A • Repeating Characters

For this problem, you will write a program that takes a string of characters, ***S***, and creates a new string of characters, ***T***, with each character repeated ***R*** times. That is, ***R*** copies of the first character of ***S***, followed by ***R*** copies of the second character of ***S***, and so on. Valid characters for ***S*** are the *QR Code* "*alphanumeric*" characters:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ$%*+-./:
```

Input

The first line of input contains a single integer ***P***, ($1 \le P \le 1000$), which is the number of data sets that follow. Each data set is a single line of input consisting of the data set number ***N***, followed by a space, followed by the repeat count ***R***, ($1 <= R <= 8$), followed by a space, followed by the string ***S***. The length of string ***S*** will always be at least one and no more than 20 characters. All the characters will be from the set of characters shown above.

Output

For each data set there is one line of output. It contains the data set number, ***N***, followed by a single space which is then followed by the new string ***T***, which is made of each character in ***S*** repeated ***R*** times.

Sample Input	Sample Output
2 1 3 ABC 2 5 /HTP	1 AAABBBCCC 2 /////HHHHHTTTTTPPPPP

B • The Rascal Triangle

The *Rascal Triangle* definition is similar to that of the *Pascal Triangle*. The rows are numbered from the top starting with 0. Each row `n` contains `n+1` numbers indexed from `0` to `n`. Using `R(n,m)` to indicate the index `m` item in the index `n` row:

`R(n,m) = 0` for `n < 0` **OR** `m < 0` **OR** `m > n`

The first and last numbers in each row (which are the same in the top row) are 1:

`R(n,0) = R(n,n) = 1`

The interior values are determined by (*UpLeftEntry* * *UpRightEntry* + 1)/*UpEntry* (see the parallelogram in the array below):

`R(n+1,m+1) = (R(n,m) * R(n,m+1) + 1)/R(n-1,m)`

```
                1
              1   1
            1   2   1
              / \
          1   3   3   1
                \ /
        1   4   5   4   1
```

Write a program which computes `R(n,m)` the m^{th} element of the n^{th} row of the *Rascal Triangle*.

Input

The first line of input contains a single integer ***P***, (`1 ≤ P ≤ 1000`), which is the number of data sets that follow. Each data set is a single line of input consisting of 3 space separated decimal integers. The first integer is data set number, ***N***. The second integer is row number ***n***, and the third integer is the index ***m*** within the row of the entry for which you are to find `R(n,m)` the *Rascal Triangle* entry `(0 <= m <= n <= 50000)`.

Output

For each data set there is one line of output. It contains the data set number, ***N***, followed by a single space which is then followed by the *Rascal Triangle* entry `R(n,m)` accurate to the nearest integer value.

Sample Input	Sample Output
5 1 4 0 2 4 2 3 45678 12345 4 12345 9876 5 34567 11398	1 1 2 5 3 411495886 4 24383845 5 264080263

C • Programming the EDSAC

The world's first full-scale, stored-program, electronic, digital computer was the EDSAC (Electronic Delay Storage Automatic Calculator). The EDSAC had an accumulator-based instruction set, operating on 17-bit words (and 35-bit double words), and used a 5-bit teletypewriter code for input and output.

The EDSAC was programmed using a very simple assembly language: a single letter opcode followed by an unsigned decimal address, followed by the the letter 'F' (for *full word*) or 'D' (for *double word*). For example, the instruction "**`A 128 F`**" would mean "*add the full word at location 128 to the accumulator*", and would be assembled into the 17-bit binary value, **`11100000100000000`**, consisting of a 5-bit opcode (**`11100`** = "*add*"), an 11-bit operand (**`00010000000`** = 128), and a single **`0`** bit denoting a *full word* operation (a 1 bit would indicate a *double word* operation).

Although arithmetic on the EDSAC was *fixed point two's complement binary,* it was not mere integer arithmetic (as is common with modern machines). The EDSAC hardware assumed a *binary point* between the leftmost bit and its immediate successor. Thus the hardware could handle only values in the range `-1.0 <= x < 1.0`. For example:

Value	Binary Representation
-1.0	1000000000000000
½	0100000000000000
¾	0110000000000000
-½	1100000000000000

As you can see, the largest possible positive value was:

`0111111111111111 = 0.9999847412109375`

and the smallest possible positive value was:

`0000000000000001 = 2`$^{-16}$ `= 0.0000152587890625`

(This also happens to be the increment between successive values on the EDSAC).

