V22.0310-001/002 Basic Algorithms April 21, 2010

Final-Project Assignment

Lecturer: Yevgeniy Dodis *Due: Wednesday, May 5*

Implementing SOLE gazzilion points

The goal of the final project is to create a program which can encode and decode streams of data on the fly according the SOLE algorithm. The focus will be on the back-end (i.e. the correctness of your implementation) but extra points will be awarded for a nice (clean and clear) user interface.[1] We strongly recommend that you use either C, C++ or Java. However, if you absolutely want to use another programming language then please first check with Yevgeniy to make sure it is OK.

Big Numbers: For most languages (such as C, C++ and Java) the normal integer data types will not have a large enough range so you will most probably need to use some extra libraries for computing with large integers. For C and C++ you can use the GMP library [GMP] while for Java you can use the java.math.BigInteger class [JBI].

Data Types: As a suggestion, it will probably make your life easier if you implement some sort of Stream type or class which is used to store both input and output code words. This class will have an array A which stores the data blocks in the word. (We recommend that each such block is stored as a big integer; see above.) You should also have a field *length*, which stores the current length n of the stream, a field *type* which indicates whether this is an input stream or an output stream, and a few other attributes you find useful. For example, you might want to store a flag *completed* to indicate if the stream has been completed (by EOF) or not, and, if not, whatever temporary variables are needed to process the next input/output block. For simplicity, I call all such fields (in addition to the array A) as state. Defining what "state" means is part of the project.

Functions: To simplify your task of implementing the SOLE algorithm, you should implement the following functions inside your main program. Below I give some guidance over the inputs to these procedures, but you can change them depending on the lower-level details of your implementation. Still, try to implement the procedures below as close as you can. Below I assume the value $B = 2^b$ is globally known.

SWITCHPAIR(x, y, A, B, A', B'): this function changes the representation (x, y) of a number from basis (A, B) to the new basis (A', B'). In other words, it takes in four positive integers A, B, A' and B', as well as two integers $x \in [A]$ and $y \in [B]$. (Implicitly this defines the number $z := xB + y$ which is to be represented in the new basis (A', B').) The function returns a pair of integers $(x', y') \in [A'] \times [B']$ such that:

$$xB + y = x'B' + y'.$$

Note that when $A * B \le A' * B'$, the operation always succeed, but otherwise the function may have to return an error when $xB + y \ge A' * B'$. Make sure you deal with such errors properly.

[1]Especially for a GUI.

PrintBlock($x, B, format$): this function should print a block $x \in [B]$, depending on the type. Currently, $format$ could be 'decimal' (meaning you print x as a decimal integer) or 'binary' (meaning you print x as binary integer). Also, when $x = B + 1$ (EOF), you should print "end-of-file".

GetBlock($y, B, format$): this function should take the input a string y representing a valid block of a given $format$ (decimal or binary) and return an integer representing it. Notice, as discussed above, such integer could be large when B is large.

NextBlockEncode($stream, x$): This function is used to process the next block of data, as it is received from the data source. As input it expects a *stream* variable (see above) and the value $x \in [B + 1]$ of the next block to be appended to the end of the stream. Once x has been processed, the function returns an updated *stream*, meaning that in addition to possibly adding new blocks to A, the *state* is updated as well. Notice, a proper implementation of this procedure will not read anything from your array A, but will only read/modify the data stored in the *state* component of your output *stream* (such as the current length, etc.), and possibly write several new output blocks to A.[2] Whenever any such block is written, you should also print it using the PrintBlock procedure.

Also notice that $x = B + 1$ indicates EOF. This also means that when the stream current length is even, you should append append two EOF symbols, as discussed in the notes.

NextBlockDecode($stream, x$) This function is analogous to the previous one, but for decoding. The main difference in the input is that now $x \in [B]$ rather then $x \in [B+1]$, since in an encoded stream x can not take on the special EOF symbols. As before, a proper implementation will not read anything from your array A, but will only read/modify the data stored in the *state* component of your input *stream* (such as the current length, etc.), and possibly write several new input blocks to A.[3] Whenever any such block is written, you should also print it using the PrintBlock procedure. Also, when you recover EOF, you should set the *copmpleted* flag accordingly.

Before discussing other functions, it is a good idea for you to test the correctness of your implementation by running the above inverse procedure against each other, to ensure that decoding the encoded stream will return the original stream.

ReadInputBlock($stream, i$) This function takes the output stream *stream* containing SOLE encoded data. It has to return (i.e. to decode) the i-th input block encoded in *stream*. Notice, in a proper implementation, this function will only access 4 output blocks to decode the corresponding input block.

ComputeOutputBlock($stream, i$) This function is analogous to the previous one, but for encoding. The only difference is that this time the stream contains the original input, and the procedure should return the i-th block of the SOLE encoding. Once again, in a proper implementation, this function will only access 4 output blocks to decode the corresponding input block.

