

WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

DAWID KOTULA

Znalezienie największego prostokąta zbudowanego z jedynek

Opiekun pracy:
dr. inż. prof. Mariusz Borkowski

Rzeszów, 2025

1. Treść zadania

Dla tablicy $M \times N$ wypełnionej zerami lub jedynekami znaleźć pole największego prostokąta zbudowanego z jedynek.

Przykład:

WEJŚCIE

[0,1,0,0,0]

[1,1,0,0,0]

[1,1,1,1,0]

[1,1,1,1,0]

[0,1,0,0,0]

WYJŚCIE: 9 (Jedynki w wierszach 1-3, kolumnach 1-3)

2. Rozwiązanie – podejście pierwsze (*brute force*)

2.1.1 Analiza Problemu

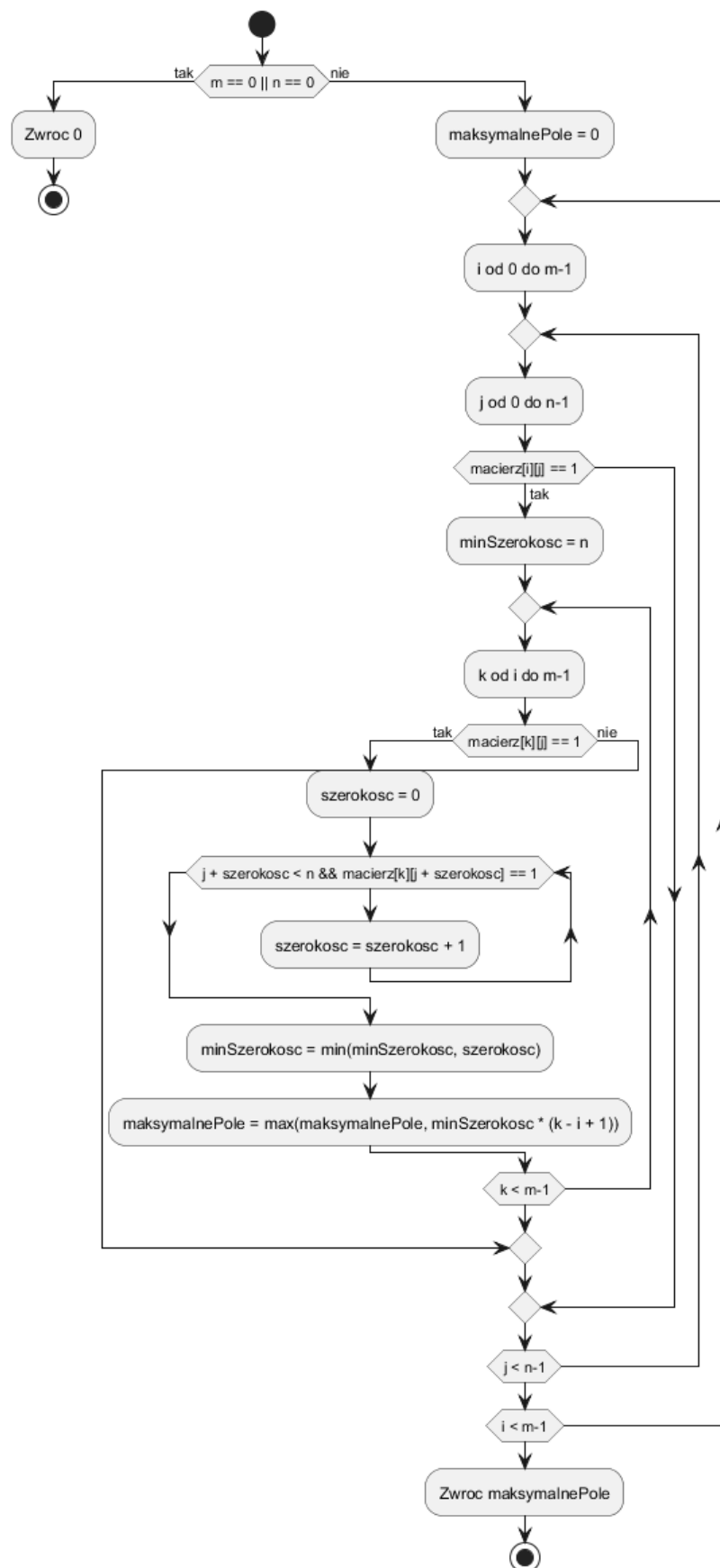
W podejściu *brute force* do znalezienia największego prostokąta z jedynek w tablicy $M \times N$ zaczynamy od utworzenia zmiennej pomocniczej `maks_pole`, w której będziemy przechowywać aktualną wartość maksymalnego pola prostokąta. Następnie, metodycznie, sprawdzamy wszystkie możliwe prostokąty w tablicy, bo obliczamy pole każdego prostokąta składającego się z jedynek i jeśli będzie ono większe od wartości przechowywanej w zmiennej `maks_pole`, przypisujemy je do tej zmiennej. Należy zwrócić uwagę na problem inicjalizacji zmiennej `maks_pole` właściwą wartością, która początkowo powinna wynosić 0. Jeśli algorytm zakończy analizę, oznacza to, że w tablicy nie znaleziono żadnego prostokąta składającego się z jedynek. Inną rzeczą, którą powinien zwrócić uwagę algorytm, jest sytuacja, gdy podana tablica jest pusta. Na ten wypadek w algorytmie zostanie dodana instrukcja warunkowa `if`, która na początku procesu petli po typie `Mtbl` sprawdzi, czy tablica ma elementy.

