Arytmetyka na sterydach

#arytmetyka_wielkich_liczb
#przeniesienie #pożyczka
#zero_wiodące

W prezentowanych do tej pory przykładach operowaliśmy liczbami całkowitymi z zakresu (na moduł) do około dwóch miliardów (typ danych `int`) lub nawet do około 10^{18} (typ danych `long long`). Sporadycznie używaliśmy typu zmiennoprzecinkowego `double`, który co prawda pozwala na rozszerzenie tego zakresu nawet do 10^{308}, ale za cenę utraty pełnej dokładności obliczeń – do około 16 cyfr znaczących.*

Zakres typów całkowitych można zwiększyć dwukrotnie, jeśli zrezygnujemy z wartości ujemnych, wtedy posługujemy się odpowiednio typami `unsigned int`† lub `unsigned long long` (z ang. *bez znaku*, wymawiaj: *ansajnd*). Jednak typy danych `unsigned` kryją w sobie pewną pułapkę obliczeniową, w szczególności polecamy Czytelnikowi sprawdzenie działania poniższego fragmentu kodu i zastanowienie się, dlaczego to tak działa:

```
unsigned int a = 7, b = 5;
cout << a - b << endl;
cout << b - a << endl;
```

Zatem z tego rodzaju typów korzystamy tylko wtedy, gdy naprawdę istnieje taka konieczność i kiedy rzeczywiście wiemy, co robimy.

A co mamy zrobić, gdybyśmy musieli wykonywać operacje arytmetyczne na naprawdę pokaźnych liczbach całkowitych, dajmy na to, 100-cyfrowych? Da się? Oczywiście, że tak, ale musimy sami zaimplementować arytmetykę tak wielkich liczb.‡

Zanim zapomnimy, warto uświadomić Czytelnikowi, że cokolwiek byśmy tutaj wydumali w zakresie własnej arytmetyki, choćby i najbardziej wyszukanego, będzie to bez porównania wolniejsze od operacji na standardowych typach liczbowych, gdyż te są wspierane *sprzętowo*, czego nie przebijemy.

Zaczniemy skromnie, od dodawania liczb o dowolnej (rozsądnej) ilości cyfr. Jasne chyba jest, że takie liczby należy reprezentować w odmienny sposób, niż zwykłe „małe” liczby całkowite – my wybierzemy wygodny w użyciu typ danych `string`.

*W języku C++ istnieje również bardziej wypasiony typ zmiennoprzecinkowy: `long double` – o jeszcze większym zakresie i lepszej dokładności.

†Można to zapisać po prostu jako `unsigned`.

‡Rzecz jasna, mądrzy ludzie coś tam już wymyślili w tym temacie i w internetach znajdziemy bibliotekę (pewnie niejedną), która umożliwia takie działania, ale chyba pouczające będzie spróbować samodzielnie napisać choćby najprostsze funkcje w tej dziedzinie.

Dodawanie wielkich liczb

Algorytm dodawania dużych liczb naturalnych (do takich się ograniczymy) nie jest żadnym odkryciem Ameryki. To dobrze nam znany algorytm dodawania liczb na sposób pisemny. Jeśli na przykład chcemy zsumować liczby 7241 oraz 368, podpisujemy jedną liczbę pod drugą, tak aby ich cyfry jedności znajdowały się w jednej kolumnie:

$$\begin{array}{r} 7241 \\ +\ 368 \\ \hline \end{array}$$

Sumujemy kolejno cyfry w kolumnach od prawej do lewej:

$$\begin{array}{r} 7241 \\ +\ 368 \\ \hline 7609 \end{array}$$

Dodajemy 1 do 8 i otrzymujemy 9, dodajemy 4 do 6 i otrzymujemy 10, liczbę większą niż 9, zatem pojawia się coś, co nazywamy *przeniesieniem* (ang. **carry**, wymawiaj: *kery*), które równe jest podłodze z dzielenia tego wyniku przez 10: tutaj jest to $\lfloor 10/10 \rfloor = 1$. Przeniesienie przechodzi dalej i jest dodawane do sumy cyfr z następnej kolumny. Przy dodawaniu dwóch liczb wartość przeniesienia wynosi zawsze albo 1 albo 0 (dlaczego?).

