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

Programming Contest **P. N. Hilfinger**
Fall 2005

2005 Programming Problems

Please make sure your electronic registration is up to date, and that it contains the correct account you are going to be using to submit solutions (we connect names with accounts using the registration data).

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 16 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). Do not make class `P`N public, or the Java compiler will complain. 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`, `sstream`, `fstream`, `map`, 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`, `java.math`,

and `java.util` and their subpackages, as well as the package `contest`, which we supply. 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.

In the actual ACM contests, you will not be given nearly as much information about errors in your submissions as you receive here. Indeed, it may occur to you to simply take the results you get back from our automated judge and rewrite your program to print them out verbatim when your program receivees the corresponding input. Be warned that I will feel free to fail any submission in which I find this sort of hanky-panky going on (retroactively, if need be).

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 -I`*our-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 P`N

when executed (so that it makes the execution of Java programs look the same as execution of C/C++ programs). To use extra Java classes that we supply in package `contest`, you must have properly set up your account first (see above). The *our-includes* directory (typically `~ctest/include`) 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://inst.cs.berkeley.edu/~ctest/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. [Miguel Revilla (from the Universidad de Valladolid problem set)] Legend has it that, after being defeated at Waterloo, Napoleon Bonaparte, recalling his days of glory, said to himself "Able was I ere I saw Elba." Although it is quite doubtful that he should have said this in English, this phrase is widely known as a typical palindrome. A palindrome is a symmetric character sequence that looks the same when read backwards, right to left. In Napoleon's case, white spaces appear at the same positions when read backwards. This is not a required condition for a palindrome. The following (by Mark Twain), ignoring spaces and punctuation marks, might have been the first conversation and the first palindromes by human beings.

> "Madam, I'm Adam."
> "Eve."

Write a program that finds all palindromes in the input lines. The input ends at the end of the file. Each line contains fewer than 3000 letters (plus an unbounded amount of other stuff).

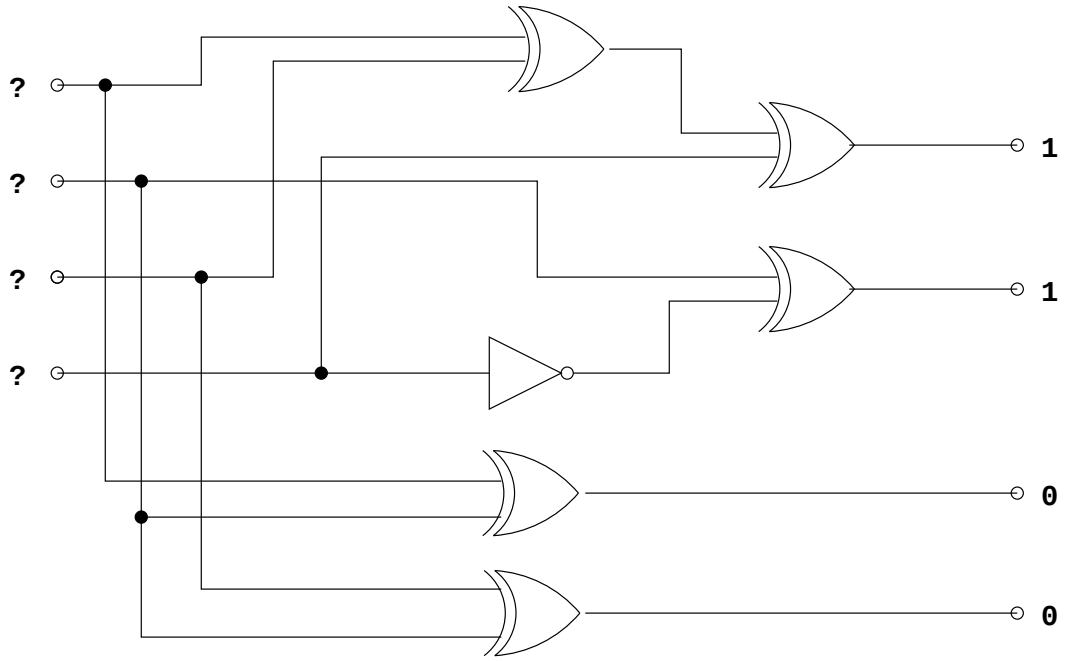
For each input line, output a line consisting of all the character sequences (letters only) of more than two characters that are palindromes in the input line. Ignore any characters in the input except letters—punctuation characters, digits, spaces, and tabs should all be ignored. Ignore case: treat uppercase and lowercase versions of a letter as identical. Whether the character sequences represent proper English words or sentences does not matter. Report palindromes in all uppercase characters, sorted lexicographically (in dictionary order) and separated by space characters when there are two or more for a given line. In any given line, report each palindrome only once, no matter how often it recurs in the line. When a palindrome overlaps with another, even when one is completely included in the other, report both, *unless* the shorter palindrome is exactly in the middle of the longer one, in which case, don't report the shorter one, even if it appears elsewhere in the line. For example, for an input line of "AAAAAA", report "AAAAAA" and "AAAAA", but not "AAAA" or "AAA". If an input line does not contain any suitable palindromes, output an empty line.

