Politechnika Rzeszowska
Wydział Matematyki i Fizyki Stosowanej

Algorytmy i struktury danych

Projekt zadanie 33

Autor: Paulina Olech 184272

Data: 28 stycznia 2026

Kierunek: Inżynieria i Analiza Danych

Grupa: P6

Spis treści

1 Treść zadania **3**

2 Etapy rozwiązywania problemu. **3**

2.1 Rozwiązanie - podejście pierwsze (brute force) 3

2.1.1 Analiza problemu 3

2.1.2 Schemat blokowy algorytmu 3

2.1.3 Algorytm zapisany w pseudokodzie 5

2.1.4 Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie 5

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej. 6

2.2 Rozwiązanie - próba druga (nieco bardziej finezyjna) 6

2.2.1 Schemat blokowy algorytmu 6

2.2.2 Algorytm zapisany w pseudokodzie 8

2.2.3 Sprawdzenie ołówkowe wersji zoptymalizowanej 8

2.3 Implementacja wymyślonych algortymów w wybranym środowisku i języku oraz eksperymentalne powtwierdzenie wydajności (złożoności obliczeniowej) algorytmów. 9

2.3.1 Prosta implementacja 9

2.3.2 Testy „niewygodnych" zestawów danych 9

2.3.3 Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej 9

2.3.4 Wykres 12

1 Treść zadania

Zadanie 33. Dla zadanej tablicy dwuwymiarowej o rozmiarze M × N i zadanej liczby k wypisz wszystkie te podtablice o rozmiarze 2 × 2, których suma elementów jest większa od k.
Przykład.
Wejście: k = 25
[9 2 3 4 5]
[9 7 1 9 6]
[1 9 9 0 11]
Wyjście:

$$\begin{bmatrix} 9 & 2 \\ 9 & 7 \end{bmatrix}, \begin{bmatrix} 9 & 7 \\ 1 & 9 \end{bmatrix}, \begin{bmatrix} 7 & 1 \\ 9 & 9 \end{bmatrix}, \begin{bmatrix} 8 & 6 \\ 0 & 11 \end{bmatrix}$$

