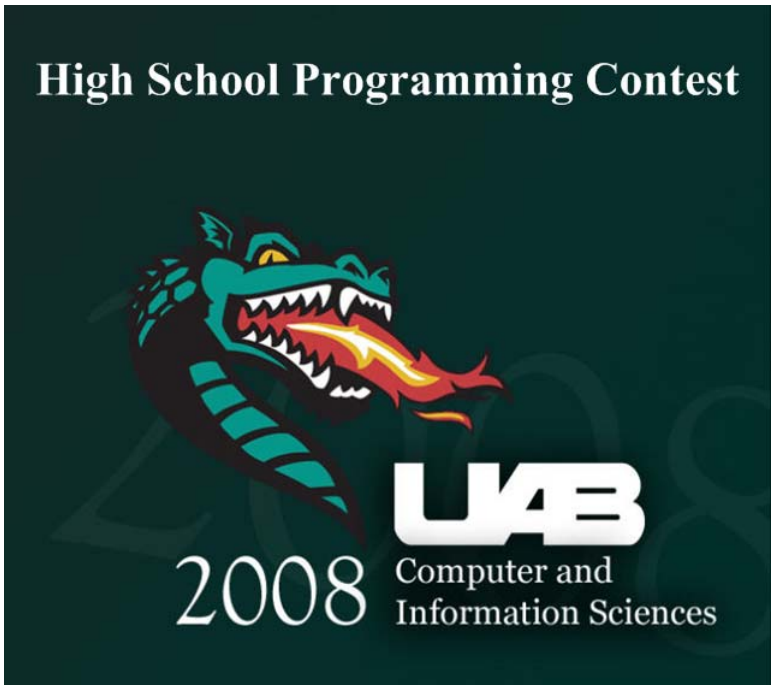

High School Programming Contest

May 10, 2008

Each problem in this packet contains a brief description, followed by two example executions of a successful implementation.

The example input and output shown for each question should be regarded only as a suggestion. We will test your programs on the examples provided, as well as two other sample input test cases.

If your program fails, the result returned by the submission system will state the counterexample test case that caused your program to be judged incorrect. Each incorrect answer will be assessed a 20 minute penalty. As noted in the rules, the overall time will be considered as a tie breaker (with total number of problems solved serving as the initial ranking metric).

Please pay close attention to the directions in each problem description. In some cases, assumptions are stated about the limitations of the input, which are designed so that you do not have to consider difficult cases or perform input validation.

Problem 1:
Simple Averages

Create a program that allows a user to enter 1 to 10 integer numbers on a single line of input (you may assume that the input contains no more than 10 numbers and no less than 1 number).

Compute both the average and median of the list and display the result.

Example 1:

```
Input list: 1 10 13
Average: 8
Median: 10
```

Example 2:

```
Input list: 1 2 3 4 5 6 7 8 9 10
Average: 5.5
Median: 5.5
```

Problem 2:
Word Shifter

Create a program that receives as input a line of text with 1 <= n <= 5 words (you may assume a "word" only contains lower-case letters). Your program should output all of the circular shifts of the n words, one per line. That is, an input with n words will output n separate lines, each with a new circular shift where the first word of the previous line is moved to the end of the next line, until the original list appears again.

Example 1:

```
Input string: ee ii ee ii oo
Output shift:
ee ii ee ii oo
ii ee ii oo ee
ee ii oo ee ii
ii oo ee ii ee
oo ee ii ee ii
```

Example 2:

```
Input string:  aa bb cc dd
Output shift:
aa bb cc dd
bb cc dd aa
cc dd aa bb
dd aa bb cc
```

Problem 3:
Credit Card Parity Checker

Portions of the following description are copied from Wikipedia.

