Problem A
Budget Travel

An American travel agency is sometimes asked to estimate the minimum cost of traveling from one city to another by automobile. The travel agency maintains lists of many of the gasoline stations along the popular routes. The list contains the location and the current price per gallon of gasoline for each station on the list.

In order to simplify the process of estimating this cost, the agency uses the following rules of thumb about the behavior of automobile drivers.

- A driver never stops at a gasoline station when the gasoline tank contains more than half of its capacity unless the car cannot get to the following station (if there is one) or the destination with the amount of gasoline in the tank.
- A driver always fills the gasoline tank completely at every gasoline station stop.
- When stopped at a gasoline station, a driver will spend $2.00 on snacks and goodies for the trip.
- A driver needs no more gasoline than necessary to reach a gasoline station or the city limits of the destination. There is no need for a “safety margin.”
- A driver always begins with a full tank of gasoline.
- The amount paid at each stop is rounded to the nearest cent (where 100 cents make a dollar).

You must write a program that estimates the minimum amount of money that a driver will pay for gasoline and snacks to make the trip.

Input

Program input will consist of several data sets corresponding to different trips. Each data set consists of several lines of information. The first 2 lines give information about the origin and destination. The remaining lines of the data set represent the gasoline stations along the route, with one line per gasoline station. The following shows the exact format and meaning of the input data for a single data set.

Line 1: One real number — the distance from the origin to the destination
Line 2: Three real numbers followed by an integer
- The first real number is the gallon capacity of the automobile’s fuel tank.
- The second is the miles per gallon that the automobile can travel.
- The third is the cost in dollars of filling the automobile’s tank in the origination city.
- The integer (less than 51) is the number of gasoline stations along the route.

Each remaining line: Two real numbers
- The first is the distance in miles from the origination city to the gasoline station.
- The second is the price (in cents) per gallon of gasoline sold at that station.

All data for a single data set are positive. Gasoline stations along a route are arranged in nondescending order of distance from the origin. No gasoline station along the route is further from the origin than the distance from the origin to the destination There are always enough stations appropriately placed along the each route for any car to be able to get from the origin to the destination.

The end of data is indicated by a line containing a single negative number.

Output

For each input data set, your program must print the data set number and a message indicating the minimum total cost of the gasoline and snacks rounded to the nearest cent. That total cost must include the initial cost of filling the tank at the origin. Sample input data for 2 separate data sets and the corresponding correct output follows.

Sample Input

```
475.6
11.9 27.4 14.98 6
102.0 99.9
220.0 132.9
256.3 147.9
275.0 102.9
277.6 112.9
381.8 100.9
516.3
15.7 22.1 20.87 3
125.4 125.9
297.9 112.9
345.2 99.9
-1
```

Output for the Sample Input

```
Data Set #1
  minimum cost = $27.31
Data Set #2
  minimum cost = $38.09
```

Problem B
Classifying Lots in a Subdivision

A subdivision consists of plots of land with each plot having a polygonal boundary. A surveyor has surveyed the plots, and has given the location of all boundary lines. That is the only information available, however, and more information is desired about the plots in the subdivision. Specifically, planners wish to classify the lots by the number of boundary line segments (*B*=3,4,5,…) on the perimeter of the lots.

Write a program that will take as input the surveyor's data and produce as output the desired information about the nature of the lots in the subdivision.

Input
The input file consists of several data sets. Each data set begins with a line containing the number of line segments (4 • *N* • 200) in the survey. The following *N* lines each contain four integers representing the Cartesian (*x*,*y*) coordinate pairs for the *N* points of a boundary line segment. The input file is terminated with a 0.

Output
For each data set, provide output listing the number of lots in each classification of boundary line segment counts (*B*=3,4,5,…). Do not include in your output those cases in which the classification has no members. The output for each data set will begin with a line containing an appropriately labeled data set number. Output for successive data sets will be separated by a blank line.

