Algorytm znajdowania sumy podzbioru

You

12 lutego 2023

Streszczenie

Problem istnienia podzbioru o sumie t zawierającego się w n-elementowym (multi)zbiorze $S \subset \mathbb{N}$, jest jednym z klasycznych problemów algorytmicznych. Problem w ogólności jest problemem NP-zupełnym jednak jeżeli t jest względnie małe i dysponujemy pamięcią $\Omega(t)$ istnieją algorytmy działające w czasie wielomianowym od długości wejścia, z czego najbardziej znanym jest działający w czasie $O(nt)$ algorytm oparty na programowaniu dynamicznym.

Celem niniejszej pracy jest dokumentacja implementacji algorytmu opracowanego przez Ce Jin i Hongxun Wu w pracy[odnośnik!!!]. Algorytm ten jest niedeterministyczny, jednak dla dużych danych prawdopodobieństwo błędu jest niewielkie. Złożoność czasowa prezentowanego algorytmu to $O((t+n)\log^2(t))$, zaś pamięciowa $O(n+t)$.

1 Wstęp

Problem sprawdzenia czy z $n-$elementowego multizbioru S liczb naturalnych jesteśmy w stanie wybrać podzbiór, którego suma elementów jest równa zadanej liczbie t jest jednym z klasycznych problemów algorytmicznych. Będziemy go w dalszej części nazywać **problem sumy podzbioru**.

Ma on zastosowanie w licznych praktycznych zagadnieniach na przykład kryptografii [Merkle–Hellman knapsack cryptosystem], analizie finansowej [Solving Subset Sum Problems using Binary Optimization with Applications in Auditing and Financial Data Analysis], czy zagadnieniach kombinatorycznych, jako specyficzny wariant problemu plecakowego (Problem plecakowy dotyczy możliwości wybrania ze zbioru przedmiotów, z których każdy ma określoną wagę i wartość, podzbioru mającego maksymalną sumaryczną wartość i jednocześnie, którego sumaryczna waga nie przekracza pewnej liczby t. Problem sumy podzbiorów odpowiada sytuacji, gdy wartości przedmiotów są wprost proporcjonalne do ich wagi).

Klasyczna wersja problem, gdzie t jest duże (gdy nie dysponujemy $O(t)$ pamięci) jest problemem NP-zupełnym. Najprostszy algorytm rozwiązujący ten problem polega na rozważeniu po kolei wszystkich możliwych podzbiorów. Złożoność tego algorytmu to $O(n2^n)$. Nieco szybszym algorytmem jest algorytm Horowitza i Sahaniego który działa w $O(2^{\frac{n}{2}}n/2)$, jednak w przeciwieństwie do algorytmu naiwnego wymagającego $O(n)$ pamięci algorytm ten wymaga $O(2^{\frac{n}{2}})$. Algorytm Schroeppela i Shamira wymaga takiego samego czasu jak algorytm Horowitza i Sahaniego, jednak potrzebuje $O(2^{\frac{n}{4}})$ pamięci. Probabilistyczny algorytm Howgrave-Grahama i Jouxa pozwolił przyspieszyć rozwiązanie problemu do $O(2^{0.337n})$ zużywając $O(2^{0.256})$ pamięci. Dalsze jego ulepszenia pozwoliły osiągnąć złożoność czasową równą $O(2^{0.291n})$.

Znacznie mniej czasu wymagają instancje problemu sumy podzbiorów gdzie t jest względnie małe i dysponujemy $O(t)$ pamięci. Oparty o programowanie dynamiczne klasyczny algorytm wymaga $O(nt)$ czasu i $O(n+t)$ pamięci. Używam go do sprawdzenia poprawności implementacji algorytmu Ce Jin i Hongxun Wu. Najszybszym znanym algorytmem dla problemu sumy podzbiorów z małym t jest algorytm Konstantinos Koiliaris i Chao Xu.

Algorytm który prezentuję w niniejszej pracy został opracowany przez Ce Jin i Honguxun Wu. Działa on w czasie $O(n+t\log^2(t))$ i wymaga $O(n+t)$ pamięci. Jest to algorytm probabilistyczny z prawdopodobieństwem błędu $O(\frac{1}{n+t})$. Moja implementacja składa się z 4 modułów: klasy field odpowiadającej symulacji działań w ciele $\mathbb{Z}_p$ reszt z dzielenia przez liczbę pierwszą p, moduł losujący liczbę p korzystający z testu Millera-Rabina, moduł zawierający implementację teorio-liczbowej szybkiej transformaty Fouriera (jest to pewna nie ortodoksja względem pracy Ce Jin i Honguxun Wu, ponieważ proponowali oni szybką transformatę Fouriera, jednak początkowe próby jej implementacji

powodowały problemy związane z niedokładnością operacji zmiennoprzecinkowych) oraz implementacja algorytmu właściwego.

Moja implementacja jest napisana w języku `C++` i korzysta ze zmiennych całkowitych 128-bitowych więc kompilacja może nie powieść się na niektórych systemach i kompilatorach w szczególności na systemach 32-bitowych. Dokładne wymagania opiszę w dalszej części pracy.

W niniejszej pracy zaprezentuję niedeterministyczny algorytm opracowany przez Ce Jin i Hongxun Wu, który korzystając ze sprytnych obserwacji na polu analizy matematycznej i algebry jest w stanie podać wynik w czasie $O(n+t\log^2(t))$, co jest czasem znacznie szybszym niż klasyczny algorytm oparty na programowanie dynamiczne.

2 Algorytm klasyczny

Specyfikacja **problemu sumy podzbiorów**, który będziemy rozwiązywać jest następująca:

Wejście: Liczba naturalna n, liczba naturalna t, zbiór S reprezentowany jako ciąg n liczb naturalnych (po wczytaniu przechowywanych w wektorze `S`).
Wyjście: `true` jeżeli istnieje $S' \subseteq S$ którego suma elementów jest równa t lub `false` w przeciwnym przypadku.

Lub alternatywnie

Wejście: Liczba naturalna n, liczba naturalna t, zbiór S reprezentowany jako ciąg n liczb naturalnych.
Wyjście: $t+1$-elementowy wektor `ans` wartości boolowskich taki, że dla każdego $i \in \{0,1,2,...,t\}$ `ans[i]==true` wtedy i tylko wtedy, gdy istnieje $S' \subseteq S$ którego suma elementów jest równa t.

Klasyczny algorytm opiera się na modyfikacji $t+1$-elementowej tablicy `DP` o komórkach przyjmujących wartości bool-owskie i opiera się na programowaniu dynamicznym. Początkowo wszystkie komórki `DP` są ustawione na `false`, oprócz komórki 0-owej, która jest ustawiona na `true`.

Dokładny algorytm przedstawia poniższy pseudokod (odpowiada on drugiej specyfikacji problemu sumy podzbiorów. Jeżeli chcielibyśmy odpowiedź na pierwszą jego wersję wystarczy zwrócić `DP[t]`, przy czym można ją zwrócić wtedy od razu gdy przyjmie wartość `true`).

```
zainicjuj n-elementowe DP i oldDP
for i <-1,2,...,t+1
    oldDP[i] = false
end for
DPold[0] = true
for i<-0,1,...,n-1
    for j <- S[i],S[i]+1,...,t
        DP[j] = oldDP[j] or oldDP[j-S[i]]
    end for
    oldDP = DP
end for
return DP
```

Dowód poprawności tego algorytmu jest prostym dowodem indukcyjnym, w którym teza indukcyjna brzmi: po wykonaniu i-tej iteracji pętli z iteratorem `i` `DP[m]==true` wtedy i tylko wtedy gdy można wybrać podzbiór zbioru $\{S[0], S[1], .., S[i]\}$ taki, że suma jego elementów wynosi m (oczywiście dla $k \in \{0,1,...,t\}$).

- Dla $i = 0$ teza jest oczywista.
- Załóżmy, że teza jest prawdziwa dla $0,1,2,...,i-1$. Jeżeli istnieje podzbiór zbioru $\{S[0], S[1], .., S[i]\}$ taki, że suma jego elementów wynosi k to jest to albo podzbiór zbioru $\{S[0], S[1], .., S[i-1]\}$ i z tezy indukcyjnej `DP[k]==true` jeszcze przed wykonaniem i-tej iteracji albo jest to podzbiór zawierający element $S[i]$, którego pozostałe elementy należące do $\{S[0], S[1], .., S[i-1]\}$ sumują

się do `k-S[i]`, więc z tezy indukcyjnej `DP[k-S[i]]==true`. Ponieważ `DP[k]` po wykonaniu i-tej iteracji przyjmuje wartość `DP[i] or DP[k-S[i]]` teza indukcyjna jest prawdziwa.

Ponieważ komórki `DP` przyjmują tylko wartości `true` i `false` do reprezentacji jej najoptymalniej użyć bitsetu a kolejne iteracje pętli zewnętrznej wykonać poleceniem `DP |= (DP » S[i])`.

Pętla zewnętrzna wykona się $O(n)$ razy i każda iteracja zajmie $O(t)$ czasu tak więc czas działania całego algorytmu zajmie $O(nt)$ i $O(t+n)$ pamięci.

Algorytm naiwny został zaimplementowany w pliku `main.cpp` jako funkcja `brutalSum`.

3 Idea algorytmu Ce Jin i Hongxun Wu

Algorytm Ce Jin i Hongxun Wu bazuje na dość prostej obserwacji, że dla danego zbioru $S = \{s_1, s_2, ..., s_n\} \subset \mathbb{N}$ istnieje jego podzbiór sumujący się do $t \in \mathbb{N}$ wtedy i tylko wtedy gdy, współczynnik przy x^t w wielomianie $A(x) := \prod_{i=1}^{n}(1+x^{s_i})$ jest niezerowy. Istotnie jeżeli ten współczynnik jest niezerowy, to istnieje ciąg indeksów $0 < i_1 < i_2 < ... < i_k < n+1$ taki, że

$$\prod_{j=1}^{k} x^{s_{i_j}} = x^{\sum_{j=1}^{k} s_{i_j}} = x^t$$

Tak więc ciąg ten odpowiada indeksom elementów które należy wybrać by uzyskać podzbiór zbioru S o sumie elementów równej t. Zamiast liczyć ten współczynnik wprost najpierw obliczymy $B(x) := \ln(\prod_{i=1}^{n}(1+x^{s_i}))$, zaś następnie $\exp(B(x)) = A(x)$.

Co to jednak znaczy, że obliczymy te funkcje? Będziemy wyliczać rozwinięcia jej w szereg Taylora. Współczynnik przy x^t w rozwinięciu w szereg Taylora $\exp(B(x))$ istotnie jest współczynnikiem przy x^t w $A(x)$. Wynika to bezpośrednio z jednoznaczności rozwinięcia w szereg potęgowy i tego, że $A(x)$ jest wielomianem.

W dalszej części sekcji będę stosował schemat notacji $F_t(x)$ jako oznaczenie rozwinięcia w szereg Taylora t-pierwszych wyrazów funkcji F, tak więc $\exp_t(x) = \sum_{i=0}^{t} \frac{x^i}{i!}$, zaś $\ln_t(1+x^a) = \sum_{i=0}^{\lfloor \frac{t}{a} \rfloor} \frac{(-1)^{i+1}x^{ai}}{i}$.

Aby znaleźć odpowiedź na omawiany w tej pracy problem wystarczy oczywiście ustalić jedynie wartość t pierwszych wyrazów rozwinięcia w szereg Taylora funkcji $\exp(B(x))$.