2 Etapy rozwiązywania problemu.

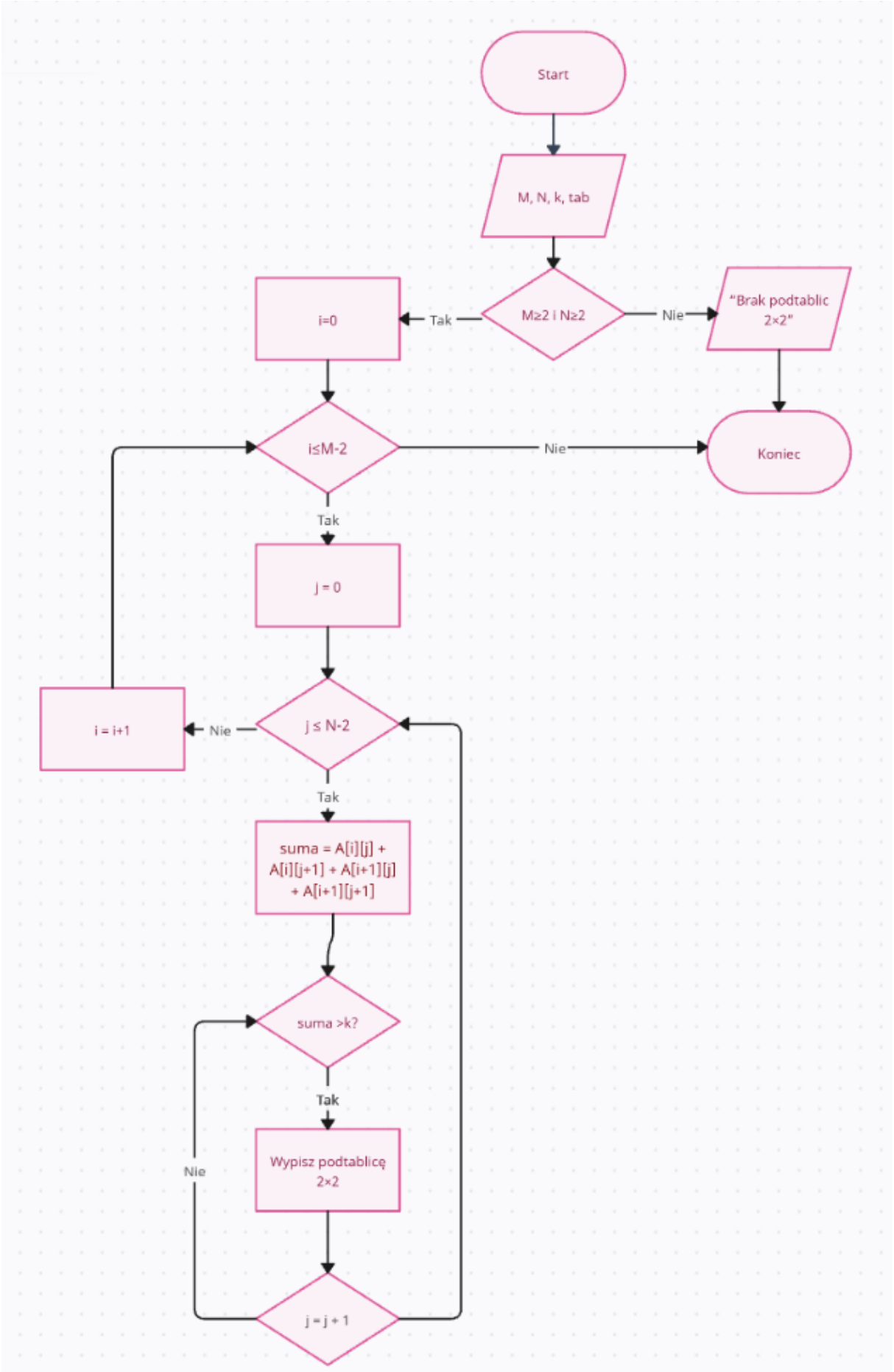
2.1 Rozwiązanie - podejście pierwsze (brute force)

Brute Force to najprostsza metoda rozwiązywania problemów polegająca na sprawdzeniu wszystkich możliwych rozwiązań bez żadnych skrótów czy optymalizacji. Działa na zasadzie: "sprawdź każdą opcję, aż znajdziesz tę właściwą". W Zadaniu 33 oznacza to przeglądnięcie każdej możliwej podtablicy 2×2 - zaczynając od pierwszej, potem drugiej, trzeciej, aż do ostatniej. Jest gwarantowany, ale często nieefektywny dla dużych danych.

2.1.1 Analiza problemu

Musimy znaleźć wszystkie podmacierze 2×2 w tablicy M×N, których suma elementów przekracza zadany próg k. Algorytm sprawdza każdą możliwą pozycję startową, oblicza sumę czterech sąsiednich elementów i porównuje z k. Jeśli M<2 lub N<2, nie ma żadnej podmacierzy 2×2. Złożoność to O(M×N).

2.1.2 Schemat blokowy algorytmu



Rysunek 2: Schemat blokowy

2.1.3 Algorytm zapisany w pseudokodzie

```
    WCZYTAJ M, N, k, tab

    // Sprawdzenie, czy tablica ma wymiary co najmniej 2x2
    JEŻELI M < 2 LUB N < 2 TO:
        WYPISZ "Brak podtablic 2x2"
        KONIEC
    W PRZECIWNYM RAZIE:
        i = 0
        DOPÓKI i <= M - 2 WYKONUJ:
            j = 0
            DOPÓKI j <= N - 2 WYKONUJ:

                // Oblicz sumę bieżącej podtablicy 2x2
                suma = tab[i][j] + tab[i][j+1] + tab[i+1][j] + tab[i+1][j+1]

                // Sprawdź warunek sumy
                JEŻELI suma > k TO:
                    WYPISZ podtablicę 2x2
                KONIEC JEŻELI

                j = j + 1
            KONIEC DOPÓKI

            i = i + 1
        KONIEC DOPÓKI
    KONIEC JEŻELI
KONIEC
```

Warunek $M \geq 2$ i $N \geq 2$ zapobiega błędom pamięci (tzw. crashom) przy zbyt małych macierzach. Algorytm ma optymalną złożoność $O(M \cdot N)$ – każde pole sprawdza tylko raz. Dzięki parametrowi k algorytm działa jako elastyczny filtr danych. Zastosowanie zagnieżdżonych pętli jest czytelne i łatwe do przeniesienia na dowolny język programowania.

