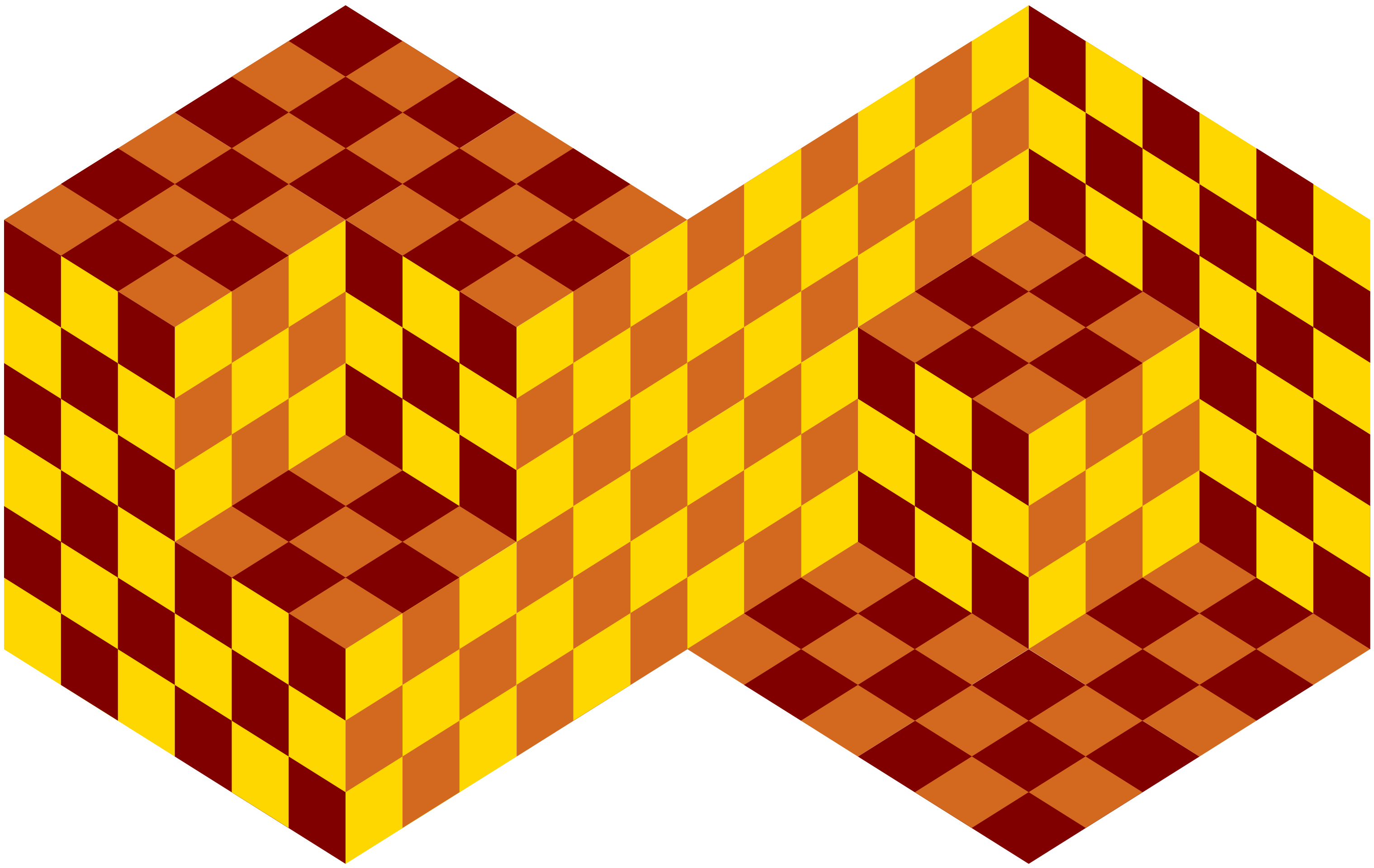
22nd Annual High School Programming Contest

April 11, 2017



Department of Mathematical and Digital Sciences
Bloomsburg University

1. Speeding Ticket

If you are ticketed for speeding in Cobblestone County, you must pay a fine calculated in terms of a base fee. The calculation has three parts:

- Pay the base fee for each MPH over the limit up to 5 MPH.
- Pay twice the base fee for each MPH over the limit from 6 up to 15 MPH.
- Pay quadruple the base fee for each MPH over the limit beyond 15.

