Philadelphia Classic 2007

The Dining Philosophers
Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
dp-exec@googlegroups.com
http://dp.seas.upenn.edu/pclassic/

Sponsored by Microsoft

Saturday, February 10, 2007

1. **Kullback-Leibler Divergence**

In the fields of artificial intelligence and machine learning, in which we are trying to teach computers how to learn and how to make decisions, the modern approach is to apply the techniques of probability theory and statistics. Probability is used as a mathematical framework to allow us to deal with the inherent uncertainty of decision making. When we ask a computer to *predict* (based on the last 100 days of weather information, what will tomorrow's average temperature be?) or *classify* (based on all the email I've received, is this new email spam or not?), we represent beliefs as a *probability distribution* over all possible options. For example, in the case of spam filtering, we would compute (based on past evidence) the probability that the email is spam and the probability that an email is not spam and then use the higher one to classify the email. As the filter sees more examples, it converges to a "true" probability distribution.

It is then useful to have a notion of *distance between probability distributions*. In particular, the *Kullback-Leibler divergence* from the field of information theory provides a kind of distance measure from some true distribution P to some other distribution Q. Generally, Q is the approximation to P that our model is currently working with. The KL divergence tells us how many mistakes we might expect to make if we think the distribution is Q instead of P.[1] The divergence between identical distributions is zero.

In this problem, you must compute the KL divergence between two distributions. A distribution is specified as a list of numbers between 0 and 1 that sum to 1, such as `0.3 0.2 0.5`; first P will be given, then Q. If $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$, then the KL divergence[2] is defined as

$$D(P||Q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}.$$

As an example, if $P = (0.2, 0.8)$ and $Q = (0.4, 0.6)$, then

$$D(P||Q) = 0.2 \log \frac{0.2}{0.4} + 0.8 \log \frac{0.8}{0.6} = 0.0915.$$

Some more examples are given below; be sure your answers are to at least 3 decimal places. Note that logarithms are with base 2. Assume that $0 \log 0 = 0$ and that $p \log \frac{p}{0}$ = Infinity, and keep in mind that adding anything to infinity or multiplying anything by infinity gives infinity. You should print out "Infinity" if the divergence does turn out to be infinite. Each distribution is given as a separate String argument so you can easily tell them apart.

Examples

Input: `"0.2 0.8" "0.4 0.6"`
Output: `0.13202999942307519`

Input: `"0.2 0.3 0.5" "0.5 0.2 0.3"`
Output: `0.27958592832197748`

Input: `"1.0 0.0" "0.0 1.0"`
Output: `Infinity`

[1] There are more concrete ways of justifying the KL divergence as a reasonable dissimilarity measure, but these are based on coding theory and information theory and so we do not go into them here.

[2] The $D(P||Q)$ notation is standard for reasons passing understanding.

2. The Voting Paradox and Arrow's Impossibility Theorem[3]

The *voting paradox* (or *Condorcet's paradox*) is an anomaly in voting systems originally noticed by the Marquis de Condorcet in the 18th century. The importance of the paradox was not fully realized until Kenneth Arrow (Professor Emeritus of Economics, Stanford University; Winner of the 1972 Nobel Prize in Economics) published *Social Choice and Individual Values* in 1951. This monograph contained *Arrow's impossibility theorem*, a famous result which shows that no voting system based on ranked preferences can produce a fair outcome when there are three or more options to choose from.[4]

A preference ordering on candidates is written in the form (C, A, B), which means that this voter prefers candidate C to A to B in that order. The voting paradox says that majority wishes can be in conflict with each other, so there can be no logical, fair way to aggregate individual preferences into a single social preference. If we have three candidates $\{A, B, C\}$ and three voters with preferences $(A, B, C), (B, C, A)$, and (C, A, B) respectively, then there are always two voters who prefer one candidate to another. If A is the winner, one could argue that C should have won instead, since two out of three voters prefer C to A; similar arguments can be made for all three candidates, so majority rule provides no clear winner. (The impossibility theorem establishes the much stronger result that *no* voting system can aggregate preferences with at least two voters and at least three candidates in a fair way.)

In this question, you will write a program that evaluates different aggregation strategies on voter preferences. Only 3 candidates will be considered. In situations where a voter gets only one choice, he makes the top choice; if there are ballots between only two of the three candidates, it's assumed that each voter will vote for the one he prefers. Consider the following five schemes:

1. Plurality winner: there is one ballot, and each voter casts their vote for their favorite candidate. Whoever gets the most votes wins, even if they don't get a majority of the votes cast.
2. Exhaustive ballot: on each ballot, each voter casts their vote for their favorite candidate (that is on the ballot). The candidate who gets the fewest votes is eliminated, and the survivors move on to the next round of voting. For three candidates, there will be just two ballots.
3. $1-2$ primary, $1-3$ primary, and $2-3$ primary: Candidates 1 and 2 (or 1 and 3, or 2 and 3, respectively) face off in a primary ballot. Whoever wins goes on to face candidate 3 (or 2 or 1) in a second ballot. Whoever wins that vote wins the election.

