Politechnika Gdańska

Wydział Elektroniki,
Telekomunikacji i Informatyki

Katedra: Architektury Systemów Komputerowych

Imię i nazwisko dyplomanta: Wojciech Pasternak

Nr albumu: 137361

Forma i poziom studiów: Stacjonarne jednolite studia magisterskie

Kierunek studiów: Informatyka

Praca dyplomowa magisterska

Temat pracy:
Obliczanie zer wielomianów

Kierujący pracą:
dr hab. inż. Robert Janczewski

Zakres pracy:
Obliczanie pierwiastków wielomianów i porównanie służących do tego struktur

Gdańsk, 2016

Spis treści

Rozdział 1

Przegląd literatury

1.1 Wielomiany

1.1.1 Definicja

Definicja 1 *Wielomianem zmiennej rzeczywistej x nazywamy wyrażenie:*

$$\begin{aligned} &W(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x + a_n, \\ &gdzie\ a_0, a_1, a_2, ..., a_{n-1}, a_n \in R, n \in N \end{aligned} \tag{1.1}$$

Liczby $a_0, a_1, a_2, ..., 1_{n-1}, a_n$ nazywamy współczynnikami wielomianu, natomiast n nazywamy stopniem wielomianu.

Szczególnym przypadkiem wielomianu jest jednomian.

Definicja 2 *Jednomianem zmiennej rzeczywistej x nazywamy wielomian, który posiada co najwyżej jeden wyraz niezerowy i określamy wzorem:*

$$W(x) = ax^n, gdzie\ a \in R, n \in N \tag{1.2}$$

Można, więc rozumieć wielomian jako skończoną sumę jednomianów. Jednomianem stopnia zerowego jest stała, pojedyncza liczba rzeczywista, która w szczególności może być zerem.

Definicja 3 *Wielomianem zerowym nazywamy, wielomian wyrażony wzorem:*

$$W(x) = 0 \tag{1.3}$$

W dalszej części, jeżeli nie zaznaczymy inaczej, mówiąc wielomian, będziemy mieli na myśli pewien wielomian, nie będący wielomianem zerowym.

1.1.2 Podstawowe działania na wielomianach

Na wielomianach, tak jak na liczbach możemy wykonywać podstawowe działania. Należą do nich: porównywanie, dodawanie, odejmowanie, mnożenie, dzielenie, a także obliczanie reszty z dzielenia

oraz NWD (największego wspólnego dzielnika). Jako, że wielomian zmiennej x możemy traktować jak funkcję jednej zmiennej, możemy także policzyć z niego pochodne.

Porównywanie wielomianów

Porównywanie należy do najbardziej elementarnych działań na wielomianach. Wymaga ono zwykłego porównania kolejnych współczynników, a jego długość trwania, zależy od ich liczby. Zapoznajmy się z twierdzeniem dotyczącym operacji porównywania wielomianów.

Twierdzenie 1 *Dwa wielomiany uważamy za równe wtedy i tylko wtedy, gdy są tego samego stopnia, a ich kolejne współczynniki są równe.*

Powyższe twierdzenie nie jest złożone, nie mniej w celu pełnego zrozumienia, zilustrujmy je przykładem.

Przykład 1 *Mamy dane wielomian W_1 oraz wielomian W_2.*

$$\begin{aligned} W_1(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \\ W_2(x) &= b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n \end{aligned} \tag{1.4}$$

Wielomiany W_1 oraz W_2 są równe wtedy i tylko wtedy gdy $\forall i \in N \ a_i = b_i$.

Można zauważyć, potencalny wpływ reprezentacji wielomianu na szybkość operacji porównania. Gdy mamy do czynienia z wielomianem, w którym uwzględniamy każdy współczynnik, także gdy jest on zerowy, złożoność czasowa porównania jest liniowa względem stopnia wielomianu. Natomiast w przypadku, gdy pomijamy wszystkie zerowe współczynniki wielomianu, złożoność również jest liniowa, ale tym razem względem liczby niezerowych współczynników wielomianów. Jak widać, w sytuacji, gdy stopień wielomianu jest znacznie większy od liczby zerowych współczynników, reprezentacja wielomianu ma niebagatelne znaczenie.
Dodatkowo, podobnie jak w przypadku porównywania liczb i sprawdzania kolejnych bitów, operacja porównania kończy się w momencie stwierdzenie, że porównywane współczynniki są różne lub porównaliśmy ze sobą już wszystkie współczynniki. Wynika z tego, że zakładając stały czas porównywania dwóch liczb, będącymi współczynnikami wielomianów, operacja porównania różnych wielomianów nigdy nie jest dłuższa od stwierdzenia, że porównywane wielomiany są równe.

Suma wielomianów

Dodawanie to kolejne elementarne działanie na wielomianach, które nie wymaga wykonywania skomplikowanych obliczeń.

Twierdzenie 2 *Aby dodać dwa wielomiany, należy dodać ich wyrazy podobne.*

Podobnie jak w przypadku porównywania czas dodawania wielomianów jest liniony, a ich reprezentacja ma zasadniczy wpływ na liczbę operacji dodawania. Pokażmy zastosowanie powyższego twierdzenia na przykładzie.

Przykład 2 *Mamy dane wielomian W_1 oraz wielomian W_2.*

$$\begin{aligned} W_1(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \\ W_2(x) &= b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n \end{aligned} \tag{1.5}$$

Zdefiniujmy trzeci wielomian: $W_3(x) = W_1(x) + W_2(x)$. Wówczas:

$$W_3(x) = (a_0 + b_0)x^n + (a_1 + b_1)x^{n-1} + ... + (a_{n-1} + b_{n-1})x + a_n + b_n \tag{1.6}$$

Na powyższym przykładzie łatwo zaobserwować, że stopień sumy dwóch wielomianów nie może być większy od większego ze stopni dodawanych wielomianów. Znajduje to potwierdzenie w twierdzeniu, dotyczącym stopnia sumy wielomianów.

Twierdzenie 3

$$\deg(W_1 + W_2) \leq max(deg(W_1),\ deg(W_2)) \tag{1.7}$$

W przedstawionym twierdzeniu, należy uwagę na operator mniejsze równe. W przypadku gdy oba te wielomiany są tego samego stopnia, o przeciwnym współczynniku przy najwyższej potędze, to stopień ten będzie mniejszy.

Różnica wielomianów

Odejmowanie to operacja bliźniacza do dodowania, nie tylko w przypadku liczb, ale także w przypadku wielomianów. By pokazać olbrzymie podobieństwo tych operacji, zacznijmy od zapoznania się z definicją wielomianu przeciwnego.

Definicja 4 *Wielomianem przeciwnym nazywamy wielomian, którego wszystkie współczynniki są przeciwne do danych.*

Spójrzmy na poniższy przykład, pokazujący, że dla każdego wielomianu można bardzo prosto zdefiniować wielomian przeciwny, zmieniając znak wszystkich jego współczynników.

Przykład 3 *Mamy dany wielomian W_1.*

$$W_1(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \tag{1.8}$$

Zdefiniujmy drugi wielomian: $W_2(x) = -W_1(x)$. Wówczas:

$$W_1(x) = -a_0x^n + (-a_1)x^{n-1} + ... + (-a_{n-1})x + (-a_n) \tag{1.9}$$

Wiemy już, czym jest wielomian przeciwny. Przedstawmy teraz twierdzenie mówiące jak odejmować od siebie wielomiany.

Twierdzenie 4 *Aby obliczyć różnicę wielomianów W_1 i W_2, należy dodać ze sobą wielomiany W_1 i $-W_2$, czyli wielomian przeciwny do wielomianu W_2.*

Jak widać, przedstawione twierdzenie potwierdza analogię obliczania róznicy i sumy wielomianów. Spójrzmy na przykład, pokazujący jak obliczać różnicę wielomianów, potrafiając już je do siebie dodawać.

Przykład 4 *Mamy dane wielomian* W_1 *oraz wielomian* W_2*.*

$$\begin{aligned} W_1(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \\ W_2(x) &= b_0x^n + b_1x^{n-1} + ... + b_{n-1}x + b_n \end{aligned} \tag{1.10}$$

Zdefiniujmy trzeci wielomian: $W_3(x) = W_1(x) - W_2(x)$*. Wówczas:*

$$W_3(x) = (a_0 - b_0)x^n + (a_1 - b_1)x^{n-1} + ... + (a_{n-1} - b_{n-1})x + a_n - b_n \tag{1.11}$$

Warto zauważyć, że wielomianem neutralnym ze względu na dodawanie i odejmowanie jest wielomian $W(x) = 0$. Oznacza to, że po dodaniu lub odjęciu wielomianu neutralnego, dostaniemy wynik, będący danym wielomianem.

Iloczyn wielomianów

Mnożenie to kolejne operacja zaliczająca się do podstawowych działań na wielomianach. Jego zasady przypominają nieco zwykłe mnożenie. Dokonujemy przemnożenia odpowiednich wyrazów, z tą różnicą, że w tym przypadku po prostu dodajemy wartości potęg dla odpowiednich współczynników. Zapoznajmy się z twierdzeniem, mówiącym dokładnie jak należy obliczać iloczyn wielomianów.

Twierdzenie 5 *Aby pomnożyć dwa wielomiany, należy wymnożyć przez siebie wyrazy obu wielomianów, a następnie dodać do siebie wyrazy podobne.*

Jak wynika z przedstawionej definicji poza mnożeniem dwóch liczb, mnożenie wielomianów w części polega na redukcji wyrazów podobnych, czyli operacji bazującej na dodawaniu. Spójrzmy na przykład, pokazujący jak definiuje się wielomian, będacy iloczynem dwóch wielomianów.

Przykład 5 *Mamy dane wielomian* W_1 *oraz wielomian* W_2*.*

$$\begin{aligned} W_1(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \\ W_2(x) &= b_0x^m + b_1x^{m-1} + ... + b_{m-1}x + b_m \end{aligned} \tag{1.12}$$

Zdefiniujmy trzeci wielomian: $W_3(x) = W_1(x) * W_2(x)$*. Wówczas:*

$$\begin{aligned} W_3(x) =& (a_0b_0)x^{n+m} + (a_0b_1 + a_1b_0)x^{n+m-1} + (a_0b_2 + a_1b_1 + a_2b_0)x^{b+m-1} + ... \\ &+ (a_{n-2}b_m + a_{n-1}b_{m-1} + a_nb_m)x^2 + (a_{n-1}b_m + a_nb_{m-1})x + a_nb_m \end{aligned} \tag{1.13}$$

Można zauważyć, że po wymnożeniu wszystkich współczynników wielomianu liczba wyrazów iloczynu wynosi $(n+1)(m+1)$. Po dokonaniu redukcji wyrazów podobnych liczba ta ulega zmniejszeniu do wartości $n + m + 2$. Oznacza to zmianę liczby wyrazów z wartości kwadratowej, do

wartości liniowej względem stopni wielomianów. Liczba wyrazów podobnych, po przemnożeniu dwóch wielomianów jest symetryczna względem wykładników potęg poszczególnych współczynników. Można zauważyć, że skrajne wyrazy posiadają tylko po jednym wyrazie potęgi, a zbliżając się do współczynników o środkowych indeksach, liczba ta wzrasta, aż do wartości równej połowie stopnia otrzymanego wielomianu.

Widzimy, że czas operacji mnożenia wielomianów jest kwadratowy, wzgędem stopni mnożonych przez siebie czynników. Należy zauważyć, że jeżeli użyjemy reprezentacji wielomianu, w których posiadamy informację tylko o jego niezerowych współczynników, to czas operacji mnożenie będzie nadal kwadratowy, jednak względem liczby tych współczynników. Dla wielomianów wysokich stopni, w których zaledwie kilka współczynników jest niezerowych różnica ta może być niegatelna i w skrajnych przypadkach czas operacji może zmniejszyć się z kwadratowego, do czasu stałego.

Na podstawie powyższego przykładu, możemy także zaobserwować, że stopień wielomianu, będącego iloczynem dwóch wielomianów niezerowych, jest standardowo równy sumie stopni tych wielomianów. Wyjątkiem jest sytuacja gdy jeden z czynników jest wielomianem zerowym. Wówczas wynik takiej operacji również będzie wielomianem zerowym. Fakt ten znajduje potwierdzenie w poniższym twierdzeniu.

Twierdzenie 6

$$\begin{aligned} &deg(W_1 * W_2) = deg(W_1) + deg(W_2),\ dla\ W_1(x)! = 0, W_2(x)! = 0 \\ &W_3(x) = W_1(x) * W_2(x) = 0,\ w\ pozostałych\ przypadkach \end{aligned} \tag{1.14}$$

Z powyższego twierdzenia można zauważyć, że stopień otrzymanego wielomianu nigdy nie będzie wyższy od dwukrotności większego ze stopni mnożonych wielomianów.

Iloraz wielomianów

Dzielenie to zdecydowanie najtrudniejsza z elementarnych operacji na wielomianach. Aby dobrze zrozumieć jego zasady zapoznajmy się z definicją podzielności wielomianów oraz dzielnika wielomianu.

Definicja 5 *Wielomian W(x) nazywamy podzielnym przez niezerowy wielomian P(x) wtedy i tylko wtedy, gdy istnieje taki wielomian Q(x), że spełniony jest warunek* $W(x) = P(x) * Q(x)$. *Wówczas: wielomian Q(x) nazywamy ilorazem wielomianu W(x) przez P(x), zaś wielomian P(x) nazywamy dzielnikiem wielomianu W(x).*

Bardzo ważnym aspektem obliczania ilorazu wielomianów jest reszta z dzielenia. Spójrzmy na poniższą definicję.

Definicja 6 *Wielomian R(x) nazywamy resztą z dzielenia wielomianu W(x) przez niezerowy wielomian P(x) wtedy i tylko wtedy, gdy istnieje taki wielomian Q(x), że spełniony jest warunek* $W(x) = P(x) * Q(x) + R(x)$.

Łatwo zauważyć analogię w wyżej przedstawionych wzorach. Różnią się one właśnie wielomianem $R(x)$, czyli resztą z dzielenia. Gdy jest ona wielomianem zerowym, to znaczy, że mamy do czynienia z dzieleniem bez reszty i mówimy o podzielności dwóch wielomianów. Spórzmy na przykład, w którym zdefiniowane zostały dwa wielomiany, będące ilorazem i resztą z dzielenia dwóch wielomianów.

Przykład 6 *Mamy dane wielomian W oraz wielomian P.*

$$\begin{aligned} W(x) &= a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n \\ P(x) &= b_0x^m + b_1x^{m-1} + ... + b_{m-1}x + b_m \end{aligned} \tag{1.15}$$

Zdefiniujmy wielomian: $Q(x) = \frac{W(x)}{P(x)}$ *oraz* $Q(x) = W(x)\ mod\ P(x)$. *Wówczas:*

$$\begin{aligned} Q(x) &= c_0x^{n-m} + c_1x^{n-m-1} + ... + c_{n-m-1}x + c_{n-m},\ gdzie\ c_0 \neq 0 \\ R(x) &= d_0x^{n-m-1} + d_1x^{n-m-2} + ... + d_{n-m-2}x + d_{n-m-1} \end{aligned} \tag{1.16}$$

Zwróćmy uwagę na potęgi stojące przy najwyższych potęgach wielomianów $Q(x)$ oraz $R(x)$. Widzimy, że stopień ilorazu wielomianów jest zawsze równy różnicy stopni wielomianów, będących dzielną i dzielnikiem. Najważniejszym aspektem jest jednak fakt, że stopień reszty z dzielenia wielomianów jest zawsze mniejszy od stopnia ilorazu. Nie można natomiast ustalić jego wartości, bez dokładnej znajomości wielomianów W i P. W przykładzie podkreślony został fakt, że współczynnik stojący przy x, o potędze $n - m - 1$ może być zerem. To samo tyczy się także kolejnych współczynników. Gdy wszystkie one są zerami, to znaczy, że mamy do czynienia z resztą, będąco wielomianem zerowym. Oznacza to wówczas, że wielomian W jest podzielny przez wielomian P. Poniżej znajduje się twierdzenie, o stopniach wielomianów, będących ilorazem i resztą z dzielenia.

Twierdzenie 7

$$deg(W_1\ mod\ W_2) < deg(W_1/W_2) = deg(W_1) - deg(W_2) \tag{1.17}$$

Jak widać, twierdzenie potwierdza nasze obserwacje i wnioski dotyczące stopni obu wielomianów.

Pochodna wielomianu

Wielomiany jako przykład funkcji ciągłej, pozwalają na obliczanie pochodnych. By przekonać się, że jest to przykład jednej z prostszych operacji na wielomianach, zapoznajmy się z definicją.

Definicja 7 *Dany jest wielomian W, określony wzorem* $W(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$. *Pochodną wielomianu W nazywamy wielomian W' i wyrażamy wzorem:*

$$W'(x) = na_0x^{n-1} + (n-1)a_1x^{n-2} + ... + 2a_{n-2}x + a_{n-1} \tag{1.18}$$

Widzimy, że powstały wielomian powstał poprzez pomnożenie wartości każdego z współczynników przez stojącą przy danym wyrazie potęgę, a następnie obniżenie jej wartości o jeden. W ten

sposób potęgi wszystkich wyrazów wielomianu obniżają się. Wyjątkiem jest tutaj potęga o wartości zero, czyli stała. Pochodna z funkcji stałej jest zawsze równa zero, dlatego została pominięta w powyższym wzorze. O stopniu pochodnej wielomianu mówi poniższe twierdzenie.

Twierdzenie 8

$$\begin{aligned} &deg(W') = deg(W) - 1,\ dla\ deg(W) > 0 \\ &W(x) = 0,\ w\ pozostałych\ przypadkach \end{aligned} \tag{1.19}$$

Jak widać dla wszystkich wielomianów, nie będących stałą liczbową, stopień ich pochodnej ulega zmniejszeniu o jeden. Czas operacji obliczania pochodnej wielomianu jest porównywalny, z obliczaniem sumy wielomianów, gdyż wystarczy, że dokonamy jednokrotnego obliczania każdego z współczynników.

1.1.3 Dodatkowe twierdzenia dotyczące wielomianów

Istnieje mnóstwo twierdzień dotyczących wielomianów. Zapoznajmy się z tymi, które ułatwią nam znajdowanie pierwiastków wielomianów. Zapoznajmy się z pierwszym twierdzeniem, mówiącym o możliwości zamienienia dowolnego wielomianu o współczynnikach wymiernych, w wielomian o współczynnikach całkowitych.

