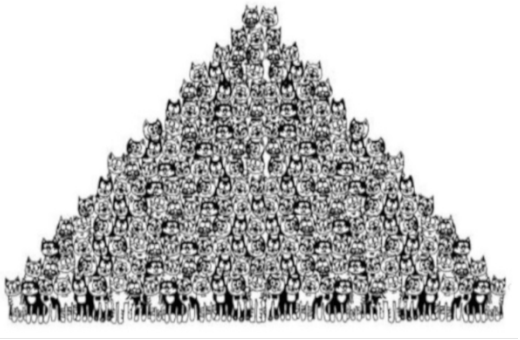
Wielkie mnożenie



#algorytm_rosyjskich_chłopów
#dzielenie_przez_2
#algorytm_Karacuby
#divide_and_conquer

Przechodzimy teraz do mnożenia wielkich liczb naturalnych: zaczniemy skromnie, od mnożenia dużej liczby przez liczbę jednocyfrową. Następnie przedstawimy „naiwny" algorytm mnożenia oparty na zwykłym mnożeniu pisemnym, potem przedstawimy algorytm mnożenia zwany *algorytmem rosyjskich chłopów*,* a na koniec zapoznamy Czytelnika z efektywnym algorytmem Karacuby.† Gwoli prawdy trzeba powiedzieć, że najbardziej efektywny znany (praktyczny) algorytm mnożenia‡ jest oparty na zastosowaniu Szybkiej Transformaty Fouriera (ang. **Fast Fourier Transform**, **FFT**, wymawiaj: *fast furier transform*, z akcentem na *o*). Jednak ten ostatni przykład związany jest z dość zaawansowaną matematyką, wykraczającą raczej poza poziom tego podręcznika.

Mnożenie przez liczbę jednocyfrową

Zajmijmy się przypadkiem, gdy mnożna jest dużą liczbą (zmienna typu `string`), zaś mnożnik jest liczbą jednocyfrową (zmienna typu `char`). Jeśli zastosujemy zwykły algorytm mnożenia pisemnego, wtedy sprawa wygląda prosto: przechodzimy mnożną od prawej do lewej, wykonujemy mnożenie w każdej kolumnie i posługujemy się przeniesieniem (zmienna *carry*) – podobnie jak przypadku dodawania liczb, na przykład:

$$\begin{array}{r} 345678 \\ \times\ 7 \\ \hline 2419746 \end{array}$$

Tym razem jednak musimy pamiętać, że przeniesienie może być liczbą większą od 1 (dlaczego?), co zresztą widzimy w powyższym przykładzie. Oczywiście, pamiętamy cały czas o różnicy pomiędzy kodem ASCII znaku, a reprezentowaną przezeń cyfrą.

Zaczniemy jak zwykle od nagłówka funkcji. Od razu możemy „wyifować" przypadek, gdy mnożnik jest równy zeru:

```
string big_product_1(const string &a, char d)
{
```

*Zwany też *algorytmem rosyjskich wieśniaków*.

†Karacuba to nazwisko rosyjskie, więc pozwoliliśmy sobie je zapisać w swojskiej transkrypcji, choć w literaturze fachowej spotkamy raczej formę *Karatsuba* i taką też formę będziemy stosować w implementacji algorytmu. (Przy zapisie kodu źródłowego stosujemy dość konsekwentnie notację angielską.)

‡Algorytm Schönhage'a-Strassena.

```
  if(d == '0') return "0";
  . . .
```

Użyliśmy tu angielskiego słowa **product** (wymawiaj: *prodakt*) oznaczającego iloczyn. Końcówka `1` wskazuje na jednocyfrowy mnożnik, bo napiszemy jeszcze inną funkcję o podobnej nazwie.

