Helion

<packt>

WYDANIE II

Struktury danych i algorytmy w języku C#

Wykorzystaj potencjał C# do projektowania efektywnych aplikacji

MARCIN JAMRO

Tytuł oryginału: C# Data Structures and Algorithms: Harness the power of C# to build a diverse range of efficient applications, 2nd Edition

Tłumaczenie: Krzysztof Bąbol (wstęp, rozdz. 1 – 5, 9), Łukasz Piwko (rozdz. 6 – 8, 10) – z wykorzystaniem fragmentów poprzedniego wydania w przekładzie K. Bąbola

ISBN: 978-83-289-1889-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/stdan2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/stdan2.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 9

Zobacz w działaniu

Wiesz już, że algorytmy są obecne prawie wszędzie i że istnieje wiele ich typów i klasyfikacji. Wsparciem dla nich są liczne struktury danych, z których część poznałeś podczas lektury poprzednich rozdziałów. Po kilku fragmentach teoretycznych najwyższy czas zabrać się za praktykę opartą na ciekawych przykładach. Reprezentują one różne typy algorytmów i stanowią podsumowanie wiele zagadnień, które dotąd omówiliśmy.

Najpierw zobaczysz, jak obliczyć określoną liczbę z **ciągu Fibonacciego** kilkoma sposobami, znacznie zróżnicowanymi pod względem wydajności, dzięki czemu dowiesz się, jak optymalizować kod. Czasem nawet drobne zmiany prowadzą do ogromnej poprawy wydajności. Potem dowiesz się, jak metodą zachłanną rozwiązać problem **wydawania reszty**, a także jak przy użyciu algorytmu typu „dziel i zwyciężaj" znaleźć **najbliższą parę punktów** na powierzchni dwuwymiarowej. Zobaczysz też piękny **fraktal** i kod pozwalający zaprojektować taką grafikę. Kolejne przykłady będą dotyczyć zastosowania przeszukiwania z nawrotami i rekurencji w rozwiązywaniu łamigłówek, mianowicie **znajdowania wyjścia z labiryntu** i **sudoku**. Zbliżając się do końca rozdziału, zobaczysz, jak przy użyciu algorytmu genetycznego, opartego na regułach sformułowanej przez Darwina teorii ewolucji i doboru naturalnego, **odgadnąć tytuł** tej książki. Ostatnim przykładem będzie algorytm siłowy **znajdowania tajnego hasła**.

Jak widzisz, przed Tobą wiele ciekawych przykładów, więc przygotuj się na pisanie dość dużej ilości kodu, a wspólnie rozwiążemy te zadania. Zaczynajmy!

W tym rozdziale omówię następujące zagadnienia:

- ciąg Fibonacciego,
- wydawanie reszty,
- najbliższa para punktów,
- generowanie fraktali,
- znajdowanie wyjścia z labiryntu,
- łamigłówka sudoku,
- odgadywanie tytułu,
- znajdowanie hasła.

Ciąg Fibonacciego

W pierwszym przykładzie przyjrzymy się obliczaniu określonej liczby z **ciągu Fibonacciego** za pomocą funkcji **rekurencyjnej** z rysunku 9.1.

$$F_n = \begin{cases} 0, jeżeli\ n = 0 \\ 1, jeżeli\ n = 1 \\ F_{n-1} + F_{n-2}, jeżeli\ n > 1 \end{cases}$$

Rysunek 9.1. Formuła pozwalająca obliczyć liczbę z ciągu Fibonacciego

Jej interpretacja jest bardzo prosta:

- *F(0)* wynosi 0,
- *F(1)* wynosi 1,
- *F(n)* jest sumą *F(n-1)* i *F(n-2)*, co oznacza, że liczba ta jest sumą dwóch poprzednich.

Na przykład wartość *F(2)* jest równa sumie *F(0)* i *F(1)*. Wynosi zatem 1, podczas gdy *F(3)* ma wartość 2. Warto wspomnieć, że istnieją dwa przypadki bazowe, dla *n* równego 0 i 1. Dla obu z nich zdefiniowane są konkretne wartości, to znaczy 0 i 1.

