Assignment 2 - LaTeX Write Up

Connor Fleischman

November 1, 2024

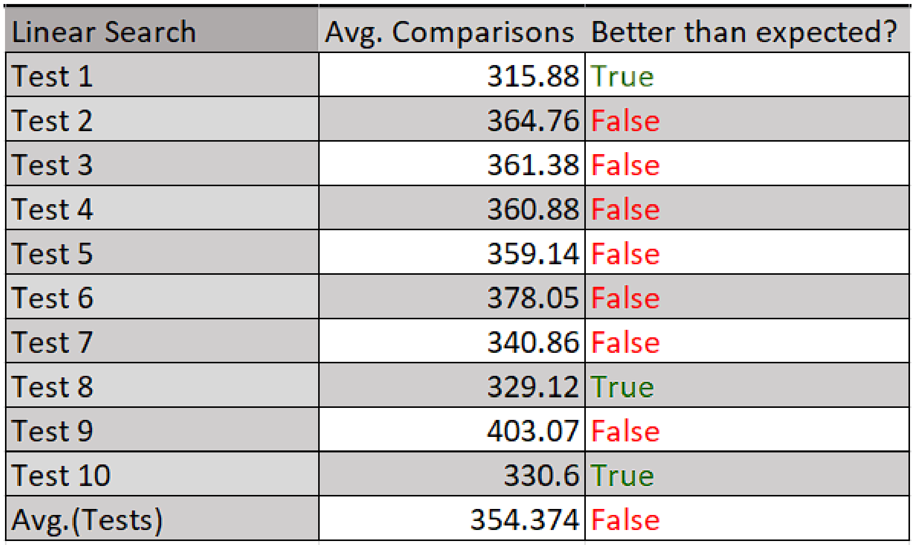
MARIST COLLEGE
ORARE ET LABORARE
1929
POUGHKEEPSIE, N.Y.

1 Assignment Results

Assignment 02 was to perform different searches on the magic items text file provided in Assignment 01 for 42 random keys, picked from magic items.

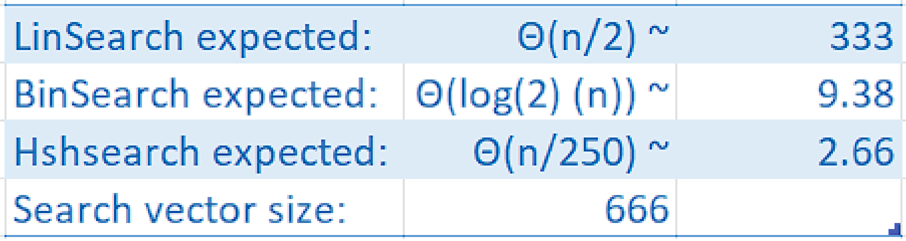
1.1 Linear Search

The first search to be performed was a linear search. This searching method takes a key and sequentially picks through each magic item, comparing the current item to the key, until the key is found. After implementing this in C++ and ensuring that my algorithm is correct, 10 tests were performed and their data recorded below.



Linear Search	Avg. Comparisons	Better than expected?
Test 1	315.88	True
Test 2	364.76	False
Test 3	361.38	False
Test 4	360.88	False
Test 5	359.14	False
Test 6	378.05	False
Test 7	340.86	False
Test 8	329.12	True
Test 9	403.07	False
Test 10	330.6	True
Avg.(Tests)	354.374	False

This table diagrams the average time each test took to find all 42 keys in magic items. It also depicts how efficient the algorithm was. Before we continue, it is important clarify a few prerequisites.

