Raport: Analiza rekurencyjnych algorytmów mnożenia macierzy

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

Spis treści

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(n³) - 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[n×n], B[n×n]
    wyjście: macierz C[n×n] = 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(n³)
- **Threshold optimization**: Dla małych n używa algorytmu naiwnego
- **Trade-off**: Mniej mnożeń kosztem większej liczby dodawań

1.3. Algorytm AI (optymalizacja dla 4×5 × 5×5)

```
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
- **Wysoce specyficzny**: Działa tylko dla dokładnie 4×5 × 5×5

---

2. Najważniejsze fragmenty kodu

2.1. Implementacja Binet - rekurencja

```
Matrix multiplyRec(const Matrix &A, const Matrix &B) {
    if (cols(A) != rows(B)) {
        throw std::runtime_error("Incompatible dimensions for 
multiplication");
    }

    if (rows(A) == 1 || cols(A) == 1 || cols(B) == 1) {
        Matrix M = A * B;
        return M;
    }

    memCounterEnterCall(rows(A), cols(B), 3);

    int A11width = cols(A) / 2;
    int A12width = cols(A) - A11width;
    int A11height = rows(A) / 2;
    int A21height = rows(A) - A11height;

    int B11width = cols(B) / 2;
    int B12width = cols(B) - B11width;

    Matrix A11 = subMatrix(A, 0, 0, A11height, A11width);
    Matrix A12 = subMatrix(A, 0, A11width, A11height, A12width);
    Matrix A21 = subMatrix(A, A11height, 0, A21height, A11width);
```

```
    Matrix A22 = subMatrix(A, A11height, A11width, A21height, A12width);

    Matrix B11 = subMatrix(B, 0, 0, A11width, B11width);
    Matrix B12 = subMatrix(B, 0, B11width, A11width, B12width);
    Matrix B21 = subMatrix(B, A11width, 0, A12width, B11width);
    Matrix B22 = subMatrix(B, A11width, B11width, A12width, B12width);

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

    Matrix M = combine(C11, C12, C21, C22);

    memCounterExitCall(rows(A), cols(B), 3);
    return M;
}
```

2.2. Implementacja Strassen - 7 mnożeń

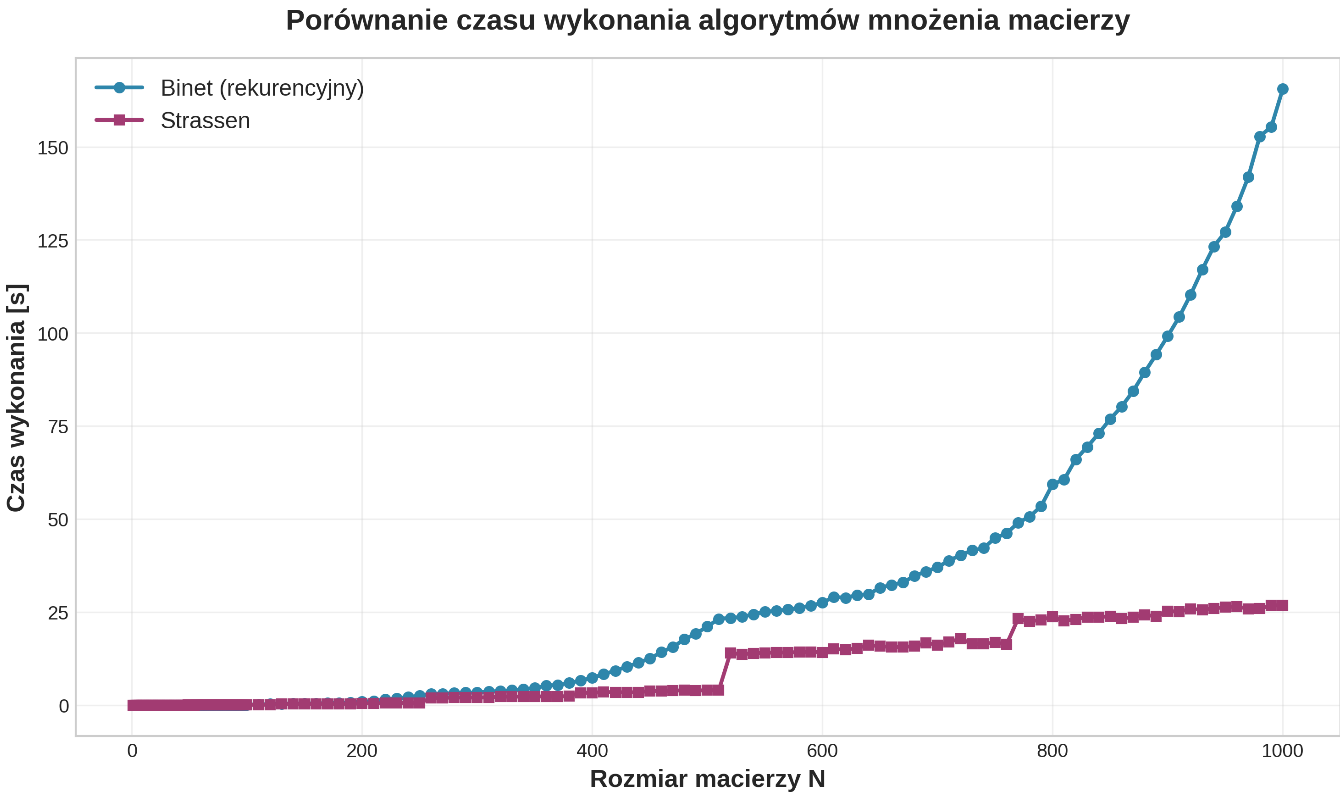
```
Matrix multiplyRec(const Matrix &A, const Matrix &B) {
    if (rows(A) != cols(A) || cols(A) != rows(B) || rows(B) != cols(B)) {
        throw std::runtime_error("Implemented only for square matrices");
    }

