Kalkulator dużych liczb w C++

Michał Bełzak
Robert Taube

27 października 2018

Streszczenie

Jednym z ograniczeń języka C++ jest obsługa liczb, których nie możemy zapisać w 8 bajtach. Ten projekt jest próbą zażegnania tego problemu. W tym celu stworzyliśmy prosty kalkulator, który demonstruje potencjalne rozwiązanie tej kwestii.

1 Wstęp

1.1 Opis problemu

Odkąd tylko człowiek zaczął liczyć niezwykle ważna była dla niego dokładność wykonywanych obliczeń. O ile nie było problemem stawianie dodatkowych miejsc po przecinku (już starożytni Babilończycy obliczyli wartość $\sqrt{2}$ z dokładnością do 6 cyfr znaczących [1]), o tyle zaczęło to być kłopotliwe przy próbie budowy jakichkolwiek maszyn liczących, gdzie możemy zaimplementować tylko przeliczalne zbiory a wszelkie stałe są trwale wpisane w architekturę rozwiązania. W programowaniu problem ten leży w sposobie zapisu i obsługi zmiennych liczbowych których wielkość ograniczana jest poprzez rozmiary pamięci zarezerwowane na poszczególne typy danych. Największymi zmiennymi liczbowymi jakie jest w stanie obsłużyć C++ bez żadnych dodatkowych bibliotek są *long long int* oraz zmiennoprzecinkowy typ *double* opierające się na 8 bajtach, ich rozmiar zatem zawiera się w zakresie $[-2^{63}, 2^{63}-1]$. Liczby te są bardzo duże, istnieją jednak zastosowania w których wymagana jest precyzja wyższa niż 15 cyfr znaczących bądź obliczenia na liczbach większych niż $2^{64}-1$. Niniejszy tekst jest opisem naszej próby zaadresowania tego problemu tak aby jedynym ograniczeniem wielkości naszych obliczeń była całkowita pamięć komputera.

1.2 Założenia projektu:

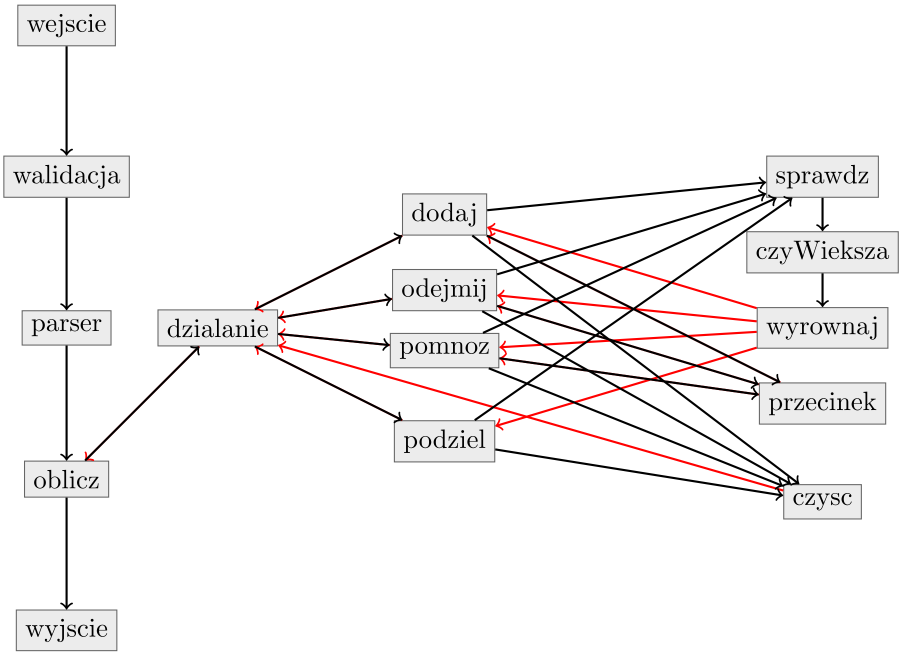
1. możliwość wprowadzenia ciągu instrukcji i ich poprawne wykonanie,
2. maksymalny możliwy rozmiar i dokładność przeprowadzanych obliczeń
3. obsługa działań podstawowych (dodawanie, odejmowanie, mnożenie, dzielenie) oraz nawiasów
4. modułowość

