SPIRAL WALK EXPOSED

Boyko Bantchev

Institute of Mathematics and Informatics
boykobb@gmail.com

***Abstract**: Traversing a matrix along a spiral is a popular small programming problem, but one which is often solved incorrectly, or only partially, or in an ill-structured way. To benefit teaching and learning good programming style, we considered worthy to present the construction of several solutions to this problem in an expository manner. Some related combinatorial problems are also discussed. Hopefully, the text can be useful to high and higher school teachers and students, as well as to practicing programmers.*

***Keywords**: algorithm, spiral, traversal, programming, calculation.*

Introduction

Traversing a matrix along a spiral is a popular small problem, commonly used as an exercise for students in programming, and sometimes for testing practical skills at programming interviews [2,3,4]. Simple though the problem may seem, most of the solutions that one can see on the World Wide Web and even in textbooks are incorrect, incomplete, ill-structured, overly complicated, or more than one of these. Therefore, it might be beneficial to both learners and teachers of programming to expose correct and well-structured solutions to the said problem, along with discussion of some related problems. This is the subject of the present article. It is written to be accessible to anyone having a basic knowledge of the C programming language.

The text can also be viewed as exposing a series of exercises in the so called 'code kata' style (see e.g. [1]), where a certain programming problem is being worked through again and again in possibly different ways. In this respect it might be of interest to practicing programmers.

Problem statement

A matrix is a rectangular table of elements of a kind, and traversing it means producing a sequence of these elements. There may be different ways of mapping the table's elements to a spiral-like curve. The spiral can evolve from the inside outwards, or it can shrink from the outside inwards, and it can be directed clockwise or inversely. A shrinking spiral will typically start, and an expanding spiral will typically end, at any of the four corners of the table. In the following we consider a clockwise shrinking spiral starting at the table's upper left corner.

If the table has *m* rows and *n* columns, we characterize it by saying that it is a *m*×*n* table. Here are examples of 3×4 and 4×5 tables of letters and their traversals:

```
a–b–c–d          a–b–c–d–e
      |                  |
e–f–g h          f–g–h–i j
|     |          |     | |
i–j–k–l          k l–m–n o
                 |       |
                 p–q–r–s–t
```

The traversal visits the elements `a b … g` and `a b … l`, respectively. But as the type of the elements and their specific values are immaterial to the traversal, we can consider only the size m×n of the table, and enumerate the coordinates of the elements rather than the elements themselves. For the above two tables, the result of traversing is then `1 1  1 2  … 2 3` and `1 1  1 2  … 3 2`.

Thus, the problem is, given *m* and *n* (*m*,*n*≥1), to produce the sequence of coordinates as explained.

Programming

In order to solve a programming problem, one needs to find a suitable conceptualization for it. For more complex problems, suitability may imply considerations of various kinds, including data structure selection and resource efficiency. In the present case, we are only interested in simplicity. We assume that a conceptualization is simple if it admits a straightforward and sufficiently precise yet concise description in a natural language.

One way to conceptualize the spiral traversal of a table is to think of it as 'cutting' the table along its borders. This is illustrated on the left part of Figure 1: the topmost row is cut first by enumerating its elements from left to right; then the rightmost column is cut by enumerating the respective elements in downward direction; then the bottommost row is cut by walking it through from right to left; finally, the leftmost column is cut, i.e. enumerated going upwards; then these steps are repeated. The general idea is that each of the four cuts leaves a smaller table than the one to which it applies, but a rectangular table anyway. Thus, the traversal problem simplifies to doing cuts in succession until the table is empty.

Figure 1

We must be careful to note when, after a cut, the resulting table gets empty, and end the traversal. Not doing this may entail an error, such as visiting cells of the table that have been already visited, or even failing to terminate the process altogether. It is easy to observe (and prove) that for a square initial table of size *n*×*n* the resulting table gets empty after cutting some row. This leads to the following procedure, where a is the coordinate of the current left, and simultaneously the current top border, and b is the right and simultaneously the bottom border:

```
 1: void spiral1(int n) {
 2:   int a, b, i, s;
 3:   for (s=0,a=1,b=n;; ++a,--b) {
 4:     for (i=a; i<=b; ++i) VISIT(a,i);
 5:     if (a==b) return;
 6:     for (i=a+1; i<=b; ++i) VISIT(i,b);
 7:     for (i=b-1; i>=a; --i) VISIT(b,i);
 8:     if (b-a==1) return;
 9:     for (i=b-1; i>a; --i) VISIT(i,a);
10:   }
11: }
```