Teraz czas na deklaracje najważniejszych zmiennych i główną pętlę po cyfrach mnożnej. Zadeklarowaliśmy pomocniczą zmienną `d_int`, gdzie przechowamy liczbową wartość mnożnika d. W zmiennej *product* będziemy zbierać kolejne cyfry iloczynu, a przeniesienie *carry* na początku jest równe zero:

```
int d_int = d - '0', carry = 0;
string product;
for(int i = a.size() - 1; i >= 0; i--)
{
  . . .
```

Zmienna *result* będzie przechowywać wynik mnożenia w danej kolumnie – do zmiennej *product* dopiszemy wynik modulo 10, przeniesienie będzie równe podłodze z wyniku dzielonego przez 10:

```
  int result = (a[i] - '0') * d_int + carry;
  product = char(result % 10 + '0') + product;
  carry = result / 10;
}
```

Kiedy pętla się skończy, trzeba jeszcze uwzględnić możliwe przeniesienie i zwrócić wartość ciągu *product*:

```
  if(carry > 0)
    product = char(carry + '0') + product;
  return product;
}
```

Dla porządku przytoczymy cały tekst funkcji `big_product_1()`:

```
string big_product_1(const string &a, char d)
{
  if(d == '0') return "0";
  int d_int = d - '0', carry = 0;
  string product;
  for(int i = a.size() - 1; i >= 0; i--)
  {
    int result = (a[i] - '0') * d_int + carry;
    product = char(result % 10 + '0') + product;
    carry = result / 10;
  }
  if(carry > 0)
    product = char(carry + '0') + product;
  return product;
}
```

Mnożenie pisemne

Jeśli mnożnik posiada więcej niż jedną cyfrę, wtedy mnożenie – prowadzone według klasycznych reguł ze szkoły powszechnej – rozciąga się na tyle wierszy, ile cyfr ma mnożnik, na przykład:

$$\begin{array}{rl} 2345 & \\ \times\ 678 & \\ \hline 18760 & (6) \\ 164150 & (7) \\ +\ 1407000 & (8) \\ \hline 1589910 & \end{array}$$

Wyniki kolejnych mnożeń dodaje się, a na czerwono zaznaczyliśmy zera, których zwykle się nie pisze, zastępując je przesunięciem iloczynu w danym wierszu w lewo. Jednak w naszej funkcji mnożącej trzeba będzie dodać na końcu takie zera, gdyż po prostu tak będzie najłatwiej zrealizować dodawanie cząstkowych iloczynów – mamy do tego gotową funkcję `big_sum()`.

Zaczynamy od nagłówka funkcji i „wyifowania” przypadku, gdy któryś z czynników jest zerem:

```
string big_product(const string &a, const string &b)
{
  if(a == "0" || b == "0")
    return "0";
  . . .
```

Teraz deklarujemy zmienną *result*, do której dodawać będziemy iloczyny z kolejnych wierszy (konieczna jest jej inicjalizacja na stringowe zero) oraz pusty na razie ciąg zer (tych czerwonych) – *suffix* (z angielska: *sufiks*, *przyrostek*,[§] wymawiaj: *safiks*). No i zaczynamy pętlę po cyfrach mnożnika b, oczywiście „od tylca”:

```
string result = "0", suffix;
for(int i = b.size() - 1; i >= 0; i--)
{
  . . .
```

Do obliczenia iloczynu w danym wierszu wykorzystamy gotową funkcję `big_product_1()`. Pomnożymy mnożną a przez jednocyfrową liczbę $b[i]$ i wynik dodamy do dotychczasowej wartości zmiennej *result*, a tę sumę podstawimy znów za zmienną *result*:

```
  result = big_sum(result, big_product_1(a + suffix, b[i]));
  suffix += '0';
}
```

Zwróćmy uwagę, że zmienna *result* wystąpi tutaj w podwójnej roli: argumentu funkcji oraz lewej strony instrukcji podstawienia. W deklaracji funkcji `big_sum()` argumenty funkcji poprzedzone są słowem kluczowym `const`, ale podczas działania funkcji `big_sum()` nie zmieniamy

[§]Czyli po naszemu: końcówka ciągu znaków. Językoznawcy troszkę inaczej rozumieją przyrostek – jako fragment wyrazu występujący po rdzeniu, a nie będący końcówką fleksyjną. Ale nie będziemy się chyba z nimi kłócić...

wartości żadnego jej argumentu (w tym zmiennej *result*), a podstawienie następuje dopiero po zakończeniu działania tej funkcji.

No tak, poza tym dodaliśmy jedno „czerwone” zero do zmiennej $suffix$.

