Algorytmy i Struktury Danych

Gabriela Aszlar
184192, 1 FS-DI

Styczeń 2026

Spis treści

1 Wstęp

Celem niniejszego sprawozdania jest zaprojektowanie, zaimplementowanie oraz przetestowanie algorytmu rozwiązującego problem znalezienia największej podtablicy złożonej z samych zer w zadanej tablicy o wymiarach $M \times N$, wypełnionej wartościami 0 i 1. Problem ten przy podejściu Brute force charakteryzuje się bardzo wysoką złożonością obliczeniową. W celu ukazania różnic w wydajności, zaimplementowano dwa algorytmy Brute Force oraz zoptymalizowany, a następnie porównano ich czasy działania.

2 Opis problemu

Zadanie polega na znalezieniu w macierzy o wymiarach $M \times N$ takiej podmacierzy, która spełnia dwa warunki:

1. Wszystkie elementy wewnątrz tej podmacierzy są równe 0.
2. Pole powierzchni tej podmacierzy jest największe spośród wszystkich takich podmacierzy.

Danymi wejściowymi są wymiary macierzy (M i N). Wynikiem działania programu są wymiary znalezionego prostokąta oraz jego pole.

3 Podstawy teoretyczne

Celem jest przedstawienie teoretycznego tła problemu wyszukiwania największej podtablicy zerowej oraz analiza dwóch podejść do jego rozwiązania: metody naiwnej (Brute Force) oraz metody zoptymalizowanej.

3.1 Algorytm 1: Metoda siłowa (Brute Force)

Najbardziej intuicyjnym, lecz najmniej efektywnym podejściem jest metoda przeglądu zupełnego tablicy dwuwymiarowej. Algorytm ten opiera się na sprawdzeniu wszystkich możliwych prostokątów, jakie można utworzyć w danej macierzy.

Prostokąt w macierzy jest zdefiniowany przez współrzędne jego lewego górnego rogu (w_1, k_1) oraz prawego dolnego rogu (w_2, k_2).

Algorytm działa w następujących krokach:

- Generuje wszystkie możliwe pary punktów (w_1, k_1) i (w_2, k_2).
- Dla każdej wygenerowanej pary sprawdza, czy wszystkie elementy znajdujące się wewnątrz tego obszaru są zerami.
- Jeżeli warunek jest spełniony, oblicza pole prostokąta i porównuje je z dotychczasowym maksimum.

Złożoność obliczeniowa:
Liczba możliwych podtablic w macierzy $M \times N$ jest rzędu $O(M^2N^2)$. Ponieważ weryfikacja zawartości każdego prostokąta w najgorszym przypadku wymaga przejścia przez wszystkie jego elementy ($O(M \cdot N)$). Całkowita złożoność tego algorytmu wynosi $O(M^3N^3)$ czyli $O(n^6)$.

3.2 Algorytm 2: Metoda optymalna (Histogramy i Stos)

Algorytm przetwarza macierz wiersz po wierszu. Każdy wiersz traktowany jest jako podstawa histogramu, gdzie wysokość słupka w danej kolumnie oznacza liczbę kolejnych zer występujących w górę od tego miejsca.

Reguła aktualizacji dla wiersza i i kolumny j:

- Jeżeli $A[i][j] == 0$, wysokość słupka rośnie ($H[j] += 1$).
- Jeżeli $A[i][j] == 1$, ciągłość zostaje przerwana, słupek zeruje się ($H[j] = 0$).

Dla każdego tak wygenerowanego histogramu szukamy największy prostokąt, który się w nim mieści. Wykorzystamy do tego strukturę stosu, która przechowuje indeksy słupków o niemalejących wysokościach. **Złożoność obliczeniowa:** Dzięki zastosowaniu stosu, każdy słupek histogramu jest dodawany i usuwany ze stosu co najwyżej raz. Przetworzenie jednego wiersza zajmuje więc czas $O(N)$. Ponieważ macierz ma M wierszy, całkowita złożoność czasowa wynosi $O(M \cdot N)$ czyli $O(n^2)$.