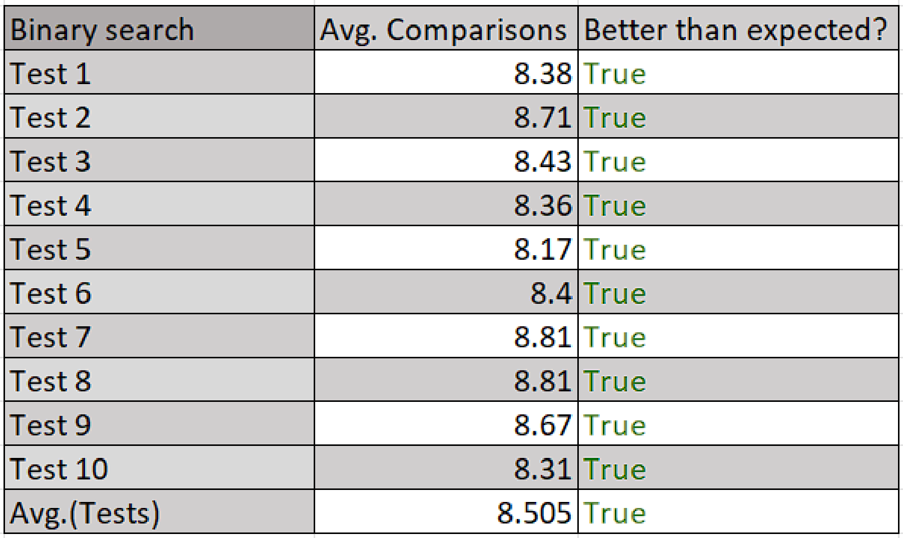


LinSearch expected:	Θ(n/2) ~	333
BinSearch expected:	Θ(log(2) (n)) ~	9.38
Hshsearch expected:	Θ(n/250) ~	2.66
Search vector size:	666	

With these in mind, the ”Better than expected?” column now has a basis to compare to. So over the performed tests, the data shows that only 30% of all searches for the 42 keys performed were more efficient than the expected value. Not an amazing look for my code, however only 10 tests were performed, who’s to say that if 100 tests took place it wouldn’t spread to 50%?

1.2 Binary Search

A binary search was the next task to conquer. This search takes a key and the middle value of the sorted magic items. Compares if the middle item is or is not the key, if it is not, then it compares if the middle item is larger or smaller than the key. Finally if the key is larger than the middle item, the search is ran again on the half greater than the middle. Otherwise the key is smaller and the search is ran on the half smaller than the key.



Binary search	Avg. Comparisons	Better than expected?
Test 1	8.38	True
Test 2	8.71	True
Test 3	8.43	True
Test 4	8.36	True
Test 5	8.17	True
Test 6	8.4	True
Test 7	8.81	True
Test 8	8.81	True
Test 9	8.67	True
Test 10	8.31	True
Avg.(Tests)	8.505	True

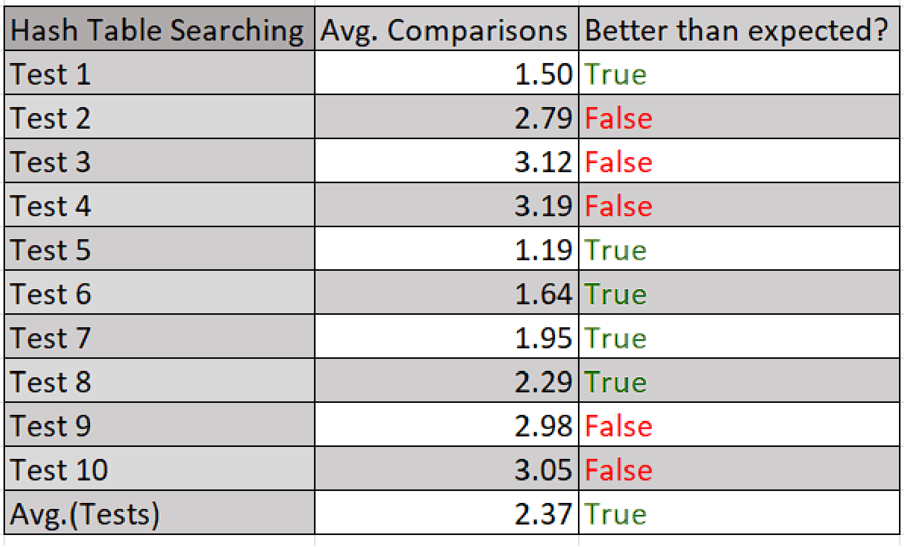
The same 10 tests were performed for binary search with each test’s keys being the same for each numbered test. Meaning, the keys used for test one in linear search were the same keys used for binary search (and for hash searching).