Assumptions:
1. *Each data set corresponds to a rectangular subdivision (as in Figures 1 and 2). The boundaries of the rectangular subdivision are parallel to the x and y axes.*
2. *All coordinates in the input file are positive integers in the range 1 to 10000.*
3. *Boundary line segments in the input file do not extend past corners of lots. For example, in Figure 1 the surveyor must survey from the point (10,41) to (15,41) and from (15,41) to (20,41) rather than surveying the entire line (10,41) to (20,41).*
4. *At least one boundary line segment in each lot lies on the subdivision's bounding rectangle.*

Figures 1 and 2 show two hypothetical subdivisions. In Figure 1 there are 12 boundary line segments, and in Figure 2 there are 27. The sample input file below contains the data for these two test cases. The plot in the upper left hand corner of Figure 2 has one line running from (16,16) to (17,18) and another from (17,18) to (19,22). Thus this lot has a perimeter comprised of 5 boundary line segments, though geometrically the lot is a 4-sided region. Similarly the perimeter of the plot in the upper left hand corner of Figure 1 is comprised of 6 boundary line segments, though the lot is pentagonal in shape.

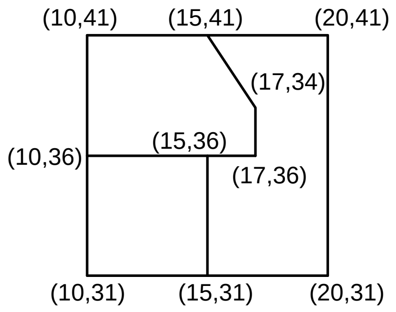


Figure 1

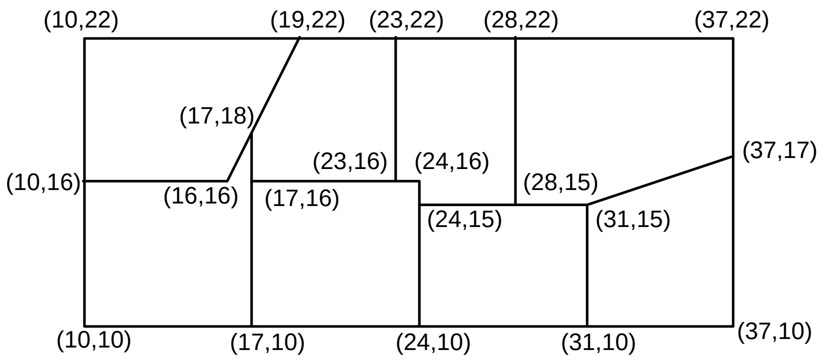


Figure 2

Sample Input

```
12
10 41 15 41
15 41 20 41
10 36 15 36
15 36 17 36
10 31 15 31
15 31 20 31
10 41 10 36
10 36 10 31
15 41 17 34
17 34 17 36
15 36 15 31
20 41 20 31
27
10 22 19 22
19 22 23 22
23 22 28 22
28 22 37 22
10 16 16 16
17 16 23 16
23 16 24 16
24 15 28 15
28 15 31 15
10 10 17 10
17 10 24 10
24 10 31 10
31 10 37 10
10 22 10 16
10 16 10 10
17 18 17 16
17 16 17 10
24 16 24 15
24 15 24 10
23 22 23 16
28 22 28 15
31 15 31 10
37 22 37 17
37 17 37 10
16 16 17 18
17 18 19 22
31 15 37 17
0
```

Output for the Sample Input

```
Case 1
Number of lots with perimeter consisting of 4 surveyor’s lines = 1
Number of lots with perimeter consisting of 6 surveyor’s lines = 1
Number of lots with perimeter consisting of 7 surveyor’s lines = 1
Total number of lots = 3

Case 2
Number of lots with perimeter consisting of 4 surveyor’s lines = 1
Number of lots with perimeter consisting of 5 surveyor’s lines = 4
Number of lots with perimeter consisting of 6 surveyor’s lines = 3
Total number of lots = 8
```

Problem C
Kissin' Cousins

The Oxford English Dictionary defines *cousin* as follows:

> **cous'in** (k▪ zn), n. (Also *first cousin*) child of one's uncle or aunt; *my second (third…) cousin*, my parent's first (second…) cousin's child; *my first cousin once (twice…) removed*, my first cousin's child (grandchild…), also my parent's (grandparent's…) first cousin.

