Divakar Pandey
DS-1
44

Assignment -02

Design and Analysis of Algorithms

Ans 1:- If a array is sorted, a linear search may not be the efficient approach.

```
int linearsearch (int arr[], int n, int num)
{
        for (int i=0; i<n; i++)
        {
            if (arr[i]== num)
            {
               return-1;
            }
            else if
            {
               return-1;
        }   }
   return-1
}
```

The loop terminates early if the current element in the array is greater than num.

Ans 2:- pseudo code iterative.

```
void insertionsort( 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;
}
```

pseudo code recursive

```
void insertionsort ( int arr[], int n)
{
        if (n<=1)
          return;
```

```
insertionsort (arr, n-1)
int key = arr[n-1]
int j = n-2
while (j >= 0 && arr[j] > key)
{   arr[j+1] = arr[j]
    j = j-1
}
arr[j+1] = key;
}
```

Insertion sort is also known as online sorting algorithm because it can sort a list as it receive new element. In online sorting new elements are continuously added to existing sorted list and insertion sort maintain the new sorted order.

Other sorting algorithm.

Bubble sort:- Not suitable for large dataset. It repeatedly steps through the list, compare adjacent element and swap them if they are in wrong order.

Selection sort:- select the minimum elements from the unsorted portion and swap it with unsorted element.

Mergesort:- A divide and conqueur algorithm that recursively divide the array into halves, sort each half and then merge them together.

Quick sort:- Another divide and conqueur algorithm that partition the array into smaller segment and recursively applies the same process to each segment.

Heap sort:- Build the max/min heap and extrat the minimum element to put it at the end of array.

Ans:3

Sorting algorithm	Time complexity Best	Worst	Average
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Quick sort	$O(n\log n)$	$\cancel{O(n\log n)}$ $O(n^2)$	$O(n\log n)$
Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$

Ans 4:-

sorting algorithm	inplace	stable	online
Insertion	✓	✓	✓
Bubble	✓	✓	✗
Selection	✓	✗	✗
quick	✓	✗	✗
merge	✗	✓	✗

Ans 5:-

```
int binarysearchr ( int arr[] ,int low , int high, int num)
{.     if (low<= high)
       {
              int mid = low+ (high-low)/2.
              if arr[mid]== target
              {
                     return mid;
```

```
else if (arr[mid] > target)
    return binary search r (arr, low, mid-1, num) -

else   return binary.search r( arr, mid+1, high, target);

}

    }
    return -1
}
```

Binary search iterative.

```
int binary search i ( vector<int> arr, int target, ~~int~~
{   int low = 0
    int high = arr.size() - 1
    while (low <= high)
    {
        int mid = low + (high - low)/2;
        if (arr[mid] == target)
            return mid;
        else if ( arr[mid] > target)
            high = mid - 1
        else
            low = mid + 1;
    }
    return -1;
}
```

Time complexity	Best	Worst	Average	Space complexity
Recursive binary	O(1)	O(logn)	O(logn)	O(logn)
Iterative binary	~~O(logn)~~ O(1)	O(logn)	O(logn)	O(1)
linear search	O(1)	O(n)	O(n/2)	O(1)

Ans 6:- T(n) be the time complexity of binary search then recurrance relation is given by

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

$T(n/2)$ represent the time complexity of the binary search on subarray of size half the size.

$O(1)$ represent the time complexity of associated with comparison and recursive call.

Ans 7:- We use unordered map to store the complement of each element the time complexity of this code is O(n). The unordered map allow for constant time average-case lookup, making the algorithm efficient

```
pair < int, int> FindIndexForSum ( vector <int> &arr, int k)
{
    unordered_map < int, int> ComplementMap;
    for ( int i=0; i< arr.size(); i++)
    {
        int complement = k- arr[i]
        auto it = ComplementMap.find(Complement)
        if ( it != ComplementMap.end())
        {
            return { it-> second, i};
        }
        ComplementMap[ arr[i]] = i
    }
    return {-1, -1};
}.
```

Ans 8: Quick sort is the fastest general purpose sort. In most pratical situation, quick sort is the method of choice of stability is important and space is available, merge sort might be the best. Insertion sort perform well on the small dataset.