1.3 Główne wyzwania, które stawia projekt:

- zamiana wejścia na stos działań w postaci *odwrotnej notacji polskiej* (algorytm stacji rozrządowej [2])
- implementacja kontenera danych typu *vector* jako typu zmiennoprzecinkowego
- implementacja działań na nowo powstałym kontenerze danych

2 Opis działania programu

2.1 Schemat blokowy



2.2 Opis słowny

Program rozpoczyna pracę od wyświetlenia komunikatu zachęty i oczekiwania na wprowadzenie wejścia od użytkownika, po otrzymaniu zadanej sekwencji obliczeń zostaje ona przekazana do funkcji *walidacja()*, która sprawdza czy sekwencja może być interpretowana jako działania matematyczne i czy jest poprawna w sensie formalnym (to jest sprawdza występowanie niedomkniętych nawiasów oraz błędów typu dwa przecinki w jednej liczbie). Jeżeli walidacja przebiegnie pomyślnie to dane trafiają do parsera, jeżeli nie to zostanie wywołany komunikat o błedzie oraz pytanie, czy użytkownik życzy sobie wypisania instrukcji użytkowania.
Po pomyślnej walidacji ciąg poleceń trafia do parsera, który używając algorytmu Dijkstry dokonuje transformacji poleceń na ciąg postfixowy a następnie przygotowany wektor wysyła do funkcji *oblicz()*. Funkcja ta odczytuje polecenia i przekazuje parametry dzialania do funkcji *dzialanie()*, która zajmuje się rozdzielaniem zadań pomiędzy odpowiednie funkcje liczące.
Wyniki z funkcji liczących zostają zwrócone do funkcji *oblicz()*, która dodaje je na swój stos roboczy i kontynuuje swoją pracę aż do momentu wykonania ostatniego działania, wynik zostaje wtedy wypisany na ekranie i program kończy pracę.

3 Działanie poszczególnych funkcji

3.1 Dodawanie

Wejście Dwa wektory A i B typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Zmodyfikowany wektor B - wynik dodawania $A + B$.

Algorytm Na początku zostają wywołane funkcje *sprawdz()* oraz *wyrownaj()* przygotowujące wektory A i B do operacji. Następnie zostaje wywołana funkcja *przecinek()*, która zapamiętuje pozycję przecinka oraz go usuwa, sprowadzając problem do liczb całkowitych. Główna część algorytmu dodawania działa analogicznie do algorytmu *"dodawania na kartce"*. Funkcja interpretuje i-te komórki wektora A i B jako cyfry i w zależności od ich sumy ustala wartości komórek $B[i]$ oraz $B[i-1]$ ($i = n \wedge i \to 0$ gdzie $n \equiv max(A.size, B.size)$). Po wykonaniu działania jest wywołana funkcja *czysc()* usuwająca zbędne zera, zostaje dopisany usunięty wcześniej przecinek, oraz zostaje zwrócony wynik w postaci wektora B.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv max(A.size, B.size)$.

Złożoność pamięciowa $\theta(1)$.

3.2 Odejmowanie

Wejście Dwa wektory A i B typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Zmodyfikowany wektor B - wynik odejmowania $A - B$.

