2007 Stanford Local Programming Contest

Saturday, October 6, 2007

Read these guidelines carefully!

Rules

1. You may use resource materials such as books, manuals, and program listings. You may *not* search for solutions to specifics problems on the Internet, though you are permitted to use online language references (e.g., the Java API documentation) or algorithms references equivalent to what would be found in a standard algorithms textbook (e.g., CLRS). You may *not* use any machine-readable versions of software, data, or existing electronic code. That is, all programs submitted *must be typed during the contest.* No cutting and pasting of code is allowed.
2. Students may not collaborate in any way with each other or anybody else, including people contacted via Internet.
3. You are expected to adhere to the honor code. You are still expected to conduct yourself according to the rules, even if you are not in Gates B02.

Guidelines for submitted programs

1. All programs must be written in C, C++, or Java. For judging, we will compile the programs in the following way:
   - `.c`: using `gcc -O2 -lm` (GCC version 4.0.3)
   - `.cc`: using `g++ -O2 -lm` (GCC version 4.0.3)
   - `.java`: using `javac` (Sun Java version 1.6.0)

   All programs will be compiled and tested on a Leland `myth` machine. The `myth` machines are dual Xeon 3.20 GHz machines with 1 GB RAM running Ubuntu Linux 6.06. Java programs will be invoked with the following command line:

   ```
   java -Xms512m -Xmx512m problemname
   ```

   which sets the amount of memory allocated for the program heap to 512 MB. Compilation errors or other errors due to incompatibility between your code and the `myth` machines will result in a submission being counted incorrect. The base filename for each problem will be listed in the problem statement; please use only the listed filename extensions.
2. Make sure you `return 0;` in your `main()`; **any non-zero return values will be interpreted by the automatic grader as a runtime error**.
3. **Java users:** Please place your `public static void main()` function in a public class with the same name as the base filename for the problem. For example, a Java solution for the `test` program should be submitted in the file `test.java` and should contain a `main()` in `public class test`.
4. All solutions must be submitted as a single file.

5. All programs should accept their input on **stdin** and produce their output on **stdout**. They should be batch programs in the sense that they do not require human input other than what is piped into stdin.

6. Be sure to follow the output format described in the problem exactly. We will be judging programs based on a `diff` of your output with the correct solution, so your program's output must match the judge output **exactly** for you to receive credit for a problem. As a note, each line of an output file must end in a newline character, and there should be no trailing whitespace at the ends of lines.

How will the contest work?

1. If you chose to work remotely from a home computer, we recommend that you test out the submission program to see if it works for you. To do this, **submit a solution for the test problem** shown on the next page. You may not see a result from the grading program immediately. However, you should receive a grading response by Friday midnight; if you do not, please let us know. We will also be grading test problems from 12:00 to 12:50 pm on Saturday.

2. For those who choose to participate onsite, from 12:00 to 12:50 pm, you will select a computer, set up your workspace and complete a test problem. Space in Gates B02 and the PUP cluster is limited, and will be available on a first-come first-served basis. You may also choose to work directly on one of the `myth` machines in Gates B08, although technically we do not have that room reserved for the contest.

3. At 1:00 pm, the problems will be posted on the main contest page in PDF and PS format, all registered participants will be sent an e-mail that the problems have been posted, and we will distribute paper copies of the problems to contestants who choose to compete in either Gates B02 or the PUP cluster.

4. For every run, your solution will be compiled, tested, and accepted or rejected for one of the following reasons: *compile error*, *run-time error*, *time limit exceeded*, *wrong answer*, or *presentation error*. In order to be accepted, your solution must match the judge output exactly (according to `diff`) on a set of hidden judge test cases, which will be revealed after the contest.

   - Source code for which the compiler returns errors (warnings are ok) will be judged as *compile error*.
   - A program which returns any non-zero error code will be judged as *run-time error*.
   - A program which exceeds the time allowed for any particular problem will be judged as *time-limit exceeded* (see below).
   - A program which fails a `diff -w -B` will be judged as *wrong answer*.
   - A program which passes a `diff -w -B` but fails a `diff` (i.e., output matches only when ignoring whitespace and blank lines) will be judged as *presentation error*.
   - A program which passes a `diff` and runs under the time constraints specified will be judged as *accepted*.

