Solution 1: The Triangle

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.

Solution 2: The Castle

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.

Solution 3: The Primes

Problem Analysis

Because *all* 5x5 squares of digits, satisfying certain requirements, must be generated (instead of just one), a systematic approach is necessary to exhaust all possibilities. One important subtask is to recognize five-digit primes with a given digit sum.

Let us investigate some numbers, such as the number of 5x5 digit squares. There are 25 digits in the square; the digit in the top left-hand corner is given. Thus, there are at most 10^24 squares to be considered. This number can be reduced considerably. For instance, the top row and the left column cannot contain any zeroes. Also the digit sums of five rows, five columns, and two diagonals are given, twelve digit sums altogether. That **roughly** means that twelve digits are determined by the others. Also observe that the nine digits in the bottom row and the rightmost column are odd. Forgetting about the details of primes, this means that the number of candidate 5x5 squares is on the order of 9^6*10^6/2^9, or approximately 10^9. This is more than you can expect to investigate in 90 seconds on a 33 MHz machine (90 seconds then give you 3*10^9 clock cycles).

Of course, many combinations can be ruled out at an early stage, because of the primality condition. We can either check primality of 5-digit numbers on-the-fly (while filling in the square), or make a table of them at the beginning (before filling in the square). What does it cost to check primality on-the-fly? Consider for instance the method that works by trial division. There are 65 primes <=sqrt(99,999). (The 65th and 66th prime are 313 and 317, furthermore 313*313=97,969 and 317*317=100,489. I cheated here: I asked Mathematica. You can also make a rough estimate by using the **Prime Number Theorem**.) This means that primality testing can be done by at most 65 divide operations. For primes it indeed takes at least that many divisions. This method seems too time-costly to do on-the-fly.

Generating Primes

We decide to pre-compute a table of 5-digit primes with a given digit sum. This is also useful because it will tell us how many such primes there are. In fact, we can make a file with all 5-digit primes together with their digit sum. Our square-generating program then starts off to read the primes with the appropriate digit sum from this file. Since this way the prime finding is kept outside the square generator, we do not have to worry too much about how we check primality. Here is a function IsPrime that works by trial division with all integers less than the square root (for n=99,999, this involves no more than 316 divisions):

```
function IsPrime(n: integer): boolean ; { pre: n >= 4 }
  var i, j, h: integer ;
  begin
  h := trunc(sqrt(n)) + 1 ;
  i := 2 ; j := h ; { bounded linear search for smallest divisor }
  while i <> j do
    if n mod i = 0 then j := i { i divides n, n is not prime }
    else inc(i) ;
  IsPrime := (i = h)
  end { IsPrime } ;
```

We will also count how many primes there are for each combination of digit sum and first digit. In order to get some confidence in this program, we would like to verify the results (do you know all 5-digit primes by heart; do you even know how many there are?). Therefore, we generalize the program to generate all primes with a given number ND of digits. The results for 2-digit primes can easily be verified by just looking at them. Also the 5-digit primes from the solution in the problem statement can be verified.

```
const
  ND = 5 ; { generate primes with ND digits }

var
  count: array[2..ND*9, 1..9] of integer ;
    { count[s, f] = # ND-digit primes with digit sum s and first digit f }

procedure Generate ;
  var first, i, s, f: integer ;
  begin
  first := 1 ;
  for i := 1 to pred(ND) do first := 10*first ; { first = 10^(ND-1) }
  for i := first to pred(10*first) do
```

```
    if IsPrime(i) then begin
      f := i ; s := i mod 10 ;
      while f >= 10 do begin f := f div 10 ; s := s + f mod 10 end ;
      { s = digit sum ; f = first digit }
      writeln(primes, i:ND, ' ', s:2) ;
      inc(count[s, f])
      end { if }
  end { Generate } ;
```

Program generate.pas generates the ND-digit primes in one file (e.g., primes-2.dat for ND=2 and primes-5.dat for ND=5) and the table of counts in another file (e.g., counts-2.dat for ND=2 and counts-5.dat for ND=5). From the table for ND=5 we learn that in total there are 8363 five-digit primes, and that digit sum 23 yields the most, namely 757 primes.

Generating Prime Squares

Let us introduce some notation. We number the rows of the square from top to bottom starting at 1, and the columns from left to right also starting at 1. The digit in row r and column c is denoted by S[r, c]. Here are some Pascal declarations:

```
type
  digit = 0..9 ;
  index = 1..5 ;

var
  ds: integer ; { given digit sum }
  tld: digit ; { given digit in top left-hand corner }
  S: array [index, index] of digit ; { the square }
```

We will fill and check the square systematically in twelve steps. Each step concerns a sequence of five digits, called a slot: there are five horizontal slots, five vertical slots, and two diagonal slots. During this process some positions in the square have been filled with a digit and others not yet. Whenever a new slot is considered, we fill it with all possible candidate primes from the table (one by one). Of course, such a prime has to match the digits that are already in the square. When all twelve slots have been filled with primes we have a solution. This programming technique is known as *backtracking*.

