Wydział Matematyki i Fizyki Stosowanej
Politechniki Rzeszowskiej

DAWID KOTULA

Znalezienie najwiekszego prostokata zbudowanego z jedynek

Opiekun pracy:
dr. inż. prof. Mariusz Borkowski

Rzeszów, 2025

1. Treść zadania

Dla tablicy `MxN` wypełnionej zerami lub jedynkami znaleźć pole najwiekszego prostokata 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)
```

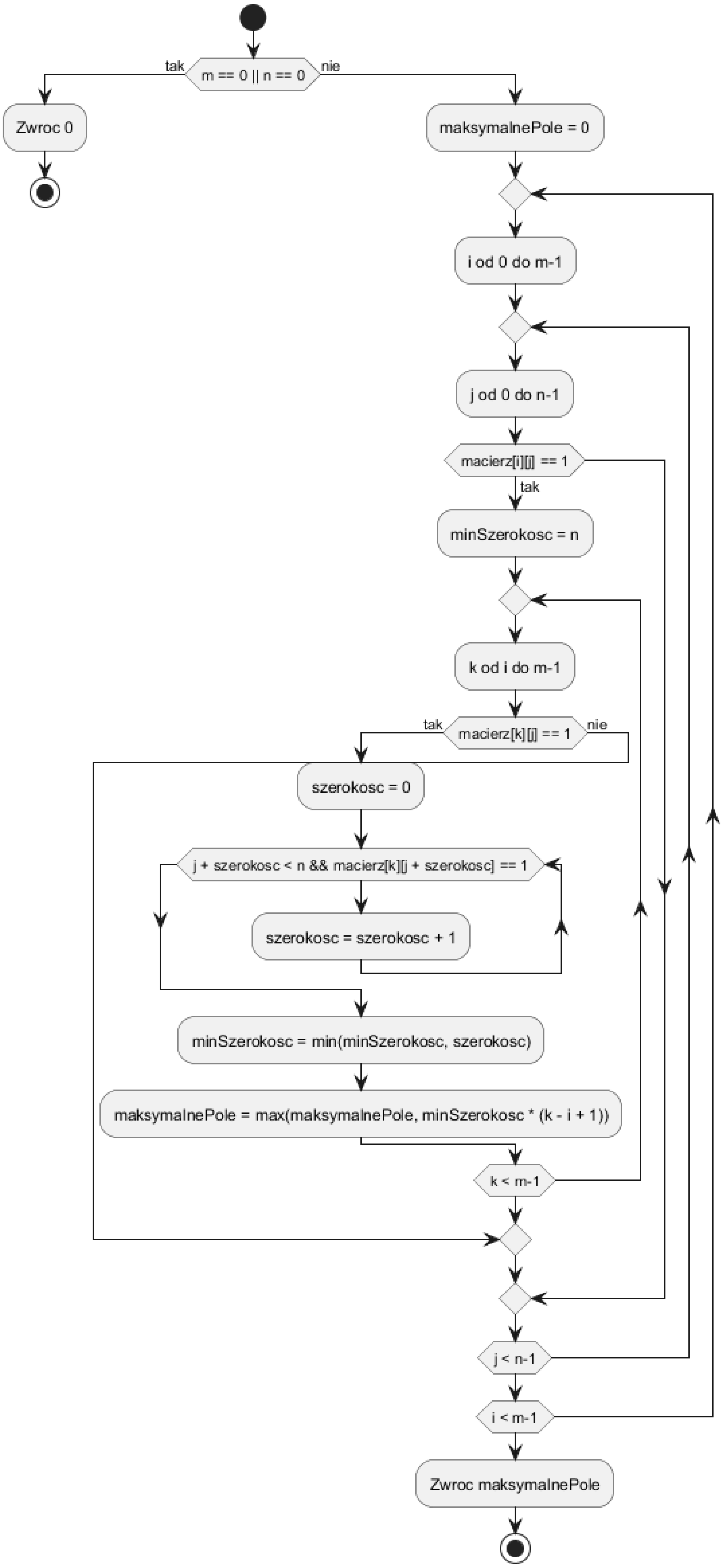
2. Rozwiazanie – podejście pierwsze (*brute force*)

2.1.1 Analiza Problemu

W podejściu *brute force* do znalezienia najwiekszego prostokata z jedynek w tablicy `MxN` zaczynamy od utworzenia zmiennej pomocniczej `maks_pole`, w której bedziemy przechowywać aktualna wartość maksymalnego pola prostokata. Nastepnie, metodycznie, sprawdzamy wszystkie możliwe prostokaty w tablicy, bo obliczamy pole każdego prostokata składajacego sie z jedynek i jeśli bedzie ono wieksze od wartości przechowywanej w zmiennej `maks_pole`, przypisujemy je do tej zmiennej. Należy zwrócić uwage na problem inicjalizacji zmiennej `maks_pole` właściwa wartościa, która poczatkowo powinna wynosić 0. Jeśli algorytm zakończy analize, oznacza to, że w tablicy nie znaleziono żadnego prostokata składajacego sie z jedynek. Inna rzecza, która powinien zwrócić uwage algorytm, jest sytuacja, gdy podana tablica jest pusta. Na ten wypadek w algorytmie zostanie dodana instrukcja warunkowa `if`, która na poczatku procesu petli po typie `Mtbl` sprawdzi, czy tablica ma elementy.

1. Dane wejściowe: Dane wejściowe w naszym algorytmie użytkownika:
   - Strukture danych typu tablica (macierz), przechowujaca wartości tablicy `Mtbl` z jedynkami i zerami.
2. Dane wyjściowe: Algorytm zwróci wartość najwiekszego pola prostokata 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 pomiedzy analiza problemu a opracowaniem algorytmu, a sama implementacja w konkretnym jezyku programowania, która zostanie przedstawiona w kolejnych rozdziałach. Pseudokod poprawia zrozumienie logiki algorytmu bez konieczności skupiania sie na składni konkretnego jezyka programowania. Dzieki temu możemy łatwiej przeanalizować i zweryfikować poprawność algorytmu przed jego implementacja.

2.1.4 Sprawdzenie poprawności algorytmu poprzez ołówkowe" rozwiazanie problemu

Aby przekonać sie, że zaproponowany algorytm rzeczywiście jest w stanie rozwiazać zadany problem, wystarczy kartka i ołówek". Przeanalizowanie jego działania krok po kroku, zgodnie z tym, co zostało zapisane w postaci pseudokodu, pozwala wykryć trywialne błedy i niespójności jeszcze na poczatkowym etapie pracy nad projektem. Dzieki temu ograniczamy czas spedzony nad sama implementacja, unikajac potencjalnych problemów i błedów w kodzie. Taka analiza ołówkowa" jest kluczowym etapem weryfikacji poprawności algorytmu przed przystapieniem do jego implementacji i konkretyzacji w jezyku programowania.

Prezentacja poprawnego wykonania tej cześci 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 przystapić do programowania (!), lecz zanim to zrobimy, warto jeszcze zastanowić sie nad złożonościa opracowanego algorytmu i przewidzieć prosta analize teoretyczna. Taka analiza pozwoli lepiej zrozumieć, jakie efekty możemy uzyskać i czy nasz algorytm nadaje sie do użycia w praktycznych zastosowaniach. Dzieki temu bedziemy mogli nie tylko poprawić nasz kod wejściowy, ale też działać sprawnie nawet dla dużych zestawów danych.

