Algorithms and Data Structures

Wk6 – exercise –Birthday's paradox

In this exercise you will practice different implementations with different computational efficiencies and memory footprints of some algorithms on a list of numbers. A start-project is available.

The application studies the expectation of how many people from a group of N will celebrate their birthday on the same day in the year. It appears that already from groups of 1000 people onwards you will find low probability that somebody celebrates her birthday alone!

So let's explore some statistics!

The constructor

```
public BirthdaysList(int numberOfPeople)
```

generates a list of N=numberOfPeople random integer day numbers between 1 and 365 (We ignore leap years for this exercise).

You are not allowed to sort the list at any time after generation…

Part1: basic solutions with low memory footprint.

Make best effort to provide the most efficient implementations of the following methods, without the use of auxiliary arrays of size N or size 365 or above. (If the method returns a List, you may instantiate an ArrayList only for building the result of the method).

```
public int countBirthdaysOn(int birthday)
```

Counts the number of people in the list that celebrate their birthday on the given birthday argument.

```
public int countBirthdaysBetween(int fromBirthday, int toBirthday)
```

Counts the number of people in the list that celebrate their birthday between the given birthday boundaries inclusive.

```
public int findLastBirthdayOfYear()
```

Finds the last day in the year (1 <= day <= 365) on which somebody in the list celebrates his birthday

```
public int countNumUniqueBirthdays()
```

Counts the number of people in the list that celebrate their birthday alone, on a day where nobody else also celebrates.

```
public int findFirstNonBirthday()
```

Finds the first day in the year (1 <= day <= 365) on which nobody in the list celebrates her birthday.

```
public int maxPeopleWithSameBirthday()
```

Counts the maximum number of people that all celebrate their birthday on the same day.

```
public int maxPeopleWithBirthdayInSameWeek()
```

Counts the maximum number of people that all celebrate their birthday within one week.
One week is any sequence of 7 consecutive days, i.e.
day 1..7 or day 2..8 or day 3..9 etc. etc. until the last week of day 359..365

```
public int findMedianBirthday()
```

[Brainteaser] Finds the day in the year so that half the people celebrate their birthday on that day or before and the other half of the people celebrate their birthday on that same day or after.

```
public List<Integer> findAllBirthdaysWithMaxPeople()
```

Find all days in the year on which the maximum number of people celebrate their birthday together.

```
public List<Integer> findBirthdaysCoveringHalfOfThePeople()
```

[Codeteaser] Find the minimum set of days such that all birthdays of half of the people are covered.

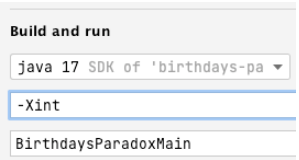
Unit tests are available in the start project to help you find correct implementations.

Part2 Efficiency.

Measure the performance of each of the methods for different numbers of people in the birthdays collection. E.g. try N=1000, 2000, 4000 upto N=64000 depending on the performance of your computer. (Don't waste your time waiting on results of too high problem sizes.

```
Testing class BirthdaysList with a list of 2500 birthdays:
Found 6 people on day-123 after 0,044 msec
Found 318 people between day-100 and day-150 after 0,051 msec
Found last birthday of year at day-365 after 0,039 msec
Found 0 unique birthdays after 9,562 msec
Found first non-birthday at day--1 after 1,122 msec
Found max 15 people with same birthday after 2,935 msec
Found max 72 people with birthday in same week after 3,023 msec
Found median birthday at day-196 after 0,460 msec
Found birthdays [281, 285] with maximum number of people after 2,945 msec
Found 131 birthdays that cover half the people after 98,528 msec
[281, 285, 205, 227, 307, 270, 274, 309, 316, 2, 65, 87, 98, 154, 266, 306, 334, 345,
 346, 13, 55, 141, 184, 256, 263, 317, 327, 28, 38, 79, 105, 125, 127, 151, 174, 176,
 191, 195, 212, 234, 242, 249, 265, 297, 304, 312, 331, 335, 343, 359, 362, 17, 32, 43,
  63, 78, 84, 100, 102, 134, 135, 136, 137, 140, 158, 165, 181, 182, 193, 200, 201,
 218, 228, 233, 237, 245, 261, 269, 273, 282, 290, 308, 319, 321, 330, 336, 349, 350,
 353, 355, 365, 1, 5, 9, 12, 22, 34, 59, 62, 66, 68, 77, 83, 101, 109, 118, 119, 124,
 128, 139, 152, 156, 162, 170, 175, 178, 203, 206, 211, 214, 220, 225, 241, 251, 255,
 262, 264, 267, 271, 276, 283]
```

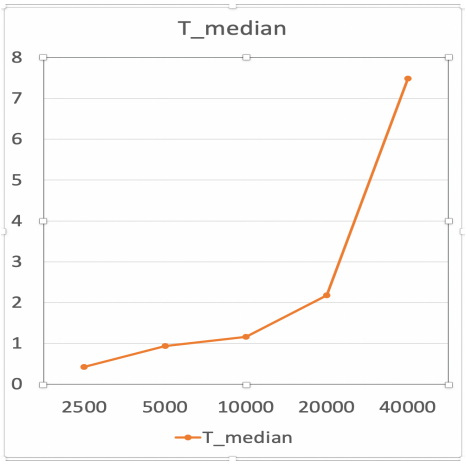
The starter project shows you how to measure nano seconds and force a sweep of the garbage collector, such that memory management doesn't play into your measurements. Complete the static method to gather performance info of all above methods. Also don't forget to disable the JIT compilation option in the run configuration.



Collect your measurements in an excel sheet, and product a graph to visualize run times.

Determine for each method the average growth factor of execution times each time when the problem size doubles, and determine the bigO computational complexity from that.
Does that match your expectations from what you expect from the structure of your code?
If it doesn't match, obtain more accurate measurements by running the method multiple times for each problem size, calling the .shuffle() method in between. Use the average time measurement for your analysis.

N	T_median
2500	0,428
5000	0,94
10000	1,16
20000	2,182
40000	7,49



Part3 Optimisation.

For some of the methods you can code a faster algorithm if you can use an auxiliary array to store intermediate results.

Override these methods in the class FasterBirthdaysList which extends BirthdaysList.

Without too much additional coding you can run your performance measurements on this class.

Redo the complexity analysis of part-2 on these new results.