Exercises of programming v. 0.1

Andrea Marin

February 1, 2013

1 Exercises of cycles

This section contains some exercises that aim at improving the skills of the students on the usage of cycles. The exercises must be solved without using any C library but the `stdio.h` and adopting the cycle which is mostly suitable to the solution proposed. In order to achieve this, keep in mind what follows:

- You use the `for` cycle when, once the values of the variables are know, you can predict the number of iterations just by reading the heading of the cycle;
- You use the `do .. while` cycle when the `for` is inappropriate and you are sure that the cycle's block must be executed at least once;
- You use the `while` cycle in all the other cases.

Do not use the *jumping* instructions as *break*, *continue* and *return* to abort a cycle.

1.1 Co-prime numbers

Write a C program that reads two integers from the standard input and decides if they are coprime. The program must write the answer on the standard output. Two numbers are coprime if their only common positive divisor is 1.

Solution. First let us understand the meaning of coprimality. 5 and 12 are coprime because they do not share any divisor but 1 and -1; indeed the same holds for -5 and 12. If we consider 21 and 27 they are not coprime because they share 3 as a common divisor. Notice that 1 is coprime with every number (included 0), but 0 is coprime only with 1.

The easiest solution consists in using the *universal property* pattern. Indeed, if the two input numbers (say n_1 and n_2) are positive, we can reformulate the coprime condition as: *all the numbers in the interval* $[2, \min(n_1, n_2)]$ *are not simultaneously divisors of* n_1 *and* n_2. However, we must consider the case of negative numbers which is easy because we can just work with the absolute values. Finally, the case of $n_1 = 0$ or $n_2 = 0$ can be treated separately; however observe that if we consider the divisors in the interval $[2, \max(n_1, n_2)]$, then we immediately solve the problem when one of the two numbers is 0 and the other is not. So the case $n_1 = 0$ and $n_2 = 0$ must be considered separately anyway. The solution is shown in Table 1. A more efficient solution relies on the Euclide's

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n1, n2;    /*the two input numbers*/
   int coprime;   /*Are they coprime?*/
   int div;       /*Candidate common divisor*/


   printf("Give the first number: ");
   scanf("%d", &n1);
   printf("Give the second number: ");
   scanf("%d", &n2);
   /* compute the absolute value */
   if (n1 < 0)
      n1 = -n1;
   if (n2 < 0)
      n2 = -n2;

   /*Universal property: first assume it is true!*/
   coprime = 1;

   /*Start from divisor  2*/
   div = 2;
   while ( (div <= n1 || div <= n2) && coprime) {
      if(n1 % div == 0 && n2 % div == 0)
         coprime = 0;    /*Counterexample has been found!*/
      div = div + 1;
   }

   /*case of both number equal to zero*/
   if (n1 == 0 && n2 == 0)
      coprime = 0;

   /*output*/
   if (coprime)
      printf("The numbers are coprime.\n");
   else
      printf("The numbers are not coprime.\n");

   return 0;
}
```

Table 1: Decision of coprimality between two integers based on testing a universal property.

algorithm for the computation of the largest common divisor. Also in this case pay attention to the case when one or both the input numbers are 0.

1.2 Exercise on sequence, give the sum and the quantity of numbers

Write a C program that reads from the standard input a sequence of numbers; the program must stop reading from the input sequence when it finds two equal consecutive numbers. Compute the sum and the quantity of all the read numbers.

Solution. The easiest way to address the problem consists in using the *sequence pattern*, i.e., we use variable *prec* to store the last read number and variable *cur* to store the current input. At each iteration of the cycle the value of *cur* is stored in *prec* and *cur* is newly read. The other patterns to apply is that of the *accumulator* to store the partial sum of the input sequence and of the *counter* to store the partial counting of the read numbers. Notice that we must do at least two read operations from the standard input, therefore, we do one reading before the cycle block and then we choose the `do .. while` cycle to ensure a second reading.

1.3 Printing the odd numbers in $[1, 1000]$

Write on standard output all the odd numbers included in the interval [1, 1000] with a newline every 10 printed numbers.

Solution. The exercise is very simple, the key-point here is counting the number of written numbers and adding a new line printing every ten. The solution is shown in Table 3. Try to rewrite the solution avoiding the use of variable *printed*.

1.4 Compute the integer $r-$th root of a positive number

Write a C program that reads from standard input two positive integers, n and r, and compute the floor approximation of the $r-$root of n:

$$x = \lfloor \sqrt[r]{n} \rfloor \qquad r, n \in \mathbb{N}^+$$

Solution. The problem can be a little difficult because we do not want to use the C Mathematics library. Let us imagine we are able to compute in some way x^r. In this case the exercise can be reformulated as follows: *find the smallest x such that $x^r > n$*, i.e., we can see the solution as an application of the *find the edge* pattern: so we start from $x = 1$ and we try all the successive values until we find the first for which $x^r > n$. Then the solution is $x - 1$. However computing x^r is easy because it is sufficient to multiply x by itself r times. The solution is depicted by Table 4, observe carefully where the initialisation of the variables are put.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int sum, count; /*accumulator and counter*/
   int cur, prev; /*current and previous values of the seqeunce*/

   scanf("%d", &cur);
   sum = cur;
   count = 1;
   do {
      prev = cur;
      scanf("%d", &cur);
      count = count + 1; /*counter*/
      sum = sum + cur; /*accumulator*/
   } while (prev != cur);
   printf("Read numbers: %d\nSum: %d\n", count, sum);

   return 0;
}
```

Table 2: Read a sequence of integers and stop when two consecutive numbers are equal. Compute the sum and count the quantity or read input.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
        int i; /*odd number to be printed*/
        int printed; /*counting the printed numbers*/

        printed = 0;
        for (i = 1; i < 1000; i = i + 2) {
                printf("%d ", i);
                printed = printed + 1;

                if (printed % 10 == 0)
                        printf("\n");
        }
        return 0;
}
```

Table 3: Printing the odd numbers in $[1, 1000]$; new line every 10.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {
   int n, r;  /*input variables*/
   int root, pow;
   int i;

   printf("Write n: ");
   scanf("%d", &n);
   printf("Write r: ");
   scanf("%d", &r);

   if (n > 0 && r > 0) {
      root = 1;
      pow = 1;
      while (pow <= n) {
         root = root + 1;
         pow = 1;
         for (i = 0; i < r; i++)
            pow = pow * radice;
      }

      printf("The floor approximation of
        the %d-root of %d is %d.\n", r, n, radice - 1);
   } else
      printf("Invalid input.\n");

   return 0;
}
```

Table 4: Compute $\lfloor \sqrt[r]{n} \rfloor$.