Insertion sort -

- in-place comparison sorting algorithm
- A sub-list is maintained which is always sorted
- We take a element & find its appropriate place and move elements greater than it to one step ahead and insert this element correctly.

-> T - $O(n^2)$
S - $O(1)$

How does it work? -

14	33	27	10	35	19	42	44
14	33	27	10	35	19	42	44
14	33	27	10	35	19	42	44
14	27	33	10	35	19	42	44
14	27	33	10	35	19	42	44
14	27	10	33	35	19	42	44
14	10	27	33	35	19	42	44
10	14	27	33	35	19	42	44
10	14	27	33	35	19	42	44
10	14	27	33	19	35	42	44
10	14	27	19	33	35	42	44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

Algorithm - <A simplified version>

1. If it first element, it is already sorted.
2. Pick next element
3. Compare with all elements in the sorted sublist
4. Shift all the elements in the sorted sublist that is greater than the value to be sorted
5. Insert the value
6. Repeat until list is sorted

Code -

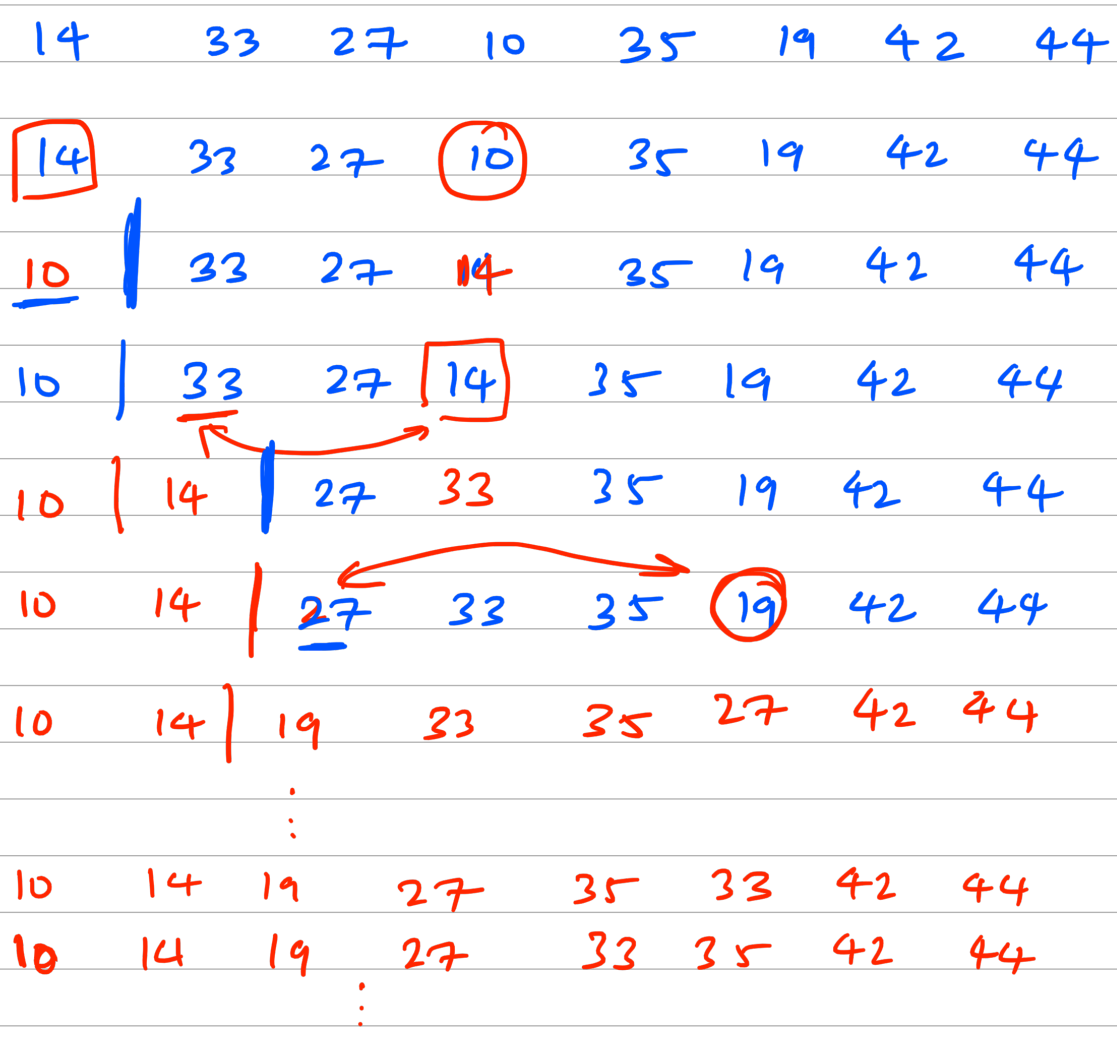
```
void insertionSort(int[] a) {

    for (int i=1; i < a.length; i+=1) {
        int position=i;
        int valueToInsert= a[i];

        while ( position > 0 && a[position-1]
            > valueToInsert]{

            a[position]= a[position-1];
            position -= 1;
```

```
        }
        a[position] = valueToInsert;
    }
}
```

