SRM 270 problemset

Michal Forišek

<misof@mfnotes.ksp.sk>
Department of Computer Science
Faculty of Mathematics, Physics, and Informatics
Comenius University, Bratislava, Slovakia

November 2005

This is a set of problems I created for TopCoder's Algorithm Competition, Single Round Match (SRM) 270.

The problems are presented in (subjectively) increasing order of difficulty. The actual point values used in the SRM are given in the following table.

	Division 1	Division 2
Buying Cheap		250
Tipping Waiters		600
Countries Ranklist	300	900
Salesman's Dilemma	600	
Packing Shapes	900	

Contents

1 Problem statements

1.1 Buying Cheap

Problem Statement

Steve would like to buy a new car. He isn't wealthy, so he would prefer a reasonably cheap car. The only problem is that the quality of the cheapest cars is... let's say questionable.

Thus Steve decided to make a list of car prices and to buy a car with the third lowest price.

You will be given a list of integers **prices**. The same price may occur multiple times in **prices**, but it should count only once in the ordering of available prices. See examples for further clarification.

Write a function that returns the third lowest price in this list. If there are less than three different car prices in **prices**, you should return -1.

Constraints

prices will contain between 1 and 50 elements, inclusive.

Each element in **prices** will be between 1 and 1000, inclusive.

Examples

input

```
{10, 40, 50, 20, 70, 80, 30,
90, 60}
```

output

```
30
```

input

```
{10, 10, 10, 10, 20, 20, 30,
30, 40, 40}
```

output

```
30
```

The lowest price is 10, the second lowest is 20 and the third lowest is 30.

input

```
{10}
```

output

```
-1
```

input

```
{80, 90, 80, 90, 80}
```

output

```
-1
```

1.2 Tipping Waiters

Problem Statement

In a restaurant, if you were pleased by the waiter's service, you may leave him a tip – you pay him more than the actual value of the bill, and the waiter keeps the excess money. In some countries, not leaving a tip for the waiter is even considered impolite. During my recent holiday I was having dinner in a foreign restaurant. The pamphlet from my travel agency informed me that the proper way of tipping the waiter is the following:

- The sum I pay must be round, i.e., divisible by 5.
- The tip must be between 5% and 10% of the *final sum I pay*, inclusive.

Clearly, sometimes there may be multiple "correct" ways of settling the bill. I'd like to know exactly how many choices I have in a given situation. I could program it easily, but I was having a holiday... and so it's you who has to solve this task. You will be given:

- an int **bill** – the amount I have to pay for the dinner
- an int **cash** – the amount of money I have in my pocket

Write a function that computes how many different final sums satisfy the conditions above.

Notes

Assume that both **bill** and **cash** are in dollars.

All the money I have is in one-dollar banknotes.

Constraints

bill and **cash** are between 1 and 2,000,000,000, inclusive.

bill doesn't exceed **cash**.

Examples

input	output
4 100	0

4 isn't a round sum, and 5 is already too much.

input	output
23 100	1

The only correct choice is to pay 25 dollars, thus leaving a tip of 2 dollars.

input

```
23
24
```

output

```
0
```

The same bill, but I don't have enough money to leave an appropriate tip.

input

```
220
239
```

output

```
1
```

This time, it is appropriate to pay either 235 or 240 dollars. Sadly, I don't have enough money for the second possibility.

input

```
1234567
12345678
```

output

```
14440
```

A large bill, but with that much money I don't care.

input

```
1880000000
1980000000
```

output

```
210527
```

input

```
171000000
179999999
```

output

```
0
```

1.3 Countries Ranklist

Problem Statement

An unnamed international contest just finished. There were exactly four contestants from each of the participating countries. During the contest each of the contestants achieved a non-negative integer score (the higher, the better). The contestants were sorted according to their scores and the first part of the overall results (i.e., the best few contestants) was announced during the final ceremony. The organizers of the contest decided not to publish the remaining, lower part of the results.

In the Countries Ranklist the countries are ordered (in decreasing order) by the total score of their four contestants. If two or more countries have the same score, they are tied for the best place from the corresponding interval, and the places of the lower ranked countries remain

unaffected. For example, if the total scores of countries A, B, C and D are 100, 90, 90 and 80, respectively, then B and C are tied for second place, and D is fourth. For further clarification, see examples 2 and 4.

The published part of the results is given in **knownResults**, with each of the elements describing one of the announced contestants. The elements have the form "`COUNTRY CONTESTANT SCORE`", where `COUNTRY` is the name of the country, `CONTESTANT` is the name of the contestant and `SCORE` is his score.