In the example below, the second line in the output is empty, corresponding to the second input line containing no palindromes. Some of the palindromes in the third input line, namely "ADA", "MIM", "AMIMA", "DAMIMAD", and "ADAMIMADA", are not reported because they appear at the center of reported ones. "MADAM" is reported, as it does not appear in the center, but only once, disregarding its second occurrence.

Example.

Input	Output
`As the first man said to the`	`TOT`
`first woman:`	
`"Madam, I'm Adam."`	`MADAM MADAMIMADAM`
`She responded:`	`DED ERE`
`"Eve."`	`EVE`

2. Consider a circuit with N inputs and N outputs connected with wires, xor gates (as for the C or Java `^` operator: `0^0` = `1^1` = 0, `1^0` = `0^1` = 1), and not gates. Given the values of the outputs, you are to determine the values of the inputs that produced those outputs, assuming that the answer is unique. For example, in the circuit



the indicated outputs result when all the inputs (from top to bottom) are 1.

The input will consist of a sequence of sets. Each set begins with an integer N, with $0 < N \leq 100$, indicating the number of inputs and outputs. There then follow N output specifications. Each specification starts with two integers, K and V, indicating the number of input specifications that follow and the desired output value (0 or 1). Each of the K input specifications, I_j is the number of an input ($1 \leq I_j \leq N$, for $1 \leq j \leq K$) or its negative (indicating an inverted value). For example, the output specification

```
2 1 2 -4
```

means "an output of 1 results from the xor of input #2 and the inverse of input #4." (Since xor is associative and commutative, this input format allows us to specify any circuit, although not necessarily compactly.) The last set of input will be followed by the integer 0.

For each input set, the ouput will consist of a sequence of N 0's and 1's indicating the input values that give rise to the indicated outputs. There will always be a unique solution. Use the format shown in the example.

Hint: I would advise against using an exhaustive search.

Example.

Input	Output
4 3 1 1 3 4 2 1 2 -4 2 0 1 2 2 0 2 3 4 3 1 1 3 4 2 1 2 -4 2 1 1 2 2 0 2 3 0	Circuit 1: 1 1 1 1 Circuit 2: 1 0 0 0

3. Sodoku is a kind of puzzle that is popular in Japan, and increasingly so elsewhere. Each puzzle consists of a 9×9 grid of squares, some of which are filled in with digits from 1–9. The grid is further subdivided into 9 3×3 subgrids. The idea is to fill in the rest of the squares so that each row, each column, and each 3×3 subgrid contains each of the digits 1–9. For example, the filled-in grid on the right below is a solution to the puzzle on the left.

				1				
3		1	4			8	6	
9			5			2		
7			1	6				
	2		8		5		1	
				9	7			4
		3			4			6
	4	8			6	9		7
				8				

4	5	2	6	1	8	3	7	9
3	7	1	4	2	9	8	6	5
9	8	6	5	7	3	2	4	1
7	3	4	1	6	2	5	9	8
6	2	9	8	4	5	7	1	3
8	1	5	3	9	7	6	2	4
2	9	3	7	5	4	1	8	6
1	4	8	2	3	6	9	5	7
5	6	7	9	8	1	4	3	2

The input to your program will consist of a single puzzle consisting of 81 one-digit integers containing the given grid entries from left-to-right, top-to-bottom. Blank entries are indicated with the digit 0. You may assume the puzzle always has a solution.

The output should consist of a 9×9 grid in the format shown in the example. The data will allow exactly one solution in each case.

Example.

