Raport

Z wykonania projektu z kursu Organizacja i Architektura Komputerów

Szybkie potęgowanie wielkich liczb.

Maciej Ciura

248879

Prowadzący: Prof. Janusz Biernat

Spis treści

1. Wstęp

1.1 Cel projektu

Zadaniem projektowym było zaimplementowanie algorytmu szybkiego potęgowania liczb naturalnych o dowolnej długości. Program miał być jak najbardziej optymalny.

1.2 Założenia

Podczas opracowywania i implementacji algorytmu zostały przyjęte następujące założenia:

- Algorytm powinien operować na liczbach w reprezentacji heksadecymalnej.
- Podstawa potęgi powinna być długości większej niż 4096b,
- Wykładnik potęgi powinien być liczbą z zakresu $<0, 2^{32}>$,
- Złożoność obliczeniowa algorytmu O(log n)

1.3 Algorytm potęgowania

Zaimplementowany algorytm to tzw. *„Szybkie potęgowanie"*.

Wykorzystując zależność $a^{n+m} = a^n \cdot a^m$, jesteśmy w stanie w teorii zmniejszyć złożoność obliczeniowa z O(n) do O(log n).
Kolejne wartości potęg obliczane są dynamicznie:

$$a_0 = a^1 = a$$
$$a_1 = a^2 = a_0 \cdot a_0$$
$$a_2 = a^4 = a_1 \cdot a_1$$
$$a_3 = a^8 = a_2 \cdot a_2$$

Dane wejściowe:
- podstawa potęgi – dowolnej długości liczba zapisana w reprezentacji heksadecymalnej jako ciąg znaków ASCII,
- wykładnik potęgi – liczba naturalna z zakresu $<0, 2^{32}>$,
Dane wyjściowe:
- wynik – liczba o długości n * k, gdzie n – długość podstawy, k – wykładnik

Algorytm zapisany w pseudokodzie:

K01: w ← "1"	*ustawiamy wynik na 1*
K02: **Jeśli** (*b* **and** 1) = 1, **to** *w* ← mnóż (*w*, *a*)	*jeśli bit b_i = 1, to wymnóż w przez a_i*
K03: *b* ← *b* **shr** 1	*przesuń bity w b o jedną pozycję w prawo*
K04: **Jeśli** *b* = 0, **to zakończ**	*reszta bitów jest zerowa, kończymy*
K05: *a* ← mnóż (*a*, *a*)	*oblicz kolejne a_i*
K06: **Idź do** kroku K02	*kontynuuj pętlę*

1.4 Algorytm mnożenia

Algorytm mnożenia został zrealizowany jako klasyczne mnożenie.
Każda cyfra mnożnej jest mnożona przez każdą cyfrę mnożnika. Jest to mało efektywny algorytm o złożoności $O(n^2)$ (przy założeniu, że mnożenie realizuje operację n^2). Mała efektywność algorytmu prawdopodobnie znacząco wpłynie na ostateczną złożoność całego algorytmu potęgowania.

2. Realizacja

2.1 Struktura programu

Projekt dzieli się na 3 części:

-program główny – napisany w języku C++, zawiera algorytm potęgowania oraz potrzebne struktury,

-algorytm mnożenia – w całości zaimplementowany w języku asemblera x86,

-skrypty testowe – skrypty napisane w języku python, pozwalające na sprawdzenie poprawności wyników programu, przeprowadzenie testów oraz opracowanie danych.

Program można skompilować w dwóch różnych wersjach – testowej oraz demonstracyjnej.

Całość była projektowana, oraz wykonana na systemie operacyjnym Linux, przy użyciu następujących narzędzi:

- język programowania – C++, x86, python
- edytor tekstu – Atom, Vim
- kompilator – GCC
- analiza danych – moduł python matplotlib
- debugowanie – GDB, Valgrind, kdbg
- inne – Make

2.2 Wprowadzanie danych

Dane wprowadzane są w postaci ciągu znaków, składającego się z cyfr reprezentacji heksadecymalnej podstawy potęgi, oraz z reprezentacji heksadecymalnej wykładnika.