By a curious coincidence (or an elegant design decision), the opcode for the *add* operation (`11100`) was the same as the teleprinter code for the letter '**A**'. The opcode for *subtract* was the same as the teleprinter code for '**S**' (`01100`), and so on. This simplified the programming for the assembler (which, incidentally, was a mere 31 instructions long). The EDSAC teleprinter alphabet was "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV" (with 'P' = `00000`, 'Q' = `00001`, and so on, up to 'V' = `11111`).

Unfortunately, the EDSAC assembler had no special directives for data values. On the other hand, there was no reason that ordinary instructions couldn't be used for this, thus, an EDSAC programmer desiring to reserve space for the constant ¾ (represented as `0110000000000000`) would use the instruction "`S 0 F`" and for $^1/_3$ (which is approximately represented as `0010101010101010`) "`T 682 D`", and so on.

Your job is to write a program that will translate decimal input values into the appropriate EDSAC instructions.

Input

The first line of input contains a single integer ***P***, ($1 \le P \le 1000$), which is the number of data sets that follow. Each data set is a single line that consists of two space separated values ***N*** and ***D***. ***N*** is the data set number. ***D*** is the decimal number of the form ***sd.ddd***…., where ***s*** is an optional minus sign, and ***d*** is any decimal digit (`0-9`). There will be at least 1 and at most 16 digits after the decimal point.

Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, followed by the EDSAC instruction necessary to specify the given constant. The instruction should be printed as follows: the "opcode" character followed by a space followed by the operand (as a non-negative decimal integer) followed by a space followed by an 'F' or 'D' (as appropriate). If the constant cannot be represented exactly in 17 bits, the value is to be rounded toward zero (up for negative, down for positive numbers). If the input value ***D*** is not in the range `-1.0 <= D < 1.0`, the string "`INVALID VALUE`" should be printed instead of an EDSAC instruction.

Sample Input	Sample Output
16 1 0.5 2 -0.5 3 -1.0000000 4 0.1 5 0.0000152587890625 6 0.0000152587890624 7 0.0000152587890626 8 -0.0000152587890625 9 -0.0000152587890624 10 -0.0000152587890626 11 0.9999999999999999 12 -0.9999999999999999 13 -5.3 14 9.1 15 -1.0000000000000001 16 0.31415926	1 I 0 F 2 & 0 F 3 ? 0 F 4 Q 1228 D 5 P 0 D 6 P 0 F 7 P 0 D 8 V 2047 D 9 P 0 F 10 V 2047 D 11 * 2047 D 12 ? 0 D 13 INVALID VALUE 14 INVALID VALUE 15 INVALID VALUE 16 T 54 F

D • Decoding EDSAC Data

The world's first full-scale, stored-program, electronic, digital computer was the EDSAC (Electronic Delay Storage Automatic Calculator). The EDSAC had an accumulator-based instruction set, operating on 17-bit words (and 35-bit double words), and used a 5-bit teletypewriter code for input and output.

The EDSAC was programmed using a very simple assembly language: a single letter opcode followed by an unsigned decimal address, followed by the the letter 'F' (for *full word*) or 'D' (for *double word*). For example, the instruction "**`A 128 F`**" would mean "*add the full word at location 128 to the accumulator*", and would be assembled into the 17-bit binary value, **`11100000100000000`**, consisting of a 5-bit opcode (**`11100`** = "*add*"), an 11-bit operand (**`00010000000`** = 128), and a single **`0`** bit denoting a *full word* operation (a 1 bit would indicate a *double word* operation).

Although arithmetic on the EDSAC was *fixed point two's complement binary,* it was not mere integer arithmetic (as is common with modern machines). The EDSAC hardware assumed a *binary point* between the leftmost bit and its immediate successor. Thus the hardware could handle only values in the range `-1.0 <= x < 1.0`. For example:

Value	Binary Representation
`-1.0`	`1000000000000000`
½	`0100000000000000`
¾	`0110000000000000`
-½	`1100000000000000`

As you can see, the largest possible positive value was:

`0111111111111111 = 0.9999847412109375`

and the smallest possible positive value was:

`0000000000000001 = 2`$^{-16}$ `= 0.0000152587890625`

(This also happens to be the increment between successive values on the EDSAC).

