Imię kota

#kombinatoryka
#permutacje
#szybkie_potęgowanie

W tym podrozdziale zajmiemy się problemem *kombinatorycznym*. Mówiąc w dużym skrócie i uproszczeniu naszym zadaniem będzie wyliczenie, na ile sposobów można coś tam zrobić: tym razem będzie chodziło o prosty problem: mamy daną garstkę klocków Scrabble'a* i ile różnych imion kocich można ułożyć z tych klocków – za każdym razem korzystając ze wszystkich klocków. Uprościmy sobie sprawę: imię kocie to dowolny wyraz ułożony z klocków, niekoniecznie sensowny. No i tradycyjnie ograniczymy się do małych liter alfabetu łacińskiego (we wprowadzanych danych). Czy to będzie trudne? Maturzyści zdający matematykę na poziomie rozszerzonym nie mają problemów z tak prostymi zadaniami, prawda? Ale my napiszemy program, który oblicza tę ilość możliwości. Zaproponujemy Czytelnikowi dość dużą „garstkę" klocków – tak maksymalnie do 100. W związku z tym możemy dostać naprawdę pokaźną liczbę wariantów, zatem zadowolimy się jej resztą z dzielenia przez liczbę $M = 10^9 + 7 = 1000000007$.†

Każdy, kto choć trochę liznął kombinatoryki, stwierdzi że ten problem na milę pachnie *permutacjami*, czyli ustawianiem skończonej ilości elementów w różnej kolejności. A skoro permutacje, to i funkcja silnia się pojawi. Ale po kolei, jeśli wszystkie klocki są różne, to wtedy sprawa jest prosta: dla n klocków mamy $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ możliwości. Dlaczego? Ano, pierwszą literę wybieramy na n sposobów, drugą – na $n-1$ (bo jeden klocek już zabraliśmy), trzecią – na $n-2$, i tak dalej, aż do 1 (ostatni klocek). Wymienione liczby należy przemnożyć przez siebie, bo wybory kolejnych liter są niezależne od siebie (zmniejsza się tylko liczba możliwości wyboru), stąd mamy silnię. Zatem na przykład dla $n = 3$ mamy $3! = 6$ sposobów wybrania imienia. Jeśli mieliśmy, dajmy na to, klocki **a**, **b** oraz **c**, to imiona mogą brzmieć: **abc**, **acb**, **bac**, **bca**, **cab** lub **cba**.‡

Jeśli klocki się powtarzają, wtedy mamy pewien problem. Weźmy dla przykładu zestaw klocków korespondujący ze słowem **abrakadabra**, mamy tutaj łącznie 11 liter: liter **a** mamy pięć, liter **b** oraz **r** po dwie, prócz tego po jednej literze **d** oraz **k**. Możemy sobie ustawić 11 klocków na wszystkie możliwe sposoby i będziemy mieć $11! = 39916800$ słów/imion, ale nie wszystkie będą różne. Kto odróżni, czy litery **b** są użyte w kolejności $\mathbf{b}_1, \mathbf{b}_2$, czy też $\mathbf{b}_2, \mathbf{b}_1$? One są nierozróżnialne! Zatem wynik należy podzielić przez 2, bo ich ewentualna zamiana miejscami nie zmienia wyrazu. Tak trzeba zrobić dla każdej powtarzającej się litery, ale co zrobić dla litery **a**, która występuje aż 5 razy? Jakakolwiek zmiana kolejności wystąpień tej litery (przy zachowaniu kolejności pozostałych liter) daje w wyniku to samo imię, a tych możliwości zamiany kolejności jest właśnie 5! (permutacje).

*O takich klockach była wzmianka w podrozdziale *Anagramy*.

†Jest to liczba pierwsza, która dość często pojawia się w tego typu problemach algorytmicznych.

‡Kto by tak kota nazwał? Choć Kocurro miał kiedyś kotka, który wabił się Richard Parker...