2.1.5 Teoretyczne oszacowanie złożoności obliczeniowej

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

Dla każdego elementu tablicy `matrix[i][j]`, algorytm sprawdza wszystkie możliwe prostokaty zaczynajace sie 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 wiec całkowita liczba operacji to suma operacji dla wszystkich elementów tablicy. Dla tablicy o wymiarach $M \times N$:

- Dla $i = 0, j = 0$ bedzie to $M \times N$ operacji.
- Dla $i = 0, j = 1$ bedzie to $(M - 1) \times N$ operacji.
- Dla $i = 1, j = 0$ bedzie 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 bedzie musiał wykonać operacje proporcjonalne do:

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

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

2.1.6 Implementacja algorytmu

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

2.2. Rozwiazanie - próba druga (nieco bardziej finezyjna)

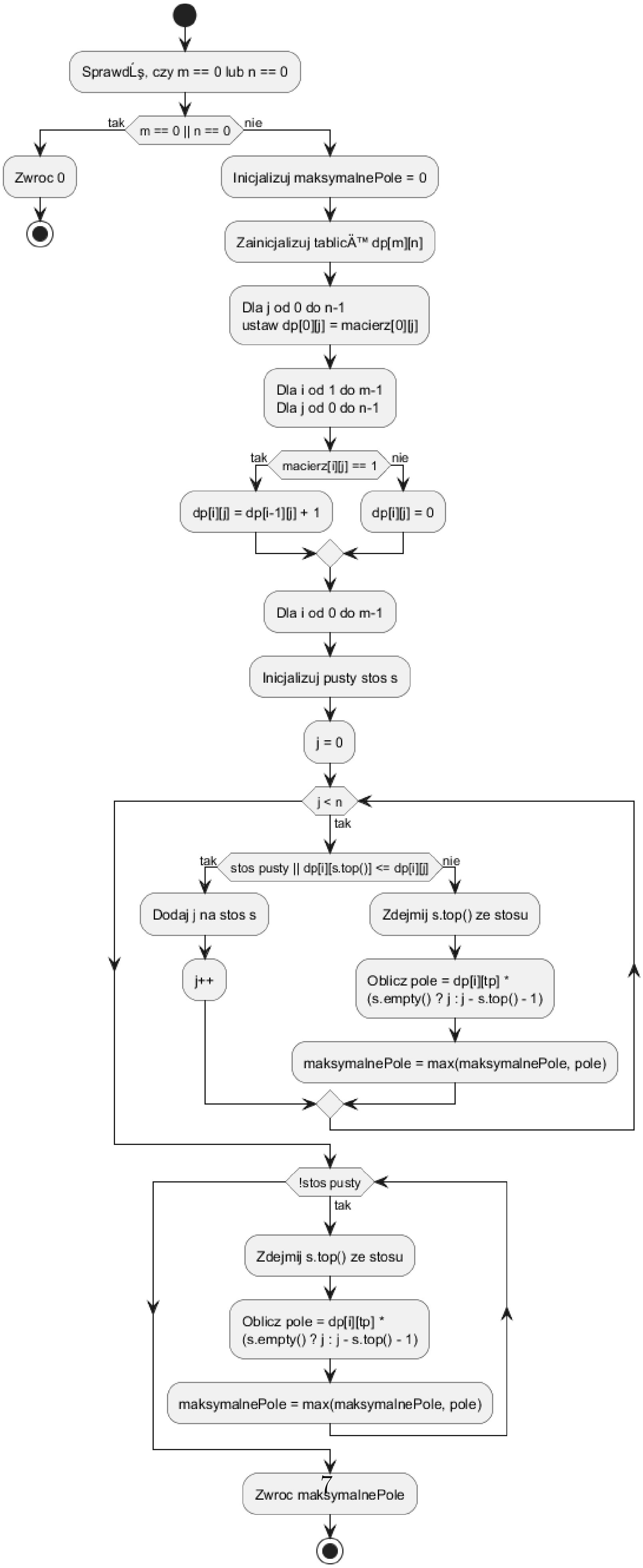
2.2.1. Ponowne przemyślenie problemu i próba wymyślenia algorytmu wydajniejszego

Dogłebna analiza problemu czesto pozwala dostrzec możliwości poprawy wydajności algorytmu. W tym przypadku możemy rozważyć podejście polegajace na przechodzeniu tablicy od poczatku (od lewej strony), jednocześnie przechowujac wartość najwiekszego prostokata znalezionego do tej pory (`aktualny_max`) oraz maksymalne pole prostokata (`maks_pole`).

Jeśli podczas przegladania tablicy natrafimy na element, który nie może być cześcia wiekszego prostokata, aktualizujemy wartość `aktualny_max`. Jeśli element nie może być cześcia wiekszego prostokata, obliczamy pole prostokata zbudowanego z jedynek, który kończy sie na tym elemencie. Jeśli pole to jest wieksze niż obecna wartość `maks_pole`, aktualizujemy `maks_pole`.

Możemy na przykład zastosować dynamiczne programowanie, aby zoptymalizować nasz algorytm. Przechodzac przez tablice, możemy przechowywać wysokości kolumn, szerokości wierszy oraz najwieksze prostokaty, które możemy utworzyć do tej pory. Dzieki temu możemy zmniejszyć złożoność czasowa 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 pozosta␣lych 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 obliczeniowa dla drugiego algorytmu, który wykorzystuje dynamiczne programowanie, przeanalizujmy 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 najwiekszego prostokata w każdym wierszu:**
   Dla każdego wiersza obliczamy najwiekszy prostokat zbudowany z jedynek, korzystajac z algorytmu do znajdowania najwiekszego 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 wykorzystujacy dynamiczne programowanie jest znacznie bardziej wydajny.

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

2.3.1. Prosta implementacja

Rozważamy na poczatek najprostsze podejście, w którym cały program umieszczamy w jednym pliku. Aby zachować modularność, do algorytmu do znajdowania najwiekszego 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, wywołujac je dla tych samych danych wejściowych.

Przykład takiego programu wraz z wynikami działania obu algorytmów przedstawia sie nastepujaco:

```cpp
#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");
```

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

    if (!inputFile) {
        cerr << "Nie mozna otworzyc pliku wejsciowego!" << 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, sprawdzajacych działanie algorytmów dla specyficznych (z punktu widzenia zadanego algorytmu) zestawów danych, aby przekonać sie, 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 ciagi wartości nierosnacych, które moga być trudne do przetworzenia. Testowanie algorytmów na takich danych pozwala upewnić sie, że sa one odporne na "niewygodne" zestawy danych i działaja poprawnie w różnych scenariuszach.

2.3.3. Testy wydajności algorytmów - eksperymentalne sprawdzenie złożoności czasowej

O wiele ciekawsze bedzie przeprowadzenie bardziej zaawansowanych testów, które pozwola potwierdzić skuteczność algorytmu obliczajacego maksymalne pole prostokata z jedynek w macierzy binarnej. Aby to osiagnać, porównamy czasy działania algorytmu dla różnych zestawów danych wejściowych. Dla rosnacej liczby danych wejściowych (n) zgromadzimy czasy obliczeń (t(n)). Te wyniki nastepnie można przedstawić w formie tabeli lub wykresu.

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

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

Generowanie zestawów danych testowych Zapamietanie wyników testów Wyświetlenie wyników testów Najwygodniej bedzie 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 generujaca 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 miare, na razie pozostaniemy na tych dwóch funkcjach, a cześć kodu zawierajaca testy umieścimy bezpośrednio w funkcji main.

