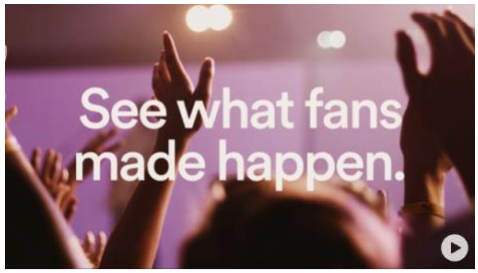
Algorithms and Data Structures

Advanced sorting for Spotify Charts

Assignment-3

Version: September 25th, 2023



Introduction

At https://charts.spotify.com/home you can learn how hundreds of millions of listeners shape today's streaming charts. In this assignment you will prepare such charts with optimal computational efficiency from raw data snapshots, following ranking criteria of different listener communities.

You will explore your own implementations of three sorting algorithms:

i. A choice of selection-sort, insertion-sort or bubble-sort.
ii. An implementation of quicksort.
iii. An implementation of heapsort for a partial sort of a top-25 hits chart.

The ranking criteria basically follow the total number of streams that have occurred globally in the past week, but national charts may give priority to songs in their local language.

Then you will verify their computational efficiencies amongst each other and against the List.sort of the Java Collection Framework.

After completing reading of this document, you can start to prepare your solution with help of the provided starter project. Only the Song class and SorterImpl class in this project have missing code sections to be completed by you. These sections are marked with // TODO comment lines. We also expect you to add another test class with additional unit tests and prepare a report with code explanations and efficiency measurements and analysis.

Preparing the test data

Three classes have been provided for preparing test data of songs and their stream counts:

1. The **Song** class tracks all information about one song, such as title, artist name, language, and the number of downloads of the song during the current charting period, split by country. For simplicity we consider only seven countries which have been enumerated in the Song class in
   `public enum Country { … }`
   It is your task to choose an appropriate data structure and add an instance variable to the Song class to represent these per-country streams counts and encapsulate its access via three public methods:
   ```
   public void setStreamsCountOfCountry(Country country, int streamsCount) { … }
   public int getStreamsCountOfCountry(Country country) { … }
   public int getStreamsCountTotal( ) { … }
   ```
   Try to select the simplest data structure that does the job well and efficiently while using clear code with low cognitive complexity.
2. The **SongBuilder** class is a helper class to generate a collection of unique songs for testing of your sorters. You can have a look at it, but there is no need to modify this code.
3. The **ChartsCalculator** class uses the SongBuilder to generate a collection of unique songs of a given size, and then populates the per-country streams counts with some semi-random sample

data using the access methods that you have prepared in the Song class.
You can have a look at the ChartsCalculator, but there is no need to modify this code.

Please also implement Song.toString to produce nicely readable representations of songs in your output in the format "*artist*/*title*{*language*}(*total streamsCount*)"

Sorting the charts

We would like you prepare sorted charts according to two different ordening criteria:

1. The **global popularity chart** is sorted by decreasing total number of streams (accumulated across all countries.).
   Please implement this criterium in Song.compareByHighestStreamsCountTotal.
2. The **Dutch national chart** ranks all songs in the Dutch national language before other songs in other languages, but then still uses the total number of streams across all countries as a secondary (decreasing) sorting criterium.
   Please implement this criterium in Song.compareForDutchNationalChart.

The capabilities of the sorting algorithms are promised by the generic **Sorter** interface. You are required to provide a generic implementation of this interface in the **SorterImpl** class. (You may also choose to provide a specific implementation in **SongSorter**):

a) Complete the selInsBubSort method with an implementation of selection sort, insertion sort or bubble sort at your choice.
b) Complete the quickSort method with a recursive implementation of the quick sort, that is different from the example implementation that was given in the lectures.
   You may need to add local private methods as part of your implementation.
   Don't forget to add references to the external sources which you have used for this, if any.
c) Complete the topsHeapSort, heapSink and heapSwim methods that implement a partial heap sort for efficient calculation of hall-of-fame type rankings.
   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 ordening of the remainder.

