SRM 287 problemset

Michal Forišek

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

February 2006

This is a set of problems I created for TopCoder's Algorithm Competition, Single Round Match (SRM) 287.

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**
Customer Statistics		250
Two Equations	300	600
Fixed Size Sums		1000
Moore's Law	450	
Coin Game	1000	

Contents

1 Problem statements

1.1 Customer Statistics

Problem Statement

You will be given a list of customer names **customerNames** extracted from a database. Your task is to report the customers that occur more than once in this list, and the number of occurrences for each of the repeated customers.

Your method should return the report in the following form: Each element in the report should be of the form "`NAME OCCURS`", where `NAME` is the name of one customer and `OCCURS` is the number of times his name occurs in **customerNames**. Sort the result in alphabetical order by customer name.

Constraints

customerNames contains between 1 and 50 elements, inclusive.

Each element of **customerNames** contains between 1 and 50 characters, inclusive.

Each element of **customerNames** contains uppercase letters ('A'-'Z') only.

Examples

input

```
{"JOHN", "BOB", "JOHN",
 "BILL", "STANLEY", "JOHN"}
```

output

```
{"JOHN 3"}
```

The only repeated name is `JOHN`, and it occurs three times.

input

```
{"YETTI", "YETTI", "YETTI",
"BIGFOOT", "BIGFOOT"}
```

output

```
{"BIGFOOT 2", "YETTI 3"}
```

Note the sorting order.

input

```
{"ANDREW", "BILL", "CINDY",
 "DOH", "ERGH", "FOO",
 "GOO", "HMPH"}
```

output

```
{}
```

No repeated names this time.

input	output
{"THEONLYCUSTOMER"}	{}

Again, no repeats.

input	output
{"A", "B", "A", "C", "A", "B", "A", "D", "D", "D"}	{"A 4", "B 2", "D 3"}

1.2 Two Equations

Problem Statement

You are given two strings, **first** and **second**. Each of these strings contains a simple linear equation with variables `x` and `y`. Your task is to determine whether this pair of equations has a unique solution, and if so, to compute it.

Each of the equations is of the form "`A*X + B*Y = C`". The spaces must appear exactly as in this example, i.e., there is exactly one space both before and after the signs "`+`" and "`=`", and there are no other spaces. The coefficients `A`, `B`, and `C` are integers. If a coefficient is non-negative, the equation contains just the number, without the unary plus sign. If a coefficient is negative, it is always enclosed in parentheses, and it contains the unary minus sign. No coefficient will contain unnecessary leading zeroes.

If the pair of equations has a unique solution, return the solution formatted as a string of the form "`X=A/B Y=C/D`", where `A`, `B`, `C`, and `D` are integers such that both fractions `A/B` and `C/D` are reduced, and the numbers `B` and `D` are positive. The numbers should be encoded in the same way as the coefficients of the equations – i.e., negative numbers must be enclosed in parentheses.

If the pair of equations has more than one solution, return the String "`MULTIPLE SOLUTIONS`". If the pair of equations has no solutions, return the String "`NO SOLUTIONS`".

Notes

A fraction `A/B` is called reduced if and only if the greatest common divisor of `A` and `B` is 1. Note that each rational number corresponds to exactly one reduced fraction with `B>0`.

Constraints

first and **second** will be formatted as described in the problem statement.

All coefficients in both equations will be integers between -9 and 9, inclusive.

