Huffman Codes

Description Suppose that we have to store a sequence of symbols (a file) efficiently, namely, we want to minimize the amount of memory needed. For the sake of simplicity, we assume that the symbols are restricted to the first six letters of the alphabet. For example, let us assume that the frequency of different symbols that you have to store is the following:

symbol	frequency
A	1000
B	150
C	200
D	800
E	300
F	50
Total	2500

As we have to store 6 different symbols, the obvious way is to encode each of them in 3 bits, as with 3 bits it is possible to encode 2^3 different symbols. With this encoding, we need $2500 \times 3 = 7500$ bits to store the above symbols. A different way to address the problem is the following. Instead of assigning to each symbol a code with the same length (i.e., number of bits), we assign shorter codes to symbols that are more frequent, and longer codes to symbols that are less frequent. One possible encoding according to this sequence is the following.

symbol	encoding
A	0
B	10101
C	1011
D	11
E	100
F	10100

According to this encoding, the number of required bits is:

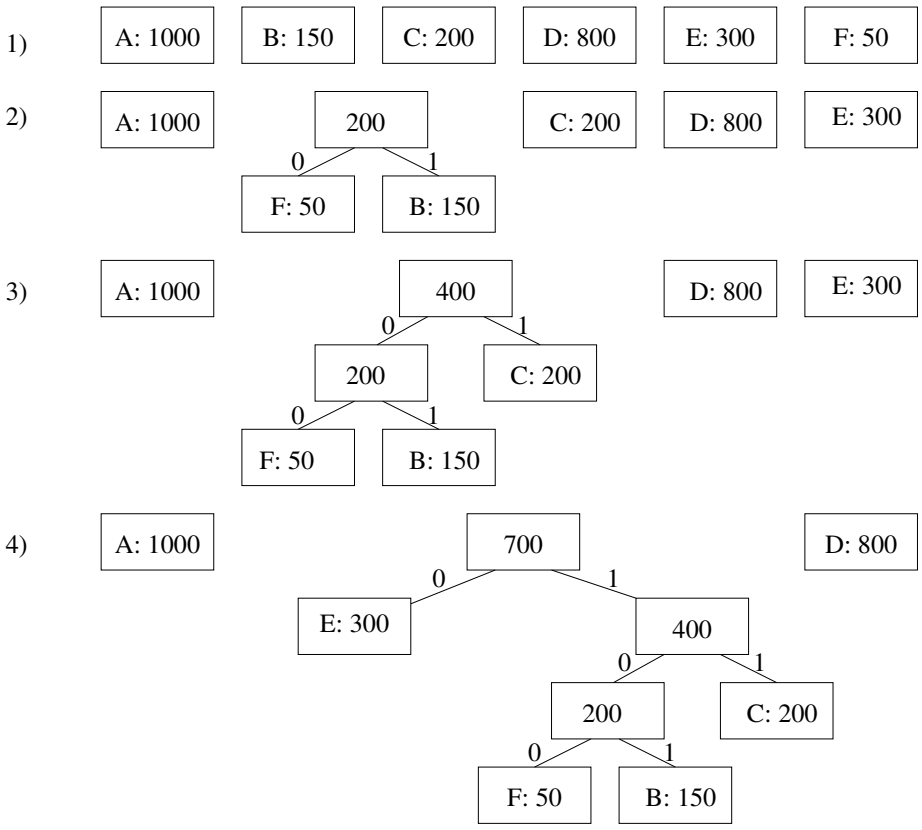
$$1000 \times 1 + 150 \times 5 + 200 \times 4 + 800 \times 2 + 300 \times 3 + 50 \times 5 = 5300.$$

This idea is the basis for the programs used to compress files. First, they analyze the input, then choose the codes, and then recode the input according to the determined codes.

Although this idea brings benefits in terms of space requirements, the use of variable-length codes presents some problems. Once we have coded a file according to a variable-length code, we must also be able to decode it in the original format (i.e., once we have compressed the file, we want to be able to decompress it). The encoding works only if the codes assigned to different characters are such that no code is a prefix of any other code. If this property does not hold, there is an ambiguity problem when trying to decompress the sequence.

You can prove that in the depicted example, no code is a prefix of any other code. For example: no code starts with 0 except for the code of A. So, while decompressing the file, if we find a symbol whose code starts with 0, we know that it is A. If we find a character whose code starts with 11, we know that it is D. It cannot be any other symbol, as no code starts with 11 other than D's code. And so on. How do we assign codes? This is done using a greedy

algorithm. We assign the shortest code to the most frequent character, the second longest code to the second most frequent character, and so on. The figure below illustrates the first few stages of the algorithm.



Given N characters with their respective frequencies, the algorithm initially builds N trees, each consisting of only a single node (step 1, in the figure). Then, iteratively, it joins together the trees whose roots have the lowest frequencies (steps 2, 3, etc. in the figure). *The tree with the lowest root frequency becomes the left child and the tree with the second-lowest root frequency becomes the right child*. The left children are associated with bit 0, and the right children are associated with bit 1. The internal nodes (i.e., root nodes created) can be thought of as dummy nodes storing a fictitious character (which does not appear in our sequence). This procedure is iterated until there is only one tree. At this point, to know the code associated with one symbol, you simply need to concatenate the 0s and 1s you encounter while moving from the root down to the symbol.

Note that the greedy strategy is applied in the reverse way. Symbols with low frequencies end up down in the tree (i.e., they are associated with long codes), while nodes with high frequencies are near the root (i.e., they are assigned short codes).

Questions and input structure The input consists of six integers, one per line. Each integer represents the frequency of the characters, A, B, C, D, E, and F, in this order. The test cases have been built so that while building trees it never happens that the same frequency appears twice. Then, the decision about which tree goes to the left and which one goes to the right is always straightforward, and the final tree should be unique.

Examples of input and output

Input

```
15
11
5
1
2
4
```

Output

```
A:0
B:10
C:110
D:11100
E:11101
F:1111
```

See the lab guidelines for submission/grading, etc., which can be found in Files/Labs.