Input	Output
0 0 0 0 1 0 0 0 0 3 0 1 4 0 0 8 6 0 9 0 0 5 0 0 2 0 0 7 0 0 1 6 0 0 0 0 0 2 0 8 0 5 0 1 0 0 0 0 0 9 7 0 0 4 0 0 3 0 0 4 0 0 6 0 4 8 0 0 6 9 0 7 0 0 0 0 8 0 0 0 0	4 5 2 6 1 8 3 7 9 3 7 1 4 2 9 8 6 5 9 8 6 5 7 3 2 4 1 7 3 4 1 6 2 5 9 8 6 2 9 8 4 5 7 1 3 8 1 5 3 9 7 6 2 4 2 9 3 7 5 4 1 8 6 1 4 8 2 3 6 9 5 7 5 6 7 9 8 1 4 3 2

4. [R. Fateman] Consider the following equation:

$$\frac{1}{(x-4)^3(x-3)^2} = \frac{3}{x-4} + \frac{-2}{(x-4)^2} + \frac{1}{(x-4)^3} + \frac{-3}{x-3} + \frac{-1}{(x-3)^2}$$

In fact, any left-hand side of this form may be expressed similarly:

$$\frac{1}{(x+c)^a(x+c+1)^b} = \frac{p_1}{x+c} + \frac{p_2}{(x+c)^2} + \ldots + \frac{p_a}{(x+c)^a} + \frac{q_1}{x+c+1} + \ldots + \frac{q_b}{(x+c+1)^b}$$

where c and all the p_i and q_i are integers. One way to do this is to multiply both sides through by $(x+c)^a(x+c+1)^b$, and then observe that, to make the two sides equal for all x, the constant term on the right must be 1 and all the coefficients of $x, x^2, \ldots x^{a+b-1}$, have to be 0. That gives you enough information to find all the p_i and q_i.

You are to write a program to compute the p_i and q_i for such problems. The input will consist of a sequence of sets in free format. Each set consists of three integers: c, a, and b, where $c \neq 0$, $c \neq -1$, $0 \leq a$, $0 \leq b$, $a + b > 0$. For example, the input for the problem above would be

```
-4 3 2
```

The last set of input will be followed by three integer literal 0s. You may assume that we will choose input data such that the intermediate calculations in the procedure suggested above are of "reasonable" size.

For each set, the output will consist of two lists, the first containing c followed by $p_1, \ldots, p_a$ and the second containing $c + 1$ followed by $q_1, \ldots, q_b$, using the format shown in the sample output below. Separate sets with a single blank line.

Example.

Input	Output
`-4 3 2`	`Expansion 1: (-4,3,-2,1) (-3,-3,-1)`
`3 3 0`	
`0 0 0`	`Expansion 2: (3,0,0,1) (4)`

5. [From the Universidad de Valladolid problem set] In Contract Bridge, players first *bid* against each other; bids consist of a suit (clubs, diamonds, hearts, or spades) and a number of tricks they expect to take beyond the first six. First, though, they must assess the strength of the 13 cards in their hands to determine a reasonable bid. You are to write a program that performs a simple version of this assessment (for the first bidder only) and decides only on an opening suit. First, we compute a *point value* for the hand:

1. Each ace counts 4 points. Each king counts 3 points. Each queen counts 2 points. Each jack counts one point.
2. Subtract a point for any king in a suit with only one card (that is, where the king is the only card of that suit in your hand).
3. Subtract a point for any queen in a suit with no more than two cards.
4. Subtract a point for any jack in a suit with no more than three cards.
5. Add a point for each suit of which the hand contains exactly two cards.
6. Add two points for each suit of which the hand contains exactly one card.
7. Add three points for each suit of which the hand contains no cards.

A suit is *stopped* if it contains an ace, or if it contains a king and at least one other card, or if it contains a queen and at least two other cards.

During the opening assessment, the three most common possibilities are:

- If the hand evaluates to fewer than 14 points, then the player must pass.
- One may open bidding in a suit if the hand evaluates to 14 or more points. Bidding is always opened in one of the suits with the most cards.
- One may open bidding in "no-trump" if the hand evaluates to 16 or more points *ignoring rules 5, 6, and 7,* and if all four suits are stopped. A no-trump bid is always preferred over a suit bid when both are possible.

