Design & Analysis of Algorithm

Tutorial - 3

MOHAK KALA
2014727
Section - D; Class R.No.- 29

① Linear Search Pseudocode =>

```
int UnOrderedLinearSearch (int A[], int n, int data)
{
    for (int i = 0; i < n; i++)
    {
        if (A[i] == data)
            return i;
    }
    return -1;
}
```

Time Complexity = O(n) worst case

Space Complexity = O(1)

② Insertion Sort =>

Iterative =>

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

② Recursive ⇒

```
void insertionsort recursive (int arr[], int n)
{
  if (n <= 1)
    return;
  insertionsortRecursive (arr, n-1);
  int last = arr[n-1];
  int j = n-2;
  while (j >= 0 && arr[j] > last)
  { arr[j+1] = arr[j];
    j--;
  }
  arr[j+1] = last;
}
```

③ Complexity of Sorting Algorithms ⇒

Algorithms	Time Complexity		
	Best	Average	Worst
1. Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
2. Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
3. Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
4. Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
5. Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
6. Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$

it is the total sorted pair.

(4) Sorting Algorithms =)

(i) Inplace => Bubble Sort, Selection Sort, Insertion Sort, Heap Sort

As in-place alg. is an alg. that does not need an extra space & produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variable is allowed.

(ii) Stable => Merge Sort, Insertion sort, Bubble Sort Counting Sort

If 2 objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

(iii) Online => Insertion Sort

If you give input one by one to the algorithm and each input produces some partial solution with available input data.

④

⑤ Binary Search ⇒ ~~Iterative~~ Recursive

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

Iterative Method ⇒

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

	Time Complexity	Space Complexity
Linear Search	$O(n)$	$O(1)$
Binary Search	$O(\log n)$	$O(1)$

(6) Recurrence Relation for binary search

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

(7) 2 indexes such that $A[i] + A[j] = k$ in the min. time complexity

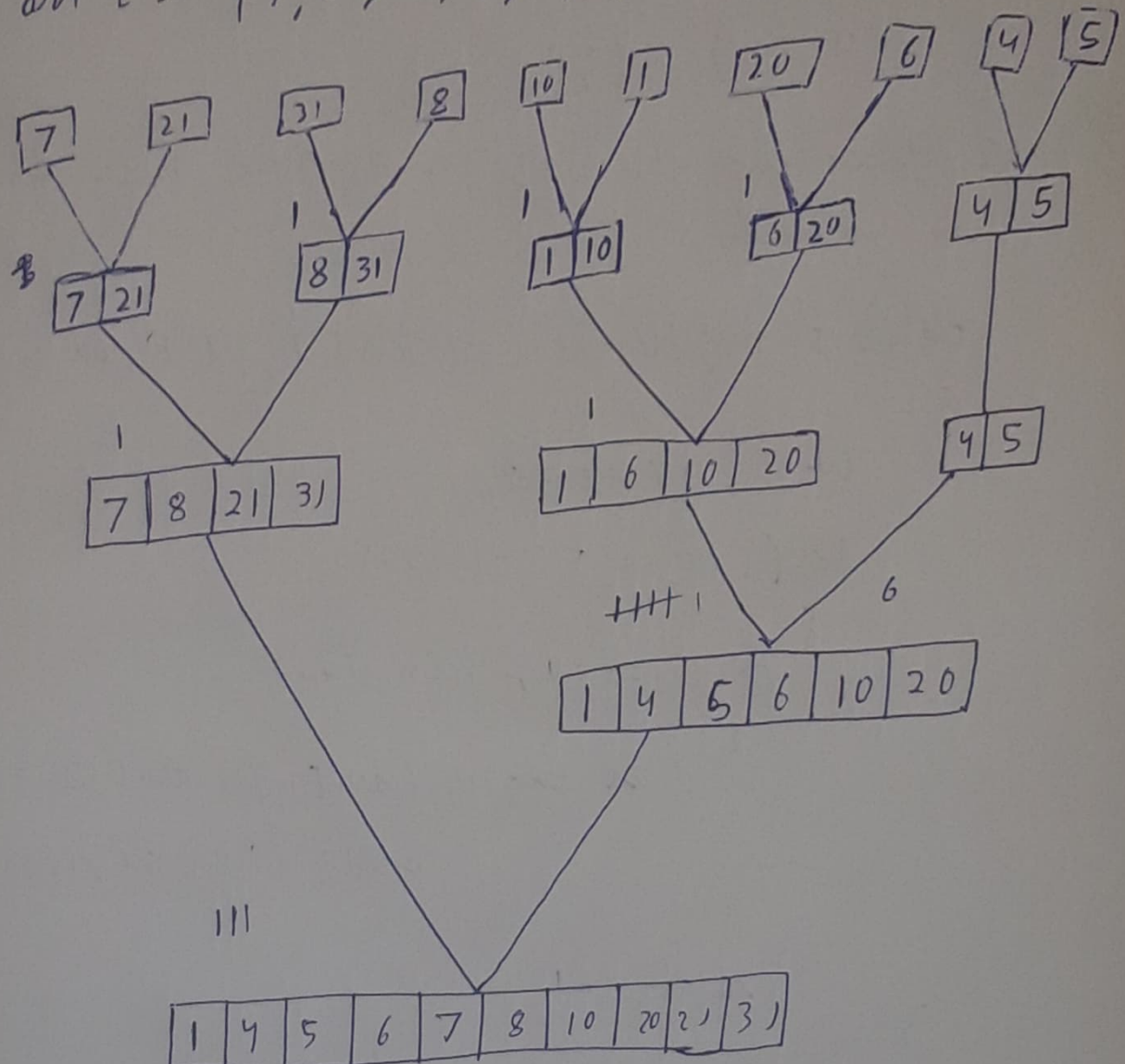
```
int sumofElement (int arr[], int k, int n)
{
    int i, j;
    for(i=0; i < n-1; i++)
    {
        for(j=i+1, j<n; j++)
        {
            if (arr[i] + arr[j] == k)
                printf("%d, %d", i, j);
        }
    } return -1;
}
```