Niestety obliczenia potrzebne do znalezienia tych współczynników mogą wymagać użycia bardzo dużych liczb długości $O(n)$. Obliczenia na nich mogą być więc czasochłonne.

Ce Jin i Hongxun Wu skorzystali z faktu, że pochodna nie jest jedynie obiektem, który można zdefiniować analitycznie jako przekształcenie funkcji $f(x)$ w funkcję zwracającą dla argumentu x wartość $\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$, ale i przekształcenie czysto algebraiczne, które przekształca szereg $\sum_{i=0}^{\infty} f_i x^i$ w $\sum_{i=1}^{\infty} i f_i i x^{i-1}$.

Tak zdefiniowana pochodna zachowuje pewne algebraiczne właściwości($f' + g' = (f+g)'$,$(fg)' = f'g + g'f$, $(f(g))' = f'(g)g'$) bez względu na to do jakiego ciała należą współczynniki tych szeregów, tak więc również pojęcie szeregu Taylora (z pewnymi ograniczeniami) możemy rozpatrywać w innych ciałach niż $\mathbb{R}$. W naszym przypadku będzie to ciało $\mathbb{Z}_p$ reszt z dzielenia przez p, gdzie p jest losową liczbą pierwszą na tyle dużą, że prawdopodobieństwo fałszywego zakwalifikowania jakiejś liczby jako 0 jest niewielkie, jednak na tyle małą, że obliczenia na elementach $\mathbb{Z}_p$ wykonują się względnie szybko.

Ważną optymalizacją w liczeniu współczynników rozwinięcia $\exp(B(x))$ było zastąpienie jednej podwójnej pętli, której wykonanie w sposób naiwny zajmowałoby pesymistycznie $O(t^2)$ czasu pomnożeniem dwóch wielomianów stopnia co najwyżej $O(t)$, co można wykonać szybką transformatą Fouriera w czasie $O(\log(t)t)$. Aby uniknąć problemów z utratą dokładności w obliczeniach na liczbach zmiennoprzecinkowych zastosowałem Teorio-liczbową szybką transformatę Fouriera, w której współczynniki rozpatrywałem w ciele wylosowanej już na potrzeby wcześniej omawianych obliczeń liczby p.

Ogólnie implementację algorytmu można podzielić na 5 zasadniczych części

- Implementacja funkcji losującą liczbę pierwszą p o pożądanych własnościach.
- Implementacja klasy odpowiadającej za wykonywanie obliczeń w ciele Z_p.
- Implementacja Teorio-liczbowej szybkiej transformaty Furiera w ciele Z_p.
- Implementacja algorytmu właściwego.
- Implementacja kodu testującego, w tym algorytmu naiwnego.

4 Implementacja ciała Z_p

Ogólny szablon, jakie metody należy zaimplementować był mocno inspirowany implementacją klasy dużych liczb całkowitych zamieszczony na stronie [https://www.geeksforgeeks.org/bigint-big-integers-in-c-with-example/]

Ponieważ znaczną część obliczeń mój program będzie wykonywać na elementach ciała $\mathbb{Z}_p$ w którym istnieją działania arytmetyczne, aby uniknąć konieczności ciągłych operacji pobierania explicite wartości modulo p z wyników działań najwygodniejszym podejściem byłoby zaprogramowanie klasy `field` której instancje reprezentują elementy ciała $\mathbb{Z}_p$ zaś przeciążone operatory odpowiadają działaniom w $\mathbb{Z}_p$ i pisząc bardziej wysokopoziomowy kod nie musimy już się troszczyć o to, by jakkolwiek "normalizować"wyniki.

Klasa ma kilka pól statycznych. Jednym z nich jest liczba `p` typu `__int128` będąca liczbą pierwszą modulo którą będziemy wykonywać nasze obliczenia. Liczba `m` typu `__int128`, która ma zastosowanie przy wykonywaniu działania mnożenia w sposób chroniący nas przed przepełnieniem zmiennej co zostanie omówione w dalszej części.

Liczbę `p` można zapisać jako $2^k r + 1$, gdzie r jest liczbą nieparzystą zaś k liczbą naturalną. W zmiennej statycznej typu `__int128` i nazwie `odd` przechowujemy wartość owego r, zaś w statycznej zmiennej `degreeOfDegree` typu `__int128` przechowujemy wartość owego k.

Dla każdej liczby pierwszej p istnieje pierwiastek pierwotny, czyli taka liczba, że x, że nie istnieje dodatnia liczba naturalna w mniejsza niż $p-1$ taka, że $x^w \equiv 1(\mod p)$ (oczywiście z małego twierdzenia Fermata $r^{p-1} \equiv 1(\mod p)$). W dalszej części pracy stopniem liczby/pierwiastka będę oznaczał najmniejszy wykładnik naturalny do którego należy ją podnieść, aby uzyskać 1 w ciele Z_p. Stopień pierwiastka pierwotnego wynosi oczywiście $p-1$. Bardzo często będziemy chcieli szybko wyliczyć liczbę mającą stopień będący potęgą 2 o wykładniku naturalnym k. Najprościej taką liczbę uzyskać jako rezultat potęgowania pierwiastka pierwotnego, najpierw do potęgi, której wykładnik jest równy wartości zmiennej statycznej `odd`, a następnie wykonaniu `degreeOfDegree-k` podniesień wyniku do kwadratu. Wygodnie byłoby nie musieć za każdym razem wykonywać potęgowania do potęgi `odd`, dlatego w zmiennej statycznej `almostPrimitiveRoot` typu `__int128` przechowuję wartość będącą wynikiem podniesienia pierwiastka pierwotnego do potęgi `odd`.

Ostatnim polem statycznym klasy `field` jest struktura typu `std::map<__int128 , __int128 >` i nazwie `inverse` przechowująca odwrotności elementów ciała. Będzie ona dynamicznie powiększana w trakcie wykonania programu i służy ominięciu konieczności wykonywania dużej ilości powtórzeń kosztownej operacji liczenia odwrotności elementu.

Metoda `setP` ustawia wartość `p` na przyjętą jako argument wartość zmiennej `x` typu `__int128`. Ustawia ona również zmienną `m` na wartość $\lfloor \frac{2^{127}-1}{p} \rfloor$. Czyści tablicę `inverse` (wcześniej wyliczone odwrotności elementów po zmianie ciała przestają być dłużej odwrotnościami tych samych elementów). Wykonując poniższy kod znajduję wartość pól `almostPrimitiveRoot` (w zmiennej `degree`) oraz `degreeOfDegree`(w zmiennej `y`) i następnie przypisujemy je odpowiednim polom. `odd`

```
__int128 y = field::p - 1;
__int128 degree = 0;
while(y \% 2 == 0){
    degree++;
    y /= 2;
}
```

Następnie szukamy wartości pola `almostPrimitiveRoot`. Przy założeniu prawdziwości hipotezy Riemanna pierwiastek pierwotny modulo p jest stosunkowo mały wielkości $O(\log^6(p))$ [ŹROOOOOOOOOOOOOOODŁOOOOOOOOOOOO!!!!!!!!!], tak więc kandydatów możemy szukać sprawdzając kolejne liczby naturalne zaczynając od 2. Tak naprawdę nie potrzebujemy wyznaczyć pierwiastka pierwotnego, ale jedynie liczbę jedynie liczbę r, że nie zachodzi $r^{2^{degreeOfDegree-1}} \equiv 1(\mod p)$. Sprawdzenie dla liczby r czy nie zachodzi $r^{2^{\frac{p-1}{2}}} \equiv 1(\mod p)$ jest mocniejszym warunkiem i dla uproszczenia kodu on zostaje sprawdzony. Następnie jeżeli wspomniany warunek zachodzi do zmiennej `almostPrimitiveRoot` zapisujemy r^{odd}.

W module zajmującym losowaniem liczb pierwszych istnieje miejsce w którym wykonujemy operacje modularne modulo liczb które nie muszą być pierwsze jednak nie wykonujemy modulo je dzielenia. Wygodnie byłoby móc użyć metod klasy `field`. Aby uniknąć potencjalnych niespodziewanych zacho-

wań programu np podczas wyznaczania wartości pola `almostPrimitiveRoot` podczas wywołania funkcji `setP` na argumencie niebędącym liczbą pierwszą utworzyłem funkcję `setPstupid`, która przyjmuje argument `x` typu `__int128` i ustawia na niego wartość pola `p`, wartość pola `m` ustawia podobnie jak w funkcji `setP` na $\lfloor \frac{2^{127}-1}{p} \rfloor$, pozostałe pola statyczne zaś ustawia na `0` i czyści tablicę odwrotności.

Pole, które przechowuje wartość pojedynczego obiektu nazwałem `value`. Oczywiście pole to już nie może być statyczne. Stworzyłem kilka konstruktorów dla klasy `field`. Konstruktor domyślny nie przyjmuje żadnego argumentu i ustawia wartość `value` na `0`. Konstruktor kopiujący ustawia wartość `value` na wartość `value` obiektu typu `field` podanego jako argument.

Ostatni konstruktor przyjmuje liczbę całkowitą `val` typu `__int128`. Żeby uniknąć nietypowych zachowań programu chcemy, żeby wartości `value` dla każdego obiektu klasy `field` były mniejsze niż `p` i większe niż `0`. Jeżeli więc wartość `val` jest większa lub równa `0` polu `value` przypisuję jej resztę z dzielenia przez `p`. Jeżeli jest zaś jest ujemna przypisuję mu wynik działania `((val%p)+p)%p`.

Przejdę teraz do omówienia kolejnych funkcji i przeciążonych operatorów w omawianej klasie.

Utworzyłem kilka get-terów, żeby w sposób bardziej kontrolowany odwoływać się do niepublicznych pól. `getP`, `getM`, `getAlmostPrimitiveRoot`, `getOdd`, `getDegreeOfDegree` służą odpowiednio do pobrania wartości pól `p`, `m` i `almostPrimitiveRoot`.

Metody `friend field inline & operator++()`, `friend field inline operator++(int temp)`, `friend field inline & operator--()`, `friend field inline & operator++(int temp)` to odpowiednio przeciążenia operatora postinkrementacji, preinkrementacji, postdekrementacji i predekrementacji. Operator postinkrementacji korzystając z niezmiennika, mówiącego, że wartość pola `value` jest zawsze liczbą całkowitą z przedziału $[0, 1, .., p-1]$ który zachowują wszystkie funkcje oraz konstruktory modyfikujące pole `value` może przypisać polu `value` po prostu wartość wyrażenia `(value+1)%field::p`. Jeżeli `value=0` naiwna postdekrementacja powoduje uzyskanie wyniku ujemnego dlatego polu `value` w przypadku postdekrementacji zamiast `(value+1)%field::p` przypisujemy `(field::p+value-1)%field::p`. Dla operatora preinkrementacji najpierw tworzymy zmienną `aux` w której przechowujemy starą wartość obiektu, którą później zwrócimy, a następnie na oryginalnym obiekcie dokonujemy zdefiniowaną wcześniej postinkrementację. Operator predekrementacji definiujemy analogicznie jak preinkrementacji.

