Homework 4

1. Algebraic Operations on Binary Values

In this exercise you are required to implement *addition* and *multiplication*, as efficiently as possible. One way to maximize algebraic efficiency is to operate directly on the binary representation of the manipulated numbers. Three such *bitwise algorithms* are described in the appendix at the end of this document. We recommend to read this appendix now. Note that for completing this exercise you need to understand how the algorithms work, not why they are efficient.

The supplied **`BinOps`** class represents binary numbers using integer arrays of length n=16, in which each value is `0` or `1`. The algorithms described in the appendix operate efficiently on any given n, and thus the 16 constant is not important. Further, theses algorithms can be implemented in either hardware and software.

- Start by implementing and testing the `bin2String` function.
- Then implement and test the `add` function, following the addition algorithm presented in the appendix.
- Next, implement the `shift` and then the `mult` functions, following the multiplication algorithm presented in the appendix.
- Finally, implement the `int2bin` and `bin2int` functions. Lecture 2-2 presented *binary-to-decimal* and *decimal-to-binary* algorithms. The former algorithm was inefficient, and both algorithms operated on strings rather than arrays. So now you need to come up with a better *binary-to-decimal* algorithm, and implement both algorithms in the **`BinOps`** class.

Read the appendix now, and then follow the implementation guidelines.

Implementation guidelines

- In the *addition* and *multiplication* algorithms, proceed from right to left, and operate on all n bit-pairs.
- In all the algorithms, if the left-most carry bit is `1`, ignore it. The resulting value will be correct up to n bits, and that's fine.
- In `add(int[] x, int[] y)` and `mult(int[] x, int[] y)` you can assume that `x` and `y` are of the same length.
- In `int2bin(int x)` it is safe to assume that `x` is positive and smaller then 2^{16}, so that the retuned array is of length n=16.
- In all functions you can assume that input arrays (`int[] x` and `int[] y`) are binary, that is, the array values are either 0 or 1.

Examples:

```
% java BinOps 15 add 5
20
% java BinOps 15 mult 5
75
```

Appendix: Algorithms[1]

Computers perform algebraic operations like *addition*, *subtraction*, *multiplication* and *division* by operating directly on *n*-bit binary values, *n* typically being 16, 32 or 64 bits, depending on the operands' data types. Ideally, we seek algebraic algorithms whose running time is proportional in this parameter *n*. Algorithms whose running time is proportional to the *values* of *n*-bit numbers are unacceptable, since these values are exponential in *n*. For example, suppose we implement the multiplication operation $x * y$ naïvely, using the repeated addition algorithm `sum = 0`, `i = 1` ... *y* {`sum = sum +` *x*}, `return sum`. If *y* is 64-bit wide, its value may well be greater than 9,000,000,000,000,000, implying that the loop may run for billions of years before terminating.

In sharp contrast, the running times of the algebraic algorithms that we present below are proportional not to the *values* of these *n*-bit numbers, which may be as large as 2^n, but rather to *n*, the number of their digits. When it comes to algebraic efficiency, that's the best that we can possibly hope for.

Addition

The addition of three bits generates one *sum* bit and one *carry* bit. This operation is defined as follows:

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

To add two *n*-bit binary values, we proceed from right to left, adding each pair of bits and the carry bit from the previous addition. The first carry bit is 0. Here is an example (*n*=16):

carry:	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
x:	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0
+																
y:	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
sum:	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

26
+
15
41

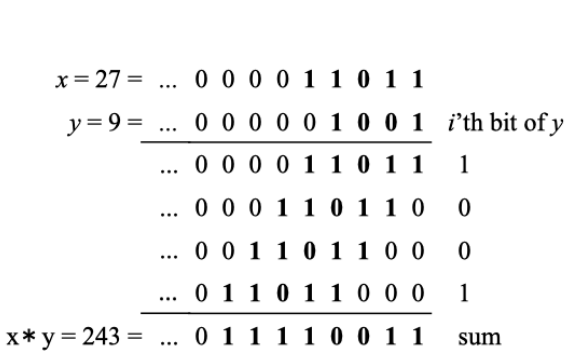
The running time of this addition algorithm is the time it takes to perform *n* 3-bit additions. Formally, we say that the running time is a polynomial function of *n*, the number of bits (in this example, *n*=16).

Technical comment: In principle, we can stop the bitwise addition operations once we have only `0` bits remaining in both operands. However, since in the worst case we have to perform *n* iterations, and since *n* is always a small constant like 16, 32, or 64, we'll always operate on *all* the bit-pairs, from right to left.

