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

Programming Contest **P. N. Hilfinger**
Fall 2000

2000 Programming Problems

To set up your account, execute

```
source ~ctest/bin/setup
```

in all shells that you are using. (This is for those of you using the C-shell. Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on 18 pages. You have 5 hours in which to solve as many of them as possible. Put each complete C solution into a file N`.c`, each complete C++ solution into a file N`.cc`, and each complete Java program into a file N`.java`, where N is the number of the problem. Each program must reside entirely in a single file. In Java, the class containing the main program for problem N must be named `P`N. Each C/C++ file should start with the line

```
#include "contest.h"
```

and must contain no other `#include` directives, except as indicated below. Upon completion, each program *must* terminate by calling `exit(0)` (or `System.exit(0)` in Java).

Aside from files in the standard system libraries and those we supply, you may not use any pre-existing computer-readable files to supply source or object code; you must type in everything yourself. Selected portions of the standard `g++` class library are included among of the standard libraries you may use: specifically, the headers `string`, `vector`, `iostream`, `iomanip`, and `fstream`. Likewise, you can use the standard C I/O libraries (in either C or C++), and the math library (header `math.h`). In Java, you may use the standard packages `java.lang`, `java.io`, and `java.util`. You may not use utilities such as `yacc`, `bison`, `lex`, or `flex` to produce programs. Your programs may not create other processes (as with the `system`, `popen`, `fork`, or `exec` series of calls). You may use any inanimate reference materials you desire, but no people. You can be disqualified for breaking these rules.

When you have a solution to problem number N that you wish to submit, use the command

```
submit N
```

from the directory containing *N*`.c`, *N*`.cc`, or *N*`.java`. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

```
submit -f N
```

which submits problem *N* without any checks.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs (if you get a message from `submit` saying that it failed, you will *not* be penalized). All tests will use the compilation command

```
contest-gcc N
```

followed by one or more execution tests of the form (Bourne shell):

```
./N < test-input-file > test-output-file 2> junk-file
```

which sends normal output to *test-output-file* and error output to *junk-file.* The output from running each input file is then compared with a standard output file, or tested by a program in cases where the output is not unique. In this comparison, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly.* It will do no good to argue about how trivially your program's output differs from what is expected; you'd be arguing with a program. Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; that is, the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 45 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc` *N*, where *N* is the number of a problem, is available to you for developing and testing your solutions. For C and C++ programs, it is equivalent to

```
gcc -Wall -o N -O -g -Iour-includes N.* -lstdc++ -lm
```

For Java programs, it is equivalent to

```
javac -g N N.java
```

followed by a command that creates an executable file called *N* that runs the command

```
java PN
```

when executed (so that it makes the execution of Java programs look the same as execution of C/C++ programs). The *our-includes* directory contains `contest.h` for C/C++, which also supplies the standard header files. The files in `~ctest/submission-tests/`*N*, where

N is a problem number, contain the input files and standard output files that `submit` uses for its simple tests.

All input will be placed in `stdin`. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you are free to use `scanf` to read in numbers and strings and `gets` to read in lines.

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. A *word* or *token,* accordingly, is a sequence of non-whitespace characters delimited on each side by either whitespace or the beginning or end of the input file.

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, and minus the time spent judging your entries. 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.

Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1 is your first submission of a problem, 2 the second, etc.), and the name of the file containing your explanation. Do not protest without first checking carefully; groundless protests will be result in a 5-minute penalty (see Scoring above).

Notices. During the contest, the Web page at URL

```
http://http.cs.berkeley.edu/~hilfingr/programming-contest/announce.html
```

will contain any urgent announcements, plus a running scoreboard showing who has solved what problems. Sometimes, it is useful to see what problems others are solving, to give you a clue as to what is easy. Be sure to reload this page from time to time, to keep information up to date.

1. A puzzle from the soon-to-be famous Garcia collection presents you with a five-by-five array of square push-buttons, some of which are lit, and asks you to find a set of moves that turn them all off. A move consists of pushing one of the buttons, which has the effect of *flipping* it and each of its north, south, east, and west neighbors (if any: one or two of these neighbors will be missing on the edges of the puzzle). Here, flipping a button means turning it off if it is on and vice-versa. You are to write a program that reads an initial configuration of lights and computes a set of moves to turn them all off. I say "set" rather than "sequence" because the flipping operations commute with each other and performing the same flip twice is an identity operation. Therefore, all we need to know is which set of lights to press, not the order in which to press them. There are thus 2^{25} possible sets of moves.