Examples

input

```
"1*X + 2*Y = 6"
"1*X + (-4)*Y = (-3)"
```

output

```
"X=3/1 Y=3/2"
```

Multiply the first equation by two, then add the second equation to the first one. You will get a new equation: `3*X = 9`. Thus `X=3`. If we substitute this into one of the original equations, we get `Y=3/2`.

input

```
"(-3)*X + 0*Y = 7"
"0*X + 8*Y = 6"
```

output

```
"X=(-7)/3 Y=3/4"
```

This time we can compute each of the variables separately. Note that in the result, the numerator (not the denominator) of `X` is negative, and that it is enclosed in parentheses. Also, note that the value of `Y` is output as a reduced fraction.

input

```
"1*X + 0*Y = 1"
"1*X + 0*Y = 1"
```

output

```
"MULTIPLE SOLUTIONS"
```

input

```
"1*X + 3*Y = 1"
"2*X + 6*Y = (-1)"
```

output

```
"NO SOLUTIONS"
```

input

```
"0*X + 0*Y = 0"
"(-3)*X + (-3)*Y = 0"
```

output

```
"MULTIPLE SOLUTIONS"
```

1.3 Fixed Size Sums

Problem Statement

Let **sum** and **count** be positive integers. Consider all possible ways to express **sum** as a sum of exactly **count** positive integers. Two ways are considered equal if we can obtain one from

the other by changing the order of the summed numbers. For example, $8 = 3+2+1+1+1$ is the same as $8 = 1+3+1+2+1$, but not the same as $8 = 3+2+2+1$. Thus we will only be interested in such summations where the summed integers are in non-increasing order.

For example, if **sum** = 8 and **count** = 3, the possible ways are:

$$\begin{aligned} 8 &= 6+1+1 \\ 8 &= 3+3+2 \\ 8 &= 4+2+2 \\ 8 &= 4+3+1 \\ 8 &= 5+2+1 \end{aligned}$$

We may now order these summations in the following way: Order them according to the first summand in decreasing order. In case of a tie, order them according to the second summand, etc. In general, to compare two summations, look at the first summand where they differ. The one where this summand is larger will be earlier in our order.

For our previous example, the correct order is:

$$\begin{aligned} 8 &= 6+1+1 \\ 8 &= 5+2+1 \\ 8 &= 4+3+1 \\ 8 &= 4+2+2 \\ 8 &= 3+3+2 \end{aligned}$$

You will be given three integers: **sum**, **count** and **index**, where **index** contains a zero-based index of a summation in the order defined above. Your method should return a string containing that summation in the form "`SUM=SUMMAND(1)+...+SUMMAND(count)`". If **index** is too large to specify a valid summation, return an empty string.

Notes

You may assume that the total number of possible summations will never exceed $2,000,000,000$.

Constraints

sum is between 1 and 150, inclusive.

count is between 1 and 150, inclusive.

index is between 0 and 2,000,000,000, inclusive.

Examples

input

```
8
3
2
```

output

```
"8=4+3+1"
```

This is the example from the problem statement. Look at the ordered list of possible summations and number them starting with zero.

input

```
13
1
0
```

output

```
"13=13"
```

There is only one possibility here.

input

```
13
13
0
```

output

```
"13=1+1+1+1+1+1+1+1+1+1+1+1+1"
```

Again, there is only one possible summation.

input

```
7
10
3
```

output

```
""
```

This is impossible. A sum of 10 positive integers is never equal to 7.

input

```
17
2
4
```

output

```
"17=12+5"
```

The first five possible summations are: $17 = 16+1$, $17 = 15+2$, $17 = 14+3$, $17 = 13+4$, *and* $17 = 12+5$.

output

```
140
17
87654321
```

output

```
"140=40+31+15+15+9+7+4+4+2+2+2+2+2+2+1+1+1"
```

1.4 Moore's Law

Problem Statement

Moore's law is a well-known prediction of the growth of computing power over time. This is the formulation we will use in this problem: *The speed of new computers grows exponentially and doubles every 18 months.* In this problem we will assume that reality precisely obeys this law.

Suppose that you have a hard computational task that would take 14 years to complete on a current computer. Surprisingly, starting its computation as soon as possible is not the best you can do. A better solution: Wait for 18 months and buy a better computer. It will be twice as fast, and therefore solve the task in 7 years. You would have the result 8.5 years from now. In the best possible solution you should wait for slightly over 4 years. The computer you'll be able to buy then will solve the task in approximately 2.2 years, giving a total of 6.2 years.

