Algorytmy macierzowe
Laboratorium 1 - Mnożenie macierzy

Kacper Wąchała, Szymon Hołysz

1. Opis zastosowanych algorytmów

1.1. Algorytm naiwny (metoda Bineta)

Algorytm polega na podziale dwóch kwadratowych macierzy A i B na cztery równe bloki, rekurencyjnym obliczeniu ich iloczynów, a następnie złączeniu wyniki w nową macierz C.

Poniżej przedstawiono pseudokod algorytmu:

```
def recursive_binet(A, B):
    n = rozmiar(A)
    jeśli n == 1:
        zwróć A * B

    podziel A na rowne bloki A11, A12, A21, A22
    podziel B na rowne bloki B11, B12, B21, B22

    C11 = recursive_binet(A11, B11) + recursive_binet(A12, B21)
    C12 = recursive_binet(A11, B12) + recursive_binet(A12, B22)
    C21 = recursive_binet(A21, B11) + recursive_binet(A22, B21)
    C22 = recursive_binet(A21, B12) + recursive_binet(A22, B22)

    zwróć macierz [[C11, C12], [C21, C22]]
```

1.2. Algorytm Strassena

Algorytm Strassena, podobnie jak poprzedni algorytm, polega na podziale kwadratowych macierzy A i B na cztery równe bloki, z tą różnicą, że następnie oblicza się przy ich użyciu 7 macierzy pomocniczych, a wynikową macierz C otrzymuje się poprzez odpowiednie dodawanie i mnożenie tych macierzy.

Poniżej przedstawiono pseudokod algorytmu:

```
def strassen(A, B):
    n = rozmiar(A)
    jeśli n == 1:
        zwróć A * B

    podziel A na rowne bloki A11, A12, A21, A22
    podziel B na rowne bloki B11, B12, B21, B22

    M1 = strassen(A11 + A22, B11 + B22)
    M2 = strassen(A21 + A22, B11)
    M3 = strassen(A11, B12 - B22)
    M4 = strassen(A22, B21 - B11)
    M5 = strassen(A11 + A12, B22)
    M6 = strassen(A21 - A11, B11 + B12)
    M7 = strassen(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6
```

```
    zwróć macierz [[C11, C12], [C21, C22]]
```

1.3. Metody generowane przy użyciu AI

Jest to zbiór metod szybkiego mnożenia macierzy o różnych rozmiarach, wyprowadzonych przez AlphaTensor, który wygenerował je korzystając z dekompozycji trójwymiarowych tensorów, umożliwiających automatyczne odkrywanie algorytmów o mniejszej liczbie mnożeń niż tradycyjne metody, w tym metoda Strassena.

2. Implementacja

W celu realizacji obliczeń na macierzach została zaimplementowana klasa `Matrix` zawierająca metody elementarnych działań i porównywania macierzy, oraz metody mnożeń przedstawione poniżej.

2.1. Metoda `recursive_binet`

```python
def recursive_binet(self, b:Matrix) -> Matrix:
        a = self
        rows, cols = a.shape
        assert rows == cols, "matrices must be square"
        assert a.shape == b.shape, "matrices must be the same shape"
        assert rows and (rows & rows - 1) == 0, "shape must be a power of 2"

        if rows == 1:
            return a.binet(b)

        p = rows // 2

        a11 = Matrix([n[:p] for n in self[:p]])
        a12 = Matrix([n[p:] for n in self[:p]])
        a21 = Matrix([n[:p] for n in self[p:]])
        a22 = Matrix([n[p:] for n in self[p:]])

        b11 = Matrix([n[:p] for n in b[:p]])
        b12 = Matrix([n[p:] for n in b[:p]])
        b21 = Matrix([n[:p] for n in b[p:]])
        b22 = Matrix([n[p:] for n in b[p:]])

        c11 = (a11.recursive_binet(b11)).__addnocount__(a12.recursive_binet(b21))
        c12 = (a11.recursive_binet(b12)).__addnocount__(a12.recursive_binet(b22))
        c21 = (a21.recursive_binet(b11)).__addnocount__(a22.recursive_binet(b21))
        c22 = (a21.recursive_binet(b12)).__addnocount__(a22.recursive_binet(b22))

        return Matrix.block([[c11, c12], [c21, c22]])
```