1. Dane wejściowe: Dane wejściowe w naszym algorytmie użytkownika:

- Struktura danych typu tablica (macierz), przechowująca wartości tablicy `Mtbl` z jedynekami i zerami.

2. Dane wyjściowe: Algorytm zwróci wartość największego pola prostokąta z jedynek w zmiennej `maks_pole`.

2.1.2 Schemat blokowy algorytmu



2.1.3 Algorytm zapisany w pseudokodzie

```
Jeśli m == 0 lub n == 0
    Zwróć 0

maksymalnePole = 0

Dla i od 0 do m-1:
    Dla j od 0 do n-1:
        Jeśli macierz[i][j] == 1:
            minSzerokosc = n
            Dla k od i do m-1 i macierz[k][j] == 1:
                szerokosc = 0
                Dopóki j + szerokosc < n i macierz[k][j + szerokosc] == 1:
                    szerokosc = szerokosc + 1
                minSzerokosc = min(minSzerokosc, szerokosc)
            maksymalnePole = max(maksymalnePole, minSzerokosc * (k - i + 1))

Zwróć maksymalnePole
```

Zauważmy, że w pseudokodzie do zapisu obu petli użyto petli typu `for`. Zapis algorytmu w pseudokodzie jest etapem pośrednim pomiędzy analizą problemu a opracowaniem algorytmu, a sama implementacja w konkretnym języku programowania, która zostanie przedstawiona w kolejnych rozdziałach. Pseudokod poprawia zrozumienie logiki algorytmu bez konieczności skupiania się na składni konkretnego języka programowania. Dzięki temu możemy łatwiej przeanalizować i zweryfikować poprawność algorytmu przed jego implementacją.

2.1.4 Sprawdzenie poprawności algorytmu poprzez ołówkowe rozwiązanie problemu

Aby przekonać się, że zaproponowany algorytm rzeczywiście jest w stanie rozwiązać zadany problem, wystarczy kartka i ołówek”. Przeanalizowanie jego działania krok po kroku, zgodnie z tym, co zostało zapisane w postaci pseudokodu, pozwala wykryć trywialne błędy i niespójności jeszcze na początkowym etapie pracy nad projektem. Dzięki temu ograniczamy czas spędzony nad samą implementacją, unikając potencjalnych problemów i błędów w kodzie. Taka analiza ołówkowa” jest kluczowym etapem weryfikacji poprawności algorytmu przed przystąpieniem do jego implementacji i konkretyzacji w języku programowania.

Prezentacja poprawnego wykonania tej części zadania zostanie przedstawiona na przykładzie danych wejściowych.

i	j	k	szerokość	minSzerokość	maksymalnePole
0	0	0	1	1	1
0	0	1	1	1	1
0	0	2	1	1	2
0	2	2	1	1	3
0	2	3	1	1	3
0	2	4	1	1	3
1	2	4	2	1	3
2	2	4	3	3	3
2	2	4	3	3	6

Dopiero teraz możemy przystąpić do programowania (!), lecz zanim to zrobimy, warto jeszcze zastanowić się nad złożonością opracowanego algorytmu i przewidzieć prosta analizę teoretyczną. Taka analiza pozwoli lepiej zrozumieć, jakie efekty możemy uzyskać i czy nasz algorytm nadaje się do użycia w praktycznych zastosowaniach. Dzięki temu będziemy mogli nie tylko poprawić nasz kod wejściowy, ale też działać sprawnie nawet dla dużych zestawów danych.

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej

Analizując algorytm można zauważyć, że podstawowa jego operacja będzie sprawdzenie wartości w tablicy `matrix` oraz obliczenie szerokości i pola prostokąta. Łatwo policzyć, ile operacji tego rodzaju zostanie wykonanych.

Dla każdego elementu tablicy `matrix[i][j]`, algorytm sprawdza wszystkie możliwe prostokąty zaczynające się od tego elementu.

W najgorszym przypadku, dla każdego elementu tablicy `matrix[i][j]`, algorytm przechodzi przez wszystkie wiersze poniżej i wszystkie kolumny na prawo od `j`.

A więc całkowita liczba operacji to suma operacji dla wszystkich elementów tablicy. Dla tablicy o wymiarach $M \times N$:

- Dla $i = 0, j = 0$ będzie to $M \times N$ operacji.
- Dla $i = 0, j = 1$ będzie to $(M - 1) \times N$ operacji.
- Dla $i = 1, j = 0$ będzie to $M \times (N - 1)$ operacji.

• ...

