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

Programming Contest **P. N. Hilfinger**
Fall 2003

2003 Programming Problems, Revised*

If you are participating in the contest, please send me e-mail as soon as possible from the account you use to submit solutions. Use

```
Contest registration
```

(written just like that) as the subject line. Include in the body of the message one line that starts with your name. If your usual e-mail address differs from the login you use for the contest, also include that e-mail address.

To set up your account, execute

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

in all shells that you are using. (This is for those of you using csh-like shells. Those using `bash` should instead type

```
. ~ctest/bin/setup.bash
```

Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on 20 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 (yes, it is OK to have a Java source file whose base name consists of a number, even though it doesn't match the name of the class). Each C/C++ file should start with the line

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

*Problem revisions 3 May 2004.

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`, `strstream`, `fstream`, and `algorithms`. 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`, `java.text`, and `java.util`, and the Berkeley packages that you may have used in CS61B. 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 roughly equivalent to

```
gcc -Wall -o N -O2 -g -Iour-includes N.*  -lm
```

For Java programs, it is equivalent to

```
javac -g 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;

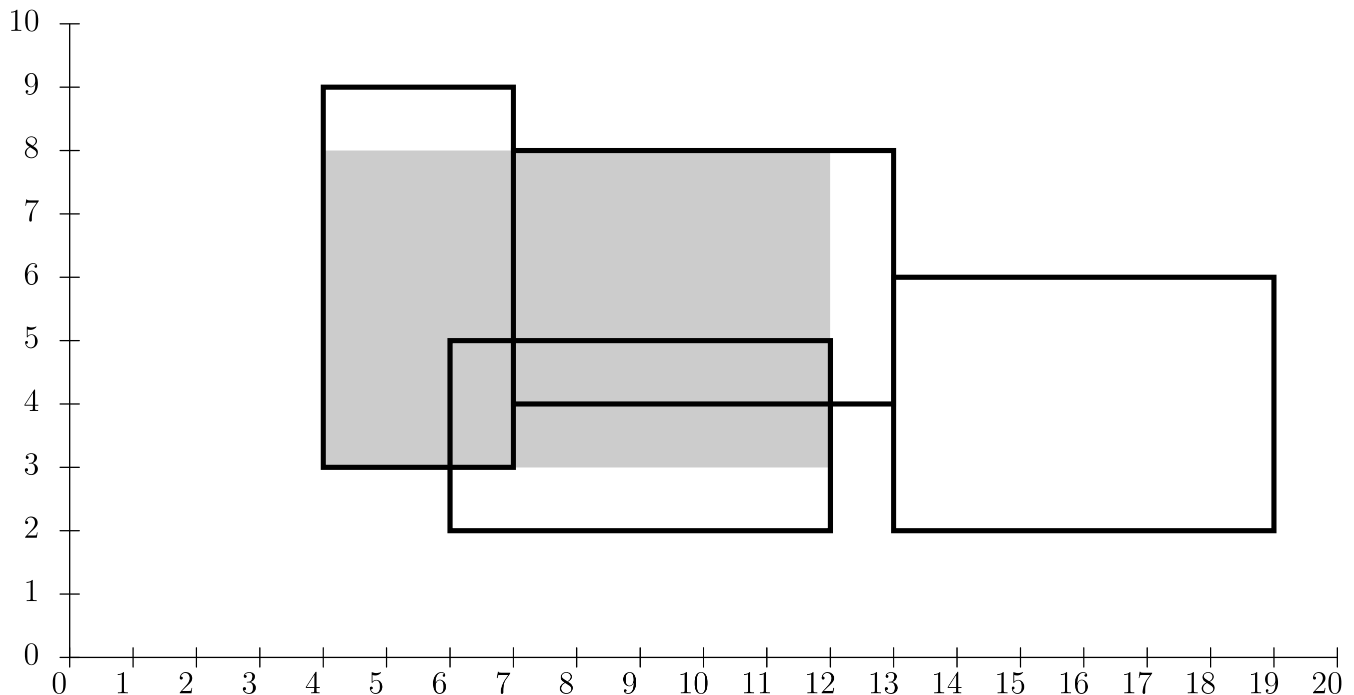
groundless protests will be result in a 5-minute penalty (see Scoring above). The Judge will *not* answer technical questions about C, C++, Java, the compilers, the editor, the debugger, the shell, or the operating system.

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.

1. [With thanks to an anonymous Microsoft interviewer] Find the largest rectangle (in area) that will fit within a union of rectangular regions. The sides of all rectangles are either horizontal or vertical. In the diagram below, for example, the gray rectangular region (area 40) is the largest that will fit in the area covered by the four outlined rectangles. The rectangles may overlap



The input to your program will consist of one or more sets of rectangles. Each set begins with an integer, N, giving the number of rectangles in the set, followed by N groups of four non-negative real numbers in free format. The numbers in each group are x, y, w, h, where (x, y) are the coordinates of the lower-left corner of a rectangle whose width is w and whose height is h. The last set of rectangles is followed by a single 0. You may assume there will be a maximum of 500 rectangles. The data will be chosen so that the sides of what are supposed to be adjacent rectangles agree exactly in their x (for vertical sides) or y (horizontal) coordinates—that is, you will not have to worry about round-off error. The range of the data can be accommodated with double-precision arithmetic.

