IOI'94 - Day 1 - Solution 2: The Castle

[Introduction] [Problem Statement] [Test Data]

Problem Analysis

A castle consists of at most 50 x 50 = 2500 modules. Therefore, the entire castle can be read from the input file and stored in program variables. Before we decide on how to represent a castle inside the program, let us look at the tasks to be accomplished.

Given a castle, we are requested to determine the number of rooms, the area of a largest room (I write `*a* largest room' instead of `*the* largest room' because there can be several rooms of maximum area), and a wall such that its removal yields a room that is as large as possible. We now rephrase this more precisely.

Two neighboring modules are said to be *connected* when there is no wall between them. A *room* is a maximal set of connected modules. The *area* of a room is the number of modules it contains. Let us define the *potential* of an interior wall as the area of the room created by removing that wall. The third item to be determined is a wall with maximum potential (a *best* wall).

A castle has at most 2500 rooms, and the maximum room area is also at most 2500 (in fact, it is at most 2499, because there are at least two rooms according to the problem statement). There are (at most, but I would expect exactly) 4*50=200 exterior walls and at most (4*50*50 - 4*50) / 2 = 4900 interior walls.

A well-known algorithm for determining rooms is based on painting the modules, using a different color for each room. Starting with color number 0 in the north west module of the castle paint it and all modules connected to it (in one or more steps). Continue with an unpainted module, using paint number 1. Repeat this until all modules have been painted.

For this approach it is necessary to traverse the (unpainted) modules of the castle, and from each module to find the modules connected to it. Once all rooms have been painted, there are several ways to determine the room areas, the maximum area, and a best wall.

The castle has M rows and N columns, with 1 <= M <= 50 and 1 <= N <= 50. We number the rows of the castle from north to south starting at 1, and the columns from west to east also starting at 1. The module in row r and column c is denoted by Map[r, c]. For each module we record its walls, by listing for each direction (west, north, east, south) whether there is a wall. The order of the directions is inspired by the encoding of walls in the input file. For each module we also maintain its room (color) number. This is captured in the following Pascal declarations.

```
const
  MaxM = 50 ;
  MaxN = 50 ;

type
  Row = 1..MaxM ;
  Column = 1..MaxN ;
  Direction = (west, north, east, south) ;
  Module = record
    wall: array [Direction] of boolean ;
    nr: integer ; { room number, -1 if unknown }
    end { Module } ;
```

```
var
  M: Row ;
  N: Column ;
  Map: array [Row, Column] of Module ;
```

In the record Module we could also have chosen to declare

```
    wall: set of Direction ;
```

Which of the two declarations is to be preferred, depends on the operations that will be done on `wall'. For this problem it would not matter much, but initializing the array of booleans is slightly simpler.

Reading (and displaying) a castle

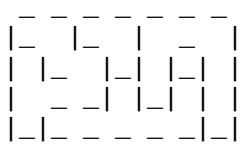
The following procedures reads a castle map from the input file `inp'. Its body follows directly from the structure of the input file. The number w that specifies the walls of a module is decoded by repeatedly inspecting and taking off the least significant bit with `odd(w)' and `w div 2'.

```
procedure ReadInput ;
  { read M, N, and Map ; initialize room numbers to -1 }
  var r: Row ; c: Column ; w: integer ; d: Direction ;
  begin
  readln(inp, M, N) ;
  if Test then writeln('Number of rows is ', M:1, ', number of columns ', N:1) ;
  for r := 1 to M do begin
    for c := 1 to N do with Map[r, c] do begin
      read(inp, w) ; { w encodes the walls of module Map[r, c] }
      for d := west to south do begin
        wall[d] := odd(w) ;
        w := w div 2
        end { for d } ;
      nr := -1
      end { for c with Map } ;
    readln(inp)
    end { for r } ;
  if Test then writeln('Input read') ;
  end { ReadInput } ;
```

When developing a program, it is good practice to produce some test output along the way to help verify that things work all right. For instance, after reading the castle, you can write it to the screen in a format that is easier to interpret than the encoded wall numbers. Here is a procedure to do so.

```
procedure WriteCastle ;
  { write Map to output }
  var r: Row ; c: Column ;
  begin
  for c := 1 to N do with Map[1, c] do
    if wall[north] then write(' _') else write('  ') ;
  writeln ;
  for r := 1 to M do begin
    for c := 1 to N do with Map[r, c] do begin
      if (c = 1) then if wall[west] then write('|') else write(' ') ;
      if wall[south] then write('_') else write(' ') ;
      if wall[east] then write('|') else write(' ')
      end { for c with Map } ;
    writeln
    end { for r }
  end { WriteCastle } ;
```

WriteCastle presents the map of the example in the problem statement as follows:



Determining the (number of) rooms

Procedure PaintMap will traverse the modules row by row (north to south), and within a row from west to east. Whenever it encounters an unpainted module, procedure PaintRoom is invoked to paint the module and all modules connected to it with the next color. We use the room number as color. PaintRoom is most easily implemented by a recursive procedure:

```
type
  RoomNumber = 0..MaxM*MaxN ;

var
  rooms: RoomNumber ; { number of rooms completely painted }

procedure PaintMap ;
  { paint the map }

  procedure PaintRoom(r: Row; c: Column) ;
    { if Map[r, c] is unpainted then paint it and all modules connected to it }
    begin
    with Map[r, c] do
      if nr = -1 then begin
        nr := rooms ;
        if not wall[west]  then PaintRoom(r, c-1) ;
        if not wall[north] then PaintRoom(r-1, c) ;
        if not wall[east]  then PaintRoom(r, c+1) ;
        if not wall[south] then PaintRoom(r+1, c)
        end { if }
    end { PaintRoom } ;

  var r: Row ; c: Column ;
  begin
  rooms := 0 ;
  for r := 1 to M do
    for c := 1 to N do
      if Map[r, c].nr = -1 then begin
        PaintRoom(r, c) ;
        rooms := succ(rooms)
        end { if }
  end { PaintMap } ;
```

