**EXPERIMENT NO. 9**

| **Objective(s):**  Implement and analyze various sorting algorithms to efficiently arrange elements in ascending or descending order. |
| --- |
| **Outcome:**  Develop functions for Bubble sort, Selection sort, Insertion sort, Quick sort, and Merge sort that accurately sort input arrays, demonstrating their respective time complexities and performance characteristics in sorting operations. |
| **Problem Statement:**  Implement Bubble sort, selection sort, Insertion sort, quick sort ,merge sort. |
| **Background Study:** 1. Bubble Sort **Objective:** Sort an array of elements by repeatedly swapping adjacent elements if they are in the wrong order. How Bubble Sort Works  1. **Iteration through Array:**    * Pass through the array multiple times.    * Compare each pair of adjacent elements.    * Swap them if they are in the wrong order (smaller follows larger). 2. **Passes:**    * After each pass, the largest unsorted element reaches its correct position at the end of the array.    * Reduce the number of comparisons in subsequent passes. 3. **Complexity:**    * **Time Complexity:** $(O(n^2))$ in the worst and average cases.    * **Space Complexity:**$(O(1))$ (in-place sorting).    2. Selection Sort **Objective:** Sort an array by repeatedly finding the minimum element (ascending order) and swapping it with the first unsorted element. How Selection Sort Works  1. **Iteration through Array:**    * Divide the array into two subarrays: sorted and unsorted.    * Find the smallest element from the unsorted subarray.    * Swap it with the first element of the unsorted subarray. 2. **Repeat:**    * Expand the sorted subarray by one element.    * Continue until the entire array is sorted. 3. **Complexity:**    * **Time Complexity:** $(O(n^2))$ in all cases (worst, average, and best).    * **Space Complexity:** $(O(1))$ (in-place sorting).    3. Insertion Sort **Objective:** Sort an array by inserting each element into its correct position in a growing sorted subarray. How Insertion Sort Works  1. **Iteration through Array:**    * Divide the array into sorted and unsorted subarrays.    * Pick an element from the unsorted subarray and insert it into its correct position in the sorted subarray. 2. **Shifting Elements:**    * Shift larger elements one position to the right to make space for the inserted element.    * Continue until all elements are sorted. 3. **Complexity:**    * **Time Complexity:** (O(n^2)) in the worst and average cases, \(O(n)\) in the best case (already sorted).    * **Space Complexity:** \(O(1)\) (in-place sorting).    4. Quick Sort **Objective:** Sort an array using a divide-and-conquer approach by recursively partitioning around a pivot element. How Quick Sort Works  1. **Partitioning:**    * Choose a pivot element (typically the last element).    * Reorder the array such that all elements less than the pivot are before it, and all greater elements are after it. 2. **Recursion:**    * Recursively apply the same partitioning to the subarrays formed by the pivot.    * Sort the subarrays until the entire array is sorted. 3. **Complexity:**    * **Time Complexity:** \(O(n \log n)\) on average, \(O(n^2)\) in the worst case (unbalanced partitions).    * **Space Complexity:** \(O(\log n)\) to \(O(n)\) depending on the implementation (typically \(O(\log n)\) due to recursion stack).    5. Merge Sort **Objective:** Sort an array using a divide-and-conquer approach by recursively splitting the array into halves, sorting each half, and merging them back together. How Merge Sort Works  1. **Divide:**    * Divide the array into two halves recursively until each subarray contains one element (base case). 2. **Merge:**    * Merge two sorted subarrays into a single sorted array by comparing elements one by one. 3. **Complexity:**    * **Time Complexity:** \(O(n \log n)\) in all cases (worst, average, and best).    * **Space Complexity:** \(O(n)\) due to the auxiliary array used for merging.    Comparison of Sorting Algorithms  * **Bubble Sort, Selection Sort, and Insertion Sort** have $(O(n^2))$ time complexity, making them inefficient for large datasets compared to $(O(n \log n))$ algorithms like Quick Sort and Merge Sort. * **Quick Sort** is typically faster than **Merge Sort** due to better cache locality and in-place partitioning, but can degrade to $(O(n^2))$ if not implemented carefully. * **Merge Sort** is stable and guarantees $(O(n \log n))$ performance but requires additional space for merging. |

