Pacific Northwest Region Programming Contest Division 1

SIMON FRASER UNIVERSITY

November 3rd, 2018

Reminders

- For all problems, read the input data from standard input and write the results to standard output.
- In general, when there is more than one integer or word on an input line, they will be separated from each other by exactly one space. No input lines will have leading or trailing spaces, and tabs will never appear in any input.
- Platform is as follows:

```
Ubuntu 18.04.1 LTS Linux (64-bit)
GNOME

vi/vim
gvim
emacs
gedit
geany
kate

Java OpenJDK version 10.0.2
C gcc 7.3.0
C++ g++ 7.3.0
Python 2.7.13 (implemented using PyPy 5.10.0).
CPython 3.6.6.
C# via Mono 5.10.1.57
Kotlin 1.2.70

Eclipse 4.8 (Photon), configured with:
   Java
   C/C++
   PyDev
IntelliJ (IDEA Community Edition 2018.2.4), configured with:
   Java
   Kotlin
CLion (version 2018.2.4), configured with
   C/C++
Pycharm Community Edition Python IDE version 2018.2.4
Code::Blocks (version 13.12+dfsg-4), configured with
   Java (OpenJDK version 1.8.0_131)
   C/C++ (CDT 9.3.0 with Ubuntu 5.4.0-6ubuntu116.04.4 5.4.0 20160609)
MonoDevelop 7.5.0.1254
Kotlinc 1.2.70
```

- Compiler options are as follows:

```
gcc -g -O2 -std=gnu11 -static $* -lm
g++ -g -O2 -std=gnu++14 -static $*
javac -encoding UTF-8 -sourcepath . -d . $*
python2 -m py_compile $*
python3 -m py_compile $*
mcs $*
kotlinc -d . ${files}
```

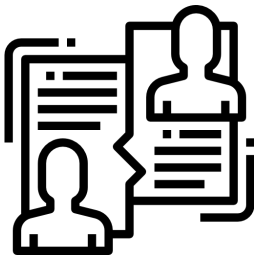
- Execution options are as follows:

```
java -Dfile.encoding=UTF-8 -XX:+UseSerialGC -Xss64m -Xms1920m -Xmx1920m $*
pypy $*
mono $*
kotlin -Dfile.encoding=UTF-8 -J-XX:+UseSerialGC -J-Xss64m -J-Xms1920m -J-Xmx1920m
```

- Python may not have sufficient performance for many of the problems; use it at your discretion. This is especially true of Python 3.

Problem A — limit 1 second

Exam



Your friend and you took a true/false exam of n questions. You know your answers, your friend's answers, and that your friend got k questions correct.

Compute the maximum number of questions you could have gotten correctly.

Input

The first line of input contains a single integer k.

The second line contains a string of n ($1 \leq n \leq 1000$) characters, the answers you wrote down. Each letter is either a '`T`' or an '`F`'.

The third line contains a string of n characters, the answers your friend wrote down. Each letter is either a '`T`' or an '`F`'.

The input will satisfy $0 \leq k \leq n$.

Output

Print, on one line, the maximum number of questions you could have gotten correctly.

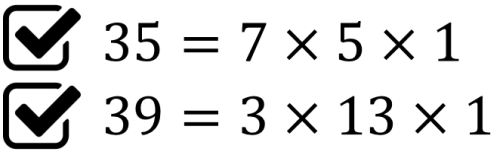
Sample Input and Output

3 FTFFF TFTTT	2

6 TTFTFFTFTF TTTTFFTTTT	9

Problem B — limit 1 second

Coprime Integers



Given intervals $[a, b]$ and $[c, d]$, count the number of pairs of coprime integers (x, y) such that $a \leq x \leq b$ and $c \leq y \leq d$.

Two numbers are coprime if their greatest common divisor is 1. A divisor of a number is a positive integer that evenly divides that number.

Input

The input consists of a single line containing space-separated integers a, b, c, and d ($1 \leq a \leq b \leq 10^7$, $1 \leq c \leq d \leq 10^7$).

Output

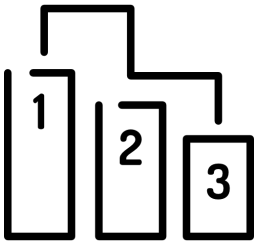
Print the result as a single integer on one line.