Całkowita liczba operacji to suma operacji dla wszystkich elementów tablicy, co można oszacować jako:

$$\sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (M-i) \cdot (N-j)$$

Przypomnijmy sobie zależność na sume kolejnych liczb naturalnych, możemy oszacować, że dla tablicy o wymiarach $M \times N$ algorytm będzie musiał wykonać operacje proporcjonalne do:

$$O(M^2 \times N^2)$$

Powiemy więc, że złożoność czasowa algorytmu wynosi $O(M^2 \cdot N^2)$, a więc czas jego wykonania będzie proporcjonalny do kwadratu liczby wierszy i kolumn tablicy wejściowej.

2.1.6 Implementacja algorytmu

Tutaj moglibyśmy przystąpić do implementacji algorytmu w wybranym przez siebie języku, jednak zanim to zrobimy, spróbujmy wywnioskować, ile czasu zajmie rozwiązanie dla danych wejściowych o rozmiarze $M^2 \cdot N^2$. Implementacja zajmiemy się w momencie, gdy będziemy mieli opracowane wszystkie szczegóły dotyczące tego, jak algorytm działa oraz jakie operacje powinien wykonać.

2.2. Rozwiązanie - próba druga (nieco bardziej finezyjna)

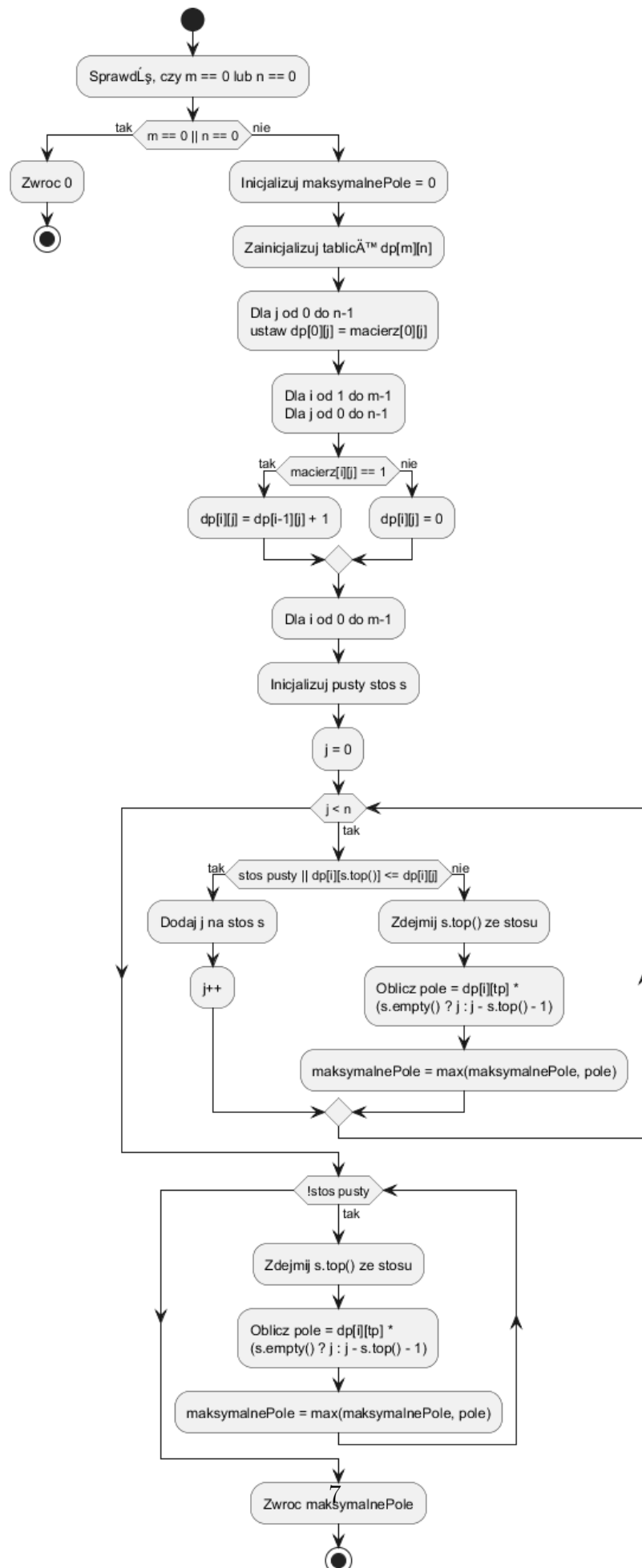
2.2.1. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego

Dogłębna analiza problemu często pozwala dostrzec możliwości poprawy wydajności algorytmu. W tym przypadku możemy rozważyć podejście polegające na przechodzeniu tablicy od początku (od lewej strony), jednocześnie przechowywać wartość największego prostokąta znalezionego do tej pory (`aktualny_max`) oraz maksymalne pole prostokąta (`maks_pole`).