Na koniec funkcja zwraca wartość zmiennej *result*:

```
  return result;
}
```

A oto i cała postać funkcji `big_product()`:

```
string big_product(const string &a, const string &b)
{
  if(a == "0" || b == "0")
    return "0";
  string result = "0", suffix;
  for(int i = b.size() - 1; i >= 0; i--)
  {
    result = big_sum(result, big_product_1(a + suffix, b[i]));
    suffix += '0';
  }
  return result;
}
```

Algorytm rosyjskich chłopów

Geneza dość niecodziennej nazwy tego algorytmu sięga XIX wieku: był on popularną metodą mnożenia liczb naturalnych polegającą na sprowadzeniu tego działania tylko do dzielenia i mnożenia przez 2 (co jest dużo łatwiejsze od mnożenia wielocyfrowych liczb) oraz dodawania. Co ciekawe, ulepszone wersje tego algorytmu stosuje się w elektronicznych układach arytmetycznych, także we współczesnych komputerach.

Załóżmy, że chcemy tą metodą pomnożyć liczbę 123 przez liczbę 19. Piszemy obydwie liczby obok siebie i jedną z nich będziemy mnożyć przez 2, a drugą dzielić przez 2 (z podłogą). Zobaczmy, jak to wygląda:

123	19
+ 246	9
492	4
984	2
+ 1968	1
= 2337	

Liczba po prawej stronie nam się skończyła, co oznacza koniec kolumny (słupka). Na czerwono zaznaczyliśmy w niej liczby *nieparzyste*, a w kolumnie po lewej pokolorowaliśmy na niebiesko odpowiadające im wartości. Jeśli teraz zsumujemy niebieskie liczby, dostaniemy prawidłowy wynik, czyli 2337. Dlaczego to działa? Trzeba sobie przypomnieć, jak zamieniało się liczbę dziesiętną na postać dwójkową. Ano, dokładnie tak, dzieląc ją przez dwa. Jeśli dzielna była nieparzysta (czyli dzielenie dawało resztę równą 1), wtedy w dwójkowej postaci liczby mieliśmy

cyfrę 1, w przeciwnym razie: 0. Istotnie $19_{10} = 10011_2$. Zatem w tej metodzie zamieniliśmy mnożnik 19 na sumę odpowiednich potęg liczby 2: $1 + 2^1 + 2^4$, co dało oczywiście ten sam wynik.

OK, z dodawaniem i z mnożeniem przez 2 poradzimy sobie, bo mamy gotowe funkcje `big_sum()` oraz `big_product_1`, ale co z dzieleniem? Jeśli dzielnik będzie ustalony i równy 2, to nie wygląda to tak strasznie: musimy tylko zaimplementować klasyczny algorytm dzielenia pisemnego.

Załóżmy, że chcemy obliczyć iloraz 4320 : 2 (a dokładniej podłogę z niego). Zapisujemy obydwie liczby, dzielną i dzielnik, i rysujemy na górze kreskę. Dzielenie wykonujemy od lewej do prawej, zapisując kolejne cyfry ilorazu nad kreską (u nas na zielono):

```
 2160
-----
 4320 : 2
-4
--
 03
 -2
---
  12
 -12
---
   00
```

Na niebiesko oznaczyliśmy kolejne cyfry dzielnej przepisywane z góry do odpowiedniego wiersza: 4, 3, 2, 0. Stanowią one cyfrę jedności liczby dzielonej w danym wierszu. Zauważmy, że pierwsza cyfra dzielnej (4) pełni tę samą rolę. Na czerwono oznaczone są reszty z dzielenia w kolejnych wierszach – przechodzą one niżej w roli cyfr dziesiątek następnej dzielonej liczby. Na przykład cyfra 1 staje się cyfrą dziesiątek w liczbie 12.[¶]

A czy byłby problem, gdyby dzielna była nieparzysta? Nie, gdyż w każdym wierszu obliczamy podłogę z ilorazu i jeśli na końcu zostanie nam „coś" mniejszego od dzielnika, po prostu to ignorujemy.

Teraz musimy zreprodukować ten schemat dzielenia (ang. **division**, wymawiaj: *dywiżn*, akcent na *i*) w języku C++. Zaczniemy pisać funkcję `big_div_2()`:

```
string big_div_2(const string &a)
{
  if(a == "0" || a == "1") return "0";
  . . .
```

Tradycyjnie obsługujemy przypadek trywialnej wartości argumentu na początku funkcji. Liczbę dzieloną w danym wierszu oznaczymy sobie *dividend* (z ang. *dzielna*, wymawiaj: *dywajdend*). Rozpoczynamy główną pętlę w funkcji:

```
string result;
int dividend = 0;
for(int i = 0; i < a.size(); i++)
{
  . . .
```

[¶]Oczywiście taki uproszczony opis jest możliwy, gdy dzielnik jest jednocyfrowy.