Ostatecznie, liczba *różnych* wyrazów z tego zestawu klocków to:

$$\frac{11!}{5! \cdot 2! \cdot 2!} = 83160.$$

Przy dwójkach też napisaliśmy znak silni, tak dla porządku.

Powoli klaruje się nam algorytm rozwiązania zadania: najpierw wczytujemy zestaw klocków (jako string z literami podanymi w losowej kolejności). Zliczamy, ile jest jakich liter, a potem się zobaczy. Zaczniemy od Bacha, *pardon*, od `main()`-a:

```
int main()
{
  string name;
  cin >> name;
  cout << name_count(name) << endl;
  return 0;
}
```

Funkcja `name_count()` (z ang. *ilość imion*, wymawiaj: *nejm kałnt*) zajmie się całą sprawą. Jej początek może wyglądać tak (trzeba tę funkcję umieścić przez funkcją `main()`):

```
int name_count(string name)
{
  int letters[26] = { };
  for(char c : name)
    letters[c - 'a']++;
  . . .
```

W tablicy *letters*[ ] (z ang. *litery*) przechowamy ilości wystąpień poszczególnych liter.[§] Podobna konstrukcja była zastosowana w podrozdziale *Anagramy*.

No dobra, bazą jest $n!$, gdzie n to długość wprowadzonego ciągu znaków. Ale jak to, u licha zapisać?! W pesymistycznym przypadku, czyli $n = 100$, silnia z tego to bez mała 160 cyfr! Trzeba znaleźć chytry sposób zapisu i tu jest dobra wiadomość, którą mogliśmy wydedukować z podrozdziału *Na drobne kawałki* – trzeba zapisać rozkład na czynniki pierwsze owego gigantycznego wyrażenia. Wiemy, że nie będzie w nim żadnej liczby większej od 100 (dlaczego? – głupie trochę pytanie). A ile jest liczb pierwszych poniżej 100? Przerabialiśmy to już w podrozdziale o liczbach pierwszych. Tam pozwoliliśmy sobie na rozmach do pierwiastka z miliarda (i krok więcej), a tutaj wystarczy nam raptem 25 początkowych liczb pierwszych od 2 do 97.

Listę potrzebnych liczb pierwszych wkleimy sobie z tamtego programu, a ich ilość obliczymy przy pomocy operatora `sizeof` (wiemy, że to będzie 25, ale taka konstrukcja jest fajna i wygodna w wielu przypadkach):

```
const int P[] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
                 29, 31, 37, 41, 43, 47, 53, 59,
                 61, 67, 71, 73, 79, 83, 89, 97};
const int NP = sizeof(P) / sizeof(int);
```

[§]Użycie pustych nawiasów klamrowych {} zapewni nam wstępne wyzerowanie całej tablicy. Nie zawsze takie „ręczne” zerowanie zmiennych jest konieczne, ale Kocurro uczył się programować, gdy było to naprawdę niezbędne i ten nawyk oszczędza mu wiele nerwów, stresu i cichaczem rzucanych przekleństw.

Mimo to nie będzie łatwo. Po pierwsze główną rolę w naszej sztuce będzie pełnić magiczna liczba M, czyli 1000000007, no to ją sobie zadeklarujmy:

```
const int M = 1000000007;
```

Główną liczbę, będącą wynikiem naszych żmudnych obliczeń, będziemy „pamiętać" jako listę wykładników potęgowych jej rozkładu na czynniki pierwsze. Potem zajmiemy się tą resztą z dzielenia przez M.

No i tu pojawia się problem: baza ($n!$) to liczba w liczniku, a ewentualne jej dzielniki (związane z powtarzaniem się liter w danych wejściowych) to dzielniki, czyli czynniki mianownika. Reprezentacja wyniku w postaci listy wykładników idealnie nadaje się do czegoś takiego: mnożenie, to dodawanie do wykładnika, a dzielenie, to odejmowanie. Kto to wymyślił? John Napier (*a.k.a.* Neper),[¶] na przełomie XVI oraz XVII w., twórca logarytmów. Szacunek, i tyle.

