Zadanie 1 (10 pkt. na pracowni, później 5 pkt.).

Zaprojektuj prostą bibliotekę funkcji matematycznych z naciskiem na dobrą organizację kodu.

Wymagane pliki projektu: `main.c`, `math_ops.h`, `math_ops.c`, `number_utils.h`, `number_utils.c`.

W pliku `math_ops.h` zadeklaruj funkcje: `int factorial(int n)`, `double power(double base, int exponent)`, `int is_prime(int number)`.

W pliku `number_utils.h` zadeklaruj funkcje: `int count_digits(int number)`, `int reverse_number(int number)`, `int digit_sum(int number)`.

Powyższe funkcje zaimplementuj odpowiednio w plikach `math_ops.c` i `number_utils.c`. Jeśli będą potrzebne funkcje pomocnicze, inne niż wyeksponowane w ww. plikach nagłówkowych, zadeklaruj je jako `static`. Dla funkcji `factorial` zadbaj o obsługę błędów, w tym także *integer overflow* (użyj wartości z `limits.h`) – co funkcja powinna w takiej sytuacji zwracać?

W pliku `main.c` zaimplementuj funkcję `main()` demonstrującą użycie funkcji wyeksponowanych w plikach nagłówkowych (każdej przynajmniej raz). Na standardowe wyjście wydrukuj nie tylko same wyniki obliczeń, ale też komentarze mówiące, co zostało obliczone.

Zadbaj o komentarze dokumentacyjne, zawierające wyczerpującą informację o tym, jakie wartości argumentów są poprawne, i co zwraca dana funkcja. Zadbaj o to, by kod był czytelny, zrozumiały, i poprawnie podzielony na moduły (w tym zadaniu jest to ważniejsze niż sama efektywność – ale ona też nie jest zupełnie bez znaczenia). Jak zawsze, warto zadawać pytania osobie prowadzącej zajęcia!

Zadanie 2 (10 pkt.). Sumę *n* kolejnych liczb naturalnych nazywamy *n*-tą liczbą trójkątną (bo jest to liczba kulek ciasno upakowanych w trójkąt równoboczny o boku *n*). Przykładowo, siódma liczba trójkątna to 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28.

Dodatkowo, okazuje się ona być najwcześniejszą liczbą trójkątną, która ma więcej niż pięć dzielników (bo ma ich sześć: 1, 2, 4, 7, 14 i 28); wcześniejsze liczby trójkątne wraz z ich dzielnikami to kolejno:

- 1 podzielne przez 1;
- 3 podzielne przez 1 i 3;
- 6 podzielne przez 1, 2, 3 i 6;
- 10 podzielne przez 1, 2, 5 i 10;
- 15 podzielne przez 1, 3, 5 i 15;

- 21 podzielne przez 1, 3, 7 i 21.

Napisz program, który wydrukuje indeks i wartość najwcześniejszej liczby trójkątnej, która ma więcej niż *N* dzielników; *N* odczytaj z jedynego obowiązkowego argumentu wywołania. Wykorzystaj do tego funkcję biblioteczną *strtoul.h* – w celu skontrolowania poprawności podaj jej nietrywialny drugi argument po to, żeby móc sprawdzić, czy konwersja "dojechała" do końca napisu.

Przetestuj swój program dla argumentów 100, 200, 300.

Zadanie 3. W bardzo odległym królestwie znajduje się *n* miast położonych wzdłuż wielkiej drogi okalającej całą krainę. Miasto *i* (dla dowolnego *i*) sąsiaduje więc z miastami *i-1* i *i+1*, przy czym miasta 1 i *n* również są sąsiednie. W każdym mieście regularnie odbywają się targi, konkretniej w mieście *i* co dokładnie *j(i)* dni jest organizowany targ.

