

Name - Tami Sabharwal
Class & Sec - MI
Roll no. - 33

Date / /
Page _____

Assignment - 2

Ques1 Write linear search pseudocode to search an element in a sorted array with minimum comparisons.

function linearSearchSorted(arr, target):

n = length of arr

for i from 0 to n-1:

If arr[i] == target:

return i

else if arr[i] > target:

return "Element not found"

return "Element not found".

Ques2 Write pseudocode for iterative and recursive insertion sort. Insertion sort, insertion sort is called online sorting. Why? What about other sorting algorithms that has been discussed in lectures?

Iterative Insertion Sort -

function iterativeInsertionsort(arr):

n = length of arr

for i from 1 to n-1:

key = arr[i]

j = i-1

while j >= 0 and arr[j] > key:

arr[j+1] = arr[j]

j = j-1

arr[j+1] = key

Recursive Insertion Sort -

function recursiveInsertionsort (arr, n):

 if $n \leq 1$:

 return

 recursiveInsertionsort (arr, n - 1)

 Key = arr[n - 1]

 j = n - 2

 while $j \geq 0$ and $arr[j] > key$:

 arr[j + 1] = arr[j]

 j = j - 1

 arr[j + 1] = key

Ques Complexity of all the sorting algorithms that has been discussed in lectures

1. Bubble Sort

→ Time Complexity

- Best case : $O(n)$

- Average case : $O(n^2)$

- Worst case : $O(n^2)$

→ Space complexity : $O(1)$

2. Selection Sort

→ Time Complexity

- Best case : $O(n^2)$

- Average case : $O(n^2)$

- Worst case : $O(n^2)$

→ Space complexity : $O(1)$

3. Insertion sort

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

4. Merge sort

- Time complexity -
 - Best case: $O(n \log n)$
 - Average case: $O(n \log n)$
 - Worst case: $O(n \log n)$
- Space complexity - $O(n)$

5. Quick sort

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

6. Count sort

- Time complexity - $O(n+K)$
- Space complexity - $O(n+K)$

Ques. Divide all the sorting algorithms into inplace/stable/online sorting.

→ 1. Inplace sorting algorithms:-

This sort the elements within the array without requiring any extra space for sorting.

• Examples include:

- Bubble Sort -
- Selection Sort
- Insertion Sort
- Quick Sort

2. Stable Sorting Algorithms :-

This maintain the relative order of records with equal keys.

• Examples include:

- Merge Sort
- Insertion Sort
- Bubble Sort

3. Online Sorting Algorithms :-

This process the data piece by piece, as it becomes available.

→ Insertion sort

Ques 5

Write recursive / iterative pseudo code for binary search. What is Time and Space complexity of linear and binary search (Recursive and iterative).

1. Recursive Binary Search :

```
function binarySearchRecursive (array, left, right, target):  
    if left <= right  
        mid = left + (right - left) // 2  
        if array [mid] == target:  
            return mid  
        elif array [mid] < target:  
            return binarySearchRecursive (array, mid + 1,  
                                         right, target)  
        else:  
            return binarySearchRecursive (array, left,  
                                         mid - 1, target)  
    else:  
        return -1
```

2. Iterative Binary Search :

```
function binarySearchIterative (array, target):  
    left = 0  
    right = length (array) - 1  
    while left <= right:  
        mid = left + (right - left) // 2  
        if array [mid] == target:  
            return mid  
        elif array [mid] < target:  
            left = mid + 1  
        else:  
            right = mid - 1  
    return -1
```

Linear search -

- Time complexity - $O(n)$
- Space complexity - $O(1)$

Binary search -

(Recursive).

- Time complexity - $O(\log n)$
- Space complexity - $O(\log n)$

Binary search -

(Iterative).

