IOI'94 - Day 1 - Solution 1: The Triangle

[Introduction] [Problem Statement] [Test Data]

Problem Analysis

Let us introduce some notation to formalize the problem. The triangle consists of N rows with 1 < N <= 100. We number the rows of the triangle from top to bottom starting at 1, and the positions in each row from left to right also starting at 1. The number appearing in row r at position p is denoted by T[r, p] where 1 <= r <= N and 1 <= p <= r. There are 1+2+...+N = N*(N+1)/2 numbers in the triangle. This amounts to 5050 numbers in the largest possible triangle (N=100). The numbers are in the range from 0 to 99.

A route starts with T[1, 1]. For r < N, a route may step from T[r, p] to either T[r+1, p] or T[r+1, p+1]. It ends in some T[N, p] with 1 <= p <= N. Thus, a route visits N numbers in N-1 steps. At each step in a route there are two possible ways to go. Hence, there are 2^(N-1) routes in the triangle. The largest possible triangle (N=100) has 2^99 routes. This is a huge number close to 10^30. The smallest value for a route sum is zero, and the largest value is N*99, which is less than 10,000.

Our first design decision concerns the input data. Two approaches are possible:

1. Read all input data at the beginning of the program and store it internally for later use.

2. Process the input data while reading it, without ever storing all of it together.

Because the amount of input data is not so large (at most 5050 small numbers), it is easy to store all of it in program variables. Unfortunately, Pascal has no triangular arrays. It is, however, not a problem to introduce a square array of 100 x 100 numbers, and use only the lower-left triangle. Here are the Pascal declarations:

```
const
  MaxN = 100 ;

type
  index = 1..MaxN ;
  number = 0..99 ;

var
  N: index ; { number of rows }
  T: array [index, index] of number ;
    { T[r, p] is the number in row r at position p }
```

(The triangle can be made into a rectangle by folding it, but this complicates the program unnecessarily.) Our first three programs start by reading all input data. In the fourth program we illustrate the other approach.

For N=1 the problem is easy because there is only one route. In that case the maximum route sum equals T[1, 1]. For N>1, two types of routes can be distinguished: one half of the routes turns left on the first step from the top, the other half goes to the right. Each route sum equals T[1, 1] *plus* the sum over the route continuing from T[2, 1] when turning left, or from T[2, 2] when turning right. Therefore, the maximum route sum equals T[1, 1] *plus* the maximum of the sums over routes starting either in T[2, 1] or in T[2, 2]. Determining the maximum sum for routes starting in T[2, p] is the same problem for a smaller triangle.

Program 1

It is now easy to write a *recursive* function, say MaxRS, that computes the maximum route sum from an arbitrary position in the triangle:

```
function MaxRS(r, p: index): integer ;
  { return maximum sum over all down routes starting in T[r, p] }
  begin
  if r = N then MaxRS := T[r, p]
  else MaxRS := T[r, p] + Max(MaxRS(r+1, p), MaxRS(r+1, p+1))
  end { MaxRS } ;
```

The invocation MaxRS(1, 1) then solves the problem. This solution is presented in Program 1. However, this program passes the first three or four tests only. For a triangle with 60 rows (test 5) the number of routes is simply too large (more than 10^17) to be investigated in 30 seconds. You can eliminate the separate function Max (for computing the maximum of two numbers) by writing it out into the body of function MaxRS, but that will not help (enough).

This problem was included precisely because it is tempting to use recursion. The `plain' recursive solution was intended to fail (on some of the tests).

Program 2

There is a standard technique to speed up such a recursive program, known by such names as *dynamic programming*, *tabulation*, *memorization*, and *memoing*. Whenever the function MaxRS is called with actual parameters r and p for the first time, the result is computed and memorized in a table, say as MRS[r, p]. For every subsequent invocation of MaxRS with these same parameters r and p, the result is not recomputed, but simply retrieved from the table as MRS[r, p].

