Opis projektu z Architektury Komputerów 2		
Prowadzący	dr inż. Dominik Żelazny	
Temat projektu	Arytmetyka modularna	
Termin zajęć	Wtorek, 13:35-15:05 TN	
Skład grupy projektowej	Daniel Leśniewicz	250996
	Patryk Fidrych	248828

Wstęp

Arytmetyka modularna jest bardzo przydatnym narzędziem matematycznym, umożliwiającym rozwiązywanie problemów algorytmicznych. Podstawowym jej celem jest zredukowanie skomplikowanych obliczeń. Polega to na zastąpieniu działań na liczbach przez działania na resztach z dzielenia tych liczb. Współcześnie jest powszechnie używana w wielu systemach kryptograficznych. Okazuje się, że jej zastosowanie pozwala znacznie usprawnić czas wykonywania obliczeń na dużych liczbach.

W naszym projekcie zaimplementowaliśmy algorytmy, które do rozwiązania problemów w swoim działaniu wykorzystują arytmetykę modularną. Dodatkowo w celu poszerzenia wiedzy na temat języka assembler, algorytmy zostały zaimplementowane w tymże języku.

Zaimplementowane algorytmy

W naszym projekcie udało się zaimplementować następujące algorytmy:

- Algorytm Euklidesa
- Naiwny Algorytm Sprawdzania Pierwszości Liczb
- Chiński Test Pierwszości
- Male Twierdzenie Fermata
- Test Millera-Rabina

Algorytm Euklidesa służy do wyznaczania NWD liczb. Pozostałe z wymienionych algorytmów zajmują się badaniem pierwszości liczb.

Zostały one opisane poniżej.

Plik main.cpp

Pliku main.cpp znajduje się wywołanie funkcji wyświetlania menu głównego programu oraz logika odpowiedzialna za wybieranie określonego algorytmu. Zostało to zaimplementowano z użyciem switch`a. W zależności od wybranej opcji, zostaje wywołany właściwy algorytm.

Fragment pliku *main.cpp*:

```
1.          switch(option){
2.          case 1:
3.              cout << "\n------ALGORYTM EUKLIDESA------" << endl;
4.              Euklides::enterValues();
5.              break;
6.          case 2:
7.              cout << "\n------NAIWNY ALGORYTM SPRAWDZANIA PIERWSZOSCI LICZB------" << endl;
8.              NaivePrime::enterValue();
9.              break;
10.         case 3:
11.             cout << "\n---------CHINSKI TEST PIERWSZOSCI---------" << endl;
12.             ChineseTestPrime::enterValue();
13.             break;
14.         case 4:
15.             cout << "\n---------MALE TWIERDZENIE FERMATA---------" << endl;
16.             LittleFermat::enterValue();
17.             break;
18.         case 5:
19.             cout << "\n---------TEST MILLERA-RABINA---------"<< endl;
20.             MillerRabin::enterValue();
21.             break;
22.         case 0:
23.             exit(0);
24.             break;
25.         default :
26.             cout << "Bledny numer wybranego algorytmu!"<< endl;
27.             break;
28.         }
29.         cout << "\n\nNaciśnij dowolny przycisk, aby przejść dalej!" << endl;
30.         getch();
31.         system("CLS");
32.
33.     }while (option != 0);
```

Plik Euklides.cpp

W tym pliku został zaimplementowany kod algorytmu Euklidesa. Sprawdza on jaki jest największy wspólny dzielnik dwóch liczb. Warto zaznaczyć, że jest to jeden z najstarszych algorytmów, został opisany ok roku 300 p.n.e. Funkcja realizująca algorytm jest napisana w assemblerze. Pobiera dwie liczby całkowite i zwraca wynik w postaci największego wspólnego dzielnika wyżej wymienionych liczb.