You have a computational task you want to solve as quickly as possible. You will be given an integer **years** giving the number of years it would take on a computer bought today. Return a double giving the least number of years in which you will have the result of the task if you use the above approach.

Notes

Your return value must have an absolute or relative error less than 1e-9.

The computation speed growth is a continuous exponential function satisfying the property from the problem statement.

Constraints

years will be between 1 and $1,000,000,000$, inclusive.

Examples

input	output
14	6.2044816339207705

The example from the problem statement.

input	output
3	2.870893001916099

input	output
47	8.82533252835082

input	output
123	10.907221008843223

1.5 Coin Game

Problem Statement

Two players play a simple game with a coin.

Given is a set S. Each element of S is a sequence of heads and tails, and all the sequences have the same length.

The first player starts the game by choosing a sequence from S. Then the second player chooses a different sequence from S. A fair coin is then thrown repeatedly until one of the chosen sequences appears as a set of consecutive throws. The player that has chosen this sequence wins.

You are given a list of strings **sequences** – the set S of allowed sequences encoded as strings of 'H's and 'T's ('H' represents a head, 'T' represents a tail). Return the probability that the first player will win this game, given that both players play optimally.

Notes

The returned value must be accurate to 1e-9 relative or absolute.

Constraints

sequences contains between 2 and 50 elements, inclusive.

Each element of **sequences** contains only uppercase 'H' and uppercase 'T' characters.

Each element of **sequences** contains between 1 and 10 characters, inclusive.

Each element of **sequences** contains the same number of characters.

No two elements in **sequences** are equal.

Examples

input	output
{"H", "T"}	0.5

This game is as simple as it gets. The first player picks a head or a tail, then they flip a coin and look at the outcome. Each player has a 50% chance of winning this game.

input

```
{"HHHHHHHHH", "TTTTTTTTT"}
```

output

```
0.5
```

Almost the same game, but quite a bit longer.

input

```
{"HH", "HT"}
```

output

```
0.5
```

This is another example of a fair game. Once a head occurs for the first time, the next throw decides which sequence will appear first.

input

```
{"TTT", "HTT"}
```

output

```
0.875
```

In this game, choice matters. The only way `TTT` *can appear before* `HTT` *is if it appears in the first three throws. Otherwise the triplet we encounter immediately before seeing* `TTT` *for the first time is always* `HTT`*. Thus in all other cases* `HTT` *appears before* `TTT`*.*

The first player knows this and he chooses `HTT`*, which gives him a* $7/8$ *chance of winning the game.*

input

```
{"HHH", "HHT", "HTH", "HTT",
 "THH", "THT", "TTH", "TTT"}
```

output

```
0.33333333333333333
```

This game contains a wonderful paradox: being the first player is not always good.

input

```
{"TTHTTHTTH", "HTTHTHHTT",
 "TTHHHHTHT", "TTTTTHTTT",
 "HHTTHTHHT"}
```

output

```
0.5019379844961235
```

2 Solutions

2.1 Customer Statistics

A straightforward way how to solve this task is to sort the customer list and then in one loop count the number of occurences of each name.

Another possibility is to use some library data structure, such as `TreeMap` in Java or `map` in C++. The advantage of this approach is that there are less special cases to consider, and thus less chances to make a mistake.

```
#define FOREACH(it,cont) for (it=(cont).begin(); it!=(cont).end(); ++it)

vector<string> reportDuplicates (vector<string> customerNames) {
  map<string,int> occurs;

  // count the number of occurences for each customer
  for (unsigned i=0; i<customerNames.size(); i++)
    occurs[ customerNames[i] ]++;

  vector<string> result;

  // process all records in the map "occurs"
  // each time we have a duplicate, report it
  FOREACH(it,occurs) {
    if (it->second > 1) { // more than one occurence
      stringstream sout;
      sout << (it->first) << " " << (it->second);
      result.push_back( sout.str() );
    }
  }
  return result;
}
```

2.2 Two Equations

I admit that this problem was nasty. Really nasty. And while it was meant to be a tough lesson that being careful pays off, I didn't expect it to be this hard. Thus I decided that in this case explaining the idea of the solution simply isn't enough. I will start this editorial with some thoughts on how to tackle problems like this one.