Put more precisely, any two persons whose closest common ancestor is (m+1) generations away from one person and (m+1)+n generations away from the other are mth cousins $n$ce removed. Normally, m • 1 and n • 0, but being used to computers counting from 0, in this problem we require m • 0 and n • 0. This extends the normal definition so that siblings are zeroth cousins. We write such a relationship as `cousin-`*m*`-`*n*.

If one of the persons is an ancestor of the other, p generations away where p • 1, they have a relationship `descendant-`*p*.

A relationship `cousin-`m_1`-`n_1 is *closer* than a relationship `cousin-`m_2`-`n_2 if $m_1 < m_2$ or ($m_1 = m_2$ and $n_1 < n_2$). A relationship `descendant-`p_1 is *closer* than a relationship `descendant-`p_2 if $p_1 < p_2$. A `descendant-`*p* relationship is always closer than a `cousin-`*m*`-`*n* relationship.

Write a program that accepts definitions of simple relationships between individuals and displays the closest `cousin` or `descendant` relationship, if any, which exists between arbitrary pairs of individuals.

Input

Each line in the input begins with one of the characters '`#`', '`R`', '`F`' or '`E`'.

'`#`' lines are comments. Ignore them.

'`R`' lines direct your program to record a relationship between two different individuals. The first 5 characters following the '`R`' constitute the name of the first person; the next 5 characters constitute the name of the second. Case is significant. Following the names, possibly separated from them by blanks, is a non-negative integer, *k*, defining the relationship. If *k* is 0, then the named individuals are siblings. If *k* is 1, then the first named person is a child of the second. If *k* is 2, then the first named person is a grandchild of the second, and so forth. Ignore anything on the line following the integer.

'`F`' lines are queries; your program is to find the closest relationship, if any, which exists between the two different persons whose 5 character names follow the '`F`'. Ignore anything on the line following the second name. A query should be answered only with regard to '`R`' lines which precede the query in the input.

There will be one '`E`' line to mark the end of the input data. Ignore anything on or after the '`E`' line.

Output

For each '`F`' line, your program is to report the closest relationship that exists between the two persons named *aaaaa* and *bbbbb* in one of the following formats:

```
   aaaaa and bbbbb are descendant-p.
   aaaaa and bbbbb are cousin-m-n.
```

with *m*, *n* and *p* replaced by integers calculated as defined above. If no relationship exists between the pair, your program is to output the following:

```
   aaaaa and bbbbb are not related.
```

Assumption:
A person is not an ancestor of himself/herself.

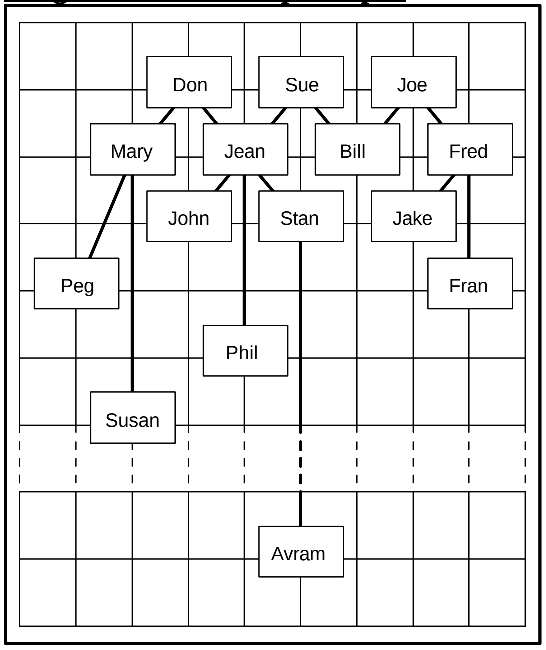
Sample Input

```
# A Comment!
RFred Joe  1 Fred is Joe’s son
RFran Fred 2
RJake Fred 1
RBill Joe  1
RBill Sue  1
RJean Sue  1
RJean Don  1
RPhil Jean 3
RStan Jean 1
RJohn Jean 1
RMary Don  1
RSusanMary 4
RPeg  Mary 2
FFred Joe
FJean Jake
FPhil Bill
FPhil Susan
FJake Bill
FDon  Sue
FStan John
FPeg  John
FJean Susan
FFran Peg
FJohn Avram
RAvramStan  99
FJohn Avram
FAvramPhil
E
```

