CIS 23: Data Structures and Algorithms

Homework 10
Prof. Sana Vaziri

Cody Vig

Abstract

Sorry in advance, this is not my best work! It has been a tremendously busy few weeks. I hope my explanations are still easy enough to read!

Problem 1

Consider the list $L = [50, 36, 78, 40, 4, 28, 90, 62, 22]$

Sort this list using bubble sort as described in the textbook. Show the list after each iteration.

Solution

The sequence of comparisons for one iteration of the bubble sort algorithm is shown. The red text denotes the two elements being compared. If a swap is made at this stage, it is shown on the right of the equals sign.

1. [50, 36, 78, 40, 4, 28, 90, 62, 22] = [36, 50, 78, 40, 4, 28, 90, 62, 22]
2. [36, 50, 78, 40, 4, 28, 90, 62, 22]
3. [36, 50, 78, 40, 4, 28, 90, 62, 22] = [36, 50, 40, 78, 4, 28, 90, 62, 22]
4. [36, 50, 40, 78, 4, 28, 90, 62, 22] = [36, 50, 40, 4, 78, 28, 90, 62, 22]
5. [36, 50, 40, 4, 78, 28, 90, 62, 22] = [36, 50, 40, 4, 28, 78, 90, 62, 22]
6. [36, 50, 40, 4, 78, 28, 90, 62, 22]
7. [36, 50, 40, 4, 78, 28, 90, 62, 22] = [36, 50, 40, 4, 78, 28, 62, 90, 22]
8. [36, 50, 40, 4, 78, 28, 62, 90, 22] = [36, 50, 40, 4, 78, 28, 62, 22, 90]

This single iteration now guarantees that the last element in the list is the largest element, so we need not compare it to anything again. We repeat this process on *all but the last element* of the list, and continue inductively until the list is sorted. The next iteration is shown, with blue text representing the portion of the list that we need not check:

1. [36, 50, 40, 4, 78, 28, 62, 22, 90]
2. [36, 50, 40, 4, 78, 28, 62, 22, 90] = [36, 40, 50, 4, 78, 28, 62, 22, 90]
3. [36, 40, 50, 4, 78, 28, 62, 22, 90] = [36, 40, 4, 50, 78, 28, 62, 22, 90]
4. [36, 40, 4, 50, 78, 28, 62, 22, 90]
5. [36, 40, 4, 50, 78, 28, 62, 22, 90] = [36, 40, 4, 50, 28, 78, 62, 22, 90]
6. [36, 40, 4, 50, 28, 78, 62, 22, 90] = [36, 40, 4, 50, 28, 62, 78, 22, 90]
7. [36, 40, 4, 50, 28, 62, 78, 22, 90] = [36, 40, 4, 50, 28, 62, 22, 78, 90]

This iteration, together with the previous, guarantees that the last *two* elements are sorted properly, so these two elements can be ignored in what follows. The next iteration is similar to the previous:

1. [36, 40, 4, 50, 28, 62, 22, 78, 90]
2. [36, 40, 4, 50, 28, 62, 22, 78, 90] = [36, 4, 40, 50, 28, 62, 22, 78, 90]
3. [36, 4, 40, 50, 28, 62, 22, 78, 90]
4. [36, 4, 40, 50, 28, 62, 22, 78, 90] = [36, 4, 40, 28, 50, 62, 22, 78, 90]

5. [36, 4, 40, 28, 50, 62, 22, 78, 90]
6. [36, 4, 40, 28, 50, 62, 22, 78, 90] = [36, 4, 40, 28, 50, 22, 62, 78, 90]

The iterations thus far guarantee that the last *three* elements are in the correct location. This process is to be repeated inductively six more times to completely sort the list. The last step of each remaining iteration is:

- [4, 36, 28, 40, 22, 50, 62, 78, 90]
- [4, 28, 36, 22, 40, 50, 62, 78, 90]
- [4, 28, 22, 36, 40, 50, 62, 78, 90]
- [4, 22, 28, 36, 40, 50, 62, 78, 90]
- [4, 22, 28, 36, 40, 50, 62, 78, 90]
- [4, 22, 28, 36, 40, 50, 62, 78, 90]
- [4, 22, 28, 36, 40, 50, 62, 78, 90]

Problem 2

Consider the list $L = [18, 8, 38, 25, 45, 12, 74, 60, 30]$.

Show this list after 5 iterations of the insertion sort algorithm as given in the textbook.

Solution

This algorithm starts by considering the element at index 0 to be on the sorted side, and considering the remaining elements as being on the *unsorted* side:

- [18, 8, 38, 25, 45, 12, 74, 60, 30].

The element at index 1 is therefore the first index out of order. It is then placed into its correct location on the sorted side to give:

1. [8, 18, 38, 25, 45, 12, 74, 60, 30].

