Algorytm znajdowania sumy podzbioru

You

9 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 nieniejszej 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 sprawdzania czy z $n-$elementowego muiltizbioru 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 podzbiorów.

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 wybrania ze zbirou przedmiotów, w którym 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 wprostproporcjonalne 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 podbiorów z małym t jest algorytm Konstantinos Koiliaris i Chao Xu.

Algorytm który prezentuję w nieniejszej 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 Z_p reszt z dzielenia przez liczbę pierwszą p, moduł losujący liczbę p kożystający z testu Millera-Rabina, moduł zawierający implementację teorio-liczbowej szybkiej transformaty Fouriera(jest to pewna nieortodoksja 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 kożysta 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 nieniejszej pracy zaprezentuję niedeterministyczny algorytm opracowany przez Ce Jin i Hongxun Wu, który kożystając 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 $SubsetSum$, który będziemy rozwiązywać jest następująca:

Wejście: Liczba naturalna n , liczba naturalna t, zbiór S reprezentowany jako ciąg n liczb naturalnych(po wczytaniu przechowywanych w wektoerze `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: Wektor $t+1$-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 algorytym 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 1 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 elemetów wynosi m(oczywiście dla $k \in \{0,1,...,t\}$).

- Dla $i=0$ teza jest oczywista.
- Załóżmy, że teza jest prawdziwa dla $i-1$. 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 $DP[k]=ture$ 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.

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

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

, więc ciąg ten odpowiada indeksom elementów które należy wybrać by uzyskaźć 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 $\ln_t(1+x^a) = \sum_{i=0}^{t} \frac{(-1)^{i-1}x^{ai}}{n}$

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 skożystali 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 pewnealgebraiczne 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 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 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 $O(t)$ współczynnikach, co można wykonać szybką transformatą Furiera w czasie $O(\log(t)t)$. Aby uniknąć problemów z utratą dokładności w obliczeniach na liczbach zmiennoprzecinkowych zastosowałem Teorio-liczbową szybką transformatę Furiera, 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 porządanych 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 elmentach ciała Z_p w którym istnieją działania arytmetycznych, aby uniknąć konieczności ciągłych operacji pobierania explicite wartości modulo p z wyników działań najwygodniejszym podejściem byłoby zaprogramowanie klasy `field` której instancje reprezentują elementy ciała Z_p zaś przeciążone operatory odpowiadają działaniom w Z_p i pisząc bardziej wysokopoziomowe funkcje nie musimy już się troszczyć o to, by jakkolwiek "nromalizować"wyniki.

Klasa ma kilka pól statycznych. Jednym z nich jest liczbna `p __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 r.

Dla każdej liczby pierwszej p istnieje pierwiastek pierwotny, czyli taka liczba, że r, że nie istnieje dodatnia liczba naturalna w mniejsza niż $p-1$ taka, że $r^w \equiv 1(\mod p)$ (oczywiście z małego twierdzenia Fermata $r^{p-1} \equiv 1(\mod p)$). W dalszej części pracy stopniem liczby 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` przechowóję 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 ominięcie 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 zazłożeniu 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 stopnia ale jedynie liczbę r dla której 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 ustawia 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`, `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 predekrementacj. Operator postinkrementacji kożystając z niezmiennika, mówiącego, że wartość pola `value` jest zawsze liczbą całkowitą z przedziału $[0, 1, .., p-1]$ który zachowują wszystkie funkcje modyfikujące pole `value` oraz konstruktory 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 operatora `-`.

Kolejnymi nieco prostszymi operatorami jakie zdefiniowałem to operatory porownójące `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` mnoże 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 w 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 typu `field` ans1 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 `b` 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 nie robię nic, tylko odrazu 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 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 operastor `^=` 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 wykożystujemy 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 konstruktóra 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 operartora `*=` 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`.

Kolwjnymi 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 rządanej 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`.

