THE UNIVERSITY OF SYDNEY

INFO1103 Assignment 2

Due: 9:00pm Monday, 2 May 2016

This assignment is worth 10% of your final assessment

Task description

In this assignment we will implement a vector computation engine that is able to construct vectors and perform vector computations in the Java programming language. Your task is to implement the incomplete functions in the provided scaffold code based on the included examples and documentation.

You are encouraged to ask questions on Ed using the assignments category. As with any assignment, make sure that your work is your own, and you do not share your code or solutions with other students.

Working on your assignment

You can work on this assignment on your own computer or the lab machines. Students can also take advantage of the online code editor on Ed. Simply navigate to the assignment page and you are able to run, edit and submit code from within your browser. We recommend that you use Safari or Chrome.

It is important that you continually back up your assignment files onto your own machine, flash drives, external hard drives and cloud storage providers. You are encouraged to submit your assignment while you are in the process of completing it. By submitting you will obtain some feedback of your progress.

Academic declaration

By submitting this assignment you declare the following:

Implementation details

Your task is to implement a vector computation engine based on the included scaffold code and the examples that are shown below. You can assume only valid input will be tested and that any commands that use vectors will be defined beforehand. All vectors will have the same length defined by $1 \leqslant length \leqslant 100,000,000$. Vector elements are indexed from 0 and can contain values up to and including the maximum value of `long`. Our test cases will not cause any integer overflows to occur.

Your program must be contained in the `Vector.java` and `VectorEngine.java` files and produce no errors when compiled using `javac` on the lab machines and Ed. Your program will be run using `java VectorEngine <length>` and read from standard input and write to standard output.

The `VectorEngine.java` file contains the `main` entry point method to the program, and is responsible for managing the list of declared vectors, reading from standard input, delegating input commands to the respective methods declared in the `Vector` class and then displaying output. This functionality has been provided in `VectorEngine.java` and should not need to be modified.

The `Vector.java` file contains the scaffold code for the `Vector` class. Method stubs have been created for the methods that you are expected to implement. Each of these stub methods contains a comment with some sample input and output to help you understand how these methods should work.

The `Vector` class contains some instance variables that you can utilise to improve the performance of your program. Since each vector is immutable we know that the results of each computation will never change, so there is no need to calculate the same result multiple times. For example, instead of calculating the sum every time the `getSum` method is called, you can instead calculate the sum once and store the result in the `sum` instance variable then simply return this value in subsequent calls to `getSum`. In addition, based on how each vector is constructed, you may be able to deduce some results based on a mathematical calculation at the time of construction. For example, suppose a vector was constructed with the uniform command, then we can immediately deduce the mode of the vector.

Running the program

The program requires you to specify the length of all vectors used in the current instance of the vector computation engine. The length is read by the program as a command line argument. For example, if you wanted every vector to contain 4 elements then you would run the program like this:

```
$ java VectorEngine 4
```

Commands

Once the program has started, you are prompted to enter the following commands:

- **`bye`** – Terminates the program.
- **`help`** – Displays the syntax of each command.
- **`set`** – Creates a new vector with the given key.
- **`show`** – Displays the vector with the given key.
- **`compute`** – Displays the result of a computation.

`set` command

The **`set`** command has the following syntax: **`set <key> = <operation> <args...>`**

- **`key`** – A string with no spaces that will be used to identify the vector.
- **`operation`** – The operation used to generate the vector.
- **`args`** – One or more optional arguments that can be passed to the operation.

Valid operations are:

- **`random <seed>`**

  Generates a new vector where each element is a random integer between 0 and 100 inclusive using the seed value. Generated using the `java.util.Random` random number generator.

- **`uniform <value>`**

  Generates a new vector where each element is set to the given **`value`**.

- **`sequence <start> <step>`**

  Generates a new vector where each element is a value of a sequence of integers starting at **`start`** and every next value is incremented by **`step`** which can be both positive and negative.

  For example, a sequence starting at 2 that has a step 2 would be:

  `2 4 6 8 10 12 ...`

- **`pq <start>`**

  Generates a new vector that consists of a sequence of pq numbers. First number in the vector should be **`start`** when pq or the next pq number that comes after.

- **`prime <start>`**

  Generates a new vector that consists of a sequence of prime numbers. First number in the vector should be **`start`** when prime or the next prime number that comes after.

- **`abundant <start>`**

  Generates a new vector that consists of a sequence of abundant numbers. First number in the vector should be **`start`** when abundant or the next abundant number that comes after.

- **`composite <start>`**

  Generates a new vector that consists of a sequence of composite numbers. First number in the vector should be **`start`** when composite or the next composite number that comes after.

- **`cloned <key>`**

  Generates a new vector with the same elements of the given vector.

- **`sorted <key>`**

  Generates a new vector with elements of the given vector in ascending order.

- **`shifted <key> <amount>`**

  Generates a new vector with elements of the given vector shifted right `n` places.