First and most important, don't reinvent the wheel! If you know that a problem is both classical and tricky, it has been solved correctly oh so many times long before you tried to solve it in the SRM. Don't be afraid to consult online references such as MathWorld, or even use Google to look for a discussion on the topic in question.

Only when you are really left "on your own", start working out all the tricky details. Again, don't be afraid. This time, don't be afraid to put away the keyboard and grab a pen and a

paper. Work out all the necessary details before you start coding, make sure you didn't forget anything, double-check your thoughts. Speed does not help if your solution fails.

If the problem contains different types of work, start with coding the simple, mechanical parts. In this case, the first step should be parsing the input. The simplest way is to split the equation as follows: `String[] F = first.split("[()+*=XY ]+");` Then you simply convert the non-empty strings in `F` into numbers. Next, write the output routine and a function to compute the greatest common divisor (in case your programming language doesn't have it built-in).

Now we can get to the important part, solving the equations.

First of all, note that an equation "`0*X + 0*Y = c`" has no solutions for $c \neq 0$, and for $c = 0$ each pair (X, Y) is a solution.

Now suppose that we have an equation "`a*X + b*Y = c`" such that $(a, b) \neq (0, 0)$. The set of solutions of each such equation corresponds to a line in the XY plane. (More exactly, if $b \neq 0$, the line is the graph of the function $Y = -(a/b) * X + (c/b)$, otherwise it is a vertical line with $X = c/a$.)

The first outline of our algorithm:

1. If one of the equations is of the form "`0*X + 0*Y = c`" with $c \neq 0$, return "`NO SOLUTIONS`".
2. If one of the equations is of the form "`0*X + 0*Y = 0`", return "`MULTIPLE SOLUTIONS`". (The other equation has got an infinite number of solutions, and all of them satisfy this equation as well.)
3. Solve the case when both equations correspond to lines.

Hopefully you now noted that we can't swap steps 1 and 2. The second step depends on the first one, we have to be sure that the other equation is solvable. Swapping these two steps was a very common mistake in the SRM.

We are left with the case that the solutions correspond to lines. Again, there are three cases:

1. The lines are not parallel, thus there is exactly one intersection.
2. The lines are parallel but not identical, there is no solution.
3. The lines are identical, there are infinitely many solutions.

There are different possibilities how to distinguish between these cases, I prefer vectors. It can be easily seen that the vector $[a, b]$ is perpendicular to the line determined by "`a*X + b*Y = c`". Thus: (the lines "`a*X + b*Y = c`" and "`d*X + e*Y = f`" are parallel) iff (vectors $[a, b]$ and $[d, e]$ are parallel) iff (their vector product $ae - bd$ is zero).

In the case that the lines are not parallel, we can find the one solution, e.g., by elimination:

If we multiply the first equation by $(-d)$, the second one by a, and add the results together, we get: "`Y*(ae-bd) = (af-cd)`".

Similarly, we can multiply the first equation by e, and the second one by $(-b)$, we get: "`X*(ae-bd) = (ce-bf)`".

Clearly, as $(ae - bd)$ is not zero, we just found the unique solution: $X = (ce - bf)/(ae - bd)$, $Y = (af - cd)/(ae - bd)$. We only have to simplify it and print it.

If the lines are parallel, we have to tell whether they are identical, in other words whether one of the equations is a multiple of the other one. This can be simply done by checking whether the vector product of $[a, b, c]$ and $[d, e, f]$ is zero.

(If you are not comfortable with vectors, you can substitute $X = 0$ or $Y = 0$ into the first equation, compute one solution of the first equation, and check whether it satisfies the other equation.)