Ponieważ produkcja wszelkich dóbr znacząco wzrosła w ostatnich latach, władze lokalne postanowiły wprowadzić także mega-targi. Każde miasto chce ściągnąć do siebie kupców również z sąsiednich miast, ustalono więc że miasto *i* będzie regularnie organizować mega-targ co dokładnie *m(i)* dni, gdzie *m(i)* jest najmniejszą wspólną wielokrotnością liczb *j(i-1), j(i), j(i+1)*, co ułatwi wizyty zainteresowanym sąsiednim kupcom.

Król, zanim udał się na polowanie, nakazał swoim rachmistrzom wyliczyć dwie wartości. Po pierwsze, zapytał jakie jest najmniejsze *k* takie, że po *k* dniach w całym królestwie odbędzie się już przynajmniej *r* mega-targów; przyjmujemy tutaj, że targi zostały odpowiednio zsynchronizowane, a zatem w mieście *i* mega-targi odbędą się dokładnie w dniach numer *m(i)*, *2m(i)*, *3m(i)*, i tak dalej. Po drugie, król podejrzewa, że może nie wszystkie miasta zasługują na tak wspaniałe wydarzenia komercyjne. Spytał więc także, ile najmniej (dowolnie wybranych) miast musi organizować mega-targi, aby do dnia *p* odbyło się ich w całym królestwie przynajmniej *q*, podczas gdy we wszystkich nie wybranych miastach mega-targi są zawieszone?

Wejście

W pierwszym wierszu wejścia znajduje się naturalna liczba $n \leq 100000$, w drugiej zaś *n* liczb naturalnych, nie większych niż milion; są to kolejne wartości *j(i)*. W ostatniej linii wejścia znajdują się liczby *r, p, q*, nie większe niż miliard. Jest gwarantowane, że odpowiedź na pierwsze pytanie króla jest nie większa niż miliard miliardów, a także że odpowiedź na drugie pytanie zawsze istnieje (jest więc równa co najwyżej *n*).

Wyjście

Na standardowe wyjście należy podać dwie oddzielone spacją liczby naturalne, będące odpowiedziami na pytania króla.

Przykłady

Przykład A

Wejście

```
3
2 5 4
10 50 3
```

Wyjście

```
80 2
```

Wszystkie miasta są sąsiednie, zatem po prostu w każdym mega-targ jest organizowany co dokładnie 20 dni. Musi więc minąć 80 dni, by sumarycznie odbyło się 10 mega-targów (tak naprawdę 12). Ponadto, jeśli chcemy by do dnia 50 odbyły się przynajmniej 3-mega targi, muszą być one organizowane w przynajmniej dwóch miastach, jedno nie wystarczy.

Przykład B

Wejście

```
4
1 2 3 4
3 500 100
```

Wyjście

```
8 1
```

Mega-targi są organizowane odpowiednio co 4, 6, 12 i 12 dni. Musi minąć 8 dni, by odbyły się trzy tego rodzaju wydarzenia. Jeśli żądamy, by do dnia 500 odbyło się 100 mega-targów, wystarczy nam jedno miasto (numer 1).

Przykład C

Wejście

```
4
4 6 8 9
8 80 6
```

Wyjście

```
96 3
```

Mega-targi są organizowane odpowiednio co 36, 24, 72 i 72 dni.

Uwagi

W tym zadaniu w sprawdzaczce została włączona flaga kompilacji *-O2* skutkująca optymalizacją kodu wynikowego. W połączeniu z innymi naszymi flagami kompilacji, spowoduje to pojawienie się pewnych nowych ostrzeżeń ze strony kompilatora. W związku z tym:

- testując rozwiązanie na swoich komputerach również warto używać tej flagi kompilacji, żeby móc zawczasu pozbyć się tych błędów;
- prawie na pewno zobaczą Państwo błąd związany z gubieniem wartości wynikowej *scanf* – o ile w bardziej użytkowych programach warto robić z tym coś mądrzejszego (np. przy użyciu instrukcji warunkowej odpowiednio obsłużyć pojawienie się niepoprawnego wejścia) tak na sprawdzaczce poprawność

wejścia jest gwarantowana i wystarczy przypisać tę wartość do jakiejś jeszcze nieużywanej zmiennej, którą później możemy czymś nadpisać.