

1999
Thomas L. Mourey Memorial
ACM/IBM Mountain Region
Scholastic Programming Contest

Table of Contents

Problem 1: The Knight, the Princess, and the Dragons

Problem 2: Roll the Die!

Problem 3: April Fool's Joke

Problem 4: Reciprocals

Problem 5: Subdirectories

Problem 6: Mountain Landscape

Problem 7: Lumber Cutting

Problem 8: Millie's Restaurant

Problem 1: The Knight, the Princess, and the Dragons (3 pages)

A one-story building consists of a rectangular array of rooms. The building will be the scene of a drama whose players are a knight in shining armor, a beautiful princess, and varying numbers of dragons in some of the building's rooms. Such a building (and its current state) will be represented in the format shown in Figure 1.

Each symbol in Figure 1 represents a room. The walls of the building are not shown ; they run in north-south or east-west direction. Specifically, the symbols have the following meaning:

K	Initial location of the knight
.	<period>
0 .. 9	A room which will always be free of dragons
#	A room which may contain some dragons; the integer represents the current number of dragons
P	A permanently obstructed room which the knight may never enter
	Location of the princess, goal of the knight's journey

- ☞ The east/west rows and north/south columns of rooms are numbered starting with row 0 and column 0 in the building's northwest corner.
- ☞ Thus the room containing 'K' in the above configuration is designated by (1, 4), meaning row 1 and column 4.
- ☞ The maximum number of east-west rows of rooms, as well as the maximum number of north-south columns of room in the building's floor plan will not exceed 12.
- ☞ The rooms along the building's outside walls are #'s.
- ☞ In those rooms which may contain dragons, the number of dragons in the room changes according to the following rules:
 - ☞ At midnight of each day the weakest dragon is eaten by the other dragon(s), if any.
 - ☞ When there is only one dragon left, it will die of loneliness after 24 hours.
 - ☞ 24 hours after the last dragon died, however, the evil spirits will place nine new dragons into the room.
- ☞ The initial configuration of the building is understood to exist shortly before noon of day 1.
- ☞ At noon of each day (including day 1), the knight may
 - ☞ either move to an adjacent room (in a north-south or east-west direction)
 - ☞ or stay idle in the current room, if that room is not dragon-prone.
- ☞ The knight may enter a dragon-prone room (at noon of some day) only if it contains exactly one dragon. The knight is powerful enough to hold one dragon at bay until midnight (when the dragon is going to die of loneliness), but the knight would not have a chance against two or more dragons. Also, if a dragon-prone room is empty at noon, the knight should know not to enter, because as soon as the clock strikes midnight, the room will contain nine dragons.
- ☞ The knight will not enter the same room twice in a path.

```
#####
#...K#
#5##0#
#...P#
#####
```

Figure 1. Building floor plan

```
#####
#...K#
#5##0#
#...P#
#####
.<<<vv>>>
#####
#...K#
#5##0#
#...P#
#####
<<<.vv>>>
#####
#...K#
#5#90#
#...P#
#####
.<<<vv>>^^
#####
#...K#
#5##0#
#...P#
#####
.<<<vv>>^^
#####
#...K#
#4##0#
#...P#
#####
.<<<vv>>>
```

Figure 2. Sample input file

The input to your program will come from the file `prog1.dat`. An example of an input file is shown in Figure 2.

- ❖ The input file will contain data for one or more knight-princess-dragons problems. (The input file shown on the preceding page contains data for six problems.) Each such problem will be represented by
 - ❖ the floor plan/initial configuration as previously discussed
 - ❖ a proposed action sequence for the knight.
- ❖ Each action sequence will be represented in the input file by one line containing at most 80 characters.
- ❖ The following characters can appear in an action sequence:

• <period>	idle
^ <caret>	move north
▼ <lower case v>	move south
< <less than>	move west
> <greater than>	move east

- ❖ The input file will not contain any blank lines, nor any leading or trailing blank spaces.

- ❖ For each knight-princess-dragons problem in the input file, your program will

- ❖ trace the knight's actions until one of three possible events:
 - ❖ the knight reaches the princess; any further actions in the action sequence will be ignored;
 - ❖ the knight attempts an impossible move or makes a move which will cost his life; again, any further actions in the action sequence will be ignored;
 - ❖ the knight has performed all actions of the action sequence without reaching the princess;
- ❖ write (to the output file `prog1.out`)
 - ❖ one blank line
 - ❖ a one-line report summarizing the outcome of the given action sequence.

?? The contents and format of the one-line report are shown in the sample output below.

If the input file `prog1.dat` has the contents shown in Figure 2, then `prog1.out` will be as follows:

Problem 1 by team x

```
Day  9: entered room      ( 3,  4); princess found
Day  9: entered room      ( 3,  4); princess found
Day 10: attempt to enter room ( 1,  3); previously entered on day  2
Day  9: attempt to enter room ( 2,  3); permanently blocked
Day  5: entered room      ( 2,  1); will be killed by dragons
End of problem 1 by team x
```

Explanation of the output:

- ❖ Each report reflects the state of affairs shortly after 12 noon of the day shown.

- ❖ First report:

- ❖ On Day 1, the knight stays idle. After three successive left moves, he reaches room (1,1) on Day 4. On Day 5, room (2,1) contains only one dragon, so the knight enters that room safely (by a move down) and the remaining moves get him to the princess on Day 9.

- ❖ Second report:

- ❖ After three successive left moves, he reaches room (1,1) on Day 3. On Day 4 (when there are two dragons in room (2,1)) he remains idle, and proceeds to his goal as in the previous action sequence.

- ❖ The last action (a move to the right) in the action sequence was never executed; it will be ignored.

- ❖ Third report:

- ❖ The knight gets past two dragon-occupied rooms at the right time, before attempting to cycle back to a previously occupied room.

- ❖ Fifth report:

- ❖ Room (2,1) now contains 4 dragons on Day 1 and no dragon on Day 5, when the knight enters it. At midnight, the room is filled with nine dragons that will immediately kill the knight.

- ❖ Observe every detail of the output, such as the exact wording and punctuation of statements, upper/lower case variations, blank lines and blank spaces

~~EE~~A few lines of the above output will be reproduced here with formatting templates:

1	2	3	4	5	6	7
1234567890123456789012345678901234567890123456789012345678901234567						
Day 10:	attempt to enter room	(1, 3)				
Day 9:	attempt to enter room	(2, 3)				
Day 5:	entered room	(2, 1)				

~~EE~~The day number is right justified in a field of width 3 in columns 4-6. It is also right justified in a field of width 3 in columns 66 - 68 when reporting the day of a previous entry.

~~EE~~The row number of a particular room is right justified in a field of width 2 in columns 32 - 33.

~~EE~~The column number of a particular room is right justified in a field of width 3 in columns 35 - 37.

Problem 2: Roll the Die! (3 pages)

A gambler's die (Figure 1) is a cube whose six faces are labeled with 1, 2, 3, 4, 5, and 6 dots, respectively.

