Trójki pitagorejskie

#twierdzenie_Pitagorasa
#preprocesor #dyrektywa
#makrodefinicja #lvalue

Jeśli trzy liczby naturalne dodatnie a, b oraz c spełniają warunek:

$$a^2 + b^2 = c^2,$$

wtedy nazywane są *trójką pitagorejską* lub *liczbami pitagorejskimi*. Zgodnie z twierdzeniem Pitagorasa,* liczby te mogą stanowić długości boków trójkąta prostokątnego. Najbardziej znaną trójką pitagorejską jest $(3, 4, 5)$. Trójkąt o takich długościach boków zwany jest *trójkątem egipskim* i znany był już w starożytności.

Naszym zadaniem jest napisanie procedury,† która znajduje wszystkie trójki pitagorejskie z zadanego zakresu (a, b oraz c nie powinny przekraczać pewnej liczby n). Problem rozwiążemy metodą „brutalnej siły" sprawdzając wszystkie możliwe trójki liczb z danego zakresu.‡ Zauważmy jednak, że możemy bez straty dla ogólności rozwiązania rozważać tylko trójki liczb spełniających następujący warunek:

$$1 \leqslant a < b < c \leqslant n.$$

Liczba a nie może nigdy być równa liczbie b, bo wtedy trójkąt byłby równoramienny. Taki trójkąt (prostokątny) nie może mieć całkowitych długości boków, bo stosunek długości jego przyprostokątnej do przeciwprostokątnej jest liczbą niewymierną (dlaczego?). Ponadto na przykład trójki $(3, 4, 5)$ i $(4, 3, 5)$ to w końcu te same trójki, więc kolejność liczb w trójce można ustalić zgodnie z powyższym wzorem.

Jaki zakres powinna przebiegać zmienna a? Najmniejszą (choć mało spodziewaną) wartością powinno być 1, natomiast maksymalna wartość to $n-2$, bo „nad" a muszą jeszcze zmieścić się b i c. Z kolei zmienna b powinna zacząć od wartości $a+1$ i dojść do wartości $n-1$. Zmienna c zaczyna od wartości $b+1$ i kończy na wartości n. Oto odpowiednia procedura zawierająca potrójną pętlę:

```
void pythagorean(int n)
{
  for(int a = 1; a <= N - 2; a++)
    for(int b = a + 1; b <= N - 1; b++)
      for(int c = b + 1; c <= N; c++)
        if(a * a + b * b == c * c)
          cout << a << ' ' << b << ' ' << c << endl;
}
```

*Pitagoras był Grekiem, żył na przełomie VI i V w. p.n.e.

†Tym mianem będziemy określać funkcję niezwracającą żadnej wartości.

‡Istnieją wzory na liczby pitagorejskie – najbardziej znany pochodzi od Euklidesa.

Nazwę funkcji wymawiaj: *pajtagorijen*, z akcentem na *i*.

Dla przykładowej wartości $n = 50$ otrzymujemy rezultat:

```
3 4 5
5 12 13
6 8 10
7 24 25
8 15 17
9 12 15
9 40 41
10 24 26
12 16 20
12 35 37
14 48 50
15 20 25
15 36 39
16 30 34
18 24 30
20 21 29
21 28 35
24 32 40
27 36 45
30 40 50
```

Widać jednak, że trójka $(3, 4, 5)$ występuje kilka razy, gdyż trójki $(6, 8, 10)$ czy $(9, 12, 15)$ można z niej uzyskać mnożąc ją przez odpowiedni czynnik. Zatem trójki $(6, 8, 10)$ i $(9, 12, 15)$ są *wtórne*, zaś trójka $(3, 4, 5)$ jest *pierwotna*. Co trzeba zmienić w naszej procedurze, aby wypisywała tylko trójki pierwotne? Wystarczy, jeśli liczby a oraz b będą liczbami względnie pierwszymi, czyli $\mathbf{gcd}(a, b)$ powinien wynosić 1 (największy wspólny dzielnik). Tak wygląda ta procedura po poprawieniu:

```
void pythagorean(int n)
{
  for(int a = 1; a <= N - 2; a++)
    for(int b = a + 1; b <= N - 1; b++)
      if(gcd(a, b) == 1)
        for(int c = b + 1; c <= N; c++)
          if(a * a + b * b == c * c)
            cout << a << ' ' << b << ' ' << c << endl;
}
```

Oczywiście w naszym programie musimy umieścić definicję funkcji **gcd**().

Tym razem jako rezultat otrzymujemy nieco krótszą listę trójek:

```
3 4 5
5 12 13
7 24 25
```

```
8 15 17
9 40 41
12 35 37
20 21 29
```

Nieco irytujący jest fakt, że nie mamy gotowego operatora kwadratu liczby całkowitej. Biblioteczna funkcja `pow()` operuje na liczbach typu `double`, a my tutaj lansujemy zasadę, by z nich nie korzystać, no chyba że musimy.

Możemy oczywiście napisać własną funkcję podnoszącą do kwadratu `sqr()` – od angielskiego **square** (wymawiaj: *skłer*), na przykład:

```
long long sqr(long long k)
{
  return k * k;
}
```

Użyliśmy typu `long long`, aby uniknąć problemów przy zbyt dużym argumencie.

