Tutorial-3

Shashwat Shah
CST-SPL-1
Roll no. 39 (4th)
2017552

Q1.

Sol :- Search (arr, n, x)
if arr[n-1] == x
return "true"
backup = arr[n-1]
arr[n-1] = x

for (i=0, i++ // no termination condition
if arr[i] == x // execute at most n times.
// that is at-most n comparison

arr[n-1] = backup
return (i < n-1) // 1 comparison.

Q2. Iterative approach.

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

Q6. Recursive approach(insitionSort(arr, i, n).

```
value = arr[i]
  j = i;

while(j>0 && arr[j-1] >value).
  {
    arr[j] = arr[j-1];
    j--;
  }
arr[j] = value;

if (i+1 <=n)
  insitionSort (arr, i+1, n);
```

If we give input one by one to the algorithm and each input produces some partial solution with available input data. Then that type of algorithm is known as an online algorithm.

Here in insition sort, we give input one by one and place each one at right order with comparison from already traces element.

Q3.

Sorting Algorithm	Best	Average	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Count Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$

Q4.

1.) Bubble sort, insertion sort & selection sort are in place sorting algorithm. Because only swapping of the element in the input array is required.

2.) Bubble sort and insertion sort can be applying as stable algorithms but selection sort can't.

3.) Merge Sort is a stable algorithm but not in place algorithm. It requires extra array storage.

Quick Sort is not stable but is as in place algorithm.

Heap Sort is an in place algorithm but is not Stable.

Q8.

Iterative Binary Search.

```
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;
     }
          return -1;
  }.
```

Time Complexity = O(logn).
Space Complexity = O(1).

Recursive Binary Search

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

Time Complexity O(logn).
Space Complexity O(logn).

Q6.

Time Complexity is O(logN)

Recurrence Relation $T(n) = T(n/2) + 1$.

1st Step :- $T(n) = T(n/2) + 1$

2nd Step :- $T(n/2) = T(n/4) + 1$

3rd Step : $T(n/4) = T(n/8) + 1$

kth step later;

$$T\left(\frac{n}{2^{k-1}}\right) = T\left(\frac{n}{2^k}\right) + 1 \text{ (k times)}$$

Adding all equation above, we get.

$$T(n) = T\left(\frac{n}{2^k}\right) + K(1) \quad \text{—— (A)}$$

Now

$$\left(\frac{n}{2^k}\right) = 1 \Rightarrow k = \log n.$$

Put $k = \log n$ in eq (A).

$$T(n) = T(1) + \log n$$

$$T(n) = 1 + \log n$$

$$\boxed{T(n) = \log n}$$ Ans

Q7.

Sample
arr = {1, 5, 3, 7, 8}. target = 10

- First, we will sort the array by quicksort
  Time Complexity = O(nlogn).

arr = {1, 3, 5, 7, 8}.

- After that, we will take two pointers.

```
l = 0 , h = arr.length - 1 = 4.
    while( l <= h) .{.
    if (arr[l] + arr[h] == target)
    {
        cout << l << h;
    }.
else if (arr[l] + arr[h] > target).
    {
        h = h - 1;.
    }.
else
    {
        l = l + 1;
    }.
}.
```

Above time complexity is O(logn).
Now, total complexity of code is

O(logn + logn) = O(2logn)

T(n) = O(logn)

Q 8.

Quicksort is the fastest general-purpose sort. In most practical situations, quick sort is the method of choice.

Q9.

Inversion Count for an array indicates - how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

```
int getInvCount(int arr[], int n).
{
    int inv_count = 0;
    for (int i = 0; i < n; i++) {.
        for (int j = i+1; j < n; j++) {.
            if (arr[i] > arr[j]).
                inv_count++;.
        }
    }
    return inv_count;
}.
```

Q10.

The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

$T(n) = O(n^2)$.

The best case of quick sort is when we will select pivot as a mean element.

$T(n) = O(n \log n)$

Q11.

```
void stableSelectionSort(int a[], int n)
{
  for(int i=0; i<n-1; i++)
  {
    int min=i;
    for(int j=i+1; j<n; j++)
    {
      if(a[min]>a[j])
        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; i<n-1; i++).
   {
   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;.
}.
```

Q140.

We use external sorting for this type of sorting.

External Sorting

In external sorting data is stored outside memory (like on disc) and only loaded into memory in small chunks.

Internal Sorting

In internal sorting, all the data to sort is stored in memory at all times while sorting is in progress.