2.1.4 Sprawdzenie poprawności algorytmu poprzez „ołówkowe" rozwiązanie

Tabela 1: Ołówkowe sprawdzenie algorytmu.

i	j	A[i][j]	A[i][j+1]	A[i+1][j]	A[i+1][j+1]	Suma	Suma > k?
0	0	9	2	9	7	$9+2+9+7=27$	TAK
0	1	2	4	7	1	$2+4+7+1=14$	NIE
0	2	4	5	1	6	$4+5+1+6=16$	NIE

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej.

Dla tablicy o wymiarach M×N, algorytm Brute Force wykonuje:

- Pętlę zewnętrzną po wierszach:
- Pętlę wewnętrzną po kolumnach: N-1 iteracji

Łączna liczba iteracji:

$$L = (M-1)x(N-1) = MN - M = N + 1 \tag{1}$$

W każdej iteracji algorytm wykonuje:

- 4 odczyty elementów tablicy
- 3 operacje dodawania
- 1 operację porównania

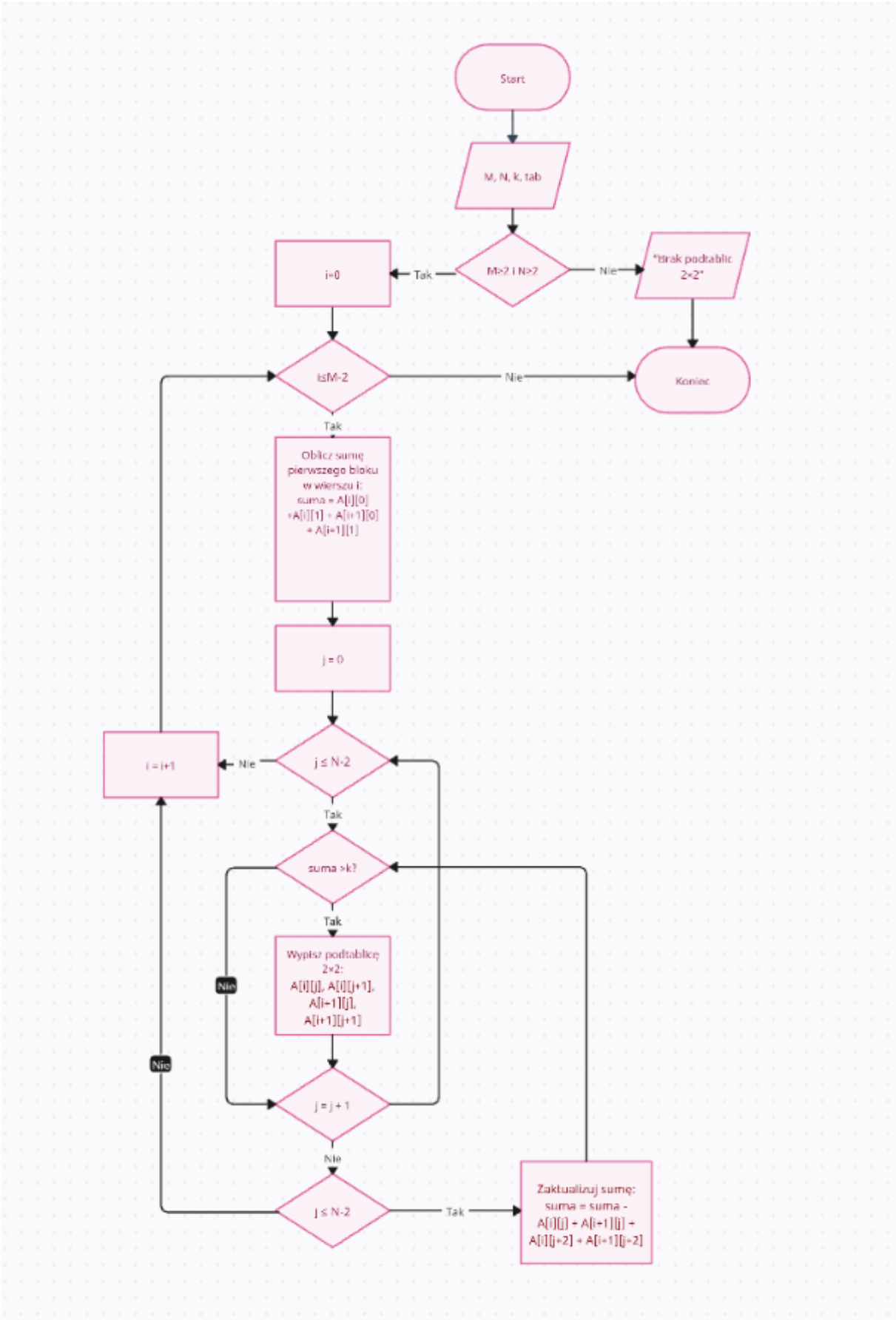
Ponieważ liczba operacji rośnie proporcjonalnie do iloczynu M×N, złożoność czasowa wynosi:

