Samiksha Mishra
Spe - 1
41

Date.
Page No.

Tutorial - 3

1. int linear-search (int *arr, int n, int key)
   for i >= 0 to n-1
      if arr[i] == key
         return i
   return -1

2. iterative insertion sort
   void insertionSort (int arr[], int n)
   {
      int i, temp, j;
      for i ← 1 to n
         temp ← arr[i]
         j ← i-1
         while (j >= 0 AND arr[j] > temp)
            arr[j+1] ← arr[j]
            j ← i-1
         arr[j+1] ← temp

   recursive insertion sort
   void insertionSort (int arr[], int n)
      if (n <= 1)
         return;
      insertionSort (arr, n-1)
      last = arr[n-1]
      j = n-2
      while (j >= 0 && arr[j] > last)
         arr[j+1] = arr[j]
         j--
      arr[j+1] = last

Date.
Page No.

→ insertion sort is called online sorting because it does not need to know anything about what values it will sort and the information is requested while the algorithm is running.

3 i Selection sort -
time complexity :-
Best case - $O(n^2)$, Worst case - $O(n^2)$
space complexity - $O(1)$

ii Insertion sort -
time complexity -
Best case - $O(n^0)$, worst case - $O(n^2)$
space complexity - ~~O(nn)~~ $O(1)$

iii Merge sort -
time complexity -
Best case - $O(n \log n)$, Worst case - $O(n \log n)$
space complexity - $O(n)$.

iv Quick sort -
time complexity -
Best case - $O(n \log n)$, worst case - $O(n^2)$
space complexity - $O(n)$

v Heap sort -
time complexity -
Best case - $O(n \log n)$ worst case - $O(n \log n)$
space complexity - $O(1)$

Date.
Page No.

vi Bubble Sorting -
time complexity -
Best case - $O(n^2)$ Worst case - $O(n^2)$
Space complexity - $O(1)$

4.

Sorting	inplace	stable	online
selection	✓		
insertion	✓	✓	✓
merge		✓	
quick	✓		
heap	✓		
bubble	✓	✓	

5. Iterative binary search

```
int binarySearch(int arr[], int l, int r,
                 int x)
{
    while (l <= r) {
        int m <- (l + r) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l <- m + 1;
        else
            r <- m - 1;
    }
    return -1;
}
```

Date.
Page No.

→ Time complexity
Best Case = O(1)
Average Case = $O(\log_2 n)$
Worst Case = $O(\log n)$

Recursive Binary Search -

```
int binsearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid ← (l+r)/2
    if (arr[mid] == x)
            return mid;
    else if (arr[mid] > x)
        return binsearch(arr, l, mid-1, x)
    else
        return binsearch(arr, mid+1, r, x)
    }
    return -1;
}
```

Time Complexity -
Best Case = O(1)
Average Case = O(logn)
Worst case = O(logn)

6 Recursion relation for binary recursive search

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

Date.
Page No.

7) A[i] + A[j] = K

8. QuickSort is the fastest general-purpose sort. In most practical situations, quicksort is the method of choice. If stability is important & space is available, merge sort might be best.

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

arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

```
#include <bits/stdc++.h>
using namespace std;
int mergeSort(int arr[], int temp[],
    int left, int right);
int merge(int arr[], int temp[],
    int left, int mid, int right);
```

Date. _____
Page No. _____

```
int mergeSort (int arr[], int array_size
{
    int temp [array_size];
    return mergesort (arr, temp, 0,
                      array_size - 1);
}

int - mergeSort (int arr[], int temp[]
    int left, int right) {
    int mid, inv_count = 0;
    if ( right > left ) {
        mid = (right + left) / 2;
        inv_count += mergSort
                (arr, temp, left, mid);
        inv_count += mergeSort (
                arr, temp, mid+1, right)
        inv_count += merge (arr,
                temp, left, mid+1, right)
    }
    return inv_count;
}
int merge (int arr[], int temp [], int
    left, int mid, int right ) {
    int i, j, k;
    int inv_count = 0;
    i = left;
    j = mid;
    k = left;
```

Date.
Page No.

```
while((i<=mid-1)&&(j<=right)){
    if(arr[i]<=arr[j])
        temp[k++]=arr[i++];
    else{
        temp[k++]=arr[j++];
        inv_count=inv_count+
                  (mid-i);
    }
}
while(i<=mid-1)
    temp[k++]=arr[i++];
while(j<=right)
    temp[k++]=arr[j++];
for(i=left;i<=right;i++)
    arr[i]=temp[i];
return inv_count;
}
int main(){
    int arr[]={7,21,31,8,10,1,20,6,4,5};
    int n=sizeof(arr)/sizeof(arr[0]);
    int ans=mergeSort(arr,n);
    cout<<"No. of inversion"<<ans;
    return 0;
}
```

Date.
Page No.

10. The worst case time complexity of quick sort is $O(n^2)$. The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happen when input array is sorted or reverse sorted & either first or last element is picked as pivot.

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

11. Recurrance relation of:

a) Merge sort ⇒ $T(n) = 2T(n/2) + n$

b) quick sort ⇒ $T(n) = 2T(n/2) + n$

⇒ Merge sort is more efficient & works faster than quick sort in case of larger array size or datasets.

⇒ Worst case complexity for quick sort is $O(n^2)$ whereas $O(n \log n)$ for merge sort.

Date.
Page No.

12. Stable Selection Sort

```
#include <iostream>
using namespace std;
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;
    }
}
int main(){
    int a[] = {4, 5, 3, 2, 4, 1};
    int n = sizeof(a)/sizeof(a[0]);
    stableSelectionSort(a, n);
    for(int i=0; i<n; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

Date.
Page No.

14. The easiest way to do this is to use ~~extreme~~ external sorting like divide our source file into temporary files of size equal to the size of the RAM & first sort these files.

- External sorting - If the input data is such that it cannot adjusted in the memory entirely at once it needs to be stored in a hard disk., floppy disk or any other storage device. This is called external sorting.
- Internal sorting - If the input data is such that it can adjusted in the main memory at once it is called internal sorting.

Date.
Page No.

13.
```
#include<iostream>
using namespace std;
void bubblesort(int a[], int n){
    int flag, count=0, i;
    for(i=0; i<n-1; i++){
        flag=0;
        for(int j=0; j<n-1-i; j++)
        {
            if(a[j]>a[j+1]){
                int t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
                flag=1;
            }
        }
        if(flag==0)
            break;
    }
    cout<<"terminated at"<<i+1<<
        "passes"<<endl;
```