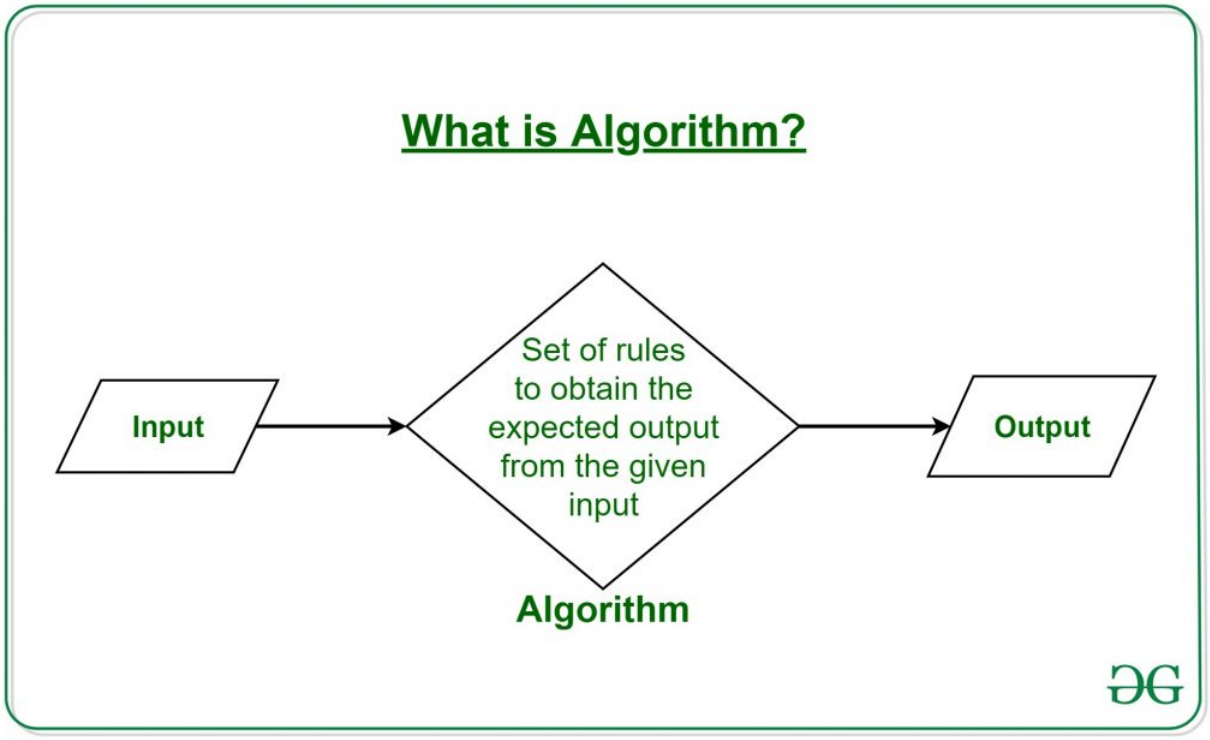
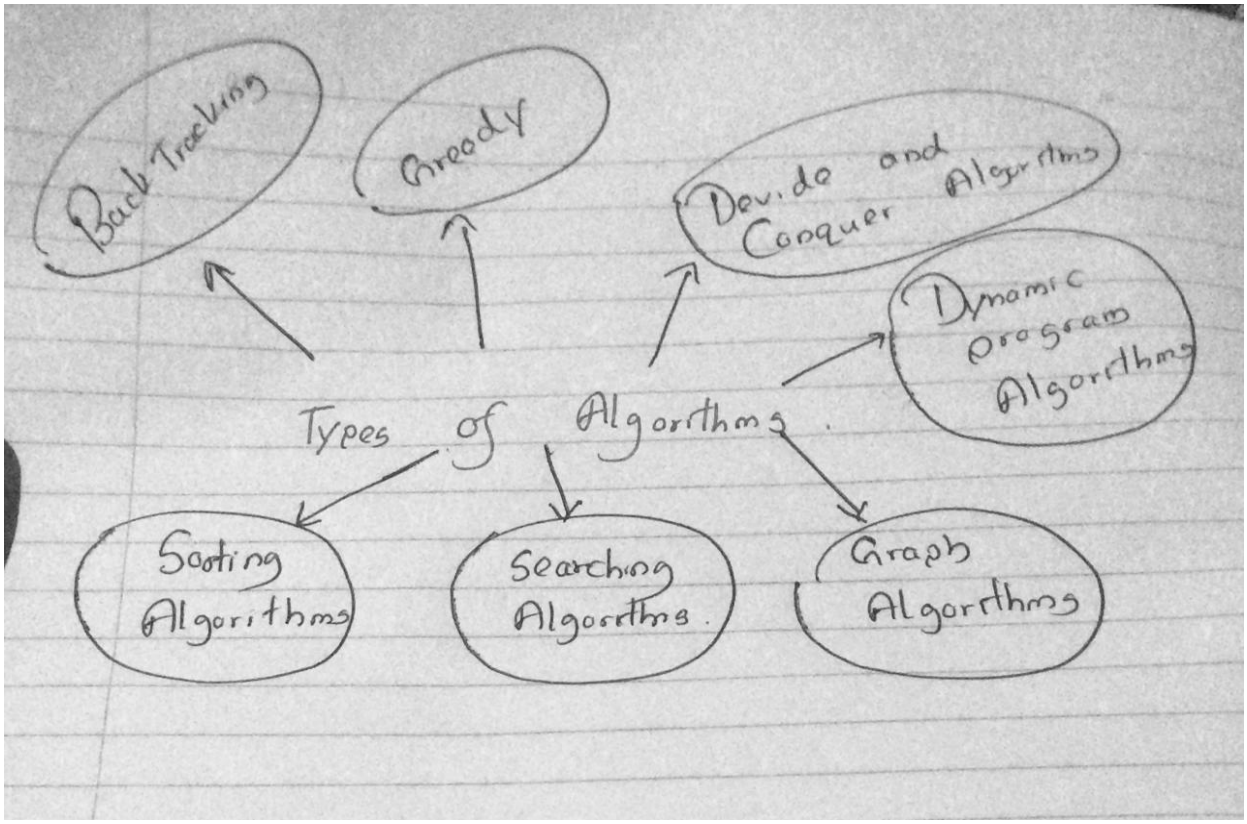
Lets learn Algorithms #1

