COM2031 Advanced Algorithms, Autumn Semester 2019

Lab 1: Minimum Missing Element

Purpose of the lab

Core: This lab explores the algorithms discussed in the Week 1 lectures to find the minimum positive integer value not contained in an input set of values. You are asked to implement several of the algorithms discussed and consider their computational complexity by running them on inputs of various sizes.

Extensions: The lab also asks you to adapt your algorithm to solve a 2-D variation of the problem: to find the unoccupied point on a grid closest to the origin according to the "Manhattan Distance".

Getting started

Download the zip file COM2031_Lab01.zip into Eclipse. This includes a template you will write your solutions into below. It also includes unit tests for you to test your solutions.

1. Minimum missing element

Implement three of the algorithms as described in the lectures, to find the solution to the following problem:

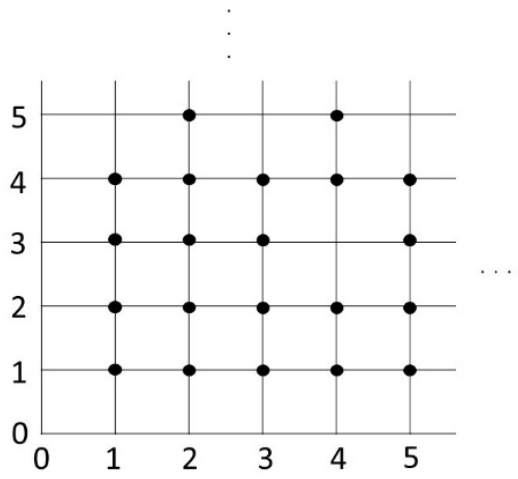
> **Input:** An array of positive integers (possibly including repeats)
>
> **Output:** The minimum positive integer not in the array

1. Implement the brute force algorithm from slide 45 as **findMissingMinimum_Brute_Force**
2. Implement the sorting approach from slide 58 of (a) sorting the values and the (b) finding the minimum missing number, as **findMissingMinimum_Sorting**
3. Implement the radix-sorting approach as explained from slide 61 as **findMissingMinimum_Radix**

In each case consider the time complexity of the algorithm by evaluating it on the provided input problems of varying sizes.

2. Grid point closest to origin – Manhattan distance

Now we consider a problem similar to the problem above, but in two dimensions. Consider a 2-dimensional grid of points with positive integer coordinates. For example, in the figure below, there is a point at (3,2), whereas there is not a point at (3,5).



The **Manhattan Distance** from a point to the Origin (0,0) is the distance taken to travel from (0,0) to the point if you can only travel horizontally or vertically along the grid lines. For example the Manhattan Distance from (0,0) to (3,2) is 3+2 = 5.

You are now asked to provide an algorithm to solve the following problem:

> **Input:** An array of positive integer coordinates on a 2-D grid, corresponding to a set of points
>
> **Output:** Coordinates of the unoccupied point with positive integer coordinates with minimal Manhattan Distance to (0,0)

e.g. in the figure above the solution would be the point (1,5): this is the unoccupied point closest to (0,0) by Manhattan Distance,

If there are two points with equal Manhattan Distance then you should take the leftmost one. For example, if (1,3) and (3,1) are both possible solutions then you should take the point (1,3).

You are provided with a Class called Point, which manages points on the grid. It contains an x coordinate and a y coordinate, and also allows for Points to be compared for equality and also compared to see which one is closer to the Origin according to the requirements of the problem above. In particular $(x,y) \leq (z,w)$ if

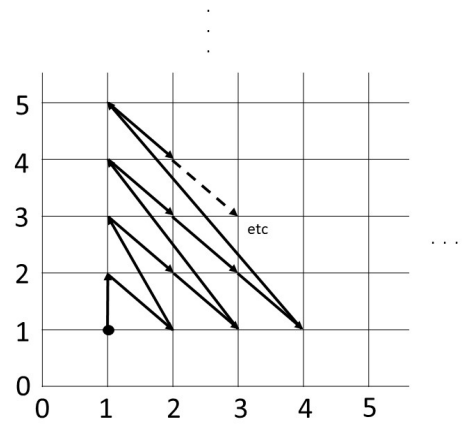
- either: $x+y < z+w$ (the Manhattan distance to (x,y) is less than the distance to (z,w))
- or: $(x+y = z+w)$ and $x \leq z$ (the Manhattan distances are equal and (x,y) is more to the left)

This provides a linear order on points.