- Time complexity - $O(\log n)$
- Space complexity - $O(1)$

Ques Write recurrence relation for binary recursive search.
 Time complexity is $O(\log N)$ -

Recurrence relation $\rightarrow T(n) = T(n/2) + 1$

Derivation \rightarrow

$$\text{1st step} \Rightarrow T(n) = T(n/2) + 1$$

$$\text{2nd step} \Rightarrow T(n/2) = T(n/4) + 1 \dots [T(n/4) = T(n/2^2)]$$

$$\text{3rd step} \Rightarrow T(n/4) = T(n/8) + 1 \dots [T(n/8) = T(n/2^3)]$$

Ques 7. Find two indexes such that $A[r] + A[s] = k$ in minimum time complexity.

```
def find_indices_with_sum(arr, K):
    hash_set = set()
    for i, num in enumerate(arr):
        complement = K - num
        if complement in hash_set:
            return arr.index(complement), i
        hash_set.add(num)
    return "No such pair exists!"
```

arr = [1, 4, 3, 2, 10]

K = 12

```
print(find_indices_with_sum(arr, K))
```

Ques 8. Which sorting is best for practical use? Explain

→ The "best" sorting algorithm for practical use depends on various factors such as the size of the dataset, the distribution of the data, memory constraints, and the desired stability of the sorting.

1. Quick Sort: widely used for general purpose sorting, efficient for large datasets but not stable.
2. Merge Sort: offers stable sorting with guaranteed $O(n \log n)$ time complexity, suitable for large datasets & external sorting.
3. Tim Sort: hybrid of merge sort and.
3. Insertion Sort: efficient for small datasets & nearly sorted arrays, stable and adaptive.

Ques 9

what do you mean by no. of inversions in an array? Count the no. of inversions in Array arr[]

\Rightarrow In an array, an inversion occurs when there are two elements $arr[i]$ and $arr[j]$ such that $i < j$ but $arr[i] > arr[j]$. In simpler terms, an inversion represents a pair of elements that are out of order.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr, 0
```

$$\text{mid} = \text{len}(arr) // 2$$

left_half, left_count = merge_sort(arr[:mid])

right_half, right_count = merge_sort(arr[mid:])

sorted_arr, merge_count = merge(left_half,
right_half)

$$\text{total_count} = \text{left_count} + \text{right_count} + \text{merge_count}$$

return sorted_arr, total_count

```
def merge(left, right):
```

merged = []

count = 0

$i = j = 0$

while $i < \text{len}(left)$ and $j < \text{len}(right)$:

if $left[i] \leq right[j]$:

merged.append(left[i])

$i += 1$

else:

merged.append(right[i:j])

j += 1

count += len(left) - i

merged.extend(left[i:j])

merged.extend(right[j:j])

return merged, count

ans = [7, 21, 31, 8, 10, 1, 20, 6, 4, 5]

sorted_ans, inversions = merge_sort(ans)

print("Number of inversions:", inversions)

Ques 10. In which cases Quick sort will give the best & the most case time complexity?

⇒ (1) Best Case Time Complexity:

→ The best case time complexity of Quick sort is $O(n \log n)$

→ This occurs when the pivot elements divide the array into roughly equal partitions in each recursive call,

(2) Worst case Time Complexity:

→ The worst case time complexity of Quick sort is $O(n^2)$

→ This occurs when the pivot selection consistently results in unbalanced partitions such as when the pivot is the smallest or largest element in the array.

Q1. Write Recurrence Relation of Merge and Quick sort in best and worst case? what are the similarities and differences between complexities of two algorithms and why?

=)

→ Merge sort -

- Best case : $T(n) = \alpha T(n/2) + O(n)$
- Worst case : $T(n) = \alpha T(n/2) + O(n)$

→ Quick sort -

- Best case : $T(n) = \alpha T(n/2) + O(n)$
- Worst case : $T(n) = T(n-1) + O(n)$

→ Similarities -

1. Both merge and quick have a best-case time complexity of $O(n \log n)$ when the input data is well-distributed or random.
2. They both use a divide-and-conquer approach to sort the array.

→ Differences :

1. Worst-case Time Complexity
2. Stability
3. Space Complexity

Ques

selection sort is not stable by default
but can you write a version of
stable selection sort

⇒ def stable_selection_sort(arr):

n = len(arr)

for i in range(n-1):

 min_index = i

 for j in range(i+1, n):

 if arr[j] < arr[min_index]:

 min_index = j

 key = arr[min_index]

 while min_index > i:

 arr[min_index] = arr[min_index - 1]

 min_index -= 1

 arr[i] = key

return arr

arr = [7, 3, 5, 1, 7, 2, 5]

sorted_arr = stable_selection_sort(arr)

print(sorted_arr)