

# Assignment 2 - LaTeX Write Up

Connor Fleischman

November 1, 2024



# 1 Assignment Results

Assignment 2 was to perform different searches on the magic items text file provided in Assignment 1 for 42 random keys, picked from the 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:	$\Theta(n/2) \sim$	333
BinSearch expected:	$\Theta(\log(2) (n)) \sim$	9.38
Hshsearch expected:	$\Theta(n/250) \sim$	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 result 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 - main.cpp

My code consists of three C++ documents. A 'main.cpp' file housing the majority of the computations, and two assistive files, 'sortItems.h' and 'build-HashTable.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 './Assignment 2/src' folder.

## 2.1 Breakdown - main()

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

```
1 int main() // Defines the main function
2 {
3     vector<string> keys = sort().first; // Defines the 42
4     key items being searched for (from shuffleItems.h)
5     sorted = sort().second; // Defines the
6     sorted vector of items (from shuffleItems.h)
7
8     for (const string &key : keys) // For every key
9     {
10         linearSearch(sorted, key); // Search the sorted
11         vector for that key using a linear search
12     }
13     averageComparisons(); // Calculate the average number
14     of comparisons made
15
16     for (const string &key : keys) // for every key
17     {
18         binarySearch(sorted, key); // Search the sorted
19         vector for that key using a binary search
20     }
21     averageComparisons(); // Calculate the average number
22     of comparisons made
23
24     sorted.clear(); // Empties the sorted
25     vector
26     hashTable = createTable(); // Builds a hash table (
27     from buildHashTable.h) to hashTable
28
29     for (const string &key : keys) // For every key
30     {
31         searchItem(key); // Search the hash table for that
32         key
```

```

24     }
25     averageComparisons(); // Calculate the average number
                           // of comparisons made
26
27     comparisonCount = 0; // Reset's the comparison count
28     keys.clear();       // Empties the keys
29     hashTable.clear();  // Empties the hash table
30
31     return 0; // Return 0
32 }

```

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 of vectors, for more detail see section 3.1. Then *sorted* is instantiated as `sort.second()`, from 'sortItems.h', for more detail see section 3.1.

A linear search is then performed for each key in *keys* using the sorted magic items. Next, the average comparisons is calculated and produced for that search on each key. After 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.

## 3 Code Breakdown - Linear & Binary Search

Assignment 2 required us to perform a linear search and a binary search, using 42 randomly chosen keys, on the sorted list of magic items.

### 3.1 Breakdown - sortItems.h

The main function in section 2.1 uses a function from 'sortItems.h'. Specifically, it uses the pair of return values given from the `sort()` method. It sets a vector of strings called *keys* to `sort().first()`, and *sorted*, which was defined

in section 5.2, as `sort().second` afterwards.

```
1      int n = lines.size();           // n = the length of
      lines vector
2      for (int i = 1; i < n; i++) // For every item i in
      the vector other than the first
3      {
4          int j = i;                  // Let j = i
5          while (lines[j - 1] > lines[j]) // While the
      previous element is larger than the current
6          {
7              swap(lines[j], lines[j - 1]); // Swap the
      positions of the current and previous elements
8              j--;                      // Decrement j
9          }
10     }
11     return lines; // Returns the sorted array
12 }
13
14 pair<vector<string>, vector<string>> sort() // Comprised
      of the function calls to shuffle the items
15 {
16     string line; // Declares line as a string
17
18     while (getline(file, line)) // While there is a new
      line in the file
19     {
20         lines.push_back(line); // Add the new line to the
      vector lines
21     }
22
23     file.close(); // Closes the file
24
25     srand(static_cast<unsigned int>(time(0))); // Seeds
      the random number generator with the time
26     int start = rand() % 625;                //
      Declares start as a random number between 0-624 (
      allowing for key space and out of bounds)
27
28     for (int i = 0; i < KEYS_SIZE; i++) // For 42 keys
```

```

29 {
30     keys.push_back(lines[i + start]); // Select the
    keys as 42 consecutive strings at some distance from
    the beginning of the list
31 }
32
33 return make_pair(keys, insertionSort()); // Returns
    the sorted vector "lines"
34 }

```

As we can see on line 35, `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 27-33.