2.1 Brute Force

The first approach will be to apply the brute force approach to find the minimal missing point. To do this you need to go through all the possibilities in order, starting from (1,1), and see whether it

appears in the collection of Points. To go through them in order you will need to work out the next Point to look for. The following diagram illustrates which Point is next from any given Point:



Implement the method: **nextPoint,** defined mathematically as follows:

nextPoint(x,y) = (x+1, y-1) if y > 1
= (1,x+1) if y = 1

For Brute force search you need to consider each point in turn to see if it is contained in the set of points, until you find the first point that does not appear.

Implement a brute force search to find the minimal missing point.

2.2 Using the Sorting approach

Now apply the sorting algorithm approach, by sorting the set of points using the ordering between points, and then search through the points from the beginning until you reach one that does not appear. Since the Points Class has a linear order you can use a standard sort method to sort the Points into order, and then examine each point in turn.

Implement the approach of sorting and then searching to find the minimal missing point.

Further challenges to think about:

3.1 Can you see how you might be able to adapt the Radix approach to solve the Grid Point Closest to Origin problem?

3.2 [Harder] Can you find the grid point with the least **Euclidean distance** $\sqrt{(x^2 + y^2)}$ to Origin (where we want the leftmost point if two have the same Euclidean distance) In the figure above this is the point (4,3).

3.2.1 Hint: work with the square of the Euclidean distance instead (i.e. $x^2 + y^2$), since the point with the smallest Euclidean distance will also be the point with the smallest square of the Euclidean distance

3.2.2 Hint: work out what the **nextPoint** function will be again – this is the difficult part

Testing and visualizing your implementations:

The zip file COM2031_Lab01.zip contains a java project that you can import into Eclipse. You will find three classes. Start with ***FindMinMissing.java***. The main method calls 2 methods. First is ***startSimulating***. The argument is a Boolean that controls whether you want to display a chart of the simulations results that will be explained in a following paragraph. The second method call is to test using all test cases that will be described in a following paragraph.

The ***startSimulating*** method calls the three algorithms discussed above inside a loop that controls the input size and number of simulations. You will find txt files with prefix "number_n.txt" in which n is the size of the array of elements. Each of these files has a sequence of numbers of 1 to n + 1 with missing value. Test Your algorithm in these files, but you might need to reduce the number of iterations starting from 1 file only, until your logic is working very well, then add another test case, and so forth. You can also start with smaller files of size n = 10 for instance.

To visualize the performance difference, the code adds a series for every algorithm, in which it adds points of the data size (on the x-axis) and the elapsed time (on the y-axis) when calling the algorithm method. The more simulations you run, the more points you will have. If the Boolean passed to the ***startSimulating*** method is true, a chart is displayed of these series. This chart is implemented using the second class found in the src folder "***chartComparisons.java***".

The second method called in the main "***testFindMinMissing***", is for testing all possible minimum values for arrays of sizes of 1 to 10 for the three algorithms you implemented. If you receive no failure assertions, then everything is covered in your code. Otherwise, continue debugging until it works fine.

The third class is "***Point.java***". It implements a 2D grid points. The main method mimics the example above, by creating a list of points for grid size 5x5, add all points to a list of points as occupied, and only removes the missing points above. It defines the closest one, then calls the three algorithms methods using the same naming convention and assert that the returned minimum is what is the closest to the origin indeed. Test until this test case is successful, and use the similar method to the above, here it is called "***testFindMinMissing2D***" for complete test cases to take care of.