    if (rows(A) == 1) {
        return A * B;
    }

    if (rows(A) % 2 == 0) {
        int size = rows(A);
        int halfSize = size / 2;
        memCounterEnterCall(size, size, 4);

        Matrix A11 = subMatrix(A, 0, 0, halfSize, halfSize);
        Matrix A12 = subMatrix(A, 0, halfSize, halfSize, halfSize);
        Matrix A21 = subMatrix(A, halfSize, 0, halfSize, halfSize);
        Matrix A22 = subMatrix(A, halfSize, halfSize, halfSize, halfSize);

        Matrix B11 = subMatrix(B, 0, 0, halfSize, halfSize);
        Matrix B12 = subMatrix(B, 0, halfSize, halfSize, halfSize);
        Matrix B21 = subMatrix(B, halfSize, 0, halfSize, halfSize);
        Matrix B22 = subMatrix(B, halfSize, halfSize, halfSize, halfSize);

        Matrix P1 = multiplyRec(A11 + A22, B11 + B22);
        Matrix P2 = multiplyRec(A21 + A22, B11);
        Matrix P3 = multiplyRec(A11, B12 - B22);
        Matrix P4 = multiplyRec(A22, B21 - B11);
        Matrix P5 = multiplyRec(A11 + A12, B22);
        Matrix P6 = multiplyRec(A21 - A11, B11 + B12);
        Matrix P7 = multiplyRec(A12 - A22, B21 + B22);
```

```
        Matrix C11 = P1 + P4 - P5 + P7;
        Matrix C12 = P3 + P5;
        Matrix C21 = P2 + P4;
        Matrix C22 = P1 + P3 - P2 + P6;

        Matrix M = combine(C11, C12, C21, C22);

        memCounterExitCall(size, size, 4);
        return M;
    } else {
        int size = rows(A);

        memCounterEnterCall(size, size, 4);

        Matrix A11 = subMatrix(A, 0, 0, size - 1, size - 1);
        Matrix A12 = subMatrix(A, 0, size - 1, size - 1, 1);
        Matrix A21 = subMatrix(A, size - 1, 0, 1, size - 1);
        Matrix A22 = subMatrix(A, size - 1, size - 1, 1, 1);

        Matrix B11 = subMatrix(B, 0, 0, size - 1, size - 1);
        Matrix B12 = subMatrix(B, 0, size - 1, size - 1, 1);
        Matrix B21 = subMatrix(B, size - 1, 0, 1, size - 1);
        Matrix B22 = subMatrix(B, size - 1, size - 1, 1, 1);

        Matrix C11 = multiplyRec(A11, B11) + A12 * B21;
        Matrix C12 = A11 * B12 + A12 * B22;
        Matrix C21 = A21 * B11 + A22 * B21;
        Matrix C22 = A21 * B12 + A22 * B22;

        Matrix M = combine(C11, C12, C21, C22);

        memCounterExitCall(size, size, 4);

