UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Programming Contest **P. N. Hilfinger**
Fall 1992

Programming Problems

You have 5 hours in which to solve as many of the following problems as possible. Put each complete solution into a single `.c` file. It must not `#include` any non-standard files. When you have a solution to problem number N (in the range 1–6) in file F`.c` that you wish to submit, mail it using the command

```
submit N F.c
```

You will be penalized for incorrect submissions, so be sure to compile and test your programs, using gcc. All tests will use the compilation command

```
gcc -O foo.c -lm
```

followed by one or more execution tests of the form

```
a.out < test-input-file | diff -b - test-output-file
```

There are examples of input and output in the directory D`/contest/examples`, where D is `~c164` on po, and `~hilfingr` on the Classic cluster. The test files used to test your submissions are not the same as these examples. Also, if you make multiple submissions of the same problem, the tests used may not be the same. You will be advised by mail whether your submissions pass.

All input will be placed in `stdin`, and all output must be produced on `stdout`. Anything written to `stderr` will be ignored. Your program *must* exit with a status code of 0 (use `exit(0)`). Because the testing of programs is automatic, you should be careful to adhere closely to the output format described. Make sure that the last line of output ends with a newline.

Scoring. Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.

Terminology. The term *free-form input* indicates that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character.

1. You are given a rectilinear array of bits, A, with dimensions $d_1 \times \cdots \times d_n$ and asked to find the smallest *tile* that describes this array. Here, a tile is an n-dimensional subpattern of the bits in A such that A can be described as consisting of non-overlapping copies of the tile pattern, all oriented the same way and laid side-by-side. For a sufficiently irregular array of bits, the smallest tile may be A itself.

For example, for $n = 1$, A might be

```
1 0 1 1 0 1 1 0 1
```

in which case the smallest tile is '`1 0 1`'. For $n = 2$, A could be

```
0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
```

in which case the smallest tile is

```
0 1
1 1
0 0
```

For $n = 3$, A could be

```
0 1 0 1       1 0 1 0        0 1 0 1       1 0 1 0
1 0 1 0       0 1 0 1        1 0 1 0       0 1 0 1
0 1 0 1       1 0 1 0        0 1 0 1       1 0 1 0
1 0 1 0       0 1 0 1        1 0 1 0       0 1 0 1
```

(all bits in the leftmost group have a second coordinate of 0, all those in the next group to right have a second coordinate of 1, etc. The third coordinate varies most rapidly going left to right.) For this, a tile would be

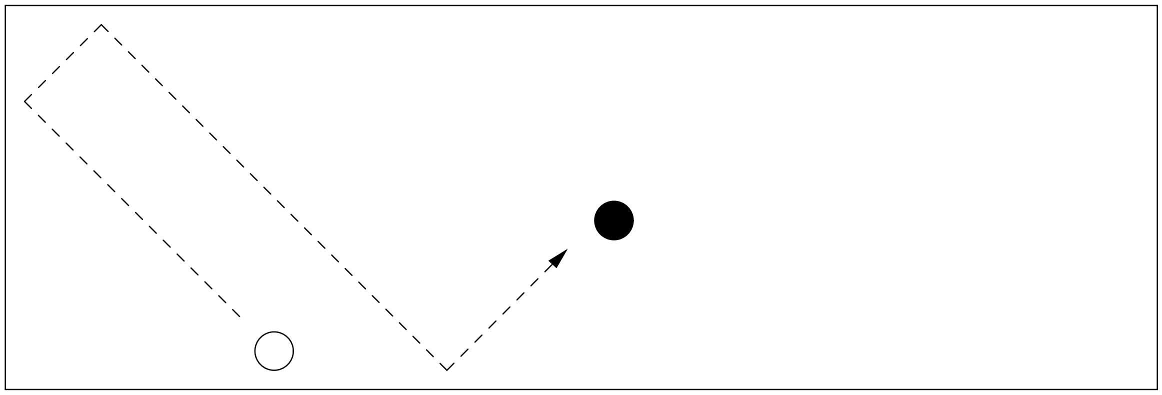
```
0 1    1 0
1 0    0 1
```

The input to your program will consist of the following (in free format):

- A value for n (a positive integer).
- The sequence $d_1, \cdots, d_n$, with all $d_i > 0$.
- A sequence of $d_1 \cdot d_2 \cdots d_n$ bits (each `1` or `0`), presented with rightmost subscript varying most rapidly.

Your output is to consist of a sequence of n positive integers (printed on a single line, separated by single spaces) giving the dimensions of the tile (this determines the contents of the tile, which therefore need not be printed). You may assume that $n \leq 7$ and that the total number of bits will not exceed 2^{16}.