A tricky solution

Note that in the general case when there is one solution, the values of X and Y have (before the reduction) the same denominator. Also, it can be shown that for the allowed input values the magnitude of the denominator and also the magnitudes of both numerators don't exceed 200.

Thus, let $S = \{u/v\ ;\ -200 \leq u \leq 200\ \wedge\ 1 \leq v \leq 200\}$. We already know that if the system of equations has exactly one solution, both X and Y for the solution lie in S. Also, it can easily be shown that any valid equation other than "`0*X + 0*Y = c`" for $c \neq 0$ has more than one solution in S.

It follows that we can simply write three nested loops to check all "candidates", and in the end count the solutions we found.

2.3 Fixed Size Sums

This was a pretty straightforward dynamic programming problem. With a different (easier) medium problem I would expect that more coders should be able to solve this one.

The combinatorial object described in the problem statement is called integer partitions. In this case we are interested in partitions with a fixed number of elements.

The common trick in almost all combinatorial problems is to generate the answer sequentially. In this problem, we can start by asking the question: what is the first element of the partition?

To answer this question quickly, we have to count the partitions. In this case there are more suitable ways of doing it, we will present one of them. Let $C(N, K, L)$ be the count of partitions that have K elements that sum up to N, and the first (largest) element is L.

For example, for $N = 8$ and $K = 3$ we have $C(8, 3, 1) = 0$, $C(8, 3, 2) = 0$, $C(8, 3, 3) = 1$, $C(8, 3, 4) = 2$, $C(8, 3, 5) = 1$, $C(8, 3, 6) = 1$, $C(8, 3, 7) = 0$, and $C(8, 3, 8) = 0$.

If we knew the values $C(N, K, L)$ for all L, we could easily find the first element of the partition we seek: simply start with $L = N$ and decrease it until you find enough partitions.

The rest of the partition can be determined in a very similar way. Note that the rest of the partition is a partition of $N - L$ into $K - 1$ elements, where the first element can range from 1 to L. Moreover, we can easily compute the new index of this smaller partition.

For example, for $N = 8$, $K = 3$ and $index = 3$ we have:

- There are 0 partitions with the first element ≥ 8.
- There are $0 + 0 = 0$ partitions with the first element ≥ 7.
- There are $0 + 0 + 1 = 1$ partitions with the first element ≥ 6.
- There are $0 + 0 + 1 + 1 = 2$ partitions with the first element ≥ 5.
- There are $0 + 0 + 1 + 1 + 2 = 4$ partitions with the first element ≥ 4. These partitions will have indices ranging from 0 to 3. Thus in the partition we seek the first element is 4. Out of all partitions starting with the element 4 the one we seek has the index 1. (We subtracted the count of partitions starting with a greater element.)

Now we are left in a situation where $N = 4$, $K = 2$, $index = 1$ and the last element was 4.

- $C(4, 2, 4) = 0$, thus there are no partitions where the next element is 4.
- $C(4, 2, 3) = 1$, thus there is one partition where the next element is 3, this is not enough.
- $C(4.2.2) = 1$, and we are (almost) done. The resulting partition is $8 = 4 + 2 + 2$.

The remaining question is how to compute the values $C(N, K, L)$. The idea behind the answer will be remarkably similar to the ideas used above. Look at any partition of N into K elements such that the first element is L. If we remove the first element, we get a partition of $N - L$ into $K - 1$ elements such that the first element is at most L. This gives the general recurrence equation:

$$C(N, K, L) = C(N - L, K - 1, 1) + C(N - L, K - 1, 2) + \ldots + C(N - L, K - 1, L)$$