An input to your program will consist of five rows of five characters, each character a '1' or a '0', with the rows separated by whitespace. A '1' indicates a light that is on, and '0' a light that is off. Output consists of an echo of the input, and a solution that also consists of of five rows of five '1's and '0's, with '1' this time indicating a button that must be pushed, and '0' a button not pushed. Use the format shown in the examples below.

Example 1:

Input

```
01000 11100
01000
00000 00000
```

Output

```
Solution to:
   01000
   11100
   01000
   00000
   00000
is to push:
   00000
   01000
   00000
   00000
   00000
```

Example 2:

Input

```
00000
00000
10101
10101
01110
```

Output

```
Solution to:
   00000
   00000
   10101
   10101
   01110
is to push:
   00100
   01110
   10001
   00000
   00100
```

2. A certain compiler writer wants to eliminate redundant tests. For this purpose, he takes an exceedingly abstract view of a program language, in which all statements have one of the forms

`L`N	A statement label.
x `=`	An assignment of some unknown quantity to variable x.
`goto L`N	Unconditional branch to label
`if` x `<=` M `goto L`N	Branch to `L`N iff variable $x \leq M$.
`if` x `>=` M `goto L`N	Branch to `L`N iff variable $x \geq M$.

Here, N and M are ordinary decimal numerals, with $0 < N < 1000$, and x is a variable name: a single lower-case letter. You may assume that each instruction is on one, separate line, and that there is whitespace space separating each **if**, **goto**, variable name, and numeral M from the surrounding tokens.

After a sequence like

```
if x <= 2 goto L3
some statements
if x <= 4 goto L4
```

we know as long as `x` is not assigned to in *some statements* and there are no labels in *some statements,* the second test is redundant and can be removed. In general,

- If the only ways to reach a conditional statement from the beginning of the program all guarantee that its condition is true, we replace the statement with a plain **goto**.
- If the only ways to reach a conditional statement from the beginning of the program all guarantee that the condition is false, we can remove the statement entirely.
- A label with no branches to it may be removed.
- All statements between an unconditional **goto** and the next label are unreachable and can be removed.

You are to use these rules repeatedly to simplify a given program as much as possible, printing the results in the same format as the input, as shown in the examples on the next page. Remember that when you remove a statement, you may remove the last branch to a label, and that when you turn an **if** into a plain **goto**, you may make the following statements unreachable. Assume no program has more than 10000 statements.

Example 1:

Input	Output
`if x >= 11 goto L2`	`if x >= 11 goto L2`
`L1`	`z =`
`z =`	`L2`
`if x >= 20 goto L1`	
`L2`	

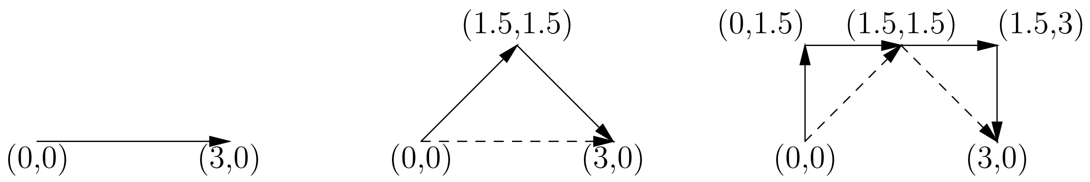
Example 2:

Input	Output
`if x >= 11 goto L2`	`if x >= 11 goto L2`
`L1`	`L1`
`x =`	`x =`
`if x >= 20 goto L1`	`if x >= 20 goto L1`
`L2`	`L2`

Example 3:

Input	Output
`x =`	`x =`
`if x >= 11 goto L100`	`if x >= 11 goto L100`
`L2`	`L2`
`y =`	`y =`
`if x >= 12 goto L102`	`z =`
`z =`	`if y >= 100 goto L101`
`if y >= 100 goto L101`	`goto L2`
`if x <= 20 goto L2`	`L100`
`L102`	`r =`
`q =`	`L101`
`goto L100`	
`L100`	
`r =`	
`L101`	

3. [D. Garcia] One can obtain a nice fractal pattern by starting with a set of directed line segments (a *directed line segment* has one end that is designated the "start" and one the "end"), then "breaking" each line segment, and repeating this operation some specified number of times. Breaking a directed line segment means treating it as the hypotenuse of an isosceles right triangle and replacing it with the two segments that make up the other two sides of this triangle:



This example shows two steps. The line segment at the left (the arrow shows direction) is first broken into two. The break is to the left relative to the direction of the arrow. Next each of the two resulting segments is broken to form the figure on the right. The dashed lines show the lines that were just broken in each case (the dashed lines are not part of the output).

