Code Jam 2012 - Round 1A

Analysis: Password Problem

The first challenge with Password Problem is wrapping your head around expected values. These things come up all the time and are extremely useful, so they are well worth learning. (For example, expected value is key to solving our last problem on the 2011 finals!)

Once you understand what the problem is asking, you mainly need to evaluate the expected number of keystrokes for each strategy:

- *Strategy 1:* Finish typing the password, then press "enter".

  The probability that this works on the first try is $x = p_1 * p_2 * ... * p_{\mathbf{A}}$. In this case, you need **B** - **A** + 1 keystrokes. In the other case, you need 2**B** - **A** + 2 keystrokes. Therefore, the expected number of keystrokes for this strategy is:

  B - **A** + 1 + (**B** + 1) * (1 - x).

- *Strategy 2:* Hit backspace k times, then proceed as in Strategy 1.

  The probability that this works on the first try is $x_k = p_1 * p_2 * ... * p_{\mathbf{A}-k}$. In this case, you need **B** - **A** + 2k + 1 keystrokes. In the other case, you need 2**B** - **A** + 2k + 2 keystrokes. Therefore, the expected number of keystrokes for this strategy is

  B - **A** + 2k + 1 + (**B** + 1) * (1 - x_k).

- *Strategy 3:* Press enter immediately, and retype the whole password.

  This always takes **B** + 2 keystrokes.

The problem is asking you to calculate the minimum of all these values. There is one more catch though: if you compute each x_i from scratch, your program will probably be too slow. It might take 99999 multiplications to calculate x, 99998 multiplications to calculate x_1, 99997 multiplications to calculate x_2, and so on. Instead, you should calculate them all together:

- $x_{\mathbf{A}} = 1$
- $x_{\mathbf{A}-1} = x_{\mathbf{A}} * p_1$
- $x_{\mathbf{A}-2} = x_{\mathbf{A}-1} * p_2$
- etc.

Making sure your solution is fast enough to complete in the time limit is an important part of the Google Code Jam!

Here is a short Python solution:

```
import sys

for tc in xrange(1, int(sys.stdin.readline())+1):
  A, B = [int(w) for w in sys.stdin.readline().split()]
  p = [float(w) for w in sys.stdin.readline().split()]
  best, x = B + 2.0, 1
  for i in xrange(A):
    x *= p[i]
```

```
    best = min(best, (B - i) + (A - i - 1) + (B + 1) * (1 - x))
  print 'Case #%d: %f' % (tc, best)
```

The Large Input and Underflow

There is one trick on this problem that caught quite a few contestants, and that involves a subtlety with how floating point numbers work. Let's suppose you calculate x_i in a slightly different way:

- $x = p_1 * p_2 * ... * p_{\mathbf{A}}$
- $x_1 = x / p_{\mathbf{A}}$
- $x_2 = x_1 / p_{\mathbf{A}-1}$
- etc.

At first glance, this looks completely equivalent to the solution above. Unfortunately, it is wrong to do this for two reasons. First, if one of the p_i values is 0, you are in trouble. Second, it turns out that even a 64-bit floating point number reserves only 11 bits for the exponent. This means that it cannot store values much less than 2^{-1000}. These very small values get rounded down to 0. Normally, you wouldn't care about values this small, but if you are multiplying 100,000 probabilities together, it becomes an issue. After rounding to 0, you will end up with every x value, including $x_{\mathbf{A}}$, being 0, and then you are in trouble!

Some people reported failing tests due to this bug, and were then able to fix it by switching to the "long double" data type in C++. However, they were lucky! Even long double has the same issues.

If you ever need to combine a lot of multiplications and divisions in the future, where intermediate values might get very small, it helps to work in log space:

```
A * B / C = exp( log(A) + log(B) - log(C) )
```

Do you see why this approach avoids the problem?