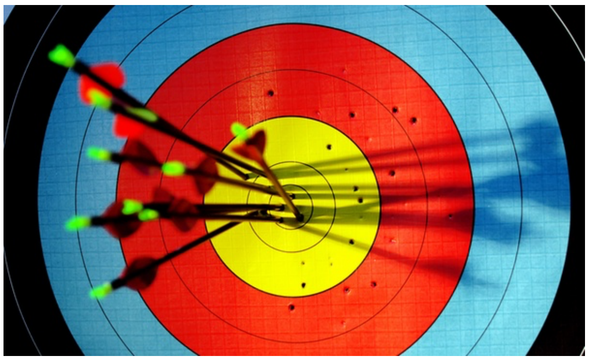
Algorithms and Data Structures

Advanced sorting for an Archery competition

Assignment-3

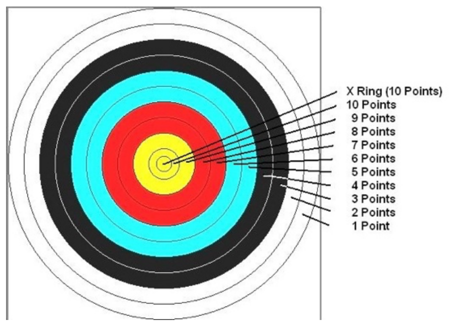
Version: October 20th, 2021



Introduction

You are asked by a little start-up to write a program that can be used during an archery competition. Another software engineer has been asked to create a mobile application that is used by the judges for entering the scores for each archer, so you can focus on the back-end of the application.

If you can write the program well it might be a goldmine for the start-up and you, as the creator of this program, can of course share in the profits. Because there is little time left for developing the application, which is common practice for start-ups, you much write (almost) bug free. Unfortunately, the mobile app is still in development, so you have to use have some software that acts as the mobile app and that will generate the results for each archer. This code is almost ready and needs some last-minute adjustments and improvements.



Archers shoot during a competition of 10 rounds. Per round they shoot 3 arrows. So, in total each archer will shoot 30 arrows. Depending on the exact kind of competition the ranking can be determined using different schemes. This means that it is important to keep track of each arrow (points) in each round. (You should not let an archer shoot 30 arrows and then only register the total number of points they have shot.) The number of points ranges from 0 (the arrow is outside the white rings) to 10. (See the image at the left.)

Register archers and scores

Two classes have been provided for registering all information about the archers and the competition:

1. The **Archer** class tracks all information about one archer, such as first name, last name, and the scores of all arrows that have been shot. Every archer also gets a unique registration id, when being enrolled into the competition. The public method `registerScoreForRound` is made available to the judges to register the scores of one round of three arrows for a specific archer.
2. The **ChampionSelector** class enrolls archers into the competition and calculates overall results. For convenience of testing, a so-called factory method, `enrollArchers`, is available to generate and enroll a random batch of sample archers. A factory method is a method that takes care of all the details of construction complex instances of classes. In this case the factory method takes care of generating and enrolling multiple archers into the competition and let them all shoot 10 rounds of 3 arrows including the registration of the scores by the judges. (So, when the list of archers is returned by this factory method, all archers have been notified of their scores.)

The seed of the constructor of ChampionSelector seeds the random number generator, such that you can reproduce results from the same (non-zero) seed. Use seed=0 to generate a truly random sample in each run.

If all information has been registered, then the results of the competition can be shows with the method **ChampionSelector.showResults**(). This method relies on sorting the archers according to various criteria. For that it uses some different sorting methods as promised by the **Sorter** interface. You may choose to complete the generic **SorterImpl** implementation class (such that you can later reuse this class for other sorting assignments) or provide specific sorters for archers in the **ArcherSorter** class.

Missing code

Here you find some information about your coding tasks to complete. Only the **Archer** class and the **SorterImpl** class have missing code, indicated by // TODO comments. Additional information can be found in the JavaDoc in the provided starter project.

Complete the basics of the Archer class.

R1. The `toString()` method of Archer should return a `String` in the following format: “135787 (225) Nico Tromp”, so first the id, then between brackets the total number of points followed by first name and surname separated by spaces.

R2. Ensure that each archer gets a unique ID that increases by one for each new archer (the first archer created has ID 135788).
Ensure that the ID can never change once it has been assigned a number.

R3. Add an instance variable to store the scores of 10 rounds of 3 arrows each.
The method `registerScoresForRound` should be able to update the 3 arrows of one specific round without impacting any scores of other rounds. Think of a nice data structure that can be used to store these scores. Should it be a 2-dimensional array, a list of arrays, a list of lists of int’s, or … ?

Scoring scheme.

As mentioned before different competitions may apply different scoring rules. That is the reason why your application must be able to handle different scoring schemes. For this assignment we want you to implement the following rules:

i. When comparing the total number of points between two archers the one with the most points ‘wins’.

ii. If the number of total points is equal, the archer with the least number of misses wins. (A miss is an arrow that scores 0)

iii. If it is still a tie, then the archer who got registered first wins. (Archers with a lower ID have been registered before archers with a higher ID.)

R4. The Archer class provides an instance method `compareByHighestTotalScoreWithLeastMissesAndLowestId` which you shall complete to be useful as a Comparator<Archer> function in the sorting methods.