Liczby są zapisywane odpowiednio do typu string i unsigned int. Zmienna typu string zawierająca podstawę jest zaszyta w klasie o nazwie BigNum.

W wersji demonstracyjnej dane pobierane są bezpośrednio ze strumienia standardowego, w testowej zaś ze specjalnie wygenerowanego skryptem ***test.py*** pliku. Plik ten zawiera:

n- długość podstawy w bajtach,
k- wykładnik,
k par liczb:

i-ta linia: podstawa długości n bajtów
i+1 linia: wykładnik = i, gdzie i ∈ <0,k>

2.3 Struktura danych

Podstawa potęgi, jak i wyniki pośrednie przechowywane są w obiektach klasy BigNum:

```
class BigNum {

public:
    unsigned int size;
    unsigned int *tab;

    BigNum();
    explicit BigNum(unsigned int size);
    explicit BigNum(std::string str);
      ~BigNum();

      BigNum &operator=(BigNum const & num);
      unsigned int operator[](int n);
  };
```

Klasa ta zawiera:

-tablicę dynamiczną typu unsigned int - do przechowywania liczb. Z racji użycia tego typu danych, liczba jest dzielona na fragmenty 4 bajtowe.
- rozmiar tablicy – 8 razy mniejszy niż ilość znaków w reprezentacji liczby

W pliku BigNum obok klasy znajdują się też przeładowane operatory, bardzo pomocne w organizacji kodu, oraz funkcja pomocnicza charToInt():

```
int charToInt(char c);
std::istream & operator>>(std::istream &in, BigNum &num);
std::ostream & operator<<(std::ostream &out, BigNum &num);
```

2.4 Algorytm potęgowania

Dzięki wyodrębnieniu funkcji oraz przeciążeniu operatora ‘=’ było możliwe zaimplementowanie algorytmu w bardzo przejrzystej formie:

```
while( true )
    {
        if( b & 1 )    result = multiply ( result, a );
        if( b >>= 1 ) a = multiply ( a, a ); else break;
    }
```

2.5 Algorytm mnożenia

Algorytm mnożenia został zaimplementowany jako funkcja w języku asemblera x86:

```
void multiply_as(unsigned int *a, unsigned int a_size, unsigned int
*b, unsigned int b_size, unsigned int *result);
```

Do funkcji multiply() przekazywane są dwa obiekty klasy BigNum. Funkcja tworzy nowy obiekt o rozmiarze (a.size + b.size) – maksymalnym rozmiarze wyniku mnożenia. Po wykonanym mnożeniu zmniejsza rozmiar tablicy, by przy kolejnej iteracji oszczędzić pamięć i czas.

```
BigNum multiply (BigNum& a, BigNum& b)
{
    BigNum temp(a.size+b.size);
    multiply_as(a.tab, a.size, b.tab, b.size, temp.tab);  //asm fun
    int i=temp.size-1;
    while(temp[i]==0) i--;
    temp.size = i+1;
    return temp;
}
```

Funkcja multiply_as zaimplementowana w języku x86:

```
.data
_a_size:      .zero 4
_b_size:      .zero 4
_result_size: .zero 4

.global multiply_as
.text
multiply_as:
push %ebp
mov  %esp, %ebp
subl $20, %esp

movl 12(%ebp), %eax
mov $4, %edx
mull %edx
movl %eax, _a_size

movl 20(%ebp), %eax
mov $4, %edx
mull %edx
movl %eax, _b_size

movl _a_size, %eax
addl _b_size, %eax
movl %eax, _result_size

xor %ebx, %ebx

loop_outer:                    # outer loop
    movl 8(%ebp), %eax         # loads first 4-byte part   INPUT_A
    addl %ebx, %eax
    mov (%eax), %eax
```

```
    xor %ecx, %ecx              # clears index for second loop

loop_inner:                     # inner loop
               pushl %ecx                      # pushes register values
                                               # on the stack
               pushl %ebx                      # for later use
    pushl %eax

               xor  %edx, %edx                 # clears edx register

    movl 16(%ebp), %edx
    addl %ecx, %edx
    mov (%edx), %edx

    mull %edx

    addl %ecx, %ebx                     # calculates index for second number
               clc                      # clearing CF flag
    movl 24(%ebp), %ecx
    addl %ebx, %ecx
    addl %eax, (%ecx)
    #addl %eax, BUFFER(,%ebx,1)         # low part of the product
    pushf
    addl $4, %ecx                       # can increment because
                                        # value is on the stack
    popf                                # adcl %edx, BUFFER(,%ebx,1)
                                        # high part of the product
    adcl %edx, (%ecx)
    jnc no_carry

   carry:                                      # recursive carry handling
               clc
               addl $4, %ecx
               addl $1, (%ecx)
               jc carry

no_carry:
               popl %eax
               popl %ebx                       # pops register values
               popl %ecx
               addl $4, %ecx
               cmp _b_size, %ecx
               jl loop_inner                   # condition for inner loop

  addl $4, %ebx
        cmp _a_size, %ebx
        jl loop_outer                          # condition for outer loop

exit:
clr %eax

addl $20, %esp
popl %ebp
ret
```

