Tutorial – 3

Sol (1) int Linear-Search (int * arr, int n, int key)

for i>0 to n-1

{ if arr [i] = key

return i

} return -1

Sol (2) Iterative insertion sort

Void insertion-sort (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 ← j-1

arr [j+1] ← temp

Recursive insertion sort

Void insertion-sort (int arr[], int n)

if (n≤1)

return

insertion sort (arr, n-1)

last = arr [n-1]

j = n-2

Ishika

E 15 Ishika DAA

Page :
Date: / /

```
while (j>=0 && arr[j] > last)
        arr[j+1] = arr[j]
                j--
        arr[j+1] = last
```

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

Sol ③ • Selection Sort
Time Complexity : Best case :- $O(n^2)$; worst case = $O(n^2)$
Space Complexity :- $O(1)$

• Insertion Sort
TC :- best Case :- $O(n)$; worst case = $O(n^2)$
S.C :- $O(1)$

• Merge Sort
T.C = Best Complexity :- $O(n\log n)$; worst Case = $O(n\log n)$
S.C = $O(n)$

• Quick Sort
T.C :- Best Case :- $O(n\log n)$; worst Case = $O(n^2)$
S.C :- $O(n)$

• Bubble Sort
T.C - Best Case = $O(n^2)$, Worst Case = $O(n^2)$
S.C = $O(1)$

- Heap sort
  TC : Best Case - $O(n\log n)$; worst Case - $O(n^2)$
  SC - $O(1)$

Sol (4)

Sorting	inplace	stable	Online
Selection sort	✓		✓
Insertion sort	✓	✓	
Merge sort		✓	
Quick sort	✓		
heap sort	✓		
Bubble sort	✓	✓	

Sol (5) Iterative Binary Search

```
int Binary-search (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;
}
```

Page :
Date: / /

Time Complexity :- Best Case :- $O(1)$
Average Case :- $O(\log_2 n)$
Worst Case :- $O(\log_2 n)$

Recursive Binary Search

```
int binary-search (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 binary-search (arr, l, mid-1, x)
    else
        return binary-search (arr, mid+1, r, x)
    }
    return -1;
}
```

Time Complexity :- Best Case :- $O(1)$
Average Case :- $O(\log n)$
Worst Case :- $O(\log n)$

Sol (6) Recurrence relation for binary recursive search

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

Sol (7) $A[i] + A[j] = K$

Ashika

Page :
Date: / /

Sol (8) Quick sort is the fastest general purpose sort, In most practical situation quick sort is the method of choice If stability is important and space is available, merge sort, might be best.

Sol (9) Inversion count for any array indicates :- how far the array is from being sorted If the array is already sorted, then the 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 l, int r);
int merge (int arr[], int temp[], int l, int mid, int r);

int merge_sort (int arr[], int array_size)
{
    int temp [array_size];
    return _mergesort (arr, temp, 0, array_size -1);
}
int _mergesort (int arr[], int temp[], int l, int r)
{
    int mid, inv_count = 0;
    if (r > l)
    {
        mid = (r+l)/2;
```

Ishita

Page :
Date: / /

```
inv-count + = mergesort (arr, temp, l mid);
inv-count + = mergesort (arr, temp, mid+1, r);
inv-count + = merge (arr, l, mid+1, r);
}
return inv-count;
}
int merge (int arr[], int temp[], int l, int mid, int r)
{
    int i, j, k;
    int inv-count = 0;
        i = l;
        j = mid;
        k = r;
    while ((i<=mid) && (j<=r))
    {   if (arr[i] <= arr[j])
            temp [k++] = arr [i++];
    else
        { temp [k++] = arr [j++];
    for (i = l ; i<= r ; i++)
            arr [i] = temp[i];
            return inv-count;
    }
    int arr[] = {7,21,31,8,10,20,6,4,5}
    int n = size of (arr) / size of arr [0];
    int arr = merge sort (arr, n);
    Cout << " no. of inversions are " << arr;
        return 0;
}
```

Ashika

Sol(10) The worst case time complexity of quicksort is $O(n^2)$. The worst case occur when the picked pivot is always an extreme (smallest or largest) element. This happen when input array is sorted or reverse sorted and 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.

Sol(11) Recurrence 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 and works faster than quick sort in case of large array size of dataset.

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

Ishika

Page :
Date: / /

(10) Stable Selection Sort

```
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[mid] ;
        while (min > i)
        {
            a[min] = a[min -1];
            min--;
        }
        a[i] = key;
    }
}

int main () {
    int a[] = { 4,5,3,2,4,1 };
    int n = size of (a);
    stableSelectionSort (a, n);
    for ( int i=0 ; i<n ; i++)
        Cout << a[i] << " " ;
    Cout << endl;
    return 0;
}
```

Ishika

Page :
Date: / /

2 Sol (13) The easiest way to use external sorting we divide our source file into temporary file of size equal to 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 stored in a harddisk floppy disk or any other storage device, this is called external sorting

- Internal sorting :- If the input data is such that it can be adjusted in the main memory at once it is called Internal sorting

Scanned by CamScanner