Jeśli podczas przeglądania tablicy natrafimy na element, który nie może być częścią większego prostokąta, aktualizujemy wartość `aktualny_max`. Jeśli element nie może być częścią większego prostokąta, obliczamy pole prostokąta zbudowanego z jedynek, który kończy się na tym elemencie. Jeśli pole to jest większe niż obecna wartość `maks_pole`, aktualizujemy `maks_pole`.

Możemy na przykład zastosować dynamiczne programowanie, aby zoptymalizować nasz algorytm. Przechodząc przez tablice, możemy przechowywać wysokości kolumn, szerokości wierszy oraz największe prostokąty, które możemy utworzyć do tej pory. Dzięki temu możemy zmniejszyć złożoność czasową algorytmu.

2.2.2. Schemat blokowy algorytmu



2.2.3. Algorytm zapisany w pseudokodzie

```
Jeśli m == 0 lub n == 0
    Zwróć 0

maksymalnePole = 0
dp = nowa macierz o wymiarach m x n

// Inicjalizacja pierwszego wiersza
Dla j od 0 do n-1
    dp[0][j] = macierz[0][j]

// Inicjalizacja pozostałych wierszy
Dla i od 1 do m-1
    Dla j od 0 do n-1
        Jeśli macierz[i][j] == 1
            dp[i][j] = dp[i-1][j] + 1
        Inaczej
            dp[i][j] = 0

// Obliczanie maksymalnego pola prostokata dla każdego wiersza
Dla i od 0 do m-1
    Stwórz pusty stos s
    j = 0
    Dopóki j < n
        Jeśli stos s jest pusty lub dp[i][s.top()] <= dp[i][j]
            s.push(j)
            j = j + 1
        Inaczej
            tp = s.top()
            s.pop()
            pole = dp[i][tp] * (jeśli stos s jest pusty ? j : j - s.top() - 1)
            maksymalnePole = max(maksymalnePole, pole)
    Dopóki stos s nie jest pusty
        tp = s.top()
        s.pop()
        pole = dp[i][tp] * (jeśli stos s jest pusty ? j : j - s.top() - 1)
        maksymalnePole = max(maksymalnePole, pole)

Zwróć maksymalnePole
```

2.2.4. Ołówkowe” sprawdzenie poprawności algorytmu nr 2

i	j	s (stos)	dp[i][j]	pole	maksymalnePole
0	0	[0]	1	-	0
0	1	[0, 1]	0	-	0
0	2	[0, 2]	1	-	0
0	3	[0, 2, 3]	0	-	0
0	4	[0, 2, 4]	0	-	0
0	-	[0, 2]	-	1	1
0	-	[0]	-	1	1
0	-	[]	-	1	1
1	0	[0]	2	-	1
1	1	[0, 1]	0	-	1
1	2	[0, 2]	2	-	1
1	3	[0, 2, 3]	1	-	1
1	4	[0, 2, 3, 4]	1	-	1
1	-	[0, 2, 3]	-	1	1
1	-	[0, 2]	-	2	2
1	-	[0]	-	2	2
1	-	[]	-	2	2
2	0	[0]	3	-	2
2	1	[0, 1]	1	-	2
2	2	[0, 1, 2]	3	-	2
2	3	[0, 1, 2, 3]	2	-	2
2	4	[0, 1, 2, 3, 4]	2	-	2
2	-	[0, 1, 2, 3]	-	2	2
2	-	[0, 1, 2]	-	4	4
2	-	[0, 1]	-	6	6
2	-	[0]	-	6	6
2	-	[]	-	6	6
3	0	[0]	4	-	6
3	1	[0, 1]	0	-	6
3	2	[0, 1, 2]	0	-	6
3	3	[0, 1, 2, 3]	3	-	6
3	4	[0, 1, 2, 3, 4]	0	-	6
3	-	[0, 1, 2, 3]	-	3	6
3	-	[0, 1, 2]	-	3	6
3	-	[0, 1]	-	3	6
3	-	[0]	-	3	6
3	-	[]	-	3	6

2.2.5. Teoretyczne oszacowanie złożoności obliczeniowej dla algorytmu 2.

Aby oszacować złożoność obliczeniową dla drugiego algorytmu, który wykorzystuje dynamiczne programowanie, przeanalizujemy jego działanie krok po kroku.

1. Inicjalizacja zmiennych:

Tworzymy tablice `dp` o długości równej liczbie kolumn macierzy, co zajmuje $O(N)$ czasu.

2. Iteracja przez wiersze macierzy:

Dla każdego wiersza aktualizujemy tablice `dp`, co zajmuje $O(N)$ czasu.

3. Obliczanie największego prostokata w każdym wierszu:

Dla każdego wiersza obliczamy największy prostokąt zbudowany z jedynek, korzystając z algorytmu do znajdowania największego prostokata w histogramie. To zajmuje $O(N)$ czasu.

Ponieważ iterujemy przez wszystkie wiersze (M) i dla każdego wiersza wykonujemy operacje w czasie $O(N)$, całkowita złożoność czasowa algorytmu wynosi:

$$O(M \cdot N)$$

W porównaniu do podejścia *brute force*, które miało złożoność $O(M^2 \cdot N^2)$, algorytm wykorzystujący dynamiczne programowanie jest znacznie bardziej wydajny.