2.2. Metoda `strassen`

```
def strassen(self, b: Matrix) -> Matrix:
        rows, cols = self.shape

        assert rows == cols, "matrices must be square"
        assert self.shape == b.shape, "matrices must be the same shape"
        assert rows and (rows & rows - 1) == 0, "shape must be a power of 2"

        if rows == 1:
            return self.binet(b)

        p = rows // 2

        a11 = Matrix([n[:p] for n in self[:p]])
        a12 = Matrix([n[p:] for n in self[:p]])
        a21 = Matrix([n[:p] for n in self[p:]])
        a22 = Matrix([n[p:] for n in self[p:]])

        b11 = Matrix([n[:p] for n in b[:p]])
        b12 = Matrix([n[p:] for n in b[:p]])
        b21 = Matrix([n[:p] for n in b[p:]])
        b22 = Matrix([n[p:] for n in b[p:]])

        m1 = (a11 + a22).strassen(b11 + b22)
        m2 = (a21 + a22).strassen(b11)
        m3 = a11.strassen(b12 - b22)
        m4 = a22.strassen(b21 - b11)
        m5 = (a11 + a12).strassen(b22)
        m6 = (a21 - a11).strassen(b11 + b12)
        m7 = (a12 - a22).strassen(b21 + b22)

        c11 = m1 + m4 - m5 + m7
        c12 = m3 + m5
        c21 = m2 + m4
        c22 = m1 - m2 + m3 + m6

        return Matrix.block([[c11, c12], [c21, c22]])
```

2.3. Metoda `ai`

Metoda `ai` służy do zaimportowania i wywołania jednej z funkcji z modułu `generated_multiplications`. Zawiera on algorytmy opracowane przez AlphaTensor wygenerowane z dekompozycji tensorów mnożenia macierzy dostępnych w repozytorium dołączonym do omawianego artykułu[1].

```
    def ai(self, B:Matrix) -> Matrix:
        A = self
        assert A.shape[1] == B.shape[0]

        package_name = ("generated_multiplications.m"
                        + "_".join([str(A.shape[0]),
                                    str(B.shape[0]),
                                    str(B.shape[1])])
                        + "_generated")
        module = importlib.import_module(package_name)
```

Dekompozycje tensorów postaci $T = \sum_i (u_i \otimes v_i \otimes w_i)$ przechowywane są w plikach `.npy`. Można z nich odczytać macierze U, V, W; a następnie odczytując z nich niezerowe wartości wygenerować kolejne składniki kombinacji liniowych.

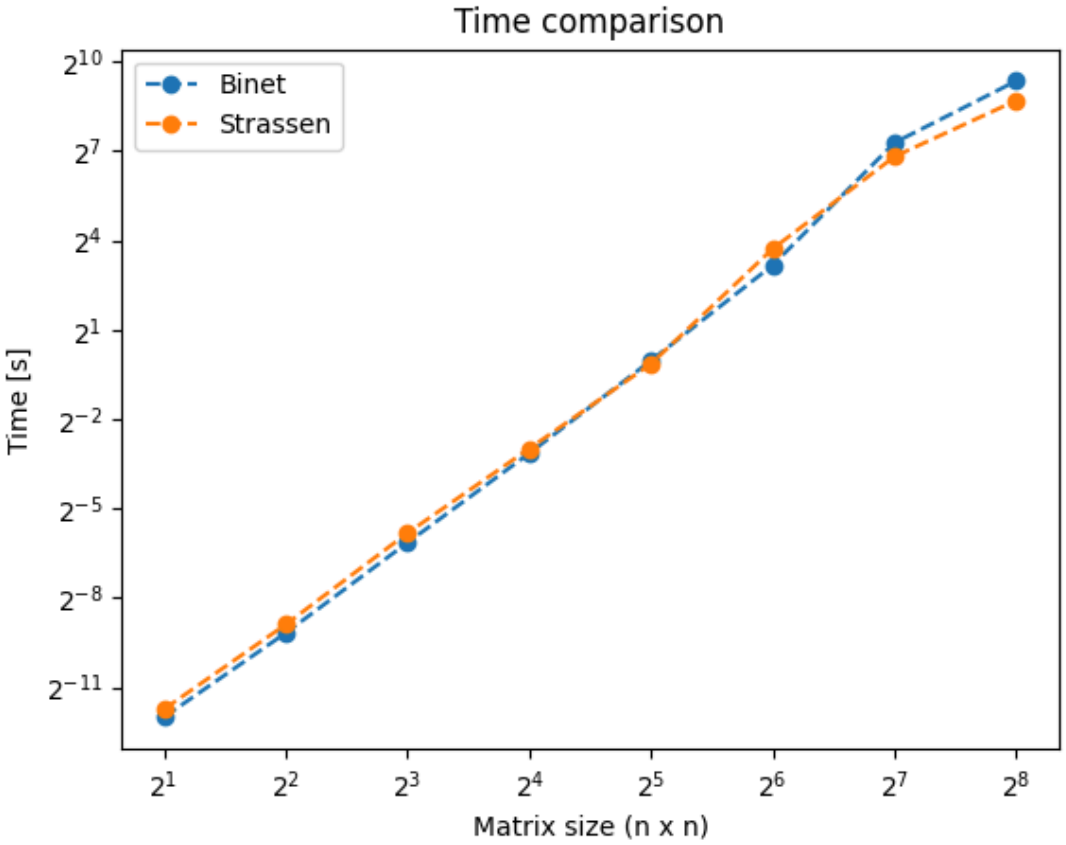
Przykładowy kod wygenerowany powyższym sposobem służący do mnożenia macierzy kwadratowej rozmiaru 2 (równoważny algorytmowi Strassena):

```
# A: 2×2, B: 2×2, C: 2×2
# Using 7 multiplications

from multiply import Matrix

def multiply(A, B):
  C = Matrix([[0 for _ in range(2)] for _ in range(2)])
  M0 = (A[1][0] + -A[1][1]) * B[0][1]
  M1 = (A[0][0] + A[1][0] + -A[1][1]) * (B[0][1] + B[1][0] + B[1][1])
  M2 = (A[0][0] + -A[0][1] + A[1][0] + -A[1][1]) * (B[1][0] + B[1][1])
  M3 = A[0][1] * B[1][0]
  M4 = (A[0][0] + A[1][0]) * (B[0][0] + B[0][1] + B[1][0] + B[1][1])
  M5 = A[0][0] * B[0][0]
  M6 = A[1][1] * (B[0][1] + B[1][1])

  C[0][0] = M3 + M5
  C[1][0] = -M1 + M4 + -M5 + -M6
  C[0][1] = -M0 + M1 + -M2 + -M3
  C[1][1] = M0 + M6
  return (C, 7)
```