Algorytm Na początku zostają wywołane funkcje *sprawdz()* oraz *wyrownaj()* przygotowujące wektory A i B do operacji. Funkcja *czyWieksza()* analizuje która z liczb jest większa i odpowiednio dobiera kolejność działania. Następnie zostaje wywołana funkcja *przecinek()*, która zapamiętuje pozycję przecinka oraz go usuwa, sprowadzając problem do liczb całkowitych. Główna część algorytmu odejmowania działa analogicznie do algorytmu *"odejmowania na kartce"*. Funkcja interpretuje i-te komórki wektora A i B jako cyfry i w zależności od ich różnicy ustala wartości komórek $B[i]$ oraz k komórek $A[j]$ ($i = n \wedge i \to 0$ gdzie $n \equiv max(A.size, B.size)$, k - ilość komórek którym trzeba nadać wartość 9, $j = i - 1 \wedge j \to i - 1 - k$). Po wykonaniu działania jest wywołana funkcja *czysc()* usuwająca zbędne zera, zostaje dopisany usunięty wcześniej przecinek, oraz zostaje zwrócony wynik w postaci wektora B.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv max(A.size, B.size)$.

Złożoność pamięciowa $\theta(1)$.

3.3 Mnożenie

Wejście Dwa wektory A i B typu *char* o długości nie większej niż $2^{32} - 2$.

Wyjście Zmodyfikowany wektor A - wynik mnożenia $A * B$.

Algorytm Na początku zostają wywołane funkcje *sprawdz()* oraz *wyrownaj()* przygotowujące wektory A i B do operacji. Następnie zostaje wywołana funkcja *przecinek()*, która zapamiętuje pozycję przecinka oraz go usuwa, sprowadzając problem do liczb całkowitych. Główna część algorytmu mnożenia działa analogicznie do algorytmu *"mnożenia na kartce"*. Zostaje utworzony pomocniczy wektor dp w którym będziemy zapisywać wynik mnożenia. Mnożymy każdą komórkę $A[i]$ przez każdą komórkę $B[j]$ $(i = n-1 \wedge i \to 0$ gdzie $n \equiv A.size \wedge j = m-1 \wedge i \to 0$ gdzie $m \equiv B.size)$. Na podstawie tego iloczynu ustalamy wartość komórek $dp[x]$ i $dp[x-1]$ $(x \equiv n+m-i-j)$. Po wykonaniu obliczeń konwertujemy wektor dp na wektor typu *char* A, zostaje wywołana funkcja *czysc()* usuwająca zbędne zera, zostaje dopisany usunięty wcześniej przecinek oraz zostaje zwrócony wynik w postaci wektora A.

Złożoność obliczeniowa $\theta(n^2)$ gdzie $n \equiv max(A.size, B.size)$.

Złożoność pamięciowa $\theta(2n)$.

3.4 Dzielenie

Wejście Dwa wektory A i B typu *char* o długości nie większej niż $2^{64}-1$.

Wyjście Wektor W - wynik dzielenia A/B.

Algorytm Na początku zostają wywołane funkcje *sprawdz()* oraz *wyrownaj()* przygotowujące wektory A i B do operacji. Następnie zostaje wykonana analogiczna funkcja do funkcji *przecinek()*, która zapamiętuje pozycję przecinka oraz go usuwa, sprowadzając problem do liczb całkowitych. Zostaje również sprawdzone czy B jest różne od zera. Główna część algorytmu mnożenia działa analogicznie do algorytmu *"dzielenia na kartce"*. Zostaje utworzony wektor C do którego będziemy sukcesywnie dopisywać kolejne wartości ciągu A i odejmować od niego dzielnik. Wektor W jest wektorem wynikowym *("To co jest nad kreską")*. Długość wektora A i W zależy od obranej dokładności. Po wykonaniu obliczeń zostaje wywołana funkcja *czysc()* usuwająca zbędne zera, zostaje dopisany usunięty wcześniej przecinek oraz zostaje zwrócony wynik w postaci wektora W.

Złożoność obliczeniowa $\theta(n^2)$ gdzie $n \equiv max(A.size, B.size) +$ *dokładność*.

Złożoność pamięciowa $\theta(2n)$.

3.5 Przecinek

Wejście Dwa wektory A i B typu *char* o długości nie większej niż $2^{64}-1$.

