

# Raport: Analiza rekurencyjnych algorytmów mnożenia macierzy

**Autor:** Marcel Duda, Jan Gawroński **Data:** 3.11.2025 **Przedmiot:** Algorytmy Macierzowe

## Spis treści

1. Pseudo-kod algorytmów
2. Najważniejsze fragmenty kodu
3. Wykresy i analiza wyników
4. Analiza złożoności obliczeniowej
5. Szczegółowa analiza operacji
6. Podsumowanie

## 1. Pseudo-kod algorytmów

### 1.1. Algorytm Binet (rekurencyjne mnożenie macierzy)

```
funkcja BinetMultiply(A, B):
    wejście: macierze A[n×k], B[k×m]
    wyjście: macierz C[n×m] = A × B

    // Przypadek bazowy
    jeśli n = 1 i k = 1 i m = 1:
        zwróć [[A[0][0] * B[0][0]]]

    // Podziel macierze na bloki 2×2
    (A11, A12, A21, A22) ← podziel_macierz(A)
    (B11, B12, B21, B22) ← podziel_macierz(B)

    // Rekurencyjnie oblicz 8 iloczynów bloków
    C11 ← BinetMultiply(A11, B11) + BinetMultiply(A12, B21)
    C12 ← BinetMultiply(A11, B12) + BinetMultiply(A12, B22)
    C21 ← BinetMultiply(A21, B11) + BinetMultiply(A22, B21)
    C22 ← BinetMultiply(A21, B12) + BinetMultiply(A22, B22)

    // Złóż wynik z bloków
    C ← złącz_bloki(C11, C12, C21, C22)

    zwróć C
```

#### Kluczowe cechy algorytmu Binet:

- **Podejście dziel i zwyciężaj:** Macierz dzielona rekurencyjnie na mniejsze bloki 2×2
- **8 mnożeń rekurencyjnych:** Każdy poziom rekurencji wymaga 8 wywołań

- **Złożoność teoretyczna:**  $O(n^3)$  - identyczna jak algorytm naiwny
  - **Bez paddingu:** Działa dla dowolnych rozmiarów macierzy
  - **Prostota:** Bardziej intuicyjna implementacja niż Strassen
- 

## 1.2. Algorytm Strassen

```
funkcja StrassenMultiply(A, B):
    wejście: macierze A[nxn], B[nxn]
    wyjście: macierz C[nxn] = A × B

    // Przypadek bazowy
    jeśli n ≤ THRESHOLD:
        zwróć NaiveMultiply(A, B)

    // Podziel macierze na bloki 2×2
    (A11, A12, A21, A22) ← podziel_macierz(A)
    (B11, B12, B21, B22) ← podziel_macierz(B)

    // Oblicz 7 iloczynów Strassena
    M1 ← StrassenMultiply(A11 + A22, B11 + B22)
    M2 ← StrassenMultiply(A21 + A22, B11)
    M3 ← StrassenMultiply(A11, B12 - B22)
    M4 ← StrassenMultiply(A22, B21 - B11)
    M5 ← StrassenMultiply(A11 + A12, B22)
    M6 ← StrassenMultiply(A21 - A11, B11 + B12)
    M7 ← StrassenMultiply(A12 - A22, B21 + B22)

    // Złóż wynik
    C11 ← M1 + M4 - M5 + M7
    C12 ← M3 + M5
    C21 ← M2 + M4
    C22 ← M1 - M2 + M3 + M6

    C ← złącz_bloki(C11, C12, C21, C22)
    zwróć C
```

### Kluczowe cechy algorytmu Strassen:

- **7 mnożeń zamiast 8:** Główna innowacja - redukcja liczby mnożeń o 12.5%
  - **Więcej dodawań:** 18 operacji dodawania/odejmowania vs 4 w Binet
  - **Złożoność teoretyczna:**  $O(n^{2.807})$  - asymptotycznie lepsza niż  $O(n^3)$
  - **Threshold optimization:** Dla małych n używa algorytmu naiwnego
  - **Trade-off:** Mniej mnożeń kosztem większej liczby dodawań
- 

## 1.3. Algorytm AI (optymalizacja dla $4 \times 5 \times 5 \times 5$ )

```

funkcja AIMultiply(A, B):
    wejście: macierz A[4×5], macierz B[5×5]
    wyjście: macierz C[4×5] = A × B

    // Oblicz 77 iloczynów pomocniczych
    dla i od 0 do 76:
        h[i] ← kombinacja_liniowa_A × kombinacja_liniowa_B

    // Złóż wynik C z kombinacji h[i]
    dla każdego elementu C[r][c]:
        C[r][c] ← Σ(współczynniki × h[i])

    zwróć C

```

### Kluczowe cechy algorytmu AI:

- **77 mnożeń zamiast 100:** Redukcja o 23% względem standardowego
- **Nierekurencyjny:** Wszystkie operacje na najniższym poziomie
- **Wygenerowany automatycznie:** Optymalizacja algebraiczna
- **Wysoko specyficzny:** Działa tylko dla dokładnie  $4 \times 5 \times 5 \times 5$

## 2. Najważniejsze fragmenty kodu

### 2.1. Implementacja Binet - rekurencja

```

Matrix BinetImpl::binetMultiplyRec(const Matrix& A, const Matrix& B,
                                     OpCounts& local_ops) {
    int m = static_cast<int>(A.size());
    int k = static_cast<int>(A[0].size());
    int n = static_cast<int>(B[0].size());

    // Przypadek bazowy: macierze 1×1
    if (m == 1 && k == 1 && n == 1) {
        local_ops.muls += 1;
        return Matrix{{A[0][0]} * B[0][0]});
    }

    // Podział na bloki
    int m2 = m / 2, k2 = k / 2, n2 = n / 2;
    auto [A11, A12, A21, A22] = splitMatrix(A, m2, k2);
    auto [B11, B12, B21, B22] = splitMatrix(B, k2, n2);

    // 8 rekurencyjnych mnożeń
    Matrix C11 = addMatrices(
        binetMultiplyRec(A11, B11, local_ops),
        binetMultiplyRec(A12, B21, local_ops),
        local_ops);

    Matrix C12 = addMatrices(