[1] This appendix is an excerpt from *The Elements of Computing Systems*, by Nisan and Schocken.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 73, we line up the two numbers one on top of the other, right-justified. Next, we multiply 356 by 3. Next, we "shift" 356 to the left one position, and multiply 3560 by 7 (which is the same as multiplying 356 by 70). Finally, we sum up the columns and obtain the result. This procedure is based on the insight that $356 \times 73 = 356 \times 70 + 356 \times 3$. The binary version of this procedure is illustrated below, using another example:




```
// Returns x * y, where x, y ≥ 0.
mult(x, y):
   sum = 0
   shiftedx = x
   for i = 0 ... n - 1  do {
      if ((i'th bit of y) == 1)
         sum = sum + shiftedx
      shiftedx = 2 * shifted
   }
   return sum
```

Let's inspect the multiplication procedure illustrated at the left of this figure. For each *i*'th bit of *y*, we "shift" *x i* times to the left (same as multiplying *x* by 2^i). Next, we look at the *i*'th bit of *y*: If it's 1, we add the shifted *x* to an accumulator; otherwise, we do nothing. The algorithm shown on the right formalizes this procedure. Note that $2 * shiftedx$ can be computed efficiently by either left-shifting the bit-wise representation of *shiftedx*, or by adding *shiftedx* to itself.

Running time: The multiplication algorithm performs *n* iterations, where *n* is the bit-width of the inputs. In each iteration, the algorithm performs a few addition and comparison operations. It follows that the total running-time of the algorithm is $a + b \cdot n$, where *a* is the time it takes to initialize a few variables, and *b* is the time it takes to perform a few addition and comparison operations. Formally, the algorithm's running time is a polynomial function of *n*, the bit-width of the inputs.

To reiterate, the running time of this $x \times y$ algorithm does not depend on the values of *x* and *y*; rather, it depends on the *bit-width* of the inputs. In computers, the bit-width is normally a small constant like 16 (`short`), 32 (`int`), or 64 (`long`), depending on the data types of the inputs. Suppose that the data type is `short`. If we assume that each iteration of the multiplication algorithm entails about 10 machine instructions, it follows that each multiplication operation will require at most 160 computer cycles, irrespective of the size of the operands. Compare this to the $10 \cdot 2^{16} = 655{,}350$ cycles that will be needed by algorithms whose worst case running time is proportional not to the *bit-width* , but rather to the *values*, of their inputs.

Technical comment: Same as in the end of the addition algorithm.

Division

(This is an optional section. You don't have to read it nor implement the division algorithm).

The naïve way to compute the division of two *n*-bit numbers *x* / *y* is to count how many times *y* can be subtracted from *x*, until the remainder becomes less than *y*. The running time of this algorithm is proportional to the value of the dividend *x*, and thus is unacceptably exponential in the number of bits *n*.

To speed up things, we can try to subtract large chunks of *y*'s from *x* in each iteration. For example, suppose we have to divide 175 by 3. We start by asking: What is the largest number, *x* = (90, 80, 70, ... 20, 10) so that $3 \cdot x \leq 175$? The answer is 50. In other words, we managed to subtract 50 3's from 175, shaving 50 iterations from the naïve approach. This accelerated subtraction leaves a remainder of $175 - 3 \cdot 50 = 25$. Moving along, we now ask: What is the largest number, *x* = (9, 8, 7, ... 2, 1) so that $3 \cdot x \leq 25$? The answer is 8, so we managed to make 8 additional subtractions of 3, and the answer, so far, is 50 + 8 = 58. The remainder is $25 - 3 \cdot 8 = 1$, which is less than 3, so we stop the process and announce that 175 / 3 = 58 with a remainder of 1.

The technique just described is the rationale behind the dreaded school procedure known as *long division*. The binary version of this algorithm is identical, except that instead of accelerating the subtraction using powers of 10 we use powers of 2. The algorithm performs *n* iterations, *n* being the number of digits in the dividend, and each iteration entails a few multiplication (actually, shifting), comparison, and subtraction operations. Once again, we have an x/y algorithm whose running time does not depend on the values of *x* and *y*. Rather, the running time is a polynomial function of *n*, the bit-width of the inputs.