Output for the Sample Input

```
Fred  and Joe   are descendant-1.
Jean  and Jake  are not related.
Phil  and Bill  are cousin-0-3.
Phil  and Susan are cousin-3-1.
Jake  and Bill  are cousin-0-1.
Don   and Sue   are not related.
Stan  and John  are cousin-0-0.
Peg   and John  are cousin-1-1.
Jean  and Susan are cousin-0-4.
Fran  and Peg   are not related.
John  and Avram are not related.
John  and Avram are cousin-0-99.
Avram and Phil  are cousin-2-97.
```

Diagram of the Sample Input



Problem D
Golygons

Imagine a country whose cities have all their streets laid out in a regular grid. Now suppose that a tourist with an obsession for geometry is planning expeditions to several such cities. Starting each expedition from the central cross-roads of a city, the intersection labelled (0,0), our mathematical visitor wants to set off north, south, east or west, travel one block, and view the sights at the intersection (0,1) after going north, (0,-1) after going south, (1,0) after going east or (-1,0) after going west. Feeling ever more enthused by the regularity of the city, our mathematician would like to walk a longer segment before stopping next, going two blocks. What's more, our visitor doesn't want to carry on in the same direction as before, nor wishes to double back, so will make a 90° turn either left or right. The next segment should be three blocks, again followed by a right-angle turn, then four, five, and so on with ever-increasing lengths until finally, at the end of the day, our weary traveller returns to the starting point, (0,0).

The possibly self-intersecting figure described by these geometrical travels is called a golygon.

Unfortunately, our traveller will making these visits in the height of summer when road works will disrupt the stark regularity of the cities' grids. At some intersections there will be impassable obstructions. Luckily, however, the country's limited budget means there will never be more than 50 road works blocking the streets of any particular city. In an attempt to gain accountability to its citizens, the city publishes the plans of road works in advance. Our mathematician has obtained a copy of these plans and will ensure that no golygonal trips get mired in molten tar.

Write a program that constructs all possible golygons for a city.

Input
Since our tourist wants to visit several cities, the input file will begin with a line containing an integer specifying the number of cities to be visited.

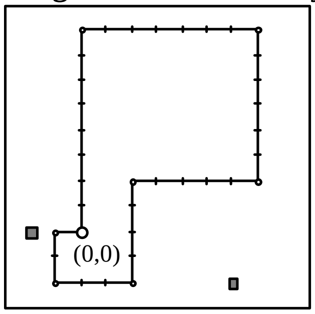
For each city there will follow a line containing a positive integer not greater than 20 indicating the length of the longest edge of the golygon. That will be the length of the last edge which returns the traveler to (0,0). Following this on a new line will be an integer from 0 to 50 inclusive which indicates how many intersections are blocked. Then there will be this many pairs of integers, one pair per line, each pair indicating the *x* and *y* coordinates of one blockage.

Output
For each city in the input, construct all possible golygons. Each golygon must be represented by a sequence of characters from the set {`n`,`s`,`e`,`w`} on a line of its own. Following the list of golygons should be a line indicating how many solutions were found. This line should be formatted as shown in the example output. A blank line should appear following the output for each city. Sample input and output are below.