For each set, the output is the area of a rectangular region of maximum size that lies within the union of rectangles, rounded down to the next integer. Use the format shown in the example on the next page.

Example.

Input	Output
`4`	`Set 1. Maximum region: 40`
`4 3 3 6   7 4 6 4    6 2 6 3`	`Set 2. Maximum region: 42`
`13 2 6 4`	
`4`	
`4 3 3 6   7 3.15 6 4`	
`6 2 6 3   13 2 6 4`	
`0`	

2. [D. Garcia] Your job is to buy a certain number of widgets, W, for a certain amount of money, $\$B$. Widgets come in N different varieties, and the cost of one widget of variety i is C_i, $1 \leq i \leq N$. For example, if there are $N = 4$ varieties of widget whose costs are $C_1 = 3$, $C_2 = 4$, $C_3 = 7$, and $C_4 = 18$, and you have $B = 20$ dollars with which to buy exactly $W = 5$ widgets, then you could buy one \$7 widget, one \$4 widget, and three \$3 widgets.

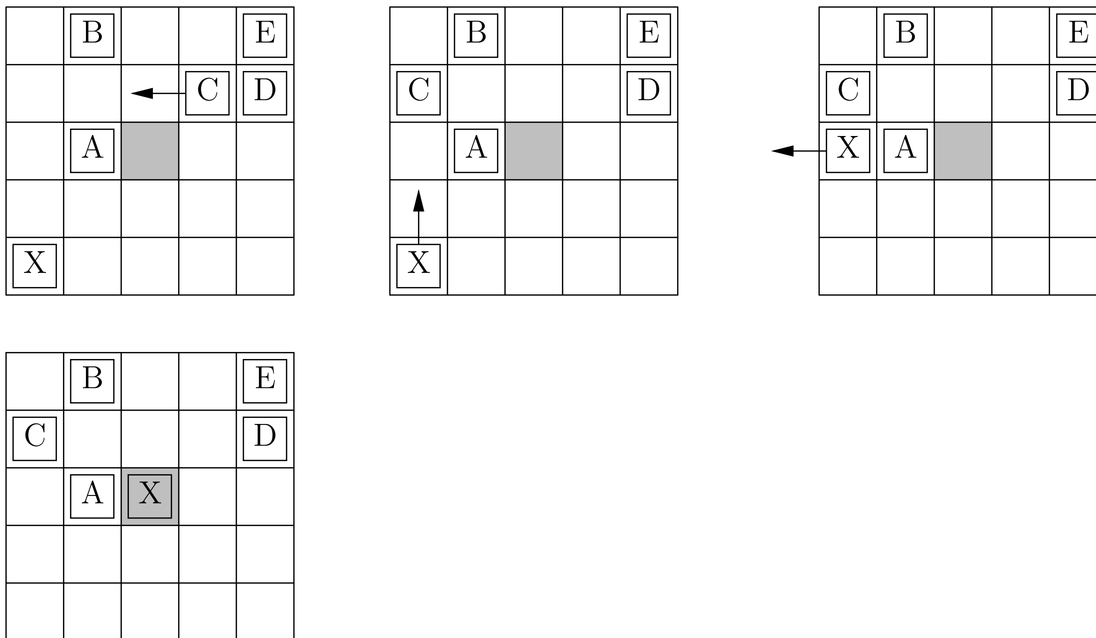
The input to your program is in free format and consists of one or more sets. Each set starts with three positive integers that supply values for W, B, and N, respectively. These are followed by N positive integers $C_1, \ldots, C_N$. A set that starts with three 0's ends the input.

For each set of data in the input, produce either a list of numbers of widgets to buy in each of the N categories (in order by increasing category number), including only those types of widgets of which you buy at least one. If no solution is possible, print the word "Impossible". Put a blank line between sets of output. Use the format given in the example below.

Example.

Input	Output
`5 20 4  3 4 18 7`	`Set 1:`
`5 20 4`	`   3 of type 1`
`  3 9 6 15`	`   1 of type 2`
`0 0 0`	`   1 of type 4`
	`Set 2:`
	`   Impossible`

3. [D. Garcia] Toroidal Lunar Lockout is a game played on a 5×5 board that consists of attempting to move a certain piece (marked 'X') in the diagram below to a central home square (shaded in the diagrams). Each move in this game consists of choosing one of the pieces (either X, or a helper 'bot (robot) labeled with a letter A–J or X) and moving it in one of four directions—up, down, right, or left—until it runs into another piece. The board is toroidal, so that pieces that cross over the left edge of the board end up in the rightmost square of the same row, and likewise for the right, top, and bottom. It is illegal to move a piece in a direction where it never encounters another piece (and would therefore travel forever). For example, starting in the configuration on the left below, the three moves shown solve the puzzle.