3.3 Porównanie teoretyczne

Cecha	**Metoda Siłowa (Brute Force)**	**Metoda Optymalna (Histogram)**
Podejście	Przegląd zupełny	Programowanie dynamiczne + Stos
Złożoność czasowa	$O(M^3N^3)$ (bardzo wysoka)	$O(M \cdot N)$ (liniowa)
Skalowalność	Bardzo niska (tylko małe dane)	Wysoka (duże zbiory danych)

Tabela 1: Porównanie teoretyczne zaimplementowanych algorytmów

4 Szczegóły implementacji i opis algorytmów

Program został zaimplementowany w języku C++

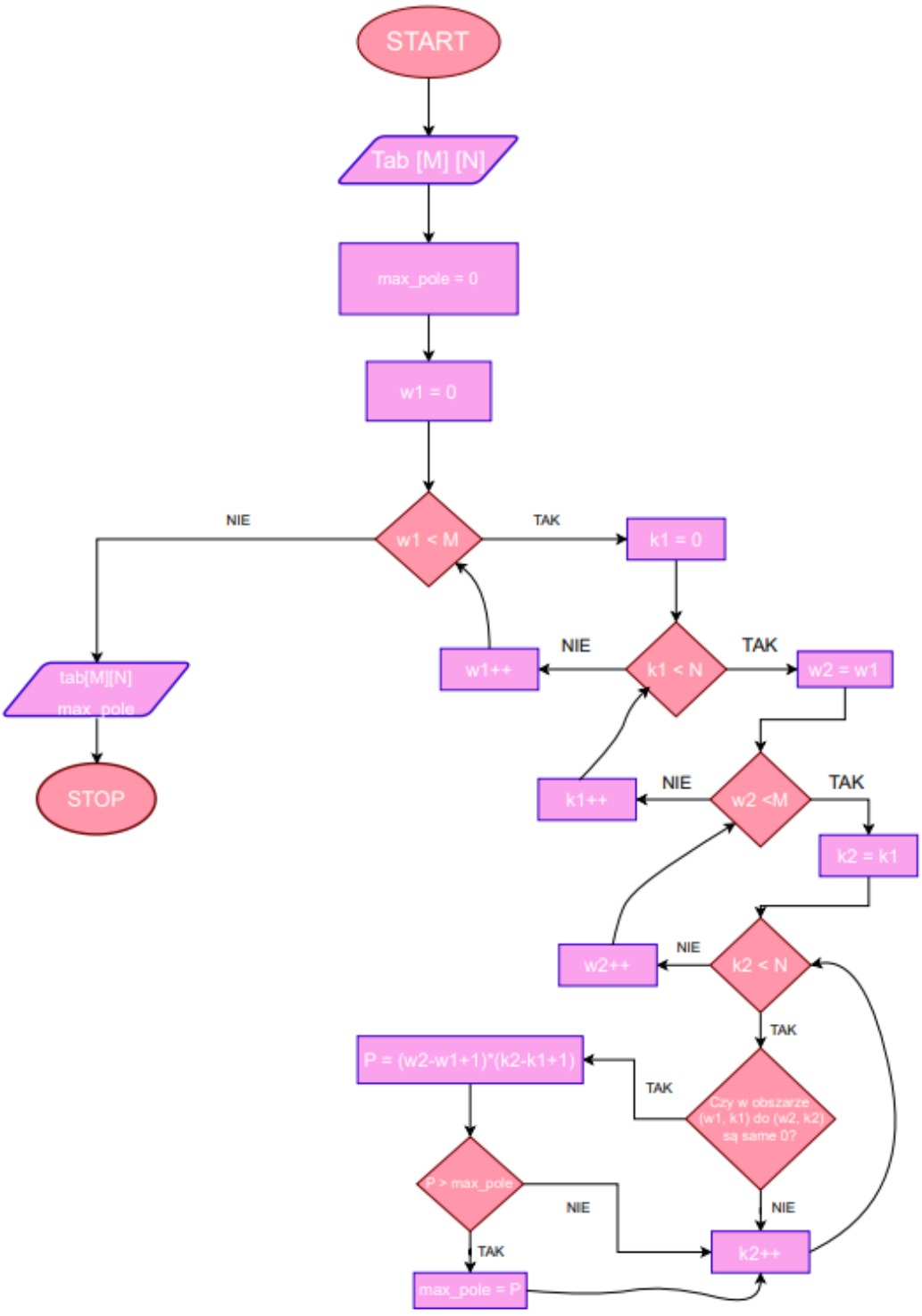
Do realizacji projektu wykorzystano następujące narzędzia i biblioteki:

