

**FACULTY - DR. RANJNA JAIN**

Practical File

**6th Semester (**Academic Year 2024 - 2025**)**

**SUBJECT NAME: ANALYSIS & DESIGN OF ALGORITHMS**

**SUBJECT CODE: CSH204B-P**

Submitted By:- Submitted To:-

**Student Name:Tamanna Faculty Name: Dr. Ranjna Jain Roll No.: 2K22CSUN01038 Designation: Lab Coordinator Branch: BTech CSE**

**Section: 6A**

# Concept:

**Bubble sort** is a simple sorting algorithm. This sorting algorithm is **comparison-based algorithm** in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of **O(n2)** where n is the number of items.

1. The array is sorted through multiple iterations. In each iteration, the largest unsorted element moves to its correct position at the end. After the first pass, the largest element is in place; after the second pass, the second-largest is positioned correctly, and so on.
2. With each pass, fewer elements need to be considered because the largest elements have already settled at the end. After kk passes, the kk largest elements are correctly positioned.
3. During each pass, adjacent elements are compared, and if the preceding element is greater than the next one, they are swapped. This ensures that the largest element among the unsorted portion moves to its correct position.
4. The process continues until a full pass occurs without any swaps, which indicates that the array is completely sorted.

# Algorithm:

bubbleSort(A): n = length(A)

for i from 0 to n-1:

for j from 0 to n-i-2: if A[j] > A[j+1]:

swap A[j] and A[j+1]

# Java Code:

class BubbleSort {

static void bubbleSort(int arr[]) { int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

int arr[] = {95, 34, 10, 12, 35, 85, 100};

System.*out*.println("Unsorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

*bubbleSort*(arr);

System.*out*.println("\nSorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

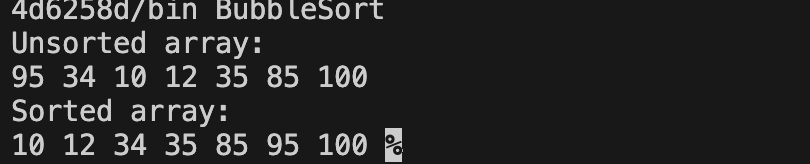
}

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(n) (Already Sorted Array)
* Worst Case Time Complexity: O(n²)
* Space Complexity: O(1)

# Output:

****

**Concept:**

**Selection Sort** is a simple sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the first unsorted element. It is an ****in-place sorting algorithm**** with a time complexity of ****O(n²)**,** making it inefficient for large datasets but useful when minimizing swaps is important.

1. Divide the array into two parts: a sorted part (initially empty) and an unsorted part (initially the entire array).
2. Find the minimum element in the unsorted part and swap it with the first element of the unsorted portion. Now, the sorted part has one element.
3. Move the boundary of the sorted section one step forward and repeat the process for the remaining unsorted portion.
4. Continue this process until all elements are placed in their correct positions, resulting in a sorted array.

# Algorithm:

selectionSort(A): n = length(A)

for i from 0 to n-1: minIndex = i

for j from i+1 to n:

if A[j] < A[minIndex]: minIndex = j

swap A[minIndex] and A[i]

# Java Code:

class SelectionSort {

static void selectionSort(int arr[]) { int n = arr.length;

for (int i = 0; i < n - 1; i++) { int minIndex = i;

for (int j = i + 1; j < n; j++) { if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

int temp = arr[minIndex]; arr[minIndex] = arr[i]; arr[i] = temp;

}

}

int arr[] = {95, 34, 10, 12, 35, 85, 100};

System.*out*.println("Unsorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

*selectionSort*(arr);

System.*out*.println("\nSorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

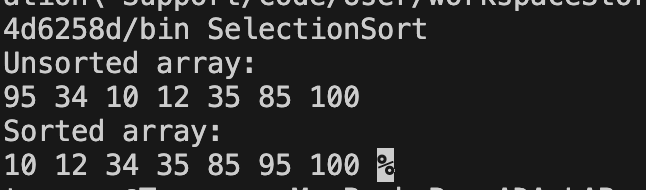
}

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(n2)
* Worst Case Time Complexity: O(n²)
* Space Complexity: O(1)

# Output:

****

**Concept:**

**Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

1. We start with second element of the array as first element in the array is assumed to be sorted.
2. Compare second element with the first element and check if the second element is smaller then swap them.
3. Move to the third element and compare it with the first two elements and put at its correct position
4. Repeat until the entire array is sorted.