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

2.3.1. Prosta implementacja

Rozważamy na początek najprostsze podejście, w którym cały program umieszczamy w jednym pliku. Aby zachować modularność, do algorytmu do znajdowania największego prostokata z jedynek w tablicy zapiszemy w postaci oddzielnych funkcji: jeden oparty na podejściu histogramowym, a drugi na dynamicznym programowaniu. Poprawność i efektywność obu metod można przetestować, wywołując je dla tych samych danych wejściowych.

Przykład takiego programu wraz z wynikami działania obu algorytmów przedstawia się następująco:

```

#include <iostream>
#include <algorithm>
#include <fstream>
#include <cstdlib>
#include <stack>

using namespace std;

// Funkcja obliczająca maksymalne pole prostokąta metodą brute force
int maksymalnePoleProstokataBruteForce(int** macierz, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }

    int maksymalnePole = 0;

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (macierz[i][j] == 1) {
                int minSzerokosc = n;
                for (int k = i; k < m && macierz[k][j] == 1; ++k) {
                    int szerokosc = 0;
                    while (j + szerokosc < n && macierz[k][j + szerokosc] == 1) {
                        ++szerokosc;
                    }
                    minSzerokosc = min(minSzerokosc, szerokosc);
                    maksymalnePole = max(maksymalnePole, minSzerokosc * (k - i + 1));
                }
            }
        }
    }

    return maksymalnePole;
}

// Funkcja obliczająca maksymalne pole prostokąta metodą dynamicznego programowania
int maksymalnePoleProstokataDP(int** macierz, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }

    int maksymalnePole = 0;
    int dp[m][n];

```

```

// Inicjalizacja pierwszego wiersza
for (int j = 0; j < n; ++j) {
    dp[0][j] = macierz[0][j];
}

// Inicjalizacja pozostałych wierszy
for (int i = 1; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        if (macierz[i][j] == 1) {
            dp[i][j] = dp[i-1][j] + 1;
        } else {
            dp[i][j] = 0;
        }
    }
}

// Obliczanie maksymalnego pola prostokąta dla każdego wiersza
for (int i = 0; i < m; ++i) {
    stack<int> s;
    int j = 0;
    while (j < n) {
        if (s.empty() || dp[i][s.top()] <= dp[i][j]) {
            s.push(j++);
        } else {
            int tp = s.top();
            s.pop();
            int pole = dp[i][tp] * (s.empty() ? j : j - s.top() - 1);
            maksymalnePole = max(maksymalnePole, pole);
        }
    }
    while (!s.empty()) {
        int tp = s.top();
        s.pop();
        int pole = dp[i][tp] * (s.empty() ? j : j - s.top() - 1);
        maksymalnePole = max(maksymalnePole, pole);
    }
}

return maksymalnePole;
}

int main() {
    ifstream inputFile("cyferki.txt");

```

```

ifstream inputFile("cyferki.txt");
ofstream outputFile("cyferki_wynik.txt");

if (!inputFile) {
    cerr << "Nie mozna otworzyc pliku wejscowego!" << endl;
    return 1;
}

int m = 5, n = 5;
int** macierz = new int*[m];
for (int i = 0; i < m; ++i) {
    macierz[i] = new int[n];
}

// Wczytywanie danych wejściowych z pliku
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        inputFile >> macierz[i][j];
    }
}

inputFile.close();

int wynikBruteForce = maksymalnePoleProstokataBruteForce(macierz, m, n);
int wynikDP = maksymalnePoleProstokataDP(macierz, m, n);

cout << "Najwieksze pole prostokata z jedynek (Brute Force): " << wynikBruteForce << endl;
cout << "Najwieksze pole prostokata z jedynek (Dynamic Programming): " << wynikDP << endl;

// Zapis wyników do pliku
outputFile << "Najwieksze pole prostokata z jedynek (Brute Force): " << wynikBruteForce << endl;
outputFile << "Najwieksze pole prostokata z jedynek (Dynamic Programming): " << wynikDP << endl;
outputFile.close();

for (int i = 0; i < m; ++i) {
    delete[] macierz[i];
}
delete[] macierz;

return 0;
}

```

2.3.2. Testy „niewygodnych” zestawów danych

Kod powyżej przedstawia proste wywołanie dwóch funkcji w celu sprawdzenia ich działania dla niewielkiego, "recznie zdefiniowanego" zestawu danych testowych. Ten program moglibyśmy uzupełnić o kilka dodatkowych testów, sprawdzających działanie algorytmów dla specyficznych (z punktu widzenia zadanego algorytmu) zestawów danych, aby przekonać się, że jest on odporny na "niewygodne" zestawy danych.

```

[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

```

Tablica ta zawiera różne kombinacje jedynek i zer, co stanowi wyzwanie dla algorytmów. W szczególności, zawiera ona ciągi wartości nierosnących, które mogą być trudne do przetworzenia. Testowanie algorytmów na takich danych pozwala upewnić się, że są one odporne na "niewygodne" zestawy danych i działają poprawnie w różnych scenariuszach.