Zmienna `fromString` kożysta z funkcji `toStringOneNumber`, która zakładam, że przyjmuje liczbę typu `__int128` i zwraca jej zapis jej reszty z dzielenia przez `10` w formnie 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 inicjalizuje zmienną wynikową `ans` na słowo puste, a następnie wykonuje pętlę

```
while(B != 0){
    if(B \% 2 == 1)ans *= multiplier;
    multiplier *= multiplier;
    B /=2;
}
```

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

Definicja: $\pi(x)$ jest funkcją $\mathbb{N} \to \mathbb{N}$ przypisująca argumentowi x liczbę liczb pierwszych niewiększych 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 $O((n+t)^2)$. 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 $\prod_{i=1}^{t}(1+x^{s_j})$ 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 niewiększy 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 Furiera, opartej na liczbach zespolonych, a teorio-liczbowej szybkiej transformaty Furiera wykonującej obliczenia w ciele Z_p. Tak więc w tym ciele Z_p musi istnieć pierwiastek z 1 stopnia r gdzie r jest postaci 2^k, dla pewnej liczby naturalnej k, oraz r jest większe lub równe stopniowi wielomianu, który powstanie po pomnożeniu wielomianów, które mnożymy naszą transformatą powiększonego o 1. Mnożone wielomiany są stopnia co najwyżej t, tak więc możemy przyjąć, że 2^k to najmniejsza potęga 2 o całkowitym wykładniku większa niż $2t+2$, zaś p ma postać $r2^k+1$. 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 tego pierwiastka podniesiony do potęgi r wynosi 2^k.

Intuicjyjnie 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 $[t+1,(n+t)^3]$. Dla uproszczenia możemy badać przedział $[0,(n+t)^3]$, ponieważ $t+1$ jest względnie małe w stosunku do $(n+t)^3$. Ostatecznie nasza hipoteza jest taka, że jeżeli $\pi'(x)$ to liczba liczb pierwszych postaci $r2^k+1$, gdzie r jest liczbą naturalną niewiększą od x. To $lim_{x\to\infty}\frac{\pi'(x)}{\frac{x}{\ln(x)}}$ jest równe 1.

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 i przy użyciu programu Jupyter Notebook.

Oczywiście eksperymentalne wyznaczenie $\pi'(x)$ dla dużych x-ów byłoby bardzo czasochłonne więc wartość $\frac{\pi'(x)}{x}$ szaacowałem przy pomocy funkcji, która najpierw losowała 100000 liczb typu $UInt64$ modulo x, mnożyłem, przez 2^k i dodawałem 1, zaś następnie sprawdzałem przy pomocy funkcji *isprime* należącej do pakietu *Primes* czy otrzymana tak liczba jest pierwsza i jeśli odpowiedź była pozytywna inkrementowałem zmienną ans, której początkowa wartość wynosiła 0. Następnie zwracałem $\frac{ans}{100000}$, czyli liczbę stosunek wylosowanych liczb pierwszych do ilości losowań. Wartość $\pi'(x)$ wyliczałem dla x postaci $100000000000000k+1$, gdzie $k \in \{0,1,..99\}$, oraz gdzie parametr 2^k jest równy $1,2,4,8,16,64,256,1024,2^{20},2^{30}$ i 2^{50}. Żeby uniknąć problemów z tym, że liczby przekraczać będą limity zmiennej całkowitej 64-bitowej bez znaku odrazu po wyliczeniu losowanej liczby modulo x rzutowałem ją na zmienną BigInt.

Na poniższym wykresie prezentuję jak poniższa metoda pozwoliła szacować $\frac{\pi'(x)}{\frac{x}{\ln(x)}}$ (oś pionowa), w zależności od argumentu x(oś pionowa) i parametru 2^k(kolor).

Jak widać z wykresu nawet dla bardzo dużych 2^k i x, wartości $\frac{\pi'(x)}{\frac{x}{\ln(x)}}$ są rzędu wielkości $O(1)$, co więcej dla dość małych wartości $2^k > 1$ można zauważyć, że $\frac{\pi'(x)}{\frac{x}{\ln(x)}}$ jest większe niż 1 i zbliżone do 2. Zgadza się to z intuicją, że jeżeli wybieramy jedynie liczby nieparzyste, to prawdopodobieństwo natrafienia na liczbę złożoną jest około 2 razy mniejsze, bo "odpadają"nam liczby parzyste, które poza 2 są zawsze

złożone. Być może jeszcze zwiększając zakres x okazałoby się, że że dla $2^k = 2^{50}$ również badana wartość zaczęłaby się zbliżać do 2, jednak otrzymane szacowania i tak są już dla nas satysfakcjonujące, tym bardziej, że mnożenie wielomianów rozmiarów rzędu 2^{50} nawet przy pomocy szybkiej transformaty Furiera i tak dla zdecydowanej większości współczesnych komputerów jest zadaniem niemożliwym do wykonania w czasie, który można by nazwać "rozsądnym"a sama pamięć potrzebna do przechowania tak dużej ilości współczynników byłaby rzędu petabajtów.