For example, suppose the base fee is $10 and you are caught speeding 23 MPH over the limit. The fine is calculated as follows:

- 5 MPH * 10 = 50.
- 10 MPH * 20 = 200.
- 8 MPH * 40 = 320.

You must therefore pay a total of $570.

Write a program that prompts the user for a speed in MPH over the limit and the base fee, and outputs the cost of the ticket.

The following sample executions illustrate the required I/O format. User input is shown in bold.

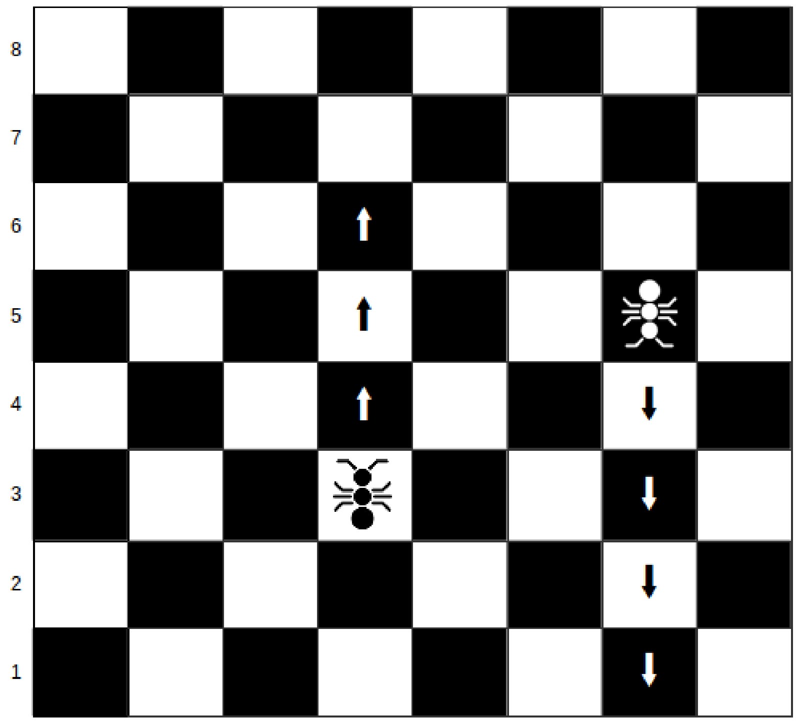
```
Speed in MPH over the limit: 3
Base fee: 8
Cost of ticket: $24


Speed in MPH over the limit: 9
Base fee: 6
Cost of ticket: $78


Speed in MPH over the limit: 28
Base fee: 30
Cost of ticket: $2310
```

The results shown above were calculated as follows:

- 24 = 3 MPH * 8
- 78 = (5 MPH * 6) + (4 MPH * 12)
- 2310 = (5 MPH * 30) + (10 MPH * 60) + (13 MPH * 120)

2. Ant Chess

Two ants are placed on a chessboard with rows numbered 1 to 8 as shown below. One of the ants faces up and the other faces down. They take turns moving, each advancing one square in the direction it faces. The winner is the first ant to reach the top or bottom row.



In the scenario portrayed in the picture, the UP ant starts in row 3 and the DOWN ant starts in row 5. The DOWN ant moves first. They progress as shown by the arrows, with the DOWN ant winning the game by reaching row 1 when the UP ant is still at row 6. In this particular case, the outcome would have been the same even if the UP ant had moved first.

Write a program that prompts the user for a starting configuration expressed as a string of three characters indicating the UP ant's starting row, the DOWN ant's starting row, and who gets to move first ('u' or 'd'). For example, "35d" describes the configuration shown in the picture. The program outputs the winner and the number of moves made by the winning ant

Each ant can begin in any row between 2 and 7 inclusive. You may assume that they are in different columns, so they never block each other.

The following execution snapshots illustrate the required I/O format.

```
Starting configuration: **35d**
The DOWN ant wins after 4 move(s).
```

```
Starting configuration: **37u**
The UP ant wins after 5 move(s).
```

3. Legs

I had a dinner party at my house last week. Five people were there, including me, and as we started to eat it occurred to me that there were 84 legs in the room. I was including spiders and cockroaches. Here is the calculation, exactly as I wrote it on a napkin:

```
     5 people * 2 legs each = 10 legs
    4 spiders * 8 legs each = 32 legs
7 cockroaches * 6 legs each = 42 legs
                              -------
                              84 legs
```

This fact amused people, and someone wondered how many different ways there are for some number of people, spiders, and cockroaches to have a combined total of 84 legs. It turns out that there are 88 possibilities. For example, 7 people, 8 spiders, and 1 cockroach have 84 legs. Another possibility would be 18 people, 6 spiders, and no cockroaches.

