**Java program:** Prob15.java

**Input File:** Prob15.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Part numbers are a big deal to any company, and the pattern of a part number can contain a lot of imbedded information. Your task will be to write a program that will take a list of part numbers as input and output the list of parts that conforms to a list of 15 part number patterns that you have been given below so these parts can be set aside for further processing.

Part number pattern rules:

* Characters that are outside a pair of parenthesis ( ) are considered to be static and cannot change, meaning that the input part number must contain the exact string in the pattern.
  + Example: The pattern ANABC-123( )-B-(A,B,C) would match part number ANABC-1234-B-A but not ANABC-1224-B-A.
* Anything inside a pair of parenthesis ( ) is considered to be dynamic and must conform to the pattern in the parenthesis. Patterns separated by commas are considered to be a list of values or range of values that are valid for that pattern.
  + Example: The pattern ANABC-123(A,B,C)-C would match only the part numbers ANABC-123A-C, ANABC-123B-C, and ANABC-123C-C.
* Empty parenthesis (nothing inside) indicates a wild-card, meaning that any length of character(s) may exist or not.
  + Example: The pattern ANABC( )-C would match both ANABCGOOD123-C and ANABC-C.

The following is the list of part number patterns to match against:

CW3101A( )-( )(P,S)

CW426( )-( )

CW427-( )C( )

CW500A4-(3 THRU 8)

CW507B1032R(8 THRU 32 BY 2)

CW3085-(001 THRU 050,102 THRU 178,201 THRU 284)

CWCG20Z-(M,N,P,Q,R,S)(101 THRU 999)B

CWDPX2-( )(P,S)( )(P,S)33-00( )

CWT02(E,P)18-(11,32)(P,S)( )

CW12326(E,G,J,K,L,M,N,P,R,T)(00375 THRU 20000)(A,S)

CW15232C(02,04,06,08,3 THRU 6)(-,H)(3 THRU 16,18 THRU 48 BY 2)

CW15263-(02,04,06,08,3,4,5,6)(-,H)(3 THRU 16,18 THRU 48 BY 2,59)

CW20001(C,P)(H,X,Y)(2,3,4,5,6,8,9,10,12,14,16,17)-(0100 THRU 7200)

CW102-2-(6 THRU 10,12 THRU 50 BY 2)-(6 THRU 10,12 THRU 50 BY 2)

CW8602-( )B( )PNSPM26

**Program Input**

The file Prob15.in.txt will contain a list of part numbers to be checked against the patterns.

**Example Input:**

CW3101AABC-123-P

CW426-3

CW427-C0DEWARSCR0CKS

CW500A4-7

CW507B1032R8

CW3085-001

CWCG20Z-M101B

CWDPX2-PS33-00

CWT02E18-11PS10

CW12326E00375A

CW15232C02-16

CW15263-02-3

CW20001CH2-0100

CW102-2-6-6

CW8602-( )B( )PNSPM26

ThisWillNotMatch

**Program Output**

Your program should output only the part numbers in the list that conform to any of the patterns given above. Your output should be in the same order that the input file was in. Part numbers that do not match any pattern should be ignored.

**Example Output:**

CW3101AABC-123-P

CW426-3

CW427-C0DEWARSCR0CKS

CW500A4-7

CW507B1032R8

CW3085-001

CWCG20Z-M101B

CWDPX2-PS33-00

CWT02E18-11PS10

CW12326E00375A

CW15232C02-16

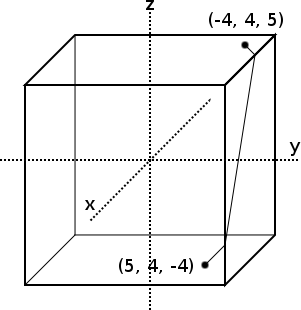
CW15263-02-3

CW20001CH2-0100

CW102-2-6-6

CW8602-( )B( )PNSPM26

**Java program:** Prob16.java

**Input File:** Prob16.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Dr. Bizarro has been tasked with developing a planetary rover prototype to explore the surface of an implausible set of cube-shaped planetoids discovered in the far reaches of our solar system. He wants you to develop a program that determines the shortest distance to travel between two points on the surface of one of these planetoids – whose faces are also implausibly flat – so that vital statistics such as the amount of power required to travel between the two points can be determined.