The s variable keeps track of the number, in the order of traversal, of the cell to visit, and VISIT(u,v) is a macro for visiting the cell with coordinates u,v:

```
#define VISIT(u,v) printf("%d: %d,%d\n",++s,u,v)
```

The four inner loops implement the cuts. They are varyingly initialized, and their terminating conditions are similarly varied. The outer loop is nominally indefinite, the actual termination taking place at lines 5 and 8.

For a table *m*×*n* that is not necessarily square, we need to introduce two more variables. As before, a and b point at the top and bottom rows of the current table, but now c and d are the coordinates of its leftmost and rightmost columns:

```
 1: void spiral2(int m, int n) {
 2:   int a, b, c, d, i, s;
 3:   for (s=0,a=1,b=m,c=1,d=n;;) {
 4:     for (i=c; i<=d; ++i) VISIT(a,i);  // row a
 5:     if (a<b) ++a; else break;
 6:     for (i=a; i<=b; ++i) VISIT(i,d);  // column d
 7:     if (c<d) --d; else break;
 8:     for (i=d; i>=c; --i) VISIT(b,i);  // row b
 9:     if (a<b) --b; else break;
10:     for (i=b; i>=a; --i) VISIT(i,c);  // column c
11:     if (c<d) ++c; else break;
12:   }
13: }
```

A bit of reasoning or experimenting reveals that, for a rectangular table, any of the four kinds of cuts may happen to be the last one, so we are careful to terminate the traversal after any of the four inner loops, upon a specific condition.

Interestingly, `spiral2`, which is a more general procedure that `spiral1`, is also more regular with respect to structure! The four inner loops are written in a very alike manner, as are the four exiting conditions at lines 5, 7, 9, and 11. Besides, each of `a`, `b`, `c`, and `d` changes at exactly the right place: immediately after the loop that affects it, and only if it does make sense to change – when the traversing is not yet finished. That generalization can lead to a more regular, simple, and elegant program at practically no cost (the size increases only insignificantly) is an important lesson to learn on this example.

There is, however, another way to look at how a table can be traversed spirally. Whenever $m,n \geq 2$, instead of doing individual cuts along the borders, we can think of stripping entire rectangular frames comprised of these borders. If each side of a rectangle is taken to consist of all cells in the respective direction but the last one, as depicted on Figure 1, right, then each side is nonempty, the lengths of horizontal sides being both equal to $n-1$, and those of the vertical sides to $m-1$.

Note that, according to this conception of the problem, the sides of a rectangle correspond to half-open ranges – the kind of ranges that one is used to in modern programming and that is well supported in languages such as C, C++, Java, etc.

For a square table of size $n \times n$ we now obtain the following traversal procedure:

```
 1: void spiral3(int n) {
 2:   int a, b, i, j, s;
 3:   for (s=0,a=1,b=n; a<b; ++a,--b) {
 4:     i = j = a;
 5:     for (; j<b; ++j) VISIT(i,j);
 6:     for (; i<b; ++i) VISIT(i,j);
 7:     for (; j>a; --j) VISIT(i,j);
 8:     for (; i>a; --i) VISIT(i,j);
 9:   }
10:   if (n%2) VISIT(a,a);
11: }
```

Line 10 caters for the fact that for odd values of n, after traversing $(n-1)/2$ rectangles, there remains a single unvisited cell at the centre of the table.

It can be observed how much simpler and more elegant `spiral3` is, compared to `spiral1`. This is due to two reasons. One is that there is no need now to check at two different places for whether the traversal should be terminated. More importantly, the spiral walk is now more regular, for there is a single pair `i j` of coordinates used to visit all cells. That pair is initialized at line 4 once for each iteration of the main loop (i.e., once for a rectangle). Each of the inner loops

advances `i` or `j`, but this never causes the pair `i j` to lose track of the rectangular walk, because each loop leaves the respective variable pointing precisely at the starting cell of the next loop. That is why the internal loops need no, and have no, explicit initialization – each of them inherits its initialization from the previous one.