Zatem w następnej kolumnie mamy $2+3+1$ (przeniesienie), czyli 6, a w ostatniej kolumnie mamy tylko 7 z pierwszego składnika i (w domyśle) 0 z drugiego składnika, stąd wynik w tej kolumnie wynosi 7, całkowita suma: 7609.

Zaczniemy może od końca, pisząc funkcję `main()`, w której przetestujemy funkcję `big_sum()` (z ang. *wielka suma*, wymawiaj: *byg sam*) obliczającą sumę dwóch wielkich liczb:

```
int main()
{
  string a, b;
  cin >> a >> b;
  cout << big_sum(a, b) << endl;
}
```

Zatem definicja funkcji sumującej powinna zaczynać się w ten sposób (rezultat dodawania umieścimy w zmiennej *c*):

```
string big_sum(const string &a, const string &b)
{
  string c;
  . . .
```

Kombinację słowa kluczowego `const` oraz symbolu referencji `&` już znamy, prawda?

Najwygodniej będzie zrealizować sumowanie przy pomocy dwóch zmiennych-indeksów *ia* oraz *ib*, oznaczających numery znaków w stringach *a* oraz *b* odpowiadających bieżącej kolumnie sumowania. Ponieważ liczby dosunięte są do prawej strony, więc na początku indeksy powinny wskazywać na ostatnie znaki w stringach:

```
int ia = a.size() - 1, ib = b.size() - 1;
```

Przed przystąpieniem do sumowania przeniesienie wynosi 0:

```
int carry = 0;
```

W głównej pętli tej funkcji będziemy stopniowo zmniejszać indeksy *ia* oraz *ib*, aż osiągną wartość ujemną. Jeśli chociaż jeden z nich jest nieujemny, to znaczy, że dana liczba jeszcze się nie skończyła. Sumowanie w danej kolumnie rozpoczniemy od uwzględnienia przeniesienia. W zmiennej *result* (z ang. *wynik*, *rezultat*, wymawiaj: *rezalt*, z akcentem na *a*) przechowamy wynik dodawania w bieżącej kolumnie:

```
while(ia >= 0 || ib >= 0)
{
  int result = carry;
  . . .
```

Cyfrę z danej liczby dodajemy tylko wtedy, jeśli indeks ma wartość nieujemną (po czym go zmniejszamy):

```
if(ia >= 0)
{
  result += a[ia] - '0';
  ia--;
}
if(ib >= 0)
{
  result += b[ib] - '0';
  ib--;
}
```

Pamiętamy oczywiście o odjęciu kodu ASCII znaku `'0'`, dzięki czemu otrzymamy wartość liczbową danej cyfry. (Była o tym mowa w podrozdziałach o układzie dwójkowym i dziesiętnym.)

Teraz należy rozstrzygnąć, jaką cyfrę wpiszemy jako wynik sumowania w danej kolumnie oraz jaka będzie wartość przeniesienia. Jako wynik wpisujemy cyfrę odpowiadającą reszcie z dzielenia zmiennej *result* przez 10, a przeniesienie to wspomniana już podłoga z *result*/10:

```
  c = char(sum % 10 + '0') + c;
  carry = sum / 10;
}
```

Kolejną cyfrę w stringu *c* dopisujemy na jego początku, gdyż cyfry te obliczamy od ostatniej do pierwszej.

Kiedy skończą nam się cyfry w obydwu liczbach, sprawa nie jest bynajmniej zakończona: musimy bowiem sprawdzić, czy nie zostało nam jakieś przeniesienie. Wszakże dodawanie liczb – na przykład – trzycyfrowych może dać w wyniku liczbę czterocyfrową, mamy choćby 333+777 = 1110.

Zatem zakończenie funkcji `big_sum()` powinno wyglądać tak:

```
  if(carry > 0)
    c = '1' + c;
  return c;
}
```

Dobrnęliśmy do końca funkcji sumującej i teraz dla porządku przytoczymy ją w całości:

```
string big_sum(const string &a, const string &b)
{
  string c;
  int ia = a.size() - 1, ib = b.size() - 1;
  int carry = 0;
  while(ia >= 0 || ib >= 0)
  {
    int result = carry;
    if(ia >= 0)
    {
      result += a[ia] - '0';
      ia--;
    }
    if(ib >= 0)
    {
      result += b[ib] - '0';
      ib--;
    }
    c = char(result % 10 + '0') + c;
    carry = result / 10;
  }
  if(carry > 0)
    c = '1' + c;
  return c;
}
```

Po uruchomieniu programu i wpisaniu przykładowych liczb:

```
7241
368
```

otrzymamy wynik:

```
7609
```

Sumowanie jakoś poszło, teraz weźmiemy się za odejmowanie.