Sample Input and Output

1 5 1 5	19
12 12 1 12	4
1 100 1 100	6087

Problem C — limit 1 second

Contest Setting



A group of contest writers have written n problems and want to use k of them in an upcoming contest. Each problem has a *difficulty level*. A contest is valid if all of its k problems have different difficulty levels.

Compute how many distinct valid contests the contest writers can produce. Two contests are distinct if and only if there exists some problem present in one contest but not present in the other.

Print the result modulo 998,244,353.

Input

The first line of input contains two space-separated integers n and k ($1 \leq k \leq n \leq 1000$).

The next line contains n space-separated integers representing the difficulty levels. The difficulty levels are between 1 and 10^9 (inclusive).

Output

Print the number of distinct contests possible, modulo 998,244,353.

Sample Input and Output

5 2 1 2 3 4 5	10

5 2 1 1 1 2 2	6

12 5 3 1 4 1 5 9 2 6 5 3 5 8	316

Problem D — limit 1 second

Count The Bits

Given an integer k and a number of bits b ($1 \leq b \leq 128$), calculate the total number of 1 bits in the binary representations of multiples of k between 0 and $2^b - 1$ (inclusive), modulo 1,000,000,009.

Input

The input will consist of two integers k and b on a single line, with $1 \leq k \leq 1000$ and $1 \leq b \leq 128$.

Output

Write your result as an integer on a single line.

Sample Input and Output

1 4	32
10 5	8
100 7	3
3 28	252698795

11 128	856188165

1 26	872415232

876 128	530649653

Problem E — limit 5 seconds

Cops And Robbers

The First Universal Bank of Denview has just been robbed! You want to catch the robbers before they leave the state of Calirado.

The state of Calirado is a perfect grid of size m-by-n. The robbers will try to escape by going from grid square to grid square (only through edges, not corners) until they reach the border of the state.

You can barricade some grid squares to stop the robbers from using them. However, depending on the terrain, different grid squares may cost different amount of money, and some grid squares may not be barricaded at all!

Find the cheapest set of grid squares to barricade that guarantees no escape for the robbers.

Input

The first line of input contains three space-separated integers n, m, and c ($1 \leq n, m \leq 30$, and $1 \leq c \leq 26$), where n and m are the dimensions of the grid, and c is the number of types of terrain in Calirado.

Each of the next m lines contains n letters each, representing the grid.

- Character '`B`' indicates the First Universal Bank of Denview; it is where the robbers currently are.
- Characters '`a`' through '`z`' represent the different types of terrain. Only the first c alphabets will appear in the grid.
- A dot ('`.`') represents a grid square that cannot be barricaded.

It is guaranteed that the grid will contain exactly one `B` character.

Finally, the last line of the input contains c space-separated integers between 1 and 100,000 (inclusive), representing the cost of barricading a single grid square of type '`a`', '`b`', and so on.

Output

Print, on one line, the minimum total cost of barricading plan that guarantees no exit for the robbers.

If there is no way to prevent the robbers from escaping, print `-1` instead.

In the first example, the minimum cost is to barricade the central three squares on each side for a total cost of 12.

In the second example, since the bank is on the border, and we cannot barricade the bank, there is no way to prevent the robbers from escaping the state.

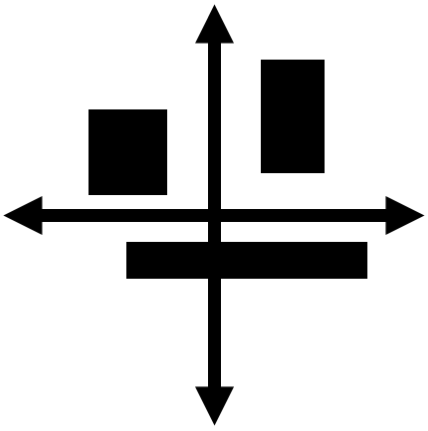
Sample Input and Output

5 5 1 aaaaa a...a a.B.a a...a aaaaa 1	12

2 2 1 aB aa 1	-1

Problem F — limit 2 seconds

Rectangles



You are given several axis-aligned rectangles. Compute the sum of the area of the regions that are covered by an odd number of rectangles.

Input

The first line of input contains a single integer n ($1 \leq n \leq 10^5$), representing the number of rectangles.

Each of the next n lines contains four space-separated integers x_1, y_1, x_2, and y_2, each between 0 and 10^9, describing the coordinates of a rectangle.