Twierdzenie 9 *Dowolny wielomian* $W_1(x) = \frac{a_n}{b_n}x^n + \frac{a_{n-1}}{b_{n-1}}x^{n-1} + ... + \frac{a_1}{b_1}x + \frac{a_0}{b_0}$*, o współczynnikach wymiernych, można przekształcić w wielomian* $W_2(x) = k * W_1(x)$*, o współczynnikach całkowitych i tych samych pierwiastkach, co wielomian W. Wówczas:*

$$k = m * NWW(b_0, b_1, ..., b_{n-1}, b_n),\ gdzie\ m \in Z \tag{1.20}$$

Twierdzenie to oznacza, że przy dysponując wyłącznie współczynnikami, będącymi liczbami całkowitymi, jesteśmy w stanie przedstawić dowolny wielomian o współczynnikach wymiernych. Przejdźmy teraz do twierdzenia, mówiącego o wartości wielomianu w danym punkcie.

Twierdzenie 10 *Jeżeli wielomian W(x) podzelimy przez dwumian* $x - x_0$*, to reszta z tego dzielenia jest równa wartości tego wielomianu dla* $x = x_0$*.*

W szczególnym przypadku reszta ta może być równa zero. Oznacza to, że liczba x_0 jest pierwiastkiem tego wielomianu. Wynika z tego bezpośrednio kolejne twierdzenie, znane jako twierdzenie Bezout.

Twierdzenie 11 (Bezout) *Liczba* x_0 *jest pierwiastkiem wielomianu W(x) wtedy i tylko wtedy, gdy wielomian jest podzielny przez dwumian* $x - x_0$*.*

Dodatkowo zapoznajmy się jeszcze z krótkim dowodem poprawności powyższego twierdzenia.

Dowód Jeżeli liczba x_0 jest pierwiastkiem wielomianu W, to wielomian ten możemy wyrazić jako iloczyn dwumianu $x - x_0$ oraz pewnego wielomianu Q: $W(x) = (x - x_0) * Q(x)$. Wyznaczając z

tego wyrażenia wielomian Q, otrzymujemy $Q(x) = \frac{W(x)}{x-x_0}$. Widzimy zatem, że dzieląc wielomian W(x) przez dwumian $x - x_0$, otrzymujemy bez reszty, wielomian Q(x).

Przejdźmy teraz, to twierdzenia mówiącego o pierwiastkach wielokrotnych, bazującego na twierdzeniu Bezout.

Twierdzenie 12 *Liczba x_0 jest pierwiastkiem k-krotnym wielomianu W(x) wtedy i tylko wtedy, gdy wielomian jest podzielny przez $(x - x_0)^k$ i nie jest podzielny przez $(x - x_0)^{k+1}$.*

Dowód poprawności jest analogiczny, jak dla twierdzenia Bezout. Jedyną różnicą jest to, że wielomian W przedstawiamy jako: $W(x) = (x - x_0)^k * Q(x)$. Zapoznajmy się z twierdzeniem, mówiących o możliwej wartości pierwiastków, o ile są one całkowite.

Twierdzenie 13 *Dany jest wielomian $W(x) = x^n + a_{n-1}x^{n-1} + ... + a_1x + a_0$, o współczynnikach całkowitych. Jeżeli wielomian W posiada pierwiastki całkowite, to są one dzielnikami wyrazu wolnego a_0.*

Przejdźmy teraz do twierdzeń mówiących o możliwości rozłożenia wielomianu na czynniki.

Twierdzenie 14 *Każdy wielomian W(x) nie będący wielomianem zerowym jest iloczynem czynników stopnia co najwyżej drugiego.*

Oznacza to, że każdy wielomian stopnia co najmniej trzeciego jest rozkładalny na czynniki. Z powyższego twierdzenia wynika kolejne, mówiące, że rozkład ten jest zawsze jednoznaczny.

Twierdzenie 15 *Niezerowy wielomian, o współczynnikach rzeczywistych, jest jednoznacznie rozkładalny na czynniki liniowe lub nierozkładalne czynniki kwadratowe, o współczynnikach rzeczywistych.*

Oznacza to, że nie da się rozłożyć jednego wielomianu na czynniki, na dwa różne sposoby, tzn. tak, by istniał chociaż jeden czynnik (lub jego proporcjonalny odpowiednik), nie występujący w drugim rozkładzie. Ważnym aspektem, wynikającym z powyższego twierdzenia jest mowa, o tym że nie wszystkie wielomiany da się rozłożyć na czynniki liniowe, o współczynnikach całkowitych. Spójrzmy na pokazujący to przykład.

Przykład 7 *Mamy dany wielomian $W(x) = x^3 - 1$. Rózłóżmy go na czynniki. Wiemy, że pierwiastkiem tego wielomianu jest $x_0 = 1$, zatem możemy przedstawić wielomian W jako iloczyn wielomianu $x - 1$ oraz drugiego wielomianu.*

$$\begin{aligned} W(x) =& x^3 - 1 = x^3 - x^2 + x^2 - x + x - 1 = (x-1)x^2 + (x-1)x + x - 1 = \\ &= (x-1)(x2 + x + 1) \\ &\Delta = 1^2 - 4 * 1 * 1 = 1 - 4 = -3 < 0 - brak\ pierwiastków\ rzeczywistych \end{aligned} \tag{1.21}$$

Jak widać drugi z czynników jest właśnie przykładem nierozkładalnego czynnika kwadratowego, o współczynnikach rzeczywistych.

Powyższy czynnik da się rozłożyć na dwa czynniki liniowe, o współczynnikach zespolonych. W pracy tej będziemy jednak mówić wyłącznie o współczynnikach rzeczywistych, najczęściej zawężając jeszcze zbiór potencjalnych współczynników, do liczb wymiernych. Przejdźmy do kolejnego twierdzenia, wynikającego bezpośrednio z dwóch poprzednich.

Twierdzenie 16 *Każdy wielomian stopnia nieparzystego, ma przynajmniej jeden pierwiastek rzeczywisty.*

Oznacza to, że każdy wielomian stopnia nieparzystego jesteśmy w stanie przedstawić jako iloczyn dwóch czynników, z których jeden jest czynnikiem liniowym, a drugi czynnikiem stopnia parzystego, który z kolei być może da się dalej rozłożyć, na wielomiany, o mniejszych stopniach.

1.2 Eliminacja pierwiastków wielokrotnych

Ważnym aspektem obliczanie zer wielomianów jest eliminacja pierwiastków wielokrotnych. Jest ona niezbędna, by móc skorzystać z metody Sturmy. Zapoznajmy się z twierdzeniem dotyczącym krotności pierwiastków pochodnej wielomianu.

Twierdzenie 17 *Jeżeli liczba jest pierwiastkiem k-krotnym wielomianu W, to jest pierwiastkiem (k-1)-krotnym pochodnej tego wielomianu.*

By lepiej zrozumieć twierdzenie, spójrzmy na poniższy przykład.

Przykład 8 *Mamy dany wielomian* $W(x) = x^3 + 2x^2 + x$. *Obliczmy teraz kolejne pochodne wielomianu W.*

$$\begin{aligned} W'(x) &= 3x^2 + 2 * 2x + 1 = 3x^2 + 4x + 1 \\ W^{(2)}(x) &= 2 * 3x + 4 = 6x + 4 \\ W^{(3)}(x) &= 6 \end{aligned} \tag{1.22}$$

Obliczmy teraz pierwiastki wielomianu W i jego kolejnych pochodnych.

$$\begin{aligned} &W(x) = x^3 + 2x^2 + x = x(x^2 + 2x + 1) = x(x+1)^2 \\ &x_1 = 0,\ k_1 = 1,\ x_2 = -1,\ k_2 = 2 \\ &W'(x) = 3x^2 + 4x + 1 \\ &\Delta = 4^2 - 4 * 3 * 1 = 16 - 12 = 4 \\ &\sqrt{\Delta} = 2 \\ &x_1 = \frac{-4-2}{2*3} = \frac{-6}{6} = -1,\ k_1 = 1,\ x_2 = \frac{-4+2}{2*3} = \frac{-2}{6} = -\frac{1}{3},\ k_2 = 1 \\ &W^{(2)}(x) = 6x + 4 = 6(x + \frac{2}{3}) \\ &x_1 = -\frac{2}{3},\ k_1 = 1 \\ &W^{(3)}(x) = 6 \ - \ \textit{brak pierwiastków} \end{aligned} \tag{1.23}$$

Jak widać powyższy przykład potwierdza zastosowanie przedstawionego twierdzenia. Widzimy, że krotność wszystkich pierwiastków ulega zmniejszeniu o jeden, w kolejnej pochodnej. Dodatkowo możemy zauważyć, że pochodna może zawierać także pierwiastki, których nie miał dany wielomian. Ma to miejsce w przypadku, gdy wielomian, posiada przynajmniej dwa różne pierwiastki. Potwierdza to poniższe twierdzenie.

Twierdzenie 18 *Liczba nowych pierwiastków pochodnej wielomianu W' (takich których nie posiadał wielomian W) jest równa liczbie różnych pierwiastków wielomianu pomniejszonej o jeden.*

By dowieść powyższego twierdzenia przeprowadźmy rozumowanie dowodzące jego poprawności.

Dowód Zdefiniujmy wielomian: $W(x) = (x-x_1)^{k_1} * (x-x_2)^{k_2} * ... * (x-x_{m-1})^{k_{m-1}} * (x-x_m)^{k_m}$. Przyjmijmy, że wielomian W jest stopnia n i posiada m różnych pierwiastków, gdzie $1 \leq m \leq n$, $k_1+k_2+...+k_{m-1}+k_m = n$. Zdefiniujmy wielomian $P(x) = (x-x_1)*(x-x_2)*...*(x-x_{m-1})*(x-x_m)$, w którym każdy z pierwiastków wielomianu W występuję dokładnie jeden raz. Pierwiastków jest m, zatem wielomian P jest stopnia m. Dzieląc wielomian W przez wielomian P, otrzymujemy bez reszty wielomian $Q(x) = (x-x_1)^{k_1-1} * (x-x_2)^{k_2-1} * ... * (x-x_{m-1})^{k_{m-1}-1} * (x-x_m)^{k_m-1}$, który posiada każdy z pierwiastków wielomianu W, krotności pomniejszonej o jeden. Skoro krotność każdego z m pierwiastków uległa zmniejszeniu o jeden, to stopień wielomianu Q w stosunku do wielomianu W zmniejszył się o m. Stopień wielomianu Q jest wieć równy n-m. Na mocy twierdzenia wiemy także, że wielomian Q(x) jest podzielny również przez wielomian W', stopnia n-1. Zatem wielomian W' możemy przedstawić jako iloczyn wielomianu Q oraz innego wielomianu: $W'(x) = Q(x) * V(x)$. Wielomian V, zawiera wszystkie pierwiastki W', których nie posiadał wielomian W. Łatwo policzyć, że stopień wielomianu V jest równy: $\deg(V) = deg(W') - deg(Q) = (n-1) - (n-m) = n-1-n+m = m-1$. Jak widać, właśnie udowodniliśmy, że stopień ten jest równy liczbie różnych pierwiastków wielomianu W pomniejszonej o jeden.

Spróbujmy teraz skorzystać z przedstawionego twierdzenia w przykładzie dokonującym eliminacji pierwiastków wielokrotnych.

Przykład 9 *Dany jest wielomian W, określony wzorem:* $W(x) = x^6 - 6x^4 - 4x^3 + 9x^2 + 12x + 4$.
Dokonajmy eliminacji pierwiastków wielokrotnych dla wielomianu W.
Obliczamy pochodną wielomianu.

$$\begin{aligned} W'(x) &= 6 * x^5 - 4 * 6x^3 - 3 * 4x^2 + 2 * 9x + 12 = \\ &= 6x^5 - 24x^3 - 12x^2 + 18x + 12 = 6(x^5 - 4x^3 - 2x^2 + 3x + 2) \end{aligned} \tag{1.24}$$

Obliczmy teraz resztę z dzielenia wielomianu W przez wielomian W'.

$$\begin{array}{l} x \\ \hline x^6 - 6x^4 - 4x^3 + 9x^2 + 12x + 4 : (x^5 - 4x^3 - 2x^2 + 3x + 2) \\ \underline{- x^6 + 4x^4 + 2x^3 - 3x^2 - 2x} \\ - 2x^4 - 2x^3 + 6x^2 + 10x \end{array} \tag{1.25}$$

Kluczowa jest wartość reszty z dzielenia. Gdyby otrzymana reszta z dzielenia była wielomianem zerowym, to pochodna wielomianu było równocześnie NWD(W, W'). W tym przypadku tak jednak nie jest, więc wykonujemy analogiczną operację z tą różnicą, że nową dzielną jest dotychczasowy dzielnik, a reszta z wielomianu jest nowym dzielnikiem. Tę operację wykonujemy tak długo, jak otrzymana reszta jest niezerowego stopnia. W przypadku gdy jest ona jednocześnie wielomianem zerowym, to podobnie jak wyżej, aktualny dzielnik, jest naszym NWD(W, W'). W przypadku gdy otrzymana reszta jest niezerowym wielomianem stopnia zerowego, to największym wspólnym dzielnikiem wielomianów jest pewna niezerowa stała.

1.3 Twierdzenie Sturma

Przeanalizujmy na początek sposób konstruowania ciągu Sturma dla wielomianu W. Pierwszym wyrazem ciagu jest sam wielomian W. Z kolei drugim wyrazem jest pochodna wielomianu W. Kolejne wyrazy ciągu Sturma wyznaczamy obliczając resztę z ilorazu dwóch poprzednich wyrazów. Dzieje się to do uzyskania pierwszej reszty wielomianu, będącą wielomianem stopnia zerowego. Za każdym razem dzieląc wielomian stopnia n, przez wielomian stopnia m, gdzie $m < n$, mamy gwarancję, że wielomian będacy resztą tego ilorazu będzie stopnia mniejszego niż m. Korzystając z tego faktu, wiemy, że liczba wyrazów ciągu Sturma dla wielomianu W jest niewiększa od $\deg(W)+1$.

Definicja 8 *Ciągiem Sturma dla wielomianu W nazywamy ciąg wielomianów: $X_0, X_1, X_2, ...,$, takich że $X_0 = W$, $X_1 = W'$, a kolejne wyrazy definiuje się jako wielomian przeciwny to reszty z dzielenia dwóch poprzednich wyrazów, przy czym ostatnim wyrazem ciągu Sturma jest pierwszy wielomian stopnia zerowego.*

Wiemy już jak definiować ciąg Sturma. Przeanalizujmy teraz jaki wpływ ma uzyskany ciąg Sturma na obliczanie pierwiastków wielomianu. By to zrobić, zapoznajmy się z kolejną definicją, mówiącą o liczbie zmian znaków ciagu Sturma.

Definicja 9 *Liczbą zmian znaków ciągu Sturma dla wielomianu W(x) w punkcie x, obliczamy zliczając liczbę zmian pomiędzy kolejnymi wyrazami, pomijac te o wartości równej zero w punkcie x. Liczbę zmian znaku w punkcie $x = x0$ definiujemy jako wartość funkcji $Z(x_0)$.*

Wiemy już jak obliczać liczbę zmian ciagu Sturma. Sprawdźmy zatem, jak przekłada się ona na liczbę pierwiastków w danym przedziale.

Twierdzenie 19 *Jeżeli wielomian W(x) nie ma pierwiastków wielokrotnych, to liczba pierwiastków rzeczywistych w przedziale $a < x \leq y$), jest równa $Z(a) - Z(b)$.*

W ogólności twierdzenie Sturma można zastosować dla przedziału $(-\infty, +\infty)$, dopuszczając w ciągu Sturma wartości niewłaściwe $+\infty$ oraz $-\infty$. Wówczas $Z(-\infty)$ będzie oznaczać liczbę zmian znaków w ciągu $W(-\infty), W_1(-\infty), W_2(-\infty), ..., W_m(-\infty)$, zaś $Z(+\infty)$ liczbę zmian w ciągu $W(+\infty), W_1(+\infty), W_2(+\infty), ..., W_m(+\infty)$.

Twierdzenie 20 *Liczba różnych pierwiastków rzeczywistych wielomianu W(x) jest równa* $Z(-\infty)-Z(+\infty)$.

Stosując twierdzenie Sturma dla coraz mniejszych przedziałów możliwe jest wyznaczenie pierwiastków wielomianu z dowolną dokładnością. Sposób ten określony jest mianem metody Sturma. Twierdzenie Sturma jest bardzo mocnym środkowiem używanym do znajdowania pierwiastków w określonym przedziale. Może być używana nie tylko dla wielomianów, ale dowolnej różniczkowalnej funkcji ciągłej. Zapoznajmy się z twierdzeniem, mówiącym o ograniczeniach dotyczących maksymalnej liczby zmian znaków dla Ciąg Sturma.

Twierdzenie 21 *Liczba zmian znaków ciągu Sturma dla wielomianu W(x) jest mniejsza od liczby wyrazów tego ciągu i niewiększa od stopnia wielomianu W(x).*

Można, więc zauważyć, że warunkiem wystarczającym i jednocześnie koniecznym do tego by wielomian W(x) stopnia n, posiadający wyłącznie pierwiastki jednokrotne, posiadał n pierwiastków rzeczywistych jest to by wartość ciągu Sturma wynosiła n dla $x = -\infty$ oraz 0 dla $x = +\infty$.

Warto zauwazyć, że dla liczb dostatecznie dużych, co do wartości bezwzględnej, z uwzględnieniem wartości niewłaściwych $+\infty$ oraz $-\infty$ liczba zmian znaków zależy wyłącznie od współczynnika stojącego przy najwyżej potędze wielomianu. Znajduje to potwierdzenie w poniższym twierdzeniu.

Twierdzenie 22 *Jeżeli współczynnik stojący przy najwyższej potędze wielomianu jest większy od 0, to wartość tego wielomianu jest większa od zera w punkcie* $x = +\infty$ *oraz mniejsza od zera w punkcie* $x = -\infty$. *Z kolei, gdy współczynnik stojący przy najwyższej potędze wielomianu jest mniejszy od 0, to wartość tego wielomianu jest mniejsza od zera w punkcie* $x = +\infty$ *oraz większa od zera w punkcie* $x = -\infty$.