Your task will be to compute the best and the worst possible placement in the Countries Ranklist for each of the participating countries. You shall assume that from each country at least one contestant was announced and that all contestants *not* in the available part of the results scored *strictly less* than the worst contestant in the available part of rankings. (For example, if the worst announced contestant scored 47 points, then each of the not announced contestants from each of the participating countries could have scored at most 46 points.)

You are to return a sequence of strings with one element for each country. The form of each element must be "`COUNTRY BEST WORST`", where `COUNTRY` is the name of the country, `BEST` and `WORST` are the best and the worst position this country could possibly have in the Countries Ranklist. Order this list so that the country names are given in alphabetical order. Note that country names are case sensitive, and that in alphabetical order all uppercase letters come before lowercase letters. The numbers `BEST` and `WORST` mustn't contain leading zeroes.

Notes

All scores (even the unknown ones) are non-negative integers.

Constraints

knownResults contains between 1 and 50 elements, inclusive.

Each of the elements in **knownResults** is of the form "`COUNTRY CONTESTANT SCORE`".

Each `COUNTRY` and `CONTESTANT` in **knownResults** are strings containing between 1 and 10 letters ('a'-'z', 'A'-'Z').

For each country **knownResults** contains at most 4 contestants.

Each `SCORE` in **knownResults** is an integer between 1 and 600, inclusive, with no leading zeroes.

Examples

input

```
{"Poland Krzysztof 101",
 "Ukraine Evgeni 30",
 "Ukraine Ivan 24"}
```

output

```
{"Poland 1 1",
 "Ukraine 2 2"}
```

The worst announced contestant is Ivan with 24 points. Each of the contestants that weren't announced had to score strictly less, i.e., at most 23 points. Thus the total score of Ukraine is at most 30+24+23+23 = 100 and Poland surely wins.

input

```
{"Poland Krzysztof 100",
 "CzechRep Martin 30",
 "CzechRep Jirka 25"}
```

output

```
{"CzechRep 1 2",
 "Poland 1 2"}
```

This time, if the two missing Czech competitors scored 24 points each (and the remaining three from Poland scored 0), Czech Republic could still win. Note the order in which the countries are reported in the output.

input

```
{"Slovakia Marian 270",
 "Hungary Istvan 24",
 "Poland Krzysztof 100",
 "Hungary Gyula 30",
 "Germany Tobias 27",
 "Germany Juergen 27"}
```

output

```
{"Germany 2 4",
 "Hungary 2 4",
 "Poland 2 2",
 "Slovakia 1 1"}
```

This is an interesting case. Slovakia is sure to win, and Poland is sure to be second. But it is possible that Germany, Hungary and Poland have an equal total score of 100. In this case they are all tied for second place.

input

```
{"usa Jack 14",
 "USA Jim 10",
 "USA Jim 10",
 "USA Jim 10",
 "USA Jim 10",
 "usa Jack 14",
 "usa Jack 14",
 "Zimbabwe Jack 10"}
```

output

```
{"USA 2 2",
 "Zimbabwe 3 3",
 "usa 1 1"}
```

Case matters, "USA" and "usa" are two different countries. Contestant names don't matter, i.e., from "USA" there are four different contestants, all named "Jim".

input

```
{"A a 9", "A b 9", "A c 9",
 "A d 9", "B e 9", "B f 9",
 "B g 8", "B h 8", "C i 9",
 "C j 9", "C k 9", "C l 7",
 "D m 1", "D n 1", "D o 1",
 "D p 1"}
```

output

```
{"A 1 1",
 "B 2 2",
 "C 2 2",
 "D 4 4"}
```

All results have been announced, so everything is clear. A is first, B and C are tied for second, and D is fourth.

1.4 Salesman's Dilemma

Problem Statement

Travelling salesmen (and of course, travelling saleswomen, too) usually have lots of problems. And some of them are pretty hard to solve algorithmically. Let's take a look at one such problem.

In the country there are several towns. Let **towns** be their count. The towns are numbered from 0 to **towns** − 1. Our travelling salesman wants to travel through this country. He starts his journey in the town **origin** and wants to end it in the town **destination**.

There are several means of transportation he may use. For each of them he knows the source and destination town and its cost. Furthermore, each time he enters a town, he will be able to sign some contracts there and make some money. This amount of money may differ from town to town, but for each town it is a fixed amount. He does sign contracts at the beginning of his journey (in the town **origin**). Of course, the salesman's goal is to maximize the amount of money he has at the end of his journey.

The various travel possibilities are given in **travelCosts**, and the profits for each town are given in **profits**. Each element of **travelCosts** is of the form "`SOURCE DESTINATION COST`", where `SOURCE` and `DESTINATION` are town numbers and `COST` is the cost of using this transportation. In profits there are exactly **towns** elements, and the i-th of them is the amount of money gained when entering town i.