  For example:

  $[1, 2, 3, 4]$ shifted right by 1 place would be $[4, 1, 2, 3]$.
  $[1, 2, 3, 4]$ shifted right by 2 places would be $[3, 4, 1, 2]$.
  $[1, 2, 3, 4]$ shifted right by 3 places would be $[2, 3, 4, 1]$.
  $[1, 2, 3, 4]$ shifted right by 4 places would be $[1, 2, 3, 4]$.
  $[1, 2, 3, 4]$ shifted right by 5 places would be $[4, 1, 2, 3]$.

- **`reversed <key>`**

  Generates a new vector with elements of the given vector in reverse order.

- **`scalar#add <key> <value>`**

  Generates a new vector that is the result of adding the scalar to each element in the vector.

- **`scalar#mul <key> <value>`**

  Generates a new vector that is the result of multiplying the scalar to each element in the vector.

- **`vector#add <key1> <key2>`**

  Generates a new vector that is the result of adding the respective elements of both vectors.

- **`vector#mul <key1> <key2>`**

  Generates a new vector that is the result of multiplying the respective elements of both vectors.

`show` command

The **`show`** command has the following syntax: **`show <key> <index>`**

- **`key`** – The key of the vector to display.
- **`index`** – Optional index. When specified, display the element at this index.

`compute` command

The `compute` command has the following syntax: `compute <operation> <key> <arg>`

- `operation` – The operation to perform on the vector.
- `key` – The key of the vector to perform the given operation on.
- `arg` – An optional argument that may be required for the given operation.

Valid operations are:

- `sum <key>`

  Display the sum of all vector elements.
- `mode <key>`

  Display the most occurring value in the vector, or -1 if the most ocurring number is not unique.
- `median <key>`

  Display the upper median which is the value that lies in the middle of the vector when sorted. When there are an even number of elements in the vector, then display the upper median.
- `minimum <key>`

  Display the minimum value in the vector.
- `maximum <key>`

  Display the maximum value in the vector.
- `frequency <key> <value>`

  Display the number of occurrences of the value in the vector.

Example 1

```
$ java VectorEngine 4
```

```
> SET a = uniform 4
ok

> SHOW a
4 4 4 4

> SET b = sequence 3 2
ok

> SHOW b
3 5 7 9

> SET c = random 1
ok

> SHOW c
97 5 21 41

> SET d = pq 100
ok

> SHOW d
106 111 115 118

> SET e = prime 100
ok

> SHOW e
101 103 107 109

> SET f = abundant 100
ok

> SHOW f
100 102 104 108

> SET g = composite 100
ok

> SHOW g
100 102 104 105

> BYE
bye
```

Example 2

```
$ java VectorEngine 4
```

```
> SET a = uniform 2
ok

> SHOW a
2 2 2 2

> SET b = sequence 1 1
ok

> SHOW b
1 2 3 4

> SET c = scalar#add b 4
ok

> SHOW c
5 6 7 8

> SET d = scalar#mul b 4
ok

> SHOW d
4 8 12 16

> SET e = vector#add a b
ok

> SHOW e
3 4 5 6

> SET f = vector#mul a b
ok

> SHOW f
2 4 6 8

> BYE
bye
```

Example 3

```
$ java VectorEngine 4
```

```
> SET a = sequence 1 1
ok

> SHOW a
1 2 3 4

> SET b = cloned a
ok

> SHOW b
1 2 3 4

> SET c = sorted a
ok

> SHOW c
1 2 3 4

> SET d = shifted a 3
ok

> SHOW d
2 3 4 1

> SET e = reversed a
ok

> SHOW e
4 3 2 1

> BYE
bye
```

Example 4

```
$ java VectorEngine 10
```

```
> SET a = random 59
ok

> SHOW a
62 84 32 56 98 97 32 70 10 21

> SHOW a 9
21

> COMPUTE sum a
562

> COMPUTE mode a
32

> COMPUTE median a
62

> COMPUTE minimum a
10

> COMPUTE maximum a
98

> COMPUTE frequency a 32
2

> COMPUTE frequency a 22
0

> BYE
bye
```

Writing your own testcases

We have provided you with some test cases but these do not not test all the functionality described in the assignment. It is important that you thoroughly test your code by writing your own test cases.

Submission details

Final deliverable for the performance will be due in week 9. Further details will be announced on Ed.

You must submit your code using the assignment page on Ed. To submit, simply place your files and folders into the workspace, click run to check your program works and then click submit.

You are encouraged to submit multiple times, but only your last submission will be marked.

Marking

In this assignment, you will receive marks for correctness and performance of your program. However, submissions are only eligible for the performance component if they pass all of the correctness tests.

8 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will use our own hidden test cases that cover every aspect of the specification. To pass test cases your solution must produce the correct output within the imposed time limit.

2 **marks** are assigned based on the *performance* of your code relative to the benchmark.
This component is tested on a separate machine in a consistent manner. Submissions faster than the benchmark implementation will receive 2 marks, with successively slower ones receiving lower marks. Submissions faster than our basic implementation will receive at least 1 mark.

Your program will be marked automatically, so make sure that you carefully follow the assignment specifications. Your program must match the exact output in the examples and the test cases on Ed.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specification, or your code is unnecessarily or deliberately obfuscated.