DATA STRUCTURE

Chapter 8

Searching & Sorting

Engineering Notes for B.Tech / Diploma Students

Topics Covered

- 8.1 Sorting – An Introduction
- 8.2 Efficiency of Sorting Algorithms
- 8.3 Bubble Sort
- 8.4 Selection Sort
- 8.5 Quick Sort
- 8.6 Insertion Sort
- 8.7 Merge Sort
- 8.8 Binary Tree Sort
- 8.9 Radix Sort
- 8.10 Shell Sort
- 8.11 Heap Sort
- 8.12 Searching – An Introduction, Binary Search
- Multiple Choice Questions (15) with Answer Key
- Broad / Long Answer Questions (10)

8.1 Sorting – An Introduction

Sorting is the process of arranging the elements of a list or array in a particular order, which may be ascending or descending, based on some key value associated with each element. It is one of the most fundamental operations in computer science because a large number of problems, such as searching, merging and preparing reports, become far simpler and faster once the underlying data has been sorted.

A sorting algorithm takes an unordered collection of data as input and produces an ordered collection as output. The elements may be numbers, strings, or records containing multiple fields, in which case sorting is usually performed on one particular field known as the key.

Need for Sorting

- It makes searching for a particular element much faster, since techniques such as Binary Search can only be applied on sorted data.
- It helps in organising data in a meaningful and readable sequence, such as arranging student records by roll number or marks.
- It is a preliminary step for many other algorithms, including merging of files, duplicate elimination and database query optimisation.
- It improves the efficiency of report generation, statistical analysis and set operations such as union and intersection.

Classification of Sorting Techniques

Sorting algorithms can be broadly classified on the basis of the following criteria:

- Internal Sorting – The entire data to be sorted is small enough to fit into the main memory (RAM) at one time. Examples include Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Heap Sort and Shell Sort.
- External Sorting – The data is too large to fit into main memory and must be stored on secondary storage such as a disk, with sorting performed in stages. Example: External Merge Sort.
- Comparison-based Sorting – The algorithm sorts elements by comparing pairs of elements, e.g. Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort.
- Non-comparison-based Sorting – The algorithm sorts elements without directly comparing them, typically using the digits or characteristics of the key, e.g. Radix Sort, Counting Sort, Bucket Sort.
- Stable Sorting – A sorting method is said to be stable if it preserves the relative order of records with equal keys, e.g. Bubble Sort, Insertion Sort, Merge Sort.
- In-place Sorting – A sorting method that requires only a constant amount of extra memory space in addition to the input array, e.g. Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Heap Sort.

8.2 Efficiency of Sorting Algorithms

The efficiency of a sorting algorithm is generally measured in terms of the amount of time it takes to sort a given number of elements and the amount of extra memory space it requires. Since the actual running time depends

on the machine and compiler used, efficiency is expressed using asymptotic notations, primarily Big-O notation, which describes how the running time grows as the size of the input, n, increases.

Time Complexity

Time complexity refers to the number of comparisons and swaps (or moves) an algorithm performs as a function of the input size n. It is generally analysed under three cases:

- Best Case – The minimum time required when the input is already in a favourable arrangement, e.g. already sorted.
- Average Case – The expected time required over all possible arrangements of the input.
- Worst Case – The maximum time required, which typically occurs when the input is in reverse sorted order.

Space Complexity

Space complexity refers to the amount of additional memory, other than the input array itself, that an algorithm needs to complete its execution. Algorithms such as Bubble Sort and Insertion Sort require only O(1) extra space, whereas Merge Sort requires O(n) extra space because it uses auxiliary arrays during the merging process.

Comparison of Standard Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Shell Sort	O(n log n)	$O(n^{1.3})$	$O(n^2)$	O(1)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)

(Here, n denotes the number of elements to be sorted and k denotes the number of digits or the range of the key in Radix Sort.)

Factors Affecting the Choice of a Sorting Algorithm

- Size of the data set to be sorted.
- Whether the data is already partially sorted or completely random.
- The amount of memory space available for auxiliary storage.
- Whether stability of equal elements needs to be preserved.
- Whether the data resides in main memory or on external secondary storage.

8.3 Bubble Sort

Bubble Sort is one of the simplest sorting techniques. It works by repeatedly stepping through the list, comparing each pair of adjacent elements and swapping them if they are in the wrong order. This process is repeated until no more swaps are required, which indicates that the list is sorted. Because smaller (or larger) elements slowly "bubble" towards the top of the list with each pass, the technique is named Bubble Sort.

