Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
Final Examination: Part 2 of 2

Guy-Vincent Jourdan, Mehrdad Sabetzadeh, and Marcel Turcotte

April 2020, duration: 2 hours

Instructions

1. This is an open book examination and the only authorized resources are:
   - Course lecture notes, laboratories, and assignments;
   - Java Development Kit (JDK) on your local computer.
2. By submitting this examination, you agree to the following terms:
   - You understand the importance of professional integrity in your education and future career in engineering or computer science.
   - You hereby certify that you have done and will do all work on this examination entirely by yourself, without outside assistance or the use of unauthorized information sources.
   - Furthermore, you will not provide assistance to others.
3. Anyone who fails to comply with these terms will be charged with academic fraud.

Marking scheme

Question	Maximum
1	35
2	5
Total	**40**

Question 1 (35 marks)

A **sparse vector** is a **vector** of numerical values in which many entries have a value of zero. Such vectors find applications in data science and machine learning. Implementing a sparse vector using an array would result in inefficient operations and a waste of memory. Here, we propose a compressed representation using linked elements. Specifically, the implementation only stores the values that are non-zero. The values that are not physically stored in the list are all zero (0).

The following examples illustrate the concepts as well as the intended behaviour. First, a sparse vector of dimension 1024 is created.

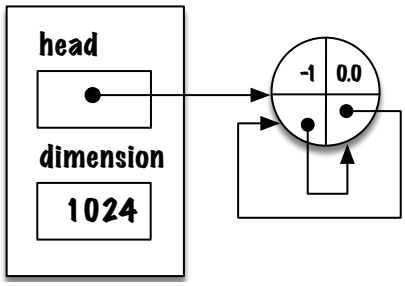
```
SparseVector v;
v = new SparseVector(1024);

System.out.println(w.getDimension());
```

Executing the above program produces the following output:

```
1024
```

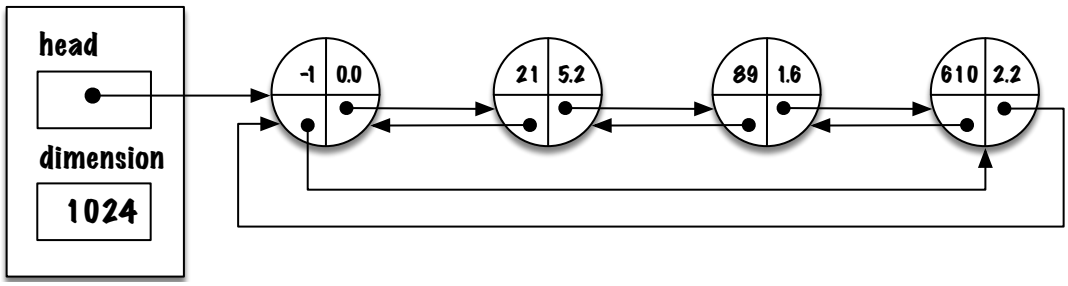
The figure below shows the corresponding memory diagram. Initially, the list consists only of the dummy node.



Now calling the method **set** three times:

```
v.set(21, 5.2);
v.set(610, 2.2);
v.set(89, 1.6);
```

Changes the memory representation as follows.



In the second example, a sparse vector of dimension 5 is created. The value at position 2 is set to 42.0. The content of the sparse vector is then printed. Notice that the index of the first value is 0.

```
SparseVector w;
w = new SparseVector(5);

w.set(2, 42.0);

for (long d=0; d<w.getDimension(); d++) {
    System.out.println(w.get(d));
}
```

Executing the above program produces the following output:

```
0.0
0.0
42.0
0.0
0.0
```

Finally, in the third example, a sparse vector of dimension **Long.MAX_VALUE** is created:

```
SparseVector x;
x = new SparseVector(Long.MAX_VALUE);

System.out.println(x.getDimension());
```

Executing the above program produces the following output:

```
9223372036854775807
```

In this compressed representation, the number of elements (nodes in the list) is always equal to the number of non-zero values in the sparse vector (plus the dummy node), not its dimension, which is why we can create a sparse vector with such large dimension. In the first example, the sparse vector designated by **v** has only three non-zero values, and thus the list contains only three elements (plus the dummy node). In the last example, the sparse vector designated by **x** has no non-zero values, its list contains only the dummy node.

Files

- `SparseVector.java`
- `Iterator.java`
- `TestSparseVector.java` (unit tests)
- `TestIterator.java` (unit tests)

Question 1.1 Elem (4 marks)

In the classs **SparseVector**, implement a nested class called **Elem** to store the values of the sparse vector. Notice that neither **SparseVector** nor **Elem** have a type parameter.

- The list of elements always starts with a dummy node, which is used as a marker for the beginning of the list. The dummy node is never used to store data. The empty list is made up of the dummy node only;
- Each element in the list stores an index (of type **long**) as well as a value (of type **double**);
- In the implementation for this question, the nodes in the list are doubly linked;
- In this implementation, the list is circular, i.e. the reference from the last node in the list points to the dummy node, the reference **previous** of the dummy node points to the last node in the list;
- In the empty list, the dummy node is the first and last node of the list, the references **previous** and **next** point to the node itself;
- The last node is easily accessible, because it's always the node preceding the dummy node, the header of the list does not have a pointer to the tail element.

Question 1.2 Instance variables (2 marks)

Declare the necessary instance variables with the appropriate visibility, as seen in class.

Question 1.3 SparseVector(long dimension) (3 marks)

Implement the constructor **SparseVector(long dimension)**, which constructs a sparse vector with the specified **dimension**. A sparse vector has a fixed dimension, specified when a sparse vector object is created. The valid indices for a sparse vector are in range 0 to **dimension-1**.