Plik typu ipynb w którym przechchowuję kod użyty w omawianych wyżej eksperymentach znajduje się w dołączonym do pracy repozytorium i nosi nazwę "primeExperiment.ipynb".

Omówię teraz już samą implementację modułu szukającego liczby pierwszej. Wszystkie wymienione niżej funkcje zostały zaimplementowane w języku C++.

Składa się on z kilku funkcji, z których zdecydowanie najprostsza jest *absolutevalue* która zwraca po prostu wartość bezwzględną liczby którą przyjmuje jako argument. Służy ona po prostu łatwemu pozbyciu się problemu z tym, że niektóre liczby losowane przez używaną w dalszej części kodu funkcję *rand*() liczby mogą być ujemne, podczas gdy iteresują nas tylko liczby dodatnie. Druga bardzo prosta funkcja to funkcja *pow*2 która dla przyjmowanej jako argument liczby t zwraca najmniejszą potęgę 2 z wykładnikiem naturalnym, która jest większa niż t. Jest to po prostu mnożenie przez 2 zmiennej *ans* której początkowa wartość jest równa 1 dopóki nie będzie większa niż t. Musimy wykonać co najwyżej 63 mnożenia, co jest na tyle szybkie, że nie ma potrzeby stosowania jakiś bardziej zaawansowanych algorytmów jak na przykład używających algorytm szybkiego potęgowania.

Funkcja *randomLongLong* służy losowaniu liczby 64-bitowej niewiększej niż podana jako argument liczba *mod*. Należąca do standardu C++ funkcja *rand* zwraca liczby 32-bitowe, co może nie być dla nas wystarczające. Jeżeli $mod < 2^{31}$ zwracam po prostu resztę z dzielenia przez *mod* wartości bezwzględnej wyniku wywołania *rand*(). Jeżeli zaś jest ona większa niż 2^{31} najpierw pobieramy jej 31 najmniej znaczących bitów z wylsowanej liczby *candidate*1, zaś następnie losujemy liczbę całkowitą *candidate*2 z przedziału $[0, \frac{mod}{2^{31}}]$. Potem łączymy obie liczby wyliczając wartość $candidate2 \cdot 2^{31} + candidate1$ co jest faktycznie losową liczbą z zadanego przedziału.

Kolejną nieco bardziej złożoną funkcją jest funkcja *isprime* sprawdzającą czy podana jako argument liczba x jest pierwsza. Jeżeli $x \leq 1$ oczywiście zwracamy fałsz. Jeżeli zaś jest równe 2 zwracamy prawdę. Jeżeli jest liczną parzystą większą niż 2 zwracamy fałsz. Następnie zmienną i o początkowej wartości 3 iterujemy się po kolejnych licznach nieparzystych dopóki zachodzi warunek $i \cdot i \leq x$. Jeżeli i dzieli x przerywamy iterację i zwracamy fałsz. Jeżeli zaś zakończymy omawianą pętlę przed zwróceniem fałszu zwracamy prawdę. Istotnie liczba nieparzysta złożona musiałaby mieć dzielnik nieparzysty mniejszy lub równy jej pierwiastkowi kwadratowemu. Niepokoić może fakt, że musimy w pesymistycznym przypadku wykonać $O(\sqrt{x})$ iteracji. W praktyce jednak nawet jeżeli x jest bardzo duże, zbliżone do górnego zakresu zmiennych typu *long long* wykonanie tej funkcji zajmuje najwyżej kilka sekund, a ponieważ zdecydowana większość liczb złożonych ma dość mały najmniejszy dzielnik pierwszy to, w praktyce o ile liczba podana na wejście nie jest pierwsza, dla nawet bardzo dużych x szybko przerwiemy wykonanie omawianej pętli i nie będzie konieczności czekać nawet tych kilku sekund.