Do zmiennej *dividend* dodajemy wartość kolejnej cyfry dzielnej. Podłoga z dzielenia tej liczby przez 2 to kolejna cyfra wyniku (budowanego w zmiennej *result*). Ale uwaga: tym razem cyfry wyniku dokładamy od lewej do prawej, stąd inna kolejność łączenia znaków. Reszta z dzielenia zmiennej *dividend*, pomnożona przez 10, przechodzi dalej:

```
    dividend += a[i] - '0';
    result += char(divident / 2 + '0');
    divident = (divident % 2) * 10;
  }
```

I to w zasadzie wszystko, pozostaje zwrócić wartość zmiennej *result* jako wartość funkcji. *Really?* Uważny Czytelnik na pewno zauważy, że jeśli pierwszą cyfrą przynajmniej dwucyfrowej dzielnej jest 1, wtedy nasza funkcja zwróci wynik zaczynający się od cyfry 0 (dlaczego?). Na pewno tego nie chcemy, zatem posłużymy się tym razem funkcją `erase()` do usunięcia niechcianego wiodącego zera:

```
    if(result[0] == '0')
      result.erase(0, 1);
    return result;
  }
```

Funkcja (procedura) `erase()` ma dwa argumenty: numer znaku, od którego zaczynamy usuwanie oraz ilość usuwanych znaków.

Proszę zwrócić uwagę, że „wyifowanie" przypadku dzielnej mniejszej od dzielnika gwarantuje nam poprawne działanie naszej funkcji (dlaczego?).

Oto i nasze dzieło w całości:

```
string big_div_2(const string &a)
{
  if(a == "0" || a == "1") return "0";
  string result;
  int dividend = 0;
  for(int i = 0; i < a.size(); i++)
  {
    dividend += a[i] - '0';
    result += char(dividend / 2 + '0');
    dividend = (dividend % 2) * 10;
  }
  if(result[0] == '0')
    result.erase(0, 1);
  return result;
}
```

Jeszcze taki drobiazg: użycie funkcji `erase()` wymaga dodania `#include <string>` na początku programu.

Mając takie narzędzia bez trudu zmontujemy teraz funkcję `big_russian()` (wymawiaj: *byg raszn*) realizującą algorytm rosyjskich drobnych producentów rolnych.

Zaczynamy od nagłówka funkcji i „wyifowania" przypadku zerowej mnożnej. W zmiennej *result* będziemy zbierać sumę wybranych liczb niebieskich (tych z lewej kolumny tabelki zaprezentowanej na początku tej sekcji):

```
string big_russian(string a, string b)
{
  if(a == "0") return "0";
  string result = "0";
  . . .
```

Tym razem przed argumentami funkcji nie występuje ani słowo `const`, ani symbol referencji `&`, ponieważ i tak musimy skopiować stringi a i b do funkcji, a tam będziemy je modyfikować.

Główna pętla przypomina tę z przykładu o zamianie na układ dwójkowy. Liczby niebieskie dodajemy do zmiennej *result*. A po czym rozpoznajemy, że liczba czerwona jest nieparzysta? Wystarczy sprawdzić jej ostatnią cyfrę – ona decyduje o parzystości (pominęliśmy odejmowanie znaku `'0'`, bo jego kod ASCII jest parzysty):

```
while(b > "0")
{
  if(b[b.size() - 1] % 2 > 0)
    result = big_sum(result, a);
  a = big_product_1(a, '2');
  b = big_div_2(b);
}
```

Pozostaje tylko zwrócić wartość zmiennej *result* jako wartość funkcji.

Oto i cały kod funkcji:

```
string big_russian(string a, string b)
{
  if(a == "0") return "0";
  string result = "0";
  while(b > "0")
  {
    if(b[b.size() - 1] % 2 > 0)
      result = big_sum(result, a);
    a = big_product_1(a, '2');
    b = big_div_2(b);
  }
  return result;
}
```

Niech Czytelnik się zastanowi, dlaczego powyższy kod działa, choć narzucałoby się, aby w warunku kontynuacji pętli `while` użyć operatora `!=` zamiast znaku większości `>`.