One thing that we have not discussed is the order in which we fill the slots. To reduce the number of possibilities as much as possible, we should try to fill slots for which as many digits as possible are already known, because this restricts the number of primes that will fit there. Many orders are possible. Let me introduce names for the slots as follows: H1 to H5 for the horizontal slots (rows 1 to 5), V1 to V5 for the vertical slots (columns 1 to 5), and D1 and D2 for the diagonal slots (D1 from top-left to bottom-right and D2 from bottom-left to top-right). We choose the order:

H1, V1, D2, H2, V2, H3, V3, H4, V4, H5, V5, D1

The digits are then produced in the following order:

```
 1  2  3  4  5
 6 13 14 12 15
 7 16 11 18 19
 8 10 20 22 23
 9 17 21 24 25
```

A nice property of this order is that the first digit is already known for every slot that is being filled: The top-left digit is given and it is the first digit of H1 and V1. The first digits of all (other) slots are covered by H1 and V1. Since the first digit of each slot is known, we can restrict attention to primes from the table that start with that digit. To exploit this we organize the table of primes as follows:

```
type
  number = record
    d: array [index] of digit ; { d[i] is i-th digit of v }
    v: integer { 5-digit prime with digit sum ds }
    end { number } ;

var
  n: integer ; { # primes }
  prime: array [0..800] of number ; { prime[0..n-1] filled in }
  first, last: array [digit] of integer ;
    { prime[i].d[1] = f for first[f] <= i <= last[f] }
```

The record `number' is introduced because we need to access the digits of each prime individually and also the `whole value' (the latter will become clear later on). The arrays `first' and 'last` indicate for each digit f the first and last index (in array `prime') of primes with first digit f. They are filled in when reading the primes from the file:

```
procedure ReadPrimes ;
  { read primes with digit sum ds from file }
  { pre: primes are sorted in increasing order }
  var primes: text ; p, s: integer ; i: index ; f: digit ;
  begin
  assign(primes, 'primes-5.dat') ; reset(primes) ;
  for n := 1 to 9 do begin first[n] := -1 ; last[n] := -2 end ; { empty ranges }  n :=
0 ;
  while not eof(primes) do begin
    readln(primes, p, s) ; { read a prime p and its digit sum s }
    if s = ds then with prime[n] do begin
      v := p ;
      for i := 5 downto 1 do begin d[i] := p mod 10 ; p := p div 10 end ;
      if first[d[1]] = -1 then first[d[1]] := n ;
      last[d[1]] := n ;
      inc(n)
      end { if with }
    end { while } ;
  if Test then begin
    writeln('Number of 5-digit primes with digit sum ', ds:1, ' is ', n:1) ;
    writeln('Number of these primes with first digit f:') ;
    write('  f = ') ;
    for f := 1 to 9 do write(f:4) ;
    writeln ; write('  # = ') ;
    for f := 1 to 9 do write(last[f]+1-first[f]:4) ;
    writeln
    end { if }
  end { ReadPrimes } ;
```

At the end of the procedure we write out some test data to help us verify that reading from the file works all right (the test output can be compared to the table of counts from the prime generator).

All that now remains is to fill the slots. For each slot we write a procedure that has the obligation to fill that slot in all possible ways and for each successful filling to call the procedure dealing with the next slot.

Procedure `Compute` that finds all solutions then simply becomes:

```
var
  solutions: integer ;

procedure ComputeAnswer ;
  begin
  S[1, 1] := tld ;
  solutions := 0 ;
  H1 ;
  if Test then writeln('Number of solutions is ', solutions:1)
  end { ComputeAnswer } ;
```

We will explain the procedures for some of the slots only, the others being very similar. For slot H1 we have

```
procedure H1 ;
  const R = 1 ;
  var i: integer ; c: index ;
  begin
  for i := first[tld] to last[tld] do with prime[i] do
    if d[2] <> 0 then
      if d[3] <> 0 then
        if d[4] <> 0 then begin
          for c := 2 to 5 do S[R, c] := d[c] ;
          V1
          end { if }
  end { H1 } ;
```

Only the first digit is known. All primes with this first digit are traversed by the for-loop. Only the primes without zeroes are filled in, after which V1 continues. The procedure for V1 is almost the same. For slot D2 we have

```
procedure D2 ;
  var i: integer ;
  begin
  for i := first[S[5, 1]] to last[S[5, 1]] do with prime[i] do
    if d[5] = S[1, 5] then begin
```

```
      S[4, 2] := d[2] ; S[3, 3] := d[3] ; S[2, 4] := d[4] ;
      H2
      end { if }
  end { D2 } ;
```

The first digit of slot D2 is in position S[5, 1]. For each prime with this first digit, we check whether its last digit d[5] matches the digit already present in position S[1, 5] (filled in by H1). If that is the case, the other three digits of the prime are copied into the square and H2 continues. Procedures H2, V2, H3, V3, and H4 are all very similar, only more digits are known and must be checked, whereas fewer digits are copied.
Procedure V4 is slightly different, in that all but the last digit are already known. We can simply compute the remaining digit because the digit sum is given. When this digit is computed, all that remains is to check whether the resulting number is prime. For that purpose we have introduced a function to look up a number in the table of primes. Since the table of primes is sorted on magnitude we can do a *binary search* (this is the reason why not only the digits of the primes are needed but also their `whole values'). Here is function IsPrime:

```
function IsPrime(w: integer): boolean ;
  { return: w is a 5-digit prime with digit sum ds }
  var i, j, h: integer ;
  begin
  i := 0 ; j := n ; { binary search }
  { w in prime[0..n-1].v == w in prime[i..j-1].v }
  while i <> pred(j) do begin
    h := (i+j) div 2 ;
    if prime[h].v <= w then i := h else j := h
    end { while } ;
  IsPrime := (prime[i].v = w)
  end { IsPrime } ;
```