If it is not possible to reach the destination town at all, return the string "`IMPOSSIBLE`". If it is possible to reach the destination town with an arbitrarily large final profit, return the string "`ENDLESS PROFIT`". Otherwise, return the string "`BEST PROFIT: X`", where `X` is the best profit he can make. The value `X` must be printed with no unnecessary leading zeroes.

Notes

We are interested in the amount the salesman earns during the journey, i.e., the difference between the balance of his account at the end and at the beginning of his journey. This difference may also be negative in case the salesman has to pay more than he earns during his journey.

The salesman may travel to the same town multiple times. He gains the profit from the town once for each visit he makes.

All means of transportation can only be used in the specified direction, i.e., they are one-way. The salesman may use each of them multiple times.

Constraints

towns is between 1 and 50, inclusive.

origin and **destination** are between 0 and **towns** − 1, inclusive.

travelCosts has between 1 and 50 elements, inclusive.

Each element of **travelCosts** is of the form "`SOURCE DESTINATION COST`".

Each `SOURCE` and `DESTINATION` in **travelCosts** are integers between 0 and **towns** − 1, inclusive, with no unnecessary leading zeroes.

Each `COST` in **travelCosts** is an integer between 1 and 1,000,000, inclusive, with no leading zeroes.

profits has exactly towns elements.

Each element of **profits** is between 0 and 1,000,000, inclusive.

Examples

input

```
5
0
4
{"0 1 10", "1 2 10", "2 3 10",
 "3 1 10", "2 4 10"}
{0, 10, 10, 110, 10}
```

output

```
"ENDLESS PROFIT"
```

The profit in each town exactly covers the cost of travelling there. The single exception is town 3. Each time the salesman travels (from town 2) to town 3, he pays 10 for the travel and gains 110 from sales in town 3, resulting in a net gain of 100.

He is able to reach an arbitrarily large profit in the following way: Start by travelling from 0 to 1, travel around the circle $1 \to 2 \to 3 \to 1$ *sufficiently many times, then travel from 1 to 2 and from 2 to 4.*

input

```
5
0
4
{"0 1 13", "1 2 17", "2 4 20",
 "0 3 22", "1 3 4747",
 "2 0 10", "3 4 10"}
{0, 0, 0, 0, 0}
```

output

```
"BEST PROFIT: -32"
```

There is no profit here, so the salesman has to find the cheapest way of travelling.

input

```
3
0
2
{"0 1 10", "1 0 10", "2 1 10"}
{1000, 1000, 47000}
```

output

```
"IMPOSSIBLE"
```

The destination town is unreachable.

input

```
2
0
1
{"0 1 1000", "1 1 10"}
{11, 11}
```

output

```
"ENDLESS PROFIT"
```

Loops are allowed.

input

```
1
0
0
{"0 0 10"}
{7}
```

output

```
"BEST PROFIT: 7"
```

Loops are not always useful.

input

```
5
0
4
{"0 1 13", "1 2 17", "2 4 20",
 "0 3 22", "1 3 4747",
 "2 0 10", "3 4 10"}
{8, 10, 20, 1, 100000}
```

output

```
"BEST PROFIT: 99988"
```

1.5 Packing Shapes

Problem Statement

Little Timmy has a rectangular frame and several shapes cut out from cardboard. He tries to fit each of the shapes into the frame (one shape at a time). Sometimes the piece fits easily, sometimes it is clear that the shape is too big to fit... and sometimes Timmy just doesn't know. Then he always comes to ask you for help.

You decided to write a program that will answer Timmy's questions.

You will be given the **width** and **height** of the frame and a description of Timmy's **shapes**. Each of the shapes is either a circle or a rectangle. Each element of **shapes** is of the one of the following forms:

- "`CIRCLE RADIUS`", where `RADIUS` is the radius of the circle.
- "`RECTANGLE WIDTH LENGTH`", where `WIDTH` and `LENGTH` are the dimensions of the rectangle.

Return a sequence of strings, where the i-th element is "YES" or "NO", depending on whether the i-th shape fits into the empty frame.

Notes

The shapes may be rotated arbitrarily.

The shapes may touch the frame, and they may even have a common part of the boundary.

Constraints

width and **height** are between 1 and 1000, inclusive.

shapes contains between 1 and 50 elements, inclusive.

Each element of **shapes** has the form described above.

All numbers in **shapes** are integers between 1 and 1000, inclusive, with no leading zeroes.