This organization of the program clearly demonstrates the importance of the seemingly minor change from closed to half-open ranges in representing a rectangular frame.

For rectangular tables `spiral3` generalizes to the following procedure, where we introduce `c` and `d` just like we did for `spiral2`:

```
 1: void spiral4(int m, int n) {
 2:   int a, b, c, d, i, j, s;
 3:   for (s = 0, i = j = a = c = 1, b = m, d = n
 4:       ; a<=b && c<=d
 5:       ; ++a, --b, ++c, --d, ++i, ++j) {
 6:     for (; j<d; ++j) VISIT(i,j);
 7:     for (; i<b; ++i) VISIT(i,j);
 8:     if (a==b || c==d) {VISIT(i,j); return;}
 9:     for (; j>c; --j) VISIT(i,j);
10:     for (; i>a; --i) VISIT(i,j);
11:   }
12: }
```

Like in `spiral3`, we need to be careful of the existence of 'unreachable' cell(s) of the table – those that remain after all possible rectangles are traversed, and therefore have to be visited separately. For a square table, the only unreachable cell is at the centre, if there is one. What are these cells for a rectangular table $m \times n$?

Figure 2

A bit of investigation shows that unreachable cells exist if and only if $\min(m,n)$ is odd. In this case, there is a middle row ($m \le n$) or a middle column ($m \ge n$) in the table, a part of which remains after traversing all enclosing rectangles (Figure 2). Let's call that remaining part a *spine*. No matter whether a spine is horizontal or vertical, if we

start to traverse it like a rectangle, but only do half of the work, i.e., walk along the upper and the right sides (one of these walks being necessarily void), there remains only one unreachable cell – the last one along the spine (on Figure 2 that cell is cross-hatched). This is why in `spiral4` the main loop's body is split by line 8, at which the presence of an unreachable cell is ascertained by checking whether the current rectangle is actually a spine. Note that the cell is nevertheless pointed at by the pair `i j`, like all other cells that are being visited.

Now that we have a form of traversal in which the coordinate pair `i j` moves smoothly along each rectangle, the following observations are in order:

- at each move from a cell to another, exactly one of the values `i` and `j` changes, and the change is either by 1 or by –1;
- at corner positions, the running variable always changes from `i` to `j` or v.v.;
- at two diagonal corner positions, the value to advance the running variable with changes from 1 to –1 or v.v.

These are invariant properties of traversing a table. However, in `spiral4` they are not explicitly expressed, as e.g. we use different loops for the different sides of a rectangle, and the only information passed from a loop to another is the values of `i` and `j`.

In order to make use of the observed regularities, we now redo `spiral4`, in particular unifying the four inner loops. The result is `spiral5`:

```
 1: void spiral5(int m, int n) {
 2:   int i, j, km, kn, k, d, *p, s;
 3:   for (s=0, i=j=1, km=m, k=kn=n, d=1, p=&j
 4:       ; km>0 && kn>0
 5:       ; km-=2, k=kn-=2, ++i, ++j)
 6:     for (int _0=2; _0; --_0) {
 7:       for (int _0=2; _0; --_0) {
 8:         for (int _0=k-1; _0; --_0)
 9:           {VISIT(i,j); *p += d;}
10:         p = p==&i ? &j : &i;  k = km+kn-k;
11:       }
12:       if (km==1 || kn==1) {VISIT(i,j); return;}
13:       d = -d;
14:     }
15: }
```

Instead of by boundary values `a`, `b`, `c`, `d`, the traversed rectangle is now represented only by its height `km` and width `kn`. Each side of a rectangle has length `k`, where `k` is either `km` or `kn`, so the loop at lines 8-9 says 'do `k-1` times: visit a cell and advance to the next one'. As the pointer `p` refers to either `i` or `j`, it is one of

these variables that is incremented at line 9, the increment `d` being 1 or –1. A rectangle is traversed by splitting the work – the loop at lines 6-14 – into two halves, in each half dealing with two sides – the loop at lines 7-11. Each time a half ends, `d` switches to the opposite value (line 13). After traversing a side, at line 10, `p` switches from (pointing at) `i` to `j` or v.v., and `k` switches from `km` to `kn` or v.v.

