Resit mid-exam Imperative Programming
Oct. 14 2016, 14:00-17:00h

- You can solve the problems in any order. Solutions must be submitted to the automated judgment system Justitia. For each problem, Justitia will test ten different inputs, and checks whether the outputs are correct.
- Grading: you get one grade point for free. The remaining nine points are based solely on the judgment given by Justitia. The first problem is worth one grade point. The remaining four problems are worth two grade points each, of which you score the full two points if you passed the complete test set of the problem (i.e. 10 test cases), or one grade point if you passed at least 5 (out of 10) test cases.
- Inefficient programs may be rejected by Justitia. In such cases, the error will be 'time limit exceeded'. The time limit for each problem is one second.
- The number of submissions to Justitia is unlimited. No points are subtracted for multiple submissions.
- There will be no assessment of programming style. However, accepted solutions are checked manually for cheating: for example, precomputed answers will not be accepted, even though Justitia accepts them.
- Note the hints that Justitia gives when your program fails a test.
- Needless to say: you are not allowed to work together. If plagiarism is detected, both parties (supplier of the code and the person that sends in a copy) will be excluded from any further participation in the course. You are not allowed to use mail, phones, tablets, calculators, etc. There is a calculator available on the exam computers (see icon on the desktop). You are allowed to consult the reader, the book, and submissions previously made to Justitia (from the labs or the midexam).
- For each problem, there are three examples of inputs and associated expected outputs. These examples are the first three test cases of a test set. The input files of these examples can be found in Justitia, together with the corresponding output files. These files are called `1.in`, `2.in`, `3.in` and `1.out`, `2.out` and `3.out`. These files can be used to test whether the output of your program matches the requested layout, so that there can be no misunderstanding about the layout and spaces in the output.

Problem 1: 9's complement subtraction

The *9's complement* of a positive integer n is the number that is obtained by replacing each digit d of n by $9 - d$. For example, the 9's complement of 713 is 286 (since 9-7=2, 9-1=8, 9-3=6). Another way of looking at this, is to say that n is subtracted from the number $10^m - 1$, where m is the number of digits of n, so for the given example this would be $(10^3 - 1) - 713 = 999 - 713 = 286$.

If we want to compute $x - y$, where $0 \leq y \leq x$, then we can use the following computational trick. We start by computing the 9's complement of x, and add y to it. Next, we compute the 9's complement of the result, and get the final answer $x - y$. At first sight, this may seem like magic, but it is actually not. Let m be the number of digits of n, then what this process is doing is simply $(10^m - 1) - (((10^m - 1) - x) + y) = x - y$. For example, in the computation of $713 - 125$, we take the 9's complement of 713 (which is 999-713=286) and add 125 to it, yielding 411. The 9's complement of 411 is 999-411=588, which is indeed 713-125.

The input of this problem consists of two integers `a`, and `b` such that $0 \leq$ `b` $\leq$ `a` $< 1000000 = 10^6$. The output should be the calculation of `a-b` using 9's complement addition. Numbers should not be printed with superfluous leading zeroes (i.e. $999 - 992 = 7$, and not 007). Make sure that the output of your program is in compliance with the following examples. The output should not contain any spaces, and must end with a newline (`\n`):

Example 1:

input:
```
713 125
```
output:
```
713-125=999-((999-713)+125)=999-(286+125)=999-411=588
```

Example 2:

input:
```
873 218
```
output:
```
873-218=999-((999-873)+218)=999-(126+218)=999-344=655
```

Example 3:

input:
```
17 12
```
output:
```
17-12=99-((99-17)+12)=99-(82+12)=99-94=5
```

Problem 2: Sorting numbers

The input for this exercise consists of a (possibly very long) series of positive integers less than 100. The series is terminated by the number 0. Write a program that reads the input, and outputs the input numbers in decreasing order. The output must be a single line, with numbers separated by a comma. The line should not contain any spaces, and it should end with a newline (`\n`):

Example 1:

input:
```
1 2 3 4 5 0
```
output:
```
5,4,3,2,1
```

Example 2:

input:
```
1 3 5 2 1 2 42 6 7 8 3 42 84 0
```
output:
```
84,42,42,8,7,6,5,3,3,2,2,1,1
```

Example 3:

input:
```
5 5 5 0
```
output:
```
5,5,5
```

Problem 3: Mother of all primes

A *prime number* is a natural number greater than 1 that has no positive divisors other than 1 and itself.
We say that a number n is a *k-descendant* of a prime number p if and only if n is a prime number and $n = k \times p + k - 1$.