2.3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej

O wiele ciekawsze będzie przeprowadzenie bardziej zaawansowanych testów, które pozwolą potwierdzić skuteczność algorytmu obliczającego maksymalne pole prostokąta z jedynek w macierzy binarnej. Aby to osiągnąć, porównamy czasy działania algorytmu dla różnych zestawów danych wejściowych. Dla rosnącej liczby danych wejściowych (n) zgromadzimy czasy obliczeń ($t(n)$). Te wyniki następnie można przedstawić w formie tabeli lub wykresu.

Uwaga: Aby zauważyć istotne różnice w czasach działania algorytmu, zestawy danych muszą być odpowiednio duże.

W tym celu potrzebne będzie stworzenie odpowiedniego kodu pomocniczego, którego zadaniem będzie:

Generowanie zestawów danych testowych Zapamiętanie wyników testów Wyświetlenie wyników testów Najwygodniej będzie zapisać poszczególne zadania w postaci oddzielnych funkcji. W najprostszym podejściu, możemy zrealizować te zadania jako oddzielne funkcje w naszym kodzie.

1. Generowanie losowych danych testowych:

```
void Generuj(int *tab, int n, int nmax) {
    for (int i = 0; i < n; i++)
        tab[i] = rand() % nmax;
}
```

2. Wyświetlanie tablicy z danymi wejściowymi: Jest to funkcja pomocnicza, której poniżej nie używamy, ale może ona być przydatna w fazie implementacji w celu sprawdzenia, czy funkcja generująca dane testowe działa poprawnie.

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

W celu niekomplikowania tego dokumentu ponad miarę, na razie pozostaniemy na tych dwóch funkcjach, a część kodu zawierająca testy umieścimy bezpośrednio w funkcji main.

```

#include <iostream>
#include <algorithm>
#include <fstream>
#include <cstdlib>
#include <stack>
#include <ctime>

using namespace std;

// Funkcja obliczająca maksymalne pole prostokąta metodą brute force
int maksymalnePoleProstokataBruteForce(int** macierz, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }

    int maksymalnePole = 0;

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (macierz[i][j] == 1) {
                int minSzerokosc = n;
                for (int k = i; k < m && macierz[k][j] == 1; ++k) {
                    int szerokosc = 0;
                    while (j + szerokosc < n && macierz[k][j + szerokosc] == 1) {
                        ++szerokosc;
                    }
                    minSzerokosc = min(minSzerokosc, szerokosc);
                    maksymalnePole = max(maksymalnePole, minSzerokosc * (k - i + 1));
                }
            }
        }
    }

    return maksymalnePole;
}

// Funkcja obliczająca maksymalne pole prostokąta metodą dynamicznego programowania
int maksymalnePoleProstokataDP(int** macierz, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }

    int maksymalnePole = 0;
    int dp[m][n];

```



```

// Inicjalizacja pierwszego wiersza
for (int j = 0; j < n; ++j) {
    dp[0][j] = macierz[0][j];
}

// Inicjalizacja pozostałych wierszy
for (int i = 1; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        if (macierz[i][j] == 1) {
            dp[i][j] = dp[i-1][j] + 1;
        } else {
            dp[i][j] = 0;
        }
    }
}

// Obliczanie maksymalnego pola prostokąta dla każdego wiersza
for (int i = 0; i < m; ++i) {
    stack<int> s;
    int j = 0;
    while (j < n) {
        if (s.empty() || dp[i][s.top()] <= dp[i][j]) {
            s.push(j++);
        } else {
            int tp = s.top();
            s.pop();
            int pole = dp[i][tp] * (s.empty() ? j : j - s.top() - 1);
            maksymalnePole = max(maksymalnePole, pole);
        }
    }
    while (!s.empty()) {
        int tp = s.top();
        s.pop();
        int pole = dp[i][tp] * (s.empty() ? j : j - s.top() - 1);
        maksymalnePole = max(maksymalnePole, pole);
    }
}

return maksymalnePole;
}

void Generuj(int *tab, int n, int nmax) {

```

```

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

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

int main() {
    ifstream inputFile("cyferki.txt");
    ofstream outputFile("cyferki_wynik.txt");

    if (!inputFile) {
        cerr << "Nie można otworzyć pliku wejściowego!" << endl;
        return 1;
    }

    int m = 5, n = 5;
    int** macierz = new int*[m];
    for (int i = 0; i < m; ++i) {
        macierz[i] = new int[n];
    }

    // Wczytywanie danych wejściowych z pliku
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            inputFile >> macierz[i][j];
        }
    }

    inputFile.close();

    clock_t begin, end;
    double czasBruteForce, czasDP;

    // Pomiar czasu dla algorytmu brute force
    begin = clock();
    int wynikBruteForce = maksymalnePoleProstokataBruteForce(macierz, m, n);
    end = clock();
    czasBruteForce = double(end - begin) / CLOCKS_PER_SEC;
}