The input to your program will consist of five numeric words in free form: x_0 y_0 x_1 y_1 B. (x_0, y_0) and (x_1, y_1) are the starting and ending points of the initial line segment. The x_i are floating-point numbers in general. B is an integer giving the number times to break all the line segments. Zero breaks produce just the single input line as output. One break produces something like the middle figure in the example above, and two breaks produces something like the right figure.

Your output will consist of lines giving the starting and ending coordinates of each resulting line segment, one segment per line, in the format

x_0 y_0 x_1 y_1 `v`

(that's a literal letter 'v' at the end). After all the lines comes the single line

```
done
```

Don't forget that last line. There are examples on the next page. (For your viewing pleasure, if you create a file, `foo`, containing a copy of the file `~ctest/lib/fractal.pro` followed by the output of your program, you can view the result using ghostscript:

```
gs foo
```

or print it.)

Example 1:

Input	Output
`0 0 3 0 0`	`0.00000e+00 0.00000e+00 3.00000e+00 0.00000e+00 v`
	`done`

Example 2:

Input	Output
`0 0 3 0 1`	`0.00000e+00 0.00000e+00 1.50000e+00 1.50000e+00 v`
	`1.50000e+00 1.50000e+00 3.00000e+00 0.00000e+00 v`
	`done`

Example 3:

Input	Output
`0 0 3 0 2`	`0.00000e+00 0.00000e+00 0.00000e+00 1.50000e+00 v`
	`0.00000e+00 1.50000e+00 1.50000e+00 1.50000e+00 v`
	`1.50000e+00 1.50000e+00 3.00000e+00 1.50000e+00 v`
	`3.00000e+00 1.50000e+00 3.00000e+00 0.00000e+00 v`
	`done`

4. There are numerous systems today that provide ways for laying text out in tabular format. Typically, one specifies a format for each column (whether text is centered, left-justified, or right-justified), and then gives a sequence of rows, with the text for each column in that row. The program then figures out how wide each column must be based on the largest width of the text for that column over all the rows, and prints the text for each column of each row according to the specified format. This is your task.

The input is in free form. It begins with a specification of the form

`\begin{tabular}{`$s_0 f_1 s_1 f_2 \cdots f_n s_n$`}`

Each s_i is either missing or is the single character `|`. Each f_i is either the single character `l`, `r`, or `c`. The specification is a single word with no embedded blanks. Next, there are zero or more text lines of the form

Text_1 `&` Text_2 `&` $\cdots$ `&` Text_n `\\`

Each Text_i consists of a sequence of words that don't include `&` or `\` characters. The ampersand and double-backslash symbols are themselves words (separated from any surrounding text words by whitespace). After all the rows, there is the closing sequence (again one word):

```
\end{tabular}
```

Your output consists of a table with n columns. Each row is formatted from one of the text lines according to the specification at the beginning as follows. Each s_i is copied to an output line literally. Each f_i is replaced by the corresponding Text_i, but with each sequence of whitespace squeezed down to a single blank. The size of f_i is equal to the length of the longest Text_i (after squeezing that is), plus 2 extra characters: a blank at the left and the right. If f_i is '`l`', the text is left justified (after the leading blank)—that is, blanks are added after the text as needed to fill up the field. If f_i is '`r`', the text is right justified, with blanks added to the left to fill the field. If f_i is '`c`', blanks are added to the left and right (with one more to the left, if there are not an even number to add) so as to center the field. Because of the blanks added at the beginning and end, the first and last characters of each field will be blank, and adjacent fields will be separated by two blanks, or a blank, vertical bar, and a blank.

The result is illustrated in the example on the next page.