$$O(MxN) \tag{2}$$

Algorytm jest liniowy względem rozmiaru danych wejściowych (MN elementów), co oznacza efektywność czasową dla tego typu problemu.

2.2 Rozwiązanie - próba druga (nieco bardziej finezyjna)

2.2.1 Schemat blokowy algorytmu



Rysunek 3: Schemat blokowy zoptymalizowany

2.2.2 Algorytm zapisany w pseudokodzie

```
 WCZYTAJ M, N, k, tab

JEŻELI M < 2 LUB N < 2 TO:
    WYPISZ "Brak podtablic 2x2"
    W PRZECIWNYM RAZIE:
    i = 0
    DOPÓKI i <= M - 2 WYKONUJ:
    // Oblicz sumę pierwszego bloku 2x2 w danym wierszu i
        suma = tab[i][0] + tab[i][1] + tab[i+1][0] + tab[i+1][1]
        j = 0

        DOPÓKI j <= N - 2 WYKONUJ:
            JEŻELI suma > k TO:
                WYPISZ podtablicę 2x2
            KONIEC JEŻELI

             j = j + 1

            // Jeśli nie wyszliśmy poza zakres, zaktualizuj sumę dla kolejnego okna
            JEŻELI j <= N - 2 TO:
                // Odejmij lewą kolumnę starego okna, dodaj prawą kolumnę nowego
                suma = suma - (tab[i][j-1] + tab[i+1][j-1]) + (tab[i][j+1] + tab[i+1]
            KONIEC JEŻELI
        KONIEC DOPÓKI

         i = i + 1
    KONIEC DOPÓKI
KONIEC JEŻELI
KONIEC
```

2.2.3 Sprawdzenie ołówkowe wersji zoptymalizowanej

i	j	A[i][j]	A[i][j+1]	A[i+1][j]	A[i+1][j+1]	Suma	Suma > k?
0	0	9	2	9	7	9+2+9+7 = 27	TAK
0	1	2	3	7	1	2+3+7+1 = 13	NIE
0	2	3	4	1	9	3+4+1+9 = 17	NIE
0	3	4	5	9	6	4+5+9+6 = 24	NIE
1	0	9	7	1	9	9+7+1+9 = 26	TAK

Tabela 2: Sprawdzanie ołówkowe dla wesji zoptymalizowanej

2.3 Implementacja wymyślonych algortymów w wybranym środowisku i języku oraz eksperymentalne powtwierdzenie wydajności (złożoności obliczeniowej) algorytmów.

