Jun Ha (Andy) Lee

916007621

Program 1: Report

Data

# of books (n)	# of requested books (m)	Linear search time	Binary search time
50	10	125 microseconds	152 microseconds
50	50	594 microseconds	242 microseconds
200	10	476 microseconds	690 microseconds
200	100	4780 microseconds	890 microseconds
200	200	6400 microseconds	1160 microseconds
1000	10	2430 microseconds	4143 microseconds
1000	500	24800 microseconds	5760 microseconds
1000	1000	44700 microseconds	5950 microseconds
2000	10	4910 microseconds	6100 microseconds
2000	1000	85000 microseconds	6627 microseconds
2000	2000	163500 microseconds	7400 microseconds
100000	10	430000 microseconds	1.38e+06 microseconds

Linear Search

```
111  int linearSearch(const std::vector<Book>& bookList,
112          const std::vector<Book>& requestList) { // linear search
113      int count = 0;
114      for (unsigned int i = 0; i < requestList.size(); i++) { // iterate each request book list
115          for (unsigned int j = 0; j < bookList.size(); j++) { // find request book from book list
116              if (compareBook(bookList[j], requestList[i])) { // if two books are same
117                  ++count; // if two books are same increase count by one
118                  break; // break for loop to not count duplicates
119              }
120          }
121      }
122      return count;
123  }
```

The linear search runs as a nested for loop by iterating through each request list and then by finding each request through a for loop of book list. As lines 115 to 120 compares the request book against every book it has the run time of O(n) where n is the size of the book list. However, this for loop will be iterated for every request which is on line 114. The for loop on line 114 will

have O(m) where m is the size of the request book list. However, since it is a nested for loop, the total run time of linear search would be O(n*m).

Looking at the data, it can be seen that the time taken is proportional to the size of the book list or request list agreeing to O(n*m). For example, comparing the case n=1000 and m=10 with n=2000 and m=10, the time taken to complete each search doubled as n doubled. For another example, comparing the case n=1000 and m=500 with n=1000 and m=1000, the time taken to complete each search also doubled as m doubled. From these examples, it can be seen that O(n*m) holds true.

Binary search

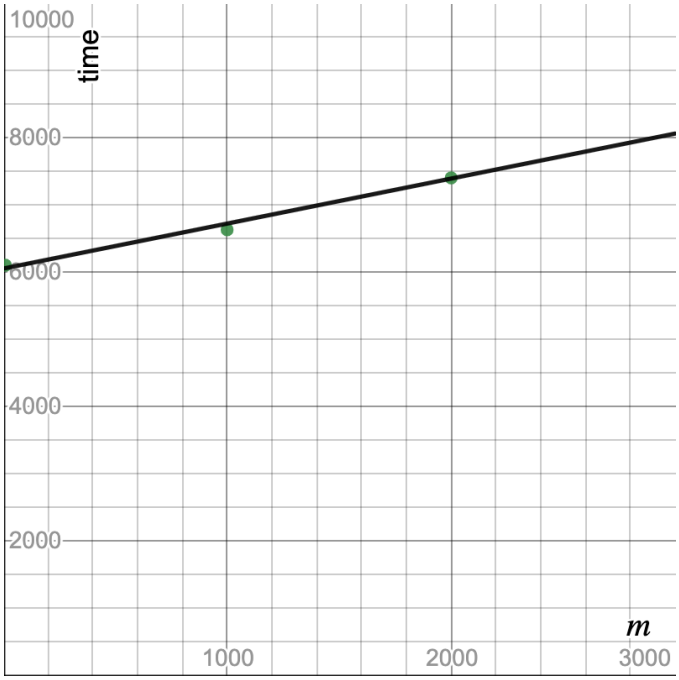
```
125  int binarySearch(const std::vector<Book>& bookList,
126          const std::vector<Book>& requestList) { // binary search
127      int count = 0;
128      for (unsigned int i = 0; i < requestList.size(); i++) { // iterate each request book list
129          int left = 0; // left most index
130          int right = bookList.size()-1; // right most index
131          while (left < right) { // stops when there is one book left to read
132              int m = (left+right)/2; // find the middle index of sorted list
133              if (bookList[m] < requestList[i]) { // compare two books
134                  left = m + 1; // search right half list
135              } else {
136                  right = m; // search left half list
137              }
138          }
139          if (compareBook(requestList[i], bookList[left])) {
140              ++count; // if two books are same increase count by one
141          }
142      }
143      return count;
144  }
```

Binary search (line 131 to line 138) itself without sorting algorithm has a time complexity of O(log n) as a vector of book list is divided into two until there is one book remaining. For example, if we had a vector of books with size n=16, in our worst case, we would look if the book is in the upper half or lower half of the list resulting in a vector with size 8. It would repeat to have size 4, size 2, and size 1 which would give us the result. This results in $\log_2(16)=4$ steps for Binary search. However, we would iterate line 131 to line 138 for every request list of size m times shown in line 128. This would mean that the time complexity for lines 125 to 144 would be O(mlogn).

