Selection Sort

Selection Sort is a straightforward and intuitive sorting algorithm that can be a good starting point for beginners in computer science. Here’s a step-by-step explanation and implementation in Python.

Explanation

Selection Sort works by repeatedly finding the minimum (or maximum) element from the unsorted portion of the list and placing it at the beginning (or end). Here’s how it works step-by-step:

1. **Initialize**: Start with the first element of the list.
2. **Find the Minimum**: Find the smallest element in the list from the current position to the end.
3. **Swap**: Swap this smallest element with the element at the current position.
4. **Move to Next Position**: Move to the next position in the list.
5. **Repeat**: Repeat the process until the entire list is sorted.

Step-by-Step Example

Consider an array `[64, 25, 12, 22, 11].`

1. **First Iteration**:
    - Current position: `0`
    - Find the minimum from `64, 25, 12, 22, 11` which is `11`.
    - Swap `64` and `11`.
    - Array becomes: `[11, 25, 12, 22, 64]`.
2. **Second Iteration**:
    - Current position: 1
    - Find the minimum from **25, 12, 22, 64** which is **12**.
    - Swap 25 and 12.
    - Array becomes: [**11, 12, 25, 22, 64**].
3. **Third Iteration**:
    - Current position: 2
    - Find the minimum from **25, 22, 64** which is 22.
    - Swap 25 and 22.

- Array becomes: [11, 12, 22, 25, 64].

4. **Fourth Iteration**:

    - Current position: `3`
    - Find the minimum from `25, 64` which is `25`.
    - No swap needed.
    - Array remains: **`[11, 12, 22, 25, 64].`**

5. **Fifth Iteration**:

    - Current position: `4`
    - Only one element left, no need to check further.
    - Array remains: **`[11, 12, 22, 25, 64].`**

The array is now sorted.

```java
package com.mphasis.sorting;
public class SelectionSort {

    // Method to perform selection sort
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        // One by one move the boundary of the unsorted subarray
        for (int i = 0; i < n - 1; i++) {
            // Find the minimum element in the unsorted array
            int minIdx = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIdx]) {
                    minIdx = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = arr[minIdx];
            arr[minIdx] = arr[i];
            arr[i] = temp;
        }
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the sorting algorithm
    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        System.out.println("Original array:");
        printArray(arr);

        selectionSort(arr);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

How It Works

1. **Outer Loop**: The outer loop runs from the beginning of the array to the second-to-last element. This loop selects each element one by one as the current position to find the minimum element in the remaining unsorted part of the array.
2. **Inner Loop**: The inner loop runs from the element just after the current position to the end of the array, finding the smallest element in this unsorted part.
3. **Swapping**: After finding the smallest element in the unsorted part, it is swapped with the element at the current position.
4. **Repeat**: This process repeats until the entire array is sorted.

Key Points

- **Time Complexity**: O(n^2) - Selection Sort is not efficient for large datasets.
- **Space Complexity**: O(1) - Selection Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional memory.
- **Stability**: Selection Sort is not stable - it does not necessarily preserve the relative order of equal elements.

Bubble Sort

Bubble Sort is a simple comparison-based algorithm where each pair of adjacent elements is compared, and the elements are swapped if they are in the wrong order. This process is repeated until the array is sorted.

Step-by-Step Example

Consider an array `[64, 34, 25, 12, 22, 11, 90]`.

1. **First Pass**:

    - Compare `64` and `34`, swap (since `64 > 34`).
    - Compare `64` and `25`, swap (since `64 > 25`).
    - Compare `64` and `12`, swap (since `64 > 12`).
    - Compare `64` and `22`, swap (since `64 > 22`).
    - Compare `64` and `11`, swap (since `64 > 11`).
    - Compare `64` and `90`, no swap (since `64 < 90`).

- Array after first pass: `[34, 25, 12, 22, 11, 64, 90]`.

2. **Second Pass**:

   - Compare `34` and `25`, swap (since `34 > 25`).
   - Compare `34` and `12`, swap (since `34 > 12`).
   - Compare `34` and `22`, swap (since `34 > 22`).
   - Compare `34` and `11`, swap (since `34 > 11`).
   - Compare `34` and `64`, no swap (since `34 < 64`).
   - Array after second pass: `[25, 12, 22, 11, 34, 64, 90]`.

3. **Subsequent Passes**:

   - Continue this process for the remaining passes until no more swaps are needed.

Java Implementation

Here is a Java function that implements Bubble Sort:

```
package com.mphasis.sorting;
public class BubbleSort {

    // Method to perform bubble sort
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;
        // Loop through each element in the array
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            // Compare adjacent elements and swap if needed
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j + 1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // If no elements were swapped, the array is 
already sorted
            if (!swapped) {
                break;
            }
        }
    }

    // Method to print the array
    public static void printArray(int[] arr) {
```

```java
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the sorting algorithm
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        System.out.println("Original array:");
        printArray(arr);

        bubbleSort(arr);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

How It Works

1. **Outer Loop**: The outer loop runs from the beginning of the array to the second-to-last element. This loop ensures that we pass through the entire array multiple times.
2. **Inner Loop**: The inner loop runs from the beginning of the array to the `n - i - 1` element, comparing each pair of adjacent elements and swapping them if they are in the wrong order.
3. **Swapping**: If a pair of adjacent elements is in the wrong order, they are swapped.
4. **Optimization**: The `swapped` boolean variable is used to optimize the algorithm. If no elements were swapped during a pass, the array is already sorted, and we can break out of the loop early.

Key Points

- **Time Complexity**: O(n^2) - Bubble Sort is not efficient for large datasets.
- **Space Complexity**: O(1) - Bubble Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional memory.
- **Stability**: Bubble Sort is stable - it preserves the relative order of equal elements.
- ==

Insertion Sort

Explanation

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Step-by-Step Example

Consider an array `[64, 34, 25, 12, 22, 11, 90]`.

1. **First Pass**:

    - The first element, `64`, is considered sorted.
    - Take the next element `34` and insert it into the sorted part.
    - Since `34` is less than `64`, move `64` to the right and insert `34` at the beginning.
    - Array becomes: `[34, 64, 25, 12, 22, 11, 90]`.

2. **Second Pass**:

    - Take `25` and insert it into the sorted part `[34, 64]`.
    - Move `64` to the right and then `34` to the right, and insert `25` at the beginning.
    - Array becomes: `[25, 34, 64, 12, 22, 11, 90]`.

3. **Subsequent Passes**:

    - Continue this process for each element in the array.

Java Implementation

Here is a Java function that implements Insertion Sort:

```java
package com.mphasis.sorting;
public class InsertionSort {

    // Method to perform insertion sort
    public static void insertionSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;

            // Move elements of arr[0..i-1] that are greater than key
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the sorting algorithm
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        System.out.println("Original array:");
        printArray(arr);

        insertionSort(arr);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

How It Works

1. **Outer Loop**: The outer loop iterates over each element of the array starting from the second element (since the first element is already considered sorted).
2. **Key Element**: The current element to be inserted into the sorted part is stored in the `key` variable.
3. **Inner Loop**: The inner loop compares the `key` with elements in the sorted part of the array. It shifts elements that are greater than the `key` to one position ahead of their current position.
4. **Insertion**: The `key` is inserted into its correct position in the sorted part of the array.

Key Points

- **Time Complexity**: O(n^2) - Insertion Sort is not efficient for large datasets.
- **Space Complexity**: O(1) - Insertion Sort is an in-place sorting algorithm, meaning it requires a constant amount of additional memory.
- **Stability**: Insertion Sort is stable - it preserves the relative order of equal elements.
- ==

Merge Sort

Merge Sort is a classic divide-and-conquer sorting algorithm that efficiently sorts an array by dividing it into smaller sub-arrays, sorting those, and then merging the sorted sub-arrays. Here’s how you can implement Merge Sort in Java:

Explanation

Merge Sort works as follows:

1. **Divide**: Split the array into two halves until each sub-array contains a single element.
2. **Conquer**: Recursively sort the sub-arrays.
3. **Combine**: Merge the sorted sub-arrays to produce the final sorted array.

Step-by-Step Example

Consider an array `[38, 27, 43, 3, 9, 82, 10]`.

1. **Divide**:

   - Split into `[38, 27, 43, 3]` and `[9, 82, 10]`.
   - Further split `[38, 27, 43, 3]` into `[38, 27]` and `[43, 3]`, and `[9, 82, 10]` into `[9]` and `[82, 10]`.
   - Continue splitting until each sub-array contains a single element.

2. **Conquer**:

   - Sort each small array: `[38, 27]` becomes `[27, 38]`, `[43, 3]` becomes `[3, 43]`, `[82, 10]` becomes `[10, 82]`.

3. **Combine**:

   - Merge sorted arrays: `[27, 38]` and `[3, 43]` into `[3, 27, 38, 43]`, and merge `[9]` with `[10, 82]` into `[9, 10, 82]`.
   - Finally, merge `[3, 27, 38, 43]` with `[9, 10, 82]` to get the sorted array `[3, 9, 10, 27, 38, 43, 82]`.

Java Implementation

Here's a Java implementation of Merge Sort:

```
package com.mphasis.sorting;
public class MergeSort {

    // Method to merge two sorted sub-arrays
    private static void merge(int[] arr, int left,
int mid, int right) {
        // Find sizes of two sub-arrays
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays
        int[] L = new int[n1];
        int[] R = new int[n2];

        // Copy data to temporary arrays
        for (int i = 0; i < n1; ++i) {
            L[i] = arr[left + i];
        }
        for (int j = 0; j < n2; ++j) {
            R[j] = arr[mid + 1 + j];
        }

        // Merge the temporary arrays back into
arr[left..right]
        int i = 0; // Initial index of first sub-
array
        int j = 0; // Initial index of second sub-
array
        int k = left; // Initial index of merged sub-
array

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[], if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements of R[], if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Method to implement merge sort
    private static void mergeSort(int[] arr, int
left, int right) {
        if (left < right) {
            // Find the middle point
            int mid = (left + right) / 2;

            // Recursively sort first and second
halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the sorting algorithm
    public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Original array:");
        printArray(arr);

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

How It Works

1. **Merge Method**:

    - Merges two sorted sub-arrays into a single sorted array.
    - Uses temporary arrays to hold the sub-arrays and then merges them into the original array.

2. **Merge Sort Method**:

    - Recursively divides the array into two halves until each sub-array has one element.
    - Calls the `merge` method to combine the sorted halves.

3. **Print Method**:

    - Prints the elements of the array.

Key Points

- **Time Complexity**: O(n log n) - Merge Sort is efficient for large datasets.
- **Space Complexity**: O(n) - Merge Sort requires additional space for the temporary arrays.
- **Stability**: Merge Sort is stable - it preserves the relative order of equal elements.

Example Output

When you run the above Java program, you should see the following output:

```
Original array:
38 27 43 3 9 82 10
Sorted array:
3 9 10 27 38 43 82
```

Quick Sort

Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer approach. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Explanation

Here's how Quick Sort works:

1. **Choose a Pivot**: Select an element from the array to serve as the pivot.
2. **Partition**: Rearrange the array so that elements less than the pivot come before it, and elements greater than the pivot come after it.
3. **Recursively Apply**: Apply the same process recursively to the sub-arrays formed by the partition.

Step-by-Step Example

Consider an array `[10, 80, 30, 90, 40, 50, 70]` with the pivot as `50`:

1. **Partitioning**:

- Rearrange elements around `50` so that elements less than `50` are on the left and those greater are on the right.
- Array becomes: `[10, 30, 40, 50, 90, 80, 70]`.

2. **Recursive Sorting**:

   - Apply Quick Sort to `[10, 30, 40]` and `[90, 80, 70]`.

Java Implementation

Here's a Java implementation of Quick Sort:

```
package com.mphasis.sorting;
public class QuickSort {

    // Method to perform partitioning
    private static int partition(int[] arr, int low, int high)
{
        // Choose the pivot element
        int pivot = arr[high];
        int i = (low - 1); // Index of smaller element

        for (int j = low; j < high; j++) {
            // If current element is smaller than the pivot
            if (arr[j] < pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i + 1] and arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    // Method to implement quick sort
    private static void quickSort(int[] arr, int low, int
high) {
        if (low < high) {
            // Partition the array and get the pivot index
            int pi = partition(arr, low, high);

            // Recursively sort the sub-arrays
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the sorting algorithm
    public static void main(String[] args) {
        int[] arr = {10, 80, 30, 90, 40, 50, 70};
        System.out.println("Original array:");
        printArray(arr);

        quickSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array:");
        printArray(arr);
    }
}
```

How It Works

1. **Partition Method**:

    - Selects a pivot (usually the last element in the current sub-array).
    - Rearranges elements so that elements less than the pivot come before it, and elements greater come after it.
    - Returns the index of the pivot after partitioning.

2. **Quick Sort Method**:

    - Recursively applies Quick Sort to the sub-arrays formed by partitioning.
    - Calls `partition` to sort the elements around the pivot.

3. **Print Method**:

    - Prints the elements of the array.

Key Points

- **Time Complexity**: O(n^2) in the worst case (when the pivot is the smallest or largest element). On average, it is O(n log n), which is efficient for large datasets.
- **Space Complexity**: O(log n) on average due to the recursive stack.
- **Stability**: Quick Sort is not stable - it does not necessarily preserve the relative order of equal elements.