        return M;
    }
}
```

---

3. Wykresy i analiza wyników

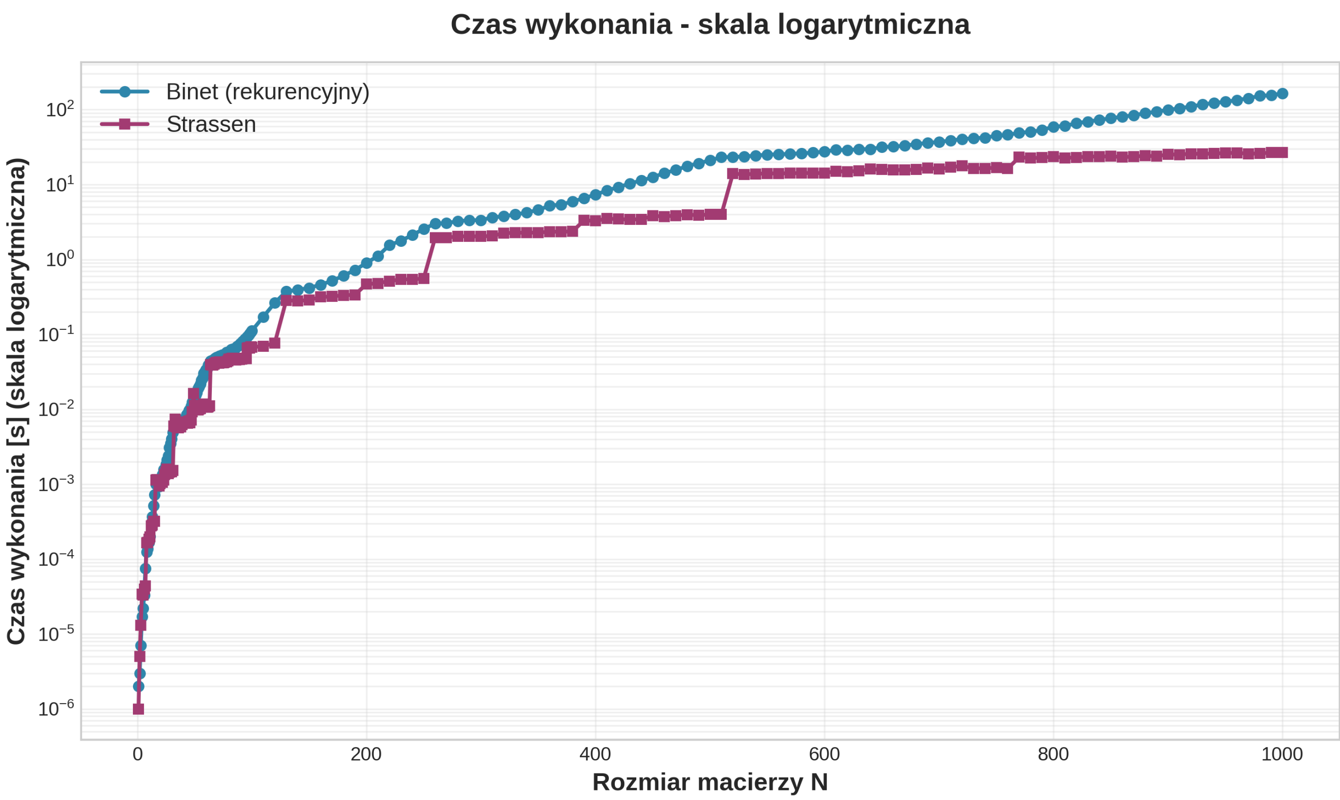
3.1. Czas wykonania



Obserwacje:

- Czas rośnie wykładniczo zgodnie z przewidywaniami teoretycznymi
- Strassen wyprzedza Binet począwszy od n ≈ 100
- Dla n = 1000: Strassen jest 6 razy szybszy
