Assignment Three

Josh Seligman

joshua.seligman1@marist.edu

October 19, 2022

1 Linear Search

1.1 The Algorithm

Linear search is a searching algorithm that walks through an array and continues on until either it finds the target element or reaches the end of the array. As shown in Algorithm 1, the function has to compare the target value with every element in the array until the condition in the while loop becomes false. Since the entire array is being searched, no assumptions have to be made about the initial status of the array, which means that the array does not have to be sorted or in any particular order prior to running the search.

Algorithm 1 Linear Search Algorithm

```
1: procedure LinearSearch(arr, target)
2:     i ← 0    // Start at the beginning of the array
3:     while i < len(arr) && arr[i] != target do    // Search through the entire array or until the target is found
4:         i++
5:     end while
6:     if i == len(arr) then
7:         i = −1    // Set i to -1 to note that the target is not in the array
8:     end if
9:     return i
10: end procedure
```

1.2 Asymptotic Analysis and Comparisons

As mentioned in Section 1.1, performing a linear search requires going through each element until the target is found or until the end of the array is reached. Also, worst case, the number of iterations will be equal to the number of elements in the array. Listing 1 provides the C++ implementation of linear search and demonstrates the need to iterate through the entire list with its loop on lines 6-12. Therefore, as its name implies, linear search runs in linear time $O(n)$.

2 Insertion Sort

2.1 The Algorithm

Insertion sort in a sorting algorithm that places an element in its sorted position by sliding previously sorted elements over until the sorted position is found. 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

```
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 and Comparisons

The code implementation of insertion sort is in Listing **??** on lines 4-33. The worst case for insertion sort is when the list is in reverse order. In this scenario, 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 **??**, 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 5.1, 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 **??**) 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 sorted, 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

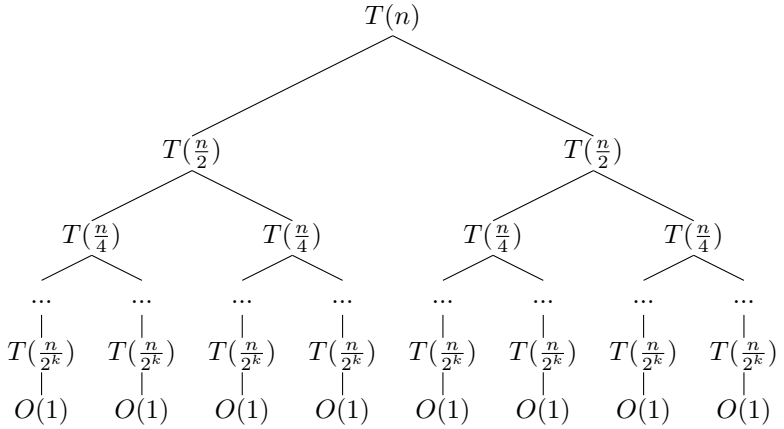
3.1 The Algorithm

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.

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
```

Figure 3.1: Recursion tree for merge sort.



3.2 Asymptotic Analysis and Comparisons

The C++ implementation of merge sort can be found in Listing **??**, more specifically lines 6-24. As shown on lines 17 and 20, 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 in Figure 3.1.

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 26-74 of Listing **??**. The beginning of the function (lines 28-36) 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 71-73 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 5.1, merge sort is significantly more efficient at sorting the magic items by using around 5,400 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 41 and 45 in Listing **??** 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 Quicksort

4.1 The Algorithm

Quicksort is similar to merge sort in that it is a divide and conquer sorting algorithm. Rather than dividing an array in half and then eventually merging the 2 subarrays back together, quicksort divides an array into 2 partitions around a pivot value so that all elements in the left partition are less than the pivot value and all elements in the right partition are greater than the pivot value. In other words, the elements are getting sorted as they are being divided up, which will result in the entire array being sorted once there are n subarrays of size 1. The performance of quicksort is also dependent on the method used to choose the pivot, which will be discussed in detail in Section 4.2.

Algorithm 4 Quicksort Algorithm

```
1:  procedure QUICKSORT(arr)
2:      if length(arr) > 1 then    // An array of size 1 is already sorted
3:          p ← choosePivotIndex(arr)    // More details on how to choose a pivot element in Section 4.2
4:          newPivotIndex ← Partition(arr, p)    // Partition the array around the pivot
5:          Quicksort(arr[0 : newPivotIndex − 1])    // Run quicksort on the left partition
6:          Quicksort(arr[newPivotIndex + 1 : length(arr) − 1])    // Run quicksort on the right partition
7:      end if
8:  end procedure
9:
10: procedure PARTITION(arr, pivotIndex)
11:     swap(arr, pivotIndex, length(arr) − 1)    // Move the pivot to the end of the array
12:     lastLowPartitionIndex ← −1    // Initially no elements are in the low partition
13:     for i ← 0, length(arr) − 2 do    // Iterate through all but the pivot
14:         if arr[i] < arr[length(arr) − 1] then
15:             lastLowPartitionIndex + +    // Found another element for the low partition
16:             swap(arr, i, lastLowPartitionIndex)    // Place the element in the low partition
17:         end if
18:     end for
19:     swap(arr, length(arr) − 1, lastLowPartitionIndex + 1)    // Place the pivot in the appropriate place
20:     returnlastLowPartitionIndex + 1
21: end procedure
```

