



Programowanie z nawrotami na przykładzie "problemu hetmanów"

Veronika Tronchuk

I rok Informatyki

Nr albumu: 30019

Problem N-hetmanów to jedno z najbardziej znanych i klasycznych zadań algorytmicznych, stosowanych w nauczaniu technik programowania, a w szczególności – rekurencji oraz przeszukiwania z nawrotami (ang. backtracking). Zadanie to polega na rozmieszczeniu N hetmanów na szachownicy o wymiarach $N \times N$ w taki sposób, aby żaden z nich nie atakował innego – czyli aby nie znajdowali się w tej samej kolumnie, wierszu lub na tych samych przekątnych.

Celem sprawozdania było szczegółowe zbadanie i zaimplementowanie rozwiązania wersji problemu, przy wykorzystaniu algorytmu typu backtracking. Główne założenia to:

- zaprojektowanie algorytmu, który umożliwia umieszczanie N hetmanów na planszy N na N bez wzajemnych ataków,
- stworzenie równoległego rozwiązania dla wież – z możliwością wyboru liczby wież mniejszych lub równych N,
- zaprogramowanie aplikacji, która umożliwia wybór trybu działania (hetmany lub wieże),
- pomiar czasu działania programu oraz analiza jego efektywności, przygotowanie wizualnej reprezentacji rozwiązań – szachownicy z rozmieszczonymi figurami

W celu rozwiązania zadania zarówno dla hetmanów, jak i wież, zastosowano klasyczne podejście rekurencyjne z nawrotami. W każdym kroku program próbuje ustawić figurę w jednej z możliwych pozycji w danym wierszu lub kolumnie. Jeśli pozycja jest dozwolona rekursja przechodzi do następnego wiersza czy kolumny. W przeciwnym razie cofa się i testuje inne opcje.

Dzięki temu algorytm „buduje” poprawne rozwiązanie kawałek po kawałku, poruszając się po drzewie decyzyjnym, w którym każda gałąź reprezentuje jedną możliwość ustawienia figury.

Do przechowywania aktualnego stanu planszy wykorzystano wektor dwuwymiarowy `vector<vector<int>> board`, w którym:

- wartość 1 oznacza obecność figury,

- wartość 0 oznacza puste pole.

Dzięki temu możliwe jest szybkie sprawdzenie, czy dana pozycja jest zajęta, a także łatwe cofnięcie się do poprzedniego stanu (ustawienie 0 zamiast 1).

Pierwszym etapem rozbudowy programu było dodanie licznika czasu, który mierzy, jak długo trwa znalezienie rozwiązania problemu – zarówno dla hetmanów, jak i wież. Taki pomiar pozwala później analizować efektywność algorytmu w zależności od rozmiaru planszy i liczby figur.

Aby możliwe było korzystanie z pomiaru czasu, na początku pliku źródłowego dodano dyrektywę:

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;
using namespace chrono;
```

#include <chrono> – pozwala korzystać z narzędzi do pracy z czasem.

using namespace chrono; – umożliwia skrócony zapis (nie trzeba pisać "chrono::" przed każdą funkcją).

Pomiar rozpoczyna się tuż przed wywołaniem funkcji rozwiązującej (solveNQueens lub solveWieża).

```
int main() {
    auto start = chrono::high_resolution_clock::now();
```

high_resolution_clock to najbardziej precyzyjny zegar dostępny w chrono.

Pomiar kończy się po zakończeniu działania funkcji.

```
if (solveNQueens(board, 0, N)) {
    auto stop = chrono::high_resolution_clock::now();
    auto czas = chrono::duration_cast<chrono::milliseconds>(stop - start);
```

Program zapisuje drugi moment czasu – zaraz po zakończeniu obliczeń.

obliczamy różnicę między czasem zakończenia i rozpoczęcia, czyli czas trwania algorytmu.

duration_cast<chrono::milliseconds> zamienia ten czas na milisekundy

Różnica zostaje zapisana jako czas i wypisana na ekranie.

