Sixth Annual
University of Central Florida

acm • ΥΠΕ

High School Programming Tournament

Problems

Problem Name	Filename
Any Way You Read It	ANYWAY
There and Back Again	BACK
Hexagons!	HEXAGONS
The Lunch Machine	MACHINE
Stuck in the Middle	MIDDLE
Numbers of Numbers in Numbers	NUMBERS
The Impulsive Traveling Salesman	SALESMAN
Three Letter Acronyms	TLAS
Trooji	TROOJI

Call your program file: *Filename*.PAS or *Filename*.C
Call your input file: *Filename*.IN

For instance, if you are solving Three Letter Acronyms:

Call your program file: TLAS.PAS or TLAS.C
Call your input file: TLAS.IN

Any Way You Read It

Filename: ANYWAY

Who said anyone had to read from left to right? If you know what you're reading, you can read from left to right, right to left, up, down, or any way!

The Problem:

Write a program that figures out how to read the official name of this contest when the name is written multi-directionally in an 11 by 11 matrix. The official contest name (which here should be read left to right, top to bottom) is

UNIVERSITY OF CENTRAL FLORIDA
ASSOCIATION FOR COMPUTING MACHINERY
UPSILON PI EPSILON
HIGH SCHOOL PROGRAMMING TOURNAMENT

The Input:

There will be several 11 by 11 matrices in the input (11 lines of 11 characters each). Each matrix will contain the official contest name, with the words delimited by the '#' character, i.e., "#UNIVERSITY#OF#CENTRAL#FLORIDA#ASSOCIATION#FOR#COMPUTING#MACHI NERY#UPSILON#PI#EPSILON#HIGH#SCHOOL#PROGRAMMING#TOURNAMENT#" -- here a continuous string of characters (ignore the line break). Note that there are exactly 121 characters, to fit the 11 by 11 matrix. Each character of the contest name string will appear in the input matrix immediately below, above, right, or left of the character preceding it in the string (no diagonals). The line following each matrix will contain two integers from 1 to 11, indicating the row and column (respectively) in the matrix where the contest name string begins.

The Output:

The program must print an 11 by 11 matrix of symbols that show the directions being used to read the contest name string. Each symbol in the output matrix represents the direction taken from the corresponding character in the input matrix (the character in the same row and column) to reach the next character of the contest name string. Use the following characters as symbols to show directions:

Symbol		Next character of string is located:
>	"greater than"	to the right
<	"less than"	to the left
^	caret (Shift-6)	above
v	lowercase letter 'v'	below
*	asterisk (Shift-8)	[indicates last character of string]

Print a blank line between output matrices.

Sample Input:

```
#UNIVERSITY
LARTNEC#FO#
#FLORIDA#AS
F#NOITAICOS
OR#COMPUTIN
YRENIHCAM#G
#UPSILON#PI
IH#NOLISPE#
GH#SCHOOL#P
#GNIMMARGOR
TOURNAMENT#
1 1
OCIATIOR#G#
SROLF#NOCNM
SITRAL#FOIA
ADNEC#F#MTC
#A#UNIOTPUH
ILOREV#NANI
S#NSITYEMRN
PHIL#PAMOUE
EHGOORRMT#R
##SCHOGINGY
IP#NOLISPU#
5 3
```

Sample Output:

```
>>>>>>>>>>v
v<<<<<<<<<<
>>>>>>>>>>v
v<<<<<<<<<<
>>>>>>>>>>v
v<<<<<<<<<<
>>>>>>>>>>v
v<<<<<<<<<<
>>>>>>>>>>v
v<<<<<<<<<<
>>>>>>>>>>*

>>>>>>v>v>v
^v<<<<v^v^v
^v>>>^>^v^v
^v^<<<<*v^v
^<>>>v^^>^v
>>vv<<^^v<v
^v<>>>^^<^v
^>v>>v>v>^v
^v<^<v^v^<v
^>>>^>^>>^v
^<<<<<<<<<<
```

There and Back Again

Filename: BACK

Consider a function $f(n)$, where n is a non-negative integer. If the function is defined as

