Assignment Two

Josh Seligman

joshua.seligman1@marist.edu

September 30, 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

```
procedure SelectionSort(arr)
    for i ← 0, n − 2 do    // Iterate through the second to last element as an array of size 1 is sorted
        smallestIndex ← i
        for j ← i + 1, n − 1 do    // Iterate through the remainder of the array
            if arr[j] < arr[smallestIndex] then
                smallestIndex ← j    // Set the new smallest index if a smaller element is found
            end if
        end for
        swap(arr, i, smallestIndex)    // Place the smallest item in the subarray into its sorted place
    end for
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. As shown in Algorithm 2, the element that is being sorted is only compared to elements in positions $[0, i-1]$ because these are the elements that have been worked with so far and are known to be in order. Therefore, the elements in this area, as described before, are shifted over until one is less than the value being sorted or until $j < 0$, which will break out of the while loop. Unlike selection sort, the performance of insertion sort varies based on the state of the input array, which will be explored more in detail in Section 2.2.

Algorithm 2 Insertion Sort Algorithm

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

2.2 Asymptotic Analysis and Comparisons

The code implementation of insertion sort is in Listing 2 on lines 7-34. The worst case for insertion sort is when the list is in reverse order. In all cases, the outer loop on line 7 will always run n times. However, when the list is in reverse order, every element that gets compared to in the inner loop will be greater than the element being sorted, which means that the inner loop will run until $j < 0$ so the item gets placed in the front of the array. As explored in Section 1.2, the total number of iterations in the case of a reversed list for insertion sort will be $\sum_{k=1}^{n-1} k$, which solves to be an $O(n^2)$ runtime. However, the best case for insertion sort is when the array is already sorted. In this situation, the outer loop will still be run, but the inner loop will never be entered because the element being sorted is always going to be greater than or equal to the element in the position before it. Therefore, since the array is only being iterated through from the outer loop, the runtime of insertion sort improves to $\Omega(n)$.

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 previously mentioned, 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, 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.

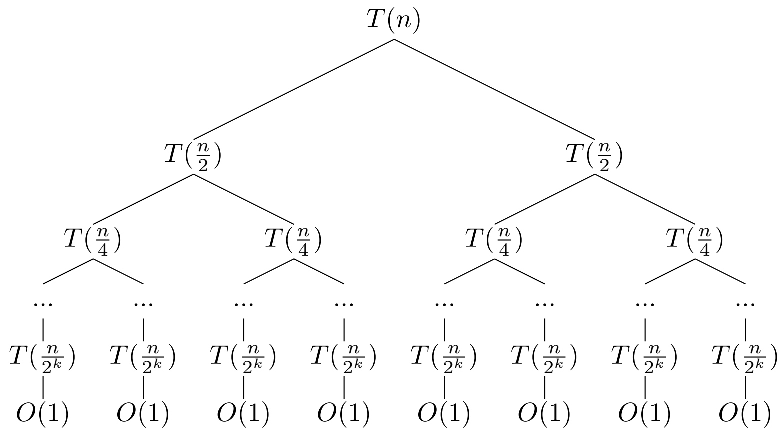
3.2 Asymptotic Analysis and Comparisons

The C++ implementation of merge sort can be found in Listing 3, more specifically lines 6-28. As shown on line 14, the array is always split in half for each successive call of merge sort until the subarray is of length 1. The recursion tree for merge sort is displayed below.

Algorithm 3 Merge Sort Algorithm

```
procedure MergeSort(arr)
    if length(arr) > 1 then    // An array of size 1 is already sorted
        mid = floor((length(arr))/2)    // Get the middle index of the array for splitting it in half
        MergeSort(arr[0 : mid])    // Perform merge sort on the first half of the array (index 0 - mid, inclusive)
        MergeSort(arr[mid + 1 : length(arr) − 1])    // Perform merge sort on the second half of the array
        Merge(arr, mid)    // Merge the 2 subarrays together in order
    end if
end procedure

