|  |  |
| --- | --- |
| https://lh6.googleusercontent.com/NfxikauFutj0NGA6tZLo-9OYuuo3IKL8zEJJ1vcz211S75EzE4cV8OiG_9xiNDI7UXEpwJU3L0HucAlThGrGrN0dBZNqE1lpoGJHT3oGyRmOalINq0YRgVxjjMgLbZ3n8Pq0wpa0thnjQcjh_kKTHUlnU-ExSfeM05HoXY1-s1p-Qtg0OyqjFjrwB05Nj4mxriktv4A | SVKM’s NMIMS  School of Technology Management & Engineering Navi Mumbai Campus |
| Department of Computer Engineering |

|  |  |
| --- | --- |
| Name: Yash Patil | SAP ID: 70472200140 |
| Semester: III | Year: II |
| Subject: Data Structures and Algorithm | Roll No.: A176 |
| Practical: 11 | Batch: 1 |

**Aim:–**

Implementation of any two sorting algorithm from Selection Sort, Merge sort and Insertion sort

**Theory:–**

Selection Sort and Merge Sort are two common sorting algorithms, each with its own approach and efficiency characteristics.

Selection Sort:

- Algorithm: Selection Sort is an in-place comparison-based sorting algorithm. It divides the input array into two parts: the sorted part on the left and the unsorted part on the right. In each iteration, it finds the minimum element from the unsorted part and swaps it with the first element in the unsorted part. This process is repeated until the entire array is sorted.

- Efficiency: Selection Sort has a time complexity of O(n^2) in the worst and average cases, where 'n' is the number of elements in the array. It performs a constant number of swaps, which is why it is considered an inefficient sorting algorithm for large datasets.

Merge Sort:

- Algorithm: Merge Sort is a divide-and-conquer sorting algorithm that recursively divides the input array into two halves, sorts each half, and then merges the sorted halves back together. The merge step combines two sorted arrays into a single sorted array.

- Efficiency: Merge Sort has a time complexity of O(n log n) in all cases, making it much more efficient than Selection Sort for large datasets. It consistently performs well regardless of the initial order of the elements. However, it requires additional memory space for the merging step, which can be a drawback for very large datasets.

Key Differences:

1. Approach: Selection Sort is an in-place sorting algorithm that repeatedly selects the minimum element from the unsorted part and places it at the beginning of the sorted part. Merge Sort, on the other hand, is a divide-and-conquer algorithm that divides the array into halves and then combines the sorted halves to produce a fully sorted array.

2. Time Complexity: Selection Sort has a time complexity of O(n^2), which makes it less efficient for large datasets. Merge Sort has a time complexity of O(n log n), which is more efficient, especially for large datasets.

3. Stability: Selection Sort is not a stable sorting algorithm, meaning the relative order of equal elements may not be preserved. Merge Sort is a stable sorting algorithm, which means it maintains the relative order of equal elements.

4. Memory Usage: Selection Sort is an in-place sorting algorithm, meaning it doesn't require additional memory space. Merge Sort, however, requires additional memory space for the merging step, which can be a drawback for very large datasets.

**Code/Implementation –**

#include <stdio.h>

void selectionSort(int arr[], int n) {

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

int min\_index = i;

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

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

}

// Swap arr[i] and arr[min\_index]

int temp = arr[i];

arr[i] = arr[min\_index];

arr[min\_index] = temp;

}

}

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) {

L[i] = arr[left + i];

}

for (int i = 0; i < n2; i++) {

R[i] = arr[mid + 1 + i];

}

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

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);

}

}

int main() {

int n, choice;

printf("Enter the number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter the elements:\n");

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

scanf("%d", &arr[i]);

}

do {

printf("\nMenu:\n");

printf("1. Selection Sort\n");

printf("2. Merge Sort\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

selectionSort(arr, n);

printf("Sorted using Selection Sort:\n");

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

printf("%d ", arr[i]);

}

printf("\n");

break;

case 2:

mergeSort(arr, 0, n - 1);

printf("Sorted using Merge Sort:\n");

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

printf("%d ", arr[i]);

}

printf("\n");

break;

case 3:

printf("Exiting the program.\n");

break;

default:

printf("Invalid choice. Please enter a valid option.\n");

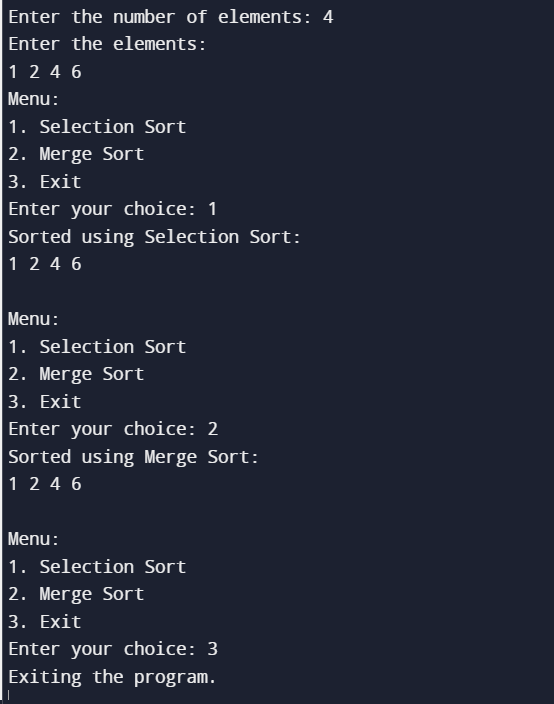
}

} while (choice != 3);

return 0;

}

**Output:-**



**Conclusion:-**

*In conclusion, Selection Sort is a straightforward but inefficient sorting algorithm that works well for small datasets but has a time complexity of O(n^2), making it impractical for larger datasets. On the other hand, Merge Sort is a highly efficient sorting algorithm with a consistent time complexity of O(n log n) for all cases. Its divide-and-conquer approach and stability make it a preferred choice for larger datasets, though it does require additional memory for merging. The choice between the two depends on the specific requirements and constraints of the sorting task, with Merge Sort being a more scalable and efficient option for most scenarios.*