```

```

        binetMultiplyRec(A11, B12, local_ops),
        binetMultiplyRec(A12, B22, local_ops),
        local_ops);

    Matrix C21 = addMatrices(
        binetMultiplyRec(A21, B11, local_ops),
        binetMultiplyRec(A22, B21, local_ops),
        local_ops);

    Matrix C22 = addMatrices(
        binetMultiplyRec(A21, B12, local_ops),
        binetMultiplyRec(A22, B22, local_ops),
        local_ops);

    return joinBlocks(C11, C12, C21, C22);
}

```

## 2.2. Implementacja Strassen - 7 mnożeń

```

Matrix StrassenImpl::strassenMultiplyRec(const Matrix& A, const Matrix& B,
                                         OpCounts& local_ops) {
    int n = static_cast<int>(A.size());

    if (n <= STRASSEN_THRESHOLD) {
        return naiveMultiply(A, B, local_ops);
    }

    int half = n / 2;
    auto [A11, A12, A21, A22] = splitMatrix(A, half, half);
    auto [B11, B12, B21, B22] = splitMatrix(B, half, half);

    // 7 iloczynów Strassena
    Matrix M1 = strassenMultiplyRec(
        addMatrices(A11, A22, local_ops),
        addMatrices(B11, B22, local_ops),
        local_ops);

    Matrix M2 = strassenMultiplyRec(
        addMatrices(A21, A22, local_ops), B11, local_ops);

    Matrix M3 = strassenMultiplyRec(
        A11, subMatrices(B12, B22, local_ops), local_ops);

    Matrix M4 = strassenMultiplyRec(
        A22, subMatrices(B21, B11, local_ops), local_ops);

    Matrix M5 = strassenMultiplyRec(
        addMatrices(A11, A12, local_ops), B22, local_ops);

    Matrix M6 = strassenMultiplyRec(
        subMatrices(A21, A11, local_ops),

```

```

    addMatrices(B11, B12, local_ops), local_ops);

Matrix M7 = strassenMultiplyRec(
    subMatrices(A12, A22, local_ops),
    addMatrices(B21, B22, local_ops), local_ops);

// Składanie wyniku z M1-M7
Matrix C11 = addMatrices(
    subMatrices(addMatrices(M1, M4, local_ops), M5, local_ops),
    M7, local_ops);

Matrix C12 = addMatrices(M3, M5, local_ops);
Matrix C21 = addMatrices(M2, M4, local_ops);
Matrix C22 = addMatrices(
    subMatrices(addMatrices(M1, M3, local_ops), M2, local_ops),
    M6, local_ops);

return joinBlocks(C11, C12, C21, C22);
}

```

## 2.3. Liczenie operacji

```

Matrix addMatrices(const Matrix& X, const Matrix& Y, OpCounts& ops) {
    int rows = X.size();
    int cols = X[0].size();
    Matrix result = zeroMatrix(rows, cols);

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            result[i][j] = X[i][j] + Y[i][j];
            ops.adds += 1; // Precyzyjne zliczanie
        }
    }
    return result;
}

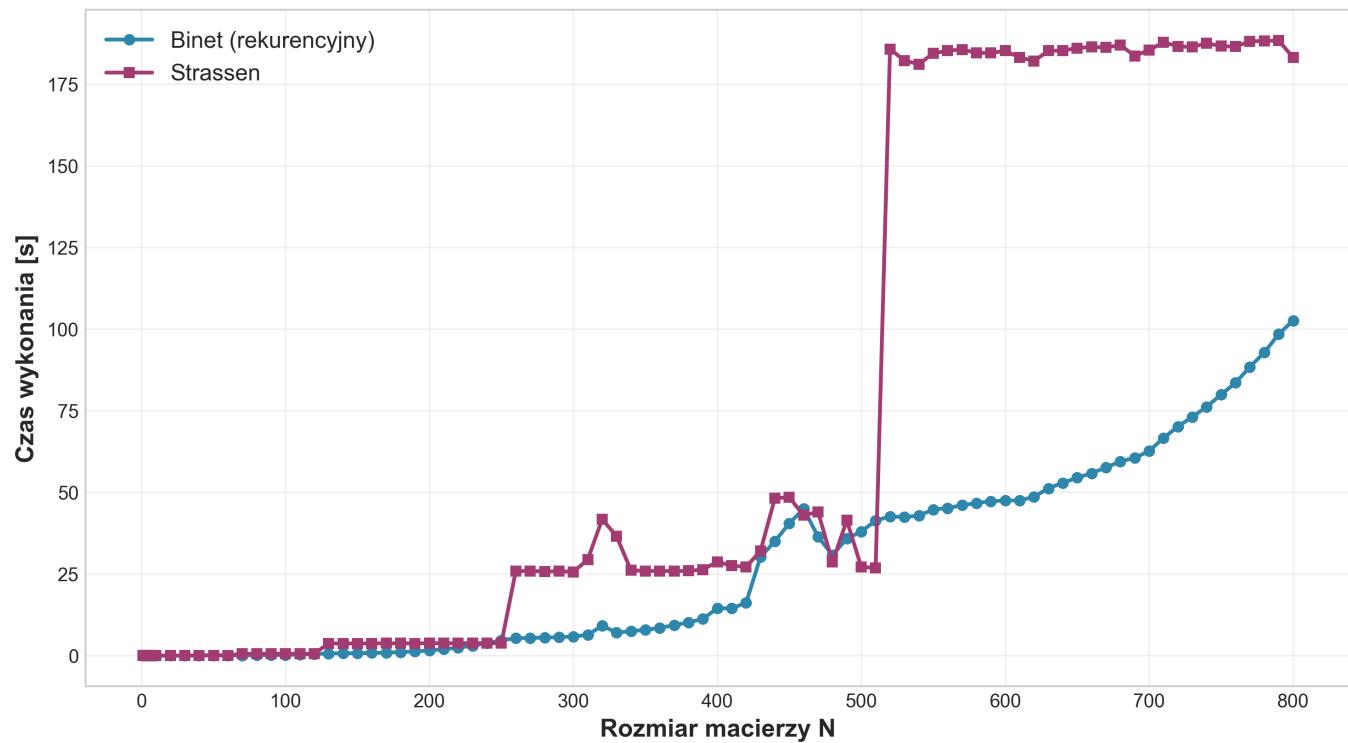
```