No dobra, krok po kroku. Najpierw zbudujemy ten rozkład $n!$ na czynniki pierwsze, do tego posłuży nam funkcja `split_factorial()` (z ang. **split** = *rozłożenie, rozkład*, **factorial** = *silnia*). Dodamy jej ekstra argument `sign` (z ang. *znak*, wymawiaj: *sajn*), aby potrafiła dokładać czynniki zarówno do licznika, jak i mianownika wyniku.

Na potrzeby programu zadeklarujemy sobie tablicę *powers*[ ], gdzie będziemy zapisywać rzeczone wykładniki, zainicjalizowane oczywiście na 0, ale później niemiłosiernie molestowane na plus czy na minus, jak tam wypadnie. To będzie lokalna tablica w funkcji `name_count()`. Aby całą silnię obsłużyć, trzeba się przespacerować po wszelkich jej czynnikach od 2 do n włącznie. To już chyba gdzieś tam robiliśmy, to znaczy to, co trzeba zrobić z każdym takim czynnikiem (podrozdział *Na drobne kawałki*).

Dla każdego czynnika występującego w silni dokonujemy jego rozkładu na czynniki pierwsze przy użyciu tablicy $P[\,]$. Jeśli dana liczba pierwsza nam pasuje, wtedy dodajemy wartość zmiennej `sign` do odpowiedniego wykładnika i usuwamy ten dzielnik z czynnika silni. Jeśli `sign` będzie równe 1, wtedy będzie to odpowiadać mnożeniu, a jeśli będzie równe -1, wówczas będziemy mieć mnożenie mianownika, czyli dzielenie:

```
void split_factorial(int n, int powers[], int sign)
{
  for(int k = 2; k <= n; k++)
  {
    int k_copy{ k }, i{ };
    while(k_copy > 1)
      if(k_copy % P[i] == 0)
      {
        powers[i] += sign;
        k_copy /= P[i];
      }
      else
        i++;
  }
}
```

[¶] **also known as**, *również znany pod imieniem*, wymawiaj: *olsoł noln ez*, dźwięczne *z*.

Wśród argumentów funkcji jest tablica – nie podaje się wtedy jej rozmiaru. Warto zwrócić uwagę na użycie pomocniczej zmiennej *k_copy*: oryginalny czynnik z silni (k) musi zostać zastąpiony przez kopię, bo podczas rozkładu zmieniamy jego wartość. Wewnętrzna pętla kręci się, póki rozkładana zmienna jest większa od 1, czyli że zawiera jeszcze choć jeden dzielnik pierwszy.

W funkcji `name_count()` możemy zatem dopisać kolejny fragment, obliczający wykładniki potęgowe z licznika ($sign = 1$) i mianownika ($sign = -1$) – dla powtarzających się liter:

```
int powers[NP] = { };
split_factorial(name.size(), powers, 1);
for(int q : letters)
  if(q > 1)
    split_factorial(q, powers, -1);
```

Zmienna q przebiega tablicę liczników *letters*[ ]. Dla wartości większych od 1 usuwamy zbędne powtórzenia. (Efektywnie ilość możliwości ulega podzieleniu przez $q!$).

Warto zwrócić uwagę na jeden drobiazg: przy wywołaniu funkcji argumenty przekazywane są przez wartość, to znaczy, że funkcja operuje tylko na ich kopiach. Wyjątkiem jest przekazywanie argumentu przez referencję (było opisane w podrozdziale *Kto lepszy?*). Tutaj mamy zmienną tablicową jako argument, czyli wskaźnik do zawartości tablicy. Tak więc funkcja `split_factorial()` samego wskaźnika nie zmieni, ale może zmienić wartość komórek tablicy *powers*[ ] i o to właśnie nam chodzi.