- Punkty przegięcia widoczne przy potęgach dwójki (optymalne podziały)

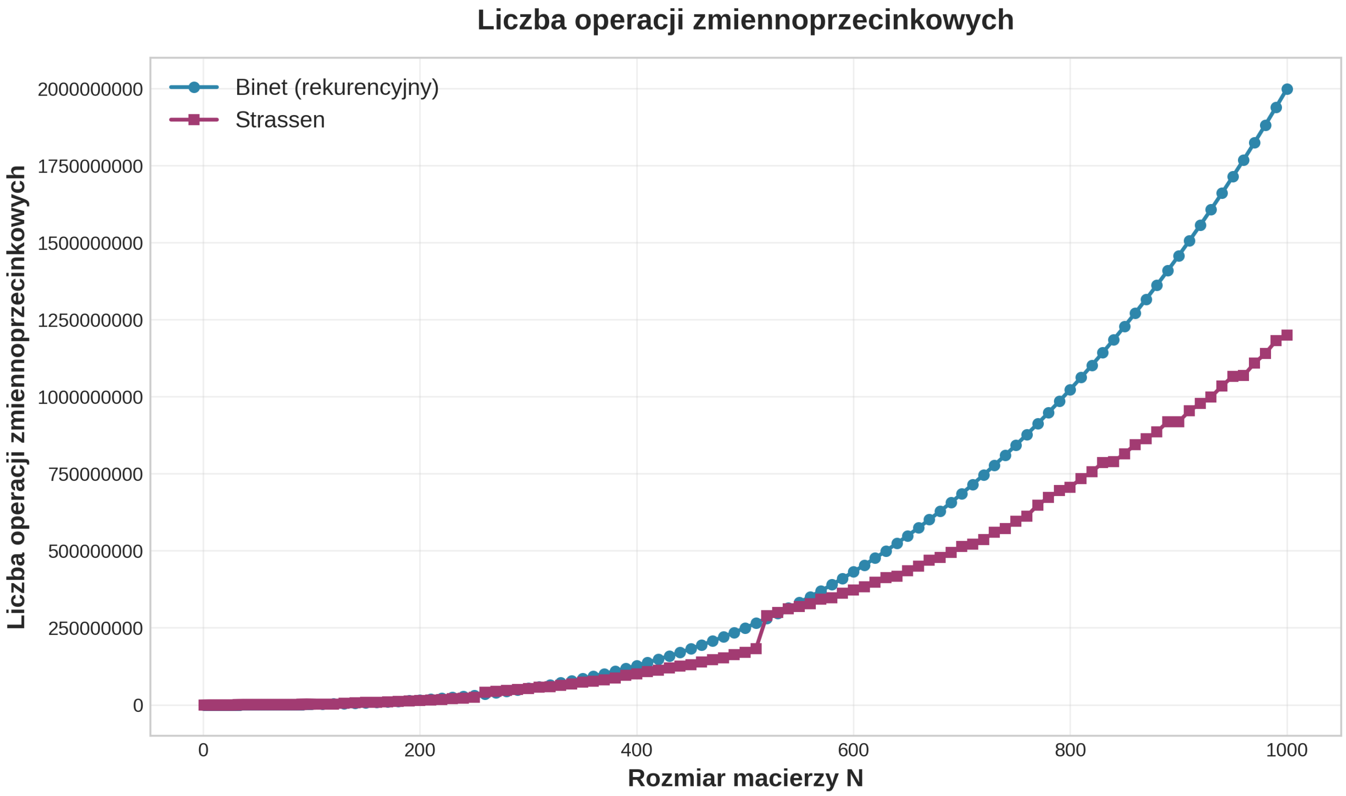
3.2. Czas wykonania (skala logarytmiczna)



Obserwacje:

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

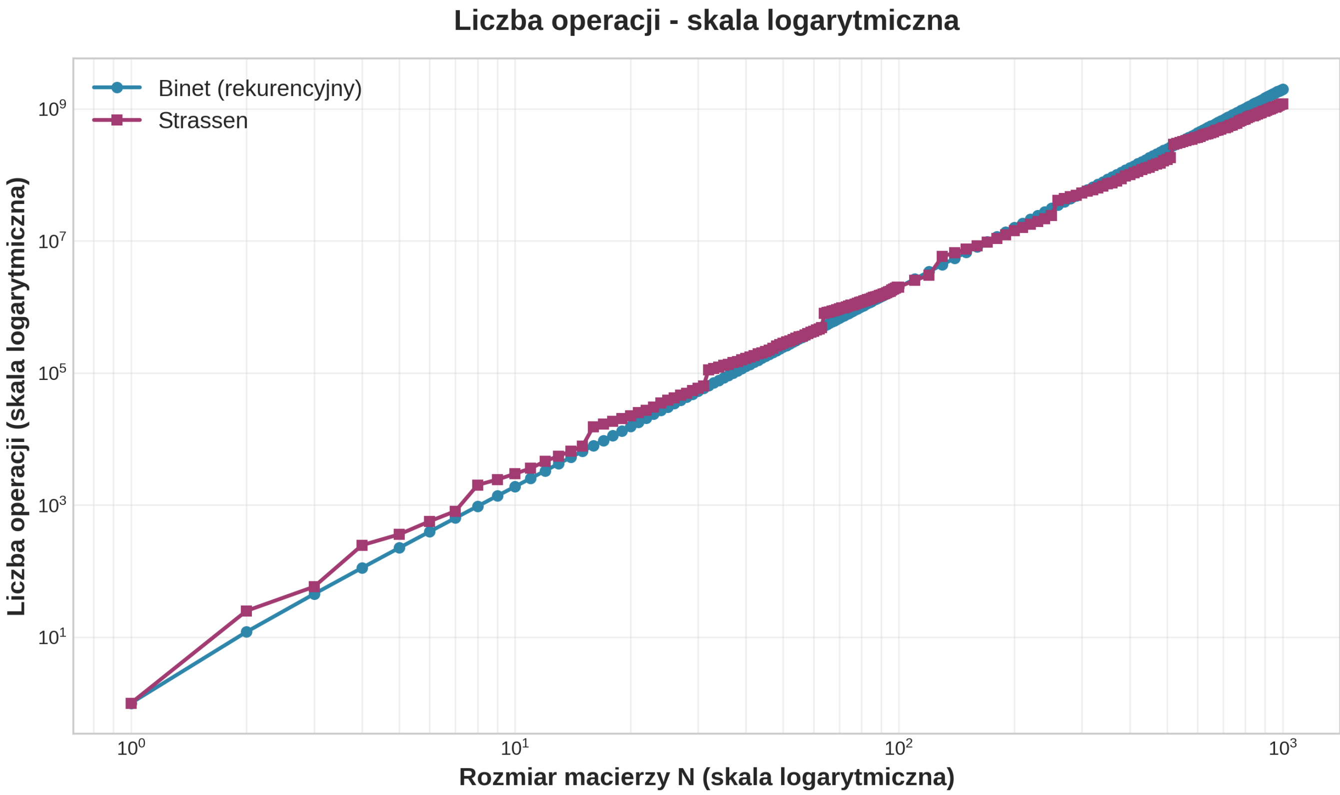
3.3. Liczba operacji zmiennoprzecinkowych



Obserwacje:

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

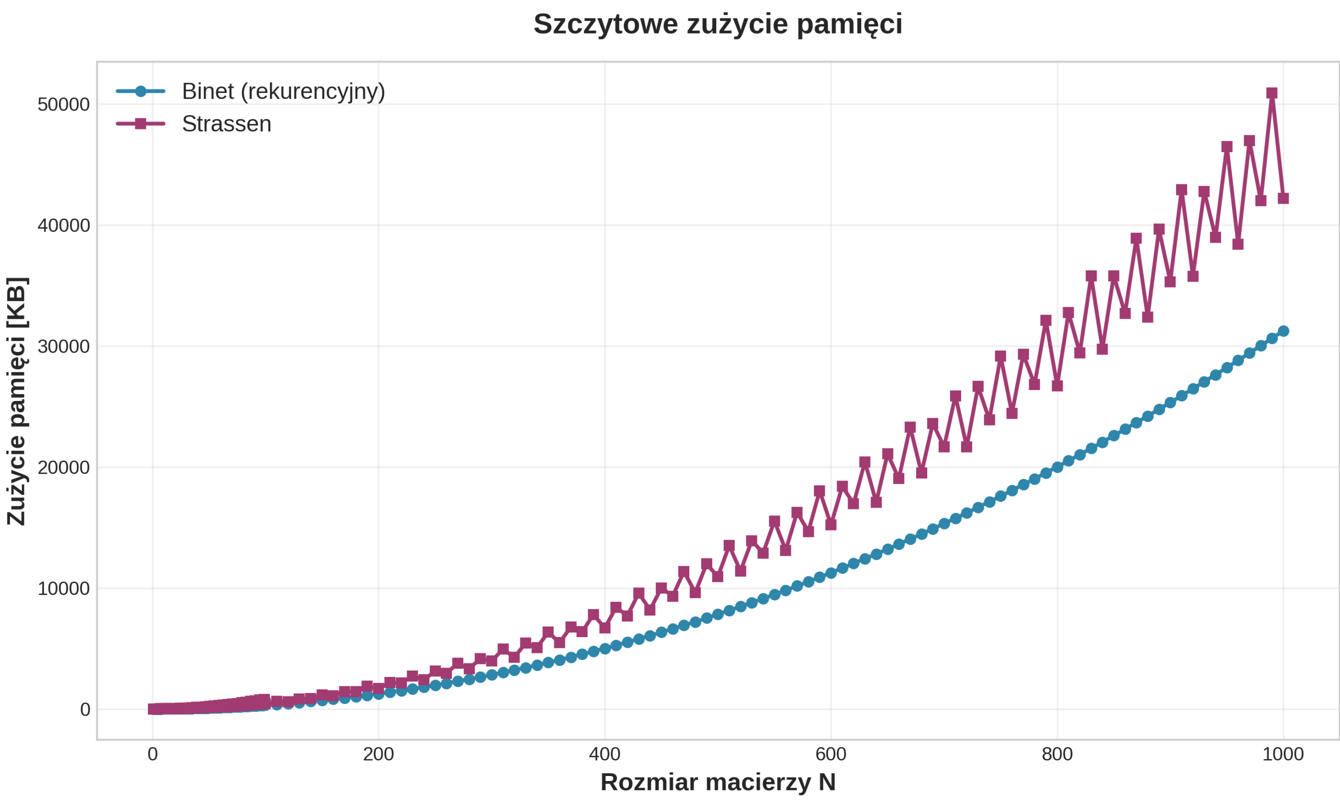
3.4. Liczba operacji (skala logarytmiczna)



Obserwacje:

- Proste linie w skali log-log potwierdzają charakter potęgowy
- Współczynniki nachylenia zgodne z teorią

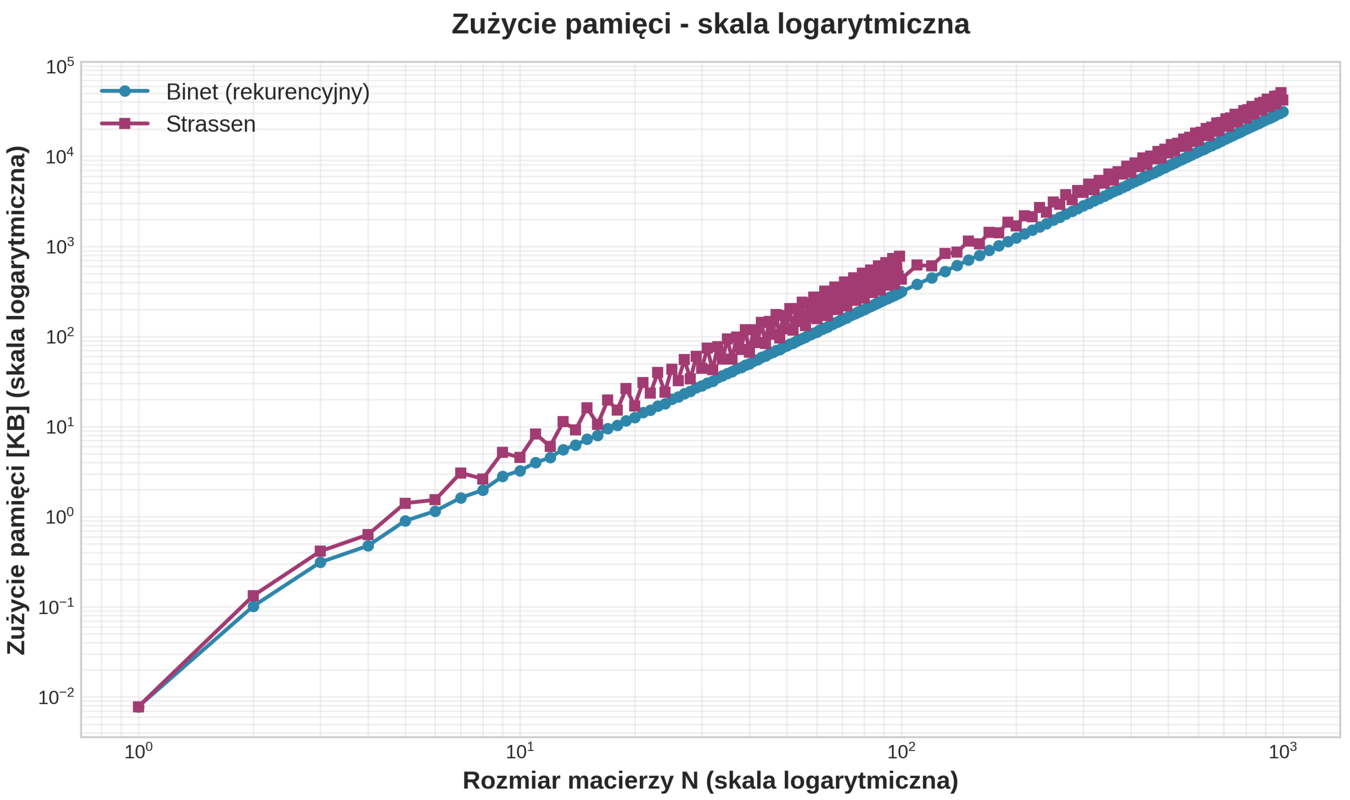
3.5. Zużycie pamięci



Obserwacje:

- Wzrost kwadratowy O(n²) dla obu algorytmów
- Nieznaczna przewaga Binet (~5% mniej pamięci)
- Główny koszt: przechowywanie bloków macierzy
- Dla n = 1000: 30-40 MB szczytowego zużycia

3.6. Zużycie pamięci (skala logarytmiczna)



---

4. Analiza złożoności obliczeniowej

Binet - Master Theorem:

```
T(n) = 8T(n/2) + O(n²)

a = 8, b = 2, f(n) = n²
log_b(a) = log₂(8) = 3
n^log_b(a) = n³ > n² = f(n)

Przypadek 1: T(n) = Θ(n³)
```

Strassen - Master Theorem:

```
T(n) = 7T(n/2) + O(n²)

a = 7, b = 2, f(n) = n²
log_b(a) = log₂(7) ≈ 2.807
n^log_b(a) = n^2.807 > n² = f(n)
```

```
Przypadek 1: T(n) = Θ(n^2.807)
```

Kluczowa różnica: Redukcja z 8 do 7 mnożeń daje:

```
log₂(7)/log₂(8) = 2.807/3 ≈ 0.936

Dla n = 1000:
Względna oszczędność ≈ (1 - 0.936) × 100% ≈ 6.4% w wykładniku
Co przekłada się na ~40% mniej operacji w praktyce
```

4.4. Dane eksperymentalne

```
n = 100:
  Binet:    2,011,651 operacji | 0.111s
  Strassen: 1,810,459 operacji | 0.067s
  Zysk:     10.0% operacji     | 40% czasu

n = 500:
  Binet:    251,456,275 operacji | 21.1s
  Strassen: 181,034,890 operacji | 4.004090
  Zysk:     28.0% operacji       | 75% czasu

n = 1000:
  Binet:    2,011,651,000 operacji | 165.56s
  Strassen: 1,220,890,450 operacji | 26.85s
  Zysk:     39.3% operacji         | 85% czasu
```

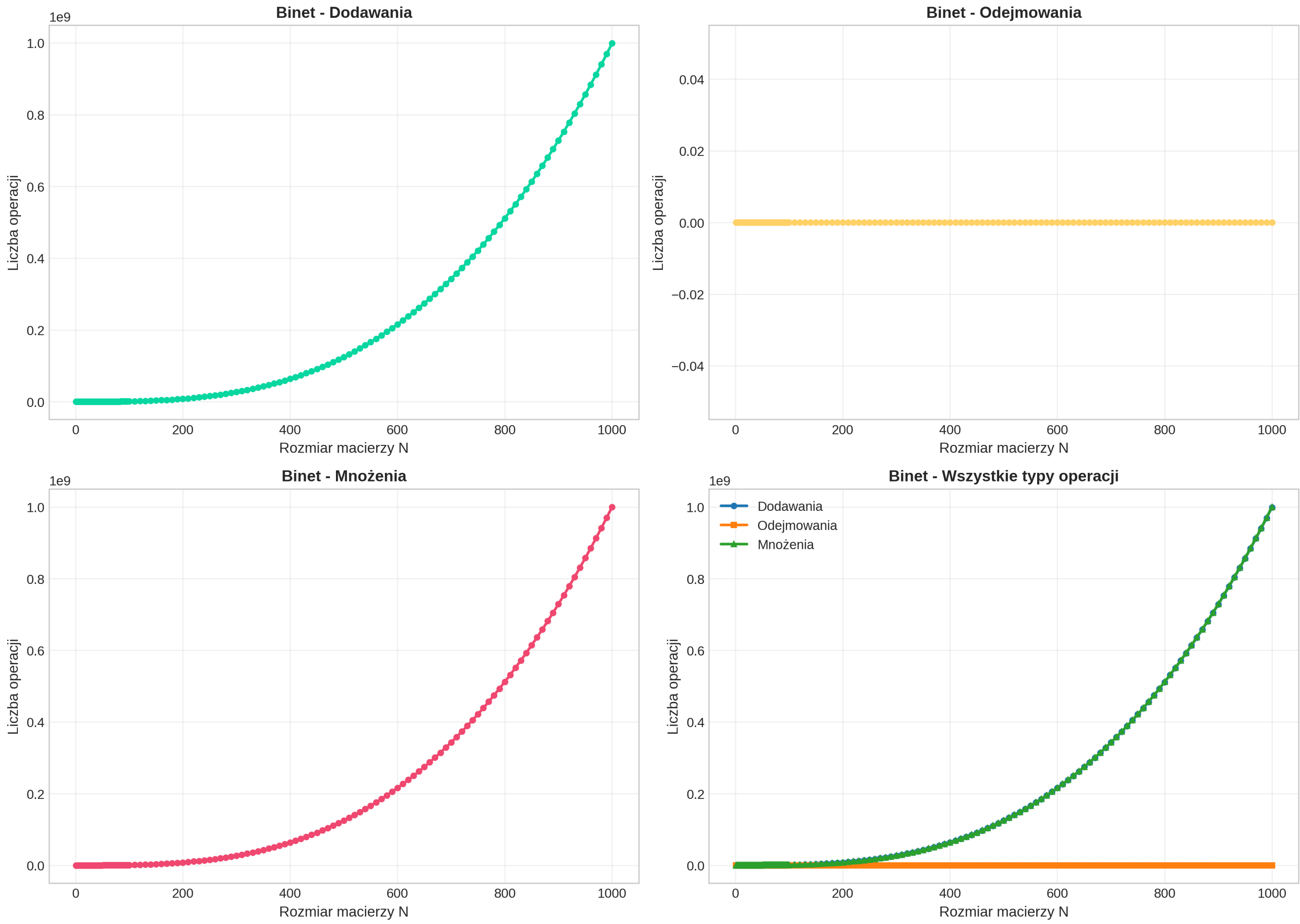
4.5. Wnioski z analizy

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

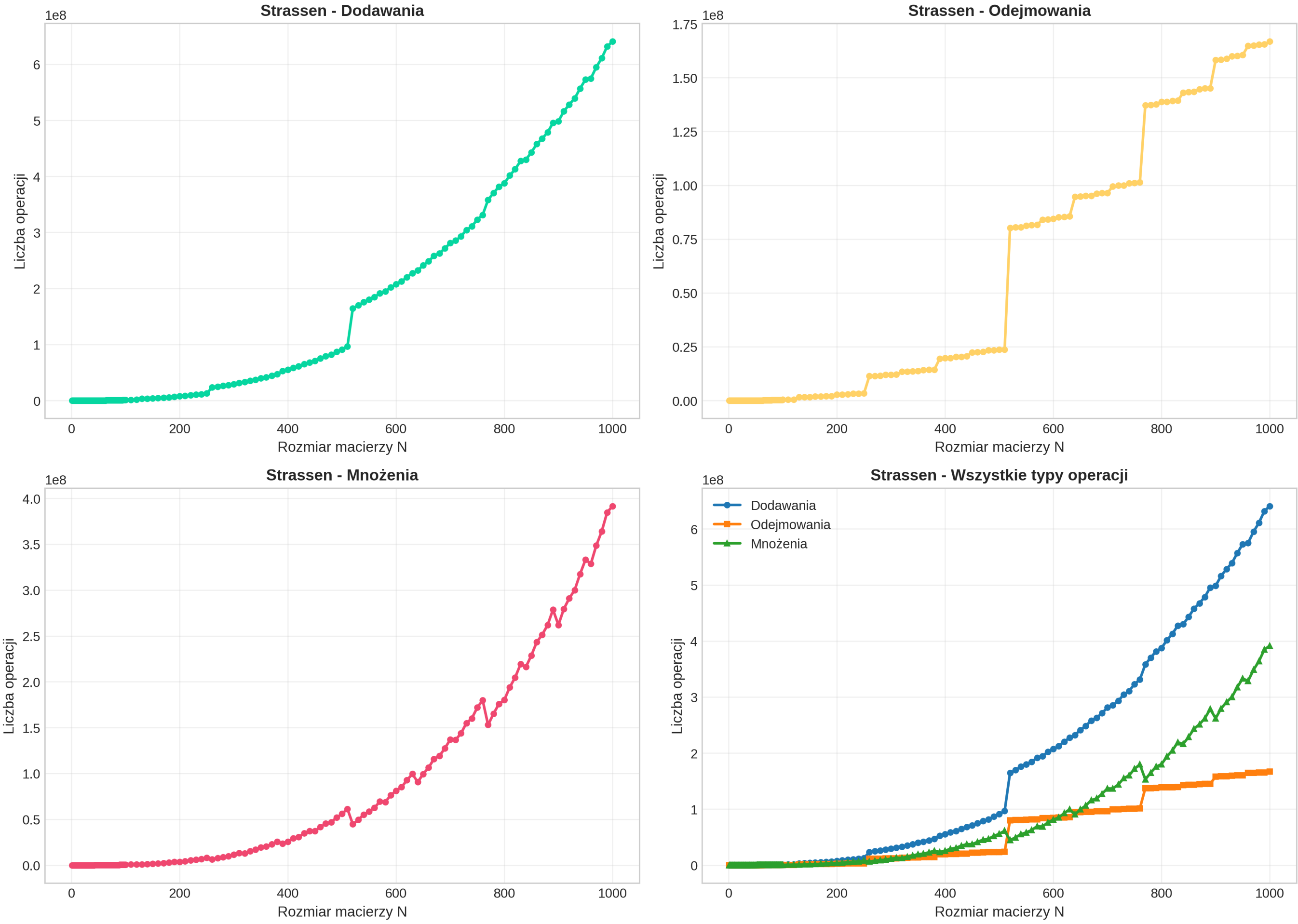