Na podstawie powyższego twierdzenia można zauważyć, że nie ma potrzeby wyliczania wartości wyrazów ciągu sturma dla wartości niewłaściwych, tzn. $x = -\infty$ oraz $x = +\infty$. Dzieje się tak dlatego, że wartość wielomianu nie jest ważna, a istotny jest jedynie znak. Fakt ten ma duży wpływ na optymalizacje obliczeń, gdyż dla wielomianów wysokich stopni ograniczamy się jedynie do sprawdzenia znaku przy najwyższej potędze, pomijąc wykonywanie potęgowania, mnożenia i sumowania kolejnych wyrazów wielomianu. Jeżeli zależy nam, na obliczeniu wartości wielomianu, dla odpowiednio dużych x, będącego ilorazem dwóch wielomianów, możemy skorzystać z poniższego twierdzenia.

Twierdzenie 23 *Dany jest wielomian* $W_1(x) = a_0x^n+a_1x^{n-1}+a_2x^{n-2}+...+a_{n-1}x+a_n$, *gdzie* $a_0 \neq 0$, *stopnia n oraz wielomian* $W_2(x) = b_0x^{n-1} + b_1x^{n-2} + b_2x^{n-3} + ... + b_{n-2}x + b_{n-1}$, *gdzie* $b_0 \neq 0$, *stopnia n-1. Wówczas* $\lim_{x\to+\infty} \frac{W_1(x)}{W_2(x)} = \frac{a_0}{b_0}$ *oraz* $\lim_{x\to-\infty} \frac{W_1(x)}{W_2(x)} = -\frac{a_0}{b_0}$.

Jak widać jesteśmy w stanie ustalić znak, a nawet dokładną wartość, największego współczynnika wielomianu, będącego ilorazem dwóch innych wielomianów, wykonując proste dzielenie dwóch liczb, będących odpowiednio najwyższymi współczynnikami wielomianów - dzielnej i dzielnika.

1.4 Biblioteka Mpir

1.4.1 Podstawowe informacje

Wstęp

Mpir jest przykładem biblioteki napisanej w języku C, pozwalającą operować na liczby o dowolnej wielkości i dokładności. Jej celem było dostarczenie narzędzia dla arytmetyki na liczbach, które nie są wspierane przez podstawowe typy wbudowane w języku C.

Kluczowy jest fakt, że biblioteka ta została zoptymalizowana w taki sposób, by w zależności od potrzeb używać odmiennych algorytmów. Stało się tak dlatego, że algorytm działania na małych liczbach z niewielką precyzją jest prosty i znacznie różni się od wymyślnej metody, używanej w przypadku wielkich liczb, z setkami cyfr po przecinku.

Warto zauważyć, że optymalizacja kodu odbywa się także w zależności od rodzaju procesora, na którym są wykonywane obliczenia. Biblioteka przeznaczona dla procesorów Intel i7 dostarcza inny kod niż dla procesora Pentium 4 lub Athlon. Dzięki temu, z pewnością, działa w sposób bardziej wydajny, kosztem wygody programistów, którzy jej używają. Nie są oni w łatwy sposób zapewnić optymalnego działania programu, niezależnie od sprzętu na którym jest on uruchamiany, co może znacznie utrudnić proces powstawania nowego oprogramowania, bazującego na bibliotece mpir.

Licencja

Biblioteka mpir jest przykładem wolnego oprogramowania, charakteryzującego się kilkoma podstawowymi założeniami. Pierwszą z nich jest wolność pozwalająca użytkownikowi na uruchamianie programu, w dowolnym celu. Drugą jest jego analiza oraz dostosowywania go do swoich potrzeb. Kolejną cechą jest możliwość dowolnego rozpowszechnianie kopii programu. Ostatnią zaś jest udoskonalanie programu i publiczne rozprowadzanie własnych ulepszeń, z których każdy będzie mógł skorzystać.

Nazewnictwo i typy

Biblioteka mpir posiada odpowiedniki dla wszystkich podstawowych typów liczbowych, które są charakterystyczne dla języka C. Odpowiednikiem typów całkowitych jest typ mpz_t, zaś odpowiednikiem liczb zmiennoprzecinkowych, typu float, jest mpq_t. Dla liczb całkowitych możemy łatwo zdecydować, czy interesuje nas liczba ze znakiem, czy bez niego. Oba przypadki posiadają prawie identycznie nazwane funkcje. Jedyną różnicą jest to, że dla zmiennych ze znakiem występuje przyrostek si, od wyrazu angielskiego wyrazu signed, a dla zmiennych bez znaku przyrostek ui, od wyrazu unsigned. Typ mpq_t daje użytkownikowi możliwość zdefiniowana dowolnej liczby, dającej się przedstawić w postaci ułamka, czyli dowolnej liczby wymiernej. Ograniczenie to spowodowane jest faktem, iż liczba taka składa się z dwóch liczb typu mpz_t. Pierwsza z nich jest licznikiem, a druga mianownikiem.

Funkcje

W bibliotece przyjęto konwencję charakterystyczną, dla języka C, polegającą na tym, że zarówno argumenty wejściowe jak i wyjściowe funkcji są przekazywane za pomocą referencji do funkcji, która zazwyczaj jest typu void, czyli nie zwraca żadnej wartości. Warto zauważyć, że argumenty wyjściowe są podawane przed wejściowymi. Stało się tak poprzez zachowanie analogii do operatora przypisania, w którym to zmienna wynikowa znajduje się po lewej stronie. W bibliotece mpir zmienna po zadeklarowaniu nie jest natychmiastowo gotowa do użycia, ponieważ wymaga inicjalizacji, przydzielającej jej miejsce w pamięci. Możemy to zrobić poprzez wywołanie odpowiedniej funkcji init, w argumencie podając odpowiednią zmienną. Analogicznie, aby zwolnić zajmowany, przez zmienną, obszar pamięci należy wywołać funkcję clear. Istnieją także funkcje realokujące, czyli zwiększające lub zmniejszające, zajmowany obszar. Użycie ich jest jednak opcjonalne, gdyż zmiana rozmiaru jest automatycznie wykrywana. Raz zainicjalizowaną zmienną, można używać dowolną liczbę razy. Twórcy biblioteki zalecają unikanie nadmiernego korzystania z funkcji init i clear, ponieważ są to operacje czasochłonne i zbyt duża liczba ich wywołań może mieć negatywny wpływ na wydajność naszego programu.

Zarządzanie pamięcią

Biblioteka mpir, alokując w pamięci nowe obiekty, stara się minimalizować wielkość, zajmowaną przez nie, miejsca. Dodatkowa pamięć jest przyznawana dopiero, kiedy jej aktualna wielkość jest niewystarczająca. W przypadku zmiennych typów mpz_t i mpq_t, raz zwiększony rozmiar danej zmiennej nigdy nie zostanie zmniejszony. Zazwyczaj jest to najlepsza praktyka, ponieważ częsta alokacja pamięci ma niekorzystny wpływ na wydajność programu. Jeżeli aplikacja potrzebuje zmniejszyć zajmowane przez daną zmienną miejsce, można dokonać realokacji albo całkowicie zwolnić, zajmowane przez nią miejsce. Dodatkowo kolejne zwiększania rozmiaru przeznaczonego na poszczególne zmienne może powodować zauważalny spadek wydajności lub fragmentację danych. Nie da się uniknąć, jeżeli chcemy zaalokować więcej miejsca dla danej zmiennej, w momencie, gdy w jej sąsiedztwie znajdują się już inne zmienne. Wówczas zależnie od implementacji mamy do czynienia z jednej z dwóch wariantów. W pierwszym przypadku, alokujemy nowe miejsce w pamięci, uzupełniając odpowiednio jego wartość, a następnie zwalniając wcześniej zajmowane miejsce. Alternatywą jest pozostawienie zajmowanej już pamięci i przydzielenie dodatkowej w innym miejscu. W takim przypadku jesteśmy zmuszeni do odpowiedniego zarządzania nieciągłymi fragmentami pamięci, co również może mieć bardzo negatywny wpływ na wydajność aplikacji. Z kolei, zmienne typu mpf_t mają niezmienny rozmiar, zależny od wybranej precyzji.
Warto zauważyć, że biblioteka standardowo do alokowania pamięci domyślnie używa funkcji malloc, ale możliwe jest także korzystanie z funkcji alloca. Pierwszy alokator zajmuje się tylko przydzieleniem pamięci, a użytkownik musi sam pamiętać, o jego zwolnieniu, w odpowiednim momencie. Z kolei, drugi z nich zdejmuje tę odpowiedzialność ze strony programisty i sam dba o zwolnienie pamięci, gdy jest ona już nie używana.

Kompatybilność

Poza kilkoma wyjątkami, biblioteka mpir jest kompatybilna z odpowiednimi wersjami biblioteki gmp. Dodatkowo, jej twórcy starają się nie usuwać istniejących już funkcji. W przypadku, gdy któraś przestanie być rekomendowana, zostaje po prostu zaznaczona jako przestarzawała, ale jej implementacja nie przestaje istnieć w kolejnych wydaniach biblioteki. Dzięki temu program, bazujący na starszej wersji biblioteki, zadziała też na nowszej edycji.

Wydajność

Dla małych liczb, narzut na korzystanie z biblioteki może być znaczący w porównaniu do typów prostych. Jest to nieuniknione, ale celem biblioteki jest próba znalezienia złotego środka pomiędzy wysoką wydajność, zarówno dla małych, jak i dużych liczb.

Operacje w miejscu

Operacje obliczania wartości bezwzględnej i negacji danej liczby są bardzo szybkie, gdy obliczane są w miejscu, tzn. wtedy, gdy zmienna wejściowa jest również zmienną wyjściową. Wówczas nie ma potrzeby alokacji i zwalnia pamięci, a cała funkcja sprowadza się do ustawienia odpowiedniego bitu, mówiącego o znaku liczby. Według specyfikacji biblioteki mpir, zauważalny powinien być także zysk, w przypadku operacji dodawania, odejmowania i mnożenie w miejscu. W momencie, gdy drugim argumentem jest nieduża liczba całkowite operację te są nieskomplikowane i bardzo szybkie.

1.4.2 Instalacja

Instalacja biblioteki mpir jest różna w zależności od systemu operacyjnego. W systemach unixowych instalacja polega na zbudowaniu i instalacji, korzystając ze źródeł. Pod windowsem jest ona równie łatwa i przebiega analogicznie, o ile używamy cygwina lub mingw. Są to narzędzia, które zapewniają, programom działającym pod systemem Windows, funkcjonalność przypominającą system Linux. Bardziej skomplikowane będzie użycie biblioteki bez wyżej wymienionych narzędzi, ale warto zaznaczyć, że biblioteka ta może być budowana z użyciem Microsoft Visual Studio, począwszy od wersji 2010, korzystając z programu o nazwie yasm asembler. Dodatkowo korzystanie z biblioteki różni się w zależności od tego czy potrzebujemy jej statyczną (mpir/lib) czy dynamiczną (mpir/dll) wersję.

Aby używać biblioteki mpir w naszym programie musimy do naszego programu dodać następującą linię:

```
#include <mpir.h>
```

Pozwoli ona nam na używanie wszystkich funkcji i typów, które udostępnia dla nas biblioteka. Dodatkowo wymagana jest kompilacja z dołączeniem naszej biblioteki, poprzez dodanie opcji -lmpir, np.:

```
gcc myprogram.c -lmpir
```

Jeżeli chcemy skorzystać z biblioteki, wspierającej język C++ dodatkowo musimy dodać także opcję -mpirxx, np.:

```
g++ myprogram.cc -lmpirxx -lmpir
```

1.4.3 Operacje na liczbach całkowitych

W niniejszym podrozdziale, omówię podstawowe funkcje liczbowe, które mają zastosowanie, zarówno dla liczb całkowitych typu mpz_t, jak i rzeczywistych typu mpq_t.

Funkcje inicjalizujące

```
void mpz_init (mpz_t integer)
```

Alokuje w pamięci miejsce na zmienną integer i ustawia jej wartość na 0.

```
void mpz_clear (mpz_t integer)
```

Zwalnia miejsce zajmowaną przez zmienną. Funkcja ta powinna być używana dla każdej zmiennej, w momencie, gdy nie ma już potrzeby, by z niej korzystać.

```
void mpz_realloc2 (mpz_t integer, mp bitcnt_t n)
```

Zmienia rozmiar zajmowanego przez zmienną miejsca. Funkcja jest używana z wartością n, większą od aktualnej, by zagwarantować zmiennej określone miejsce w pamięci. Z drugiej strony, zostaje wywołana z wartością mniejszą, jeżeli chcemy zmniejszyć liczbę zajmowanego przez zmienną miejsca. Gdy nowy rozmiar jest wystarczający by pomieścić aktualną wartość zmiennej, to zostaje ona zachowana. W przeciwnym razie zostanie ona ustawiona na 0. Istnieje również przestarzała funkcja mpz_realloc, ale jej użycie jest niezalecane. Nie została usunięta, by zachować kompatybilność wsteczną.

Funkcje przypisania

```
void mpz_set (mpz_t rop, mpz_t op)
```

Pozwala na przypisanie wartości tych samów typów. Wartość zmiennej op jest ustawiana jako rop. Jest to równoważnik operatora przypisania.

```
void mpz_set_sx (mpz_t rop, intmax_t op)
```

Pozwala na przypisanie zmiennej typu całkowitego do wartości typu mpz_t.

```
void mpz_set_d (mpz_t rop, double op)
```

Zapisuje wartość typu double do zmiennej typu mpz_t.

```
void mpz_set_q (mpz_t rop, mpq_t op)
```

Pozwala na zrzutowanie liczby rzeczywistej, typu mpq_t do zmiennej mpz_t. Gdy wartość mpq_t nie jest liczbą całkowitą, zostaje ona zaokrąglona w dół, poprzez obcięcie jej części ułamkowej.

```
int mpz_set_str (mpz_t rop, char *str, int base)
```

Konstruuje liczbę typu mpz_t, na podstawie podanego łańcucha znaków, reprezentującego daną wartość. Zmienna base mówi o podstawie podanej liczby. Dopuszcza się występowanie białych znaków, które są ignorowane. Z kolei zmienna base dopuszcza 0 oraz wartości z zakresu (2; 61). Gdy jest ona równa 0, podstawowa zostaje ustalona, bazując na początkowych znakach. Dla prefixów 0x oraz 0X, reprezentacja liczby zostaje ustalona jako szesnastkowa. Gdy jest ona równa 0b lub 0X, to liczba ta jest binarna, a gdy rozpoczyna się od zera, a drugi znak jest inny od wymienionych, to zostaje uznana za liczbę oktalną. W pozostałych wypadkach jest to liczba dziesiętna. Kolejne wartości dziesiętne od 10 do 35 są reprezentowane jako litery od a do z, przy czym nie ma rozróżnienia ze względu na wielkość liter. W przypadku wyższych wartości podstawy, wielkość liter ma znaczenia, przy czym liczby od 10 do 35 reprezentowane są przez duże litery, a liczby od 36 do 61 przez małe. Funkcja dokonuje weryfikacji, czy podany ciąg znaków w całości reprezentuje poprawną liczbę. Jeżeli tak, to zwraca ona wartość 0. W przeciwnym wypadku jest to 1.

```
void mpz_swap (mpz_t rop1, mpz_t rop2)
```

Używana jest do zamiany wartości pomiędzy dwoma zmiennymi typu mpz_t. Jej użycie jest rekomendowane, ze względów wydajnościowych. W przypadku braku tej funkcji, potrzebaby zmiennej tymczasowej. W tym celu musiała by ona na początku zostać zaalokowana w pamięci, a na końcu zwolniona. Obie operacje są czasochłonne i zaleca się minimalizować ich użycie, więc użycie zoptymalizowanej funkcji służącej do zamiany wartości dużych liczb wydaje się zdecydowanie najlepszą i najszybszą opcją.

Funkcje konwersji

```
intmax_t mpz_get_sx (mpz_t op)
```

Zwraca zmienną typu int, a znak liczby zostaje przepisany. Jeżeli wartość jest poza zakresem liczb typu int, to rzutowanie następuje poprzez pozostawienie najmniej znaczącej części. Powoduje to, że funkcja w wielu przypadkach może okazać się bezużyteczna.

```
double mpz_get_d (mpz_t op)
```

Pozwala zrzutować typ mpz_t na zmienną typu double. Jeżeli jest to konieczne, stosowane jest zaokrąglenie. W przypadku, gdy eksponent jest za duży, zwrócony wynik jest zależny od danego systemu. Jeżeli jest dostępna, zwrócona może być wartość nieskończoności.

```
char * mpz_get_str (char *str, int base, mpz_t op)
```

Zwraca ciąg znaków reprezentujących liczbę op., o podstawie danej w paremetrze o nazwie base. Funkcja ta jest analogiczna do mpz_set_str. Jeżeli parametrem str jest NULL, to ciąg

znaków zostaje zwrócony przez funkcję. W przeciwnym razie funkcja umieszcza go pod adresem, na który wskazuje zmienna str. Należy zadbać o to, by bufor, do zapisania rezultatu funkcji, był wystarczający. Powinien on być 2 bajty dłuższy niż długość zwróconej liczby w danym systemie liczbowym. Pierwszy bajt służy na wpisanie ewentualnego znaków minus, natomiast drugi na znak ‘
0’ kończący łańcuch znaków. Długość zasadniczej części liczbowej można pobrać używając funkcji mpz_sizeinbase (op, base), w której podajemy daną liczbę oraz podstawę.

Funkcje arytmetyczne

```
void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość rop jako sumę op1 i op2.

```
void mpz_sub (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość rop jako różnicę op1 i op2.

```
void mpz_mul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Ustawia wartość rop jako iloczyn op1 i op2.

```
void mpz_addmul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Oblicza wartość iloczynu czynników op1 i op2, a następnie zwiększa rop o otrzymany wynik. Tożsame z wykonaniem działania rop = rop + op1 * op2.

```
void mpz_submul (mpz_t rop, mpz_t op1, mpz_t op2)
```

Oblicza wartość iloczynu czynników op1 i op2, a następnie zmniejsza rop o otrzymany wynik. Tożsame z wykonaniem działania rop = rop - op1 * op2.

```
void mpz_neg (mpz_t rop, mpz_t op)
```

Jako rezultat ustawia liczbę przeciwną do danej. Gdy rop i op są tą samą zmienną to mamy do czynienia z przykładem operacji w miejscu. Wówczas całe funkcja jest bardzo szybka, gdyż opiera się tylko na zmianie bitu, mówiącego o znaku danej liczby.

```
void mpz_abs (mpz_t rop, mpz_t op)
```

Funkcja w swoim działaniu jest bardzo podobna do funkcji mpz_neg. Jedyną róznicą jest to, że bit znaku nie zostaje zanegowany, a ustawiony tak, by wskazywał na wartość nieujemną.