Output

Print, on one line, the total area covered by an odd number of rectangles as an exact integer.

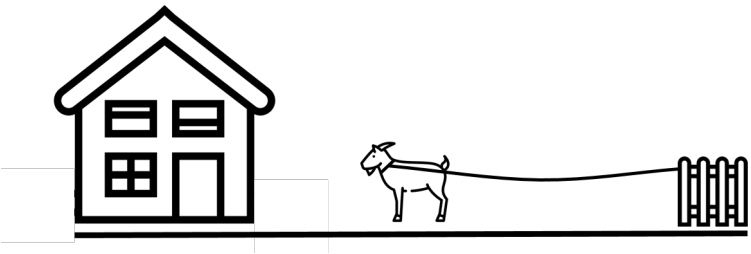
Sample Input and Output

2 0 0 4 4 1 1 3 3	12

4 0 0 10 10 1 1 11 11 2 2 12 12 3 3 13 13	72

Problem G — limit 1 second

Goat on a Rope



You have a house, which you model as an axis-aligned rectangle with corners at (x_1, y_1) and (x_2, y_2).

You also have a goat, which you want to tie to a fence post located at (x, y), with a rope of length l. The goat can reach anywhere within a distance l from the fence post.

Find the largest value of l so that the goat cannot reach your house.

Input

Input consists of a single line with six space-separated integers x, y, x_1, y_1, x_2, and y_2. All the values are guaranteed to be between -1000 and 1000 (inclusive).

It is guaranteed that $x_1 < y_1$ and $x_2 < y_2$, and that (x, y) lies strictly outside the axis-aligned rectangle with corners at (x_1, y_1) and (x_2, y_2).

Output

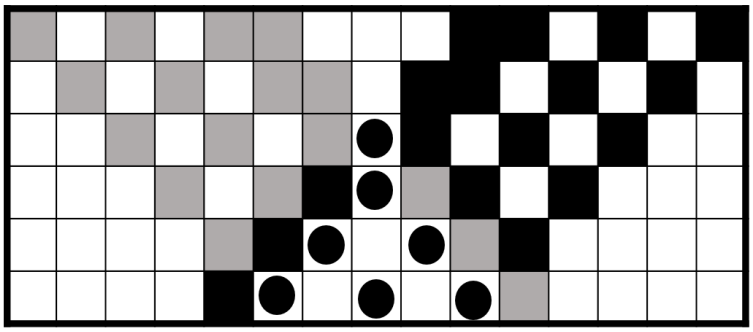
Print, on one line, the maximum value of l, rounded and displayed to exactly three decimal places.

Sample Input and Output

7 4 0 0 5 4	2.000
6 0 0 2 7 6	2.000
4 8 7 8 9 9	3.000

Problem H — limit 1 second

Repeating Goldbachs



The Goldbach Conjecture states that any even number $x \geq 4$ can be expressed as the sum of two primes. It can be verified that the conjecture is true for all $x \leq 10^6$.

Define a *Goldbach step* as taking x ($4 \leq x \leq 10^6$), finding primes p and q (with $p \leq q$) that sum to x, and replacing x with $q - p$. If there are multiple pairs of primes which sum to x, we take the pair with the largest difference. That difference must be even and less than x. Therefore, we can repeat more Goldbach steps, until we can reach a number less than 4.

Given x, find how many Goldbach steps it takes until reaching a number less than 4.

Input

The input will consist of a single integer x ($4 \leq x \leq 10^6$).

Output

Print, on a single line, the number of Goldbach steps it takes to reach a number less than 4.

Sample Input and Output

20	3
30	4
40	5

50	6
60	7
70	8

Problem I — limit 5 seconds

Inversions

You are given a sequence $p = (p_1, \ldots, p_n)$ of n integers between 1 and k, inclusive. Some of the integers are not determined, but they must still be between 1 and k.

An inversion is a pair of indices (i, j) with $i < j$ and $p_i > p_j$.

Find the maximum possible number of inversions in p obtained by replacing the undetermined numbers with values between 1 and k. Note that the numbers can be duplicated.

Input

The first line of input contains two space-separated integers n and k ($1 \leq n \leq 200{,}000$ and $1 \leq k \leq 100$). The next n lines describe the sequence p, and each contains a single integer between 0 and k, inclusive. The value 0 represents an undetermined number.

