

# Assignment Two

---

Josh Seligman

joshua.seligman1@marist.edu

September 23, 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  $i \leftarrow 0, n - 2$  do    // Iterate through the second to last element as an array of size 1 is sorted
3:     smallestIndex  $\leftarrow i$ 
4:     for  $j \leftarrow i + 1, n - 1$  do    // Iterate through the remainder of the array
5:       if  $arr[j] < arr[smallestIndex]$  then
6:         smallestIndex  $\leftarrow j$     // 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 takes 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. Lastly, lines 22-24 are all assignments, which run in constant time, and are located in the outer loop. Thus, they run in  $(n - 1) * C_6$  time, where  $C_6$  is the time needed to perform the swap. When adding up the runtimes of each line and dropping the constants, the sum is  $3 * (n - 1) + 3 * (\frac{1}{2}n^2 - \frac{1}{2}n) = \frac{3}{2}n^2 + \frac{3}{2}n - 3 \approx n^2 + n$  is  $O(n^2)$ .

## 2 INSERTION SORT

### 2.1 THE ALGORITHM

---

**Algorithm 2** Insertion Sort Algorithm

---

```

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

```

---

### 2.2 ASYMPTOTIC ANALYSIS

## 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
15         // than the
16         // current element that is being sorted
17         while (j >= 0 && cur.compare(data->arr[j]) < 0) {
18             // We made a comparison so increment it
19             comparisons++;
20
21             // Shift the compared element over 1 to make room for the element being sorted
22             data->arr[j + 1] = data->arr[j];
23             j--;
24         }
25
26         // After the loop, we want to increment comparisons only if j >= 0 because
27         // if j < 0, then the boolean expression would have immediately returned false
28         // without making
29         // a comparison
30         if (j >= 0) {
31             comparisons++;
32         }
33
34         // Place the value in its proper place
35         data->arr[j + 1] = cur;
36     }
37
38     // Return the number of comparisons
39     return comparisons;
40 }

```

38 }

## Listing 2: Insertion Sort (C++)