Sorting methods.

Now that you know how to rank the archers it is time to find the most efficient way of determining the champions. You need to sort the archers in such a way that the champion is the first in the list and the archer with the lowest score is at the end.

You will be evaluating up to four different sorting methods:

I. The Java Collections sort as offered by the **List<E>** interface.
II. Either a selection or an insertion sort (you may choose your algorithm of preference).
III. Quick sort (any variant of your choice that you can explain well).
IV. [OPTIONAL for excellent grade] tops Heap sort
This algorithm finds the top-M items of a list of N and sorts them into the front part of the list, without worrying about the sorting order of the remainder.
This sorter is very useful for creation a (small) 'hall-of-fame' out of a large collection, which may not even fit in memory… (N >> M).

R5. Complete the sorter method implementations II) III) and optionally also IV) in the **SorterImpl** class. (Or provide specific implementations of the Sorter<Archer> interface in the **ArcherSorter** class first, if you find it difficult to code a generic implementation immediately).
Your implementations shall comply with the method signatures as you find them in the interface specification.

Tips and constraints.

- Signatures of public methods shall not be changed.
- You may add private methods and instance variables as you deem appropriate.
Provide appropriate in-line documentation for these.
- Apply proper encapsulation to all your coding.
- Minimize cyclomatic and cognitive complexity.
Add useful in-line comments.
Document loop-invariants and representation invariants in your code.
- Your code should pass all unit tests.
Also add additional unit tests as appropriate (in a separate test class)!
(The provided unit tests do not cover all functionality!)

Presenting the results:

If your code is complete and passes all unit tests, you can present your sponsor with the results of the competition. Below you find the expected output of the main program:

```
Welcome to the HvA Archery Champion Selector


1001 archers have participated in this competition
The first three archers to enroll were: [135788 (147) Billy STANLEY, 135789 (162) Dorothy LOPEZ,
135790 (153) Ryan HOWELL]
The first three archers by alphabet are: [136759 (156) Amber ADAMS, 135866 (131) Dennis ADAMS,
135842 (130) Jessica ADAMS]
At 4th thru 10th place of the rankings we find: [135930 (198) Danielle ROGERS, 135973 (196) Grace
MARTIN, 135805 (195) Sophia ROBINSON, 136529 (195) Carol RAY, 136421 (194) Ethan GRAHAM,
136362 (194) Logan CARTER, 136670 (192) Lori ELLIOTT]
The top-3 price winners of the competition are: [136216 (200) Ronald CARLSON, 136693 (199) Emily
STEWART, 136062 (198) Donna HUDSON]
```

Efficiency

Besides correctly calculating the outcome of the competition, we want you to perform a big-O time complexity analysis of each sorting method, supported by analysis of measured execution times.

Determine how long it takes for each algorithm to sort a list of archers and their scores. You start with 100 archers and multiply it by 2 and keep doing this until you reach 5.000.000 archers or until sorting the list takes more than 20 seconds. *When measuring the time, it takes to sort the archers, make sure you only measure the time needed for actually sorting the archers, and not for creating them! Also, run System.gc() after each generation of an input set, so that garbage collection will not impact the measurements!*

To ensure that your comparisons are fair it is important that you use the same unordered list of archers for each algorithm. So, once you have a list of archers make four copies of it and use each copy for a different algorithm.

Because the algorithms may perform differently on different specific input, and your computer has more tasks to perform (playing songs through Spotify, scanning files for viruses) than only determine the champion for the competition, you must run each experiment 10 times each with a different seed for the ChampionSelector and use the average over these 10 runs as input for the determination of the efficiency.

R6. Code your full performance test into an additional unit test class.

Run your tests with the JVM parameter `–Xint` to eliminate impact of the JIT compiler on your measurements (See ppt slides of lecture and exercise on efficiency.)

Now that you have the numbers you need to determine (by using math) the efficiency of each algorithm. For the topsHeapSort algorithm both the size N of the list and the number of tops M impact the time complexity. Use both N and M in your big-O expression.

Include both the numerical data and a graphical representation of measured time T vs. problem size N in your report.

Report

Your report shall include the following:

1. Seven code snippets with explanation/justification of your code, with clear rationale. (A Dutch or English description of the statements in your code does not add any information.)
2. At least one use of a loop-invariant to argue correctness of your code.
3. Tables with numerical data of measured execution times of all your sorting methods for problem sizes of 100, 200, 400, 800, etc. util execution time exceeds 20sec.
4. Graphical representation of execution time vs problem size of above performance results.
5. Big-O complexity analysis of each implemented sorting method based on structure of the code. For the tops-M Heap Sort algorithm, use problem size N and tops size M in your big-O formula.
6. Big-O complexity analysis of the measured execution times, by comparing ratio's of measured execution times against ratio's of problem sizes. (See ppt slides of lecture on efficiency)

Grading

Basic grading criteria can be found in the Rubrics at the DLO.

In addition, the following is applied.

1. The maximum score for solutions without a specific implementation of the topsHeapSort method is 'Good'.
2. The maximum score for solutions with a specific ArcherSorter class instead of the generic SorterImpl class is 'Good'