What is an Algorithm?

The word Algorithm means “A set of rules to be followed in calculations or other problem-solving operations” Or “A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations “.





Algorithms can be classified into various types based on their design, functionality, and application. Here are some common types of algorithms:

1. Sorting Algorithms:
    - Examples: Bubble Sort, Merge Sort, QuickSort
    - Purpose: Rearrange elements in a specific order, such as ascending or descending.
2. Searching Algorithms:
    - Examples: Binary Search, Linear Search
    - Purpose: Locate a specific item or value within a dataset.
3. Graph Algorithms:
    - Examples: Depth-First Search (DFS), Breadth-First Search (BFS), Dijkstra's Algorithm
    - Purpose: Solve problems related to graphs, such as finding paths, connectivity, or shortest paths.

4. Dynamic Programming Algorithms:
    - Examples: Fibonacci sequence calculation, Knapsack problem
    - Purpose: Optimize solutions to problems by breaking them into subproblems and solving them recursively.
5. Divide and Conquer Algorithms:
    - Examples: Merge Sort, QuickSort
    - Purpose: Break down a problem into smaller subproblems, solve them independently, and combine their solutions.
6. Greedy Algorithms:
    - Examples: Kruskal's Algorithm, Prim's Algorithm
    - Purpose: Make locally optimal choices at each stage with the hope of finding a global optimum.
7. Backtracking Algorithms:
    - Examples: N-Queens problem, Sudoku solver
    - Purpose: Systematically explore all possible solutions to a problem by trying different options and backtracking when necessary.

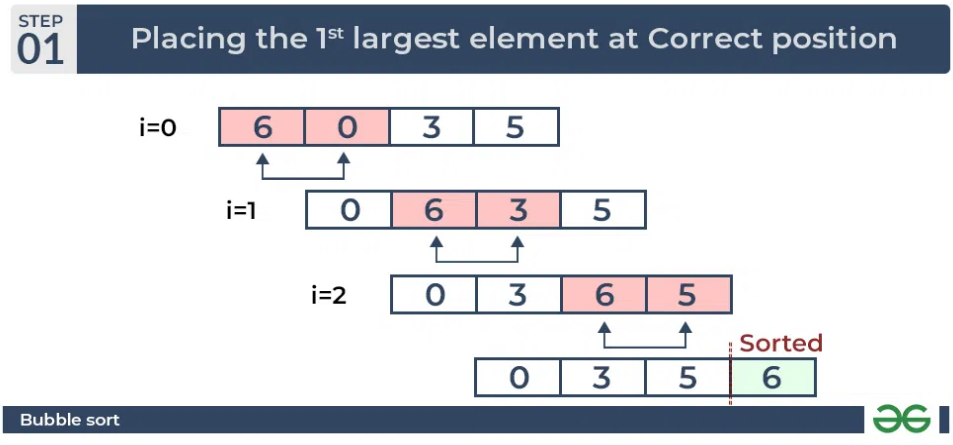
Bubble Sort:

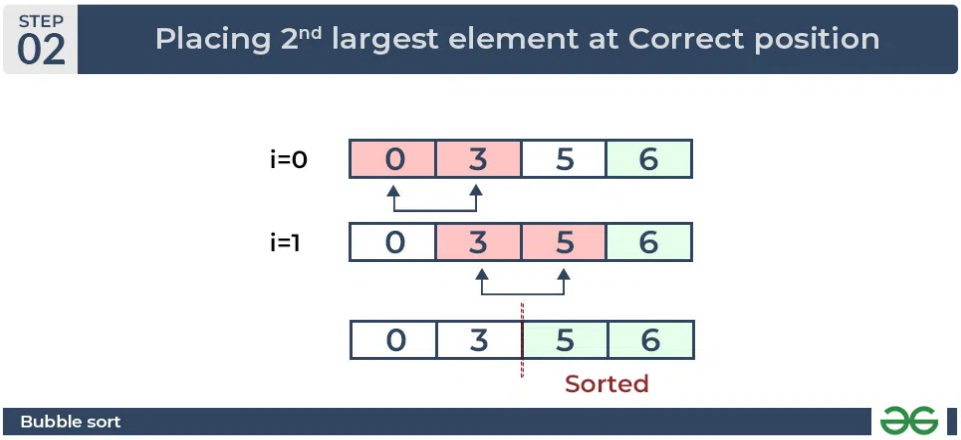
Introduction:

