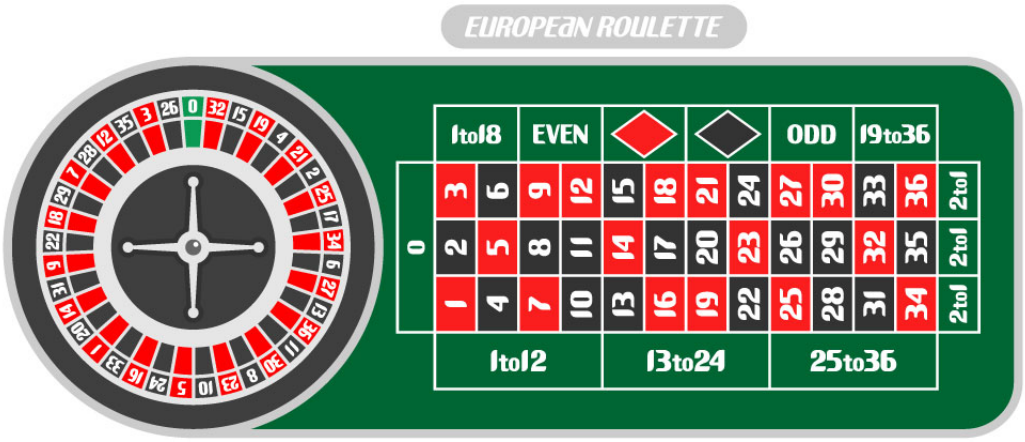
Lab session 3: Programming Fundamentals

Note: In this 3rd lab session you are only allowed to make use of constructs that were taught in the first three weeks of the course.

Problem 1: Martingale Red

Roulette is a casino game named after the French word meaning 'little wheel'. In each play of roulette, a roulette wheel spins in one direction and a ball in the opposite direction. The ball gradually loses momentum, and eventually falls into one of several coloured and numbered pockets along the edge of the wheel.

Players can make several bets before spinning the wheel. One of those bets is based on the colour of the number where the ball lands. This colour can be red or black. If a player bets some amount of money, say n euros, on the colour red and the ball lands in the pocket of a red coloured number, then the player receives $2n$ euros. In other words he wins n euros. If the ball lands in a black coloured pocket, or in the pocket with the number zero (which is the only pocket coloured green), then the player loses the bet (and thus the n euros). The colours of the numbers can be found in the following figure.



The *Martingale Red System* is a common roulette strategy. In this strategy the player always bets on the colour red. After a loss the player doubles the next bet. After a win, the player starts over with the initial bet amount. This way, in theory, after each winning bet the player always has a win of one unit.

For example, let the initial bet be 1 euro, and there are 4 consecutive draws of a black number. Hence, the player has lost $1 + 2 + 4 + 8 = 15$ euros. Then, according to the strategy, in the next round the player makes a bet of 16 euros. If the player wins the next round, then the total win is 1 euro (a loss of 15 euros followed by a win of 16 euros).

Of course, the system is not flawless. The problem is that doubling the bet on each loss requires an unlimited amount of money. In practice, players have a finite amount of money. We say that a player is *bust* if he has not enough money to make the next bet. So, if the next bet would be 32 euros, while the player only has 10 euros, then we still consider the player bust.

Write a program that reads from the input a starting amount of money (the budget). Next is a series of draws of numbers (spins of the wheel) terminated by the number -1 (note that valid draws are numbers in the range 0 upto 36). Assume that a player plays the Martingale Red system with the initial bet of 1 euro. The output of your program should be the amount of money that the player has after the series of spins, or `BUST` if the player went bust at some point in the game.

Example 1:

input:
```
10
1 36 -1
```
output:
```
12
```

Example 2:

input:
```
10
2 17 -1
```
output:
```
7
```

Example 3:

input:
```
10
2 17 35 -1
```
output:
```
BUST
```

Problem 2: Last n Digits

In the lecture we discussed how to compute (a^e) **mod** m for large values of e without overflow. For example, $2^{100} = 1267650600228229401496703205376$ which clearly does not fit in an `int` or `long`. However, (2^{100}) **mod** $100 = 76$, which clearly fits in an `int`.

Write a program that reads from the input three integers a, e, and n (where $0 \leq a < 2^{31}$, $0 \leq e < 2^{31}$, and $1 \leq n \leq 4$ and outputs the last n digits of a^e. Note that the last 3 digits of a number like 4096 are 096 and not 96.