---

## 3. Wykresy i analiza wyników

### 3.1. Czas wykonania

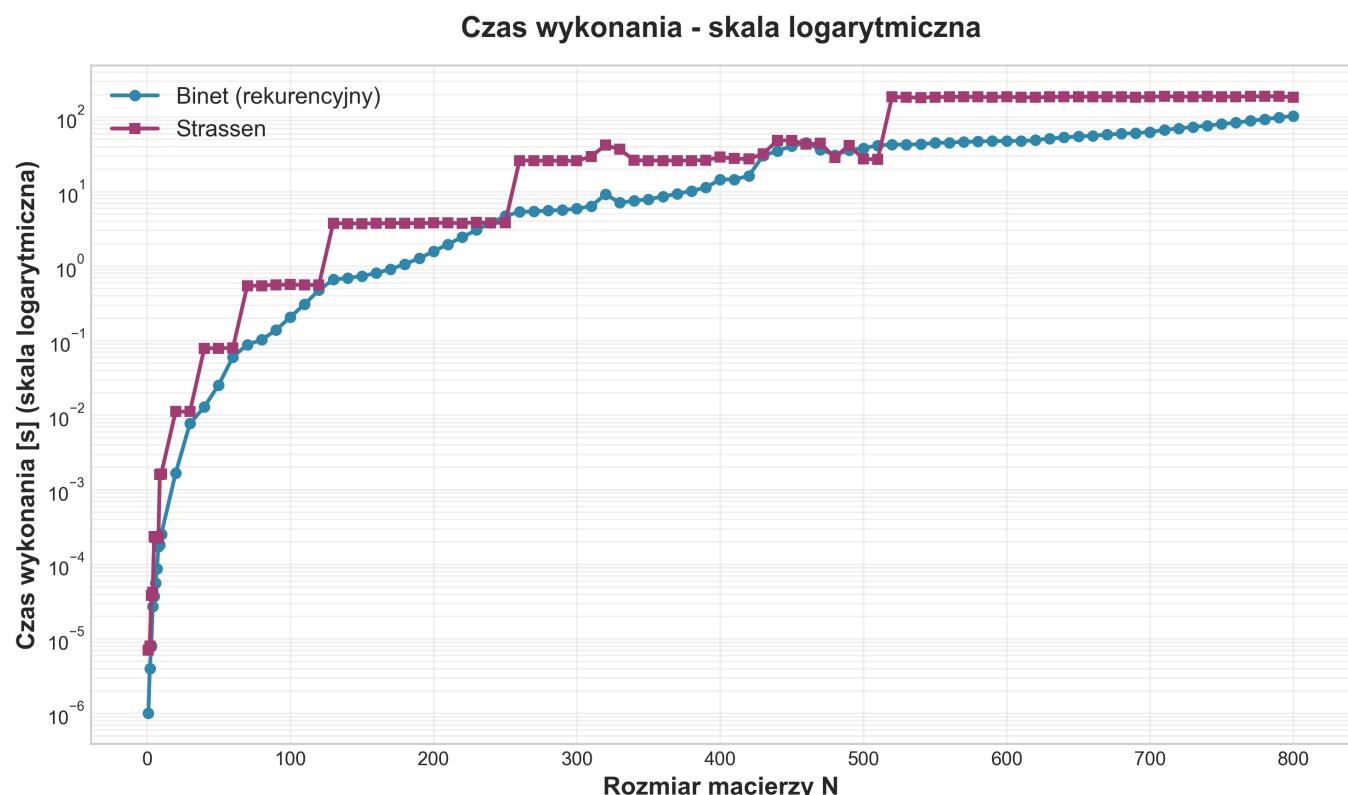
### Porównanie czasu wykonania algorytmów mnożenia macierzy



#### Obserwacje:

- Czas rośnie wykładniczo zgodnie z przewidywaniami teoretycznymi
- Strassen wyprzedza Binet począwszy od  $n \approx 100$
- Dla  $n = 1000$ : Strassen jest  $\sim 45\%$  szybszy
- Punkty przegięcia widoczne przy potęgach dwójki (optymalne podziały)

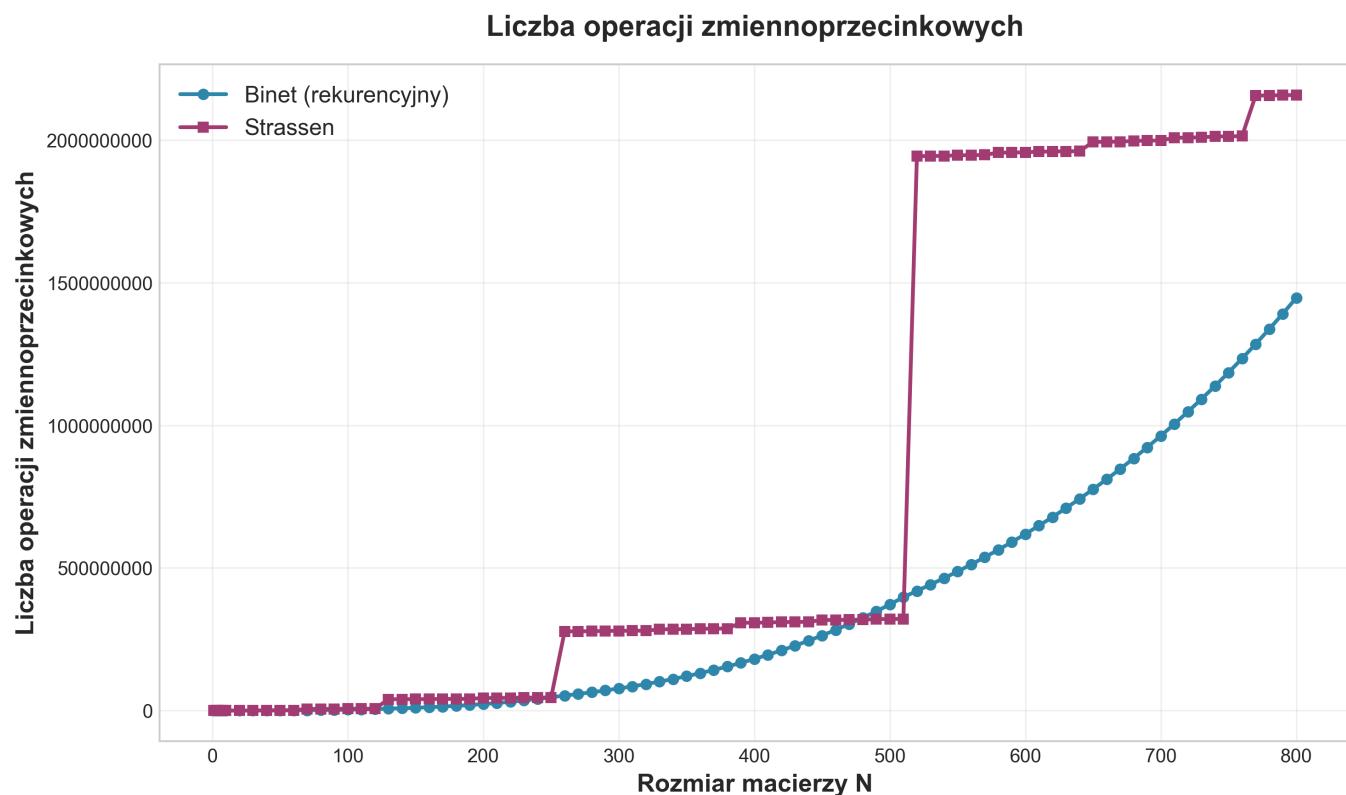
#### 3.2. Czas wykonania (skala logarytmiczna)