Ans 9:- Inversion count for an array indicate how far the array is from being sorted.

original array = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5}

1	20	6	4	5

Original array :- 7, 21, 31, 8, 10, 1, 20, 6, 4, 5

7, 21, 31, 8, 10

7, 21, 31   8, 10

7   21, 31

~~Sont~~
count inversion (7, 21)

7, 21, 31   8, 10

7, 21, 31   8   10

7 ~~7, 21,~~ 31   8   Count inversion (31, 8)

7 21 31 | 8, 10

7, 21, 31 | 8, 10

(21, 8) (31, 8) (31, 10) count inversion

7, 8, 10, 21, 31

1, 20, 6, 4, 5

1, 20 | 6, 4, 5

6 | 4, 5

Count Inversion (6, 4)

1, 20 | 4, 5, 6

Count Inversion (20, 4) (20, 5)

1, 4, 5, 6, 20 | 21, 31

(21, 6) (31, 6) (31, 4)

(31, 5)

4 Inversion

1, 4, 5, 6, 20 | 20, 31

(6, 1) (8, 1) (10, 1)

(8, 4) (10, 4) (10, 5)

(8, 5)

1, 4, 5, 6, 8, 10, 20, 21, 31 |]

(7, 6) (7, 4) (7, 5) (7, 8)

(7, 10) (7, 20) (7, 21)

(7, 31)

1, 4, 5, 6, 7, 8, 10, 20, 21, 7]

Total inversion 8+7+4+1+1+1+1+1 = 24

Ans 10:- Best case:- The base case time complexity for each sort quick sort occurs when partitioning is perfectly balanced meaning the algorithm consistently divides the input in two equal halves. This lead to a well-balanced recursive tree and each level of the tree processes roughly half the element of previous level. The best case time complexity is $O(n \log n)$. The pivot chosen for partitioning is the median of the input array. The input is already sorted or nearly sorted.

Worst case:- The worst case T.C. for quick sort occurs when partitioning is highly unbalanced leading to a skewed recursive tree. This happen when the chosen pivot consistently divides the array into one partiation with almost all the element and another with only few. The pivot chosen is smallest or the largest element in the array. The input array is sorted in ascending or descending order. The worst case T.C is $O(n^2)$.

Ans 11:- ~~Best case~~
Recurrence Relation for Merge sort.

Best Case:-

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

Worst Case

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

Quick sort

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

$$T(n) = T(n-1) + O(n)$$

Merge sort

follow divide and conqueur rule

$T(n) = 2T(n/2) + O(n)$ Best Case

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

Worst Time complexity $O(n\log n)$

Stable sorting algorithm

additional memory proportional

~~use po~~

- use merging step

Quick Sort.

follow divide and conqueur rule.

Best:- $T(n) = 2T(n/2) + O(n)$

Worst:- $T(n) = T(n-1) + O(n)$

Worst T.C:- $O(n^2)$

not stable sorting algorithm.

less additional memory.

uses a partitioning step.

Ans 12:-

```
void stable selectionsort (vector <int> & arr)
{
    int n = arr.size();
    for (int i=0; i<n-1; i++)
    {
        int minIndex = i;
        for (int j = i+1; j<n; j++)
        {
            if (arr[j] == arr [minIndex] &&
                                j < minIndex)
            {
                minIndex = j;
            }
            elseif ( arr[j] < arr [minIndex])
            {
                minIndex = j;
            }
        }
        int temp = arr [min Index];
        while ( min Index > i)
        {
            arr [minIndex] = arr [minIndex-1];
            minIndex--;
```

```
    }
    arr[i] = temp;
  }
}
```

Ans 13:-

```
void Bubblesort (vector<int>& arr)
{
    int n = arr.size();
    bool swapped;
    for (int i=0; i<n-1; i++)
    {
        swapped = false;
        for (int j=0; j<n-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
    }
    if (!swapped)
    {
        break;
    }
}
```

Ans 14: When dealing with a dataset that is larger than the available physical memory (RAM), external sorting technique become essential. One commanly used algorithm for external sorting in Merge Sort.

Internal sorting:- refer to the purpose of sorting data that can be ~~accoma~~ accommodated entirely with the computer's main memory (RAM).

All data fits into the primary memory.

Fast access to data in RAM leads to quick sorting.

No Need for time-consuming input/output operatio

External Sorting:- External sorting is used when the size of data to be sorted exceed the capacity of the computer's main memory.

Allow sorting of dataset larger than available RAM.

Slower compared to internal sorting due to the involved of disk I/O.