(NOTE: succ(v) is a Standard Pascal notation for the successor of v. For integer v we have succ(v) =v+1. I happen to like succ.) For every unpainted module, PaintRoom is called once; it then colors that module and makes at most four more calls to PaintRoom. Thus, altogether at most 2500*(1+4) = 12,500 calls to PaintRoom are made. This should be feasible within the time limit. (What are the precise minimum and maximum number of calls to PaintRoom for a 50 x 50 castle?)

For testing purposes it is convenient to write a color map of the castle. This is done by procedure WriteColors:

```
procedure WriteColors ;
  { write Map colors to output }
  var r: Row ; c: Column ;
```

```
  begin
  for r := 1 to M do begin
    for c := 1 to N do write(Map[r, c].nr:2) ;
    writeln
    end { for r }
  end { WriteColors } ;
```

After calling PaintMap, WriteColors presents the map of the example in the problem statement as follows:

```
 0 0 1 1 2 2 2
 0 0 0 1 2 3 2
 0 0 0 4 2 4 2
 0 4 4 4 4 4 2
```

Determining the room areas

While rooms are painted, their area can be computed, and the maximum can be maintained as well. There is no need to store all areas computed. However, for the next task it is convenient to have a table that gives the area of each room:

```
var
  area: array[RoomNumber] of integer ; { area[n] is area of room nr. n }
  maxarea: integer ; { maximum room area }
```

Instead of modifying the procedure PaintRoom to compute the area as well (you run the risk of introducing errors; see Program 2 below), we write a separate procedure MeasureRooms that computes the areas of all rooms, and also the maximum area.

```
procedure MeasureRooms ;
  var r: Row ; c: Column ; n: RoomNumber ;
  begin
  for n := 0 to pred(rooms) do area[n] := 0 ;
  for r := 1 to M do
    for c := 1 to N do
      inc(area[Map[r, c].nr]) ;
  maxarea := 0 ;
  for n := 0 to pred(rooms) do
    if area[n] > maxarea then maxarea := area[n]
  end { MeasureRooms } ;
```

(NOTE: pred(v) is a Standard Pascal notation for the predecessor of v. For integer v we have pred(v) =v-1. inc(v) is a Turbo Pascal notation for v:=succ(v). It avoids duplicate determination of v's identity (address), which is useful here.)

Determining a wall with maximum potential

Recall that the potential of an interior wall is the area of the room created by removing that wall. For each interior wall we can easily compute its potential. Observe that this area is not necessarily the sum of the areas of the rooms on either side of the wall. (I made this mistake in my first program.) It could be that the same room appears on both sides of a wall, that is, the wall lies inside a single room. In that case its removal does not create a larger room. The following procedure BestWall considers all interior walls and determines a wall with maximum potential.

```
var
  bestrow: Row ; bestcol: Column ; bestdir: Direction ;
```

```
procedure BestWall ;
  var r: Row ; c: Column ; maxp: integer ;

  procedure Update(k1, k2: RoomNumber; d: Direction) ;
    var p: integer ;
    begin
    if k1 = k2 then p := area[k1] else p := area[k1] + area[k2] ;
    if p > maxp then begin
      maxp := p ; bestrow := r ; bestcol := c ; bestdir := d
      end { if }
    end { Update } ;

  begin
  maxp := 0 ;
  for r := 1 to M do
    for c := 1 to N do with Map[r, c] do begin
      if (r >< M) and wall[south] then Update(nr, Map[r+1, c].nr, south) ;
      if (c >< N) and wall[east]  then Update(nr, Map[r, c+1].nr, east) ;
      end { for c with Map }
  end { BestWall } ;
```

Note that the if-statements inside the nested for-loops cannot be eliminated by changing the upper bounds of the for-loops in the following way:

```
  for r := 1 to pred(M) do
    for c := 1 to pred(N) do ...
```

because this way possibly some interior walls (namely south walls of the east-most modules and east walls of the south-most modules) are forgotten! Test 3 would catch this error; the erroneous output would be:

```
9
36
1 1 S
```

One may wonder whether the maximum potential can be determined more efficiently. Inspecting all interior walls may seem overkill. However, the amount of work involved in procedure BestWall is on the same order as reading the input file and determining the rooms and their areas. Of course, it would suffice to inspect just neighboring rooms (and not *all* the module walls in between them). But it is difficult to collect and store this information more efficiently. Note that a wall of maximum potential not necessarily involves a room of maximum area!

Programs

Program 1 is the complete program. Program 2 is a variant where we compute the room areas while painting.

Variants of this problem

It is challenging to solve this problem with different constraints. For instance, what about a castle that is so big that it cannot be completely stored in program variables? Say the east-west dimension of the castle is at most 1000 modules and the north-south dimension at most 10,000 modules. If this is helpful, you may also assume that there are no more than 100 rooms.

Modify the program to find *all* rooms of maximum area and *all* walls with maximum potential indicating which room pairs are involved. Check whether any of the provided test input files is such that none of the walls with maximum potential involve a room of maximum area.

When judging programs for this problem, it is necessary to produce test input. Write a program that given a map of the castle (as produced by WriteCastle) generates an input file that encodes the walls with numbers.

Tom Verhoeff
Eindhoven University of Technology