Example 1:

Input

```
\begin{tabular}{l|c|r}
Jack & Professor & 70,000 \\
Herman & CEO & 350,000 \\
Susan & Personal
Trainer & 60,000 \\
\end{tabular}
```

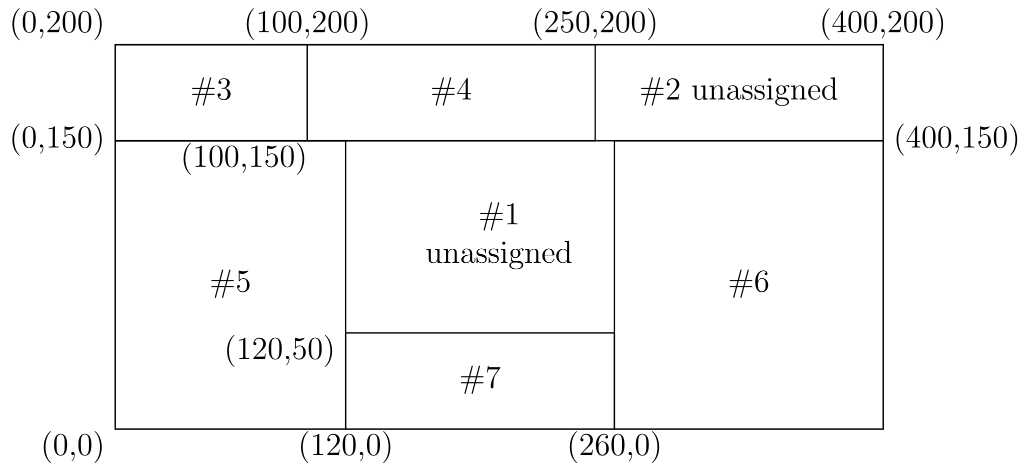
Output

```
Jack   |      Professor     |  70,000
Herman |         CEO        | 350,000
Susan  | Personal Trainer   |  60,000
```

5. One of the functions performed by a typical GUI-building package (GUI: Graphical User Interface) is to lay out a set of subwindows within some larger window (in Java, for example, this is the job of `LayoutManagers`). In this problem you'll build part of such a facility. The basic units we deal with are what we'll call *windows,* each of which is a rectangle whose sides are either vertical or horizontal (no slanting lines). A window may be a *subwindow* of a another, in which case it lies entirely inside the other (which is called its *parent*).

Here, we'll do a simplified layout manager. We start with an empty window containing no children. At each point in the process, there will be a rectangular, unassigned area of the window, initially all of it. We can add a child window to the top, bottom, left, or right of this unassigned area. When adding a window to the top or bottom, we must specify its height, but not width. When adding to the left or right, we specify width, but not height. The idea is that the child is positioned at the indicated side of the unassigned area, and that it expands in width (on the top and bottom) or height (on the left or right) to fill the whole side of the unassigned area. After doing this, we end up with a hierarchy of nested windows. We'll assume there is one outermost window containing all the rest. To complete the process, we specify both a height and width for this outermost window.

For example, consider 7 windows: #1 is the outermost, size 400×200 (width times height). First, we place window #2 with height 50 on the top of #1. Then we put window #3 with width 100 on the left of #2. Then we place window #4 with width 150 on the left of the remaining space of #2. Next, we put window #5 with width 120 on the left of the remainder of window #1. Then, window #6 on the right of #1 with width 140. Finally, we put window #7 at the bottom of #1 with height 50. Here is the result:



Window #2 here encloses #3 and #4, and window #1 includes all the rest. Windows #3–7 are all unassigned space, since they contain no children.

Input to your program is in free form. It begins with two integers, W, and H, the width and height of window #1, which is the outer window. Next there are entries for each of the remaining windows, each of the form

$N \;\; P \;\; s \;\; d$

where N is the number of the window, P the number of its parent, s the dimension (width or height, depending on placement) of the window, and d is one of the letters `T`, `B`, `L`, or `R`, for top, bottom, left, or right. Assume that window numbers are consecutive starting at 2 for the first non-top-level window, and that parents always precede children.

The output consists of a list of the number, coordinates of the lower-left corner, width, and height of each window. Use the format shown in the example below, which shows the input and results from the preceding example.