2. A billiard ball is rolling on a frictionless, flat, rectangular table. There is another stationary ball somewhere on the table. You are to determine whether the balls collide for the first time after the first has rebounded from the cushions enclosing the table exactly three times. The situation might look like this, for example. The dashed arrow indicates the path of the center of the cue (white) ball, which banks (rebounds) three times and strikes the black ball.



The balls collide with each other when their centers are 2.0 units apart. The cue ball collides with a cushion if its center comes to within 1.0 unit of the cushion. As suggested by the diagram, the angle at which the cue ball collides with a cushion is equal to that at which it leaves the cushion.

The input to your program consists of the following sequence of numbers separated by whitespace in free form.

- Floating-point numbers x_c and y_c, the initial position of the center of the cue ball. The point (0,0) is what appears in the diagram as the lower-left corner.
- Floating-point numbers x_s and y_s, the position of the center of the stationary ball.
- Floating-point numbers d_x and d_y, the initial direction in which the cue ball moves. That is, the cue ball initially moves so that at time t, it is at

  $$(x_c + td_x, y_c + td_y).$$

  At least one of d_x and d_y must be non-zero.
- Floating-point numbers l_x and l_y, the dimensions of the table in the x and y directions, respectively.

Your program should print either the message `Balls collide`, `Balls do not collide`, or `Balls collide before three bounces`, as appropriate.

For the example above, the output would be `Balls collide` and the input might be as follows.

```
14.0 2.0    32.5 10.75
-2 2
60 20
```

3. The text-formatting program TeX uses the following algorithm for hyphenation (taken from *The TeXbook* by Donald E. Knuth).

1. A word to be hyphenated is extended on either end by special markers. For the word "hyphen-ation," we get `.hyphenation.` when we use '.' as the special marker.
2. The extended word has subwords

   ```
   . h y p h e n a t i o n .
   ```

   of length one,

   ```
   .h hy yp ph he en na at ti io on n.
   ```

   of length two,

   ```
   .hy hyp yph phe hen ena nat ati tio ion on.
   ```

   of length three, etc.
3. Each subword of length k has $k+1$ small integer "interletter" values having to do with the desirability of hyphenation in those places. These are all 0 unless that subword appears in the *pattern dictionary* described below. In the case of "hyphenation," for example, this pattern dictionary contains the subwords

   $$_0h_0y_3p_0h_0 \quad _0h_0e_2n_0 \quad _0h_0e_0n_0a_4$$
   $$_0h_0e_0n_5a_0t_0 \quad _1n_0a_0 \quad _0n_2a_0t_0 \quad _1t_0i_0o_0 \quad _2i_0o_0$$

   as patterns, and no others.
4. For each position in the word, find the maximum interletter value contained in any pattern touching that position. For example, between 'e' and 'n' in "hyphenation" there are five relevant values: 2 from $_0h_0e_2n_0$, 0 from $_0h_0e_0n_0a_4$, 0 from $_0h_0e_0n_5a_0t_0$, 1 from $_1n_0a_0$, and 0 from $_0n_2a_0t_0$. The maximum of these is 2. The result for all maximizations is

   $$._0h_0y_3p_0h_0e_2n_5a_4t_2i_0o_0n_0.$$
5. A hyphen is considered to be acceptable between two letters if the associated interletter value is *odd*. Thus, `hy-phen-ation`.

Write a program to read in words (given free-format on `stdin`) of up to 100 characters, and perform hyphenation as described, printing out each word on a separate line of `stdout` with possible hyphens inserted (e.g., `hy-phen-ation`). The pattern dictionary is in `~c164/hyphen.tex` on po and `~hilfingr/hyphen.tex` on the Classic cluster. Create a symbolic link to it in your directory with the name `hyphen.tex`, and refer to it in your program by that name (i.e., `fopen("hyphen.tex",...)`). Each line of the dictionary contains a subword, with single-digit integers at places with non-zero interletter values. For example,

```
.ach4
z4is
4z1z2
```

mean $._0a_0c_0h_4$, $_0z_4i_0s_0$, and $_4z_1z_2$.

4. A propositional logical formula is either

- An *atom*, denoted by a letter (upper and lower case are distinct), or
- A *composite formula*; recursively, if $\mathcal{A}$ and $\mathcal{B}$ are formulae, then so are `(`$\mathcal{A}$`|`$\mathcal{B}$`)`, meaning "$\mathcal{A}$ or $\mathcal{B}$," `(`$\mathcal{A}$`&` $\mathcal{B}$`)`, meaning "$\mathcal{A}$ and $\mathcal{B}$," `~`$\mathcal{A}$, meaning "not $\mathcal{A}$," and `(`$\mathcal{A}$`->`$\mathcal{B}$`)`, meaning "$\mathcal{A}$ implies $\mathcal{B}$," or equivalently "$\mathcal{B}$ or not $\mathcal{A}$."