[2]Notice, in normal situations no block is output after each odd block, and two blocks are output after each even block (except the first, when only one output block is output).

[3]Notice, in normal situations no block is output after each even block, and two blocks are output after each odd block (except the first when nothing is output yet).

ModifyInputBlock($stream, i, x'$) This function takes the output stream *stream* which contains a SOLE encoding of some data. It also takes in an index i and a new value $x' \in [B+1]$. It updates the output stream so that now it contains the SOLE encoding of the same data as before, except that the i-th input block was changed to the value x'. In other words it must:

1. Decode the i-th input block from the output *stream*.
2. Change the input value to x'.
3. Recompute the four encoded blocks of *stream* that are potentially affected by the new value and fix them in the stream.

Notice, in case $x' = B+1$, this really corresponds to truncating the file, so you should be careful to properly update all the state variables. Conversely, if the original block x was EOF, and $x' \neq$ EOF, you should interpret this operation as *appending* x' to the end of the stream. This means that at the end of this operation the stream is still completed and its length increased by 1.

Large Files: In your procedures you can assume that the main SOLE constraint $B \geq 2n^2$ (for odd n) should hold. I.e., if the input/out stream becomes too long, simply output an error. Honors students, however, should be able to handle arbitrary large values of n. Namely, if $n_0 = 2^{(b-1)/2} - 1$ is the largest length which can be handled and n gets above n_0, you should be able to start a new stream, as hinted in the notes.

Interface: While a command line based interface is sufficient, extra points will be awarded for additional features such as a GUI. Remember, the best project is eligible to be publish on the web as a reference implementation of the SOLE algorithm. So ideally we would like it to be as instructive, clear and user-friendly as possible.

Most importantly, the main program should some variant of the following minimal interface (feel free to add extra stuff!).

- **Step 1:** ask for the value of block length $b > 0$.
- **Step 2:** ask whether the block should be entered in binary or decimal (e.g., for $b = 5$ if you expect 00111 or 7)
- **Step 3:** ask which of the following operations should be done:
    1. *Enter fresh stream for encoding.* Here you enter the input blocks (in appropriate format) one by one, use the GetBlock procedure to convert them to integers x, and then incrementally build your output stream using the NextBlockEncode($stream, x$) procedure. After each block, you should also print your current state (whatever it is) and if you output any output blocks. Entering EOF (e.g., anything in the wrong format) exits this process.
    2. *Enter fresh stream for decoding.* Here you enter the output blocks (in appropriate format) one by one, use the GetBlock procedure to convert them to integers x and then incrementally build your input stream using the NextBlockDecode($stream, x$) procedure. After each block, you should also print your current state (whatever it is) and if you output any input blocks. The exit is done the moment EOF is printed.

3. *Recover some input block i.* Here the user first enters i, and then is asked to enter 4 appropriate output blocks. At the end, the corresponding input block i is output (in the appropriate format). Notice, since there is no stream in this interface, you might not be able to directly call the ReadInputBlock($stream, i$) procedure, but can probably find a way to use large chunks of its code to accomplish this task.
4. *Compute some output block i.* Here the user first enters i, and then is asked to enter 4 appropriate input blocks. At the end, the corresponding input block i is output (in the appropriate format). Notice, since there is no stream in this interface, you might not be able to directly call the ComputeOutputBlock($stream, i$) procedure, but can probably find a way to use large chunks of its code to accomplish this task.
5. *Modify some input block i.* Here the user first enters i and a block value x', and then is asked to enter 4 appropriate output blocks. At the end, you output the modified 4 output block to change the i-th input block to x'. Notice, $x' = \text{EOF}$ corresponds to file truncation. Once again, since there is no stream in this interface, you might not be able to directly call the ModifyInputBlock($stream, i, x'$) procedure, but can probably find a way to use large chunks of its code to accomplish this task.

   For honors students, when original i-th block x was EOF, you should output 4 or 6 new output blocks resulting by appending x' to the stream (depending on whether i was even or odd).
6. *Quit.*

What to Submit: You should email your projects to Joel, and include the following two attachments.

- A zip file containing all your source files, documentation, etc. required to run your program. After unzipping your file in a fresh directory, it should be very easy to test your implementation. Make sure that there is a separate file containing *only your concise implementation of the required procedures*, but nothing else (GUI, putting all together, etc.). Try to put good comments in your code.

- Another file (text, word, pdf, etc.) explaining precisely how to run your program and what interface to expect. Make sure to include examples of a few sample runs of your program, illustrating at least the five possible operations I want you to implement (new encode, new decode, read input block, read output block, modify input block, but feel free to include more). The file should also include whatever else you want to tell us.

Teams? Given that I post these details a bit late, I do not mind if you work in teams of at most two people each.

Good luck!

References

[GMP] GNU Multiple Precision Library. `http://gmplib.org/`

[JBI] Java Big Integer Class: java.math.BigInteger `http://java.sun.com/javase/6/docs/api/java/math/BigInteger.html`