Następnie definiuję przeciążenia operatorów `friend inline field &operator+=`, `friend inline field &operator+`, `friend inline field &operator-=`, `friend inline field &operator-`. Operator `+=` przyjmuje przez referencję dwa argumenty typu `field`, gdzie pierwszy oznaczam jako `a`, zaś drugi jako `b`. Wartości pola `value` przypisujemy wartość `( a.value + b.value ) %field::p` gdzie operator `%` służy zachowaniu niezmiennika, by wartości pola `value` były mniejsze niż `p`. Z kolei dla operatora `-=` wartości analogicznego pola przypisujemy wartość wyrażenia `(a.value-b.value+field::p)%field::p` gdzie nieobecne w poprzednim przypadku dodanie `p` służy niedopuszczeniu do sytuacji gdy `value` stanie się ujemne jeśli `b.value>a.value`. W implementacji operatora `+` tworzymy tymczasowy obiekt `temp` klasy `field` o wartości początkowej pola `value` równej `a.value`, a następnie przy pomocy zdefiniowanego wcześniej operatora `+=` dodajemy do tego obiektu obiekt `b`. Analogicznie definiujemy operator `-`.

Kolejnymi nieco prostszymi operatorami jakie zdefiniowałem to operatory porównujące `friend inline bool operator==`, `friend inline bool operator!=`, `friend inline bool operator<`, `friend inline bool operator<=`, `friend inline bool operator>`, `friend inline bool operator>=` które dla argumentów `a` i `b` zwracają wartości boolowskie odpowiednio wyrażeń `a.value == b.value`, `a.value != b.value`, `a.value < b.value`, `a.value <= b.value`, `a.value >= b.value`.

Nieco bardziej skomplikowana jest implementacja operatora `friend field &operator*=`. Ponieważ liczba `p` mnożę być rzędu $O(t(n+t)^3)$ gdybyśmy postąpili naiwnie i (przyjmując oznaczenie pierwszego argumentu jako `a`, drugiego jako `b`) przypisali polu `value` wyniku wartość wyrażenia `(a.value*b.value)%field::p` oznaczałoby to pobranie reszty z dzielenia przez `p` z wartości wyrażenia `a.value*b.value`, która byłaby rzędu $O(t^2(n+t)^6)$, co już dla `t` rzędu 10^5 mogłoby powodować wychodzenie poza zakres nawet zmiennej typu `__int128`. Oczywiście tak mały zakres liczb mocno wpłynąłby na użyteczność zaimplementowanych funkcji. Definiuję więc zmienną `m`, będącą górną wartością `b`, dla którego możemy wykonać mnożenie w sposób naiwny (i następnie wyciągnąć tylko resztę z dzielenia przez `p`) bez obaw o przepełnienie. W przeciwnym wypadku tworzę zmienną pomocniczą `A` będącą wartością pola value obiektu `a` oraz zmienną `B` będącą wartością pola value obiektu `b`.

Na potrzeby dalszej części pracy definiuję operator matematyczny % który dla lewego argumentu będącego liczbą naturalną A i prawego będącego liczbą naturalną dodatnią B zwraca resztę z dzielenia A przez B. Jest więc bardzo podobny do operatora `%` w składni C++.

Zauważmy, że $A \cdot B = A \cdot \lfloor \frac{B}{m} \rfloor + A \cdot (B\%m)$. Wartość `m` jest zależna od `p` i dlatego ustawiamy ją podczas operacji `setP` o czym pisałem w jednym poprzednich akapitów. Inicjalizuję obiekt `ans1` klasy `field` o wartości pola `value` równej `A`. Następnie mnożę go rekurencyjnie przy pomocy operatora `*=` przez `field(fied::m)`, a następnie przez `field(B/field::m)`. Inicjalizuję też obiekt `ans2` o początkowej wartości pola `value` równej `A` i mnożę rekurencyjnie przy pomocy operatora `*=` przez field(B%field::m). Po tych operacjach wartość pola `value` obiektu `ans1` wynosi $A \cdot \lfloor \frac{B}{m} \rfloor$, zaś obiektu `ans2` $A \cdot (B\%m)$. Do `a` zapisujemy więc `ans1+ans2` i zwracamy `a`.

Kolejnym operatorem jest operator `friend field operator*`. Argumenty, które otrzymuje to obiekty typu `field` o nazwach `a` i `b`. Tworzę tymczasowy obiekt typu `field` i nazwie `temp` o tej samej wartości pola `value` co `a` i następnie mnożę go przez `b` przy pomocy zdefiniowanego wcześniej operatora `*=` i zwracam tak zmodyfikowany obiekt `temp`.

Kolejnym operatorem jest operator potęgowania `friend field inline &operator^=`. Pierwszym argumentem jaki przyjmuje jest obiekt typu `field` o nazwie `a` oraz zmienna typu `__int128` o nazwie `b`. Zauważmy, że z małego twierdzenia Fermata (jeżeli p nie dzieli a, jeżeli jednak dzieli również własność zachodzi, czego dowód jest trywialny) $a^{k(p-1)+l} \equiv (a^{p-1})^k a^l \equiv 1^k a^l \equiv a^l (\mod p)$ dla dowolnych naturalnych a, k, l, tak więc zamiast podnosić `a` do potęgi `b` możemy podnieść ją do potęgi `b%(p-1)`. Teoretycznie moglibyśmy rozpatrywać `((b%(p-1))+p-1) % (p - 1)`, jednak z pewnych powodów, o których napiszę w dalszej części pracy zdecydowałem się rozpatrzyć przypadki `b` ujemnego i dodatniego osobno.

Należy również pamiętać, że po zaimplementowaniu funkcji `setPstupid` nasza klasa może symulować niektóre działania modulo `p` dla `p` nie będącego liczbą pierwszą. Oczywiście wtedy małe twierdzenie Fermata przestaje pozwalać na ową redukcję `b` do reszty z dzielenia przez `p-1`, tak więc przypadek ten skorygowałem prostym `if`-em, sprawdzającym czy wartość pola `odd` jest równa `0`, co może się zdarzyć tylko wtedy gdy klasa jest inicjalizowana funkcją `setPstupid`.

Jeżeli `b==0` przypisuję `a` wartość `1` i zwracam wynik. Jeżeli `b==1` nie robię nic, tylko od razu zwracam `a`. Jeżeli `b==-1` to jeżeli `a.value==0` wyrzucam błąd dzielenia przez `0`, jeżeli zaś nie wykonuję następującą procedurę. Ponieważ potęgowanie do `-1` to znalezienie odwrotności `a` w $\mathbb{Z}_p$ z małego twierdzenia Fermata oznacza to podniesienie `a` do potęgi `p-2`. Potęgowanie można zaimplementować, by wykonywało się w czasie logarytmicznym względem stopnia, jednak mnożenie ze względu na groźbę przepełnienia też może w pesymistycznym przypadku zabierać logarytmicznie dużo czasu. Znalezienie odwrotności obiektu jest dość podstawową operacją, którą możemy chcieć wykonywać bardzo często, tak więc czas $O(\log^2(p))$ może nie być zadowalający. Dlatego klasa `field` posiada jeszcze jedno pole, którym jest statyczna mapa o nazwie `inverse` z wejściami typu `<__int128,__int128>`, która przypisuje liczbie `x` liczbę która jest wartością odwrotną w ciele $\mathbb{Z}_p$. Mapę tą czyścimy, za każdym razem gdy zmieniamy wartość `p`. Istotnie wtedy odwrotności tych samych liczb mogą stać się inne, bo zmienia się ciało w którym są tymi odwrotnościami. Operacja `insertInverse` przyjmuje zmienne typu `field`, które zakładamy explicite, że są elementami wzajemnie odwrotnymi i oznaczamy je jako `a` oraz `b`, a następnie, jako, że $(a^{-1})^{-1} = a$ wykonujemy od razu 2 przypisania

```
field::inverse[a.value] = b.value;
field::inverse[b.value] = a.value;
```

Gdy mamy wyliczyć potęgę o wykładniku `-1` jakiegoś elementu `a` typu `field` sprawdzam najpierw czy w mapie odwrotności jest przypisana wartość dla klucza `a.value`, a następnie jeżeli kluczowi temu przypisana jest wartość ustawiamy `a.value` na nią i zwracamy `a`, zaś w przeciwnym przypadku wykonujemy operację `field::insertInverse(a, A^=(( long long)(field::p-2)))`, gdzie `A` to kopia obiektu `a`. (rekurencyjnie wywołujemy operator `^=` dla wartości nieujemnej) i ponownie pobieramy wartość dla klucza `a.value`. Pozwala nam to dla każdego elementu $\mathbb{Z}_p$ wyliczyć jego odwrotność co najwyżej raz, bez względu na ilość wywołań tego wyliczenia w wyżej poziomowym kodzie.

Kolejnym przypadkiem, który należy rozważyć to gdy `b>1`. Wykonuję wtedy iteracyjną wersję szybkiego potęgowania. Tworzę pomocniczy obiekt `multiplier` będący kopią `a` oraz obiekt `ans` o wartości pola `value` równej 1. Następnie wykonuję pętlę

```
while(b != 0){
```

```
    if(b % 2 == 1)ans *= multiplier;
    multiplier *= multiplier;
    b /=2;
}
```

i następnie kopiuję wartość `ans` do obiektu `a` oraz zwracam `a`.

Kolejnym i już ostatnim przypadkiem jest gdy `b < -1`. W tym przypadku wykorzystujemy fakt, że $a^b = (a^{-1})^{-b}$. Tak więc kolejno wykonujemy zdefiniowane wcześniej podniesienie `a` do potęgi `-1` przy pomocy operatora `^ =` a następnie znów przy pomocy tego samego operatora podnosimy `a` do potęgi `-b>1`, co też zostało już wcześniej zdefiniowane. Po wykonaniu tych obliczeń zwracam `a`.

Kolejnym operatorem jest `^`. Pobiera on argument `a` typu `field` i `b` typu `__int128`, tworzy przy pomocy konstruktora kopiującego tymczasowy obiekt `temp` typu `field` z taką samą wartością pola `value` co `a`, następnie przy pomocy operatora `^=` podnosi `temp` do potęgi `B` i zwraca tak zmodyfikowaną wartość `temp`.

Kolejnym operatorem będzie operator dzielenia `\=` przyjmujący dwa argumenty typu `field` oznaczone odpowiednio jako `a` i `b`. Dzielenie w ciele modulo nie oznacza "klasycznego"dzielenia arytmetycznego, ale pomnożenie przez element odwrotny. Tak więc tworzymy nowy obiekt `B` będący kopią `b` i mnożymy `a` przy pomocy operatora `*=` przez `B^(longlong)(-1)`. Następnie zwracamy tak zmodyfikowane `a`.

Kolejnym operatorem jest `\`. Otrzymuje on dwa argumenty typu `fieled` o nazwach `a` i `b`. Tworzę obiekt `temp` typu field i przypisuję mu wartość `a`, następnie dzielę przez `b` przy pomocy operatora `\=` i zwracam `temp`.

Kolejnymi operatorami są operatory strumieniowe
`std::istream &operator» (std::istream &in,field&a)` i
`std::ostream &operator«(std::ostream &out,const field &a)`.

Wypisanie/wczytanie elementu będzie polegać na wypisaniu/wczytaniu wartości jego pola `value`. Jednak kompilator `g++` nie ma zdefiniowanych operatorów strumieniowych dla zmiennych typu `__int128`. Dlatego będę w przypadku operatora » najpierw wczytywał zapis żądanej wartości pola `value` do zmiennej typu `std::string`, a następnie przy pomocy funkcji `fromString` przekształcał ją do zmiennej typu `__int128` i zwracał podany jako argument strumień `in`. Z kolei w przypadku operatora « będę konwertował zmienną typu `__int128` do zmiennej typu `std::string` i dopiero po konwersji wypisywał na podany jako argument strumień `out` i zwracał ten strumień.

