Date

Page

DAA
Assignment 2

Q1:- Write linear search pseudocode to search an element in a sorted array with minimum comparisions.

ans:- linear search pseudocode:

```
int linearsearchsorted (int array[], int n,
                        int key) {

 int index = -1;
for (int i=0; i<n; i++) {
 if (array[i] == key)
{
  index = i;
  break;
}
if (array[i] > target)
{
break;
}
}
return index;
}
```

Q2:- Write pseudocode for iterative and recursive insertion sort. Insertion sort is called online sorting why? What about other sorting algorithms that has been discussed?

Iterative insertion sort:-

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

Date

Page

Recursive Insertion sort

```
void insertion ( int array[ ], int n)
{
 if ( n <= 1)
   return ;
 insertion ( array , n-1 );
 int key = array [n-1 ];
  int j = n-2;

while ( j >=0 && array[j] > key ) {
    array [j+1] = array [j ];
      j-- ;
 }

   array[j+1] = key ;
}
```

Online sorting refers to the ability of sorting Algo to dynamically sort a list as elements are presented, without requiring the entire list to be known in advance.

Insertion sort is considered online because it efficiently builds a sorted list incrementally, making it suitable for real time, dynamic data input.

Date
Page

The online nature of insertion sort is beneficial in scenarios where data is continuously streaming and you need to maintain a sorted order without having the entire dataset available at the start.

Quick sort, Merge sort

- Divide and conquer algorithms.
- Need full datasets for partitioning & merging.

Selection sort, Bubble sort, Heap sort

- Rely on entire array or heap for sorting.

Radix sort

- Sorts digits at different places.
- Require knowledge of entire dataset.

Bucket sort

- Can be adapted for online scenarios.
- Standard version assumes knowledge of entire value range.

Hence, as above is not called online sorting Algorithm

Count sort

- Require full dataset
- Not Adaptive to real time, incremental input.

Date

Page

Q3: Complexity of all sorting algorithms that has been discussed.

1) Bubble sort

Best case	$O(n)$
Worst case	$O(n^2)$
Average case	$O(n^2)$
Space complexity	$O(1)$

2) Selection sort

Best case	$O(n^2)$
Worst case	$O(n^2)$
Average case	$O(n^2)$
Space complexity	$O(1)$

3) Merge sort

Best case	$O(n \log n)$
Worst case	$O(n \log n)$
Average case	$O(n \log n)$
space complexity	$O(n)$

Date
Page

4) Quick sort

Best case	$O(n \log n)$
Worst case	$O(n^2)$
Average case	$O(n \log n)$
Space complexity	$O(\log n)$

5) Insertion sort

Best case	$O(n)$
Worst case	$O(n^2)$
Average case	$O(n^2)$
space complexity	$O(1)$

6) Counting sort

Best case	$O(n+k)$
Worst case	$O(n+k)$
Average case	$O(n+k)$
Space complexity	$O(k)$

n → no of elements in array.
K → diff b/w max & min value.

7) Radix sort

Best case	$O(nk)$
Worst case	$O(nk)$
Average case	$O(nk)$
space complexity	$O(n+k)$

Date

Page

8) Heap sort

Best case	O(nlogn)
Worst case	O(nlogn)
Average case	O(nlogn)
Space complexity	O(1)

9) Bucket sort

Best case	O(n+K)
Worst case	$O(n^2)$
Average case	$O(n^2)$
space complexity	O(n+K)

Q4:- Divide all sorting algorithm into inplace / stable / online sorting.

inplace sorting → sorting Algo that donot requir additional memory proportional to the input size; it sorts the data with only a constant amount of extra memory.

Date
Page

2) Stable sorting
A sorting Algo is stable if it maintains the relative order of equal elements in the sorted output as they appeared in input.

3) Online sorting :-
A sorting Algo is considered online if it can handle data elements as they arrive, updating the sorted structure dynamically without needing the entire dataset at once.

4) External sorting :-
sorting algo designed to handle massive datasets that do not fit into computer's main memory, often involving the use of external storage like disks.