Input	Output
`400 200`	`1. 400x200 @ (0,0)`
`2 1 50 T`	`2. 400x50 @ (0,150)`
`3 2 100 L`	`3. 100x50 @ (0,150)`
`4 2 150 L`	`4. 150x50 @ (100,150)`
`5 1 120 L`	`5. 120x150 @ (0,0)`
`6 1 140 R`	`6. 140x150 @ (260,0)`
`7 1 50 B`	`7. 140x50 @ (120,0)`

6. The phrase *propositional calculus* is just a pretentious term for a logic involving simple logical formulae containing variables and the logical connectives '`&`' (and), '`|`' (or), '`~`' (not), and `>` (implies). For convenient input, we can write a formula such as '`(x&y) > (~x | y)`' in prefix form like this:

```
>&xy|~xy
```

that is, each (sub)expression is either a simple variable or is an operator followed by one or two operand subexpressions in the same format (recursively). For simplicity, assume that all logical variables are one-character lower-case letters. The presence or absence of whitespace has no effect on the meaning of the expression.

An *axiom schema* has the same format, but may also contain upper-case letters, which we'll call *pattern variables:*

```
> A
  > B A
```

An axiom schema *matches* a formula $\mathcal{F}$ if one can substitute for all its pattern variables so as to get $\mathcal{F}$. Thus, '`> A > B A`' matches '`> x > x x`' or '`> & x z > y & x z`' but it does not match '`> x & x x`' or '`> x > x y`'. A formula (i.e., not a schema) matches another formula if they are identical. We never match one schema to another.

A *proof with k givens* is a sequence of axiom schemata and formulas, $X_1 \cdots X_n$, such that for all $1 \le i \le n$, either

- $i \le k$ (i.e., "X_i is a given."), or
- There is some $j < i$ such that X_j matches X_i (X_i must be a formula, not a schema), or
- For some $j < i$ and $k < i$, X_j is the formula `(>` X_k X_i`)`. (This is the inference rule known as *modus ponens:* if A is true and A implies B, then infer B is true.)

Write a program that reads in a sequence of formulae and indicates whether it is a proof according to these rules. The givens are separated from the rest of the proof by a single underscore, '`_`', which will always appear. The output consists of an echo of the proof up to either its end, if it is valid, or the first erroneous line, if it is not, with a message about whether it checks. Use the format illustrated in the examples. [Hint: The conventional meanings of the connectives is irrelevant in this problem.]

Example 1:

Input

```
> A > B A
x
_
> x
  > y x
> y
  x
```

Output

```
> A > B A
x
_
> x > y x
> y x
*Proof OK*
```

Example 2:

Input

```
> A ~A
x
_
> x ~y
~y
```

Output

```
> A ~ A
x
_
> x ~ y
*Error in proof*
```

Example 3:

Input

```
> A   > B A

> > A B
  > > A  > B C
    > A C
-

> x  > x x

> > x > x x
  > > x > > x x x
    > x x

> > x > > x x x
  > x x

> x > > x x x

> x x
```

Output

```
> A > B A
> > A B > > A > B C > A C
_
> x > x x
> > x > x x > > x > > x x x > x x
> > x > > x x x > x x
> x > > x x x
> x x
*Proof OK*
```

7. In my recent interstellar travels, I had occasion to visit the ancient feudal people of Outer Xyyyy. Their society sets great store on social precedence. Through a complex set of customs, individuals rank themselves relative to each other, but the customs do not always guarantee that individuals will agree as to their respective ranks. At parties, it is the job of the host to determine his guests' ideas of rank and to specify arrival times on their invitations that guarantee to the extent possible that whenever individual A believes himself of higher rank than B, he arrives at the party *later* than B. It is awkward when both A and B believe themselves to be of higher rank, or when A and B don't know each other, but A believes himself above C who believes himself about B, and B believes himself above D who believes himself above A. In all such cases, custom reluctantly allows for all the individuals involved to arrive simultaneously. In the absence of this sort of disagreement, however, the higher-ranking individual arrives strictly later. When one individual does not know his rank relative to another, he does not care when he arrives relative to the other (sooner, later, or simultaneously).

Since parties are typically large, and hosts could use some help. Your problem is to devise a system to schedule the arrival of guests. Groups of guests should arrive at intervals of 1 decicronon (a unit of time out there). The idea is for each guest to arrive in some group as early as possible while obeying the constraints described above.

