IOI'94 - Day 2 - Solution 2: The Buses

[Introduction] [Problem Statement] [Test Data]

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.

Tom Verhoeff
Eindhoven University of Technology