Examples

input

```
100
100
{"RECTANGLE 3 3",
 "RECTANGLE 3 230",
 "RECTANGLE 140 1"}
```

output

```
{"YES", "NO", "YES"}
```

The first rectangle clearly fits, but the second one clearly doesn't. The third one can be placed inside the frame after it is rotated 45 degrees.

input

```
100
100
{"RECTANGLE 100 100",
 "CIRCLE 50"}
```

output

```
{"YES", "YES"}
```

Touching the frame is allowed.

input

```
10
100
{"RECTANGLE 99 9",
 "CIRCLE 22"}
```

output

```
{"YES", "NO"}
```

The rectangle can be rotated, but the circle is too large.

input

```
170
900
{"RECTANGLE 200 700",
 "RECTANGLE 3 910",
 "RECTANGLE 1000 7",
 "CIRCLE 5",
 "CIRCLE 50",
 "CIRCLE 500",
 "RECTANGLE 1000 99"}
```

output

```
{"NO", "YES", "NO", "YES",
 "YES", "NO", "NO"}
```

2 Solutions

2.1 Buying Cheap

This problem was as simple as it gets. Maybe the only thing worth noting is: don't reinvent the wheel. If there is a library function for a problem, use it.

In this case, most programming languages include a function to remove duplicate elements from a (sorted) list. A very nice and simple way to solve this task in C++ is to convert the input vector into a set (it gets sorted and duplicates are removed), back into a vector and output its third element (or -1).

```
int thirdBestPrice(vector prices) {
  set S ( prices.begin(), prices.end() );
  vector V ( S.begin(), S.end() );
  if (V.size() < 3) return -1;
  return V[2];
}
```

The same without complicated data structures:

```
int thirdBestPrice(vector<int> prices) {
  sort( prices.begin(), prices.end() );
  int cnt = 1;
  for (int i = 1; i < prices.size(); i++)
    if (prices[i] != prices[i - 1] && ++cnt == 3)
      return prices[i];
  return -1;
}
```

2.2 Tipping Waiters

The moral of the story for this task is: don't use "real" numbers if you don't have to.

The problem was solvable by brute force: For each round number between the bill and the amount I have, calculate the tip and check whether its size is appropriate. (As an optimization, you may stop as soon as the tip exceeds 10% of the sum you pay).

However, there is also a solution in constant time. Let B be the value of the bill and S the sum I'll pay. Then the tip is $S - B$. For it to be appropriate, this value must be between $S/20$ and $S/10$, inclusive. We get two inequalities: $S/20 \leq S - B \ \wedge \ S - B \leq S/10$. We may rewrite these equations to get $S \geq 20B/19$ and $S \leq 10B/9$.

In other words, sums that lead to an appropriate tip lie in the interval $[20B/19, 10B/9]$. The boundaries of this interval are not necessarily integers. To get integral bounds, we take the ceiling of the lower bound and the floor of the upper bound. Now we only have to compute the number of round sums in this interval, which is a pretty easy task.

```
int possiblePayments(int bill, int cash) {
  // compute the smallest and largest valid sums
  int minPay = bill + (bill + 18) / 19; // the ceiling of the lower bound
  int maxPay = bill + bill / 9;
  if (cash < maxPay) maxPay = cash;

  // adjust the bounds to be round
  while (minPay % 5 != 0) minPay++;
  while (maxPay % 5 != 0) maxPay--;

  int result = 0;
  if (maxPay >= minPay) result = 1 + (maxPay - minPay) / 5;
  return result;
}
```

2.3 Countries Ranklist

There are not many tasks that can be shared as an easy problem in Division 1 and a hard problem in Division 2, but the results show that this was one of them. It required some thinking before you started to write code, but once you saw how to solve this task, the code becomes

pretty straightforward. As a consequence, this problem had a pretty high success rate in both divisions.

First of all, suppose that I know the exact scores for each of the countries. How to compute the place of a given country? Simply, it's one plus the number of countries that had a higher total score.

Knowing this, how to compute the best possible placement for some country C? We start by finding the worst announced coder. Let his score be S. Now the best scenario for C is that each of its missing contestants scored $S-1$ points and all the other missing contestants scored zero. This uniquely determines the score of each of the countries, hence we can compute the best possible placement for C.

When looking for the worst possible placement the computation is reversed, i.e., contestants from C score zero and all the others score $S-1$ points.

Note: The presented solution was quadratic in the number of countries. There is a linear time solution. We leave it as an exercise for the readers.

2.4 Salesman's Dilemma

Tricky, tricky, tricky! I don't remember seeing such a low success rate for quite a while... The number of submissions was much higher than I expected – and as the success rate shows, many of the coders weren't alerted by the high point value and they didn't spend enough time testing their solutions.

