Course Code: CL-2001	Course : Data Structures - Lab
Instructor(s) :	Abeer Gauher, Sobia Iftikhar

LAB - 6

Implementation of Sorting Algorithms

Different Sorting Algorithms

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quicksort
6. Counting Sort
7. Radix Sort
8. Bucket Sort
9. Heap Sort

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

Insertion Sort

Insertion sort is a simple and efficient comparison sort. In this algorithm, each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through.

Key Points:

1. Simple implementation
2. Efficient for small data
3. Adaptive: If the input list is presorted [may not be completely] then insertions sort takes O(n + d), where d is the number of inversions
4. Practically more efficient than selection and bubble sorts, even though all of them have O(n 2) worst case complexity
5. Stable: Maintains relative order of input data if the keys are same
6. In-place: It requires only a constant amount O(1) of additional memory space
7. Online: Insertion sort can sort the list as it receives it

Algorithm:

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

```
void insertionSort(int array[]) {
    int size = array.length;
    for (int step = 1; step < size; step++) {
      int key = array[step];
      int j = step - 1;
     key<array[j] to key>array[j].
      while (j >= 0 && key < array[j]) {
        array[j + 1] = array[j];
        --j;
      }
      array[j + 1] = key;
    }
  }
```

Bubble Sort

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to the last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no more swaps are needed

1. Bubbles out the largest element after each pass.
2. Inplace, Stable, not adaptive.
3. Modified Bubble sort is adaptive.

Algorithm:

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

```
static void bubbleSort(int array[]) {
  int size = array.length;
  // loop to access each array element
  for (int i = 0; i < size - 1; i++)
    // loop to compare array elements
    for (int j = 0; j < size - i - 1; j++)
      if (array[j] > array[j + 1]) {
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
}
```

Merge Sort

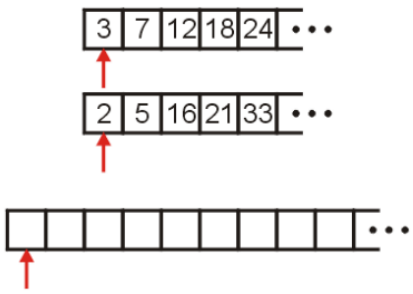
The merge sort algorithm is defined recursively:

– If the list is of size 1, it is sorted—we are done;

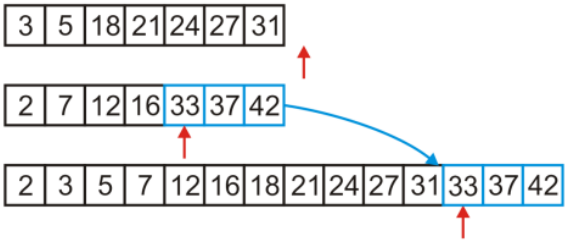
– Otherwise:

- Divide an unsorted list into two sub-lists,
- Sort each sub-list recursively using merge sort, and
- Merge the two sorted sub-lists into a single sorted list

EXAMPLE




```
while ( i1 < n1 && i2 < n2 ) {
    if ( array1[i1] < array2[i2] ) {
        arrayout[k] = array1[i1];
        ++i1;
    } else {
        arrayout[k] = array2[i2];
        ++i2;
    }
    ++k;
}
```



The algorithm:

– Split the list into two approximately equal sub-lists
– Recursively call merge sort on both sub lists
– Merge the resulting sorted lists

Implementation

```
void merge_sort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        int midpoint = (first + last)/2;

        merge_sort( array, first, midpoint );
        merge_sort( array, midpoint, last );
        merge( array, first, midpoint, last );
    }
}
```

Quick Sort

Key Points:

Step:1 Consider the last element of the list as pivot (i.e., Element at last position in the list).
Step 2 - Define two variables low and high. Set low and high to first and last elements of the list respectively. and set pivot to arr[high]//last element.
Step 3 - Initialize the smaller element i to low-1// i = low-1 and next element j. Set j= low.
Step 4 - Traverse the elements from j to high -1(second last element)
Step 4.1- If arr[j] < pivot then increment i and swap arr[i] and arr[j].
Step 6 - Repeat steps 3,4 & 4.1 until j recahes high -1.
Step 7 - Exchange the pivot element with arr[i + 1] element.
Step 8 - Return i when added 1.
arr[] = {10, 80, 30, 90, 40, 50, 70} Indexes: 0 1 2 3 4 5 6
low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1
Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot,
do i++ and swap arr[i] with arr[j] i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of the loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it and all elements greater than 70 are after it.

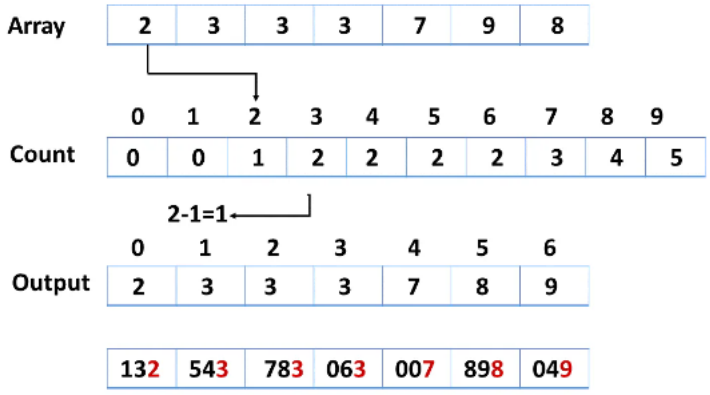
Radix Sort:

Let's start with [132, 543, 783, 63, 7, 49, 898]. It is sorted using radix sort, as illustrated in the figure below.

- Find the array's largest element, i.e., maximum. Consider A to be the number of digits in maximum. A is calculated because we must traverse all of the significant locations of all elements.

The largest number in this array [132, 543, 783, 63, 7, 49, 898] is 898. It has three digits. As a result, the loop should be extended to hundreds of places (3 times).

- Now, go through each significant location one by one. Sort the digits at each significant place with any stable sorting technique. You must use counting sort for this. Sort the elements using the unit place digits (A = 0).



- Sort the elements now by digits in the tens place.

- Finally, sort the elements by digits in the hundreds place.

Implementation

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Lab Tasks:

1. Consider a Two Dimensional Array of NxN, Input must be taken by user Write a program that will iterate through each row of the given 2D array, and sort elements of each row using an efficient sorting algorithm.

2. Write a program that will count the number of swap required to sort the array using insertion sort algorithm.
   Input: A[] = {2, 1, 3, 1, 2}
   Output: 4
   Explanation:
   Step 1: arr[0] stays in its initial position.
   Step 2: arr[1] shifts 1 place to the left. Count = 1.
   Step 3: arr[2] stays in its initial position.
   Step 4: arr[3] shifts 2 places to the left. Count = 2.
   Step 5: arr[5] shifts 1 place to its right. Count = 1.

3. We are given an array. We need to sort the even positioned elements in the ascending order and the odd positioned elements in the descending order.
   Examples:
   Input : a[] = {7, 10, 11, 3, 6, 9, 2, 13, 0}
   Output : 11 3 7 9 6 10 2 13 0
   Even positioned elements after sorting int
   ascending order : 3 9 10 13
   Odd positioned elements after sorting int
   descending order : 11 7 6 2 0

4. Write a program that will sort the given string Array using Bubble Sort.

```
Strings in sorted order are :
String 1 is Adam
String 2 is Adrm
String 3 is Apple
String 4 is Cat
String 5 is Cloths
String 6 is Orange
```

5. You don't need to take the input or print anything. Your task is to complete the function merge() which takes arr[], l, m, r as its input parameters and modifies arr[] in-place such that it is sorted from position l to position r, and function mergeSort() which uses merge() to sort the array in ascending order using merge sort algorithm.