| **Algorithm (Student Work Area):**  1. Bubble Sort **Objective:** Sort an array of elements by repeatedly swapping adjacent elements if they are in the wrong order.  **Algorithm:**   1. Start from the beginning of the array. 2. Compare each pair of adjacent elements. 3. If the elements are in the wrong order (larger followed by smaller for ascending order), swap them. 4. Repeat steps 1-3 for each pair of adjacent elements until no more swaps are needed in a pass.   2. Selection Sort **Objective:** Sort an array by repeatedly finding the minimum element and swapping it with the first unsorted element.  **Algorithm:**   1. Iterate through the array from the beginning. 2. Assume the current element is the minimum. 3. Compare it with each subsequent element to find the actual minimum. 4. Swap the current element with the minimum element found. 5. Repeat steps 1-4, expanding the sorted portion of the array by one element each time, until the entire array is sorted.   3. Insertion Sort **Objective:** Sort an array by inserting each element into its correct position in a growing sorted subarray.  **Algorithm:**   1. Start from the second element of the array. 2. Compare the current element with elements in the sorted subarray (to its left). 3. Shift all larger elements one position to the right. 4. Insert the current element into its correct position in the sorted subarray. 5. Repeat steps 1-4 for each subsequent element until the entire array is sorted.   4. Quick Sort **Objective:** Sort an array using a divide-and-conquer approach by recursively partitioning around a pivot element.  **Algorithm:**   1. Choose a pivot element from the array (usually the last element). 2. Partition the array into two subarrays:    * Elements less than the pivot (left subarray).    * Elements greater than or equal to the pivot (right subarray). 3. Recursively apply the same partitioning process to the left and right subarrays. 4. Concatenate the sorted left subarray, pivot, and sorted right subarray to get the sorted array.  5. Merge Sort **Objective:** Sort an array using a divide-and-conquer approach by recursively splitting the array into halves, sorting each half, and merging them back together.  **Algorithm:**   1. Divide the array into two halves. 2. Recursively sort each half by applying merge sort. 3. Merge the two sorted halves into a single sorted array:    * Compare elements from the two halves.    * Place the smaller element in the merged array.    * Move to the next element in the respective subarray. 4. Repeat step 3 until all elements are merged into a single sorted array. |
| --- |

| **Code:**  **Bubble Sort**  #include <stdio.h>  void bubbleSort(int array[], int size) {  for (int i = 0; i < size - 1; ++i) {  // Flag to optimize by stopping if no swap is made  int swapped = 0;  for (int j = 0; j < size - 1 - i; ++j) {  // Swap if current element is greater than next element  if (array[j] > array[j + 1]) {  int temp = array[j];  array[j] = array[j + 1];  array[j + 1] = temp;  swapped = 1;  }  }  // If no elements were swapped, array is sorted  if (swapped == 0) {  break;  }  }  }  // Example usage  int main() {  int array[] = {64, 25, 12, 22, 11};  int size = sizeof(array) / sizeof(array[0]);  bubbleSort(array, size);  printf("Sorted array: ");  for (int i = 0; i < size; ++i) {  printf("%d ", array[i]);  }  printf("\\n");  return 0;  }  **Selection