Funkcja `fromString` używa funkcji `charToDigit`, która przyjmując znak `x` typu `char` będący zapisem jakiejś cyfry w języku "ludzkim"zwraca liczbę typu `__int128` mającą wartość tej cyfry. Funkcja `fromString` przyjmuje zmienną `x` typu `std::string` jako argument a następnie przechodząc w pętli od tyłu po znakach tego słowa dodaje do zmiennej wynikowej `ans` kolejne potęgi `10` o wykładniku naturalnym pomnożone przez wartość wyniku funkcji `charToDigit` na rozpatrywanym znaku. Jeżeli zaś napotka na znak `-` mnoży `ans` przez `-1`.

Funkcja `toString` korzysta z funkcji `toStringOneNumber`, która przyjmuje liczbę typu `__int128` i zwraca zapis jej reszty z dzielenia przez `10` w formie zmiennej typu `string`. Funkcja `toString` przyjmuje wartość `x` typu `__int128` Jeżeli `x==0` zwraca `"0"`. Jeżeli `x` jest ujemne zapamiętuje to w zmiennej `minus` typu `bool` i następnie mnoży w miejscu `x` przez `-1`. Funkcja inicjalizuję zmienną wynikową `ans` na słowo puste, a następnie wykonuje pętlę

```
    while(x != 0){
                ans = toStringOneNumber(x)+ans;
                x /=10;
        }
```

W przypadku gdy zmienna `minus` została ustawiona na `true` dodaję z przodu wyniku słowo `"-"`. Po wykonaniu omówionych operacji zwracam wynik.

Omówiony w powyższej sekcji kod został napisany w języku $C++$ i znajduje się w pliku $field.h$ w dołączonym do pracy repozytorium.

5 Szukanie liczby pierwszej

W pracy Jin i Wu liczba pierwsza którą losujemy ma tylko jedno zastosowanie. Jest nim wyznaczenie ciała w którym będziemy wykonywać nasze obliczenia. Losowanie jej odbywa się spośród liczb na

przedziale $[t+1,(n+t)^3]$.

Definicja: $\pi(x)$ jest funkcją $\mathbb{N} \to \mathbb{N}$ przypisująca argumentowi x liczbę liczb pierwszych nie większych niż x.

Twierdzeniem o liczbach pierwszych mówi, że: [ŹROOOOOOOOOOOOOOODŁOOOOOOOOOOOO]

$$\lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{\ln(x)}} = 1$$

Tak więc liczb pierwszych na przedziale $[t+1,(n+t)^3]$ jest $\Omega((n+t)^2)$. Celem naszego algorytmu jest sprawdzenie czy dla danego multizbioru $\{x_1,x_2,...,x_n\} \equiv \mathbb{N}$ i liczby t współczynnik przy x^t w wielomianie $\prod_{i=1}^{t}(1+x^{s_i})$ jest dodatni. Ponieważ obliczenia wykonujemy w ciele $\mathbb{Z}_p$ jeżeli ów współczynnik byłby podzielny przez p fałszywie uznalibyśmy go za zerowy. Jednak współczynnik ten będzie nie większy od 2^n, tak więc ma najwyżej $O(n)$ dzielników pierwszych, więc prawdopodobieństwo, że wylosowana liczba dzieli ten współczynnik jest $O(\frac{1}{n+t})$

Różnica, między moją implementacją, a algorytmem opisanym przez Ce Jin i Hongxun Wu jest jednak taka, że nie używałem klasycznej wersji szybkiej transformaty Fouriera, opartej na liczbach zespolonych, a teorio-liczbowej szybkiej transformaty Fouriera wykonującej obliczenia w ciele $\mathbb{Z}_p$. W tym ciele musi istnieć pierwiastek z 1 stopnia r gdzie r jest postaci 2^k dla pewnej liczby naturalnej k, oraz r jest większe niż stopnień wielomianu, który powstanie po pomnożeniu wielomianów, do których mnożenia używamy naszą transformatę. Mnożone wielomiany są stopnia co najwyżej t, tak więc możemy przyjąć, że 2^k to najmniejsza potęga 2 o całkowitym wykładniku większa niż $2t$. zaś p ma postać $r2^k+1$.

Twierdzenie: Jeżeli w ciele $\mathbb{Z}_p$ istnieje pierwiastek stopnia 2^k z 1 to p ma postać $r2^k+1$, gdzie r jest liczbą naturalną.

Istotnie dla każdej liczby pierwszej p istnieje jej pierwiastek pierwotny a, którego stopień jest równy $p-1$, czyli w tym przypadku $r2^k$, zaś stopień a^r jest równy 2^k.

Intuicyjnie można przypuszczać, że ponieważ rozmieszczenie liczb pierwszych jest trudne do opisania prostym wzorem, to podobny procent liczb postaci $r2^k+1$ gdzie r jest liczbą naturalną z przedziału $[t+1,(n+t)^3]$ jest pierwszy, co po prostu, procent liczb pierwszych na przedziale $[t+1,(n+t)^3]$.

Definicja: $\pi'(x,t)$ to funkcja $(\mathbb{N}\times\mathbb{N}) \to \mathbb{N}$, która dla argumentów x i t zwraca liczbę liczb pierwszych postaci $r2^k+1$, gdzie 2^k to najmniejsza potęga 2 o wykładniku naturalnym większa niż $2t$, zaś r jest liczbą naturalną mniejszą lub równą x.

Losowanie liczby pierwszej p o żądanych właściwościach będzie polegać najpierw na wylosowaniu liczby naturalnej r z przedziału $[t+1,(n+t)^3]$, a następnie wyliczeniu liczby $k2^r+1$, gdzie 2^k to najmniejsza potęga 2 o wykładniku naturalnym większa niż $2t$ i sprawdzeniu, czy tak wyliczona liczba jest pierwsza.

Dla danych t i n tym sposobem możemy wylosować $\pi'((n+t)^3,x)-\pi'(t+1,x)$ różnych liczb pierwszych. Jeżeli więc $\pi'((n+t)^3,x)-\pi'(t+1,x)$ jest co najmniej $\Omega((n+t)^2)$ to zachodzi szacowanie prawdopodobieństwa błędu analogiczne do szacowania Jin i Wu. Wyliczenie dokładnej wartości $\pi'((n+t)^3,x)-\pi'(t+1,x)$ dla dużych n i t jest trudne dlatego oszacowałem je poprzez losowanie `1000000` liczb z przedziału $[1+t,(n+t)^3]$, pomnożeniu przez 2^k, gdzie 2^k to najmniejsza potęga 2 o wykładniku naturalnym większa niż $2t$ i dodaniu 1, oraz zliczeniu w zmiennej `ans` ile tak wylosowanych liczb jest pierwsze. $\frac{ans}{1000000}((n+t)^3-(1+t)+1) \approx \pi'(n,t)$.

Poniższa heat-mapa ukazuje tak przybliżoną wartość $\log_2(\frac{\pi'(n,t)}{(n+t)^2})$, w zależności od $\log_2(n)$ i $\log_2(t)$. Jak widać zdecydowanie liczba liczb pierwszych możliwych do uzyskania w omówiony sposób jest $\Omega((n+t)^2)$.

W przeciwieństwie do większości pozostałego kodu, który przygotowałem, ze względu na jednoczesne łatwe wykonywanie wykresów, oraz szybkość działania eksperymentalne weryfikowanie tej hipotezy przeprowadziłem w języku Julia. Kod użyty do testowania znajduje się w pliku `PrimesExperiment.jl`.

Omówię teraz już samą implementację modułu szukającego liczby pierwszej. Wszystkie wymienione niżej funkcje zostały zaimplementowane w języku C++ i znajdują się w pliku `FindPrimes.cpp`.

Moduł losujący składa się z kilku funkcji, z których zdecydowanie najprostszą jest `pow2` która dla przyjmowanej jako argument liczby `t` zwraca najmniejszą potęgę `2` z wykładnikiem naturalnym, która jest większa niż `2t`. Polega ona na mnożeniu przez `2` zmiennej `ans` której początkowa wartość jest równa `1` dopóki nie będzie większa niż `2*t`. Musimy wykonać co najwyżej kilkadziesiąt mnożeń, co jest na tyle szybkie, że nie ma potrzeby stosowania jakiś bardziej zaawansowanych algorytmów jak na przykład używających algorytmu szybkiego potęgowania.

Funkcja `randomLongLong` służy losowaniu liczby dodatniej 64-bitowej nie większej niż podana jako argument liczba `mod`. Należąca do standardu C++ funkcja `rand` zwraca liczby 32-bitowe, co może nie być dla nas wystarczające. Jeżeli `mod` mieści się w zakresie zmiennej typu `int` zwracam po prostu resztę z dzielenia przez `mod` wartości bezwzględnej wyniku wywołania `rand()`. Jeżeli zaś jest większe niż $2^{31}-1$ `31` ostatnich bitów wyniku zapisujemy w zmiennej `candidate` jako resztę z dzielenia przez 2^{30} wartości bezwzględnej wyniku wywołania funkcji `rand()`, a następnie losuję bity wiodące jako wynik operacji `std::abs(((long long)std::rand())%((long long)mod/(long long)1073741824))` i zapisuję je w zmiennej `candidate2`. Może się zdażyć, że niektóre liczby mniejsze niż `mod` będą nieosiągalne, jednak jest z pewnością mniej niż $\frac{mod}{2}$, więc nie będzie to wpływać na asymptotyczną złożoność prawdopodobieństwa błędu.

Podobną konstrukcję ma funkcja `random128` służąca do losowania dodatniej liczby typu `__int128` z przedziału $[0, 1, ..., mod]$, gdzie `mod` to liczba typu `__int128` podana jako argument. Rolę jaką w funkcji `randomLongLong` pełnią funkcja `rand` i liczby $2^{31}-1$i 2^{30} w kodzie `random128` pełnią odpowiednio funkcja `randomLongLong` i liczby $2^{63}-1$ i 2^{62}. Nie musimy też w funkcji również troszczyć się o wyciąganie wartości bezwzględnej bo funkcja `randomLongLong` zwraca tylko

Sprawdzenie czy wylosowana liczba jest pierwsza wykonuję przy pomocy testu Millera-Rabina.

Jego idea jest następująca: każdą liczbę pierwszą p większą niż 2 da się jednoznacznie zapisać w postaci $d2^s+1$ gdzie d jest nieparzyste, zaś s naturalne. Dla dowolnej liczby $a \in \{2, 3, ..., p-1\}$ z małego twierdzenia Fermata p dzieli

$$a^{p-1}-1 = a^{d2^s}-1 = (a^d)^{2^s}-1 = ((a^d)^{2^{s-1}}+1)((a^d)^{2^{s-1}}-1) =$$

$$((a^d)^{2^0}-1)((a^d)^{2^0}+1)((a^d)^{2^1}+1)((a^d)^{2^2}+1)...((a^d)^{2^{s-1}}+1)$$

Tak więc z wyboru a wiemy, że któraś z liczb $((a^d)^{2^0}+1), ((a^d)^{2^1}+1), ..., ((a^d)^{2^{s-1}}+1)$ jest podzielna przez p, czyli dla pewnego $i \in \{0, 1, 2, ..., s-1\}$ zachodzi

$$(a^d)^{2^i}+1 \equiv 0(\mod p)$$

czyli równoważnie

$$(a^d)^{2^i}+1 \equiv n-1(\mod p)$$

Test Millera-Rabina opiera się na następującej obserwacji:

Obserwacja 1: Dla większej niż 2 liczby $n = 2^s d+1$, gdzie s jest naturalne, zaś d naturalne nieparzyste i liczby $a \in \{2, 3, ..., n-1\}$ jeżeli istnieje liczba $i \in \{0, 1, 2, .., s-1\}$ taka, że $(a^d)^{2^i} \equiv n-1(\mod p)$ to n prawdopodobnie jest pierwsza. Jeżeli takie i nie istnieje n liczbą pierwszą nie jest na pewno.

prawdopodobieństwo błędu można minimalizować wybierając kilka liczb a.

