Manthan Kumar Rajput. Assignment - DAA.

04

CE

Q.1→ Write linear search Pseudocode to search an element in a sorted array with minimum comparison.

```
Function linear_search_sorted (arr, x):
    n = length(arr)
    i = 0
    while i<n and arr[i]<=x:
        if arr[i]==x;
            return i
        i = i+1
        return -1
```

Q2. Write a Pseudo code for iterative and recursive insertion sort. Sort is called online sorting. why? what about other sorting algorithms that has been discussed in lecture.

Iterative

```
Function insertion_sort(arr):
    n = length(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] = arr[j]
        j = j-1
        arr[j+1] = key
        return arr.
```

Recursive.

```
Function recursive_insertion_sort(arr, n)
    if(n<=1:
        return arr
    recursive_insertion_sort (arr, n-1)
        key = arr(n-1)
    j = n-1
    while j>=0 and arr[j]>key:
        arr[j+1] = arr[j]
        j = j-1
        arr[j+1] = key
    return
```

Other sorting algo. that have been discussed in lecture.

- Bubble sort : This algorithm repeatedly compares adjacent element and swaps them if they are in the wrong order until the entire array is sorted.
- selection sort : This algorithm repeatedly selects the minimum elements from the unsorted part of the array and swaps it with the first element of the unsorted part until the entire array is sorted.
- Merge sort: This algo divides the array into two halves recursively sort the two halves, and then merge the two sorted halves and then merge the two sorted halves into a singel sorted array.
- Quick sort: This sort algo picks an element as a pivot, the array around the pivot and then recursively sorts of two subarrays on either side of the pivot.

Q.3 Complexity of all sorting algo. that has been discussed in lecture.

1. Bubble sort
   - W-case time complexity = $O(n^2)$
   - B-case " " = $O(n)$
   - Avg. " " " = $O(n^2)$
   - Spac " " = $O(1)$

2. Selection sort:
   - W-case time complexity : $O(n^2)$
   - B= " " " $O(n^2)$
   - Avg " " " $O(n^2)$

   Spac complexity $O(1)$

3. Insertion sort:
   - W-case time complexity = $O(n^2)$
   - B-" " " = $O(n)$
   - Avg. " " " = $O(n^2)$
   - Spac complexity = $O(1)$

4. Merge sort:
   - W-case time complexity = $O(n\log n)$
   - B-case " " = $O(n\log n)$
   - Avg " " " = $O(n\log n)$
   - Space complexity = $O(n)$

5. Quick sort:
   - W-case time complexity : $O(n^2)$
   - B-" " " : $O(n\log n)$
   - Avg. " " " : $O(n\log n)$

   Spac complexity : $O(\log n)$

Manthan Kumar Rajput
04
CE

Q.4. Divide all the sorting algo into inplace / stable / online sorting.

| Inplace | Stable | Online |
|---|---|---|
| • Bubble sort | • Insertion sort | • Insertion sort |
| • Selection sort | • Merge sort | |
| • Insertion sort | | |
| • Quick sort | | |

Q.5. Write recursive / iterative pseudo code for binary search what is the time space complexity of linear and binary search (recursive and iterative).

Recursive binary search:

```
binary search (arr, left, right, x):
if right >= left :
    mid = left + (right + left) // 2
    if arr[mid] == x:
        return mid
    elif arr[mid] > x:
        return binary search (arr, left, mid - 1, x)
    else:
        return binary search (arr, mid + 1, right, x)
    else:
        return -1
```

Iterative Binary Search:

```
Function binary_search (arr, x):
  left = 0
  right = len(arr) - 1
  while left <= right :
    mid = left + (right - left) // 2
  if arr[mid] == x:
        return mid
  elif arr[mid] > x:
        right = mid - 1
  else:
     left = mid + 1
  return -1
```

Binary

| Linear Search | Recursive Binary | Iterative Binary |
|---|---|---|
| B-case time complexity - O(1) | B-case time comp - O(1) | B-case time - O(1) |
| W- " " " - O(n) | W- " " " - O(log n) | W- " " - O(log n) |
| Space complexity O(1) | Space complexity - O(log n) | Space complexity - O(1) |

Q.6 Manthan Kumar Rajput.
04
CE

Q.6. Write recurrence relation for binary recursive search.

The recurrence relation for binary search is.

$$T(n) = T(n/2) + C$$

T(n) represents the time complexity of searching for an element in an array of n element using binary recursive search.

n/2 represents the size of the subproblem obtained by dividing the input array into two halves.

C represent the constant.

The base case for this reccurence relation is when the size of the subarray become -1 i.e $T(1) = C$.

The recurrence relation can be solved using master theorem.

The master theorem gives time complexity of $O(\log n)$. where n is the number of element in the array.

2. Find two indexes such that $A[i] + A[j] = K$ in minimum time complexity.

algo

Step 1 : Initialize an empty hash table.