# Algorithm:

insertionSort(A): n = length(A)

for i from 1 to n-1: key = A[i]

j = i - 1

while j >= 0 and A[j] > key: A[j+1] = A[j]

j = j - 1 A[j+1] = key

# Java Code:

class InsertionSort {

static void insertionSort(int arr[]) { int n = arr.length;

for (int i = 1; i < n; i++) { int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

public static void main(String args[]) {

int arr[] = {95, 34, 10, 12, 35, 85, 100};

System.*out*.println("Unsorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

*insertionSort*(arr);

System.*out*.println("\nSorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

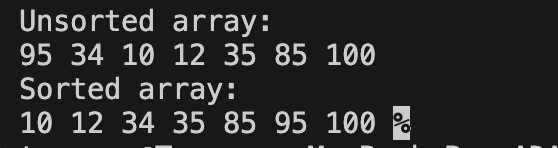
}

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(n) (Already Sorted Array)
* Worst Case Time Complexity: O(n²)
* Space Complexity: O(1)

# Output:

****

**Concept:**

**Quick Sort** works on the principle of divide and conquer, breaking down the problem into smaller sub- problems.

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array**: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

# Algorithm:

quickSort(A, low, high): if low < high:

pivotIndex = partition(A, low, high) quickSort(A, low, pivotIndex - 1) quickSort(A, pivotIndex + 1, high)

partition(A, low, high): pivot = A[high]

i = low - 1

for j from low to high-1: if A[j] < pivot:

i = i + 1

swap A[i] and A[j] swap A[i + 1] and A[high] return i + 1

# Java Code:

class QuickSort {

static void quickSort(int arr[], int low, int high) { if (low < high) {

int pivot = *partition*(arr, low, high); *quickSort*(arr, low, pivot - 1); *quickSort*(arr, pivot + 1, high);

}

}

static int partition(int arr[], int low, int high) { int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) { if (arr[j] < pivot) {

i++;

int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;

}

}

int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp; return i + 1;

}

public static void main(String args[]) { int arr[] = {95, 34, 10, 12, 35, 85, 100};

System.*out*.println("Unsorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

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

System.*out*.println("\nSorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

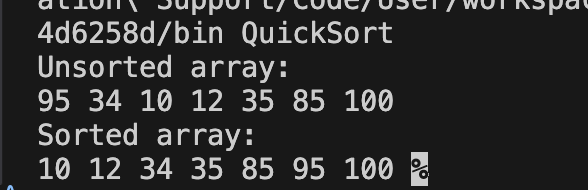
}

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(n log n)
* Worst Case Time Complexity: O(n²)
* Best Case Space Complexity: O(log n)
* Worst Case Space Complexity: O(n)

# Output:

****

**Concept:**

**Merge Sort** is an efficient and stable sorting algorithm that follows the divide-and-conquer approach. It recursively breaks down an array into smaller subarrays, sorts them, and then merges them back in sorted order.

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

# Algorithm:

mergeSort(A, left, right): if left < right:

mid = (left + right) / 2 mergeSort(A, left, mid) mergeSort(A, mid + 1, right) merge(A, left, mid, right)

merge(A, left, mid, right):

create leftArray and rightArray

merge leftArray and rightArray back into A

# Java Code:

class MergeSort {

static void mergeSort(int arr[], int left, int right) { if (left < right) {

int mid = left + (right - left) / 2; *mergeSort*(arr, left, mid); *mergeSort*(arr, mid + 1, right); *merge*(arr, left, mid, right);

}

}

static void merge(int arr[], int left, int mid, int right) { int n1 = mid - left + 1;

int n2 = right - mid;

int leftArray[] = new int[n1]; int rightArray[] = new int[n2];

for (int i = 0; i < n1; i++) leftArray[i] = arr[left + i];

for (int j = 0; j < n2; j++) rightArray[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left; while (i < n1 && j < n2) {

if (leftArray[i] <= rightArray[j]) { arr[k] = leftArray[i];

i++;