For the case of 20 legs, here are all the possibilities:

people	spiders	cockroaches
0	1	2
1	0	3
2	2	0
3	1	1
4	0	2
6	1	0
7	0	1
10	0	0

Write a program that prompts the user for a number of legs and calculates the corresponding number of possible combinations of people, spiders, and cockroaches.

Execution snapshots:

```
Number of legs: 12
4 possible combinations of people, spiders, and cockroaches.

Number of legs: 20
8 possible combinations of people, spiders, and cockroaches.

Number of legs: 720
5521 possible combinations of people, spiders, and cockroaches.
```

4. Counters

Imagine a device consisting of a row of three buttons, each with an integer counter. The counters are initially set to zero. Pressing a button causes its counter to increase by one, and at the same time each of the other two counters decreases by one.

Here is an illustration:

After pressing the first button:

After pressing the third button:

After pressing the third button again:

Write a program that prompts the user to enter the final counter values and outputs the number of times that each button was pressed in order to reach those values. You may assume that the user enters a valid set of counter values (i.e., a solution exists).

Execution snapshots (the first one illustrates the scenario described above):

```
Final counter values: -1 -3 1
Counter hits: 1 0 2

Final counter values: -6 -4 -2
Counter hits: 3 4 5

Final counter values: 1 -5 -3
Counter hits: 4 1 2

Final counter values: -5 -3 1
Counter hits: 1 2 4
```

5. Bounded Weight Sequences

We define the weight of a positive integer to be the number of its digits. Thus weight(43) = 2 and weight(101763) = 6. The weight of a sequence of numbers is simply the sum of the weights of the individual numbers in the sequence. For example, the weight of (8, 9, 10, 11) is 6.

For this problem, we will be interested in sequences of consecutive positive integers starting at *k* and of weight not exceeding *n*, where *k* and *n* are chosen by the user.

To illustrate, suppose *k* = 7 and *n* = 10.

Integer	Weight
7	1
8	1
9	1
10	2
11	2
12	2

The weight of the sequence (7, 9, 9, 10, 11, 12) is 9, and the sequence cannot be extended to 13 without exceeding the weight bound.

Write a program that prompts the user for a starting integer and a weight bound, and outputs the longest possible sequence of consecutive integers starting at the specified number and not exceeding the weight bound.

The required I/O format is illustrated by the following execution snapshots:

```
Starting integer: 7
Weight bound: 10
The sequence (7-12) has weight 9.

Starting integer: 23
Weight bound: 100
The sequence (23-72) has weight 100.

Starting integer: 99
Weight bound: 99999
The sequence (99-22258) has weight 99997.
```

6. SubSub

Write a program that prompts the user to enter a string and then outputs the longest substring that appears twice without any intervening characters. If there is more than one such substring of maximal length, output the one that occurs first when reading from left to right. If there are no such substrings, output NO SOLUTION.

Execution snapshots:

```
 Input: ABCCBA
Output: C

 Input: ABCBCBA
Output: BC

 Input: ABCBCCCCD
Output: BC

 Input: SAX+OZX+OZAX
Output: X+OZ

 Input: 012120452045045
Output: 2045

 Input: ABACBDAD
Output: NO SOLUTION
```

7. Zeros Downstream

Let *g*(*n*) denote the product of the positive integers from 1 to *n*:

g(1) = 1
g(2) = 1 * 2 = 2
g(3) = 1 * 2 * 3 = 6
g(4) = 1 * 2 * 3 * 4 = 24
g(5) = 1 * 2 * 3 * 4 * 5 = 120
g(6) = 1 * 2 * 3 * 4 * 5 * 6 = 720

and so on. Let *h*(*n*) denote the number of zeros after the last non-zero digit of *g*(*n*). To illustrate, here is a table showing *g*(*n*) and *h*(*n*) for selected values of *n*:

n	*g*(*n*)	*h*(*n*)
4	24	0
5	120	1
10	3,628,800	2
15	1,307,674,368,000	3
20	2,432,902,008,176,640,000	4
25	15,511,210,043,330,985,984,000,000	6
30	265,252,859,812,191,058,636,308,480,000,000	7
40	815,915,283,247,897,734,345,611,269,596,115,894,272,000,000,000	9

Write a program that prompts the user for a positive integer *n* and outputs *h*(*n*).

