Q1

```
int Binary(int arr[], int n, int key)
{   for(i=0 to n-1)
      if(arr
{   int mid = arr[(n-1)/2];
    if (key == mid)
       return 1;
    else if (key > mid)
    {for(int i=mid+1, to n-1)
      {  if(arr[i] == key)
            return 1;
      }
    }
    else{for(i=0 to mid-1)
          { if (arr[i] == key)
                return 1;
          } }
  return -1;
}
```

No. of comparisions:-

$\left(\frac{n}{2}+1\right)$

Q2. Iterative Way Insertion Sort:-

```
void insSort (int arr[], int n)
{   for (i=1 to n)
    {   int value = arr[i]
        int j=i
        while (j>0 && arr[j-1] > value)
        {   arr[j] = arr[j-1];
            j--
        }
        arr[j] = value;
    }
}
```

Recursive Way Insertion Sort:-

```
void insSort (int arr[], int i, int n)
{   int value = arr[i];
    int j=i
    while (j>0 && arr[j-1] > value)
    {   arr[j] = arr[j-1]
        j--
    }
    arr[j] = value;
    if (i+1<=n)
      insSort (arr, i+1, n)
}
```

An online algorithm is one that can process its input piece-by-piece without having the entire input available from the beginning.

Insertion sort is considered as online as it looks at one input element per iterations & produces a partial solution without considering future elements.

Other algorithms, like selection sort considers the whole input for sorting by repeatedly finding the minimum element from unsorted part & putting it at the beginning, which requires accers to the entire input.

Q3

Algorithm	Time Complexity		
	Best	Average	Worst
Bubble Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$
Quick Sort	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n^2)$
Heap Sort	$\Omega(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$
Count Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$ where k: range of nos.

Q4

Algorithm	Type of Sorting
Insertion Sort	Inplace / Stable / Online
Quick Sort	Inplace / Unstable
Heap Sort	Inplace / Unstable
Bubble Sort	Inplace / Stable
Merge Sort	Stable
Selection Sort	& Unstable ~~/Online~~
Count Sort	Stable

Q5 Iterative Method

```
int binarySearch(int A[], int x)
{ int low=0, high = A.length-1
  while (low <= high)
  { int mid = low + (high-low)/2
    if (x == A[mid])
        return mid;
    else if (x > A[mid])
        low = mid+1;
    else
        high = mid-1;
  }
}
```

Recursive Method.

```
int binarySearch (int A[], int low, int high, int x)
{ if (low > high)
      return -1      // base condition
  int mid = low + (high - low) /2
  if (x == A[mid])
    return mid;
  else if (x > A[mid])
    return binarySearch (A, mid+1, high, x)
  else
    return binarySearch (A, low, mid-1, x)
}
```

T.C of Linear Search :- $O(n^2)$

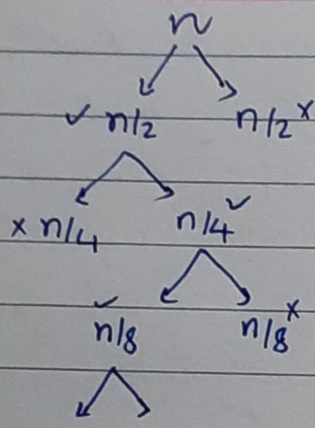
T.C of Binary Search :- $O(\log n)$

S.C of Linear search = $O(1)$

S.C of Binary Search = $O(1)$ (It)

S.C of Binary Search = $O(\log n)$. (Rec)