Insertion sort is defined on line 1 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 2.1) and '`sorted = sort().second`' (ln 4, sec 2.1). Finally in the `sort()` function we read the file '`magicItems.txt`' into a vector of strings called *lines* for manipulation.

## 3.2 Breakdown - `linearSearch()`

Below is my implementation of a linear search algorithm which takes a searching vector and a key. This function sequentially searches each item in the vector until it finds the key, incrementing a counter for each comparison made. Then when the value is found it will output the number of comparisons performed to find it and add that to the comparison count.

If the value is not found, then it will not increment the comparison count and simply return that the value was not found after being compared to every other value.

```

1 string linearSearch(const vector<string> &lines, const
    string &key) // Defines the linear search function
2 {
3     int numComparisons = 0; // Defines the number of
    comparisons to be 0
4
5     for (int position = 0; position < lines.size(); ++
        position) // For every element in lines

```



```

6      {
7          numComparisons++; // Increment numComparisons
8
9          if (lines[position] == key) // If the current
            element is the key
10         {
11             cout << "Number of comparisons: " <<
numComparisons << endl; // Print to console the
            number of comparisons
12             comparisonCount += numComparisons;
                // Increment comparison count with the
            number of comparisons made to find key
13             return lines[position];
                // Return the found item
14         }
15     }
16     // If element is not found:
17     cout << "[NF] Number of comparisons: " <<
numComparisons << endl; // Print to console the
            number of comparisons
18     comparisonCount += numComparisons;
                // Increment comparison count with the
            number of comparisons made to find key
19     return "";
                // Return nothing
20 }

```

A linear search algorithm should, on average, find its key  $\frac{n}{2}$  times where  $n$  is the number of values being searched through, in this case 666. So we expect the search to return its key after an average of 333 ( $\frac{666}{2}$ ) comparisons. As seen above, my code does linearly search through every item in magic items until the key is found, therefore my algorithm is of  $\frac{n}{2}$  efficiency on average.

### 3.3 Breakdown - binarySearch()

Listed below is the binary search algorithm I created. This function will take the middle value of the sorted magic items and compare it to the key. After checking for equality, if not equal, then it will compare if the key is greater than or less than the middle value. Depending on if the key is larger or smaller, the binary search will then recursively call itself on whatever half

the key belongs to until the value is found at the middle.

This method is inherently more efficient than a linear search as the amount of values to be search through decreases by half each time the function recurs.

```
1 string binarySearch(const vector<string> &lines, const
2     string &key) // Defines the binary search functio
3 {
4     int start = 0; // Defines the starting
5     value
6     int stop = lines.size() - 1; // Defines the stopping
7     value (at end -1)
8     int numComparisons = 0; // Defines the number of
9     comparisons to be 0
10
11     while (start <= stop) // While the start is <= to
12     stop
13     {
14         int middle = (start + stop) / 2; // The middle is
15         split between the start and stop
16         numComparisons++; // Increment
17         numComparisons
18
19         if (lines[middle] == key) // If the middle element
20         is the key
21         {
22             cout << "Number of comparisons: " <<
23             numComparisons << endl; // Print to console the
24             number of comparisons
25             comparisonCount += numComparisons;
26             // Increment comparison count with the
27             number of comparisons made to find keu
28             return lines[middle];
29             // Return the found item
30         }
31         else if (lines[middle] < key) // If the middle
32         element is less than the key
33         {
34             start = middle + 1; // Set the new stop to be
35             one less than the middle (since element in bottom
```

```

21     half)
22     }
23     else // If the middle element is greater than the
24     key
25     {
26         stop = middle - 1; // Set the new start to be
27         one more than the middle (since element in top half)
28     }
29 }
30 // If element is not found:
31 cout << "[NF] Number of comparisons: " <<
32 numComparisons << endl; // Print to console the
33 number of comparisons
34 comparisonCount += numComparisons;
35 // Increment comparison count with the
36 number of comparisons made to find key
37 return "";
38 // Return nothing
39 }

```

Binary searching, as mentioned, is more efficient than linear searching. And rightfully so since linearly searching through a graph is slower than repeatedly cutting it in half. The average efficiency of a binary search is  $\log_2 n$ , where  $n$  is 666 (the size of the search vector).