Algo.	In place (or)	Stable	Online	External
Bubble	✓	✓	×	×
selection	✓	×	×	×
Merge	×	✓	×	✓
Quick	✓	×	×	✓
Insertion	✓	✓	✓	×
Counting	×	✓	×	×
Radix	×	✓	×	×
Heap	✓	×	×	×
Bucket	×	✓	×	×

Q5: Write recursive / iterative pseudocode for binary search. What is time and space complexity of linear and binary search (Recursive and Iterative)

Recursive binary search

```
int binarysearch (int array[], int key, int low, int high);
if (low <= high)
{
int mid = (low) + (high - low)/2;
if ( arr[mid] == key)
  return mid ;
elseif (array[mid] < key)
 {
return binarysearch (array, key, mid+1, high);
 }
else
{
return binarysearch ( array, key, low, mid-1);
}
} else {
   return -1;
  }
}
```

Date
Page

Iterative binary search

```
int binarysearch (int array[], int key, int n)
{
int low=0;
int high= n-1;
while (low <= high)
{
int mid= low + (high-low)/2;
if (array[mid] == key)
  return mid;
else if (array[mid] < key) {
 low = mid +1;
 } else
 {high = mid +1;
 }
}
return -1;
}
```

Binary search

time complexity

worst case → O(log n)

space complexity → O(log n) → recursive
due to recursive call stack.

→ O(1) → iterative

Linear search

time complexity (worst) → O(n)

Space → O(1)

Date

Page

Q6: Write recurrence relation for binary recursive search.

Recurrence relation

↳ is used to determine the relationship b/w time complexity of problem and its subproblems.

T(n) → bool binarySearch (int arr[], int l, int r, int key)

{

if(l > r)

return false;

int mid = l + (r-l)/2;

if(arr[mid] == key) → O(1)

return true;

else if (arr[mid] < key) → T(n/2)

return binarySearch (arr, mid+1, r, key);

else

return binarySearch (arr, l, mid-1, key); → T(n/2)

}

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

$\boxed{T(n) = T(n/2) + 1}$ → recurrence relation

finding complexity using master theorem.

$T(n) = T(n_1) + 1$

$a = 1 \quad b = 2$

$c = \log_b a$

$c = \log_2 1$

$= \log 1$

$= 0$

$n^c = f(n)$

$n^0 = 1$

$c = 0$

$c = n^c \rightarrow O(n^c \log n)$

$= O(n^0 \log n)$

$= O(\log n)$ Ans

Q7:- find two indexes such that $A[i] + A[j] = K$ in minimum time complexity.

Using hash table:

```
void pairwithsum (int arr[], int n, int K) {
int seen[100000] = {0};
for (int i=0; i< n; i++) {
int complement = K - arr[i];
if (seen[complement]) {
printf("pair found at index %d and %d:
%d + %d = %d \n",
seen[complement]-1, i, arr[seen[complement]
-1], arr[i], K);
return;
}
```

```
        seen[arr[i]] = i+1;
    }
    printf("No pair found\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int K = 8;
    findPairSum(arr, n, K);
    return 0;
}
```

Q6:- What/which sorting is best for practical uses? Explain.

Choice of best sorting Algo for practical uses dependent on various factors such as size of dataset, the distribution of data, memory constraints etc.

1. Quick sort

Advantages :-

fast average use performance, inplace sorting, widely used in practice.

→ frequently used in practice for general purpose sorting

Consideration :-
can have poor performance on already sorted data, not stable.

2) Merge sort :-

Advantages :-
Stable, consistent performance across different cases, suitable for linked list.
→ commonly used for general purpose sorting when stability & consistency are imp.

Consideration :-
Requires additional space (non-in place)
may not be as fast as quick sort.

3) Heap sort :-

Advantages :-
In place sorting, worst case O(nlogn), no worst case scenario data dependency.

Consideration :-
slower in place on average compared to quick sort and merge sort.

Practical use :-
Used in scenario where in place sorting & worst case time complexity are critical.

Date
Page

4) Insertion sort

Advantages :-
Simple, efficient for small datasets, adaptive (performs well on partially ordered data).