Output

Print, on a single line, the maximum possible number of inversions.

Sample Input and Output

6 9 0 8 4 3 0 0	15

10 9 5 2 9 0 7 4 8 7 0 0	28

10 9 7 4 0 0 8 5 0 0 3 1	36

Problem J — limit 1 second

Time Limits

A contest setter wants to determine the time limits for a given problem. There are n model solutions, and solution k takes t_k milliseconds to run on the test data. The contest setter wants the time limit to be an integer number of seconds, and wants the time limit to be at least s times larger than the slowest model solution. Compute the minimum time limit the contest setter can set.

Input

The first line of input contains two space-separated integers n and s ($1 \leq n \leq 100$ and $1 \leq s \leq 20$).

The second line of input contains n space-separated integers $t_1, \ldots, t_n$ ($1 \leq t_k \leq 2000$ for all $k = 1, \ldots, n$).

Output

Print, on one line, the minimum time limit (in seconds) as a single integer.

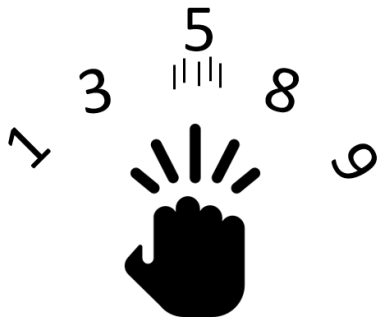
Sample Input and Output

2 5 200 250	2

3 4 47 1032 1107	5

Problem K — limit 2 seconds

Knockout



The solitaire game Knockout is played as follows. The digits from 1 to 9 are written down on a board, in order. In each turn, you throw a pair of six-sided dice. You sum the dice and cross out some set of digits that sum to the same total. If you cannot, the game ends and your score is the concatenation of the remaining digits. Otherwise, you throw the dice again and continue.

This game can be played to either minimize or maximize your score. Given a position of the game (what digits remain) and a roll of the dice, compute which digits you should remove and what your expected total score is for both minimizing and maximizing versions of the game.

Input

The input contains a single line with three space-separated integers. The first is the board state, containing some nonempty subset of the digits 1 through 9, in order. The next two integers are numbers between 1 to 6 (inclusive), representing the roll of the two dice.

Output

On the first line of output, print the digit(s) you should remove to minimize the expected score, followed by the expected score. On the second line of output, do the same for the maximizing version of Knockout.

If multiple digits are removed, list the digits in order with no space separating them. If you cannot remove digits to match the sum of the dice, print '`-1`' for your move instead.

The expected scores should be printed with exactly five digits after the decimal point.

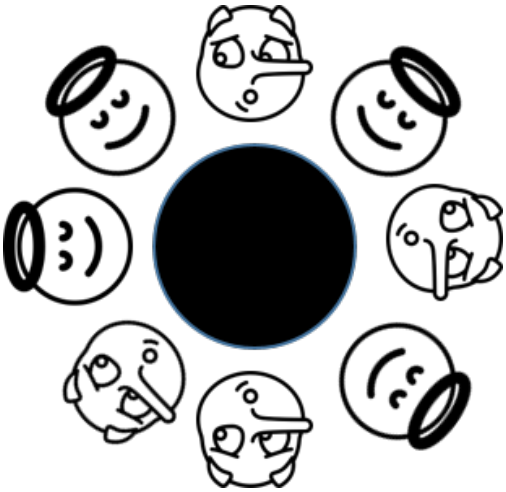
Sample Input and Output

1345 1 1	-1 1345.00000 -1 1345.00000

12349 3 1	13 151.70370 4 401.24546

Problem L — limit 1 second

Liars



There are n people in a circle, numbered from 1 to n, each of whom always tells the truth or always lies.

Each person i makes a claim of the form: "the number of truth-tellers in this circle is between a_i and b_i, inclusive."

Compute the maximum number of people who could be telling the truth.

Input

The first line contains a single integer n ($1 \leq n \leq 10^3$). Each of the next n lines contains two space-separated integers a_i and b_i ($0 \leq a_i \leq b_i \leq n$).

Output

Print, on a single line, the maximum number of people who could be telling the truth. If the given set of statements is inconsistent, print `-1` instead.