For the first example below, the evaluation starts with 6 points for the two kings, 4 for the ace, 6 for the three queens, and one for the jack. To this tally of 17 points, we add 1 point for having only two cards in spades, and subtract 1 for having a queen in spades with only one other card in spades. The resulting 17 points is enough to justify opening in a suit.

The evaluation for no-trump is 16 points, since we cannot count the one point for having only two spades. We cannot open in no-trump, however, because the hearts suit is not stopped. Hence we must open bidding in a suit. The two longest suits are clubs and diamonds, with four cards each, so the possible choices are to bid clubs or bid diamonds. For this problem, we will prefer diamonds to clubs (and hearts to diamonds and spades to hearts).

Input to your program consists of a series of hands, each hand containing 13 cards. Each card consists of two characters. The first represents the rank of the card: 'A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K'. The second represents the suit of the card: 'S', 'H', 'D', 'C', standing for "spades", "hearts", "diamonds", and "clubs", respectively. Cards are separated by whitespace, but the rank and suit are immediately adjacent, without intervening whitespace.

For each hand, print one line containing the recommended bid, either "PASS", "BID *suit*", where *suit* is 'S', 'H', 'D', or 'C' (in this order of preference if two or more are possible), or "BID NO-TRUMP". Use the format shown in the example.

Example.

Input	Output

`KS QS TH 8H 4H AC QC TC 5C KD QD JD 8D`
`AC 3C 4C AS 7S 4S AD TD 7D 5D AH 7H 5H`
	`Hand 1: BID D`
	
	`Hand 2: BID NO-TRUMP`

6. Define a *standard term* recursively to be either

- A variable (a string of lower-case letters), or
- A pair of terms separated by whitespace and surrounded by parentheses, or
- A *lambda term* of the form $(\lambda x.\ E)$, where x is a variable, and E is a term. Inside this lambda term, any occurrence of the variable x is said to be *bound* by the lambda term, unless it is part of another, nested lambda term naming the same variable.

For example,

```
x
foo
(λ q.  (λ bar.  (q (bar foo))))
((λ y.  y) (λ z.  z))
```

are all terms. Here, we will concern ourselves only with terms in which all variables are bound by some lambda term; such terms are called *combinators.* Only the last example above is a combinator.

Renaming all occurrences of a variable that are bound to a particular lambda term does not change the meaning of the term: $(\lambda x.\ x)$ and $(\lambda y.\ y)$ mean the same thing. Likewise $(\lambda x.\ ((\lambda x.\ x)\ x))$ and $(\lambda x.((\lambda y.\ y)\ x))$ mean the same thing. The fact that there are therefore an infinite number of ways to write any term is sometimes undesirable, and there are various alternative ways to define terms that avoid it. One such is the *de Bruijn term,* ("Bruijn" is pronounced roughly "brown"), which for our purposes may be defined as

- A non-negative integer numeral, or
- A pair of de Bruijn terms separated by whitespace and surrounded in parentheses, or
- A lambda term of the form $\lambda.\ B$, where B is a de Bruijn term.

In a de Bruijn term, the numeral k is interpreted as being bound to the k^{th} most deeply nested lambda term that surrounds it (numbering from 0). So $(\lambda x.\ (\lambda y.\ (x\ (y\ y))))$ becomes $(\lambda.\ (\lambda.\ (1\ (0\ 0))))$, and both $(\lambda x.\ x)$ and $(\lambda y.\ y)$ become $(\lambda.\ 0)$.

Write a program that converts standard terms into de Bruijn terms. The input will consist of standard terms in free format, except that they will use a backslash ('\') instead of λ. Extra whitespace is allowed anywhere except in the middle of a variable. You may assume that each term is a proper combinator (all variables are bound).

The output will consist of de Bruijn terms in the format shown in the example below (with spaces only between the terms in a parenthesized pair).

Example.

