Problem Notes

General Comments

The problems used in the test can be classified in four different classes, according to their difficulty level:

- **Easy**: These are very easy problems. We suppose they can be solved in an hour by a first or second year student.
- **Easy/Intermediate**: This class is formed by problems that do not need much technical knowledge, but involve some reasoning and good skills in data structures. We suppose a third year student can solve them in less than an hour.
- **Intermediate**: Intermediate problems involve some special data structure, algorithm or algorithm design technique, like graph algorithms and dynamic programming. An experienced contestant is supposed to solve them in an hour.
- **Hard**: Problems that involve advanced data structures and algorithm design techniques. We suppose an experienced contestant can solve it during the contest time.

We have also distributed the problems among different flavors, involving data structures, graphs, geometry, dynamic programming. Moreover, we avoided problems with extremely large statements or whose solution needed loads of lines of code. We ensure that no problem require a source file with more than 200 lines.

Problem A - Rubik Cube

Level: Easy/Intermediate

A simple approach consists of modeling the cube as six 3x3 matrices of letters, representing each face of it. So, you may write simple procedures for each movement. Notice that a counterclockwise rotation is equivalent to three clockwise rotations, which may save some lines in your algorithm.

Problem B - This Sentence is False

Level: Easy/Intermediate

The solution for this problem is very intuitive. Each sentence may be seen as a vertex in a graph. If sentence *i* is about sentence *j*, create an edge from vertex *i* to vertex *j* using as its label what *i* says about *j* (if it is true or false). It is worth seeing that this graph will have various components, each one composed of a unique cycle and some "tails" formed by vertices which are connected to the cycle but are not part of it. In each component, once you define one of its vertices (sentences) as true or false, all the other sentences of the component can have a value assigned. If a vertex *i* has value *v*, each vertex *j* connected to it by an outgoing edge (from *i*'s viewpoint) will have value *(v == label(i,j))*, where *label(i,j)* refers to the

label of the edge which connects vertex *i* to vertex *j*; and each vertex *k* connected to it by an incoming edge will have value *(v == label(k,i))*, where *label(k,i)* refers to the label of the edge which connects vertex *k* to vertex *i*. By this principle, two things may be inferred:

- An instance with a cycle containing an odd number of "false" edges is inconsistent. (Because every vertex in the cycle has its veracity implying its falsity)
- There are exactly two possible assignments in a consistent graph component. If a sentence is true (false) in one assignment, it is false (true) in the other one.

Based on this information, many algorithms can be developed to solve the problem. A simple approach follows:

```
{ This algorithm solves only one instance }
create the graph
separete each component of it
inconsistent := false
solution := 0
for each component do
     if the component's cycle has an odd number of false edges then
          inconsitent := true
          { If you want, you may break the for here }
     else
          Assign the value true to one vertex in the component
          Calculate the values of the other vertices
          Let tv be the number of vertices labelled true
          Let fv be the number of vertices labelled false
          solution := solution + max(tv,fv)
     end
end
if inconsistent then
     print "Inconsistent"
else
     print solution
end
```

The previous algorithm might be implemented using a simple depth-first search, requiring only O(n) time and space.

Problem C - Will Indiana Jones Get There?

Level: Intermediate

A simple approach consists of constructing a complete undirected graph where each vertex corresponds to a wall section. Every edge *(i,j)* has a cost *c(i,j)* associated, which is 0 when the two segments intersect or corresponds to the distance between the two segments if they do not intersect. After that, you may execute an algorithm very similar to Dijkstra's shortest path algorithm.

Maintain a set *S*, initially empty, of vertices whose minimum size wooden board necessary to get to it has already been calculated. Create an array board[N] which contains the value of the smallest board Indiana will need to get to every wall section passing only through the wall sections in S. Let *s* be the vertex which refers to the wall section on which Indiana stand. During initialization, make board[*s*] equal to 0 and board[*i*] equal to INFINITE, for every *i* different from *s*.