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ 4 \cdot f(n-1) + 3, & \text{if } n > 0 \end{cases}$$

and we want to find the value of $f(4)$, we do this by repeatedly applying the formula for f until we get to $f(0)$, i.e., going "there," and then substituting computed values back in to equations until we find $f(4)$, i.e., coming "back again:"

$$\begin{aligned} f(4) &= 4 \cdot f(3) + 3 \\ f(3) &= 4 \cdot f(2) + 3 \\ f(2) &= 4 \cdot f(1) + 3 \\ f(1) &= 4 \cdot f(0) + 3 \\ f(0) &= 1 \quad \text{[by definition]} \\ f(1) &= 4 \cdot 1 + 3 = 7 \\ f(2) &= 4 \cdot 7 + 3 = 31 \\ f(3) &= 4 \cdot 31 + 3 = 127 \\ f(4) &= 4 \cdot 127 + 3 = 511 \end{aligned}$$

Thus $f(4) = 511$. Notice that the process of going "there" is superficial; we could just think of evaluating $f(4)$ by starting at $f(0)$ and computing successive values until we reach $f(4)$. In other words, we only need to come "back again."

Now consider a more general form of this problem. Let us define the function as

$$f(n) = \begin{cases} c, & \text{if } n = 0 \\ a \cdot f(n-1) + b, & \text{if } n > 0 \end{cases}$$

where a, b, and c are all integer constants. The method for finding $f(n)$ is the same as before.

The Problem:

Write a program that computes $f(n)$, given values for n, a, b, and c.

The Input:

There will be several lines of input. Each line will contain the values n, a, b, and c in that order, separated by spaces. n will be a non-negative integer less than or equal to 1500 and a, b, and c will all be integers, and have values such that the products $a \cdot f(n-1)$, $a \cdot f(n-2)$, $a \cdot f(n-3)$, and so on, down to $a \cdot f(0)$, will be less than 32768 in absolute value.

The Output:

For each line in the input, your program must print the following message:

```
f(n) = value
```

where `n` is the input value of n and `value` is the integer obtained from evaluating $f(n)$. `value` is guaranteed to be less than 32768 in absolute value, as are all values of $f(n)$ for the given a, b and c required to calculate `value`. Print each message on a new line, and print a blank line between messages.

Sample Input:

```
4 4 3 1
2 1 2 1
3 1 1 -4
87 -1 -6 0
```

Sample Output:

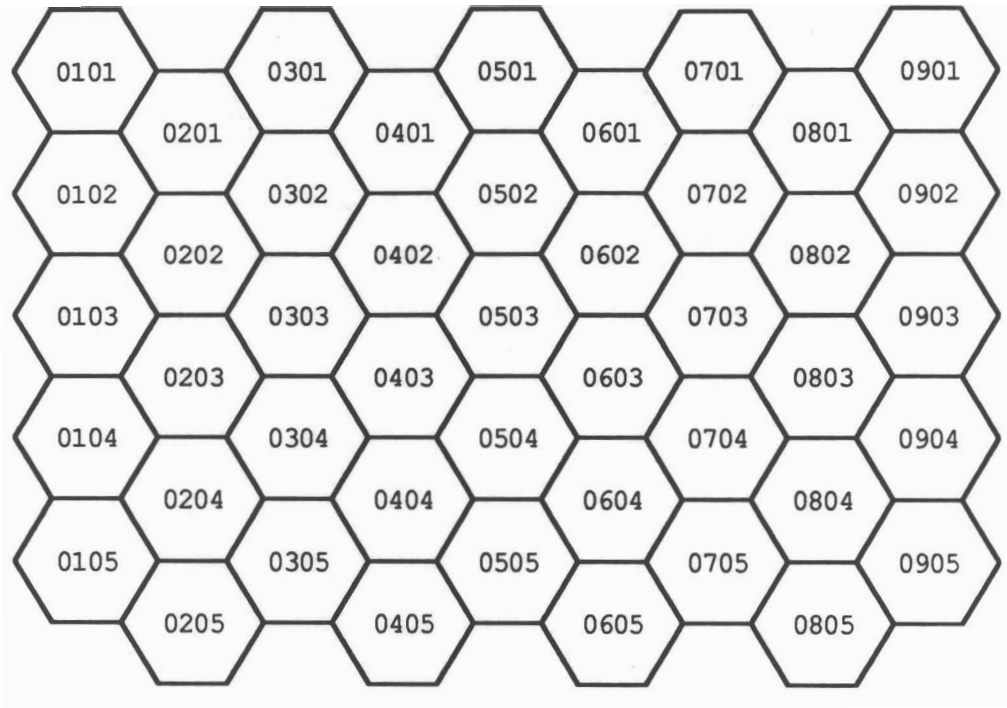
```
f(4) = 511

f(2) = 5

f(3) = -1

f(87) = -6
```

Hexagons!