4.2 Asyptotic Analysis and Comparisons

The C++ implementation for quicksort and partitioning can be found in Listing **??**. Since quicksort divides the array into 2 partitions and continues to work until the subarray is of size 1, the recursion tree for quicksort will be the same as merge sort (see Figure 3.1) assuming that the partitions are balanced and approximately the same size. Additionally, the work done to partition the array is $O(n)$ because it iterates through the entire array once (length of subarrays * number of subarrays in the level of the tree). Therefore, on average, the runtime complexity for quicksort will be the same as merge sort at $O(n * log_2 n)$.

However, as mentioned in Section 4.1, how the pivot is chosen will impact the runtime in the worst case scenario, which is an already sorted array or a reversed array. In these situations, if the pivot is either the first or last element in the array, the partitions would be of size 0 and size $n - 1$ because the pivot is either going to be greater than all the elements or less than all the elements. As a result, the algorithm would work like selection sort in that it would make sure the one element gets swapped into place and have to iterate through the rest of the array, which is not known to be sorted. Therefore, in these situations, quicksort downgrades to $O(n^2)$.

Therefore, the main problem with quicksort is how to guarantee somewhat balanced partitions. One solution is to use the median of 3 method to pick the partition. The implementation for this approach is found on lines 21-71 of Listing **??**. Instead of choosing 1 pivot, the median of 3 approach picks 3 random elements in the array for potential use as the actual pivot and then uses the middle element as the pivot. As a result, there will always be at least 1 element in each partition for all levels of the recursion tree. Even if the partitions are unbalanced, quicksort will still run in linearithmic time, but the logarithm will just have a different base to represent the uneven divide in the elements, and $O(n)$ work will still be done for partitioning at each level of the recursion tree. Also, as shown in the implementation of the median of 3 approach, only random number computations, assignments, and comparisons are needed, which all run in constant time and do not impact the overall runtime complexity of quicksort.

As shown in Table 5.1, both the number of comparisons and runtime are most similar to those for merge sort, at approximately $n * log_2 n$ comparisons, which lines up with the algorithm's complexity of $O(n * log_2 n)$. The number of comparisons for quicksort, however, is greater than that for merge sort for 2 main reasons: quicksort does not have an autocomplete on its partition like merge sort has for merging and there is a tradeoff with the number of comparisons made to prevent $O(n^2)$ runtime. First, since quicksort partitions the array when dividing, every element has to be compared to ensure it is placed in the correct partition. This is very different from merge sort, which just places the rest of a subarray into the merged array when the other subarray being merged is completely merged (lines 41-49 of Listing **??**). Also, as shown in lines 36-71, several comparisons have to be made to pick a pivot that prevents quicksort from downgrading to $O(n^2)$, which is a tradeoff that is made because of how much more efficient $O(n * log_2 n)$ is compared to $O(n^2)$. Also, despite having more comparisons than merge sort, quick sort often ran in less time. Although constants are dropped when doing a Big-Oh analysis, the merge function has much larger constants than the partition function for quicksort. As shown in the merge function on lines 26-74 of Listing **??**, there are 2 loops that iterate through the length of the merged subarray, which means that merge sort actually does $2n$ work for each level of the recursion tree. On the other hand, quicksort's partition function on lines 82-114 of Listing **??** only iterates through the array once and does n work at each level of the tree, which can save more time as the size of the array increases.

5 Appendix

5.1 Comparisons Table

Algorithm	List	Comparisons	Time
Selection Sort	666 magic items, shuffled	221445	19916376 ns
	20 Yankees greats, sorted	190	12564 ns
	20 Yankees greats, reversed	190	12104 ns
Insertion Sort	666 magic items, shuffled	112474	8952454 ns
	20 Yankees greats, sorted	19	1586 ns
	20 Yankees greats, reversed	190	13012 ns
Merge Sort	666 magic items, shuffled	5404	1069105 ns
	20 Yankees greats, sorted	48	9518 ns
	20 Yankees greats, reversed	40	6164 ns
Quicksort	666 magic items, shuffled	8092	951497 ns
	20 Yankees greats, sorted	72	8052 ns
	20 Yankees greats, reversed	75	8463 ns

Table 5.1: A table of the number of comparisons made and time to complete each sort on a variety of lists.

5.2 Linear Search

```cpp
int linearSearch(StringArr* data, std::string target, int* comparisons) {
    // Start with the first element in the array
    int i = 0;

    // Iterate through the array, looking for the target
    while (i < data->length && data->arr[i].compare(target) != 0) {
        if (comparisons != nullptr) {
            // Increment the number of comparisons
            (*comparisons)++;
        }
        i++;
    }

    // Default to -1 as the output if the target is not in the array
    int out = -1;

    // Add a comparison and set the index because we found the element
    if (i < data->length) {
        (*comparisons)++;
        out = i;
    }

    // Return the position of the target element
    return out;
}
```

Listing 1: Linear Search (C++)