In every step, select a vertex *i* which is not in S and whose value board[*i*] is minimal. **Add *i* to S**. It is worth seeing that if board[*i*] is minimal, then it will need a bigger wooden board to get to *i* through another vertex not in S. So, the best way to get to vertex *i* is using the wooden board previously

calculated. Now that board[*i*] is fixed, we can recalculate board[*j*], for all *j* not in S, testing the best path through *i*. So, for all *j* not in S, execute the following line:

```
board[j] = min(board[j],max(board[i],c(i,j)))
```

When the vertex representing the wall the trapped person stands on is included in S, you may stop executing the algorithm and print out board[*i*]. This simple approach runs in O(n^2) time and uses O(n) space (once *c(i,j)* can be calculated only when it is required).

Problem D - Duty Free Shop

Level: Easy/Intermediate

This is a variation of the classical Knapsack problem which is found in many books. A common strategy consists of maintaining an array V of size M+1 (range 0..M), representing the number of Mindt chocolates that can be grouped in small boxes. V[*j*] is 0 if it is not possible to group *j* Mindt chocolates in small boxes and it is *1<=k<=N* if it is possible to group *j* Mindt chocolates in small boxes, being the *k*-th small box (in the order given by the input) the last one used. Initially, make V[0] equal to -1 to indicate that 0 Mindt chocolates can be boxed without any small box.

On step *i*, take the *i*-th small box and let *vi* be its capacity. Then, execute the following:

```
for j := M downto vi do
     if (V[j] == 0) and (V[j-vi] <> 0) then
          V[j] := i
     end
end
```

The **for** goes backwards in order to avoid using the same small box more than once in a single step. By induction, you see that after step *i*, V[*j*] is 0 if it is not possible to group *j* Mindt chocolates using some of the *i* first small boxes and it is *1<=k<=i* if it is possible to group *j* Mindt chocolates using some of the *i* first small boxes, being the *k*-th small box the last one used. V[0] is the only exception for this rule, once its -1 value means that no small box is used. So, after N steps V[*j*] <> 0 if, and only if, there is a subset of the small boxes whose sum is *j* and, thus, can box *j* Mindt chocolates (once *0<=j<=M*).

After that, it is very easy to check if it is possible to distribute the two brands of chocolates among the small boxes. Let S be the sum of the capacities of all the small boxes; check if there is a value *j* such that V[*j*] <> 0 and S - *j* < L. If there is no such *j*, it is impossible to distribute the chocolates. Otherwise, the array V has enough information to print a subset of boxes whose sum is *j*. Starting from *j*, push V[*j*] in a stack. Let *i* be the small box indicated by V[*j*] and let *vi* be its value; subtract *vi* from *j* and continue this procedure until *j* becomes 0 (zero). At this moment, print the size of the stack and pop every element of it, printing it one by one. This can be done very easily using a recursive procedure.

This solution takes O(M*N) time and needs O(M+N) space.

Problem E - Not Too Convex Hull

Level: Hard

This is the most difficult problem in the problem set. It involves geometry and dynamic programming. A possible approach starts ordering (clockwise or counterclockwise) the nails around the Origin. Let's say that N is the number of points without the Origin (actually, in the problem statement, N includes the Origin). So, after the ordering, we have defined points 1..N. Define a circular queue of the points, so that

when we refer to points 4 .. 2 in a set of 6 points, we mean a working set composed of points 4, 5, 6, 1 and 2. After that, it is necessary to calculate a matrix CH[N][N], where CH[*i*][*j*] corresponds to the area used by a single rubber band wrapping points *i* ... *j* (in the order defined before). If the angle formed by points *i* and *j* with the Origin is bigger than PI (in radians), define CH[*i*][*j*] as INFINITE because a rubber band involving them does not touch the Origin. Also define CH[*i*][*j*] as INFINITE if *i* == *j* because a wrapping containing only two nails is invalid. Otherwise CH[*i*][*j*] can be calculated using any convex hull algorithm together with some area calculation algorithm. A good approach would use a variation of the Graham's Scan algorithm, using the Origin as pivot in such a way that an entire line CH[*i*] can be calculated in O(N) time. So, calculating the entire matrix CH would take O(N^2) time.