It is important to note that each of the loops at lines 6, 7, and 8 merely establishes a certain number of repetitions of the respective body. For that purpose each loop introduces a variable, but the latter is not used in the loop; it just counts from the given initial value down to 0. This 'non-use' is accentuated by giving each of the three variables a formal and unusual name, a kind of non-name.

To stress even more that the three nested loops are mere repetitions, we introduce the following macros:

```
#define REP(t) for (int _0=t; _0>0; --_0) {
#define END }
```

Using them, the above procedure takes the even cleaner form:

```
 1: void spiral6(int m, int n) {
 2:   int i, j, km, kn, k, d, *p, s;
 3:   for (s=0, i=j=1, km=m, k=kn=n, d=1, p=&j
 4:       ; km>0 && kn>0
 5:       ; km-=2, k=kn-=2, ++i, ++j)
 6:     REP(2)
 7:       REP(2)
 8:         REP(k-1) VISIT(i,j); *p += d; END
 9:         p = p==&i ? &j : &i;  k = km+kn-k;
10:       END
11:       if (km==1 || kn==1) {VISIT(i,j); return;}
12:       d = -d;
13:     END
14: }
```

Calculating

Traversing whatever structure is generating a sequence of its elements. For each such problem, there is a related one: given an element of the structure, to find its ordinal number in the generated sequence without actually generating the latter. This is a combinatorial problem requiring calculation. The ability to solve such problems is a part of what makes a competent programmer.

Related to traversing a table as presented here, the above problem is, given the size of a table and a pair of cell coordinates, to find the ordinal number of the cell in the order of traversal (the value of `s` in each of the procedures above). The problem can be solved as follows.

Recalling that the spiral traversal consists of properly alternated two kinds of horizontal and two kinds of vertical linear traversals, we can categorize the cells of the table into four groups. A cell is north (N) if it is visited within a horizontal traversal from left to right; east (E) if visited within a vertical, downwards-directed traversal; south (S) if visited 'horizontally', moving from right to left; and west (W) if visited when moving upwards. The four groups meet each other in pairs along the four diagonals, as well as along the spine, as shown on Figure 2 by the grey cells.

In other words, the sets N, E, S, and W consist of, respectively, rows scanned eastwards, columns scanned southwards, rows scanned westwards, and columns scanned northwards. Besides, each row and column, and thus each cell, belongs to exactly one rectangle, these rectangles being nested within each other, together constituting the table. Nesting immediately suggests sequential numbering, and knowing the ordinal number T of a rectangle in that sequence lets one compute the total number of cells on the rectangles that precede T. For a given rectangle and a cell on it, we can also find, depending on the N/E/S/W category of the cell, the number of cells on the rectangle that precede the given cell in the traversal.

Therefore, the ordinal number of a cell in the traversal sequence can be found by finding the category of the cell, obtaining the ordinal number T of the respective rectangle, and, on this basis, finding the total number of cells on the same and other rectangles that precede the given cell in the traversal.

The equations of the four diagonal lines of a table $m \times n$ are:

NW: $i-j = 0$ NE: $i+j = n+1$

SW: $i+j = m+1$ SE: $i-j = m-n$.

Let i and j be the coordinates of a specific cell in the table. From the above equations it is easy to find the category of the cell as depending on i and j:

N: $i \le (m+1)/2$, $i \le j \le n-i+1$,

E: $j \ge (n+1)/2$, $n-j+1 \le i \le m-n+j$,

S: $i > (m+1)/2$, $m-i+1 \le j \le i-m+n$,

W: $j < (n+1)/2$, $j < i \le m-j+1$.

Note that the cells on the NE diagonal are considered to belong to both N and E categories, and similarly for SE and SW diagonals; the cells on NW are only N. The cells of a horizontal (vertical) spine are considered N (E), as they should.

The number T for the rectangle on which the cell (i,j) is can be determined as the distance of the cell from the table border that corresponds to the cell's category. Thus, by category, T equals: i if N, $n-j+1$ if E, $m-i+1$ if S, and j if W.

