Na dwoje kotka wróżyła

#liczby_losowe
#rand #srand #time
#komentarz #// #/*_*/

W wielu problemach algorytmicznych, matematycznych czy przyrodniczych przydaje się możliwość skorzystania z *generatora liczb losowych* (ang. **random number generator**, wymawiaj: *random namber dżenerejtor*). Jest to funkcja (występująca w każdym języku programowania), która za każdym razem zwraca inną wartość, na pozór niezwiązaną z poprzednią. Algorytmy generowania takich liczb są oczywiście ściśle deterministyczne* – opierają się na teorii liczb – i mają specyficzną cechę: są okresowe, to znaczy po pewnym czasie zaczynają powtarzać sekwencję wyników. Jeśli okres jest duży, wtedy nie stanowi to wielkiego problemu, bo wykorzystuje się tylko niewielki podzbiór zbioru liczb zwracanych przez generator.†

W języku C++ generator liczb losowych nazywa się `rand()` i zwraca on liczbę całkowitą ze zbioru $\{0, 1, 2, \ldots, RAND_MAX\}$, choć niekoniecznie każdą. Stała *RAND_MAX* może być inna dla Windows i dla Linuksa,‡ ale generalnie jest to niemała liczba całkowita. Zresztą, co tu dużo gadać, możemy po prostu sprawdzić jej wartość:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  cout << RAND_MAX << '\n';
  return 0;
}
```

U Kocurra wyszło 2147483647 (w systemie Linux). Jeśli u Ciebie wyszło mniej, nie łam się. Nie znaczy to, że jesteś gorszego sortu, po prostu tak wyszło.

Spróbujmy teraz wypisać tuzin przykładowych liczb losowych:

```
#include <bits/stdc++.h>
using namespace std;

int main()
```

*Dlatego ich wyniki zwane są też *liczbami pseudolosowymi*.

†Kocurrowi przyszło kiedyś napisać samodzielnie taki generator z okresem około 10^{20} i tu nieoceniona okazała się książka Ryszarda Zielińskiego *Generatory liczb losowych*, wydana jeszcze wtedy, gdy w Krakowie po Alejach Trzech Wieszczów spacerowały niedźwiedzie.

‡Dokładniej: dla różnych kompilatorów języka C++.

```
{
  for(int i{ }; i < 12; i++)
    cout << rand() << '\n';
  return 0;
}
```

Na ekranie pojawi się sekwencja liczb (być może u Ciebie będzie inna):

```
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421
1025202362
1350490027
```

No fajnie, liczby rzeczywiście wyglądają na przypadkowe – do czasu, kiedy uruchomimy ponownie nasz program i – o zgrozo! – otrzymamy dokładnie taką samą listę liczb. Czyli cała ta losowość to lipa! No tak, pisaliśmy o deterministycznym sposobie generowania tych liczb, więc w jaki sposób tchnąć w nie trochę spontaniczności? Otóż należy spowodować, aby pierwsza (początkowa) z generowanych liczb była wybierana za każdym razem inaczej. A co zmienia się od jednego uruchomienia programu do drugiego? Czas! Zatem tę pierwszą liczbę[§] należy jakoś związać z chwilą uruchomienia programu. Służy do tego funkcja `srand()`, która ustawia początek sekwencji liczb losowych w oparciu o wartość swego argumentu, a w jego roli wystąpi funkcja `time()`. Funkcja ta jest zwyczajowo uruchamiana z argumentem 0, wtedy po prostu zwraca bieżący czas w formie liczby całkowitej.[¶]

Do naszego programu dodamy więc użycie funkcji `srand()` (wymawiaj: *si-maθ*):

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  srand(time(0));
  for(int i{ }; i < 12; i++)
    cout << rand() << '\n';
  return 0;
}
```

Uruchamiając ten program uzyskamy zupełnie inną sekwencję liczb, niż poprzednio, i o to chodziło.

[§]Zwaną **random seed** (wymawiaj: *random sid* z twardym *s* i długim *i*) – słowo **seed** oznacza *ziarno* (nasiono).

[¶]Zwanej **timestamp** (wymawiaj: *tajmstamp*) – słowo **stamp** oznacza tutaj *znak*.

Na cóż jednak mogą przydać się nam takie dziwne liczby? Na przykład do symulacji rzutów kostką do gry (taką zwykłą, sześcienną). Weźmy taką funkcję `dice()` (wymawiaj: *dajs*, to po angielsku znaczy „kostka do gry”), która zwraca liczbę ze zbioru $\{1, 2, 3, 4, 5, 6\}$:

```
int dice()
{
  return 1 + rand() % 6;
}
```

To działa, ponieważ `rand()%6` to liczba naturalna z zakresu od 0 do 5, a powiększona o 1 daje nam zakres taki, jaki generuje kostka do gry.

Teraz możemy wymyślić sobie jakiś problem z dziedziny „kostkologii” i zasymulować przebieg eksperymentu, przez co znajdziemy rozwiązanie.

