Assignment Two

Josh Seligman

joshua.seligman1@marist.edu

September 24, 2022

1 Selection Sort

1.1 The Algorithm

Selection sort is a sorting algorithm that, for each iteration of the array, selects the smallest (or largest) element of the unsorted part of the array and places the element into its sorted position. As shown in the pseudocode for the sort in Algorithm 1, selection sort works with the subset of the array in the range $[i,\ n)$ in each iteration because the elements in the indices less than i are already sorted and do not have to be checked. Thus, as more elements get sorted, the quicker each iteration becomes because a smaller portion of the array is compared until $i = n - 2$, which is the final iteration of the algorithm. Selection sort is also very consistent in that it runs in the same amount of time regardless of the order of the elements and has both a best and worst case of n^2, which will be analyzed in further detail in Section 1.2.

Algorithm 1 Selection Sort Algorithm

```
1:  procedure SelectionSort(arr)
2:      for i ← 0, n − 2 do    // Iterate through the second to last element as an array of size 1 is sorted
3:          smallestIndex ← i
4:          for j ← i + 1, n − 1 do    // Iterate through the remainder of the array
5:              if arr[j] < arr[smallestIndex] then
6:                  smallestIndex ← j    // Set the new smallest index if a smaller element is found
7:              end if
8:          end for
9:          swap(arr, i, smallestIndex)    // Place the smallest item in the subarray into its sorted place
10:     end for
11: end procedure
```

1.2 Asymptotic Analysis

Listing 1 contains the C++ code implementing selection sort on lines 6 - 25. Line 6 defines a loop that iterates $n - 1$ times and contains 2 assignments and a comparison, all of which operate in constant time for each iteration. Thus, line 6 will take $(n-1) * C_1$ time, where C_1 is the time needed for each of the operations.

Next, line 8 is an assignment, which take a constant time and executes $n-1$ times because it is in the outer loop, resulting in a time of $(n-1)*C_2$, where C_2 is the constant time needed for the assignment. Line 11, similar to line 6, defines a loop with 3 constant time expressions, which can be marked as C_3. However, since it is nested inside of the loop on line 6, the total number of iterations of the inner loop is more complex. In the first iteration of the outer loop, the inner loop runs $n-1$ times. From there, each corresponding iteration of the outer loop results in one less iteration of the inner loop with a minimum of 1 pass on the inner loop when $i = n-2$. Therefore, the total number of times the inner loop on line 11 will be called is $\sum_{k=1}^{n-1} k$, which by the formula for the sum of the first N natural numbers, is equal to $\frac{(n-1)(n-1+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Thus, the total time to execute line 11 is $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_3$. Next, line 13 contains a comparison that, since it is nested inside the inner loop, will run in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_4$ time, where C_4 is the time needed to make the comparison. Line 15 is a simple assignment and, just like line 14, will run in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_5$, where C_5 is the time to perform the assignment. The assignment on line 18 is purely for collecting data and not part of the algorithm and, therefore, will be excluded from the asymptotic analysis of selection sort. Line 19 is the end of the inner loop, and represents an unconditional branch back to the top of the loop, which means it runs the same number of iterations as the loop, which is $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_6$, where C_6 is the time needed to execute the branch. Next, lines 22-24 are all assigments, which run in constant time, and are located in the outer loop. Thus, they run in $(n-1) * C_7$ time, where C_7 is the time needed to perform the swap. Lastly, line 25 is the close and unconditional branch for the outer loop, which will run in $(n-1) * C_8$ time, where C_8 is the time to execute the unconditional branch. Overall, when adding up the runtimes of each line and dropping the constants, the sum is $4 * (n-1) + 4 * (\frac{1}{2}n^2 - \frac{1}{2}n) = 2n^2 + 2n - 4 \approx n^2 + n$ is $O(n^2)$.