```
printBoard(board, N);  
cout << "czas wykonania: " << czas.count() << endl;
```

Funkcja **count()** zwraca konkretną liczbę milisekund.

Wynik jest wyświetlany użytkownikowi.

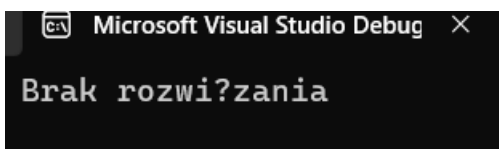
Dalej należało uruchomić program dla różnych wartości N, czyli rozmiarów szachownicy od 1 na 1 do co najmniej 12 na 12, i sprawdzić, czy dla każdej wartości N program znajduje poprawne rozwiązanie oraz ile czasu to zajmuje.

Zadanie to pozwala przeanalizować, jak algorytm zachowuje się przy różnych wielkościach problemu – zarówno pod kątem poprawności, jak i wydajności.

Hetmany

Dla N = 3:

```
int main() {  
    auto start = chrono::high_resolution_clock::now();  
    int N = 3; // Można zmienić na dowolną wartość
```



Dla wartości N = 3 program nie znajduje rozwiązania dla problemu N hetmanów. Nie jest to błąd w programie – jest to matematycznie poprawny wynik.

Nie da się umieścić trzech hetmanów na planszy 3 na 3 w taki sposób,

aby nie atakowały się nawzajem. Każda próba rozmieszczenia kończy się konfliktem na przekątnej, wierszu lub kolumnie. Problem ten został udowodniony matematycznie i jest klasycznym wyjątkiem.

Dla N = 8:

```
int main() {  
    auto start = chrono::high_resolution_clock::now();  
    int N = 8; // Można zmienić na dowolną wartość  
    vector<vector<int>> board(N, vector<int>(N, 0));
```

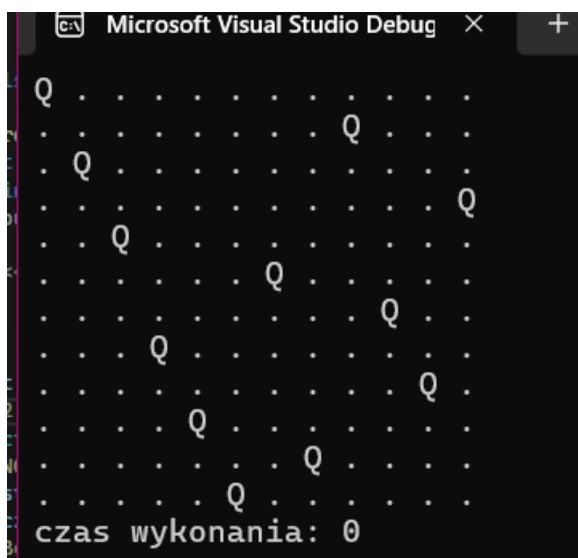


```
Q . . . . . . .  
. . . . . Q .  
. . . . Q . .  
. . . . . Q .  
. Q . . . . .  
. . . Q . . .  
. . . . Q . .  
. . Q . . . .  
czas wykonania: 0
```

po uruchomieniu programu z wartością N = 8, algorytm przeszukiwania z nawrotami poprawnie znajduje jedno z wielu możliwych rozwiązań.

Dla N = 12:

```
int main() {  
    auto start = chrono::high_resolution_clock::now();  
    int N = 12; // Można zmienić na dowolną wartość
```