Wyjście Liczba, która oznacza pierwotną pozycję przecinka w wektorze A i B.

Algorytm Funkcja sprawdza i-te wyrazy ciągu A aż do napotkania przecinka $(i = n \wedge i \to 0$ gdzie $n \equiv A.size)$. Zostaje usunięty przecinek z wektora A i B oraz zostaje zwrócona pozycja przecinka.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv A.size$.

Złożoność pamięciowa $\theta(1)$.

3.6 Parser

Wejście Zwalidowany ciąg poleceń w notacji infiksowej pod postacią wektora typu *char*.

Wyjście Ciąg poleceń w notacji postfixowej pod postacią wektora typu *char*.

Algorytm Zastosowany algorytm jest wariacją algorytmu Dijkstry (znanego także jako shunting-yard algorithm[2]). Funkcja wczytuje znaki dopóki są jakieś do sprawdzenia, jeżeli istnieją, to:

- Lewy nawias, to dopisuje go na stos
- Prawy nawias, to przepisuje elementy ze stosu do wektora wynikowego aż do momentu dotarcia do lewego nawiasu, usuwa oba nawiasy nie wpisując ich do wyniku
- Operator działania, to:
  - dopisuje do wyniku literkę c
  - jeżeli stos jest pusty LUB najwyższy element stosu jest operatorem o niższej precedencji niż badany, to dopisz do stosu badany operator
  - jeśli nie to przepisuj elementy ze stosu do wyjścia aż do momentu spełnienia warunku a następnie dopisz do stosu badany operator
- cyfra, to dopisz ją do wyjścia
- jeżeli był to ostatni element sprawdzanego ciągu to przepisz cały stos do wyniku.

Po zakończeniu się pętli następuje zwrócenie wyniku i kolaps funkcji.

Złożoność obliczeniowa $\theta(2n)$

Złożoność pamięciowa $\theta(2n)$

3.7 Oblicz

Wejście Uporządkowany wektor poleceń typu *char* zapisanych w postaci postfixowej.

Wyjście Wektor typu *char* zawierający wynik działania całego programu.

Algorytm Dopóki program ma co wczytywać, to:

- jeżeli znak jest operatorem, to wczytaj dwa najwyższe elementy stosu, usuń je i zastąp wynikiem b(operator)a
- jeżeli znak jest równy c, to przejdź do kolejnego okrążenia pętli
- jeśli nic z powyższych, to do momentu napotkania znaku c lub operatora dopisuj znaki do wektora pomocniczego, a po napotkaniu jednego z wymienionych terminatorów przepisz wektor pomocniczy do stosu.

Po wczytaniu ostatniego znaku linii poleceń zwraca najwyższy element stosu, czyli wynik obliczeń.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv komendy.size$.

Złożoność pamięciowa $\theta(n)$.

3.8 Wyrównywanie

Wejście Wektor A i B typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Brak.

Algorytm Funkcja wyrównuje liczby A i B względem siebie tj. po wykonaniu funkcji $\exists m \in <0, A.size - 1>$: $A[m] = B[m] = ","$. Pierwsza pętla sprawdza i-te wyrazy ciągu A aż do napotkania przecinka ($i = 0 \wedge i \rightarrow n$ gdzie $n \equiv A.size - 1$). Na podstawie wartości i oraz długości ciągu A jest obliczana ilość cyfr przed oraz za przecinkiem. Druga pętla przeprowadza analogiczne obliczenia dla ciągu B. Następnie dopisujemy zera na początku ciągu, którego liczba cyfr przed przecinkiem jest mniejsza. Liczba dopisanych zer jest równa wartości bezwzględnej z różnicy ilości cyfr przed przecinkiem dwóch liczb. Analogicznie dopisujemy zera na końcu ciągu, który ma mniej cyfr za przecinkiem. Funkcja jest typu *void* - wynikiem jej działania jest modyfikacja wejścia.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv A.size + B.size$.