Funkcje dzielenia

Uzywając niżej wymienionych funkcji należy pamiętać, ze dzielenie przez 0, zarówno w przypadku obliczania wartości ilorazu, jak i reszty z dzielenia jest nielegalne. W przypadku, gdy wystąpi taka sytuacja, biblioteka zachowa się jak w przypadku dzielenia przez 0, dla liczb typu int, tzn. skończy działanie funkcji, rzucając odpowiedni wyjątek.

```
void mpz_tdiv_q (mpz_t q, mpz_t n, mpz_t d)
```

Funkcja oblicza iloraz z liczb n i d. Litera przed ‘div’ w nazwie funkcji mówi, o jej zachowaniu w przypadku, gdy rezultat nie jest liczbą całkowitą. Symbol ‘t’ (’truncate’), oznacza obcięcie części ułamkowej. Zalecanym użyciem jest przypadek, gdy nie ma znaczenia wartość reszty z dzielenia.

```
void mpz_tdiv_r (mpz_t r, mpz_t n, mpz_t d)
```

Oblicza wartość rezsty z dzielenia liczb n i d. Podobnie jak w fukcji mpz_tdiv_q, mamy do czynienia z przypadkiem obcięcia ewentualnej części ułamkowej ilorazu. Oznacza to, że zwracana reszta zawsze będzie tego samego znaku co zmienna n. Rekomendowana, gdy wartość ilorazu z dzielenia nie jest istotna.

```
void mpz_tdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)
```

Łączy w sumie dwie powyższe funkcje. Ze względów wygody i wydajności, zaleca się jej użycie wówczas, gdy potrzebujemy obliczyć zarówno iloraz, jak i resztę z dzielenia. Ważne jest by pamiętać, że przekazane argumenty q i r, muszą być referencjami do różnych zmiennych. W przeciwnym razie, działanie funkcji będzie nie poprawne, a wynik najprawdopodobniej błędny.
Oprócz wyżej wymienionych funkcji, istnieją także ich odpowiedniki, o odmiennym zachowaniu w kwestii zaokrąglania części ułamkowej ilorazu. Pierwszym z nich jest grupa funkcji, posiadająca w nazwie ’cdiv’, gdzie litera ‘c’ (’ceil’) oznacza sufit. Powoduje to, że w razie potrzeby, iloraz zostanie zaokrąglony w górę. Wówczas otrzymana reszta r będzie przeciwnego znaku do liczby d. Funkcjami z tej grupy są: mpz_cdiv_q, mpz_cdiv_r oraz mpz_cdiv_qr. Drugim odpowiednik jest grupa funkcji, posiadająca w nazwie ’fdiv’, gdzie ‘f’ (’floor’) oznacza podłogę. W tym przypadku, w razie potrzeby wynik zostanie zaokrąglony w dół. Reszta r będzie w tym przypadku tego samegu znaku co liczba d. Przykładami funkcji z tej grupy są: mpz_fdiv_q, mpz_fdiv_r oraz mpz_fdiv_qr. Zauważmy, że we wszystkich trzech przypadkach zostaje spełnione równanie: $n = q * d + r,\ gdzie\ 0 \leq |r| < |d|$.

```
int mpz_divisible_p (mpz_t n, mpz_t d)
```

Funkcja sprawdza, czy d dzieli liczbę n. Jeżeli nie, to zwracana 0, w przeciwnym razie zwraca wartość różną od zera.

```
void mpz_divexact (mpz_t q, mpz_t n, mpz_t d)
```

Oblicza iloraz z dzielenia liczb n przez d. Działa poprawnie tylko w przypadku, gdy n jest podzielna przez d. By to sprawdzić można użyć funkcji mpz_divisable_p. Jest znacznie szybsza niż pozostałe funkcje, umożliwiające obliczanie ilorazu z dzielenia. Dlatego zawsze, gdy wiemy, że spełniony jest powyższy warunek, użycie tej funkcji jest wysoko rekomendowane.

Funkcje porównania

```
int mpz_cmp (mpz_t op1, mpz_t op2)
```

Porównuje wartości dwóch liczb. Zwraca wartość dodatnią, gdy $op1 > op2$, ujemną, gdy $op1 < op2$, oraz 0, gdy obie wartości są równe.

```
int mpz_cmpabs (mpz_t op1, mpz_t op2)
```

Porównuje wartość bezwzględną dwóch liczb. Zwraca wartość dodatnią, gdy $|op1| > |op2|$, ujemną, gdy $|op1| < |op2|$, oraz 0, gdy obie wartości są równe lub przeciwne.

```
int mpz_sgn (mpz_t op)
```

Określa znak danej liczby. Zwraca 1 dla wartości dodatnich, -1 dla ujemnych oraz 0 w przypadku zera. Należy zwrócić uwagę, ze w obecnej implementacji mpz_sgn jest makrem i rozpatruje dany argument wielokrotnie.

Pozostałe funkcje

```
void mpz_pow_ui (mpz_t rop, mpz_t base, mpir_ui exp)
```

Podnosi liczbę base do potęgi exp.

```
void mpz_sqrt (mpz_t rop, mpz_t op)
```

Oblicza pierwiastek całkowity dla danej liczby. W przypadku, gdy wynik nie jest liczbą całkowitą, część ułamkowa zostaje obcięta.

```
void mpz_sqrtrem (mpz_t rop1, mpz_t rop2, mpz_t op)
```

Jest bardzo podobna do funkcji mpz_sqrt. Poza obliczeniem wartości pierwiastka z danej liczby, zwraca także różnicę pomiędzy daną liczbą, a kwadratem tego pierwiastka. Wartości zwracane można określić równaniem:

$$\begin{aligned} Rop1 &= \sqrt{op} \\ Rop2 &= op - (\sqrt{rop1})^2 \end{aligned} \tag{1.26}$$

Zauważmy, że jeżeli rop2 = 0, to rop1 jest pierwiastkiem kwadratotowym z op.

1.4.4 Operacje na liczbach wymiernych

W tym podrozdziale, rozpatrzę funkcje dla zmiennych typu mpq_t. Skupię się w nim na funkcjach, które nie występują dla liczb całkowitych typu mpz_t, opisanych w poprzednim podrozdziale lub ich implementacja znacząco się różni, od już przedstawionej. Na początku warto zaznaczyć, że zmienne mpq_t reprezentują wyłącznie liczby wymierne. Jest to spowodowane ich implementacją. Każda liczba typu mpq_t składa się z dwóch liczb całkowitych typu mpz_t, reprezentujących licznik i mianownik.

```
void mpq_init (mpq_t dest_rational)
```

Funkcja alokuje miejsce w pamięci i inicjuje wartość danej liczby, ustawiając licznik na 0 i mianownik na 1.

```
void mpq_canonicalize (mpq_t op)
```

Znajduje wspólne dzielniki licznika i mianownika, skracając ułamek. Dodatkowo zapewnia, że mianownik jest liczbą dodatnią, w razie potrzeby, mnożąc licznik i mianownik przez liczbę -1.

```
int mpq_set_str (mpq_t rop, char *str, int base)
```

Tworzy liczbę na podstawie wartości podanej jako łańcuch znaków. Jej wartość podawana jest w postaci najpierw licznik, a następnie mianownik, na zasadach podobnym jak w funkcji mpz_set_str. Obie liczby oddziela znak operatora dzielenia '/'. Funkcja zwraca wartość 0, w przypadku, gdy cały łańcuch wejściowy jest poprawny oraz -1 w innym przypadku. Jeżeli argument base jest równy 0, to format liczbowy, dla licznika i mianownika, jest ustalany osobno. Oznacza to, że możemy podać liczby w dwóch różnych formatach, np. „0xFF/256" . Warto zauważyć, że funkcja mpq_canonicalize nie jest wołana automatycznie, więc jeżeli podaliśmy ułamek, który chcemy skrócić, to musimy pamiętać o jej wywołaniu.

```
char * mpq_get_str (char *str, int base, mpq_t op)
```

Funkcja zwracająca liczbę w postaci ułamka, tzn. licznik i mianownik, oddzielony znakiem kreski ułamkowej. Jest analogiczna do funkcji mpz_get_str. Bufor wyjściowy, w którym chcemy umieścić rezultat funkcji powinien być, wystarczający i wynosić $mpz\_sizeinbase(mpq\_numref(op), base)+ mpz\_sizeinbase(mpq\_denref(op), base) + 3$. Jeden dodatkowy bajt w porównaniu do funkcji mpz_get_str, spowodowany jest koniecznością umieszczenia w buforze kreski ułamkowej.

Rozdział 2

Opis rozwiązania

2.1 Podział na moduły

Aplikacja została podzielona na trzy główne moduły: statyczna biblioteka, aplikacja konsolowa oraz testy jednostkowe. Każdy z nich został zawarty w osobnym projekcie, stworzonym w programie Microsoft Visual Studio 2015. Moduły te zostały ze sobą odpowiednio powiązane, razem tworząc solucje. Pomiędzy modułami występują jasno określone zależności. Aplikacja konsolowa i projekt testowy są od siebie całkowicie niezależne, więc zmiany i błędy w jednym z nich, nie mają wpływu na drugi. Obie te części zależą od biblioteki statycznej, bezpośrednio się do niej odwołując. Aplikacja konsolowa udostępnia interfejs użytkownika do wspomnianej funkcjonalności, a framework testowy weryfikuje jej poprawność. Dodatkowo oba projekty są zależne od biblioteki mpir, która musi być dołączana w sposób dynamiczny, poprzez dodanie odpowiedniego pliku typu dll.

2.1.1 Statyczna biblioteka

Statyczna biblioteka zawiera wszystkie logiczne aspekty projektu, posiada odpowiednie klasy, zdefiniowane funkcje i zaimplementowane algorytmy, gotowe do wykorzystania i przetestowania. To w tej częsci projektu są zaprojektowane klasy pomocniczne - CharsConstants i StringManager oraz klasa Number, pozwalająca na łatwe korzystanie z dużych liczb wymiernych. Ta ostatnia jest warstwą pośrednią pomiędzy biblioteką mpir a pozostałymi częściami projektu. Klasa ta, jako jedyna zależy w bezpośredni sposób od implementacji wspomnianej biblioteki. Innymi słowy, nawet w przypadku bardzo radykalnych zmian w bibliotece mpir, jest to jedyna klasa, wymagająca zmiany. Jest to bardzo wygodny i bezpieczny sposób, na całkowite uniezależnienie pozostałej części od implementacji innego projektu, na którego kierunek rozwoju programista pozostaje bez wpływu.

Moduł zawiera także najważniejszą część projektu, czyli klasy umożliwiające wykonanie działań na wielomianach. Istnieją dwie takie klasy - PolynomialMap i PolynomialVector, używajace odpowiednio mapy i tablicy, do przetrzymywania struktury wielomianów. Dla obu takich klas została zdefiniowana klasa abstraktycjna Polynomial, która implementuje wspólne funkcja, a także

definiuje interfejs dla klas pochodnych. Działanie obu klas jest analogiczny, ale zostało zaimplementowane w odmienny sposób, co ma zasadniczny wpływ na wydajność rozwiązania, dla konkretnych typów wielomianów.

Część ta jest zależna od odpowiednio skompilowanej biblioteki mpir. W przeciwieństwie do aplikacji konsolowej i frameworku testowego, wystarczająca jest jej statycznie dołączana wersja. Warto zaznaczyć, że wersja ta jest inna dla różnych komputerów. Jest ona zależna od rodzaju procesora, ponieważ jej kod asemblerowy został zoptymalizowany, pod jego konkretny typ. Może to powodować różnice w wydajności biblioteki, ale nie powinno mieć żadnego wpływu na jej deterministyczne zachowanie. Dlatego jej działanie zawsze powinno być przewidywalne i takie samo, niezależne od rodzaju platformy, na której jest używana.

2.1.2 Aplikacja konsolowa

Aplikacja konsolowa pozwala, w łatwy, dla użytkownika, sposób, na tworzenie wielomianów i obliczanie wartości ich pierwiastków w zadanym przedziale. Użytkownik podaje odpowiednie dane wejściowe z linii poleceń, a te są następnie przetwarzane. Otrzymane wyniki są wypisywane na standardowe wyjście w przystępny sposób, tzn. w systemie dziesiętnym i z określoną precyzją.

Warto zaznaczyć, że użytkownik może wprowadzać wielomiany w dowolny sposób. Mogą być one, zarówno w postaci sumy kolejnych współczynników, jak i skomplikowanych iloczynów, składających się z poszczególnych wielomianów. Jedynym ograniczeniem jest to, by wyrażenie było poprawne składniowo oraz zawierało wyłącznie całkowite współczyniki i naturalne potęgi. Nie powinno być to jednak przeszkodą, w obliczaniu pierwiastków dla skomplikowanych wielomianów. Należy bowiem pamiętać, że kazdy wielomian o współczynnikach wymiernych, a tylko takie będziemy rozwazać w niniejszej pracy, da się przedstawić przy pomocy proporcjonalnych liczb całkowitych. Odpowiednie ułamki zwykłe, zawsze możemy w ten sposób zamienić, wyciągając z danego wielomianu czynnik, będący odwrotnością najmniejszej wspólnej wielokrotności, wszystkich współczynników. Jako, że czynnik ten będzie stałą, to przy obliczaniu pierwiastków, można go po prostu pominąć, gdyż nie ma on na nie żadnego wpływu.

Użytkownik może korzystać z aplikacji tak długo, jak będzie sobie tego życzył. Zapytanie o kolejne wielomiany przeprowadzonę są w pętli i trwają, dopóki użytkownik nie wprowadzi ciągu znaków, odpowiedzialnego za wyjście z programu. Szczegółowe informację, o tym jak korzystać z aplikacji są podane w jego instrukcji.

2.1.3 Framework testowy

Testy jednostkowe umożliwiają użytkownikowi sprawdzenie, czy określona funkcjonalność działa bezbłędnie. Zgodnie z ich założeniami, są w nich zdefiniowe podstawowe testy, weryfikujące różne przypadki testowe. W tym przypadku jest nieco inaczej, ponieważ w postaci testów jednostkowych, zostały zawarte również bardziej skomplikowane testy funkcjonalne. Wśród nich znajdują się też takie, które testują główne zadanie projektu, czyli dla wielomianu, tworzonego na podstawie otrzymanego wejścia, w postaci łańcucha znaków, potrafią, z określoną precyzją, znaleźć jego wszystkie

pierwiastki rzeczywiste.

Jako środowisko testowe wybrany został framework wbudowany w narzędzie Microsoft Visual Studio 2015. Rozważana była też biblioteka o nazwie Google Test. Jej istotną zaletą, była przenośność, pomiędzy platformami, ale ostatecznie koncepcja ta została odrzucona. Spowodowane było to faktem, iż cały projekt pisany był w Visual Studio, a łatwość i wygoda, uruchamiania w nim testów jednostkowych jest bardzo duża. Dzięki temu wyborowi, każda, nawet najmniejsza zmiana, mogła być bardzo szybka przetestowana, bez potrzeby wprowadzania jakichkolwiek dodatkowych zmian, co znacznie przyspieszało proces implementacji i weryfikacji przyjętego rozwiązania.

2.2 Główne klasy

2.2.1 CharsConstants

Klasa CharsConstants została zaimplementowana w celu zapewnienia większej czytelności kodu. Ważnym aspektem było zdefiniowane i łatwe rozpoznawanie wszystkich, mogących występować w łańcuchu wejściowym, legalnych znaków. Takie podejście ogranicza możliwość popełnienia prostych, a trudnych do wykrycia błędów w przetwarzaniu otrzymanego tekstu, np. literówki. W przypadku, gdy są zdefiniowane odpowiednio stałe i funkcji, ewentualna literówka, skończy się zasygnalizowaniem błędu już na etapie kompilacji. Przypadek taki jest sygnalizowany i łatwy do naprawienia, dzięki czemu programista nie musi zastanawiać się dlaczego, poprawny, wydałoby się, kod nie działa. Poniżej przedstawiam strukturę klasy, wraz ze zdefiniowanymi w niej stałymi i statycznymi metodami.

```
class CharsConstants
{
        static const char Space = ' ';
        static const char Tab = '\t';
        static const char NewLine = '\n';
        static const char LeastDigit = '0';
        static const char GreatestDigit = '9';
        static const char LeastUppercase = 'A';
        static const char GreatestUppercase = 'Z';
        static const char LeastLowercase = 'a';
        static const char GreatestLowercase = 'z';

public:
        static const char Plus = '+';
        static const char Minus = '-';
        static const char Mul = '*';
        static const char Div = '/';
        static const char Exp = '^';
        static const char OpeningParenthesis = '(';
```

```
        static const char ClosingParenthesis = ')';
        static const char Var = 'x';

        static int CharToInt(char c);
        static bool IsDigit(char c);
        static bool IsLetter(char c);
        static bool IsUppercase(char c);
        static bool IsLowercase(char c);
        static bool IsWhitespace(char c);
        static bool IsPlus(char c);
        static bool IsMinus(char c);
        static bool IsMul(char c);
        static bool IsDiv(char c);
        static bool IsExp(char c);
        static bool IsOpeningParenthesis(char c);
        static bool IsClosingParenthesis(char c);
        static bool IsVar(char c);
        static bool IsOperator(char c);
        static bool IsLegalValue(char c);
        static bool IsLegalOpeningOperator(char c);
};
```

2.2.2 StringManager

Klasa StringManager oferuje przydatne funkcje, na łańcuchach znaków. Wykorzystywana jest w metodach klasy Parser, w celu łatwiejszej jego implementacji oraz zwiększenia czytelności. Poniżej deklaracja klasy StringManager i udostępnianych przez nią metod.

```
class StringManager
{
public:
        static string EmptyString();
        static bool IsEmptyString(string s);
        static char ReturnLastChar(string s);
        static bool LastCharIsADigit(string s);
        static bool LastCharIsALetter(string s);
        static bool LastCharIsADigitOrALetter(string s);
        static bool LastCharIsAdigitOrALetterOrAParenthesis(string s);
        static int FindFirst(string s, char c);
        static int FindLast(string s, char c);
        static string Substr(string s, int first, int last);
        static vector<string> Split(string s, string operators);
```

```
        static int FindClosingParenthesis(string s);
        static string ParenthesisContent(string s);
};
```

