Name - Akshay Kumar Surapu
Sec - ML
Roll no. - 61
Date / /
Page

Assignment - 2

Ques 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".
```

Ques 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?

Ans 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 <= 1:
        return
    recursiveInsertionSort(arr, n-1)
    key = arr[n-1]
    j = n-2
    while j >= 0 and arr[j] > key:
        arr[j+1] = arr[j]
        j = j-1
    arr[j+1] = key
```

Ans3 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 - $O(n+K)$

Q. 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 maintains 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

Q.No.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 = len(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 → $T(n) = T(n/2) + 1$

Derivation →

1st step => $T(n) = T(n/2) + 1$

2nd step => $T(n/2) = T(n/4) + 1$ ... [$T(n/4) = T(n/2^2)$]

3rd step => $T(n/4) = T(n/8) + 1$ ... [$T(n/8) = T(n/2^3)$]

Ques 7. Find two indexes such that A[x] + A[y] = K with 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, 45, 7, 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 larg datasets & external sorting.
3. Tim Sort: hybrid of merge sort and
3. Insertion sort: efficient for small datasets & nearly sorted arrays, stable and adaptive.

Be Positive

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

⇒ 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 simple terms, an inversion represents a pair of elements that are out of order.

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

    mid = 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)

    total_count = left_count + right_count +
                  merge_count
    return sorted_arr, total_count

def merge(left, right):
    merged = []
    count = 0
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
```

```
else:
    merged.append(right[j])
    j += 1
    count += len(left) - i

merged.extend(left[i:])
merged.extend(right[j:])
return merged, count

arr = [7, 21, 31, 8, 10, 1, 20, 6, 4, 5]
sorted_arr, inversions = merge_sort(arr)
print("Number of inversions:", inversions)
```

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

=> (1.) Best Case Time Complexity:

→ The best case time complexity of Quick Sort is O(nlogn)

→ 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.

Q11. 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 : $T(n) = 2T(n/2) + O(n)$
- Worst case : $T(n) = 2T(n/2) + O(n)$

→ Quick sort -

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

→ Similarities -

1. Both merge and quick sort 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 0 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)
```