Given an initial configuration of this game, you are to find a solution of minimum length. Where more than one solution is possible, find the one in which at each step, the piece moved is furthest to the left on the lowest row.

The input will consist of one or more sets of numbers in free format. Each set starts with an integer, $0 < N \leq 10$, giving the number of 'bots. The names of the 'bots are the first N letters of the alphabet. There is always one X piece. Next come $N + 1$ pairs of integers giving first the row number (0–4, numbering from the bottom) and then the column number (0–4, numbering from the left) of each piece. The 'X' piece is the first set of numbers, then 'A', 'B', and so forth.

The last set is followed by a 0. Output for each set should consist of the set number followed either by a list of moves, or the phrase "No solution". The list of moves has the form of a list of groups P–d, where P is the name a piece, and the d indicate the direction of its moves: U for up, D for down, L for left, R for right (group adjacent moves for a single piece together). Use the format shown in the example below, separating output sets with a single blank line.

Example.

Input	Output
`5  0 0 2 1 4 1 3 3 3 4 4 4`	`Set 1:`
`3  0 0 1 1 2 2 3 3`	`   C-L X-U X-L`
`0`	
	`Set 2:`
	`   No solution`

4. The technique of *shotgun sequencing* is being used these days to determine the DNA sequence of a growing list of organisms. Here, we'll consider a vastly simplified procedure; for many reasons, it could only work only for the DNA of some unspecified extra-terrestrial organisms (whose DNA looks to us like "junk DNA"), and would require perfect lab technique.

Abstractly, we can specify a single DNA molecule as a string from an alphabet containing the four characters A(denine), G(uanine), and T(hymine), and C(ytosine)—the first letters of the names of four nitrogen bases. The string of bases is actually just one of two complementary sequences. For this problem, we are going to ignore this complication and pretend that we can look selectively at just one of the sequences.

We have no way of determining the entire string directly for DNA of any realistic size, so instead we create many duplicate molecules of the DNA being studied and then break them up into fragments at random locations, and obtain instead a large number of *substrings* of the string. Because of how they are created, many of these substrings will overlap. Assume there is a fragment that covers every base in either string. The problem is: given this set of fragments, reconstruct the sequence. There is not generally a unique solution, so we will find one that yields the shortest possible string (for earthly life, this would not be a good heuristic).

For example, here are a set of fragments aligned above a strand they might come from:

```
                 GTTATTGCCAACAGCA
TACTGTTGCAT               AACAGCAGTGGA
       GCATGTTG                       GGGGGTATCGCAGGCCT
         ATGTTGGAGTTATT           TGGAGGGGGTA    AGGCCTGATCGCAC
        CATGTTGGAGTT          AGCAGTGG
---------------------------------------------------------------
TACTGTTGCATGTTGGAGTTATTGCCAACAGCAGTGGAGGGGGTATCGCAGGCCTGATCGCAC
```

The input to your program will consist of one or more sets of data in free format. Each set starts with an integer, N, followed by N fragments. Each fragment is a string of ≤ 200 characters from the (upper-case) AGTC alphabet. A set starting with the integer 0 terminates the input. Use the format shown in the example on the next page for the output, printing in ten groups of three per line (or possibly fewer at the end of the string), and leaving a blank line between sets. Assume that there is a unique solution for each input.

Example.

Input	Output

10
TGGAGGGGGTA
GTTATTGCCAACAGCA
AGCAGTGG
TACTGTTGCAT AGGCCTGATCGCAC
GGGGGTATCGCAGGCCT
GCATGTTG AACAGCAGTGGA
ATGTTGGAGTTATT
CATGTTGGAGTT
 	
0
	Set 1:
	  TAC TGT TGC ATG TTG GAG TTA TTG CCA ACA
	  GCA GTG GAG GGG GTA TCG CAG GCC TGA TCG
	  CAC

5. The Recidivists, one of the two major political parties in the country of LaTexas, would like to increase still further their control over the government with a little creative redistricting (what we in this country call "gerrymandering"). Their country is shaped like a rectangle, which was long ago surveyed and divided into $W \times H$ grid squares—W squares west-to-east, and H squares north-to-south. There are supposed to be D non-overlapping districts of roughly equal population, each comprising an integral number of grid squares. The total population of LaTexas is T, so each district should have about T/D people in it. There are data from a recent poll and census that give the numbers of Recidivists and Demagogues (the opposing party) in each grid square. People in LaTexas tend to have strong opinions; nobody is "undecided" as to party.

The Recidivists want an assignment of grid squares to districts that will maximize the number of districts in which they have a majority. They must comply with certain constraints, lest they push the Demagogues too far, and cause them to decamp for Tahiti in order to deprive the LaTexan legislature of a quorum:

1. Any grid square in a district must be reachable from any other by a path that lies entirely within that district and proceeds only vertically or horizontally (never diagonally) from one square to the next.
2. The population of a district D may be lower than T/D if there is a single grid square adjacent to one of its grid squares (horizontally or vertically) that would bring its population above T/D if it were added to the district.
3. There may be at most one district whose population is above T/D.

For example, given the data below, and $D = 3$, we can give the Recidivists a 2/3 majority of districts by dividing the districts as shown. (The thick lines are district boundaries. The notation d/r means "100d Demagogues and 100r Recidivists."

5/1	2/3	4/4	6/7	3/4
6/2	5/1	2/1	6/4	5/6
3/4	2/5	9/3	7/2	5/1
1/6	4/3	4/3	2/2	5/5

The input to your program will consist of one or more sets of data in free format. Each set begins with three integers: W, H, and D. These are followed by $2WH$ values indicating the populations of Demagogues and Recidivists in each district from west to east (left to right), and north to south. The last set is followed by three 0's.

The output should be a single line for each set in the format shown in the example. A single blank line separates sets. You may assume that it is possible to solve each set.

Example.

Input	Output
`5 4 3`	`Set 1: 2 out of 3 districts can be Recidivist.`
`5 1 2 3 4 4 6 7 3 4`	
`6 2 5 1 2 1 6 4 5 6`	`Set 2: 3 out of 3 districts can be Recidivist.`
`3 4 2 5 9 3 7 2 5 1`	
`1 6 4 3 4 3 2 2 5 5`	
`5 4 3`	
`1 5 3 2 4 4 7 6 4 3`	
`6 2 1 5 1 2 4 6 6 5`	
`4 3 5 2 3 9 2 7 1 5`	
`6 1 3 4 3 4 2 2 5 5`	
`0 0 0`	

6. A number of systems now allow one to maintain multiple versions of a single file by, in effect, saving only the *changes* from one version to the next. Consider a scheme for storing N versions of a file, numbered from 1 (earliest) to N (latest), in two files: a *data file,* containing the actual characters ("the bits"), and an *administrative file,* which tells how to reconstruct any desired version of the file's contents from the data file or from previous versions.

For example, consider a one-line file that has gone through the following versions (the two lines at the end show character indices into the files' contents):