Additionally, binary search can only happen when the list is sorted. Therefore, the time complexity of sorting is also included. However, O(mlogn) would have a higher time complexity than sorting, resulting in O(mlogn).

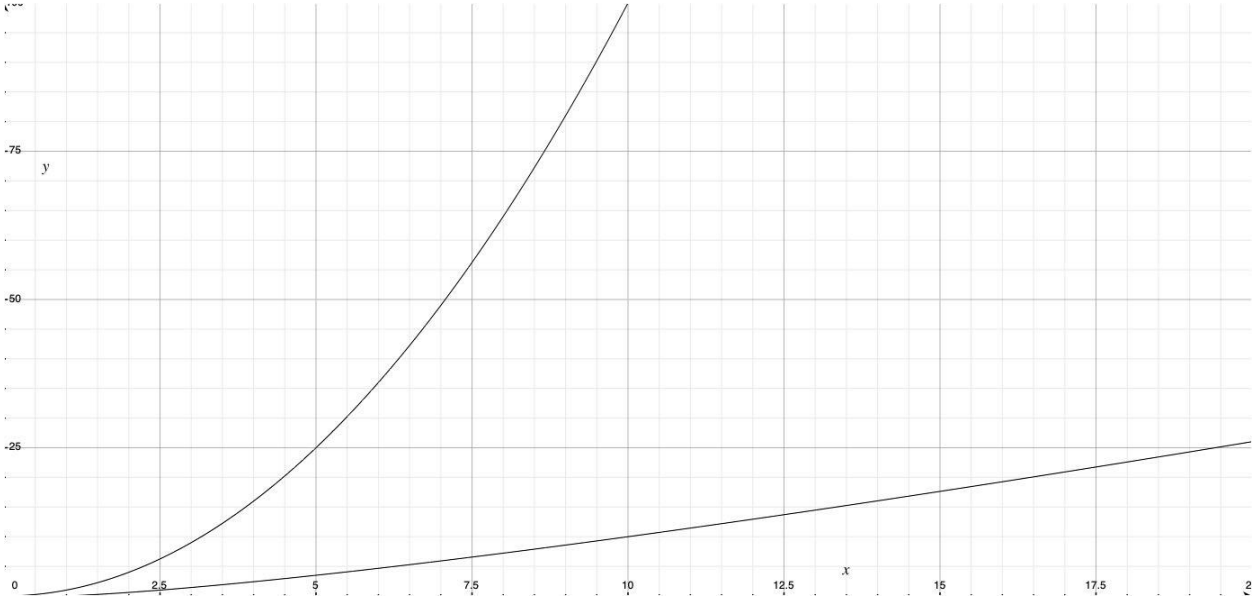
While harder to show proportional relationships in binary search, the time taken to for each search with the same n with different m did not differ as much. This is due to the time to sort a list of n. Once sorted, the time would not differentiate as much as linear search as the time complexity of the search is O(mlogn).

For example, if we look at cases with n=2000, the time should be proportional to $m_2\log(2000)/m_1\log(2000)$. Therefore, if we consider the time to sort as b, we should create a linear relationship of time = ax+b which is shown by the graph below.



Comparison

From the data set, it can be seen that the linear search is faster than binary search for cases where the size of requested books m<50. However, the difference quickly increases for a larger number of m. This can be further explained by the graph below of nlog(n) and n^2.



As seen above, the difference between nlog(n) and n^2 increases for a higher number of n. For a small number of n, the time for linear search would be faster than binary search as binary search would need sorting.

In conclusion, the linear search would be more valuable if we are requesting a small number of books. On the other hand, the binary search would be more valuable if we are requesting to find

a large number of books. Even for a large number of books, if we request only a small number of books, the linear search would be faster. For example, if we have a case of n=1000000 and m=10, the time would still be faster for linear search as seen in the data table. In a real-life context, I would expect the user to request a small number of books from the library meaning that linear search would be more efficient. However, it would be beneficial for the library to maintain a sorted catalog of the books. Considering that a sorted list of books already exists, it would be much faster for a binary search as the time complexity is O(mlogn) compared to O(m*n).