```
Q . . . . . . . . . . . .  
. . . . . Q . . . . . .  
. . . . . . . . . . Q .  
. . . . Q . . . . . . .  
. . . . . . . Q . . . .  
. . . . . . . . . Q . .  
. . . . . Q . . . . . .  
. . . . . . . . . . Q .  
. . . . Q . . . . . . .  
. . . . . . . . . . . Q  
. . . . . . . . Q . . .  
. . . . . . . . . . . .  
czas wykonania: 0
```

po uruchomieniu programu z wartością $N = 12$, algorytm przeszukiwania z nawrotami poprawnie znajduje jedno z wielu możliwych rozwiązań.

Dla $N = 15$:

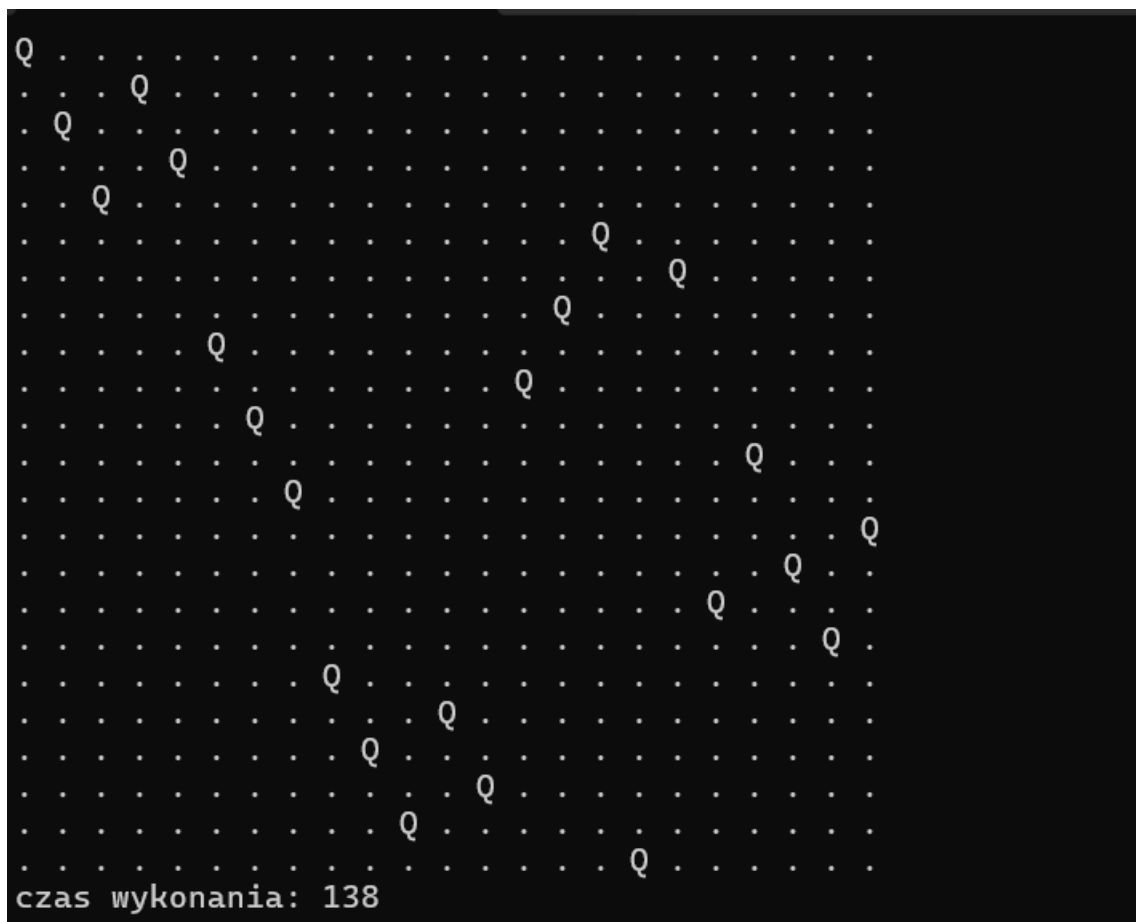
```
int main() {
    auto start = chrono::high_resolution_clock::now();
    int N = 15; // Można zmienić na dowolną wartość
```

[illegible]

po uruchomieniu programu z wartością $N = 15$, algorytm przeszukiwania z nawrotami poprawnie znajduje jedno z wielu możliwych rozwiązań.

Dla $N = 23$:

```
int main() {
    auto start = chrono::high_resolution_clock::now();
    int N = 23; // Można zmienić na dowolną wartość
```



po uruchomieniu programu z wartością $N = 23$, algorytm przeszukiwania z nawrotami poprawnie znajduje jedno z wielu możliwych rozwiązań.

nr	Rozmiar tablicy	ilosc figur	Czas wykonania [ms]
1	3	-	-
2	8	8	0
3	12	12	0
4	15	15	3
5	23	23	138

Najważniejszym czynnikiem wpływającym na **czas** działania programu jest rozmiar szachownicy N na N . Im większe N , tym więcej potencjalnych konfiguracji figur musi zostać rozważonych przez algorytm. W szczególności:

Dla problemu hetmanów, liczba możliwych ustawień rośnie wykładniczo – w najgorszym przypadku złożoność to $O(N!)$, ponieważ każda kolumna

ma N możliwych pozycji.

Dla wież, choć złożoność również jest znacząca, to możliwe rozwiązania można opisać jako permutacje kolumn (co jest prostsze niż analiza przekątnych dla hetmanów).

W przypadku **hetmanów** algorytm musi sprawdzać nie tylko wiersze i kolumny, ale także dwie przekątne (górną lewą i dolną lewą). To powoduje większą liczbę warunków i dłuższy czas sprawdzania pozycji.

W przypadku **wież** wystarczy kontrolować tylko rzędy i kolumny – co sprawia, że sprawdzanie jest szybsze, a zatem ogólny czas wykonania programu jest mniejszy.

Czas wykonania może się nieznacznie różnić w zależności od:

- szybkości procesora,
- obciążenia systemu,
- sposobu kompilacji programu

Dalej należało rozszerzyć program o możliwość wyboru przez użytkownika innej figury – wieży – zamiast domyślnych hetmanów. Oznaczało to wprowadzenie menu początkowego, gdzie użytkownik decyduje, czy chce rozwiązywać problem hetmanów (tryb 1) czy wież (tryb 2).

```
cout << "1 - Queens, 2 - Wieza \n Podaj numer: ";  
cin >> numer;
```

Użytkownik wpisuje 1 lub 2, a program wybiera odpowiedni algorytm.

Następnie w zależności od wpisanej wartości program wykonuje jeden z warunków:


```

if (numer == 1) {
    rozw = solveNQueens(board, 0, N);
}
else if (numer == 2) {
    cout << "Podaj liczbe wiez: ";
    cin >> liczbaWiez;
    rozw = solveWieza(board, 0, N, liczbaWiez);
}
else {
    cout << "Blad" << endl;
    return 1;
}

```

```

bool isSafeWieza(vector<vector<int>>& board, int wiersz, int kolumna, int N) {
    for (int i = 0; i < N; i++) {
        if (board[wiersz][i] == 1)
            return false;
    }
}

```

bool isSafeWieza(...)

bool – typ zwracany: funkcja zwraca true albo false.

sSafeWieza – nazwa funkcji, oznacza „czy bezpieczna pozycja dla wieży”.

vector<vector<int>>& board

Dwuwymiarowy wektor (szachownica) reprezentujący planszę.

Przekazany referencją (&) – czyli bez kopiowania.

int wiersz, int kolumna

Pozycja, w której chcemy umieścić wieżę.

int N

Rozmiar planszy (czyli N×N).

Ciało funkcji:

for (int i = 0; i < N; i++)

Pętla sprawdzająca wszystkie kolumny w wierszu oraz wszystkie wiersze w kolumnie.

if (board[wiersz][i] == 1)

return false;

if (board[i][kolumna] == 1)

return false;

board[wiersz][i] — sprawdzamy kolumny w tym samym wierszu.

board[i][kolumna] — sprawdzamy wiersze w tej samej kolumnie.

== 1 — oznacza, że w tym miejscu już stoi jakaś wieża.

return false;

Jeśli znaleziono już figurę w tej linii — nie można tu nic ustawiać.