Example Output

When you run the above Java program, you should see the following output:

```
Original array:
10 80 30 90 40 50 70
Sorted array:
10 30 40 50 70 80 90
```

Linear Search

Linear Search is a straightforward searching algorithm used to find an element in a list. It works by checking each element in the list one by one until the desired element is found or the end of the list is reached.

Explanation

Here's how Linear Search works:

1. **Start at the Beginning**: Begin at the first element of the list.
2. **Compare**: Compare the current element with the target element.
3. **Check for Match**: If the current element matches the target, return the index of the current element.
4. **Continue**: If the current element does not match the target, move to the next element.
5. **Finish**: If the end of the list is reached and the target has not been found, return a value indicating that the target is not in the list (e.g., `-1`).

Java Implementation

Here’s a Java implementation of Linear Search:

```
package com.mphasis.sorting;
public class LinearSearch {

    // Method to perform linear search
    public static int linearSearch(int[] arr,
int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Target found,
return the index
            }
        }
        return -1; // Target not found
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the searching
algorithm
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60,
70};
        int target = 40;

        System.out.println("Array:");
        printArray(arr);

        int result = linearSearch(arr, target);
        if (result != -1) {
            System.out.println("Element " +
target + " is present at index " + result +
".");
        } else {
            System.out.println("Element " +
target + " is not present in the array.");
        }
    }
}
```

How It Works

1. **Linear Search Method**:
   - Iterates through each element of the array.
   - Compares each element with the target value.
   - Returns the index of the target if it is found.
   - Returns `-1` if the target is not found in the array.
2. **Print Method**:
   - Prints the elements of the array.

Key Points

- **Time Complexity**: O(n) - Linear Search checks each element once, so it scales linearly with the size of the list.
- **Space Complexity**: O(1) - Linear Search is an in-place algorithm that requires a constant amount of additional memory.
- **Efficiency**: Linear Search is less efficient compared to more advanced search algorithms (like Binary Search) for large datasets but is simple and works on unsorted data.

Example Output

When you run the above Java program, you should see the following output:

```
Array:
10 20 30 40 50 60 70
Element 40 is present at index 3.
```

Binary Search

Binary Search is a more efficient search algorithm than Linear Search, but it requires that the array be sorted before searching. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty.

Explanation

Here's how Binary Search works:

1. **Initialize**: Start with two pointers, `low` and `high`, representing the bounds of the search interval. Initially, `low` is set to the index of the first element, and `high` is set to the index of the last element.
2. **Calculate Middle**: Compute the middle index of the current interval.
3. **Compare**:
   - If the target value is equal to the middle element, the search is successful, and the index of the middle element is returned.
   - If the target value is less than the middle element, adjust the `high` pointer to be `mid - 1` to search in the left half.
   - If the target value is greater than the middle element, adjust the `low` pointer to be `mid + 1` to search in the right half.

4. **Repeat**: Repeat the process until the `low` pointer is greater than the `high` pointer.
5. **Not Found**: If the interval is empty, return `-1` indicating that the target value is not in the array.

Java Implementation

Here's a Java implementation of Binary Search:

```java
package com.mphasis.sorting;
public class BinarySearch {

    // Method to perform binary search
    public static int binarySearch(int[] arr, int 
target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if target is present at mid
            if (arr[mid] == target) {
                return mid;
            }

            // If target is greater, ignore the left 
half
            if (arr[mid] < target) {
                low = mid + 1;
            }
            // If target is smaller, ignore the right 
half
            else {
                high = mid - 1;
            }
        }

        // Target not found
        return -1;
    }

    // Method to print the array
    public static void printArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    // Main method to test the searching algorithm
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50, 60, 70};
        int target = 40;

        System.out.println("Array:");
        printArray(arr);

        int result = binarySearch(arr, target);
        if (result != -1) {
            System.out.println("Element " + target +
" is present at index " + result + ".");
        } else {
            System.out.println("Element " + target +
" is not present in the array.");
        }
    }
}
```

How It Works

1. **Binary Search Method**:

   - Initializes `low` and `high` pointers.
   - Computes the middle index.
   - Compares the middle element with the target.
   - Adjusts `low` and `high` based on whether the target is less than or greater than the middle element.
   - Continues until the target is found or the search interval is empty.

2. **Print Method**:

   - Prints the elements of the array.

Key Points

- **Time Complexity**: **O(log n)** - Binary Search is much faster than Linear Search for large datasets, as it halves the search space with each step.

- **Space Complexity**: **O(1)** - Binary Search requires a constant amount of additional memory.
- **Requirements**: The array must be sorted before performing Binary Search.

Example Output

When you run the above Java program, you should see the following output:

```
Array:
10 20 30 40 50 60 70
Element 40 is present at index 3.
```