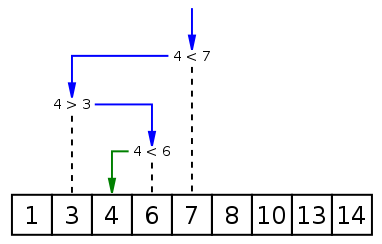
Binary Search Assignment



Kshitij Shah

March 22, 2017

ICS4U1

M. Berry

Introduction

Binary search is a way to search through sorted data to locate very efficiently a specific search term. Linear search is a search that sequentially compares the search term to all elements of the data in an array, if the list is too long, it’s a less efficient method of searching, but the benefit of the linear search is that the data does not have to be sorted. The program detailed in the following report uses the example of a text file containing 1500 phone numbers sorted in ascending numerical order to count the number of comparisons require to execute each type of search for various cases to see which method is more efficient different situations. This report contains the code tested, explanations of values and a discussion of the benefits and drawbacks of both search types.

Testing

I tested the program using a variety of numbers to check if both searches found the number on the right line in an appropriate number of comparisons. I provided some reasoning as to why I chose some numbers.

For best-case testing, I predicted because of the way linear search works, the optimal search term was the first element of the array. To confirm, I checked the number of comparisons the linear search ran by having a running total global integer counter that incremented every time the loop ran. By manually checking the 1st, 100th, 200th, 300th… 1500th and checked the value of the comparison counter I could confirm the relationship is linear, and the first element did take the fewest comparisons (2). For binary search, the algorithm suggested that the middle term (n/2) would be the ideal scenario. By running the search, and checking the counter, I determined it took fewer searches compared to any other value in the list (n/4 or 3n/4 were the second most efficient). It took 2 comparisons for the n-1/2 element.

For worst case, I predicted for both, an element not present in the array would be the worst-case scenario, as it would have to check to the last possible element as each search would, and then compare one more to determine that it does not exist in the array. The alternative was the last element in the array for both (1st element would be also equivalent in binary search). To test, I ran both searches on the last element, and a not found element and determined that the not found element resulted in 1 more comparison in both searches. In linear, it took 1501 comparisons, and 13 in binary search. Both numbers are greater than when it’s the last element of the array, which took 1500 and 12 comparisons respectively, this fact was confirmed by an internet search.   
  
For average case, linear was simple to determine. It was equally likely that the search term would be present on the second half as it was to be on the first half of the array, because of this a simple calculation was done to find the average index value, 749. This resulted in 751 comparisons, an average amount when compared to searching for the first and last element. Binary search presented some difficulty in determining the average case scenario. The more times the method runs, the greater the number of elements that become accessible to the search to find. The fact that every time the method runs, if you were to make a tree of the process, every iteration allows twice as many values compared to previous run. Therefore, the average case was the maximum number of iterations that can be run without resulting in a not found. Since every iteration divides the array in half, 2n = # elements in array, n being the number of comparisons. Therefore, log2(n) = # of comparisons, log2(1500) = 10.55 = ~11. This means that >50% of the elements are only accessible after 11 comparisons. An example of this would be the last element in the array, as to arrive at it, it could not be the result of the division of a sub array, but as a result of the algorithm checking it as a last possible element, taking all 11 comparisons.

Discussion Questions:

*1. How large was the data file (# records)?*

The data file was 1500 lines long, containing a unique phone number on each line in ascending numerical order. This was pre-sorted, making it a good data file to use binary search with. This, for a modern computer is not a very large set of data, meaning both linear and binary search are completed very quickly.

*2. What was the best-case scenario for each search type?*

Linear:

Linear search runs through the array comparing each element from start to finish to search term, this means the earlier in the in the list the match occurs, the fewer comparisons must be done by the program. This means the best-case scenario for the linear search would be if the search term (phone number) matched the first element of the array. In this case, that would be ‘905-200-2301’ at line 1.

Binary:

Unlike linear, binary search divides the array into two parts, and jumps directly to the middle element of the search area in an ordered set of data. It then determines whether to look through the second or first half by comparing search term to element in the middle of the search area. If there are no more elements in the array to search, it returns the number wasn’t found, if it finds the number, it returns the location where it was found, otherwise it runs binary search again on the further subdivided section of the array. The best-case scenario is the first one the search compares to search term, which is the middle term in the array, which is array length – 1 /2. In this database of 1500 lines, it would be the 749th element, the phone number ‘905-542-2924’ at line 750.