procedure Merge(arr, mid)
    leftIndex ← 0    // Index for the left subarray
    rightIndex ← mid + 1    // Index for the right subarray
    newArr ← [ ]
    for i ← 0, length(arr) − 1 do    // Iterate through all elements
        if rightIndex >= length(arr) then    // All the right subarray items are already in newArr
            newArr[i] = arr[leftIndex]    // Add the next item from the left subarray
            leftIndex + +
        else if leftIndex > mid then    // All the right subarray items are already in newArr
            newArr[i] = arr[rightIndex]    // Add the next item from the right subarray
            rightIndex + +
        else if arr[leftIndex] < arr[rightIndex] then    // The next element from the left subarray is less than the next element from the right subarray
            newArr[i] = arr[leftIndex]    // Add the next item from the left subarray
            leftIndex + +
        else
            newArr[i] = arr[rightIndex]    // Add the next item from the right subarray
            rightIndex + +
        end if
    end for
    for j ← 0, length(arr) − 1 do
        arr[j] ← newArr[j]    // Transfer the sorted elements to the original array
    end for
end procedure
```



As shown in the recursion tree, the array is broken up into smaller subarrays until the subarray is small enough that it is solved using a constant time operation. Additionally, the last call is on an array of size $\frac{n}{2^k}$. Since the array is being divided in half each time, k is the number of levels in the tree, which is $log_2 n$.

Next, after the 2 subarrays are sorted, they are merged together for the conquer phase, which is on lines 30-80 of Listing 3. The beginning of the function (lines 32-40) are all assignments, which run in constant time. Next, there is a for loop that iterates through the length of the subarray being merged. The body is a large if-else block with comparisons and assignments, which is all $O(1)$ and makes the loop it is contained in run in $O(n)$ time. Lastly, lines 74-76 define a loop that iterates through each element in the subarray, which is also $O(n)$. Therefore, the runtime for the merge function is about $2n + 1$, which is $O(n)$.

At each level of the tree, the merge function will work with exactly n elements. For instance, in the first level of the tree, the left subarray will be merged from 2 subarrays of size $\frac{n}{4}$, which means that the merge function will work with $\frac{n}{2}$ elements. However, since there are 2 subarrays of size $\frac{n}{2}$, the merge function will be called again to handle the other subarray and, therefore, the level will end up merging n elements. Thus, one can multiply n by the number of levels to get the runtime complexity, which 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
```

```cpp
    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

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

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

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

    // Sort the first half
    mergeSortWithIndices(data, start, mid, comparisons);

    // Sort the second half
    mergeSortWithIndices(data, mid + 1, end, comparisons);

    // Merge both halves
    merge(data, start, end, mid, comparisons);
}

void merge(StringArr* data, int start, int end, int mid, int* comparisons) {
```

```cpp
    // The left half is at the start
    int leftIndex = start;

    // The right half starts at the midpoint + 1
    int rightIndex = mid + 1;

    // Get the size of the array that the 2 halves will merge into
    // and create the merged sub array
    int subArrLength = end - start + 1;
    std::string newSubArr[subArrLength];

    // Iterate through the entire merged subarray
    for (int i = 0; i < subArrLength; i++) {
        // If the rightIndex > end, then the entire right half is already merged
        if (rightIndex > end) {
            // Add the next element from the left half
            newSubArr[i] = data->arr[leftIndex];
            leftIndex++;
        } else if (leftIndex > mid) { // If the leftIndex > mid, then the entire left half is already merged
            // Add the next element from the right half
            newSubArr[i] = data->arr[rightIndex];
            rightIndex++;
        } else if (data->arr[leftIndex].compare(data->arr[rightIndex]) < 0) { // Compare the 2 elements from each half
            // Add the next element from the left half
            newSubArr[i] = data->arr[leftIndex];
            leftIndex++;

            // Increment the number of comparisons made
            if (comparisons != nullptr) {
                (*comparisons)++;
            }
        } else {
            // Add the next element from the right half
            newSubArr[i] = data->arr[rightIndex];
            rightIndex++;

            // Make sure to increment comparisons because a comparison was made in the last else-if condition
            if (comparisons != nullptr) {
                (*comparisons)++;
            }
        }
    }

    // Transfer the merged subarray to the actual array
    for (int j = 0; j < subArrLength; j++) {
        data->arr[start + j] = newSubArr[j];
    }
}

void quickSort(StringArr* data, int* comparisons) {
    // Run the helper function for the entire array
    quickSortWithIndices(data, 0, data->length - 1, comparisons);
}
```

Listing 3: Merge Sort (C++)