Assignment 1
Searching in Arrays & Linked Lists

Release Date	Due Date
January 16, 2020	January 30, 2020

Objectives

- Experience various techniques to search arrays and linked lists
- Practice developing high-performance solutions
- Compare theoretical vs empirical complexities
- Compare linear and binary search of arrays

Problem Specification

Linear and binary searches are common techniques used to search in linked lists and arrays. In this assignment, you will get experience in implementing these search algorithms and comparing their performances. You will also gain experience in extending binary search.

Guild Wars 2 is an MMORPG that allows you to equip a number of bags that give you inventory space. Each bag has 25 slots. In this assignment, you will write a program to simulate this inventory system and search through it for various items.

In this assignment, items will consist of only weapons. The text file included has data for ~750 different weapons in the game. There are 4 columns which list a weapon's “Item name”, “Min Strength”, “Max Strength”, and “Rarity”.

Write an application to solve the following problem:

1) Read in the “items” text file and store them in a doubly linked list, called *itemsLinkedList*.
   - Create a data structure to represent an item. This data structure should store the item's name, minimum strength, maximum strength, rarity, and current strength (current strength is not in the data set file. It will be assigned later).
2) Create *n* number of bags and populate them completely with items chosen from the previous item list (duplicate items are allowed). Remember, a bag has 25 slots. When you place an item in a bag, randomly assign its "current strength" to a number between its minimum strength and maximum strength (inclusive).
   - For example, the Hatchet has a minimum strength of 134, and a maximum strength of 163. A random value between those two values (inclusive) should be assigned to the item, such as 145, 157, or 163.
   - Fill your *n* bags sequentially. For example, the 1st item will go in the 1st bag, and the 2nd will also go into the 1st bag... up to the 25th item which will yet again go in the 1st bag. The 26th item will now go in the 2nd bag, and so on. The bags are meant to represent one contiguous inventory in sections of 25 items. When you run out of items, cycle through itemsLinkedList, i.e., start with the first item in this linked list again.
     - *Pause:* What type of data structure would be a good design for simulating these bags?

3) Randomly choose an item from the original items linked list and then search for any occurrence of it in your simulated bag system. Both the name of the item and the rarity of the item must match. Report the item you are searching for. If found, report the bag number in which it was found, the position within the bag, and the item's current strength. Report the time spent searching for the item.

- For example, if you were searching for a Rare Hatchet...
    - You might find it in the 12^{th} slot of the 5^{th} bag.
    - You might find a Fine Hatchet, but this would not count as a match, because the rarity does not match.

4) Further test the speed of your search algorithm. Repeat the process in step 3) multiple times, and report the average time searching. For this step, you do not have to print anything except the final average time that you calculated.

5) Sort your inventory by item name. Sort your entire inventory as a whole, not each bag separately.

- After sorting your items by name, further sort each subgroup of weapons with the same name by their current strength. Clearly delineate your code for doing this by adding comments.

6) Sort your inventory by item name (and each subgroup of weapons with the same name by their current strength) for each bag individually and then merge the sorted bags to sort the entire inventory. Call this Multi-MergeSort. Compare the timings of steps 5 and 6 – theoretical as well as empirical.

7) Repeat step 3) and 4), now using Binary search instead of Linear search.

8) Your program should work with an arbitrary value for *n*. Generate sample output files for cases where *n* = 1, 10, 100, 1000, and 10000.

Pause: If your code runs very fast (especially for small input sizes) then the elapsed time (using system supplied time functions) might be too small to be recorded. What can you do to remedy the situation?

Longer Pause: Remember binary search works most efficiently on a sorted array! So for m searches, there is no point in sorting repeatedly and then searching or searching repeatedly using linear searches, but sort once and then perform multiple searches. Find the range of *m* for your code where the solution of *m* linear searches is better than the solution of sort and m binary-searches.

Sample output

```
n=2
Bags before sorting:
Bag 1:
       Fine Fireman's Axe, 180
       Rare Lionguard Axe, 377
       Rare Flame Cleaver, 624
       Fine Skale Zapper, 188
       Basic Pistol, 111
       ...
Bag 2:
       Masterwork Bandit Revolver, 735
       Fine Champion's Mace, 163
       Rare Pirate Bloodletter, 397
       Fine Carving Knife, 191
       Fine Jeev's Dusty Focus, 239
       ...

Searching for Rare Ebon Vanguard Hammer...
Found in bag 2, slot 12. Strength: 901.
Single search time: xxxxx nanoseconds.
Average search time: yyyyy nanoseconds.
Bags after sorting:
Bag 1:
       Fine Ancient Longbow, 168
       Fine Ancient Short Bow, 155
       Rare Aureate Warhammer, 655
       Masterwork Bandit Revolver, 735
       Masterwork Bloodsaw Sword, 257
       ...
Bag 2:
       Rare Glyphic Longbow, 450
       Rare Glyphic Longbow, 390
       Fine Hemlocked Sword, 168
       Masterwork Jeev’s Glossy Scepter, 300
       Rare Krytan Scepter, 388
       ...

Searching for Rare Glyphic Edge...
Not found.
Single search time: xxx nanoseconds.
Average search time: yyy nanoseconds.
```

Output Requirements

- Report the value of *n*.
- If n <= 10, Print out the first 5 items of each bag, before and after sorting. Otherwise, don't print out the contents of the bags.
- Step 3's output requirements (Average time spent searching, searched item's names, locations, strengths).
- Steps 5 and 6’s measured timings.

Design Requirements

Code Documentation

For this assignment, you must include documentation for your code. This includes how to compile and run your program. Also, you **must** give outputs of your program with n = 1, 10, 100, 1000, and 10000. If you don't provide outputs for these 5 inputs, it means your program does not execute properly.

Coding Conventions and Programming Standards

You must adhere to all conventions in the CS 3310 Java coding standard. This includes the use of white spaces for readability and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions for naming files, classes, variables, method parameters and methods. Read the material linked from our class web-pages (in case you can't recall programming styles and conventions from your CS1 and CS2 courses).

Testing

Make sure you test your application with several different values capturing different cases, to make sure it works.

Assignment Submission

- Generate a .zip file that contains all your files, including:
  - Plagiarism Declaration
  - Source code files
  - Including any input or output files
  - Documentation of your code – e.g. using Javadoc if using Java
  - A brief report (in a pdf file) on your observations of comparing theoretical vs empirically observed time complexities. Note this report will include (a) a brief description of problem statement(s), (b) algorithms descriptions (if these are standard, commonly known algorithms, then just mention their names along with customization to your specific solution(s), otherwise give the pseudo-code of your algorithms, (c) theoretically derived complexities of the algorithms used in your code, (d) table(s) of the observed time complexities, and (e) plots comparing theoretical vs. empirical along with your observations (e.g. do theoretical agree with your implementation, why? Why not?).

- Don't forget to follow the naming convention specified for submitting assignments