3. Wyniki

3.1. Porównanie czasów wykonania

Aby zbadać efektywność zaimplementowanych metod rekurencyjnych, zmierzono czasy ich wykonania dla rozmiarów macierzy 2^n, $n = 1, 2, 3, \ldots, 8$. Poniżej przedstawiono wykres porównania czasów wykonania (przyjęto skalę logarytmiczną na obu osiach):

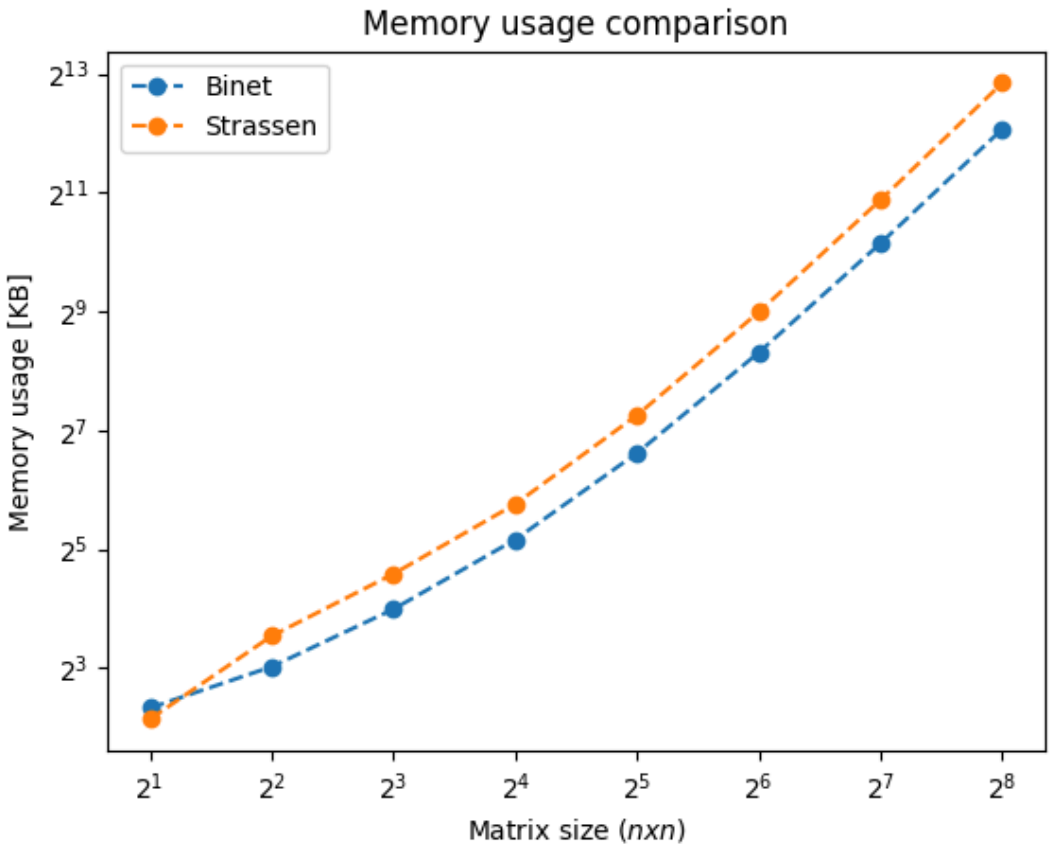


Rysunek 1: Porównanie czasu wykonania algorytmów rekurencyjnych mnożenia macierzy

Na wykresie można zauważyć, że obie z rozpatrywanych metod mają bardzo zbliżone do siebie czasy wykonania