For example, the prime number 3 has a 2-descendant (being $7 = 2 \times 3 + 2 - 1$), and a 3-descendant (being $11 = 3 \times 3 + 3 - 1$). However, 3 has no 4-descendant because $4 \times 3 + 4 - 1 = 15$, and 15 is not a prime number. Note that each prime is its own 1-descendant.

Of all prime numbers below one million, the prime number 154769 is quite special, since all numbers $k \times 154769 + k - 1$ are prime for $k = 1, 2, 3, 4, 5, 6$. No other number below one million has more than 6 *consecutive* k-descendants starting from $k = 1$.

The input of this problem consists of two integers `a` and `b`, where $1 \leq$ `a` $\leq$ `b` $\leq 1000000 = 10^6$. The output should be the prime `p`, where `a` $\leq$ `p` $\leq$ `b`, with the largest number of consecutive k-descendants starting from $k = 1$, and the corresponding number of descendants. If this prime number is not unique (i.e. several solutions exist), then print the smallest one. On the other hand, if no prime number exists in the given interval, then your program should output the text `NONE`. Note that there are no extra spaces in the output, and that the output line must end with a newline (`\n`):

Example 1:
input:
```
1 1000000
```
output:
```
154769 6
```

Example 2:
input:
```
100 200
```
output:
```
173 3
```

Example 3:
input:
```
1000 1005
```
output:
```
NONE
```

Problem 4: Palindromic substrings

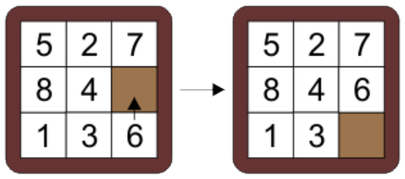
In this problem, a string is considered a *substring* of another string if it occurs in the other string ignoring spaces. For example, `"RUMOR"` is a substring of `"RUM OR WHISKY"`.
Recall that a *palindrome* is a sequence of characters which reads the same backward or forward, such as *radar* or *kayak*.

Write a program that reads a sentence from the input that is terminated by a dot '.'. The sentence consists solely of uppercase letters and spaces, and will not exceed 100 characters (including the dot). Your program must output the longest palindromic substring of the input sentence. In case a sentence has more than one longest palindromic substring (see example 2), then the one that occurs earliest in the sentence should be printed. Note that there are no spaces in the output, and that the output line must end with a newline (`\n`):

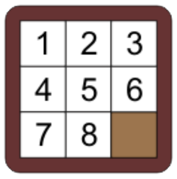
Example 1:
input:
```
BOB AND ANNA HAVE PALINDROMIC NAMES.
```
output:
```
ANNA
```
Example 2:
input:
```
RADARS DO NOT DETECT KAYAKS.
```
output:
```
RADAR
```
Example 3:
input:
```
NEVER ODD OR EVEN.
```
output:
```
NEVERODDOREVEN
```

Problem 5: Sliding n-puzzle

A *sliding n-puzzle* is a number puzzle, that is played on an $n \times n$ grid containing $n^2 - 1$ tiles, and an empty space. The tiles are numbered 1, 2, .., n^2-1. Tiles can be moved using the moves `UP`, `DOWN`, `LEFT`, `RIGHT`. For example, in the following figure, the move made is `UP 6`, meaning that the tile with the number 6 was moved up (i.e, the empty space went down).



The object of the puzzle is to place the tiles in ascending order by making a series of sliding moves. A possible solution is given in the following figure. Note that the location of the empty space may be anywhere in a solution, so the empty space need not be in the bottom right position (as in the figure).



Write a program that reads from the input an integer `n`, where $2 \le$ **`n`** ≤ 8. The next `n` lines contain the rows of an **`n`** $\times$ **`n`** grid of numbers. The grid is the initial configuration of an `n`-puzzle. The empty space is encoded as the number 0. The rest of the input consists of a series of moves, terminated using the word END.

Your program should output `SOLVED` if the series of moves (starting from the initial configuration) solves the puzzle. If the puzzle is not solved, your program should output `UNSOLVED`. The program should output `INVALID` in case an invalid move occurs in the move series. Note that there are no spaces in the output, and that the output ends with a newline (`\n`):

Example 1:

input:
```
3
1 3 0
4 2 5
6 7 8
RIGHT 3
UP 2
END
```
output:
```
SOLVED
```

Example 2:

input:
```
3
1 0 3
4 2 5
6 7 8
LEFT 3
RIGHT 3
END
```
output:
```
UNSOLVED
```

Example 3:

input:
```
2
3 1
0 2
LEFT 2
DOWN 1
RIGHT 3
RIGHT 2
END
```
output:
```
INVALID
```