The range of candidates can already be reduced because we know the first digit. However, the gain is not much (check the table of counts). You are challenged to try the following approach yourself and compare the execution times:

```
function IsPrime2(w: integer; f: digit): boolean ;
  { pre: f = first digit of w }
  var i, j, h: integer ;
  begin
  i := first[f] ; j := succ(last[f]) ; { binary search }
  if i >= j then IsPrime2 := false
  else { i < j } begin { w in prime[0..n-1].v == w in prime[i..j-1].v }
    while i <> pred(j) do begin
      h := (i+j) div 2 ;
      if prime[h].v <= w then i := h else j := h
      end { while } ;
    IsPrime2 := (prime[i].v = w)
    end { else }
  end { IsPrime2 } ;
```

In calls to IsPrime2 you have to supply the (known) first digit of the actual parameter for w. Note that we have to be careful: the range of primes with the required first digit may well be empty. [Story: My first program did not have the part `if i >= j then IsPrime2 := false' to guard for an empty range (because I naively modified IsPrime above). That program worked fine for all test data of the jury. I only found the mistake when trying the program for all possible input combinations (also see at the end). The first failure did not occur until digit sum 40. In all preceding cases there apparently never was the need to check primality with an empty range.] My advice: keep it simple.

Procedure V4 is now coded as follows:

```
procedure V4 ;
  const C = 4 ;
  var d, w: integer ; r: index ;
  begin
  d := ds ; w := 0 ;
  for r := 1 to 4 do begin
    d := d - S[r, C] ;
    w := 10*w + S[r, C]
    end { for } ;
  if odd(d) and (0 <= d) and (d <= 9) then
    if IsPrime(10*w+d) then begin
      S[5, C] := d ;
      H5
      end { if }
  end { V4 } ;
```

Procedure H5 is similar to V4. For slots V5 and D1 all digits are known and the only remaining task is to check primality. Here is procedure D1:

```
procedure D1 ;
  var w: integer ; rc: index ;
  begin
  w := 0 ;
  for rc := 1 to 5 do w := 10*w + S[rc, rc] ;
  if IsPrime(w) then WriteSolution
  end { D1 } ;
```

Altogether this yields **Program 1**. Once you have made some design decisions, this program is not difficult to write. There are, however, three thing that I dislike about it.

Firstly, it is easy to make a typing mistake that is very hard to find. The obligations of the twelve slot-filling procedures are crystal clear and it is just a matter of writing them. No problem. But it is easy to make a small mistake that is hard to spot afterwards: I made three typing errors. Don't kid yourself: you are not going to read those twelve procedures very carefully when under pressure. Lesson: programs that are easy to write are not necessarily easy to read. In a competition readability is important as well.

Secondly, when you do make a mistake (and this is quite likely), it may be easy to find out that there is a mistake *somewhere*. But for this program it is difficult to produce appropriate intermediate output to help diagnose errors. For instance, at each moment only a subset of the positions in the square contain valid digits (from slots already filled in). The other digits are junk, but you cannot see that by inspecting them. The knowledge of which digits are valid is encoded in the *structure* of the program, it is not encoded in its variables. Thus, it is difficult to print only the valid digits of a partially filled square.
Program 2 contains some diagnosing facilities. The main idea is to introduce an imaginary digit (undef=10) to mark an unfilled position (invalid digit). Each slot-filling procedure puts back `undef' after each attempt (it takes away the digits filled in). Procedure WriteSquare writes the valid digits of a square. This makes it possible to trace the filling process. (And even then it is still difficult to pinpoint errors.)
My third objection to Program 1 is that it is difficult to change the order in which slots are filled. The reason is that the choice of order is encoded, again, in the *structure* of the program. Of course, the slot-filling procedures can (relatively easily) be called in a different order (for instance, procedure `Compute` starts with D1, and D1 calls H1, etc.). But their obligations (and thus their program texts) also depend on that order. The filling order is quite critical in this problem. I believe that I made a reasonable choice, but you cannot always tell from the start. If you do find out that your program is too slow because you chose an inconvenient order, then you may want to change that order, without major rewrites that can introduce mistakes.

For this problem, it is rather difficult to write a program that is both general (in that it allows you to change the filling order without major program modifications) and efficient. The point is that efficiency is precisely obtained by treating each slot in an optimal way, using knowledge about what is already filled in. If you have enough time, you can write a program that given an order generates an efficient Pascal program for generating squares based on that order.

Experiments

It is instructive to experiment with slight modifications in the program. For instance, how important is it to avoid zeroes in the top row and left column (in procedures H1 and V1). That is, how much slower (or faster?) becomes the program if you do not eliminate zeroes as early as possible? [N.B. Entries first[0] and last[0] are not defined in Program 1 because they are not needed. When zeroes are not directly eliminated, these entries become necessary.]

Also experiment with different orders. For instance, the order starting

H1, V1, H2, V2, ...
is interesting because the missing center digit of D2 can then be computed. When of the remaining 8 digits, two are chosen, the six others **can be computed**. All that then remains is to check primality of eight numbers.

Another experiment to carry out is generating the table of primes inside the program instead of reading it from a file.