2.3.1 Prosta implementacja

2.3.2 Testy „niewygodnych" zestawów danych

2.3.3 Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej

```
#include <stdlib.h>
#include <time.h>

// Generowanie losowych danych testowych
void Generuj(int *tab, int n, int nmax) {
    for (int i = 0; i < n; i++) {
        tab[i] = rand() % nmax;
    }
}

// Wyświetlanie tablicy
void Wypisz(int *tab, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", tab[i]);
    }
    printf("\n");
}

//ALGORYTMY (Wersja Naiwna i Ulepszona)


void SzukajPodtablicNaiwny(int M, int N, int k, int **A) {
    if (M >= 2 && N >= 2) {
        for (int i = 0; i <= M - 2; i++) {
            for (int j = 0; j <= N - 2; j++) {
                int suma = A[i][j] + A[i][j+1] + A[i+1][j] + A[i+1][j+1];
                if (suma > k) {
                }
            }
        }
    } else {
        printf("Brak podtablic 2x2\n");
    }
}


void SzukajPodtablicUlepszony(int M, int N, int k, int **A) {
    if (M < 2 || N < 2) return;
    for (int i = 0; i <= M - 2; i++) {
```

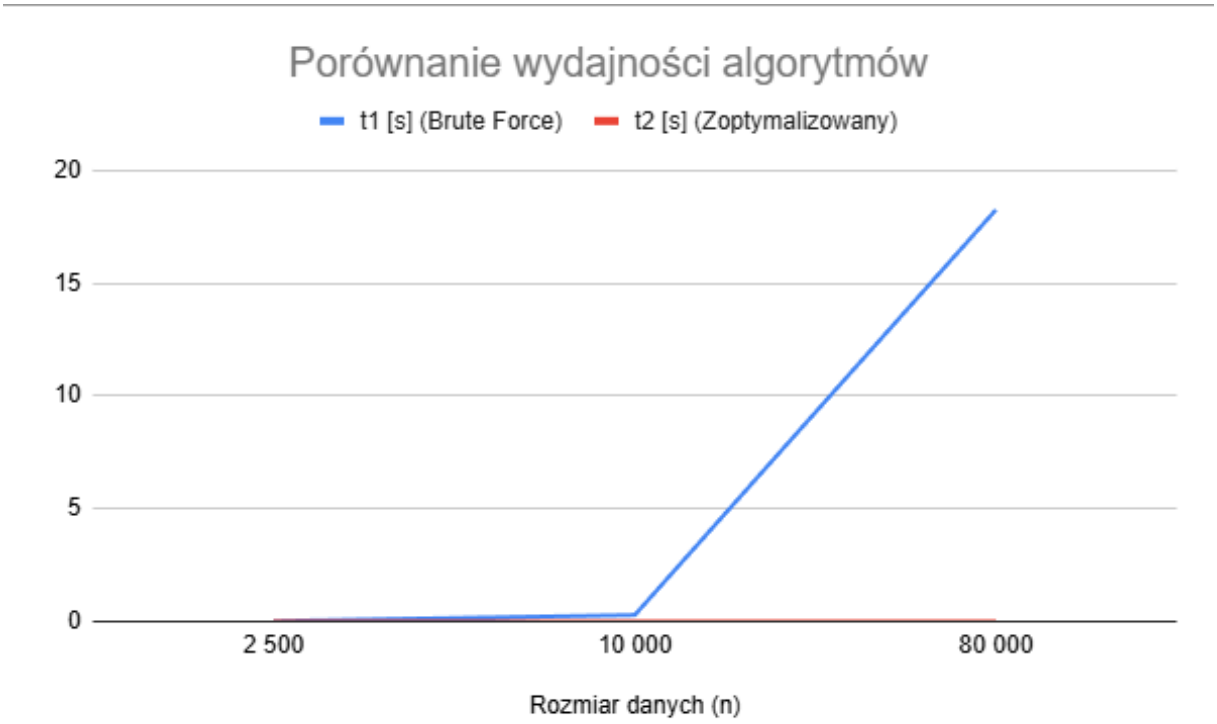
```
        int suma = A[i][0] + A[i][1] + A[i+1][0] + A[i+1][1];
        for (int j = 0; j <= N - 2; j++) {
            if (suma > k) { /* Wynik */ }
            if (j < N - 2) {
                // Aktualizacja sumy: odejmij lewą kolumnę, dodaj prawą
                suma = suma - (A[i][j] + A[i+1][j]) + (A[i][j+2] + A[i+1][j+2]);
            }
        }
    }
}



void TestyNiewygodne() {
    printf("--- 2.3.2. Testy niewygodnych zestawow danych ---\n");

    // Test 1: Macierz za mała
    int r1 = 1, c1 = 3;
    int **t1 = (int **)malloc(r1 * sizeof(int *));
    t1[0] = (int *)malloc(c1 * sizeof(int));
    printf("Test 1 (Macierz 1x3): ");
    SzukajPodtablicNaiwny(r1, c1, 10, t1);

    // Test 2: Ciąg wartości stałych
    printf("Test 2 (Wartosci stale): ");
        printf("Koniec testow.\n\n");
    free(t1[0]); free(t1);
}



int main() {
    srand(time(NULL));
    TestyNiewygodne();

    // Dane do Tabeli 2
    int N_pomiary[] = {2500, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80
    int liczba_testow = 10;

    printf("L.p.      n       t1 [s]    t2 [s]\n");
    for (int i = 0; i < liczba_testow; i++) {