Sample Input and Output

3 1 1 2 3 2 2	2

8 0 1 1 7 4 8 3 7 1 2 4 5 3 7 1 8	-1

Problem M — limit 1 second

Mobilization

In some strategy games you are required to mobilize an army. There are n troops to choose from, each of which has a unit cost c_i, health h_i, and potency p_i. You can acquire any combination of the troop types (even fractional units), such that the total cost is no more than C. The *efficacy* of the army is equal to its total health value, multiplied by its total potency. What is the greatest efficacy you can achieve given the troops available?

You may assume that there are always sufficient troops to buy as many as you want (subject to the total cost constraint).

Input

The first of input consists of two space-separated integers n and C ($1 \leq n \leq 30{,}000$ and $1 \leq C \leq 100{,}000$).

Each of the next n lines starts with an integer c_i ($1 \leq c_i \leq 100{,}000$), followed by two decimal values h_i and p_i ($0.0 \leq h_i, p_i \leq 1.0$). The numbers are space-separated.

Output

Print, on a single line, the greatest possible efficacy, with exactly two digits after the decimal point.

Sample Input and Output

4 100000 300 1 0.02 500 0.2 1 250 0.3 0.1 1000 1 0.1	19436.05

2 100 1 0.1 1 1 1 0.1	3025.00

Count The Bits

This problem was a dynamic programming problem based on a function that maps the count of bits and the current value modulo k to a sum of bits. The trick here was realizing that to calculate the sum of bits for values divisible by k, we gain speed by calculating the sum of bits for values equivalent to any j modulo k. That is, we expand the problem somewhat to permit a simple recursive formulation.

We define $f[i][j]$ to be the total number of bits in all numbers with i bits, that are equal to j modulo k. Further, we define $c[i][j]$ to be the count of numbers with i bits that are equal to j modulo k. Clearly $f[0][j]$ and $c[0][j]$ are all equal to zero, except that $c(0,0)$ is equal to one. What happens if we add a zero bit to the right? The modulo value is multiplied by two. What happens if we add a one bit to the right? The modulo value is multiplied by two, and a one is added. Equivalently we can add a bit to the left, which involves tracking the value of that bit modulo k.

So we iterate over the count of bits. From $c[i][j]$ we contribute $c[i][j]$ more bits to $c[i+1][2j \bmod k]$ and also to $c[i+1][(2j+1) \bmod k]$. From $f[i][j]$ we contribute $f[i][j]$ to $f[i+1][2j \bmod k]$, and $f[i][j] + c[i][j]$ to $f[i+1][(2j+1) \bmod k]$.

In the end we read our final result from $f[b][0]$.

Contest Setting

This problem is solved using dynamic programming, with the two dimensional state space mapping the number of difficulty classes considered and the number of distinct problems required into the total number of contests possible.

So we start by calculating how many different problems there are for a given difficulty. We can do this with a hash map. For the third input, which was

```
12 5
3 1 4 1 5 9 2 6 5 3 5 8
```

we end up with the following difficulty counts:

```
1:2 2:1 3:2 4:1 5:3 6:1 8:1 9:1
```

After calculating these counts the actual difficulty doesn't matter at all; we can rearrange this into an array v giving just the counts:

```
[2 1 2 1 3 1 1 1]
```

Define $f(i, j)$ to be the number of contests possible with j different problems using problems from just the first i classes. There's only a single distinct contest with no problems, no matter how many problem classes we consider:

$$f(i, 0) = 1$$

Further, there are no contests possible with at least one problem if there are no problem categories:

$$f(0, j) = 0 \qquad (j > 0)$$

Given a contest with j problems from the first i categories, we can expand it to a contest with $j+1$ problems from the first $i+1$ categories by adding any of the $v[i]$ problems in category i, or we can just ignore that category altogether:

$$f(i+1, j+1) = v[i]f(i, j) + f(i, j+1) \qquad (i, j > 0)$$

We can implement this using recursion with memoization or with iterative dynamic programming.

Coprime Integers

The given limits should have made it clear that calculating the greatest common divisor explicitly for all pairs would not nearly have run in time. Instead, we use inclusion-exclusion on the square-free numbers.

Roughly started, we start with the set of all pairs. We subtract all pairs where both numbers are divisible by 2. We subtract all pairs where both numbers are divisible by 3. We don't need to worry about all pairs divisible by 4, because we eliminated them all when we eliminated all numbers divisible by 2. We subtract all pairs where both numbers are divisible by 5.

