DS-Searching-Unit-5

23 April 2025 18:46

What is Searching?

In computer science, searching is the process of finding an item with specified properties from a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or they may be elements of other search spaces.

Searching is one of the core computer science algorithms. We know that today's computers store a lot of information. To retrieve this information proficiently we need very efficient searching algorithms.

There are certain ways of organizing the data that improves the searching process. That means, if we keep the data in proper order, it is easy to search the required element.

Sorting is one of the techniques for making the elements ordered.

Types of Searching:

Following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Interpolation search
- Binary Search Trees (operates on trees and refer Trees chapter)
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

Unordered Linear Search:

Let us assume we are given an array where the order of the elements is not known. That means the elements of the array are not sorted. In this case, to search for an element we have to scan the complete array and see if the element is there in the given list or not.

```
int UnOrderedLinearSearch (int A[], int n, int data) {
    for (int i  = 0; i < n; i++) {
        if(A[i] == data)
            return i;
    }
    return -1;
}
```

Time complexity: O(n),
in the worst case we need to scan the complete array.
Space complexity: O(1)

1. Sorted/Ordered Linear Search:

   If the elements of the array are already sorted, then in many cases we don't have to scan the complete array to see if the element is there in the given array or not.

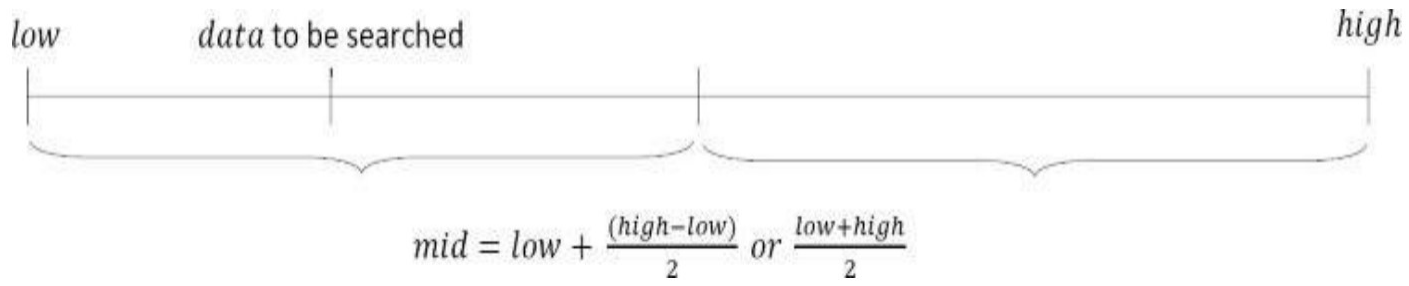
   In the algorithm below, it can be seen that, at any point if the value at A[i] is greater than the data to be searched, then we just return –1 without searching the remaining array

```
int OrderedLinearSearch(int A[], int n, int data) {
    for (int i  = 0; i < n; i++) {
            if(A[i] == data)
                    return i;
            else if(A[i] > data)
                    return -1;
    }
    return -1;
}
```

Time complexity of this algorithm is O(n).
This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is the same.
Space complexity: O(1).

Binary Search:

Let us consider the problem of searching a word in a dictionary.
Typically, we directly go to some approximate page [say, middle page] and start searching from that point. If the name that we are searching is the same then the search is complete.
If the page is before the selected pages then apply the same process for the first half; otherwise apply the same process to the second half.
Binary search also works in the same way. The algorithm applying such a strategy is referred to as binary search algorithm



$$mid = low + \frac{(high-low)}{2} \text{ or } \frac{low+high}{2}$$

Screen clipping taken: 23-04-2025 19:12

```
//Iterative Binary Search Algorithm
int BinarySearchIterative(int A[], int n, int data) {
    int low = 0;
    int high = n-1;
    while (low <= high) {
        mid = low + (high-low)/2; //To avoid overflow
        if(A[mid] == data)
            return mid;
        else if(A[mid] < data)
            low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Screen clipping taken: 23-04-2025 19:15