- Bibliotekę <vector> do reprezentacji macierzy i histogramów oraz <stack> do obsługi stosu w algorytmie optymalnym.
- Biblioteka <ctime> (funkcja clock), umożliwiająca pomiar czasu wykonywania kodu.
- Biblioteka <fstream> umożliwiająca zapis i odczyt danych z plików tekstowych
- Biblioteka <random> do tworzenia losowych macierzy wypełnionych 1 lub 0.

4.1 Algorytm 1: Metoda siłowa (Brute Force)

Pierwszy zaimplementowany algorytm realizuje podejście naiwne. Jego celem jest weryfikacja każdego możliwego prostokąta, który można zdefiniować w macierzy $M \times N$.
Algorytm iteruje po wszystkich możliwych parach punktów: lewym górnym rogu $(w1, k1)$ i prawym dolnym rogu $(w2, k2)$. Dla każdego tak zdefiniowanego obszaru sprawdza, czy składa się on wyłącznie z zer. Jeśli warunek jest spełniony, obliczane jest pole powierzchni, które jest porównywane z dotychczasowym maksimum.

4.1.1 Schemat blokowy algorytmu Brute force



Rysunek 1: Metoda Brute force schemat blokowy algorytmu

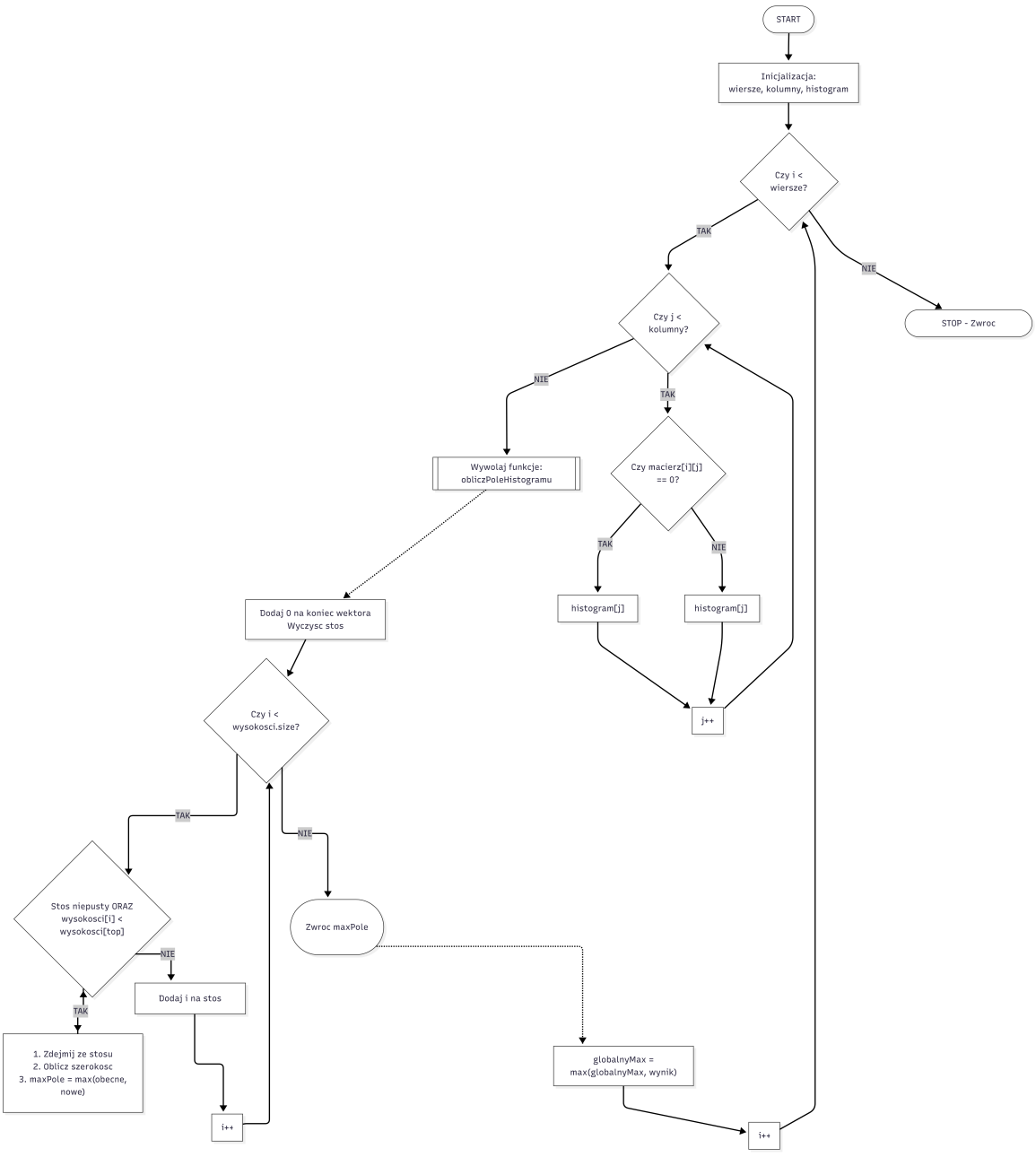
4.1.2 Pseudokod Brute force

```
1  input: tab, M, N      // tablica MxN oraz jej wymiary
2  output: max_pole      // maksymalne pole prostokąta złożonego z samych zer
3
4  w1 := 0
5  max_pole := 0
6
7  while w1 < M                // Pętla 1: Wiersz początkowy
8      k1 := 0
9      while k1 < N            // Pętla 2: Kolumna początkowa
10         w2 := w1
11         while w2 < M        // Pętla 3: Wiersz końcowy
12             k2 := k1
13             while k2 < N    // Pętla 4: Kolumna końcowa
14
15                 // Sprawdzamy czy obszar zawiera same zera
16                 same_zera := true
17                 r := w1