Kod funkcji:

```
1.  int Euklides::NWD(int a, int b){
2.          asm("\
3.          # 8(%ebp) - zmienna a                                          \n\
4.          # 12(%ebp) - zmienna b                                         \n\
5.          # %ecx - rejestr pomocniczy                                    \n\
6.                                                                         \n\
7.          start_loop:                                                    \n\
8.              cmpl $0, 12(%ebp)         # jesli b=0 to koniec            \n\
9.              je end_loop                                                \n\
10.             movl 12(%ebp), %ecx       # przechowanie wartosci b        \n\
11.                                                                        \n\
```

```
            # OPERACJA MODULO                                                   \n\
            # Wykonanie dzielenia: podwojne slowo w %edx:eax                    \n\
            # przez argument instr. div                                         \n\
            # W tym przypadku a/b. Resza dzielenia w %edx                       \n\
            movl $0, %edx                                                       \n\
            movl 8(%ebp), %eax                                                  \n\
            movl 12(%ebp), %ebx                                                 \n\
            divl %ebx                                                           \n\
                                                                                \n\
            # Przypisanie reszty z dzielenia do b                               \n\
            movl %edx, 12(%ebp)                                                 \n\
            # Stara wartosc b jest nowa dzielna                                 \n\
            movl %ecx, 8(%ebp)                                                  \n\
            jmp start_loop                                                      \n\
        end_loop:                                                               \n\
    ");

    return a;
```

W działaniu algorytmu wykorzystywana jest operacja modulo. W implementacji funkcji, pierwszą przekazaną do funkcji liczbę reprezentuje parametr *a*, a druga parametr *b*. Na początku wykonywane jest dzielenie z resztą pierwszej liczby przez drugą liczbę. W zależności od otrzymanego wyniku operacji modulo podejmowane są odpowiednie decyzje. Jeśli reszta z pierwszego dzielenia jest równa 0, to największym wspólnym dzielnikiem jest druga z liczby przekazanych do funkcji. Gdy reszta jest różna od zera to następuje przypisanie parametru *a* wartości parametru *b*. Do parametru *b* przypisywana jest wartość reszty z dzielenia. Następnie ponownie jest wykonywane dzielenie parametru *a* przez parametr *b* dopóki reszta nie będzie równa 0. Operacja jest wykonywana w pętli *while*, co pozwala na zwrócenie prawidłowego wyniku.

Plik NaivePrime.cpp

Kolejnym algorytmem, który został zaimplementowany jest naiwny algorytm sprawdzania pierwszości liczb. Funkcja przedstawiona poniżej zwraca *1* jeśli dana liczba jest liczbą pierwszą lub *0* jeśli nie jest liczbą pierwszą.

Kod funkcji:

```
int NaivePrime::checkPrime(int a){

    int root=sqrt(a);

    asm("\
        # 8(%ebp) - zmienna a                                                   \n\
        # -12(%ebp) - sqrt(a)                                                   \n\
                                                                                \n\
        movl 8(%ebp), %ecx      # zapis do %ecx sprawdzanej liczby              \n\
        movl $2, %ebx           # licznik pierwszosci do symulowania zakresu    \n\
                                # a do sqrt(a)                                  \n\
                                                                                \n\
        cmpl $1, %ecx           # jesli liczba jest 1, to nie jest pierwsza     \n\
        je niePierwsza                                                          \n\
                                                                                \n\
        start_loop:                                                             \n\
        cmpl %ebx, -12(%ebp)    # jesli róznica: sqrt(a)-licznik pierw. jest    \n\
        jl loop_exit            # mniejsze od 0 to koniec petli                 \n\
        movl %ecx, %eax         # zapis do %eax sprawdzanej liczby              \n\
                                                                                \n\
        movl $0, %edx           # dzielenie %edx:%eax przez arg instr. divl     \n\
        divl %ebx               # czyli wartosc licznika pierwszosci            \n\
                                                                                \n\
```

```
24.                                 # reszta z dzielenia jest w %edx             \n\
25.         cmpl $0, %edx           # jesli reszta rowna 0 to liczba              \n\
26.         je niePierwsza          # nie jest pierwsza - koniec petli            \n\
27.                                                                               \n\
28.                                                                               \n\
29.         incl %ebx               #zwieksz licznik                            \n\
30.         jmp start_loop                                                        \n\
31.                                                                               \n\
32.         loop_exit:                                                            \n\
33.         movl $1, 8(%ebp)        # jesli jest pierwsza to a=1                  \n\
34.         jmp koniec                                                            \n\
35.                                                                               \n\
36.         niePierwsza:            # jesli nie jest pierwsza to a=0              \n\
37.         movl $0, 8(%ebp)                                                      \n\
38.                                                                               \n\
39.         koniec:                                                               \n\
40.     ");
41.
42.     return a;
43. }
```