3. Testy

3.1 Założenia

Testy miały na celu wyznaczenie złożoności obliczeniowej algorytmu. Teoretyczna oczekiwana złożoność algorytmu szybkiego potęgowania to O(log n), natomiast biorąc pod uwagę nieoptymalny algorytm mnożenia $O(n^2)$, spodziewana ostateczna złożoność to $O(n^2)$. Dane wejściowe są generowane losowo, o długości zadanej przy uruchamianiu skryptu testowego.

3.2 Plan testów

Procedura testu:

1. Skompilowanie programy do wersji testowej za pomocą komendy:

```
$ make test
```

2. Wygenerowanie pliku wsadowego za pomocą skryptu „test.py”, dane wejściowe: n - ilość bajtów wygenerowanej losowo podstawy, k – wykładnik potęgi. Po wprowadzeniu danych skrypt generuje dwa pliki:
   a. data_input – plik z danymi wsadowymi do programu głównego,
   b. python_result – plik z referencyjnymi wynikami odpowiadającymi wygenerowanym danym.
3. Uruchomienie programu głównego w wersji testowej - „OiAK_Potegowanie_TEST”, program generuje dwa pliki:
   a. times_n_k – plik z pomiarami czasu dla każdej iteracji, gdzie n i k to parametry podane przy generowaniu danych skryptem „test.py”
   b. result_cpp – plik z wynikami obliczeń, zawiera wyniki w reprezentacji heksadecymalnej, oddzielone znakiem nowej linii.
4. Porównanie wyników otrzymanych w pliku wynikowym „result_cpp”, z wynikami referencyjnymi z pliku „result_python” komendą:

```
$ diff result_cpp result_python
```

5. Opracowanie wyników. Ewentualne odrzucenie błędów grubych, uruchomienie skryptu „plot.py” w celu zrzutowania ich na wykres.

Zostało wybrane pięć rozmiarów liczb wejściowych:

- podstawa: 16 B, wykładnik: 1024,

- podstawa: 32 B, wykładnik: 1024,

- podstawa: 512 B, wykładnik: 255,

- podstawa: 1024 B, wykładnik: 255,

- podstawa: 10240 B, wykładnik: 60.

3.3 Implementacja

Plik „main_test.cpp” to zmodyfikowany na potrzeby testów plik „main.cpp”. Zawiera on dwie zagnieżdżone pętle: jedną iterującą po parach liczb zawartych w pliku wejściowym data_input, drugą powtarzającą pomiar na jednym zestawie danych dziesięć razy w celu uśrednienia wyników.

```
for(int i=0;i<k;i++){
BigNum result("1");
    data_file>>a;
    data_file>>std::hex>>b;
```

```
 for(int m=0;m<10;m++)
       {
    t1 = std::chrono::high_resolution_clock::now();

    while( true )
    {
        if( b & 1 )    result = multiply ( result, a );
        if( b >>= 1 ) a = multiply ( a, a ); else break;
    }
      t2 = std::chrono::high_resolution_clock::now();
      std::chrono::duration<double> time_span =
std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
      time_avg += time_span.count();
      }
      result_file<<result;
      time_file<<time_avg/10<<std::endl;
```