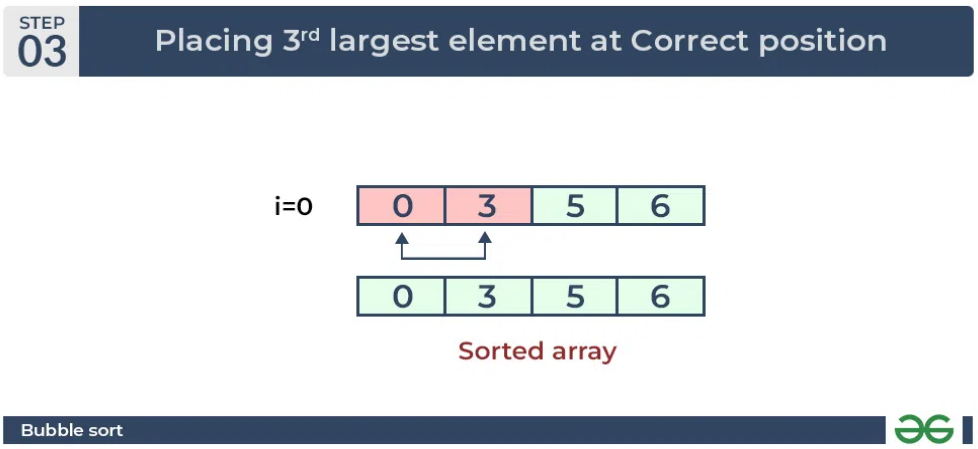
Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Real-World Analogy:

Imagine you have a row of people standing in a line, and you want to arrange them in ascending order of height. In each pass, you compare the height of adjacent people and swap them if they are in the wrong order. You repeat this process until the entire line is sorted by height.


STEP 01
Placing the 1st largest element at Correct position
i=0
6 0 3 5
i=1
0 6 3 5
i=2
0 3 6 5
Sorted
0 3 5 6
Bubble sort





- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2

Pseudocode

```
procedure bubbleSort(list)
    n = length(list)
    for i from 0 to n-1
        for j from 0 to n-i-1
```

```
        if list[j] > list[j+1]
            swap(list[j], list[j+1])
```

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Bubble Sort function
void bubbleSort(int arr[], int n) {
    // Outer loop for each pass
    for (int i = 0; i < n - 1; i++) {
        // Inner loop for each comparison in the pass
        for (int j = 0; j < n - i - 1; j++) {
            // Compare adjacent elements and swap if they are
in the wrong order
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
// Main function
```

```
int main() {
    // Input array
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // Calculate the size of the array
    int n = sizeof(arr) / sizeof(arr[0]);
    // Call Bubble Sort function
    bubbleSort(arr, n);
    // Display the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

1. swap Function:
    - This function swaps the values of two elements using a temporary variable.
2. bubbleSort Function:
    - The outer loop iterates through each pass of the Bubble Sort.
    - The inner loop compares adjacent elements and swaps them if they are in the wrong order.
3. main Function:
    - The input array is defined.
    - The size of the array is calculated.
    - The bubbleSort function is called to sort the array.
    - The sorted array is displayed.

Merge Sort:

Introduction:

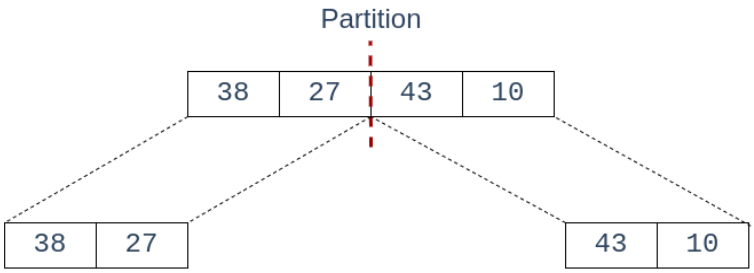
Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, recursively sorts them, and then merges the sorted halves to produce a fully sorted array.

Real-World Analogy:

Imagine you have a deck of cards, and you want to sort them. Divide the deck into two halves, sort each half separately, and then merge them back together in sorted order.