Sample Input
```
2
8
2
-2 0
6 -2
8
2
2 1
-2 0
```

Output for the Sample Input
```
wsenenws
Found 1 golygon(s).

Found 0 golygon(s).
```

Diagram of the 1st City



Problem E
MIDI Preprocessing

MIDI (Musical Instrument Digital Interface) is a standard for communication involving computers and synthesized music instruments. Part of the standard defines commands, which when transmitted to a synthesizer, begin and end the sounding of a particular note. In this problem we will consider processing simple MIDI "programs." In the following example, three simultaneous notes (a chord, with note numbers 60, 70 and 80) are played for 10 time units immediately followed by a single note (number 62) for 2 time units.

```
0 ON 60
0 ON 70
0 ON 80
10 OFF 60
10 OFF 80
10 OFF 70
10 ON 62
12 OFF 62
```

Much existing music cannot be directly translated to this program form. Sometimes a note is already "on" when the written music indicates that it is to be sounded again. For example:

```
0 ON 60
10 ON 60
12 OFF 60
20 OFF 60
```

A synthesizer will interpret this program to sound note 60 for 12 time units, not 20 as indicated. We will not hear the separate sounding of the note at time 10, since turning on a note that is already sounding will be ignored. By analogy, consider turning a light on and off. If it's on, turning it on again is ineffective. Likewise, the first time that a light is turned off, it is off!

When a note already on is to be sounded again, the program can be "fixed" by inserting an OFF command for that note 1 time unit before the second ON command. Since there are already at least two OFF commands in such circumstances, only the last of these should be retained; the other should be eliminated from the program. The "fixed" program will cause the synthesizer to behave as if the same note had been played twice in rapid succession.

Another problem exists in programs that turn a note on and off at the same time. Depending on the ordering of the events in the program, either the note will be prematurely ended (if the OFF command appears after the ON), or the second sounding of the note will not be heard. For example:

```
0 ON 60              0 ON 60
10 ON 60             10 OFF 60
10 OFF 60            10 ON 60
20 OFF 60            20 OFF 60
```

In the example on the left, the note will be turned off at time 10. The example on the right doesn't leave the note off long enough to allow a human listener to detect the "punctuation" in the sound. In both cases the correction is the same: move the OFF command so it is executed by the synthesizer 1 time unit before the corresponding ON command.

If an OFF command inserted 1 time unit before an ON as a result of the "fix" occurs at exactly the same time as the preceding ON, the second ON and the OFF that occurs at the same time should be eliminated.

Write a program that will accept an arbitrary number of MIDI programs and "fix" them as described above.

Input

Each program contains an arbitrary number of lines. Each line contains, in order, the time that the command is sent to the synthesizer (a non-negative integer), a command (either ON or OFF), and a note (an integer in the range 1 to 127). These items are separated by one or more blanks. Each program except the last is terminated with a line containing only the integer -1. The last program is terminated by a line containing only the integer -2.

Output

The output is to be a "fixed" MIDI program in the same format as the input.

Assumptions

1. *The ON and OFF commands will always be in upper case letters.*
2. *The times associated with programs are in non-decreasing order.*
3. *All notes are initially OFF.*
4. *If different notes are to be turned on or off simultaneously, the order in which the corresponding commands appear is unimportant.*
5. *Each ON command will have a matching OFF command following it in the program.*

Sample Input

```
0 ON 60
10 ON 60
12 OFF 60
20 OFF 60
-1
0 ON 60
5 ON 70
10 ON 60
10 OFF 60
15 OFF 70
15 ON 70
20 OFF 60
20 OFF 70
-1
0 ON 60
1 OFF 60
1 ON 60
10 OFF 60
-2
```

Output for the Sample Input

```
0 ON 60
9 OFF 60
10 ON 60
20 OFF 60
-1
0 ON 60
5 ON 70
9 OFF 60
10 ON 60
14 OFF 70
15 ON 70
20 OFF 60
20 OFF 70
-1
0 ON 60
10 OFF 60
-2
```

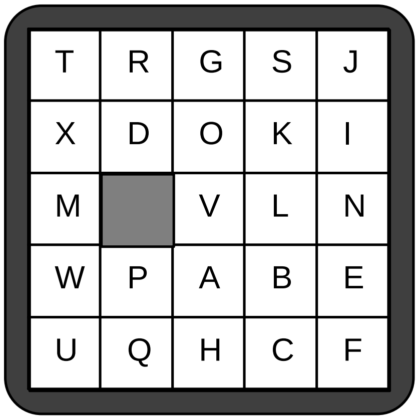
1993 ACM International Collegiate Programming Contest Finals
sponsored by AT&T EasyLink Services

