A1 Problem Statement

Unit Pricing

Given up to 10 items (weight in ounces and cost in dollars) determine which one by order (e.g. third) is the cheapest item in terms of cost per ounce. Also output the cost per ounce of the cheapest item. The first input will be the number of items. Also indicate a tie for the cheapest item if it occurs.

Example:

Number of items: 3

Weight 1: 20

Cost 1: 30.00

Weight2: 10

Cost2: 6

Weight2: 4

Cost 3: 6

Answer: items 1 and 3 are the cheapest at 1.50 per ounce

A1 Solution
Unit Pricing

```
int main()
{
	double W[4], P[4], CPU[4], MinCPU;
	int n,i,MinI;
	std::cout << "\nenter number of choices: ";
	std::cin >> n;
	for (i = 0; i < n; i++) {
		std::cout << "\nenter weight " << i+1 << ": ";
		std::cin >> W[i];
		std::cout << "\nenter price " << i+1 << ": ";
		std::cin >> P[i];
		CPU[i] = P[i] / W[i];
	}
	MinCPU = CPU[0]; MinI = 1;
	for (i = 1; i < n;i++) {
		if (MinCPU > CPU[i]) { MinCPU = CPU[i]; MinI = i + 1; }
	}
	std::cout << "\nItem " << MinI
		<< "is a best or equal purchase. It is $"
		<< MinCPU << " per ounce.";
	}
return 0;
}
```

A2 Problem Statement
Dice Game

There are several games and puzzles that are done using dice. This problem will use standard dice that have six sides, numbered 1, 2, 3, 4, 5, and 6. If we roll some dice, the total sum is the sum of values on the top face of the dice. Given the total sum and the number of dice that were rolled, output all the combinations of rolls that are equal to the total sum. Your program should also output the total number of combinations found.

Input Format

The input format is the number of dice as a whole number, followed by the total as a whole number.

Output Format

All of the dice combinations should be outputted first, one per line. Finally, the total number of dice combinations should be outputted on its own line.

Sample Input

```
3 15
```

Sample Output

```
(3, 6, 6)
(4, 5, 6)
(4, 6, 5)
(5, 4, 6)
(5, 5, 5)
(5, 6, 4)
(6, 3, 6)
(6, 4, 5)
(6, 5, 4)
(6, 6, 3)
10
```

```python
import itertools


def get_input():
    s = input().split(" ")
    return int(s[0]), int(s[1])


numbers = [1, 2, 3, 4, 5, 6]
results = []
num_dice, total = get_input()
count = 0

for roll in itertools.product(numbers, repeat=num_dice):
    if sum(roll) == total:
        print("({})".format(",".join(str(n) for n in roll)))
        count += 1

print(count)
```

A3 Problem Statement

Encoding Roman Numerals

Roman numerals were invented by the ancient Romans and served as the Western number system for over a thousand years. The Roman system uses letters for specific values: “I” for one; “V” for five; “X” for ten; “L” for fifty; “C” for hundred; “D” for five hundred; “M” for thousand. The symbols were generally listed from highest value to lowest. E.g. MCCLXII represents 1262: M is 1000; CC is 200; LX is 60; and II is 2. The exceptions to the ordering are: “I” before a “V” represents 4; “I” before an “X” represents 9. Similarly, “XL” represents 40; “XC” 90; “CD” 400; “CM” 900. (based on Wikipedia’s “Roman numerals”).

Given a decimal integer, print the Roman numeral equivalent.