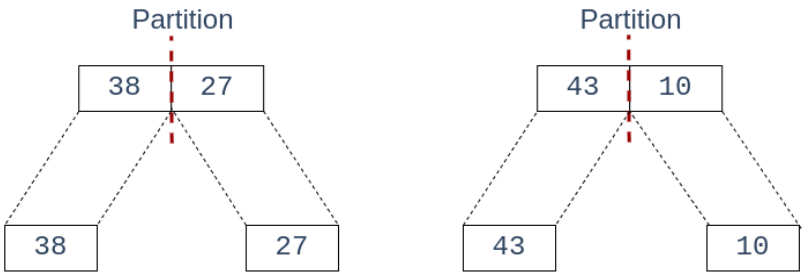
STEP 01

Splitting the Array into two equal halves



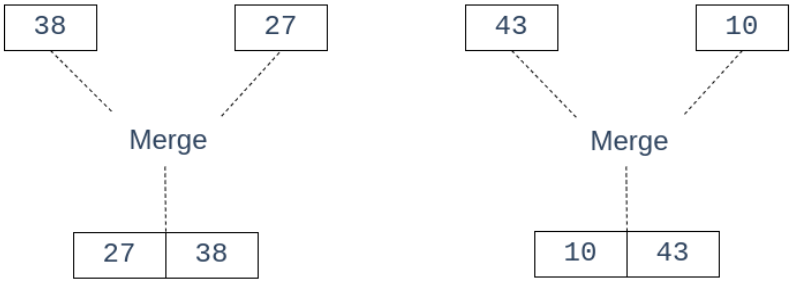
STEP 02

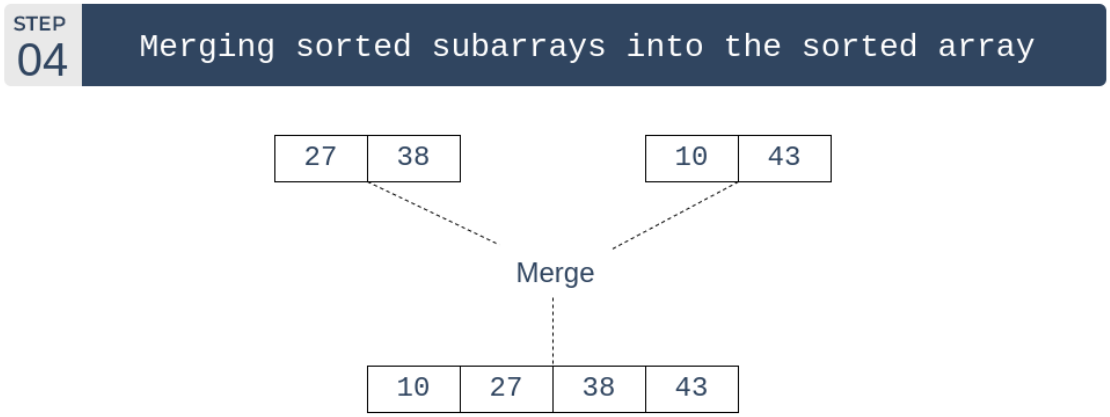
Splitting the subarrays into two halves



STEP 03

Merging unit length cells into sorted subarrays





Merge Sort

```
procedure mergeSort(list)
    if length(list) <= 1
        return list
    mid = length(list) / 2
    left = mergeSort(list[0:mid])
    right = mergeSort(list[mid:])
    return merge(left, right)
procedure merge(left, right)
    result = []
    while left is not empty and right is not empty
        if left[0] <= right[0]
            append left[0] to result
            remove left[0] from left
        else
            append right[0] to result
            remove right[0] from right
    append remaining elements of left to result
    append remaining elements of right to result
    return result
```

```
#include <stdio.h>
// Function to merge two sorted arrays
void merge(int arr[], int l, int m, int r) {
```

```
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
// Create temporary arrays
int L[n1], R[n2];
// Copy data to temporary arrays L[] and R[]
for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];
// Merge the temporary arrays back into arr[l..r]
i = 0;
j = 0;
k = l;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}
// Copy the remaining elements of L[], if there are any
while (i < n1) {
    arr[k] = L[i];
    i++;
```

```
        k++;
    }
    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Merge Sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for large l and r
        int m = l + (r - l) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
// Main function
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    // Call Merge Sort function
    mergeSort(arr, 0, n - 1);
    // Display the sorted array
```

```
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

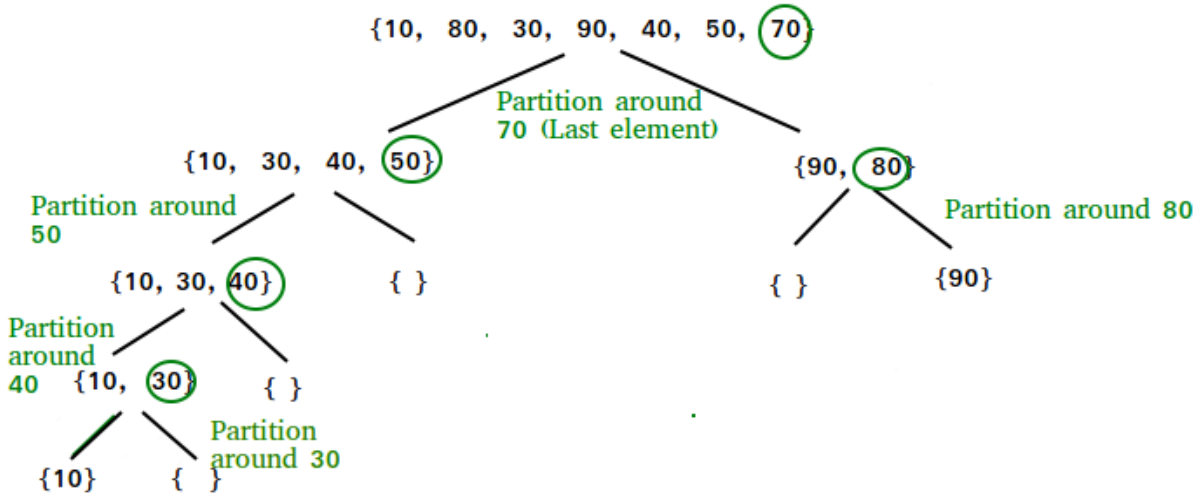
QuickSort:

Introduction:

QuickSort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into two sub-arrays, such that elements less than the pivot are on the left, and elements greater than the pivot are on the right. The sub-arrays are then sorted recursively.

Real-World Analogy:

Imagine you have a shelf of books, and you want to organize them. Choose a book as a pivot, place smaller books to its left and larger books to its right. Repeat this process for each sub-shelf until the entire shelf is sorted.