Filename: HEXAGONS

Many popular board games are played on boards of squares. Examples include chess, checkers, Othello, etc. However, there are many board games and role-playing games (e.g. Battletech) which instead use boards of interlocking hexagons. Hexagons are preferable to squares for such games, because they are better models of real-world characteristics such as distances between points and terrain shapes.

A coordinate notation commonly used for hexagonal boards uses four digits to denote each hexagon. The first two digits indicate the column in which the hexagon is located, and the second two digits give the position of the hexagon in that column, starting from the top. Using this notation, the largest possible board is 99 columns of 99 hexagons each. The upper left portion of a such a hexagonal game board is given below. Note that odd-numbered columns are higher than even-numbered columns.



Suppose you wanted to move from one hexagon on such a board to another hexagon by traveling through a sequence of hexagons that are connected. There would be a set of shortest possible paths--paths consisting of the smallest number of steps. The number of steps required for such a path is called the *distance* between the two hexagons. For example, the distance between hexagon 0101 and hexagon 0303 is 3, because a shortest path is 0201-0302-0303. The paths 0201-0202-0303 and 0102-0202-0303 are also shortest paths, but there is no path shorter than 3 hexagons.

The Problem:

Write a program to determine the distance between two hexagons on a hexagonal game board of this type that is the largest possible size, 99 columns of 99 hexagons.

The Input:

Each line of the input will contain the coordinates of two hexagons in a 99 by 99 board. The coordinates will range from 0101 to 9999, using the coordinate system given. There will be exactly one space between the coordinates for the two hexagons.

The Output:

For each pair of hexagons in the input file, your program must print the message:

`The distance from` *`hexagon1`* `to` *`hexagon2`* `is` *`distance`*`.`

where *`hexagon1`* is the coordinate notation for the first hexagon, *`hexagon2`* is the coordinate notation for the second hexagon, and *`distance`* is the distance between the hexagons. Print the messages one per line on consecutive lines.

Sample Input:

```
0101 0303
0201 0602
0101 0203
0101 1001
```

Sample Output:

```
The distance from 0101 to 0303 is 3.
The distance from 0201 to 0602 is 4.
The distance from 0101 to 0203 is 3.
The distance from 0101 to 1001 is 9.
```

The Lunch Machine

Filename: MACHINE

Ah, lunch in the age of technology! No waiting for a table, no waiting in line, no drive-thru nightmares: just you and the vending machine--assuming it works properly. One such automated vendor has four kinds of products: gum, chips, soda, and sandwiches. The machine accepts nickels, dimes, quarters, and one-dollar bills. To make a purchase, the buyer must insert enough money to equal or exceed the cost of the item, and then press a button indicating her selection. If the buyer selects the **change** button, all of the money that has been inserted is returned to her. If she selects a food item, the machine checks to make sure that enough money has been inserted. If so, it dispenses the item and returns any extra money. If not, it gives her a message indicating that more money is required.

The current prices of the items in this machine are as follows:

GUM	35 cents
CHIPS	40 cents
SODA	50 cents
SANDWICH	85 cents

The Problem:

Write a program to simulate the operation of this vending machine. Given a series of actions taken by the buyer, the program must output the appropriate responses of the machine. The machine starts with no money in it. Whenever money is inserted into the machine, the program should output nothing but keep track of the amount. If a product button is pressed (i.e., GUM, CHIPS, SODA, or SANDWICH) the program must verify that enough money has been inserted to cover the amount of purchase. If there is not enough money in the machine, the program must give an appropriate message to the buyer but not return any money. If the purchase is valid, however, the program should indicate the amount of money to return to the buyer, and clear the amount that was stored in the machine. If the **change** button is pressed, the amount of money to return to the buyer should be given, and the amount of money in the machine should be cleared.

The Input:

Each line of the input file will contain only a one-word string, in all uppercase letters, which represents an action of the buyer. The words `GUM`, `CHIPS`, `SODA`, `SANDWICH`, and `CHANGE` indicate buttons pressed by the buyer. The words `NICKEL`, `DIME`, `QUARTER`, and `DOLLAR` indicate money inserted into the machine. Nothing other than these words will appear in the input.