i ciężko jest zauważyć znaczące różnice między nimi. Można natomiast oszacować ze skali, że współczynnik nachylenia obu wykresów wynosi ≈ 3, co oznacza, że złożoności obu algorytmów są w przybliżeniu rzędu $O(n^3)$.

3.2. Porównanie zużycia pamięci

Następnie dokonano porównania zużycia pamięci przez oba algorytmy. Poniżej przedstawiono uzyskane wyniki (przyjęto skalę logarytmiczną na obu osiach):

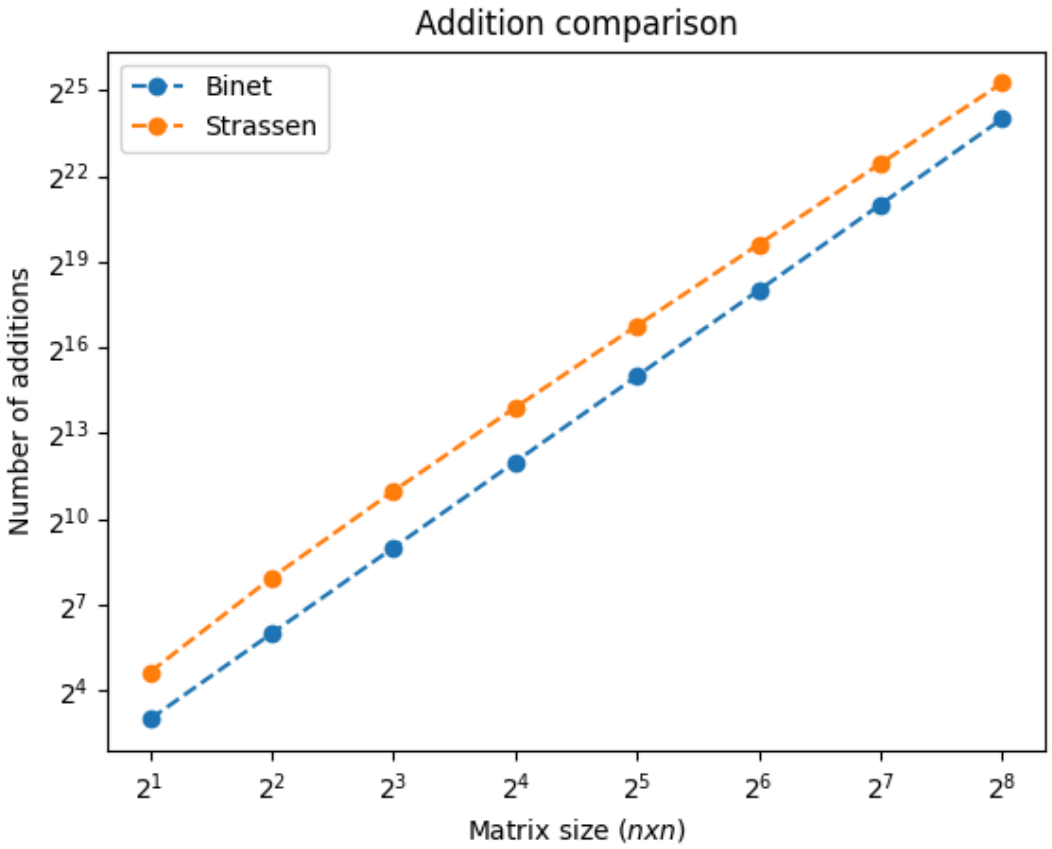


Rysunek 2: Porównanie zużycia pamięci algorytmów rekurencyjnych mnożenia macierzy