Examples:

Input: 14

Output: XIV

Input: 1955

Output: MCMLV

Input: 2017

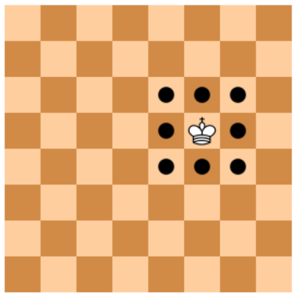
Output: MMXVII

A3 Solution
Encoding Roman Numerals

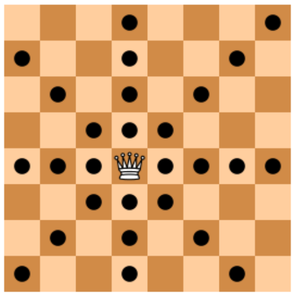
```cpp
int main()
	int n;
	{
		std::cout << "\nenter number: ";
		std::cin >> n;
		std::cout << "\n";
		while (n >= 1000) { std::cout << "M"; n = n - 1000; }
		if (n >= 900) { std::cout << "CM"; n = n - 900;}
		if (n >= 500) { std::cout << "D"; n = n - 500; }
		if (n >= 400) { std::cout << "CD"; n = n - 400; }
		while (n >= 100) { std::cout << "C"; n = n - 100;}
		if (n >= 90) { std::cout << "XC"; n = n - 90; }
		if (n >= 50) { std::cout << "L"; n = n - 50; }
		if (n >= 40) { std::cout << "XL"; n = n - 40; }
		while (n >= 10) { std::cout << "X"; n = n - 10;  }
		if (n == 9) { std::cout << "IX"; n = n - 9; }
		if (n >= 5) { std::cout << "V"; n = n - 5; }
		if (n == 4) { std::cout << "IV"; n = n - 4; }
		while (n > 0) { std::cout << "I"; n = n - 1;}
	}
	return 0;
```

A4 Problem Statement
Chess

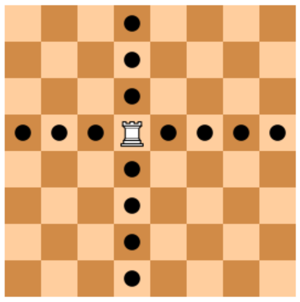
Chess is a very popular board game played between two players on a 8 x 8 checkered board. There are several different pieces used in the game. Given a chess piece and its initial location, calculate all the valid single moves a chess piece can make on the board. You can assume no other pieces on the board except the one that is given. Moves cannot go outside the bounds of the board. See below for descriptions of how each piece can move. We will not be considering the pawn piece for this problem. Each square on the board is numbered, starting with (1,1) in the bottom left hand corner. The x value is in the horizontal direction.



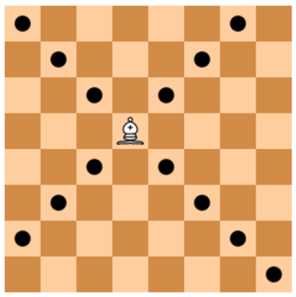
The **king** can move one square in any direction.



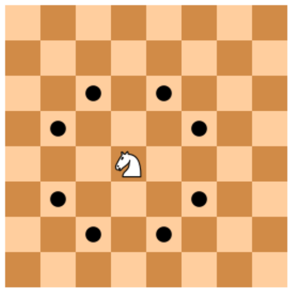
The **queen** can move any number of squares in the vertical, horizontal, or diagonal directions



The **rook** can move any number of squares in the horizontal or vertical directions.



The **bishop** can move any number of squares in the diagonal direction.



The **knight** can move in an "L" shape. This equates to either:

- two squares vertically and one square horizontally
- or two squares horizontally and one square vertically

Input Format

The chess piece will be entered first, followed by the x and y location as integers. The chess piece will be abbreviated**: K** for **king**, **Q** for **queen**, **B** for **bishop**, **R** for **rook**, and **N** for **knight**.

Output Format

You should first output the number of moves that are possible on a single line, followed by each x and y location of all the squares that can be reached in a single move on separate lines.