Fakt 1: Jeżeli hipoteza Riemanna jest prawdziwa wystarczające okazuje się sprawdzenie sprawdzenie a z mniejsze lub równe $\max(n-2, 2\ln^2(n))$.[ŹROOOOOOOOOOOOOOODŁOOOOOOOOOOOO]

Moja implementacja testu Millera Rabina składa z dwóch funkcji.

Pierwsza z nich to `MillerRabinOne` mająca za zadanie sprawdzić warunek z **Obserwacji 1** dla określonych `n` i `a` zwracając `false` gdy wiemy, że liczba nie jest pierwsza. Na początku rozważam przypadki gdy `n` i `a` nie spełniają początkowych założeń dla których **Hipoteza 1** była formułowana przy pomocy następujących zapytań warunkowych:

```
\begin{}t
if(n == 2) return true;
if(n < 2) return false;
if(n \% 2==0) return false;
if(a \% n == 1) return true;
if(a > n) return true;
```

Jeżeli warunki początkowe są spełnione przy pomocy funkcji `field::setPstupid` ustawiam wartości pól statycznych klasy `field` tak, żeby symulowała ona obliczenia w pierścieniu $\mathbb{Z}_n$. Oczywiście nie wszystkie działania zdefiniowane w klasie `field` działają w pierścieniu $\mathbb{Z}_n$(na przykład dzielenie). Jednak możemy to zignorować po prostu ograniczając się do działań dodawania, odejmowania, mnożenia i potęgowania z wykładnikiem naturalnym.

Wyznaczam `d` i `s` zgodnie z oznaczeniami z **Hipotezy 1**.

Następnie tworzę obiekt `x` klasy `field` z wartością pola `value` równą `a`. Podnoszę `x` do potęgi `d` przy pomocy operatora `^=` zdefiniowanego w module definiującym klasę `field`. Następnie `s`-krotnie podnoszę w miejscu `x` do kwadratu przed każdym kolejnym podniesieniem sprawdzając, czy `x==n-1` co oznaczałoby, że liczba `n` jest "raczej pierwsza"i zwracamy wtedy `true` lub `x==1` co oznaczałoby, że `x` po kolejnych podniesieniach do kwadratu będzie równe 1, więc `n` nie jest pierwsza i zwracamy `false`. Jeżeli żaden z tych warunków nigdy nie zaszedł po `s-1` podniesieniach do kwadratu zwracamy `false`.

Kolejną funkcją jest `MillerRabin`, która przyjmuje liczbę `n` typu `__int128` i wywołuje funkcję `MillerRabinOne` z `n` jako pierwszym argumentem oraz drugim będącym kolejnymi liczbami ze zbioru $\{2, 3, ..., \min(n-2, 2\ln^2(n))\}$ i jeżeli, któreś z tych wywołań zwróci `false` funkcja `MillerRabin` również zwraca `false`. Jeżeli jednak tak nie będzie to na podstawie **Faktu 1** zwracamy `true` oznaczające, że `n` z bardzo małym prawdopodobieństwem błędu jest pierwsza.

Kolejną już ostatnią funkcją w tym module jest funkcja `find_prime`. Przyjmuje ona jako argumenty liczbę `n` i `t`, zaś następnie zwraca liczbę pierwszą postaci $r2^k + 1$, gdzie r jest liczbą naturalną z przedziału $[t + 1, (n + t)^3]$. Najpierw przy pomocy wywołania `srand((unsigned int)time(NULL))` ; zwiększam losowość testu. Następnie przy pomocy funkcji `pow2` wyznaczam 2^k. Następnie losuję potencjalne liczby `candidate` przy pomocy wywołania funkcji `random128` na $(n + t)^3$. Jeżeli wylosuję liczbę nie większą niż `t` losuję jeszcze raz. Dla dużych `n` i `n` prawdopodobieństwo tego jest znikome, a ewentualny koszt niewielki. Następnie mnożę wylosowane r z wyznaczonym 2^k i dodaję do wyniku 1. Kolejnym krokiem jest dla tak uzyskanej liczby sprawdzenie, czy jest pierwsza przy pomocy funkcji `MillerRabin`.

Omówione w tej sekcji funkcje znajdują się w pliku `find_prime.cpp` i są napisane w języku C++.

6 Teorio-liczbowa szybka transformata Furiera

Dyskretna transformata Furiera służy przyspieszeniu mnożenia wielomianów. Wielomiany możemy reprezentować jako ciąg współczynników lub jako ciąg wartości w ustalonych punktach , których liczba przekracza stopień reprezentowanego wielomianu. Pierwsza reprezentacja jest wygodna do wyliczania wartości w dowolnym punkcie z dziedziny, jednak mnożenie wielomianów w takiej postaci jest czasochłonne i ma czas kwadratowy od długości wielomianów. Druga reprezentacja pozwala na mnożenie wielomianów w czasie liniowym, jednak wyliczanie ich wartości w innych punktach jest trudne. Dyskretna transformata Fouriera służy do przechodzenia między tymi postaciami w relatywnie szybkim czasie $O(\log(n)n)$, gdzie n to stopień tych wielomianów.

Wybór punktów opiera się na prostej obserwacji, że jeżeli $A(x) = a_0x^0 + a_1x^1 + .. + a_{2n-1}x^{2n-1}$ (jeżeli stopień A ma stopień parzysty przyjmujemy po prostu, że a_{2n-1} jest równe 0). jest równe sumie $A_0(x^2)$ gdzie $A_0(t)$ jest równe $\sum_{i=0}^{n-1} a_{2i}t^i$, oraz $A_1(x^2)x$, gdzie $A_1(t)$ jest równe $\sum_{i=0}^{n-1} a_{2n+1}t^i$. Tak więc jeżeli dwie liczby a i b mają te same wartości swoich kwadratów możemy obliczyć tylko raz wartości A_1 i A_2 od tego kwadratu, a następnie w czasie stałym połączyć te wyniki, tak aby uzyskać wartości wielomianu A w punktach a i b.

Klasyczna wersja dyskretnej transformaty Furiera oblicza wartości wielomianu A dla argumentów z ciała liczb zespolonych. Przyjmujemy następujące oznaczenie $A(x) = \sum_{i=0}^{\infty} a_ix^i$, przy czym istnieje I takie, że $\forall i > I : a_i = 0$. I oznaczając dalej $A_0(x) = a_0x^0 + a_2x^1 + .. + a_{2^m-2}x^{2^{m-1}-1}$ oraz $A_1(x) = a_1x^1 + a_3x^1 + .. + a_{2^m-1}x^{2^{m-1}-1}$, przy czym 2^m jest najmniejszą potęgą 2 o wykładniku naturalnym, większym niż stopień wielomianu będącego wynikiem optymalizowanego przez nas mnożenia.

Będę oznaczał ω_m jako $e^{\frac{i2\pi}{2^m}}$. Zaważmy, że :

$$\begin{cases} A(\omega_m^1) & = A_0(\omega_m^2) + A_1(\omega_m^2)\omega_m^1 = A_0(\omega_{m-1}^1) + A_1(\omega_{m-1}^1)\omega_m^1 \\ A(\omega_m^2) & = A_0(\omega_m^4) + A_1(\omega_m^4)\omega_m^2 = A_0(\omega_{m-1}^2) + A_1(\omega_{m-1}^2)\omega_m^2 \\ A(\omega_m^3) & = A_0(\omega_m^6) + A_1(\omega_m^6)\omega_m^3 = A_0(\omega_{m-1}^3) + A_1(\omega_{m-1}^3)\omega_m^3 \\ & \dots \\ A(\omega_m^{2^m}) & = A_0(\omega_m^{2^{m+1}}) + A_1(\omega_m^{2^{m+1}})\omega_m^{2^m} = A_0(\omega_{m-1}^{2^m}) + A_1(\omega_{m-1}^{2^m})\omega_m^{2^m} \end{cases}$$

Tak więc gdy mamy wektory $V_0 = \left(A_0(\omega_{m-1}^0), A_0(\omega_{m-1}^1), ..., A_0(\omega_{m-1}^{2^{m-1}})\right)$ oraz $V_1 = \left(A_1(\omega_{m-1}^0), A_1(\omega_{m-1}^1), ..., A_1(\omega_{m-1}^{2^{m-1}})\right)$ możemy w czasie liniowym wyliczyć wektor $V = \left(A(\omega_m^0), A(\omega_m^1), ..., A(\omega_m^{2^m})\right)$, jednak wyliczenie wektorów V_1 i V_2 można znów rozbić na wyliczenie najpierw 2 wektorów długości 2^{m-2} i następnie połączenie tych dwóch wektorów w czasie liniowym od ich długości. Możemy tak rozbijać kolejne wektory, aż dojdziemy do wektorów długości 1 które przechowują wartości pewnych wielomianów 0-stopnia będący po prostu jednym ze współczynników A.

Niech $O(t)$ będzie czasem policzenia wartości wielomianu A w $n = 2^m$ punktach przy pomocy dyskretnej transformaty Furiera. Furiera. Wiemy, że $O(t) = 2O(\frac{t}{2}) + O(n)$. Rozwiązaniem takiego równania jest $O(t) = O(n\log(n))$. Okazuje się, że po wyliczeniu wartości wielomianu AB w omawianych punktach możemy "wyciągnąć"z tych wartości w czasie $O(n)$ wartości współczynników wielomianu AB, gdzie n to ilość tych punktów, tak więc cały algorytm mnożenia odbywa się w czasie $O(\log(n)n) + O(n) = O(\log(n)n)$

Moja implementacja korzysta z dwóch klasycznych ulepszeń dyskretnej transformaty Furiera.

Pierwsza to tak zwana "Teorio-liczbowa transformata Furiera". Polega ona na nie wykonywaniu obliczeń w ciele C lecz w Z_p, przy czym oczywiście musi w ciele tym istnieć pierwiastek z 1 stopnia 2^m, gdzie 2^m jest potęgą 2 o wykładniku naturalnym i większa niż stopień zwracanego wielomianu. Pierwiastek ten też podniesiony do jakiejkolwiek potęgi o wykładniku naturalnym dodatnim, mniejszym niż 2^m musi być różny od 1. Tak naprawdę oznacza to po prostu, że p ma postać $r2^m + 1$, gdzie r jest liczbą naturalną, o czym pisałem w poprzedniej sekcji.

Jako $A[i : j]$ będę oznaczał komórki tablicy ze współczynnikami wielomianu A o indeksach większych lub równych niż i oraz mniejszych lub równych j.

Drugim ulepszeniem jest użycie tak zwanej szybkiej transformaty Furiera. Opiera się ona na obserwacji że rekurencyjna wersja dyskretnej transformaty Furiera polega najpierw na podzieleniu współczynników wielomianu na coraz mniejsze grupy, aż do zawierających tylko pojedynczy współczynnik, a następnie łączeniu pojedynczych współczynników w pary zawierające informację o 2 współczynnikach, potem czwórki, ósemki itd aż do połączenia ich w jeden duży blok zawierający informacje o wszystkich współczynnikach, a rekurencyjne wywołania służą jedynie pogrupowaniu w jakiej kolejności będziemy wykonywać łączenia.(NIE WIEM JAK TO NAPISAC LEPIEJ????)