*3. What was the worst-case scenario for each search type?*

Linear:

Linear search makes comparisons until it finds the search term, so if never finds the search term, it must compare search term with every element of the array, then determine it does not exist in the array. This means the worst-case scenario would be a search term (phone number) that isn’t found in the array, for example in this case ‘999-555-0100’, a number reserved to be fake for use in movies or other fiction media. One could also make it the last number + 1, as the list is organized ascending order, so if it’s the last number + 1, it doesn’t already exist in the list. This is more of an arbitrary worst case for ordered list using linear search. This would take the most number of comparisons, in my program, 1501 comparisons.

Binary:

Like linear search, binary search takes the most comparisons when it searches for a number that does not exist in the array, because it would have to go to through all divisions until it gets to one element in the array and is still not found. This number can be generated similarly as done for linear, by either manually inputting a non-existent phone number, or taking the last phone number in the sorted list and adding one to it, as it would not be in the list. This would certainly work for binary search, because to use binary search, the data must be sorted. That however is not the case for linear and unless the data is sorted, you wouldn’t know what the highest or lowest values are. In my testing, I found testing a not found number resulted in 13 comparisons.

*4. What was the average case scenario for each search type?*

Linear:

The average case for linear search is very easy to figure out, as it’s the middle element of the array, because it is equally likely the number be earlier in the array, or later. Therefore, the average case would be the number of lines/2. In this case, that number was ‘905-542-2924’ at line 750 (1500/2).

Binary:

The average case scenario for binary search is a lot more complicated than in linear search, because the relationship of the number of comparisons to the input is not linear, but rather logarithmic. Because of this, every iteration allows there to be twice as many elements accessible to the search, so therefore, at the last (11th) iteration, >50% of the values are possible accessible. This means that a run that takes 11 comparisons would be the average case. This could be a lot of numbers, easiest to determine would be the last in the array, as I previously explained, it would not be arrived at unless it is the last possible element to be checked. The index then of array[array.length-1] would take 11 comparisons.

*5. Describe how you determined the average case scenario. Use external resources if desired.*

Linear:

It’s very easy to determine the average case for linear search because the number of comparisons increases linearly with the amount of data to be searched through, therefore the average case for “n” records it n/2 comparisons, with the search term being located half way through the data.

Binary:

As mentioned previously, the average case for binary is more complicated than it is with linear. Because with every iteration of the recursive method, more elements are accessible to be found, I believed that the number should be greater than half of the maximum comparisons. Then I made a tree of the possible indexes the binary search could access every iteration, and I’ve charted it out:

|  |  |  |
| --- | --- | --- |
| **# of Comparisons (n)** | **# of Accessible Elements (2n)** | **% of Elements Accessible (out of 1500)** |
| 1 | 1 | 0.06% |
| 2 | 2 | 0.13% |
| 3 | 4 | 0.26% |
| 4 | 8 | 0.53% |
| 5 | 16 | 1.06% |
| 6 | 32 | 2.13% |
| 7 | 64 | 4.26% |
| 8 | 128 | 8.53% |
| 9 | 256 | 17.06% |
| 10 | 512 | 34.13% |
| 11 | 1024 | 68.26% (>50%) |

*6. Is it always more efficient to sort a set of data and perform a binary search on it than to simply do a linear search on the unsorted data?*

No, it’s not, because if the data is unsorted, to run binary search, one must first sort the data, as efficiently as possible (merge sort usually) then run binary search through it. If the number of comparisons it takes to do the sort and search is greater than the length of the array, then it is impractical, and would be better to just run linear search. Doing the linear search in that case would be both easier on the hardware and the programmer. If there is a lot of data, then it becomes inefficient to use linear search as both merge sort and binary search have nlog(n) and log(n) comparisons to run respectively. This means that the number of comparisons doesn’t get linearly increased as the input size increases. Basically, if the length of the array is greater than the value of 2(nlog(n)), then it’s more efficient to run sort and search instead of linear search. If the plan is to increase the amount of data, it might be a good idea to sort the current amount and resave that to run binary search on later, when it further expands, even if at the moment linear is more practical.