We may use dynamic programming or memoization to compute all the values $C(N, K, L)$. Afterwards, we determine the elements of the sought partition one by one. C++ code follows.

```
int C[160][160][160];

string kthElement(int sum, int count, int index) {
  memset(C,0,sizeof(C));

  // fill in the table, first the base case...
  for (int i=1; i<=sum; i++) C[i][1][i]=1;
  // ... then the general case
  for (int c=2; c<=count; c++)
    for (int s=c; s<=sum; s++)
      for (int l=1; l<=s-c+1; l++)
        for (int k=1; k<=l; k++)
          C[s][c][l] += C[s-l][c-1][k];

  // count all partitions of sum into count parts
  int all = 0;
  for (int i=1; i<=sum; i++) all += C[sum][count][i];

  // if there are not enough of them, return an empty string
  if (all <= index) return "";

  stringstream sres;
  sres << sum << "=";

  int remains = sum, last = sum;
  for (int i=0; i<count; i++) {
```

```
      // determine the i-th element of the partition:
      // start with the value of the previous element,
      // decrease it until you find enough partitions

      int element = last+1, seen = 0;
      while (seen <= index) { element--; seen += C[remains][count-i][element]; }

      // output the new element and update the current state
      sres << (i?"+":"") << element;
      index -= (seen - C[remains][count-i][element]);
      remains -= element;
      last = element;
    }

    return sres.str();
  }
```

Exercise. The time complexity of this solution is $\Theta(\mathbf{sum}^3 \cdot \mathbf{count})$. Find a way to improve it to $\Theta(\mathbf{sum}^2 \cdot \mathbf{count})$.

2.4 Moore's Law

There were essentially two different ways of solving this problem. In both of them, we will start by formalizing the problem statement. Suppose that we have a task that would take Y years to complete on a computer bought today. Let $f(t)$ be the computation time on a computer bought after t years. We were told that $f(t)$ is an exponential function such that $f(0) = Y$ and $f(1.5) = Y/2$. Thus clearly $f(t) = Y \cdot 2^{-t/1.5}$.

Our goal is to find a non-negative waiting time w such that the total time of the waiting and the computation is minimal. This time can be computed as $w + f(w)$.

The calculus approach

We want to find the minimum of the function $g(t) = t + Y \cdot 2^{-t/1.5}$. In general, the minimum occurs in a point where the derivative of this function is zero. We have $g'(t) = 1 + Y \cdot \ln 2 \cdot 2^{-t/1.5}/(-1.5)$, where ln is the natural logarithm. Solving for $g'(t) = 0$ gives a unique solution $t = (-1.5)\log_2(1.5/(Y \ln 2))$, and we are almost done.

Why almost? First, we should check that this is really a local minimum. This should be clear from the nature of the problem, formally it can be done by computing the second derivative in that point.

The second issue is more important. In this problem we are only interested in non-negative values of t (we can't start the computation sooner than now). If the function g reaches its minimum for a negative value of t, the minimal value for non-negative values of t will clearly be achieved for $t = 0$.

(This could be discovered by carefully testing your solution on boundary cases. If you forgot to check for negative t, for $Y = 1$ you got a result of less than 0.5 years. This is obviously wrong, as even a computer bought after half a year still needs more than 0.5 years to complete the task.)

The numeric approach

We will now describe a method of finding the minimum of a function known under the name ternary search. First, we will introduce the method in general.

Let $[a, b]$ be a closed interval and let f be a function defined on this interval. Moreover, f must be such that there is a point c in the interval $[a, b]$ such that f is decreasing on $[a, c]$ and increasing on $[c, b]$. We want to find the values c and $f(c)$, i.e., the minimum of f on the given interval.

We will approximate c by repeatedly shrinking the interval in which we search. Each iteration looks as follows:

1. Compute the values $x = (2a + b)/3$ and $y = (a + 2b)/3$. (Note that the values x and y split the interval $[a, b]$ into three equal parts, hence the name ternary search.)
2. Compute $f(x)$ and $f(y)$.
3. If $f(x) > f(y)$, the new interval will be $[x, b]$. Otherwise, the new interval is $[a, y]$.