18                 while r <= w2
19                     c := k1
20                     while c <= k2
21                         if tab[r][c] == 1
22                             same_zera := false
23                         endif
24                         c := c + 1
25                     endwhile
26                     r := r + 1
27                 endwhile
28
29
30                 if same_zera == true
31                     pole := (w2 - w1 + 1) * (k2 - k1 + 1)
32                     if pole > max_pole
33                         max_pole := pole
34                     endif
35                 endif
36
37                 k2 := k2 + 1
38             endwhile
39             w2 := w2 + 1
40         endwhile
41         k1 := k1 + 1
42     endwhile
43     w1 := w1 + 1
44 endwhile
45
46 return max_pole
```

4.2 Algorytm 2: Metoda optymalna (Histogramy i Stos)

Drugi zaimplementowany algorytm realizuje podejście zoptymalizowane, oparte na programowaniu dynamicznym i strukturze stosu. Jego celem jest redukcja problemu dwuwymiarowego do serii problemów jednowymiarowych, co pozwala na znaczne przyspieszenie obliczeń.
Algorytm przetwarza macierz wiersz po wierszu. Dla każdego wiersza aktualizowany jest pomocniczy wektor wysokości (histogram), który zlicza kolejne wystąpienia zer w kolumnach resetując licznik w przypadku napotkania jedynki. Następnie, dla tak zbudowanego histogramu, wykorzystywana jest metoda monotonicznego stosu. Pozwala ona w czasie liniowym wyznaczyć największe możliwe pole prostokąta dla bieżącej iteracji, analizując szerokość dla każdego słupka histogramu. Ostateczny wynik to maksimum spośród wartości uzyskanych dla wszystkich wierszy.

