PROSZĘ UWAŻNIE CZYTAĆ POLECENIA I STOSOWAĆ SIĘ DO ZALECEŃ!
ZADANIA MOŻNA ROBIĆ W DOWOLNEJ KOLEJNOŚCI.

Zadanie 0

Liczbą pierwszą nazywamy liczbę naturalną większą od 1, która ma dokładnie dwa dzielniki naturalne: 1 i samą siebie. Napisz program, który prosi użytkownika o podanie liczby, sprawdza czy ta liczba jest pierwsza i wyświetla stosowny komunikat.
Przykład działania:

Podaj liczbę:
-2
Podana liczba nie jest pierwsza.

Podaj liczbę:
2
Podana liczba jest pierwsza.
Uwaga: Nie musisz sprawdzać, czy liczba jest całkowita, możesz to założyć.
Podpowiedź: Użyj operatora reszty z dzielenia (modulo).

Zadanie 1

Napisz program, który wczytuje w pętli ciąg liczb całkowitych aż do momentu, gdy użytkownik poda liczbę 0, która jest tylko sygnałem końca danych i nie jest dalej brana pod uwagę. Następnie program wypisuje wartości najmniejszego i największego elemenentu wczytanego ciągu oraz liczbę wystąpień tych wartości w całym ciągu. Na przykład dla ciągu (-9, 2, 11, 10, 11, -9, -9, -8, 0) program powinien wypisać:

Min = -9 3 razy
Max = 11 2 razy

Uwaga: Nie wolno stosować tablic ani innych kolekcji. Nie trzeba sprawdzać, czy podane liczby są całkowite – można to założyć. Można również założyć, że podawane liczby nie są bardzo duże albo bardzo małe, np. w przedziale $[-10^6, 10^6]$.
Podpowiedź: Użyj pętli, która nie wymaga, żeby z góry znać ilość iteracji. Zacznij od zrobienia jednego wariantu (min/max), a drugi zrób w analogiczny sposób.

Zadanie 2

Napisz program wyglądający tak:

```
#include <iostream>

void rekur()
{
        // TYLKO TUTAJ COS DODAJEMY
```

```
}

int main()
{
        std::cout << "Podaj liczby" << std::endl;
        rekur();
        return 0;
}
```

Program powinien wczytywać liczby całkowite z konsoli aż użytkownik poda 0. Wtedy wszystkie wczytane liczby (bez kończącego zera) powinny być wypisane w odwrotnej kolejności niż były podawane.
Przykład:

Podaj liczby:
5 13 -4 9 3 0
3 9 -4 13 5

Uwaga: Uzupełnij **tylko** definicję funkcji rekurencyjnej rekur. Nie zmieniaj i nie dodawaj niczego innego. Nie używaj tablic, kolekcji ani pętli. Załóż, że wpisywane są liczby całkowite w jednej linii.
Podpowiedź: Funkcja rekurencyjna woła samą siebie i "czeka" na wynik tego wywołania. Oznacza to, że wywołując funkcję rekurencyjną tworzymy coraz głębsze "poziomy". Jako pierwszy wykona się najgłębszy poziom, później poziom od niego wyższy, później kolejny aż do pierwszego. Jeżeli w ciele funkcji rekurencyjnej umieścimy instrukcje **po** wywołaniu rekurencyjnym (samej siebie) to te instrukcje zostaną wykonane dopiero po zakończeniu rekurencji, czyli po wyjściu z niższego poziomu.

Zadanie 3

Największy wspólny dzielnik NWD – dla danych dwóch (lub więcej) liczb całkowitych największa liczba naturalna dzieląca każdą z nich.
Napisz program, który poprosi użytkownika o wpisanie dwóch liczb całkowitych dodatnich a i b (nie musisz sprawdzać czy są całkowite, ale jeśli są niedodatnie to wypisz błąd i zakończ działanie programu). Nastepnie program wykona poniższy algorytm w celu znalezienia dla nich NWD:

1. oblicz c jako resztę z dzielenia a przez b
2. zastąp a liczbą b, a następnie b liczbą c
3. jeśli wartość b wynosi 0, to a jest szukaną wartością NWD, w przeciwnym wypadku przedź do kroku 1

Przykład działania:

Podaj dwie liczby całkowite dodatnie
5 0
Złe dane! Kończę pracę!