Now, think about all pairs where both numbers are divisible by 6. We subtracted them once when we handled all pairs divisible by 2, and once again when we handled all pairs divisible by 3, so we have already subtracted these numbers twice. We need to correct for this by adding back the count of all pairs that are divisible by 6.

And this continues. For every square-free integer k, we either add or subtract all pairs divisible by k. To determine whether to add or subtract, we count the number of distinct prime factors in k; if that number is even, we add; if it is odd, we subtract.

How do we determine how many pairs divisible by k there are with the first number between a and b inclusive, and the second number between c and d inclusive? This is just

$$(\lfloor\frac{b}{k}\rfloor - \lfloor\frac{a-1}{k}\rfloor)(\lfloor\frac{d}{k}\rfloor - \lfloor\frac{c-1}{k}\rfloor)$$

To generate the square-free numbers, we can either test each individually, but that will usually be too slow. Instead, we generate all primes less than a million using a sieve, and then use a simple recursion that either includes or excludes each prime, stopping with the product gets larger than the maximum of b and d. As we generate we can also accumulate the count of primes in each number, so we know whether to add or subtract.

Cops And Robbers

This problem was undisguised minimum-cut (equivalent to maximum flow). Each square gets two nodes, one for entering the square and one for exiting the square, with an edge between them of capacity equivalent to the cost of barricading that square. All squares on the edges evacuate to the sink, and the source was the bank square.

The problem was of sufficient size that a basic Ford-Fulkerson flow algorithm probably would not complete in time, but Edmond-Karp or similar algorithms easily do.

Exam

You want to know how many problems you answered differently from your friend (d), and how many you answered the same e. To maximize your score you allocate as much of your friend's score s to the ones you answered the same (as many as possible), and then the rest to the ones you answered differently. The final result is then

$$\min(e, s) + \min(n - e, n - s)$$

Goat on a Rope

There were multiple ways to solve this problem. One was to enumerate all possible eight cases for where the goat stake could be, and calculate the distance there. Another was to treat the house as a large origin, and essentially compress it to a point.

The problem with enumerating cases is it's very easy to get one wrong.

Ultimately a single line worked for this problem.

```
hypot(max(x1-x, x-x2, 0), max(y1-y, y-y2, 0))
```

Greedy Scheduler

Let's simulate the queue, processing all customers in order from front to back.

At $t = 0$, all of the cashiers are unoccupied and ready to take a customer. Therefore, the first customer will go to cashier 1, occupying that register until $t = t_1$.

In general, say we're about to process customer i. By now we've computed that, due to the shopping carts of customers 1 to $i - 1$, each cashier j is occupied until some time $T(i - 1, j)$, at which point they will become free to take a new customer. Therefore, customer i will go to the cashier j with smallest $T(i-1, j)$, breaking ties by smallest j. As a result, this cashier will become occupied for an additional t_i seconds.

Implementing such a recurrence using saved values is called dynamic programming. It yields a solution with time and memory complexity $O(nc)$, which is sufficient for the provided constraints. The memory complexity can be reduced to $O(n)$ by noticing that there's no need to keep track of $T(i, j)$ for old values of i. Thus, the index i can be suppressed

Finally, we can speed up the search for a minumum time using a partially sorted data structure, such as a balanced binary search tree or min heap. The final run-time is $O(c \log n)$. The limits on this problem were sufficiently relaxed so this was not necessary in this case.

House Numbers

The intended solution here was a simple simulation. Since we are given m, we can simply start with $x = m + 1$ and $n = m + 2$, and accumulate a sum of the numbers from m to $x - 1$ and from m to n. For each x starting with $m + 1$ and increasing by 1, we add $x - 1$ to the sum of houses to the left of us, then increase n (adding it to the sum of the houses to our right) until that sum is greater than or equal to twice the sum to the left of us, plus x. If we hit equality, we are done, else we continue to the next x.

The problem statement guarantees there is a solution that we will find quickly.

Inversions

The unknown values should always be replaced by a nonincreasing sequence. This is easily proved by contradiction. So we just need to find that nonincreasing sequence. A simple dynamic programming approach works, where the state space is (position, rightmost unknown value), working left to right.