5. For each problem, the time allowed for a run (consisting of multiple test cases) will be 10 seconds total for all test cases. The number of test cases in a run may vary from 20 to 200 depending upon the problem, so write efficient code!

6. Watch your **e-mail on Leland**. We will be sending out any necessary messages as well as notification if your programs are accepted or rejected via e-mail, and the e-mails will go to `your_leland_id@stanford.edu`.

7. If you want to follow the progress of the contest, go to the **standings web page**. A link to this page exists on the main contest page, `http://ai.stanford.edu/~chuongdo/acm/2007/`.

8. At 5:00 pm, the contest will end. No more submissions will be accepted. Contestants will be ranked by the number of solved problems. Ties will be broken based on total time, which is the sum of the times for correct solutions; the time for a correct solution is equal to the number of minutes elapsed since 1:00 pm plus 20 penalty minutes per rejected solution. No penalty minutes are charged for a problem unless a correct solution is submitted. After a correct submission for a problem is received, all subsequent incorrect submissions for that problem do not count towards the total time.

9. The top six contestants will advance to the regionals. Full results will be posted as soon as possible after the competition.

Helpful hints

1. **Make sure your programs compile and run properly on the `myth` machines.** If you choose not to develop on the Leland systems, you are responsible for making sure that your code is portable.

2. **Read (or skim) through all of the problems at the beginning to find the ones that you can code quickly.** Finishing easy problems at the beginning of the contest is especially important as the time for each solved problem is measured from the beginning of the contest. Also, check the leaderboard frequently in order to see what problems other people have successfully solved in order to get an idea of which problems might be easy and which ones are likely hard.

3. If you are using C++ and unable to get your programs to compile/run properly, try adding the following line to your `.cshrc` file

   ```
   setenv LD_LIBRARY_PATH /usr/pubsw/lib
   ```

   and re-login.

4. The `myth` machines in Gates B08 are not officially reserved for the contest, but these will be the machines used for judging/testing of all programs. You may find it helpful to work on these machines in order to ensure compatibility of your code with the judging system.

5. If you are a CS major and have a working `xenon` account, please work in the **PUP cluster** rather than Gates B02; the PUP cluster has UNIX machines, which may be a more convenient programming environment if you intend to use Emacs, etc. If you don't know where the PUP cluster is, just ask!

6. If you are working on a PC in Gates B02, it may be helpful to run a VNC session if you don't like coding from a terminal. Check out the IT services page on using VNC, which can be found at `http://unixdocs.stanford.edu/moreX.html`. If you wish to use an IDE (e.g., Visual Studio or Eclipse), please make sure that you know how to set this up yourself beforehand. We will not be able to provide technical support related to setting up IDEs during the contest.

7. If you need a clarification on a problem or have any other questions, come talk to us in **Gates B02** or the **PUP cluster**, or send an e-mail to `chuongdo@cs`.

The directions given here are based on those taken from Brian Cooper's 2001 Stanford Local Programming Contest problem set. Many thanks to Sonny Chan, David Arthur, and Tomas Rokicki for their help in writing and debugging problems!

0 Test Problem (`test.{c,cc,java}`)

0.1 Description

This is a test problem to make sure you can compile and submit programs. Your program for this section will take as input a single number N and return the average of all integers from 1 to N, inclusive.

To submit a solution, you must log into an `myth` machine on the Leland network. Then, run

```
/afs/ir.stanford.edu/users/c/h/chuongdo/acm/submit
```

with a single argument: the name of the file you are submitting. For each problem, your solution must adhere to the naming convention specified next to the problem name! For example, to submit a C++ solution to this problem, type (make sure to run this from a `myth` machine):

```
$ ~chuongdo/acm/submit test.cc
```

When you submit your solution, the judge will receive an e-mail and will judge your solution as soon as possible (please be patient!). Once this is finished, you will receive an e-mail stating that you have completed the problem or that you have not, and a reason why not. You can resubmit rejected solutions as many times as you like (though incurring a 20 minute penalty for each rejected run of a problem you eventually get right). Once you have submitted a correct solution, future submissions of that problem will still be graded but will not count towards your final score or total time.

Note that you do not need to submit this problem during the actual contest!

