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

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

2.1 The Algorithm

Binary search is a searching algorithm that takes an already sorted list and progressively cuts it in half until there is only 1 element left, which is the one that is being searched for. As shown in Algorithm 2, each recursive call on lines 8 and 10 makes the problem smaller by moving the start or stop limits to a single side of the midpoint, which effectively cuts the array in half at each level of the recursion tree. Also, Figure 2.1 illustrates how binary search divides the array in half to eventually find the target element.

Algorithm 2 Binary Search Algorithm

```
1:  procedure BinarySearch(arr, target, start, stop)
2:      out ← −1    // Assume the element is not found, by setting the default output to -1
3:      if start <= stop then    // Working in a valid range
4:          mid ← ⌊(start+stop)/2⌋    // Get the middle of the range
5:          if target == arr[mid] then
6:              out ← mid    // Target found at position mid
7:          else if target < arr[mid] then    // Target is in bottom half of the array
8:              out ← BinarySearch(arr, target, start, mid − 1)    // Do binary search on lower half of array
9:          else    // Target is in top half of the array
10:             out ← BinarySearch(arr, target, mid + 1, stop)    // Do binary search on top half of array
11:         end if
12:     end if
13:     return out
14: end procedure
```

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Figure 2.1: A visualization of the binary search algorithm. The blue shaded areas are the parts of the array being considered at each step of the recursion tree until there is only the target element left.

2.2 Asymptotic Analysis and Comparisons

Binary search, similar to merge sort and quicksort, divides the array in half at each step of the recursion tree. This makes the recurrence relation of binary search to be $T(n) = T(\frac{n}{2}) + C$, where $T(\frac{n}{2})$ is the time to perform binary search on the half of the array and C is the constant time to perform the comparisons. Thus, since the array is being divided in half until the size of the array being considered is 1, the number of times binary search will be recursively called at worst case will be $log_2 n$ times, which makes it run in $O(log_2 n)$ time.

$O(log_2 n)$ is a huge improvement over $O(n)$ for linear search. However, the one tradeoff is that binary search needs to have an already sorted array. Thus, if the array is not sorted, an additional $n * log_2 n$ time will need to be added to put the array in a state to use binary search. Therefore, linear search may be the better option for one and done search operations, but binary search with a sort will catch up as the number of search operations increases. The mathematical equation to determine this point is $x * n > x * log_2 n + n * log_2 n$, where x is the number of searching operations that have to be done on the unsorted array.

3 Appendix

3.1 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 3.1: A table of the number of comparisons made and time to complete each sort on a variety of lists.

3.2 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++)