If all is done, then the ChartsCalculator will produce the Spotify.Charts of this week! Below you can find the result that can be expected from the main program. You can vary the seed of the constructor of ChartsCalculator to also test with different sample data.

```
Welcome to the HvA Spotify Charts Calculator


263 songs have been included in this week's charts


The five most streamed songs are:
[Beyoncé/CUFF IT{EN}(241723), The Weeknd/Stargirl Interlude{EN}(233514), Steve Lacy/Bad
Habit{EN}(233432), Eminem/Superman{EN}(231412), DI/RECT{EN}(218694)]


The top-five in the Dutch-language national chart are:
[Suzan & Freek/Honderd Keer{NL}(115248), Kris Kross Amsterdam/Vanavond (Uit M'n Bol){NL}(113414),
Antoon/Leuk{NL}(108697), Snelle/Blijven Slapen{NL}(104532), $hirak/Als Je Bij Me Blijft{NL}(103444)]


The bottom-ten least streamed songs are:
[Natte Visstick/Visstick Gooi Die Kanker Kick{NL}(38850), Rels B/cómo dormiste?{SP}(44849),
Antoon/Olivia{NL}(48248), Suzan & Freek/Kwijt{NL}(48319), Jinho 9/Interessant{NL}(51788), Bad
Bunny/Party{SP}(52335), Feid/Feliz Cumpleaños Ferxxo{SP}(52425), Zoë Tauran/Solo{NL}(53787), Kris
Kross Amsterdam/Vluchtstrook{NL}(54618), Antoon/Hallo{NL}(55115)]
```

Other tips and constraints.

- Signatures of public methods shall not be changed.
  You may add private methods and instance variables as you deem appropriate.
- Apply proper encapsulation to all your coding.
  Isolate duplicate or similar functionality into reusable methods.
- Minimize cyclomatic and cognitive complexity.
  Add useful comment lines.
- Your code should pass all unit tests…

Unit tests

1) Some unit tests are provided to assist you to verify correctness of your code.
2) But these unit tests are not exhaustive for all possible scenarios:
   Add another test class with two unit-tests which verify stability of each of the ranking comparator methods in the Song class, i.e.:
   a. compare(song, song) returns 0 for any song compared with itself.
   b. compare(song1, song2) == - compare(song2,song1) for any two songs.
3) Add at least two more unit-tests for scenario's that were impacted by a defect during your initial development work but were not easy traceable from the available unit tests.
4) Add one unit test class or main class that reproduces the results of your performance measurements as explained below.

Efficiency

Besides correctly calculating the outcome of the charts, 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 the global popularity chart of sizes 100, 200, 400, 800, 1600, … etc., until you reach 5.000.000 songs or until sorting the list takes more than 20 seconds.

When measuring the time:

A. Make sure you only measure the time needed for sorting the songs, and not include the time needed to setup the test data!
B. Run System.gc() after each generation of an input set, so that garbage collection will not impact the measurements!
C. Run your tests with the JVM parameter -Xint to eliminate impact of the JIT compiler on your measurements (See ppt slides of the lecture and exercise on efficiency.)
D. Make sure that you use the same unordered list of songs for each algorithm, ensuring that your evaluation of the sorting algorithms is fair. (E.g., once you have generated a list of songs of a given size, make four copies of it, each for use by a different algorithm.)
E. *Run each measurement 10 times, each with a different seed for the ChartsCalculator (or seed==0) and use the average result over these 10 runs as input for the evaluation of the efficiency.*
You may find varying performances of the same algorithm on different input lists of the same size, because your algorithms may perform differently on specific input, and your computer may be busy with different tasks at the same time (receiving e-mail, updating your you-tube subscriptions, indexing your file system, scanning files for viruses, etc.)

Now that you have the numbers you need to determine the efficiency of each of your selInsBubSort, quicksort and topsHeapSort implementations (by using math…):

a) Provide a static analysis and big-O assessment of CPU-time complexity based on the code structure of each implementation.

b) Provide a numerical analysis and big-O assessment of CPU-time complexity based on the computation time measurements of each implementation.

For the topsHeapSort algorithm both the size N of the list and the number of tops M impact the time complexity. So, use both N and M in your big-O expression here.

Include both the numerical data and a graphical representation of measured time T vs. problem size N in your report (i.e. problem size on the horizontal axis, measured time on the vertical axis of your graph; combine all sorts in one graph; chose the range of the vertical axis such that outliers are cut off and the differences between the algorithms can be observed.)

Report

Your report shall include the following:

1. Seven code snippets with explanation/justification of your code, with clear rationale.
   (A Dutch or English transcription 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 and the collection sort for problem sizes of 100, 200, 400, … etc. until the 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 ratios of measured execution times against ratios of problem sizes. (See ppt slides of lecture on efficiency)

Grading

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

In addition, the following is applied.

1. If you failed to complete topsHeapSort algorithm successfully, you still can earn a sufficient score provided that you have submitted a serious attempt to succeed.
2. The maximum score for a solution with a specific implementation in the SongSorter class instead of the generic SorterImpl class is ‘Good’