Execution snapshots:

```
Enter a positive integer: 10
h(n) = 2

Enter a positive integer: 30
h(n) = 7

Enter a positive integer: 123456789
h(n) = 30864192
```

You may assume *n* ≤ 1,000,000,000.

8. Uninteresting Numbers

If you are like me, you find numbers having three consecutive identical digits to be completely uninteresting. Let's avoid them. Write a program that prompts the user to enter a positive integer *k* and outputs the smallest integer greater than *k* that does not have three consecutive identical digits. You may assume $k \leq 10^{20}$.

Execution snapshots:

```
 Input: 1234555
Output: 1234556

 Input: 999
Output: 1001

 Input: 12344440
Output: 12344500

 Input: 12233300000
Output: 12233400100

 Input: 1919666619191919
Output: 1919667001001001

 Input: 1111111111111111111
Output: 1120010010010010010
```

9. Cheap Integers

A positive integer can be represented in various ways as a combination of sums and products, which we will call a simple representation of the integer. Here are a few simple representations of the integer 21:

- 21 = 10 + 11.
- 21 = 4 * 4 + 5.
- 21 = (1 + 2 * 3) * 3.
- 21 = 1 + 4 * 5.

Let's define the cost of a simple representation to be the sum of the numbers that appear in it. For example, the simple representations shown above have the following costs:

Simple Representation	Cost
10 + 11	21
4 * 4 + 5	13
(1 + 2 * 3) * 3	9
1 + 4 * 5	10

It turns out that among all simple representations of 21, the lowest possible cost is nine. We will define the cost of a positive integer to be the minimal cost of a simple representation of it. So, for example, the cost of 21 is nine.

Write a program that prompts the user for a positive integer and outputs the corresponding cost.

Execution snapshots:

```
Enter a positive integer: 21
Cost: 9

Enter a positive integer: 50
Cost: 12

Enter a positive integer: 67
Cost: 14

Enter a positive integer: 888
Cost: 20
```

You may assume that the input will be a positive integer at most 9999.

10. Antisocial Seating

There are *N* empty chairs in a row. *N* people enter the room one at a time and each person must choose an unoccupied chair and sit there. These people wish to avoid sitting next to other people if possible. If that is not possible, the next best option is to sit next to only one person. And if *that* is not possible, a person will have no choice but to sit next to two other people.

Suppose, for example, *N* = 4. Here is one way in which the chairs might be filled: ACDB. This notation means that the first person (A) enters and sits in the first chair, then the second person (B) enters and sits in the fourth chair, and so on. In this scenario, here is how the seating arrangment would look after each person enters the room and sits:

```
A . . .
A . . B
A C . B
A C D B
```

Note that the first person (A) will not necessarily choose to sit in the first or last chair, even though any other choice will result in being next to two other people *eventually*. The decision about where to sit is based only on the *current* seating arrangement. For example, the chairs might fill up like this:

```
. . A .
B . A .
B . A C
B D A C
```

Let *S*(*N*) denote the set of all possible final seating arrangements. For example:

S(4) = {ACDB, ADBC, ADCB, BCDA, BDAC, BDCA, CADB, CBDA}.

S(5) = {ACEBD, ADBEC, ADCEB, ADEBC, AEBDC, AECBD, AECDB, AEDBC, BCEAD, BDAEC, BDCEA, BDEAC, BEADC, BECAD, BECDA, BEDAC, CADEB, CAEBD, CAEDB, CBDEA, CBEAD, CBEDA, CDAEB, CDBEA, CEADB, CEBDA, DACEB, DAEBC, DAECB, DBCEA, DBEAC, DBECA}.

We can number the arrangements according to their alphabetical ordering. In *S*(4), ACDB is the first possible arrangement, ADBC is the second, and so on.

Let *T*(*N*, *K*) denote the *K*th arrangement in S(N). For example, *T*(4, 2) = ADBC.

Write a program that prompts the user for the number of chairs (*N*) and an index (*K*), and outputs *T(N*, *K*). You may assume that *N* will be between 3 and 10 inclusive.

See the next page for execution snapshots.

Number of chairs: **3**
Index: **2**
Arrangement of index 2: BAC

Number of chairs: **4**
Index: **2**
Arrangement of index 2: ADBC

Number of chairs: **6**
Index: **70**
Arrangement of index 70: DAEBFC

Number of chairs: **8**
Index: **999**
Arrangement of index 999: CHDEGAFB