The element out of order is 38, the element at index 2. It is then placed in its correct location on the sorted side (which, incidentially, is where it already is):

2. [8, 18, 38, 25, 45, 12, 74, 60, 30].

Now the element at index 3 is the first index out of order. Placing it into its correct location on the sorted side gives:

3. [8, 18, 25, 38, 45, 12, 74, 60, 30].

Next, the element at index 4 is out of order, and placing it at its correct location on the sorted side yields:

4. [8, 18, 25, 38, 45, 12, 74, 60, 30].

At the fifth iteration, the item at index 5 is the first out of order, and so placing it in its correct location on the sorted side gives the list:

5. [8, 12, 18, 25, 38, 45, 74, 60, 30].

This above is the list after five iterations of the insertion sort algorithm.

Problem 3

Consider Insertion Sort and Selection Sort (as described in the book):

Insertion Sort

```
template <class elemType>
void insertionSort(elemType list[], int length)
{
    for (int firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++)
    {
        if (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
        {
            elemType temp = list[firstOutOfOrder];
            int location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
    }
} // end insertionSort
```

Selection Sort

```
template <class elemType>
void selectionSort(elemType list[], int length)
{
    int minIndex;

    for (int loc = 0; loc < length; loc++)
    {
        minIndex = minLocation(list, loc, length - 1);
        swap(list, loc, minIndex);
    }
} // end selectionSort
```

Both algorithms grow a "sorted" side by moving elements from one side to another.

Explain how the outer for loop of `insertionSort()` is similar to the for loop of `selectionSort()`. Explain the main difference between how the two algorithms grow their "sorted" sides.

Solution

The `insertionSort()` algorithm begins by partitioning the list into two sides, one "sorted" and the other "unsorted", at index 1. At the next iteration, the element at index 1 is placed into its correct location on the *sorted* side of the list by comparing it with the element at index 0. The process is repeated recursively, where the element at index i is placed into its correct location on the *sorted* side of the list by looping through the sorted side until the correct location is found.

The `selectionSort()` algorithm begins by finding the smallest number in the list and swapping it with the item at index 0. After one iteration, the smallest element is at the beginning of the list, and the remaining

elements at indices 1, 2, ..., n are unsorted. The process is repeated recursively by replacing the indices i, $(i+1)$, ..., n with $(i+1)$, $(i+2)$, ..., n.

Both algorithms "grow" a sorted side by moving one element at a time from the unsorted side into its correct location. The main difference is that in `insertionSort()`, the element at index i is moved into its correct location on the sorted side at iteration i of the algorithm, whereas in `selectionSort()`, the minimum element of the unsorted side of the list is placed into the sorted side at index i at iteration i of the algorithm. Essentially, the step of the iteration controls what element in the *unsorted* side gets moved in `insertionSort()`, while the step of the iteration controls what element in the *sorted* side gets added in `selectionSort()`.

Problem 4

Consider the list $L = [48, 30, 66, 50, 9, 95, 80, 15, 25, 18, 94, 55, 3, 22, 62]$.

Suppose this list is to be sorted using Quick Sort as given in the textbook. Let the pivot be the median value of the first, middle, and last element in the list. Answer the following:

1. What is the pivot when partition is called for the first time?
2. What is the list after the first call to partition?
3. What are the sizes of the two sublists created by partition?

Solution

1. The pivot in this algorithm is defined to be the median of the first, middle, and last element in the list, which, in order, are the numbers 15, 48, and 62. The median of $\{15, 48, 62\}$ is 48, and so the pivot is 48.
2. A single iteration of the `partition()` function simultaneously finds the correct location for the pivot in the list *and* moves all elements which are smaller than the pivot in front of the pivot while otherwise preserving their order. Since the correct location of the pivot is index 7, we have the following list after the first call to `partition()`:

- [30, 9, 15, 25, 18, 3, 22, 48, 66, 50, 95, 80, 94, 55, 62].

3. The elements to the left of the pivot comprise the left sublist, and the elements to the right of the pivot comprise the right sublist. The left sublist has length 7, and the right sublist incidentally also has length 7.

Problem 5

Both the merge sort and quick sort algorithms sort a list by partitioning it. Explain how the merge sort algorithm differs from the quick sort algorithm in partitioning the list.

Solution

In the merge sort algorithm, the list is recursively bisected until only one element is in each sublist. Then, the two adjacent sublists are merged in a way that preserves their relative order. Sorting happens granually, and then the constituent sublists are merged.

In the quick sort algorithm, a pivot (typically the element at the middle index) is chosen, and the list is partitioned so that the elements that are smaller than the pivot are placed to its left, and the elements that are larger than the pivot are placed to its right. Some level of sorting is happening at each iteration of the pivot.

Essentially, in merge sort, sorting happens *after* the lists are partitioned, whereas in quick sort, a sorting is taking place *during* each partition.