Pozostaje nam wyliczyć ostateczny wynik, modulo ta kosmiczna liczba M. I tu jest właściwy moment, aby przedstawić Czytelnikowi bardzo sympatyczny i efektywny algorytm obliczający potęgę dodatniej liczby całkowitej modulo M. To znaczy, o tym modulo, to za chwilę, na razie po prostu potęga x^n – o wykładniku naturalnym, rzecz jasna. Rozważymy trzy możliwości:

1. $n = 0$,
2. n parzyste, większe od zera,
3. n nieparzyste.

Jeśli wykładnik jest równy 0, to potęga jest równa 1, nie ma co kombinować. Drugi przypadek: jeśli wykładnik jest parzysty, to można wyliczyć x do potegi $\frac{n}{2}$ i podnieść do kwadratu, dla większych n na pewno się opłaci. Trzeci przypadek: można wyliczyć x do potegi $\left\lfloor \frac{n}{2} \right\rfloor$, podnieść do kwadratu i jeszcze pomnożyć przez x. Mogłoby to wyglądać tak:

```
typedef long long ll;

ll quick_power(int x, int n)
{
  if(n == 0)
    return 1;
  ll y = quick_power(x, n/2);
  y = y * y;
  if(n % 2 > 0)
    y = y * x;
```

```
  return y;
}
```

Posłużyliśmy się typem `long long` (`ll`), żeby nie kusić licha.

Jednak nam chodzi o obliczanie potęgi modulo M (stała globalna), więc musimy dopisać to do naszej funkcji, najlepiej przy każdym mnożeniu:

```
ll modular_power(int x, int n)
{
  if(n == 0)
    return 1;
  ll y = modular_power(x, n/2);
  y = (y * y) % M;
  if(n % 2 > 0)
    y = (y * x) % M;
  return y;
}
```

I tyle. Przy okazji zmieniliśmy jej nazwę na `modular_power()`.

No to teraz już zostaje dokończyć funkcję `name_count()`: trzeba wymnożyć wszystkie potęgi liczb pierwszych, oczywiście modulo M. To jedna prosta pętla:

```
  . . .
  ll result{ 1 };
  for(int i = 0; i < NP; i++)
    result = (result * modular_power(P[i], powers[i])) % M;
  return result;
}
```

To już wszystko, lecz dla porządku przytoczymy jeszcze kompletny program, od początku do końca:

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

const int P[] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
                 29, 31, 37, 41, 43, 47, 53, 59,
                 61, 67, 71, 73, 79, 83, 89, 97};
const int NP = sizeof(P) / sizeof(int);
const int M{ 1000000007 };

typedef long long ll;

ll modular_power(int p, int exponent)
{
  if(exponent == 0)
    return 1;
```

```
  ll x = modular_power(p, exponent / 2);
  x = (x * x) % M;
  if(exponent % 2 > 0)
    x = (x * p) % M;
  return x;
}

void split_factorial(int n, int powers[], int sign)
{
  for(int k = 2; k <= n; k++)
  {
    int k_copy{ k }, i{ };
    while(k_copy > 1)
      if(k_copy % P[i] == 0)
      {
        powers[i] += sign;
        k_copy /= P[i];
      }
      else
        i++;
  }
}

int name_count(string name)
{
  int letters[26] = { };
  for(char c : name)
    letters[c - 'a']++;
  int powers[NP] = { };
  split_factorial(name.size(), powers, 1);
  for(int q : letters)
    if(q > 1)
      split_factorial(q, powers, -1);
  ll result{ 1 };
  for(int i = 0; i < NP; i++)
    result = (result * modular_power(P[i], powers[i])) % M;
  return result;
}

int main()
{
  string name;
  cin >> name;
  cout << name_count(name) << endl;
  return 0;
}
```

Czytelnikowi pozostawimy sprawdzenie, że takie obliczanie reszty z dzielenia przez M na każdym etapie mnożenia daje w końcu poprawny rezultat.