All solution counts

We ran the program for all possible combinations of digit sum and top left-hand digit. The results are tabulated in file **solcount.dat**. Altogether there are 2676 prime squares. Top left-hand digit 3 and digit sum 23 yield the maximum number of 152 solutions. By the way notice the regular pattern of empty bands in the table of solution counts (there are only solutions for digit sums 11 and 13, 17 and 19. 23 and 25, 29 and 31, 35 and 37). Can you explain this pattern? (Let me know if you do, because I have not given it much thought. In September 1996, I received **Mark Dettinger's explanation**. Thank you, Mark!)

Variants of this problem

Of course, you can try this problem for squares of other dimensions than 5x5. How about more than two dimensions, for example, a 5x5x5 cube? What about other number systems than base 10?

Some combinations of top left-hand digit and digit sum have many solutions. Find the ones with the fewest number of different primes in the square. What is the smallest number of different primes to make a 5x5 prime square?

Find all rotation-invariant 5-digit primes, that is, find all 5-digit primes, all of whose rotations are prime as well. The four rotations of 12345 are 23451, 34512, 45123, and 51234.

Solution 4: The Clocks

Problem Analysis

At first sight, this may look like a backtrack problem, in which the program systematically tries move sequences until one is found that turns the dials to the desired state (12 o'clock). There is, however, a difficulty in this approach.

In which order should we try the move sequences? We need to make sure that we do not miss the right one(s). If we cannot give an upper bound on the length of candidate move sequences, then we should, for instance, try them in order of increasing length. First we deal with all move sequences of length zero (only one), next all of length one (only nine), then length two, etc. This is called a *breadth-first search*. It is often more complicated then a *depth-first search* (where the move sequences are tried in some lexicographic order).

If the desired move sequence is sufficiently short, then it can be found quickly by a breadth-first search. But the problem statement did not say that all input would result in short move sequences. We have to take longer sequences into account as well.

Let us assume for a moment that we need no more than k moves of each of the nine move types. In that case the maximum length of a move sequence is 9k and there are (9k)!/((k!)^9) such maximal move sequences. For k=1 this equals 9!=3.6e5, for k=2 we get 18!/2^9=1.3e13, and for k=3 it is 27!/6^9=1.1e21. Of course, all the shorter move sequences should be considered as well. Therefore, these numbers do not look promising. For k=3 we cannot hope to try all sequences within the time limit. Even for k=2 we are hard pressed. Hoping for k=1 is wishful thinking.

We need a better idea. The effect of a move is to turn a subset of the dials over 90 degrees (clockwise). The net effect of two moves executed one after the other is cumulative. If move 1 turns clocks A and B over 90 degrees, and move 2 turns B and C over 90 degrees, then their combination turns A and C over 90 degrees and B over 2*90=180 degrees. Therefore, the net effect does not depend on the order of execution. (This resembles the *superposition principle* for waves, fields, etc. in physics.) Consequently, for determining the net effect of a move sequence on the dials, we need to know only *how many* moves of each type are involved and not *in what order* the moves are made.

The observation above drastically reduces the number of candidate move sequences to be considered. First of all, it is now obvious that each move type need not appear more than three times, because four quarter turns is the same is no turn at all. Secondly, since each of the 9 move types can appear from 0 to 3 times, the number of candidate move sequences equals 4^9=262,144. This can be accomplished in the given time.

Program 1

Program 1 tries all candidate move sequences in nine nested for-loops until a solution is found. (N.B. It was stated in the **Competition Rules** that every input file would have a solution.) Program 1 relies on the fact that doing one type of move four times is the same as doing nothing.

This program has not been optimized in any way. It shows that a rough idea may work all right. Anything you do to improve it is a waste of (your) time, since Program 1 provides the solution well within the time limit.

However, Program 1 leaves a number of interesting questions unanswered. Furthermore it is still not very efficient, even though it is fast enough for our purpose. There are two questions that the jury needed to answer in order to judge the programs:

1. Can every starting configuration be solved?
2. How many solutions (modulo 4) does a solvable starting configuration have?

Observe that the number of starting configurations is 4^9, because for each of the nine clocks there is a choice among four states. This is the same as the number of candidate move sequences for a solution. Therefore, either (i) each starting configuration has a unique solution, or (ii) there are some starting configurations without solution and some with multiple solutions.

Exercise: Modify Program 1 to generate all solutions. Because the problem statement does not restrict the input, we have assumed in Program 1 that each starting configuration has a solution, which then is unique modulo 4.

Given a move sequence it is easy to compute its net effect on the dials. Assume move type j occurs tj times in the sequence. **Figure 2** in the problem statement implicitly tells us how clock A is affected by the moves. Here is that table again in a slightly different format:

```
Move   Affected clocks

 1     A B . D E . . . .
 2     A B C . . . . . .
 3     . B C . E F . . .
 4     A . . D . . G . .
 5     . B . D E F . H .
 6     . . C . . F . . I
 7     . . . D E . G H .
 8     . . . . . . G H I
 9     . . . . E F . H I
```

We conclude that the net effect on dial A is t1+t2+t4 quarter turns. Here is a complete table giving for each clock the moves that affect it (essentially obtained by reflecting the matrix above in the upper-left-to-lower-right diagonal, also called *transposition*):

```
Clock   Affected by moves

 A      1 2 . 4 . . . . .
 B      1 2 3 . 5 . . . .
 C      . 2 3 . . 6 . . .
 D      1 . . 4 5 . 7 . .
 E      1 . 3 . 5 . 7 . 9
 F      . . 3 . 5 6 . . 9
 G      . . . 4 . . 7 8 .
 H      . . . . 5 . 7 8 9
 I      . . . . . 6 . 8 9
```