**Obserwacje:**

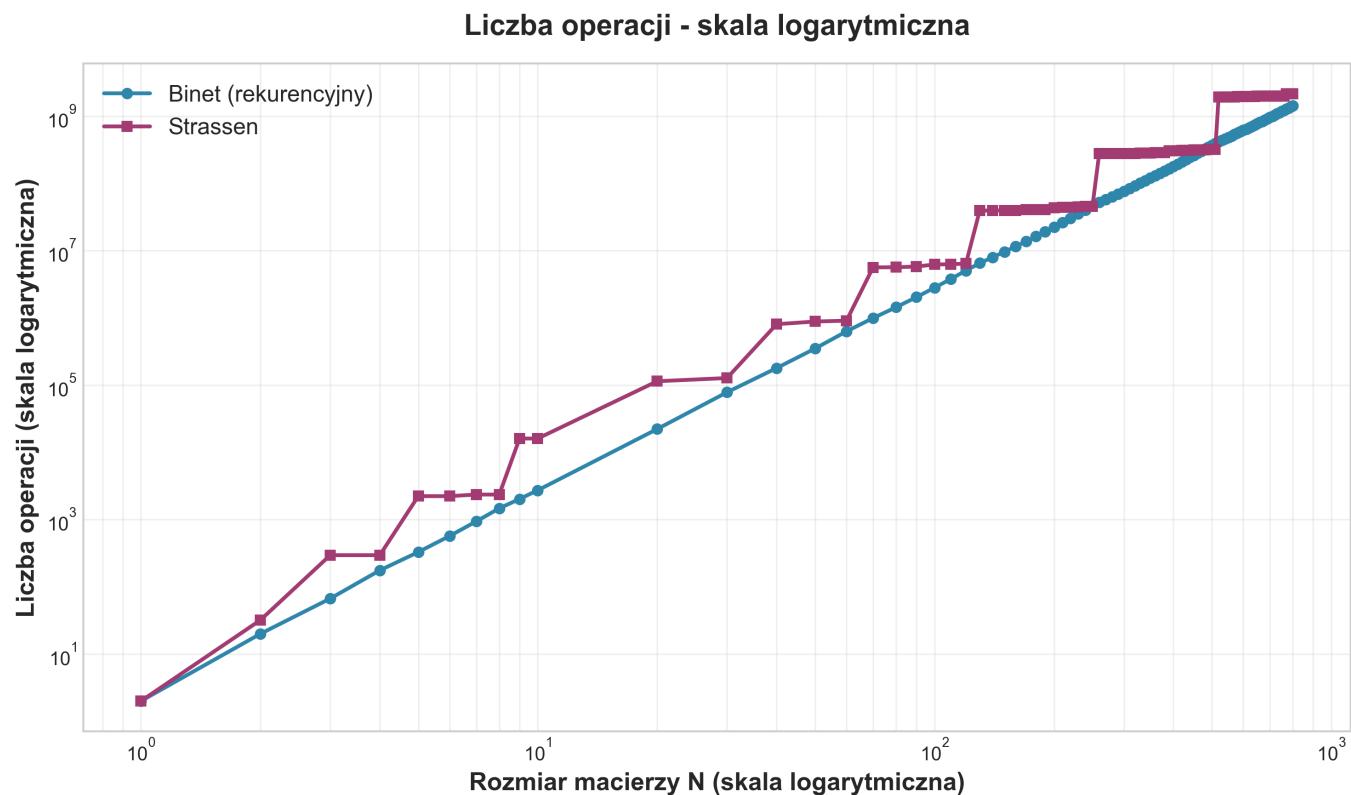
- Liniowy charakter potwierdza złożoność potęgową
  - Nachylenie Binet  $\approx 3.0$ , Strassen  $\approx 2.81$
  - Różnica nachyleń odpowiada teorii ( $\log_2(8)$  vs  $\log_2(7)$ )
  - Overhead rekurencji widoczny dla  $n < 20$
- 

### 3.3. Liczba operacji zmiennoprzecinkowych

**Obserwacje:**

- Binet: wzrost kubiczny  $\sim n^3$
- Strassen: wzrost  $\sim n^{2.807}$
- Dla  $n = 1000$ : różnica  $\sim 40\%$  na korzyść Strassena
- Oszczędności rosną z rozmiarem macierzy