- ❖ Each of the faces 1, 2, and 3 is adjacent to the other two.
- ❖ If the die is viewed so that faces 1, 2, and 3 are visible, face 1 is on top, and face 2 is in front, then face 3 will be seen on the right side of the die, as in the figure on the right.
- ❖ Faces 4, 5, and 6 are opposite to faces 3, 2, and 1, respectively.

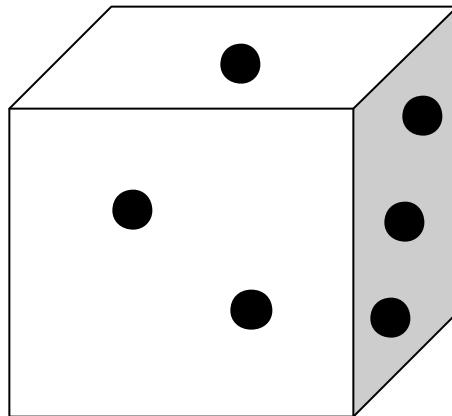


Figure 1.

In the scenario of this problem, a die will be placed on top of a horizontal xy -plane (Figure 2).

- ❖ One unit length in the xy -plane is equal to the side length of the die.
- ❖ Initially, the center of the bottom face of the die will coincide with point $(0, 0)$ of the xy -plane.
- ❖ The xy -plane is situated so that the positive y -axis points to the north, and the positive x -axis points to the east.
- ❖ The die will always be observed from the south (looking north), so that the front side of the die is the one facing south.

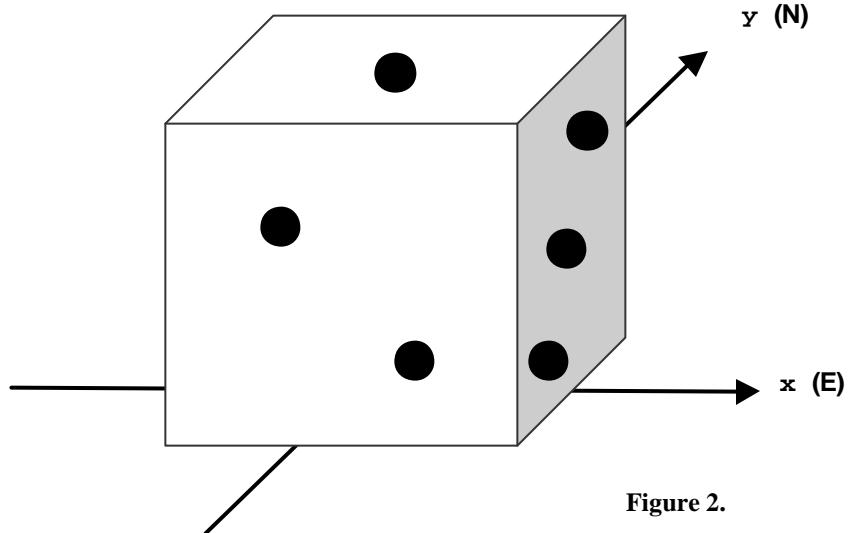


Figure 2.

The action which will be performed on the die will be referred to as "rolling the die". This action, however, will be very different from the conventional meaning of rolling a die.

- ❖ The die will always be rolled in one of the four directions north, south, east, and west.
- ❖ Rolling the die to the north, for example, will mean that the die is rotated through an angle of 90 degrees about its north bottom edge.

Starting with position and configuration shown in Figure 2, let us roll the die to the north.

- ❖ The position of the die will now change from $(0, 0)$ to $(0, 1)$.
- ❖ The face 2 will now be on top and the face 6 will be in front.

The resulting position and configuration are shown in Figure 3.

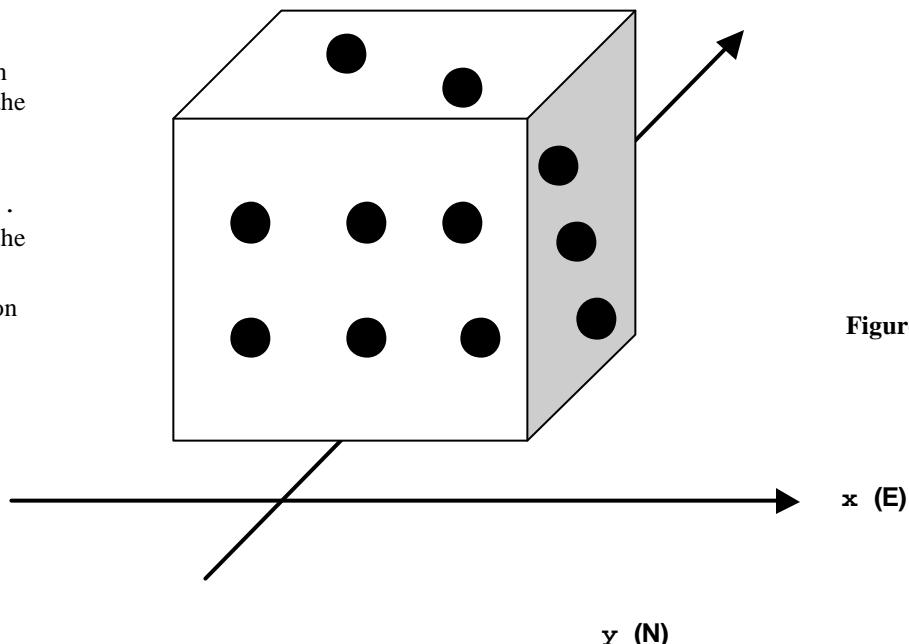


Figure 3.

A further roll to the east will change the position to $(1, 1)$ and bring face 4 to the top, as shown in Figure 4.

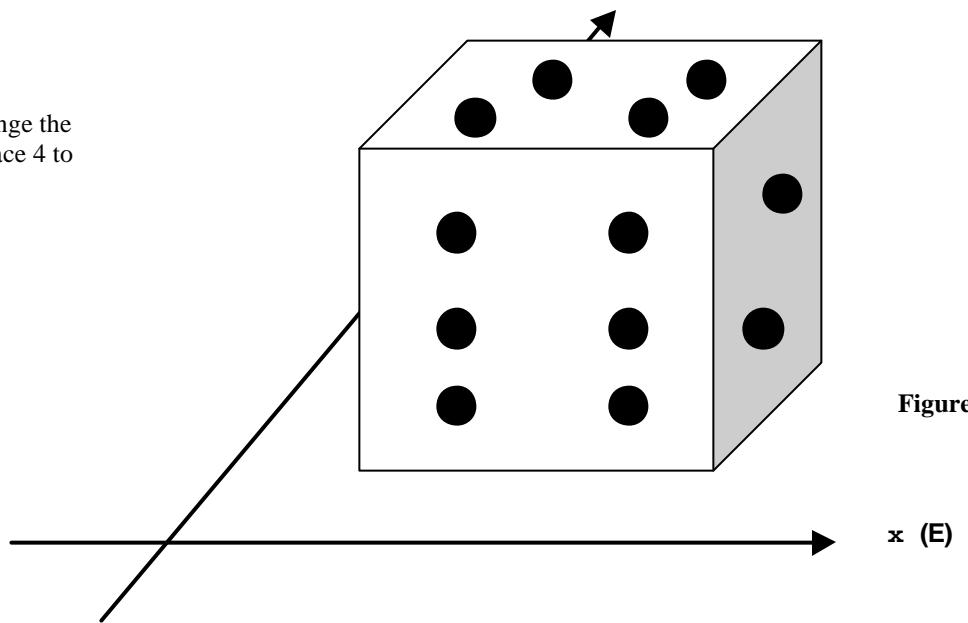


Figure 4.