As shown above, the binary searching algorithm I implemented was quite efficient. But the same logic applied to linear search applies here, although 100% of tests were more efficient than the average case, does not mean that with more testing it can’t be inefficient.

1.3 Hash Table Search

The last search assigned was to take the magic items, create a hash table of 250 buckets, populate it with the items, and search for the keys using the hash table. A hash table is a 2-dimensional table where each value in the table has its own table of values.

So in relation to the problem, the hash table stores each magic item in a bucket corresponding to its ASCII value. This allows a search to narrow down the possible answers much more quickly than linear and binary search, leading to a much more efficient algorithm.



Hash Table Searching	Avg. Comparisons	Better than expected?
Test 1	1.50	True
Test 2	2.79	False
Test 3	3.12	False
Test 4	3.19	False
Test 5	1.19	True
Test 6	1.64	True
Test 7	1.95	True
Test 8	2.29	True
Test 9	2.98	False
Test 10	3.05	False
Avg.(Tests)	2.37	True

After conducting the same 10 tests, using the same 42 keys per tests, the data has shown that 50% of tests performed were above the average efficiency of a hash search. It also shows how efficient hash searching is compared to the other searching algorithms above. It would be interesting to see if the average of all tests would fall below the expected threshold if more tests were performed.

2 Code Breakdown

My code consists of three C++ documents. A ’main.cpp’ file housing the majority of the computations, and two assistive files, ’sortItems.h’ and ’buildHashTable.h’. These files are woven together through ’main.cpp’ so that when executed it prints out the results of the searches to the console. These files are in the ’./Assignment2/src’ folder.

2.1 Prerequisites

```cpp
#include "buildHashTable.h"
#include "sortItems.h"
#include <iostream>
#include <iomanip>

using namespace std; // Globally used namespace

int comparisonCount = 0;          // Declares comparisonCount as 0 to store the total number of comparisons made for a search
vector<string> sorted;            // Declares the global sorted vector of strings
vector<vector<string>> hashTable; // Declares a hash table comprised of a vector of vectors of strings
```

Beginning ’main.cpp’ we include ’buildHashTable.h’ and ’sortItems.h’ as well as ’iostream’ and ’iomanip’. These allow us to use functions like createTable() and sort.first() or .second() for our algorithms. They also allow for use of console output and floating point number precision respectively. We also use the std namespace as to not have to add ”std::_” to the code.

Globally scoped variables:
comparisonCount - the total number of comparisons for a search
sorted - a vector of strings to store the sorted magic items
hashTable - a vector of vectors of strings, or a vector of buckets of strings

2.2 Main

Our program begins through the main function below. Do not worry if this seems complex, we will break it down piece by piece.

```cpp
int main() // Defines the main function
{
    vector<string> keys = sort().first; // Defines the 42 key items being searched for (from shuffleItems.h)
    sorted = sort().second;             // Defines the sorted vector of items (from shuffleItems.h)

    for (const string &key : keys) // For every key
    {
        linearSearch(sorted, key); // Search the sorted vector for that key using a linear search
    }
    averageComparisons(); // Calculate the average number of comparisons made

    for (const string &key : keys) // for every key
    {
        binarySearch(sorted, key); // Search the sorted vector for that key using a binary search
    }
    averageComparisons(); // Calculate the average number of comparisons made

    sorted.clear();              // Empties the sorted vector
    hashTable = createTable(); // Builds a hash table (from buildHashTable.h) to hashTable

    for (const string &key : keys) // For every key
    {
        searchItem(key); // Search the hash table for that key
    }
    averageComparisons(); // Calculate the average number of comparisons made

    comparisonCount = 0; // Reset's the comparison count
    keys.clear();        // Empties the keys
    hashTable.clear();   // Empties the hash table

    return 0; // Return 0
}
```

When the executable is ran a vector of strings called keys is declared and instantiated as sort.first(). This comes from the 'sortItems.h' file which returns a pair, for more detail see section 2.3. Then sorted is instantiated as sort.second(), from 'sortItems.h', for more detail see section 2.3. A linear search is then performed for each key in keys using the sorted magic items. After, the average comparisons is calculated and produced for that search on each key. Next a binary search is used for the same keys as before to search through magic items. Again, the program calculates and produces the average number of comparisons for a key based off the 42 keys.