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

⑥

⑨ No. of inversions ⇒ No of inversions in an array are the no. of total swaps performed on the elements of the array to satisfy the whole array in a particular condition.

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



(10) Quick Sort has a time complexity of

$\Omega(n \log(n))$ Best Case => ~~This is when the array is already sorted~~ which is when the partition process always picks the middle element as the pivot

$O(n^2)$ worst case => This is when the partition process picks up the extreme element (smallest or largest) element.

This happens when input array is sorted or reverse sorted and either first or last element is picked up as pivot for partition.

(11) Merge Sort => Recurrence Relation =>

Best Case => $T(n) = O(n \log n)$

Worst Case => $T(n) = O(n \log n)$

Quick SORT => Recurrence Relation =>

Best Case => $T(n) = 2T\left(\frac{N}{2}\right) + N-1$

Worst Case => $T(n) = T(n-2) + N-1$

Similarities => Both the sorting algorithms work on the principle of "Divide & Conquere"
Both are comparison based sorting based algorithm where each of the inputs are compared

(8)

Differences => 1. The worst case complexity of quick sort is $O(n^2)$, whereas of merge sort is $O(n \log n)$

(12) Stable Selection Sort => Selection sort is not stable by default, but any comparison based sorting algorithm which is not stable by nature can be modified to be stable by modifying the key comparison operation so that the comparison of 2 keys considers position as a factor for objects with equal key or by tweaking it in a way such that its nearing does not change and it becomes stable as well.

Selection Sort can be made stable if instead of swaping, the minimum element is placed in its position without swaping i.e. by placing the no. in its position by pushing every element one step forward.

```
void stable_SelectionSort (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;
}
```

(13) Short Bubble Sort => Best Case => O(n)

We can improve bubble sort by using one extra flag. No more swaps indicate the completion of sorting. If the list is already sorted, we can use this flag to skip the remaining passes.

```
void BubbleSortImproved (int A[], int n)
{
    int pass, i, temp, swapped = 1;
    for (pass = n-1; pass >= 0 && swapped; pass--)
    {
        swapped = 0;
        for (i = 0; i <= pass - 1; i++)
        {
            ~~swapped = 0;~~ if (A[i] > A[i+1])
            {
                temp = A[i];
                A[i] = A[i+1];
                A[i+1] = temp;
                swapped = 1;
            }
        }
    }
}
```

(13) If our computer has a RAM (Physical Memory) of 2 GB and we are given an array of 4 GB for sorting.

We'll use External Sorting, which is used in such cases where our file is ~~goes~~ bigger than the main memory.

As with internal sorting algorithms, there are a no. of algorithms, there are a number of algs for external sorting. One such ~~external~~ algo is External Merge -Sort. In practice, these external sorting algorithms are being supplemented by internal sorts.

Simple External Mergesort ⇒

A no. of records from each tape are read into main m/m, sorted using an internal sort, and then output to the tape.

1. Read 500 MB of data into main memory and sort by some conventional method (let us say quick sort)
2. Write sorted data to disk.
3. Repeat steps 1 & 2 until all of the data is sorted in chunks of 500 MB, now we need to merge them into one single sorted output file.
4. Read the first 100 MB of each sorted chunk (input buffers) in main m/m (400 MB in total) and allocate remaining 100 MB for output buffer.
5. Perform the 4 way merge sort & store the result in output buffer, & if output buffer is full, write it to the final sorted file. -S