```

```

// Pomiar czasu dla algorytmu dynamicznego
begin = clock();
int wynikDP = maksymalnePoleProstokataDP(macierz, m, n);
end = clock();
czasDP = double(end - begin) / CLOCKS_PER_SEC;

cout << "Najwieksze pole prostokata z jedynek (Brute Force): " << wynikBruteForce << " (czas: " << czasBruteForce << " s)" << endl;
cout << "Najwieksze pole prostokata z jedynek (DP): " << wynikDP << " (czas: " << czasDP << " s)" << endl;

// Zapis wyników do pliku
outputFile << "Najwieksze pole prostokata z jedynek (Brute Force): " << wynikBruteForce << " (czas: " << czasBruteForce << " s)" << endl;
outputFile << "Najwieksze pole prostokata z jedynek (DP): " << wynikDP << " (czas: " << czasDP << " s)" << endl;
outputFile.close();

for (int i = 0; i < m; ++i) {
    delete[] macierz[i];
}
delete[] macierz;

return 0;

```

L.p.	n	t_1 [s] (Brute Force)	t_2 [s] (Dynamic Programming)
1	500	0.216731	0.179145
2	1000	0.995965	0.615200
3	1500	2.266834	1.442626
4	2000	3.948277	2.547369

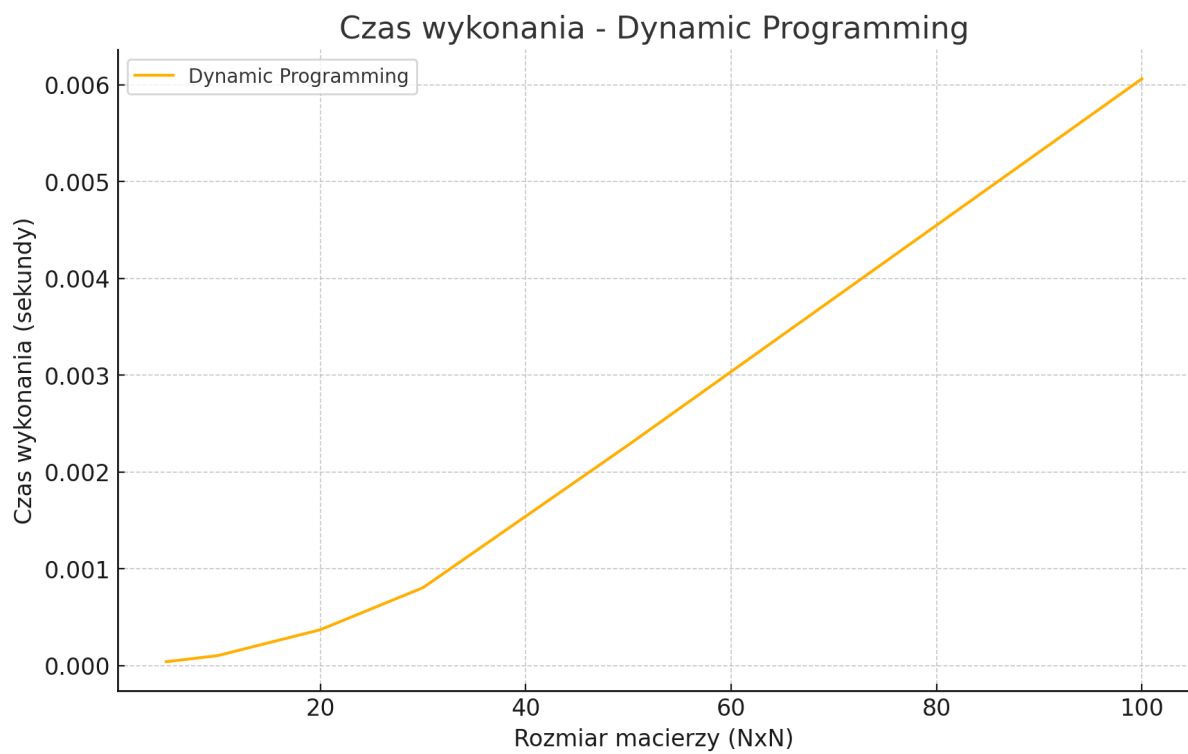
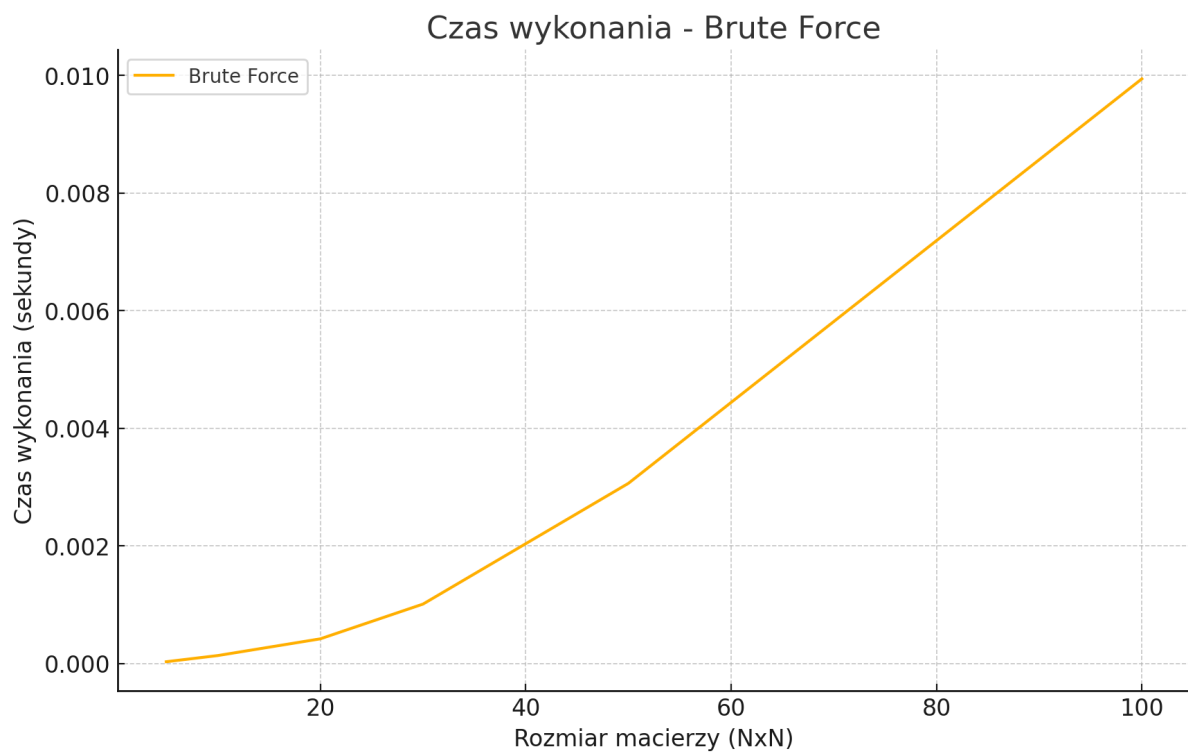
Table 1: Tabela czasów wykonania dla programów (ograniczone dane do 2000x2000)