Why does this method converge on the value we seek? Note that in the third step we drop one third of the search interval. We have to prove that it doesn't contain the point where f is minimal. For the first case: suppose that $c < x$. But this means that on the whole $[x, y]$ f has to be increasing, and thus we must have $f(x) < f(y)$. As this is not the case, we know that $c \geq x$, and thus we may really drop the interval $[a, x)$ from our search.

(If you are not sure whether you understand it correctly, try drawing a picture of the situation.)

Note that if originally $b - a = L$, then after N iterations we have a search interval of the length $L \cdot (2/3)^N$. Already for $N = 60$ this is approximately $3L10^{-11}$.

In our current problem, we want to find the minimum of the function g (defined above) on the interval $[0, \infty)$. Clearly it is enough to search in the interval $[0, Y]$, as we know that waiting for more than Y years won't give us an optimal solution. And we can easily verify (or guess) that the function g satisfies the conditions for ternary search to apply. To find the solution we simply iterate the search until we have a result that's precise enough:

```
double speed(double delay) {
  return Math.pow(2.0,delay / 1.5);
}

double necessaryTime(double taskLength, double delay) {
  return delay + taskLength / speed(delay);
}

public double shortestComputationTime(int years) {
  double taskLength = years;
  double mn = 0, mx = taskLength;
```

```
    for(int i=0; i<1000; i++) {
      double t1 = mn*2/3 + mx/3;
      double t2 = mn/3 + mx*2/3;
      if (necessaryTime(taskLength,t1) < necessaryTime(taskLength,t2))
        mx=t2; else mn=t1;
    }

    return necessaryTime(taskLength,mn);
  }
```

The Eryx approach

Another possible approach: Guess the approximate closed form of the function giving the answer, and use the values from the problem statement to find the constants such that our function interpolates the given values.

2.5 Coin Game

Also this problem admitted several different approaches to solving it. We will present some of them. We will concentrate on the main problem: given two sequences, determine the probability that the first one will occur earlier than the second one. The rest of the problem is easy.

Clearly we can't afford to repeatedly throw coins and wait for `HHTHTHTTTH` to appear, we have to think of something more clever.

Observation 1. Let L be the length of the given sequences. Suppose that we have a game in progress and that at least L throws were already made. Then the probability that the first player will win depends only on the outcome of the last $L-1$ throws.

This observation leads to the conclusion that any allowed game can be in one of roughly 500 different states. This number is still quite large, we have to think of something better. We will start with an example.

Observation 2. Suppose that the first player waits for `HHTHTHH`, and the second one waits for `HHHTHHH`. Let p be the probability that the first player wins given that the last six throws were `TTTTTH`. Let q be the probability that the first player wins given that the last six throws were `HTTTTH`. Then $p = q$. This is because in the current situation the first throw is useless, no player may use it in his sequence.

How to formalize the above observation? Let P be the set containing all prefixes of the two input strings. Then the state of the game is uniquely determined by the longest element of P that is the suffix of the sequence of previous throws. In other words, the state of the game tells us which player may use more previous throws, and how many of them.

(For the example in Observation 2, the state of the game corresponds to the string `H` – no player can use earlier throws to form his sequence. If the last six throws were `TTHHHT`, the state would be `HHHT`.)

We only have roughly 20 states in the game. Let's look at what happens to the state of the game when we throw a next coin.

For any string S let $crop(S)$ be the longest suffix of S that is in P. Thus if S contains enough previous throws, $crop(S)$ is the corresponding state. If we are in the state Z (that doesn't correspond to either player winning the game), after the next throw we will move either to $crop(Z + \texttt{'H'})$, or to $crop(Z + \texttt{'T'})$.

Using this information we may find the probability that player 1 wins as follows:

We will simulate the game and for each state keep the probability that after the current number of throws we are in this state. We can easily build a $|P|$ times $|P|$ matrix A such that multiplying the vector of probabilities by A corresponds to one step of our simulation. But then 2^{200} steps of the simulation correspond to multiplying by $B = A^{2^{200}}$. The matrix B can be computed using 200 matrix multiplications using repeated squaring. And 2^200 coin throws is quite a lot.

