# Assignment Two

Josh Seligman

joshua.seligman1@marist.edu

September 29, 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 and Comparisons

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 $O(n^2)$.

As shown in Table 4.1, selection sort is very consistent with the number of comparisons made as, regardless of the state of the list, it always makes $\frac{1}{2}n^2 - \frac{1}{2}n$ comparisons. This is no coincidence as it is also the number of times the algorithm's inner loop iterates, which means that the selection sort will run very consistently for all lists, no matter the state of the array prior to running the algorithm.

# 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.

## 2.2 Asymptotic Analysis and Comparisons

The code implementation of insertion sort is in Listing 2 on lines 7-34. As mentioned in Section 2.1, the worst case for insertion sort is when the array is in reverse order because every element will have to be compared to every element within the sorted portion of the array. First, the outer loop begins on line 7 and contains 2 assignments and a comparison. Based on the definition of the loop, these statements will run $n-1$ times, which means the line will run in $(n-1) * C_1$ time, where $C_1$ is the time needed to execute these statements. Next, line 9 is a basic assignment and, since it is in the loop, will run in $(n-1) * C_2$ time, where $C_2$ is the time to perform the assignment. Line 12 is also an assignment in the outer loop, which will run in $(n-1) * C_3$ time, where $C_3$ is the time to execute the assignment. Next, line 16 contains the definition for a while loop, which

**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
```

has 2 comparisions. Since the worst case requires each element to be compared to all of the other sorted elements, the while loop will terminate when $j = -1$. Thus, when $i = 1$, the while loop will only iterate once, and the number of iterations for the while loop will increment with a max of $n - 1$ for when $i = n - 1$. The total iterations is explained in Section 1.2 as being $\sum_{k=1}^{n-1} k = \frac{(n-1)(n-1+1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$, which means the loop on line 16 will run in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_4$ time, where $C_4$ is the time to perform the comparisons. Line 18 is just for counting the comparisons and will be excluded from the analysis of the algorithm. Next, the assignment on line 21 runs in constant time and, since it is in the inner loop, will run in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_5$ time, where $C_5$ is the time to perform the assignment. Line 22 also contains an assignment and runs in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_6$ time, where $C_6$ is the time to perform the assignment. Line 23 is the end of the while loop, which translates to an unconditional branch to the top of the loop, which will run in $(\frac{1}{2}n^2 - \frac{1}{2}n) * C_7$ time, where $C_7$ is the time to execute the branch. Lines 28-30 are used for counting the comparisons and are excluded from the analysis of insertion sort. Next, line 33 is an assignment to put the element in its sorted place, which will run in $(n - 1) * C_8$ time because it is in the outer loop, where $C_8$ is the time to perform the assignment. Lastly, line 34 is the end of the for loop, which is an unconditional branch to the top of the loop, which will run in $(n - 1) * C_9$, where $C_9$ is the time to perform the branch. Overall, when summing up each runtime and dropping the constants, the total is $5 * (n - 1) + 4 * (\frac{1}{2}n^2 - \frac{1}{2}n) = 2n^2 + 3n - 5 \approx n^2 + n$ is $O(n^2)$.

In Table 4.1, insertion sort is shown to have 3 very different outcomes for the lists that were used for testing relative to selection sort. First, insertion sort used about half the number of comparisons as selection sort for a list of 666 shuffled magic items. This is because the inner loop of insertion sort may terminate when $arr[j] < currentVal$ (see line 16 in Listing 2) and in a randomly shuffled list, the probability of $arr[j] < currentVal$ will be around 50%. Therefore, insertion sort will on average be about 50% more efficient than selection sort, but is still classified as $O(n^2)$ beause it is still running at a function of $n^2$. Next, when the list is already shuffled, insertion sort only makes $n - 1$ comparisons. As mentioned in Section 2.1, the best case for insertion sort is $\Omega(n)$ because the inner loop will never be entered as the second condition for $arr[j] < currentVal$ will always return false. This means there will be only 1 comparison made for each iteration of the outer loop, which equates to $n-1$ comparisions. Lastly, Section 2.1 mentioned that the worst case for insertion sort is when the list is in reverse order because every element will have to compare itself with all of the elements in the sorted portion of the array. This causes insertion sort to have the same number of comparisons as selection sort for a reversed list at $\frac{1}{2}n^2 - \frac{1}{2}n$, which is also the same number of iterations as the inner loop for insertion sort and makes insertion sort $O(n^2)$.

# 3 Merge Sort

## 3.1 The Algorithm

Merge sort is a divide and conquer sorting algorithm that continues to divide an array up until it has $n$ subarrays of size 1, which, by definition, are all sorted. From there, the subarrays are merged together by comparing the elements in each subarray to determine the sorted order of the combined subarrays. Eventually, the full array will be merged back together will all of the elements fully sorted. As displayed in Algorithm 3 in the *MergeSort* procedure, since the sort is a divide and conquer algorithm, merge sort takes advantage of recursion to make the problem smaller until it reaches its base case of $length(arr) <= 1$, which is shown on line 2. Additionally, on lines 4 and 5, merge sort always divides a given array in half, which makes its performance very predictable and consistent, which will be discussed more in detail in Section 3.2.

**Algorithm 3** Merge Sort Algorithm