Therefore, when we calculate  $\log_2(666)$  we find that the expected number of comparisons required to find a key is 9.38. Comparing this to my code we see that the algorithm does recursively call itself on the half that the key belongs to, making it  $\log_2 n$ . So this implementation of binary search is effective.

## 4 Code Breakdown - Hash Table Searching

Assignment 2 also required us to build a hash table with a bucket size of 250. Sort the magic items, load them into these buckets, and perform searches for the keys using this table.

## 4.1 Breakdown - buildHashTable.h

The file 'buildHashTable.h' is ran through the createTable() function on line 29. In this function, a vector of strings called magic items is made and populated with the magic items from 'magicItems.txt' (ln 31 & 32). Then the hash table is created with a bucket size of 250 (ln 33). Afterwards, the hash table is populated with the magic items, this creates each bucket and fills it with its respective items (ln 34). Finally the table is returned (ln 35).

```
1   string line;                                // Defines a
   line
2   ifstream fileStream("magicItems.txt"); // Reads and
   opens the file
3   while (getline(fileStream, line))          // While there
   is a new line in the file
4   {
5       vector.push_back(line); // Push the string on that
   line into the vector of strings
6   }
7   fileStream.close(); // Close the file
8 }
9
10 int makeHashCode(const string &item) // Function to take
   an item and turn it into it's ASCII value
11 {
12     int letterTotal = 0;                // Defines the letter
   total as 0
13     for (const char &ch : item) // For every character in
   the item
14     {
15         letterTotal += toupper(ch); // Add its uppercase
   ASCII value to the total
16     }
17     return letterTotal % HASH_TABLE_SIZE; // Return the
   letter total 'mod' hash table size to get its bucket
18 }
19
20 void buildTable(const vector<string> &items, vector<
   vector<string>> &hashTable) // Function to build the
   hash table
21 {
```

```

22     for (const string &item : items) // For each item in
    the vector of strings
23     {
24         int hashCode = makeHashCode(item); // Find the
    hash code for that item
25         hashTable[hashCode].push_back(item); // Insert
    item into correct bucket based on the hash code ASCII
    value
26     }
27 }
28
29 vector<vector<string>> createTable() // Function to
    create the table (used by main.cpp)
30 {
31     vector<string> magicItems; //
    Declares the vector of strings to store the magic
    items
32     readFileIntoVector(magicItems); //
    Populates the vector
33     vector<vector<string>> hashTable(HASH_TABLE_SIZE); //
    Declares a hash table comprised of a vector of
    vectors of strings with fixed size of 250
34     buildTable(magicItems, hashTable); //
    Build hash table with the table and the vector of
    strings for buckets
35     magicItems.clear(); //
    Empties the magic items
36
37     return hashTable; // Returns the hash table
38 }

```

This code was influenced from the 'Hashing\_cpp.txt' file (from labouseur.com) as many of its methods are similar. The `makeHashCode()` function turns a given item into the sum of each character's ASCII value. It then returns the remainder of that value divided by the number of buckets in the hash table to get the value of the item's bucket.

The `buildTable()` function takes a list of items and the unpopulated hash table and populates it. Specifically, the function will take every item in the list of items, create its hash code for it, then push that code into the hash table (making its bucket).

## 4.2 Breakdown - searchItem()

The function `searchItem()`, in 'main.cpp' is used to search for a specific item in the hash table. It is called for each key in the list of keys and afterwards, the average number of comparisons for all the key's searches is computed.

```
1 string searchItem(const string &item) // Defines the
   function to search an item in the hash table
2 {
3     int numComparisons = 0; // Defines the number of
   comparisons to be 0
4
5     int hashCode = makeHashCode(item);
   // Defines the hash code for an item
6     for (const string &storedItem : hashTable[hashCode])
   // For every item in the hash table with the same
   hash code value as the item to be found
7     {
8         numComparisons++;           // Increment
   numComparisons
9         if (storedItem == item) // If the item to be found
   is equal to the sorted item
10        {
11            cout << "Number of comparisons: " <<
   numComparisons << endl; // Print to console the
   number of comparisons
12            comparisonCount += numComparisons;
   // Increment comparison count with the
   number of comparisons made to find keu
13            return storedItem;
   // True = found
14        }
15    }
16    // If element is not found:
17    cout << "[NF] Number of comparisons: " <<
   numComparisons << endl; // Print to console the
   number of comparisons
18    comparisonCount += numComparisons;
   // Increment comparison count with the
   number of comparisons made to find keu
19    return "";
```

```

20 }                                     // False = not found

```

As we can see, on line 5 the hash code value of the key is calculated. Then we check, for every item in the bucket corresponding to the hash value of the key, if the value is not equal to the key, check the next value.