A $p \times q$ rectangle has $2(p+q)-4$ cells on it. Each rectangle is by two cells smaller in both height and width than the immediately enclosing one, so the former consists of 8 cells less than the latter. It follows that the k-th rectangle has $2(m+n)-8k+4$ cells. Summing up for k=1,2,…,$T-1$, the rectangles that come prior to T in the

traversal together contain $2(T-1)(m+n-2T+2)$ cells. Substituting for T depending on the category of the cell, we obtain for this quantity:

N: $2(i-1)(n+m-2i+2)$,

E: $2(n-j)(m-n+2j)$,

S: $2(m-i)(n-m+2i)$,

W: $2(j-1)(n+m-2j+2)$.

In order to find the ordinal number of the cell on its own rectangle T, we note that the rectangle has its NW corner at coordinates (T,T) and is of height $m-2(T-1)$ and width $n-2(T-1)$. Then, using the coordinates (i,j) and the cell's category, we add up the needed amounts. For example, an E cell would be preceded by the $n-2(T-1)$ cells on the N side, and $i-T-1$ other cells on the E side of the rectangle. That cell's ordinal number on the rectangle is therefore $n+i-3T+2 = i+3j-2n-1$. Doing as said for all the four categories and adding the above found amounts for the preceding rectangles, we finally obtain for the ordinal number of (i,j) in the overall traversal, by category:

N: $(i-1)(2n+2m-4i+3)+j$,

E: $2(n-j)(m-n+2j-1)+i+j-1$,

S: $(m-i+1)(2n-2m+4i-1)-m-j$,

W: $2j(n+m-2j)-i+j+1$.

Note that the assumption that a diagonal point is categorized ambivalently means that, for such a point, two of the above formulae are equally applicable.

A number of other traversal-related calculational problems suggest themselves. For example:

- Which cell is the last one (or has a given ordinal number)?
- Given the coordinates of a cell, which cell is the previous one visited, and which is the next one to visit?
- How many 90° turns are there in a spiral traversal?
- Which of the four directions is the last one taken (or which is the one with a given ordinal number in the sequence of directions)?
- At which cell does the last turn (or the turn with a given number) occur?

Some of the problems can be solved by immediate reasoning, some others by applying the already obtained results, and yet some others – by using similar techniques. One can also consider other kinds of spiral traversal, including outwards-directed, and pose similar questions.

Concluding remarks

Taking spiral traversing as an example of educational problem solving, we have developed solutions of different kinds. Further analysis of the solutions can elucidate

what features, and in what respect, make one or another of the solutions 'better'. We leave this to the interested reader.

A path not explored here is solving the problem in a functional – as opposed to imperative – manner. Using a functional programming language leads to a very different style of expression and extremely concise – up to one line! – and elegant, although not necessarily resource-efficient solutions. The functional point of view certainly adds to the educational value of discussing this programming problem.

Finally, we would like to stress that solving calculational problems related to a programming problem should never be underestimated. Beside its immediate utility as an exercise in combinatorics, it helps fully comprehend the original problem, and can provide insights on building better programming solutions.

References

1. *Code Katas.* http://codekatas.org.
2. E. McDonnell. *At play with J: Volutes.* Vector, Vol.13 (1996), No.2, 144-153. http://jsoftware.com/papers/play132.htm.
3. *Rosetta code: Spiral matrix.* http://rosettacode.org/wiki/Spiral_matrix.
4. М. Бърнева. *Компютърните програми – верни, надеждни и красиви.* Сб. „Информатика и икономика", Шумен 2003, стр.48-51.

ОБХОЖДАНЕТО ПО СПИРАЛА ОБЯСНЕНО

Бойко Банчев

Институт по математика и информатика

***Резюме**: Обхождането на таблица по спирала е популярна малка задача по програмиране, но често бива решавана погрешно или частично или пък зле структурирано. В името на преподаването и усвояването на добър стил на програмиране намерихме за добре да представим с пояснения построяването на няколко решения на тази задача. Разглеждат се и някои свързани с нея комбинаторни задачи. Надяваме се, че текстът би бил полезен за преподаватели и учащи от средното и висшето училища, както и за занимаващите се практически с програмиране.*