Algorithm

1. Start from the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next pair of elements and repeat steps 2 and 3 until the end of the array is reached; this completes one pass.
5. Repeat the entire process for (n – 1) passes, or until a complete pass occurs with no swaps, which means the array is sorted.

Pseudocode

```
for i = 0 to n-2
    for j = 0 to n-2-i
        if arr[j] > arr[j+1] then
            swap(arr[j], arr[j+1])
        end if
    end for
end for
```

Characteristics

- Time Complexity: Best Case – O(n) (when an optimised version detects no swaps); Average and Worst Case – $O(n^2)$.
- Space Complexity: O(1), since sorting is done in place.
- Stable and easy to implement, but inefficient for large data sets due to the large number of comparisons and swaps.

8.4 Selection Sort

Selection Sort works by dividing the array into a sorted part and an unsorted part. In each pass, it selects the smallest (or largest) element from the unsorted part and places it at the end of the sorted part by swapping it into its correct position. This process continues until the entire array is sorted.

Algorithm

6. Set the first element of the unsorted part as the minimum.
7. Compare this minimum with each of the remaining elements in the unsorted part.
8. Whenever a smaller element is found, mark it as the new minimum.
9. After scanning the entire unsorted part, swap the minimum element with the first element of the unsorted part.

10. Move the boundary between the sorted and unsorted parts one position to the right and repeat the process until the whole array is sorted.

Pseudocode

```
for i = 0 to n-2
    min_index = i
    for j = i+1 to n-1
        if arr[j] < arr[min_index] then
            min_index = j
    end for
    swap(arr[i], arr[min_index])
end for
```

Characteristics

- Time Complexity: $O(n^2)$ in the best, average and worst cases, since the algorithm always scans the remaining unsorted elements regardless of their arrangement.
- Space Complexity: O(1).
- Not stable in its basic form, but performs fewer swaps than Bubble Sort, which makes it useful when the cost of swapping is high.

8.5 Quick Sort

Quick Sort is a highly efficient, divide-and-conquer sorting algorithm. It works by selecting an element from the array, called the pivot, and partitioning the remaining elements into two sub-arrays: those smaller than the pivot and those greater than the pivot. The pivot is then placed in its correct sorted position, and the same process is applied recursively to the two sub-arrays.

Algorithm

11. Choose a pivot element from the array (commonly the first, last, or middle element).
12. Partition the array so that all elements smaller than the pivot are placed to its left and all elements greater are placed to its right.
13. The pivot is now in its correct final position within the sorted array.
14. Recursively apply steps 1 to 3 to the sub-array of elements to the left of the pivot.
15. Recursively apply steps 1 to 3 to the sub-array of elements to the right of the pivot.
16. The recursion terminates when a sub-array has zero or one element, since such an array is already sorted.

Pseudocode

```
function quickSort(arr, low, high)
    if low < high then
        pi = partition(arr, low, high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
    end if

function partition(arr, low, high)
    pivot = arr[high]
```

```
    i = low - 1
    for j = low to high-1
        if arr[j] <= pivot then
            i = i + 1
            swap(arr[i], arr[j])
    swap(arr[i+1], arr[high])
    return i + 1
```

Characteristics

- Time Complexity: Best and Average Case – O(n log n); Worst Case – $O(n^2)$, which occurs when the pivot chosen is always the smallest or largest element, as with an already sorted array.
- Space Complexity: O(log n) on average, due to the recursive call stack.
- Not stable in its standard implementation, but is one of the fastest general-purpose sorting algorithms in practice due to good cache performance.

8.6 Insertion Sort

Insertion Sort builds the final sorted array one element at a time. It works in a manner similar to the way a person sorts playing cards in their hands – each new card (element) is picked up and inserted into its correct position among the cards already sorted.

Algorithm

17. Consider the first element of the array to be a trivially sorted sub-array of size one.
18. Pick the next element from the unsorted part; call it the key.
19. Compare the key with the elements in the sorted sub-array, moving from right to left.
20. Shift each compared element that is greater than the key one position to the right.
21. Insert the key into the position left vacant after shifting.
22. Repeat steps 2 to 5 for all remaining elements until the entire array is sorted.

Pseudocode

```
for i = 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
    end while
    arr[j+1] = key
end for
```

Characteristics

- Time Complexity: Best Case – O(n), when the array is already sorted; Average and Worst Case – $O(n^2)$.
- Space Complexity: O(1).
- Stable and efficient for small data sets or data that is already nearly sorted; also useful as the final step in more advanced hybrid sorting algorithms.