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 $[t+1, (n+t)^3]$. Na początku przy pomocy funkcji *pow*2 wyznaczam 2^k. Następnie losuję potencjalne liczby r modulo $(n+t)^3$, przy pomocy funkcji *randomLongLong*. Jeżeli wylosuję liczbę niewiększą niż t losuję jeszcze raz. Dla zwłaszcza dużych n i t prawdopodobieństwo tego jest znikome, a nawet jeśli się zdaży, to czas powtórzenia losowania jest znikomy. 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 *isprime*. Niepokoić może, koniczność wykonywanie niewiadomej ilości losowań i sprawdzeń, pierwszości wylosowanych liczb. Warto jednak zauważyć, że oczekiwana liczba losowań wynosi $O(\log(n+t))$ i większość przypadków sprawdzania złożoności liczb złożony zachodzi bardzo szybko, bo bardzo szybko w funkcji *isprime* znajdujemy jej dzielnik pierwszy(na przedziale $[0, N]$ dla dużych N zaledwie nieco ponad 0.08 liczb ma dzielnik pierwszy większy niż 1000000 co szybko może sprawdzić poniższy krótki skrypt w Julii:

```
zans=1
for 3 in 1:1000000
    if isprime(i)
        ans *=(i-1)
        ans /= i
```

```
    end
    println(ans)
end
```

Kod funkcji omówionych w tej sekcji znajduje się w pliku $find\_prime.h$.

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

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 $\sum_{i=0}^{n-1} a_{2n}t^i$, 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 istneie I takie, że $\forall i > I : a_i = I$. 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 wykadniku naturalnym, większym niż stopień wielomianu będącego wynikiem optymalizowanego przez nas mnożenia.

Będę oznaczał ω_m jako $e^{\frac{i2\pi}{2^m}}$. Zaóważmy, 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 wektory V_1 i V_2 można znów rozbić na wyliczenie najpierw 2 wektorów długości 2^{m-2} i połączenia tych dwóch wektorów w czasie liniowym od ich długośći. 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 kożysta 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 nauralną, o czym pisałem w poprzedniej sekcji.

Drugim ulepszeniem jest użycie tak zwanej szybkiej transformaty Furiera. Opiera się ona na obserwacji że rekurencyjna wersja dyskretnej transformaty Furiera polega najpierw na dzieleniu współczynników wielomianu na coraz mniejsze grupy, aż do zawierających tylko pojedynczy współczynnik, a następnie łączeniu pojedynczych współczynników w pary zawieerające 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 wywoływa transformaty po prostu ustawiając obok siebie współczynniki w takiej kolejności żeby pary indeksów $(0,1)$, $(2,3)$,...,$(2^m-2, 2^m-1)$ odpowiadały polom w wektorze łączonych na najgłębszym poziomie rekurencji, $(0,1,2,3)$, $(4,5,6,7)$, $(8,9,10,11)$,...,$(2^m-4, 2^m-3, 2^m-2, 2^m-1)$ na drugim najgłębszym poziomie rekurencji itd.

Przejdźmy do dokładnego opisu naszej implementacji.