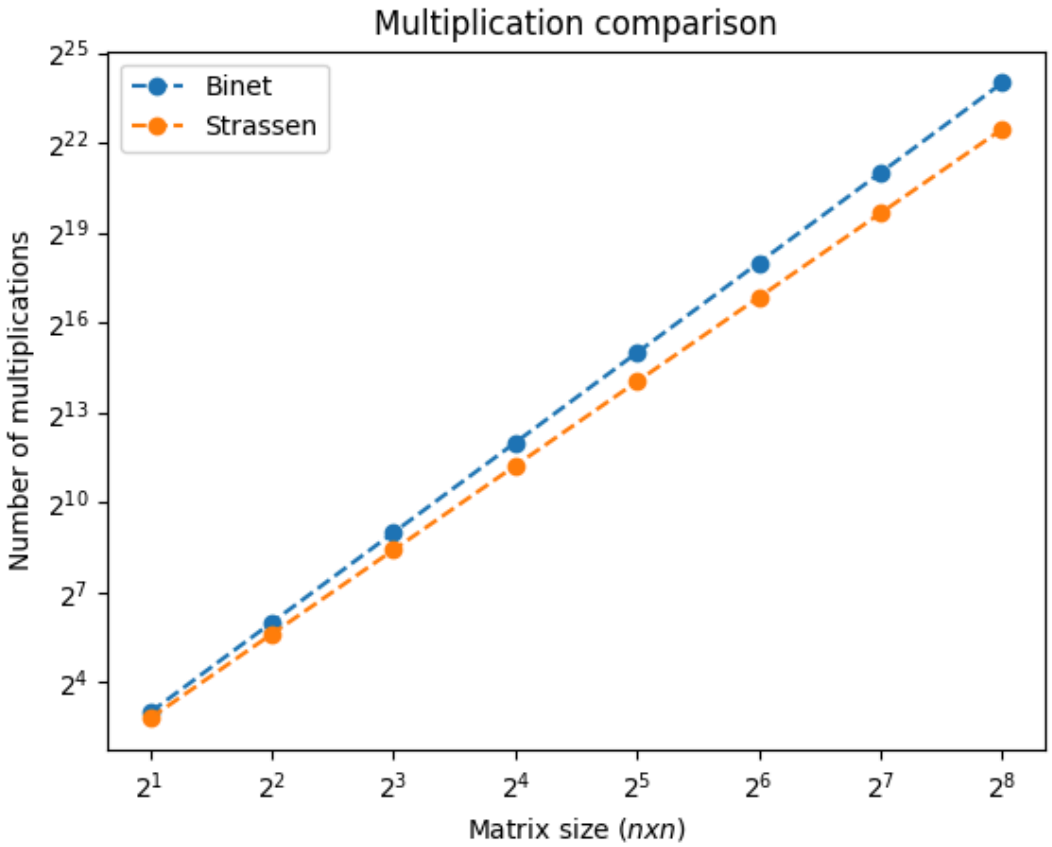
Można zaobserwować, że algorytm Strassena zużywa więcej pamięci niż metoda Bineta. Wykresy dla tych algorytmów są natomiast równoległe, co oznacza że zużycie pamięci rośnie względem rozmiaru macierzy w przybliżeniu w tym samym tempie (ich złożoność pamięciowa jest tego samego rzędu), a różnica w położeniu linii wynika ze stałego czynnika.

3.3. Porównanie liczby wykonanych operacji zmiennoprzecinkowych

Porównano również liczbę wykonanych operacji w obu algorytmach (dodawanie i mnożenie). Poniżej przedstawiono wykres (przyjęto skalę logarytmiczną na obu osiach):