Możemy jednak to grupowanie wykonać bez kolejnych rekurencyjnych wywoływań transformaty po prostu ustawiając obok siebie współczynniki w takiej kolejności żeby współczynniki wielomianów stopnia $2^l - 1$ które w wersji rekurencyjnej powstają na $m-l$ poziomie rekursji były w spójnych segmentach $A[0 : 2^l - 1]$,$A[2^l : 2 \cdot 2^l - 1]$, $A[2 \cdot 2^l : 2(2^l) - 1]$,...,$[2^m - 2^l, 2^m - 1]$. Przy czym współczynniki wielomianu powstałego na $l+1$ poziomie rekursji, które w wielomianie powstałym na l-tym poziomie rekursji były przy potęgach o wykładniku parzystym zostają umieszczone w "pierwszej połowie"segmentu długości 2^l w którym były umieszczone na poprzednim etapie, zaś te, które były przy potęgach nieparzystych w drugim.

Zaważmy, że pierwsze "rozbicie"wektora współczynników w wersji rekurencyjnej sprowadza się, do wybrania osobno współczynników parzystych i nieparzystych. Tak więc nasze grupowanie powinno w polach o indeksach $(0, 1, 2, ..., 2^{m-1} - 1)$ umieścić liczby znajdujące się najpierw w polach o indeksach parzystych, czyli mające ostatni bit równy `0`, z kolei w polach o indeksach $(2^{m-1}, 2^{m-1} + 1, ..., 2^m - 1)$ powinno umieścić liczby nieparzyste, czyli mające ostatni bit równy `1`.

Przyjmijmy, że pierwszy poziom rekursji rozbija wielomian $A(x) = \sum_{i=0}^{2^m-1} a_i x^i$, na $A_0(x) = \sum_{i=0}^{2^{m-1}-1} a_{2i} x^i$ oraz $A_1(x) = \sum_{i=0}^{2^{m-1}-1} a_{2i+1} x^i$.

Gdy umieścimy współczynniki A_0 w $A[0, 2^{m-1} - 1]$, zaś A_1 w $A[2^{m-1}, 2^m - 1]$ należy umieścić współczynniki przy potęgach parzystych w wielomianie A_0 w pierwszej połówce $A[0, 2^{m-1} - 1]$, nie-

parzyste zaś w drugiej. I analogicznie współczynniki przy potęgach parzystych w wielomianie A_1 w pierwszej połówce $A[2^{m-1}, 2^m - 1]$, nieparzyste zaś w drugiej. To który współczynnik jest przy potędze parzystej czy nie w wielomianach A_0 i A_1 decyduje to czy początkowo był przy potędze której wykładnik miał drugi bit równy 0, czy 1. Dalsze kontynuowanie tego rozumowania prowadzi do wniosku, że jeżeli odwrócimy $m-1$ ostatnich bitów liczby i otrzymamy numer komórki w której powinniśmy umieścić liczbę a_i. Istotnie załóżmy, że liczba powstała po odwróceniu m ostatnich bitów liczby i jest większa niż że liczba powstała po odwróceniu m ostatnich bitów liczby j. Prosta indukcja pozwala stwierdzić, że współczynniki, których indeksy mają k wspólnych ostatnich bitów zostają na k-tym poziomie rekursji umieszczone w tym samym wielomianie. To czy zostaną umieszczone w pierwszej czy drugiej połowie segmentu do którego były przypisane na k-tym poziomie rekursji determinuje to czy ich $k+1$ bit jest równy 0 czy 1.

To, że liczba powstała po odwróceniu m ostatnich bitów liczby i jest większa niż liczba powstała po odwróceniu m ostatnich bitów liczby j oznacza, że pewna liczba k(być może równa 0)ostatnich bitów liczby i jest taka sama jak liczby j zaś $k+1$ ostatni bit liczby i jest równy 1, podczas gdy $k+1$ ostatni bit liczby j jest równy 0. Tak więc do $k-tego$ poziomu rekursji a_i i a_j pozostają w tym samym wielomianie $B(x) = \sum_{l=0}^{2^{m-k}-1} b_l x^l$, czyli w wersji iteracyjnej są w tym samym segmencie długości 2^{m-k} i na $k+1$ poziomie rekursji a_j jest umieszczane w wielomianie ze współczynnikami $b_0, b_2, ..., b_{2^{m-k}-2}$, zaś a_i w wielomianie ze współczynnikami $b_1, b_3, ..., b_{2^{m-k}-1}$, tak więc w wersji iteracyjnej a_j zostaje w umieszczone w pierwszej połowie segmentu w którym są współczynniki $b_0, b_1, ..., b_{2^{m-k}-1}$ zaś a_i w drugiej, tak więc a_i zostaje umieszczone w komórce o większym indeksie niż a_j. Ponieważ indeksy tablicy $A[0 : 2^m - 1]$ wyznaczają wszystkie możliwe binarne liczby m-bitowe to istotnie jeżeli odwrócimy $m-1$ ostatnich bitów liczby i otrzymamy numer komórki w której powinniśmy umieścić liczbę a_i.

Przejdźmy do dokładnego opisu naszej implementacji.

Najpierw definiuję funkcję `enoughGoodRoot` która przyjmuje liczbę `k` typu `__int128` i zwraca element typu `field`, który jest pierwiastkiem z 1 stopnia 2^k. Jest to po prostu podniesienie elementu `field::almostPrimitiveRoot` (będącego pierwiastkiem z 1 stopnia $2^{field::degreeOfDegree}$) `field::almostPrimitiveRoot-k` razy do kwadratu.

Kolejnym etapem jest odpowiednie ustawienie wartości w komórkach wektora ze współczynnikami wielomianu(nazwę go `coefficients`, tak by można było wykonać na niej iteracyjną wersję szybkiej transformaty Furiera.

Funkcja `reverseBits` służy znalezienia indeksu komórki, w której powinna być umieszczona liczba znajdująca się początkowo w komórce o indeksie `x` gdzie `x` jest pierwszym argumentem funkcji `reverseBits` typu `long long`, zaś drugim argumentem jest liczba `k` typu `long long` wyznaczająca ile ostatnich bitów `x` należy odwrócić.

Na początku zamieniamy miejscami bity o indeksach parzystych `x` z nieparzystymi. Robimy to najpierw tworząc dwie kopie zmiennej `x`. W jednej z nich zerujemy (and-ując ją z odpowiednią maską) bity o indeksach parzystych, zaś w drugiej nieparzystych. Pierwszą kopię przesuwamy o 1 bit w prawo, tak, że bity o indeksach parzystych stały się bitami o indeksach nieparzystych, zaś drugą w lewo i wykonujemy na tak zmodyfikowanych kopiach operację `or`-a bitowego.

Analogiczną operację jak omówiona w poprzednim akapicie dla bloków składających się z 1 liczby wykonujemy na blokach 2-bitowych, 4-bitowych, 8-bitowych, 16-bitowych, i 32-bitowych.

Tak zmodyfikowana zmienna `x` jest odwróconą bitowo początkową wartością zmiennej `x`. Ponieważ chcieliśmy odwrócić jedynie jej ostatnie `k` bitów przed zwróceniem `x` jako wyniku przesuwamy ją jeszcze o `64-k` bitów w prawo.

Gdy mamy już funkcję `reverseBits` możemy stworzyć funkcję `setToDo` transformującą wektor współczynników w "surowej"formie w wektor gotowy do wyliczenia jego szybkiej transformaty Furiera. Jako argumenty dla tej funkcji otrzymujemy wektor elementów typu `field` o nazwie `coefficients` oraz zmienną `long long` o nazwie `size` oznaczającą żądaną długość wektora wynikowego(jest to odpowiednio duża potęga dwójki o wykładniku naturalnym). Na początku dopełniamy wektor `coefficients` elementami równymi `field(0)` do rozmiaru `size`. W zmiennej `log` zapisujemy wartość logarytmu o podstawie 2 na argumencie będącym długością wektora `coefficients` Następnie w pętli z iteratorem `i` przechodzącym po wszystkich liczbach naturalnych od `0` do `size-1` i jeżeli `reverseBits(i,log) > i`(warunek ten służy temu, by każdy element został zamieniony dokładnie raz, zauważmy też, że `reverseBits(reverseBits(i,log),log)=i`) zamieniam w tabeli pole o indeksie `i` z polem o indeksie `reverseBits(i,log)` miejscami. Funkcja nic nie zwraca, a jedynie modyfikuje wektor podany jako argument.

Przejdźmy do funkcji `DFT`, która otrzymując wektor współczynników wielomianu `coefficients`, liczbę `size` nie mniejszą niż długość tego wektora i będący potęgą 2 o wykładniku naturalnym oraz obiekt `omegaM` typu `field`, który jest pierwiastkiem z 1 stopnia `size` zwraca wektor zawierający wartości tego wielomianu w punktach kolejno $omegaM^0$, $omegaM^1$,...,$omegaM^{size}$.

Poniżej umieściłem kod tej procedury.

```
void inline DFT(std::vector<field>&coefficients, long long size, field omegaM){
    setToDo(coefficients,size);
    std::stack<field> omegasM;
    while(omegaM != 1){
        omegasM.push(omegaM);
        omegaM *= omegaM;
    }
    long long m = 1;
    while(!omegasM.empty()){
        field currentOmegaM= omegasM.top();
        std::cout<<std::endl;
        omegasM.pop();
        field omega = 1;
        m*=2;
        std::cout<<m;
        for(long long j = 0; j<m/2;j+=1){
            for(long long k = j; k<size; k+=m){
                field t = omega * coefficients[k+m/2];
                field u = coefficients[k];
                coefficients[k] = u+t;
                coefficients[k+m/2] =  u - t;
            }
            omega *= currentOmegaM;
        }
    }
}
```

Na początku przygotowujemy wektor do wykonania dalszej części obliczeń wywołując zdefiniowaną w poprzednim paragrafie funkcję `setToDo` na wektorze `coefficients` i dla liczby `size`.

Tworzymy stos na który kładziemy wyniki kolejnych złożeń podniesienia do kwadratu liczby `omegaM` aż do `-1`.

Póki stos nie będzie pusty będę w każdej iteracji pętli `while` ściągał z niego kolejne wartości i przypisywał je do zmiennej `currentOmegaM`.

W dalszej części będę oznaczał jako ω_i pierwiastek z 1 stopnia 2^i, w Z_p.

Przyjmijmy oznaczenie, że na początku `i`-tej iteracji wektor `coefficients` ma postać $\frac{size}{2^{i-1}}$ bloków postaci $(f_{i,k}(\omega_i^0), f_{i,k}(\omega_i^1), ..., f_{i,k}(\omega_i^{2^{i-1}-1}))$. gdzie k to numer bloku. Chcemy następujące po sobie pary bloków połączyć w jeden blok postaci $(f_{i+1,k}(\omega_{i+1}^0), f_{i+1,k}(\omega_{i+1}^1), ..., f_{i+1,k}(\omega_{i+1}^{2^i-1}))$, przy czym $f_{i+1,k}(\omega_{i+1}^j) = f_{i,2k}(\omega_i^{\frac{j}{2}}) + f_{i,2k+1}(\omega_i^{\frac{j}{2}})\omega_{i+1}^j$.