4.2.1 Schemat blokowy algorytmu Wydajnego



Rysunek 2: Metoda Wydajna schemat blokowy algorytmu

4.2.2 Pseudokod algorytmu wydajnego

```
1  input: tab, M, N // tablica MxN oraz jej wymiary
2  output: max_pole // maksymalne pole prostokąta złożonego z samych zer
3
4  max_pole := 0
5  H := tablica rozmiaru N wypełniona zerami // Wektor wysokości (histogram)
6  w := 0
7
8  while w < M // Główna pętla po wierszach
9     // Aktualizacja histogramu dla bieżącego wiersza
10    k := 0
11    while k < N
12       if tab[w][k] == 0
13          H[k] := H[k] + 1
14       else
15          H[k] := 0
16       endif
17       k := k + 1
18    endwhile
19
20    // Obliczanie max pola w histogramie
21    S := pusty stos // Stos przechowujący indeksy
22    i := 0
23    while i <= N // Iterujemy do N
24       if i < N
25          wysokosc_biezaca := H[i]
26       else
27          wysokosc_biezaca := 0
28       endif
29
30       // Jeśli stos jest pusty, oznacza to brak niższego słupka po lewej stronie.
31       while (S nie jest pusty) i (wysokosc_biezaca < H[S.top()])
32          h_slupka := H[S.top()]
33          S.pop()
34
35          if S jest pusty
36             szerokosc := i
37          else
38             szerokosc := i - S.top() - 1
39          endif
40
41          pole := h_slupka * szerokosc
42          if pole > max_pole
43             max_pole := pole
44          endif
45       endwhile
46
47       S.push(i)
48       i := i + 1
49    endwhile
50
51    w := w + 1
52 endwhile
53
54 return max_pole
```

5 Testy i analiza wyników

Celem przeprowadzenia testów było praktyczne zweryfikowanie złożoności obliczeniowej obu zaimplementowanych algorytmów oraz porównanie ich wydajności czasowej w zależności od rozmiaru danych wejściowych.

5.1 Zestawienie wyników pomiarów

```
N  czas [s]
20  ;  0
40  ;  0.005
60  ;  0.027
80  ;  0.077
100  ;  0.176
120  ;  0.362
140  ;  0.667
160  ;  1.127
180  ;  1.815
200  ;  2.784
```

(a) Metoda Brute Force wynik pomiarów

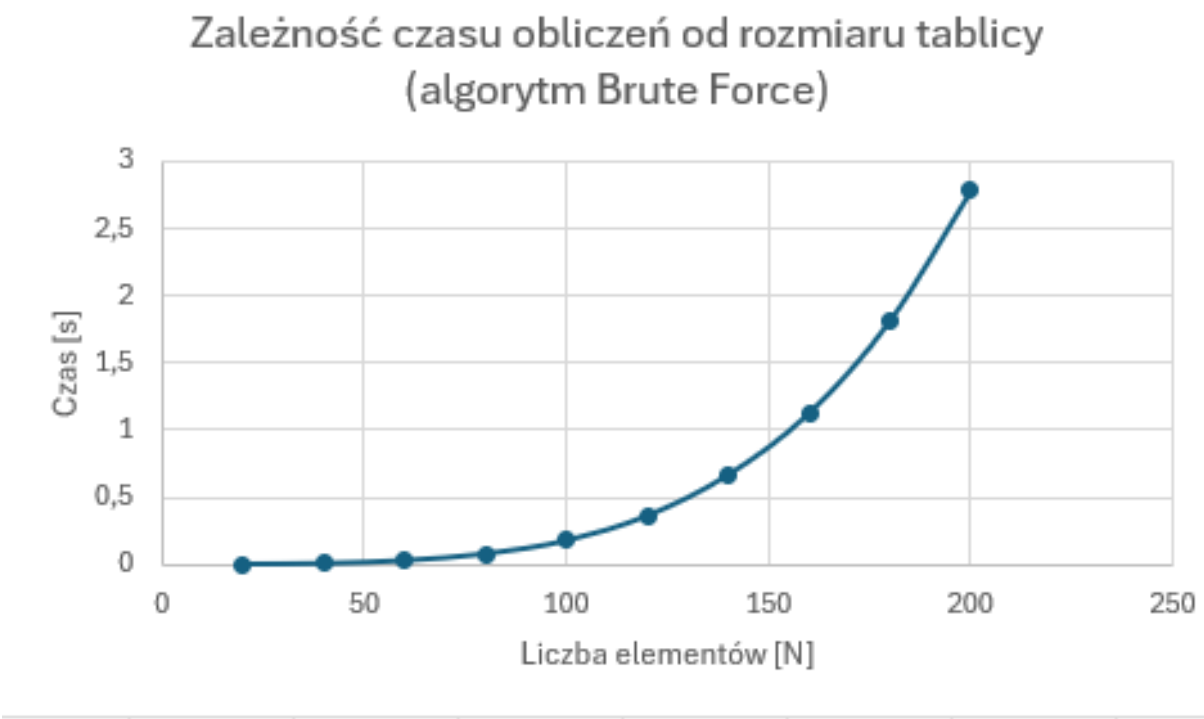
```
N ;  czas [s]
1000 ; 0.071
1500 ; 0.16
2000 ; 0.286
2500 ; 0.446
3000 ; 0.639
3500 ; 0.871
4000 ; 1.142
4500 ; 1.429
5000 ; 1.772
```