Algorytm Karacuby

Czas na najbardziej zaawansowany algorytm mnożenia, który zamierzamy tutaj przedstawić – algorytm Karacuby. Załóżmy, że mamy do pomnożenia dwie długie liczby a oraz b, z których dłuższa ma L cyfr. (Oczywiście, mogą być one tej samej długości.) Niech n będzie dowolną dodatnią liczbą całkowitą mniejszą od L. Wtedy czynniki mnożenia możemy przedstawić w następujący sposób:

$$a = a_1 \cdot 10^n + a_0,$$
$$b = b_1 \cdot 10^n + b_0.$$

gdzie liczby a_1, a_0, b_1, b_0 są liczbami naturalnymi oraz a_0 i b_0 są mniejsze od 10^n. Na przykład jeśli $a = 1234$, $b = 567$ oraz $n = 2$, wtedy mamy:

$$a = 12 \cdot 10^2 + 34,$$

$$b = 5 \cdot 10^2 + 67.$$

czyli:

$$a_1 = 12, \quad a_0 = 34, \quad b_1 = 5, \quad b_0 = 67.$$

Jeśli wykonamy mnożenie tak przedstawionych liczb a i b, otrzymamy następujące wyrażenie:

$$a \cdot b = (a_1 \cdot 10^n + a_0) \cdot (b_1 \cdot 10^n + b_0)\,,$$

czyli:

$$a \cdot b = k_2 \cdot 10^{2n} + k_1 \cdot 10^n + k_0,$$

gdzie

$$k_2 = a_1 \cdot b_1,$$
$$k_1 = a_1 \cdot b_0 + a_0 \cdot b_1,$$
$$k_0 = a_0 \cdot b_0.$$

Powyższe wzory wymagają czterech mnożeń (widać to?). Można jednak sprytnie je przekształcić tak, aby wystarczyły tylko trzy mnożenia – zrobił to właśnie Anatolij Karacuba w 1960 roku. Zauważmy bowiem, że:

$$k_1 = (a_1 + a_0) \cdot (b_1 + b_0) - a_1 \cdot b_1 - a_0 \cdot b_0 = (a_1 + a_0) \cdot (b_1 + b_0) - k_2 - k_0.$$

Zatem jeżeli najpierw wyliczymy k_2 oraz k_0, to do obliczenia k_1 będziemy potrzebować tylko jednego mnożenia. OK, będzie więcej dodawań i odejmowań, ale te ostatnie działania są znacznie szybsze od mnożenia. Istotnie, dodawanie (lub odejmowanie) dwóch liczb o n cyfrach to około n operacji, zaś mnożenie (pisemne) dwóch takich liczb to około n^2 operacji. Przy dużych wartościach n zysk jest znaczący. Poza tym zamieniliśmy mnożenie liczb o L cyfrach na operacje na liczbach krótszych – zwykle wybiera się $n \approx L/2$, co daje największe oszczędności czasowe.

Prześledźmy dokładnie, jak to działa na naszych przykładowych liczbach $a = 1234$, $b = 567$ oraz $n = 2$. Obliczamy kolejno:

$$k_2 = a_1 \cdot b_1 = 12 \cdot 5 = 60,$$

$$k_0 = a_0 \cdot b_0 = 34 \cdot 67 = 2278.$$

Teraz mamy (ostatnie dwa składniki zebraliśmy razem w nawiasie):

$$k_1 = (a_1 + a_0) \cdot (b_1 + b_0) - (k_2 + k_0) = (12 + 34) \cdot (5 + 67) - (60 + 2278) =$$

$$= 46 \cdot 72 - 2338 = 974.$$

Ostatecznie tą metodą otrzymujemy:

$$a \cdot b = k_2 \cdot 10^{2n} + k_1 \cdot 10^n + k_0 = 60 \cdot 10^4 + 974 \cdot 10^2 + 2278 =$$

$$= 600000 + 97400 + 2278 = 699678.$$

Łatwo sprawdzić tradycyjnym sposobem na kalkulatorze, że 1234 pomnożone przez 567 daje właśnie taki wynik.

No fajnie, zamiast początkowego mnożenia liczb o długości L mamy działania na dwukrotnie krótszych liczbach, które jednak też mogą być pokaźnych rozmiarów. I co z tym począć? Nic prostszego, jak zastosować rekurencyjnie ten sam algorytm do krótszych liczb, aż dojdziemy do sytuacji, kiedy jeden z czynników będzie jednocyfrowy, z czym sobie już umiemy radzić.

