1/3rd term Programming Fundamentals Oct. 6 2023, 18:30-21:30h

- You can solve the problems in any order. Solutions must be submitted to the automated judgement system Themis. For each problem, Themis will test ten different inputs, and check 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 Themis. The first problem is worth one grade point. The remaining four problems are worth two grade points each.
- You can score partial points if a program passes several but not all test cases. For example, if you pass 6 out of 10 test cases, then you are awarded 60% of the grade points for that exercise.
- Inefficient programs may be rejected by Themis. In such cases, the error will be 'time limit exceeded'. The time limit for each problem is one second..
- The number of submissions to Themis is not limited. 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 Themis accepts them.
- 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 copied code) will be excluded from any further participation in the course.
- You are not allowed to use email, phones, tablets, calculators, etc. There is a calculator available on the exam computers (see icon on the desktop). You are allowed to consult the ANSI C book and a dictionary. You are not allowed to use a printed copy of the reader or the lecture slides, however they are available digitally (as a pdf) in Themis. You are allowed to access your own submissions previously made to Themis.
- For each problem, the first three test cases (input files) are available on Themis. These input files, and the corresponding output files, are called `1.in` , `2.in`, `3.in`, `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.
- **If you fail to pass a problem for a specific test case, then you are advised not to lose much time on debugging your program, and continue with another prob lem. In the last hour of the exam, all test cases will be made visible in Themis.**

Problem 1 is worth 1 grade point. The other exercises are worth 2 grade points (totaling 9 points, you get 1 point for free).

Problem 1: Cigarette butts

A poor homeless man can make exactly one cigarette out of four cigarette butts. One day he finds 31 butts. How many cigarettes can he smoke that day?

Well, from 31 butts he can make 7 cigarettes (using $7 \times 4 = 28$ butts, leaving 3 butts). So, after having smoked these 7 cigarettes he has 7 new butts, totalling $7 + 3 = 10$ butts, which is enough to make another 2 cigarettes (leaving $10 - 2 \times 4 = 2$ butts). After having smoked these two cigarettes, he again has two new butts. Together with the remaining 2 butts, he can make another (last) cigarette. So, in total he can smoke $7 + 2 + 1 = 10$ cigarettes.

Write a program that reads from the input a positive integer n, which is the number of butts that the man finds on some day. The output should be the number of cigarettes that he can smoke that day. You may assume that $0 \leq n < 1000000$.

Example 1:

input:
```
3
```
output:
```
0
```

Example 2:

input:
```
4
```
output:
```
1
```

Example 3:

input:
```
31
```
output:
```
10
```

Problem 2: Right-truncatable primes

A *right-truncatable prime* is a prime which remains prime when the least significant digits are successively removed. For example, 7393 is a right-truncatable prime, because 7393, 739, 73, and 7 are all prime.

The input for this problem is an integer n, where $0 \leq n \leq 10^9$. The output must be `YES` if n is a right-truncatable prime, and `NO` otherwise.

Example 1:

input:
```
7393
```
output:
```
YES
```

Example 2:

input:
```
42
```
output:
```
NO
```

Example 3:

input:
```
3137
```
output:
```
YES
```

Problem 3: Running in circles

Consider the following process. Given two positive moduli `modA` and `modB`. We start with two variables `a` and `b` which are both initially set to zero. Next we keep iterating the assignments `a=(a+1)%modA; b=(b+1)%modB` until we reach the initial position `a==0` and `b==0` again. For example, with `modA=3` and `modB=2` we get the following sequence of steps:
$(\texttt{a},\texttt{b}):\ (0,0) \to (1,1) \to (2,0) \to (0,1) \to (1,0) \to (2,1) \to (0,0)$
As you can see, starting from $(0,0)$ it takes 6 steps to reach $(0,0)$ again. We call 6 the *cycle length* for the moduli 3 and 2. Of course, we can extend this process for multiple moduli. For example, if you try the above process for three variables `a`, `b`, `c`, and `modA=2`, `modB=3`, and `modC=5` then you will find that the cycle length for the moduli 2, 3, and 5 is 30.

Write a program that reads from the input a series of positive moduli. The series is terminated by a zero (so, zero itself is not a modulus). You may assume that there are at most ten moduli on the input. The output must be the cycle length of the input moduli. You may assume that the cycle length fits in a standard `int`.

Example 1:
input:
```
3 2 0
```
output:
```
6
```

Example 2:
input:
```
5 7 0
```
output:
```
35
```

Example 3:
input:
```
2 3 5 0
```
output:
```
30
```

Problem 4: DNA matching

The input for this problem consists of two lines. The first line contains four non-negative integers: `nA`, `nC`, `nT`, and `nG`. The second line contains a DNA string containing only the characters '`A`', '`C`', '`T`', and '`G`'. You may assume that this string is no longer than 100000 characters.

The output must be the starting indices of all *consecutive substrings* that contain exacly `nA` times the character '`A`', `nC` times the character '`C`', `nT` times the character '`T`', and `nG` times the character '`G`'. A consecutive substring is a sequence of characters that appear in adjacent positions of the DNA string.

For example, for `nA=nT=nG=1`, `nC=2`, and the DNA string `"GGTACCGACTGTAT"` we see that it has three substrings that match the criteria:

G**GTACC**GACTGTAT GG**TACCG**ACTGTAT GGTAC**CGACT**GTAT

The starting indices of these substring in the DNA string are 1, 2, and 5. Note that the index of the first character of the DNA string is 0 (which is a standard C convention).

The output of your program should be `NO MATCH` if no matching substrings can be found.

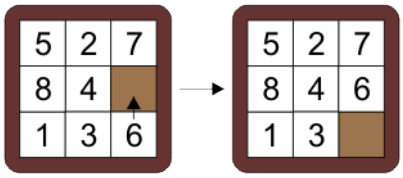
Example 1:
input:
```
1 1 1 1
TACTGTAT
```
output:
```
1
```

Example 2:
input:
```
1 2 1 1
GGTACCGACTGTAT
```
output:
```
1,2,5
```

Example 3:
input:
```
1 1 1 1
AACCTTGG
```
output:
```
NO MATCH
```

Problem 5: Sliding 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 grid cell. The tiles are number 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 $2 \leq$ `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
```