Lab 08 Implementation of Sorting Algorithms

Learning Objectives

- Learn to implement different sorting algorithms using integer arrays
- Theoretically estimate and compare running times of different sorting algorithms
- Empirically test and compare running times of different sorting algorithms

Pre-Lab Reading

Bubble Sort Algorithm:

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sort, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.
This procedure of sort is called bubble sort because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

```
Algorithm Bubblesort(input: n, V)
def n : number of values to sort;
def V[n] : array of size n;
def temp, i, j: integer variables;
1. i := n - 1;
2. while ( i >= 1 ) do
3.     j := 0;
4.     while ( j < i ) do
5.           if ( V[j] < V[j+1] )
6.                 temp   := V[j];
7.                 V[j]   := V[j+1];
8.                 V[j+1]:= temp;
9.           end if
10.          j := j + 1;
11.    end while
12.    i := i - 1;
13. end while
14. return V;
```

Algorithm 1: Bubble Sort

Selection Sort Algorithm:

Selection sort is a sorting algorithm that has a quadratic running time complexity of $\mathbf{O(n^2)}$, thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Algorithm 2: Selection Sort

Insertion Sort Algorithm:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort. However, insertion sort provides several advantages like simplicity, efficiency on small datasets, more efficient compare to selection sort and bubble sort, in-place sorting and stability.

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Algorithm 3: Insertion Sort

Merge Sort Algorithm:

All of the comparison-based sorting algorithms that we've seen thus far have sorted the array in place (used only a small amount of additional memory). Mergesort is a sorting algorithm that requires an additional temporary array of the same size as the original one. It needs O(n) additional space, where n is the array size. It is based on the process of merging two sorted arrays into a single sorted array.

To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

We repeatedly do the following:

- compare A[i] and B[j]
- copy the smaller of the two to C[k]
- increment the index of the array whose element was copied
- increment k

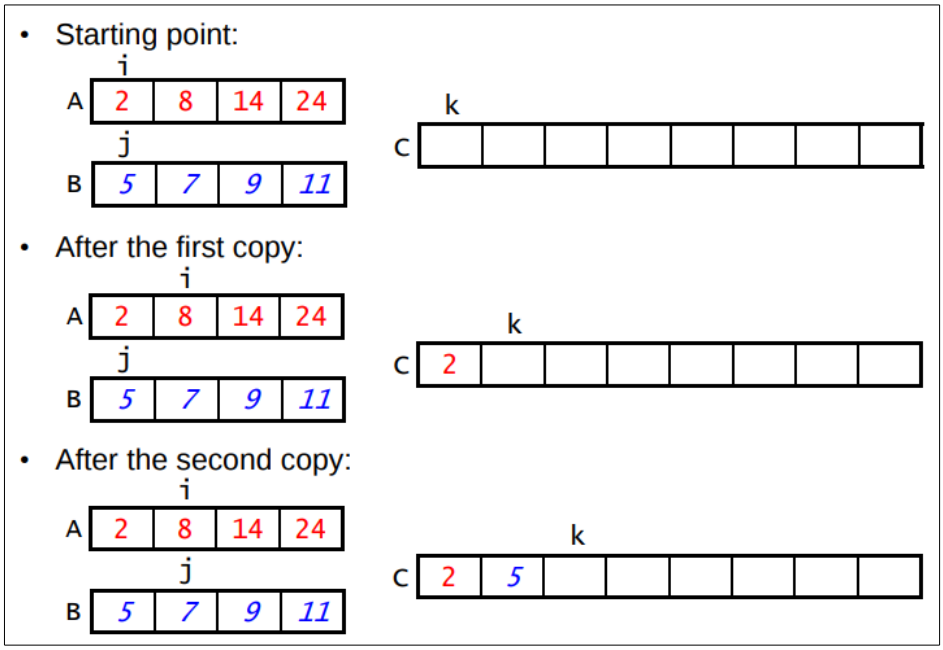

Figure 1 Merging two sorted arrays (start)