By a curious coincidence (or an elegant design decision), the opcode for the *add* operation (`11100`) was the same as the teleprinter code for the letter '**A**'. The opcode for *subtract* was the same as the teleprinter code for '**S**' (`01100`), and so on. This simplified the programming for the assembler (which, incidentally, was a mere 31 instructions long). The EDSAC teleprinter alphabet was "PQWERTYUIOJ#SZK*?F@D!HNM&LXGABCV" (with 'P' = `00000`, 'Q' = `00001`, and so on, up to 'V' = `11111`).

Unfortunately, the EDSAC assembler had no special directives for data values. On the other hand,

there was no reason that ordinary instructions couldn't be used for this, thus, an EDSAC programmer desiring to reserve space for the constant ¾ (represented as `0110000000000000`) would use the instruction "`S 0 F`" and for $^1/_3$ (which is approximately represented as `0010101010101010`) "`T 682 D`", and so on.

Your job is to write a program that will translate EDSAC instructions into the appropriate decimal fractions.

Input

The first line of input contains a single integer ***P***, ($1 \le P \le 1000$), which is the number of data sets that follow. Each data set is a single line that contains ***N*** (the dataset number), followed by a space, followed by an EDSAC instruction of the form: *c*␣*d*␣*s*, where *c* is a single character in the EDSAC alphabet, *d* is an unsigned decimal number ($0 <= d < 2^{11}$), and s is either a '`D`' or '`F`'. Note:␣ represents a single space.

Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, followed by the exact decimal fraction represented by the by the EDSAC instruction, including a minus sign (for negative values). The format for the decimal fraction is: **`sb.ddd`**…, where **`s`** is an optional minus sign, **`b`** is either a `1` or `0`, and **`d`** is any decimal digit (0-9). There must be at least 1 and at most 16 digits after the decimal point. Trailing zeros in the fraction *must* be suppressed.

Sample Input	Sample Output
13 1 P 0 F 2 I 0 F 3 & 0 F 4 ? 0 F 5 Q 1228 D 6 P 0 D 7 V 2047 D 8 * 2047 D 9 ? 0 D 10 P 256 F 11 V 1536 F 12 T 682 D 13 T 54 F	1 0.0 2 0.5 3 -0.5 4 -1.0 5 0.0999908447265625 6 0.0000152587890625 7 -0.0000152587890625 8 0.9999847412109375 9 -0.9999847412109375 10 0.0078125 11 -0.015625 12 0.3333282470703125 13 0.31414794921875

E • Route Redundancy

A city is made up exclusively of one-way streets. Each street in the city has a capacity, the maximum number of cars it can carry per hour. Any route (path) also has a capacity, which is the minimum of the capacities of the streets along that route.

The *redundancy ratio* from point ***A*** to point ***B*** is the ratio of the maximum number of cars that can get from ***A*** to ***B*** in an hour using all routes simultaneously, to the maximum number of cars that can get from ***A*** to ***B*** in an hour using just one route. The minimum redundancy ratio is the number of cars that can get from ***A*** to ***B*** in an hour using all possible routes simultaneously, divided by the capacity of the single route with the largest capacity.

Input

The first line of input contains a single integer ***P***, ($1 \leq P \leq 1000$), which is the number of data sets that follow. Each data set consists of several lines and represents a directed graph with positive integer weights.

The first line of each data set contains five space separated integers. The first integer, ***D*** is the data set number. The second integer, ***N*** (2 <= ***N*** <= 1000), is the number of nodes in the graph. The third integer, ***E***, (***E*** >= 1), is the number of edges in the graph. The fourth integer, ***A***, (0 <= ***A*** < ***N***), is the index of point **A**. The fifth integer, ***B***, (0 <= ***B*** < ***N*****,** **A** != **B**), is the index of point **B**.

The remaining ***E*** lines describe each edge. Each line contains three space separated integers. The first integer, ***U*** (0 <= ***U*** < N), is the index of node **U**. The second integer, ***V*** (0 <= ***V*** < ***N*****,** V **!=** U), is the index of node **V**. The third integer, ***W*** (1 <= ***W*** < 1000), is the capacity (weight) of the path from **U** to **V**.