Problem F
Puzzle

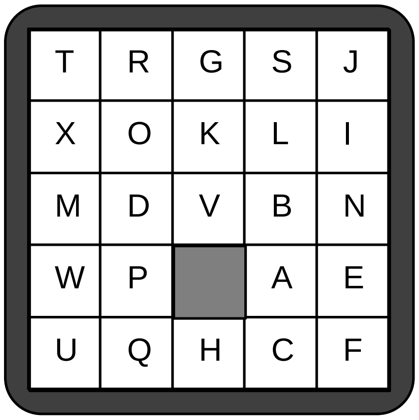
A children’s puzzle that was popular 30 years ago consisted of a 5∞5 frame which contained 24 small squares of equal size. A unique letter of the alphabet was printed on each small square. Since there were only 24 squares within the frame, the frame also contained an empty position which was the same size as a small square. A square could be moved into that empty position if it were immediately to the right, to the left, above, or below the empty position. The object of the puzzle was to slide squares into the empty position so that the frame displayed the letters in alphabetical order.

The illustration below represents a puzzle in its original configuration and in its configuration after the following sequence of 6 moves:

1) The square above the empty position moves.
2) The square to the right of the empty position moves.
3) The square to the right of the empty position moves.
4) The square below the empty position moves.
5) The square below the empty position moves.
6) The square to the left of the empty position moves.



Original puzzle configuration.



Puzzle configuration after the sequence of described moves.

Write a program to display resulting frames given their initial configurations and sequences of moves.

Input
Input for your program consists of several puzzles. Each is described by its initial configuration and the sequence of moves on the puzzle. The first 5 lines of each puzzle description are the starting configuration. Subsequent lines give the sequence of moves.

The first line of the frame display corresponds to the top line of squares in the puzzle. The other lines follow in order. The empty position in a frame is indicated by a blank. Each display line contains exactly 5 characters, beginning with the character on the leftmost square (or a blank if the leftmost square is actually the empty frame position). The display lines will correspond to a legitimate puzzle.

The sequence of moves is represented by a sequence of *A*s, *B*s, *R*s, and *L*s to denote which square moves into the empty position. *A* denotes that the square above the empty position moves; *B* denotes that the square below the empty position moves; *L* denotes that the square to the left of the empty position moves; *R* denotes that the square to the right of the empty position moves. It is possible that there is an illegal move, even when it is represented by one of the 4 move characters. If an illegal move occurs, the puzzle is considered to have no final configuration. This sequence of moves may be spread over several lines, but it always ends in the digit 0. The end of data is denoted by the character Z.

Output

Output for each puzzle begins with an appropriately labeled number (Puzzle #1, Puzzle #2, etc.). If the puzzle has no final configuration, then a message to that effect should follow. Otherwise that final configuration should be displayed.

Format each line for a final configuration so that there is a single blank character between two adjacent letters. Treat the empty square the same as a letter. For example, if the blank is an interior position, then it will appear as a sequence of 3 blanks—one to separate it from the square to the left, one for the empty position itself, and one to separate it from the square to the right.

Separate output from different puzzle records by at least one blank line.

Sample input and the corresponding correct output are shown below. The first record corresponds to the puzzle illustrated on the other side of this page.

<u>Sample Input</u>

```
TRGSJ
XDOKI
M VLN
WPABE
UQHCF
ARRBBL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAA
LLLL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAAAABBRRRLL0
Z
```

<u>Output for the Sample Input</u>

```
Puzzle #1:
  T R G S J
  X O K L I
  M D V B N
  W P   A E
  U Q H C F

Puzzle #2:
     A B C D
   F G H I E
   K L M N J
   P Q R S O
   T U V W X

Puzzle #3:
   This puzzle has no final configuration.
```

Problem G
Resource Allocation

A software development firm is willing to hire new programmers and to spend more money for hardware and software systems in order to increase productivity in its programming divisions. For lack of a better idea, management has defined increased productivity for a division as “incremental lines of code” that the division produces. The company needs a resource allocation model to determine how the money and new programmers should be divided among the divisions in order to maximize the total productivity increase.