```
bool solveWieża(vector<vector<int>>& board, int wiersz, int N, int liczbaWież) {
    if (liczbaWież == 0)
        return true;
    if (wiersz >= N)
        return false;
    for (int kolumna = 0; kolumna < N; kolumna++) {
        if (isSafeWieża(board, wiersz, kolumna, N)) {
            board[wiersz][kolumna] = 1;
            if (solveWieża(board, wiersz + 1, N, liczbaWież - 1))
                return true;
            board[wiersz][kolumna] = 0;
        }
    }
    return solveWieża(board, wiersz + 1, N, liczbaWież);
}
```

bool solveWieża(...)

Funkcja zwraca true, jeśli udało się ustawić wszystkie wieże.

Nazwa oznacza „rozwiąż problem rozmieszczenia wież”.

if (liczbaWież == 0) return true;

Jeśli nie ma już żadnych wież do ustawienia — zakończ (sukces).

To warunek zakończenia rekursji z sukcesem.

if (wiersz >= N) return false;

Jeśli skończyły się wiersze, a wieże nadal zostały — to błąd.

Warunek zakończenia z niepowodzeniem.

for (int kolumna = 0; kolumna < N; kolumna++)

W każdej kolumnie aktualnego wiersza sprawdzamy, czy da się postawić wieżę.

if (isSafeWieża(...))

Jeśli dana pozycja jest bezpieczna (brak wieży w tym wierszu i kolumnie)...

board[wiersz][kolumna] = 1;

Stawiamy wieżę na tej pozycji.

if (solveWieża(wiersz + 1, ..., liczbaWież - 1))

Przechodzimy do następnego wiersza i zmniejszamy liczbę wież.
Jeśli ta ścieżka prowadzi do rozwiązania — zwracamy true.

board[wiersz][kolumna] = 0;

Jeśli próba nie powiodła się — usuwamy wieżę (backtracking).

return solveWieża(wiersz + 1, ..., liczbaWież);

Spróbuj pominąć ten wiersz i ustawić wieżę gdzie indziej.
To jest potrzebne wtedy, gdy wież do ustawienia jest mniej niż N.

Dalej należało dodać możliwość, aby użytkownik mógł samodzielnie wpisać liczbę wież (liczbaWież), które mają zostać rozmieszczone na planszy.

Dodano w funkcji main() dodatkowy krok po wyborze trybu wieży:

```
cout << "Podaj liczbę wież: ";  
cin >> liczbaWież;  
rozw = solveWieża(board, 0, N, liczbaWież);
```

Wartość liczbaWież jest następnie przekazywana jako argument do funkcji solveWieża(...).

Dalej należało porównać czasy wykonania programu w dwóch trybach:

1. Dla problemu N hetmanów (pełny backtracking, przekątne + rzędy i kolumny),
2. Dla problemu M wież (tylko rzędy i kolumny).

Porównanie miało na celu zbadanie, który algorytm działa szybciej oraz jak wzrost N wpływa na czas działania w obu przypadkach.

wieże

Dla n = 3:

```
int main() {  
    int N = 3;  
    int numer, rozw;  
    int liczbaWież = N;
```

```
Microsoft Visual Studio Debug X
1 - Queens, 2 - Wieza
Podaj numer: 2
Podaj liczbe wiez: 3
Q . .
. Q .
. . Q
Czas wykonania: 938
```

Dla N = 8:

```
int main() {
    int N = 8;
    int numer, rozw;
    int liczbaWiez = N;
```

```
1 - Queens, 2 - Wieza
Podaj numer: 2
Podaj liczbe wiez: 8
Q . . . . .
. Q . . . . .
. . Q . . . .
. . . Q . . .
. . . . Q . .
. . . . . Q .
. . . . . . Q
Czas wykonania: 6510
```

```
1 - Queens, 2 - Wieza
Podaj numer: 2
Podaj liczbe wiez: 3
Q . . . . .
. Q . . . . .
. . Q . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
Czas wykonania: 2990
```

```
1 - Queens, 2 - Wieza
Podaj numer: 2
Podaj liczbe wiez: 9
niema rozwiazan
```

Dla N = 12:

```
int main() {  
    int N = 12;  
    int numer, rozw;  
    int liczbakwiaz = N;
```