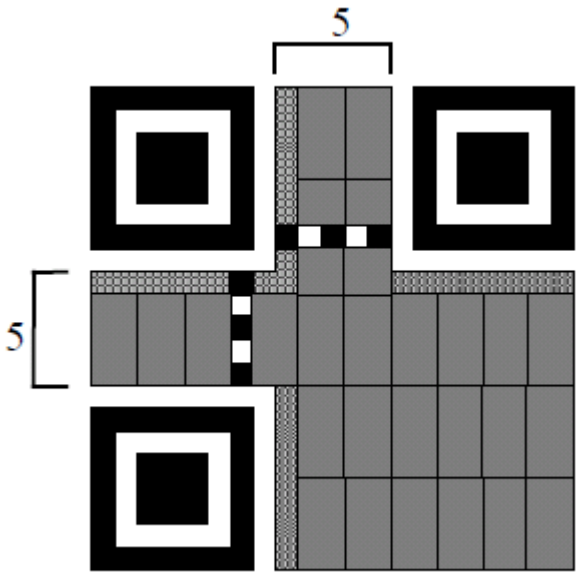
Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, followed by a floating-point value which is the minimum *redundancy ratio* to 3 digits after the decimal point.

Sample Input	Sample Output
1 1 7 11 0 6 0 1 3 0 3 3 1 2 4 2 0 3 2 3 1 2 4 2 3 4 2 3 5 6 4 1 1 4 6 1 5 6 9	1 1.667

F • QR

QR Codes (the smallest, which is 21 pixels by 21 pixels, is shown below) are square arrays of black or white pixels (modules) which include *Position Detection Patterns* (the square bull's-eye patterns), *Timing Patterns* (the alternating black and white lines), *Alignment Patterns* in larger *QR Codes*, *Format Information* (the stippled pixels), *Version information* in larger *QR Codes* and *Data and Error Correction Codewords* (gray 8 pixel blocks).



The 21-by-21 *QR Code* has *26 data and error correction codewords*. At the lowest error correction level for this code, 19 are *data codewords* and 7 are *error correction codewords*. Data may be encoded as numeric at 3 numbers per 10 bits, as alphanumeric at 2 characters per 11 bits, as 8 bit bytes or as Kanji at 13 bits per character. Data is encoded in groups of (mode, character count, character data bits). The mode can change within the data stream. The mode is specified by a 4 bit code and the character count by a varying number of bits depending on the mode and *QR Code* size. For the 21-by-21 code, the character count bits are:

Mode Name	Mode Bits	Count Bits
Numeric	0001	10
Alphanumeric	0010	9
8 bit byte	0100	8
Kanji	1000	8
Termination	0000	0

The entire data stream ends in the *termination code* which may be truncated if there is not enough room. Any partially filled codeword after the termination code is filled with 0 bits. Any remaining codewords are set to `11101100` followed by `00010001` alternating.

Numeric strings are encoded 3 digits at a time. If there are remaining digits, 2 digits are encoded in 7 bits or 1 digit in 4 bits. For example:

`12345678` → `123 456 78` → `0001111011 0111001000 1001110`

Prefix with mode (`0001`) and count (`8` → `0000001000`) is (`4 + 10 + 10 + 10 + 7`) bits:

`0001 0000001000 0001111011 0111001000 1001110`

Alphanumeric strings encode the characters (<SP> represents the space character):

`0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ<SP>$%*+-./:`

as numbers from `0` to `44`, then two characters are encoded in `11` bits:

<first char code> `* 45 +` *<second char code>*

if the number of characters is odd, the last character is encoded in 6 bits. For example:

`AC-42` → `(10, 12, 41, 4, 2)` → `10*45 + 12 = 462, 41*45 + 4 = 1849, 2` →
`00111001110 11100111001 000010`

Prefix with mode and count is $(4 + 9 + 11 + 11 + 6)$ bits:

`0010 000000101 00111001110 11100111001 000010`

The 8 bit binary and Kanji modes will be straightforward for the purposes of this problem. Kanji codes will just be opaque 13 bit codes; you need not decode the characters they represent, just the hexadecimal values. For example:

`8 bit 0x45 0x92 0xa3` → `01000101 10010010 10100011`

Prefix with mode and count is (`4 + 8 + 8 + 8 + 8`) bits:

`0100 00000011 01000101 10010010 10100011`

Kanji `0x1ABC 0x0345` → `1101010111100 0001101000101`

Prefix with mode and count is (`4 + 8 + 13 + 13`) bits:

`1000 00000010 1101010111100 0001101000101`

To illustrate forming the 19 codeword content of a *QR Code*, combine the first 3 sequences above (for numeric, alphanumeric and bytes). Concatenate the bits, split into 8 bit code words add the termination codeword, any fill bits and fill bytes (`41 + 41 + 36` *data bits* + 4 bit *termination code* = `122` → `6` fill bits are needed to get `16` bytes, and to fill out the 19 bytes, 3 fill bytes are needed):

```
0001 0000001000 0001111011 0111001000 1001110
0010 000000101 001110011110 111001111001 000010
0100 00000011 01000101 10010010 10100011
0000 000000 11101100 00010001 11101100
```

split into 8 bit codewords:

```
00010000 00100000 01111011 01110010 00100111 00010000 00010100 11100111
01110011 10010000 10010000 00001101 00010110 01001010 10001100 00000000
11101100 00010001 11101100 → HEX 10207B72271014E77390900D164A8C0EC11EC
```

Write a program to read 19 codewords and print the corresponding data.