Example 1:

input:
```
2 100 2
```
output:
```
76
```

Example 2:

input:
```
5 3 3
```
output:
```
125
```

Example 3:

input:
```
5 10 3
```
output:
```
625
```

Problem 3: Super Primes

We can repeatedly reduce a positive integer number by summing its digits. We stop with this process when we reached a number less than 10. For example, the number 1994373911 can be reduced in 3 steps: $1994373911 \rightarrow 47 \rightarrow 11 \rightarrow 2$.

As you know, an integer n is a prime number if and only if $n > 1$ and n has exactly two divisors: 1 and n itself. We call a number n a *super prime* if n is prime, and each step in the reduction process also yields a prime.

As an example, 1994373911 is prime and so are 47, 11, and 2. Therefore, 1994373911 is a super prime. The number 997 (which is prime) is not a superprime, since $9 + 9 + 7 = 25$ is not prime.

Write a program that reads from the input two `int`s a and b (where $a < b$) and outputs the number of super primes n with $a \leq n < b$.

Example 1:

input:

```
1 100
```

output:

```
12
```

Example 2:

input:

```
1 1000
```

output:

```
41
```

Example 3:

input:

```
1 10000
```

output:

```
242
```

Problem 4: Palindromic Proth Primes

An integer N of the form $N = k \times 2^n + 1$ where k and n are positive integers, k is odd and $k < 2^n$ is called a *Proth number*. A Proth number which is prime is called a *Proth prime*. The first five Proth primes are 3, 5, 13, 17, and 41.

An integer is called *palindromic* if reversing its digits yields the same number. For example, 12321 is palindromic.

Write a program that accepts on its input a positive `int` and determines whether this number is a Proth number. If it is not, then it should output this (see example 1). If it is, then it should output whether the number is a plain Proth number, a palindromic Proth number, a Proth prime, or a palindromic Proth prime (examples 2-5). The output should match the exact format of the following examples.

Example 1:

input:

```
42
```

output:

```
42 is not a Proth number.
```

Example 2:

input:

```
25
```

output:

```
25 is a Proth number.
```

Example 3:

input:

```
52225
```

output:

```
52225 is a palindromic Proth number.
```

Example 4:

input:

```
1217
```

output:

```
1217 is a Proth prime.
```

Example 5:

input:

```
929
```

output:

```
929 is a palindromic Proth prime.
```

Problem 5: Resolution Rule

In propositional logic, a *clause* is an expression which is the *disjunction* of a finite collection of literals (atoms or their negations).
In this exercise, we impose two extra rules on clauses:

- For the atoms, we only allow the upper case letters A, B, .., Z. Hence, there can be at most 26 atoms in a clause.

- A clause never contains an atom and its negation. This makes sense, because $A \vee \neg A \vee \alpha$ reduces to **true** for any expression α.

An example of a valid clause is:

$$A \vee B \vee \neg P \vee \neg C \vee \neg Q \vee E$$

We represent this clause with the following ASCII notation (where ~ is used to denote negation): `[A,B,~P,~C,~Q,E]`.

The *resolution rule* is an inference rule that produces a new clause implied by two clauses containing *complementary* literals. Two literals are said to be complements if one is the negation of the other. The resolution rule is given by the following formula ((where α and β are propositional expressions):

$$\frac{A \vee \alpha, \quad \neg A \vee \beta}{\alpha \vee \beta}$$

The concluding expression $\alpha \vee \beta$ is called the *resolvent*.

As a concrete example, from the clauses $A \vee \neg B \vee C$ and $A \vee P \vee B \vee \neg D$ we produce $A \vee C \vee A \vee P \vee \neg D$ which reduces to (eliminating the duplicate occurence of A) the resolvent $A \vee C \vee P \vee \neg D$.

Write a program that reads from the input the ASCII representation of two clauses. The output of the program should be the ASCII representation of the resolvent of the two clauses, or `NO RESOLVENT` if the two clauses do not contain complementary literals. If the resolvent exists, then the ASCII representation of the resolvent must be sorted in alphabetical order. Note that you may assume that the two input clauses are chosen such that they produce at most one resolvent (in other words, there is at most one pair of complementary literals).

Example 1:

input:
`[A,~B,C][A,P,B,~D]`
output:
`[A,C,~D,P]`

Example 2:

input:
`[A][~A]`
output:
`[]`

Example 3:

input:
`[A,B][C,~D]`
output:
`NO RESOLVENT`