Kolejne iteracje pętli z iteratorem `j` odpowiadają kolejnym punktom w których wyliczamy wartości odpowiednich wielomianów, zaś pętla z iteratorem `k` numery kolejnych rozpatrywanych wielomianów. Ponieważ $\omega_{i+1}^{2^i} = -1$ wyliczenie $f_{i+1,k}(\omega_{i+1}^j)$ i $f_{i+1,k}(\omega_{i+1}^{j+2^i})$ wyliczam w jednej iteracji pętli.(jak lepiej to napisać?)

Przejdźmy do pełnej procedury mnożenia.

```
std::vector<field>multiplication(std::vector<field> A, std::vector<field> B){
    long long size = 1;
    while(size < (long long)((A.size() + B.size()) +2 )){
        size *= 2;
    }
    long long l = log(size);
    field omegaM = enoughGoodRoot(l);
```

```
    DFT(A, size ,omegaM);
    DFT(B, size ,omegaM);
    for(long long i=0;i<size;i++){
        A[i] = B[i] * A[i];
    }
    omegaM = field(1)/omegaM;
    DFT(A, size ,omegaM);
    for(long long i=0;i<size;i++){
        A[i] /= field(size);
    }
    return A;
}
```

Wektorów nie przekazuję przez referencje bo będę je modyfikował.

Najpierw wliczamy wartość `size` będącą liczbą punktów w których będziemy liczyć wartości wielomianu AB(jest to najmniejsza liczba będąca potęgą 2 o wykładniku naturalnym większa niż stopień AB). Kolejnym etapem jest znalezienie pierwiastka stopnia `size` z 1 w $\mathbb{Z}_p$, nie będącym w tym ciele jednocześnie pierwiastkiem z 1 niższego naturalnego stopnia. Następnie wyliczamy przez wywołania DFT na odpowiednich argumentach wektor $(A(omegaM^0), A(omegaM^1), ..., A(omegaM^{size}))$ oraz $(B(omegaM^0), B(omegaM^1), ..., B(omegaM^{size}))$, następnie w kopii wektora A zapisuję wartości AB w kolejnych punktach będące wartościami odpowiednich mnożeń.

Okazuje się, że aby zmienić, wektor $(AB(omegaM^0), AB(omegaM^1), ..., AB(omegaM^{size}))$ w wektor kolejnych współczynników AB wystarczy wywołać na nim DFT z drugim argumentem równym $size$ i trzecim będącym równym odwrotności $omegaM$ w Z_p, a następnie podzielić w Z_p każdy jego element przez $size$.

7 Pochodna algebraiczna i jej własności

Klasyczna analityczna definicja pochodnej jest to przekształcenie funkcji $f(x)$ w funkcję $f'(x)$ taką, że $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$. Definicja taka traci jednak cały sens jeżeli chcemy zmienić dziedzinę funkcji f i f' z liczb rzeczywistych na dziedzinę gdzie nie możemy zmniejszać h w taki sposób by było dowolnie małe lecz niezerowe. Przykładem takiej dziedziny jest ciało Z_p.

Zauważmy, że jeżeli funkcja f $\mathbb{R} \to \mathbb{R}$ jest gładka w okolicy 0, możemy ją utożsamić z jej szeregiem Taylora $\sum_{i=0}^{\infty} f_i x^i$.

Weźmy liczbę pierwszą p. W dalszej części sekcji będę zakładał, że współczynniki szeregu Taylora rozważanej funkcji są postaci $\frac{a_i}{b_i}$, gdzie $a, b \in \mathbb{Z}$ i b jest niepodzielne przez p. Przy takim założeniu napis $\sum_{i=0}^{\infty} \frac{a_i}{b_i} x^i$ zachowuje algebraiczny sens również w ciele reszt Z_p, w którym liczbę całkowitą utożsamiamy z jej resztą z dzielenia przez p. W języku szeregów Taylora pochodną można zdefiniować jako przekształcenie funkcji $f(x) = \sum_{i=0}^{\infty} f_i x^i$ w funkcję $f'(x) = \sum_{i=0}^{\infty} (i+1) f_{i+1} x^i$.

Udowodnię teraz, że tak zdefiniowana pochodna, dla funkcji $f(x) = \sum_{i=0}^{\infty} f_i x^i$ i $g(x) = \sum_{i=0}^{\infty} g_i x^i$ zachowuje własności $f' + g' = (f+g)'$, $f' - g' = (f-g)'$, $(fg)' = f'g + fg'$ oraz $f(g)' = f'(g)g'$.

Lemat: $f' + g' = (f+g)'$ oraz $f' - g' = (f-g)'$
Dowód: $f' + g' = \sum_1^{\infty} i f_i x^{i-1} + \sum_1^{\infty} i g_i x^{i-1} = \sum_1^{\infty} i(f_i + g_i) x^{i-1} = (f+g)'$ i analogicznie $f' - g' = \sum_1^{\infty} i f_i x^{i-1} - \sum_1^{\infty} i g_i x^{i-1} = \sum_1^{\infty} (f_i - g_i) x^{i-1} = (f-g)'$

Lemat: $(fg)' = f'g + fg'$.
Dowód: $(fg)' = ((\sum_{i=0}^{\infty} f_i x^i)(\sum_{i=0}^{\infty} g_i x^i))' = (\sum_{i=0}^{\infty} (\sum_{j=0}^{i} f_j g_{i-j}) x^i)' = \sum_{i=1}^{\infty} (\sum_{j=0}^{i} f_j g_{i-j}) i x^{i-1} = \sum_{i=0}^{\infty} (\sum_{j=0}^{i} f_j g_{i+1-j})(i+1) x^i$ z kolei
$f'g + fg' = (\sum_0^{\infty} (i+1) f_{i+1} x^{i-1})(\sum_{i=0}^{\infty} g_i x^i) + (\sum_0^{\infty} (i+1) g_{i+1} x^i)(\sum_{i=0}^{\infty} f_i x^i) = \sum_{i=0}^{\infty} (\sum_{j=0}^{i} f_{i+1-j} g_j (i+1-j)) x^i + \sum_{i=0}^{\infty} (\sum_{j=0}^{i} g_{i+1-j} f_j (i+1-j)) x^i = \sum_{i=0}^{\infty} (\sum_{j=0}^{i} g_{i+1-j} f_j (i+1-j+j)) x^i = \sum_{i=0}^{\infty} (\sum_{j=0}^{i} g_{i+1-j} f_j)(i+1) x^i = (fg)'$

Lemat: $(f(g))' = f'(g)g'$.
Dowód: Na początek przyjmijmy, że $f(x) = x^k$, gdzie k jest liczbą naturalną. Dla $k = 0$ i $k = 1$ teza jest spełniona. Niech teza jest spełniona dla $f(x) = x^l$ dla każdego naturalnego l mniejszego niż k. Wtedy $(g^k)' = ((g^{k-1})g)' = (g^{k-1})g' + g(g^{k-1})' = (g^{k-1})g' + g((k-1)g^{k-2}g') = (g^{k-1})g' + ((k-1)g^{k-1}g') = kg^{k-1}g'$, tak więc na mocy zasady indukcji dla dowolnego k naturalnego jeżeli $f(x) = x^k$ to $(f(g))' = f'(g)g'$ dla dowolnego g postaci $\sum_1^\infty g_i x^i$. W takim razie dla dowolnego f postaci $\sum_1^\infty g_i x^i$ zachodzi $(f(g))' = (\sum_{i=0}^\infty f_i g^i)' = \sum_{i=0}^\infty f_i (g^i)' = \sum_{i=0}^\infty f_i i g^{i-1} g' = f'(g)g'$.

8 Implementacja algorytmu Jin i Wu

W tej części pracy dla funkcji $F(x)$, której szereg Taylora ma postać $\sum_{i=0}^\infty f_i x^i$ jako F_t będziemy oznaczać $\sum_{i=0}^t f_i x^i$.

Zasadnicza implementacja algorytmu Jin i Wu składa się z 4 funkcji.

Pierwszą z nich jaką omówię jest funkcja `B`, której zadaniem jest wyliczenie pierwszych $t+1$ współczynników szeregu Taylora $B(x) = \ln(\prod_{i=1}^n (1 + x^{s_i}))$.

Funkcja `B` wyznacza wektor $t + 1$ pierwszych wyrazów szeregu Taylora w ciele Z_p następującej funkcji:

$$B(x) = \ln(\prod_{i=1}^n (1 + x^{s_i})) = \sum_{i=1}^n \ln(1 + x^{s_i}) = \sum_{i=1}^n (\sum_{j=1}^\infty \frac{(-1)^{j-1}}{j} x^{s_i j})$$

Niech a_k będzie oznaczać liczbę elementów S równych k.

Przy tak przyjętych oznaczeniach

$$B_t(x) = \sum_{i=1}^n (\sum_{j=1}^{\lfloor \frac{t}{s_i} \rfloor} \frac{(-1)^{j-1}}{j} x^{s_i j}) = \sum_{k=1}^t (\sum_{j=1}^{\lfloor \frac{t}{k} \rfloor} \frac{a_k (-1)^{j-1}}{j} x^{jk})$$

Spójrzmy na implementację funkcji B.

```
std::vector<field> B(std::vector<field> s, long long t){
    std::vector<field> a(t+1,field(0));
    std::vector<field> ans(t+1,field(0));
    int K;
    for(int i =0; i<s.size(); i++){
        if(s[i].getValue()<=t)a[s[i].getValue()] ++;
    }
    for(long long k = 1; k <= t; k++){
        for(long long j = 1; j <= t/k ; j++){
            field x = field(-1);
            ans[k*j] = ans[k*j] + a[k]*(x^(j-1))/field(j);
        }
    }
    return ans;
}
```

Na początku tworzę wektor `a` którego `k`-ty element oznacza zdefiniowane wcześniej a_k. Reszta kodu jest prostym przetłumaczeniem matematycznej notacji sumy na składnię zawierającą pętle, przy czym pole $ans[i]$ odpowiada współczynnikowi przy x^i w rozwinięciu w szereg Taylora funkcji $B(x)$.

Kolejne dwie funkcje to `compute` i `mainCompute`.

Ich celem jest wyliczenie $(\exp(B(x)))_t = (\sum_{i=0}^\infty \frac{B(x)^i}{i!})_t$

Algorytm ten bazuje na spostrzeżeniu, że żeby wyliczyć $G_t(x)$,gdzie $G(x) = \exp(F(x))$ i $F(x) = \sum_{i=1}^\infty f_i x^i$, jeżeli znamy $G(0)$ wystarczy znaleźć wartości $f_0, f_1, ..., f_t$. Co więcej $G_t(F(x)) = G_t(F_t(x))$

Niech rozwinięcie w szereg Taylora funkcji G ma postać $\sum_{i=0}^\infty g_i x^i$

Zauważmy, że $G'(x) = (\exp(F(x)))' = \exp(F(x))F'(x) = G(x)F'(x)$, tak więc $\sum_{i=0}^\infty (i+1) g_{i+1} x^i = (\sum_{i=0}^\infty g_i x^i)(\sum_{i=1}^\infty i f_i x^{i-1}) = \sum_{i=0}^\infty (\sum_{j=1}^{i+1} j f_j g_{i+1-j}) x^i$

Tak więc $(i+1)g_{i+1} = \sum_{j=1}^{i+1} jf_j g_{i+1-j}$, czyli $g_{i+1} = (i+1)^{-1}(\sum_{j=1}^{i+1} jf_j g_{i+1-j})$. Zauważmy, że $G(0) = g_0$, tak więc mając wyliczone $f_0, f_1, ..., f_t$ jesteśmy w stanie wyliczyć $g_1, g_2, ..., g_t$, wyliczając je po kolei.