The judge solution does it slightly differently; it first calculates the result assuming only the lowest value is available for substitution. Then, using the intermediate results it calculates results assuming the lowest two values are available, and so on and so forth. The state space is the same as the previous approach but the order of computation may be different.

Knockout

The state space for this problem, even starting with a full board, was small enough that a simple recursive solution worked with no need for memoization or dynamic programming. To calculate the expected value, you needed to calculate the probability of each throw and then recursively determine the best actions based on the throw, and return the result up to the top.

One trick to make the same code work for both the minimize and maximize cases was to write the code to maximize the final result, and to compute the minimize case, run it again but negate the final score value in the base case if no result was possible.

Liars

This problem sounds like it requires a complicated solution strategy, but in reality all you need to do is try all the possibilities. For any given number of liars, you can easily check how many people are telling the truth, and see if that is a possible solution. So iterate from 1 to n, check each value to see if it works, and print that value if it does. If no values work, then the statements are mutually inconsistent, so print –1.

Mobilization

Every type of unit can be visualized on a 2D plane with potency vs health (per unit dollar for both). Any linear combination of types of units will lie within the convex hull of these points. The maximum product will also lie on the convex hull, either on an extreme point or on a line of the hull.

So the solution is to compute the hull, and then consider each point and each line in the hull. Optimizing the product of xy on a line in the plane is straightforward calculus, or alternatively it can be done using search.

Interestingly, none of the judge solutions properly dealt with repeated data points, and thus some of the judge data was broken. Luckily this was caught and the data fixed before it affected the contest.

Paper Cuts

This problem was a dynamic programming problem. The function being optimized mapped the used characters in the first string and the final character position from the first string chosen to the fewest number of cuts required to get to that state. For n input characters the total state space was somewhat less than $n2^n$, which for 18 is sufficiently small to easily run in time.

The set of characters already used should be represented as a bitmask in a single integer, rather than a more complicated data structure (like a set of characters); this "bitmap DP" is a standard trick in programming contests.

No additional optimizations were necessary.

Pizza Deal

For both the slice and the whole pizza, simply calculate which has a larger pizza-to-cost ratio. Since the input is integral, and the area of a whole pizza is always π times an integer and thus irrational, it's impossible for there to be a tie. You need to know the formula for the area of a circle given its radius.

Poker Hand

We need to find which rank is represented the most. An easy way to do this is to iterate through all the given cards, and for each one, iterate through all given cards again counting how many match in rank. Return the maximum of this value.

Random Index Vectors

Solving this problem required implementing the operations on Random Index Vectors as described, without expanding the vector to its uncompressed size since that would be too slow. The code that iterated over a pair of RIVs, finding and processing both matching and unmatched indices, is a bit tedious; using functional programming techniques to share this code could simplify the code a bit.

Rectangles

Each vertical line has some x coordinate, so we only need to consider the sweep line right before and right after these x values. For one x value, we multiply the length of the sweep line that had an odd number of rectangles by the previous interesting x value. We then process vertical lines at that position by flipping the parity of some range on the sweep line. We can then continue onto

the next interesting x coordinate.

To simulate these operations efficiently, build a segment tree on the sweep line representing the current parity on each region. Since there are only $O(n)$ interesting y values they could cut the sweep line, there are only n regions. Each vertical line flips some consecutive sequence of regions, which can be done with lazy propagation. We also need to know the length of regions which is odd which can also be maintained by the tree.

Repeating Goldbachs

This problem is just a simulation problem; execute the process described to get the result.

You need a prime sieve to generate the primes less than one million quickly. Then, you just need to scan the primes with two iterators, one increasing and one decreasing, at each step.

The difficult part of this problem was realizing that the process is quadratic if not written carefully. You want to initialize the upper iterator to be the smallest prime not greater than half the current value being split, and you want to initialize the lower iterator to be the largest prime not less than half the current value being split. Scanning the list from either the front or the back for this point is too slow. An easy way to do this is to realize that the required point in the prime list is very close to what it was last time, so searching nearby the most recent point works well.

Time Limits

First, find the slowest solution; this is the one that sets the time limit. You want to calculate the smallest value r that is at least s times the input value, when the input value is given in milliseconds. To calculate this, multiply s times the current input value, and then round up to the nearest second. This can be tricky to get precisely right, but it's not too bad; if the maximum input is v, the result is just

$$\left\lfloor \frac{sv + 999}{1000} \right\rfloor$$