Programming Hand in 4

Brooks & Odderskov

For this handin you must work in your semester project group.

The objective of this hand in is to make you apply the concepts of data structures, sorting and graphs to new problems. The scope of the content were covered during Lectures 9, 11 and 12:

- Balanced Binary Trees (Program 1)
- Sorting (Program 2)
- Graphs (Program 3)

Please note that the exercises in this hand in are more complex than in the previous hand ins. This will be taken in to consideration in the assessment. If you get stuck in an exercise, you are advised to seek out the student instructor for help at the study café (on Discord). If you are still stuck, skip the exercise and move on to the next one.

For each of the below exercises there is an associated template. You can find all templates in the zip file 'dmaProg4.zip'. Create a new project in IntelliJ for all your DMA programming handins. It is up to you how you want to structure your hand ins but it may be a good idea to have a module for each of the four hand ins. Unpack the zip file to the newly created project folder.

Do not change any of the existing code, e.g. do not rename methods, etc., although you may need to change the reference to the package (first line in templates). You will only need to add the body of your methods in the relevant template. For each of the programs, replace the comment with your code. Note, you do not need to create methods for user inputs. All you really need to do is to write the logic of the methods, i.e. the algorithms. But it may be beneficial in some cases to create additional classes.

Each exercise is accompanied by a test class. You can use this to test whether your method works correctly. Note, there are also complexity tests for the first two programs to check whether you found a fast solution as defined in the exercise. In program 3, no additional points are available.

For this hand in, you do not need to do complexity analysis.

You upload your hand in as a zip-file where you simply zip the handin4 module with all your code/files. **If you copy or in any other way use code from other sources, please reference these sources as a comment in your program.**

Program 1: `Median`

In this exercise you should implement a data structure named `Median` that supports the two operations:

- `void add(int v)` - add a number to the data structure
- `int median()` - return the median of the numbers in the data structure

In this exercise, the **median** of a set of N numbers is the number in the set that comes **after** the first N/2 (rounded down) numbers in the sorted order. In Java, this means that the median is the element at index `N/2` in the sorted order (since integer division in Java rounds down, and Java uses 0-based indexing).

Example 1: If the set contains the numbers:

```
2, 5, 3, 7
```

then the sorted order is `[2, 3, 5, 7]`, so the median is 5.

Example 2: If the set contains the numbers:

```
2, 4, 7, 9, 3
```

then the median is 4.

Example 3: Note that the median is not necessarily close to the average. For instance, if the set contains the numbers:

```
32, 59, 32673, 58586, 14162, 288, 411
```

then the median is 411 (although the average is 15173).

Concretely, you should use the template `Median.java` and implement the methods `add` and `median` in `Median`, and you are allowed to add your own private fields to the class.

Input constraints:

- Number of insertions is at most 500000
- All numbers added are distinct (that is, the same number is not added multiple times to the data structure)
- Each number is an integer between 1 and 1000000000
- The first operation is `add()`

Scoring:

- 1 point for correct algorithm
- 1 extra point for correct and fast algorithm

An algorithm that spends $O(\log N)$ time per operation is fast enough for the extra point.

Hint for the fast solution: Use two priority queues that implement a heap. Note, a heap is simply a special case of a balanced binary tree data structure which we covered in Lecture 9.

Program 2: `Inversions`

Let $A[1..N]$ be an array of N integers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A. In this task you must implement an algorithm to count the inversions in an array of N integers from 1 to N.

For instance, the list

```
1, 4, 2, 3
```

contains two inversions, namely (2, 3) and (2, 4), since $A[2] > A[3]$ and $A[2] > A[4]$.

The list

```
5, 4, 3, 2, 1
```

has 10 inversions, namely (1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5).

Concretely, you should use the template `Inversions.java` and implement a public method named `countInversions` that takes an `ArrayList<Integer> input` as argument and returns an `int`.

Input constraints:

- $1 \leq N \leq 65536$
- Each element is between 1 and N
- Each number between 1 and N occurs exactly once in the list, i.e. there are no duplicate elements
- The number of inversions fits in a Java `int` without overflowing

Scoring:

- 1 point for correct algorithm
- 1 extra point for correct and fast algorithm

An $O(N \log N)$ algorithm is fast enough for the extra point.

Hint for the fast solution: Consider how merge sort works and modify it. The best place to start is the pseudo-code from the slides, you will want to modify Merge and MergeSort and perhaps just call them from countInversions. Note, finding a fast algorithm is a difficult task, and this should be seen as a challenge exercise. Finding a $O(n^2)$ algorithm should be manageable and does not require looking in to merge sort (you can solve it using two loops).

Program 3: `Maze`

In this exercise you must compute the shortest path in a maze from the top left corner to the bottom right corner. A maze is an H-by-W matrix A (H rows, W columns), where each cell $A[i][j] (0 \leq i < H, 0 \leq j < W)$ is either a '.' (free space) or a 'o' (wall):

```
ooooooooooooo
o.o.o.....o.o
o.o.ooo.ooo.o
o.......o...o
ooooo.o.o.ooo
o.....o.....o
o.ooooooo.o.o
o.ooo.o.o.o.o
o...o.ooooo.o
ooo.o.......o
o.......ooo.o
ooooooooooooo
```

We can model the maze as a graph as follows. The vertices of the graph are the pairs (i, j) where $A[i][j] ==$ '.', and the edges out of a vertex (i, j) are the four neighbors $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$, except where the neighbor is a wall 'o'.

The number of vertices and edges is $O(W \cdot H)$ (you should convince yourself that this is true). The length of a path is simply the number of edges in the path. In the example above, the shortest path consists of 20 vertices, which means that the shortest path has length 19:

```
ooooooooooooo
oxo.o.....o.o
oxo.ooo.ooo.o
oxxxxxxxo...o
ooooo.oxo.ooo
o.....oxxxxxo
o.ooooooo.oxo
o.ooo.o.o.oxo
o...o.oooooxo
ooo.o......xo
o.......oooxo
ooooooooooooo
```

Concretely, you must implement a method named `shortestPath` that accepts a `char[][] maze` as parameter and returns an `int` indicating the length of the shortest path from `maze[1][1]` to `maze[H-2][W-2]` (where `H = maze.length` and `W = maze[0].length`), or `Integer.MAX_VALUE` if there is no path.
Use the template file `Maze.java`.

Input constraints:

- $3 \leq W, \quad H \leq 1000$
- Each `maze[i][j]` is either '.' or 'o'
- `maze[1][1]` and `maze[H-2][W-2]` are both equal to '.'.
- The first row, last row, first column, and last column all consist of walls ('o') (see the examples above)

Scoring:

- 1 point for correct and fast implementation

An $O(WH)$ algorithm is fast enough.