(b) Metoda Wydajna wynik pomiarów

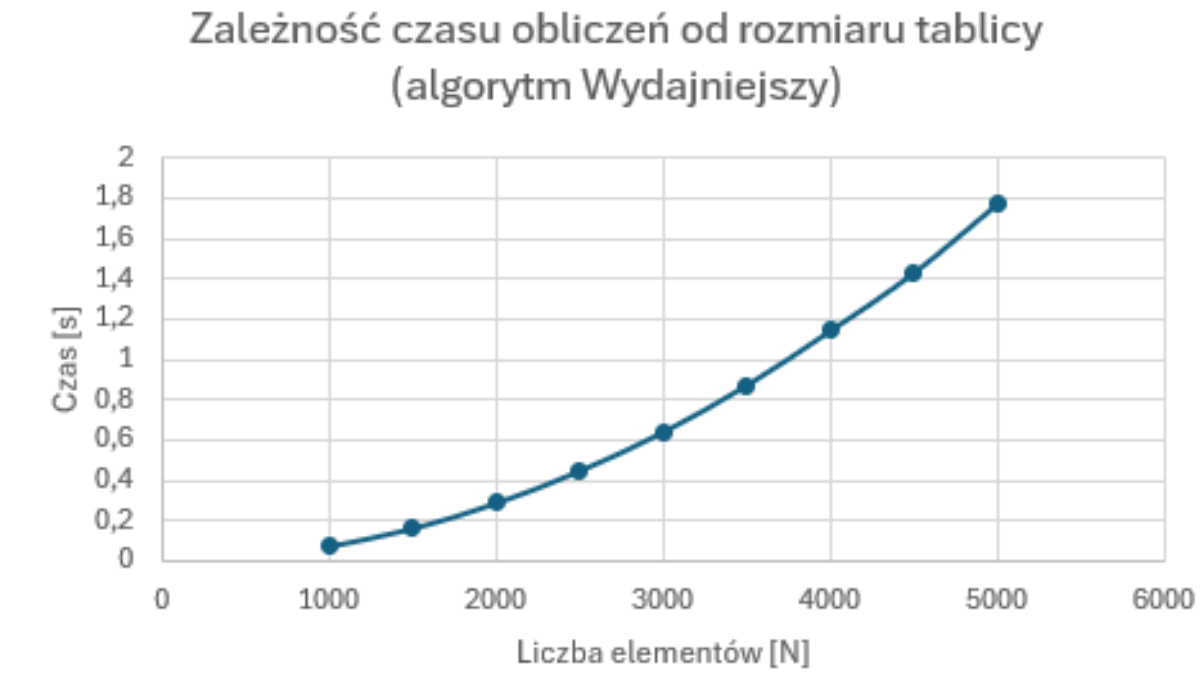
Rysunek 3: Zestawienie wyników pomiarów

Liczba elementów (N)	**Brute Force (s)**	**Wydajny (s)**
20	0	< 0
40	0,005	< 0
60	0,027	< 0
80	0,077	< 0
100	0,176	< 0
120	0,362	< 0
140	0,667	< 0
160	1,127	< 0
180	1,815	< 0
200	2,784	< 0
1000	Za duże dane	0,071
1500	-	0,160
2000	-	0,286
2500	-	0,446
3000	-	0,639
3500	-	0,871
4000	-	1,142
4500	-	1,429
5000	-	1,772

