**PROJECT REPORT**

**NAME: RAVITEJ KRISHNA MURTHY**

**STUDENT NUMBER: 1002198790**

**PROJECT-01(SORTING ALGORITHMS)**

**Implement and compare the following sorting algorithm:**

* **Merge Sort**
* **Heap Sort**
* **Quick Sort (Regular quick sort\* and quick sort using 3 medians)**
* **Insertion Sort**
* **Selection Sort**
* **Bubble Sort**

**Merge Sort:**

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach to sort an array or list of elements. The main idea behind merge sort is to divide the array into smaller subarrays, sort each subarray recursively, and then merge the sorted subarrays to produce the final sorted array.

**Pseudocode:**

function mergeSort(arr[], l, r)

if l < r then

middle = (l + r) / 2

// Call mergeSort for the first half

mergeSort(arr, l, middle)

// Call mergeSort for the second half

mergeSort(arr, middle + 1, r)

// Merge the two halves sorted in step 2 and step 3

merge(arr, l, middle, r)

end function

function merge(arr[], l, middle, r)

// Find sizes of two subarrays to be merged

n1 = middle - l + 1

n2 = r - middle

// Create temporary arrays

L[n1], R[n2]

// Copy data to temporary arrays L[] and R[]

for i = 0 to n1 - 1 do

L[i] = arr[l + i]

for j = 0 to n2 - 1 do

R[j] = arr[middle + 1 + j]

// Merge the temporary arrays back into arr[l..r]

i = 0 // Initial index of first subarray

j = 0 // Initial index of second subarray

k = l // Initial index of merged subarray

while i < n1 and j < n2 do

if L[i] <= R[j] then

arr[k] = L[i]

i++

else

arr[k] = R[j]

j++

k++

// Copy the remaining elements of L[], if any

while i < n1 do

arr[k] = L[i]

i++

k++

// Copy the remaining elements of R[], if any

while j < n2 do

arr[k] = R[j]

j++

k++

end function

**Heap Sort:**

Heap sort is an efficient comparison-based sorting algorithm that uses the heap data structure to sort elements in ascending or descending order. It starts by building a max heap from the input array, then repeatedly extracts the maximum element from the heap and places it at the end of the array, reducing the size of the heap each time until all elements are sorted.

**Pseudocode:**

function heapSort(arr[])

n = length of arr

// Build a max heap from the input array

for i = n/2 - 1 to 0 do

heapify(arr, n, i)

// One by one extract elements from the heap

for i = n - 1 to 0 do

// Swap the root (maximum element) with the last element

swap(arr[0], arr[i])

// Reduce the size of heap and heapify the root

heapify(arr, i, 0)

end function

function heapify(arr[], n, i)

largest = i

left = 2\*i + 1

right = 2\*i + 2

// If left child is larger than root

if left < n and arr[left] > arr[largest] then

largest = left

// If right child is larger than largest so far

if right < n and arr[right] > arr[largest] then

largest = right

// If largest is not root

if largest != i then

swap(arr[i], arr[largest])

heapify(arr, n, largest)

end function

function swap(arr[], i, j)

temp = arr[i]

arr[i] = arr[j]

arr[j] = temp

end function

**Quick Sort (Regular):**

Quicksort is a highly efficient sorting algorithm based on the divide-and-conquer approach. It works by selecting a pivot element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot. The subarrays are then recursively sorted independently.

**Pseudocode:**

function quickSort(arr[], low, high)

if low < high then

// Partition the array and get the pivot index

pivotIndex = partition(arr, low, high)

// Recursively sort elements before and after the pivot

quickSort(arr, low, pivotIndex - 1)

quickSort(arr, pivotIndex + 1, high)

end function

function partition(arr[], low, high)

// Selecting the last element as the pivot

pivot = arr[high]

i = low - 1

for j = low to high - 1 do

// If current element is smaller than or equal to pivot

if arr[j] <= pivot then

// Increment index of smaller element

i = i + 1

// Swap arr[i] and arr[j]

swap(arr, i, j)

// Swap arr[i+1] and arr[high] (pivot)

swap(arr, i + 1, high)

return i + 1

end function

function swap(arr[], i, j)

temp = arr[i]

arr[i] = arr[j]

arr[j] = temp

end function

**Quick Sort (3-Median):**

The quicksort algorithm with the 3-median technique aims to improve the pivot selection process by choosing a pivot that is closer to the median of the array, resulting in more balanced partitions and better performance, especially for nearly sorted or partially sorted arrays.