The Output:

If the buyer attempts to make a purchase without putting enough money in the machine, the program should print the message

```
Insufficient funds
```

otherwise, when the buyer presses a button, the program should print out the amount of money to be returned, in the format

```
$x.xx
```

that is, with at least one digit before the decimal point and exactly two digits after the decimal point. Each line of output should immediately follow the previous line.

Sample Input:

```
DIME
QUARTER
CHANGE
DIME
DIME
DIME
QUARTER
SODA
DOLLAR
NICKEL
SANDWICH
NICKEL
NICKEL
DIME
NICKEL
CHIPS
DOLLAR
QUARTER
CHANGE
```

Sample Output:

```
$0.35
$0.05
$0.20
Insufficient funds
$1.50
```

Stuck in the Middle

Filename: MIDDLE

Signmakers and graphic designers, among others, often have a need to center text. In order to do this, it is useful to know which characters are in the middle of a line. Some do this so much that they are able to spot the middle characters at a glance. Out of habit, their gaze is "stuck in the middle."

The Problem:

Write a program that spots the middle character or characters of a line of text. If the line of text has an even number of characters, there will be two middle characters. If the line of text has an odd number of characters, only one will be in the middle.

The Input:

The input will consist of several lines of characters. Every line will contain at least one and no more than seventy characters. No line will end with the space character.

The Output:

For each line of the input file, your program should print the middle character or characters in the order they appear in the input, surrounded by square brackets [ ]. Print the bracketed character sets one per line on consecutive lines, as shown in the sample below.

Sample Input:

```
12345
xyzw
Chlorine Sale: Come in and Look
------> Notice <------
** Yes! We have GRITS! **
    $4.00 Specials
```

Sample Output:

```
[3]
[yz]
[C]
[ti]
[a]
[0 ]
```

Numbers of Numbers in Numbers

Filename: NUMBERS

There are any number of numerical sequences. Here's one of our favorites. Start with any non-negative integer *s*. The next term of the sequence is determined by counting consecutive occurrences of each of the digits of *s*, and replacing the consecutive occurrences with the count and a single digit. Continue this process to compute successive terms in this sequence. For example, the starting term 124 has one 1 followed by one 2 followed by one 4, i.e., 1 1, 1 2, and 1 4. The second term of this sequence would be 111214. In turn, this term has three 1's followed by one 2 followed by one 1 followed by one 4, so the third term of this sequence is 31121114. Similarly, the fourth term will be 1321123114, and so on.

The Problem:

Write a program which will determine the *n*th term of the sequence that starts with *s* when given the integers *n* and *s*.

The Input:

There will be several sets of data in the input, each occupying two lines of the input file. The first line of each data set will contain the starting term *s*, a non-negative integer up to 50 digits long. The second line will contain *n*, a positive integer less than or equal to 15. All numbers of each sequence up to and including the *n*th term are guaranteed to be no more than 50 digits long.

The Output:

For each pair of values for *s* and *n* in the input, your program must print the *n*th term of the sequence starting in the leftmost column. Print each term on a new line, separating terms with a blank line.

Sample Input:

```
124
4
313
7
333221999999999999
3
```

Sample Output:

```
1321123114

31232112311321322113

2322311219
```

The Impulsive Traveling Salesman

Filename: SALESMAN

One of the classic problems in computer science is known as the Traveling Salesman Problem: given a group of cities that a salesman must visit, determine the shortest route which visits all of the cities exactly once. Don't panic! You don't have to solve this. It belongs to a class of problems known as *NP-Complete* (which is Latin for "a pain in the neck"), meaning that the problem grows more difficult in leaps and bounds as its size increases. Consider instead a salesman who is impulsive--that is, after visiting a city, he moves on to the nearest city that he has not yet visited. This will give the salesman a reasonably short path, although almost never the shortest path.

The Problem:

Write a program which, given a list of cities and their positions on a map, will determine the final city which is visited by the salesman.

