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, $target$)
  $i \leftarrow 0$ // *Start at the beginning of the array*
  while $i < len(arr)$ && $arr[i] \mathrel{!}= target$ **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, 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 moves 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.

Algorithm 2 Binary Search Algorithm

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

2.2 Asymptotic Analysis and Comparisons

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

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

```cpp
    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++)