```
Microsoft Visual Studio Debug × +  
1 - Queens, 2 - Wieza  
Podaj numer: 2  
Podaj liczbe wierz: 12  
Q . . . . .  
 . Q . . . . .  
 . . Q . . . . .  
 . . . Q . . . . .  
 . . . . Q . . . . .  
 . . . . . Q . . . . .  
 . . . . . . Q . . . . .  
 . . . . . . . Q . . . . .  
 . . . . . . . . Q . . . . .  
 . . . . . . . . . Q . . . . .  
 . . . . . . . . . . Q . . . . .  
 . . . . . . . . . . . Q . . . . .  
Czas wykonania: 832
```

Dla N = 15:

```
int main() {  
    int N = 15;  
    int numer, rozw;
```

```
1 - Queens, 2 - Wieza  
Podaj numer: 2  
Podaj liczbe wierz: 15  
Q . . . . .  
 . Q . . . . .  
 . . Q . . . . .  
 . . . Q . . . . .  
 . . . . Q . . . . .  
 . . . . . Q . . . . .  
 . . . . . . Q . . . . .  
 . . . . . . . Q . . . . .  
 . . . . . . . . Q . . . . .  
 . . . . . . . . . Q . . . . .  
 . . . . . . . . . . Q . . . . .  
 . . . . . . . . . . . Q . . . . .  
 . . . . . . . . . . . . Q . . . . .  
 . . . . . . . . . . . . . Q . . . . .  
 . . . . . . . . . . . . . . Q . . . . .  
Czas wykonania: 1612
```

Dla N = 23:

```
int main() {
    int N = 23;
    int numer, rozw;
```

[illegible]

nr	Rozmiar tablicy	Ilosc figur	Czas [ms]
1	3	3	938
2	8	8	6510
3	8	3	2990
4	8	9	-
5	12	12	832
6	15	15	1612
7	23	23	2405

Hetmany

nr	Rozmiar tablicy	ilosc figur	Czas wykonania [ms]
1	3	-	-
2	8	8	0
3	12	12	0
4	15	15	3
5	23	23	138

wieże

nr	Rozmiar tablicy	Ilość figur	Czas [ms]
1	3	3	938
2	8	8	6510
3	8	3	2990
4	8	9	-
5	12	12	832
6	15	15	1612
7	23	23	2405

Hetmany:

Algorytm zatrzymuje się po pierwszym rozwiązaniu → bardzo krótki czas działania (do $N = 15$),

Nawet przy $N = 23$, czas jest akceptowalny (138 ms).

Wieże:

Zadziwiająco długi czas działania nawet dla $N = 3$ czy $N = 8$

$N = 8$, liczba wież = 9 – brak rozwiązania:

całkowicie zgodne z teorią: maksymalna liczba wież na planszy $N \times N$ to N (bo musi być 1 wieża na 1 wiersz i 1 kolumnę).

Program poprawnie to wykrył i zakończył bez wyniku.

$N = 3$, hetmany – brak rozwiązania:

zgodne z teorią – dla $N = 2$ i $N = 3$ nie istnieje żadne ustawienie hetmanów bez konfliktu.

Wnioski

Algorytm hetmanów działa szybko, ale nie zlicza wszystkich możliwych układów – tylko pierwszy.

Algorytm wież działa znacznie wolniej, ponieważ prawdopodobnie zlicza wszystkie możliwe konfiguracje

Hetmany wymagają sprawdzania przekątnych, co zwiększa złożoność logiczną algorytmu, ale jednocześnie program kończy działanie po pierwszym poprawnym rozmieszczeniu – dzięki czemu czas wykonania jest krótki.

Wieże są prostsze do rozmieszczenia (brak przekątnych), jednak w obecnej wersji programu zliczają wszystkie możliwe poprawne ustawienia, co wydłuża czas działania nawet dla niewielkich wartości N.

Wady:

- Program nie znajduje rozwiązania dla tych wartości, co może wyglądać jak błąd.
- Dla niektórych rozmiarów planszy, program w trybie wież działa długo (nawet kilkadziesiąt sekund).
- Użytkownik może wpisać np. 9 wież na planszy 8×8, co kończy się błędem lub brakiem wyniku.
- Przy N = 23 czas znacznie się wydłużył 138 ms, co mogło wyglądać jak zawieszenie.
- W printBoard() zawsze wyświetla się Q zamiast W, nawet jeśli tryb to wieże.