We introduce a table MRS that is initialized at -1 to indicate that the results are not yet known (note that all route sums are at least 0). The function MaxRS can be modified as follows:

```
function MaxRS(r, p: index): integer ;
  { return maximum sum over all down routes starting in T[r, p] }
  begin
  if MRS[r, p] = -1 then
    if r = N then MRS[r, p] := T[r, p]
    else MRS[r, p] := T[r, p] + Max(MaxRS(r+1, p), MaxRS(r+1, p+1)) ;
  MaxRS := MRS[r, p]
  end { MaxRS } ;
```

This idea is incorporated in Program 2. It is still a simple program and now it is also fast enough, because not all routes are followed completely. A disadvantage of this solution is that it requires additional storage for the table. In fact, Program 2 uses at least twice the amount of data memory compared to Program 1 (such a trade-off between speed and memory usage is a common dilemma). You can squeeze the table into the upper-right triangle of the square array, but that would complicate the program needlessly.

Program 3

When program 2 is analysed a bit further, it becomes clear that table MRS is filled in a particular order. Assuming that in the call Max(E, F), expression E is, for instance, always evaluated before F, the recursion pattern turns out to be very regular. Filling starts at the lower-left corner and continues from the bottom in diagonals slanting upward to the left. For five rows the order is:

```
        15
      10  14
     6   9  13
   3   5   8  12
 1   2   4   7  11
```

It is now a small step to come up with a program that fills table MRS in a non-recursive way. Furthermore, this can be done in a more convenient order, say row by row from the bottom. That is, for five rows in the order:

```
        15
      13  14
    10  11  12
   6   7   8   9
 1   2   3   4   5
```

The following procedure computes MRS non-recursively (also called *iteratively*):

```
procedure ComputeAnswer ;
  { m := the maximum route sum }
  var r, p: index ;
  begin
  for r := N downto 1 do
    for p := 1 to r do
      if r = N then MRS[r, p] := T[r, p]
      else MRS[r, p] := T[r, p] + Max(MRS[r+1, p], MRS[r+1, p+1]) ;
  m := MRS[1, 1]
  end { ComputeAnswer } ;
```

(Of course, you can eliminate the if-statement by introducing a separate p-loop for r=N, but the program is fast enough as it is and better to understand this way.) Observe that each number T[r, p] is now inspected exactly once. Therefore, we do not need a separate table for storing MRS if we put MRS[r, p] at T[r, p]. The type of T must be adjusted because the input T-values are numbers in the range 0..99, but MRS-values possibly not. This idea is worked out in Program 3.

Program 4

In Program 3, all input data must still be read and stored before the actual computation starts, because MRS is computed from bottom to top. Routes in the triangle can, however, also be considered from bottom to top by flipping the direction. If we take that point of view, then we can say that

- a route starts somewhere on the base at T[N, p] for 1 <= p <= N;
- it steps upward either left or right (on the boundary there is no choice), that is, from T[r, p], with 1 < r <= N and 1 <= p <= r, it may step to T[r-1, p-1] if 1 < p, and to T[r-1, p] if p < r;
- it always terminates at T[1, 1].

We now redefine MRS[r, p] as the maximum sum over all *up*-routes starting in T[r, p]. The final answer is then obtained as the maximum value among MRS[N, p] with 1 <= p <= N. The following recurrent relationships hold for MRS:

- MRS[1, 1] = T[1, 1]
- MRS[r, 1] = T[r, 1] + MRS[r-1, 1] if 1 < r
- MRS[r, p] = T[r, p] + Max(MRS[r-1, p-1], MRS[r-1, p]) if 1 < r and 1 < p < r
- MRS[r, r] = T[r, r] + MRS[r-1, r-1] if 1 < r

The case distinction can be avoided by defining MRS[r, p] = 0 for r < 1 or p < 1 or p > r. We then

simply have

- MRS[r, p] = T[r, p] + Max(MRS[r-1, p-1], MRS[r-1, p])

MRS can be computed iteratively from top to bottom. Therefore it is not even necessary to store all input values. Only one row of MRS needs to be stored. From this row and an input row the next row of MRS is computed (this computation is a little tricky). The final answer is the maximum of the last row computed. Program 4 implements this approach.

Variants of this problem

Here are some variations on this problem. Instead of just producing the maximum route sum, produce a path that attains this maximum, for instance, as a sequence of L's and R's, for left and right turns respectively.

Given a triangle with numbers, compute how many routes take on the maximum route sum. Similarly, compute the mean path sum.

Tom Verhoeff
Eindhoven University of Technology