❖ Here is an example of the input file prog2.dat to your program:

```
1 2 W W W N
1 2 N W W W
1 6 S S S W W W W W
3 3 N N S
4 6 E W W S S S S E S S S S S S S E
```

❖ Each line of the input file consists of:

- ❖ Two integers in the range $1 \dots 6$, separated by one blank space.
- ❖ These integers specify the initial orientation of the die. (As stated above, the initial position of the die is always $(0, 0)$, so it does not need to be specified.)
- ❖ The first of the two integers specifies the top face of the die, the second one specifies its front face.
- ❖ One or more (but not more than 35) upper case letters, each of which is 'N', 'S', 'E', or 'W'.
- ❖ Each letter is preceded by one blank space.

- These letters specify a sequence of rolls of the die. Each roll will occur to the north, south, east, or west, as indicated by the corresponding upper case letter.

Given the above input, your program will write the following output to prog2.out:

Problem 2 by team x

```

Initial orientation:      top = 1  front = 2
Moves: W W W N
Final orientation and position: top = 2  front = 3  x = -3  y = 1

Initial orientation:      top = 1  front = 2
Moves: N W W W
Final orientation and position: top = 4  front = 6  x = -3  y = 1

Invalid initial orientation:      top = 1  front = 6
Invalid initial orientation:      top = 3  front = 3

Initial orientation:      top = 4  front = 6
Moves: E W W S S S S E S S S S S S E E S S S S S S E
Final orientation and position: top = 2  front = 4  x = 3  y = -17
End of problem 2 by team x

```

For each line of input that contains a valid initial orientation (such as lines 1, 2, and 4 in the above example), there will be exactly four lines of output:

A blank line.

A line specifying the initial orientation, read from the input file.

A line specifying the sequence of moves, read from the input file.

A line specifying the final orientation and position of the die. You may assume that the values of *x* and *y* will be in the range -99..99.

For each line of input that contains an invalid initial orientation (the two given faces of the die are not adjacent, or the same face is given for the top and the front), there will be exactly two lines of output:

A blank line.

A line stating that the initial orientation is invalid and specifies the values of *top* and *front*, read from the input file.

Observe every detail of the output, such as the exact wording and punctuation of statements, upper/lower case variations, blank lines and blank spaces. Note in particular, that the values of *top* and *front* are printed in fields of width 2, whereas the values of *x* and *y* are printed in fields of width 4 (including the blank space which follows the equal sign).

A few lines of the above output will be reproduced here with formatting templates:

```

1      2      3      4      5      6      7
123456789012345678901234567890123456789012345678901234567890
Invalid initial orientation:      top = 1  front = 6

Initial orientation:      top = 4  front = 6
Moves: E W W S S S S E S S S S S S E E S S S S S S E
1      2      3      4      5      6      7
123456789012345678901234567890123456789012345678901234567890
Final orientation and position: top = 2  front = 4  x = 3  y = -17
End of problem 2 by team x

```

Problem 3: April Fool's Joke (2 pages)

High on the wall of the church tower in a small village in Transylvania is an analog clock. It has an hour hand and a minute hand and a conventional 12-hour dial. The clock's mechanism advances both hands by the appropriate amount once every minute.

During the night preceding April Fool's Day, two mischievous and mechanically skilled persons embark on a clandestine expedition. They ascend the church tower and interchange ("swap") the clock's two hands, so that each hand is pointing in the direction in which the other hand should be pointing.

For example, suppose the true time is 7:18. Then we have the scenario depicted in Figures 1a and 1b:

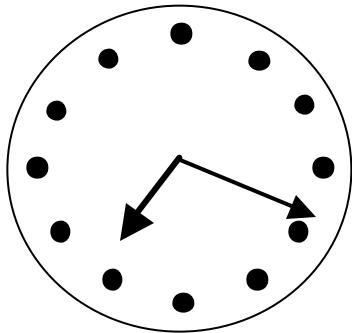


Figure 1a: True time

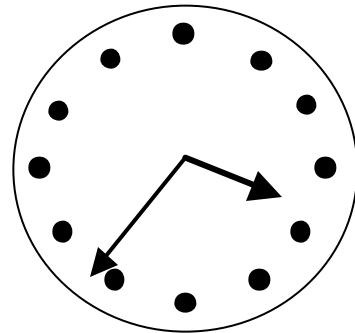


Figure 1b: Swapped time

The observer who looks at the swapped clock at this time (assuming he/she has no preconceived notion of what time it ought to be) will say: "It is 3:37."

Just two minutes later, at 7:20, however, the scenario will be very different, as shown in Figures 2a and 2b:

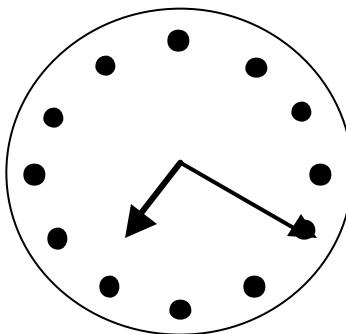


Figure 2a: True time

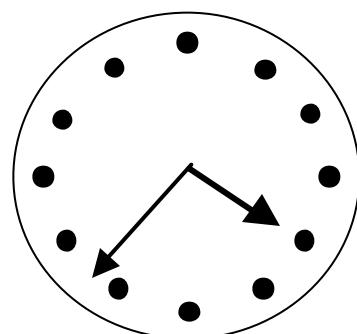


Figure 2b: Swapped time

At this point, the observer who looks at the swapped clock will notice that the hour hand is pointing directly at the 4 o'clock position, so it ought to be very close to 4 o'clock. But the minute hand indicates that it is 37 minutes past the hour, which is not compatible with the hour hand's position. In this case, we say that the swapped clock's configuration is not valid.

Here is an example of the input file **prog3.dat** for your program:

```
7      18
7      19
7      20
3      0
3      2
4      30
12     15
```

Each line of the input file contains two integers (separated by one or more blank spaces) that specify the hours and minutes of the current true time, respectively. The value of hours will be in the range 1 .. 12, the value of minutes will be in the range 0 .. 59.

For each line of input, your program will determine whether the swapped clock's hands will form a valid configuration, and if so, print out the time shown by the swapped clock.