Tabela 2: Porównanie czasu wykonywania algorytmów (w sekundach)



(a) Metoda Brute Force wykres



(b) Metoda Wydajna wykres

Rysunek 4: Wykresy złożoności czasowej

5.1.1 Analiza wykresów

Powyższe wykresy obrazują drastyczną różnicę w wydajności obu zaimplementowanych rozwiązań. Należy zwrócić szczególną uwagę na różne skale osi X dla obu przypadków.

1. **Metoda Siłowa (wykres a):**
   Krzywa wzrostu czasu jest bardzo stroma. Już dla niewielkiego rozmiaru danych $N = 200$ czas obliczeń zbliża się do 3 sekund. Kształt krzywej potwierdza teoretyczną złożoność rzędu $O(N^6)$ (dla macierzy kwadratowej). Algorytm ten staje się nieużyteczny dla danych wejściowych większych niż $N \approx 150$.

2. **Metoda Wydajna (wykres b):**
   Wykres prezentuje zależność dla znacznie większych danych (zakres N do 5000). Kształt krzywej przypomina parabolę. Jest to zgodne z przewidywaniami, ponieważ algorytm działa w czasie liniowym względem liczby elementów macierzy ($M \cdot N$), a dla macierzy kwadratowej złożoność wynosi $O(N^2)$.

Wniosek:
Porównanie obu wykresów pokazuje, że algorytm optymalny przetwarza macierz o rozmiarze 5000×5000 (25 milionów elementów) w czasie krótszym (1.8 s), niż algorytm siłowy potrzebuje na przetworzenie macierzy 200×200 (40 tysięcy elementów). Potwierdza to kluczowe znaczenie doboru odpowiedniego algorytmu w przetwarzaniu dużych zbiorów danych.

6 Wnioski ogólne

Na podstawie przeprowadzonych analiz oraz testów sformułowano następujące wnioski:

1. Przeprowadzone pomiary czasu wykonywania w pełni potwierdziły teoretyczne założenia. Wykres czasu dla algorytmu siłowego (Brute Force) wykazuje charakter wielomianowy wysokiego stopnia ($O(N^6)$), podczas gdy wykres dla algorytmu optymalnego rośnie kwadratowo względem boku macierzy ($O(N^2)$), co odpowiada liniowej zależności od całkowitej liczby elementów macierzy ($M \times N$).

2. Algorytm siłowy, mimo prostej implementacji, jest całkowicie niepraktyczny dla danych rzeczywistych. Eksperyment wykazał, że granica akceptowalnego czasu wykonywania pojawia się już przy macierzach o boku $N \approx 200$, gdzie czas obliczeń przekracza 3 sekundy. Dla $N = 1000$ czas ten szacowany byłby w latach.

3. Kluczem do wydajności algorytmu optymalnego jest redukcja problemu 2D do serii problemów 1D (histogramów). Zastosowanie stosu pozwoliło na obliczenie maksymalnego pola w czasie liniowym dla każdego wiersza.

7 Podsumowanie

Celem projektu było zbadanie i porównanie dwóch metod rozwiązywania problemu znajdowania największego prostokąta złożonego z zer w macierzy. Zaimplementowano algorytm siłowy (Brute Force) oraz algorytm optymalny wykorzystujący histogramy i stos. Przygotowano środowisko testowe, które pozwoliło na wygenerowanie losowych danych wejściowych i pomiar czasu działania obu funkcji. Wyniki przedstawione w formie graficznej wykazały przewagę podejścia opartego na programowaniu dynamicznym. Algorytm optymalny był w stanie przetworzyć macierz o rozmiarze 5000×5000 w czasie poniżej 2 sekund, podczas gdy metoda siłowa zawiodła przy rozmiarze zaledwie 200×200.