Finally, the code clears the sorted list and creates a hash table from 'buildHashTable.h' using createTable(). The same search is performed for the same 42 keys as prior, but instead of searching through the magic items, the program uses a hash table populated with the magic items. Then the average comparisons is found and output and the comparison count is reset to 0 and keys and the hash table are all cleared.

2.3 Sort Items

The main function firstly uses a function from 'sortItems.h'. Specifically, it uses the pair of return values given from the sort() method in 'sortItems.h'. It sets a vector of strings called keys to sort().first, and sorted, which was defined in section 2.2, as sort().second afterwards.

```
vector<string> lines;            // Declares the lines a vector of strings
vector<string> keys;             // Declares keys a vector of strings
int KEYS_SIZE = 42;              // Declares a key selection size of 42
ifstream file("magicItems.txt"); // Opens and reads the file

```

```cpp
vector<string> insertionSort() // Defines the insertionSort method
{
   int n = lines.size();       // n = the length of lines vector
   for (int i = 1; i < n; i++) // For every item i in the vector other than the first
   {
      int j = i;                      // Let j = i
      while (lines[j - 1] > lines[j]) // While the previous element is larger than the current
      {
         swap(lines[j], lines[j - 1]); // Swap the positions of the current and previous elements
         j--;                          // Decrement j
      }
   }
   return lines; // Returns the sorted array
}

pair<vector<string>, vector<string>> sort() // Comprised of the function calls to shuffle the items
{
   string line; // Declares line as a string

   while (getline(file, line)) // While there is a new line in the file
   {
      lines.push_back(line); // Add the new line to the vector lines
   }

   file.close(); // Closes the file

   srand(static_cast<unsigned int>(time(0))); // Seeds the random number generator with the time
   int start = rand() % 625;                  // Declares start as a random number between 0-624 (allowing for key space and out of bounds)

35     for (int i = 0; i < KEYS_SIZE; i++) // For 42 keys
36     {
37        keys.push_back(lines[i + start]); // Select the keys as 42 consecutive strings at some distance from the beginning of the list
38     }
39 
40     return make_pair(keys, insertionSort()); // Returns the sorted vector "lines"
41 }
```

As we can see on line 40, sort().first is keys and sort().second is the insertionSort() function. Keys is defined as a vector of strings which, after randomly selecting a starting value within the bounds of the data, we store the next 42 values as keys. This is done on lines 32-38.

Insertion sort is defined on line 6 and has a return value of a vector of strings. Insertion sort takes the unsorted magic items, written to a vector of strings, sorts it using an insertion sort, and returns it to sort().second. This is why we set 'keys = sort().first' (ln 3, sec 2.2) and 'sorted = sort().second' (ln 4, sec 2.2).

2.4 Insertion Sort

The second sorting method was Insertion sort which takes the list and compares each element one by one. Meaning the first element is compared to the second, if its greater than the second, swap, then go to the third element, if its less than the second swap, if its less than the first, swap again. This continues until the array is sorted

```
vector<string> insertionSort() // Defines the insertionSort method
{
   int n = lines.size();       // n = the length of lines vector
   for (int i = 1; i < n; i++) // For every item i in the vector other than the first
   {
      int j = i;                       // Let j = i
      while (lines[j - 1] > lines[j]) // While the previous element is larger than the current
```

```
        {
            swap(lines[j], lines[j - 1]); // Swap the positions of the current and previous elements
            j--;                          // Decrement j
        }
    }
    return lines; // Returns the sorted array
}
```

Note the asymptotic running time of each search and explain why it is that way. Including the asymptotic running time of hashing with chaining and explain why it is that way.

3 Final Thoughts

text