Input	Output

`(\s.(\foo.foo))`
`(\m.(\n.(\s.(\z.((m s)`
`((n z) s))))))`
`(\f.((\x.(f (\y.((x x) y))))`
`(\x.(f (\y.((x x) y))))))`
	`Term 1: (\.(\.0))`
	`Term 2: (\.(\.(\.(\.((3 1) ((2 0) 1))))))`
	`Term 3: (\.((\.(1 (\.((1 1) 0)))) (\.(1 (\.((1 1) 0))))))`

7. Consider a set of pairs of values:

$$\{(a, x), (a, y), (b, y), (b, x), (b, z)\}$$

We could specify this in a fashion reminiscent of Unix makefiles:

```
a : x
a : y
b : y
b : x
b : z
```

Furthermore, we can introduce an abbreviation in which we allow more than one item to the left or right of the colons:

```
a b : x y
b : z
```

The first line above is supposed to indicate a cross-product: every item on the left ('a' and 'b') is paired with every item on the right ('x' and 'y'). Define the *size* of one of these specifications as the total number of items on all the left and right sides of the colons (that is, the total number of tokens in the description, not including colons). So the size of the first specification is 10, and the size of the second is 6. The problem is to find a specification of minimal size for a given set of pairs.

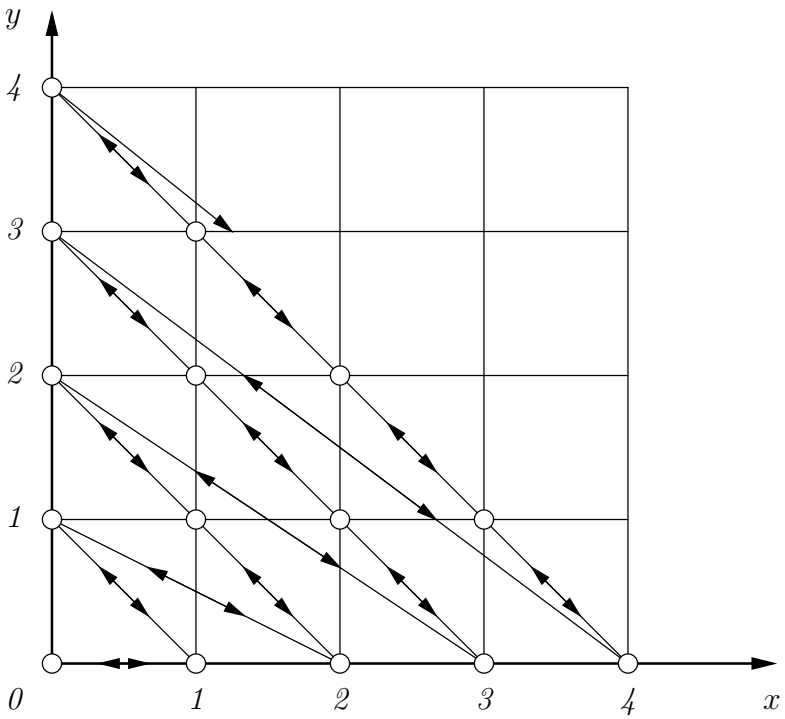
The input will consist of one of more sets of pairs. Each set of pairs will consist of an even number of words in free format—where a *word* is defined as a non-empty string of letters, digits, and underscores)—followed by two periods separated by whitespace.

For each set of pairs, output a minimal specification, using the format shown in the example. The order in which you output lines or identifiers within a line is irrelevant; any minimal solution will do.

Example.

Input	Output
a x a y b y b x b z . . a b c d . .	Set 1: a b : x y b : z Set 2: a : b c : d

8. [Anupam Bhattacharjee (from the Universidad de Valladolid problem set)] In the diagram below, each small circle has non-negative integral coordinates in the usual Cartesian coordinate system. You can move from one circle to another following a path denoted by the ¡arrow symbols. To go from a source circle to a destination circle, the total number of step(s) needed is one more than the number of intermediate circles you pass through. For example, to go from (0, 3) to (3, 0), you have to pass through (1, 2) and (2, 1), so in this case, the total number of steps needed is 3. In this problem, you are to calculate the (minimal) number of step(s) needed to go from a given source circle to a given destination circle.



The input is in free format, starting with the number of test cases, N. This is followed by N groups of four integers, the first pair of which are the coordinates of the source circle (x then y) and the second of which represents those of the destination circle. You may assume that x and y are non-negative integers less than 2^{31}, and that the result is also a non-negative integer less than 2^{31}.

For each pair of integers your program should output the case number, source and destination points, and the number of step(s) to reach the destination from the source in the format shown in the example.

Example.

Input	Output
3 0 0 1 0 0 0 0 1 1 2 1 1	Case 1: (0, 0) to (1, 0) takes 1 step Case 2: (0, 0) to (0, 1) takes 2 steps Case 3: (1, 2) to (1, 1) takes 4 steps