Odejmowanie wielkich liczb

Odejmowanie liczb naturalnych nie jest dużo trudniejsze od dodawania, ale jednak trzeba pamiętać o kilku niuansach. Przede wszystkim zamiast *przeniesienia* będziemy mieć teraz *pożyczkę* (ang. **borrow**,[§] wymawiaj: *boroł*). Pozostaje jednak kwestia znaku wyniku odejmowania.

Jeśli odjemnik (tutaj: 567) jest mniejszy od odjemnej (tutaj: 1234), wtedy dostajemy wynik dodatni, na przykład:

$$\begin{array}{r} 1234 \\ -\ 567 \\ \hline 667 \end{array}$$

Jeśli odjemna i odjemnik są równe, wtedy wynik jest liczbą zero:

$$\begin{array}{r} 8901 \\ -\ 8901 \\ \hline 0 \end{array}$$

Jeśli odjemna (tutaj: 678) jest mniejsza od odjemnika (tutaj: 5432), wtedy dostajemy wynik ujemny:

$$\begin{array}{r} 678 \\ -\ 5432 \\ \hline -\ 4754 \end{array}$$

Ten ostatni przypadek jest szczególny, bo wymaga dodania znaku minus ($-$) przed wynikiem. Najłatwiej byłoby to zrealizować tak: jeśli rzeczywiście on zachodzi, wtedy można zamienić miejscami odjemną i odjemnik, odjąć po Bożemu, a na koniec dopisać znak minus przed wynikiem. Do tego potrzebujemy jednak funkcji sprawdzającej, czy dana liczba jest mniejsza od drugiej. *Really*? A zwykły znak mniejszości nie wystarczy? Niestety nie, gdyż liczby mamy reprezentowane przez stringi, dla których znak $<$ oznacza porównywanie według porządku leksykograficznego (to już gdzieś tam było, prawda?). Istotnie, dla ciągów znaków na przykład poniższa relacja jest prawdziwa:

$$\text{"1111"} < \text{"222"}$$

Musimy więc sami napisać odpowiednią funkcję (taki komparator) na nasze potrzeby – nazwiemy ją `smaller()` (z ang. *mniejszy*, wymawiaj: *smoler*):[¶]

```
bool smaller(const string &a, const string &b)
```

Generalnie, funkcja powinna zwrócić wartość prawdy, jeśli a jest mniejsze od b – w sensie porównywania liczb. Zauważmy, że jeśli a jest krótsze niż b, wtedy na *sicher*[‖] zwracamy prawdę. Z kolei jeśli a jest dłuższe od b, wtedy wynik to na pewno fałsz. A jeżeli długości obydwu liczb są takie same, wtedy zadziała zwykły znak $<$ (dlaczego?).

[§] Warto pamiętać, że w języku angielskim istnieją dwa odpowiedniki polskiego terminu *pożyczka*: **borrow** (gdy pożyczamy coś od kogoś) oraz **loan** (gdy pożyczamy coś komuś).

[¶] Z punktu widzenia językowego bardziej pasowałoby nazwać ją `less()`, jednak ta nazwa jest już zastrzeżona.

[‖] Wymawiaj: *zicher*, to po teutońsku.

Oto proponowana przez nas postać funkcji `smaller()`:

```
bool smaller(const string &a, const string &b)
{
  if(a.size() == b.size())
    return a < b;
  return a.size() < b.size();
}
```

Jak widać, konsekwentnie unikamy niepotrzebnych instrukcji warunkowych, żeby nie rozbudowywać zbytnio kodu programu.