Najpierw definiuję funkcję *enoughGoodRoot* która przyjmuje liczbnę k typu *long long* i zwraca element typu $field$ na którym aby był równy 1 trzeba wykonać dokładnie k podniesień jej do kwadratu. Na początku deklaruję zmienną $powerof2$ typu *long long* będącą największą potęgą 2 o wykładniku naturalnym takim, że dzieli on zmienną statyczną p klasy $field$ pomniejszoną o 1, którą definiowałem w poprzedniej sekcji. Liczbę tą wyznaczam przez przypisanie jej początkowej wartości równej 1, i mnożeniu jej w pętli przez 2 dopóki wartość wyrażenia $(field :: p-1)/powerof2$ nie będzie liczbą nieparzystą. Następnie tworzymy zmienną *odd* typu *long long* będącą po prostu liczbą po której pomnożeniu przez $powerof2$ i dodaniu 1 otrzymyujemy $field :: p$. Wiemy, z małego twierdzenia Fermata, że każda liczba względnie pierwsza z $field :: p$ podniesiona do $2^{powerof2}odd$ przystaje do 1 modulo $field :: p$. Dalsza część algorytmu polega na losowaniu niezerowego obiektu klasy $field$ o nazwie *candidate*, podniesieniu go w miejscu do potęgi *odd*, a następnie sprawdzeniu ile razy tak uzyskany obiekt należy podnieść do kwadratu, nim będzie on równy 1. W tym celu podnosimy w pętli do kwadratu kopię *candidate* i póki nie dojdziemy do 1 inkrementujemy zmienną $funnyLog$, która początkowo jest równa 0 i po wykonaniu omawianej pętli ma ona wartość równą liczbie tych podniesień do kwadratu. Jeżeli $funnyLog$ jest większe niż k (zakładamy, że $2^k < powerof2$) to liczba powstała przez podniesienie *candidate* do kwadratu $funnyLog - k$ razy jest naszym szukanym pierwiastkiem. Jeżeli tak nie jest losujemy do skutku nowego *candidate*, jednak za każdym razem prawdopodobieństwo sukcesu jest nie mniejsze niż $\frac{1}{2}$, bo z prawdopodobieństwem $\frac{1}{2}$ wylosujemy niereszte kwadratową $field :: p$. (ZRODOLOOO ZNALEZC!!!!!!!!!!!!!!!!!!!!).

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. Okazuje się, że należy je uszeregować w takim porządku, że wartość znajdująca się w polu $coefficients[n]$ musi znaleźć się w polu $reverseBits(n,k)$, która oznacza zmienną powstałą po odwróceniu k ostatnich bitów zmiennej n, gdzie 2^k to ilość pól wynikowego wektora. Funkcja $reverseBits$ składa się z klasycznego algorytmu odwócenia kolejności bitów zmiennej, polegającym na najpierw zamianie ze sobą dwóch sąsiednich bitów przy pomocy operatorów |, przesunięć bitowych i wyzerowaniu odpowiednich bitów przy pomocy operatora & i odpowiednich masek bitowych, następnie podobnej zamieniam sąsiednie bloki po 2 bity, następnie 4, 8, 16 i 32. Potem koryguję wynik przesówając go bitowo o $64-k$ bitów w lewo i wtedy tak uzyskaną zmienną zwracam jako wynik funkcji.

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 rządaną długość wektora wynikowego(tak naprawdę jest to odpowiednio duża potęga dwójki o wykładniku naturalnym). Na początku dopełniamy wektor $coefficients$ elementami równymi $field(0)$ do rozmiaru *size*. Następnie w pętli z *iteratorem* i przechodzącym po wszystkich liczbach naturalnych od 0 do $size-1$ i jeżeli $reverseBits(i, \log_2(size)) > i$(warunek ten służy temu, by każdy element został zamieniony dokładnie raz, zauważmy też, że $reverseBits(reverseBits(i, \log_2(size)), \log_2(size)) = i$) zamieniam w tabeli pole o indeksie i z polem o indeksie $reverseBits(i, \log_2(size))$ 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* takim, że podniesiony do jakiejkolwiek liczby naturalnej mniejszej niż size nie jest równy 0 zwraca wektor zawierający wartości tego wielomianu w punktach kolejno $omegaM^0$, $omegaM^1$,...,$omegaM^{size}$.

Poniżej umieściłem kod tej procedutry.

```
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 przygotowywujemy 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 podniesiania 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 który podniesiony do żadnej mniejszej niż 2^i potęgi o wykładniku naturalnym nie jest równy 1, zaś f_j

Przyjmijmgy 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 wiliczamy wartość $size$ będącą liczbą punktów w których będziemy liczyć wartości wielomianu AB(jest to najmniejsza liczba będąca potgą 2 o wykładniku naturalnym większa niż stopień AB). Kolejnym etapem jest znalezienie pierwiastka stopnia $size$ z 1 w Z_p, nie będącym w tym ciele jednocześnie pierwiastkiem z 1 niższeg 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.

Zaóważmy, 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 algorytmyu 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 $t+1$ pierszych 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 funkcjie 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 rządania 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 niewiększe 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 kożysta 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 wykożystanie 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 przezemnie 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 napewno jest niewiększa niż 2^n, więc ma co najwyżej n różnych czynników pierwszych. Eksperymenty w sekcij 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 $O(\frac{n}{(n+t)^2}) = O(\frac{1}{n+t})$.