More specifically, here is the output file **prog3.out** resulting from the input file listed above:

Problem 3 by team x

```
=====
True      True      Minutes From      Minutes From      Discrepancy      Swapped      Swapped
Hours     Minutes   Minute Hand In   Hour Hand In   Of Minutes      Hours       Minutes
                                                Swapped Clock   Swapped Clock
=====
7        18          37                36                1            3          37
7        19          37                48                11           3          37
7        20          37                0                 23           --          --
3        0           15                0                 15           --          --
3        2           15                24                9            12          15
4        30          23                0                 23           --          --
12       15          1                 0                 1            3          1
=====
```

End of problem 3 by team x

- The numbers in the first two vertical fields of the output just echo the input.
- From the input values, it is straightforward to calculate the position of the minute hand in the swapped clock, which determines a certain number of minutes past the hour, as listed in the third field of the output.
- From the input values, it is also straightforward to calculate the position of the hour hand in the swapped clock. The position of the hour hand actually determines hours as well as minutes; the minutes are listed in the fourth field of the output.
- The discrepancy between the two results for minutes, listed in the fifth field of the output, is first calculated as the absolute value of the difference between the two results for minutes. This discrepancy, however, cannot exceed 30 minutes, so (as on the third line of numerical output) it may need to be subtracted from 60 minutes.
- Results in the third and fourth output fields are rounded to the nearest integer, if necessary. The discrepancy (in the fifth output field) is calculated from the integer values displayed in the third and fourth output fields.
- The configuration of the swapped clock is considered to be valid if the discrepancy does not exceed 12 minutes.
 - In this case, the swapped clock's time is printed in the sixth and seventh fields of the output. The value of minutes displayed in the seventh field is identical to the value displayed in the third field.
 - Otherwise, two hyphens are displayed in each of the sixth and seventh output fields.
- Observe every detail of the output, such as the exact wording of the headers, upper/lower case variations, separator lines (of equal signs) and blank spaces.
- A few lines of the above output will be reproduced here with a formatting template:

```
1      2      3      4      5      6      7
1234567890123456789012345678901234567890123456789012345678
True      True      Minutes From      Minutes From      Discrepancy      Swapped      Swapped
Hours     Minutes   Minute Hand In   Hour Hand In   Of Minutes      Hours       Minutes
                                                Swapped Clock   Swapped Clock
=====
7        18          37                36                1            3          37
=====
```

Problem 4: Reciprocals (1 page)

The reciprocal of a positive integer is 1 divided by that integer. For example, the reciprocal of 37 is $1/37$. When the division is carried out to evaluate a reciprocal in decimal form, the result often turns out to be an infinitely repeating decimal. For example, $1/37 = 0.0270270270\dots$ where the group of digits 027 repeats infinitely.

The subject of this problem, however, is a positive integer whose reciprocal, when written in the decimal point format, ends after a finite number of digits. Such a positive integer can be

- a power of 10, or
- a power of 2, or
- a power of 2 followed by trailing zeros, or
- a power of 5, or
- a power of 5 followed by trailing zeros.

The input to your program, `prog4.dat`, will contain a list of such positive integers, one per line, as shown in the following example:

2
100
6250000
20480000000000000
8470329472543003390683225006796419620513916015625

Each positive integer in the input file will

~~✓~~ begin in column 1
~~✓~~ be greater than 1
~~✓~~ contain at most 72

It is easy to see that each of the first four numbers in the example shown here falls into one of the five categories listed above. The fifth one happens to be 5 raised to the 70^{th} power.

For each number in the input, your program will write three lines of output to `prog4.out`:

- ❑ one blank line,
- ❑ one line that reproduces the given number from the input, in the format shown in the sample output below,
- ❑ one line that displays the exact value of the decimal representation of the reciprocal of the given number.

For the input shown above, the output is the following:

Problem 4 by team x

$$\begin{array}{r} 1 \quad / \quad 2 \quad = \\ 0.5 \end{array}$$

$$1 / 100 =$$

$$1 / 6250000 = 0.00000016$$

1 / 20480000000000000000 =
0.00000000000000048828125

☞ Pay attention to every detail of the output format. In particular:

- each non-blank line will begin in column 1;
- the only blank lines are the ones specified above;
- there is exactly one blank on each side of the “/” and one blank preceding the equal sign;
- there is exactly one 0 preceding the decimal point in the representation of the reciprocal.

~~2.2~~ You may assume that the decimal representation of the reciprocal will not contain more than 78 characters (including the leading 0 and the decimal point).

Problem 5: Subdirectories (3 pages)

Your program will simulate the creation of subdirectories (folders) on one of the disks of a computer. The input file to your program, *prog5.dat*, will contain a sequence of commands that a user might enter from a command line, and the output file *prog5.out* will contain the operating system's responses to these commands. Below is an example of an input file, and on the right is the listing of the corresponding output file.

```
dir
mkdir sub6
mkdir sub3
mkdir sub4
dir
mkdir sub4
cd sub3
cd sub3
mkdir sub3
mkdir sub6
mkdir sub4
dir
cd sub6
mkdir sub666
dir
up
up
dir
up
```

```
Problem 5 by team X
Command: dir
Directory of root:
No subdirectories
Command: mkdir sub6
Command: mkdir sub3
Command: mkdir sub4
Command: dir
Directory of root:
sub3 sub4 sub6
Command: mkdir sub4
Subdirectory already exists
Command: cd sub3
Command: cd sub3
Subdirectory does not exist
Command: mkdir sub3
Command: mkdir sub6
Command: mkdir sub4
Command: dir
Directory of root\sub3:
sub3 sub4 sub6
Command: cd sub6
Command: mkdir sub666
Command: dir
Directory of root\sub3\sub6:
sub666
Command: up
Command: up
Command: dir
Directory of root:
sub3 sub4 sub6
Command: up
Cannot move up from root directory
End of problem 5 by team X
```

The four commands that can appear in the input file are:

dir	Display the path and the subdirectories of the current default directory, the latter in lexicographic order.
mkdir <name>	Create a subdirectory of the current default directory with the specified name.
cd <name>	Change the default to a specified subdirectory of the current default directory.
up	Change the default to the parent directory of the current default directory.

Each line of the input file begins with one of the four commands, in lower case letters.

The commands *mkdir* and *cd* in the input file will be followed (starting in column 9) by an argument (the subdirectory name).

The argument will consist of at least one and at most six characters that can be upper or lower case letters, digits, or underscores; in particular, the argument will not contain blanks.

The argument will be followed immediately by the end-of-line character.

The commands *dir* and *up* do not take an argument, they will be followed immediately by the end-of-line character.

Before responding to each command in the input file, your program will echo the command :

The command will be displayed starting in column 10 (after "Command : ");

If there is an argument, it will be displayed starting in column 18.

In response to the `dir` command, your program will:

Display the path from the root directory to the current default directory, in the format of the sample output

In the example shown, the `dir` command appears four times in the input file. The first two times the path is `root`, the second time it is `root\sub3`, the third time it is `root\sub3\sub6`.

