Searching and Sorting
Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.

However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
• There are two popular methods for searching the array elements:

  *linear search*
  *binary search*

• The algorithm that should be used depends entirely on how the values are organized in the array.

• For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.
Linear Search

- Linear search, also called as **sequential search**, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array $A[]$ is declared and initialized as,

  ```
  int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
  Val = 2 ➔ Pos = 2
  ```
LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = VAL
    SET POS = I
    PRINT POS
    Go to Step 6
    [END OF IF]
    SET I = I + 1
    [END OF LOOP]
Step 5: IF POS = -1
    PRINT VALUE IS NOT PRESENT IN THE ARRAY
[END OF IF]
Step 6: EXIT
**Linear Search**

**Complexity of Linear Search Algorithm**

Linear search executes in $O(n)$ time where $n$ is the number of elements in the array.

Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.

Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, $n$ comparisons will have to be made.
Binary Search

Binary search is a searching algorithm that works efficiently with a sorted list.

The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

The same mechanism is applied in the binary search.
Binary Search

Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array $A[]$ that is declared and initialized as

$$\text{int } A[] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\};$$

value to be searched is $VAL = 9$

The algorithm will proceed in the following manner.

$\text{BEG} = 0, \text{END} = 10, \text{MID} = (0 + 10)/2 = 5$

A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 = 16/2 = 8


A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.

So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
Binary Search

The algorithm will terminate when $A[\text{MID}] = \text{VAL}$. When the algorithm ends, we will set $\text{POS} = \text{MID}$. $\text{POS}$ is the position at which the value is present in the array.

However, if $\text{VAL}$ is not equal to $A[\text{MID}]$, then the values of $\text{BEG}$, $\text{END}$, and $\text{MID}$ will be changed depending on whether $\text{VAL}$ is smaller or greater than $A[\text{MID}]$.

(a) If $\text{VAL} < A[\text{MID}]$, then $\text{VAL}$ will be present in the left segment of the array. So, the value of $\text{END}$ will be changed as $\text{END} = \text{MID} - 1$.

(b) If $\text{VAL} > A[\text{MID}]$, then $\text{VAL}$ will be present in the right segment of the array. So, the value of $\text{BEG}$ will be changed as $\text{BEG} = \text{MID} + 1$. 

Prepared By : Kaushik Vaghani
**Binary Search**

**BINARY_SEARCH(A, lower_bound, upper_bound, VAL)**

Step 1: [INITIALIZE] SET BEG = lower_bound  
END = upper_bound, POS = -1  
Step 2: Repeat Steps 3 and 4 while BEG <= END  
Step 3: SET MID = (BEG + END)/2  
Step 4: IF A[MID] = VAL  
   SET POS = MID  
   PRINT POS  
   Go to Step 6  
ELSE IF A[MID] > VAL  
   SET END = MID - 1  
ELSE  
   SET BEG = MID + 1  
[END OF IF]  
[END OF LOOP]  
Step 5: IF POS = -1  
   PRINT “VALUE IS NOT PRESENT IN THE ARRAY”  
[END OF IF]  
Step 6: EXIT
Sorting

• Bubble Sort
• Selection Sort
• Quick Sort
• Merge Sort
Selection Sort

Selection sort is a sorting algorithm that has a quadratic running time complexity of $O(n^2)$.

Procedure:
Consider an array ARR with $N$ elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap $ARR[POS]$ and $ARR[0]$. Thus, $ARR[0]$ is sorted.
- In Pass $N-1$, find the position POS of the smaller of the elements $ARR[N-2]$ and $ARR[N-1]$. Swap $ARR[POS]$ and $ARR[N-2]$ so that $ARR[0], ARR[1], ..., ARR[N-1]$ is sorted.
**Sorting**