You will be provided with the length of an edge on the planetoid (in km) as well as the location of each point on the surface in a Cartesian format (x,y,z). The origin point (0,0,0) of the Cartesian system will be set as the center of each world’s mass and all edges of the world will be parallel to one of the system’s axes.

Remember, the distance between two points on a Cartesian plane can be determined using the following formula:

**Program Input**

The file Prob16.in.txt will contain planetoid data in groups of three lines:

1. The first line of each group will be the length of the planetoid’s sides.
2. The second line of each group will be the location of the starting point in Cartesian format.
3. The third line of each group will be the location of the ending point in Cartesian format.

**Example Input:**

10

5 4 -4

-4 4 5

200

1 1 -100

-35 -100 -100

30

-15.0 15.0 15

15 -15.0 -15

**Program Output**

Your program should calculate the distance between the two points, and print it out to 4 decimal places using standard rounding rules.

**Example Output:**

14.1421

107.2241

67.0820

**Java program:** Prob17.java

2

1

3

4

5

7

6

8

9

10

12

11

13

14

15

17

16

18

19

20

22

21

23

24

25

**Input File:** Prob17.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

A puzzle is played on a 5x5 grid of 25 squares. The squares are numbered as shown in the diagram. You start in the square numbered 1 (upper left hand corner). You can travel up, down, right, or left (not diagonally) to another square containing a wall in common with the square you are in. One square contains the treasure (your goal).

Some of the other squares contain blocks. A block fills the entire square and prevents you from traveling through that square unless you push the block out of the way. You can push a block to another square as long as there is an empty square on the opposite side of the block. You cannot push a block on top of the treasure. You cannot push a block diagonally or outside the grid.

A move consists of pushing a block from one square to an adjacent square. A move is recorded as a pair of numbers representing the square a block is moving from and a square that block is moving to, separated by a hyphen. For example, if you pushed a block from square 13 to square 8, you would record the move as 13-8. The object of the puzzle is to reach the square containing the treasure in the fewest number of moves. Movement from square to square without pushing any blocks is free.

**Program Input**

The file Prob17.in.txt will contain lines of lists of 25 integers separated by spaces representing the contents of the 25 numbered squares on the board. A "0" indicates that the square is empty as you start the puzzle. A "1" indicates that the square contains a block. A "2" indicates that the square contains the treasure. Square number 1 will always contain a zero at the start because that is your starting position, and only one square will contain a "2". Each new line is a new puzzle.

**Example Input:**

0 0 1 0 0 0 1 0 1 0 0 1 1 0 1 0 0 1 0 2 0 0 0 1 1

0 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 2 0

Example 2

Example 1

T

T

**Program Output**

Your program should print the shortest series of moves that opens a path to the treasure for each puzzle. In the event that the treasure is already accessible at the beginning of the puzzle, your program should print out the phrase "No moves necessary". In the event that the treasure cannot be reached, your program should print out the phrase "No solution".

**Example Output:**

3-4 9-10

15-20 20-25 19-18

**Java program:** Prob18.java

**Input File:** Prob18.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Companies spend a lot of money on integrated circuit testing to make sure that their manufacturing processes are as good as possible. There are many things that can go wrong with making a circuit – too much metal at a given location, not enough vertical distance between crossing wires, poorly soldiered contacts…the list goes on and on. Fortunately, many of these defects can be effectively modeled as something called a "stuck-at fault". This means that a given node in the circuit is assumed to be stuck at a given value. Your task is to write a program that will read a circuit model and determine the correct output for a given set of inputs either with or without stuck-at faults applied to the circuit.

The only valid values for a node in the circuit are 0 and 1. To implement your program, you will need to know about the different logic gates that can be used to construct circuits:

|  |  |  |
| --- | --- | --- |
| **Gate Type** | **Description** | **Symbol** |
| PI (Primary Input) | The PI gate is a placeholder for the primary input signals. The inputs to your program will set the values of the PI gates. | N/A |
| AND | The AND gate outputs a 1 if and only if all its inputs are a 1. Otherwise, it outputs a 0. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-AND_ANSI.svg.png |
| NAND | The NAND is the negation of the AND gate. It outputs a 0 if and only if all its inputs are a 1. Otherwise it outputs a 1. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-NAND_ANSI.svg.png |
| OR | The OR gate outputs a 1 if any of its inputs are 1. If all inputs are 0, it outputs a 0. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-OR_ANSI.svg.png |
| NOR | The NOR gate is the negation of the OR gate. It outputs a 0 if any of its inputs are 1. If all inputs are 0, it outputs a 1. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-NOR_ANSI.svg.png |
| XOR | The XOR gate outputs a 1 if it has an odd number of inputs that are 1. Otherwise, it outputs a 0. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-XOR_ANSI.svg.png |
| XNOR | The XNOR gate is the negation of the XOR gate. It outputs a 0 if it has an odd number of inputs that are 1. Otherwise, it outputs a 1. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-XNOR_ANSI.svg.png |
| NOT | The NOT gate outputs a 1 if its input is a 0. Otherwise, it outputs a 1. | C:\Users\trinkmr\Desktop\CodeQuest\gates\100px-NOT_ANSI.svg.png |
| PO (Primary Output) | The PO gate is a placeholder for your program’s output. You will be printing out the value of the PO gates. | N/A |

PI, PO, and NOT gates may only contain a single input. Other gate types can have an unbounded number of inputs.

**Program Input**

The file Prob18.in.txt will contain two sections:

1. The circuit model section will contain lines in the following format:

NodeNumber GateType InputList

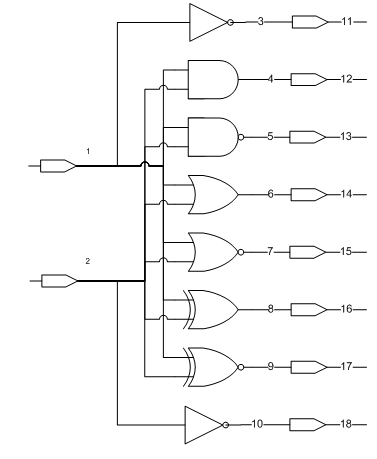
Where NodeNumber is the unique id of the node, GateType is one of the gate types described in the table above, and InputList is a list of node numbers that act as the inputs to the current gate being described. All values will be separated by spaces. A line with only the word "INPUTS" marks the end of the circuit model section.

1. The circuit input section will contain an ordered list of values for the PI gates. The values for the PI gates should be applied in the order that the PI gates were encountered in the input section, not necessarily in numerical node order.

If there are stuck-at fault values to be applied to the circuit, the following repeating pattern will follow the PI inputs on the same line, once for each stuck-at fault:

NODE NodeNumber SA Value

Where NodeNumber is the node that has the fault and Value is the value that the node is stuck at.

**Example Input:**

1 PI

2 PI

3 NOT 1

4 AND 1 2

5 NAND 1 2

6 OR 1 2

7 NOR 1 2

8 XOR 1 2

9 XNOR 1 2

10 NOT 2

11 PO 3

12 PO 4

13 PO 5

14 PO 6

15 PO 7

16 PO 8

17 PO 9

18 PO 10

INPUTS

0 0

0 1

1 0

1 1

1 1 NODE 3 SA 1 NODE 4 SA 0

**Program Output**

Your program should print the values of the PO gates in the order that they were encountered in the circuit model section. Print each PO on its own line, and insert a blank line between each set of outputs to denote the running of each input set. PO values should be printed as follows:

PO NodeNum = Value

**Example Output:**

PO 11 = 1

PO 12 = 0

PO 13 = 1

PO 14 = 0

PO 15 = 1

PO 16 = 0

PO 17 = 1

PO 18 = 1

PO 11 = 1

PO 12 = 0

PO 13 = 1

PO 14 = 1

PO 15 = 0

PO 16 = 1

PO 17 = 0

PO 18 = 0

PO 11 = 0

PO 12 = 0

PO 13 = 1

PO 14 = 1

PO 15 = 0

PO 16 = 1

PO 17 = 0

PO 18 = 1

PO 11 = 0

PO 12 = 1

PO 13 = 0

PO 14 = 1

PO 15 = 0

PO 16 = 0

PO 17 = 1

PO 18 = 0

PO 11 = 1

PO 12 = 0

PO 13 = 0

PO 14 = 1

PO 15 = 0

PO 16 = 0

PO 17 = 1

PO 18 = 0