The Luhn algorithm, also known as the "modulus 10" or "mod 10" algorithm, is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers. The formula verifies a number against its included check digit, which is usually appended to a partial account number to generate the full account number. This account number must pass the following test:

1. Counting from rightmost digit (which is the check digit) and moving left, double the value of every even-positioned digit. For any digits that thus become 10 or more, take the two numbers and add them together. For example, 1111 becomes 2121, while 8763 becomes 7733 (from 2×6=12 → 1+2=3 and 2×8=16 → 1+6=7).
2. Add all these digits together. For example, if 1111 becomes 2121, then 2+1+2+1 is 6; and 8763 becomes 7733, so 7+7+3+3 is 20.
3. If the total ends in 0 (put another way, if the total modulus 10 is congruent to 0), then the number is valid according to the Luhn formula; else it is not valid. So, 1111 is not valid (as shown above, it comes out to 6), while 8763 is valid (as shown above, it comes out to 20).

Create a program that takes as input a single number sequence (each sequence has at least 2 numbers and no more than 10 numbers). Determine if the number is correct according to the Luhn algorithm.

Example 1:

```
Input number: 446667652
Output: Number is not correct
```

Example 2:

```
Input number: 446667651
Output: Number is correct
```

Problem 4:
Password Cracker

Create a program that prints a particular password combination created from a set of 11 characters. The set of 11 possible characters is as follows:

a b c d 0 1 2 3 " ! {

The program should take as input one integer number per line. For each input number, the program must display the password combination generated at that iteration.

The order of generated passwords must adhere to the following rules:

1. Passwords must be generated in the order in which the above characters are displayed.
2. All passwords of a given length must be generated before moving on to the next length.

The following in an example of correct password generation and demonstrates the relationship between the iteration number and the expected password:

Password 1: a
Password 2: b
Password 3: c
...
Password 11: {
Password 12: aa
Password 13: ab
...
Password 8784: 1111

Example 1:

```
Password iteration: 20
Expected password: a"
```

Example 2:

```
Password iteration: 6789
Expected password: 0aab
```

Problem 5:
Pascal's Triangle

In mathematics, Pascal's triangle is a geometric arrangement of the binomial coefficients in a triangle. A simple construction of the triangle proceeds in the following manner. On the zeroth row, write only the number 1. Then, to construct the elements of following rows, add the number directly above and to the left with the number directly above and to the right to find the new value. If either the number to the right or left is not present, substitute a zero in its place. For example, the first number in the first row is 0 + 1 = 1, whereas the numbers 1 and 3 in the third row are added to produce the number 4 in the fourth row.

Create a program that will allow the user to enter between 1 and 50. This number represents the number of rows in Pascal's triangle to print to the screen. Your program should approximate some level of horizontal spacing for each row, as shown in the output below.

Example 1:

```
Number of rows: 5
Output:

            1
         1     1
      1     2     1
   1     3     3     1
1     4     6     4     1
```

Example 2:

```
Number of rows: 7
Output:

                  1
               1     1
            1     2     1
         1     3     3     1
      1     4     6     4     1
   1     5    10    10     5     1
1     6    15    20    15     6     1
```

Problem 6:
Airline Seat Packer

Imagine you work for an airline company that wants to figure out how to fly only with full planes while optimizing their profit. The airline fleet consists of two types of aircraft: *Type A* aircraft cost $cost_A$ dollars to operate per flight and can carry $passengers_A$ passengers. *Type B* aircraft cost $cost_B$ dollars to operate per flight and can carry $passengers_B$ passengers. Create a program that *fills each aircraft to capacity* and also *minimizes the total cost of operations* for that route.

Your program should request the number of total passengers for the route. The program should also ask the user for the values of $cost_A$ and $passengers_A$, as well as $cost_B$ and $passengers_B$ (please see the example below). The output should state the specific combination of *Type A* and *Type B* flights that should be made that will *ensure all flights are full while also optimizing the cost of full flights*. The total cost should also be stated in the output. In some cases, it may not be possible to offer a solution (that is, there is no combination that will allow all planes to be full), which should be indicated in the output if that is the case.

Example 1:

```
How many passengers on this route? 550
Enter cost and occupancy of Type A: 1 13
Enter cost and occupancy of Type B: 2 29
This route should use 20 Type A and 10 Type B at cost 40.
```

Example 2:

```
How many passengers on this route? 599
Enter cost and occupancy of Type A: 11 20
Enter cost and occupancy of Type B: 22 40
This route cannot be flown at full capacity.
```

This page intentionally left blank for scratch paper.