Writing down this algorithm as we have done for multiplication is not difficult. To make things interesting, the following figure presents another division algorithm which is just as efficient, but more elegant and easier to implement.

```
// Returns the integer part of x / y,
// where x ≥ 0 and y > 0.
divide(x, y):
    if  (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

Suppose we have to divide 480 by 17. The algorithm shown above is based on the insight: $480/17 = 2 \cdot (240/17) = 2 \cdot (2 \cdot (120/17)) = 2 \cdot (2 \cdot (2 \cdot (60/17))) = ...$, and so on. The depth of this recursion is bounded by the number of times *y* can be multiplied by 2 before reaching *x*. This also happens to be, at most, the number of bits required to represent *x*. Thus the running time of this algorithm is also a polynomial function of *n*, the bit-width of the inputs.

One snag in this algorithm is that each multiplication operation also requires *n* operations. However, an inspection of the algorithm's logic reveals that the value of the expression `(2 * q * y)` can be computed without multiplication. Instead, it can be obtained from its value in the previous recursion level, using addition.

2. Prime numbers

A prime number is a natural number that is divisible only by 1 and itself. There is an infinite number of prime numbers; some examples are: 2, 3, 5, 7, 11, 13, 17, and 19.

Write a program `Primes` that reads a command-line integer *N*, and prints all the prime numbers smaller than *N*. You may assume *N* is even and larger than 2.

Your program should implement the *Sieve of Sundaram*, which is somewhat more sophisticated than the *Sieve of Eratosthenes* presented in lecture 4-2. The *Sieve of Sundaram* sieves out the *composite* numbers just as the *Sieve of Eratosthenes* does, but even numbers are not considered. The algorithm of the *sieve of Sundaram* is presented below.

1. Start with the integers from 1 to *n,* where $n=(N-2)/2$.
2. Remove all integers of the form $i + j + 2ij$ where:
   a. $1 \leq i \leq j$
   b. $i + j + 2ij \leq n$
3. The remaining numbers are doubled and incremented by one, producing all the odd prime numbers (i.e., all primes except 2) below $2n + 2$.

Think: when iterating over *i* and *j*, what should be the bounds on *i* and *j*?

Here are examples of the program execution (the `main` function is supplied in `Primes.java`):

```
% java Primes 40
The prime numbers up to 40 are:
2 3 5 7 11 13 17 19 23 29 31 37


% java Primes 400
The prime numbers up to 400 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127
131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257
263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397
```

3. Language identification

In many applications of natural language processing (NLP) the first step is to identify the language in question. A simple method for language identification is to compare the frequencies of letters to known distributions for a set of languages. For example, in English the most frequent letters are **etaoi**, in French they are **esait**, and in German they are **ensri**.

Implement a program `LetterFreq` that reads text from the input, computes the frequencies of each letter in the text, and compares these letter frequencies to those of English, French, and German.

For example, for the following text in the supplied **`English.txt`**:

`% more English.txt` // the "more" command can be used to list the contents of a file; use "type" on windows

```
Like all Vogon ships it looked as if it had been not so much designed as con- gealed. The
unpleasant yellow lumps and edifices which protuded from it at unsightly angles would have
disfigured the looks of most ships, but in this case that was sadly impossible. Uglier things
have been spotted in the skies, but not by reliable witnesses. In fact to see anything much
uglier than a Vogon ship you would have to go inside and look at a Vogon. If you are wise,
however, this is precisely what you will avoid doing because the average Vogon will not think
twice before doing something so pointlessly hideous to you that you will wish you had never been
born - or (if you are a clearer minded thinker) that the Vogon had never been born.
```

The program will output scores for three languages; the lower the score, the more likely this is the right language:

```
% java LetterFreq < English.txt
en 0.0028980848443422784
de 0.011142722614205293
fr 0.009481364988177894

% java LetterFreq < French.txt
en 0.008859697462879132
de 0.007577847692666364
fr 0.0011959851118153015
```

For simplicity, we focus only on the letters a-z. Uppercase letters are converted to lowercase, and all other letters (è, ù, etc.), as well as spaces and punctuation (.,:), are disregarded. The frequencies are saved in a `double` array called q. The frequencies are the number of times each letter appeared in the text divided by the total number of letters (without punctuation, etc.). Therefore, the elements of q are between 0 and 1 and the sum of the array is 1. The first element, q_0, is the frequency of the letter **a** in the text. The last element q_{25} is the frequency of the letter **z** in the text.

The score S for each language is computed by comparing its letter frequencies array to q. The language frequencies arrays are supplied in the class **`LetterFreq`**: **`en`** for English, **`fr`** for French, and **`de`** for German. These arrays have the same structure as the text frequencies array. The score is computed by summing over the squared differences between the text letter frequencies and the language letter frequencies. That is, to compare the text frequencies array q to a language frequencies array f, we compute:

$$S(f,q) = \sum_{i\in\{a,\ldots,z\}} (f_i - q_i)^2 = (f_a - q_a)^2 + (f_b - q_b)^2 + \cdots + (f_z - q_z)^2$$

To check that the implementation is correct, you can verify that $S(f,f) = 0$ for any language frequencies array f, which can be either *en* or *fr* or *de*.

Complete the supplied `LetterFreq` class. To test your implementation, run it both with the file **`English.txt`** and with the file **`French.txt`**. You can also try it with your own text.

Submission

Before submitting your solution, inspect your code and make sure that it is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Both documents can be found in the Moodle site, it is your responsibility to read them. Any deviations from these guidelines will result in points penalty. Submit the following files only:

- `BinOps.java`
- `Primes.java`
- `LetterFreq.java`

Deadline: Submit Homework 4 no later than December 22 23:55. You are welcome to submit earlier.