The Input:

There will be several sets of data in the input, each representing a group of cities. Each data set will start with a line containing a single integer *n*, which may range from 1 to 20 inclusive, indicating the number of cities in that group. Descriptions for the *n* cities will begin on the next line. Each city description will consist of two lines, the first of which will contain the city name (a string of uppercase and lowercase letters and spaces). No city name will be longer than 50 characters. The second line of the city description will contain two integers *x* and *y*, indicating the coordinates of the city on the map. *x* and *y* are guaranteed to be less than 32768 in absolute value. The intrepid salesman, anxious to get started, will always start his route at the first city listed. Fortunately for the salesman (who can get quite frustrated when he's in a hurry), there will always be exactly one unvisited city closest to his current location.

The Output:

For each data set, your program should print the message

`Salesman ends up in city.`

where `city` is the name of the city where the salesman's journey ends. Print each message on a new line. There should be no blank lines between messages.

Sample Input:

```
4
Boston
10 10
Los Angeles
0 1
Buffalo
8 10
San Francisco
0 3
3
Jacksonville
1 3
Tallahassee
-2 3
Orlando
0 0
```

Sample Output:

```
Salesman ends up in Los Angeles.
Salesman ends up in Orlando.
```

Three Letter Acronyms

Filename: TLAS

Acronyms have long been used to simplify the discussion and writing of technical terms and phrases, both for clarity and for expediency. The most popular kind of acronym is the **TLA**, which itself is a Three Letter Acronym. More familiar to programmers are TLAs such as **CRT**, for "cathode ray tube," and **TSR**, for "terminate and stay resident."

The Problem:

Write a program which, given a list of TLA definitions and a paragraph of text, will expand any defined TLAs it finds in the text and will reformat the text as needed to maintain the original margins, spacing, and sentence capitalization.

The Input:

The first line of input will contain a positive integer *n*, less than 20. The next *n* lines of input will each contain one TLA definition. Each TLA definition will consist of three uppercase letters (the Three Letter Acronym), followed by a single space, followed by the expansion of the TLA. The expansion of the TLA will consist of one or more words (a **word** is any continuous group of uppercase and/or lowercase letters), separated by single spaces, and will contain at least three but no more than 50 characters total. No expansion will contain a defined TLA.

The paragraph of text will begin on the line immediately following the last TLA definition. The text will consist only of uppercase and lowercase letters, spaces, and periods. Any two words on a given line will be separated from each other by a single space, or, between sentences, by a period and a single space. No line will start or end with a space. Each line will contain as many whole words as possible without making the line more than 70 characters long. No word in the text will be more than 50 characters long. Every sentence will begin with a capital letter. The last line of the paragraph will be terminated by end-of-file. There will be only one paragraph.

The Output:

Your program must print out the paragraph of text with all of the defined TLAs replaced by their expansions. On any line where the TLA expansion causes the line to be longer than 70 characters, words at the end of the line must be moved to the beginning of the next line. Move as few words as possible when doing this. Note that this may cause the next line to exceed the 70 character limit, in which case words on that line will also have to be moved to its successive line, and so on.

All periods should immediately follow the last word of a sentence--on the same line. If a TLA expansion appears at the beginning of a sentence, your program should make sure that the first word of the expansion begins with a capital letter. Otherwise, the expansion should appear precisely as it is given in its definition in the input. A TLA in the text should only be expanded if it appears as a single word in all uppercase letters. All words in the paragraph must be spaced in the same manner as the input text.

Sample Input:

```
7
CRT cathode ray tube
TSR terminate and stay resident
LAN local area network
UCF University of Central Florida
UPE Upsilon Pi Epsilon
OUT OPERATING UNDER TIMESLICE
RAM random access memory
Last year at the UCF ACM UPE High School Programming Tournament I saw
a message appear on the CRT. I asked a contest official what caused it
to appear. He said that a TSR program allowed the computer to display
messages sent through the LAN. He pulled the LAN card out of the
computer and fixed it. The message said RAMDISK OUT. RAM caching with
a fast hard disk usually works better than using a RAMDISK.
```

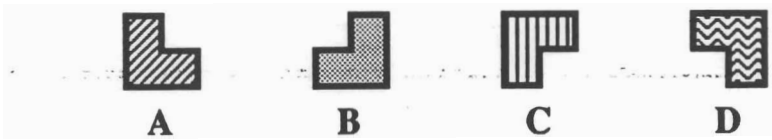
Sample Output:

```
Last year at the University of Central Florida ACM Upsilon Pi Epsilon
High School Programming Tournament I saw a message appear on the
cathode ray tube. I asked a contest official what caused it to appear.
He said that a terminate and stay resident program allowed the
computer to display messages sent through the local area network. He
pulled the local area network card out of the computer and fixed it.
The message said RAMDISK OPERATING UNDER TIMESLICE. Random access
memory caching with a fast hard disk usually works better than using a
RAMDISK.
```