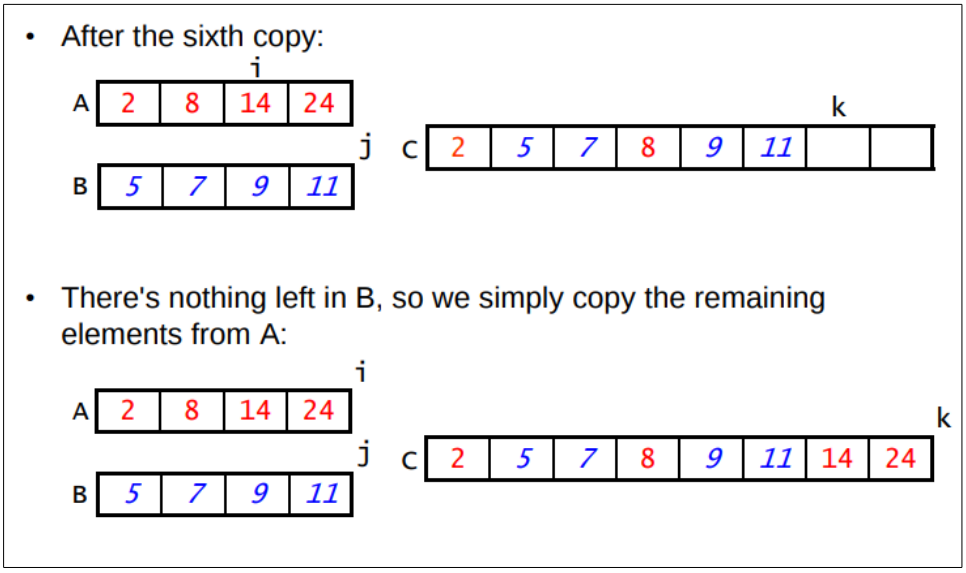

Figure 2: Merging two sorted arrays (end)

The divide and conquer approach of the algorithm first divides the input array into two sub-arrays and calls itself recursively for each sub-array. The base case is when we are left with one element. Then it merges the two returning sub-arrays using the algorithm depicted in Figures 1 and Figure 2. The complete process is shown in Figure 3.

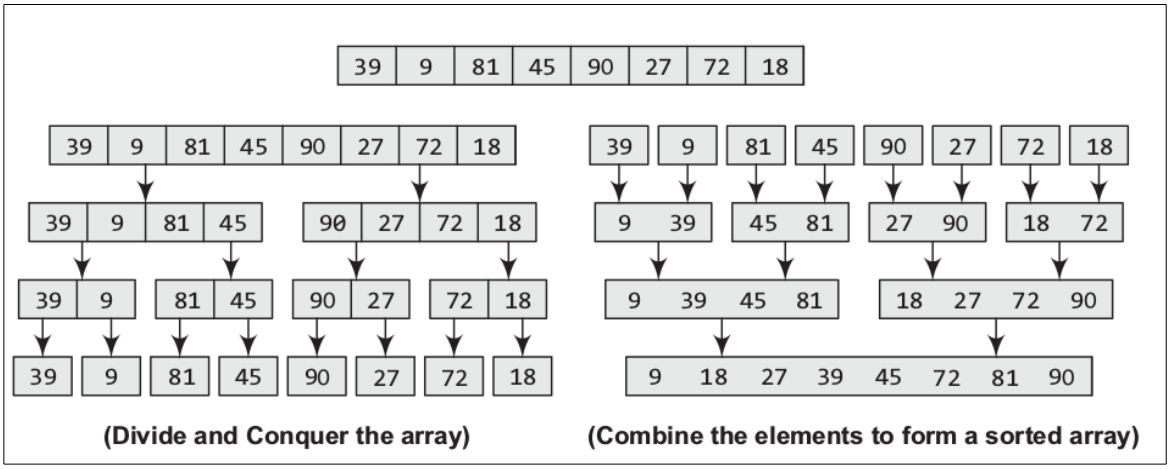

Figure 3: Working of Merge Sort

For further insight, read chapter **14.7** to **14.10** (**Pages 434 – 445**) from the book "***Data Structures using C***", by Reema Thareja, 2nd Edition.

In-Lab Task 1: Complete the functions for Selection Sort and Insertion Sort

You are provided with the skeleton code for this lab. This program uses Bubble Sort, Selection Sort, Insertion Sort and Merge Sort to sort an array of integers. Bubble Sort implementation is already completed and is provided for reference so that you may familiarize yourselves with the working of the program.

The program computes the time taken by the sorting function and displays this after sorting has finished. **Your task** is to complete the **Selection Sort** and **Insertion Sort** parts of the code.

In-Lab Task 2: Empirically compute the execution times for the three sorting methods

Here you will run the program with different data sizes and for different sorting techniques, while compiling the resulting execution times in the table provided below.

	Data Size ↓	**Bubble Sort**	**Selection Sort**	**Insertion Sort**	**Merge Sort**
1	16				
2	128				
3	1024				
4	16384				
5	131072				

Post Lab Task: Complete the Merge Sort Function and empirically determine its time complexity.