8.7 Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to produce the final sorted array. Unlike Quick Sort, the actual sorting work is performed during the merge step rather than during partitioning.

Algorithm

23. If the array has more than one element, divide it into two roughly equal halves.
24. Recursively apply Merge Sort to the left half.
25. Recursively apply Merge Sort to the right half.
26. Merge the two sorted halves into a single sorted array by repeatedly comparing the smallest unmerged elements of each half and copying the smaller one into the result.
27. Continue merging until both halves are completely merged into one fully sorted array.

Pseudocode

```
function mergeSort(arr, l, r)
    if l < r then
        m = (l + r) / 2
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
    end if

function merge(arr, l, m, r)
    create temp arrays L[] and R[]
    copy data into L[] and R[]
    merge L[] and R[] back into arr[l..r]
    in sorted order
```

Characteristics

- Time Complexity: O(n log n) in the best, average and worst cases, making its performance highly predictable.
- Space Complexity: O(n), since it requires auxiliary arrays to hold elements during merging.
- Stable, and particularly well suited to sorting linked lists and large data sets stored on external storage (External Merge Sort).

8.8 Binary Tree Sort

Binary Tree Sort (also called Tree Sort) is a sorting technique that makes use of a Binary Search Tree (BST). The elements of the array are inserted one by one into a BST, and then an in-order traversal of the tree is performed, which visits the nodes in ascending sorted order because of the defining property of a BST – the left subtree of any node contains only smaller values and the right subtree contains only larger values.

Algorithm

28. Create an empty Binary Search Tree.

29. Insert each element of the array into the BST one at a time, following the standard BST insertion rule (smaller values go to the left, larger values go to the right of a node).
30. Once all elements have been inserted, perform an in-order traversal of the tree (Left – Root – Right).
31. Store the elements visited during the in-order traversal back into the array; this gives the elements in ascending sorted order.

Characteristics

- Time Complexity: Best and Average Case – O(n log n), when the tree remains reasonably balanced; Worst Case – $O(n^2)$, when the tree becomes skewed, such as when the input array is already sorted.
- Space Complexity: O(n), since additional memory is required to build the tree structure.
- It can be made stable by storing duplicate values as a list or counter within the same node, and its worst case can be avoided by using a self-balancing BST such as an AVL tree or a Red-Black tree.

8.9 Radix Sort

Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits. Instead of comparing whole elements against one another, it distributes elements into buckets according to each digit, starting from the least significant digit (LSD) to the most significant digit (MSD), or vice versa.

Algorithm (LSD Radix Sort)

32. Find the maximum number in the array to determine the number of digits, d.
33. Starting from the least significant digit (units place), distribute all elements into 10 buckets (numbered 0 to 9) according to the value of the current digit.
34. Collect the elements from the buckets in order from bucket 0 to bucket 9; this forms a new, partially ordered arrangement of the array.
35. Repeat steps 2 and 3 for the tens place, hundreds place, and so on, until all d digits of the maximum number have been processed.
36. After processing the most significant digit, the array is completely sorted.

Characteristics

- Time Complexity: O(nk) in the best, average and worst cases, where n is the number of elements and k is the number of digits in the largest number.
- Space Complexity: O(n + k), since additional space is required for the buckets used during distribution.
- Stable, and highly efficient for sorting large sets of integers or fixed-length strings, but not suitable when the key range or digit count is very large.

8.10 Shell Sort

Shell Sort is an improvement over Insertion Sort. Instead of comparing only adjacent elements, it first compares elements that are far apart from each other, defined by a value called the gap, and gradually reduces the gap in

successive passes until it becomes 1, at which point the algorithm performs a final pass equivalent to a standard Insertion Sort. This allows elements to move long distances quickly in the early passes, reducing the total amount of shifting required later.

Algorithm

37. Choose an initial gap value, typically n / 2, where n is the number of elements.
38. Compare elements that are 'gap' positions apart and swap them if they are out of order, effectively performing an Insertion Sort on each sub-list formed by the gap.
39. Reduce the gap (commonly by dividing it by 2) and repeat step 2 for the new gap value.
40. Continue reducing the gap and repeating the comparisons until the gap becomes 1.
41. Perform a final pass with a gap of 1, which is a standard Insertion Sort, to complete the sorting process.

