# Assignment - 2

(1)

```
for (i=0; i<length ; i++)
{
    if ( arr[i] == target )
    {
        return i;
    }
    else if ( arr[i] > target)
    {
        return -1;
    }
}
```

(2)

```
void iterative_insertion ( 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 = j-1;
        }
        arr[j+1] = key;
    }
}
```

```
void recursive_insertionhelper ( int arr[] ,
int n)
{
    if ( n <= 1 )
    {
        return;
    }
    recursive_insertionhelper ( arr , n-1 );
    int last_element = arr[n-1];
    int j = n-2;
    while ( j >= 0 && arr[j] > last_
    element )
    {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = last_element;
}

void recursive_insertion ( int arr[] , int n)
{
    int x = recursive_insertion (arr , n);
}
```

(3)

| Insertion sort | Selection sort | Bubble sort |
|---|---|---|
| T.C : | T.C : | T.C : |
| Best $O(n)$ | Best $O(n^2)$ | Best $O(n^2)$ |
| Avg $O(n^2)$ | Avg $O(n^2)$ | Avg $O(n^2)$ |
| worst $O(n^2)$ | worst $O(n^2)$ | worst $O(n^2)$ |
| S.C : $O(1)$ | S.C : $O(1)$ | S.C : $O(1)$ |

## Quick sort

T.C :
Best $O(n \log n)$
Avg $O(n \log n)$
worst $O(n^2)$

Space :
$O(\log n)$

## merge sort

T.C :
Best $O(n \log n)$
Avg $O(n \log n)$
worst $O(n \log n)$

Space :
$O(n)$

(4)

| Algorithm | In-place | stable | Online |
|---|---|---|---|
| Bubble sort | Yes | Yes | No |
| selection sort | Yes | No | No |
| Insertion sort | Yes | Yes | Yes |
| merge sort | No | Yes | No |
| Quick sort | Yes | No | No |
| Heap sort | Yes | No | No |
| count sort | No | Yes | No |
| Radix sort | No | Yes | No |

(5)

```
int recursive_binary_search(int arr[], int target, int low, int high)
{
    if(low > high)
    {
        return -1;
    }
    int mid = low + (high - low)/2;
    if(arr[mid] == target)
    {
        return mid;
    }
    else if(arr[mid] > target)
    {
        return recursive_binary_search
        (arr, target, low, mid-1);
    }
    else
    {
        return recursive_binary_search
        (arr, target, mid+1, high);
    }
}

int iterative_binary_search(int arr[], int target, int n)
{
    int low = 0;
    int high = n-1;
    while(low <= high)
    {
        int mid = low + (high - low)/2
```

```
if (arr[mid] == target)
{
    return mid;
}
else if (arr[mid] < target)
{
    low = mid + 1;
}
else
{
    high = mid - 1;
}
}
return -1;
}
```

## Linear search

T.C recursive

T.C → O(n)
S.C → O(1)

Iterative

T.C → O(n)
S.C → O(1)

## Binary search

recursive

T.C → O(log n)
S.C → O(log n)

Iterative

T.C → O(log n)
S.C → O(1)

(6) In each recursive call, the search space is halved, therefore, the time taken to search an array of size n can be expressed as follows:

- If the element is found at the middle index, the time taken is constant.
- If the element is not found and we continue the search.

$$T(n) = T(n/2) + O(1)$$

$T(n)$ → time taken to search an array of size $n$,
$T(n/2)$ → time taken to search an array of $n/2$,
$O(1)$ → time taken for constant operation.

Time complexity of binary search is $O(\log n)$.

(7)
```
void twosum(int arr[], int n, int target)
{
    int l = 0;
    int h = n-1;
    sort(arr, arr+n);
    while(l<=n)
    {
        if((arr[l] + arr[h]) == target)
        {
            return cout << l << " " << h;
        }
```

```
        else if (arr[l] + arr[h] > target)
        {
            h--;
        }
        else
        {
            l++;
        }
    }
```

(8)

- Quick sort is best for practical uses, because efficient for large datasets and generally outperforms other sorting algorithms
- Performs well in average and best case scenario.
- Not stable, but it can be modified to achieve stability at the cost of additional overhead.
- Requires less additional space compared to merge sort, making it suitable for memory-constrained environments.

(9)

In the context of sorting algorithms an inversion occurs when two elements in an array are out of order relative to each other. Formally, in an array arr[], a pair (arr[i], arr[j]) forms an

inversion if $i < j$ but $arr[i] > arr[j]$

for example, in the array {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} there are several inversions.

- (7, 8)
- (21, 8)
- (31, 8)
- (21, 10)
- (31, 10)
- (31, 20)
- (8, 6)
- (21, 6)
- (31, 6)
- (21, 4)
- (31, 4)
- (20, 6)
- (20, 4)
- (6, 5)

24 inversions

(10) Quick sort's time complexity varies depending on the input data and the chosen pivot elements. In the the

Best-case time complexity:

Quick sort achieves its best-case time complexity when the pivot element is chosen such that it consistently divides the array into two sub-arrays. In this scenario, the algorithm's partitioning process efficiently divides the array into smaller sub-arrays. The best case time complexity of Quick sort is $O(n \log n)$, where n is number of elements. This scenario typically occurs when the input array is already sorted or nearly sorted.

Worst-case time complexity:

- Quick sort exhibits its worst-case time complexity when the pivot element is consistently chosen in a way that one sub-array has significantly more elements.
- In this scenario, the algorithm performs poorly as it fails to divide the array into smaller sub-arrays.
- The worst case time complexity of Quick sort is $O(n^2)$
- This scenario typically occurs when the input array is already sorted and pivot selection algorithm chooses first or last element as pivot selection.

(11) 1. Merge sort:
- Recurrence relation
$T(n) = 2T(n/2) + O(n)$

2. Quick sort:
- Recurrence relation (average case)
$T(n) = T(k) + T(n-k-1) + O(n)$

- Recurrence relation (worst case)
$T(n) = T(n-1) + O(n)$

similarities:
- Both merge sort and Quick sort have a time complexity of $O(n \log n)$ in the average case.
- Both algorithms divide the input array into smaller sub-arrays, recursively sort the sub-arrays and then merge or combine the sorted sub-arrays.
- Both algorithms have a worst-case time complexity of $O(n^2)$.

Differences:
- Merge sort guarantees a time complexity of $O(n \log n)$ in all cases (best, average and worst case), making it a reliable choice.
- Quick sort has a worst-case time complexity of $O(n^2)$ but typically performs better in practice than merge sort.
- Merge sort is stable, meaning it preserves the relative order of equal

elements, while Quick sort is not stable by default.
Merge sort requires additional space for the merge step, while quick sort is an in-place sorting algorithm.

(13)

(12)

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

(13)

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

(14) In this scenario we need to use the external sorting like merge sort. Merge sort is well suited for external sorting because it divides the data into smaller chunks that can fit into the available RAM.

- Internal sorting
- Internal sorting refers to sorting

data that can fit entirely into the main memory.
- Algorithms like quick sort, merge sort are commonly used.
- In internal sorting, all data resides in the main memory.

- External sorting
- External sorting refers to sorting data that exceeds the available RAM and requires external storage.
- Algorithms like merge sort.
- In external sorting data is divided into smaller chunks that can fit into the available RAM.
- External sorting is essential for handling large datasets that cannot be processed entirely in memory.

(2) continued

Insertion sort is often referred to as an "online sorting" algorithm because it can sort a list as it receives it. Other sorting algorithms discussed in lecture includes

1. Bubble sort
2. Selection sort
3. Merge sort
4. Quick sort