Considerations :-
Inefficient for large datasets, not suitable for highly unsorted data.

5) Radix sort

Advantages :-
Linear time complexity, stable, efficient for size fixed integer.

Consideration :-
Requires additional space, limited to specific types of data.

6) Bucket sort :-

Advantages :-
Liner time complexity, efficient for uniformly distributed data.

Date
Page

Consideration
Aquires knowledge of data distribution, additional space.

7) Counting sort:

Advantages :- Linear time complexity, stable.

Disadvantages :- Limited to specific type of data, requires additional space.

Practical use :- Efficient for sorting integers when the range of values is limited.

8) Selection sort :

Advantages :- Simple & easy to implement.

Disadvantage :- Inefficient for large dataset.

Practical use :- ~~for~~ limited to small dataset or as an educational tool.

Date

Page

9) Bubble sort

Advantage :- Simple implementation

Disadvantage :- Inefficient for large datasets.

Practical Use :- Limited to small datasets or educational purposes.

Q9:- What do you mean by no. of inversions in an array? Count the no. of inversions in Array arr[] = {7, 21, 31, 8, 10, 1, 20, 6, 4, 5} using merge sort.

An inversion in an array occurs when 2 elements are out of their natural or sorted order.

More formally, for an array 'arr', if there are indices 'i' and 'j' such that $i < j$ and $arr[i] > arr[j]$, then pair of elements $[arr[i], arr[j]]$ form an inversion

Date _____

Page _____

```
int mergeandcountinversion (int arr[], int
                            temp[], int left, int
                            right)

{
int i = left, j = mid+1, k = left;
int inv_count = 0;
while (i <= mid && j <= right) {
  if (arr[i] <= arr[j])
  {
  temp[k+1] = arr[i+1];
  }
else {
  temp[k+1] = arr[j+1];
  int count = inv_count + (mid - i+1);
  }
}
while (i <= mid) {
  temp[k+1] = arr[i+1];
}
while (j <= right) {
  temp[k+1] = arr[j+1];
}
for (i = left; i <= right ; i++) {
  arr[i] = temp[i];
  }
return inv_count;
}
```

Date _____

Page _____

```
int mergesort_and_count_inversion (int arr[], int temp[],
                                   int left, int right)
{
  int inv_count = 0;
  if (left < right)
  {
    int mid = (l) + (r-l)/2;
    int l_count = inv_count + mergesort_and_count_inv
                              (arr, temp, left, mid);
    int r_count = inv_count + mergesort_and_cou
                              (arr, temp, mid+1, right
    int l_r_count = inv_count + mergesort_and_count
                              (arr, temp, left, mid,
                               right);

  }
  return inv_count;
}

int count_inversions (int arr[], int n){
    int temp[n];
    return merge_sort_and_count_inv (arr, temp, 0,
                                     n-1);
}
```

Date
Page

```
int main () {
int arr [] = { 7, 21, 31, 8, 10, 1, 20, 6, 4, 5};
int n = size of (arr) / size of (arr[0]);
int temp[n];
int inversion = count inversion (arr, n);
scheno;
}
```

no. of inversion = 27

10] In which case Quick sort will give best and worst case time complexity?

⇨ Best case

In best case, the array is partitioned into 2 roughly equal halves at each level of recursion. This leads to balanced recursive tree,

time complexity $O(n \log n)$

⇨ Worst case

In worst case, array is consistently partitioned into one significantly smaller & one significantly larger subarray. This result in unbalanced recursive tree, resembling a linear structure,

time complexity $O(n^2)$

Date

Page

Average case

Time complexity → $O(n \log n)$

This assumes that pivot value are chosen randomly leading to balanced partitioning most of time.

11) Write Recurrence relation of Merge & Quick sort in best & worst case? what are the similarities & diff b/w complexities of 2 algorithm & why?

Merge sort

Recurrence relation

Best case $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Worst case $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Quick sort

Recurrence relation

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

Worst $T(n) = T(n-1) + \Theta(n)$

Date
Page

Similarities :

1) Both Merge & Quick sort have a divide & conquer approach.