```cpp
#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) {
```

```cpp
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 mozna otworzyc pliku wejsciowego!" << 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;
```

```cpp
    // 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

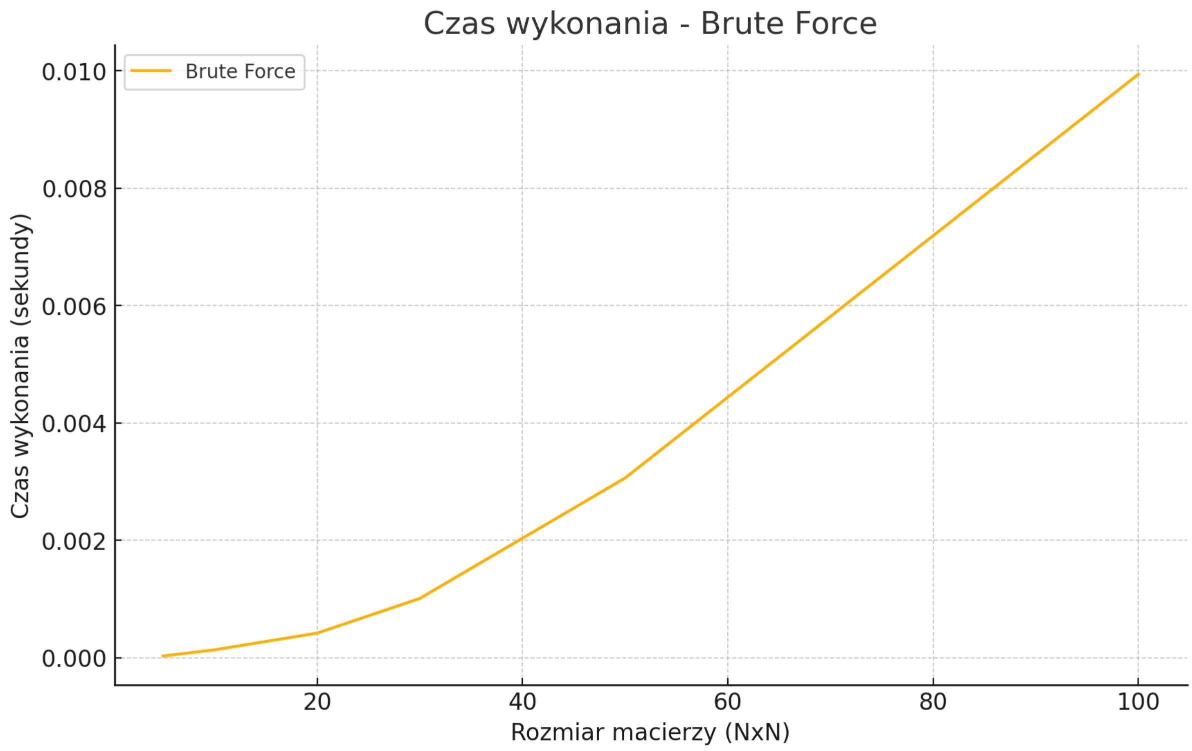
Table 1: Tabela czasów wykonania dla programów (ograniczone dane do 2000x2000)

