TOPIC - SEARCHING AND SORTING

SEARCHING

Introduction:

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These types of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Searching Algorithms:

1. Linear Search

A simple approach is to do a linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of the elements, return -1.

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

Time Complexity : O(n)

It is possible to improve the worst case complexity of linear search by two methods,

1. if element Found at last O(n) to O(1)
2. if element Not found O(n) to O(n/2)

The implementation of above two methods can be found here.

Code for Linear Search:

```
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

1. Binary Search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

One important aspect to note in binary search is that the array must be sorted. Binary search doesn't work on unsorted arrays. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Stepwise algorithm for Binary Search:

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.

3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Time Complexity : O(logn)

The time complexity of Binary Search can be written as

T(n) = T(n/2) + c

The above recurrence can be solved either using the Recurrence Tree method or Master's method. It falls in case 2 of the Master's Method and the solution of the recurrence is Θ(logn).

Auxiliary Space: O(1) in case of iterative implementation. In the case of recursive implementation, O(Logn) recursion call stack space.

Code for Binary Search:

```
// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}
```

SORTING

Introduction:

Sorting is any process of arranging items systematically, and has two common, yet distinct meanings:

1. Ordering: arranging items in a sequence ordered by some criterion.
2. Categorizing: grouping items with similar properties.

Sorting Algorithms:

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

For example, the below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

BFMEIETNITK => BEEFIIKMNTT

Sorting Terminology:

In-Place Sorting:

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

Stable Sorting:

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Sorting Algorithms:

Some of the most commonly used popular sorting algorithms are:

1. Selection Sort
2. Bubble Sort

3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort

1. Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray. Following example explains the above steps:

arr[ ] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

Implementation of Selection Sort:
https://www.geeksforgeeks.org/selection-sort/

Time Complexity: O(n^2) as there are two nested loops.

Auxiliary Space: O(1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

2. Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), the algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2** **4** 5 8) –> (1 **2** **4** 5 8)
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)

Implementation of Bubble Sort: https://www.geeksforgeeks.org/bubble-sort/

Optimized Implementation:

The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if the inner loop didn't cause any swap.

Worst and Average Case Time Complexity: O(n^2). Worst case occurs when the array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when the array is already sorted.
Auxiliary Space: O(1)
Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Bubble sort is an In-Place Stable sorting algorithm. Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

3. Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)
i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

Implementation of Insertion Sort:
https://www.geeksforgeeks.org/insertion-sort/

Time Complexity: O(n^2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Even Insertion sort is an **In-Place Stable** sorting algorithm.

Uses: Insertion sort is used when the number of elements is small. It can also be useful when the input array is almost sorted, only few elements are misplaced in a complete big array.

Binary Insertion Sort

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in the worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n^2) because of the series of swaps required for each insertion. Refer this for implementation.

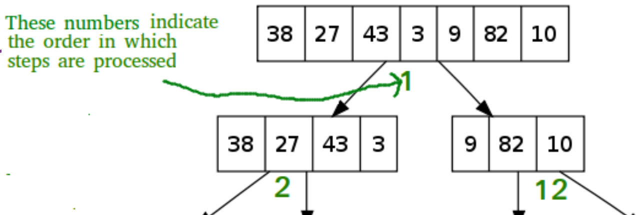
4. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

Algorithm: MergeSort(arr[], l, r)

```
If r > l
  1. Find the middle point to divide the array into two halves:
       middle m = l+ (r-l)/2
  2. Call mergeSort for first half:
       Call mergeSort(arr, l, m)
  3. Call mergeSort for second half:
       Call mergeSort(arr, m+1, r)
  4. Merge the two halves sorted in step 2 and 3:
       Call merge(arr, l, m, r)
```

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12

Implementation of Merge Sort: https://www.geeksforgeeks.org/merge-sort/

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \theta(n)$
The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is $\theta(nLogn)$. Time complexity of Merge Sort is $\theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

Applications of Merge Sort

- Useful for sorting linked lists in O(nLogn) time
- Inversion Count Problem
- Used in External Sorting

Drawbacks of Merge Sort

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of 0(n) for the temporary array.
- It goes through the whole process even if the array is sorted.

5. Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
```

```
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore the current element.

Pseudo code for partition()

```
/* This function takes last element as pivot, places the pivot
element at its correct position in sorted array, and places all
smaller (smaller than pivot) to left of pivot and all greater
elements to right of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element and indicates the
                   // right position of pivot found so far

    for (j = low; j <= high- 1; j++)
    {
            // If current element is smaller than the pivot
            if (arr[j] < pivot)
            {
                i++;    // increment index of smaller element
                swap arr[i] and arr[j]
            }
        }
        swap arr[i + 1] and arr[high])
        return (i + 1)
}
```

Illustration of partition() :

arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Implementation of Quick Sort: https://www.geeksforgeeks.org/quick-sort/

Analysis of QuickSort:

Time taken by QuickSort, in general, can be written as follows.

T(n) = T(k) + T(n-k-1) + Θ(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks the greatest or smallest element as pivot. If we consider the above partition strategy where the last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is a recurrence for the worst case.

T(n) = T(0) + T(n-1) + Θ(n)

which is equivalent to

$T(n) = T(n-1) + \Theta(n)$

The solution of above recurrence is $\Theta(n^2)$

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + \Theta(n)$

The solution of above recurrence is $\Theta(n\text{Log}n)$. It can be solved using case 2 of Master Theorem.

Average Case:

To do an average case analysis, we need to consider all possible permutations of an array and calculate the time taken by every permutation which doesn't look easy.

We can get an idea of the average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in another set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \Theta(n)$

Solution of above recurrence is also $O(n\text{Log}n)$

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

6. Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that the value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.

3. Repeat step 2 while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