Rysunek 3: Porównanie liczby dodawań algorytmów rekurencyjnych mnożenia macierzy

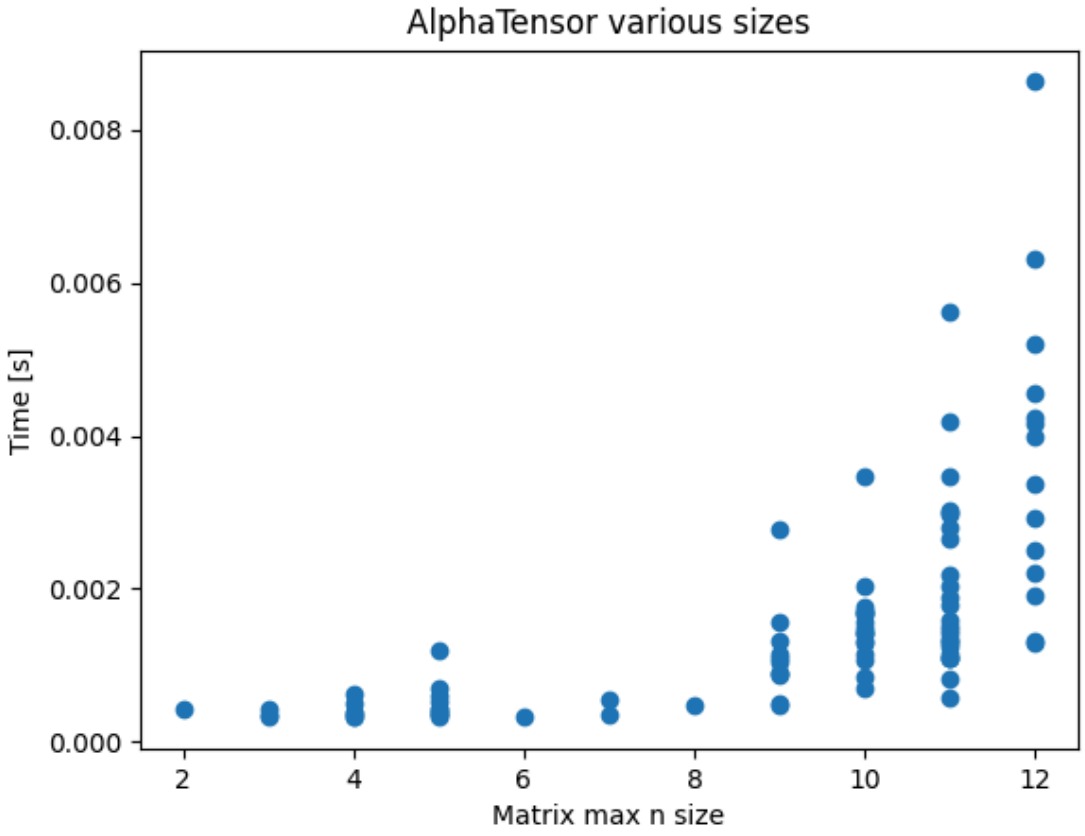


Rysunek 4: Porównanie liczby mnożeń algorytmów rekurencyjnych mnożenia macierzy

Można dostrzec, że algorytm Strassena kosztem zwiększenia liczby dodawań, ma mniejszą liczbę mnożeń w trakcie wykonania, co przyczynia się do tego, że jego złożoność teoretyczna jest mniejszego rzędu niż dla metody Bineta. Wykresy zarówno liczby dodawań jak i mnożeń dla algorytmu Strassena charakteryzują się też mniejszym nachyleniem, co niejako jest potwierdzeniem teorii, że algorytm ten ma lepszą złożoność obliczeniową niż metoda Bineta ($O(n^{2.81})$ w porównaniu do $O(n^3)$).

3.4. Wyniki metod generowanych AI

Dla różnych metod wygenerowanych przez AI przeprowadzono porównanie czasu wykonania. Za oś poziomą przyjęto wartość maksymalną z wymiarów mnożonych przez siebie macierzy. Poniżej przedstawiono wykres:



Rysunek 5: Porównanie czasu wykonania dla algorytmów wygenerowanych przez AI

4. Wnioski

Wykonane ćwiczenia pozwoliły zapoznać się z dostępnymi metodami mnożenia macierzy oraz z możliwościami ulepszania algorytmów. Omawiane metody istotnie zmniejszają czas obliczeń, ale trzeba mieć na uwadze, że dostępne biblioteki wykonujące operacje na macierzach są wielokrotnie lepiej zoptymalizowane i to z nich (a nie z własnych implementacji) lepiej korzystać w profesjonalnych zastosowaniach.

Bibliografia

[1] A. Fawzi *i in.*, „Discovering faster matrix multiplication algorithms with reinforcement learning”, *Nature*, t. 610, nr 7930, s. 47–53, 2022, doi: 10.1038/s41586-022-05172-4.