Gaussian elimination

The above approach alone probably wasn't fast enough to solve the largest inputs, however, with some time spent on optimizing your solution it could be made fast enough to (barely) pass the system tests.

However, if you see a simulation like this one, you should always ask: Can I compute the exact solution the simulation converges to? Usually, the answer is positive.

For each state (i.e., for each prefix Q in P) let $prob(Q)$ be the probability that from the state Q the first player will win the game. We want to know the value $prob(\texttt{""})$, and we can get it by computing all the values $prob(Q)$ at the same time.

For each state we can write one linear equation that describes what happens when we throw a coin in this state. As an example, consider the situation from Observation 2, the players wait for `HHTHTHH` and `HHHTHHH`, respectively. For the state `HHH` we get the equation: $prob(\texttt{"HHH"}) = 0.5\,prob(\texttt{"HHH"}) + 0.5\,prob(\texttt{"HHHT"})$. For the state `H` we get: $prob(\texttt{"H"}) = 0.5\,prob(\texttt{"HH"}) + 0.5\,prob(\texttt{""})$. For the state `HHTHTH` we get: $prob(\texttt{"HHTHTH"}) = 0.5 + 0.5\,prob(\texttt{""})$.

In this way we construct a set of $|P|$ linear equations with $|P|$ variables. These equations have an unique solution that can be found using Gaussian elimination. A (somewhat numerically stable) piece of code follows.

```
// initialize the matrix with C = |P| rows
double A[][] = new double[C][C+1];
for (int i=0; i<C; i++) for (int j=0; j<=C; j++) A[i][j] = 0.0;

// fill in the equations
for (int i=0; i<C; i++) {
  if (P[i].equals( firstString )) { A[p][p]=1.0; A[p][C]=1.0; continue; }
  if (P[i].equals( secondString )) { A[p][p]=1.0; A[p][C]=0.0; continue; }

  String T1 = crop( P[i]+'H' ), T2 = crop( P[i]+'T' );
  int x1 = getIndex( P, T1 ), x2 = getIndex( P, T2 );
```

```
  A[p][p] += 1.0; A[p][x1] -= 0.5; A[p][x2] -= 0.5;
}

// create a triangular matrix
for (int col=0; col<C; col++) {
  double max=1e-12; int kde=-1;

  // select a row where the element has the largest magnitude
  // note that for the first row, the first column will be selected
  for (int row=col; row<C; row++) {
    if (A[row][col] > max) { max=A[row][col]; kde=row; }
    if (-A[row][col] > max) { max=-A[row][col]; kde=row; }
  }
  // won't happen here, we have a regular set of equations
  if (kde==-1) return -47;

  // swap the selected row into its proper place
  if (kde>col) for (int c2=col; c2<=C; c2++) {
    double x=A[col][c2]; A[col][c2]=A[kde][c2]; A[kde][c2]=x;
  }
  if (A[col][col]<0) for (int c2=col; c2<=C; c2++) A[col][c2]=-A[col][c2];

  // subtract a suitable multiple of our row from the other ones
  for (int row=col+1; row<C; row++) {
    double mul = A[row][col] / A[col][col];
    for (int c2=col; c2<=C; c2++) A[row][c2] -= mul * A[col][c2];
  }
}

// substitute the values
for (int col=C-1; col>=0; col--) {
  A[col][C] /= A[col][col];
  for (int row=0; row<col; row++) A[row][C] -= A[row][col] * A[col][C];
}
```

Even more maths

The game is known under the name "Penney ante", and it is described e.g. in chapter 8.4 of the book Concrete Mathematics by Graham, Knuth and Patashnik. The probability that the first player wins can be computed in an even more simple way, simply by examining the patterns.