} else {

arr[k] = rightArray[j]; j++;

} k++;

}

while (i < n1) {

arr[k] = leftArray[i]; i++;

k++;

}

while (j < n2) {

arr[k] = rightArray[j]; j++;

k++;

}

}

public static void main(String args[]) { int arr[] = {45, 34, 10, 8, 35, 12,170};

System.*out*.println("Unsorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

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

System.*out*.println("\nSorted array:"); for (int num : arr) {

System.*out*.print(num + " ");

}

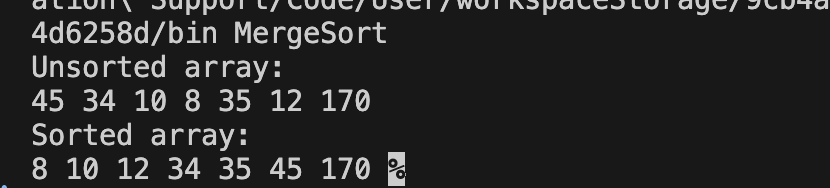
}

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(n log n)
* Worst Case Time Complexity: O(n log n)
* Space Complexity: O(n)

# Output:

****

**Concept:**

**Linear Search** is a simple searching algorithm used to find an element in an array or list. It works by checking each element one by one until the desired element is found or the list ends.

# Algorithm:

linearSearch(A, target):

for i from 0 to length(A) - 1: if A[i] == target:

return i return -1

# Java Code:

class LinearSearch {

static int linearSearch(int arr[], int target) { for (int i = 0; i < arr.length; i++) {

if (arr[i] == target) { return i;

}

}

return -1;

}

public static void main(String args[]) { int arr[] = {10, 20, 30, 40, 50};

int target = 30;

int result = *linearSearch*(arr, target); if (result == -1)

System.*out*.println("Element not found");

result);

}

else

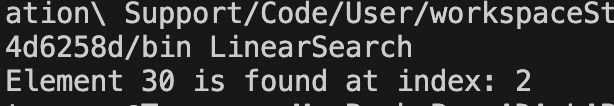
System.*out*.println("Element " + target + " is found at index: " +

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(1)
* Worst Case Time Complexity: O(n)
* Space Complexity: O(1)

# Output:

****

**Concept:**

**Binary Search** Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

1. Divide the search space into two halves by finding the middle index “mid”.
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
   1. If the key is smaller than the middle element, then the left side is used for next search.
   2. If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

# Algorithm:

binarySearch(A, target, low, high): while low <= high:

mid = (low + high) / 2 if A[mid] == target:

return mid

else if A[mid] < target: low = mid + 1

else:

high = mid - 1 return -1

# Java Code:

class BinarySearch {

static int binarySearch(int arr[], int target, int low, int high) { while (low <= high) {

int mid = low + (high - low) / 2; if (arr[mid] == target)

return mid;

else if (arr[mid] < target) low = mid + 1;

else

high = mid - 1;

}

return -1;

}

public static void main(String args[]) { int arr[] = {12, 20, 70, 68, 35};

int target = 68;

int result = *binarySearch*(arr, target, 0, arr.length - 1);

if (result == -1) System.*out*.println("Element not found");

result);

}

else

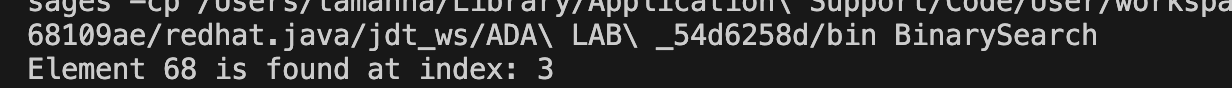
System.*out*.println("Element " + target +" is found at index: " +

}

# Analysis (Time and Space Complexity):

* Best Case Time Complexity: O(1)
* Worst Case Time Complexity: O(log n)
* Space Complexity: O(1)

**Output:**

****