2.2.3 Parser

Parser Klasa umożliwiająca użytkownikom podanie wielomianów poprzez standardowe wejście. Składa się tylko z dwóch metod. Zadaniem pierwszej z nich jest unifikacja wielomianu podanego na wejściu. Drugą metodą klasy jest tworzenie obiektu, reprezentującego wielomian. Metoda UniformInputString jest wykonywana zawsze na początku funkcji ConvertToPolynomial. Następnie na podstawie otrzymanego wyniku, funkcja ta dokonuje kolejnych obliczeń. Argument type, mówi o tym jakie typu wielomian ma zostać stworzony wewnątrz funkcji. Zwracana referencja wskazuje na obiekt wybranego typu. Dla wartości 0 lub 1, wybierany jest odpowiedni typ - PolynomialMap lub PolynomialVector. Poniżej, krótka definicja klasy Parser oraz jej metod.

```
class Parser
{
        string UniformInputString(string s);
        Polynomial& ConvertToPolynomial(string inputS, int type = 0);
};
```

2.2.4 Number

Klasa Number stanowi warstwę pośrednia pomiędzy klasą wielomianu, a wykorzystaniem typów liczbowych, z biblioteki mpir. Została ona stworzona, by uniezależnić implementacje wielomianów od wykorzystanego sposobu reprezentacji dużych liczb. Dzięki temu jakakolwiek zmiana w udostępnianych przez bibliotekę mpir, klasach i funkcjach, wymaga zmiany kodu aplikacji, tylko w jednym miejscu. Ma to niebagatelny wpływ na łatwość utrzymania aplikacji. Dodatkowo, wprowadzając metody opakowujące funkcje biblioteczne, możliwe jest automatyczne wykonanie dodatkowych operacji i stworzenie nowych funkcji, ułatwiających tworzenie klas i zwiększających czytelność kodu. Narzut czasowy związany z koniecznością wywoływania funkcji pośrednich został oceniony jako niewielki, a ich wpływ na przeprowadzone testy i otrzymane rezultaty jako pomijalny. Zapoznajmy się teraz z definicją klasy Number.

```
class Number
{
public:
        mpq_t value;
        explicit Number();
        explicit Number(double value);
        Number(const Number &bigNumber);
        ~Number();
```

```
Number Neg();
Number Abs();
Number Copy();
void SetMaxNegativeValue();
void SetMaxValue();
bool IsPlusInfinity();
bool IsMinusInfinity();
bool IsInfinity();
bool IsVerySmallValue();
bool IsZero();
int IsInVector(vector<Number> v);

bool operator == (Number bigNumber);
bool operator != (Number bigNumber);
bool operator > (Number bigNumber);
bool operator < (Number bigNumber);
bool operator >= (Number bigNumber);
bool operator <= (Number bigNumber);
Number operator = (Number bigNumber);
Number operator + (Number bigNumber);
Number operator - (Number bigNumber);
Number operator * (Number bigNumber);
Number operator / (Number bigNumber);
Number operator ^ (int power);
Number operator += (Number bigNumber);
Number operator -= (Number bigNumber);
Number operator *= (Number bigNumber);
Number operator /= (Number bigNumber);
Number operator ^= (int power);
bool operator == (double value);
bool operator != (double value);
bool operator > (double value);
bool operator < (double value);
bool operator >= (double value);
bool operator <= (double value);
Number operator = (double value);
Number operator + (double value);
Number operator - (double value);
Number operator * (double value);
Number operator / (double value);
Number operator += (double value);
```

```
        Number operator -= (double value);
        Number operator *= (double value);
        Number operator /= (double value);
        string ToString();
        void Print();
}
```

Jak widać, przeciążone zostały wszystkie przypadne, w przypadku liczb, operatory, zarówno dla klasy Number jak i dla typu double. W języku C++ wszystkie wbudowane typu liczbowe - int, long oraz float, można łatwo zrzutować na ten zmienną typu double. Wynika, z tego, że przy pomocy powyższych operatorów, jesteśmy w stanie w łatwy sposób, dokonać dowolnego działania na dużych liczbach, niezależnie od typu drugiego operandu. Wyjątkiem jest operator potęgowania, dla którego możliwe jest podniesienie do dowolnej potęgi, pod warunkiem, że jej wartość jest liczbą naturalną.

Klasa posiada tylko jeden element – obiekt typu mpq_t, reprezentujący właściwą liczbę wymierną. W klasie zostały zdefiniowane podstawowe funkcje, takie jak obliczenie wartości bezwzględnej oraz liczby przeciwnej. W celu zwiększenia czytelności i łatwiejszej implementacji, zostały przeciążone wszystkie przydatne operatory. Rozpatrzmy dwa przykłady użycia biblioteki mpir – pierwszy z bezpośrednim użyciem dostępnych funkcji i drugi z wykorzystaniem klasy Number.

```
mpq_t simple_function(int a, int b)
{
        mpq_t value1, value2, value3, value4;
        mpq_inits(value1, value2, value3, value4);
        mpq_set_d(value1, (double) a);
        mpq_set_d(value2, (double) b);

        mpq_mul(value3, value1, value2);
        if (mpq_cmp(value1, value2) > 0)
                mpq_add(value4, value1, value3);
        else
                mpq_add(value4, value2, value3);

        mpq_clears(value1, value2, value3);
        return value4;
}

mpq_t simple_function_with_Number(int a, int b)
{
        Number& number1 = new Number(a);
        Number& number2 = new Number(b);
        Number& number3 = new Number(a*b)
        Number& number4 = new Number();
```

```
        if (number1 > number2)
                number4 = number1 + number3;
        else
                number4 = number2 + number3;

        delete(number1);
        delete(number2);
        delete(number3);
        return number4;
}
```

Łatwo zauważyć, że w drugim przypadku kod jest zdecydowanie czytelniejszych. Poprzez przeciążenie operatorów, operacje na dużych liczbach wymiernych wyglądają identycznie jak działania na wbudowanych typach liczbowych. Dzięki temu, wyeliminowano bezpośrednie wywoływanie funkcji bibliotecznych. Ich użycie, pomimo charakterystycznych nazwa, zawsze wymagało chwili zastanowienia nad kolejno przekazywanymi argumentami. W przypadku nieskomplikowanych funkcji, stosunkowo duży narzut związany jest z koniecznością inicjalizowania referencji i zwalniania miejsca, przez obiekty na które wskazują. Nie stanowi to jednak sporej zmiany, w stosunku do przykładu pierwszego, w którym, na początku funkcji nastąpiły deklaracja i inicjalizacja zmiennych, a na końcu ich zwolnienie z pamięci.

2.2.5 Polynomial

Klasa Polynomial to klasa abstrakcyjna, posiadająca zdefiniowane wszystkie funkcje, niezbędne do znalezienia pierwiastków danego wielomianu. Jest ona klasą bazową dla klas PolynomialMap i PolynomialVector, reprezentujących wielomiany, przy pomocy odpowiedniej struktury. Pierwsza z nich bazuje na mapie, czyli strukturze opartej na parach (klucz, wartość). Klucze muszą być różnowartościowe, co umożliwia jednoznaczne znalezienie wartości dla dowolnego z nich. Pozwala to na posiadanie informacji wyłącznie o niezerowych współczynnikach wielomianu. Druga z nich bazuje na wektorze, jako przykładzie tablicy, której kolejne elementy są położone w pamięci operacyjnej obok siebie. Pierwszy element tablicy to współczynnik wielomianu, stojący przy potędze zerowej, a następne wartości to współczynniki, stojące przy kolejnych, coraz to wyższych potęgach. Dzięki temu dostęp do dowolnego wyrazu wielomianu jest bardzo szybki. Jednocześnie jednak, konieczne jest przetrzymywanie informacji o wszystkich współczynnikach wielomianu, stojących przy kolejnych potęgach, od potęgi zerowej, aż do najwyższej potęgi z niezerowym współczynnikiem, równej stopniowi wielomianu.

Klasa Polynomial posiada zarówno już napisane metody, jak i takie, które zostały tylko zaprojektowane i oznaczone jako czysto wirtualne, czyli koniecznie do zaimplementowania w klasie dziedziczącej. Do tego pierwszego zbioru zalicza się część metod, która nie odnosi się do konkretnej struktury wielomianu. Dzięki temu, w klasach podrzędnych nie ma konieczności ponownego ich pisania. Posiadanie jednej implementacji zamiast dwóch pozwalało na łatwiejszą i dużo sprawniej-

szą konstrukcję obu klas dziedziczących. Spore zmiany projektowe i niewielkie korekty wystarczyło wprowadzić tylko raz, zamiast niepotrzebnie je powielać. Niestety, w przypadku sporej części funkcji, mimo bardzo zbliżonej implementacji, nie było możliwe ich połączenie i umieszczenie w klasie bazowej. Spowodowane to było koniecznością odwołań do konkretnego typu danych, w którym zostały umieszczone wyrazu wielomianu. Uogólnione są natomiast wszystkie pozostałe funkcje, w tym także te, które posiadają wywołania funkcji czysto wirtualnych. W przypadku tym, zastosowany został polimorfizm, czyli jeden z paradygmatów programowania obiektowego. Pozwala on odwoływać się do zdefiniowanej metody, bez znajomości jej implementacji. Wywołana funkcja zadziała różnie, zależnie od typu obiektu, na którym zostanie wykonana. Z punktu widzenia poprawnego działania klasy Polynomial, bez znaczenia jest implementacja operatora przypisania, o ile spełnia on swoją rolę. W ten sposób możliwe jest działanie na obiekcie wielomianu, niezależnie, czy bazuje on na tablicy, czy mapie.

W języku C++, by można zastosować polimorfizm, poszczególne metody muszą operować na wskaźnikach lub referencjach. Nie możliwe jest użycie tego mechanizmu w przypadku przekazywaniu obiektów poprzez wartość. Wytłumaczenie tego jest bardzo proste i opiera się na zmiennym rozmiarze obiektów klas pochodnych. Uniemożliwia to odwołanie się do danego obiektu, póki nie znamy jego dokładne typu. Inaczej jest w przypadku referencji i wskaźników, ponieważ ich rozmiar jest stały, a zmieniać się może jedynie ich wartość, czyli miejsce w pamięci, do którego się odwołują.

Z uwagi na powyższy fakt, w programie wszystkie obiekty klasy Polynomial są przekazywane poprzez referencję. By referencję, można przekazywać pomiędzy funkcjami, obiekt na który wskazuje, musi zostać stworzony dynamicznie. Można to zrobić poprzez użycie, znanej z języka C, funkcji malloc, alokującej miejsce w pamięci lub typowego dla języka C++, operatora new. W przeciwnym wypadku, zmienna będzie widoczna tylko w miejscu, w którym zostanie stworzona, np. wewnątrz funkcji. Próba odwołania się do takiej wartości, w miejscu, w którym zmienna nie jest widoczna, skończy się błędem czasu wykonania i komunikatem o naruszeniu dostępu. Stanie się tak, ponieważ referencja w takim przypadku będzie dalej istnieć, ale obiekt, na który wskazuje, już nie. Takie błędy często są popełniane przez niedoświadczonych programistów, a próba ich lokalizacji i znalezienia przyczyny, na pierwszy rzut oka, nie jest oczywista. W momencie, gdy skorzystamy z dynamicznego tworzenia obiektu, będzie on istniał tak długo, dopóki nie zostanie jawnie usunięty w kodzie programu lub aplikacja nie skończy swojego działania, zwalniając przy tym całą zajmowaną pamięć. Kiedy nie wszystkie stworzone obiekty zostają usunięte, dochodzi do tzw. wycieków pamięci. W zależności od ich rozmiarów i czasu działania aplikacje, ich konsekwencje mogą być bardzo różne. W skrajnym wypadku, może dojść do wyczerpania całej dostępnej pamięci. Gdy zaczyna jej brakować, system operacyjny zapisuje jej ostatnio nieużywaną część na dysku, by w razie potrzeby móc ją odczytać. Z uwagi na to, że pamięć operacyjna jest wielokrotnie szybsza od dysków twardych, taka operacja powoduje gigantyczne opóźnienia w pracy komputera. Gdy szybkość kolejnych alokacji pamięci jest większa, niż jej zrzucanie na dysk, w pewnym momencie komputer ulegnie całkowitemu zawieszeniu. Wówczas jedynym wyjściem jest, często bardzo niepożądany, restart systemu. Bezpiecznym rozwiązaniem jest ustalenie limitu na wykorzystanie przez pojedynczą aplikację. Może to znacznie ułatwić debugowanie i znalezienie ewentualnego błędu, a także zabezpieczy użytkownika przed uruchomieniem złośliwego oprogramowania, mającego na

celu doprowadzenie do wspomnianej wyżej sytuacji.

Poniżej przedstawiam definicję klasy abstrakcyjnej Polynomial. Wyróżnić w niej można kolejne sekcje - zmienne klasy, konstruktory, funkcje i operatory. Te ostatnie możemy podzielić pomiędzy te, które posiadają implementacje w klasie abstrakcyjnej Polynomial oraz te, które zostały zdefiniowane jako czysto wirtualne i konieczne jest ich nadpisanie w klasie pochodnej.

```
class Polynomial
{
public:
        MAP m;
        VECTOR v;
        bool isNew = true;
        int type = 0;
        string inputS = "";
        vector<Number> roots;
        int id = 0;

        explicit Polynomial();
        explicit Polynomial(Number number);
        ~Polynomial();

        virtual Polynomial& CreatePolynomial() = 0;
        virtual Polynomial& CreatePolynomial(Number number) = 0;
        virtual void Clear() = 0;
        virtual bool IsZero() = 0;
        virtual int Size() = 0;
        virtual int PolynomialDegree() = 0;
        virtual Number Value(int power) = 0;
        virtual pair<Polynomial&, Polynomial&> \
                DividePolynomials(Polynomial& p1, Polynomial& p2) = 0;
        virtual void SetNumberValue(int power, Number number) = 0;
        virtual int NumberOfChangesSign(Number a) = 0;
        virtual Polynomial& NegativePolynomial() = 0;
        virtual Polynomial& Derivative() = 0;
        virtual Number PolynomialValue(Number a) = 0;
        virtual string ToString() = 0;
        virtual void SturmClear() = 0;
        virtual int TheLowestNonZeroValue() = 0;

        virtual bool operator==(Polynomial& p2) = 0;
        virtual Polynomial& operator = (Polynomial& p2) = 0;
        virtual Polynomial& operator + (Polynomial& p2) = 0;
```

```
virtual Polynomial& operator - (Polynomial& p2) = 0;
virtual Polynomial& operator * (Polynomial& p2) = 0;

virtual VECTOR VectorValuesExceptValueOfPolynomialDegree \
        (int degree) { return{}; };
virtual MAP MapValuesExceptValueOfPolynomialDegree \
        (int degree) { return{}; };

bool Set(string s);
bool IsNew();
PAIR ValueOfPolynomialDegree();

bool ValueEquals(int power, Polynomial& p2);
void SetValue(int power, int value);
void Add(int power, Number number);
void Sub(int power, Number number);
PAIR Mul(int power1, Number number1, int power2, Number number2);
PAIR Div(int power1, Number number1, int power2, Number number2);
Polynomial& Nwd(Polynomial& p1, Polynomial& p2);
Polynomial& PolynomialAfterEliminationOfMultipleRoots();
void Normalize();
Number CoefficientValue(PAIR pair1, Number a);
Number NextNumberFromRange(Number a, Number b);
int NumberOfRoots(Number a, Number b);
int AddNextRoot(Number x);
vector<Number> FindRoots(Number a, Number b);
void PrintRoots(double a, double b);

bool operator!=(Polynomial& p2);
Polynomial& operator / (Polynomial& p2);
Polynomial& operator % (Polynomial& p2);
Polynomial& operator ^ (int power);
Polynomial& operator += (Polynomial& p2);
Polynomial& operator -= (Polynomial& p2);
Polynomial& operator *= (Polynomial& p2);
Polynomial& operator /= (Polynomial& p2);
Polynomial& operator %= (Polynomial& p2);
Polynomial& operator ^= (int power);
void Print();
void PrintInput();
};
```

2.2.6 PolynomialVector

Klasy PolynomialMap i PolynomialVector są implementacjami klasy abstrakcyjnej Polynomial. Obie one nadpisują wszystkie czysto wirtualne metody klasy bazowej. Wiele z tych metom wygląda bardzo podobnie, natomiast istotną różnicą jest konieczność odwołania się do konkretnej reprezentacji danych. To właśnie dlatego zostały one oznaczone w klasie bazowej, jako metody abstrakcyjne. Takie zaprojektowanie klasy, wymusza na użytkowniku korzystanie z interfejsu, udostępnionego przez klase bazowe. Jego poprawne zaimplementowanie daje gwarancje, że klasa będzie wykonywać to czego oczekuje bazowa, nie narzucajac sposóbu, w jaki ma to robić.

Jak zostało wspomniane, różnicą obu klas jest podejście do współczynników zerowych. Została ona oparta na typie vector, z biblioteki STL. Tablica zakłada, że kolejne współczynniki bedą zapisywane w jej kolejnych komórkach, przy czym indeks w tablicy, począwszy od zerowego, określa potęgę dla współczynnika, którego wartość jest tam zapisana. Aplikacja musi pamiętać wskaźniki na pierwszy i ostatni element. Na ich podstawie jest ona w stanie stwierdzić, ile elementów zawiera, co w sposób bezpośredni przekłada się na stopień danego wielomianu. Największym mankamentem takie podejścia jest konieczność posiadania informacji, o wszystkich współczynnikach, także zerowych. W przypadku wielomianów rzadkich, o wysokim stopniu, np. $x^{1000} - 1$, konieczne jest przechowywanie 999 zer w kolejnych komórkach. Poza negatywnym wpływem na złożoność pamięciową takiego podejścia, istotniejszym wydaje się fakt, że znalezienie wyłącznie niezerowych elementów, wymaga przejrzenia wszystkich komórek tablicy. Zatem, by dodać do siebie dwa wielomiany x^{1000} oraz $2x^{1000}$, konieczne jest wykonanie aż 1001 sumowań lub sprawdzenie dla każdego z nich, czy jest potrzebne.