**Pseudocode:**

function quickSort(arr: array of elements, low: integer, high: integer)

if low < high then

// Finding the pivot using median of three technique

pivot = medianOfThree(arr, low, high)

// Partition the array around the pivot

pi = partition(arr, low, high, pivot)

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1)

quickSort(arr, pi + 1, high)

end if

end function

function medianOfThree(arr: array of elements, low: integer, high: integer) -> integer

mid = low + (high - low) / 2

// Compare and swap elements to ensure the middle element is the median

if arr[low] > arr[mid] then

swap(arr, low, mid)

if arr[mid] > arr[high] then

swap(arr, mid, high)

if arr[low] > arr[mid] then

swap(arr, low, mid)

return arr[mid]

end function

function partition(arr: array of elements, low: integer, high: integer, pivot: integer) -> integer

i = low - 1

for j = low to high - 1 do

if arr[j] <= pivot then

i = i + 1

swap(arr, i, j)

swap(arr, i + 1, high)

return i + 1

end function

function swap(arr: array of elements, i: integer, j: integer)

temp = arr[i]

arr[i] = arr[j]

arr[j] = temp

end function

**Insertion Sort:**

Insertion sort is a simple sorting algorithm that builds the final sorted array one element at a time. It iterates through the array, moving each element into its correct position relative to the elements before it.

**Pseudocode:**

// Function to perform insertion sort on an array

function insertionSort(arr[])

n = length of arr

// Iterate through each element of the array

for i = 1 to n - 1 do

// Set the current element as the key

key = arr[i]

// Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position

j = i - 1

while j >= 0 and arr[j] > key do

arr[j + 1] = arr[j]

j = j - 1

arr[j + 1] = key

end function

**Selection Sort:**

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 element at the beginning of the unsorted part. This process continues until the entire array is sorted.

**Pseudocode:**

function selectionSort(arr[])

n = length of arr

// Iterate through each element of the array

for i = 0 to n - 2 do

// Find the index of the minimum element in the unsorted part of the array

min = i

for j = i + 1 to n - 1 do

if arr[j] < arr[min] then

min = j

// Swap the minimum element with the current element

temp = arr[min]

arr[min] = arr[i]

arr[i] = temp

end function

**Bubble Sort:**

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 no swaps are needed, which indicates that the list is sorted.

**Pseudocode:**

function bubbleSort(arr[])

n = length of arr

// Iterate through each element of the array

for i = 0 to n - 2 do

// Iterate through each element of the array from the beginning to the end of the unsorted part

for j = 0 to n - i - 2 do

// If the current element is greater than the next element, swap them

if arr[j] > arr[j + 1] then

swap(arr[j], arr[j + 1])

end function

**Code Explanation:**

All the above functions are being called from the main method of the Main.java file to execute the corresponding sorting function. The resultant time taken by individual Sorting algorithm is calculated and plotted in a bar graph with the help of GUI. The input size is taken from the user through the GUI. The GUI is implemented using JavaFX. There are multiple methods in Main.java where individual Sorting Algorithms are being called with the help of Object of the class of the individual sorting algorithms.

**RESULTS:**

**Let’s consider different input sizes:**

* N = 1000

Execution of Merge Sort has taken :0.5 ms

Execution of Heap Sort has taken :0.6 ms

Execution of Quick Sort has taken :0.38 ms

Execution of Quick Sort(3 Median) has taken :0.38 ms

Execution of Insertion Sort has taken :1.94 ms

Execution of Selection Sort has taken :1.98 ms

Execution of Bubble Sort has taken :1.18 ms

A screenshot of a graph

Description automatically generated

* N = 5000

Execution of Merge Sort has taken :1.13 ms

Execution of Heap Sort has taken :1.92 ms

Execution of Quick Sort has taken :0.7 ms

Execution of Quick Sort(3 Median) has taken :0.8 ms

Execution of Insertion Sort has taken :3.43 ms

Execution of Selection Sort has taken :5.9 ms

Execution of Bubble Sort has taken :4.76 ms

A screen shot of a graph

Description automatically generated

* N = 9000

Execution of Merge Sort has taken :1.96 ms

Execution of Heap Sort has taken :1.04 ms

Execution of Quick Sort has taken :0.48 ms

Execution of Quick Sort(3 Median) has taken :0.9 ms

Execution of Insertion Sort has taken :5.42 ms

Execution of Selection Sort has taken :15.52 ms

Execution of Bubble Sort has taken :15.85 ms

A screenshot of a graph

Description automatically generated

* N = 100

Execution of Merge Sort has taken :0.18 ms

Execution of Heap Sort has taken :0.04 ms

Execution of Quick Sort has taken :0.09 ms

Execution of Quick Sort(3 Median) has taken :0.11 ms

Execution of Insertion Sort has taken :0.15 ms

Execution of Selection Sort has taken :0.22 ms

Execution of Bubble Sort has taken :0.22 ms

A screenshot of a computer

Description automatically generated

* N = 500

Execution of Merge Sort has taken :0.05 ms

Execution of Heap Sort has taken :0.05 ms

Execution of Quick Sort has taken :0.04 ms

Execution of Quick Sort(3 Median) has taken :0.04 ms

Execution of Insertion Sort has taken :0.16 ms

Execution of Selection Sort has taken :0.07 ms

Execution of Bubble Sort has taken :0.07 ms

A screenshot of a computer

Description automatically generated

**CONCLUSION:**

We can say that for a large array size Insertion, Selection and bubble sort are not preferrable as they take a long time compared to quick, heap or merge sort. Quick sort using 3 Median almost does the sorting at the same time as regular quick sort algorithm.

For smaller array sizes the time taken for sorting is relatively closer for all the algorithms.