Pseudocode

```
gap = n / 2
while gap > 0
    for i = gap to n-1
        temp = arr[i]
        j = i
        while j >= gap and arr[j-gap] > temp
            arr[j] = arr[j-gap]
            j = j - gap
        arr[j] = temp
    gap = gap / 2
```

Characteristics

- Time Complexity: depends on the gap sequence chosen; commonly Average Case – about $O(n^{1.3})$, Worst Case – $O(n^2)$.
- Space Complexity: O(1), since sorting is performed in place.
- Not stable, but significantly faster than simple Insertion Sort or Bubble Sort for medium-sized arrays, and does not require any extra memory.

8.11 Heap Sort

Heap Sort is a comparison-based sorting technique that makes use of a Binary Heap data structure, most commonly a Max Heap. It first converts the input array into a Max Heap, in which the value of each parent node is greater than or equal to the values of its children, so that the largest element is always at the root. It then repeatedly removes the root (the maximum element) and places it at the end of the array, reducing the heap size by one each time.

Algorithm

42. Build a Max Heap from the given array of n elements, using the heapify procedure on all non-leaf nodes starting from the last internal node up to the root.
43. Swap the root of the heap (the maximum element) with the last element of the heap.

44. Reduce the size of the heap by one, effectively removing the last element (the maximum) from consideration.
45. Apply the heapify procedure on the new root to restore the max-heap property.
46. Repeat steps 2 to 4 until the heap size is reduced to one, at which point the array is fully sorted in ascending order.

Pseudocode

```
function heapSort(arr, n)
    for i = n/2 - 1 down to 0
        heapify(arr, n, i)
    for i = n-1 down to 1
        swap(arr[0], arr[i])
        heapify(arr, i, 0)

function heapify(arr, n, i)
    largest = i, left = 2*i+1, right = 2*i+2
    if left < n and arr[left] > arr[largest]
        largest = left
    if right < n and arr[right] > arr[largest]
        largest = right
    if largest != i
        swap(arr[i], arr[largest])
        heapify(arr, n, largest)
```

Characteristics

- Time Complexity: O(n log n) in the best, average and worst cases, since building the heap takes O(n) and each of the n extractions takes O(log n).
- Space Complexity: O(1), as it sorts the array in place without needing auxiliary storage.
- Not stable, but its guaranteed O(n log n) worst-case performance and constant extra space make it valuable for systems with limited memory.

8.12 Searching – An Introduction, Binary Search

Searching – An Introduction

Searching is the process of finding the location of a particular element, called the search key, within a given collection of data. A search operation is said to be successful if the element is found and unsuccessful if it is not present in the collection. Searching is one of the most frequently performed operations in computer applications, ranging from looking up a record in a database to finding a word in a document.

Types of Searching Techniques

- Linear Search (Sequential Search) – Each element of the list is examined one after another, starting from the first, until the required element is found or the list is exhausted. It works on both sorted and unsorted data and has a time complexity of O(n).
- Binary Search – Works only on sorted data and repeatedly divides the search interval in half, giving a much faster time complexity of O(log n).

- Other advanced techniques include Interpolation Search, Jump Search, Exponential Search and Hashing-based search, which are used in specific situations to further improve performance.

Binary Search

Binary Search is an efficient algorithm for finding a target value within a sorted array. It works on the principle of divide and conquer: it repeatedly compares the target value to the middle element of the current search interval. If they are equal, the search is successful. If the target is smaller than the middle element, the search continues in the left half; if it is larger, the search continues in the right half. This process is repeated, halving the size of the search interval at each step, until the element is found or the interval becomes empty.

Algorithm

47. Set two pointers, low and high, to the first and last index of the sorted array respectively.
48. Repeat the following steps as long as low is less than or equal to high.
49. Calculate the middle index: mid = (low + high) / 2.
50. If arr[mid] is equal to the search key, the search is successful; return mid as the position of the element.
51. If the search key is less than arr[mid], discard the right half by setting high = mid – 1.
52. If the search key is greater than arr[mid], discard the left half by setting low = mid + 1.
53. If low exceeds high before the element is found, the search is unsuccessful and the element is not present in the array.

Pseudocode

```
function binarySearch(arr, key, low, high)
    while low <= high
        mid = (low + high) / 2
        if arr[mid] == key then
            return mid
        else if arr[mid] < key then
            low = mid + 1
        else
            high = mid - 1
    return -1   // element not found
```

Characteristics

- Time Complexity: Best Case – O(1), when the middle element itself is the target; Average and Worst Case – O(log n).
- Space Complexity: O(1) for the iterative version and O(log n) for the recursive version, due to the function call stack.
- Pre-condition: the array must already be sorted before applying Binary Search; it is far more efficient than Linear Search for large sorted data sets.

Multiple Choice Questions (MCQs)

Answer key is provided at the end of this section.