Input

The first line of input contains a single integer ***P***, (`1 ≤ P ≤ 1000`), which is the number of data sets that follow. Each data set is a single line of input consisting of the data set number, ***N***, followed by a single space and 38 hexadecimal digits giving the 19 bytes of *QR Code* data. The valid hexadecimal digits are `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E` and `F`.

Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, the number of *QR decoded* characters in the result, a single space and the character string corresponding to the *QR Code* data. In the output string, printable ASCII characters (in the range `0x20` to `0x7e`) are printed as the ASCII character *EXCEPT* that backslash (****) is printed as **** and pound sign (**#**) is printed as **\#**. *Non-printable* 8 bit data is output as **\xx**, where **x** is a hexadecimal digit (e.g. **\AE**). *Non-printable* 8 bit data is any value that is less than the ASCII value of a space (`0x20`) or greater than `0x76`. 13 bit Kanji values are printed as **#bxxx**, where **b** is **0** or **1** and **x** is a hexadecimal digit (e.g. **#13AC**).

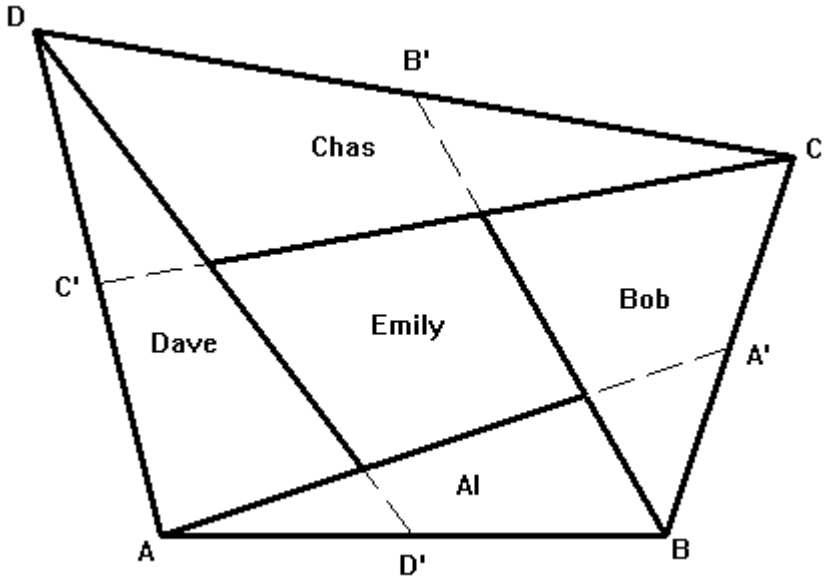
Sample Input
4 1 10207B72271014E77390900D164A8C00EC11EC 2 802D5E0D1400EC11EC11EC11EC11EC11EC11EC 3 20BB1AA65F9FD7DC0ED88C973E15EF533EB0EC 4 2010B110888D9428D937193B9CEA0D7F45DF68

Sample Output
1 16 12345678AC-42E\92\A3 2 2 #1ABC#0345 3 23 HTTP://WWW.ACMGNYR.ORG/ 4 36 3.14159265358979323846264338327950 28

G • Rancher's Gift

Rancher Joel has a tract of land in the shape of a convex quadrilateral that he wants to divide among his sons Al, Bob, Chas and Dave, who wish to continue ranching on their portions, and his daughter Emily, who wishes to grow vegetables on her portion.

