DELIVERY (FIN)

There is a large amount of packets that need to be delivered, within a certain time limit if possible. There are two vehicles available for making the deliveries, a small van and a large lorry.

Each packet is either *small* or *large*. A small packet can be delivered either with the van or with the lorry, but a big one requires the lorry. For each packet we know the amount of time it would take to deliver it; we call this the *time requirement* of the packet. One vehicle can deliver only one packet at a time.

We are also given a *deadline*, which is the total amount of time within which we should deliver as many packets as possible. We do not care about what happens after the deadline

Thus, from our point of view, the deliveries take place as follows.

- A certain amount of small packets will be delivered by the van, and the total time requirement of these packets must not exceed the deadline, and
- a certain amount of small and large packets will be delivered by the lorry, and the total time requirement of these packets must also not exceed the deadline.

The task in this problem is to determine the largest total amount of packets that can be delivered in this manner.

Input data

The input will be in the file DELIVERY.IN.

The first input line contains a single integer *T* (1 ≤ T ≤ 1000) - the deadline.

The second input line contains a single integer *N* (1≤ N≤500) - the total number of small packets.

The input lines 3 through *N*+2 each contain a single integer between 1 and 1000 (inclusive). These are the time requirements of the small packets. Time requirements are given in *ascending order*.

The input line number *N*+3 contains a single integer *M* (1≤M≤500) - the total number of large packets.

The input lines *N*+4 through *N*+*M*+3 each contain a single integer between 1 and 1000 (inclusive). These are the time requirements of the large packets. These time requirements are also given in ascending order.

Output data

The only line of the output file DELIVERY.OUT should contain a single integer - the largest number of packets that can be delivered within the deadline.

Example

Input data(file DELIVERY.IN)

```
10
8
2
2
2
2
2
4
4
4
4
3
3
6
6
```

Output data(file DELIVERY.OUT)

```
8
```

The deadline is 10 time units. There are 8 small packets: 5 with time requirement 2 and 3 with time requirement 4. There are 4 large packets, 2 with time requirement 3 and 2 with time requirement 6.

The 8 packets can be delivered in 10 time units by delivering the small packets with time requirement 2 with the van, and using the lorry to deliver one small packet with time requirement 4 and 2 large packets with time requirement 3.

NEW FUN (LAT)

Programmers often are asked to rewrite old program and the new program version must keep its functionality. It is nice, if such program is well documented and source code is available. But not always life is so lucky.

You must rewrite earlier written function FUN about which is known the following:

1) FUN's arguments are 4 positive integers a_1,a_2,a_3 and a_4,
2) FUN's value is one of its arguments values,
3) FUN realizes one function of the following kind:

```
IF a_i op a_j THEN FUN:=a_k ELSE FUN:=a_m ;
```

where possible *op* values are $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, and $1 \leq i,j,k,m \leq 4$.

If `i=1, j=2, k=4, m=1`, and *op* is '≠', then function

```
IF a1≠a2 THEN FUN:=a4 ELSE FUN:=a1;          (¶)
```

is realized and we will obtain, for example, `FUN(2,1,1,3) = 3, FUN(4,4,2,3) = 4`.

It is known, that in practice the existing version is too slow and works properly only if its argument values doesn't exceed 5. You are asked to write a new function version which have to work properly for all positive integer values which doesn't exceed 32767.

Old FUN version is written so, that this, exactly which function is realized depends on data file in the current catalog FUN.DAT - so it will not be enough if you will write version for one fixed FUN.DAT. New version must work properly for any valid FUN.DAT .

Your program will always work in the same environment as old FUN, so you, for example, may include in your program old FUN calls (of course, keeping in mind restrictions for argument values). You can asume that in the current catalog always will be valid file FUN.DAT and your new program never will be used without it.

You must write program which realizes the same function as old FUN (for any valid FUN.DAT) and for argument values given in input file counts function values and output them to output file.

Input data

In the first line of text file NEWFUN.IN number of argument sets N ($1 \leq N \leq 1000$) is given. In the each of the following N lines one argument set (four positive integer values separated by single space characters) is given. All argument values are less or equal to 32767.

Output data

Text file NEWFUN.OUT must contain exactly N rows. In the i-th file row one positive integer - "new FUN" value for i-th argument set must be written.

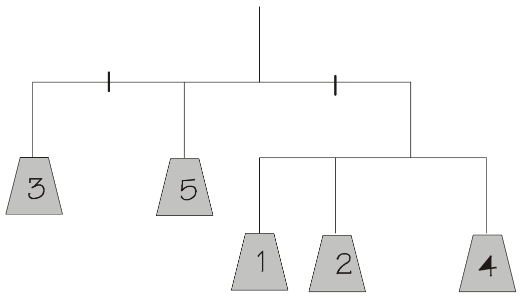
Environment and example

On your diskettes files which realize old FUN (FUNIT.TPU, FUNIT.OBJ, FUNIT.H, FUN.DAT) are given. Given FUN.DAT corresponds to function described earlier in example (¶). In the files MAIN.PAS, MAIN.C and DEMO.PRJ old FUN version call examples are given.

```
Input data (File NEWFUN.IN)   Output data(File NEWFUN.OUT)
5                             3
2 1 1 3                       4
4 4 2 3                       32766
32767 1 1 32766               6
6 6 6 6                       10
7 8 9 10
```

Remark: In the given example the last three sets will not be processed properly by old `FUN` version.