If we find the value then it will print to the console the number of comparisons it took to find the value. That number is then added to the running comparison count. If the value is not found after searching through the bucket, then it prints that the value was not found to the console. Comparison count is incremented by 0, so nothing changes.

## 5 Misc.

Below is all remaining code. Anything not mentioned before this point is defined and explained below.

### 5.1 averageComparisons()

Here we are calculating the average number of comparisons needed to find every key in the sorted, or hashed, data set.

```

1 float averageComparisons() // Defines the method to
  compute the average number of comparisons
2 {
3     float average = static_cast<float>(comparisonCount) /
        KEYS_SIZE; // Defines the
        average as a statically cast float (comparisonCount / #
        of keys (defined in shuffleItems.h))
4     cout << fixed << setprecision(2) << "Average
        comparisons per search: " << average << endl; //
        Print to console the average comparisons per search
5     comparisonCount = 0;
        // Reset's the
        comparison count for the next search
6
7     return average; // Returns the average number of
        comparisons needed to find the elements for a given
        search

```

8 }

To round to two decimal places, as specified in the Assignment 2 instructions, we must use a fixed precision floating point number. To find the average of the comparisons we use a globally scoped variable called `comparisonCount` defined in section 5.2. With this we divide it by the total number of keys, defined in section 3.1 on line 3. This gives us the average which is then output to the console to two decimal places. Then the comparison count is reset to 0 for the next search.

## 5.2 main.cpp - Prerequisites

```
1 #include "buildHashTable.h"
2 #include "sortItems.h"
3 #include <iostream>
4 #include <iomanip>
5
6 using namespace std; // Globally used namespace
7
8 int comparisonCount = 0;           // Declares
    comparisonCount as 0 to store the total number of
    comparisons made for a search
9 vector<string> sorted;             // Declares the global
    sorted vector of strings
10 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



### 5.3 sortItems.h - Prerequisites

```
1 #include <fstream>
2 #include <vector>
3
4 using namespace std; // Declaring the name space to use
   std::_
5
6 vector<string> lines;           // Declares the lines a
   vector of strings
7 vector<string> keys;           // Declares keys a
   vector of strings
8 int KEYS_SIZE = 42;           // Declares a key
   selection size of 42
9 ifstream file("magicItems.txt"); // Opens and reads the
   file
```

In 'sortItems.h' we make use of importing 'fstream' for reading the 'magicItems.txt' file to a vector. Also, 'vector' is used for vector manipulation, and 'algorithm' to use the "swap" instruction. The std namespace is still used for simplicity.

Globally scoped variables:

- lines - where magic items will be written to

- keys - the 42 keys to be searched for

- KEYS\_SIZE - specifies that we will be searching for 42 keys

- file - the file to be read from when making lines

## 5.4 buildHashTable.h - Prerequisites

```
1 #include <fstream>
2 #include <vector>
3
4 using namespace std; // Declaring the name space to use
   std::_
5
6 const int HASH_TABLE_SIZE = 250; // Declares the hash
   table's size
```

The 'buildHashTable.h' file uses two include statements, one for 'fstream' which, same as before, is used to read the file to a vector. The second is 'vector' which allows us to control the data in a specified vector. Again, the std namespace is used.

Globally scoped variables:

HASH\_TABLE\_SIZE - the number of buckets used in the hash table

## 6 Final Thoughts

This assignment was, at least for myself, more straightforward and easier to comprehend than Assignment 1. This could be because I find myself more well versed in C++ and LaTeX as the semester continues, or maybe its because I actually learned something. Regardless, this assignment was very enjoyable to code, and even debug. Although C++ can be weird, especially when it comes to memory allocation, it is a fun language to learn and I look forwards to the upcoming assignments.