Dodatkowo, w przypadku, gdy w wyniku działania, zmienia się stopień wielomianu, konieczne jest rozszerzenie lub pomniejszenie tablicy, poprzez aktualizacje odpowiednich wskaźników. Jeżeli chodzi o złożoność czasową funkcji, nie ma to jednak wpływu, ponieważ dla wszystkich typów operacji, i tak jesteśmy zmuszeni przejrzeć wszystkie elementy. Wyjątkiem jest stwierdzenie, czy mamy do czynienia, z wielomianem zerowym, którego złożoność jest stała, gdyż wartość ustalana jest na podstawie rozmiaru tablicy.

Jeżeli chodzi o zalety korzystania z tablicy, jako struktury do przechowywania współczynników wielomianu, to najważniejszy jest stały czas dostępu do dowolnego z nich, poprzez możliwość odwołania się do konkretnego elementu tablicy. W przypadku zapisu pod wskazany adres, czas ten może się wydłużyć, w przypadku zmiany stopnia wielomianu. Widać więc, że użycie tablicy wydaje się uzasadnione w przypadku operacji na wielomianach gęstych, których stopień wielomianu nie zmienia się zbyt często. Struktura ta doskonale sprawdzi się do zsumowania dwóch wielomianów, stopnia setnego, zawierających wszystkie współczynniki równe 1. Z kolei, dla przypadku, gdy wielokrotnie sumujemy rzadkie wielomiany, o przeciwnych współczynnikach, wydajność tej struktury prawdopodobnie nie będzie zadowalająca.

2.2.7 PolynomialMap

Klasa PolynomialMap zakłada strukturę wielomianu, reprezentującą wyłącznie niezerowe współczynniki. Pozwala to zaoszczędzić miejsce w pamięci, dla przechowywania wartości zerowych i łatwo stwierdzić ile ich jest. Również w tej klasie wykorzystałem typ z bibiolteki STL, jakim jest map. Mój wybór był spowodowany faktem, że jest to rodzaj kontenera danych, w którym mamy łatwy dostęp do żądanej wartości. Nie jest on tak szybki, jak w przypadku vectora, bo w każdym zapytaniu, musimy dowiedzieć się o położenie konkretnej wartości. Mapa, nie gwarantuje nam, że trzymane w niej danę będą znajdować się, w położonej blisko siebie pamięci. Dzięki temu, modyfikacja jej rozmiaru, jest bardzo łatwa, gdyż wystarczy zmienić tylko te wartości, które rzeczywiście się zmieniają. Gdy wielomian $x^{100}+1$, zamieniamy na $x^{10}+1$, wystarczy, jedyne co musimy zrobić, to usunąć wartość dla klucza równego 100, a dodać klucz równy 10, z wartością 1.

W porównaniu z PolynomialVector, klasa ta wydaje się lepsza dla rzadkich struktur wielomianów, a gorsza dla gęstych. Przeanalizujmy krótko, dlaczego tak podpowiada intuicja. Niepodwarzalne jest, że pojedynczy dostęp do dowolnego elementu, w przypadku tablicy jest szybszy, niż dla jakiejkolwiek innej struktury. Dzieje się tak, ponieważ na podstawie miejsca początku tablicy, dla dowolnego jej elementu, na podstawie indeksu, z góry znamy jego dokładny adres. W przypadku wszystkich kontenerów danych, których kolejne elementy, nie są położone w pamięci obok siebie, najpierw musimy je znaleźć, tj. uzyskać ich adres.

Rozważmy strukturę danych, jaką jest drzewo. Jego elementy są posortowane, ale bezpośredni dostęp mamy tylko do korzenia. Do innych elementów, można dostać się w sposób, wyłącznie pośredni. Niezależnie od rodzaju drzewa, jaki wybierzemy, nie jesteśmy przekroczyć pewnych charakterystycznych dla niego wartości. W przypadku drzewa, średnia złożoność, znalezienia wybranej wartości jest logarytmiczna. Mowa tu o najkorzystniejszym wariancie, czyli przypadku, gdy drzewo to jest zrównowazone, tzn. takie, w którym, odległość do jego liści jest różniąca się maksymalnie o jeden i bliska wartości logarytmu z rozmiaru drzewa. Dodatkowo każde dodanie, modyfikacja i usunięcie dowolnego elementu, zaczyna się od jego znalezienia. Oznacza to, że złożoność logarytmiczna jest w takim przypadku, wartością graniczną.

Wydaje się więc, że aby uzyskać w przypadku takiej struktury, wydajność lepszą niż dla tablicy, liczba ich elementów musi się odpowiednio róznić. Potrzebuje ona mieć odpowiedni zapas liczby operacji, by ich czas, pomimo długości każdej z nich, był łącznie mniejszy.

2.3 Główne funkcje

2.3.1 Metody klasy Parser

Unifikacja wejściowego łańcucha znaków

```
inline string Parser::UniformInputString(string s)
```

Zadaniem funkcji jest ujednolicenie podanego na wejściu, wtaki sposób, był jednoznaczny i łatwy do dalszego przetworzenia. Jako argument przyjmuje ona pojedynczy obiekt klasy string.

Po dokonaniu odpowiednich operacji, jako rezultat, zwraca również typ string. Użycie jej ma na celu ułatwienie przetwarzania w kolejnym etapie, w którym na podstawie podanego ciągu znaków, tworzony będzie obiekt wielomianu. Głównym zadaniem funkcji, jest weryfikacja, czy ciąg znaków podanych na wejściu reprezentuje poprawny składniowo wielomian. Sprawdzana jest liczba nawiasów otwierających i zamykających oraz fakt, czy w każdym miejscu wyrażenia liczba otwartych nawiasów jest nie mniejsza niż liczba nawiasów zamkniętych. Dodatkowo sprawdzane jest, czy na sąsiednich miejscach nie występują dwa operatory. Poprawne wyrażenie nigdy nie kończy się operatorem, a zaczynać może się tylko minusem, literałem, cyfrą lub nawiasem otwierającym. W czasie unifikacji wyrażenia, ignorowane są wszystkie występujące w nim białe znaki. Warto zaznaczyć, że dopuszczalny jest tylko jeden znak, reprezentujący zmienną wielomianu. Nie ma ograniczeń co do wartości tego znaku, może być to znak 'a', 'x' lub jakikolwiek inny, ale powinniśmy zadbać, by w całym wyrażeniu występował on w tej samej postaci. Sporym ułatwieniem w interfejsie jest brak konieczności wpisywania operatorów mnożenia ('*') i potęgowania ('^') w oczywistych miejscach. Z punktu widzenia aplikacji, wyrażenia 4x oraz 4*x są identyczne. Podobnie jest w przypadku x3 oraz x^3. Funkcja analizuje wyrażenie i zwraca je w odpowiedniej postaci. Dla przykładu wyrażenia x3+2x2+3x+1, x^3 +2x^2 + 3x+1 zostaną zamienione w x^3+2*x^2+3*x+1.

Konwersja łańcucha znaków do postaci wielomianu

```
inline Polynomial& Parser::ConvertToPolynomial(string inputS, int type)
```

Funkcja ma na celu stworzenie wielomianu na podstawie wartości znakowej i zwrócenie odpowiedniej referencji. W zależności od parametru type, wielomian reprezentowany jest przez jeden z dwóch rodzajów struktur. Pierwszą z nich jest wektor - począwszy od wyrazu stojącego przy najwyższej potędze, reprezentowane są wszystkie współczynniki, także zerowe. Drugim jest mapa - która posiada informacje tylko o niezerowych współczynnikach.

Na początku funkcji wykonywana jest metoda UniformInputString. Jeżeli zwróci ona pustą wartość, to funkcja przerywa swoje działanie, zwracając pusty obiekt wielomianu. W przeciwnym wypadku następuje iteracja po kolejnych znakach łańcucha wejściowego. Ze względu na kolejność wykonywania działań, w funkcji występują trzy obiekty wielomianów. Pierwszy z nich posiada informacje o aktualnie przetwarzanym fragmencie wejścia. Drugi mówi o wartości fragmentu wielomianu, który wystąpił przed znakiem mnożenia lub dzielenia, stanowiąc, obok aktualnego wielomianu, drugi operand, dla wskazanej operacji. Z kolei, trzeci jest analogiczny i mówi, o wartości drugiego operandu, dla dodawania i odejmowania. Wszystkie działania zostają dokonywane w momencie, gdy natrafimy na operator.

W przypadku, gdy mamy do czynienia, ze znakiem nawiasu otwierającego, zostaje wyszukany odpowiedni nawias zamykający, a na wewnętrznym zakresie jest wywoływana rekurencyjnie ta sama funkcja. Zwracana wartość zapisywana jest jako aktualny element, a wskaźnik zostaje przesunięty na kolejny. Gdy aktualnie przetwarzanym znakiem jest potęgowanie zachowanie funkcji jest analogiczne. Wielomian zostaje podniesiony do odpowiedniej potęgi, otrzymana wartość zapisana, a wskaźnik ustawiony na kolejny element, występujący po wykładniku.

W przypadku pozostałych operatorów działanie funkcji jest nieco odmienne. Spowodowane jest to faktem, że trafiając na dany operator, nie znamy jeszcze jego obu argumentów, a jedynie pierwszy z nich. Drugi jest nieznany, a dodatkowo sam w sobie może zawierać operatory z wyższym priorytetem, np. w przypadku wielomianu x+3*2 potrzebujemy najpierw obliczyć wartość wyrażenia 3*2 i dopiero wówczas uzyskany wynik zsumować z pierwszym operandem, równym x. Fakt ten, dopiero w momencie natrafienia na dany operator, narzuca wykonywanie poprzedniego działania. Uwaga ta tyczy się jednak tylko natrafienia na znaki dodawania, odejmowania, mnożenia i dzielenia, ponieważ, jak zostało wspomniane wcześniej, operator potęgowania i nawiasy posługują się osobnym algorytmem. Zachowanie to, można porównać do działania stosu, na którym mogą leżeć tylko dwa argumenty. Pierwszym z nich jest operacja dodawania, a drugim mnożenia. Za pomocą nich można przedstawić także odejmowanie i dzielenie, zatem funkcje te nie będą przeze mnie rozpatrywane osobno. Operator mnożenia jest traktowany jako działanie z wyższym priorytetem, a dodawania z niższym. Oznacza to, że aby wykonać sumowanie, należy zadbać o to by ewentualna operacja mnożenia została wykonana wcześniej. Tak więc, w momencie natrafienia na znak mnożenia lub dzielenia, wystarczy przemnożyć lub podzielić przez siebie dwa operandy. W momencie dodawania, sytuacja jest bardziej skomplikowana. Zostaje wykonane sprawdzenie, czy odłożony został operator mnożenia. Jeżeli tak, to wartość aktualnego elementu zostaje przemnożona przez zapisany argument, w przeciwnym razie pozostaje bez zmian. Następnie, wynik operacji zostaje dodany lub odjęty od zapisanego argumentu. W momencie, gdy zakończymy iterowanie po całym łańcuchu wejściowym, należy zadbać o ewentualne wykonanie mnożenia oraz dzielenia, czyniąc to analogicznie jak w momencie rozpatrywania znaku dodawania lub odejmowania.

2.3.2 Metody klasy Number

Sortowanie liczb typu mpq_t

```
inline vector<Number> SortNumbers(vector<Number>v)
```

Zadaniem tej metody jest posortować liczby, z biblioteki mpir, w kolejności niemalejącej. Zarówno wejściem, jak i wyjściem funkcji jest wektor obiektów klasy Number. Metoda jest używana w celu sortowania tablicy, o niewielkiej liczbie elementów. Oznacza to, że w takim przypadku, bardzo dobrze sprawdzi się nieskomplikowany algorytm sortujący, np. sortowanie bąbelkowe. Z uwagi na mały zestaw danych, rzędu kilkudziesięciu elementów, kwadratowa złożoność czasowa, będzie akceptowalna, gdy weźmiemy pod uwagę bardzo niewielką liczbę koniecznych operacji, w jednym obiegu pętli. Metoda skorzysta z operatora porównania i metody zamiany elementów klasy Number, bazujących na funkcjach mpq_cmp oraz mpq_swap, z biblioteki mpir.

Porównywanie wektorów liczbowych

```
inline int VectorsAreEqual(vector<Number>v1, vector<Number>v2)
```

Głównym celem metody jest sprawdzenie, czy dwa wektory liczbowe są sobie równe, tzn. czy zawierają wszystkie te same elementy, w dowolnej kolejności. Funkcja przyjmuje, jako argumenty,

dwa wektory, a zwraca liczbę typu int. Jest ona równa 0, gdy wektory są różne oraz 1, gdy są równe. W pierwszej kolejności porównywana jest liczba elementów obu wektorów. Gdy jest ona różna, wówczas mamy pewność, że wektory są różne. Gdy liczba elementów, jest równa, elementy w obu tablicach zostają posortowane, pry pomocy funkcji SortNumbers. Następnie porównywane są kolejne elementy wektorów. Wektory uznajemy za inne, gdy chociaż jedna para jest różna.

Wyszukiwanie danej liczby w tablicy

```
inline int Number::IsInVector(vector<Number> v)
```

Metoda stwierdza, czy obiekt, na którym będzie wykonywana, jest element wektora, podanego jako argument funkcji. Zwraca ona -1, gdy tablica nie zawiera danej wartości liczbowej. W przeciwnym wypadku, rezultatem funkcji jest indeks w podanej tablicy, przy czym pierwszy jej element ma indeks równy 0. Z uwagi, na to, że dany wektor, nie zawsze jest posortowany, nie możemy skorzystać z algorytmu wyszukiwania binarnego, posiadającego logarytmiczną złożoność, ze względu na liczbę jego elementów. By z niego skorzystać, musielibyśmy wykonać sortowanie, a także zapamiętywać informację o indeksie danej liczby, w wektorze inicjalnym. Najszybszy algorytm sortowania, wykonywana na jednym wątku ma złożoność liniową. Jego użycie jest, wiec zupełnie nie opłacalne, jeżeli weźmiemy pod uwagę, że przejrzenie wszystkich liczb tablicy, ma dokładnie taką samą złożoność. Dodatkowo zastosowany algorytm ma własność stopu, tzn. gdy natrafi na element, którego szukamy, kończy swoje działania, zwracając odpowiedni indeks.

Zwracanie wartości liczby w postaci łańcucha znaków

```
inline string Number::ToString()
```

Funkcja ma celu przekształcenie danej liczby, standardowo podawanej jako ułamek, w postaci - licznik i mianownik, na liczbę, zapisaną w postaci dziesiętnej. By to osiągnąć, potrzebna jest informacje, o wymaganej precyzji obliczeń. Na początku liczba zostaje przyrównana do zera, gdy jest mniejsza, jako pierwszy znak wyniku, zostaje ustawiony minus, a dalej liczba jest zawsze rozpatrywana jako nieujemna. Na początku zostaje obliczony iloczyn licznika i liczby równej 10^x, gdzie x = precyzja, wyrażona w ważnych cyfrach po przecinku. Następnie obliczany jest całkowitoliczbowy wynik dzielenia. Do otrzymanego, w ten sposób, rezultatu wystarczy już tylko dodać separator, oddzielający część całkowitą i ułamkową w odpowiednim miejscu. Przypada ono w takim miejscu, by po stronie ułamkowej znajdowała się dokładnie żądana liczba cyfr. Na koniec, należy jeszcze zadbać, by otrzymany wynik był dobrze sformatowany, trzeba więc usunąć, nieznaczące zera w części ułamkowej. Dodatko potrzeba pamiętać, że po wykonaniu takiej operacji, może zdarzyć, się że część ułamkowa nie zawiera już żadnych cyfr. Wówczas należy nie zapomnieć, o usunięciu znaku separatora, tak by dla powstałej liczby całkowitej, ostatnim w kolejności znakiem, była cyfra jedności.

2.3.3 Metody klasy Polynomial

Największy wspólny dzielnik wielomianów

```
inline Polynomial& Polynomial::Nwd(Polynomial& p1, Polynomial& p2)
```

Jest to funkcja obliczająca największy wspólny dzielnik dwóch wielomianów. Wynik jest zwracany przy pomocy referencji, do uzyskanego wielomianu. Na początku, obliczana jest wartość reszty z dzielenia ilorazu podanych wielomianów. Funkcja wołana jest w sposób rekurencyjny. Kolejnymi argumentami funkcji dla wielomianów p1 i p2 są wielomian p2 oraz reszta z dzielenia wielomianu p1 przez wielomian p2. Funkcja wołana jest rekurencyjnie, pod warunkiem, że otrzymana reszta, nie jest wielomianem stopnia zerowego. W takim przypadku zwracana jest wartość p2, jeżeli dzieli ona bez reszty wielomian p1 lub wielomian $W(x) = 1$, w przeciwnym wypadku. Warto zauważyć, że z każdym wywołaniem funkcji, stopień wielomianów, które są jej argumentami, maleje. Oznacza to, że dla wielomianów stopnia n i m, maksymalna liczba rekurencyjnych wywołań funkcji jest równa $min(n, m) + 1$.

Eliminacja pierwiastków wielokrotnych wielomianu

```
inline Polynomial&
        Polynomial::PolynomialAfterEliminationOfMultipleRoots()
```

Funkcja dla danego wielomianu, dokonuje eliminacji pierwiastków wielokrotnych i zwraca otrzymany w ten sposób wielomian. Na początku funkcji, obliczana jest pochodna wielomianu, a następnie dla tych wielomianów, znajdowany jest największy wspólny dzielnik. Na otrzymywanych wielomianach dokonywana jest operacja normalizacji, tzn. podzielenie wszystkich jest współczynników, przez wartość, równą współczynnikowi, stojącemu przy najwyższej potędze. W ten sposób otrzymamy wielomian, o tych samych pierwiastkach, ze współczynnikiem, przy najwyższej potędze równym 1. Pozwala to na dokonanie łatwiejszych, dzięki czemu często szybszych, obliczeń, na danych wielomianach. W drugiej części funkcji, dokonywana jest eliminacja pierwiastków wielokrotnych, poprzez podzielenie wielomianu W, przez wielomian równy $NWD(W, W')$. Otrzymany wynik jest rezultatem funkcji. Zarówno dzielna, jak i dzielnik, są wielomianami, dla których dokonaliśmy już normalizacji, wiadomo więc, że wynikowa wartość, jest już znormalizowana.