This is a rigid syntax. Only and all the parentheses mentioned must be there, and no whitespace.

A formula is *satisfiable* if some assignment of truth values ('true' or 'false') to the atoms in it yields true. For example, the following formulae are each satisfiable.

```
q
(a|(b&c))
((a&~a)->z)
```

The following are not satisfiable.

```
(q&~q)
(((a|~b)&(~a|b))&(a&~b))
```

Write a program that reads any number of formulae from the standard input—each written with no whitespace inside it, and separated from each other by whitespace—echoes each formula on a separate line (no preceding whitespace), followed by either "`is satisfiable`" or "`is unsatisfiable`." For example, for the inputs above, the output should be

```
q is satisfiable
(a|(b&c)) is satisfiable
((a&~a)->z) is satisfiable
(q&~q) is unsatisfiable
(((a|~b)&(~a|b))&(a&~b)) is unsatisfiable
```

5. An m-bit floating-point number is a value

$$\pm 0.d_1 \cdots d_m \times 2^e,$$

where e is an integer (i.e., positive or negative), and each d_i is either 0 or 1. We call the fraction $0.d_1 \cdots d_m$ the *significand* and e the *exponent*. If the fraction meets the condition that either $d_1 = 1$ or all the $d_i = 0$, we say that the significand is *normalized*.

The m-bit *floating-point sum* of two such values is the m-bit floating-point value that is closest to the mathematical sum. In case the mathematical sum is equally close to two m-bit floating-point values, we choose the one having $d_m = 0$.

Write a program to repeatedly read in an integer, m and two normalized m-bit floating-point values, and print out the normalized m-bit sum. The format of the input will be as in the following example (for $m = 6$,); it is free-form, as usual.

```
6 +0.110010_-1 +0.110110_0
```

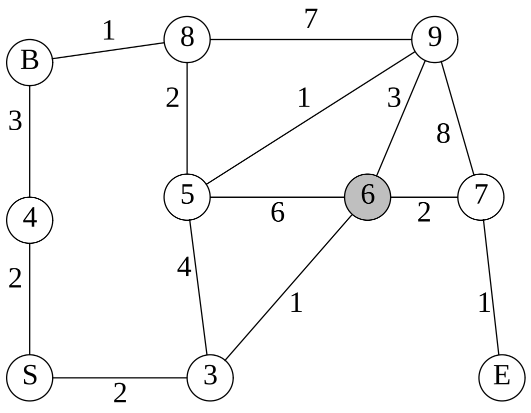
Denoting the two numbers 0.110010×2^{-1} and 0.110110×2^0. The result should be a single, normalized value in the same format; for this example the result is as follows.

```
+0.101000_1
```

Separate results are to be printed on separate lines. In this case, the actual result was 0.1001111×2^1. Rounding to 6 binary digits gives us the actual result. You may assume that $m < 256$. Prefix all 0 values with a '+' sign and give them an exponent of 0.

6. A borogove and a snark find themselves in a maze of twisty little passages that connect numerous rooms, one of which is the maze exit. The snark, being a boojum, finds borogoves especially tasty after a long day of causing people to softly and silently vanish away. Unfortunately for the snark (and contrariwise for his prospective snack), borogoves can run twice as fast as snarks and have an uncanny ability of finding the shortest route to the exit. Fortunately for the snark, his preternatural senses tell him precisely where the borogove is at any time, and he knows the maze like the back of his, er, talons. If he can arrive at the exit or in any of the rooms in the borogove's path before the borogove does (strictly before, not at the same time), he can catch it. The borogove is not particularly intelligent, and will always take the shortest path, even if the snark is waiting on it.

Thus, for example, in the following maze, the snark (starting at 'S') will dine in the shaded room, which he reaches in 6 time units, and the borogove (starting at 'B') in 7. The numbers on the connecting passages indicate distances (the numbers inside rooms are just labels). The snark travels at 0.5 units/hour, and the borogove at 1 unit/hour.



Write a program to read in a maze such as the above, and print one of two messages: `Snark eats`, or `Borogove escapes`, as appropriate. The input is as follows.

- A positive integer $N \geq 3$ indicating the number of rooms. You may assume that $N < 1024$. The rooms are assumed to be numbered from 0 to $N - 1$. Room 0 is always the exit. Initially room 1 contains the borogove and room 2 contains the snark.
- A sequence of edges, each consisting of two room numbers (the order of the room numbers is immaterial) followed by an integer distance.

Assume that whenever the borogove has a choice between passages to take (i.e., all lead to a shortest path), he chooses the one to the room with the lowest number.

For the maze above, a possible input is as follows.

```
10
2 3 2   2 4 2           3 5 4   3 6 1         4 1 3
5 6 6   5 8 2   5 9 1   6 7 2   6 9 3
7 0 1   7 9 8           1 8 1
8 9 7
```