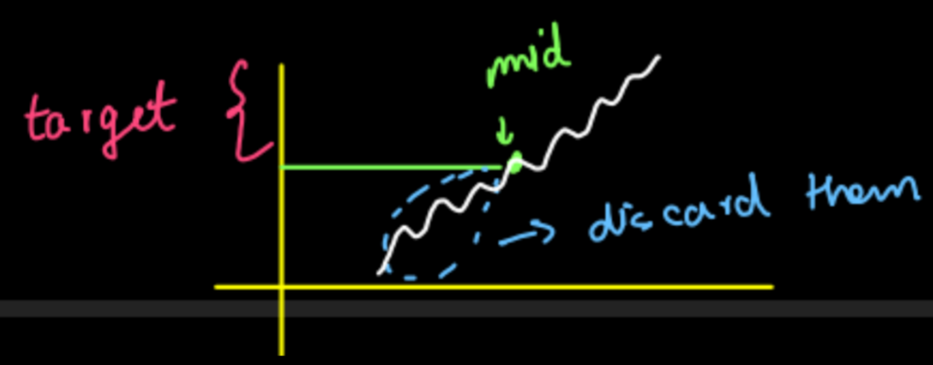
$T(n) = T(n/2) + K$ $\quad O(\log_2 n)$

target => 60

-40	-20	-10	0	10	30	30	60	80	90	100
0	1	2	3	4	5	6	7	8	9	10

low, mid, high

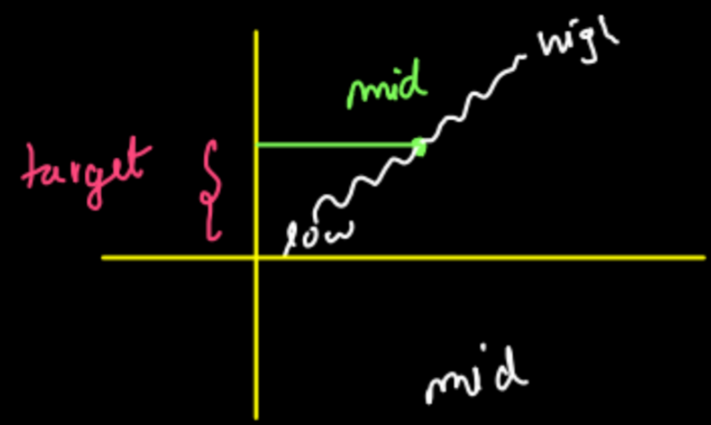
-40	-20	-10	0	10	30	30	60	80	90	100
0	1	2	3	4	5	6	7	8	9	10



So, low = mid + 1

low mid high

-40	-20	-10	0	10	30	30	60	80	90	100
0	1	2	3	4	5	6	7	8	9	10



So, high = mid - 1

mid low high

-40	-20	-10	0	10	30	30	60	80	90	100
0	1	2	3	4	5	6	7	8	9	10

low = mid + 1

low high mid

-40	-20	-10	0	10	30	30	60	80	90	100
0	1	2	3	4	5	6	7	8	9	10

in case of unsuccessful search low > high & element is not found.

```java
public int search(int[] nums, int target) {
  int left = 0, right = nums.length -1;

   while(left <= right) {
       int mid = (left + right)/2;

       if(nums[mid] == target) {
           return mid;
       } else if(nums[mid] < target) {
           left = mid + 1;
       } else {
           right = mid - 1;
       }
   }

   return -1; //search unsuccessful
}
```

mid = (left + right) / 2 → can cause overflow

⇒ instead use

mid = left + (right - left) / 2;

Sort Dates

```
5
12041996
20101996
05061997
12041989
11081987
```

Dates given in DD|MM|YYYY

⇒ Sort on the basis of days first then month, then year

12/04/1996
20/10/1996
05/06/1997
12/04/1989
11/08/1987

Given dates

12041996 ⇒ to get date = ((12041996) / 10^6) % 100

12041996 ⇒ to get month = ((12041996) / 10^4) % 100

12041996 ⇒ to get year = (12041996) % 10^4

```
5
12041996
20101996
05061997
12041989
11081987
```

Sort Dates Procedure

12041996
20101996
05061997
12041987
11081987

date wise stable sort →

05061997
11081987
12041996
12041987
20101996

month wise stable sort

12041996
12041987
05061997
11081987
20101996

year wise stable sort

12041987
11081987
12041996
20101996
05061997

Sort Dates Code

```java
public static void sortDates(String[] arr) {
  countSort(arr, 1000000, 100, 32); //days are from 1-31
  countSort(arr, 10000, 100, 13); //months are from 1-12
  countSort(arr, 1, 10000, 2501); //given in the question
}

public static void countSort(String[] arr,int div, int mod, int range) {
  String[] ans = new String[arr.length];
  // make frequency arr
  int[] farr = new int[range];
  for (int i = 0; i < arr.length; i++) {
    farr[Integer.parseInt(arr[i], 10) / div % mod]++;
  }
  // convert it into prefix sum array
  for (int i = 1; i < farr.length; i++) {
    farr[i] += farr[i - 1];
  }
  // stable sorting(filling ans array)
  for (int i = arr.length - 1; i >= 0; i--) {
    int pos = farr[Integer.parseInt(arr[i], 10) / div % mod] - 1;
    ans[pos] = arr[i];
    farr[Integer.parseInt(arr[i], 10) / div % mod]--;
  }
  // filling original array with the help of ans array
  for (int i = 0; i < arr.length; i++) {
    arr[i] = ans[i];
  }
}
```

base 10 is req otherwise 06 will be read as an octal No i.e any no starting with 0 will be converted to its binary eq considering it to be octal.