IOI'94 - Day 2 - Solution 3: The Circle

[Introduction] [Problem Statement] [Test Data]

Problem Analysis

Let me start by introducing some terminology, given a circular arrangement of numbers. Number `p` (not necessarily appearing in the circular arrangement) is said to be *creatable*, when there is a segment of one or more adjacent numbers in the circular arrangement with sum `p`. The *tail* of number `m` is defined as the number `t`, `t >= m`, such that all numbers from `m` to `t` are creatable and number `t+1` is not creatable (if `m` is not creatable then its tail is defined as `m-1`).

We can now reformulate the problem as follows. Given are numbers `n`, `m`, and `k` with `1 <= n <= 6` and `1 <= m,k <= 20`. The objective is to find *all* circular arrangements of `n` numbers, each number being at least `k`, such that the tail of `m` is as large as possible. It is also required to output that tail. In fact, the problem statement prescribes the output more precisely: the first line contains the maximum tail, the following lines present the circular arrangements (one per line), such that they start with their smallest number.

Observe that creatable numbers are at least `k`. Thus, for `m < k` the tail of `m` is `m-1`, because `m` itself is not creatable. For `k <= m` the maximum tail is *at least* `m+n-1`, because of the circular arrangement consisting of the `n` numbers `m`, `m+1`, ..., `m+n-1` *in arbitrary order*. Since in the output these arrangements should all start with the smallest number---that is, with `m`---there are `(n-1)!` such arrangements. These arrangements and also their tail are called *trivial*. The trivial tail is a *lower bound* on the maximum tail (provided that `m >= k`).

N.B. The number `(n-1)!` of trivial arrangements is *not* an *upper bound* on the number of arrangements to be output. In fact, it turns out that the input `n,m,k = 5,10,5` has 32 arrangements for the maximum tail of 14.

> From now on we assume `k <= m` (the Competition Rules explicitly state that all test data will have solutions; this also implies that 1 is a *lower bound* on the number of arrangements to be output). Furthermore, when we speak of the tail, we mean the tail of `m`.

The cases `n = 1, 2, 3` are special, because *every* non-empty subset of the numbers appears as a segment of adjacent numbers in the circle. The case `n=1` is uninteresting: the maximum tail is `m` obtained by the unique arrangement `m`.

The cases `n = 2, 3` can also be solved ``by hand'', but they are tricky! For instance, for `n=2` and `m > 1` there is the arrangement `m`, `m+1` yielding the maximum tail `m+1` (observe that `m+2` is not creatable since `2m+1 > m+2`). But if `k=1` then there is also the arrangement `m`, `1` yielding tail `m+1`. If `m=1`, then the arrangement `1`, `2` yields the maximum tail `m+2 = 3`.

We have already given the lower bound `m+n-1` on the maximum tail. We can also give an *upper bound*. The maximum tail is at most the sum of all numbers in the circle. This upper bound is not very instructive. Observe that for `1 <= p < n`, there are `n` segments of `p` adjacent numbers, Furthermore, there is a single segment of `n` adjacent numbers (consisting of *all* numbers in the circle). Thus, in total there are `(n-1)*n+1` segment sums. In the best case every such segment creates a different number. This means that the maximum tail is at most `m+(n-1)*n`.

Systematic investigation of arrangements

Let us introduce some constants, types, and variables:

```
const
  Max_n =  6;
  Max_m = 20;
  Max_k = 20;

type
  Circle = array [1..Max_n] of integer; { intended: array [1..n] of integer }

var
  n: 1..Max_n; { input value }
  m: 1..Max_m; { input value }
  k: 1..Max_k; { input value }
```

Reading the input is easy. After that we will go through all possible arrangements once and store the best arrangements so far.

```
var
  BestCount: integer; { # best arrangements so far }
  BestArr: array [1..1000] of Circle;
    { BestArr[1..BestCount] = best arrangements so far }
  BestTail: integer; { maximum tail found so far }
```

It is not clear at this point what upper bound to choose for the array `BestArr` with best arrangements so far. We have just picked 1000. If there is enough time, then we could first go through the possible arrangements to find out what the maximum tail is and in a second phase go through the arrangements again to filter out the ones that have this maximum tail (possibly exploiting knowledge about the maximum tail). That would avoid storing an unknown number of intermediate results. Yet another solution is presented later.