```
Version 1: the what she did.
Version 2: the who of the what she did.
Version 3: the who of the what of the what she did.
Version 4: The who of the what of the what she did.
           +---------+---------+---------+---------+
           0         10        20        30        40
```

This could be represented with the following data file:

```
the what she did.who ofT
+---------+---------+---------+
0         10        20        30
```

and the following administrative file:

```
4
    0 0 17  -1
    1 0 4  0 17 6  0 20 1  1 0 17  -1
    2 0 20  3 8 12 2 20 8 -1
    0 23 1  3 1 39  -1
```

The administrative file starts with an integer, N, giving the number of versions, followed by a sequence of N *version constructors.* A version constructor consists a sequence of *substring selectors,* terminated by a `-1`. Each substring selector has the form $V \quad K \quad L$, where V is a version number or 0, K is a character index within the version (0-based, as shown), and L is a length. A substring selector selects the L-character substring of version V that begins at the Kth character. Version 0 refers to the raw contents of the data file. The concatenation of the substrings specified by the substring selectors in the order they appear is the text of a version. As the example shows, a substring selector that is part of a version constructor for version V_1 may specify any V such that $0 \leq V < V_1$. It may also specify $V = V_1$, but only if K and L refer to a portion of the text that has already been constructed by previous substring selectors for version V_1 (for example, see the substring selector "`3 8 12`" in the version constructor for version 3, above).

The input to your program will consist of a data file terminated by a line that begins with the '@' character. All end-of-line characters preceding the '@' are part of the data file. As per the UNIX convention, ends of lines are marked with single newline characters.

You may assume that the size of the data file is $\leq 2^{16}$. Following the form feed is the administrative file described above in free format.

The output should consist of the reconstructed versions, printed as shown in the example below.

Example.

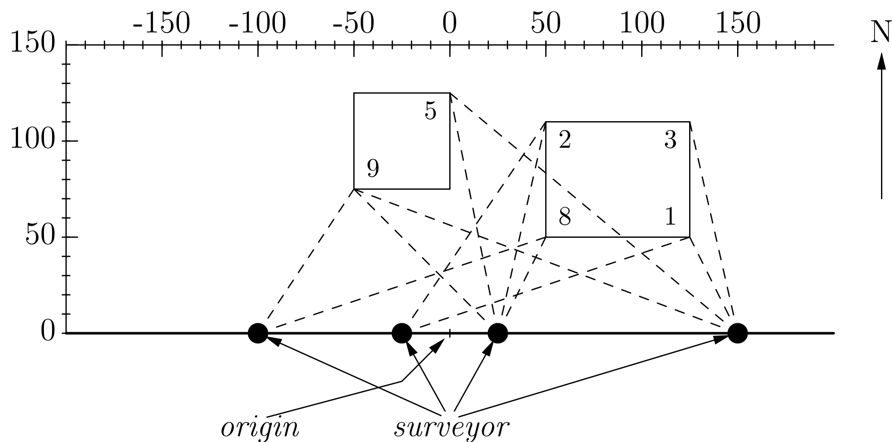
<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>
<pre>
the what she did.who ofT
@
4
0 0 17 -1
1 0 4 0 17 6 0 20 1 1 0 17 -1
2 0 20 3 8 12 2 20 8 -1
0 23 1 3 1 39 -1
</pre>
</td><td>
<pre>
Version 1:
the what she did.

Version 2:
the who of the what she did.

Version 3:
the who of the what of the what she did.

Version 4:
The who of the what of the what she did.
</pre>
</td></tr>
</table>

7. A lazy surveyor is attempting to determine the dimensions of a set of buildings in a city. He stands on a straight east-west running path the lies south of the city limits. His instruments allow him to determine the angle of elevation from the ground and azimuth (the angle left or right of due north) of any point in the city he can see. Unfortunately, between the fog and the pollution, he can only see the flashing warning lights that are mounted at the four top corners of each building, and can't always see them all. He is able to move east and west along the path and get new sightings, but not always of the same flashing lights. Fortunately, the lights all flash in subtly different patterns, so he knows when he has multiple sightings of the same light. The tops of each building form a perfectly level rectangle (their corners are all at the same height), with edges aligned east-west or north-south. No two buildings have the same height.



Given a set of readings of azimuth and elevation, you are to reconstruct the positions and dimensions of as many buildings as possible. Report only those buildings for which you have enough data to give all information.

The input will come in free-format sets. Each set begins with an integer, $N > 0$, giving the number of readings that follow. Each reading consists of an integer, K, followed by three floating-point numbers: p, a, and e. The value K identifies each light; it is arbitrary, but two readings are for the same light if and only if they have the same K (the small numbers in the diagram above indicate K values). The value p is the position of the surveyor on the path at the time of the reading, in feet east of an arbitrary origin on the path ($p < 0$ for positions west of the origin). Azimuth a is the angle clockwise from north (in degrees) of the light ($-90 < a < 90$). Elevation e is the angle above the horizon of the light in degrees ($0 < e < 90$). For simplity, let's pretend the surveyor takes his measurements while lying on the ground. The last input set is followed by the integer 0.

Your output should give the position of the southwest corner of each building whose position and dimensions can be totally reconstructed from the data, followed by the width east-to-west, the depth north-to- south, and the height. Positions are specified as (x, y), where x is the position in feet east of the origin on the path, and y is feet north ($y > 0$). Use the format shown in the example, separating sets of output with a blank line. Round all numbers to the nearest foot. List buildings found in order of increasing height.

Example.

Input