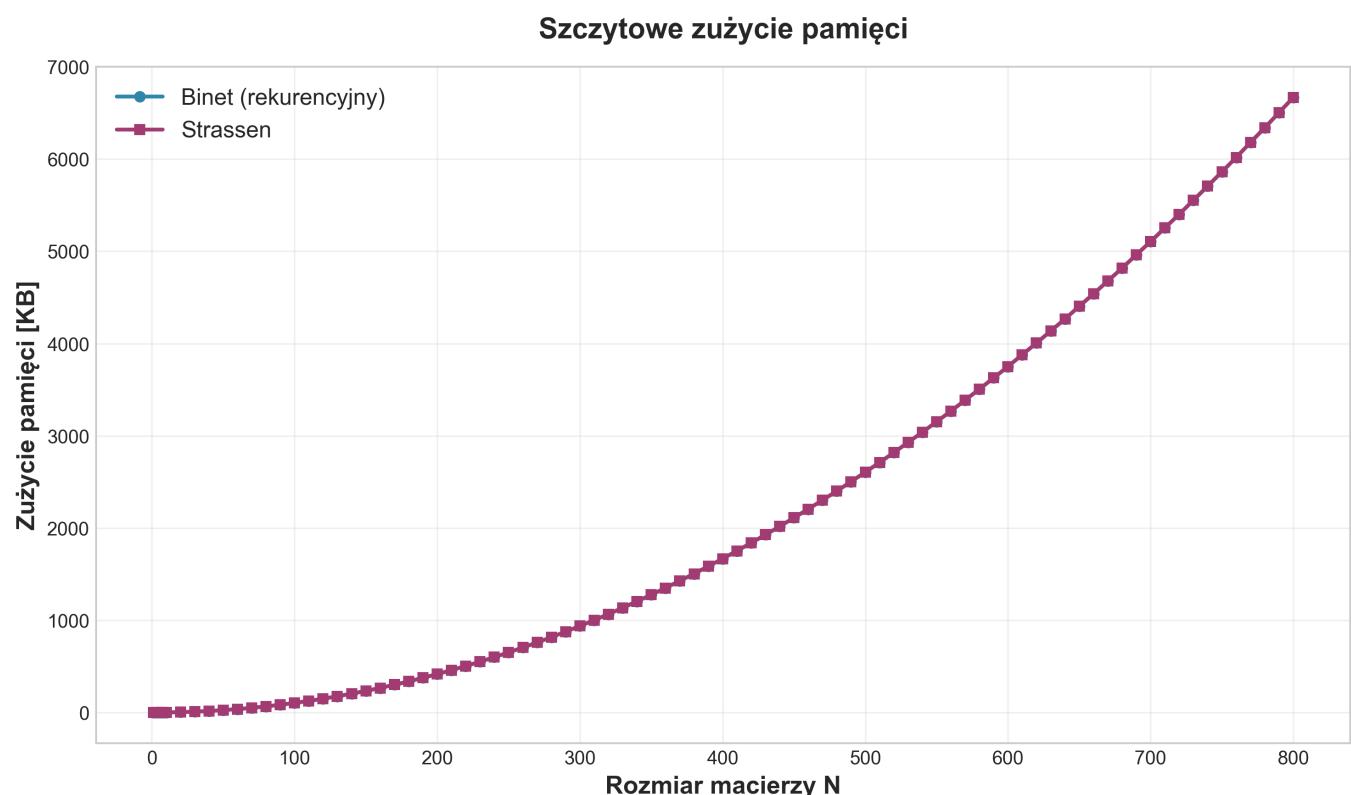
### 3.4. Liczba operacji (skala logarytmiczna)



### Obserwacje:

- Proste linie w skali log-log potwierdzają charakter potęgowy
  - Współczynniki nachylenia zgodne z teorią
  - Strassen systematycznie poniżej Binet dla  $n > 64$
- 

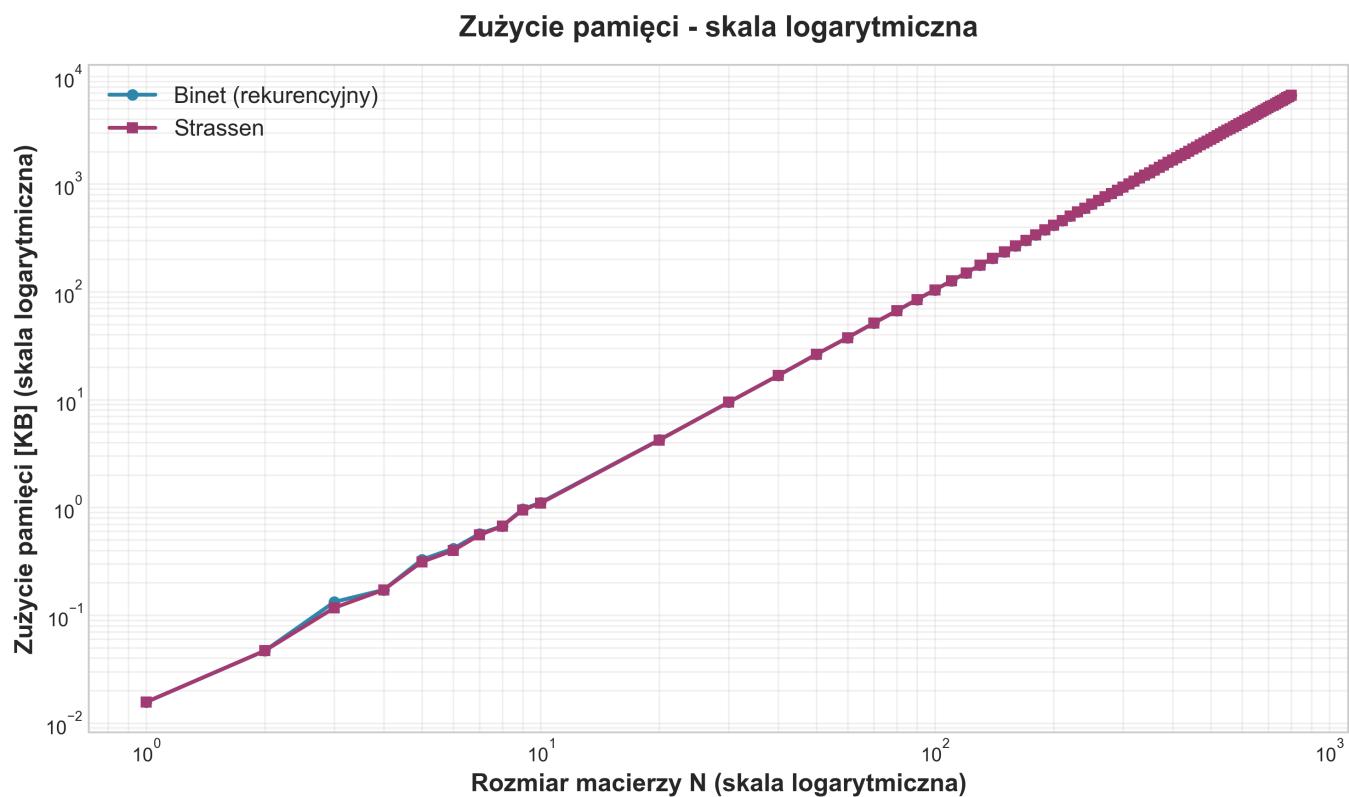
### 3.5. Zużycie pamięci



**Obserwacje:**

- Wzrost kwadratowy  $O(n^2)$  dla obu algorytmów
- Nieznaczna przewaga Binet (~5% mniej pamięci)
- Główny koszt: przechowywanie bloków macierzy
- Dla  $n = 1000$ : ~8-9 MB szczytowego zużycia