The final output is produced by procedure `WriteOutput`:

```
procedure WriteOutput;
  var i, j: integer;
  begin
  writeln(out, BestTail:1) ;
  for i := 1 to BestCount do begin
    for j := 1 to n do write(out, ' ', BestArr[i][j]:2) ;
    writeln(out)
    end { for i } ;
  if Test then writeln('Max tail = ', BestTail:1, '; # arr. = ', BestCount:1)
  end { WriteOutput } ;
```

We will use the following global variables for constructing and checking arrangements:

```
var
  Arr: Circle; { Arr[1..n] is the circular arrangement }
  Tail: integer; { tail of Arr[1..n] }
```

Given an arrangement of `n` numbers, procedure `ComputeTail` determines the tail of `m`. The idea is to compute the sums of all segments of adjacent numbers in the circular arrangement `Arr`. This generates the set of creatable numbers, from which the tail is readily derived.

```
procedure ComputeTail;
  { post: Tail = tail of m for circular arrangement Arr[1..n] }
  var
```

```
    a, b, s, u: integer ;
    Creatable: array[1..51] of boolean ; { Creatable[i] = i is creatable }
  begin
  u := 1 + m + (n-1)*n ; { 1 + upper bound on maximum tail }
  for a := 1 to u do Creatable[a] := false ;
  for a := 1 to n do begin
    s := 0 ; { s = sum of Arr[a..b] }
    for b := a to n do begin
      s := s + Arr[b] ;
      if s <= u then Creatable[s] := true
      end { for b } ;
    for b := 1 to a-2 do begin
      s := s + Arr[b] ;
      if s <= u then Creatable[s] := true
      end { for b }
    end { for a } ;
  Tail := m ; { linear search for smallest uncreatable number }
  while Creatable[Tail] do inc(Tail) ;
  dec(Tail)
  end { ComputeTail } ;
```

There are a few things to be pointed out about `ComputeTail`. Because of our choice for `u` we know that at least one of the numbers from `m` to `u` is *not* creatable. In the for-loops, `a` is the first sector of the segments considered. The first `b`-loop considers segments that start at sector `a` and that do not cycle beyond sector `n`. The second `b`-loop cycles around to sector 1 and beyond. Here we need not go further than `a-2` because the segment consisting of all numbers was already coverd by the first loop for `a = 1` (strictly speaking it has not first sector).

For each arrangement constructed, the variables `t`, `BestArr`, `BestTail` are updated by procedure `CheckArrangemant`:

```
procedure CheckArrangement;
  begin
  ComputeTail ;
  if Tail > BestTail then begin { improved arrangement }
    BestCount := 1 ; BestArr[BestCount] := Arr ; BestTail := Tail
    end { then }
  else if Tail = BestTail then begin { another arrangement with same tail }
    inc(BestCount) ; BestArr[BestCount] := Arr
    end { then }
  end { CheckArrangement } ;
```

We would like to write `n` nested for-loops to go through all possible arrangements. This is a little difficult since `n` is a variable. It can be accomplished by a recursive procedure. We have named it `FillRemainder`. The outermost loop is a special case treated below.

```
procedure FillRemainder(i: integer);
  { Fill remaining sectors Arr[i..n] in all possible ways }
  { pre: i > 1 }
  var j, u: integer;
  begin
  if i > n then { all sectors filled, check whether arrangement is useful }
    CheckArrangement
  else begin { fill sector i in all possible ways }
    if Trace then begin
      for j := 1 to pred(i) do write(Arr[j]:3) ;
      writeln
      end { if } ;
    u := m+(n-1)*n ; { naive upper bound on numbers to try }
    for j := Arr[1] to u do begin { N.B. Arr[1] is smallest number }
      Arr[i] := j ;
```

```
      FillRemainder(succ(i))
      end { for j }
    end { then }
  end { FillRemainder } ;
```

The for-loop tries all possible numbers `j` at sector `i`. Since sector 1 contains the smallest number, `j` can start at `Arr[1]`. the upper bound for `j` is less straightforward. We have picked `m+(n-1)*n` because this is the upper bound on the maximum tail. It does not make sense to include larger numbers. It should be noted that this upper bound is rather rough, and may cause the investigation of too many arrangements.

Procedure `FillRemainder` is called by procedure `ComputeAnswer` that also provides the outermost for-loop:

```
procedure ComputeAnswer;
  { pre: k <= m }
  var j: integer;
  begin
  BestCount := 0 ; BestTail := m+n-1 ;
  for j := k to m do begin
    Arr[1] := j ; { N.B. this is the smallest number in the circle }
    FillRemainder(2) ;
    end { for j }
  end { ComputeAnswer } ;
```

The only numbers to try in sector 1 are from `k` to `m`, since smaller numbers are not allowed by definition, and with larger numbers `m` would not even be creatable. Note that for the best arrangements we need not necessarily have `Arr[1] = m`. An example is provided by the fourth test case (see input-4.txt and output-4.txt).

All of this is put together into Program 1. This program solves test cases 1, 2, and 4 within the time limit; for the others that is doubtful.

Another remark about Program 1 is that if `BestTail` would be initialized to 0 instead of `m+n-1`, then the input combination `n,m,k = 5,17,5` would cause an overflow of the list of (intermediate) best arrangements: the program would have to skip 1261 arrangements with tail 20, before finding the first arrangement with (maximum) tail 21 (of which there are only 24). It is hard to give an upper bound of the number of (intermediate) best arrangements. In the next program we will just start writing the best arrangements to the output file, and overwrite it if we find an improvement.