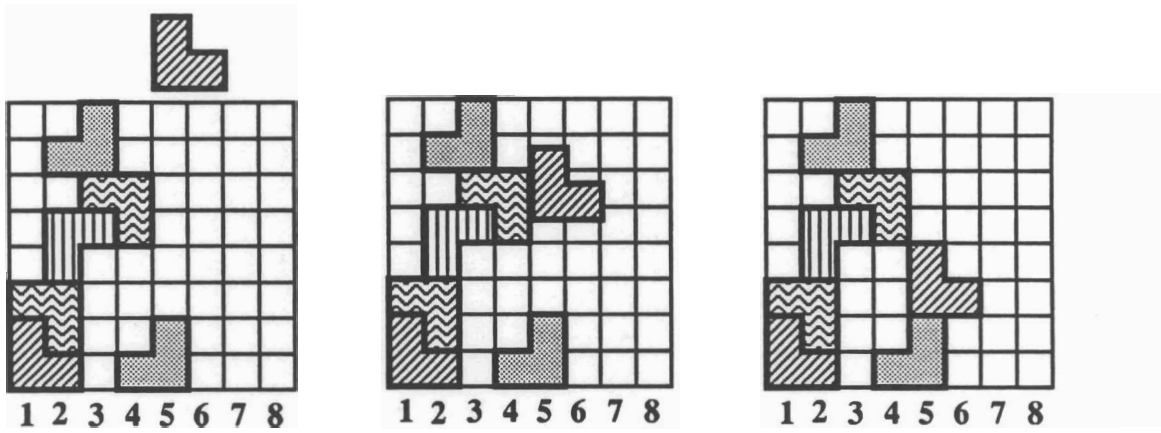
Trooji

Filename: TROOJI

The game Trooji involves dropping differently-shaped 2-dimensional pieces into a vertical 8 by 8 grid. The four pieces (each 3 connected squares) used are designated by letters as follows:



The pieces are dropped into the grid from above, one at a time, into a pair of adjacent columns in the grid. No piece rotates or moves sideways once it has been dropped, but falls from a position above the top row down through each successive row, until it is blocked by another piece or lands in the last row. The following illustration shows the progression of a type A piece falling into columns 5 and 6 of a grid that already has had several pieces dropped into it:



If a piece dropped into the grid does not fit completely (i.e., it would stick out the top) it is **rejected**--that is, removed from the grid entirely. In the grid shown above, any piece dropped into columns 2 and 3 will be rejected. The only piece that would *not* be rejected if dropped into columns 1 and 2 is a type C piece.

The Problem:

Write a program which will take as input a list of pieces sequentially dropped into an empty Trooji grid and the respective columns into which they were dropped, and produce a list of rejected pieces.

The Input:

There will be several sets of data in the input file, each representing a group of pieces dropped into an empty Trooji grid. The first line of each data set will contain an integer *n*, indicating the number of pieces dropped into the empty grid. The following *n* lines will each contain a single uppercase letter (A-D, the piece type), followed by a space and an integer from 1 to 7 (the left column of the pair into which the piece was dropped). The pieces are dropped in the order they occur in the input.

The Output:

For each rejected piece, your program must print the message

`Reject piece` *n*`:` *type column*

where *n* is the number of the piece (start counting at the first piece dropped for each grid) and *type* and *column* are the piece type and column number given in the input. List the rejected pieces in the order they were dropped. Print all messages for a given grid on consecutive lines, but print a blank line at the end of output for each grid.

Sample Input:

```
15
A 1
B 4
D 1
C 2
D 3
B 2
A 5
A 2
B 2
C 2
D 2
A 1
B 1
C 1
D 1
8
C 3
D 3
B 4
B 5
A 4
C 4
C 3
D 7
```

Sample Output:

```
Reject piece 8: A 2
Reject piece 9: B 2
Reject piece 10: C 2
Reject piece 11: D 2
Reject piece 12: A 1
Reject piece 13: B 1
Reject piece 15: D 1

Reject piece 5: A 4
Reject piece 7: C 3
```