Computer Science 320SC – (2018)

Programming Assignment 1

Due: Saturday, July 28 (11.59pm)

Version 1.01, 2018-07-17: corrected day of week in the due date
Version 1.1, 2018-07-18: adjusted output spec of problem 1 for consistency; fixed two typos
Version 1.11, 2018-07-18: corrected values of pfe(3) and pfe(13) in textual description
Version 1.12, 2018-07-19: adjusted the introduction in Problem 1 to emphasise that the prime factors must be listed in strictly-increasing order

Requirements

This first assignment lets you get familar with submission of correct and efficient implementations of simple (but not quite trivial!) algorithms to the CompSci 320 automated marker. You may implement your algorithms in any language accepted by the automarker e.g. Java, Python 3, Python 2, C or C++.

This assignment is worth 5% of your total course marks. Future programming assignments will require much more work to obtain the same number of marks, so you are encouraged to make a serious attempt on this one!

Problem Statement 1: Prime-Factor Encoding of 3-Digit Integers

The input to your program is single non-negative integer $n < 1000$.

The output to your program is in tabular form, with a single-line header followed by n lines of output, where the k-th line of output indicates how the integer k can be expressed in two different ways as a string.

The first line of output should be the 10-character string `"  k pfe(k)"`, where there are two spaces before the `"k"` and one space after the `"k"`.

The first string on the $(k+1)$-st line of output ($1 \leq k \leq n$) is the value of k expressed as a right-adjusted decimal string $d(k)$ that is exactly three characters long, for example $d(1)$ = `"  1"` and $d(999)$ = `"999"`.

The second string on the k-th line of output is its prime-factor encoding pfe(k). We'll start with some examples of this encoding, rather than a formal definition: pfe(2) = `"2"`, pfe(4) = `"2^2"`, pfe(6) = `"2*3"`, pfe(8) = `"2^3"`, and pfe(12) = `"2^2*3"`. Please note that the exponentiation operator `"^"` takes precedence over the multiplicative operator `"*"`. Also note that the factors are listed in strictly-increasing order, e.g. `"3*3"` is *not* a pf-encoding of any integer, because the factor 3 appears twice. The integer 9 is pf-encoded as `"3^2"`.

Exponents e_i are encoded as decimal strings $d(e_i)$ however the bases b_i are coded somewhat more cleverly – as the decimal encoding $d(j)$ of the *index* j of this prime factor $p_j = b_i$. Accordingly, pfe(1) = `"1"`, pfe(2) = `"2"`, pfe(3) = `"3"`, pfe(5) = `"4"`, pfe(7) = `"5"`, pfe(11) = `"6"`, pfe(13) = `"7"`. Note that I have introduced pfe(1) = `"1"` as a special case in our encoding – because I want to have a non-null representation of the integer 1.

The two strings on the $(k+1)$-st line of output ($1 \leq k \leq n$) should be separated by a single space.

The input should be taken from keyboard/stdin/System.in.

Sample Input:

```
17
```

Your program's precisely-formatted output should be sent to console/stdout/System.out.

Sample Output:

```
 k pfe(k)
 1 1
 2 2
 3 3
 4 2^2
 5 4
 6 2*3
 7 5
 8 2^3
 9 3^2
10 2*4
11 6
12 2^2*3
13 7
14 2*5
15 3*4
16 2^4
17 8
```

As far as I know, pf-encoding has not been studied previously – so you're exploring some novel territory for algorithmics! If you're intrigued to know more about the underlying mathematics, please see Sum of Prime Factors in Wolfram's online MathWorld resource; and if you want to dig even deeper, please see A001414 Integer log of n: sum of primes dividing n (with repetition).

Formally: the pf-encoding of a positive integer k is the concatenation of strings s_i, where each string is a factor of k of the form j_i (if the exponent $e_i = 1$) or of the form j_i^e_i (if the exponent $e_i > 1$). The j_i values are indices into the monotonically-increasing sequence of primes starting from 1. The e_i values are exponents. Both j_i and e_i are encoded as decimal strings. Adjacent strings are separated by asterisk characters `"*"`. The j_i values are monotonically increasing, for example pfe(6) = `"2*3"`, and the string `"3*2"` is not a pf-encoding of any integer.

If you're struggling to understand this problem, please ask for help among your classmates or attend your tutorial session. This is *not* a course in algebraic number theory. I will not be examining you on the (as-yet unexplored) properties of pfe(). Instead I'm trying to give you some experience with a formally-specified problem statement. Without some formalism, it is impossible to specify exactly what an algorithm "should" be doing, making it impossible to prove its correctness. However even in a formal statement there is room for ambiguity or confusion – so please please please ask questions until you know what your algorithm "should" be doing, *before* you try to design it and implement it. And, unless you're truly wizardly, you'll have to debug your implementation before it does what you think it should be doing. When you submit your implementation to the automarker, it will test your implementation on a (secret) value of input – to determine whether or not your implementation runs this particular case correctly.

Problem Statement 2: PF-Compressibility

The input format for this problem is the same as in Problem 1: a single positive integer $n < 1000$ expressed as a decimal string.

The output format for this problem is similar to Problem 1, but possibly with fewer lines of output. Your program should print only the lines for which the length of the pf-encoded representation of an integer k is *shorter* than the length of its representation $d(k)$ as a decimal string.

The input should be taken from keyboard/stdin/System.in.

Sample Input:

```
17
```

Your program's precisely-formatted output should be sent to console/stdout/System.out.

Sample Output:

```
  k pfe(k)
 11 6
 13 7
 17 8
```

Problem Statement 3: Efficient Multiplication

The input format for this problem is a space-separated series of pf-encoded integers $\text{pfe}(v_i)$, where every integer value v_i is of bounded size $v_i < 1000$. The input line may be arbitrarily long. However your implementation may have OS-dependencies which effectively put an upper bound on the length of its input lines, so (to save you the trouble of diagnosing and eliminating any such dependencies) our automarker will test your implementation with an input line of at most 2000 characters.

The output format for this problem is a single line: the pf-encoding of the product $\prod v_i$ of all values v_i in its input.

Your implementation must be efficient: it should run in $O(n)$ time on an n-character input.

The input should be taken from keyboard/stdin/System.in. Note that the spacing of the input may be irregular.

Sample Input:

```
 2^3   17  3*4
```

Your program's precisely-formatted output should be sent to console/stdout/System.out.

Sample Output:

```
2^3*3*4*17
```

Submission

For each of the problems in this assignment, name your source code `gcd.ext`, where `ext` denotes one of { `java`, `cpp`, `py2`, `py3`, `cs` } indicating its language (Java, C++, Python 2, Python 3, mono).

You must use exactly one source file per problem, with your submission for problem j named `probj.ext`.

Two marks are awarded for Problem 1, one mark for Problem 2, and two marks for Problem 3.