REPORT : SOFTWARE ENGINEERING PROJECT

BY ALEXIS LE FORESTIER

<u>Introduction :</u>

The goal of this project was implement a method of compression that does not lose direct access to the elements : Bitpacking.

<u>My approach :</u> An abstract class called BitPacking (see : BitPacking.java) that will act as the parent of three different classes (BitPackingV1, BitPackingV2, and BitPackingV3), each one being a different implementation of BitPacking.

I will use the BitSet class from the java.util package to work with bits, because it takes significantly less memory than an array of boolean.

My BitPacking class will have a method to find the lowest amount of bit needed to store the biggest integer of an array. That method will be called in the constructor because this value will be useful for BitPackingV1 and BitPackingV2.

<u>BitPackingV1 :</u>

This first version of BitPacking is able to write compressed integers on two consecutive integers.

Here’s how I handled the compression :

- First, I need to know how big will the compressed array be, so that I create an array of adequate size to store my compressed integers. I calculate the total amount of bits needed.
- I instantiate an object from the BitSet class that will store my compressed integers up until its full.
- I go through all the integers in my original array, shaving as many useless 0 as I can.
- Every time my BitSet is full, I convert it into an integer before storing it into the my final compressed array, and clearing my BitSet object.
- I repeat this loop until I filled my compressed array with as much bits as I calculated in the beginning.

How I handled the get(int I) method :

- I start by finding the first bit that will contain the first bit of the number I’m looking for in the compressed array.
- I then put it, along with the next nBitsPerInt bits into a BitSet I instantiated.
- I convert the BitSet into an integer, and return that value

For the decompression, I simply call the get() method for each element in the array.

BitPackingV2 :

This second version of BitPacking ensures that compressed integers are never written on two consecutive integers.

Here’s how I handled the compression :

- First I calculate how many numbers can fit in one compressed integer by dividing 32 by nBitsPerInt
- I determine the compressed array size by dividing total elements by this packing density
- I use a BitSet as a temporary buffer to accumulate bits from multiple numbers
- For each original number, I copy its nBitsPerInt significant bits into the BitSet at the correct position
- When the BitSet is full, I convert it to an integer, store it in the compressed array, and clear the BitSet
- I repeat until all numbers are processed.

How I handled the get(int i) method :

- I calculate which compressed integer contains the requested number
- I find the starting bit position within that compressed integer
- I extract exactly nBitsPerInt bits starting from that position
- I convert those bits back to an integer and return the value

For the decompression, I simply call the get() method for each element in the array, just like I did in BitPackingV1.

BitPackingV3 :

This third version of BitPacking include a “Overflow area” to store bigger values than the rest. The goal is to waste as little memory as possible.

Here's how I handled the compression :

- First I analyze the data distribution by finding the median value and calculating optimal bit sizes for both normal and overflow areas
- I split the data into two groups: values below the median (normal area) and values above (overflow area)

- For normal area values, I store the actual value using just enough bits plus one flag bit
- For overflow area values, I store a pointer to the overflow location in the normal area, then store the actual large value in a separate overflow area
- I use two separate BitSets to accumulate bits for both areas, converting them to integers when full
- Finally I combine both compressed areas into one final array

How I handled the get(int i) method :
- I calculate the bit position of the requested number in the normal area
- I extract the stored value along with its flag bit
- If the flag bit is 0, I return the value directly as it's a normal area value
- If the flag bit is 1, I interpret the value as a pointer to the overflow area
- I navigate to the overflow area using the pointer and extract the actual large value stored there
- I return the reconstructed value

For the decompression, same as for BitPackingV1 and BitPackingV2.

<u>Problems encountered :</u>

- For BitPackingV3, I didn't really know which metric to use differentiate between a "big integer" and a "small integer", so I use the median of the sorted array, which is probably not the best way, but that's the best one I could come up with.

- The biggest problem that I encountered is definitely how much duplicated code there is in my project. I really struggled with coming up with solutions for this project, so I went with a brute force approach of fixing problems as they come up, instead of thinking of a global idea for a solution. The result is a bit messy… I tried to refactor as much as I could, but any major attempt of refactoring broke my code, so I'm stuck with a rather difficult-to-read code (most notable in BitPackingV3). I deeply apologize about that.