W typowej grze w kości używa się pięciu kostek i punktuje się rozmaite szczególne układy liczby oczek. Jednym z takich układów jest *full*, czyli wyrzucenie takich samych wartości na trzech kostkach oraz takich samych wartości (ale różnych od tych trzech poprzednich) na dwóch pozostałych kostkach. Jakie jest prawdopodobieństwo wyrzucenia takiego układu? Każdy kumaty maturzysta zdający matematykę w zakresie rozszerzonym policzy to bez komputera, ale my pokusimy się o wykorzystanie liczb losowych. W ten sposób znajdziemy rozwiązanie przybliżone (choć być może bardzo dokładne). Po prostu rzucimy pięcioma kostkami N razy (N będzie dużą liczbą). Jeśli *full* wypadnie nam, dajmy na to, w k rzutach, wtedy szukane prawdopodobieństwo to iloraz k/N. Im więcej rzutów, tym dokładniejszy wynik otrzymamy.[‖]

Po czym ropoznamy, że wypadł *full*? Ano, wystarczy zliczyć, ile wypadło jedynek, ile dwójek, itd. To będzie 6 liczb (liczników) jednoznacznie charakteryzujących rzut 5 kostek. Jeśli wśród tych 6 liczb jest dokładnie jedna wartość 3 i dokładnie jedna wartość 2, to mamy *fulla* (skąd ta pewność?).

Napiszmy funkcję `full()`, która sprawdza, czy wyrzuciliśmy *fulla*. Funkcję `dice()` wywołamy 5-krotnie, bo tyleż mamy kostek. Wyniki rzutów będziemy zliczać przy użyciu tablicy $C[\,]$. Wywołanie funkcji `dice()` powinno zwiększać o 1 komórkę tablicy o takim numerze, jak rezultat funkcji `dice()`, czyli od 1 do 6.

Tablica $C[\,]$ może mieć 6 komórek, ale wtedy numer komórki będzie o jeden mniejszy od wartości, która jej odpowiada. Chyba lepiej zadeklarować ją o oczko większą i po prostu nie używać komórki o indeksie 0.[**]

Następnie należy sprawdzić zawartość tablicy $C[\,]$, czy zawiera wartości 3 i 2 – informację o tym przechowamy w zmiennych *found*3 oraz *found*2 (wymawiaj: *fałnd*, po angielsku: *znaleziono*). Funkcja `full()` mogłaby wyglądać na przykład tak:

[‖]No i mamy tu piękny polski akcent: taka metoda obliczania to jedna z tak zwanych metod Monte Carlo, do rozwoju których bardzo przyczynił się polski matematyk Stanisław Ulam, uczestnik projektu Manhattan (amerykańskiego programu budowy bomby jądrowej).

[**]W ten sposób łatwiej uniknąć wyjątkowo trudnych do wykrycia błędów w konstrukcji programu.

```
bool full()
{
  int C[7] = { };  // nie zapominamy o wyzerowaniu liczników
  for(int i{ 1 }; i <= 5; i++)
    C[dice()]++;
  bool found2{ }, found3{ };  // zmienne otrzymują wartość false
  for(int j{ 1 }; j <= 6; j++)
    if(C[j] == 2)
      found2 = true;
    else if(C[j] == 3)
      found3 = true;
  return found2 && found3;
}
```

No tak, sukces mamy, jeśli któryś z liczników wskaże 3, a któryś 2. *Alles klar*!

Jak ponadto widać, posłużyliśmy się tu komentarzem do kodu źródłowego: wszelki tekst po wystąpieniu znaków `//` (aż do końca wiersza) jest ignorowany przez kompilator. Przydaje się także bardziej rozbudowany komentarz, który może rozciągać się na wiele wierszy:

```
/* cokolwiek
coś jeszcze
coś innego
coś na deser */
```

Możemy wreszcie poskładać cały program (rzut pięcioma kostkami powtórzymy dziesięć milionów razy):

```
#include <bits/stdc++.h>
using namespace std;

int dice()
{
  return 1 + rand() % 6;
}

bool full()
{
  int C[7] = { };  // nie zapominamy o wyzerowaniu liczników
  for(int i{ 1 }; i <= 5; i++)
    C[dice()]++;
  bool found2{ }, found3{ };  // zmienne otrzymują wartość false
  for(int j{ 1 }; j <= 6; j++)
    if(C[j] == 2)
      found2 = true;
    else if(C[j] == 3)
      found3 = true;
  return found2 && found3;
}
```

```
int main()
{
  srand(time(0));
  const int N{ 10000000 };
  int k{ };
  for(int i{ }; i < N; i++)
    if(full()) k++;
  cout << (double)k / N << '\n';
  return 0;
}
```

Jakaś liczba w tym ilorazie (k/N) powinna być typu `double` (u nas jest to licznik), wtedy dostaniemy zwykły wynik z częścią ułamkową.

Nasz program powinien wypisać coś takiego (lub liczbę bardzo zbliżoną do tej):

```
0.0385468
```

Dokładny wynik to:[††]

$$\frac{6 \cdot 5}{6^5} \cdot \binom{5}{3} = \frac{30}{7776} \cdot 10 \approx 0,03858\ldots$$

Chyba może być...

[††] Maturzyści, potwierdzicie?