```
12
3 150 -12.8043 35.3438
1 150 -26.5651 55.0553
5 150 -50.1944 50.8693
9 150 -69.444 48.3309
5 25 -11.3099 62.0251
8 25 26.5651 55.0553
9 25 -45 66.1574
2 25 12.8043 35.3438
1 -25 71.5651 26.8378
2 -25 34.2869 31.0013
9 -100 33.6901 69.4149
8 -100 71.5651 26.8378

10
3 150 -12.8043 35.3438
1 150 -26.5651 55.0553
5 150 -50.1944 50.8693
5 25 -11.3099 62.0251
8 25 26.5651 55.0553
9 25 -45 66.1574
1 -25 71.5651 26.8378
2 -25 34.2869 31.0013
9 -100 33.6901 69.4149
8 -100 71.5651 26.8378

0
```

Output

```
Set 1:
  Building #1 is at (50,50) and has dimensions 75x60x80.
  Building #2 is at (-50,75) and has dimensions 50x50x240.

Set 2:
  Building #1 is at (-50,75) and has dimensions 50x50x240.
```

8. The game of Pherkadean Fizzbin is played by two players with a deck of the cards peculiar to the planet Pherkad Minor II. These cards have seven suits and eleven ranks. We humans generally just call the suits A–G and the ranks 1–11, in order to avoid dealing with the disturbingly bizarre names and images the Pherkadean culture has chosen for the suits (the name of the highest, for example, translates roughly as "the smell of napalm in the morning").

To play one round, the dealer gives both players some number, $1 < N \le 38$, of cards (which varies depending on the level of difficulty involved). The dealer then arranges his cards in a left-to-right row face up in front of him. His opponent must then place one of his cards next to each of the dealer's cards, after which the round is scored. Scoring works by adding individual scores for each of the opponent's cards, giving the total to the dealer (the deal alternates between players). The score for any given card is the maximum absolute value of the difference between that card's value (see below) and the values of the three nearest cards in the dealer's hand (i.e., the dealer card next to the opponent's card and the two dealer cards on either side of it, or, when the opponent's card is on an end, the three dealer cards at that end). The value of the opponent's card is its rank (1–11); that of a dealer card is its rank, plus 10 if its suit matches that of the opponent's card. Thus, an opponent's B9 next to the dealer's B2, F7, and G11 gives 3 points to the dealer (the maximum of $|9 - (10+2)|$, $|9 - 7|$, and $|9 - 11|$).

You are to write a program that, given the row of cards that the dealer has laid out, and the opponent's hand, determines the best way to for the opponent to match up his cards with the dealer's (i.e., giving the dealer the lowest possible score). For example, if $N = 5$ the dealer lays out

```
A1    B5    G11   G6    B9
```

and the opponent has `B11`, `G2`, `F10`, `B1`, and `G3`, then a best layout for the opponent best layouts is

```
A1    B5    G11   G6    B9
G2    B1    B11   F10   G3
```

for a score of 60.

The input to your program will consist of an integer, N, giving the number of cards in each hand, followed by the dealer's layout in the format shown above, followed by the opponent's cards. Input is in free format. The output should consist only of the best possible score, in the format shown in the example below.

Example.

Input	Output

`5`
`A1   B5   G11  G6 B9`
`B11 G2 F10 B1 G3`
	`Best score is 60.`