Jai Kumar
09
CE

Assignment - DAA

Q1. Write linear search pseudocode to search an element in a sorted array with minimum comparison.

Python →

```
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 Pseudo code for iterative and recursive insertion sort, Insertion sort is called online sorting. why? what about other sorting algorithms that has been discussed in lectures?

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] = 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-2
    while j >= 0 and arr[j] > key:
        arr[j+1] = arr[j]
        j = j-1
    arr [j+1] = key
    return arr
```

Jai Kumar 09 CE

Other sorting algo. that have been discussed in lectures.

- Bubble sort: This algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order until the entire array is sorted.
- Selection Sort: This algorithm repeatedly selects the minimum element 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 into a singel sorted array.
- Quick sort: This sort agd algo picks an element as a pivot, Partit the array around the pivot, and then recursively sorts the two subarrays on either side of the pivot.

Q3. Complexity of all the sorting algo. that has been discussed in lectures.

1. Bubble sort
   - W-case time complexity: $O(n^2)$
   - B-case time complexity: $O(n)$
   - Avg-case time complexity: $O(n^2)$
   - Space complexity: $O(1)$
2. Selection Sort:
   - W-case time complexity: $O(n^2)$
   - B-case time complexity: $O(n^2)$
   - Avg- time Complexity: $O(n^2)$
   - Space complexity: $O(1)$
3. Insertion sort:
   - W-case time complexity: $O(n^2)$
   - B-case time complexity: $O(n)$
   - Avg-case time ": $O(n^2)$
   - Space complexity: $O(1)$
4. Merge sort:
   - W-case time complexity: $O(n\log n)$,
   - B-case time complexity: $O(n\log n)$
   - Avg-case time ": $O(n\log n)$
   - Space complexity: $O(n)$
5. Quick sort:
   - W-case time complexity: $O(n^2)$
   - B-case time ": $O(n\log n)$
   - Avg-case time ": $O(n\log n)$
   - Space complexity: $O(\log n)$

Jai Kumar
09
CE

Q4. 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		

Q5. Write recursive/iterative pseudo code for binary search. What is the time and space complexity of Linear and Binary Search (Recursive and Iterative).

Recursive Binary Search:

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

Jai Kumar
09
CE

Iterative Binary Search:

```
Function binarySearch(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
```

Linear Search	Binary — Recursive Binary	Binary — Iterative Binary
B-case time compl - $O(1)$	B-case time compl - $O(1)$	B-case time - $O(1)$
W- " " " - $O(n)$	W- " " " - $O(\log n)$	W- " " - $O(\log n)$
space compl. - $O(1)$	Space compl. - $O(\log n)$	Space compl - $O(1)$

08. Which sorting is best for practical uses? Explain?

In general, there is no one-size fits-all answer to which sorting algo is best for partical uses. However some sorting algo are commonly used in practics because they have good avrage-case performance and are easy to impliment and under which includes.

Jai Kumar 09 CE

Q6. Write reccurrence relation for binary recursive search.

The recurrunce 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 elements using binary recursive search.

n/2 represent the size of the subproblem obtainted by dividing the input array into two halves

c represent the constant.

The base case for this reccurrence relation is when the size of the subarray becomes 1 i.e, $T(1) = C$.

The recuvrrunce relation can be solved using master Theorom. The master Theorom gives a time complexity of $O(\log n)$. where n is the number of element in the array.

Q7. 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.

09. What do you mean by number of inversions in an array? Count the number of inversions in an Array arr[] = { 7, 21, 31, 8, 10, 1, 20, 6, 4, 5 } using merge sort.

Jaikumar
09
CE

In an array of n distinct elements, an inversion is a pair of elements (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:

```
Function merge sort(arr, left, right):
    if left < right:
        mid = (left + right)/2
        inversions = mergesort(arr, left, mid)
        inversions += mergesort(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
```

Jai Kumar
09
CE

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

Quick sort has a worst-case time complexity of $O(n^2)$
The worst-case time complexity occurs when the pivot element chosen at each step divides the array into two subarrays of size 0 and n-1, respectively. This can happen when the input array is already sorted to when all the elements in the array are the same. In this case, the recursive function calls will have to process n-1 elements each time, leading to a worst-case time complexity of $O(n^2)$.
The best-case time occurs when the pivot element chosen at each step divides the array into two subarrays of roughly equal size. In this case, the recursive function calls will have to process two subarrays of size roughly n/2 each time, leading to a best-case time complexity of $O(n \log n)$.

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

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

Jai Kumar
09
CE

```
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]
```

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.

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-2
            if A[i] > A[i+1]:
                A[i], A[i+1] = A[i+1], A[i]
                sorted = False
        n -= 1
```