Algorytm polega na próbnym dzieleniu sprawdzanej liczby *a* przez liczby z zakresu od *2* do *sqrt(a)*. Przy każdej takiej operacji badana jest reszta z dzielenia. Jeśli reszta podczas którejś operacji dzielenia będzie wynosiła 0, to liczba *a* nie będzie liczbą pierwszą. Wystarczy sprawdzić liczby jedynie z zakresu *2* do *sqrt(a)*. Jeśli liczba posiada czynnik większy od *sqrt(a)*, to drugi jego czynnik musi być mniejszy od pierwiastka z *a*, aby ich iloczyn musiał być równy *a*. Zatem wystarczy podzielić liczbę *a* przez liczby z danego przedziału, aby wykluczyć liczby złożone.

Plik ChineseTestPrime.cpp

Chiński test pierwszości jest kolejnym algorytmem pozwalającym na zbadanie pierwszości liczby. Został on odkryty około 500 lat p.n.e. w Chinach. Zauważono, że jeśli liczba *a* jest liczbą pierwszą to wyrażenie $2^a - 2$ jest podzielne przez liczbę *a*. Na tej podstawie można wyciągnąć wniosek: jeśli $2^a mod\ a\ \neq 2$ to liczba a nie jest liczbą pierwszą. Jednakże podstawowym problemem było obliczanie dużych potęg liczby 2. Aby zrealizować to w prosty sposób wykorzystano arytmetyką modularną. Biorąc pod uwagę fakt, że reszta z dzielenia 2^a przez *a*, jest z przedziału <0; a-1>, w algorytmie można w prosty sposób obliczyć wyrażenie $2^a mod\ a$. Kolejne potęgi są rozbijane na operacje mnożenia modulo a. Dzięki wymnażaniu kolejnych reszt istnieje możliwość wyznaczania dużych potęg 2^a.

W celu realizacji algorytmu w programie zostały zaimplementowane dwie funkcje. Pierwsza z nich zwraca wynik następującej operacji: $(num1\ \cdot num2)\ mod\ modNum$. Podczas działaniu wykorzystuje dodawanie wielokrotności mnożnej modulo *modNum*. Druga z funkcji zwraca wartość wyrażenia $2^{num1}\ mod\ modNum$ i w celu zwrócenia prawidłowego wyniku wykorzystuje pierwszą z funkcji. Dodatkowo posłużono się możliwością przesuwania bitów. Dokładny opis zastosowania maski bitowej znajduje się w sprawozdaniu z przebiegu projektu. Gdy druga z funkcji skończy działanie to zwraca wynik, który jest porównywany z liczbą 2 i na tej podstawie jest ustalana wartość liczby. Należy również zaznaczyć, że powyższy algorytm ma również wady. Jeśli $2^a mod\ a = 2$, to liczba a jest liczbą pierwszą lub pseudopierwszą przy podstawie 2. Jednakże liczby pseudopierwsze

przy podstawie 2 występują bardzo rzadko i istnieje niewielkie prawdopodobieństwo, że algorytm zwróci niepoprawny wynik.

Ze względu na obszerność kodu, algorytm nie został on wklejony w tym dokumencie.

Plik LittleFermat.cpp

Pierre de Fermat jest autorem Małego Twierdzenia Fermata, które bazuje na chińskim twierdzeniu o pierwszości. Twierdzenie Fermata zostało ono odkryte w 1640 roku. Można je przedstawić następująco:

Dla liczby pierwszej num i dowolnej liczby naturalnej a, jeśli *NWD (num, a) = 1, to* $a^{num\ -1}$ *mod num = 1.* Twierdzenie to jest bardziej ogólne od twierdzenia chińskiego.