Step 2 : For each element A[i] in the array

a. calculate the diff. $K - A[i]$.

b. If the diff exists in the hash table, return the indices i and j such that $A[i] + A[j] = K$.

c. otherwise add the current element A[i] to hash table.

Step 3: If no such indices are found, return null or an appropriate message indicating that the sum K cannot be obtained from any two elements of the array

Q3. what do you mean by number of inversions in an array? Count the number of inversions in an array arr[] = [2, 21, 31, 8, 10, 1, 20, 6, 4, 5] using merge sort.

In an array of n distinct elements, an inversion is a pair of element (arr[i], arr[j] such that i<j and arr[i] > arr[j]. In other words it represents, how far away an array is from being sorted in ascending order.

Code:

```
merge sort(arr, left, right):
if left < right:
    mid = (left+right) //2
inversions= mergesort(arr, left, mid)
inversions = merge sort(arr, mid+1, right)
inversions = merge(arr, left, mid, right)
    return inversions
else:
  return 0:

Function merge(arr, left, mid, right
    inversion = 0
    L = arr[left: mid+1]
    R = arr[mid+1: right+1]
    i = j = 0
    K = left.
    while i<len(L) and j<len(R):
        if L[i] <= R[j]:
            arr[k] = L[i]
            i+=1
        else:
            arr[K] = R[j]
            j+=1
            inversions+ = (mid -i+1)
    K+=1
    while i<len(L):
    arr[K] = L[i]
        i+=1
        K+=1
    while j<len(R):
    arr[K] = R[j]
        j+=1
        K+=1
    return inversions
```

Manthan Kumar Rajput
CE
04

Q. 10- In which cases a quick sort will give the best and worst case time complexity?

Ans: Quicksort is a widely used and efficient comparison-based sorting algorithm that exhibits different time complexities depending on the input data characteristics.

Best case time complexity: The best case complexity of quicksort occurs when the pivot chosen during the partitioning step result in equally sized sub-array in each recursive call. This means that the pivot divides the input array into two nearly equal halves, resulting in a balanced partition. In this case, the time complexity of quicksort is $O(n \log n)$, where n is the number of elements in the input array. This occurs when the pivot selected is close to the median of median of the input array.

Worst case complexity: The worst case time complexity of quicksort occurs when the pivot chosen during the partitioning step result in highly imbalanced subarrays in each recursive call. This can happen when the input array is already sorted or nearly sorted, and the pivot chosen consistently result in one sub-array being significantly larger than the other. This worst case scenario can be mitigated by using randomized pivot selection techniques or choosing pivots strategically, such as using the "median-of-three" pivot selection strategy.

11. 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

Ans. Recurrence Relation for merge sort.

Best case :

T_merge(n) = 2T_merge(n/2) + c (where c is a constant representing the time taken for merging two sub-arrays of size n/2)

worst case :

T_merge(n) = 2T_merge(n/2) + c*n (where c is a constant representing the time taken for merging two sub-array of size n/2, and n represents the total number of elements in the initial array that need to be merged)

Recurrence Relation for Quick sort.

T_quick(n) = T_quick(n/2) + c (where c is a constant representing the time taken for Partitioning the initial array into two equal-sized sub-arrays)

worst case:

T_quick(n) = T_quick(n-1) + T_quick(0) + c*n (where c is a constant representing the time

Similarities.

1. Both merge sort and quick sort have an average time of O(nlogn) for sorting n element
2. Both algorithms use divide-and-conquer technique to sort an array
3. Both algorithms have a space complexity of O(n) as they require additional space for merging or partitioning the initial array

Difference

1. In the best case, merge sort has a time complexity of O(nlogn), whereas quick sort has a time complexity of O(nlogn) as well, but with a smaller constant factor due to fewer comparisons
2. In the worst case, merge sort has a time complexity of O(nlogn), whereas quick sort has a worst-case time complexity

Q.12 Selection sort is not stable by default but can you write a version of stable selection sort?

Ans The idea behind stable selection sort is to modify the selection sort algorithm such that it always selects the smallest element among the unsorted, but it swaps this smallest element with the left most occurrence of that element in the sorted part of the array. This ensures that the relative order of equal elements is preserved, making the algorithm stable.

```
Stable_selection_sort(A):
    n = length(A)
    for i from 0 to n-2:
        min_idx = i
        for j from (i+1) to n-1:
            if A[j] < A[min_idx]:
                min_idx = j
        for k from min_idx down to i+1:
            if A[k] == A[k-1]:
                A[k], A[k-1] = A[k-1], A[k]
            else:
                break
        A[i], A[min_idx] = A[min_idx], A[i].
```

~~Q.13 - B~~

Q.13- 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.

This optimization is known as the "flagged" or "adaptive" bubble sort algo.

```
Flagged_bubble_sort(A):
    n = length(A)
    sorted = False
    while not sorted:
    sorted = True
    for i from 0 to n-1
        if A[i] > A[i+1]:
        A[i], A[i+1] = A[i+1], A[i]
        sorted = False
        n -= 1
```