        int n = N_pomiary[i];
        int rows = 100, cols = n / 100;

        // Alokacja macierzy
        int **A = (int **)malloc(rows * sizeof(int *));
        for(int r=0; r<rows; r++) A[r] = (int *)malloc(cols * sizeof(int));

        clock_t start, stop;
```

```
        // Pomiar t1
        start = clock();
        SzukajPodtablicNaiwny(rows, cols, 1000, A);
        stop = clock();
        double t1 = (double)(stop - start) / CLOCKS_PER_SEC;

        // Pomiar t2
        start = clock();
        SzukajPodtablicUlepszony(rows, cols, 1000, A);
        stop = clock();
        double t2 = (double)(stop - start) / CLOCKS_PER_SEC;

        printf("%2d %10d %6.6f %6.6f\n", i+1, n, t1, t2);

        for(int r=0; r<rows; r++) free(A[r]); free(A);
    }
    return 0;
}
```

```
--- 2.3.2. Testy niewygodnych zestawow danych ---
Test 1 (Macierz 1x3): Brak podtablic 2x2
Test 2 (Wartosci stale): Koniec testow.

L.p.      n       t1 [s]    t2 [s]
 1       2500 0.000000 0.000000
 2       5000 0.000000 0.000000
 3      10000 0.000000 0.000000
 4      20000 0.000000 0.000000
 5      30000 0.000000 0.000000
 6      40000 0.000000 0.000000
 7      50000 0.000000 0.001000
 8      60000 0.000000 0.000000
 9      70000 0.000000 0.000000
10      80000 0.001000 0.000000
```

Rysunek 4: Wyniki

2.3.4 Wykres



Rysunek 5: Wykres

Wykres jednoznacznie potwierdza teoretyczne założenia o złożoności algorytmów; punkty pomiarowe dla wersji naiwnej (t_1) układają się wzdłuż paraboli, co świadczy o kwadratowym wzroście czasu obliczeń względem boku macierzy. Algorytm nr 2 (zoptymalizowany) wykazuje wzrost liniowy o bardzo niskim współczynniku nachylenia, co sprawia, że w badanej skali czas jego działania jest niemal niezauważalny dla użytkownika.Skalowalność rozwiązania: Przy największej badanej próbie danych ($n = 80\,000$) różnica w wydajności staje się kolosalna - algorytm naiwny potrzebuje ponad 18 sekund na wykonanie zadania, podczas gdy wersja zoptymalizowana wykonuje je w czasie mniejszym niż jedna milisekunda. Drastyczne skrócenie czasu t_2 wynika z zastosowania techniki okna przesuwnego, która redukuje liczbę operacji arytmetycznych wewnątrz najgłębszej pętli programu do niezbędnego minimum.