```
//Recursive Binary Search Algorithm
int BinarySearchRecursive(int A[], int low, int high, int data) {
    int mid = low + (high-low)/2; //To avoid overflow
    if (low>high)
      return -1;
    if(A[mid] == data)
        return mid;
    else if(A[mid] < data)
        return BinarySearchRecursive (A, mid + 1, high, data);
    else   return BinarySearchRecursive (A, low, mid - 1 , data);
    return -1;
}
```

Screen clipping taken: 23-04-2025 19:15

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using *Divide and Conquer* master theorem, we get, $T(n) = O(logn)$.

Time Complexity: O(*logn*). Space Complexity: O(1) [for iterative algorithm].

Screen clipping taken: 23-04-2025 19:16

11.8 Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Average Case
Unordered Array	n	$n/2$
Ordered Array (Binary Search)	$logn$	$logn$
Unordered List	n	$n/2$
Ordered List	n	$n/2$
Binary Search Trees (for skew trees)	n	$logn$
Interpolation search	n	$log(logn)$

Screen clipping taken: 23-04-2025 19:17

What is Sorting?

Sorting is an algorithm that arranges the elements of a list in a certain order [either ascending or descending]. The output is a permutation or reordering of the input.

Sorting can significantly reduce the complexity of a problem, and is often used for database algorithms and searches.

Classification of Sorting Algorithms

Sorting algorithms are generally categorized based on the following parameters.

1. By Number of Comparisons
2. By Number of Swaps
3. By Memory Usage
4. By Recursion
5. By Stability
6. By Adaptability

Other Classifications
Another method of classifying sorting algorithms is:
• Internal Sort
• External Sort
Internal Sort:

Sort algorithms that use main memory exclusively during the sort are called internal sorting algorithms. This kind of algorithm assumes high-speed random access to all memory.

External Sort:

Sorting algorithms that use external memory, such as tape or disk, during the sort come under this category

Bubble Sort:

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to the last, comparing each pair of elements and swapping them if needed
Bubble sort continues its iterations until no more swaps are needed.
The algorithm gets its name from the way smaller elements “bubble” to the top of the list
The only significant advantage that bubble sort has over other implementations is that it can detect whether the input list is already sorted or not.

Implementation:

```
void BubbleSort(int A[], int n) {
    for (int pass = n - 1; pass >= 0; pass--){
        for (int i = 0; i <= pass - 1 ; i++)      {
            if(A[i] > A[i+1]) {
                // swap elements
                int temp = A[i];
                A[i] = A[i+1];
                A[i+1] = temp;
            }
        }
    }
}
```

Algorithm takes O(n 2) (even in best case). We can improve it by using one extra flag. No more swaps indicate the completion of sorting. If the list is already sorted, we can use this flag to skip the remaining passes.

```
void BubbleSortImproved(int A[], int n) {
    int pass, i, temp, swapped = 1;
    for (pass = n - 1; pass >= 0 && swapped; pass--) {
        swapped = 0;
        for (i = 0; i <= pass - 1 ; i++) {
            if(A[i] > A[i+1]) {
                // swap elements
                temp = A[i];
                A[i] = A[i+1];
                A[i+1] =  temp;
                swapped = 1;
            }
        }
    }
}
```

Screen clipping taken: 24-04-2025 12:19

This modified version improves the best case of bubble sort to O(n).

Performance:
---

1. Worst case complexity : O(n 2)
2. Best case complexity (Improved version) : O(n)
3. Average case complexity (Basic version) : O(n 2)
4. Worst case space complexity : O(1) auxiliary

II. Selection Sort:
==========

Selection Sort

Selection sort is an in-place sorting algorithm. Selection sort works well for small files. It is used for sorting the files with very large values and small keys. This is because selection is made based on keys and swaps are made only when required.

Advantages
• Easy to implement
• In-place sort (requires no additional storage space)

Disadvantages
• Doesn't scale well: O(n2)

Algorithm
1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the elements until the entire array is sorted
This algorithm is called selection sort since it repeatedly selects the smallest element.

Implementation:

```
void Selection(int A [], int n) {
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++) {
        min = i;
        for (j = i+1; j < n; j++) {
            if(A [j] < A [min])
                min = j;
        }
        // swap elements
        temp = A[min];
        A[min] = A[i];
        A[i] = temp;
    }
}
```

Screen clipping taken: 24-04-2025 12:24

Performance

Worst case complexity : $O(n^2)$
Best case complexity : $O(n^2)$
Average case complexity : $O(n^2)$
Worst case space complexity: O(1) auxiliary

Screen clipping taken: 24-04-2025 12:24

III. Insertion Sort:
==========
Insertion sort is a simple and efficient comparison sort. In this algorithm, each iteration removes an element from the input data and inserts it into the correct position in the list being sorted.

The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through.

Advantages
• Simple implementation
• Efficient for small data
• Adaptive: If the input list is pre sorted [may not be completely] then insertions sort takes O(n + d), where d is the number of inversions
• Practically more efficient than selection and bubble sorts, even though all of them have O(n 2) worst case complexity
• Stable: Maintains relative order of input data if the keys are same
• In-place: It requires only a constant amount O(1) of additional memory space
• Online: Insertion sort can sort the list as it receives it

Algorithm
Every repetition of insertion sort removes an element from the input data, and inserts it into the correct position in the already-sorted list until no input elements remain. Sorting is typically done in-place.

Implementation