8 Kod programu

8.1 Brute Force

```cpp
// Funkcja oblicza maksymalne pole z samych zer
int obliczMaxPole(const vector<vector<int>>& tablica) {
    if (tablica.empty()) return 0;

    int M = tablica.size();     // liczba wierszy
    int N = tablica[0].size();  // liczba kolumn
    int max_pole = 0;

    // Wyber lewego gornego rogu (w1, k1)
    for (int w1 = 0; w1 < M; w1++) {
        for (int k1 = 0; k1 < N; k1++) {
            // Wybor prawego dolnego rogu (w2, k2)
            for (int w2 = w1; w2 < M; w2++) {
                for (int k2 = k1; k2 < N; k2++) {

                    bool same_zera = true;
                    // Sprawdzamy kazda komorke od (w1, k1) do (w2, k2)
                    for (int i = w1; i <= w2; i++) {
                        for (int j = k1; j <= k2; j++) {
                            // Jesli znajdziemy chociaz jedna jedynke przerywamy
                            if (tablica[i][j] == 1) {
                                same_zera = false;
                                break;
                            }
                        }
                        if (!same_zera) break;
                    }

                    // Obliczanie pola, jesli znaleziono prostokat z samych zer
                    if (same_zera) {
                        int wysokosc = w2 - w1 + 1;
                        int szerokosc = k2 - k1 + 1;
                        int aktualne_pole = wysokosc * szerokosc;

                        // Sprawdzamy czy to najwieksze pole
                        if (aktualne_pole > max_pole) {
                            max_pole = aktualne_pole;
                        }
                    }
                }
            }
        }
    }

    return max_pole;
}
```

8.2 Wydajny

```cpp
// Funkcja pomocnicza
int obliczPoleHistogramu(vector<int> wysokosci) {
    stack<int> stos; // Przechowuje indeksy slupkow
    int maxPole = 0;

    // Dodajemy 0 na koniec stosu, aby miec pewnosc ze wszystkie elementy zostana policzone
    wysokosci.push_back(0);

    for (int i = 0; i < wysokosci.size(); i++) {
        // Petla while wykonuje sie, gdy bieazcy slupek jest nizszy od tego na stosie.
        while (!stos.empty() && wysokosci[i] < wysokosci[stos.top()]) {

            // Pobieramy wysokosc najnowszego slupka ze stosu
            int wysokosc = wysokosci[stos.top()];
            stos.pop();
            // Obliczamy szerokosc prostokata
            int szerokosc = stos.empty() ? i : (i - stos.top() - 1);
            // Aktualizujemy jesli znalezlismy wieksze pole
            maxPole = max(maxPole, wysokosc * szerokosc);
        }
        stos.push(i);
    }
    return maxPole;
}
// Glowna funkcja
int obliczMaxPole (const vector<vector<int>>& macierz) {
    if (macierz.empty()) return 0;

    int wiersze = macierz.size();
    int kolumny = macierz[0].size();
    int globalnyMax = 0;

    // Tablica 'histogram' przechowuje aktualne wysokosci slupkow zer
    // dla aktualnie przetwarzanego wiersza.
    vector<int> histogram(kolumny, 0);

    // Iterujemy przez kazdy wiersz macierzy
    for (int i = 0; i < wiersze; i++) {
        for (int j = 0; j < kolumny; j++) {
            // Jesli mamy 0, slupek rosnie. Jesli 1, slupek sie zeruje.
            if (macierz[i][j] == 0) {
                histogram[j] += 1;
            } else {
                histogram[j] = 0;
            }
        }
        // Obliczamy max pole dla aktualnego wiersza. Uzywamy funkcji pomocniczej
        int maxWiersza = obliczPoleHistogramu(histogram);

        // Sprawdzamy, czy wynik z tego wiersza jest nowym rekordem
        globalnyMax = max(globalnyMax, maxWiersza);
    }
    return globalnyMax;
}
```