If the current default directory contains no subdirectories, display the message to that effect; otherwise display in lexicographic order all subdirectories of the current default directory.

The subdirectories will be left-justified in fields of width 8.

There is no specific limit on the number of subdirectories of a particular directory. If there are more than ten, the displayed list will wrap around to the beginning of the next line after each group of ten subdirectories. For example, if the subdirectory `sub666` in the above example had 20 siblings `sub601`, `sub602`, etc., the output from the `dir` command would be:

Directory of `root\sub3\sub6`:

```
sub601  sub602  sub603  sub604  sub605  sub606  sub607  sub608  sub609  sub610
sub611  sub612  sub613  sub614  sub615  sub616  sub617  sub618  sub619  sub620
sub666
```

The commands `mkdir`, `cd`, and `up` will not produce any immediate response, unless an error message is in order. The reasons for possible error messages are as follows:

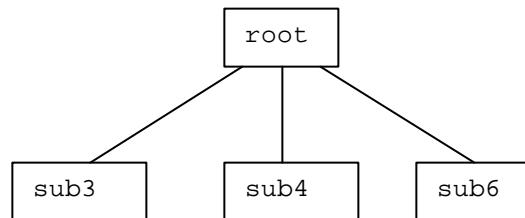
In response to the command `mkdir`: the current default directory already has a subdirectory whose name is specified in the argument.

In response to the command `cd`: the current default directory does not have a subdirectory whose name is specified in the argument.

In response to the command `up`: the current default directory is the root directory.

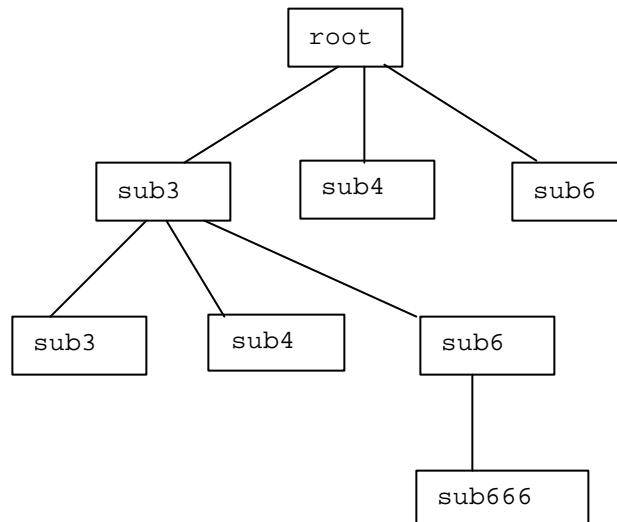
At program startup, the default directory is the root directory, and it has no subdirectories.

In the example shown above, the effect of the first group of three `mkdir` commands is to create the directory tree shown on the right.



After changing the default directory to `sub3`, executing the second group of three `mkdir` commands, changing the default directory to the subdirectory `sub6` and executing the last `mkdir` command, the resulting directory tree is shown on the right.

Note that subdirectories that are not siblings (do not have the same parent) may have the same name.



There is no limit on the number of subdirectories of a particular directory, nor is there a limit on the maximum number of levels in the directory tree. The total number of subdirectories successfully created by the program will not, however, exceed 5000.

Pay close attention to every detail of the output, such as wording and punctuation, upper/lower case variations, number of blank spaces, and the absence of blank lines.

A few lines of the above output are reproduced here with a formatting template:

123456789012345678901234567890

Command: mkdir sub4

Command: dir

Directory of root\sub3:

sub3 sub4 sub6

Command: cd sub6

Command: mkdir sub666

Problem 6: Mountain Landscape (4 pages)

Two seaports at sea level are separated by a mountainous stretch of dry land.

Between the two seaports are N mountain peaks, where N is a positive integer, in the range $1 \dots 6$.

The N mountain peaks will be denoted by P_1, P_2, \dots, P_N . The two seaports will be denoted by P_0 and P_{N+1} .

The N mountain peaks and the two seaports all lie in the same vertical plane.

The seaport P_{N+1} is situated directly due east of the seaport P_0 .

The mountain peaks P_1, P_2, \dots, P_N are numbered in west-to-east order.

It will be further assumed that neighboring points of this configuration (P_0 and P_1 , P_1 and P_2 , etc.) are in view of each other.

Figure 1 shows an example for the case $N = 3$:

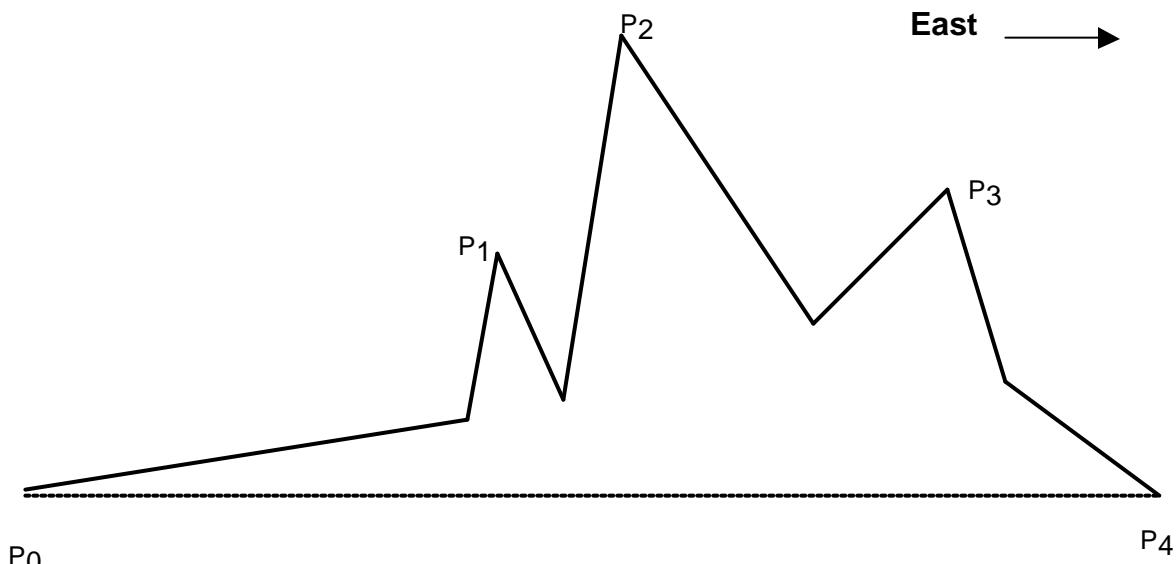


Figure 1. Two seaports and three mountain peaks

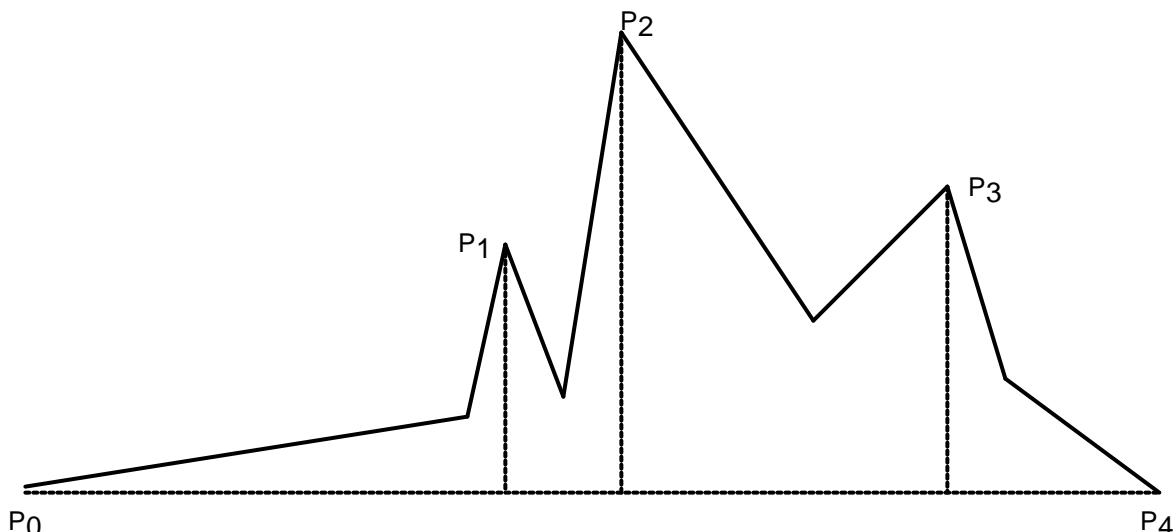


Figure 2. Task goals: determine the altitudes (vertical dotted lines) and pipeline length (P_0 to P_4)

The Corps of Engineers has been assigned the tasks of determining

the altitudes (elevations above sea level) of the N mountain peaks, and

the length of a straight pipeline which will connect the two seaports.

Figure 2 on the preceding page illustrates the altitudes and pipeline length to be determined.

In order to accomplish their mission, the engineers will take two sets of measurements:

N distance measurements, and

$N+1$ angle of elevation measurements.

Distance measurements:

The straight-line distance between each pair of points (P_k, P_{k+1}) , where $k = 0, 1, 2, \dots, N-1$, is measured. Figure 3 shows the measured distances for the example of Figure 1. (This distance could be measured by actually measuring the time it takes a sound wave to cover the distance.)

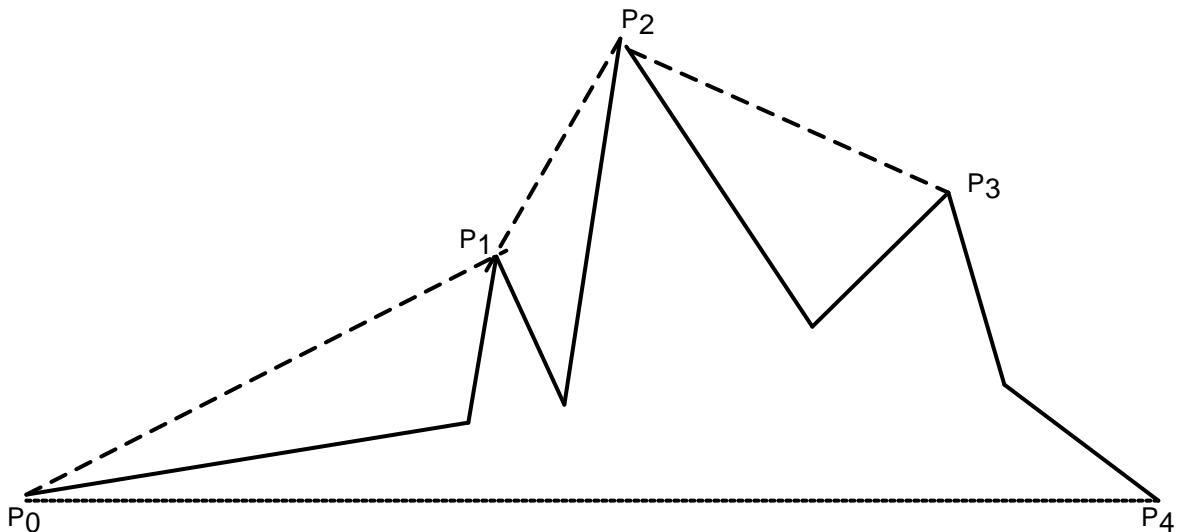


Figure 3: The measured distances are D_0, D_1 , and D_2 .

Angle of elevation measurements:

For each pair of points (P_k, P_{k+1}) , where $k = 0, 1, 2, \dots, N$, the angle of elevation of P_{k+1} , viewed from P_k , will be measured; it is the angle formed by a horizontal line and the line segment connecting P_k with P_{k+1} . Figure 4 on the following page shows the four angles A_0, \dots, A_3 to be measured for the example of Figure 1. For sake of clarity, the solid lines (representing the actual mountain landscape) have been removed from Figure 4.

The angles are measured in degrees.

When the point P_k is located at a higher elevation than the point P_{k+1} , then the angle is negative.

In Figure 4, the angles measured at P_0 and P_1 are positive, whereas the angles measured at P_2 and P_3 are negative.

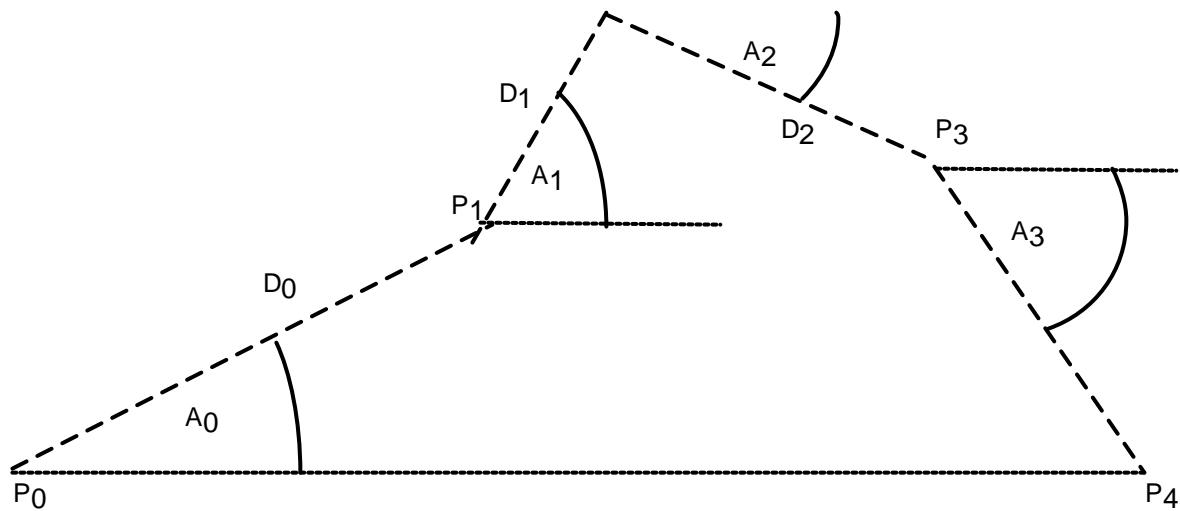


Figure 4: With 3 mountain peaks, 4 angles will be measured.