Jak wynika z powyższej tabeli, algorytmy przedstawione w kodzie różnią się znacząco pod względem czasu wykonywania obliczeń dla tych samych zestawów danych wejściowych. Wersja naiwna (algorytm brute force) wymaga znacznie więcej czasu w porównaniu do wersji zoptymalizowanej (algorytm dynamiczny). Na przykład, dla macierzy o rozmiarze 2000×2000 czas działania algorytmu brute force wynosi około 4 sekund, podczas gdy algorytm dynamiczny kończy obliczenia w mniej niż 3 sekundy.

Różnice te wynikają z odmiennych złożoności obliczeniowych obu algorytmów. Algorytm brute force charakteryzuje się złożonością kwadratową względem liczby elementów, co powoduje gwałtowny wzrost czasu obliczeń przy zwiększaniu rozmiaru danych wejściowych. Natomiast algorytm dynamiczny wykorzystuje podejście optymalizacyjne, redukując czas wykonywania dzięki liniowej zależności od liczby elementów w każdym wierszu macierzy.

Wyniki obliczeń można również zilustrować graficznie. Na wykresach przedstawiających zależność czasu obliczeń $t(n)$ od liczby danych wejściowych n widać, że czasy algorytmu brute force układają się wzdłuż paraboli, co potwierdza teoretyczna złożoność obliczeniowa $O(n^3)$. Natomiast punkty odpowiadające algorytmowi dynamicznemu tworzą niemal linię prostą, co odpowiada złożoności $O(n^2)$.

Na rysunkach poniżej za pomocą czerwonych punktów oznaczono wyniki pomiarów czasu dla macierzy o rozmiarach 500×500 , 1000×1000 , 1500×1500 , oraz 2000×2000 . Linie ciągłe przedstawiają aproksymacje wyników: krzywa paraboliczna dla algorytmu brute force oraz linie proste dla algorytmu dynamicznego. Obserwacje te dowodzą, że teoretyczna analiza złożoności algorytmów znajduje swoje odzwierciedlenie w wynikach praktycznych.



2.3.4. Testy wydajności algorytmów - złożoności optymistyczne/pesymistyczne

Wyniki testów, przedstawione na wykresach, pokazują, że algorytm brute force jest szczególnie podatny na pesymistyczne dane, gdzie czas działania znacząco wzrasta. Algorytm dynamiczny wykazuje większą stabilność, jednak również zauważalnie dłużej działa w mniej korzystnych przypadkach. Porównanie czasów obliczeń potwierdza przewidywane różnice wynikające ze złożoności obliczeniowej obu algorytmów.

2.3. Biografia

- <https://www.w3schools.com>.
- <https://www.learnlatex.org>
- <https://www.overleaf.com/learn>
- <https://en.wikibooks.org/wiki/LaTeX>
- <https://plantuml.com>