Date

Page

Assigment - 2

1.) Linear Search

```
function Ls(arr[], n) //where n = number of terms
{
    for (i to n)
```

```
function ls(arr[], n, k) /* where n = number of
                            terms and k = element to
                            search */
{   flag = 0
    for (i to n-1)
    {
        if (k == arr[i])
        {
            flag = 1
        }
    }
    return 1 ; // returns 1 if element found.
}
```

Date
Page

2. Insertion Sort

a. Recursive approach.

```
function insertion(arr[], n)
{
    if n<=1
    {
        return
    }
    insertion(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;
}
```

b. Iterative Approach

```
function insertion (arr[], n)
{
    for (i from 1 to n-1)
    {
        key = arr[i]
        j = j-1
```

Date

Page

```
while ( j>=0 and arr[j] > key)
{
   arr[j+1] = arr[j];
    j = j-1
}
    arr[j+1] = key
}
}
```

Assume an Array : [9, 4, 2, 2, 3, 4, 1]

Lets encode this array as [9a, 4a, 2a, 2b, 3a, 4b, 1a]
↳ Each iteration of insertion sort

~~1st → [9a, 4a, 2b, 3~~

1st → [9a, 4a, 2a, 2b, 3a, 4b, 1a]
2nd → [4a, 9a, 2a, 2b, 3a, 4b, 1a]
3rd → [2a, 4a, 9a, 2b, 3a, 4b, 1a]
4th → [2a, 2b, 4a, 9a, 2b, 3a, 4b, 1a]
5th → [2a, 2b, 3a, 4a, 9a, 4b, 1a]
6th → [2a, 2b, 3a, 4a, 4b, 9a, 1a]
7th → [1a, 2a, 2b, 3a, 4a, 4b, 9a]

Since in our acctual array 2a, 4a were before 2b, 4b respectively we can say that the element present in actual array is same after other is sorted array and hence insertion sort is online for sorting.

classNotes

Date

Page

Other online Sorting -

1. Insertion Sort – ✓
2. Merge Sort – ×
3. Heap Sort – ✓
4. Quick Sort – ✓
5. Radix Sort – ✓
6. Count Sort – ✓
7. Bubble – ✓
8. Selection – ✓
9. ~~Quick sort – ✓~~

3. Sort	Time Complexity	Space Complexity
1. Bubble Sort	$O(n^2)$	$O(1)$
2. Heap Sort	$O(n \log n)$	$O(1)$
3. Selection Sort	$O(n^2)$	$O(1)$
4. Insertion Sort	$O(n^2)$	$O(1)$
5. Quick Sort	$O(n^2)$	$O(n)$
6. Merge Sort	$O(n \log n)$	$O(n)$
7. Counting Sort	$O(n+k)$	$O(n+k)$
8. Radix Sort	$O(n * k)$	$O(n+k)$

* All space and time complexity for worst cases only.

Date

Page

4.

Sort	In Place	Stable	On-line
Bubble	1	1	0
Heap	1	0	0
Insertion	1	1	1
Quick	1	0	0
Count	0	1	0
Merge	0	1	0
Radix	0	1	0
Selection	1	0	0

where 1 = Yes
0 - No

5 Binary Search

Iterative Approach

```
function Bs(arr[], n, key) /* Conditoion → array must be sorted */
{   r = n
    l = 0
    ~~mid = n-1/2~~
    while (l <= r)
    {
        mid = (l + (r-1))/2
        if(arr[mid] == key)
        {
            return true
        }
        else if(arr[mid] > key)
```

classNotes

Date

Page

```
    {
        l = mid - 1
    }
    else
    {
        r = mid - 1
    }
} // loop ends if element not found
return false
}
```

Recursive Approach

~~function Bs(arr~~

```
function Bs(arr[], l, r, key)
{   if (l <= r){
        mid = (l + (r - 1)) / 2
        if (arr[mid] == key)
            return True
        else if (arr[mid] < key)
            return Bs(arr, mid+1, r, key)
        else
            return Bs(arr, 0, mid - 1, key)
    }
    return false
}
```

Date

Page

Linear Search (iterative)	Binary Search (~~Recursive~~ iterative)
T.C = O(n)	T.C = O(log(n))
S.C = O(1)	S.C = O(1)

Linear Search (recursive)	Binary Search (Recursive)
T.C = O(n)	T.C = O(log(n))
S.C = O(1) ✗	S.C = O(log(n)) ✓
No stack used	Stack to recursive calls.