Skrypt „test.py” używa modułu secrets do generowania ciągu znaków reprezentującego liczbę heksadecymalną o zadanej długości bajtów.

```
data_file.write("%d\n%d\n"%(k,n))

for i in range(0,k):

    string = (secrets.token_hex(n))

    if(len(string)>0):
        number = int(string,16)
        data_file.write(string+'\n'+hex(i)+'\n')
          result_file.write("%x\n" %number**i)

      else:
          number = 0
          data_file.write('0\n'+hex(i)+'\n')
          result_file.write("%x\n" %number**i)
```

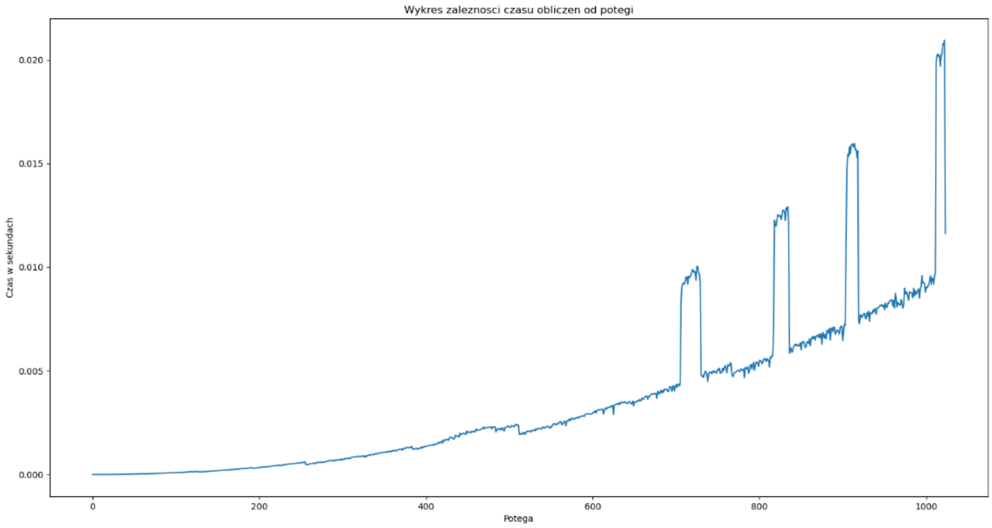
3.4 Metodologia pomiarów

Pomiary były przeprowadzane pod nieobciążonym systemem. Każdy pomiar jest uśrednioną wartością trzech pomiarów na tych samych danych. Okresowe skoki czasu wykonywania obliczeń zostały potraktowane jako błędy grube, a hipoteza na temat ich źródła została zawarta we wnioskach.

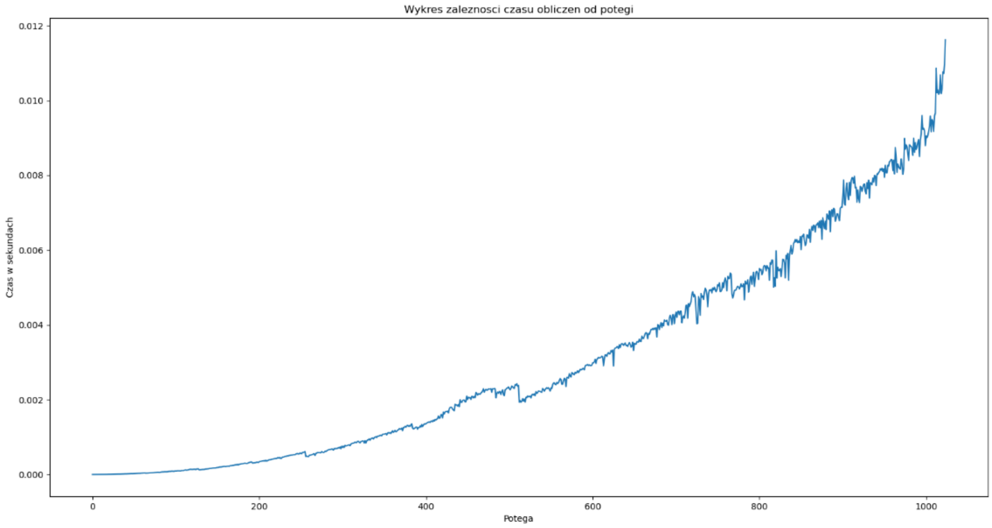
Procesor na stacji pomiarowej: Intel core i5-8250u, 1,8 – 3,4 GHz