```
1:  procedure MergeSort(arr)
2:      if length(arr) > 1 then    // An array of size 1 is already sorted
3:          mid = floor((length(arr))/2)    // Get the middle index of the array for splitting it in half
4:          MergeSort(arr[0 : mid])    // Perform merge sort on the first half of the array (index 0 - mid, inclusive)
5:          MergeSort(arr[mid + 1 : length(arr) − 1])    // Perform merge sort on the second half of the array
6:          Merge(arr, mid)    // Merge the 2 subarrays together in order
7:      end if
8:  end procedure
9:
10: procedure Merge(arr, mid)
11:     leftIndex ← 0    // Index for the left subarray
12:     rightIndex ← mid + 1    // Index for the right subarray
13:     newArr ← [ ]
14:     for i ← 0, length(arr) − 1 do    // Iterate through all elements
15:         if rightIndex >= length(arr) then    // All the right subarray items are already in newArr
16:             newArr[i] = arr[leftIndex]    // Add the next item from the left subarray
17:             leftIndex + +
18:         else if leftIndex > mid then    // All the right subarray items are already in newArr
19:             newArr[i] = arr[rightIndex]    // Add the next item from the right subarray
20:             rightIndex + +
21:         else if arr[leftIndex] < arr[rightIndex] then    // The next element from the left subarray is less than the next element from the right subarray
22:             newArr[i] = arr[leftIndex]    // Add the next item from the left subarray
23:             leftIndex + +
24:         else
25:             newArr[i] = arr[rightIndex]    // Add the next item from the right subarray
26:             rightIndex + +
27:         end if
28:     end for
29:     for j ← 0, length(arr) − 1 do
30:         arr[j] ← newArr[j]    // Transfer the sorted elements to the original array
31:     end for
32: end procedure
```

## 3.2 Asymptotic Analysis and Comparisons

The C++ implementation of merge sort can be found in Listing 3, more specifically lines 6-28. First, the base case for checking if there are 0 or 1 elements is on lines 8-11 with an if statement and a return for the number of comparisons. This block of code runs 1 time per function call and runs in $? * C_1$ time, where ? is the number of times the function gets called through recursion and $C_1$ is the time needed to execute the comparison in the codition and return the value if needed. Next, line 14 computes the middle index for dividing the array, which runs once per function call in $? * C_2$ time, where ? is the number of times the function gets called through recursion and $C_2$ is the time to perform the operation. Next, lines 17 and 18 divide the array through recursive calls to the same function, each of which runs in $? * C_3$ and $? * C_4$ time, respectively, where ? is the number of times the function gets called and $C_3$ and $C_4$ are the time for the functions to run. Lastly, line 24 contains a call for the *merge* function, which will run in ?*?? time, where ? is the number of times the function gets called and ?? is the runtime of *merge*.

The first unknown so far is the number of times that the $mergeSortWithIndices$ function gets called to reach a subarray of size 1. Since the array gets divided in half each time, the number of divides needed to reach the base case will be $log_2 n$. The other unknown is the runtime complexity of *merge*. This function is defined on lines 30-80. First, lines 32-40 are variable assignments for the left subarray index, the right subarray index, the total length between both sides, and the new merged array. These lines will run in a total of $1 * C_5$ time, where $C_5$ is the time to execute the variable assignments. Next, line 46 defines a loop with 2 assigments and a comparison, which at worst case will run in $n * C_6$ time, where $C_6$ is the time to complete the assignments and comparison. Lines 48-70 define a large if-else block. There will always be 1, 2, or 3 comparisons as defined by the if conditions in lines 48, 52, and 56, and there will always be 2 assignments regardless of the condition that is true. Therefore, at worst case, the if statements will run in $n * C_7$ time because it is nested inside of the loop, where $C_7$ is the time to compute the comparisons and the assignments. Line 71 is the end of the loop, which is an unconditional branch running in $n * C_8$ time, where $C_8$ is the time to perform the branch. Next, line 74 defines a new for loop with 2 assignments and 1 comparison. Similar to the first loop on line 46, this loop will run at worst $n$ times, which means it will require $n * C_9$ time, where $C_9$ is the time to execute the assignments and comparison. Line 75 contains an assignment within the loop, which will run in $n * C_{10}$ time, where $C_{10}$ is the time to execute the assignment. Lastly, line 76 is the end of the loop, which is an unconditional branch, running in $n * C_{11}$ time, where $C_{11}$ is the time to perform the branch. Overall, when adding up the runtimes of the *merge* function and removing constants, the sum is $1 + 6n \approx n$ is $O(n)$ and the value of ??.

When computing the runtime of merge sort by adding op the terms and dropping constants, the total is $4 * log_2 n + n * log_2 n \approx log_2 n + n * log_2 n$ is $O(n * log_2 n)$.

As shown in Table 4.1, merge sort is significantly more efficient at sorting the magic items by using around 5,500 comparisons, which is a little less than the algorithm's runtime of $O(n * log_2 n)$. The reason for the number of comparisons being less than 6246.67 ($666 * log_2 666$) is explained on lines 48 and 52 in Listing 3 as the merging of an entire subarray before the other will result in the algorithm merging the remainder of the other subarray without needing to compare the elements. Regardless, the number of comparisons is still not far from $O(n * log_2 n)$. Additionaly, merge sort is very similar to selection sort in that it will perform consistently for any permutation of an array of size $n$. This is shown in the smaller test cases, which perform almost identically despite being in differing orders. Also, one interesting point to note is the performance of insertion sort versus merge sort for the already sorted list. Since insertion sort has a best case of $\Omega(n)$, it is more efficient than merge sort when the array is already mostly sorted. This leads to the idea of hybrid algorithms that may use one sort up until a certain point and switch to a second algorithm that is more efficient in the end game of the sorting process.

# 4 Appendix

## 4.1 Comparisons Table

| Algorithm | List | Comparisons | Time |
|---|---|---|---|
| Selection Sort | 666 magic items, shuffled | 221445 | 3625271 ns |
| | 20 Yankees greats, sorted | 190 | 3673 ns |
| | 20 Yankees greats, reversed | 190 | 3636 ns |
| Insertion Sort | 666 magic items, shuffled | 104628 | 2523161 ns |
| | 20 Yankees greats, sorted | 19 | 795 ns |
| | 20 Yankees greats, reversed | 190 | 2966 ns |
| Merge Sort | 666 magic items, shuffled | 5417 | 1006113 ns |
| | 20 Yankees greats, sorted | 48 | 6751 ns |
| | 20 Yankees greats, reversed | 40 | 6745 ns |

Table 4.1: A table of the number of comparisons made and time to complete each sort on a variety of lists.

## 4.2 Selection Sort

