BLG 335E - Analysis of Algorithms I, Fall 2017

Project 1

Total worth: 7.5% of your grade

Handed out: 27 Sep 2017, Wednesday

Due: 11 Oct 2017 Wednesday - 23:59

Overview

Hearthstone™ is an online collectible card game developed by Blizzard Entertainment. The gameplay is turn-based, using constructed decks of thirty cards and a selected hero of a designated class. Players use their limited mana crystals to cast spells or summon minions to attack the opponent, with the goal to reduce to opponent's health to zero. Each player is provided a set of basic cards to build decks from, but they can add new cards to their collection over time that allow different synergies. Some cards are available only to specific classes, while others are available to all. Winning matches can earn in-game gold, which can be used to buy card packs and expand one's card collection. Expansions are periodically released, bringing new card sets and further customization to the game.

A view of the in-game card manager

You will implement sorting routines for a card manager for Hearthstone. You are provided three different card collections with 10 000, 100 000, and 1 000 000 cards. These collections are organized in tab-delimited text files (***hs-set-*[*10k*|*100k*|*1M*]*.txt***), formatted as follows:

NAME	CLASS	RARITY	SET	TYPE	COST
Soulfire	Warlock	Basic	Basic	Spell	1
Blubber Baron	Neutral	Epic	Gadgetzan	Minion	3
Al'Akir the Windlord	Shaman	Legendary	Classic	Minion	8
Molten Blade	Warrior	Rare	Un'Goro	Weapon	1
Dalaran Aspirant	Mage	Common	Grand Tournament	Minion	4

You will sort these collections based on the following two sorting procedures:

- **Full sort:** All cards will be sorted first by **CLASS**, then by **COST**, and finally by **NAME**. The result of this sorting procedure must yield results **identical** to the supplementary files with the ***-sorted*** affix, provided along with the data sets for reference.
- **Filter sort:** All cards will be sorted only by **TYPE**. Collections sorted with this procedure must display cards with the same **TYPE** together as an undivided consecutive block, but the cards within the same **TYPE** block can remain unsorted with respect to each other.

Furthermore, you will implement both **insertion sort** and **merge sort** in your code, and apply both algorithms to each of these sorting procedures, measuring runtimes for each execution.

For the smaller data sets, it is suggested that you re-run your sorting code in a redundant loop multiple (*e.g.* 1000) times, then divide the measured runtime by the same number, to achieve higher granularity in your runtime measurement. This is not mandatory, but is useful for getting better measurements if your sorting passes finish almost instantaneously.

Code (60 points)

Implement the insertion sort [20 points] and merge sort [20 points] algorithms in C++. Write your program in a class called **`CardManager`** encapsulating the necessary methods and attributes corresponding to the algorithms and data structures you will require for this project.

Your program must write the sorted output in a separate file [10 points], and print the time elapsed by the sorting procedure on the console [10 points].

Your class must have the methods **`fullSort(...)`** and **`filterSort(...)`** that respectively perform the full sort and the filter sort procedures described earlier.

You must also include a **`main(...)`** function in your code expecting four string values as command line arguments. The first argument can be either **`-full`** or **`-filter`**, specifying which sorting procedure to use. The second argument can be **`-i`** or **`-m`**, specifying insertion sort and merge sort respectively. The third and fourth arguments specify filenames for the input (unsorted) and output (sorted) collections respectively.

Your program should compile and run using commands similar to the following example:

```
[user@ssh ~]$ g++ yourStudentID.cpp -o project1
[user@ssh ~]$ ./project1 -full -i hs-set-10k.txt hs-set-10k-full.txt

   Time elapsed: 13.6 microseconds

[user@ssh ~]$ _
```

Your code will be carefully inspected and tested on different data sets from those provided with the assignment. Please do not resort to funny tricks. Happy coding!

Report (40 points)

Q1) [10 points] Provide two graphs or tables (for full sort and filter sort) comparing runtimes of insertion sort and merge sort on the three card collections provided with the assignment.

Q2) [10 points] Briefly analyze the computational performances of insertion sort and merge sort with respect to your sorting parameters. Did one of the algorithms always outperform the other? Or did one perform better than the other (or almost catch up with it) in a specific scenario? If so, briefly discuss the reason(s) why.

Q3) [10 points] Consider you had to sort cards by **RARITY**, or by **SET**, in filter sort. In what way are these two parameters similar to **TYPE**, but different from **NAME**? Could sorting by **RARITY** or **SET** in filter sort affect the performances of insertion sort and merge sort?

Q4) [10 points] What is stable sorting? Are insertion sort and merge sort stable? What could go wrong in the full sort procedure if an unstable sorting algorithm is used?

Submission

You should be aware that the Ninova system clock may not be synchronized with your computer, watch, or cell phone. Do not e-mail the teaching assistant or the instructors your submission after the Ninova submission has closed. If you have submitted to Ninova once and want to make any changes to your report, you should do it before the Ninova submission system closes. **Your changes will not be accepted by e-mail**. Connectivity problems to the Internet or to Ninova in the last few minutes are not valid excuses for being unable to submit. **You should not risk leaving your submission to the last few minutes**. After uploading to Ninova, check to make sure that your submission appears there.

Policy: You may discuss the problem addressed by the project at an abstract level with your classmates, but you should not share or copy code from your classmates or from the Internet. **You should submit your own, individual project.** Plagiarism and any other forms of cheating will have serious consequences, including failing the course.

Submission Instructions: Please submit your assignment through Ninova. Include both your report (as a PDF file) and your code (including header files) in the archive file you submit.

All your code must be written in C++, and must compile and run on the common ITU Linux Server (accessible through SSH) using g++. If your code requires non-standard compiling, please state compilation instructions in your report.

When you write your code, try to follow an object-oriented methodology with well-chosen variable, method, and class names and comments where necessary**. Your code must compile without any errors; otherwise, you may get a grade of zero on the coding part of the assignment.**

If a question is not clear, please let the teaching assistant know by email (sulubacak@itu.edu.tr).