The input will be a list of preference orderings (each written `2 1 3`) separated by commas. Print the winners under each of the five strategies in the following order: plurality winner, exhaustive ballot, $1-2$ primary, $1-3$ primary, $2-3$ primary. Note in the examples below that we give all the votes as one big String argument where each preference ordering is separated by a comma followed by a space.

Examples

Input: `"1 2 3, 1 2 3, 1 2 3, 2 1 3, 2 3 1, 3 1 2"`
Output: `1 1 1 1 1`

Input: `"1 3 2, 1 3 2, 1 3 2, 1 2 3, 1 3 2, 3 2 1, 2 3 1, 2 3 1, 2 3 1, 3 2 1, 3 2 1, 2 3 1"`
Output: `1 2 3 3 3`

[3] Idea from Lorrie Cranor, Washington University.

[4] If you are interested in finding out more, these topics are considered in detail by the field of economics, and the areas of welfare economics, social choice theory, and voting theory in particular.

3. **Large Numbers**

When we write computer programs, we are limited by the number of bits that can be held in memory at a single location (called a *register*, in this case). The number of bits a register can hold is called the *register width*. We will not go into the binary representation of integers here, but in general, an n-bit register can represent integers up to $2^n - 1$. In particular, a 32-bit register, the most common width for personal computers currently, can represent integers up to 4294967295, or about 4.2 billion.

We often need to manipulate much larger numbers (companies – and some rich people – often deal with sums in the tens of billions of dollars, for example), but regular `int` or `long` variables in Java (or other languages) cannot store numbers of this kind; another representation is required. In this question you must write a program that can multiply two arbitrarily large integers. (If it helps you, you can assume for sake of concreteness that none of the integers given as input will exceed 30 digits. Obviously, you are not allowed to use the Java `BigInteger` class.)

Examples

Input: `89172398172391 1298310928310`
Output: `115773499050825906622289210`

Input: `1782763122673676373 18329929271717282`
Output: `32677921946834328228791434019178186`

4. **Longest Palindrome**

A *palindrome* is a string that looks the same when read forward or backward, such as 'ada', 'lion oil', or '1232321'. For this problem, you must write a program that finds the *longest palindrome inside a string*. For example, if the program receives "I don't like lion oil." as input, it should be able to find "lion oil" as the longest palindrome contained in the string. Spaces should be ignored, but not punctuation; this means that you *do not* need to print out spaces in palindromes when you print your solutions. The string `lionoil` is an acceptable answer for input "lion oil". If there are ties, as in "lion oil 1232321" (both are 7 characters), you may return either one of the matches. If a string contains no palindromes longer than 1 character, you may return any character in the string.

Examples

Input: `"I dislike lion oil"`
Output: `lionoil`

Input: `abdcdcdcada`
Output: `dcdcd`

Input: `828281193939`
Output: `93939` *or* `82828`

Input: `abc`
Output: `a` *or* `b` *or* `c`

Input: `a'ba`
Output: `a` *or* `b` *or* `'`

5. **Regular Polygons**

A *regular polygon* is a polygon with all internal angles equal and all sides the same length. Squares, equilateral triangles, regular pentagons, and regular hexagons are some of the familiar regular polygons of small size. In this question, you must write a program that prints out the coordinates of the "corners" of a regular polygon with n sides. This is a simplified version of the kind of tasks that are commonplace in modern video games, among other things.

Assume that one edge is between $(0, 0)$ and $(1, 0)$ (so the length of every side is 1), and that the coordinates should be output in **counterclockwise** order. Thus, for $n = 4$ (a square), we would output the coordinates `(0,0), (1,0), (1,1), (0,1)`. Your answers should be accurate to at least 3 decimal places, and printing out things like `0.50000000001` instead of `0.5` is fine.

Note: in Java, if your solution is correct, you should be able to get exactly `1.0` as the output much of the time (occasionally it may be something like `1.0000000000000002`); in C or C++, however, you may get things like `0.9999999999999` instead. The graders will take this into account, so you don't need to worry about not being accurate to three decimal places in that case.

Examples

Input: `3`
Output: `(0, 0), (1, 0), (0.5, 0.866)`

Input: `4`
Output: `(0, 0), (1, 0), (1, 1), (0, 1)`

Input: `5`
Output: `(0, 0), (1, 0), (1.309, 0.951), (0.5, 1.538), (-0.309, 0.951)`

6. **ISBN Numbers**

ISBN numbers are the numbers next to the barcode on books that uniquely identify a book. Decades ago, when orders used to be placed by telephone, people would try to identify the book they wanted to buy from a retailer, but the signal would often get garbled over the telephone. The ISBN number system uses a built-in check so the number itself can be used to verify that the code is valid.