Weźmy liczbę pierwszą $p > t$.

$B(x) = \ln(\prod_{i=0}^{n}(1+x^{s_i})) = \sum_{i=1}^{n}(\sum_{k=1}^{\infty} \frac{(-1)^{k-1}x^{s_i k}}{k}) := \sum_{i=0}^{\infty} b_i x^i$, więc każda z liczb $b_0, b_1, ..., b_n$ da się zapisać jako liczba wymierna postaci $\frac{x}{y}$, gdzie x, y to względnie pierwsze liczby naturalne i y nie jest podzielne przez p. Jeżeli x i y utożsamimy z resztą dzielenia ich przez p to wartość wyrażenia $\frac{x}{y}$ da się wyliczyć w ciele Z_p, a z tego.

Niech $A(x) = \exp(B(x)) = \prod_{i=0}^{n}(1+x^{s_i}) := \sum_{i=0}^{\infty} a_i x^i$.

Wiemy, że $a_{i+1} = (i+1)^{-1}(\sum_{j=1}^{i+1} b_j a_{i+1-j})$, a także $a_0 = \prod_{i=0}^{n}(1+0^{s_i}) = 1$, więc każda z liczb a_i, gdzie $i \in \{0, 1, 2, .., t\}$ da się przedstawić w postaci $\frac{X_i}{Y_i}$, gdzie X_i, Y_i to względnie pierwsze liczby naturalne i Y_i nie jest podzielne przez p. Jeżeli X_i i Y_i utożsamimy z resztą dzielenia ich przez p to wartość wyrażenia $\frac{X_i}{Y_i}$ da się wyliczyć w ciele Z_p i wartość liczby a_i dla naturalnego i nie większego niż t jest równa 0 wtedy i tylko wtedy gdy dla jej postaci $\frac{X_i}{Y_i}$ p dzieli X_i. Zauważmy, że $A(x)$ jest iloczynem wielomianów o całkowitych współczynnikach, więc też jest wielomianem o całkowitych współczynnikach, więc a_i jest wartością całkowitą, która w ciele Z_p jest utożsamiona z 0 wtedy i tylko wtedy, gdy jest podzielna przez p.

Na mocy powyższych rozważań, możemy dalsze obliczenia wykonywać w ciele Z_p, chyba, że explicite zaznaczę inaczej.

Procedura `mainCompute` jako argument przyjmuje wektor `f` $t+1$ pierwszych współczynników rozwinięcia w szereg Taylora funkcji B. Na początku inicjuje wynikowy $t+1$-elementowy wektor `g` ustawiając wszystkie jego komórki poza `g[0]` na 0, z kolei `g[0]` ustawiamy na 1. Następnie uruchamiamy funkcję `compute` na argumentach `0`, `t`, `g` i `f`.

Funkcja `compute` zapisuje do wektora podanego jako trzeci argument(oczywiście przez referencję, żeby można było go modyfikować) wartości kolejnych współczynników szeregu Taylora funkcji $g(x) = \exp(\sum_{i=0}^{t} f_i x^i)$, gdzie $t+1$ to długość wektora podanego jako czwarty argument, zaś f_i to wartość i-tej komórki wektora podanego jako czwarty argument. W dalszej części jako f_i będę oznaczał wartość i-tej komórki wektora podanego jako czwarty argument, zaś jako g_i będę i-tej komórki wektora podanego jako trzeci argument. Jako t będę oznaczał `f.size()-1`

Idea funkcji `compute` bazuje na tym, że aby wyliczyć $g_i = (i+1)^{-1}(\sum_{j=1}^{i+1} jf_j g_{i+1-j})$ dla wszystkich $i \in \{1, 2, ..., t\}$ należy wykonać $O(n^2)$ dodawań składników postaci $(i+1)^{-1}jf_j g_{i+1-j}$ do odpowiednich komórek. Nie każde dodawanie można wykonać w dowolnym momencie, ponieważ wartości komórek wektora `g` ulegają zmianie. Dodawanie uznamy za dozwolone, jeżeli g_{i+1-j} obecne w dodawanym składniku $(i+1)^{-1}f_j g_{i+1-j}$ nie ulega zmianie.

Zapisana w pseudokodzie funkcja `compute` ma postać:

```
procedure compute(l,r,g,f)
    if l < r
        m <- floor((l+r)/2) #obliczenia w tej linii wykonujemy w liczbach naturalnych
        compute(l,m,g,f)
        for i <- m+1, m+2,...,r
            for j<-l,l+1,..,m
                g[i] <- g[i]+(i-j)f[i-j]g[j]/i
            end for
        end for
        compute(m+1,r,g,f)
    end if
end procedure
```

Po wykonaniu `compute(l,r,f,g)` chcemy, żeby wartości $g_l, g_{l+1}, ..., g_r$ były już ustawione na wartości docelowe, zaś przed wykonaniem `compute(l,r,f,g)` chcemy, żeby wszystkie składniki postaci $(i+1)^{-1}f_j g_{i+1-j}$ gdzie $i+1-j < l$ zostały już dodane do odpowiednich komórek `g` o indeksach należących do $\{0, 1, ..., r\}$. Chcemy też, żeby każda operacja dodawania była dozwolona.

Jeżeli długość `g` jest równa 0 to żądania napisane w poprzednim akapicie są spełnione. Załóżmy indukcyjnie, że są spełnione dla `g` długości $0, 1, 2, .., t-1$. Niech długość `g` jest równa t.

Wywołanie `compute(0,t,f,g)` sprowadza się do wywołania `compute(0,m,f,g)`, gdzie `m` to wynik wykonanej w liczbach całkowitych działania $\lfloor \frac{t}{2} \rfloor$. Po jej wykonaniu z tezy indukcyjnej $g_0, g_2, ..., g_m$

mają docelowe wartości. Następnie przechodzimy do wykonania pętli

```
for i <- m+1, m+2,...,r
    for j<-l,l+1,..,m
        g[i] <- g[i]+(i-j)f[i-j]g[j]/i
    end for
end for
```

Ponieważ indeks j jest nie większy niż m wszystkie dodawania są dozwolone. Po wykonaniu tej pętli zostają już wykonane wszystkie potrzebne dodania wyrazów postaci $g[i] + (i-j)f[i-j]g[j]/i$, gdzie j jest nie większe niż m. Następnie wykonujemy $compute(m+1, r, g, f)$, co przypomina wywołanie $compute(0, r-m-1, f, g)$ z tą modyfikacją, że w momencie gdy wykonalibyśmy linię `g[i] <- g[i]+(i-j)f[i-j]g[j]/i` każde wystąpienie zmiennych `i` i `j` zastępujemy odpowiednio zmiennymi `i+m+1` i `j+m+1`. Ponieważ zgodnie z tezą indukcyjną po wykonaniu $compute(0, r-m-1, f, g)$ g_i zostaje zwiększone o $(i+1)^{-1}(\sum_{j=1}^{i+1} jf_j g_{i+1-j})$, dla $i \in \{1, 2, ..., r-m-1\}$ to po wykonaniu `compute(m+1,r,f,g)` g_{i+m+1} zostaje zwiększone o $(m+1+i+1)^{-1}(\sum_{j=1}^{i+1} jf_{m+1+j} g_{m+1+i+1-j})$, dla $i \in \{1, 2, ..., r-m-1\}$, więc po wykonaniu $compute(m+1, r, g, f)$ $g_i = (i+1)^{-1}(\sum_{j=1}^{i+1} jf_j g_{i+1-j})$ dla każdego $i \in \{0, 1, 2, 3, ..., t\}$.

Nasza implementacja jednak korzysta z jednego usprawnienia. Zauważmy, że iloczyn wielomianów $F(x) = \sum_{k=0}^{r-l} kf_k x^k$ i $G(x) = \sum_{j=0}^{m-l} g_{j+l} x^j$, ma postać $\sum_{i=0}^{r+m-2l} (\sum_{k=0}^{r-l} kf_k g_{i-k+l}) x^i$, tak więc w tym iloczynie współczynnik przy potędze x^{i-l} podzielony przez i jest równy liczbie o którą zostaje zwiększone g_i po wykonaniu pętli

```
for i <- m+1, m+2,...,r
            for j<-l,l+1,..,m
                g[i] <- g[i]+(i-j)f[i-j]g[j]/i
            end for
end for
```

Wykonanie pętli naiwnie zajmuje czas $O(t^2)$, zaś wykorzystanie szybkiej teorio-liczbowej transformaty Fouriera do mnożenia wielomianów, w mojej implementacji przyspieszyć wykonanie tej pętli do $O(t \ln(t))$. Niech $T(t)$ oznacza czas wykonania `compute`, gdzie różnica między drugim i pierwszym argumentem wynosi $t+1$. Ponieważ $T(t) = 2T(\frac{t}{2}) + O(t \ln(t))$, to $T(t) = O(t \ln^2(t))$.

Kolejną ostatnią już implementowaną prze zemnie funkcją jest `JinWu`. Przyjmuje ona wektor `s` elementów zbioru S, oraz liczbę t. Najpierw losuje liczbę pierwszą p która jest wynikiem wykonania funkcji `find_prime(s.size(),t)`. Następnie wywołując `field::setP(p)` ustawiam zmienne statyczne klasy `field` tak, żeby obliczenia w niej wykonywane odpowiadały wykonaniu ich w klasie Z_p. Potem do wektora $Bans$ przy pomocy wykonania funkcji $B(s,t)$ zapisuję $t+1$ pierwszych współczynników(w Z_p) szeregu Taylora funkcji $B(x) = \ln(\prod_{i=0}^{n-1}(1+x^{s_i}))$, gdzie jako n oznaczam długość wektora `s`, zaś jako s_i oznaczam jego i-tą komórkę w Z_p(będę to oznaczenie stosował również w dalszej części pracy). Następnie do wektora `computeAns` zapisuję $t+1$ pierwszych współczynników(w Z_p) szeregu Taylora funkcji $A(x) = \prod_{i=0}^{n-1}(1+x^{s_i})$ jako wynik procedury `mainCompute(t,Bans)`. Zauważmy, że ponieważ $A(x)$ jest wielomianem współczynnik przy potędze x^i w jego rozwinięciu w szereg Taylora jest po prostu współczynnikiem przy potędze x^i w wielomianie $A(x)$, zaś współczynnik przy potędze x^i w $A(x)$ oznacza ilość sposobów wybrania ciągu indeksów naturalnych $-1 < i_1 < i_2 < ... < i_m < n$, takich, że $x^{s_{i_1}+s_{i_2}+...+s_{i_m}} = x^i$, czyli ilość podzbiorów S dających sumę i. Ponieważ liczby te zapisujemy jako elementy Z_p możliwe, `computeAns[t]` jest równe 0 mimo, że istnieją podzbiory S o sumie t jednak ich liczba jest podzielna przez p. Jednak liczba tych podzbiorów na pewno jest nie większa niż 2^n, więc ma co najwyżej n różnych czynników pierwszych. Eksperymenty w sekcji poświęconej losowaniu liczby pierwszej zostało wykazane, że liczba możliwych do wylosowania wartości p jest $O(\frac{(n+t)^3}{\log(n+t)}) = O((n+t)^2)$, więc prawdopodobieństwo, że p dzieli niezerową liczbę podzbiorów S o sumie t jest $O(\frac{n}{(n+t)^2}) = O(\frac{1}{n+t})$.