Można jednak tak zdefiniować funkcję `sqr()`, aby działała dla dowolnego typu danych (w tym także `double`). Tyle, że nie będzie to w pełnym tego słowa znaczeniu funkcja, ale *makrodefinicja*, popularnie zwana *makrem*.

Kluczem do rozwiązania problemu są tak zwane *dyrektywy preprocesora*, czyli specyficzne instrukcje zaczynające się od symbolu # (ang. **hash**, wymawiaj: *hasz*). Jedną taką instrukcję już znamy: `#include`, która powoduje wstawienie w danym miejscu odpowiedniego pliku nagłówkowego stowarzyszonego z jakąś biblioteką (pakietem funkcji), na przykład `iostream`.

Instrukcje preprocesora są wykonywane przed właściwą kompilacją programu: wpływają one na postać kodu źródłowego, który następnie przetwarzany jest przez kompilator. Daje to naprawdę spore możliwości, oczywiście pod warunkiem, że programista ogarnia to narzędzie.

Jedną z najczęściej używanych dyrektyw jest instrukcja `#define`. Przy jej pomocy definiuje się jakiś identyfikator, który może być wykorzystany w kodzie programu. Na przykład dyrektywa:

```
#define MAX 1000
```

definiuje stałą *MAX*, której wartość wynosi 1000. Przypomina to bardzo deklarację stałej przy pomocy słowa kluczowego `const`:

```
const int MAX = 1000;
```

To w praktyce to samo, ale jednak istnieją pewne różnice. Nie będziemy w to wchodzić, ale generalnie fachowcy zalecają ten drugi sposób deklaracji stałej.

Dyrektywa `#define` może nam się jednak przydać do utworzenia makrodefinicji, czyli takiej jak gdyby minifunkcji. Na przykład obliczanie kwadratu liczby k mogłoby być realizowane przez następujące makro:

```
#define sqr(k) k*k
```

Jednak takie makro nie będzie funkcjonować poprawnie, gdyż preprocesor podmienia w wywołaniu makra tekst w sposób literalny, zatem na przykład:

```
sqr(a+b)
```

zostanie zamienione na:

```
a+b*a+b
```

To na pewno nie jest wyrażenie, o jakie nam chodziło. Zatem w definicji makra musimy dopisać nawiasy otaczające jego argument:

```
#define sqr(k) (k)*(k)
```

Czy to wystarczy? Niestety nie, proszę sobie wyobrazić takie wyrażenie:

```
c/sqr(a+b)
```

które zostanie zamienione na:

```
c/(a+b)*(a+b)
```

No i jest źle, bo takie wyrażenia w C++ są obliczane od lewej do prawej, zatem będzie ono traktowane jako:

$$\frac{c}{a+b} \cdot (a+b)$$

I znów nie o to chodziło. Dopiero dodanie nawiasów otaczających całe wyrażenie wewnątrz makra zapewni poprawność obliczeń w każdej sytuacji:

```
#define sqr(k) ((k)*(k))
```

Jak widać, łatwo jest zrobić fatalny błąd w konstrukcji makra (przy tworzeniu analogicznej funkcji nie byłoby tych problemów). A jednak makra mają swoich zwolenników wśród programistów – głównie ze względu na szybciej wykonywany kod obliczający dane wyrażenie.

Jeśli umieścimy taką definicję makra w kodzie programu przed funkcją `pythagorean()`, wtedy instrukcja warunkowa w niej mogłaby wyglądać tak:

```
if(sqr(a) + sqr(b) == sqr(c))
  . . .
```

¡*Me gusta*! – jak mawia moja piękna znajoma Aña, nauczycielka języka hiszpańskiego.

Przy okazji warto wspomnieć o dość typowym błędzie składniowym, jaki można zrobić w powyższym wyrażeniu logicznym. Co się stanie, jeśli zamiast podwójnego znaku równości (czyli operatora porównania `==`) wpiszemy omyłkowo pojedynczy znak równości (czyli operator przypisania/podstawienia `=`)?

Dostaniemy dziwny komunikat:

```
error: lvalue required as left operand of assignment
```

Assignment (wymawiaj: *esajnment*) to „przypisanie”, **left operand** to „lewa część” (lewa strona instrukcji przypisania), **required** (wymawiaj: *rekłajerd*) to „wymagany”. Ale cóż to jest **lvalue** (wymawiaj: *el-waliu*)?! Otóż tym terminem określa się coś, za co można w jednoznaczny sposób podstawić coś innego i zostanie to przechowane – i to coś powinno stać po lewej stronie instrukcji przypisania (stąd przedrostek **l**). Dajmy przykłady pozytywne:

```
n = 3;     // Zmienna n otrzymuje wartość 3
sum += x;  // Zmienna sum zostaje powiększona o wartość zmiennej x
A[i] = 0;  // Element tablicy A o indeksie i otrzymuje wartość 0
```

Natomiast poniżej mamy przykłady błędnych instrukcji:[§]

```
x + y = z;      // Nie da się podstawić za x + y
a / b = 4 / 3   // Nie da się podstawić za a / b
f(n) = 5;       // Nie da się podstawić za f(n)
```

[§]Proszę pamiętać, że ten termin jest charakterystyczny dla języków wywodzących się z języka C (a więc C++, Java itp.). Istnieją języki o innych regułach składni, na przykład w Pythonie poprawna jest instrukcja „`a, b = 1, 2`” (zmienna a otrzymuje wartość 1, a zmienna b – wartość 2).