Analysis of Task PACK

The general strategy will be: try all 'interesting' enclosing rectangles, and save the champions. In pseudocode:

```
for all interesting rectangles (x,y) do
  process(x,y)
putout
```

The procedure `process` tries to put the pair of sides found into a data object. This object consists of the value `minarea` (the minimum area found so far), and a sorted list of different rectangles (pairs of sides) having this area.

```
procedure process(x, y: sides)
   if x * y > minarea then
      ignore x and y
   else
     if x * y < minarea then
       make list empty
       minarea := x * y
       insert (x, y)
     else
       insert(x, y)
```

The procedure `insert` puts the pair (x, y) in the right order and in the right place into the sorted list of pairs in the database (if it was already there it is ignored).

The procedure `putout` puts the contents of the data object in the right format into the file `output.txt`.

The procedures `insert` and `putout` are fairly standard procedures.

The interesting part is the implementation of:

```
for all interesting rectangles (x,y) do ...
```

A very naive approach

Let M be the sum of the longest sides of the four rectangles. Evidently a solution should fit in a $M \times M$ rectangle, and $M \leq 200$.

We may iteratively try each of the four rectangles in each place:

```
for h := 1 to M do
  for v := 1 to M do
    put rectangle[m] in place (h, v)
```

Of course every rectangle should be placed once horizontally, and once vertically. For every layout we must find out whether there are no intersections. If that is the case we calculate the sides of the enclosing rectangle (call these x and y) and do $process(x, y)$.

As we have to try all combinations we have to try $16 * M^8$ possibilities. This consumes too much time for larger values of M. (Recall that $M = 200$ may occur.)

A naive approach

The algorithm of the preceding paragraph may be improved in several ways:

- Do not place the third rectangle if the first two are not disjoint. (This comes down to backtracking). Likewise do not place the fourth rectangle if the first three are not disjoint.
- Do not place the next rectangle if the area of the current enclosing rectangle is larger than $minarea$.
- In one direction, e.g. horizontal: if a rectangle is not disjoint with one of its predecessors, jump over it, until they are disjoint.

This changes the behaviour to $O(M^{4+\epsilon})$. (Experimentally we found that the algorithm took 20 times longer when we doubled the sides of the rectangles). This will still consume too much time for larger values of M.

A fast solution

The information in Figure 1 tells us that there are essentially 6 basic layout patterns. Using this information we can find a solution running in constant time.

A given set of rectangles can be put in several ways into one of the basic layout patterns. First of all we have to decide which rectangle will be put in which place. This comes down to a permutation of the rectangles, so it can be done in $4! = 24$ ways. Next we have to choose an orientation for every rectangle, that is decide whether its longest side is 'lying' or 'standing'. We have to make this choice four times. This gives rise to 16 possible (combinations of) orientations. For each of the basic layout patterns we now have $24 * 16 = 384$ ways to place a given set of rectangles.

Given one of the basic layout patterns and given the sides of the rectangles we can easily calculate the sides of the minimal enclosing rectangle.

Let the sides of the rectangles be: x_1, y_1; x_2, y_2; x_3, y_3; x_4, y_4. (x are the horizontal sides, y the vertical sides). Further assume that the rectangles are placed in the first pattern from left to right, that is, rectangle 1 is the leftmost, and rectangle 4 is the rightmost. (It turns out that the order of the rectangles is irrelevant in this pattern, but that is not true in general). Then the height of the minimal enclosing rectangle equals:

$$max4(y1, y2, y3, y4)$$

whereas the width equals:

$$x1 + x2 + x3 + x4$$

(here $max4$ is a function returning the maximum of its four arguments).

In the same way we can write down expressions for the sides of the enclosing rectangles from the other five basic layouts. One easily checks that the fourth and the fifth layout give rise to the same expressions for the sides. This leaves us with five expression-pairs, each to be filled in 384 ways giving $5 * 384 = 1920$ pairs of numbers.

Calculating all these pairs is certainly not beyond the power of today's computers.

Next we have to calculate the minimal area (product) from these 1920, throw away all pairs with larger products, shuffling the remaining pairs in the required format and output the result. So we get the following global algorithm:

```
read rectangles
for all (24) permutations do
  for all (16) orientations do
    for all (5) layouts do
      calculate the sides of the enclosing rectangle
      process these sides
```

We now give (part of) an implementation in Pascal, in a more or less informal style.

```
type
  rectangle  = array[1..2] of integer;
  rectangles = array[1..4] of rectangle;
```

Implementation of the following procedures is left to the reader.

```
procedure readrectangles(var rects: rectangles);
procedure swaprectangles(var r1,r2 : rectangle);
procedure swapinteger(var i1, i2: integer);
```

An easy way to go through all permutations is recursively: successively put each of the rectangles on the fourth place and permute the remaining three rectangles, etc.

```
procedure permute(k: integer; rect: rectangles);
  var i : integer;
```

```
  begin
    if k = 0 then
      orient(4, rect)
    else
      for i := k downto 1 do
      begin
        swaprectangles(rect[i], rect[k]);
        permute(k - 1, rect);
      end;
  end;
```

The main program is simple:

```
var rect: rectangles;
begin
  { do some initialization }
  readrectangles(rect);
  permute(4, rect);
  { output result }
end.
```