Each programming division is limited in how effectively it can utilize any new resources. For example, one particular division will be able to use 0, 3, 5, or 6 new programmers effectively. (The personnel organization within that division prevents it from being able to use 1, 2, 4, 7 or more new programmers.) This gives 4 options for allocating new programmers to that division. There are only 3 different options for allocation of additional money to that division. Therefore, there are 12 possible allocation scenarios in this example. For each scenario, the company has estimated the incremental lines of code that would be produced by that division.

You must write a program that recommends a precise allocation of resources among the divisions. For each division, your program must determine how many new programmers and how much money should be allocated. Allocation of new programmers and money must be made to maximize the total productivity increase—the sum of incremental lines of code over all divisions. The total number of programmers allocated cannot exceed the total number of programmers that the company is willing to hire. The total amount of money cannot exceed the total amount budgeted for the entire company. In the case where there are multiple optimal solutions, your program may recommend any one of them.

Input

Input for your program consists of several allocation problems. All input data are non-negative integers. The first 3 lines of input for each problem consists of:

d	number of programming divisions (0 < d • 20 except when d is the end-of-file sentinel)
p	total number of new programmers
b	total amount of money budgeted for new computing resources

Following those 3 lines are input records for each programming division. The first record is for division #1, the second for division #2, etc. Each division record is organized as follows:

n	number of new programmer options (0 • n • 10)
$x_1\ x_2 \ldots x_n$	list of new programmer options (numbers are separated by blanks)
k	number of new budget options (0 • k • 10)
$b_1\ b_2 \ldots b_k$	cost of each new budget option (separated by blanks)
n ∞ k table of integers	the (i, j) table entry is the incremental lines of code produced for allocation of x_i new programmers and b_j additional budget

It is possible to allocate 0 new programmers to any division and $0 for new hardware and software—resulting in no increase in productivity for that division. This “null” allocation will be explicitly shown.

Each allocation problem begins on a new line. The end of input is signified by an allocation “problem” with 0 divisions. No input lines follow that line.

Output

Output for each problem begins with a line identifying the problem that is solved (problem #1, problem #2, etc.). This is followed by a blank line then 3 lines that tell the total amount of money to be spent, the total number of new programmer to be hired, and the total anticipated new productivity for an optimal resource allocation.

Output for each division comes next. The first line identifies the division by number. The remaining 3 lines indicate the division's budget, the number of new programmers for the division, and the expected incremental lines of code to be produced. One blank line appears between output for successive divisions. Two blank lines appear between output for successive problems. The exact formatting of the output is not critical, but all output must be easy to read and well-identified.

A sample input file which contains one complete allocation problem is shown below. In this problem, there are 3 programming divisions. The company is willing to hire up to 10 new programmers and spend up to $90,000 on new computing resources. For division #1, the expenditure of $50,000 on new computing resources and allocation of 6 new programmers would result in the production of 40,000 incremental lines of code.