T- $O(n^2)$
S- $O(1)$

(2) → Selection Sort -

- In-place comparision based sorting algorithm
- two lists →
  one sorted / one unsorted



Algorithm –

1. Set MIN to location 0
2. Search the minimum item in the list
3. Swap with value at location MIN.
4. Increment MIN to point to next element
5. Repeat until list is sorted.

Code –

```
void selectionSort(int[] a) {

    for(int i=0; i<a.length; i++) {
        int minIndex = i;
        for(int j=i+1; j<a.length; j++) {
            if(a[j] < a[minIndex]){
                minIndex=j;
            }
        }
        if(minIndex != i) {
            swap(a, minIndex, i);
        }
    }
}
```

T – $O(n^2)$

S – $O(1)$

Searching - Return the location of the target in a collection

Linear search -

```
int search (int[] a, int target) {
    for (int i=0; i< a.length; i+=1) {
        if (a[i]== target) {
            return i;
        }
    }
    return -1;
}
```

T - O(n)
S - O(1)

Binary search -

- Data is in sorted form
- Divide and conquer? -
- fast search algorithm

?-

- looks for a particular item by comparing the middle of item of the collection
- If a match occurs, then index of item is returned
- If the middle item is greater than the item, then the item is searched in subarray to the left of middle item.
- Otherwise, we search in the right sub array.

- This process continues on the subarray as well until the size of the subarray reduces to 0.

How?- search (31)

10	14	19	26	(27)	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

$mid = \frac{(0+9)}{2} = 4.5 \Rightarrow 4$

31

~~10 14 19 26 27~~ | 31 | 33 | (35) | 42 | 44

5 6 7 8 9

$mid = \frac{5+9}{2} = 7$

31

31 33 | (~~35~~) ~~42 44~~

5 6

$mid = \frac{(5+6)}{2} = |5.5| = 5$

return (5)

$mid = lo + \frac{(hi - lo)}{2}$;

$T = O(\log n)$

$S = O(1)$

Code-

```
int binarySearch(int[] a, int target){
    int left=0;
    int right= a.length-1;
    while(left <= right){
        int mid= left+(right- left)/2;
        if(a[mid]== target){
            return mid;
        }else if( target > a[mid]){
            left= mid+1;
        }else{
            right= mid-1;
        }
    }
    return -1;
}
```

Why ?. T→O(logn)

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

S→O(1)

$\log_b x = y$

$b^y = x$

find first and last index of element in a sorted array.

[2, 3, 6, 6, 6, 6, 8, 9, 9, 10, 12, 100]
0 1 2 3 4 5 6 7 8 9 10 11

```
int[] findFirstAndLast(int[] arr, int target) {
    int leftIdx = -1;

    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            leftIdx = mid;
            right = mid - 1;
        } else if (target < arr[mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    if (leftIdx == -1) {
        return new int[] {-1, -1};
    }

    int rightIdx = -1;
    left = 0;
    right = arr.length - 1;
```

```
while ( left <= right) {

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

    if( arr[mid ] == target) {
            rightIdx = mid;
            left = mid + 1;
      } else  if ( target > arr[mid ]) {
            left = mid + 1;
      } else {
            right = mid - 1;
      }
}

return  new int[] {
          leftIdx,
          rightIdx
};
}
```

T - O(logn)
S - O(1)