Question 1.4 long getDimension() (1 mark)

Write the instance method **getDimension()**, which simply returns the dimension of this sparse vector.

Question 1.5 void set(long index, double value) (8 marks)

Implement the method **set(long index, double value)**. It replaces the value at the specified position in this sparse vector with the specified value. The method throws an exception of type **IndexOutOfBoundsException** if the specified index is out of range, less than 0 or greater than **getDimension()-1**. Make sure to preserve the property that a sparse vector does not physically store values that are zero.

Question 1.6 double get(long index) (4 marks)

Write the implementation of the method **double get(long index)**. The method returns the value at the specified position in this sparse vector. It throws an exception of type **IndexOutOfBoundsException** if the specified index is out of range, less than 0 or greater than **getDimension()-1**.

Question 1.7 double getL1Norm() (3 marks)

Implement the method **double getL1Norm()**. The l_1 norm of a vector is simply the sum of the absolute values of its elements. Specifically, if x is a vector with dimension n, the l_1 norm of x is defined as follows:

$$||x||_1 = \sum_{i=0}^{n-1} |x_i|.$$

The execution time must be proportional to the number of non-zero values in the sparse vector, not its dimension.

Reminder: Math.abs(double a) returns the absolute value of a double value.

Question 1.8 Iterator getIterator() (2 marks)

Write the instance method **getIterator()**, which returns an iterator over this sparse vector.

Question 1.9 SparseVectorIterator (8 marks)

Implement an iterator over a sparse vector. Notice that neither **SparseVectorIterator** nor **Iterator** have a type parameter. For this question, the interface is defined as follows.

```
public interface Iterator {
    boolean hasNext();
    double next();
}
```

The iterator will return all the values of the sparse vector, **zero and non-zero values**, one at a time. See the example below.

- Declare a nested class called **SparseVectorIterator** implementing the interface **Iterator**;
- The method **boolean hasNext()** returns **true** if the iteration has more values;
- The method **double next()** returns the next value in the iteration.

The implementation of the class **SparseVectorIterator** should be such that an iteration through the entire sparse vector visits each element of the list only once.

The following example displays the intended behaviour.

```
SparseVector v;
v = new SparseVector(5);

v.set(2, 42);

Iterator i;
i = v.getIterator();

while (i.hasNext()) {
    System.out.println(i.next());
}
```

Executing the above program produces the following output:

```
0.0
0.0
42.0
0.0
0.0
```

For additional examples, consult the unit tests.

- `TestSparseVector.java`
- `TestIterator.java`

Question 2 (5 marks)

For this question, **n** digital documents are stored on **k** USB keys (flash drives). The method **getMaxDiff** is an auxilliary method for a software system to find the optimum placement of the documents. You must complete the implementation of the method **getMaxDiff(int[] sizes, int[] map, int k)**, where the parameters have the following meanings:

- the parameter **sizes** designates an array of length **n**, where the element **i** of the array is the size of the digital document **i**.
- the parameter **map** designates an array mapping the **n** digital documents to one of the **k** USB keys. Specifically, the element **i** of the array is the index of the designated USB key.
- **k** is the number of USB keys.

In order to illustrate the concepts, consider the example below:

```
@Test
public void test3() {

    int[] sizes = new int[] {100, 10, 70, 50, 20, 30, 40};

    int[] map = new int[] {0, 1, 2, 0, 1, 1, 1};

    int k = 3;

    assertEquals(80, MediaLibrary.getMaxDiff(sizes, map, k));

}
```

In the above example, there are 7 digital documents. Accordingly, the arrays designed by **sizes** and **map** are of length 7. The array designated by **sizes** contains the sizes of the 7 documents, e.g. document 0 has size 100, document 1 has size 10, document 2 has size 70, etc. The array designed by **map** defines the mapping of the **n** documents to the **k** USB keys, e.g. document 0 goes to the USB key 0, document 1 goes to USB key 1, document 2 goes to USB key 2, document 3 goes to USB key 0, and the last three documents all go to USB key 1. Here, **k** has value 3.

- The method **getMaxDiff** returns the difference in size between the USB keys with minimum and maximum sizes, where the size of a USB key is the sum of the sizes of all the documents on the USB key.

You must complete the implementation of the method **getMaxDiff** for the class **MediaLibrary**. In the example above, the expected value is 80. For additional examples consult the unit tests.

Files

- `MediaLibrary.java`
- `TestMediaLibrary.java` (unit tests)

Rules and regulation

- Submit your examination through the on-line system Brightspace.
- You must do this examination individually.
- You must use the provided template classes below.
- It is your responsibility to make sure that Brightspace has received your examination.
- Late submissions will not be graded.

Files

- Download the archive **f2_3000000.zip**;
- Unzip the file and rename the directory, replacing **3000000** by **your student id**;
- Add your **name** and **student id** in a comment in **MediaLibrary.java** and **SparseVector.java**;
- Make sure to **add comments to your code**.

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **f2_3000000**, where 3000000 is your student number. The name of the folder starts with the letter "f" (lowercase), followed by 2, since this is part 2 of the examination. The segments are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. Your submission must contain the following files, and nothing else. In particular, do not submit the byte-code (.class) files.

- Copyright.txt
- README.txt
  - A text file that contains your name and student id, as well as a declaration of integrity.
- `Iterator.java`
- `MediaLibrary.java`
- `SparseVector.java`
- `TestIterator.java` (unit tests)
- `TestMediaLibrary.java` (unit tests)
- `TestSparseVector.java` (unit tests)