<u>Sample Input</u>

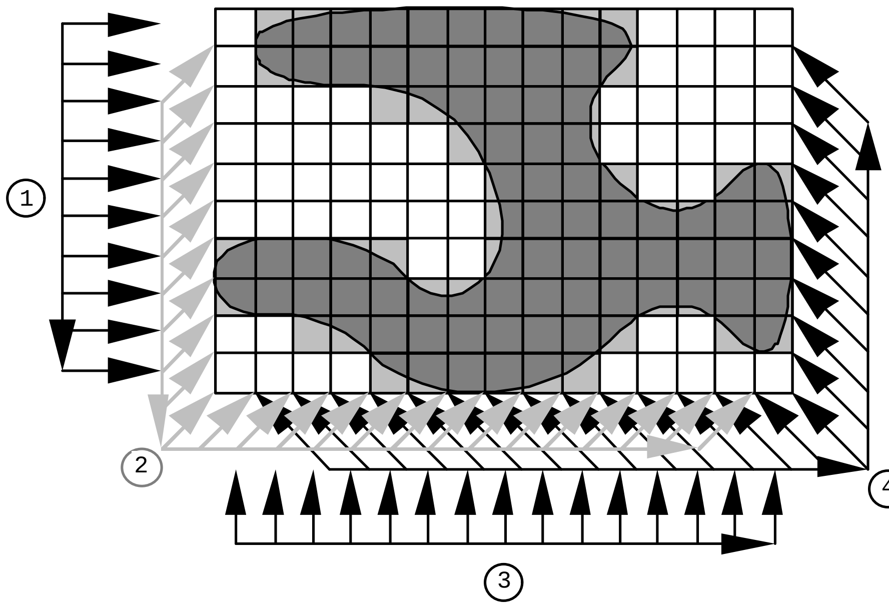
```
3
10
90000
4
0 2 5 6
4
0 20000 50000 70000
    0   10000    20000    50000
60000   20000    10000    40000
20000   10000    30000    40000
30000   10000    40000    30000
5
0 1 3 4 8
3
0 40000 80000
    0   50000   30000
50000   40000   60000
20000   30000   50000
80000   90000   50000
30000   40000   70000
3
0  4  6
5
0  50000  30000  40000  50000
    0  30000  50000  60000  30000
10000  20000  30000  40000  50000
20000  30000  40000  50000  60000
0
```

<u>Output for the Sample Input</u>

```
Optimal resource allocation problem #1

Total budget: $80000
Total new programmers: 6
Total productivity increase: 210000

Division #1 resource allocation:
Budget:  $0
Programmers: 2
Incremental lines of code: 60000

Division #2 resource allocation:
Budget:  $40000
Programmers: 4
Incremental lines of code: 90000

Division #3 resource allocation:
Budget:  $40000
Programmers: 0
Incremental lines of code: 60000
```

Problem H
Scanner

A body scanner works by scanning a succession of horizontal slices through the body; the slices are imaged one at a time. The image slices can be reassembled to form a three dimensional model of the object. Write a program to construct a two dimensional image slice using data captured during the scan.



The scanner consists of four arrays of sensors arranged around a 10∞15 matrix. Array 1 consists of 10 sensors pointing to the right, array 2 has 24 sensors pointing diagonally to the top right, array 3 has 15 sensors pointing to the top and array 4 has 24 sensors pointing to the top left. Each sensor records the thickness of that portion of the object directly in front of that sensor.

Readings from the arrays of sensors are recorded in counterclockwise order. Within an array of sensors, data are also recorded counterclockwise. A complete scan consists of 73 readings.

Input

The input file begins with a line with an integer indicating the number of image slices to follow. For each image slice, there are separate lines with 10, 24, 15, and 24 integers representing sensor data from sensor arrays 1 through 4 respectively. The order of the readings is indicated in the diagram. Although it is possible for the result of a scan to be ambiguous, the data supplied to you will have no ambiguous interpretation.

Output

For each slice, your program should print 10 lines of 15 cells. To indicate that the cell represents a part of the object, print a hash character (#) for the cell; to indicate that the cell is not a part of the object, print a period (`.`). Between successive output image slices, print a blank line.

Sample Input (describing object in diagram above)

```
1
10 10 6 4 6 8 13 15 11 6
0 1 2 2 2 2 4 5 5 6 7 6 5 6 6 5 5 6 6 3 2 2 1 0
2 4 5 5 7 6 7 10 10 10 7 3 3 5 5
0 0 1 3 4 4 4 4 3 4 5 7 8 8 9 9 6 4 4 2 0 0 0 0
```

Output for the Sample Input

```
.##########....
.##########....
....######.....
......####.....
.......####..##
.......########
#####..########
###############
..#########..##
....######.....
```