Assignment 2 - LaTeX Write Up

Connor Fleischman

November 1, 2024

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 - Linear & Binary Search

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 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
        key items being searched for (from shuffleItems.h)
4      sorted = sort().second;              // Defines the
       sorted vector of items (from shuffleItems.h)
5
6      for (const string &key : keys) // For every key
7      {
8          linearSearch(sorted, key); // Search the sorted
       vector for that key using a linear search
9      }
10     averageComparisons(); // Calculate the average number
        of comparisons made
11
12     for (const string &key : keys) // for every key
13     {
14         binarySearch(sorted, key); // Search the sorted
       vector for that key using a binary search
15     }
16     averageComparisons(); // Calculate the average number
        of comparisons made
17
18     sorted.clear();            // Empties the sorted
       vector
19     hashTable = createTable(); // Builds a hash table (
       from buildHashTable.h) to hashTable
```

```
20
21      for (const string &key : keys) // For every key
22      {
23          searchItem(key); // Search the hash table for that
         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, for more detail see section 2.2. Then sorted is instantiated as sort.second(), from 'sortItems.h', for more detail see section 2.2.

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.

2.2 Breakdown - sortItems.h

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 4.2, as sort().second afterwards.

```
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)

```

```
30      for (int i = 0; i < KEYS_SIZE; i++) // For 42 keys
31      {
32          keys.push_back(lines[i + start]); // Select the
        keys as 42 consecutive strings at some distance from
        the beginning of the list
33      }
34
35      return make_pair(keys, insertionSort()); // Returns
        the sorted vector "lines"
36  }
```

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.

2.3 Breakdown - linearSearch()

```
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 keu
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 keu
19    return "";
                  // Return nothing
20  }
```

2.4 Breakdown - binarySearch()

```
1   string binarySearch(const vector<string> &lines, const
    string &key) // Defines the binary search function
2   {
3     int start = 0;                // Defines the starting
    value
4     int stop = lines.size() - 1; // Defines the stopping
    value (at end -1)
5     int numComparisons = 0;       // Defines the number of
     comparisons to be 0
6
7     while (start <= stop) // While the start is <= to
    stop
8     {
9       int middle = (start + stop) / 2; // The middle is
    split between the start and stop
```

```
10      numComparisons++;                   // Increment
    numComparisons
11
12      if (lines[middle] == key) // If the middle element
     is the key
13      {
14          cout << "Number of comparisons: " <<
    numComparisons << endl; // Print to console the
    number of comparisons
15          comparisonCount += numComparisons;
                    // Increment comparison count with the
    number of comparisons made to find keu
16          return lines[middle];
                    // Return the found item
17      }
18      else if (lines[middle] < key) // If the middle
    element is less than the key
19      {
20          start = middle + 1; // Set the new stop to be
    one less than the middle (since element in bottom
    half)
21      }
22      else // If the middle element is greater than the
    key
23      {
24          stop = middle - 1; // Set the new start to be
    one more than the middle (since element in top half)
25      }
26  }
27  // If element is not found:
28  cout << "[NF] Number of comparisons: " <<
    numComparisons << endl; // Print to console the
    number of comparisons
29  comparisonCount += numComparisons;
                // Increment comparison count with the
    number of comparisons made to find keu
30  return "";
                // Return nothing
31 }
```

3 Code Breakdown - Hash Table Searching

3.1 buildHashTable.h

```
1  void readFileIntoVector(vector<string> &vector) // Function to read the file to a vector of strings
2  {
3      string line;                              // Defines a line
4      ifstream fileStream("magicItems.txt"); // Reads and opens the file
5      while (getline(fileStream, line))      // While there is a new line in the file
6      {
7          vector.push_back(line); // Push the string on that line into the vector of strings
8      }
9      fileStream.close(); // Close the file
10 }
11
12 int makeHashCode(const string &item) // Function to take an item and turn it into it's ASCII value
13 {
14     int letterTotal = 0;         // Defines the letter total as 0
15     for (const char &ch : item) // For every character in the item
16     {
17         letterTotal += toupper(ch); // Add its uppercase ASCII value to the total
18     }
19     return letterTotal % HASH_TABLE_SIZE; // Return the letter total 'mod' hash table size to get its bucket
20 }
21
22 void buildTable(const vector<string> &items, vector<vector<string>> &hashTable) // Function to build the hash table
23 {
```

```
    for (const string &item : items) // For each item in the vector of strings
    {
        int hashCode = makeHashCode(item);   // Find the hash code for that item
        hashTable[hashCode].push_back(item); // Insert item into correct bucket based on the hash code ASCII value
    }
}

vector<vector<string>> createTable() // Function to create the table (used by main.cpp)
{
    vector<string> magicItems;                          // Declares the vector of strings to store the magic items
    readFileIntoVector(magicItems);                     // Populates the vector
    vector<vector<string>> hashTable(HASH_TABLE_SIZE); // Declares a hash table comprised of a vector of vectors of strings with fixed size of 250
    buildTable(magicItems, hashTable);                  // Build hash table with the table and the vector of strings for buckets
    magicItems.clear();                                 // Empties the magic items

    return hashTable; // Returns the hash table
}
```

3.2 searchItem()

```
string searchItem(const string &item) // Defines the function to search an item in the hash table
{
    int numComparisons = 0; // Defines the number of comparisons to be 0

```

```
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 "";
                    // False = not found
20  }
```

4 Code Breakdown - Misc.

4.1 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 staticly 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 do to two decimal places, as specified in the Assignment 02 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 4.2. With this we divide it by the total number of keys, defined in section 2.2 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.

4.2 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

4.3 sortItems.h - Prerequisites

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std; // Declaring the name space to use
       std::_
7
8  vector<string> lines;             // Declares the lines a
       vector of strings
9  vector<string> keys;              // Declares keys a
       vector of strings
10 int KEYS_SIZE = 42;               // Declares a key
       selection size of 42
```

```
11 ifstream file("magicItems.txt"); // Opens and reads the
      file
```

4.4 buildHashTable.h - Prerequisites

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <vector>
5
6 using namespace std; // Declaring the name space to use
      std::_
7
8 const int HASH_TABLE_SIZE = 250; // Declares the hash
      table's size
```

5 Final Thoughts

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.