Złożoność pamięciowa $\theta(1)$.

3.9 Sprawdź

Wejście Wektor A typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Brak.

Algorytm Funkcja sprawdza czy wektor A posiada przecinek, jeżeli go nie ma to dopisuje go. Zostają sprawdzone i-te wyrazy ciągu A ($i = 0 \wedge i \rightarrow n$ gdzie $n \equiv A.size - 1$). Jeżeli w ciągu A wystąpi przecinek, funkcja zostaje zakończona. Jeżeli nie, do wektora A zostaje dopisany sufiks $",0"$. Funkcja jest typu *void* - wynikiem jej działania jest modyfikacja wejścia.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv A.size$.

Złożoność pamięciowa $\theta(1)$.

3.10 Czyszczenie

Wejście Wektor A typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Brak.

Algorytm Funkcja usuwa zbędne zera z liczby. Pierwsza pętla usuwa pierwszy element ciągu A dopóki $A[0] = "0"$. Jeżeli po wykonaniu pętli pierwszy wyraz $A = ","$ to funkcja dopisuje 0 na początku wektora. Druga pętla analogicznie usuwa ostatnie elementy wektora i dopisuje 0 na końcu. Funkcja jest typu *void* - wynikiem jej działania jest modyfikacja wejścia.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv A.size$.

Złożoność pamięciowa $\theta(1)$.

3.11 Czy Większa?

Wejście Wektor A i B typu *char* o długości nie większej niż $2^{64} - 1$.

Wyjście Informacja o tym, czy liczba B jest większa od liczby A.

Algorytm Na początku zostaje wywołana funkcja $wyrownaj()$, która wyrównuje liczby A i B względem przecinka. Następnie porównujemy i-te wyrazy ciągu $A[i]$ oraz $B[i]$ ($i = 0 \wedge i \to n$ gdzie $n \equiv max(A.size, B.size)$). Jeżeli w jakimkolwiek momencie $B[i] > A[i]$ oznacza to, że liczba B jest większa od liczby A. Jeżeli taka sytuacja nigdy nie nastąpi oznacza to, że $A > B$ lub $A = B$.

Złożoność obliczeniowa $\theta(n)$ gdzie $n \equiv max(A.size, B.size)$.

Złożoność pamięciowa $\theta(1)$.

4 Przeprowadzone testy

Rozbicie całego zagadnienia na osobne funkcje pozwoliło na szybką analizę problemów i debug całego programu. Oczywistym jest fakt, że nie nie jesteśmy w stanie przeprowadzić testów poprawności dla każdej liczby i kombinacji działań. Sprawdzaliśmy jedynie charakterystyczne i skrajne przypadki dla każdej funkcji: jeżeli będą zwracać poprawne wyniki, możemy wnioskować indukcyjnie, że reszta przypadków będzie również zwracać poprawny wynik, co zostało sprawdzone w testach dla losowych, niecharakterystycznych danych.

5 Możliwe ulepszenia

Mnożenie: Zastosowanie algorytmu Karacuby i uzyskanie złożoności obliczeniowej $\Theta(n^{log_2 3})$ wraz z Szybką Transformatą Fouriera (Algorytm Cooleya-Tukeya)[3].
Ogólne: Przewidywanie nieokreślonych zachowań poszczególnych funkcji[4].
Operacje na bitach oraz wykorzystanie funkcji wbudowanych w kompilator GCC [5].

Literatura

[1] en.wikipedia.org/wiki/Babylonian_mathematics

[2] en.wikipedia.org/wiki/Shunting-yard_algorithm

[3] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L, Clifford Stein, *Wprowadzenie do algorytmów* Rozdział 30.:*"Wielomiany i FFT"*, Wydawnictwo PWN

[4] en.cppreference.com/w/cpp/language/ub

[5] gcc.gnu.org/onlinedocs/gcc/