0.2 Input

The input test file will contain multiple test cases. Each test case is specified on a single line containing an integer N, where $-100 \leq N \leq 100$. The end-of-file is marked by a test case with $N = -999$ and should not be processed. For example:

```
5
-5
-999
```

0.3 Output

The program should output a single line for each test case containing the average of the integers from 1 to N, inclusive. You should print numbers with exactly two decimal places. For example:

```
3.00
-2.00
```

0.4 Sample C Solution

```
#include <stdio.h>

int main() {
  int n;
  while (1) {
    scanf("%d", &n);
    if (n == -999) break;
    if (n > 0) printf("%.2lf\n", (double)(n * (n + 1) / 2) / n);
    else printf("%.2lf\n", (double)(1 + n * (1 - n) / 2) / (2 - n));
  }
  return 0;
}
```

0.5 Sample C++ Solution

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
  cout << setprecision(2) << setiosflags(ios::fixed | ios::showpoint);
  while (true) {
    int n;
    cin >> n;
    if (n == -999) break;
    if (n > 0) cout << double(n * (n + 1) / 2) / n << endl;
    else cout << double(1 + n * (1 - n) / 2) / (2 - n) << endl;
  }
  return 0;
}
```

0.6 Sample Java Solution

```
import java.util.*;
import java.text.DecimalFormat;

public class test {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        DecimalFormat fmt = new DecimalFormat("0.00");
        while (true) {
            int n = s.nextInt();
            if (n == -999) break;
            if (n > 0) System.out.println(fmt.format((double)(n * (n + 1) / 2) / n));
            else System.out.println(fmt.format((double)(1 + n * (1 - n) / 2) / (2 - n)));
        }
    }
}
```

1 Divisibility (`divisibility.{c,cc,java}`)

1.1 Description

On the planet Zoop, numbers are represented in base 62, using the digits

0, 1, . . . , 9, A, B, . . . , Z, a, b, . . . , z

where

A (base 62) = 10 (base 10)
B (base 62) = 11 (base 10)
⋮
z (base 62) = 61 (base 10).

Given the digit representation of a number x in base 62, your goal is to determine if x is divisible by 61.

1.2 Input

The input test file will contain multiple cases. Each test case will be given by a single string containing only the digits '0' through '9', the uppercase letters 'A' through 'Z', and the lowercase letters 'a' through 'z'. All strings will have a length of between 1 and 10000 characters, inclusive. The end-of-input is denoted by a single line containing the word "end", which should not be processed. For example:

```
1v3
2P6
IsThisDivisible
end
```

1.3 Output

For each test case, print "yes" if the number is divisible by 61, and "no" otherwise. For example:

```
yes
no
no
```

In the first example, 1v3 $= 1 \times 62^2 + 57 \times 62 + 3 = 7381$, which is divisible by 61.
In the second example, 2P6 $= 2 \times 62^2 + 25 \times 62 + 6 = 9244$, which is not divisible by 61.

2 Go (go.{c,cc,java})

2.1 Description

In the game of Go, two players alternate placing black and white stones on lattice points of an $n \times n$ grid, each attempting to surround as much territory (i.e., regions of unfilled lattice points) as possible. At the end of the game, the score for each player is the total area of the territory surrounded by his or her stones. Given the locations of black and white stones on a Go board at the end of a match, your task is to compute the score of each player in order to determine the winner.[1]

Formally, two grid lattice points with coordinates (r, c) and (r', c') are *adjacent* if $|r - r'| + |c - c'| = 1$. A connected region of unfilled lattice points belongs to one player's territory if all adjacent filled lattice points contain stones belonging to that player (see Figure 1). Finally, a player's score consists of the number of unfilled lattice points in his or her territory.

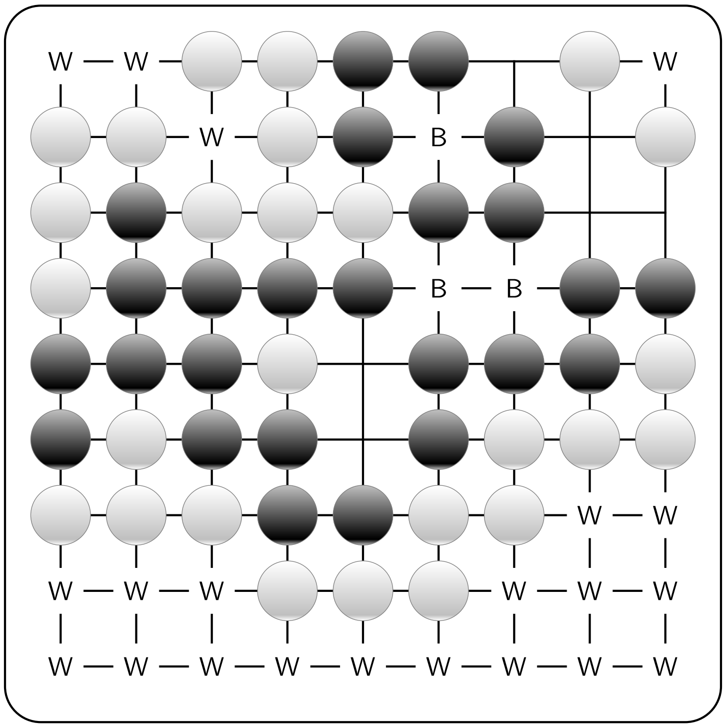

Figure 1: Diagram of a 9×9 Go board. Unfilled lattice points belonging to black's territory are marked with B, and unfilled lattice points belonging to white's territory are marked with W. Neutral unfilled lattice points are unmarked. In the game above, white wins by $21 - 3 = 18$.

[1]Note that the scoring of Go boards described here does not correspond exactly to the real game of Go: we make the simplifying assumptions that all "disputes" have been settled so that any territories surrounded by stones of both colors are considered neutral, and that all groups on the board are considered "alive."

2.2 Input

The input test file will contain multiple cases, each consisting of three lines. Each test case begins with a line containing three integers, n (where $1 \leq n \leq 19$), b, and w (where $b \geq 0$, $w \geq 0$ and $1 \leq b + w \leq n^2$). Here, n denotes the size of the board, b is the number of black pieces placed, and w is the number of white pieces placed. The second line of each test case contains b pairs of integers $r_1\ c_1\ \ldots\ r_b\ c_b$ (where $1 \leq r_i, c_i \leq n$) indicating the positions of the b black stones. The third line of each test case contains w pairs of integers $r'_1\ c'_1\ \ldots\ r'_w\ c'_w$ (where $1 \leq r'_i, c'_i \leq n$) indicating the positions of the w white stones. No two stones will be located at the same lattice point. Input is terminated by a single line containing only the number 0; do not process this line. For example:

```
1 1 0
1 1

2 0 1

1 1
5 12 4
1 1 1 2 1 3 2 1 2 3 3 1 3 3 4 1 4 3 5 1 5 2 5 3
1 4 2 4 3 4 3 5
0
```

2.3 Output

For each test case, print either "White wins by ____", "Black wins by ____", or "Draw". For example:

```
Draw
White wins by 3
Black wins by 1
```

3 Game Dice (`dice.{c,cc,java}`)

3.1 Description

In the game of Dungeons & Dragons, players often roll multi-sided dice to generate random numbers which determine the outcome of actions in the game. These dice come in various flavors and shapes, ranging from a 4-sided tetrahedron to a 20-sided isocahedron. The faces of an n-sided die, called dn for short, are numbered from 1 to n. Ideally, it is made in such a way that the probabilities that any of its n faces shows up are equal. The dice used in the game are d4, d6, d8, d10, d12, and d20.

When generating random numbers to fit certain ranges, it is sometimes necessary or desirable to roll several dice in conjunction and sum the values on their faces. However, we may notice that although different combinations of dice yield numbers in the same range, the probabilities of rolling each of the numbers within the range differ entirely. For example, a d6 and a d10 afford a range of 2 to 16 inclusive, as does two d8s, but the probability of rolling a 9 differs in each circumstance.

Your task in this problem is to determine the probability of rolling a certain number, given the set of dice used.

3.2 Input

The input test file will contain multiple cases, with each case on a single line of input. The line begins with an integer d (where $1 \leq d \leq 13$), the number of dice rolled. Following are d descriptors of dice, which can be any of "d4", "d6", "d8", "d10", "d12", or "d20". The last number on each line is an integer x (where $0 \leq x \leq 1000$), the number for which you are to determine the probability of rolling with the given set of dice. End-of-input is marked by a single line containing 0; do not process this line. For example:

```
1 d10 5
2 d6 d6 1
2 d10 d6 9
2 d8 d8 9
0
```

3.3 Output

For each test case, output on a single line the probability of rolling x with the dice, accurate to five decimal places. Note that even if there trailing zeros, you *must* show them (see Test problem for an example of decimal formatting). For example:

```
0.10000
0.00000
0.10000
0.12500
```

4 Baking Cakes (`cakes.{c,cc,java}`)

4.1 Description

Tom's birthday is coming up, and you have been put in charge of baking cakes for his monstrous birthday party. However, you have a great number of cakes to make, and a very short amount of time, so you are not sure that you will even finish before the party!

You have a list of different cakes to make, each requiring a certain amount of time to bake. You also have exactly 3 ovens to bake the cakes in, and each oven can only bake one cake at a time. Assuming that the time required to take a cake out and put another one in is negligible, can you determine the smallest amount of time you will need to spend baking, given the list of cakes to make?

4.2 Input

The input test file will contain multiple cases, with each case on a single line. The line begins with an integer n (where $1 \leq n \leq 40$), the number of cakes to bake. Following are n integers, $t_1, \ldots, t_n$ (where $1 \leq t_i \leq 30$), indicating the time in minutes required to bake each of your cakes. End-of-input is marked by a single line containing 0; do not process this line. For example:

```
1 30
3 15 10 20
5 6 7 8 9 10
0
```

4.3 Output

For each test case, output on a single line the smallest amount of time, in minutes, that you need to bake all of your cakes. For example:

```
30
20
15
```

5 Pool (`pool.{c,cc,java}`)

5.1 Description

Billiards, also commonly known as "pool," is a popular game in North America. The game is played on a rectangular table with six pockets—one at each corner and one in the middle of each of the two longer sides of the table. The object of the game is to strike a cue ball so that it collides with other balls, knocking them into the pockets.

The surface of our pool table measures 108" by 54", and to facilitate computation, we place it on the Cartesian plane with its southwest corner situated at $(0, 0)$, and its northeast corner at $(108, 54)$. Therefore, the centers of the 6 pockets, numbered 1 through 6, will have coordinates of $(0, 0)$, $(54, 0)$, $(108, 0)$, $(0, 54)$, $(54, 54)$, and $(108, 54)$, respectively (see Figure 2). The billiard balls are spherical and measure 2" in diameter.

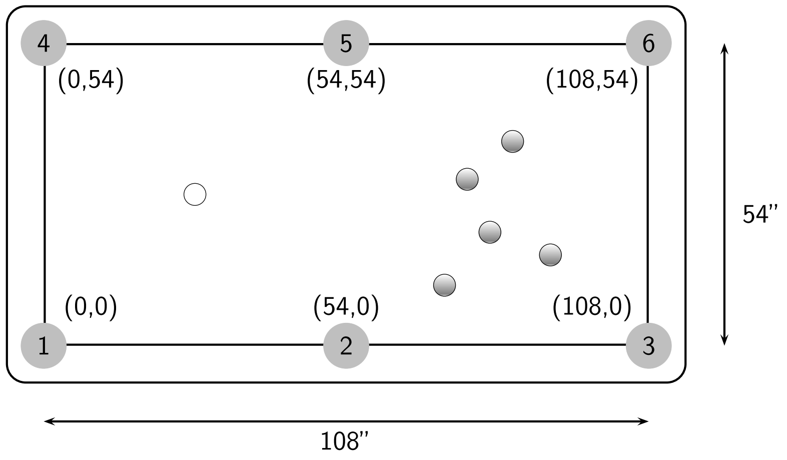

Figure 2: Diagram of pool table.

Given the location of the cue ball, a target ball, and a number of other balls on the table, your task for this problem is to write a program to determine whether or not you can successfully make a particular pool shot. The cue ball can be struck in any direction in a straight line. We consider collisions between the balls to be perfectly elastic, so that the target ball will always travel in a straight line, away from the point on its surface contacted by the cue ball (see Figure 3).[2]

A shot is considered possible if the cue ball can be struck so that it collides directly with the target ball, in turn sending the target ball directly into a pocket. Neither ball should collide with any other balls, bounce off the edges of the table (cushions), nor should their centers cross the boundaries of the table. In other words, you are not to consider any bank shots, combination shots, spin shots, or any other trick shots in this problem. Note that the difference between the incoming angle of the cue ball and the outgoing angle of the target ball must be greater than 90°. The target ball is considered to land in a pocket when its center coincides with the center of that pocket.

[2]You may assume that the cue ball disappears immediately after making contact with the target ball.

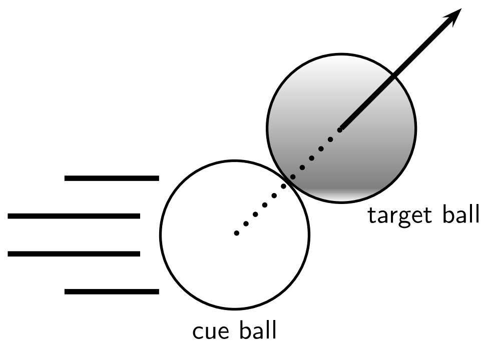

Figure 3: Diagram of pool ball collision.

5.2 Input

The input test file will contain multiple cases. The first line of each test case contains four real numbers, x_c y_c x_t y_t, where (x_c, y_c) is the location of the cue ball and (x_t, y_t) is the location of the target ball. The second line of the test case contains an integer n (where $0 \leq n \leq 14$), the number of additional balls on the table, followed by n pairs of real numbers, x_1 y_1 $\ldots$ x_n y_n, where (x_i, y_i) is the location of the ith additional (possibly obstructing) ball. No two balls overlap, and all balls are strictly in the interior of the table; in particular, all provided coordinates obey $3 < x < 105$ and $3 < y < 51$.

Input is terminated with a single line containing only the number 0; do not process this line. For example:

```
30 27 70 24
0
80 27 40 27
2 38 28 38 26
81.0 27.0 54.5 27.0
1 54.0 33.3
81.0 27.0 54.5 25.0
1 54.0 33.3
0
```

5.3 Output

For each test case, output on a single line the number(s) of the pocket(s) into which the target ball can be shot, sorted in ascending numerical order. Separate pocket numbers with a single space. If there are no pockets for which a clear shot exists, output the words "no shot". For example:

```
3 6
no shot
1 4
1 2 4
```

6 Jogger (`jogger.{c,cc,java}`)

6.1 Description

Every morning, Joe the Jogger goes for a brisk run around his neighborhood. The houses in the neighborhood (which are numbered from 1 to n) are connected together by a set of roads with the property that between every two houses, there exists exactly one unique path. That is, the graph structure of the road network is a tree of unknown topology, consisting of n leaf nodes (corresponding to each of the houses) and up to $n-1$ (but possibly fewer) internal nodes denoting intersections where roads split or merge (see Figure 4).

In this problem, your task is to help Joe plan a jogging route from one house in the neighborhood to another house. Because Joe is a veteran jogger, he wants his route to take as long as possible. Given a matrix of distances (in meters) between each pair of houses along the road graph, the number of seconds r it takes Joe to run one meter, and the number of seconds t it takes for Joe to cross each intersection along the route, determine the pair of houses for which the total travel time is as long as possible.

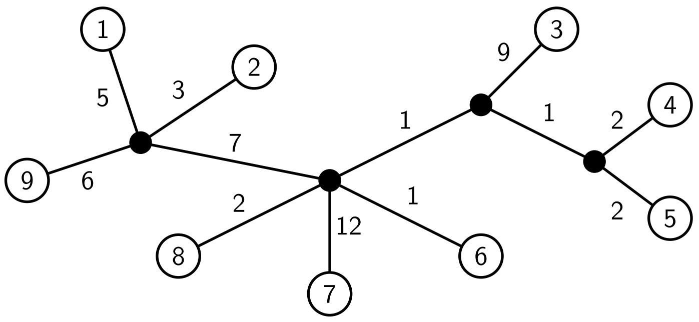

Figure 4: Diagram of neighborhood with $n = 9$. Houses correspond to numbered nodes, whereas internal nodes of the tree (i.e., intersections) correspond to filled nodes. Note that multiple roads may meet at a single intersection. Here, $d_{3,9} = 9 + 1 + 7 + 6 = 23$. If $r = 1$ and $t = 5$, then running from house 3 to house 9 takes Joe $1 \times 23 + 3 \times 5 = 38$ seconds. In the neighborhood shown above, this is the longest possible route, timewise, that Joe can take.

6.2 Input

The input test file will contain multiple cases. Each test case begins with a line containing three integers: n (where $1 \leq n \leq 50$), the total number of houses in the neighborhood, r (where $1 \leq r \leq 10$), the number of seconds per meter traveled, and t (where $1 \leq t \leq 100$), the number of seconds needed to cross each intersection. Then, the next n lines each contain n values indicating the distances d_{ij} (where $1 \leq d_{ij} \leq 1000$) between each pair of houses. The distances are specified in row-major order; i.e., the jth entry of the ith line is d_{ij}.

You are guaranteed that $d_{ii} = 0$ for each i and that $d_{ij} = d_{ji}$ for $i \neq j$. Furthermore, you may assume that the distance matrix corresponds to path lengths along some valid tree, no house coincides with any internal node of the tree, each internal node of the tree has degree ≥ 3, and all edges in the tree have positive length. Input is terminated by a single line containing the number 0; do not process this line. For example:

```
9 1 5
0 8 22 16 16 13 24 14 11
8 0 20 14 14 11 22 12 9
22 20 0 12 12 11 22 12 23
16 14 12 0 4 5 16 6 17
16 14 12 4 0 5 16 6 17
13 11 11 5 5 0 13 3 14
24 22 22 16 16 13 0 14 25
14 12 12 6 6 3 14 0 15
11 9 23 17 17 14 25 15 0
0
```

6.3 Output

For each test case, you must print a single line of output containing the longest total travel time possible. For example:

```
38
```

7 Censorship (`censorship.{c,cc,java}`)

7.1 Description

As part of the new educational reform program, the CS department has decided to engage in censorship of school texts. In this problem, you must help the department by writing a program which eliminates from an input text string all occurrences of strings from a set of words to be filtered.

More formally, a word w can be removed from another string s if w is a substring of s (i.e., the characters of w appear consecutively in s). Given a text string s and a set T of words to be filtered, return the length of the shortest possible string that can result from iteratively removing words in T from s. Each word in T may be removed from s an unlimited number of times.

7.2 Input

The input test file will contain multiple cases, with each case on a single line of input. Each test case begins with a single integer n (where $1 \leq n \leq 50$) indicating the size of the set T followed by a text string s to be processed. Then, n strings $t_1 \ \ldots \ t_n$ indicating the words of T follow. The text string and all of the filter words are guaranteed to contain only the characters 'a' through 'z' and will have lengths between 1 and 50. All filter words will be unique. Input is terminated by a single line containing the number 0; do not process this line. For example:

```
1 ccdedefcde cde
3 aabaab aa ba ab
3 aabaab aa ba bb
0
```

7.3 Output

For each test case, print a single integer indicating the minimum length resulting string possible. For example:

```
1
0
0
```

Possible reductions giving the lengths shown for the three sample inputs are:

$\text{c}\underline{\text{cde}}\text{defcde} \rightarrow \underline{\text{cde}}\text{fcde} \rightarrow \text{f}\underline{\text{cde}} \rightarrow \text{f}$
$\underline{\text{aa}}\text{baab} \rightarrow \underline{\text{ba}}\text{ab} \rightarrow \underline{\text{ab}} \rightarrow \epsilon$
$\underline{\text{aa}}\text{baab} \rightarrow \text{b}\underline{\text{aa}}\text{b} \rightarrow \underline{\text{bb}} \rightarrow \epsilon,$

where ϵ denotes the empty string.

8 Change (`change.{c,cc,java}`)

8.1 Description

Sue is waiting in line at the grocery store. Being in a hurry, she wants to pay with exact change when she gets to the front of the line. However, she does not know how much her items are going to cost; instead, she only knows an upper bound C on their total cost. Given a list of the various coins Sue has in her pocket, your goal is to determine the minimum number of coins she must take out in order to ensure that she can make exact change for every amount from 1 to C.

8.2 Input

The input test file will contain multiple cases. Each test case begins with a single line containing two integers C (where $1 \leq C \leq 1000000000$) and m (where $1 \leq m \leq 1000$), where C is the maximum amount for which Sue must be able to make change, and m is the number of unique coin denominations Sue has in her pocket. The next m lines each contain two numbers, v_i (where $1 \leq v_i \leq 1000$) and n_i (where $1 \leq n_i \leq 1000$), where v_i is the value of the ith coin denomination, and n_i is the number of coins of that denomination that Sue has in her pocket. Input is terminated by a single line containing the number 0; do not process this line. For example:

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

8.3 Output

For each test case, either print a single line containing the number of coins Sue must use in order to make exact change for all amounts up to C, or print "Not possible" if exact change cannot always be made with any combination of coins in Sue's pocket. For example:

```
3
Not possible
```