```
void InsertionSort(int A[], int n) {
    int i, j, v;
    for (i = 1; i <= n - 1; i++) {
        v = A[i];
        j = i;
        while (A[j-1] > v && j >= 1) {
            A[j] = A[j-1];
            j--;
        }
        A[j] = v;
    }
}
```

Screen clipping taken: 26-04-2025 22:26

Example

Given an array: 6 8 1 4 5 3 7 2 and the goal is to put them in ascending order.

6 8 1 4 5 3 7 2 (Consider index 0)
6 8 1 4 5 3 7 2 (Consider indices 0 - 1)
1 6 8 4 5 3 7 2 (Consider indices 0 - 2: insertion places 1 in front of 6 and 8)
1 4 6 8 5 3 7 2 (Process same as above is repeated until array is sorted)
1 4 5 6 8 3 7 2
1 3 4 5 6 7 8 2
1 2 3 4 5 6 7 8 (The array is sorted!)

Screen clipping taken: 26-04-2025 22:26

Performance
If every element is greater than or equal to every element to its left, the running time of insertion sort is Θ(n). This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

Worst case complexity: $\Theta(n^2)$
Best case complexity: $\Theta(n)$
Average case complexity: $\Theta(n^2)$

Worst case space complexity: $O(n^2)$ total, O(1) auxiliary

Screen clipping taken: 26-04-2025 22:28

Comparisons to Other Sorting Algorithms
Insertion sort is one of the elementary sorting algorithms with O(n2) worst-case time. Insertion sort is used when the data is nearly sorted (due to its adaptiveness) or when the input size is small (due to its low overhead). For these reasons and due to its stability, insertion sort is used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Notes:

- Bubble sort takes $\frac{n^2}{2}$ comparisons and $\frac{n^2}{2}$ swaps (inversions) in both average case and in worst case.
- Selection sort takes $\frac{n^2}{2}$ comparisons and n swaps.
- Insertion sort takes $\frac{n^2}{4}$ comparisons and $\frac{n^2}{8}$ swaps in average case and in the worst case they are double.
- Insertion sort is almost linear for partially sorted input.
- Selection sort is best suits for elements with bigger values and small keys.

10.9 Merge Sort

Merge sort is an example of the divide and conquer strategy.

Important Notes

- *Merging* is the process of combining two sorted files to make one bigger sorted file.
- *Selection* is the process of dividing a file into two parts: k smallest elements and $n - k$ largest elements.
- Selection and merging are opposite operations
    - selection splits a list into two lists
    - merging joins two files to make one file
- Merge sort is Quick sort's complement
- Merge sort accesses the data in a sequential manner
- This algorithm is used for sorting a linked list

Screen clipping taken: 27-04-2025 05:13

Performance

Worst case complexity : $\Theta(nlogn)$
Best case complexity : $\Theta(nlogn)$
Average case complexity : $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ auxiliary

Screen clipping taken: 27-04-2025 05:15

Heap Sort:

Heap sort is a comparison-based sorting algorithm and is part of the selection sort family.
Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case Θ(nlogn) runtime.
Heapsort is an in place algorithm but is not a stable sort.

Performance
Worst case performance: Θ(nlogn)
Best case performance: Θ(nlogn)
Average case performance: Θ(nlogn)
Worst case space complexity: Θ(n) total, Θ(1) auxiliary

Quicksort:

Quick sort is an example of a divide-and-conquer algorithmic technique. It is also called partition exchange sort. It uses recursive calls for sorting the elements, and it is one of the famous algorithms among comparison-based sorting algorithms.

Divide: The array A[low ...high] is partitioned into two non-empty sub arrays A[low ...q] and A[q+ 1... high], such that each element of A[low ... high] is less than or equal to each element of A[q + 1... high]. The index q is computed as part of this partitioning procedure.

Conquer: The two sub arrays A[low ...q] and A[q + 1 ...high] are sorted by recursive calls to Quick sort.

Algorithm
The recursive algorithm consists of four steps:
1) If there are one or no elements in the array to be sorted, return.
2) Pick an element in the array to serve as the “pivot” point. (Usually the left-most element in the array is used.)
3) Split the array into two parts – one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array

Performance
Worst case Complexity: O(n2)
Best case Complexity: O(nlogn)
Average case Complexity: O(nlogn)
Worst case space Complexity: O(1)

Comparison of Sorting Algorithms :
==========

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n + d)$, where d is the number of inversions.
Shell	-	$O(nlog^2n)$	1	no	
Merge sort	$O(nlogn)$	$O(nlogn)$	depends	yes	
Heap sort	$O(nlogn)$	$O(nlogn)$	1	no	
Quick sort	$O(nlogn)$	$O(n^2)$	$O(logn)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.
Tree sort	$O(nlogn)$	$O(n^2)$	$O(n)$	depends	Can be implemented as a stable sort.

Note: *n* denotes the number of elements in the input.

Screen clipping taken: 27-04-2025 05:29