W zaimplementowanym algorytmie na początku sprawdzana jest podzielność wprowadzonej liczby przez liczby pierwsze z przedziału <2;1000>. Umożliwia to wstępną eliminację liczb złożonych oraz liczb pseudopierwszych Carmichaela. Jeśli liczba będzie złożona, to zwracana jest od razu informacja, że nie spełnia ona zadanego warunku. Następnie w pętli sprawdzany jest warunek Fermata. Zostaje wylosowana podstawa *a*, która zawiera się w przedziale od 2 do testowanej liczby pomniejszonej o jeden. Następnie sprawdzane jest czy *a* jest względnie pierwsza z wprowadzoną liczbą *num*. Polega to inaczej mówiąc na sprawdzeniu, czy *NWD (num, a) = 1*. Jeśli tak, to testowany jest warunek: $a^{num\ -1}$ *mod num = 1*. Jeśli zostanie spełniony ten warunek, to liczba może być liczbą pierwszą. Jednak nie jest to na tym etapie pewne. Aby to potwierdzić należy sprawdzić to wykonując test Fermata np. dziesięciokrotnie. Liczba, która została uznana liczbą pierwsza może być również liczbą pseudopierwsza Carmichaela. Liczby te są jednak bardzo odległe, a stosując podzielność przez liczby pierwsze z przedziału <2;1000> można być pewnym, że algorytm daje poprawne wyniki.

Aby sprawdzić, czy dana liczba jest pierwsza należy wywołać *controller*, zarządza on powyższą logiką algorytmu. Wywołuje on funkcję *multiplicationModuloF()* oraz *powerModuloF()*. Dodatkowo w algorytmie został wykorzystany kod algorytmu Euklidesa, aby wyznaczyć NWD liczb.

Również w tym przypadku, ze względu na obszerność kodu, algorytm nie został on wklejony w tym dokumencie.

Plik MillerRabin.cpp

Ostatnim algorytmem jest test Millera-Rabina który sprawdza pierwszość liczby nieparzystej. Został on opracowany w 1975r. Test powstała oparty na deterministyczny algorytm Millera, którego poprawność zależy od nieudowodnionej uogólnionej hipotezy Riemanna. Warto zaznaczyć, że jest to metoda probabilistyczna dlatego nazywana jest testem. Metoda pozwala sprawdzić pierwszość liczby z określonym prawdopodobieństwem błędu.

Funkcja realizująca algorytm przyjmuje dwa argumenty: liczbę do przetestowania ze względu na pierwszość oraz liczbę określającą ilość powtórzeń testu. Oczywiście jeśli jest większa ilość

powtórzeń to istnieje większe prawdopodobieństwo, że test zwróci poprawny wynik. Algorytm zwraca informację o pierwszości liczby (lub silnej pseudo-pierwszości) z prawdopodobieństwem ¼^{n}. Stała n w tym przypadku oznacza ilość powtórzeń testu.

Test w swoim działaniu wykorzystuje mnożenie oraz potęgowanie modulo. Funkcje te zostały napisane w języku assembler, jednak ich wywołania są zrealizowane w języku C++. Dodatkowo warto zaznaczyć, że test jest wykorzystywany we współczesnych systemach kryptografii publicznej.

Podsumowanie

Udało nam się wykonać wszystkie założenia dotyczące projektu przesłane w pierwszym etapie. Realizacja projektu pozwoliła nam zagłębić temat arytmetyki modularnej. Dowiedzieliśmy się, że arytmetyka modularna umożliwia sprawne rozwiązywanie wielu problemów algorytmicznych. W naszym projekcie zaimplementowaliśmy 5 algorytmów, które korzystają z działań na resztach w celu rozwiązania konkretnych problemu. Pierwszy etap projektu nie sprawił nam większych trudności. Zrealizowaliśmy w nim algorytm Euklidesa oraz naiwny test pierwszości. Drugi etap był już trochę trudniejszy, ponieważ algorytm chińskiego testu pierwszości oraz małe twierdzenie Fermata były bardziej wymagające pod względem realizacji w języku assembler. Najtrudniejszym w implementacji okazał się test Millera-Rbina. Powodem może być fakt, że algorytm sprawdzania pierwszości liczby pierwszej jest dosyć złożony. Każdy z algorytmów działa prawidłowo, co potwierdza finalna wersja programu konsolowego. Gdyż algorytmy zostały zaimplementowane w języku assembler, pozwoliło to nam na znaczne poszerzenie wiedzy również na ten temat. Oczywiście postać kodu assemblera nie jest zbyt przejrzysta dlatego staraliśmy się objaśniać zasadę działania funkcji poprzez użycie komentarzy.