```
           4(0)
          /    \
       10(1)   3(2)
       /   \
    5(3)    1(4)
```

The numbers in brackets represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
           4(0)
          /    \
      10(1)    3(2)
      /    \
5(3)      1(4)
```

Applying heapify procedure to index 0:

```
          10(0)
          /   \
       5(1)    3(2)
       /   \
```

4(3)   1(4)

The heapify procedure calls itself recursively to build heap in top down manner.

Implementation of Heap Sort: https://www.geeksforgeeks.org/heap-sort/

- Heap sort is an in-place algorithm. Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array

2. k largest(or smallest) elements in an array

Heap sort algorithms have limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See Applications of Heap Data Structure

STL for Sorting in C++

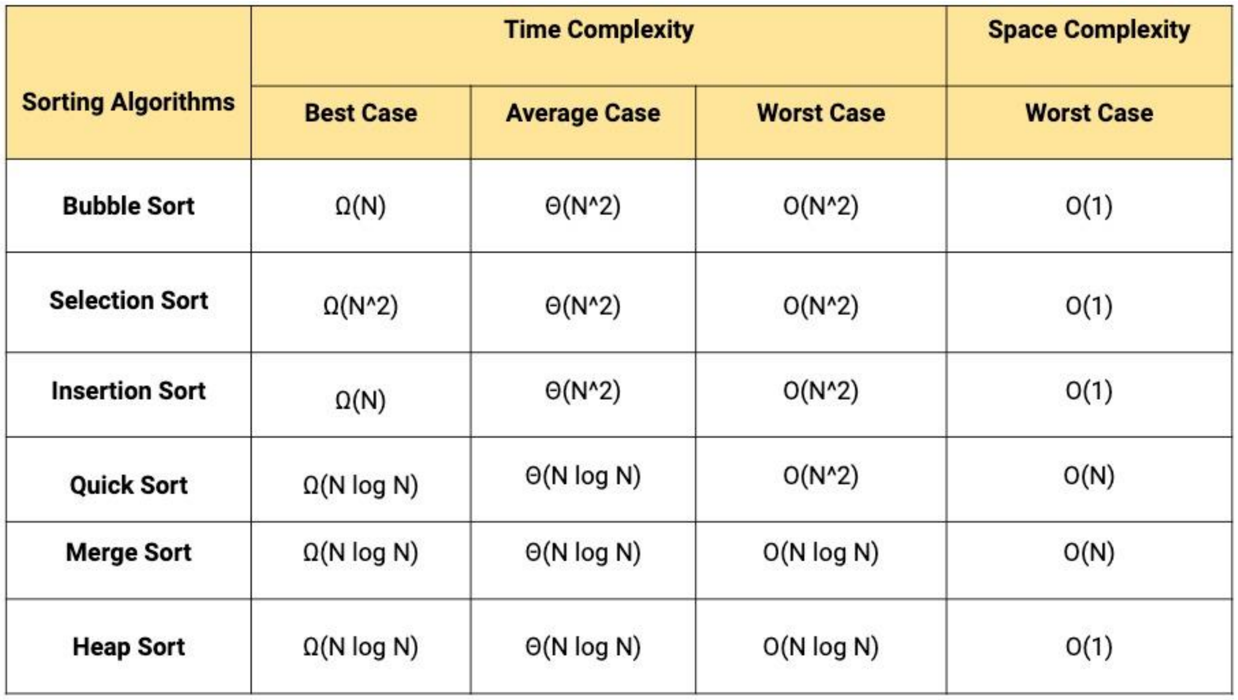
C++ STL provides a function sort that sorts a vector or array (items with random access).

It generally takes two parameters , the first one being the point of the array/vector from where the sorting needs to begin and the second parameter being the length up to which we want the array/vector to get sorted. The third parameter is optional and can be used in cases such as if we want to sort the elements lexicographically.

By default, sort() function sorts the element in ascending order.

For more details of STL for sorting refer this

Time and Space Complexities of Commonly used Sorting Algorithms:



Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Additional Topics on Sorting:
https://www.geeksforgeeks.org/sorting-algorithms/

Time Complexities of Commonly used Sorting Algorithms:
https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

Additional Resources:
https://codeburst.io/algorithms-i-searching-and-sorting-algorithms-56497dbaef20