```cpp
int selectionSort(StringArr* data) {
    // Start comparisons at 0
    int comparisons = 0;

    // Iterate through the second to last element because the last element will already be sorted as is
    for (int i = 0; i < data->length - 1; i++) {
        // The smallest index is going to start as the start of the subset of the list
        int smallestIndex = i;

        // Iterate through the rest of the list
        for (int j = i + 1; j < data->length; j++) {
            // Compare the current element to the current smallest element in the subset
            if (data->arr[smallestIndex].compare(data->arr[j]) > 0) {
                // If the current element comes first, make it the new smallest element
                smallestIndex = j;
            }
            // Increment comparisons
            comparisons++;
        }

        // Put the smallest index in its respective place
        std::string temp = data->arr[i];
        data->arr[i] = data->arr[smallestIndex];
        data->arr[smallestIndex] = temp;
    }

    // Return the number of comparisons
    return comparisons;
}
```

Listing 1: Selection Sort (C++)

## 4.3 Insertion Sort

```cpp
int insertionSort(StringArr* data) {
    // Number of comparisons starts at 0
    int comparisons = 0;

    // We begin with the second element because an array of size 1 is already sorted
    // So no need to check on the first element
```

```
    for (int i = 1; i < data->length; i++) {
        // Save the current element for later use
        std::string currentVal = data->arr[i];

        // Comparisons are going to start with the previous index
        int j = i - 1;

        // Continue until j is a valid index (< 0) or until we found an element that is less than the
        // current element that is being sorted
        while (j >= 0 && currentVal.compare(data->arr[j]) < 0) {
            // We made a comparison so increment it
            comparisons++;

            // Shift the compared element over 1 to make room for the element being sorted
            data->arr[j + 1] = data->arr[j];
            j--;
        }

        // After the loop, we want to increment comparisons only if j >= 0 because
        // if j < 0, then the boolean expression would have immediately returned false without making
        // a comparison
        if (j >= 0) {
            comparisons++;
        }

        // Place the value in its proper place
        data->arr[j + 1] = currentVal;
    }

    // Return the number of comparisons
    return comparisons;
}
```

Listing 2: Insertion Sort (C++)

### 4.4 Merge Sort

```
int mergeSort(StringArr* data) {
    // Return the number of comparisons from sorting the entire array
    return mergeSortWithIndices(data, 0, data->length - 1);
}

int mergeSortWithIndices(StringArr* data, int start, int end) {
    // Base case is array of size 1 or size 0 (if the list is completely empty)
    if (start >= end) {
        // No comparisons are needed here, so return 0
        return 0;
    }

    // Get the midpoint for the sections
    int mid = (start + end) / 2;

    // Sort the first half and get the number of comparisons needed to sort it
    int comp1 = mergeSortWithIndices(data, start, mid);

    // Sort the second haly and get the number of comparisons needed to sort it
    int comp2 = mergeSortWithIndices(data, mid + 1, end);

    // The number of comparisons for sorting the subarray is the number of comparsions made to sort the 2 halves
    // and thenumber of comparisons needed to merge the 2 halves together
    int comparisons = comp1 + comp2 + merge(data, start, end, mid);

```

```
26      // Return the total number of comparisons
27      return comparisons;
28  }
29
30  int merge(StringArr* data, int start, int end, int mid) {
31      // The left half is at the start
32      int leftIndex = start;
33
34      // The right half starts at the midpoint + 1
35      int rightIndex = mid + 1;
36
37      // Get the size of the array that the 2 halves will merge into
38      // and create the merged sub array
39      int subArrLength = end - start + 1;
40      std::string newSubArr[subArrLength];
41
42      // Start at 0 comparisons
43      int comparisons = 0;
44
45      // Iterate through the entire merged subarray
46      for (int i = 0; i < subArrLength; i++) {
47          // If the rightIndex > end, then the entire right half is already merged
48          if (rightIndex > end) {
49              // Add the next element from the left half
50              newSubArr[i] = data->arr[leftIndex];
51              leftIndex++;
52          } else if (leftIndex > mid) { // If the leftIndex > mid, then the entire left half
                is already merged
53              // Add the next element from the right half
54              newSubArr[i] = data->arr[rightIndex];
55              rightIndex++;
56          } else if (data->arr[leftIndex].compare(data->arr[rightIndex]) < 0) { // Compare the
                2 elements from each half
57              // Add the next element from the left half
58              newSubArr[i] = data->arr[leftIndex];
59              leftIndex++;
60
61              // Increment the number of comparisons made
62              comparisons++;
63          } else {
64              // Add the next element from the right half
65              newSubArr[i] = data->arr[rightIndex];
66              rightIndex++;
67
68              // Make sure to increment comparisons because a comparison was made in the last
                    else-if condition
69              comparisons++;
70          }
71      }
72
73      // Transfer the merged subarray to the actual array
74      for (int j = 0; j < subArrLength; j++) {
75          data->arr[start + j] = newSubArr[j];
76      }
77
78      // Return the number of comparisons
79      return comparisons;
80  }
```

Listing 3: Merge Sort (C++)