3.5 Wyniki pomiarów:

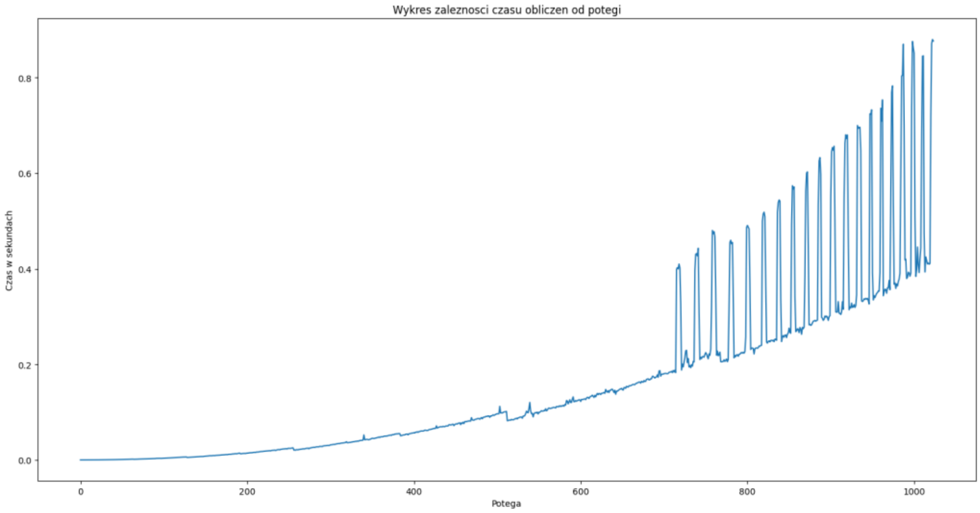
Dane wsadowe: podstawa 16 B, wykładnik = 1024



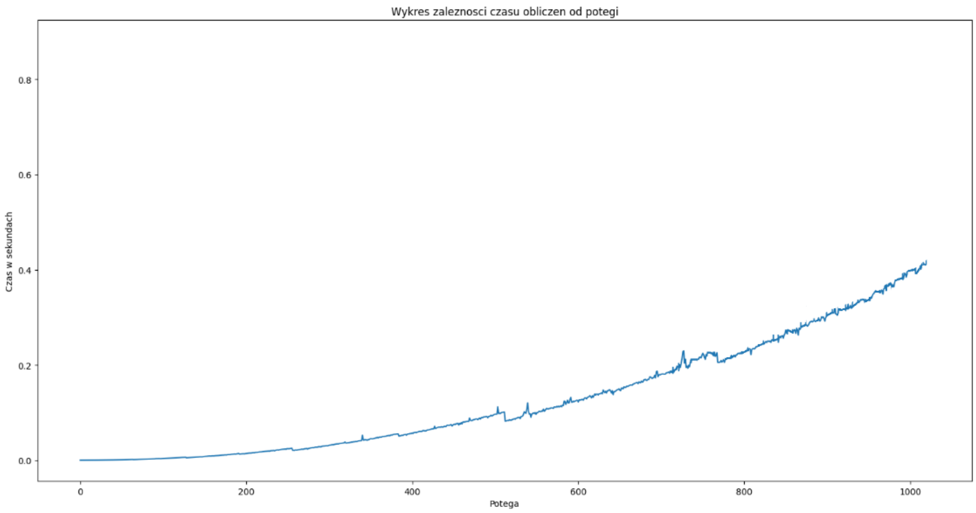
Po zniwelowaniu błędów grubych:



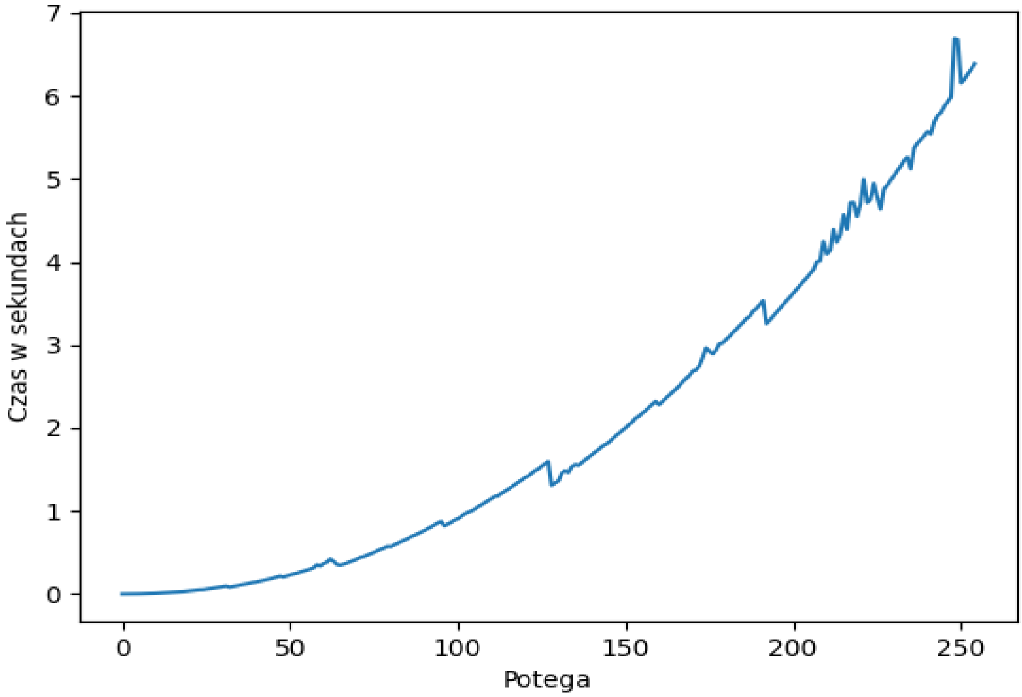
Dane wsadowe: podstawa 32 B, wykładnik = 1024



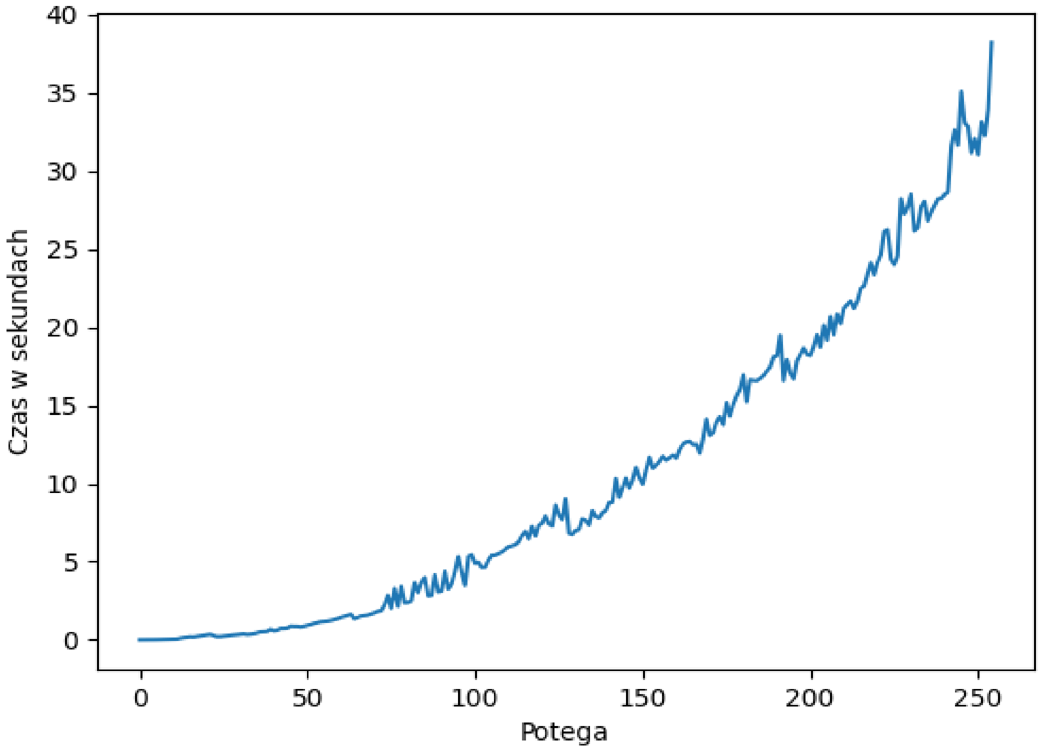
Po zniwelowaniu błędów grubych:



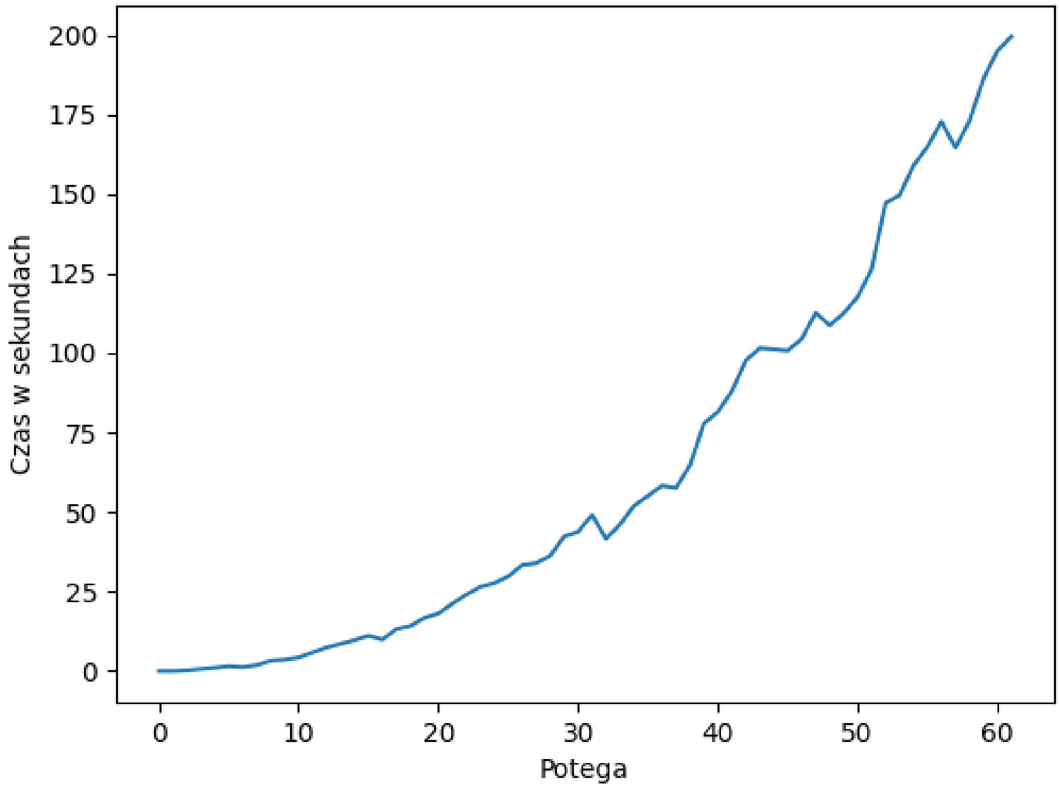
Dane wsadowe: podstawa 512 B, wykładnik = 255