Sample Input

```
K 5 5
```

Sample Output

```
8
(4, 4)
(4, 5)
(4, 6)
(5, 4)
(5, 6)
(6, 4)
(6, 5)
(6, 6)
```

```python
size = 8


def traverse(xd, yd, x, y):
    moves = []
    while True:
        x += xd
        y += yd
        if 1 <= x <= size and 1 <= y <= size:
            moves.append((x, y))
        else:
            break
    return moves


def king(x, y):
    moves = []
    # right side
    if x < size:
        moves.append((x + 1, y))
        if y < size:
            moves.append((x + 1, y + 1))
        if y > 1:
            moves.append((x + 1, y - 1))
    # bottom
    if y > 1:
        moves.append((x, y - 1))
        if x > 1:
            moves.append((x - 1, y - 1))

    # left
    if x > 1:
        moves.append((x - 1, y))
        if y < size:
            moves.append((x - 1, y + 1))
    # top
    if y < size:
        moves.append((x, y + 1))

    return moves


def queen(x, y):
    return bishop(x, y) + rook(x, y)


def bishop(x, y):
    moves = []

    moves += traverse(1, -1, x, y)
    moves += traverse(-1, 1, x, y)
    moves += traverse(-1, -1, x, y)
    moves += traverse(1, 1, x, y)

    return moves


def rook(x, y):
    moves = []

    moves += traverse(0, -1, x, y)
    moves += traverse(-1, 0, x, y)
    moves += traverse(0, 1, x, y)
    moves += traverse(1, 0, x, y)
    return moves


def knight(x, y):
    moves = []
    if x - 2 > 0:
        if y + 1 <= size:
            moves.append((x - 2, y + 1))
        if y - 1 > 0:
            moves.append((x - 2, y - 1))
    if x + 2 <= size:
        if y + 1 <= size:
            moves.append((x + 2, y + 1))
        if y - 1 > 0:
            moves.append((x + 2, y - 1))

    if y - 2 > 0:
        if x + 1 <= size:
            moves.append((x + 1, y - 2))
        if x - 1 > 0:
            moves.append((x - 1, y - 2))
    if y + 2 <= size:
        if x + 1 <= size:
            moves.append((x + 1, y + 2))
        if x - 1 > 0:
            moves.append((x - 1, y + 2))
    return moves


def output(moves):
    moves.sort()
    for move in moves:
        print(move)

s = input().split(" ")
if s[0] == 'K':
    result = king(int(s[1]), int(s[2]))
    print(len(result))
    output(result)
elif s[0] == 'Q':
    result = queen(int(s[1]), int(s[2]))
    print(len(result))
    output(result)
elif s[0] == 'B':
    result = bishop(int(s[1]), int(s[2]))
    print(len(result))
    output(result)
elif s[0] == 'R':
    result = rook(int(s[1]), int(s[2]))
    print(len(result))
    output(result)
elif s[0] == 'N':
    result = knight(int(s[1]), int(s[2]))
    print(len(result))
    output(result)
```

A5 Problem Statement

Soccer Tournament

In a 4 team round-robin tournament, each team plays all the other teams. There will be six games. A team's score is determined by 3 points for each win and 1 point for ties. Losses are zero points.

Given the largest score and two other scores, determine the remaining team's score. If there are multiple possible answers, give only one. If there are no possible answer, indicate that.

Example 1:

Largest score: 9
Additional score: 6
Additional score: 1

Remaining team's score: 1

Example 2:

Largest score: 7
Additional score: 3
Additional score: 3

Remaining team's score: 4

Example 3:

Largest score: 6
Additional score: 4
Additional score: 4

Remaining team's score: 2 or 3

Example 4:

Largest score: 7
Additional score: 6
Additional score: 6

Remaining team's score: not possible

A5 Solution
Soccer Scores

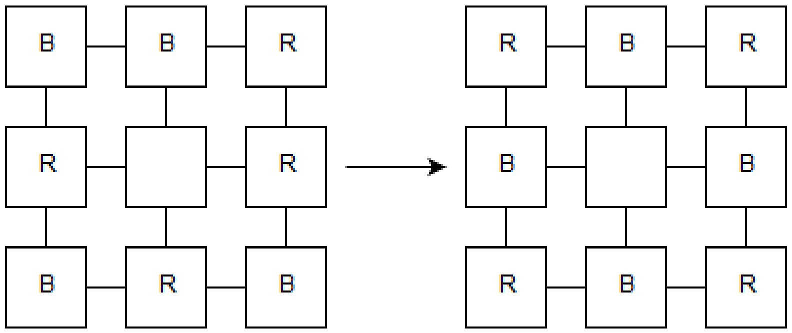
```
int main()
{

       int S1, S2, S3, Sum;
       float NumWin, NumLoss, NumTie, S4,L,W,T;
       L = 0; W = 0; T = 0;
       std::cout << "\nenter high score: ";
       std::cin >> S1;
       std::cout << "\nenter another score: ";
       std::cin >> S2;
       std::cout << "\nenter another score: ";
       std::cin >> S3;
       Sum = S1 + S2 + S3;

       if (S1 == 9) { NumWin = 3; NumLoss = 0; NumTie = 0; }
       if (S1 == 7) { NumWin = 2; NumLoss = 0; NumTie = 0.5; }
       if (S1 == 6) { NumWin = 2; NumLoss = 1; NumTie = 0; }
       if (S1 == 5) { NumWin = 1; NumLoss = 0; NumTie = 1; }
       if (S1 == 3) { NumWin = 0; NumLoss = 0; NumTie = 1.5; }
       if (S2 == 7) { NumWin = NumWin + 2;  NumTie = NumTie + 0.5; }
       if (S2 == 6) { NumWin = NumWin + 2; NumLoss++; }
       if (S2 == 5) { NumWin++; NumTie = NumTie + 1; }
       if (S2 == 4) { NumWin++; NumLoss++; NumTie = NumTie + 0.5; }
       if (S2 == 3 && S1 > 3) { NumWin++; NumLoss++; NumLoss++; }
       if (S2 == 3 && S1 == 3) { NumTie = NumTie + 1.5; }
       if (S2 == 2) { NumTie = NumTie + 1.0; NumLoss++; }
       if (S2 == 1) { NumLoss = NumLoss + 2; }
       if (S3 == 7) { NumWin = NumWin + 2;  NumTie = NumTie + 0.5; }
       if (S3 == 6) { NumWin = NumWin + 2; NumLoss++; }
       if (S3 == 5) { NumWin++; NumTie = NumTie + 1; }
       if (S3 == 4) { NumWin++; NumLoss++; NumTie = 0.5; }
       if (S3 == 3 && S1 > 3) { NumWin++; NumLoss++; NumLoss++; }
       if (S3 == 3 && S1 == 3) { NumTie = NumTie + 1.5; }
       if (S3 == 2) { NumLoss++; NumTie = NumTie + 1.0; }
       if (S3 == 1) { NumLoss = NumLoss + 2; NumTie = NumTie + 0.5; }
//     std::cout << "\nNW " << NumWin << " NL " << NumLoss << " NT " << NumTie;
       if (NumWin == 6) { S4 = 0; std::wcout << "\nAnswer: " << S4; }
       T = 2*((6 - NumWin) - NumTie);
       if (NumWin > NumLoss) { L = NumWin - NumLoss; }
       if (NumLoss > NumWin) { W = NumLoss - NumWin; }
       if (NumLoss == NumWin && NumWin == 4) { W = 1; T = 1; }
       S4 = 3 * W + T;
       std::cout << "\nL " << L << " W " << W << " T " << T;
       std::cout << "\nAnswer S4: " << S4;
//     std::cout << "\nCalc total: " << S1 + S2 + S3 + S4;
//     std::cout << "\nPossible Total: " << 18.0 - (.5*T + NumTie);
       return 0;
}
```

A6 Problem Statement
Sliding Puzzle

Sliding puzzles are fun brain teasers where you must slide tiles on a board to achieve a certain configuration. For this problem, you have a 3 x 3 tiled board with 4 **black (B)** tiles, 4 **red (R)** tiles, and one **empty** square. The winning scenario is when you have alternating red-black or black-red tiles around the edge of the board and the empty square in the center. Colored tiles can only be moved into an empty square. Likewise, tiles can only move up, left, down, or right and cannot skip over a colored tile. Write a program to solve a sliding puzzle based on the initial board configuration received as input. The program should output the state of the board after each move, and after the puzzle is solved, it should output the number of moves that was made (when a tile is moved, that counts as one move). The center square will always initially be the empty square. You may assume that the board configuration that is inputted is solvable. Your algorithm does not have to make the minimal number of moves. As such, your number/order of moves may differ from the sample output. The image below shows an initial configuration and the final solved configuration.