Let NTCH(*r*, *i*, *j*) be the total area of the best wrapping of *r* rubber bands involving points *i* ... *j* (in the order defined before). The following recurrence defines NTCH recursively and can be used to develop an efficient algorithm to solve the problem. A correctness proof for this recurrence is straightforward.

```
NTCH(1,i,j) = CH[i][j]

NTCH(r,i,j) = INFINITE { if i == j }
              min( k := i to j-1, NTCH(r-1,i,k) + CH[k+1][j]) ) { otherwise }
```

In a first step, create a matrix M[N][N] and make it equal to CH. In a step *r*, this matrix M represents NTCH(*r*). As it is seen in the recurrence above, you need only NTCH(*r*-1) and CH to construct NTCH(*r*). So, let MA represent the matrix calculated in the previous step, to calculate M you only need to follow the recurrence using MA instead of NTCH(*r*-1). Stop after step B. At this point, you have calculated all the possible wrappings using B rubber bands and containing any consecutive sequence of points. You just have to choose the best one which contains all points. This is given by the minimum value M[*i*][*i*-1]. Just print this value. This algorithm takes O(B*N^3) time and uses O(N^2) space.

Important: All the input data for this problem are integers. So, all the intermediate computations can be done using integers (even area calculation - it is worth seeing that 2*area is an integer). The usage of floating point numbers for this problem may cause errors, once some calculations return big numbers (absolute value) which may not fit in the significand of some floating point data types. In such cases, these types loose information (precision) which may lead to a wrong answer for the problem. There will be no problem if the used floating point type has a significand big enough to cope with intermediate calculations without loosing precision.

Problem F - I hate SPAM, but some people love it

Level: Easy

This easy problem consists of verifying reachability and degree of vertices in a graph. For every SPAM message, verify if it reaches a person (i.e. if there is a path from the originator to the person in the given graph). If so, the number of SPAM messages sent by that person is equal to its degree in the graph, otherwise it is zero, once it does not receive the message. According to this value, match the correct attribute and print it following the given ordering constraints. Reachability can be easily verified using a transitive closure algorithm (e.g. Floyd-Warshall).

Problem G - Noise Effect

Level: Easy

This is certainly the easiest problem in the set. Once you have read both scanned and standard images, test

all the four possible rotations of the scanned image comparing it with the standard one. Invert the image (mirror effect) and repeat the previous operations. Choose the best confidence degree and print it.

Problem H - Supermarket

Level: Intermediate

A simple approach to solve this problem consists of maintaining an array L[M+1] (range 0..M), where L[*j*] represents the minimum cost necessary to buy the *j* first items in the list following the given restrictions. Initially, make L[0] equal to 0 and L[*j*] (*j* <> 0) equal to INFINITE. On step *i*, you calculate L for the *i* first products in the supermarket, based on the array calculated in the previous step. Let LA be the array L calculated in the previous step; if the current supermarket product appears as the *j*-th element in the list, you have that L[*j*] = min(LA[*j*] , LA[*j* -1] + *its cost*). In other words, you have to choose between the two possibilities:

- to buy the *j* first elements in the list from the *i*-1 first products in the supermarket. (the previous calculated value LA[*j*])
- to buy the *j-1* first elements in the list from the *i*-1 first products in the supermarket (LA[*j* -1]) and to buy the *i*-th product in the supermarket as the *j*-th element in the list.

Here is the algorithm for a single step:

```
{ On step i }
Let sp be the i-th product in the supermarket
Let cost be its cost
for j := 1 to M do
     Let lp be the j-th product in the list
     if sp == lp then
          L[j] := min(LA[j], LA[j-1] + cost)
     else
          L[j] := LA[j]
     end
end
```

A strategy very similar to that one used on problem D (going backwards in the loop) can be used in order to avoid using the array LA.

After N steps, you have calculated L for the entire supermarket. So, if L[M] is INFINITE, it is impossible to buy all the products in the list. Otherwise, L[M] is the minimum cost he would pay to buy everything. It is worth seeing that this algorithm runs in O(M*N) and requires O(M) space.

Last modified: Fri May 16 10:54:18 BRT 2003