The input will consist of one or more sequences of names. A name is a word consisting only of letters, digits, and underscores. A word consisting of a single backslash (\) marks the end of each sequence. A sequence consisting of words N_0 N_1 $\cdots$ N_k, where $k \geq 0$, indicates that N_0 is a guest and that he considers his rank to be higher than that of N_1, $N_2, \ldots, N_k$, who are also guests. (It says nothing about the opinions of $N_1, \ldots, N_k$.) Every guest will be N_0 in exactly one sequence.

The output will consist of groups of guests numbered by arrival time, with the first group arriving at time 0 decicronons. Use the format shown in the example on the next page. Within a single group, print the names of guests in the order they appear as instances of N_0 (that is, if Xerccx appears at the beginning of one input sequence, and Alphrrt at the beginning of a later input sequence, and they are both to arrive at the same time, print Xerccx first.)

Example 1:

Input

```
Glorrk Groggg Alphrrt Kzyxx
Qqqrt \
Alphrrt Kzyxx Groggg \
Qqqrt Alphrrt \
Kzyxx Zwort \
Zwort Alphrrt \
Marvin \
Groggg Tlll \ Tlll \
```

Output

```
0. Marvin Tlll
1. Groggg
2. Alphrrt Kzyxx Zwort
3. Qqqrt
4. Glorrk
```

Example 2:

Input

```
Zwort \ Alphrrt \ Kzyxx \
```

Output

```
0. Zwort Alphrrt Kzyxx
```

8. Parties arriving at the famous Le Dindon Vert restaurant in East Biscuit, New Jersey are seated as soon as a table becomes available. Unfortunately, due to staffing problems, the restaurant's service tends to slow down considerably as more tables are filled.

The rate at which a party progresses through its meal at any time is $1/(1+0.5(N-1))$ of the rate at which a single party, alone in the restaurant, progresses, where N is the number of parties in the restaurant. As people leave, in other words, the rate of everybody else's progress increases. So, if a lone party can eat dinner in 1 hour once seated, then two can eat it in 1.5 hours. If I arrive at an empty restaurant and after 1/2 hour, another party arrives, I will take a total of 1.25 hours to eat (0.5 hour at full speed, and 0.75 hours at 2/3 speed).

Furthermore, parties differ as to tolerance for waiting to be seated, and in the time they have to eat. They will give up and leave the waiting line after some period of time after their arrival that we can call their *waiting threshold,* and they will leave the restaurant abruptly, whether or not they are finished, at some point after their arrival that we may as well call their *sitting threshold.*

Given a set of arrival times and thresholds for a sequence of parties arriving at Le Dindon Vert, you are to compute the total number of parties that reach one of their thresholds and leave, and the average waiting time and average total times spent over all parties. These averages are to include parties who give up waiting and are never seated (whose waiting time and total time are both their waiting threshold) and parties who leave before finishing (whose total time is just their sitting threshold).

Input is in free form. First, there is an integer, N, giving the number of tables. Next is an integer, E, giving the eating time of a party alone in the restaurant (in minutes). The rest of the input is a sequence of triples of integers. Each triple, $T\ \ W\ \ S$, gives the arrival time of a party (in minutes since opening time), their waiting threshold (in minutes since arrival), and their sitting threshold (in minutes since arrival). There may be any number of inputs, but you may assume that the values of T strictly increase.

Output should be in the format illustrated in the example on the next page. Internal calculations should be conducted precisely (so people can leave at fractional times), but all time averages that you report are to be rounded to the nearest minute.

Example:

Input	Comments
`3 60`	
`0 15 120`	Party 1. Eats 30 minutes at full speed + 1 minute with 2 parties at 2/3 speed + 58 2/3 minutes with 3 parties at 1/2 speed. Leaves at 89 2/3 past opening time.
`30 15 120`	Party 2. Eats 1 minute with 2 parties at 2/3 speed, and 118 2/3 minutes with 3 parties at 1/2 speed. Leaves at 149 2/3.
`31 15 160`	Party 3. Eats 120 minutes with 3 parties at 1/2 speed. Leaves at 151.
`45 15 120`	Party 4. Gives up after waiting 15 minutes.
`46 60 120`	Party 5. Waits 43 2/3 minutes. Eats for 60 minutes with 3 parties at 1/2 speed, then 1 1/3 minutes with 2 parties at 2/3 speed, then 15 minutes alone. Has to leave at 166 because of sitting threshold.

Output

```
There were 5 parties.
Average waiting time: 12 minutes
Average total time: 93 minutes
Never seated: 1
Left before finishing: 1
```