THE MOBILE (SWE)



In figure 1 you can see a mobile in perfect balance. The mobile have five weights 1 kg, 2 kg ... 5 kg. The distance between two marks is 1 m. You can check the balance through this calculation

```
-3·3 + -1·5 + 2·(1+2+4)=0
-2·1 + -1·2 + 1·4=0
```

When you get a mobile problem you will get the ***stucture*** of the mobile from a character string. The mobile in figure 1 is described in this way

```
(-3,-1,2(-2,-1,1))
```

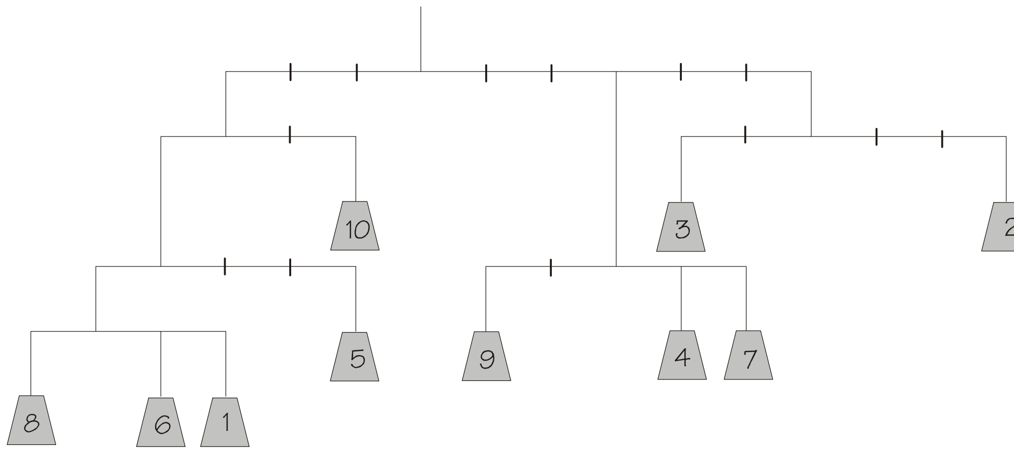
You have to calculate the weights and give the answer as another character string. From figure 1 with the five weights the answer is

```
(3,5,(1,2,4))
```

You will now write a program reading a description of a mobile from an input file, calculate the weights and writing the answer to an output file.

- If there are *n* weights in the mobile you must use the weights from 1 kg to *n* kg exactly once
- *n* ≤17
- The input string will be given as a single line in the input file MOBILE.IN. All numbers are integers which values are between -50 and 50 (inclusive).
- The output string will be given as a single line in the output file MOBILE.OUT, without extra spaces.
- You can assume that for each given test data at least one solution exists. If more than one solution is possible, you must output one of them.

Here you will have two additional examples with input and output string.

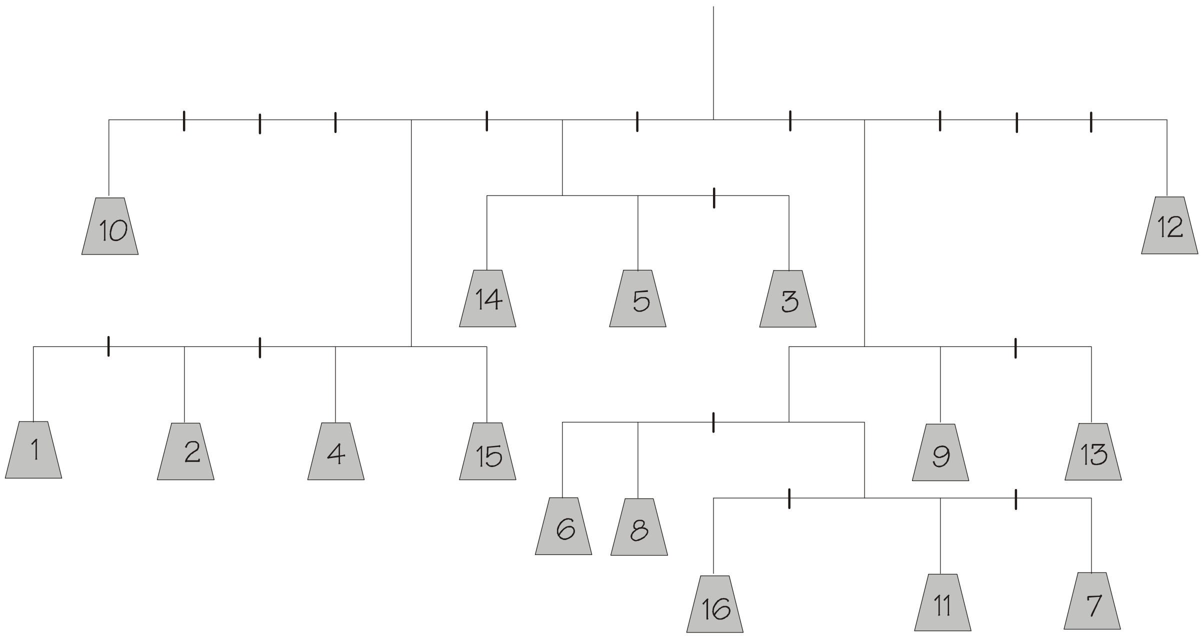


Input data (file MOBILE.IN)

```
(-3(-1(-1(-1,1,2),3),2),3(-2,1,2),6(-2,3))
```

Output data (file MOBILE.OUT)

```
((((8,6,1),5),10),(9,4,7),(3,2))
```



Input data (file MOBILE.IN)

```
(-8,-4(-5,-3,-1,1),-2(-1,1,3),2(-1(-3,-2,1(-2,1,3)),1,3),6)
```

Output data (file MOBILE.OUT)

```
(10,(1,2,4,15),(14,5,3),((6,8,(16,11,7)),9,13),12)
```