Możemy teraz napisać funkcję `big_difference()` (z ang. *wielka różnica*, wymawiaj: *byg dyfrens*) obliczającą różnicę dwóch liczb naturalnych. Odwołamy się tutaj do jeszcze nienapisanej funkcji `subtract()` (z ang. *odejmij*, wymawiaj: *sabtract*, z akcentem na drugą sylabę), której zadaniem będzie odejmowanie przy gwarancji, że odjemnik nie jest większy od odjemnej:

```
string big_difference(const string &a, const string &b)
{
  if(smaller(a, b))
    return '-' + subtract(b, a);
  return subtract(a, b);
}
```

Początek funkcji `subtract()` wygląda podobnie do funkcji `big_sum()`:

```
string subtract(const string &a, const string &b)
{
  string c;
  int ia = a.size() - 1, ib = b.size() - 1;
  int borrow = 0;
  . . .
```

Ponieważ mamy pewność, że odjemna a nie jest krótsza od odjemnika b, zatem główna pętla w funkcji może być zapisana nieco prościej:

```
while(ia >= 0)
{
  int result = a[ia] - '0' - borrow;
  ia--;
  . . .
```

Pożyczkę (*borrow*) oczywiście odejmujemy od wyniku (*result*) w danej kolumnie.

Odjemnik b może się skończyć wcześniej, niż odjemna, więc musimy się zapytać, czy możemy z niego czerpać kolejne cyfry:

```
if(ib >= 0)
{
  result -= b[ib] - '0';
  ib--;
}
```

Wynik w danej kolumnie może okazać się ujemny, wtedy trzeba zaciągnąć pożyczkę:

```
if(result < 0)
{
  result += 10;
  borrow = 1;
}
else
  borrow = 0;
```

Obliczoną cyfrę w danej kolumnie dopisujemy na początku zmiennej c:

```
  c = char(result + '0') + c;
}
```

... i tak kończy się główna pętla w funkcji `subtract()`. To jednak nie koniec samej funkcji. Zauważmy, że na początku zmiennej c mogą pojawić się zera – są to tak zwane *zera wiodące* (ang. **leading zeros**, wymawiaj: *liding ziroły*, z akcentem na i w drugim wyrazie). Na przykład jeśli funkcję `subtract()` wywołamy z argumentami $a =$ "1145" oraz $b =$ "1123" zmienna c otrzyma wartość "0022". Pasowałoby się pozbyć tych zer, zatem zliczymy, ile ich mamy na początku, ale musimy pamiętać, że c może składać się z *samych* zer – kiedy odjemna i odjemnik są równe. Zliczanie zer wiodących powinno wyglądać tak:

```
int zeros = 0;
while(zeros < c.size() && c[zeros] == '0')
  zeros++;
```

Jeśli liczba tych zer jest równa długości stringu c, wtedy rezultatem odejmowania powinna być po prostu liczba 0 (w formie ciągu znaków). W przeciwnym razie funkcja powinna zwrócić podciąg stringu c począwszy od znaku o numerze *zeros* aż do końca stringu (w takim przypadku można pominąć długość podciągu):

```
  if(zeros == c.size())
    return "0";
  return c.substr(zeros);
}
```

Funkcja `substr()` pojawiła się już w podrozdziale o Odwrotnej Notacji Polskiej – wymaga ona dyrektywy `#include <string>`.

Jak zwykle dla porządku przytoczymy całą treść budowanej po kawałku funkcji `subtract()`:

```
string subtract(const string &a, const string &b)
{
  string c;
  int ia = a.size() - 1, ib = b.size() - 1;
  int borrow = 0;
  while(ia >= 0)
  {
    int result = a[ia] - '0' - borrow;
    ia--;
    if(ib >= 0)
    {
      result -= b[ib] - '0';
      ib--;
    }
    if(result < 0)
    {
      result += 10;
      borrow = 1;
    }
    else
      borrow = 0;
    c = char(result + '0') + c;
  }
  int zeros = 0;
  while(zeros < c.size() && c[zeros] == '0')
    zeros++;
  if(zeros == c.size())
    return "0";
  return c.substr(zeros);
}
```

Definicje funkcji `subtract()` oraz `smaller()` powinny znaleźć się w pliku źródłowym przed definicją funkcji `big_difference()`.**

**Od tej zasady można odejść, posługując się *deklaracjami funkcji*, ale omówimy to w oddzielnym podrozdziale.