6.) Recurrence Relation Binary Search

Pseudo Code :-

```
T(n) => Bs(a, i, j, x)        — T(1) O(1)
        mid = (i+j)/2         — T(n) O(1)
        if (a[mid] == x)
            return (mid)
        if (a[mid] > x)
        ** return BS(a, i, mid-1, x) ← T(n/2)
        else
        ** return BS(a, mid+1, j, x) ← T(n/2)
```

reason for **

Assume x = 20

mid

10 20 30 [40] 50 60 70

← search in left half as x < a[mid]

10 [20] 30 **

return 20 O(1)

if x = 10, array would again split in half and

classNotes

Date
Page

and we can say $T(n) = T(n/2) + 1$

7. In this we will use two pointers.

Approach:

n element array

if $a[i] + a[j] == Sum$ → return
$a[i] + a[j] < Sum$ → increase i
$a[j] + a[i] > Sum$ → increase j.

} This will only work if array is sorted

if array is not sorted we have to use ~~loaner~~ linear search approach

Two pointer approach + merge sort approach	~~merge s~~ Linear Search approach
T.C merge sort → $O(n \log n)$ T.C two pointer → $O(n)$ Total = $O(n \log n + n)$ since $n \log n > n$ it will be dominant Total T.C. = $O(n \log n)$	T.C linear Search → first a element i is fixed which j iterates so $TC = O(n^2)$

On comparing we find merge sort + two pointer approach is better than linear search.

Date

Page

```
function search(A[], n, K
{
    i = 0
    l = n-1
    if((A[i] + A[l]) == k)
    {
        return <i, l>
    }
    else if
```

```
function search (A[], n, k]
{   ms(A, n) // call for merge sort.
    i = 0
    j = n-1
    while (i < j)
    {
        if((A[i] + A[j]) == k)
        {
            return i, j
        }
        else if ((A[i] + A[j]) < k)
        {
            i++;
        }
        else
        {
            j++;
        }
    }
    return -1, -1
}
```

Date
Page

8. Which algorithm is ~~best~~ best for practical use.

We can't say or assume any particular algorithm as best algorithm because each algorithm has its own pros and cons. We must consider sorting as per use. For example.

1. Merge Sort → Can be used where stability is important and has ample amount of free memory No suitable for large database.

2. Quick Sort → It can be used where data is uniformly distributed and more emphasises on ~~sto~~ time. ~~use~~ ~~test~~ less memory and can be used in database

3. Heap Sort → No additional memory is required but is little slower than above two. Can be used in DB sorting.

4. Insertion Sort → Used is smaller database or questions where some part needs to be sorted.

5. Selection Sort → T.C being high is practically not used as much but can be implied in smaller sits.

6. Bubble Sort → Rarely used.

Date

Page

9. Inversions

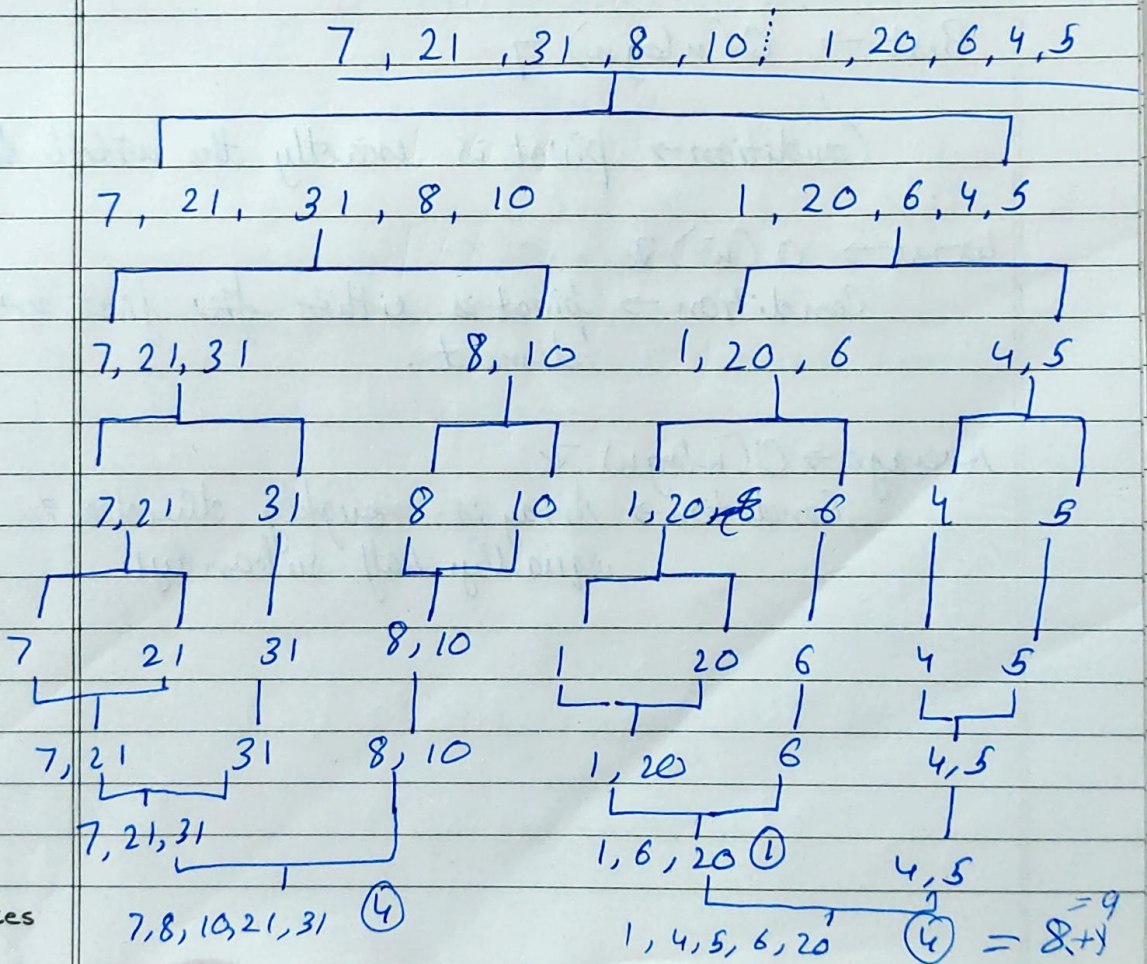
Number of inversion means how many element is shifted before placing new element at its prescribed location.