**Example:** Sort the array given below using selection sort.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>9</td>
<td>39</td>
<td>81</td>
<td>45</td>
<td>90</td>
<td>27</td>
<td>72</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>18</td>
<td>81</td>
<td>45</td>
<td>90</td>
<td>27</td>
<td>72</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>45</td>
<td>90</td>
<td>81</td>
<td>72</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>39</td>
<td>90</td>
<td>81</td>
<td>72</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>39</td>
<td>45</td>
<td>81</td>
<td>72</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>39</td>
<td>45</td>
<td>72</td>
<td>81</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>39</td>
<td>45</td>
<td>72</td>
<td>81</td>
<td>90</td>
</tr>
</tbody>
</table>
```

Prepared By: Kaushik Vaghani
Sorting

Algorithm:

**SELECTION SORT(ARR, N)**

- Step 1: Repeat Steps 2 and 3 for K = 0 to N-1
- Step 2: CALL SMALLEST(ARR, K, N, POS)
  - [END OF LOOP]
- Step 4: EXIT

**SMALLEST (ARR, K, N, POS)**

- Step 1: [INITIALIZE] SET SMALL = ARR[K]
- Step 2: [INITIALIZE] SET POS = K
- Step 3: Repeat for J = K+1 to N-1
  - IF SMALL > ARR[J]
    - SET SMALL = ARR[J]
    - SET POS = J
    - [END OF IF]
  - [END OF LOOP]
- Step 4: RETURN POS

Prepared By: Kaushik Vaghani
**Sorting**

**Merge Sort**

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. The running time of merge sort in the average case and the worst case can be given as $O(n \log n)$.

*Divide* means partitioning the $n$-element array to be sorted into two sub-arrays of $n/2$ elements. If $A$ is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide $A$ into two sub-arrays, $A_1$ and $A_2$, each containing about half of the elements of $A$.

*Conquer* means sorting the two sub-arrays recursively using merge sort.

*Combine* means merging the two sorted sub-arrays of size $n/2$ to produce the sorted array of $n$ elements.
Sorting

Merge Sort

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

→ A smaller list takes fewer steps and thus less time to sort than a large list.
→ As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

Procedure:
The basic steps of a merge sort algorithm are as follows:

→ If the array is of length 0 or 1, then it is already sorted.
→ Otherwise, divide the unsorted array into two sub-arrays of about half the size.
→ Use merge sort algorithm recursively to sort each sub-array.
→ Merge the two sub-arrays to form a single sorted list.
Example: Sort the array given below using merge sort.

39 9 81 45 90 27 72 18

(Divide and Conquer the array)

(Combine the elements to form a sorted array)
The merge sort algorithm uses a function **merge** which combines the sub-arrays to form a sorted array. While the merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

To understand the merge algorithm, consider the figure below which shows how we merge two lists to form one list. For ease of understanding, we have taken two sub-lists each containing four elements. The same concept can be utilized to merge four sub-lists containing two elements, or eight sub-lists having one element each.

![Merge Algorithm Diagram](image)

Compare ARR[I] and ARR[J], the smaller of the two is placed in TEMP at the location specified by INDEX and subsequently the value I or J is incremented.
When I is greater than MID, copy the remaining elements of the right sub-array in TEMP.
**Sorting**

Algorithm:

MERGE_SORT(ARR, BEG, END)

Step 1: IF BEG < END

    SET MID = (BEG + END)/2
    CALL MERGE_SORT (ARR, BEG, MID)
    CALL MERGE_SORT (ARR, MID + 1, END)
    MERGE (ARR, BEG, MID, END)

[END OF IF]

Step 2: END


**Sorting**

**MERGE (ARR, BEG, MID, END)**

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J<=END)

- IF ARR[I] < ARR[J]
  - SET TEMP[INDEX] = ARR[I]
  - SET I = I + 1
- ELSE
  - SET TEMP[INDEX] = ARR[J]
  - SET J = J + 1

[END OF IF]

SET INDEX = INDEX + 1

[END OF LOOP]

Cont..
Sorting

Step 3: [Copy the remaining elements of right sub-array, if any]

IF I > MID

Repeat while J <= END

SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1, SET J = J + 1
[END OF LOOP]

[Copy the remaining elements of left sub-array, if any]
ELSE

Repeat while I <= MID

SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]

[END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K= 0

Step 5: Repeat while K < INDEX

SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]

Step 6: END
Quick Sort

Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes $O(n \log n)$ comparisons in the average case to sort an array of $n$ elements. However, in the worst case, it has a quadratic running time given as $O(n^2)$. Basically, the quick sort algorithm is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

Like merge sort, this algorithm works by using a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.
Quick Sort

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)
Quick Sort

Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves. To understand how we find the pivot element, follow the steps given below. (We take the first element in the array as pivot.)

Procedure:
Quick sort works as follows:

1. Set the index of the first element in the array to loc and left variables. Also, set the index of the last element of the array to the right variable. That is, loc = 0, left = 0, and right = n–1 (where n in the number of elements in the array)
Sorting

Quick Sort

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right].

(a) If that is the case, then simply continue comparing until right becomes equal to loc. Once right = loc, it means the pivot has been placed in its correct position.
(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.
(c) Set loc = right

3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc. That is, a[loc] should be greater than a[left].

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position.
(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.
(c) Set loc = left.
Example: Sort the elements given in the following array using quick sort algorithm.

```
27  10  36  18  25  45
```

We choose the first element as the pivot. Set loc = 0, left = 0, and right = 5.

Scan from right to left. Since a[loc] < a[right], decrease the value of right.

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

Since a[loc] > a[right], interchange the two values and set loc = right.
Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.