Algorytm wyznaczania kolejnych przedziałów wyszukiwania

```
inline Number Polynomial::NextNumberFromRange(Number a, Number b)
```

Funkcja zwraca wybraną liczbę z przedziału (a, b). Algorytm funkcji działa tak, by zoptymalizować wybieranie przedziałów, w którym sprawdzane będzie istnienie pierwiastków. Ma to na celu jak najszybsze zawężenie przedziału, w którym znajduje się szukany pierwiastek. Zakładamy, że długośc przedziału jest niezerowa, a liczby a i b reprezentują odpowiednio - lewy i prawy kraniec przedziału. Z tego założenia wynika, że $a < b$ i takich parametrów spodziewa się funkcja. W celu

optymalizacji, wewnątrz funkcji nie jest sprawdzane, czy przedstawiony warunek jest spełniony. Poniżej przedstawiona została wartość zwracana przez funkcję, w zależności od otrzymanych na wejściu parametrów.

$$f(a,b) = \begin{cases} -2,\ dla\ a = -\infty \wedge b = -1 \\ -1,\ dla\ a = -\infty \wedge b = 0 \\ 0,\ dla\ a * b < 0 \\ 1,\ dla\ a = 0 \wedge b = +\infty \\ 2,\ dla\ a = 1 \wedge b = +\infty \\ a * |a|,\ dla\ a \in (0,1) \cap (1,+\infty) \cap \{-1\} \wedge b = -\infty \\ b * |b|,\ dla\ a = -\infty\ \wedge\ b \in (-\infty,-1) \cap (-1,0) \cap \{1\} \\ \frac{a+b}{2},\ w\ pozostałych\ przypadkach \end{cases}$$

Jak widać, funkcja w pierwszej kolejności stara się dokonać takie podziału, by w obu podprzedziałach znajdowały się wartości tego samego znaku. Kolejnymi wartościami, o których należy wspomnieć są liczby -1 oraz 1. Dzieje się tak z uwagi na to, żeby zoptymalizować znajdowanie pierwiastków, zarówno takich, o niewielkiej wartości bezwzględnej, bliskich zeru, jak i, bardzo dużych. Wówczas poprzez obliczanie kwadratu aktualnej wartości, z zachowaniem jej dotychczasowego znaku jesteśmy w niewielkiej liczbie iteracji maksymalnie zawęzić szukany przedział. Warto zauważyć, że dla liczb mniejszych od 1, posiadających, jako drugi kraniec przedziału 0, w tempie wykładniczym zbliżamy się do 0. Natomiast dla liczb większych od 1, gdy drugim krańcem przedziału jest $+\infty$, w tym samym tempie się od niego oddalamy. Przeanalizujmy jak powstają kolejne podziały w dwóch przypadkach, gdy przedział jest równy $(-\infty, \infty)$, a szukany pierwiastek to -0.1 oraz gdy przedziałem jest (-1, 30), a pierwiastkiem 5. W celu wygody i przejrzystości prezentowanych obliczeń, w przykładzie przyjmijmy, że szukamy pierwiastka z dokładnością 0.1. Oznacza to, że w momencie, gdy mamy przedział długości nie większej niż 0.2, o którym wiemy, że znajduje się w nim pierwiastek, to jesteśmy w stanie podać jego wartość z żądaną precyzją, wskazując dokładnie środek danego przedziału.

Przykład 10

$$\begin{aligned} &a = -\infty,\ b = +\infty\ \Rightarrow\ c = 0 \\ &a = -\infty,\ b = 0\ \Rightarrow\ c = -1 \\ &a = -1,\ b = 0\ \Rightarrow\ c = -0.5 \\ &a = -0.5,\ b = 0\ \Rightarrow\ c = -0.25 \\ &a = -0.25,\ b = 0\ \Rightarrow\ c = -0.0625 \end{aligned} \tag{2.1}$$

Jak widać dla tego specyficznego przypadku, algorytm już w 5 krokach ustalił wartość poszukiwanej liczby. Długość przedziału jest równa $-0.0625 - (-0.25) = 0.1875 < 0.2$, zatem podając środek otrzymanego przedziału, równy $\frac{-0.0625+(-0.25)}{2} = \frac{-0.3125}{2} = 0.15625$, mamy pewność, że będzie wskazywał on badany pierwiastek, z wymaganą precyzją. Teraz spójrzmy na drugi scenariusz.

Przykład 11

$$\begin{aligned}
&a = -1,\ b = 30 \ => \ c = 0\\
&a = 0,\ b = 30 \ => \ c = 15\\
&a = 0,\ b = 15 \ => \ c = -7.5\\
&a = 0,\ b = 7.5 \ => \ c = 3.75\\
&a = 3.75,\ b = 7.5 \ => \ c = 5.625\\
&a = 3.75,\ b = 5.625 \ => \ c = 4,6875\\
&a = 4.6875,\ b = 5.625 \ => \ c = 5,15625\\
&a = 4.6875,\ b = 5.15625 \ => \ c = 4.921875
\end{aligned} \tag{2.2}$$

W drugim przypadku, funkcja zakończyła działanie po 8 iteracjach. Możemy zauważyć, że liczba kroków, jest zależna bezpośrednio od żądanej precyzji. Podając jako rozwiązanie, kolejne środki przedziału, jesteśmy w czasie logarytmicznym dla badanego przedzialu, znaleźć leżący w nim pierwiastek. Należy zwrócić uwagę, że w niekorzystnym przypadku, kolejne przybliżenia niekoniecznie muszą być coraz bliższe szukanemu rozwiązaniu. Tak było by, gdy pierwiastkiem w powyższym przykładzie była liczba 5.15. Wówczas pomimo bycia w bliskim jego otoczeniu, kontynuowalibyśmy pracę algorytmu, a ten znalazł by szukane rozwiązanie, dopiero kilka iteracji później. Jest to niewątpliwy minus zastosowanego algorytmu, jednak ciężki do wyeliminowania.

Liczba pierwiastków wielomianu w badanym przedziale

```
inline int Polynomial::NumberOfRoots(Number a, Number b)
```

Funkcja zwraca liczbę pierwiastków w zadanym przedziale. Jest ona obliczana na podstawie liczby zmian znaku na krańcach przedziałach i równa liczbie takich zmian na prawym krańcu, pomniejszona o ich wartość na lewym.

Znajdowanie pierwiastków wielomianu w badanym przedziale

```
inline vector<Number> Polynomial::FindRoots(Number a, Number b)
```

Jest to najważniejsza funkcja całego projektu, ponieważ wewnątrz niej znajduje się cała logika projektu. Zwraca ona tablicę, z wartościami kolejnych pierwiastków wielomianu. Na początku funkcji sprawdzana jest liczba pierwiastków w badanym przedziale. Liczba ich nie może być ujemna, a kiedy równa jest 0, funkcja kończy swoje działanie i zwraca pusty wektor. Zgodnie z twierdzeniem, liczba pierwiastków w przedziale, wlicza także ewentualny pierwiastek, na prawym krańcu przedziału. Gdy funkcja stwierdzi, istnienie tego pierwiastka, zostaje on dodany do wektora wyjściowego. Jeżeli liczba pierwiastków w przedziale była równa 1, wiemy już, że był to jedyny pierwiastek w danym przedziale i możemy zwrócić jednoelementowy wektor, z wartościami pierwiastków. Następnie, na podstawie funkcji NextNumberFromRange, przedział zostaje podzielony

na dwa mniejsze. W otrzymanych przedziałach zostaje obliczona liczba pierwiastków. Jeżeli jest ona dodatnia, to następuje rekurencyjne wywołanie funkcji, z argumentami, będącymi granicami danego podprzedziału. Wywołana rekurencyjnie metoda, po wykonaniu, zwróci wynik, który zostanie przetworzony przez funkcje wywołującą.

2.3.4 Metody czysto wirtualne klasy Polynonial – porównanie działania metod klas PolynomialMap i PolynomialVector

Ustawianie wartości wyrazu wielomianu

```
inline void PolynomialMap::SetNumberValue(int power, Number number)
inline void PolynomialVector::SetNumberValue(int power, Number number)
```

Głównym zadaniem funkcji jest ustawienie wartości podanego współczynnika na daną wartość liczbową. W przypadku mapy, konieczna jest wyryfikacja, czy liczba jest równa zero. Jeżeli tak, to następuje sprawdzenie, czy w mapie występuje już klucz, równy podanej potędzę. W takim przypadku następuje jego usunięcie, a gdy klucz nie istnieje, jej opuszczenie, bez jakichkolwiek dodatkowych działań. Jeżeli wartość liczbowa jest niezerowa, to sprawdzenie jest analogiczne. W pierwszym przypadku następuje wówczas nadpisanie wartości o danym kluczu, a w drugim wstawienie do mapy pary – klucz (potęga), wartość (liczba).

W przypadku klasy PolynomialVector meotoda została nieco bardziej rozbudowana. Na początku następuje sprawdzenie, czy dany stopień potęgi jest wyższy od aktualnego stopnia wielomianu. Jeżeli tak, to następuje sprawdzenie, czy dana wartość jest zerowa. Wówczas funkcja kończy swoje działanie, a w przeciwnym wypadku rozpoczyna operację wstawiania do wektora żądanej pary – potęga, wartość współczynnika. Gdy stopien wielomianu zwiększa się o 1, wystarczy, po prostu, na kolejnym miejscu w tablicy wstawić daną wartość. W przeciwnym zaś przypadku, musimy zadbać o to, by na miejscach wszystkich współczynników, reprezentujących ich wartości dla kolejnych potęg, począwszy od dotychczasowego stopnia wielomianu, powiększego o 1, aż do jego nowego stopnia, pomniejszonego o 1, zostały wstawione zera. Dopiero wówczas na kolejnym miejscu w tablicy może zostać wstawiona wartość dla odpowiedniej potęgi.

Gdy, podana w argumencie funkcji, potęga, jest niewiększa od stopnia wielomianu, następuje uaktualnienie wartości na odpowiedniej pozycji w tablicy. Konieczne jest wówczas dodatkowe sprawdzenie. Jeżeli, podana para – potęga, wartość współczynika wskazują na zerowy współczynnik dla potęgi, równej stopniowi danego wielomianu, niezbędne jest przesunięcie końca tablicy o jedno miejsce. Następnie wykonywane są sprawdzenia kolejnych elementów funkcji, począwszy od końca. W przypadku, gdy mamy do czynienia z zerami, następują kolejne korekty granicy tablicy. Podana operacja trwa tak, dopóki nie zostaną przeanalizowane wszystkie elementy lub nie natrafimy na dowolną wartość różną od zera.

Warto zauważyć, że w przypadku obu klas, funkcja ma wpływ na stopień wielomianu. Dodatkowo, operacja dokładania i zdejmowania kolejnych argumentów, w przypadku wektora nie została zoptymalizowana. Możliwe, że w sytucji konieczności, wielokrotnej zmiany jej rozmiarów,

korzystniejsza byłaby pojedyncza operacja, zwłaszcza w przypadku konieczności usuwania kolejnych zerowych współczynników. Należałoby wówcas, najpierw przeanalizować kolejne ich wartości i zliczyć wszystkie zera znaczące, a następnie pojedynczą operacją, odpowiednio zmanipulować wskaźnik na ostatni element.

Stwierdzanie czy wielomian jest wielomianem zerowym

```
inline bool PolynomialMap::IsZero()
inline bool PolynomialVector::IsZero()
```

Funkcja ma na celu przeanalizowanie struktury wielomianu i stwierdzenie, czy dany wielomian jest wielomianem zerowym. Implementacja metody w obu klasach jest analogiczna. Następuje sprawdzenie, czy wielomian posiada wyłącznie zerowe współczynniki. Zgodnie z założeniami, w mapie przetrzymywane są tylko niezerowe wartości wyrazów wielomianu, a w wektorze, przechowywane są wszystkie spółczynnik, aż do ostatniego niezerowego współczynnika, stojacego przy najwyższej potędze. Na tej podstawie jesteśmy w stanie stwierdzić, że funkcja zwraca prawdę, tylko, jeżeli mapa, bądź wektor są puste.

Obliczanie stopnia wielomianu

```
inline int PolynomialMap::PolynomialDegree()
inline int PolynomialVector::PolynomialDegree()
```

Zadaniem tej metody jest obliczenie stopnia danego wielomianu. Złożoność funkcji jest zależna od wybranej klasy. W przypadku PolynomialMap konieczna jest analiza jej kolejnych elementów i stwierdzenie, jaka jest największa, występująca w niej potęga. W przypadku PolynomialVector złożonośc funkcji jest stała, gdyż stopień wielomianu jest równy rozmiarowi tablicy, pomniejszonego o jeden. Dla obu klas, gdy mamy do czynienia z wielomianem zerowym, zwracaną wartością jest -1.

Zwrócania wartości wielomianu w postaci łańcucha znaków

```
inline string PolynomialMap::ToString()
inline string PolynomialVector::ToString()
```

Celem funkcji jest zwrócenie wartości wielomianu w postaci zmiennej znakowej. Jej rezultat wygląda analogicznie, jak wynik metody UniformInputString w klasie Parser. Na podstawie odpowiednich par – potęgi i wartości współczynnika, zostaje stworzony obiekt typu string. Na podstawie kolejnych liczb, przed kolejnymi wyrazami, zostają dostawione odpowiednio znaki plus, dla wartości większych od 0 oraz minus, dla wartości mniejszych. Wyjątkiem jest pierwszy wyraz, w przypadku którego ewentualny znak minus, jest ignorowany. Pomijane są także zbędnę elementy, takie jak „1*” dla współczynników, stojących przy potęgach dodatnich oraz „*x^0” „^1” dla odpowiednio – zerowej i pierwszej potęgi.

Chociaż struktury, różnią się reprezentacją w pamięci, to obie zwracają identycznie sformatowany rezultat. Uznałem bowiem, że w przypadku tablicy, przy przyjętym formatowaniu, wypisywanie zerowych współczynników jest zupełnie zbędne i tylko uczyniłoby wynik mniej czytelnym. Opcja wypisywania tablicy z kolejnymi współczynnikami została także odrzucona z podobnych powodów. Sytuacja taka byłaby bardzo niekorzystna dla użytkownika. Zwłaszcza dla rzadkich wielomianów wysokich stopni, ich reprezentacja uległaby znacznej zmianie, a ręczne znalezienie niezerowych współczyników okazałoby się praktycznie niewykonalne.

Dzielenie wielomianów

```
inline pair<Polynomial&, Polynomial&>
PolynomialMap::DividePolynomials(Polynomial& p1, Polynomial& p2)

inline pair<Polynomial&, Polynomial&>
PolynomialVector::DividePolynomials(Polynomial& p1, Polynomial& p2)
```

Metoda dla podanych wielomianów oblicza ich iloraz oraz resztę z dzielenia i zwraca je jako parę obiektów. Na początku, funkcja oblicza wartość pierwszego wyrazu wielomianu, który zostanie zwrócony, jako iloraz. Jest on równy wynikowi dzielenia wyrazów stojących przy najwyższych potęgach wielomianów – dzielnej i dzielnika. Kolejne wyrazy drugiego z nich są mnożenie przez otrzymaną wartość, a wynik jest odejmowany od dzielnej. W rezultacie tego działania, współczynnik przy najwyższej potędze staje się równy zero i zostaje zredukowany. Daje to gwarancję, że nowy stopień wielomianu, będzie przynajmniej o jeden, mniejszy, niż stopień danego wielomianu. Funkcja, kończy swoje działanie, gdy dokonona powyższej operacji dla wszystkich wyrazów dzielnej. Wówczas aktualnie przetwarzany wielomian jest zwracany jako obliczona reszta. Warto zauważyć, że stopień tego wielomianu jest mniejszy od stopnia dzielnika i co najmniej o 2 mniejszy od stopnia dzielnej.

Obliczanie pochodnej wielomianu

```
inline Polynomial& PolynomialMap::Derivative()
inline Polynomial& PolynomialVector::Derivative()
```

Celem funkcji jest obliczenie pochodnej danego wielomianu. Jej wartość jest obliczana tylko w przypadku, gdy wielomian jest niezerowy. Dla danego wielomianu $W(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$, pochodna wielomianu wyraża wzorem: $W'(x) = na_0x^{n-1} + (n-1)a_1x^{n-2} + ... + 2a_{n-2}x + a_{n-1}$. W przypadku klasy PolynomialMap wymagane jest przejrzenie tylko niezerowych współczynników wielomianu, zatem operacja ta jest liniowa, względem ich liczb. Z kolei, dla typu PolynomialVector konieczne jest sprawdzenie wartości wszystkich współczynników, przy czym tylko te niezerowe mają wpływ na wartość pochodnej.

Obliczanie wyrazów ciagu Sturma

```
inline vector<PolynomialMap> PolynomialMap::GetSturm()
```

```
inline vector<PolynomialVector> PolynomialVector::GetSturm()
```

Funkcja ma na celu zwrócenie wektora wielomianów, w którym wartościami będą kolejne wyrazu ciągu Sturma. Na początku, przy pomocy funkcji Derivative obliczana jest pochodna wielomianu. Gdy jest ona wielomianem zerowym, zostaje zwrócony ciąg Sturma, zawierający tylko jeden element – wielomian, dla którego ciąg ten jest obliczany. Wiadomo wówczas, że liczba zmian znaków dla takie ciagu, niezależnie od wybranego punktu, będzie równa zero, co oznacza, że dany wielomian nie zawiera pierwiastków rzeczywistych.

Kolejnym krokiem jest obliczenie reszty z dzielenia wielomianu, przez jego pochodną. Jeżeli jest ona wielomianem zerowym, to zwracany jest dwuelementowy ciąg Sturma – wielomian i jego pochodna. W przeciwnym razie, wartość do niej przeciwna stanowi kolejny element ciągu Sturma. Podobnie jak w przypadku pierwszej reszty z dzielenia, także kolejne wyrazy są obliczane jako iloraz dwóch poprzednich wyrazów ciągu Sturma. Dzieje się to tak długo, dopóki otrzymany wielomian jest wielomianem zerowym. Warto zauważyć, że maksymalna liczba wyrazów ciągu Sturma jest równa stopniowi wielomianu, powiększonemu o jeden. Wówczas stopnie kolejnych wyrazów ciągu Sturma są równe kolejnym potęgom, od stopnia wielomianu, do zera.