The procedure `orient` (called from permute) goes recursively through all orientations:

```
procedure orient(k: integer; rect: rectangles);
  begin
    if k = 0 then
      layouts(rect)
    else
    begin
      orient(k-1, rect);
      swapinteger(rect[k][1],rect[k][2]);
      orient(k-1,rect)
    end;
  end;
```

For all permutations and all orientations we call the procedure layouts, which tries all layouts:

```
function min(x, y: integer): integer;
function max(x, y: integer): integer;
function max3(x, y, z: integer): integer;

procedure layouts(var rect: rectangles);
  var
```

```
    h,  v  : integer;
    x1, y1,
    x2, y2,
    x3, y3,
    x4, y4 : integer;
  begin
    x1 := rect[1][1]; y1 := rect[1][2];
    x2 := rect[2][1]; y2 := rect[2][2];
    x3 := rect[3][1]; y3 := rect[3][2];
    x4 := rect[4][1]; y4 := rect[4][2];

    h := x1 + x2 + x3 + x4;
    v := max(max(y1, y2), max(y3, y4));
    process(h, v);

    h := max(x1 + x2 + x3, x4);
    v := max3(y1, y2, y3) + y4;
    process(h, v);

    h := max(x1 + x2, x4) + x3;
    v := max(max(y1, y2) + y4, y3);
    process(h, v);

    h := x1 + x4 + max(x3, x2);
    v := max3(y1, y4, y2 + y3);
    process(h, v);

    h := max3(x1 + x2, x2 + x3, x3 + x4);
    v := max3(y1 + y3, y1 + y4, y2 + y4);
    process(h, v);
  end;
```

In fact all local variables in the procedure `layouts` are superfluous. For example we could write the formulae for the first layout as:

```
process( rect[1][1] + rect[2][1] + rect[3][1] + rect[4][1],
         max(max(rect[1][2], rect[2][2]), max(rect[3][2], rect[4][2])));
```

We consider this expression unreadable, however.

The procedure `process` adds the pair (h, v) to the scorelist. The scorelist is a global object.

```
var scorelist = record
                  minarea    : integer;
```

```
      numofrects : integer;
      rects      : array[1..1920] of rectangle;
    end;
```

The size 1920 for the list of rectangles is sufficient, but exaggerated. We argued already that the first layout does not generate 384 but at most 16 different pairs, because it is not changed by permuting the rectangles. If we consider the formulae for the other layouts carefully we find that at most 356 different pairs are possible. Of course we save only rectangles having the (current) minimal area. So far we were unable to find a problem having more than three solutions[1]. Initialization of the scorelist is simple:

```
with scorelist do
begin
  minarea := maxint;
  numofrects := 0
end;

procedure process(h, v: integer);
  var area, long, short : integer;
  begin
    long  := max(h,v);
    short := min(h,v);
    area  := long * short;
    with scorelist do
      if area < minarea then { new champ !}
      begin
        minarea := area;
        numofrects := 1;
        rects[1][1] := short;
        rects[1][2] := long;
      end
      else
        if area = minarea then
          insert(short, long);
  end;
```

[1]I would greatly appreciate motivated answers to the following questions:

1. How long is the longest list of solutions a problem may have?
2. How long is the longest list belonging to a certain (not necessarily the minimal) area?
3. If we allow reals for the sides of the rectangles, does that changes these figures?

The procedure `insert` must put the (ordered) pair in its proper place in the list. If it is already in the list it should be dismissed. We must keep in mind that the proper place might be beyond the last on. We use a sentinel for that case.

```
procedure insert(short, long: integer);
  var k, j: integer;
  begin
    with scorelist do
    begin
      rects[numofrects + 1][1] := maxint; {sentinel}
      k := 1;
      while rects[k][1] < short do
        k := k + 1;
      { now rects[k][1] >= short, sentinel assures termination }
      { if rects[k][1] = short : do nothing, the pair is already there }
      if rects[k][1] > short then
      begin
        for j := numofrects downto k do
          rects[j + 1] := rects[j];
        rects[k][1] := short;
        rects[k][2] := long;
        numofrects  := numofrects + 1;
      end;
    end;
  end;
```

Another solution

One may try to generate the basic layout patterns without using Fig. 1.

Consider the Euclidian plane, supplied with cartesian coordinates. Consider two rectangles in the plane, having their sides parallel with the axis. These rectangles are disjoint if and only if in at least one direction they have disjoint projections. In other words, the rectangles A and B are disjoint iff:

- A is at the right-hand side of B (that is: the projection of A on the first axis lies at the right hand side of B's projection), or
- B lies at the right-hand side of A, or
- A lies 'higher' than B (that is ...) or
- B lies lower than A.

Every pair of rectangles (there are 6 pairs) must have one of these relations. This gives $4^6 = 2^{12} = 4096$ possibilities. Some of these are impossible, however (e.g. A lies at the right-hand side of B, B l.a.t.r.h.s. of C and C l.a.t.r.h.s. of A). (After removing impossibilities, and redundancies caused by reflection, rotation and permutation of the rectangles we obtain the six basic patterns of Fig. 1 in the task).

Again, 4 rectangles together have 16 different orientations, so we have to go through 65786 cases. This can be handled within the time limit.

Peter Kluit
Scientific Committee IOI'95