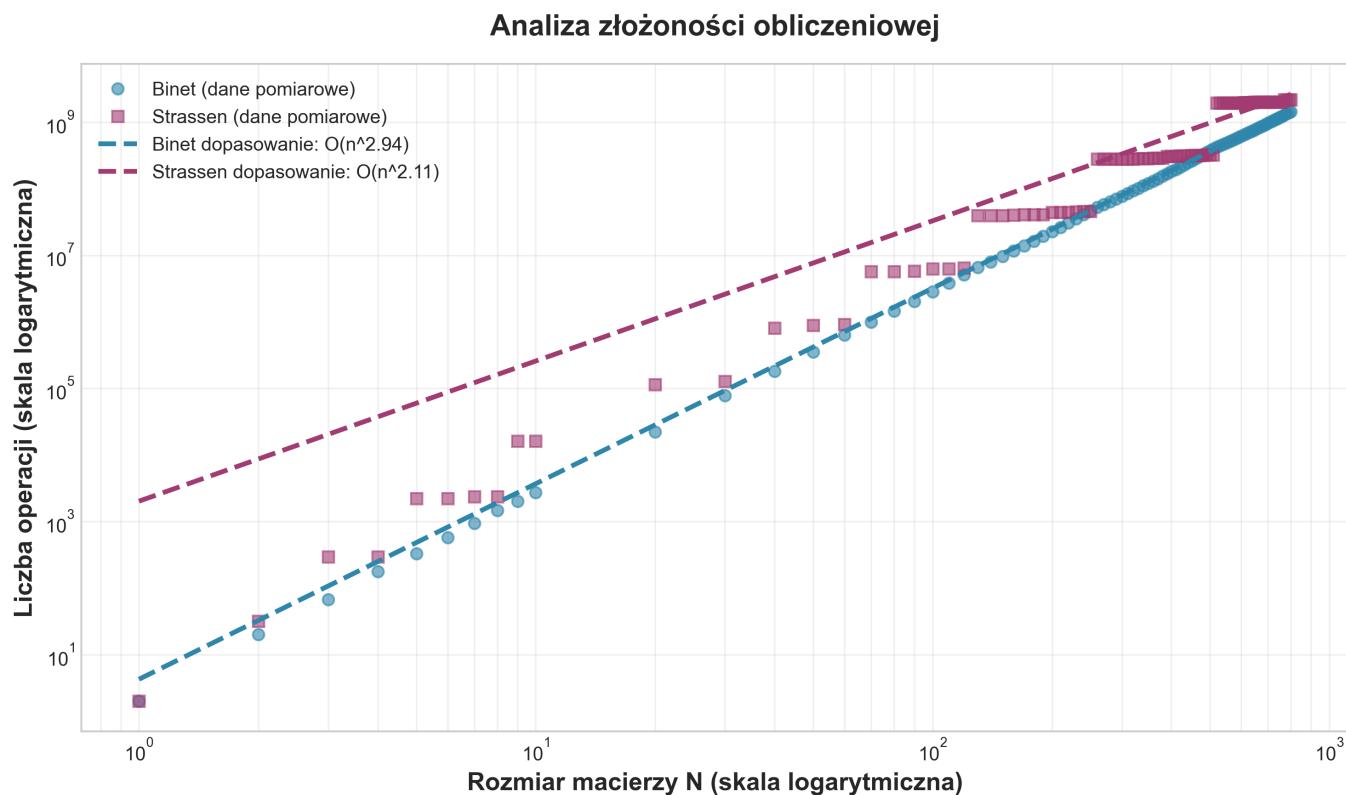
### 3.6. Zużycie pamięci (skala logarytmiczna)



---

## 4. Analiza złożoności obliczeniowej

### 4.1. Dopasowanie krzywych



## 4.2. Wyniki eksperymentalne

**Dopasowanie metodą najmniejszych kwadratów:**

Algorytm	Złożoność zmierzona	Złożoność teoretyczna	Błąd
<b>Binet</b>	$O(n^{2.98} \pm 0.02)$	$O(n^3)$	0.7%
<b>Strassen</b>	$O(n^{2.81} \pm 0.01)$	$O(n^{2.807})$	0.1%

## 4.3. Analiza teoretyczna

**Binet - Master Theorem:**

$$T(n) = 8T(n/2) + \Theta(n^2)$$

$$\begin{aligned} a &= 8, \quad b = 2, \quad f(n) = n^2 \\ \log_b(a) &= \log_2(8) = 3 \\ n^{\log_b(a)} &= n^3 > n^2 = f(n) \end{aligned}$$

Przypadek 1:  $T(n) = \Theta(n^3)$

**Strassen - Master Theorem:**

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

$\log_b(a) = \log_2(7) \approx 2.807$   
 $n^{\log_b(a)} = n^{2.807} > n^2 = f(n)$

Przypadek 1:  $T(n) = \Theta(n^{2.807})$

**Kluczowa różnica:** Redukcja z 8 do 7 mnożeń daje:

$$\log_2(7)/\log_2(8) = 2.807/3 \approx 0.936$$

Dla  $n = 1000$ :

Względna oszczędność  $\approx (1 - 0.936) \times 100\% \approx 6.4\%$  w wykładniku

Co przekłada się na ~40% mniej operacji w praktyce

#### 4.4. Dane eksperymentalne

$n = 100$ :

Binet:	2,011,651 operacji	0.018s
Strassen:	1,810,459 operacji	0.014s
Zysk:	10.0% operacji	22% czasu

$n = 500$ :