The input to your program will come from prog6.dat; here is an example of an input file:

```
27.18 705.3 57.76 328.4 -24.3 472 -54.9
45 141 30 200 -40.5 120 21.4 450 -10.3 110.3 28.3 185.2 -60
```

- ❖ The number of lines in the input file will be at least 1 but has no upper bound.
- ❖ Each line of the input file contains data for a self-contained problem of two seaports and N mountain peaks, as previously described.
- ❖ Each line of input has the format

$A_0 D_0 A_1 D_1 \dots A_{N-1} D_{N-1} A_N$

- ❖ The distances D_k and angles A_k have been previously defined.
- ❖ Each distance D_k has at most three digits before the decimal point and at most one digit after the decimal point.
- ❖ Each angle A_k has at most two digits after the decimal point. The fine points of angle measurement and representation have been discussed in previous paragraphs.
- ❖ All angles are greater than -90 degrees and less than 90 degrees.
- ❖ The first angle on each line is positive, the last angle on each line is negative.
- ❖ The maximum number of peaks in a mountain landscape represented by one line of input data is 6, which means 7 angles and 6 distances.
- ❖ The minimum number of peaks in a mountain landscape represented by one line of input data is 1, which means 2 angles and 1 distance.

The output from your program will be written to `prog6.out`. The input data file listed above will result in the following output:

Problem 6 by team x

```
Altitude of peak 1 = 322.2
Altitude of peak 2 = 599.9
Altitude of peak 3 = 405.7
Pipeline length     = 1517.9
```

```
Altitude of peak 1 = 99.7
Altitude of peak 2 = 199.7
Altitude of peak 3 = 121.8
Altitude of peak 4 = 286.0
Altitude of peak 5 = 266.2
Altitude of peak 6 = 354.0
Pipeline length     = 1259.1
End of problem 6 by team x
```

For each line of input, the output will consist of:

one blank line;

the altitude of each peak, one peak per line, enumerated in increasing order of peak numbers, in the format shown above;

the length of the pipeline, in the format shown above.

Each altitude, as well as the length of the pipeline, will be displayed rounded to one digit after the decimal point; it will thus consist of at least one but no more than four digits before the decimal point, and exactly one digit following the decimal point.

Observe every detail of the output, such as the exact wording of statements, upper/lower case variations, blank lines and blank spaces.

A few lines of the above output will be reproduced here with a formatting template:

```
1           2
123456789012345678901234567
Altitude of peak 2 = 599.9
Altitude of peak 3 = 405.7
Pipeline length     = 1517.9
```

Problem 7: Lumber Cutting (2 pages)

Carpenters are often required to cut a set of pieces of given lengths, but can only obtain lumber at a fixed length. They must therefore determine how to use the length available most economically to make the parts. Lumber companies will not refund a partially cut two-by-four, and so using one foot of a twelve-foot two-by-four is no better of a solution than using all of it, since whatever remains will be discarded.

The input to your program will come from `prog7.dat`. Each line of the input file will constitute a “job”, represented in the form

`<board_length> <saw_width> <part1_length> <part2_length> ...`

where

`<board_length>` is the length of the original lumber

`<saw_width>` is the width of the cutting tool, the amount of wood that will become sawdust with each cut

`<part1_length> <part2_length> ...` the remaining entries on the line constitute the list of the lengths of the desired parts, in non-decreasing order.

Here is an example of an input file `prog7.dat`:

```
1000 100 250 250 500 650 1000
1000 50 200 250 250 500 650 970
```

Further facts about the input:

`<board_length>` All numbers in the input will be positive integers (thousandths of inches or centimeters if you like).

`<saw_width>` The board length will not exceed 30000, the saw width will not exceed 1000, and each part length will not exceed 9999.

`<part1_length> <part2_length> ...` The saw width will be less than the smallest part length.

`<part1_length> <part2_length> ...` Each part length will be less than or equal to the board length.

`<part1_length> <part2_length> ...` The number of part lengths on each line will not exceed 12.

`<part1_length> <part2_length> ...` Each line of input will provide valid input for one job, i.e., it will give rise to some output as described below.

For each job (each line of input), the output (written to `prog7.out`) will consist of

`<output>` one blank line

`<output>` one line that displays the board length

`<output>` one line that displays the saw width

`<output>` one line that displays the number of boards needed for the job

`<output>` **the number of boards used for each job will be as small as possible**

`<output>` one or more additional lines (one for each board), each line containing

`<output>` **Board <sequence_number>:**

`<output>` where the sequence numbers will start at 1 and increase by 1 on each successive line;

`<output>` a list of (at least 1 but no more than 12) lengths of parts, in any order, that will be cut from that particular board;

`<output>` each part length will appear exactly as many times on these lines of the output as it did in the input

`<output>` the part lengths for a particular board, plus the saw width added an appropriate number of times, will not exceed the board length.

Furthermore,

`<output>` for each job there is only one correct answer for the number of boards, but

`<output>` different configurations of parts cut from each board can constitute correct output, as long as the conditions listed above are met. In fact, two different output files that could arise from the above input are listed on the next page.

The sample output files listed on the next page illustrate some of the subtleties of the problem:

`<output>` In the first job (with saw width = 100 units)

`<output>` when two parts (lengths = 250 and 650 units) are cut from a 1000 unit long board, one cut will turn 100 units of lumber into sawdust and there is no discarded lumber.

`<output>` when one 1000 unit long part is required, it is obtained as the entire board, without requiring any cuts

`<output>` when two parts (lengths = 250 and 500 units) are cut from a 1000 unit long board, two cuts are required, 200 units of lumber will turn into sawdust, leaving a 50 unit piece of discarded lumber.

`<output>` In the second job (with saw width = 50 units)

`<output>` when two parts (lengths = 250 and 650 units) are cut from a 1000 unit long board, two cuts are required, turning 100 units of lumber into sawdust and leaving no discarded lumber.

`<output>` when one 970 unit long part is required, one cut is required, turning 30 units of lumber into sawdust and leaving no discarded lumber.

Problem 7 by team x

Board length = 1000

Saw width = 100

Number of boards needed = 3

Board 1: 250 650

Board 2: 1000

Board 3: 250 500

Board length = 1000

Saw width = 50

Number of boards needed = 4

Board 1: 250 650

Board 2: 970

Board 3: 250 500

Board 4: 200

End of problem 7 by team x

Problem 7 by team x

Board length = 1000

Saw width = 100

Number of boards needed = 3

Board 1: 500 250

Board 2: 650 250

Board 3: 1000

Board length = 1000

Saw width = 50

Number of boards needed = 4

Board 1: 250 250 200

Board 2: 500

Board 3: 650

Board 4: 970

End of problem 7 by team x

Observe every detail of the output, such as the exact wording of statements, upper/lower case variations, blank lines and blank spaces.