```
procedure quickSort(list, low, high)
   if low < high
      pi = partition(list, low, high)
      quickSort(list, low, pi - 1)
      quickSort(list, pi + 1, high)
procedure partition(list, low, high)
   pivot = list[high]
```

```
    i = low - 1
    for j from low to high - 1
        if list[j] <= pivot
            i++
            swap list[i] and list[j]
    swap list[i + 1] and list[high]
    return i + 1
```

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function for QuickSort
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivot is the last element
    int i = low - 1; // Index of smaller element
    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    // Swap the pivot element with the element at (i + 1)
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
// QuickSort function
```

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);
        // Recursively sort elements before and after the partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Main function
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    // Call QuickSort function
    quickSort(arr, 0, n - 1);
// Display the sorted array
printf("Sorted array: "
);
for (int i = 0
; i < n; i++) {

printf("%d "
, arr[i]);
}
return 0
;
}
```

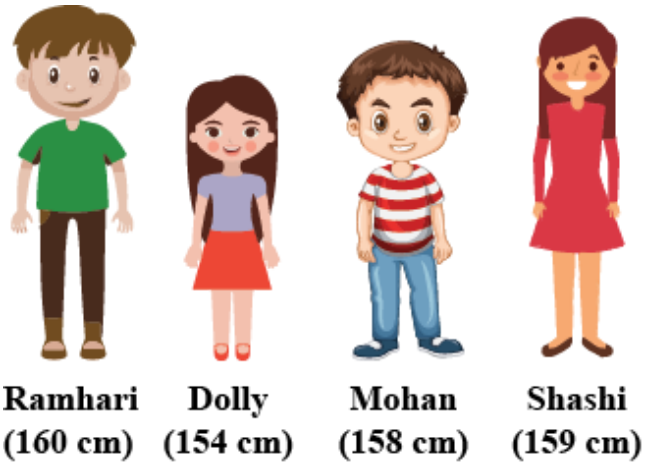
Selection Sort:

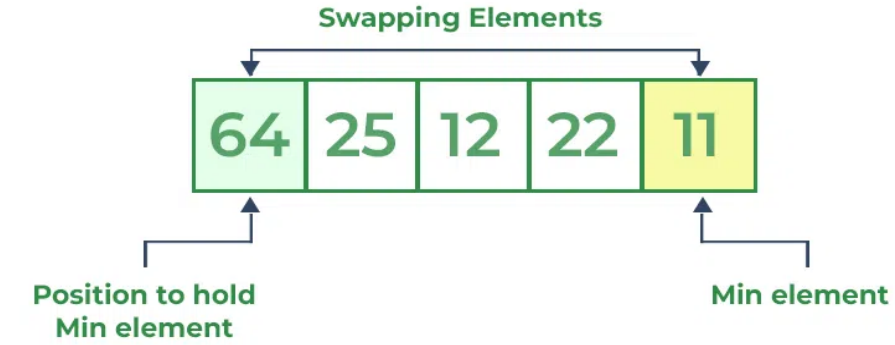
Introduction:

Selection Sort is a simple sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted part of the array and swaps it with the first unsorted element. The process is repeated until the entire array is sorted.

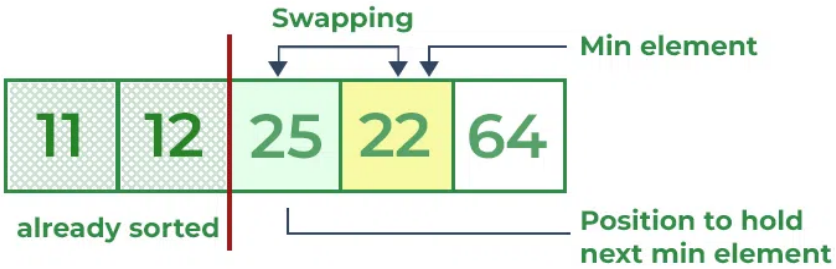
Real-World Analogy:

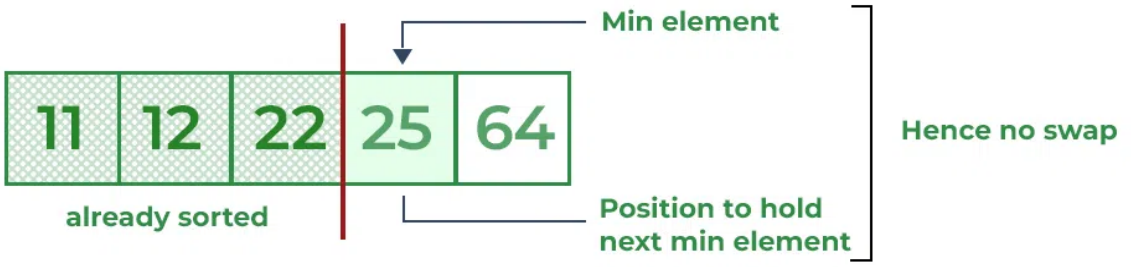
Imagine you have a row of students, and you want to arrange them in ascending order of height. In each iteration, you find the shortest student from the remaining unsorted students and place them at the beginning of the sorted section.




Swapping Elements
64
25
12
22
11
Position to hold
Min element
Min element


Swapping
Min element
11
25
12
22
64
already sorted
Position to hold
next min element


Swapping
Min element
11
12
25
22
64
already sorted
Position to hold
next min element



Selection Sort

Selection Sort

Pseudocode:

```
procedure selectionSort(list)
    n = length(list)
    for i from 0 to n-1
        // Assume the current element is the minimum
        minIndex = i
        for j from i+1 to n
            // Find the index of the minimum element
            if list[j] < list[minIndex]
                minIndex = j
        // Swap the found minimum element with the first unsorted element
```

```
      swap(list[i], list[minIndex])
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Selection Sort function
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Assume the current element is the minimum
        int minIndex = i;
        // Find the index of the minimum element in the unsorted part
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first unsorted
element
        swap(&arr[i], &arr[minIndex]);
    }
}
// Main function
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    // Call Selection Sort function
```

```
    selectionSort(arr, n);
    // Display the sorted array
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Compare and contrast the Merge Sort, Bubble Sort, and Selection Sort

1. Time Complexity:

- Merge Sort:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
- Bubble Sort:
  - Best Case: O(n) - with optimized versions.
  - Average Case: O(n^2)
  - Worst Case: O(n^2)
- Selection Sort:
  - Best Case: O(n^2)
  - Average Case: O(n^2)
  - Worst Case: O(n^2)

2. Space Complexity:

- Merge Sort:
  - Additional Space: O(n) - Requires additional space for merging.
  - In-Place Sorting: No (except for optimized versions).
- Bubble Sort:
  - Additional Space: O(1) - In-place sorting.
- Selection Sort:
  - Additional Space: O(1) - In-place sorting.

3. Stability:

- Merge Sort:

    - Stable Sorting: Yes
    - Explanation: Maintains the relative order of equal elements.
- Bubble Sort:
    - Stable Sorting: Yes
    - Explanation: Maintains the relative order of equal elements.
- Selection Sort:
    - Stable Sorting: No
    - Explanation: May change the relative order of equal elements.

4. Suitability:

- Merge Sort:
    - Large Datasets: Efficient due to its divide-and-conquer approach.
    - Linked Lists: Suitable, as it doesn't rely on random access.
- Bubble Sort:
    - Small Datasets: Simple to implement and may be suitable for small datasets.
    - Linked Lists: More challenging due to the need for swapping elements.
- Selection Sort:
    - Small Datasets: Simple to implement, but less efficient for larger datasets.
    - Linked Lists: More challenging due to the need for swapping elements.
- **Conclusion:**
    - Merge Sort:
        - Efficient for large datasets, stable, and suitable for linked lists.
    - Bubble Sort:
        - Simple to implement, suitable for small datasets, and stable.
    - Selection Sort:
        - Simple to implement but less efficient for larger datasets. Not stable

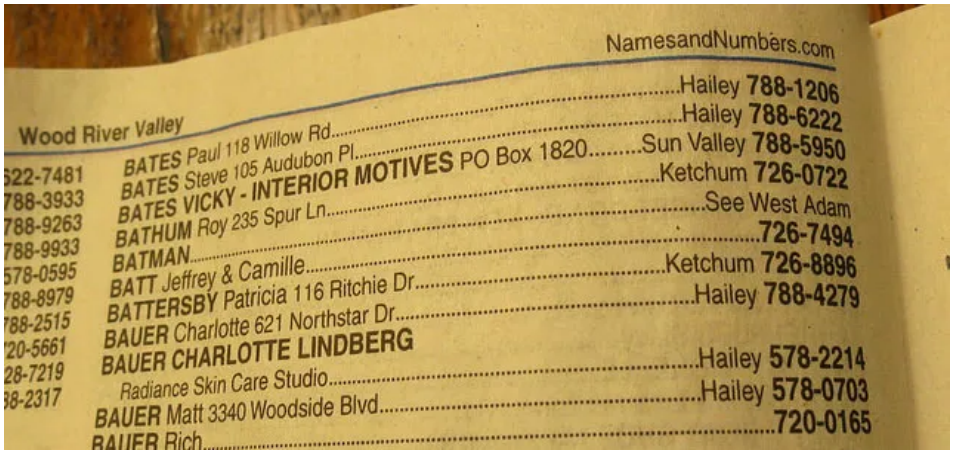
Binary Search:

Introduction:

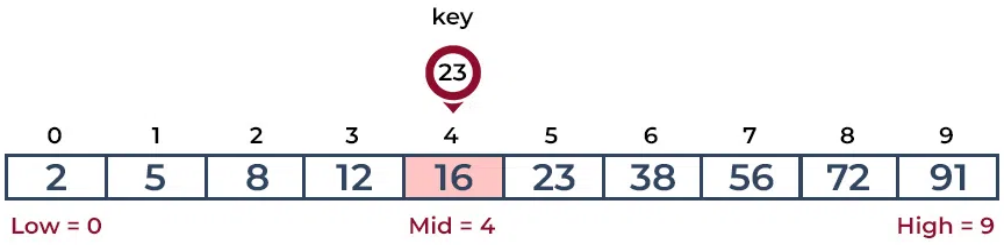
Binary Search is an efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half, comparing the middle element to the target value.

Real-World Analogy:

Imagine you have a phone book sorted by names. To find a specific person, you could open the book in the middle, check if the person's name is before or after the current page, and repeat this process until you find the person.

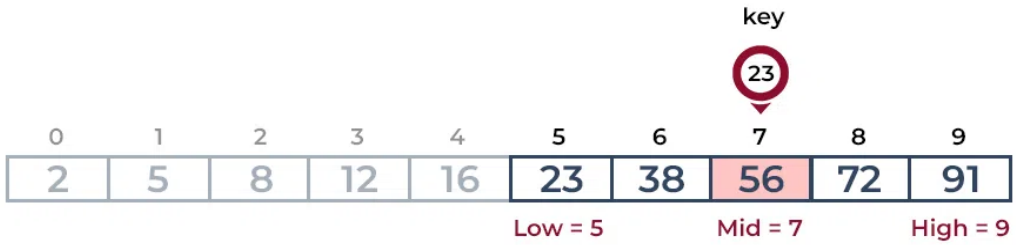


Consider an array ***arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}****, and the* ***target = 23****.*



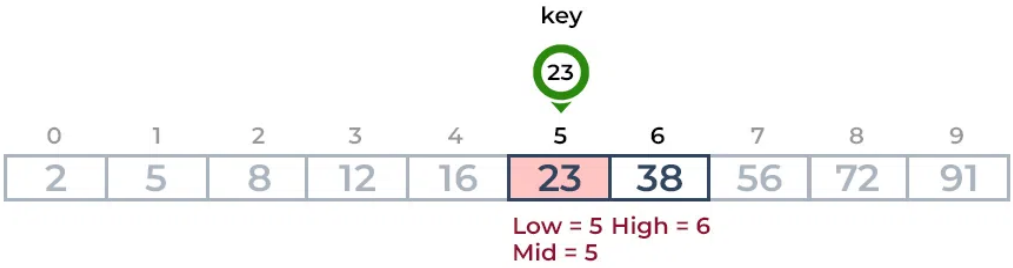
Binary search

- *Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.*



Binary search

- *Key is less than the current mid 56. The search space moves to the left.*



Binary search

Second Step: *If the key matches the value of the mid element, the element is found and stop search.*

Pseudocode:

```
procedure binarySearch(arr, target)
    low = 0
    high = length(arr) - 1
    while low <= high
        mid = (low + high) / 2
        if arr[mid] == target
            return mid  // Target found
        else if arr[mid] < target
            low = mid + 1
        else
            high = mid - 1
    return -1  // Target not found
```

```
#include <stdio.h>
// Binary Search function
int binarySearch(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target)
            return mid;  // Target found
```

```
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;  // Target not found
}
// Main function
int main() {
    int arr[] = {11, 22, 34, 45, 56, 67, 78, 89, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 56;
    // Call Binary Search function
    int result = binarySearch(arr, n, target);
    // Display the result
    if (result != -1)
        printf("Element %d found at index %d\n", target, result);
    else
        printf("Element %d not found in the array\n", target);
    return 0;
}
```

Linear Search:

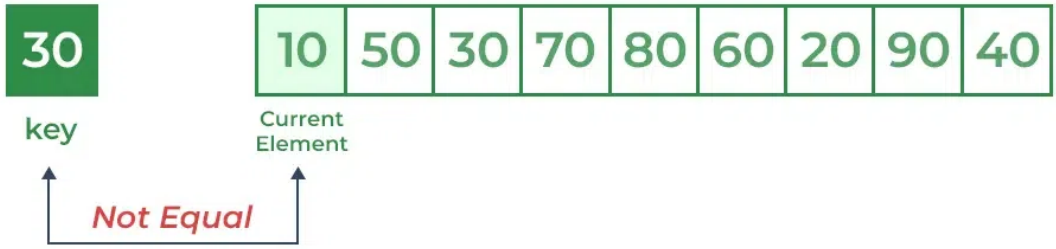
Introduction:

Linear Search is a simple search algorithm that sequentially checks each element in the array until a match is found or the entire array has been searched.