Input Format

Input will be given using 8 capital letters, **R** representing a red tile and **B** representing a black tile. There will always be four black and four red tiles. The input is one continuous string that is not separated by any delimiter. The first letter of the input is the top left tile; the following letters are the tiles in the clockwise direction. For example, “BBRRBRBR” is the initial tile configuration for the image above.

Output Format

After each move, the current state of the board should be pretty printed (see below for example). The colored tiles should be printed using R and B, and the empty square should be printed using an underscore. Once the puzzle is solved, the number of moves made should be printed.

Sample Input

BBRRBRBR

Sample Output

```
B _ R
R B R
B R B

_ B R
R B R
B R B

R B R
_ B R
B R B

R B R
B B R
_ R B

R B R
B B R
R _ B

R B R
B B R
R B _

R B R
B B _
R B R

R B R
B _ B
R B R

8
```

```python
tiles = list(input())
blank = "_"
slot = blank
moves = 0


def slide_in(i):
    global slot
    global moves
    if slot == blank and i % 2 == 1:
        slot = tiles[i]
        tiles[i] = blank
        moves += 1
        return True
    return False


def slide_out(i):
    global slot
    global moves
    if tiles[i] == blank and i % 2 == 1 and slot != blank:
        tiles[i] = slot
        slot = blank
        moves += 1
        return True
    return False


def rotate_left():
    i = tiles.index("_")

    global moves
    if i > -1:
        if i == len(tiles) - 1:
            tiles[i] = tiles[0]
            tiles[0] = blank
        else:
            tiles[i] = tiles[i + 1]
            tiles[i + 1] = blank
        moves += 1
        return True
    return False


def rotate_right():
    i = tiles.index("_")

    global moves
    if i > -1:
        tiles[i] = tiles[i - 1]
        tiles[i - 1] = blank
        moves += 1
        return True
    return False


def check_win():
    alt = tiles[0]
    for c in tiles[1:]:
        if c == blank:
            continue
        if c == alt:
            return False
        alt = c
    return True
```

```
def print_board():
    print()
    print(str.join(" ", tiles[:3]))
    print("{} {} {}".format(tiles[7], slot, tiles[3]))
    temp = tiles[4:7]
    temp.reverse()
    print(str.join(" ", temp))



"""
0-1-2
7-_-3
6-5-4
"""
while not check_win():
    moved = False
    if slot == blank:
        if tiles[1] == tiles[2] or tiles[1] == tiles[0]:
            moved = slide_in(1)
        elif tiles[3] == tiles[2] or tiles[3] == tiles[4]:
            moved = slide_in(3)
        elif tiles[5] == tiles[4] or tiles[5] == tiles[6]:
            moved = slide_in(5)
        elif tiles[7] == tiles[0] or tiles[7] == tiles[6]:
            moved = slide_in(7)
    else:
        if tiles[1] == blank and tiles[0] != slot and tiles[2] != slot:
            moved = slide_out(1)
        elif tiles[3] == blank and tiles[2] != slot and tiles[4] != slot:
            moved = slide_out(3)
        elif tiles[5] == blank and tiles[4] != slot and tiles[6] != slot:
            moved = slide_out(5)
        elif tiles[7] == blank and tiles[6] != slot and tiles[0] != slot:
            moved = slide_out(7)
    if not moved:
        # the rotate left function is not used since we don’t worry about the program being op
timal.
        # for the algorithm to make minimal moves, the logic must be adjusted to figure out
        # if a right or left rotation is optimal.  This decision was made to make the problem
        # more appropriate for the grade level.
        rotate_right()
    print_board()

print()
print(moves)
```