Q6



Recurrence Relation

$\therefore T(n) = T(n/2) + C$

Q7

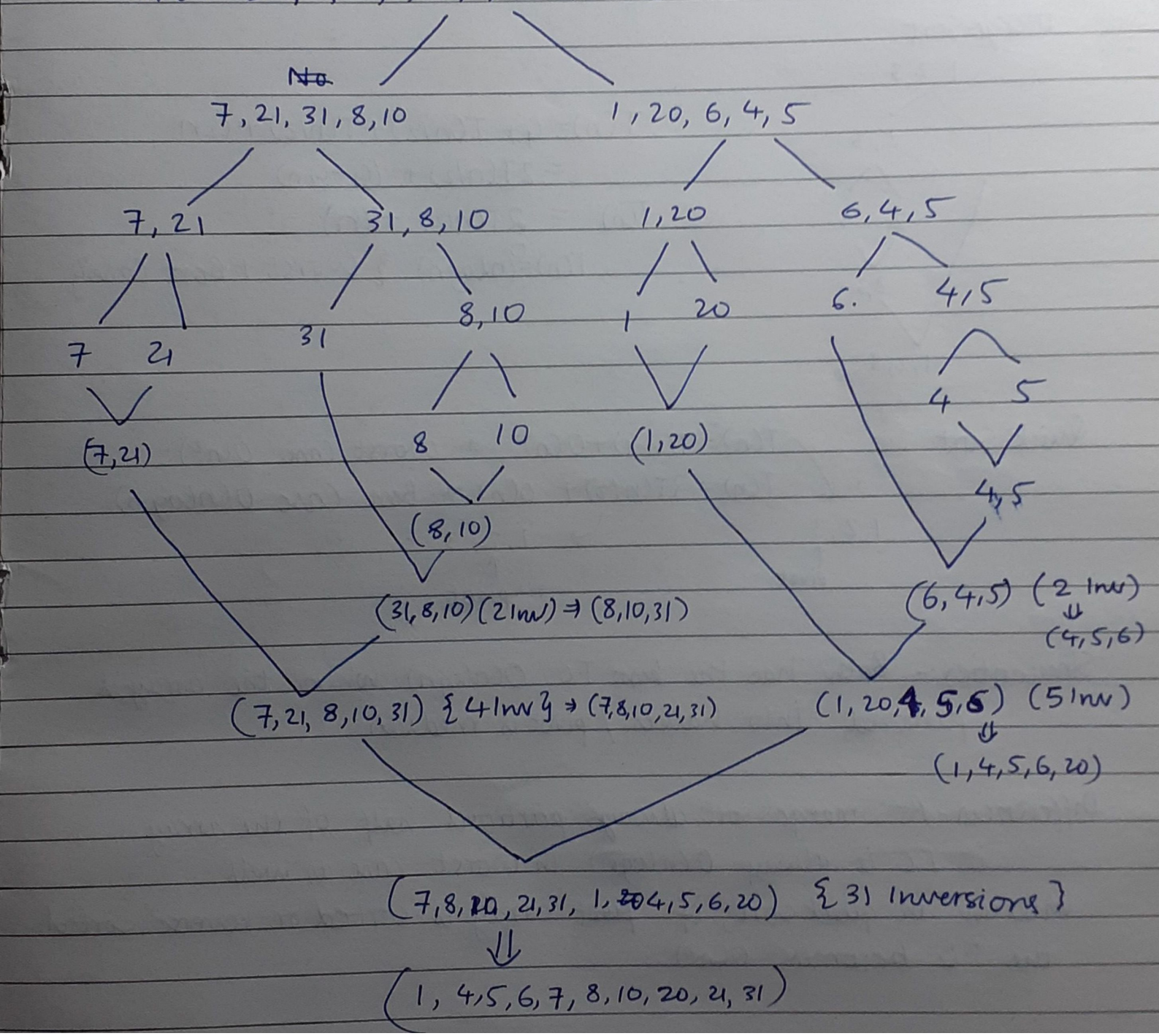
```
int find_sumPair (int A[], n, k)
{   Sort (A,n)    ⇒ merge sort ⇒ O(nlogn)
    i=0
    j=n-1

    while (i<j)
    { if (A[i] + A[j]==k)
          return 1;
      else if (A[i] + A[j]<k)
          i=i+1;
      else
      j=j-1;
    }  return -1;
}
```