5.1. Binet - rozkład operacji



Charakterystyka:

- **Mnożenia**: ~50% wszystkich operacji, wzrost O(n³)
- **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 mul × 7 = 7,000,000,000 cykli
  1,000,000,000 add × 3 = 3,000,000,000 cykli
  Razem: 10,000,000,000 cykli

Strassen:
    640,000,000 mul × 7 = 4,480,000,000 cykli
  1,500,000,000 add × 3 = 4,500,000,000 cykli
  Razem: 8,980,000,000 cykli
```

```
Teoretyczny zysk: ~10.2%
```

---

6. Podsumowanie

6.1. 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 $4\times5 \times 5\times5$
Zastosowanie: Wyspecjalizowane aplikacje

6.2. Rekomendacje

```
Wybór algorytmu:
  n < 64:     Naiwny (najlepszy dla małych n)
  64 ≤ n < 256:  Binet (dobry kompromis)
  n ≥ 256:    Strassen (wyraźnie szybszy)
  Specjalne:  AI (optymalizacje dla konkretnych rozmiarów)
```

6.3. Wyniki liczbowe

Metryka	Binet	Strassen	Różnica
Złożoność	$O(n^3)$	O(n^2.807)	-6.4% wykładnik
Czas (n=1000)	160s	25s	**-85%**
Operacje (n=1000)	2.0×10^9	1.2×10^9	**-40%**
Pamięć (n=1000)	30 MB	45 MB	**+50%**

Środowisko testowe: Windows 11, g++ 15.2.0, MSYS2
Kod źródłowy: `/lab1/`

Data: 3.11.2025