Let us denote the state of dial V before the move sequence by v and its new state after executing the sequence by v'. The new states are related to the old states by the following equations (where addition is taken modulo 4):

```
 a' = a + t1 + t2 + t4
 b' = b + t1 + t2 + t3 + t5
 c' = c + t2 + t3 + t6
 d' = d + t1 + t4 + t5 + t7
 e' = e + t1 + t3 + t5 + t7 + t9
 f' = f + t3 + t5 + t6 + t9
 g' = g + t4 + t7 + t8
 h' = h + t5 + t7 + t8 + t9
 i' = i + t6 + t8 + t9
```

Consequently, the problem our program has to solve is now translated into solving a system of nine *linear algebraic equations* with nine unknowns (t1, ..., t9), where the vector (a, ..., i) is the given initial state and the final state (a', ..., i') is the all-zero vector. All the coefficients of the unknowns are apparently either 0 or 1. We have put the coefficient matrix in text file **matrix.dat**.

Program 2

There are numerous ways to solve a system of linear algebraic equations. **Program 2** is based on a straightforward method often learned in school, called *Gauss-Jordan elimination*. It reads the coefficients from **matrix.dat**.

Note that the execution time of Program 1 depends on the actual input (in the worst case it takes 4^9 steps through the inner loop, in the best case only one). The execution time of Program 2 is almost independent of the input (procedure Solve takes on the order of 9^3=729 steps because there are three nested for-loops with at most 9 steps each). Program 2 is often faster than Program 1, but it is also more complicated and offers more opportunities for making mistakes when implementing it. Therefore, I cannot recommend it in the context of IOI'94.

Program 2 does help answer the above questions. The method it uses to solve the system of linear equations depends only in part on the input. The manipulations with the matrix A of coefficients (in procedure Solve) are independent of the input (only array b depends on the input). If the program is able to solve the problem for one particular input file, then we know that it will also succeed in solving the problem for every other input file. A simple trial run reveals that Program 2 works for the example input file **input-0.txt**. This proves that each input file has a solution and hence we know that the solutions are unique (modulo 4).

Program 3

Because the manipulations with the coefficient matrix A do not depend on the input, they can be done in a separate pre-processing phase. In particular, the coefficient matrix for the system of linear equations can be *inverted*. The inverse matrix can then be used to compute solutions very quickly. Program **invert.pas** reads the coefficients from **matrix.dat** and writes the inverse matrix to the text file **inverse.dat**. It is obtained from Program 2 by replacing the linear array b in procedure Solve by a square array B initialized to the identity matrix.

Program 3 uses this inverse matrix to compute a solution by a single matrix-vector multiplication (which takes on the order of 9^2=81 steps). This solution is extremely fast. It would be even faster if the coefficients would not be read from a file but were incorporated in the program to initialize the matrix. In Turbo Pascal this is not so difficult, but for other Pascal versions that may require 81 assignment statements. In fact, if you strip the program to the bone (you don't have to use an array) you can squeeze it into a few lines. Here is a solution to the above system of nine equations in (t1,...,t9) for given (a,...,i) and (a',...,i')=(0,...,0) derived from the inverse matrix (addition and multiplication are modulo 4):

```
 t1 = 8+ a+2b+ c+2d+2e -f+ g -h
 t2 =    a+ b+ c+ d+ e+ f+2g+   2i
 t3 = 8+ a+2b+ c -d+2e+2f    -h+ i
 t4 =    a+ b+2c+ d+ e+    g+ h+2i
 t5 = 4+ a+2b+ c+2d -e+2f+ g+2h+ i
 t6 =   2a+ b+ c+    e+ f+2g+ h+ i
 t7 = 8+ a -b+   2d+2e -f+ g+2h+ i
 t8 =   2a+   2c+ d+ e+ f+ g+ h+ i
 t9 = 8    -b+ c -d+2e+2f+ g+2h+ i
```
Program 4, based on this solution, may be instructive but it is not a recommended IOI practice.

Variants of this problem

This problem derives from Rubik's clock puzzle. In Rubik's puzzle all clocks have 12 states. Each move sets the (affected) clocks one hour forward (clockwise). But there are also the inverse moves that set the (affected) clocks backward one hour (counterclockwise).

Solve the problem for Rubik's (12-hour) clock puzzle, minimizing the number of moves (a backward move counts as one move).

Solution 5: The Buses

Problem Analysis

Bus routes

A bus route is characterized by the time of its first arrival at the bus stop (in minutes after 12:00) and its interval (number of minutes between successive stops). These two numbers determine how often a bus route stops at the observed bus stop from 12:00 to 12:59. Because this number of stops plays an important role, it will be included with the other information to describe a bus route. Here is the definition of type `BusRoute`:

```
type
  BusRoute = record
    first   : 0..29;
    interval: 1..59; { in fact, first < interval <= 59 - first }
    howoften: 2..60; { howoften = 1 + (59 - first) div interval }
    end;
```

The lower bound on `first` is zero by definition. The upper bound and the bounds on `interval` are less straighforward and we will now explain them.