Program 2

What improvements can we make to speed up Program 1? Why is it too slow? There are two things that come to mind. The first is that too many arrangements are checked. A reason for this could be that the upper bound `u` for `j` in procedure `FillRemainder` is unnecessarily large. The second is that when determining the tails of arrangements a lot of computations are duplicated. Observe that two arrangements that differ only in one sector share about half of their creatable numbers.

It is difficult to improve the speed by avoiding duplicate computations when determining tails of arrangements. The main reason for this is that (n-1)*n/2 + 1 of the `(n-1)*n+1` segment sums can only be computed when the *last* sector has been filled in. For `n=6` there are 31 segment sums to be computed, of which 16 involve on the last sector.

At the expensive of some overhead we can determine a tighter upper bound, though this is a bit complicated. Let us take the case `n,m,k = 5,3,1` as an example. Consider a state where the first

three sectors have been filled as follows:

```
    i  |  1   2   3   4   5
-------|-------------------
Arr[i] |  3   5   7
```

The call `FillRemainder(4)` will try numbers at `Arr[4]` starting from 3 up to some upper bound. In Program 1 this upper bound is 23. In fact, Program 1 will check 708,578 arrangements (and for each arrangement the tail of `m` is determined).

The three sectors that have been filled in already determine 6 segment sums. Here is a table indicating the creatable numbers:

```
          Creatable | 3   5   7 8         12       15
--------------------|-----------------------------------
not (yet) Creatable |   4   6     9 10 11    13 14    16
```

Filling in sector 4 with number `j` implies that the 11 (!) segment sums involving `Arr[4]` will be at least `j`. In total there are 21 segment sums, so there are only 21-11-6=4 segment sums that involve `Arr[5]` and *not* `Arr[4]`.

Assume we try `j >= 12` then all 11 segment sums involving `Arr[4]` will at least 12, and the 4 remaining segment sums involving `Arr[5]` can then at best create the number 4, 6, 9, and 10. This would leave 11 uncreatable, yielding a tail of at most 10. Apparently, a good upper bound for `j` is 11 (quite a bit less than 23).

More in general, a better upper bound is obtained by determining which numbers are already creatable, and by calculating the number `q` of segment sums that involve `Arr[i+1..n]` and *not* `Arr[i]`. The segment sums involving `Arr[i]` are all at least `Arr[i]`, and there will be at most `q` smaller segment sums created from `Arr[i+1..n]`. The `(q+1)`th number that is not yet creatable is a suitable upper bound for `Arr[i]` because beyond that point the tail can no longer grow.

Here is procedure `ComputeUpperBound` that computes the improved upper bound:

```
procedure ComputeUpperBound(i: integer; var ub: integer);
  { post: ub = upper bound for Arr[i] based on Arr[1..i-1] }
  var
    a, b, s, u: integer ;
    Creatable: array[1..51] of boolean ; { Creatable[p] = p is creatable }
  begin
  u := 1 + m + (n-1)*n ; { 1 + upper bound on maximum tail }
  for a := 1 to u do Creatable[a] := false ;
  for a := 1 to pred(i) do begin
    s := 0 ; { s = sum of Arr[a..b] }
    for b := a to pred(i) do begin
      s := s + Arr[b] ;
      if s <= u then Creatable[s] := true
      end { for b } ;
  a := n - i ; { a = # unfilled sectors besides Arr[i] }
  a := n*a - (a*succ(a)) div 2 + 1 ;
  { a = 1 + # segment sums involving Arr[i+1..n] and not Arr[i] }
  ub := m ;
  while a <> 0 do begin
    while Creatable[ub] do inc(ub) ;
    inc(ub) ; dec(a)
    end { while }
  end { ComputeUpperBound } ;
```

This is incorporated into a Program 2. For the case `n,m,k = 5,3,1` now only 15,173 arrangements are checked. The case `n,m,k = 6,1,1` drops from 28,629,151 arrangements checked by Program 1 to 156,072 by Program 2.

Variants of this problem

What if sectors may be used more than once? We still require that they are adjacent. For example, for `n=4`, we could have a segment involving the 5 sectors 2, 3, 4, 1, 2 (sector 2 being used twice).

Given `n,m,k` find the tail of `m` with the most arrangements. Each arrangement yields a tail of `m`. We are now interested in maximimizing the number of arrangements that yield the same tail (instead of maximizing the tail).

Find all triples `n,m,k` such that for the maximum tail of `m` there is at least one arrangement whose smallest number is less than `m`.

Find all triples `n,m,k` such that their maximum tails have a maximum number of arrangements. I noticed that `n,m,k = 6,19,6` has maximum tail 24 for which there are 150 arrangements. Can you find triples with more arrangements for their maximum tail?

To help you on your way, the file all.txt lists for each case its maximum tail and the number of corresponding arrangements.

Tom Verhoeff
Eindhoven University of Technology