T.C = T.C sorting + T.C of two pointer

= O(nlogn) + O(n) = O(logn. n)

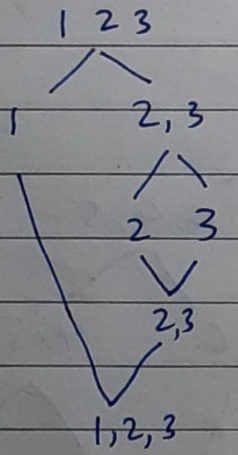
Q8. Each sorting algorithm has its best use case but if we have to select one, we can use quicksort as we are dealing with computers in our everyday life, quicksort performs best for datasets that fit in memory. Further it makes faster sortings. It is also inplace algo, which means it doesn't require extra space unlike merge sort

Q9 Number of inversion in an array means how far (or close) the array is from being sorted. Two elements a[i], a[j] form an inversion if a[i] > a[j] & i < j.



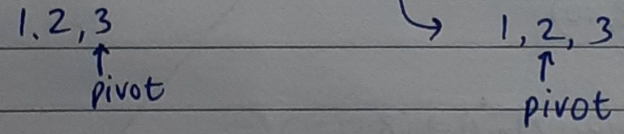
Q10 Quicksort gives worst case T.C = $O(n^2)$ when array is reverse sorted or sorted.
Quicksort gives best case T.C = $O(n \log n)$ when we will select pivot as a mean element.

Q11 Merge sort



$T(n) = c_1 + T(n/2) + T(n/2) + c_2 n$
$= 2T(n/2) + (c_1 + c_2 n)$
$T(n) = 2T(n/2) + O(n)$
$T(n) \overset{O}{=} (n \log n)$ {Worst + Best Case}

Quick Sort
$T(n) = T(n-1) + O(n)$ ⇒ worst case $O(n^2)$
$T(n) = 2T(n/2) + O(n)$ ⇒ Best Case $O(n \log n)$



Similarities :- Both has the best T.C $O(n \log n)$ when the array is partioned into halves / pivot is median.

Differences :- As merge sort always partions half of the array. its T.C is alway $O(n \log n)$ in worst case as well.
Whereas in quicksort, if ~~pivot~~ array is sorted or reverse sorted, the T.C becomes $O(n^2)$.

Q12 Selection Sort is unstable as :-

If array is $4_A, 1, 5, 3, 4_B, 2$ it will sort like this

$1, 2, 3, 4_B, 4_A, 5$ instead of

$1, 2, 3, 4_A, 4_B, 5$

It is because of swapping. Instead push every element & put minimum element at its pos.

```
void selsortstable(int arr[], int n)
 { for (i=0 to n-2)
   {  int min=i
      for (j=i+1 to n-1)
        if (a[min] > a[i])
            min=j;
      int key = a[min];
      while (min > i)
       { a[min] = a[min-1];
         min--;
       }
      a[i] = key;
   }
}.
```

Q13

```
void bubbleSort (int arr[], int n)
{  int i, j;
   bool swapped;
   for (i=0 to n-2)
   {  swapped = false
      for (j=0 ; j<n-1-i ; j++)
      {  if (arr[j] > arr[j+1])
         {  swap (arr[j], arr[j+1]);
            swapped = true;
         }
      }
   if (swapped == false)
      break;
}
```

Q14. We will use merge sort for this purpose as merge is an external sorting algo.

- External sorting handles massive amounts of data. It is used when the data being sorted can't fit into main memory (RAM). In the sorting phase, chunks of data small enough to be fit in main memory are read & sorted & written out to temp. file. In merge phase, the sorted sub files are combined into single file. Slower than Internal sorting.
- Internal sorting can handle small amounts of data. Faster than external sorting as whole process is done in main memory. Whole sorting process is done in main memory. Eg. Quicksort.