Observe that the **problem statement** implies `first < interval`, since buses are known to arrive *throughout the entire hour*. If `first >= interval`, then there would have been a stop earlier than `first` at time `first-interval >= 0`, which contradicts the definition of `first`. Also observe that the second bus arrives at time `first+interval` and since buses are known to stop at least twice we therefore have `first + interval <= 59`. This explains the bounds on `interval`. The upper bound on `first` can be obtained by adding the inequalities for `first`:

```
  first <       interval
  first <= 59 - interval
------------------------ +
2*first <  59
```

from which we infer `first <= 29`. The number `howoften` is determined by the two inequalities:

```
first + (howoften-1)*interval <= 59
first +  howoften   *interval >  59
```

These can be rewritten into

```
59-first - interval < (howoften-1) * interval <= 59-first
```

from which we infer `howoften = 1 + (59-first) div interval`. So much for the bounds. Question: How many bus routes are there?

Here is procedure for writing a bus route in a graphically appealing way. It is useful during devopment and will help understand the problem better.

```
procedure GraphBusRoute(var f: text; b: BusRoute);
  var i: integer;
  begin
  with b do begin
    write(f, 1:first+1) ;
    i := first + interval ;
    while (i <= 59) do begin
      write(f, 1:interval) ;
      i := i + interval
      end { while } ;
    write(f, ' ':62-i+interval) ;
    writeln(f, '[', first:2, ',', interval:2, ',', howoften:2, ']')
    end { with }
  end { GraphBusRoute } ;
```

`GraphBusRoute` writes a tally for each arrival of the bus route. The locations of the tallies on the output line correspond to the arrival times. At the end of the line the parameters are written. The three bus routes in the schedule appearing in the **problem statement** are shown by `GraphBusRoute` (three calls) as follows (the first two lines with time labels are produced by `WriteTimes`, which is too simple to include here):

```
000000000011111111112222222222333333333344444444445555555555
012345678901234567890123456789012345678901234567890123456789
1            1            1            1            1          [ 0,13, 5]
   1           1           1           1           1           [ 3,12, 5]
     1       1       1       1       1       1       1         [ 5, 8, 7]
```

The input data

The input data is a sorted list of arrival times, possibly containing duplicates. These are most conveniently stored by counting for each arrival time how often it occurs. For this purpose we introduce variables `s` and `a`:

```
var
  s: integer; { s = # unaccounted arrivals = sum a[0..59] }
  a: array[0..59] of integer; { a[t] = # unaccounted arrivals at time t }
```

Procedure `GraphUnaccounted` (listing not included) shows the arrival times in the same format as `GraphBusRoute`, except that now the tallies may take on values from 0 upward (0 is displayed as a space, and numbers above 9 are displayed as letters from A upward). The input for the example with 17 arrival times in the **problem statement** would be written as

```
000000000011111111112222222222333333333344444444445555555555
012345678901234567890123456789012345678901234567890123456789
1  1 1       2 1     1    11 1       1 2     1     111         total = 17
```

Compare this to the graphs of the bus routes shown above. The three rows of the bus routes nicely add up to the row of unaccounted arrival times in the input.

The input is read from file `inp` by procedure `ReadInput`:

```
procedure ReadInput;
  { read input into s and a }
  var i, j: integer;
  begin
  if Test then writeln('Reading input') ;
  readln(inp, s) ;
  if Test then writeln('Number of stops = ', s:1) ;
  for i:=0 to 59 do a[i] := 0 ;
  for i:=1 to s do begin
    read(inp, j) ;
    inc(a[j])
    end { for i } ;
  readln(inp) ;
  if Test then begin GraphUnaccounted ; writeln end
  end { ReadInput } ;
```

The following function `Fits` determines whether a given bus route `b` fits with the arrivals `a`, that is, whether all stops of `b` occur in `a`:

```
function Fits(b: BusRoute): boolean;
  { check whether b fits with a, that is, all arrivals of b occur in a }
  { global: a }
  var i, j: integer;
  begin
  with b do begin
    i := first ; j := 60 ;
    { bounded linear search for earliest a[first + k*interval] = 0 }
    while i < j do
      if a[i] <> 0 then i := i+interval
      else j := i ;
    Fits := (i >= 60)
    end { with }
  end { Fits } ;
```

Finding candidate bus routes

We will first make a list of all bus routes that fit with the arrival times in the input. These are called candidate bus routes. Observe that the total number of possible bus routes equals the number of pairs `(first,interval)` with `0 <= first <= 29` and `first+1 <= interval <= 59-first`, which is 59+57+...+3+1 = 60*30/2 = 900.

Candidate bus routes will be stored in global array `c` and counted in integer `n`:

```
var
  n: integer; { # candidate bus routes }
  c: array[0..899] of BusRoute; { c[0..n-1] are candidate bus routes }
```

Procedure `FindBusRoutes` determines the candidate bus routes that fit with the given arrival times `a`:

```
procedure FindBusRoutes;
  { post: c[0..n-1] are all bus routes fitting with a }
  { global: a, n, c }
  var f, i: integer;
  begin
  if Test then begin
    writeln('Finding candidate bus routes') ;
    WriteTimes
    end { if } ;
  n := 0 ;
  for f:=0 to 29 do begin
    if a[f] <> 0 then begin
      for i:=f+1 to 59-f do begin
        with c[n] do begin
          first := f ;
          interval := i ;
          howoften := 1 + (59 - f) div i
          end { with c[n] } ;
        if Fits(c[n]) then begin { another candidate }
          if Test then GraphBusRoute(c[n]) ;
          inc(n)
          end { if }
        end { for i }
      end { if }
    end { for f } ;
  if Test then
    writeln('Number of candidate bus routes = ', n:1)
  end { FindBusRoutes } ;
```

Procedure `FindBusRoutes` is quite straightforward. As usual we have included some diagnostic output. A few things might need further explanation.