The center of the tract is most suitable for vegetable farming so Joel decides to divide the land by drawing lines from each corner (`A`, `B`, `C`, `D` in counter clockwise order) to the center of an opposing side (respectively `A'`, `B'`, `C'` and `D'`). Each son would receive one of the triangular sections and Emily would receive the central quadrilateral section. As shown in the figure, Al's tract is to be bounded by the line from `A` to `B`, the line from `A` to the midpoint of `BC` and the line from `B` to the midpoint of `CD`; Bob's tract is to be bounded by the line from `B` to `C`, the line from `B` to the midpoint of `CD` and the line from `C` to the midpoint of `DA`, and so on.



Your job is to write a program that will help Rancher Joel determine the area of each child's tract and the length of the fence he will have to put around Emily's parcel to keep her brothers' cows out of her crops.

For this problem, `A` will always be at `(0, 0)` and `B` will always be at `(x, 0)`. Coordinates will be in *rods* (a *rod* is 16.5 *feet*). The returned areas should be in *acres* to 3 decimal places (an *acre* is 160 square *rods*) and the length of the fence should be in *feet*, rounded up to the next *foot*.

Input

The first line of input contains a single integer P, ($1 \le P \le 1000$), which is the number of data sets that follow. Each data set is a single line that contains of a decimal integer followed by five (5) space separated floating-point values. The first (integer) value is the data set number, N. The floating-point values are `B.x`, `C.x`, `C.y`, `D.x` and `D.y` in that order (where `V.x` indicates the `x` coordinate of `V` and `V.y` indicates the `y` coordinate of `V`). Recall that the `y` coordinate of `B` is always zero (`0`). The supplied coordinates will always specify a valid convex quadrilateral.

Output

For each data set there is a single line of output. It contains the data set number, N, followed by a single space followed by five (5) space separated floating point values to ***three*** (3) decimal place accuracy, followed by a single space and a decimal integer. The floating-point values are the areas in *acres* of the properties of Al, Bob, Chas, Dave, and Emily respectively. The final integer is the length of fence in *feet* required to fence in Emily's property (rounded up to the next foot).

Sample Input
3 1 200 250 150 -50 200 2 200 200 100 0 100 3 201.5 157.3 115.71 -44.2 115.71

Sample Output
1 35.000 54.136 75.469 54.167 54.666 6382 2 25.000 25.000 25.000 25.000 25.000 4589 3 29.144 29.144 29.144 29.144 29.144 4937

H • Maximum in the Cycle of 1

If ***P*** is a permutation of the integers `1, …, n`, the *maximum in the cycle of 1* is the maximum of the values ***P(1), P(P(1)), P(P(P(1)))***, etc. For example, if ***P*** is the permutation:

```
|1 2 3 4 5 6 7 8|
|3 2 5 4 1 7 8 6|
```

we have:

```
P(1) = 3
P(P(1)) = P(3) = 5
```

and

```
P(P(P(1))) = P(5) = 1
```

so the maximum in the cycle of 1 is 5.

For this problem, you will write a program which takes as input integers `n,` (`n > 0`) and `k` (`1 <= k <= n`), and returns the number of permutations of the integers `1, …, n`, for which the maximum in the cycle of 1 is `k`.

Input

The first line of input contains a single integer ***P***, (`1 ≤` ***P*** `≤ 1000`), which is the number of data sets that follow. Each data set is a single line that contains the three space separated decimal integer values. The first value is the data set number, ***N.*** The second value is the size of the permutation, **`n`** where (**`1 <= n <= 20`**), and the third value is the desired maximum in the cycle of 1, **`k`** where (**`1 <= k <= n`**).

Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, followed by a double precision floating point whole value which is the number of permutations of the integers `1, …, n`, for which the maximum in the cycle of 1 is `k`.