Zaczniemy zatem pisać funkcję `big_karatsuba()` od „wyifowania" prostych przypadków:[‖]

```
string big_karatsuba(const string &a, const string &b)
{
  if(a.size() == 1)
    return big_multiply_1(b, a[0]);
  if(b.size() == 1)
    return big_multiply_1(a, b[0]);
  . . .
```

Teraz wybieramy wartość zmiennej n, która decyduje o podziale oryginalnych czynników a i b na krótsze liczby. Bierzemy połowę liczby cyfr dłuższego czynnika (dokładniej: podłogę z tej liczby):

```
int n = max(a.size(), b.size()) / 2;
```

Czynniki a i b dzielimy tak, aby drugie ich części (a_0 oraz b_0) miały długość równą n:

```
string a_1 = a.substr(0, a.size() - n),
       a_0 = a.substr(a.size() - n),
       b_1 = b.substr(0, b.size() - n),
       b_0 = b.substr(b.size() - n);
```

Przypominamy, że jeśli funkcja `substr()` jest wywoływana tylko z jednym argumentem, wtedy zwraca ona podciąg począwszy od znaku o wskazanym numerze aż do końca ciągu.

Teraz możemy już obliczyć k_0, k_2, a później k_1 (to ostatnie obliczamy na dwa tempa, aby nie straszyć Czytelnika zbyt rozbudowanymi formułami):

[‖] Pisownię nazwiska wyjaśniliśmy na początku tego podrozdziału.

```
    string k0 = big_karatsuba(a_0, b_0),
           k2 = big_karatsuba(a_1, b_1),
           k1 = big_karatsuba(big_sum(a_0, a_1), big_sum(b_0, b_1));
           k1 = big_difference(k1, big_sum(k2, k0));
```

Pozostał jeszcze mały detal: jak pomnożyć liczbę przez potęgę liczby 10? Nic prostszego, trzeba dokleić na końcu liczby odpowiednią ilość zer (przerabialiśmy to). Przyda się tutaj wygodna konstrukcja `string(m, '0')`, której rezultatem jest m-elementowy ciąg znaków złożony z samych zer. Pozostaje zatem zwrócić wartość finalnego wyrażenia:

```
  return big_sum(k2 + string(2*n, '0'), big_sum(k1 + string(n, '0'), k0));
}
```

Jak zwykle dla porządku przytoczymy całą treść funkcji `big_karatsuba()`:

```
string big_karatsuba(const string &a, const string &b)
{
  if(a.size() == 1)
    return big_multiply_1(b, a[0]);
  if(b.size() == 1)
    return big_multiply_1(a, b[0]);
  int n = max(a.size(), b.size()) / 2;
  string a_1 = a.substr(0, a.size() - n),
         a_0 = a.substr(a.size() - n),
         b_1 = b.substr(0, b.size() - n),
         b_0 = b.substr(b.size() - n);
  string k0 = big_karatsuba(a_0, b_0),
         k2 = big_karatsuba(a_1, b_1),
         k1 = big_karatsuba(big_sum(a_0, a_1), big_sum(b_0, b_1));
         k1 = big_difference(k1, big_sum(k2, k0));
  return big_sum(k2 + string(2*n, '0'), big_sum(k1 + string(n, '0'), k0));
}
```

Algorytm Karacuby należy do algorytmów opartych na zasadzie **divide and conquer** (*dziel i zwyciężaj*, wymawiaj: *divajd end konker*, z akcentami na *a* oraz na *o*). To bardzo ważna klasa algorytmów, należą do niej na przykład algorytm sortowania szybkiego, sortowania przez scalanie czy poszukiwania pary najbliższych punktów na płaszczyźnie. Mówiąc w dużym uproszczeniu, zasada ta polega na podziale oryginalnego problemu (zbioru danych) na podproblemy (zwykle na dwa), rozwiązania podproblemów oddzielnie, a następnie na połączeniu składowych wyników w jedną całość. Takie algorytmy z natury są rekurencyjne. Wrócimy do tego tematu, myślę że nie raz i nie dwa razy.

Uff, dobrnęliśmy do końca najdłuższego (jak dotąd) podrozdziału podręcznika...