First of all, the check `if a[f] <> 0` was inserted to cut off impossible bus routes as early as possible (otherwise, for values of `f` with `a[f]=0`, all possible values of `i` would be tried in vain).
Second, one might be tempted to optimize a little more. For instance, the computation of `howoften` could have been done only if `Fits(c[n])` turned out successful. Also, the function `Fits` could be adapted to exploit the fact that `a[f] <> 0` was already tested, by starting `Fits` with `i := first+interval` instead of `i := first`. The main reasons for not doing so are that these changes do not speed up things considerably (try it), and that they may complicate later uses or changes of `Fits` (such as using `howoften` in `Fits`).
As an example of `FindBusRoutes` consider the **diagnostic output** produced when reading the example input from the **problem statement** and finding the candidate bus routes. The 17 stops of this input give rise to 42 candidate bus routes, of which only eight stop more than twice.

Here is an overview of the number of candidate bus routes in each test:

test case	number of arrivals	number of candidates
0	17	42
1	12	24
2	44	237
3	43	375

```
  4      31        136
  5      40        201
  6      70        365
```

Finding schedules

A schedule can be described as a list of bus routes (at most 17 according to the **problem statement**):

```
type
  Schedule = array [0..16] of BusRoute;
```

A schedule is written by the following procedure:

```
procedure WriteSchedule(var f: text; sc: Schedule; len: integer);
  var i: integer;
  begin
  for i:=0 to len-1 do with sc[i] do
    writeln(f, first:2, ' ', interval:2) ;
  if Test then writeln(f, '-----')
  end { WriteSchedule } ;
```

Using the candidate bus routes we can do straighforward *backtracking* to determine bus schedules that exactly account for the given arrival times. We are only interested in a bus schedule with as few bus routes as possible. For that purpose we introduce some global variables:

```
var
  t: longint; { # schedules found so far }
  freq: array [1..17] of longint; { freq[p] = # schedules with p bus routes }
  p: integer; { # buses in partial schedule so far }
  m: integer; { # buses in best schedule so far }
  sched: Schedule; { sched[0..p-1] is schedule so far }
  best: Schedule; { best[0..m-1] is best schedule so far }
```

Variables `t` and `freq` are for diagnostic purposes only.

Note that, according to the **problem statement**, the order of bus routes in a schedule is irrelevant (``the order of the bus routes does not matter'') and that a bus route may occur more than once (``buses from different routes may arrive at the same time''). To avoid duplication of work we will put bus routes in a schedule in the same order as they appear in the list of candidates and we allow multiple occurrences of the same bus route.
The state of the backtracking process is captured by the variables `s`, `a`, `p`, and `sched`. The bus routes `sched[0..p-1]` account for part of the arrival times, and the unaccounted arrival times are represented by `a` (and `s`). Procedure `Use` extends the current partial schedule with a given bus route and updates the state variables:

```
procedure Use(b: BusRoute);
  { global: s, a, p, sched }
  var i: integer;
  begin
  sched[p] := b ;
  inc(p) ;
  with b do begin
    i := first ;
    while (i <= 59) do begin
      dec(a[i]) ;
      i := i+interval
      end { while } ;
    s := s - howoften
    end { with } ;
  if Trace then begin
    WriteSchedule(output, sched, p) ;
    GraphUnaccounted(output)
    end { if }
  end { Use } ;
```

Similarly, procedure `RemoveLast` removes the last bus route that was used in the current partial schedule:

```
procedure RemoveLast;
  { global: s, a, p, sched }
  var i: integer;
  begin
  dec(p) ;
  with sched[p] do begin
    i := first ;
    while (i <= 59) do begin
      inc(a[i]) ;
      i := i+interval
      end { while } ;
    s := s + howoften
    end { with }
  end { Remove } ;
```

The recursive procedure `FindSchedules` generates all schedules (with at most 17 bus routes):

```
procedure FindSchedules(k: integer);
  { global: s, a, n, c, p, sched, m, best, t, freq }
  { Find all schedules sched[0..r-1] with p <= r <= 17 such that
    bus routes sched[0..p-1] are as given,
    sched[p..r-1] accounts for a and uses only bus routes from c[k..n-1] }
  begin
  if s = 0 then { nothing left to account for }
    ScheduleFound
  else if p = 17 then { too many bus routes: ignore }
  else { try each candidate c[k..n-1] that fits }
    while k < n do begin
      if Fits(c[k]) then begin
        Use(c[k]) ;
        FindSchedules(k) ;
        RemoveLast
        end { if } ;
      inc(k)
      end { while }
  end { FindSchedules } ;
```

`FindSchedules` is called as follows in procedure `ComputeAnswer`:

```
procedure ComputeAnswer;
  begin
  FindBusRoutes ;
  if Test then writeln('Finding schedules') ;
  for p:=1 to 16 do freq[p] := 0 ;
  t := 0 ; p := 0 ; m := 18 ;
  FindSchedules(0) ;
  if Test then begin
    writeln('Number of schedules = ', t:1) ;
    WriteFrequencies(out)
    end { if }
  end { ComputeAnswer } ;
```

For the 17 arrival times in the **problem statement**, `FindSchedules` produces 18 schedules, as shown by the **diagnostic output**. The shortest has three bus routes and is unique, the longest (of which there are three) has seven bus routes.