The last digit of a 10 digit ISBN number is a check digit or *checksum*. It can take any value between 0 and 10, where 10 is represented by the letter X. We multiply each number in the code (except the check digit) by numbers from 1 to 9 and add up the results. The remainder of this quantity after being divided by 11 must be equal to the check digit. (More formally, the dot product of $(1, 2, 3, 4, 5, 6, 7, 8, 9)$ and the code mod 11 should equal the check digit.) For example, to find the check digit for the 10 digit ISBN whose first nine digits are 0-306-40615:

$$\begin{aligned} & 1 \times 0 + 2 \times 3 + 3 \times 0 + 4 \times 6 + 5 \times 4 + 6 \times 0 + 7 \times 6 + 8 \times 1 + 9 \times 5 \\ = \; & 145, \end{aligned}$$

and $145/11 = 13\text{R}2$ (13 with a remainder of 2). So the check digit is 2, and the complete sequence is ISBN 0-306-40615-2.

The two most common errors which occur when handling an ISBN (e.g., typing it in or writing it down) are an altered digit or transposition of adjacent digits. Since 11 is a prime number, the ISBN check digit method ensures that these two kinds of errors will always be detected.

Your task is to implement an ISBN check system. You'll be given an ISBN code from a book that a store wants to sell, and you need to make sure it's a real book so a mistake isn't made. Given a code, print `SALE` if it's valid and `OOPS` if it isn't. (Hint: the % operation will help you find remainders.)

Examples

Input: `"0-321-32213-4"`
Output: `SALE`

Input: `"1-55860-604-3"`
Output: `OOPS`

Input: `"0-534-95097-3"`
Output: `SALE`

7. **Histograms**

In this question, you must take a series of numbers between 0 and 99 and output a formatted histogram of scores. For example, on input `34 23 27 93 97 92 87 86 55 43 42 43 47 49 51 99`, the output should be the following:

```
        *
        *         *
        *         *
    *   * *     * *
    * * * *     * *
0 1 2 3 4 5 6 7 8 9
```

If there are two dots above 5, it means that two numbers in the list were between 50 and 59 (inclusive).

Examples

Input: `15 25 96 27`
Output:

```
    *
  * *             *
0 1 2 3 4 5 6 7 8 9
```

8. **Graph Coloring**

A *graph* (not in the sense of "the graph of $y = x^2$") is composed of a set of *nodes* and a set of *edges* connecting those nodes. Graphs are used in all of computer science, and graph algorithms have played a very influential role both in theory and practice.

Graph *coloring* is an assignment of colors to the nodes of the graph such that no adjacent nodes (two nodes are adjacent if they are connected by an edge) have the same color. Problems concerning graph coloring have played an important role in the history of computer science and mathematics. The general problem of finding the minimum number of colors needed to color an arbitrary graph – called the chromatic number – is famously difficult: an NP-Hard problem, in the language of complexity theory. In mathematics, the *four color conjecture*, now the four color theorem, was a well-known problem concerning graph coloring that remained unsolved for over a century; indeed, much of the last century's research in graph theory has its origin in the conjecture.

Though these problems have played an important role in theoretical research, they are also very relevant in applications. For example, when we schedule classes at a university, two courses taught by the same faculty member cannot be scheduled for the same time slot, and two courses required of the same group of students also should not conflict. Determining the minimum number of time slots needed is then a graph coloring problem. Other applications include assigning radio frequencies, computer optimization, and register allocation in computer hardware.

In this question, we ask you to write a program that can decide whether an arbitrary graph can be colored with just two colors. If so, print `YES`; otherwise, print `NO`. The input will come as two arguments: the first will be an integer representing the number of nodes in the graph, and the second will be a string of ordered pairs, representing the edges. For example, the input `3 "0 1, 1 2, 2 0"` represents a graph with 3 nodes that are connected in a triangular fashion: 0 to 1, 1 to 2, and 2 to 0 (we are naming the nodes with these integers; we could just have easily written something like `"A B, BC, C A"` but this is a bit more annoying to parse). The edges do not have any direction, so 1 0 and 0 1 are the same edge. You should assume that all the nodes are connected somehow, e.g. `"0 1, 2 3"` is not a valid edge set by itself because there is no way of getting from 0 to 2; in other words, there must be a path of edges between any two nodes. (This will simplify the problem a bit.)

Examples

Input: `3 "0 1, 1 2, 2 0"`
Output: `NO`

Input: `4 "0 1, 1 2, 2 3, 3 0"`
Output: `YES`