1. **Which of the following sorting algorithms has the best worst-case time complexity?**
   (a) Bubble Sort
   (b) Selection Sort
   (c) Merge Sort
   (d) Insertion Sort

2. **The technique of 'bubbling up' the largest element to the end of the array in each pass is used in:**
   (a) Selection Sort
   (b) Bubble Sort
   (c) Quick Sort
   (d) Radix Sort

3. **Selection Sort has a time complexity of _____ in all cases (best, average and worst).**
   (a) O(n)
   (b) O(n log n)
   (c) $O(n^2)$
   (d) O(log n)

4. **Quick Sort follows which algorithm design technique?**
   (a) Greedy method
   (b) Dynamic programming
   (c) Divide and conquer
   (d) Backtracking

5. **In Quick Sort, the worst-case time complexity of $O(n^2)$ occurs when:**
   (a) The array is randomly arranged
   (b) The pivot always divides the array into two equal halves
   (c) The pivot is always the smallest or largest element
   (d) All elements of the array are distinct

6. **Insertion Sort is most efficient when the input array is:**
   (a) Randomly arranged
   (b) In reverse sorted order
   (c) Already sorted or nearly sorted
   (d) Very large in size

7. **Which sorting algorithm requires additional O(n) auxiliary space to merge sub-arrays?**
   (a) Heap Sort
   (b) Quick Sort
   (c) Merge Sort
   (d) Shell Sort

8. **Binary Tree Sort produces sorted output by performing which traversal on the BST?**
   (a) Pre-order traversal

(b) Post-order traversal
(c) In-order traversal
(d) Level-order traversal

9. Radix Sort is best described as a:

(a) Comparison-based sorting technique
(b) Non-comparison-based sorting technique
(c) Tree-based sorting technique
(d) Recursive divide and conquer technique

10. In Radix Sort, digits are commonly processed starting from the:

(a) Most significant digit (MSD) only
(b) Middle digit
(c) Least significant digit (LSD)
(d) Digit chosen at random

11. Shell Sort improves upon which basic sorting algorithm?

(a) Bubble Sort
(b) Selection Sort
(c) Insertion Sort
(d) Merge Sort

12. In Shell Sort, the value used to determine how far apart the compared elements are is called the:

(a) Pivot
(b) Gap
(c) Key
(d) Index

13. Heap Sort makes use of which data structure to sort elements?

(a) Binary Search Tree
(b) Linked List
(c) Binary Heap
(d) Stack

14. The time complexity of Heap Sort in the worst case is:

(a) O(n)
(b) $O(n^2)$
(c) O(n log n)
(d) O(log n)

15. Binary Search can be applied only when the given array is:

(a) Unsorted
(b) Sorted
(c) Circular
(d) Doubly linked

Answer Key

1-c 2-b 3-c 4-c 5-c 6-c 7-c 8-c 9-b 10-c 11-c 12-b 13-c 14-c 15-b

Broad / Long Answer Questions

Q1. What is sorting? Explain the need for sorting and classify sorting algorithms as internal versus external and stable versus unstable, giving suitable examples of each.

Q2. Explain the criteria used to judge the efficiency of a sorting algorithm. Compare the time and space complexities of Bubble Sort, Selection Sort, Quick Sort, Merge Sort and Heap Sort with the help of a table.

Q3. What is Bubble Sort? Write its algorithm and trace it step by step on the array [29, 10, 14, 37, 13], showing the array after every pass.

Q4. Explain Selection Sort with its algorithm. Sort the array [64, 25, 12, 22, 11] using Selection Sort, showing the array after each pass, and state its time complexity.

Q5. Describe the working of Quick Sort using the divide-and-conquer approach. Explain the partitioning process with a suitable example and discuss its best, average and worst-case time complexities.

Q6. What is Insertion Sort? Explain its algorithm with a suitable example and discuss why it is efficient for small or nearly sorted data sets.

Q7. Explain Merge Sort with its algorithm. Sort the array [38, 27, 43, 3, 9, 82, 10] using Merge Sort, showing the divide and merge steps, and derive its time complexity.

Q8. What is Binary Tree Sort? Explain how a Binary Search Tree can be used to sort a list of elements, and discuss its best-case and worst-case time complexities with reasons.

Q9. Explain Radix Sort and Shell Sort with their algorithms and suitable examples. Compare them on the basis of time complexity, space complexity and the type of data on which each is best suited.

Q10. Explain Heap Sort with its algorithm. Discuss the concept of a Max Heap, describe the heapify process, and explain how Binary Search works on a sorted array with a suitable example.