This task offered quite a lot of different opportunities to make a mistake – and many of them weren't covered by examples in the problem statement. If you stopped to think about this problem and discovered them, they could be (and in many cases they were) converted into "killer" test cases for the challenge phase.

The first step towards a correct solution is to construct a directed graph, where the value of each edge is its travel cost minus the profit in the destination city. In this graph we are interested in the shortest walk from our origin to our destination.

The simple case is when destination is not reachable. But if it is reachable, there may be no shortest walk in the graph, because of negative cycles. This was the problematic case we had to detect. (In the original problem statement, this was the "endless profit" case.)

What are these negative cycles we just mentioned? If the salesman discovers a cycle with a negative sum of edge costs, he may start going around this cycle, making an arbitrarily large profit. Of course, the occurrence of a negative cycle somewhere in the graph is not a sufficient condition to output "endless profit" – the cycle must be reachable from the origin and the destination must be reachable from the cycle.

So basically the problem boils down to detecting negative cycles reachable from the origin. If there is one such that the destination is reachable from it, we have the endless profit case, otherwise the shortest walk from origin to destination is a path (i.e., it visits no vertex twice) and it can be found using some standard algorithm.

Author's solution uses the algorithm by Bellman and Ford to compute the shortest paths from origin and detect reachable negative cycles. Then all vertices reachable from them are found using a simple breadth-first search. In the contest, some of the coders solved this task using the algorithm by Floyd and Warshall. Both are standard textbook algorithms, if you don't know them, there are lots of online resources on this topic.

2.5 Packing Shapes

This problem wasn't as nasty as other geometry problems, but still many of the solutions failed due to precision problems and/or handling some boundary cases incorrectly.

The circle part is simple, its diameter (=twice the given radius) mustn't exceed either dimension of the frame. The interesting part were the rectangles.

Basically, there were three possible approaches. The first one is iterative: Take the small rectangle and start to rotate it in small steps. After each rotation, measure its projection on both coordinate axes and check whether it currently fits the frame or not.

Submitting such a solution is a gamble. It may pass, if you are lucky, but it may very easily fail. And the prepared set of test cases contained both hand-crafted and autogenerated cases meant to cause such solutions to fail. As far as I was able to find out after the match, most such solutions failed.

A second, far better way, was to employ binary search on the correct angle. Imagine a rectangle lying on its longer side. If we start rotating it, its projection on the x axis will start to shrink. What we want is to find the smallest angle such that the rectangle fits into the frame horizontally... and check whether for this angle the rectangle fits also vertically.

Doubles are precise enough for this solution to work reliably, as long as we choose a suitable margin for error tolerance. In the contest this was probably the best solution.

The third possibility was to compute the correct angle directly. Here we had to be extremely careful so that we don't overlook some special corner case.

As the author of this task, I had to make sure the outputs are correct. Thus I had to choose a fourth possibility – do some exact computation, using integers only! My code follows, with some explanation below.

```
// small rectangle: c*d, large rectangle: a*b

if (c<=a && d<=b) return "YES";
if (c<=b && d<=a) return "YES";

// let
// f: y = (b-x)c / d
// g: y = -xd/c + a
// we want a point [x,y] lying below f,g such that x>0, y>0, x^2+y^2 = c^2

// check whether one of the lines cuts away the whole arc
if (a <= c && a <= d) return "NO";
```

```
if (b <= c && b <= d) return "NO";

// now we are interested in whether they intersect...
// ... and whether the intersection point lies outside the arc

// parallel if c==d, but this case should already be handled
long kc = b*c - a*d,     km = c*c - d*d;
long xc = c * kc,        yc = a * km - d * kc;

if (xc*xc + yc*yc < km * km * c * c) return "NO"; else return "YES";
```

The idea? Either the sides of the "small" rectangle are parallel with the sides of the "large" rectangle, or not. The first case is easy to check. For the second case, we may assume that both endpoints of a side with length c touch the large rectangle. Why? Consider an arbitrary correct placement of the small rectangle inside the large rectangle. We may now move it (without rotating it) until the endpoints of a side of our choice touch the large rectangle.

In the corner of the large rectangle we get a right triangle with the hypotenuse c. Let x and y be the lengths of the other two sides. Using x and y we can easily compute the width and height of the rotated "small" rectangle. None of them may exceed the corresponding dimension of the "large" rectangle. This leads to two linear inequalities for x and y. We compute whether they have a solution such that $x^2 + y^2 = c^2$. If yes, the "small" rectangle can be rotated and the answer is positive, otherwise the answer is negative.