Podaj dwie liczby całkowite dodatnie
5 13
NWD = 1

Podaj dwie liczby całkowite dodatnie
80 60
NWD = 20

Uwaga: Proszę trzymać się algorytmu, nie ma potrzeby używania tablic, kolekcji czy matematycznych funkcji z bibliotek. Można to zrobić rekurencyjnie, ale nie jest wymagane.

Zadanie 4

Napisz i przetestuj nastepujący zestaw funkcji **rekurencyjnych** (w ich treści nie może być żadnych pętli!)

- `int gcdRec(int a, int b);`

  – zwraca NWD dwóch liczb całkowitych dodatnich.

- `int sumDigits( int n );`

  – zwraca sumę (dziesiętnych) cyfr podanej liczby naturalnej
  (e.g. $34 \rightarrow 7$)

- `int numDigits( int n)`

  – zwraca liczbę (dziesiętnych) cyfr podanej liczby naturalnej (e.g. $34 \rightarrow 2$).

- `void printOddEven(int n);`

  – drukuje w jednej linii:

  – dla n parzystego: wszystkie liczby parzyste od 2 do n.

  – dla n nieparzystego: wszystkie liczby nieparzyste od 1 do n.

  `void hailstone(int n);`

  – drukuje w jednej linii wszystkie liczby ciągu Collatza (patrz niżej) od n aż do 1.

Ciąg Collatza, znany też jako ciąg Ulama lub "gradowy" (ang. hailstone), to ciąg dla którego pierwszy wyraz a_0 jest dodatnią liczbą całkowitą, a kolejne wyrazy są wyliczane ze wzoru

$$a_0 = \begin{cases} a_n/2 \text{ jeśli } a_n \text{ jest parzyste} \\ 3a_n + 1 \text{ jeśli } a_n \text{ jest nieparzyste} \end{cases}$$

Hipoteza Collatza (wciąż nieudowodniona) mówi, że niezależnie od wartości pierwszego elementu a_0, po skończonej liczbie kroków ciąg osiągnie wartość 1 (i potem będzie już cyklicznie przybierał wartości $1 \to 4 \to 2 \to 1...$). Następujący program, po zdefiniowaniu wszystkich funkcji

```
#include <iostream>
int gcdRec(int a, int b);
int sumDigits(int n);
int numDigits(int n);
void printOddEven(int n);
void hailstone(int n);

int main()
{
        std::cout << "gcdRec(12, 42) = " << gcdRec(12, 42) << std::endl
                                        << "gcdRec(12, 25) = " << gcdRec(12, 25)

        std::cout << "sumDigits(123) = " << sumDigits(123) << std::endl
                                        << "sumDigits(971) = " << sumDigits(971)

        std::cout << "numDigits(12345) = " << numDigits(12345) << std::endl
                                        << "numDigits(971) = " << numDigits(971)

        std::cout << "printOddEven(14): ";
        printOddEven(15);
        std::cout << std::endl;
        std::cout << "printOddEven(14): ";
        printOddEven(14);
        std::cout << std::endl;
        std::cout << "hailstone(13): ";
        hailstone(13);
        std::cout << std::endl;

        return 0;
}
```

powinien wypisać:

gcdRec(12, 42) = 6
gcdRec(12, 25) = 1
sumDigits(123) = 6
sumDigits(971) = 17
numDigits(12345) = 5
numdigits(971) = 3
printOddEven(15): 1 3 5 7 9 11 13 15
printOddEven(14): 2 4 6 8 10 12 14
hailstone(13): 13 40 20 10 5 16 8 4 2 1

Uwaga: Wszystkie funkcje mają być rekurencyjne. Nie trzeba sprawdzać poprawności danych. Nie trzeba pobierać danych z konsoli.