Sort**  #include <stdio.h>  void selectionSort(int array[], int size) {  for (int i = 0; i < size - 1; ++i) {  int minIndex = i;  // Find the index of the minimum element in the remaining unsorted array  for (int j = i + 1; j < size; ++j) {  if (array[j] < array[minIndex]) {  minIndex = j;  }  }  // Swap the found minimum element with the first element of the unsorted array  int temp = array[minIndex];  array[minIndex] = array[i];  array[i] = temp;  }  }  // Example usage  int main() {  int array[] = {64, 25, 12, 22, 11};  int size = sizeof(array) / sizeof(array[0]);  selectionSort(array, size);  printf("Sorted array: ");  for (int i = 0; i < size; ++i) {  printf("%d ", array[i]);  }  printf("\\n");  return 0;  }  **Insertion Sort**  #include <stdio.h>  void insertionSort(int array[], int size) {  for (int i = 1; i < size; ++i) {  int key = array[i];  int j = i - 1;  // Move elements of array[0..i-1], that are greater than key, to one position ahead of their current position  while (j >= 0 && array[j] > key) {  array[j + 1] = array[j];  j--;  }  array[j + 1] = key;  }  }  // Example usage  int main() {  int array[] = {64, 25, 12, 22, 11};  int size = sizeof(array) / sizeof(array[0]);  insertionSort(array, size);  printf("Sorted array: ");  for (int i = 0; i < size; ++i) {  printf("%d ", array[i]);  }  printf("\\n");  return 0;  }  **Quick Sort**  #include <stdio.h>  void swap(int\* a, int\* b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  int partition(int array[], int low, int high) {  int pivot = array[high]; // pivot  int i = (low - 1); // Index of smaller element  for (int j = low; j <= high - 1; j++) {  // If current element is smaller than or equal to pivot  if (array[j] <= pivot) {  i++; // increment index of smaller element  swap(&array[i], &array[j]);  }  }  swap(&array[i + 1], &array[high]);  return (i + 1);  }  void quickSort(int array[], int low, int high) {  if (low < high) {  // pi is partitioning index, array[p] is now at right place  int pi = partition(array, low, high);  // Separately sort elements before partition and after partition  quickSort(array, low, pi - 1);  quickSort(array, pi + 1, high);  }  }  // Example usage  int main() {  int array[] = {64, 25, 12, 22, 11};  int size = sizeof(array) / sizeof(array[0]);  quickSort(array, 0, size - 1);  printf("Sorted array: ");  for (int i = 0; i < size; ++i) {  printf("%d ", array[i]);  }  printf("\\n");  return 0;  }  **Merge Sort**  #include <stdio.h>  void merge(int array[], int left, int mid, int right) {  int i, j, k;  int n1 = mid - left + 1;  int n2 = right - mid;  // Create temporary arrays  int L[n1], R[n2];  // Copy data to temporary arrays L[] and R[]  for (i = 0; i < n1; i++)  L[i] = array[left + i];  for (j = 0; j < n2; j++)  R[j] = array[mid + 1 + j];  // Merge the temporary arrays back into array[left..right]  i = 0; // Initial index of first subarray  j = 0; // Initial index of second subarray  k = left; // Initial index of merged subarray  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  array[k] = L[i];  i++;  } else {  array[k] = R[j];  j++;  }  k++;  }  // Copy the remaining elements of L[], if any  while (i < n1) {  array[k] = L[i];  i++;  k++;  }  // Copy the remaining elements of R[], if any  while (j < n2) {  array[k] = R[j];  j++;  k++;  }  }  void mergeSort(int array[], int left, int right) {  if (left < right) {  // Same as (left + right) / 2, but avoids overflow for large left and right  int mid = left + (right - left) / 2;  // Sort first and second halves  mergeSort(array, left, mid);  mergeSort(array, mid + 1, right);  // Merge the sorted halves  merge(array, left, mid, right);  }  }  // Example usage  int main() {  int array[] = {64, 25, 12, 22, 11};  int size = sizeof(array) / sizeof(array[0]);  mergeSort(array, 0, size - 1);  printf("Sorted array: ");  for (int i = 0; i < size; ++i) {  printf("%d ", array[i]);  }  printf("\\n");  return 0;  } |
| --- |
| **OUTPUT :** |