Real-World Analogy:

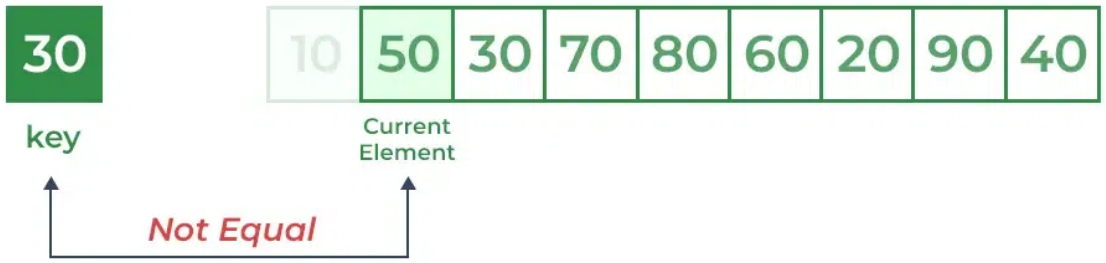
Imagine you are looking for a specific book in a shelf of books. You start from one end and check each book until you find the one you are looking for.

For example: Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30



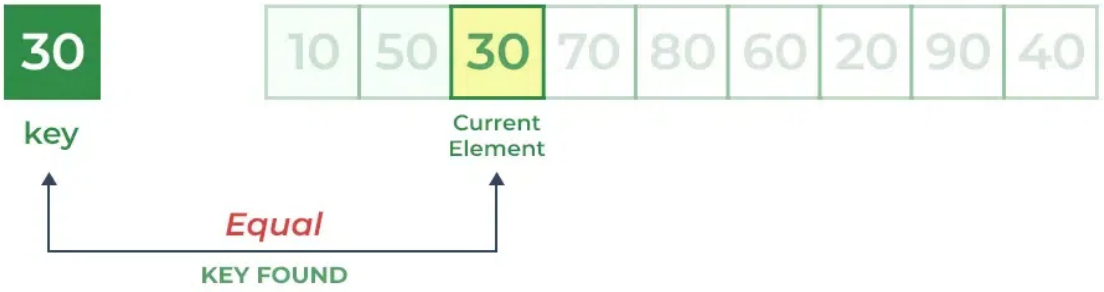
Linear Search Algorithm

- *Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.*



Linear Search Algorithm

- *Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.*



Linear Search Algorithm

Step 2: *Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).*

Pseudocode

```
procedure linearSearch(arr, target)
   for i from 0 to length(arr)-1
      if arr[i] == target
         return i  // Target found
   return -1  // Target not found
```

```
#include <stdio.h>
// Linear Search function
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target)
            return i;  // Target found
    }
    return -1;  // Target not found
}
// Main function
int main() {
    int arr[] = {11, 22, 34, 45, 56, 67, 78, 89, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int target = 56;
    // Call Linear Search function
    int result = linearSearch(arr, n, target);
    // Display the result
    if (result != -1)
        printf("Element %d found at index %d\n", target, result);
    else
        printf("Element %d not found in the array\n", target);
    return 0;
}
```

Let's compare and contrast Binary Search and Linear Search

1. Time Complexity:

- Binary Search:
  - Best Case: O(1) - constant time for finding the target in the middle.
  - Average Case: O(log n) - logarithmic time due to the division of the search space.
  - Worst Case: O(log n) - logarithmic time for a sorted array.
- Linear Search:
  - Best Case: O(1) - constant time if the target is at the beginning.
  - Average Case: O(n/2) - linear time on average, as the target might be found in the middle.
  - Worst Case: O(n) - linear time for searching the entire array.

2. Space Complexity:

- Binary Search:
  - Additional Space: O(1) - constant space for storing indices and variables.
- Linear Search:
  - Additional Space: O(1) - constant space, as no additional data structures are used.

3. Suitability:

- Binary Search:

    - Suitable for sorted arrays or lists.
    - Efficient for large datasets.
    - Requires a sorted dataset.
- Linear Search:
    - Suitable for both sorted and unsorted arrays.
    - Simple and intuitive.
    - Efficient for small datasets.

4. Search Space Division:

- Binary Search:
    - Divides the search space in half at each step.
    - Efficient for large datasets due to rapid elimination.
- Linear Search:
    - Scans the search space sequentially.
    - Suitable for small datasets, as it checks each element one by one.

5. Stability:

- Binary Search:
    - Stable.
- Linear Search:
    - Stable.

Conclusion:

- Binary Search:
    - More efficient for large sorted datasets.
    - Requires a sorted dataset.
- Linear Search:
    - Simple and applicable to both sorted and unsorted datasets.
    - More suitable for small datasets.

Doc By - AJ

Reference -

GPT 3.5

www.geeksforgeeks.org