Liczba zmian znaków w danym punkcie

```
inline int PolynomialMap::NumberOfChangesSign(Number a)
inline int PolynomialVector::NumberOfChangesSign(Number a)
```

Funkcja na postawie ciągu Sturma i metody GetSturm, oblicza liczbę zmian znaków dla podanego punktu. Porównuje ona wartości dwóch kolejnych wyrazów ciagu Sturma. Jeżeli ich iloczyn jest ujemny, następuje zwiększenie licznika o jeden. W przypadku wartości zerowych, element taki jest pomijany, tzn. ciąg jest rozpatrywany, tak jakby go nie zawierał. Warto zauwazyć, że dla odpowiednio dużych wartości absolutnym liczby a, w tym wartości niewłaściwych $-\infty$ i ∞, liczba zmian znaków, zależy wyłącznie od współczynników stojących, przy najwyższych potęgach kolejnych wielomianów. Fakt ten powoduje możliwą optymalizację działania funkcji, w zależności od otrzymanego parametru. Najbardziej czasochłonna operacja, jaką jest dzielenie wielomianów, i tak musi zostać wykonana, by znaleźć wszystkie wyrazy ciągu Sturma, ale zysk z takiej optymalizacji, może być zauważalny i znacząco poprawić wydajność funkcji.

Rozdział 3

Przeprowadzone testy

3.1 Testy funkcjonalne

3.1.1 ParserUniform

Klasa testowa pozwalająca na weryfikacje metody UniformInputString, mającej na celu unifikację otrzymanej wartości znakowej, przedstawiającej wielomian. Testy dla niej zostały napisane, pomimo, że funkcja nie jest skomplikowana. Jej funkcjonalność została uznana za podstawową, a jej poprawne działanie jest niezbędne, by korzystać z programu. Jest to jedyna funkcja, która jest wołana, niezależnie od innych.

Jej zadaniem jest nie tylko ujednolicić wprowadzaną postać wielomianu, ale przede wszystkim dokonać walidacji jej poprawności. Musi być ona odporna na róznego rodzaju przypadki, wprowadzane przez użytkownika, zarówno przypadkiem, jak i celowo. Jej zadaniem jest przeanalizować i zwrócić pusty obiekt typu string, w momencie, gdy nie uda się jej zrozumieć i w pełni przeanalizować otrzymanego wejścia. Funkcja ConvertToPolynomial wywołuje ją na początku, a następnie bazując na otrzymanym rezultacie, tworzy wybrany obiekt typu PolynomialMap lub PolynomialVector.

Testy tej funkcji zakładają sprawdzenie, że dla danego wejścia, zostaje zwrócone poprawne wyjście, czyli takie, które jest standardowe i łatwe do sparsowania. W przeciwnym wypadku, zwrócony ma zostać pusty łańcuch. Niezależnie jakie dane zostaną podane na wejściu, program nie ma praw, zakończyć się błędem, rzucając wyjątek, ani w ogóle się nie zakończyć.

Na przedstawioną funkcjonalność napisałem ponad 20 testów funkcjonalnych. Poniżej zamieszczam przykładowe z nich, przedstawiając je w postaci - wejście, wyjście.

```
"" -> ""
"2x*+4" -> ""
"3*x-2" -> "3*x-2"
"4x" -> "4*x"
"x2+5" -> "x^2+5"
```

```
"10x3 - 4 x2 + 5x" -> "10*x^3-4*x^2+5*x"
"(3x-1)(2x+4)+2(x-4)" -> "(3x-1)*(2x+4)+2*(x-4)"
```

3.1.2 ParserConvert

Klasa pozwalająca na testowanie metody ConvertToPolynomial. Wywołuje ona funkcje UniformInputString i analizuje jej wartość. Gdy jest to pusta wartość, tworzy wielomian zerowy. W przeciwnym wypadku, analizuje ona uzyskany rezultat i na jego podstawie tworzy wybrany obiekt typu PolynomialMap lub PolynomialVector. Rodzaj stworzonego wielomianu jest podawany jako parametr funkcji. W zależności od niego, także wszystkie pośrednie wielomianu, przybierają żądaną formę.

Przetestowanie tej funkcji powinno być możliwie bardzo dokładne i zawierać różne przypadki graniczne. Bez poprawnej działania tej funkcji, użytkownik w żaden sposób nie może przewidzieć jaką wartośc będzie miał dany wielomian. Ewentualny błąd w tej funkcji, może powodować nieprzewidziane zachowanie pozostałych. W zachowaniu funkcji uwzględnione przypadki niedopuszczalnych wielomianóW, które jednak przeszły weryfikacji poprawności funkcji UniformInputData. Są nimi między innymi: ujemna lub niecałkowita potęga wielomianu, czy dzielenie przez wielomian zerowy. Wejścia takie nie zostaną odrzucone, ponieważ zawierają one wielomiany poprawne składniowo, a nielegalne wartości zostaną wykrytę, dopiero w momencie dokonania obliczeń, wykonanywanych jako pierwsze, np. z powodu nawiasu.

By przetestować funkcję, napisałem blisko 50 różnych testów. Poniżej przedstawiam wybrane z nich przypadki testowe.

–

3.1.3 Operatory dodawania, odejmowania i mnożenia

PolynomialSumOperator

Jest to klasa testowa odpowiedzialna za testowanie funkcjonalne operatora dodawania. W klasie Polynomial został przeciążony operator dodawania, pozwalający dodawać dwa wielomiany, zapisując działanie to w czytelny i przejrzysty sposób. Testowanie poprawności tego działania polega na wywołaniu funkcji weryfikującej z trzema parametrami. Dwa pierwsze są składnikami dodawania, a ostatni to oczekiwana suma. Funkcja za pomocą operatora dodawania oblicza sumę wielomianów, a następnie porównuje ją z oczekiwanym wynikiem. Działanie polega na dodaniu wartości wyrazów, o tych samych potęgach, czyli redukcji wyrazów podobnych. Poniżej zamieszczam przykładowe testy.

```
"1" + "20" = "21"
"5" + "-11" = "-6"
"12x" + "0" = "12x"
"12x3" + 15x3" = "27x3"
```

```
"12x3+4x2" + "-15x2" = "12x3-11x2"
"12x3+4x+8" + "-15x2+30x+5x-5-3" = "12x3-15x2+39x"
```

PolynomialSubOperator

Jest to bliźniacza klasa testowa dla PolynomialSumOperator. Jedyną zmianą jest to, że w tym przypadku, podajemy kolejno: odjemną, odjemnik i oczekiwana różnicę. Warto zauważyć, że odejmowanie dwóch wielomianów, podobnie jak w przypadku liczb, sprowadza się do zsumowania pierwszego z nich i wartości przeciwnej do drugiego. Przeprowadzone testy są analogiczne, jak dla operatora sumy.

```
"1" - "20" = "-19"
"5" - "-11" = "16"
"12x" - "0" = "12x"
"12x3" - 15x3" = "-3x3"
"12x3+4x2" - "-15x2" = "12x3+19x2"
"12x3+4x+8" - "-15x2+30x+5x-5-3" = "12x3+15x2-31x+16"
```

PolynomialMulOperator

Jest klasą, której zadaniem jest jak najdokładniej przetestować operator mnożenia wielomianów. Testy analogicznie jak w przypadku poprzednich klas, mają na celu sprawdzenie, czy dla dwóch podanych czynników, uzyskamy pożądany wynik. Operacja iloczynu wielomianów polega na przemnożeniu wszystkich wyrazów pierwszego z nich przez wyrazy drugiego. Mnożąc dwa wyrazy, mnożymy przez siebie ich wartości, zaś wykładniki sumujemy. Następnie dokonujemy redukcji wyrazów podobnych, poprzez ich zsumowanie, a uzyskany w ten sposób wielomian jest szukanym iloczynem.

```
"1" * "20" = "20"
"0" * "-55" = "-55"
"-2x" * "7" = "-14x"
"12x" * "-5x" = "-60x2"
"12x3" * "15x3" = "180x6"
"12x3+4x2" * "-15x2" = "-180x5-60x4"
"12x3+4x+8" * "-15x2+35x-8" = "-180x5+420x4-156x3+20x2+248x-64"
```

3.1.4 Operatory dzielenia i modulo

PolynomialDivOperator

PolynomialDivOperator jest klasą testową, pozwalającą na przetestowania poprawności dzielenia wielomianów. Funkcja ją weryfikują przyjmuje trzy parametry. Są nimi dzielna, dzielnik oraz spodziewany iloraz. Podany dzielnik nie może być wielomianem zerowym, ponieważ dzielenie przez

0, również w przypadku wielomianów nie jest dopuszczalne. Operacja dzielenia jest, w przypadku wielomianów, najbardziej skomplikowaną operacją. Algorytm jej wykonania przypomina nieco dzielenie pisemne w przypadku liczb całkowitych. Wyraz, stojący przy najwyższej potędze dzielnej, jest dzielony przez wyraz, stojący przy najwyższej potędze dzielnika, a wynik jest zapisywany. Następnie otrzymany wyraz jest mnożony przez wielomian przeciwny do dzielnika, a otrzymany rezultat sumowany z dotychczasową wartością wielomianu. Uzyskany wynik staje się nową dzielną, a sytuacja powtarza się do momentu, gdy stopień dzielnej będzie niższy, niż stopień wielomianu dzielącego. Wartości otrzymane z kolejnych dzieleń tworzą wyrazy ilorazu, o którym wiemy, że jego stopień jest równy różnicy stopni argumentów operatora dzielenia.

```
"20" / "20x" = "0"
"20" / "-1" = "=-20"
"-21x" / "7x" = "-3"
"-12x2" / "-6x" = "2x"
"x3+x2+x" / "x" = "x2+x+1"
"-30x3-15x" / "-5x2" = "6x"
"x6-6x4-4x3+9x2+12x+4" / "x5-4x3-2x2+3x+2" = "x"
```

PolynomialModOperator

Jest klasą testową dla operatora modulo, którego obliczanie oparte jest na wykonywaniu operacji dzielenia, z tymże w tej sytuacji zwracany jest inny wielomian. W przypadku opisanym powyżej, tzn. w momencie gdy stopień dzielnej jest większy niż stopień aktualnie przetwarzanego wielomianu, to drugi z tych wielomianów nazywany jest resztą z dzielenia. Wiemy o nim, że jego stopień jest mniejszy niż dzielnika. W szczególnym przypadku, gdy dzielnik jest podzielny przez dzielną, uzyskana reszta jest wielomianem zerowym. Warto zaznaczyć, że podobnie jak w przypadku dzielenia, w celu weryfikacji podawane są także trzy wielomiany. Ostatni z nich jest pożądanym wynikiem operacji modulo, zaś środkowy dzielnikiem, co wymusza by jego wartość była różna od zera. Poniżej zaprezentowane zostały te same przypadki testowe jak dla dzielenia, ale w tej sytuacji wynikiem jest rezultat operacji modulo. Dodatkowo lista testów została powiększona o testy, charakteryzujące się niezerową resztą.

```
"20" % "20x" = "20"
"20" % "-1" = "=0"
"-21x" % "7x" = "0"
"-12x2" % "-6x" = "0"
"x3+x2+x" % "x" = "0"
"-30x3-15x" % "-5x2" = "-15x"
"x6-6x4-4x3+9x2+12x+4" % "x5-4x3-2x2+3x+2" = "-2(x4+x3-3x2-5x-2)"
"x3+x2+x1+11" % "x2+1" = "10"
"(x+2)^3+1" % "x+2" = "1"
"x4-4x3+6x2-7x+3" % "(x-1)^2 = "-3x+2"
```

3.1.5 PolynomialDerivative

Zadaniem tej klasy jest sprawdzenie, czy metoda, pozwalająca na obliczanie pochodnej wielomianu, działa poprawnie. Jest ona wykonywana na konkretnej instancji obiektu typu Polynomial i zwraca referencję do jego pochodnej. Funkcja bazuje na operatorach dodawania i mnożenia. Ich poprawne działanie jest zatem niezbędne, by metoda ta mogła poprawnie funkcjonować. Weryfikacje poprawności wyników następuje poprzez porównanie oczekiwanej wartości pochodnej wielomianu z wartością obliczoną. Poniżej prezentuję wybrane przypadki testowe dla tej funkcjonalności.

–

3.1.6 PolynomialAfterElimination

Jest to klasa weryfikująca, czy wartość wielomianu, po eliminacji pierwiastków wielokrotnych, jest poprawna. Metoda ta jest wykonywana dla danego obiektu typu Polynomial i zwraca referencję do wielomianu wynikowego. Jej wartość nie jest jednak bezpośrednio porównywana z oczekiwanym wynikiem. Wcześniej jest bowiem wykonywana jeszcze funkcja normalizująca uzyskany wielomian. Jest to niezbędne, gdyż w przeciwnym razie, wykonywane porównanie mogłoby wskazać fałszywy rezultat. Jest to spowodowane tym, że funkcja porównująca, ze względów wydajnościowych, nie normalizuje danych wielomianów. Decyzja taka była umotywowana faktem, że w takim przypadku porównanie całkowicie odmiennych wielomianów byłoby liniowe ze względu na ich stopień, zamiast odbywać się w czasie stałym. Aplikacja zakłada, że w każdym miejscu, gdzie jest to wymagane, będzie następowało jawne wywołanie odpowiedniej funkcji.

Aspektem bardzo ułatwiającym testowanie tej funkcjonalności jest możliwość podawania wielomianu wejściowego w postaci iloczynu wielu czynników. Pozwala to podawać kolejne czynniki, zawierające dane pierwiastki wielomianu, z określonymi krotnościami i spodziewać się dokładnie tych samych, ale jednokrotnych. Dzięki temu jest się niezależnym od innych programów, pozwalających na obliczanie zer wielomianów, ponieważ spodziewane wartości, zawarte są pośrednio w wielomianie wejściowym. Poniżej zamieszczam przykładowe testy, z uwzględnieniem podawania wejścia, w różnej postaci.

–

3.1.7 PolynomialValue

Jest klasą weryfikującą, czy wartość wielomianu w danym punkcie jest obliczana poprawnie. Funkcja weryfikująca przyjmuje dwa argumenty wejściowe – wielomian oraz dany punkt. Otrzymany wynik porównuje z trzecim parametrem, którym jest oczekiwane rezultat. Poza sprawdzeniem poprawności działania samego wielomianu, weryfikowane jest także działanie klasy Number. Ważnym aspektem jest tutaj przede wszystkim sprawdzenie zachowania funkcji, w przypadku dużych liczb, które często pojawiają się w momencie obliczania wartości wielomianów wysokich stopni. Należy wziąć pod uwagę, że obliczenie wartości w punkcie $x_0 = 10$, dla wielomianu stopnia setnego

wymaga liczby posiadającej aż 101 cyfr. To właśnie z tego względu w projekcie niezbędna była biblioteka mpir, pozwalająca na posługiwanie się takimi wartościami liczbowymi. Poniżej prezentuję przykładowe testy na obliczanie wartości wielomianu w danym punkcie.

–

3.1.8 PolynomialNumberOfRoots

Jest to klasa, sprawdzająca ile pierwiastków rzeczywistych znajduje się w podanym przedziale. Pod uwagę brane są wyłącznie rożne pierwiastki, dlatego na początku funkcji weryfikującej wywoływana jest funkcja PolynomialAfterEliminationOfMultipleRoots, a dopiero na jej znormalizowanym rezultacie wykonywana jest właściwa funkcja. Jej rezultat porównywany jest ze spodziewaną wartością, podawaną w postaci zmiennej typu int. Domyślnymi granicami przedziału są wartości niewłaściwe $-\infty$ i $+\infty$, więc gdy interesują nas pierwiastki w całym zbiorze liczb rzeczywistych, nie ma potrzeby ich podawać. W przeciwnym razie nadpisywana jest wartością zmiennych klasy – a i b, oznaczających lewy i prawy kraniec przedziału. Możliwe jest także sprecyzowanie granicy przedziału tylko z jednej strony, np. gdy szukamy liczby pierwiastków dodatnich. Poniżej zamieszczam przykładowe testy dla różnych przedziałów wyszukiwania.

–

3.1.9 PolynomialRoots

Na początku funkcji weryfikującej dokonywana jest eliminacja pierwiastków wielokrotnych. Otrzymany wielomian poddawany jest normalizacji, a wówczas znajdowane są jego pierwiastki w podanym przedziale. Podobnie jak w przypadku klasy PolynomialNumberOfRoots, domyślnym przedziałem jest zakres od $-\infty$ do $+\infty$. Pierwiastki w wektorze wyjściowym funkcji znajdują się w kolejności ich znalezienia. Także w tym przypadku, konieczne jest więc ich posortowanie. Poniżej zamieszczam większość testów, sprawdzających ostateczną funkcjonalność programu. Lista ta jest bardziej rozbudowana niż poprzednie, ponieważ uznałem, że jest ona zdecydowanie najistotniejszą częścią testów funkcjonalnych mojego programu.

–

Bibliografia

[1] E.J. Barbeau. *Polynomials*. Problem Books in Mathematics. Springer New York, 2003.

[2] D. Buell. *Algorithmic Number Theory: 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004.

[3] R.L. Burden, J.D. Faires, and A.M. Burden. *Numerical Analysis*. Cengage Learning, 2015.

[4] L.N. Childs. *A Concrete Introduction to Higher Algebra*. Undergraduate Texts in Mathematics. Springer New York, 2012.

[5] T. Granlund and G.D. Team. *Gnu MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, 2015.

[6] W. Kryszewski. *Wykład analizy matematycznej, cz. 1: Funkcje jednej zmiennej:*. Number pkt 1. Wydawnictwo Naukowe UMK, 2014.

[7] D.S. Malik. *Data Structures Using C++*. Cengage Learning, 2009.

[8] Polskie Towarzystwo Matematyczne. *Wiadomości matematyczne*. Number t. 10-11. Państwowe Wydawn. Naukowe., 1968.

[9] J.M. McNamee. *Numerical Methods for Roots of Polynomials -*. Number pkt 1 in Studies in Computational Mathematics. Elsevier Science, 2007.

[10] T. Mora. *Solving Polynomial Equation Systems I: The Kronecker-Duval Philosophy*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[11] V. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Birkhäuser Boston, 2012.

[12] W. Sierpiński and A. Mostowski. *Zasady algebry wyższej, z przypisem Andrzeja Mostowskiego Zarys teorii Galois*. Monografie Matematyczne. Nakładem Polskiego Tow. Matematycznego, 1951.

[13] Mieczysław (1918-2007) Warmus and Józef (1927-) Łukaszewicz. Metody numeryczne i graficzne. cz. 1.