2) In best case scenario, both algo have a time complexity of O(nlogn)

Difference

1) Worst case :-

Mergesort :- O(nlogn) in all cases.
Quicksort :- $O(n^2)$ in worst case.

2) stability :-

Merge sort :- stable - Equal elements maintain their relative order
Quick sort :- Not stable - Equal elements might change their relative order.

3) space complexity

Mergesort - Requires additional space for merging hence inplace.
Quicksort → generally inplace, but may require additional space for recursive call stack.

Date

Page

Why:-

↳ Worst case time complexity of quick sort is $O(n^2)$ when poorly chosen pivot lead to imbalanced partitions.

↳ Mergesort, on other hand consistently maintains $O(n \log n)$ time complexity in both best and worst case.

↳ Quick sort tends to perform well in practice due to its smaller constant factor & cache efficiency.

↳ Merge sort is preferred when stability & predictable performance are crucial & additional space is not a concern.

Date

Page

Code quick sort

```
void swap (int arr[], int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}

int position ( int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

for (int j = low; j < high ; j++) {
    if (arr[j] <= pivot)
    {
    i++;
    swap (arr, i, j);
    }

  swap (arr, i+1, high);
    return i+1;
}

void quicksort (int arr [], int low, int high) {
    if (low < high) {
    int pivotindex = partition (arr, low, high);
    quicksort (arr, low, pivotindex - 1);
    quicksort (arr, pivotindex + 1, high);
  }
}
```

* quicksort (arr, 0, n-1);

Date

Page

Q12: Selection sort is not stable by default but can you write a version of stable selection sort.

Selection sort is inherently not stable because it performs inplace swaps without considering the origin order of equal elements.

However, we can modify the selection sort Algo to make it stable by performing a stable selection within inner loop.

```
void swap(int arr[], int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}

void stableSelectionSort(int arr[], int n) {
for (int i=0; i<=n-1; i++) {
    int min = i;
for (int j=i+1; j<n; j++)
{
  if (arr[j] < arr[min]) {
      min = j;
  }
}
```

```
int minval = arr[min];
  while (min > i) {
    arr[minIndex] = arr[minIndex-1];
     minIndex--;
    }
  arr[i] = minvalue;
  }
}
```

Q13: Bubble sort scans whole array even when array is sorted. Can you modify the bubble sort so that it doesn't scan the whole array once it is sorted.

```
void swap (int arr[], int a, int b) {
   int temp = arr[a];
   arr[a] = arr[b];
   arr[b] = temp;
  }
void bubble (int arr[], int n) {
 for (int i=0; i<n-1; i++) {
    int swapped = 0;
 for(int j=0; j<n-i-1; j++) {
     if (arr[j] > arr[j+1]) {
       swap (arr, j, j+1);
       swapped = 1;
    }
 }
```

```
if (swapped == 0)
    break;
}
}
```

Q: Your computer has a RAM (physical memory) of 2 GB and you are given an array of 4 GB for sorting. What algo you are going to use for this purpose and why? Also explain concept of External and Internal sorting?

Internal sorting

↳ Sorting data that easily fits into ur computer's memory (RAM).

↳ Algo like Quick sort, Merge sort etc can be used because they operate on data that's all in RAM.

External Sorting:

1) Memory limitation:
When data to be stored is too large to fit into ur computer memory.

Date _____

Page _____

4 Requires dividing the data into smaller chunks that can fit into memory.

2) Sorting in Pieces

→ Each chunk is sorted individually.

→ Common algo for external sorting include Merge sort, External quick sort.

3) Merging Sorted Parts

→ After sorting, sorted chunks are merged to produce a final sorted result.

→ This merging process is what makes Merge sort particularly useful for external sorting.

Choosing Merge sort for External Sorting

1) Stable sorting

Merge sort maintains the order of equal elements, which is good for keeping things in order.

2) Memory efficiency

Merge sort's approach of sorting and merging in chunks works well when you can't fit everything into memory at once.

Date

Page

3) Predictable Reading / Writing

When dealing with external storage like a hard disk, Mergesort uses a pattern of reading & writing data that is easier to manage.