2 Insertion Sort

2.1 The Algorithm

Insertion sort in a sorting algorithm that places an element in its sorted place by sliding previously sorted elements over until the sorted position is found for the element. Unlike selection sort, insertion sort has a unique property in that its performance varies based on the state of the input array. For instance, if the array is already completely sorted, the while loop on line 5 in Algorithm 2 will never be entered because each element is already sorted and in its proper position. This makes the best case runtime of insertion sort $\Omega(n)$ because the inner loop is never run and the outer loop just iteraties through the array once. However, the worst case of insertion sort is when the array is in reverse order. This becomes the worst case because, as shown within the while loop in Algorithm 2, each element will have to be compared with every other element, which will cause j to end at -1 and the element gets inserted at the front of the array. This results in a worst case runtime complexity of $O(n^2)$, which will be analyzed and proved in detail in Section 2.2.

Algorithm 2 Insertion Sort Algorithm

```
1:  procedure InsertionSort(arr)
2:      for i ← 1, n − 1 do    // Start at index 1 because the first element is already sorted
3:          currentVal ← arr[i]
4:          j ← i − 1
5:          while j ≥ 0 and currentVal < arr[j] do    // Find the position to place the element
6:              arr[j + 1] ← arr[j]    // Shift the element over because it is greater than the current value
7:              j ← j − 1
8:          end while
9:          arr[j + 1] ← currentVal    // Place the element in its sorted position
10:     end for
11: end procedure
```

2.2 Asymptotic Analysis

The code implementation of nsertion sort is found in Listing 2 on lines 7-34.

3 Appendix

3.1 Selection Sort

```
1  int selectionSort(StringArr* data) {
2      // Start comparisons at 0
3      int comparisons = 0;
4
5      // Iterate through the second to last element because the last element will already be
           sorted as is
6      for (int i = 0; i < data->length - 1; i++) {
7          // The smallest index is going to start as the start of the subset of the list
8          int smallestIndex = i;
9
10         // Iterate through the rest of the list
11         for (int j = i + 1; j < data->length; j++) {
12             // Compare the current element to the current smallest element in the subset
13             if (data->arr[smallestIndex].compare(data->arr[j]) > 0) {
14                 // If the current element comes first, make it the new smallest element
15                 smallestIndex = j;
16             }
17             // Increment comparisons
18             comparisons++;
19         }
20
21         // Put the smallest index in its respective place
22         std::string temp = data->arr[i];
23         data->arr[i] = data->arr[smallestIndex];
24         data->arr[smallestIndex] = temp;
25     }
26
27     // Return the number of comparisons
28     return comparisons;
29 }
```

Listing 1: Selection Sort (C++)

3.2 Insertion Sort

```
1  int insertionSort(StringArr* data) {
2      // Number of comparisons starts at 0
3      int comparisons = 0;
4
5      // We begin with the second element because an array of size 1 is already sorted
6      // So no need to check on the first element
7      for (int i = 1; i < data->length; i++) {
8          // Save the current element for later use
9          std::string cur = data->arr[i];
10
11         // Comparisons are going to start with the previous index
12         int j = i - 1;
13
14         // Continue until j is a valid index (< 0) or until we found an element that is less
               than the
15         // current element that is being sorted
16         while (j >= 0 && cur.compare(data->arr[j]) < 0) {
17             // We made a comparison so increment it
18             comparisons++;
19
```

```
20              // Shift the compared element over 1 to make room for the element being sorted
21              data->arr[j + 1] = data->arr[j];
22              j--;
23          }
24
25          // After the loop, we want to increment comparisons only if j >= 0 because
26          // if j < 0, then the boolean expression would have immediately returned false
                without making
27          // a comparison
28          if (j >= 0) {
29              comparisons++;
30          }
31
32          // Place the value in its proper place
33          data->arr[j + 1] = cur;
34      }
35
36      // Return the number of comparisons
37      return comparisons;
38  }
```

Listing 2: Insertion Sort (C++)