This method is incorporated in **Program 1**. It is too slow for the **second test input** with 44 arrival times. **Program 1** quickly finds a schedule with four bus routes (the minimum) but then continues to look for (non-existing) improvements. Apparently there are many (longer) schedules for this test case.

Program 2

One way of improving Program 1 is by cutting off the search for schedules in a `corner' of the search space where it is impossible to find improvements of the best schedule so far. More precisely, if `p` is the number of bus routes in the current partial schedule, then we will see to it that `p < m` holds invariantly. If `s=0` then we have a schedule that is also an improvement of the best schedule so far. If, however, `s<>0` then we can stop searching in this corner if `p+1=m` since at least one more bus route is needed to complete the schedule, and therefore it will never result in an improvement. Here is the adapted code:

```
procedure FindBestSchedule(k: integer);
  { global: s, a, n, c, p, sched, m, best }
  { Find all schedules sched[0..r-1] with p <= r < m such that
    bus routes sched[0..p-1] are as given,
    sched[p..r-1] accounts for a and uses only bus routes from c[k..n-1] }
  { pre: p < m }
  begin
  if s = 0 then { nothing left to account for }
    ScheduleFound
  else { try all candidates c[k..n-1] that fit }
    while (k < n) and (p+1 <> m) do begin
      if Fits(c[k]) then begin
        Use(c[k]) ;
        FindBestSchedule(k) ;
        RemoveLast
        end { if } ;
      inc(k)
      end { while }
  end { FindBestSchedule } ;
```
Note that we have not written

```
  else if p+1 = m then { too many bus routes: ignore }
  else { try all candidates c[k..n-1] that fit }
    while k < n do begin
```
because `m` may be changed inside the while-loop by the recursive call to `FindBestSchedule`.

This idea is incorporated in **Program 2**. This program indeed only tries one schedule for **input-1.txt** and **input-2.txt**. However, on the other input files it still takes (too) long.

Program 3

Yet another idea that might help improve performance is based on reordering the list of candidate bus routes. For Program 1, the order of the candidate bus routes does not matter, since this program generates *all* schedules. Using a different order of candidate buses simply means that all schedules are found in a different order.

Program 2 might benefit from another order for the candidates, because if it finds a good schedule early on, then the built-in cut-off mechanism is more efficient. Since we are interested in schedules with the fewest bus routes, each bus route in the schedule should account for as many arrivals as possible. Therefore, we might first try bus routes that make many stops.

Program 3 sorts the candidate bus routes on `howoften` as they are found. The **diagnostic output** for the input in the **problem statement** shows the sorted list of 42 candidate bus routes. Sorting turns out to make only a small difference. Program 3 still takes too long for **input-3.txt**.

Program 4

Programs 2 and 3 aimed at reducing the number of schedules investigated. We can also directly aim at reducing the number of *partial* An adapted procedure `FindBestSchedule` is here:

```
procedure FindBestSchedule(k: integer);
  { global: s, a, n, c, p, sched, m, best }
  { Find all schedules sched[0..r-1] with p <= r < m such that
    bus routes sched[0..p-1] are as given,
    sched[p..r-1] accounts for a and uses only bus routes from c[k..n-1] }
  { pre: p < m }
  begin
  if s = 0 then { nothing left to account for }
    ScheduleFound
  else begin { try all candidates c[k..n-1] that fit }
    while (k < n) {c}and (c[k].howoften > s) do inc(k) ;
    while (k < n) {c}and (p + 1 + (s-1) div c[k].howoften < m) do begin
      if Fits(c[k]) then begin
        Use(c[k]) ;
        FindBestSchedule(k) ;
        RemoveLast
        end { if } ;
      inc(k)
      end { while }
    end { else }
  end { FindBestSchedule } ;
```
The first while-loop skips candidate bus route that make too many stops for the remaining unaccounted arrival times. The second while-loop breaks off as soon as the remaining candidate bus routes make so few stops that improvement is no longer possible. Note that I have written `{c}and` to remind myself (and you) that this conjunction is *conditional* (using short-circuit boolean evaluation). That is, if the first conjunct already evaluates to false then the second conjunct is not evaluated (in our case it is then even undefined).

Observe that this technique only works if the list of candidate bus routes is sorted on `howoften`. In that case a *lower bound* can be given on the number of bus routes needed to complete the schedule. The technique is sometimes called *branch-and-bound*. It is used in **Program 4**, which is so effective that all six input files are done in an instant. For all of them only one or two (complete) schedules are considered.

Variants of this problem

What changes should be made to the programs if all bus routes in a schedule should be *different*? What about generating *all* minimal schedules?

Write an auxiliary program that generates all bus routes, sorts them on how often they stop, and then puts this data in a file `routes.dat'. Investigate whether using this file, instead of generating them inside the main program, can speed up the generation of all candidates.

Solution 6: The Circle

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.