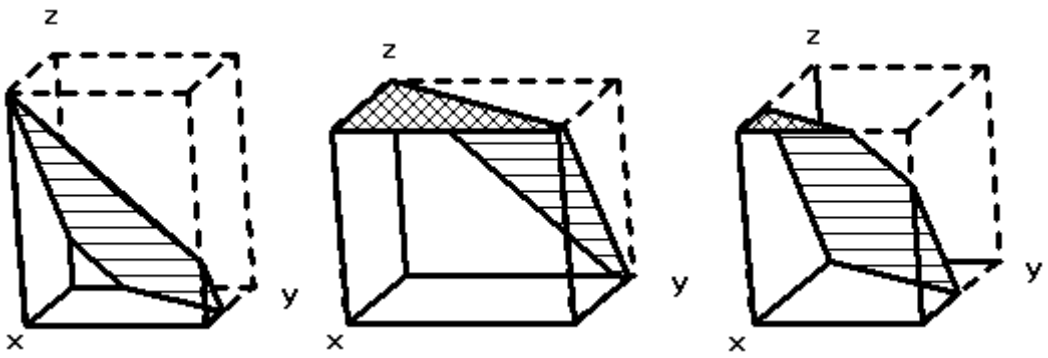
Sample Input	Sample Output
4 1 4 1 2 7 3 3 10 5 4 20 7	1 6 2 168 3 86400 4 1158524765798400

I • The Golden Ceiling

The main office of the *Bank of Zork* was built in the *Aragain Village* (later known as *Flatheadia*) in the year 722 of the Great Underground Empire (*GUE*). In 788 *GUE*, the chairman, J. Pierpont Flathead, decided (shortly before his unexplained disappearance in 789), that it was time to completely redesign the already ornate bank atrium with a ceiling covered in brilliant gold leaf.

This new ceiling was not your ordinary ceiling. Although the atrium is essentially a big box, the ceiling would be slanted (supposedly, making the atrium look bigger). At the time, the exact dimensions of the rectangular atrium and the slope and location of the slanted ceiling had not been finalized. Flathead wanted to know how much gold leaf he would have to order from the *Frobozz Magic Gold Leaf Company* to cover the ceiling for different atrium dimensions and ceiling slants. He also wanted to allow the slanted part to possibly hit the floor of the box and/or the top of the box.

Consider the following rough sketches of some possible atrium configurations:



Note: The dashed outline represents the original box, the horizontally ruled surface is the slanted part of the ceiling and the cross hatched surface is the part of the top of the box not cut off by the plane. The walls and floor of the atrium are transparent. The total area to be covered (the *ceiling*) is the slanted part plus any part of the top of the original box that is not cut off by the plane.

Your job is to write a program that Flathead could have used to calculate the amount gold leaf required to cover the *ceiling* for a particular configuration.

As a sad epilogue, the main branch was brought to ruins when the Curse of Megaboz befell it in 883*GUE*. Between the barbarian invasions of the 880's and the countless looters that had tread the underground ruins in the years that followed, the entire bank with all its valuables, as well as its very expensive gold leaf ceiling, had been removed or vandalized. More information can be found on-line at: http://www.thezorklibrary.com/history/bank_of_zork.html.

(Continued on next page)

Input

The first line of input contains a single integer ***P***, ($1 \le P \le 1000$), which is the number of data sets that follow. Each data set is a single line that contains the data set number, ***N***, followed by a space, followed by seven space separated double precision floating point values, **L**, **W**, **H**, **A**, **B**, **C** and **D**. The values **L**, **W** and **H** specify the *length, width* and *height* of the atrium in *Flathead Units* (*FU*'s), respectively, and are always positive values. The values **A**, **B**, **C** and **D** specify the coefficients of the plane equation for the slanted part of the ceiling:

$$\mathbf{A}x + \mathbf{B}y + \mathbf{C}z = \mathbf{D}$$

where: $0 <= x <= \mathbf{L}$, $0 <= y <= \mathbf{W}$, $0 <= z <= \mathbf{H}$.

One corner of the original box is always at the origin (0, 0, 0) and the other at (**L**, **W**, **H**). The plane will never be vertical (C will be >= 1.0) and the plane will always pass through the interior of the box (there will be points (x,y,z) in the box and strictly above the plane (Ax + BY +Cz > D).and others strictly below the plane (Ax + By + Cz < D).

Output

For each data set there is one line of output. It contains the data set number (***N***) followed by a single space, followed by an integer value that is the number of square *FU*'s required to cover the *ceiling* in gold leaf (rounded *up* to the next square *FU*).

Sample Input	Sample Output
3 1 10 12 15 -1.3 1 1.1 3.5 2 10 12 10 -1.3 1 1.1 11 3 13 9 10 -1.3 1 1.1 0	1 166 2 164 3 144