Dane wsadowe: podstawa 1024 B, wykładnik = 255



Dane wsadowe: podstawa 10240 B, wykładnik = 60

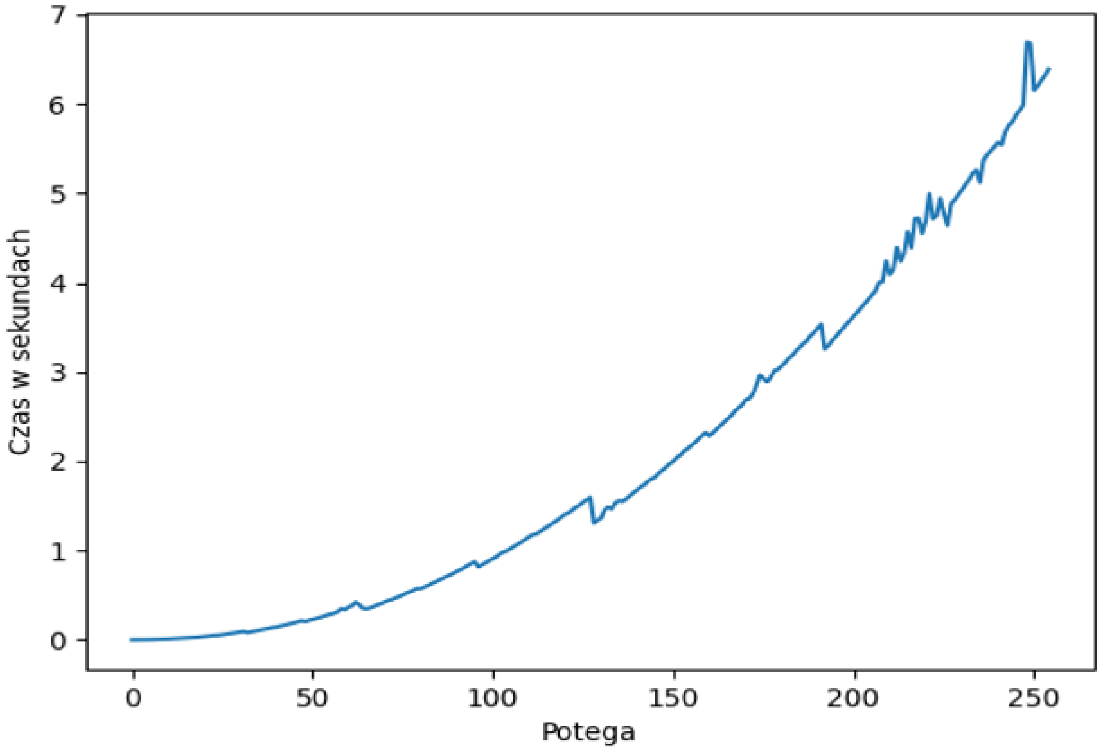


Czas wykonywania obliczenia n^k					
k / n	16	32	512	1024	10240
60	3.51951e-05	0.00134317	0.337464	1.34375	186.206
255	0.000599534	0.0248618	6.38716	38.2453	-
1024	0.0116251	0.876686	-	-	-

3.6 Wnioski

Zgodnie z przewidywaniami ostateczna złożoność algorytmu wynosi $O(n^2)$. Wynika to najprawdopodobniej z nieoptymalnego algorytmu mnożenia, bądź też z narzutów czasowych związanych z nieoptymalnym zastosowaniem struktur danych w języku C++.
Za źródło nienaturalnych wzrostów wyników można winić dławienie termiczne procesora. Proces pomiarowy zużywał 12-13% zasobów procesora, więc wysycenie na jednym z ośmiu rdzeni sięgało 100%. Procesor po osiągnięciu temperatur granicznej zmniejszał taktowanie, co skutkowało zmniejszeniem osiągów i w konsekwencji gorszymi wynikami.

Ciekawe zjawisko można dostrzec na przy wykładnikach o kolejnych potęgach dwójki. Dla przykłady wykres: podstawa 512B, wykładnik 255



Na kolejnych potęgach dwójki można zauważyć „stopnie". Wynika to z natury algorytmu. Na przykładzie gdy k =7, aby obliczyć a^k należy wykonać 5 mnożeń:

$$r = 1 * a = a$$

$$a = a * a = a^2$$

$$r = a * a^2 = a^3$$

$$a = a^2 * a^2 = a^4$$

$$r = a^3 * a^4 = a^7$$

Natomiast gdy k=8, aby obliczyć a^2 należy wykonać 4 mnożenia:

$$a = a * a = a^2$$

$$a = a^2 * a^2 = a^4$$

$$a = a^4 * a^4 = a^8$$

$$r = 1 * a^8 = a^8$$

Instrukcja kompilacji i uruchomienia programu znajduje się w pliku README.