The board length, the saw width, and the number of boards are right-justified in a field of width 6 starting immediately after the equal sign.

Each board number is right-justified in a field of width 3 starting immediately after the word Board.

Starting immediately after the colon, each part length is right-justified in a field of width 5.

A few lines of the above output will be reproduced here with a formatting template:

1	2	3	4	5	6
123456789012345678901234567890123456789012345678901234567890123456789					
Number of boards needed =	4				
Board 1:	250	250	200		

Problem 8: Millie's Restaurant (3 pages)

- On most days, the staff of Millie's Restaurant prepares one or more servings of one or more dishes before the restaurant is opened for business. At the end of the day, each serving (which was prepared fresh on that day but not sold) is individually packaged, labeled, and placed in the freezer. The label identifies the dish and the day on which it was prepared.
- When a customer orders a particular dish, the waitress will first try to fill the customer's order from the servings which were prepared fresh on that day. If a serving of a particular dish is not found among the freshly prepared dishes, then it will be taken out of the freezer. In that case, if there are servings with different preparation dates in the freezer, the serving taken out of the freezer is chosen to be one that has been in the freezer for the longest time.
- The input to your program will come from two files: prog8p.dat and prog8s.dat. These files contain, respectively, day-by-day listings of dishes prepared and sold. We will refer to them as the prepared dishes file and the sold dishes file, respectively. Here is an example for each:

prog8p.dat

```

1 10 Southern Fried Chicken
1 14 Alaskan King Crab
1 15 Vegetarian Lasagna
2 3 Vegetarian Lasagna
2 2 Southern Fried Chicken
4 4 Vegetarian Lasagna
4 7 Alaskan King Crab
4 8 Southern Fried Chicken
7 6 Vegetarian Lasagna

```

prog8s.dat

```

1 8 Southern Fried Chicken
2 4 Alaskan King Crab
2 3 Southern Fried Chicken
3 2 Vegetarian Lasagna
3 10 Alaskan King Crab
5 3 Alaskan King Crab
5 2 Southern Fried Chicken
5 15 Vegetarian Lasagna

```

- The two input files have the same format.
- In columns 1 . . 3 the day of preparation (or sale) is right-justified. The days are numbered starting at 1. The largest number of days is 999.
- Column 4 is blank.
- In columns 5 . . 6 the number of servings of a dish is right-justified. The largest number of servings that can appear on one line of either file is 99.
- Column 7 is blank.
- In columns 8 – 37, the name of a dish prepared (or sold) is left-justified.
- The days on successive lines will be in non-decreasing order (which means increasing, with possible repetitions).
- In the prepared dishes file there will be at least one entry for day 1.
- There may be days on which no dish was prepared but some dishes were sold (days 3 and 5 in the above example).
- There may be days on which no dish was sold but some dishes were prepared (day 4 in the above example).
- There may be days of total inactivity, such as day 6 in the above example.
- You may assume that all servings that appear in the sold dishes file were actually present (either as freshly prepared or as frozen dishes) on the day of sale.
- No dish will appear in either file more than once within the span of one day.
- There is no specific limit on the number of different dishes that may appear in the prepared dishes file, but at no time will the number of different dishes (whether freshly prepared or frozen) exceed 20.
- There is no explicit limit on the number of lines in the input files, except for limits that can be deduced from previously stated limits.
- The largest day number in the prepared dishes file may be less than, equal to, or greater than that in the sold dishes file.
- The output, written by your program to prog8.out, will be a sequence of daily reports on the freezer's contents. The output corresponding to the above input is listed on the next page.

Problem 8 by team x

Frozen dishes at the end of day 1:

Dish	Prepared on day	Quantity
Alaskan King Crab	1	14
Southern Fried Chicken	1	2
Vegetarian Lasagna	1	15

Frozen dishes at the end of day 2:

Dish	Prepared on day	Quantity
Alaskan King Crab	1	10
Southern Fried Chicken	1	1
Vegetarian Lasagna	1	15
	2	3

Frozen dishes at the end of day 3:

Dish	Prepared on day	Quantity
Southern Fried Chicken	1	1
Vegetarian Lasagna	1	13
	2	3

Frozen dishes at the end of day 4:

Dish	Prepared on day	Quantity
Alaskan King Crab	4	7
Southern Fried Chicken	1	1
	4	8
Vegetarian Lasagna	1	13
	2	3
	4	4

Frozen dishes at the end of day 5:

Dish	Prepared on day	Quantity
Alaskan King Crab	4	4
Southern Fried Chicken	4	7
Vegetarian Lasagna	2	1
	4	4

Frozen dishes at the end of day 6:

Dish	Prepared on day	Quantity
Alaskan King Crab	4	4
Southern Fried Chicken	4	7
Vegetarian Lasagna	2	1
	4	4

Frozen dishes at the end of day 7:

Dish	Prepared on day	Quantity
Alaskan King Crab	4	4
Southern Fried Chicken	4	7
Vegetarian Lasagna	2	1
	4	4
	7	6

End of problem 8 by team x

Starting with day 1, the output will contain the report for each day, through the last day of the period, which is either the last day in the prepared dishes file, or the last day in the sold dishes file, whichever occurred later.

For each day, the daily report will begin with the following four lines:

one blank line (there are no other blank lines in the output),

one line which identifies the day;

the day number is right-justified in a field of width 4 following the word "day";

there is no blank between the day number and the colon which follows it;

a header line,

a line of 50 equal signs.

The line of equal signs will be followed by a list of the dishes of which there are currently servings in the freezer, along with the day on which those servings were prepared, and the number of servings.

The dishes will be listed in lexicographic order under each day.

No dish name will appear more than once under one day.

There may be servings of a dish which were prepared on different days. (In the above example, on day 4, there are servings of Southern Fried Chicken which were prepared on days 1 and 4, and servings of Vegetarian Lasagna which were prepared on days 1, 2, and 4.) In this case the information about days of preparation and numbers of servings will be listed in increasing order of the preparation day, without repeating the name of the dish (the name of the dish appears only on the first line for that particular dish).

Under one dish, no preparation day will appear more than once.

The names of the dishes will be left-justified in columns 1..30.

The preparation days will be right-justified and aligned with the field headers (see the formatting template shown below).

Pay close attention to every detail of the output, such as wording and punctuation, upper/lower case variations and number of blank spaces. A few lines of the above output are reproduced here with a formatting template:

1	2	3	4	5
---	---	---	---	---

12345678901234567890123456789012345678901234567890

Frozen dishes at the end of day 3:

Dish	Prepared on day	Quantity
Southern Fried Chicken	1	1
Vegetarian Lasagna	1	13
	2	3