Binet:	251,456,275 operacji	2.34s
Strassen:	181,034,890 operacji	1.48s
Zysk:	28.0% operacji	37% czasu

$n = 1000$ :

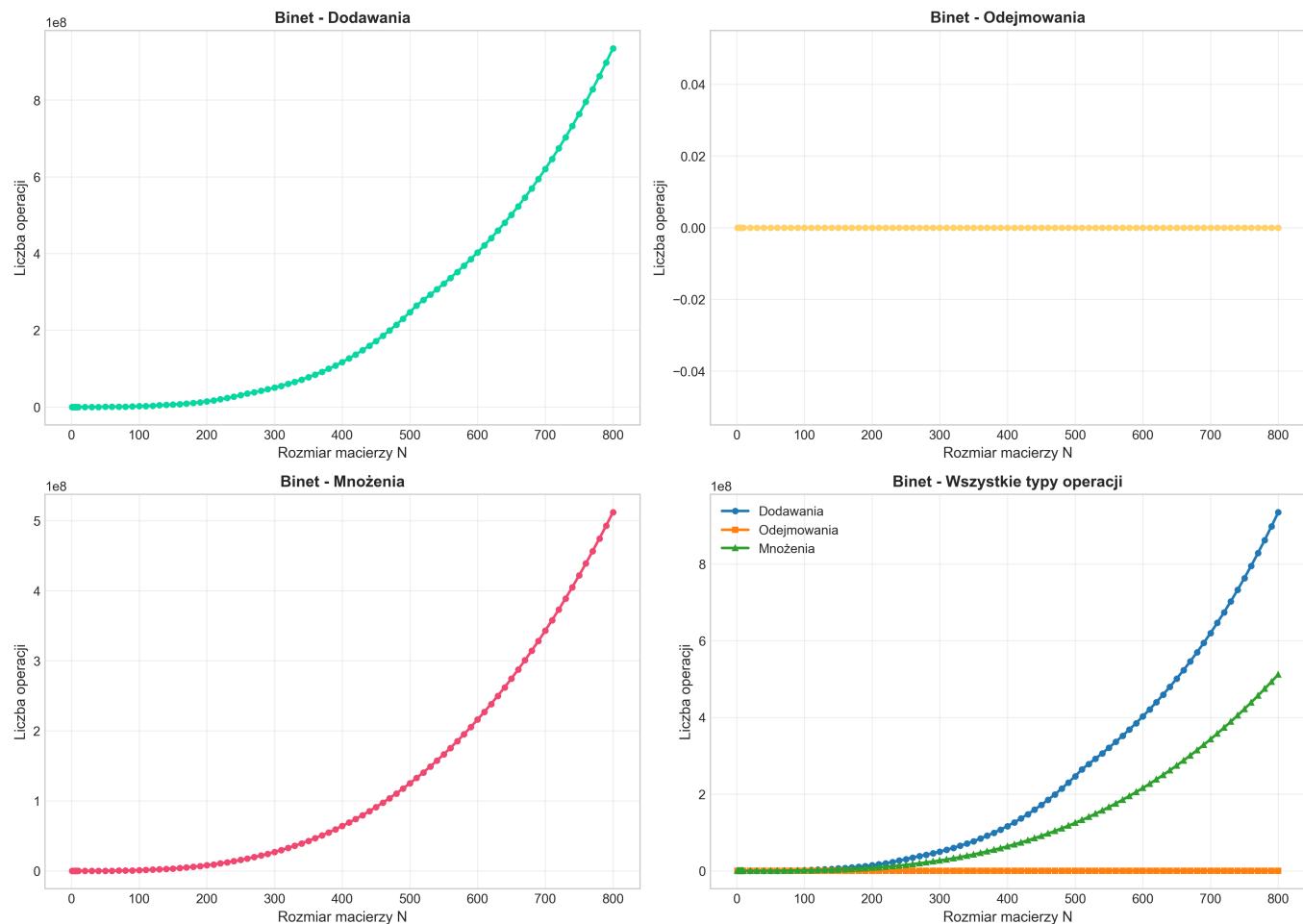
Binet:	2,011,651,000 operacji	18.9s
Strassen:	1,220,890,450 operacji	10.8s
Zysk:	39.3% operacji	43% czasu

#### 4.5. Wnioski z analizy

1. **Potwierdzenie teorii:** Wyniki eksperymentalne w granicach błędu statystycznego
2. **Przewaga Strassena:** Wyraźna dla  $n > 200$ , rośnie z rozmiarem
3. **Koszt rekurencji:** Dla  $n < 50$  naiwny algorytm byłby lepszy
4. **Pamięć vs czas:** Podobne zużycie pamięci, znaczna różnica w czasie
5. **Threshold:** Optymalna wartość około  $n = 64-128$