For eg ⇒

1, 2, 7, 10, 12, 13 has to accomodate 4

1, 2, 4, 7, 10, 12, 13 → inversion = 4

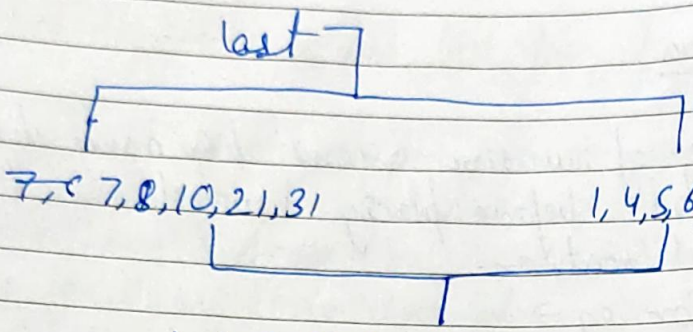
Merge Sort



classnotes

Date

Page



1, 4, 5, 7, 8, 10, 20, 21, 31 → 5 + 5 + 5 + 2 = 17

Total = 17 + 9 = 26

10. Quick Sort

Best → $O(n \log n)$ ↓

Condition → pivot is exactly the middle element

Worst → $O(n^2)$ ↓

Condition → pivot is either the first or last element.

Average → $O(n \log n)$ ↓

Condition → Array is roughly divided in equally half subarrays

Date

Page

12. **STable Quick Sort**

```
void sqs(vector<pair<int,int>& arr) {
    /* first element holds value second holds
       occurence */
    int n = arr.size() - 1;

    for(int i = 0; i < n; i++)
    {
        int min_idx = i;
        for(int j = i+1; j < n; j++)
        {
            if (arr[j].first < arr[min_idx].first)
            {
                min_idx = j;
            }
        }
        pair<int,int> key = arr[min_idx];
        while(min_idx > i){
            arr[min_idx] = arr[min_idx-1];
            min_idx--;
        }
        arr[i] = key;
    }
}
```

Date

Page

13. Bubble sort without scanning whole array.

Beside double check loop we can use single loop in which one element is at i position and other is at i+1. if a[i] < a[i+1] then go for next comparison if not then call bubble sort for i to n indexs.

Comparison Code

```
void bs(int a[], int i, int n)
{
    if (
```

13. Bubble sort after sorting.

This can be achieved by pairing element with boolean data type. if it is sorted then it will store true else false. If true is found at 0th index break the condition and come out else iterate for bubble sort.

Code

```
void bs(&vector<int, bool>& arr)
{
    int n = &arr.size() - 1;
```

Date

Page

```
for(i=0; i<n; i++)
{
    if (arr[0].second == true)
    {
        break;
    }
    else
    {
        for(j=i+1; j<n; j++)
        {
            if (arr[j]<arr[i])
            {
                Swap(i, j);
                Swap(arr, i, j);
            }
        }
    }
}
arr[0].second = true; // after first call turns a[0] true.
}
```

Note ** → if array is of size n, i.e n > n & then if a element is added after sorting array will not able to sort again. to achieve capability to sort again, while inserting a element change arr[0].second to false again

Date
Page

13. Merging of data greater than available memory

We cant sort data of 4GB in 2GB memory to do so we have to use External Sorting methods like External merge sort or External quick sort.

External Sorting → It is sorting in which data is sorted and stored in slower memory for a while. Then it is retrived from file to sort with other data. This chuncking of data allowes us to sort large data with less resorces usage.

Internal Sorting → It is sorting in which less memory is directly used and no secondary memory is used. It can only accomodate chunck smaller than available memory.

Speed Comparison.

Internal > external

Storage can be sorted

External > internal