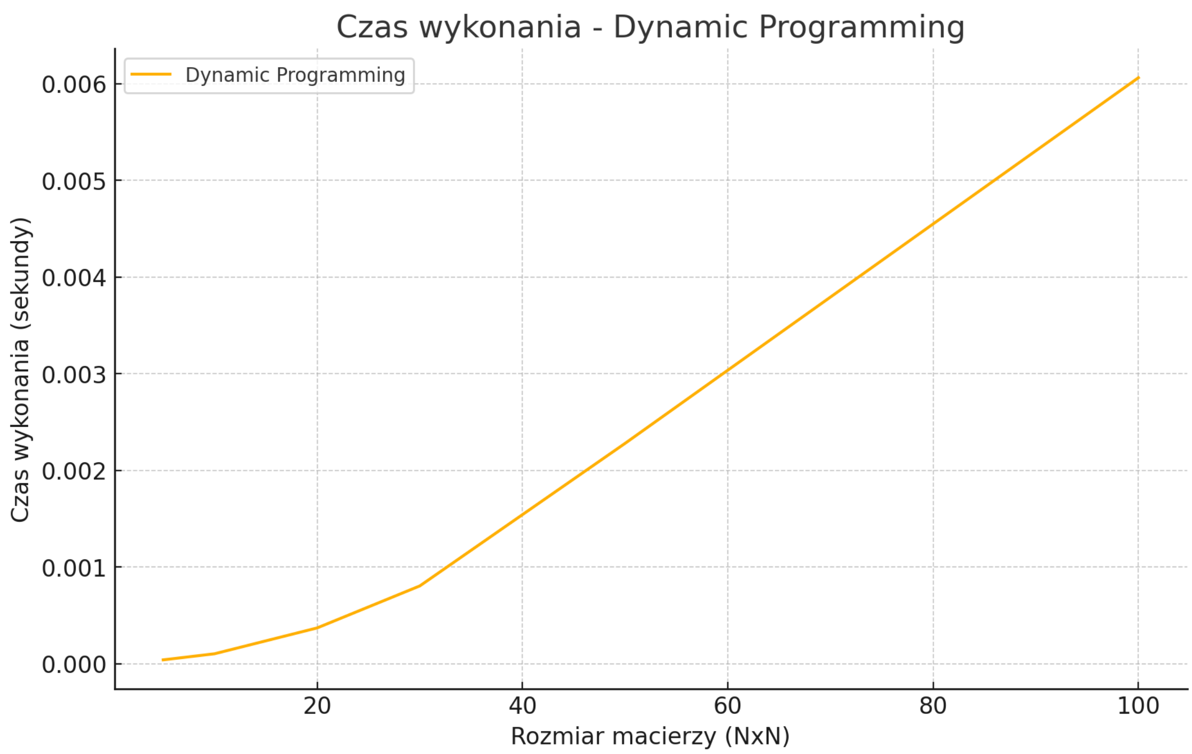
Jak wynika z powyższej tabeli, algorytmy przedstawione w kodzie różnia sie znaczaco pod wzgledem czasu wykonywania obliczeń dla tych samych zestawów danych wejściowych. Wersja naiwna (algorytm brute force) wymaga znacznie wiecej czasu w porównaniu do wersji zoptymalizowanej (algorytm dynamiczny). Na 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 wynikaja z odmiennych złożoności obliczeniowych obu algorytmów. Algorytm brute force charakteryzuje sie złożonościa kwadratowa wzgledem liczby elementów, co powoduje gwałtowny wzrost czasu obliczeń przy zwiekszaniu rozmiaru danych wejściowych. Natomiast algorytm dynamiczny wykorzystuje podejście optymalizacyjne, redukujac czas wykonywania dzieki liniowej zależności od liczby elementów w każdym wierszu macierzy.

Wyniki obliczeń można również zilustrować graficznie. Na wykresach przedstawiajacych zależność czasu obliczeń $t(n)$ od liczby danych wejściowych n widać, że czasy algorytmu brute force układaja sie wzdłuż paraboli, co potwierdza teoretyczna złożoność obliczeniowa $O(n^3)$. Natomiast punkty odpowiadajace algorytmowi dynamicznemu tworza niemal linie prosta, co odpowiada złożoności $O(n^2)$.

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


Czas wykonania - Brute Force
Brute Force
0.010
0.008
0.006
0.004
0.002
0.000
Czas wykonania (sekundy)
20
40
60
80
100
Rozmiar macierzy (NxN)


Czas wykonania - Dynamic Programming
Dynamic Programming
0.006
0.005
0.004
0.003
0.002
0.001
0.000
Czas wykonania (sekundy)
20
40
60
80
100
Rozmiar macierzy (NxN)

2.3.4. Testy wydajności algorytmów - złożoności optymistyczne/pesymistyc

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

2.3. Bibliografia

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