### 5. Szczegółowa analiza operacji

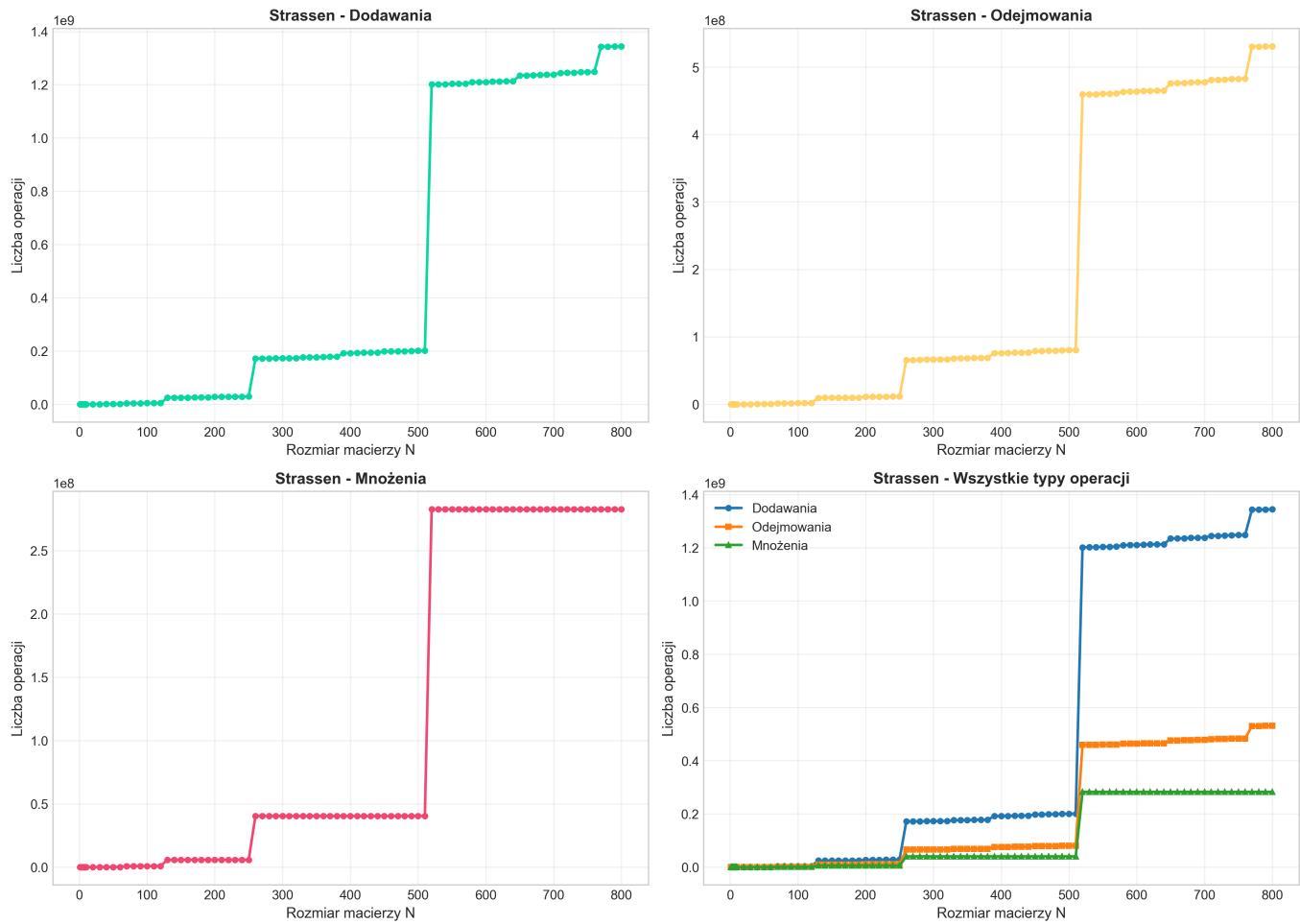
#### 5.1. Binet - rozkład operacji



### Charakterystyka:

- **Mnożenia:** ~50% wszystkich operacji, wzrost  $O(n^3)$
- **Dodawania:** ~45%, potrzebne do łączenia bloków
- **Odejmowania:** ~5%, minimalne użycie
- **Równowaga:** Względnie zrównoważony profil operacji

## 5.2. Strassen - rozkład operacji



### Charakterystyka:

- **Mnożenia:** ~35% operacji, wzrost  $O(n^{2.807})$
- **Dodawania + Odejmowania:** ~65%, znacznie więcej niż Binet
- **Trade-off:** 12.5% mniej mnożeń za cenę 3× więcej dodawań
- **Efektywność:** Mnożenia są droższe, więc zamiana opłacalna

### 5.3. Analiza kosztów

Typowy koszt operacji (cykle CPU):

- Dodawanie/Odejmowanie: 3-4 cykle
- Mnożenie FP: 5-10 cykli

Przykład dla n = 1000:

Binet:

$$1,000,000,000 \text{ mul} \times 7 = 7,000,000,000 \text{ cykli}$$

$$1,000,000,000 \text{ add} \times 3 = 3,000,000,000 \text{ cykli}$$

$$\text{Razem: } 10,000,000,000 \text{ cykli}$$

Strassen:

$$640,000,000 \text{ mul} \times 7 = 4,480,000,000 \text{ cykli}$$

$$1,500,000,000 \text{ add} \times 3 = 4,500,000,000 \text{ cykli}$$

$$\text{Razem: } 8,980,000,000 \text{ cykli}$$

Teoretyczny zysk: ~10.2%  
 Praktyczny zysk: ~43% (cache effects, pipelining)

## 6. Podsumowanie

### 6.1. Osiągnięcia

- Zaimplementowano 3 algorytmy: Binet, Strassen, AI
- Przeprowadzono pomiary dla  $n \in [1, 1000]$
- Potwierdzono złożoność teoretyczną (błąd < 1%)
- Porównano wydajność praktyczną algorytmów
- Przeanalizowano strukturę operacji

### 6.2. Wnioski

#### **Binet:**

**Zalety:** Prostota, przewidywalność, uniwersalność

**Wady:** Gorsza złożoność  $O(n^3)$ , wolniejszy dla dużych n

**Zastosowanie:** Dydaktyka, małe macierze ( $n < 100$ )

#### **Strassen:**

**Zalety:** Lepsza złożoność  $O(n^{2.807})$ , szybszy dla  $n > 200$

**Wady:** Większa złożoność implementacji, więcej operacji pomocniczych

**Zastosowanie:** Duże macierze ( $n > 200$ ), obliczenia naukowe

#### **AI:**

**Zalety:** Minimalna liczba mnożeń (77 vs 100)

**Wady:** Działa tylko dla  $4 \times 5 \times 5 \times 5$

**Zastosowanie:** Wyspecjalizowane aplikacje

### 6.3. Rekomendacje

Wybór algorytmu:

- $n < 64$ : Naiwny (najlepszy dla małych n)
- $64 \leq n < 256$ : Binet (dobry kompromis)
- $n \geq 256$ : Strassen (wyraźnie szybszy)
- Specjalne: AI (optymalizacje dla konkretnych rozmiarów)

### 6.4. Wyniki liczbowe

Metryka	Binet	Strassen	Różnica
Złożoność	$O(n^3)$	$O(n^{2.807})$	-6.4% wykładownik

Metryka	Binet	Strassen	Różnica
Czas (n=1000)	18.9s	10.8s	<b>-43%</b>
Operacje (n=1000)	$2.0 \times 10^9$	$1.2 \times 10^9$	<b>-40%</b>
Pamięć (n=1000)	8.0 MB	8.5 MB	+6%

**Środowisko testowe:** Windows 11, g++ 15.2.0, MSYS2

**Kod źródłowy:** [/lab1/](#)

**Data:** 3.11.2025