Implementacja **rekurencyjna** w języku C# przedstawia się następująco:

```
long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Jak widać, metoda `Fibonacci` wywołuje dwukrotnie samą siebie z różnymi wartościami parametrów, to znaczy mniejszymi o 1 i 2 od przekazanego do niej parametru `n`. Jeśli wywołasz tę metodę, przekazując do niej liczbę `25`, otrzymasz wynik 75 025:

```
long result = Fibonacci(25);
```

Pamiętaj, że przedstawiona tutaj rekurencyjna wersja obliczania wartości funkcji Fibonacciego jest bardzo nieefektywna i w przypadku większych wartości wejściowych będzie bardzo powolna.

Jej wydajność można znacząco poprawić z zastosowaniem **programowania dynamicznego**, metodami zstępującą albo wstępującą. Najpierw wykorzystajmy **metodę zstępującą** ze **spamiętywaniem** w celu *buforowania obliczonych wyników* podproblemów:

```
Dictionary<int, long> cache = [];

long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    if (cache.ContainsKey(n)) { return cache[n]; }
    long result = Fibonacci(n - 1) + Fibonacci(n - 2);
```

```
        cache[n] = result;
        return result;
    }
```

Klasa `Dictionary` służy za pamięć podręczną. Kluczami słownika są wartości zmiennej `n` przekazywane do metody `Fibonacci`, a wartościami są obliczone wyniki wywołania `Fibonacci(n)`. Wewnątrz metody następuje sprawdzenie, czy pamięć podręczna zawiera klucz równy `n`. Jeśli tak, nie przeprowadzamy dalszych operacji i po prostu zwracamy wartość z pamięci podręcznej. Jeśli nie zawiera ona jeszcze takiego klucza, stosujemy to samo podejście, co w wersji rekurencyjnej, a wynik obliczeń dodajemy do tej pamięci tuż przed jego zwróceniem.

Czy warto wprowadzać takie zmiany? Zobaczmy, jak wygląda czas wykonania obliczeń 50. wyrazu ciągu Fibonacciego. W podstawowej wersji rekurencyjnej na moim komputerze trwało to ponad 88 sekund. Po wprowadzeniu metody zstępującej ten sam wynik otrzymałem po… mniej niż 1 milisekundzie. To rozwiązanie jest prawie 100 tysięcy razy szybsze!

Skoro już wiesz, że programowanie dynamiczne ma bardzo duże znaczenie dla wydajności, przyjrzyjmy się obliczaniu liczby Fibonacciego **metodą wstępującą**:

```
long Fibonacci(int n)
{
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }

    long a = 0;
    long b = 1;
    for (int i = 2; i <= n; i++)
    {
        long result = a + b;
        a = b;
        b = result;
    }

    return b;
}
```

Dokonaliśmy tu większej modyfikacji, ponieważ zastąpiliśmy rekurencję iteracją. Kod jest jednak bardzo prosty, gdyż składa się tylko z jednej pętli `for` iterującej od 2 do podanej liczby i obliczającej sumę dwóch wcześniejszych. Istnieją oczywiście oddzielne warunki `if` dla parametru `n` o wartości 0 i 1.

A co w tym przypadku z wydajnością? Porównajmy obliczanie 5000. liczby z ciągu Fibonacciego metodami zstępującą i wstępującą. Przy użyciu tej pierwszej na moim laptopie trwa ono około 2 milisekund, a drugiej — niespełna milisekundę. Pamiętaj, że mówimy teraz o 5000. liczbie z ciągu Fibonacciego, a wcześniej testowaliśmy jedynie 50. Wzrost wydajności jest imponujący, nieprawdaż?

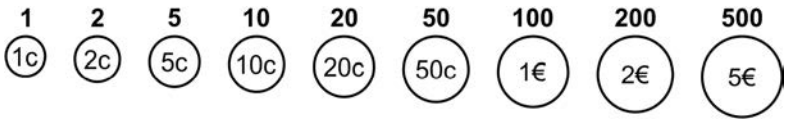
Po ukończeniu pierwszego przykładu przejdźmy do rozwiązywania problemu wydawania reszty.

Wyniki mogą się różnić

Te wyniki wydajnościowe uzyskałem na swoim komputerze w bardzo prosty sposób, nawet bez kilkukrotnych powtórzeń. Oczywiście w innych okolicznościach, na przykład gdy komputer jest intensywnie wykorzystywany, wyniki mogą być odmienne. Kluczowe było jednak przedstawienie pewnej tendencji, a nie dokładnego wyniku w milisekundach. Dzięki testom wydajnościowym chciałem pokazać, jak wielka jest różnica pomiędzy podstawową wersją rekurencyjną a każdą z wersji zoptymalizowanych za pomocą programowania dynamicznego.

Wydawanie reszty

Drugi przykład ukazany w tym rozdziale przedstawia **algorytm zachłanny** rozwiązywania problemu **wydawania reszty**, który polega na znalezieniu najmniejszej liczby monet pozwalającej uzyskać wartość podaną na wejściu (rysunek 9.2).



Rysunek 9.2. Ilustracja nominałów w przypadku waluty euro

Dla przykładu, aby w systemie nominałów 1, 2, 5, 10, 20, 50, 100, 200 i 500 uzyskać wartość 158, należy wziąć 5 monet, a mianowicie 100, 50, 5, 2 i 1. Podejście zachłanne jest bardzo proste, gdyż wystarczy *wybrać największy możliwy nominał nie większy niż pozostała suma*. Operację tę wykonujemy dopóty, dopóki pozostała wartość nie zmaleje do zera. Jak widać, w tym algorytmie nie zastanawiamy się nad ogólnym rozwiązaniem, ale na każdym kroku próbujemy wybierać takie, które jest obecnie najlepsze.

Oto implementacja w języku C#:

```
int[] den = [1, 2, 5, 10, 20, 50, 100, 200, 500];
List<int> coins = GetCoins(158);
coins.ForEach(Console.WriteLine);

List<int> GetCoins(int amount)
{
    List<int> coins = [];
    for (int i = den.Length - 1; i >= 0; i--)
    {
        while (amount >= den[i])
        {
            amount -= den[i];
            coins.Add(den[i]);
        }
    }
    return coins;
}
```

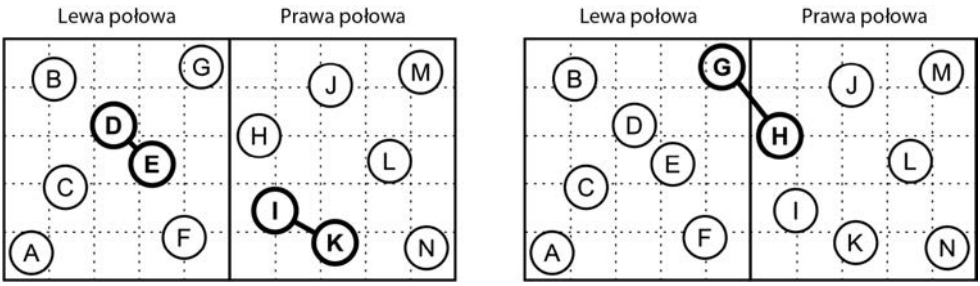
Najważniejszą rolę odgrywa metoda `GetCoins`, przyjmująca jedną wartość wejściową, to znaczy sumę do wydania. Zwraca ona listę wybranych monet. Po wywołaniu metody i przekazaniu do niej na przykład liczby 158 w konsoli zobaczymy nominały 100, 50, 5, 2 i 1.

Szybko nam poszło z tym przykładem! Przejdźmy do czegoś nieco bardziej skomplikowanego.

Najbliższa para punktów

Kolejnym przykładem będzie algorytm **znajdowania najbliższej pary punktów** znajdujących się na powierzchni dwuwymiarowej. Jest to interesujący problem algorytmiczny, który można rozwiązać w paradygmacie „**dziel i zwyciężaj**".

Każdy punkt na powierzchni jest reprezentowany przez współrzędne *x* i *y*, których wartości zaczynają się od (*0*, *0*), co odpowiada lewemu górnemu rogowi. Aby znaleźć najbliższą sobie parę punktów, najpierw sortujemy je wszystkie według współrzędnej *x*, w sposób pokazany na diagramach z rysunku 9.3, i oznaczamy literami od *A* do *N*.



Rysunek 9.3. Diagramy obrazujące algorytm znajdowania najbliższej pary punktów

Potem dzielimy powierzchnię na dwie połowy. W tym celu możemy podzielić na pół liczbę wszystkich punktów, co w tym przypadku da nam liczbę 7, i potraktować pierwsze 7 punktów jako lewą, a następne 7 jako prawą połowę tej powierzchni.

W tym zadaniu jest miejsce na **rekurencję**, musimy więc rekurencyjnie znaleźć najbliższe siebie punkty w obu połowach i zapisać ich dane jako r_l (punkty *D* i *E*) oraz r_r (punkty *I* i *K*). Aby znaleźć najbliższą parę, należy porównać ze sobą te odległości, a wynik zapisać jako *r*. W naszym przypadku będą to punkty *D* i *E*.

To jeszcze nie wszystko — trzeba też zbadać odległości pomiędzy punktami z lewej i prawej połowy, pokazane na rysunku 9.3 po prawej stronie. W tym celu musimy pobrać do tablicy dane wszystkich punktów, które, biorąc pod uwagę wyłącznie współrzędną *x*, znajdują się bliżej środka, niż wynosi odległość *r* już znalezionej pary punktów (w naszym przypadku *D* i *E*). W tablicy tej znajdujemy parę punktów położonych najbliżej siebie (tutaj: *G* i *H*). Oznaczmy ten wynik literą *s*. Aby dokończyć zadanie, trzeba sprawdzić, w której parze, *r* (*D* i *E*) czy *s* (*G* i *H*), punkty leżą bliżej siebie. Potem wystarczy zwrócić wynik (w naszym przypadku *D* i *E*).

Najważniejszy fragment kodu wygląda następująco:

```
Result? FindClosestPair(Point[] points)
{
    if (points.Length <= 1) { return null; }
    if (points.Length <= 3) { return Closest(points); }

    int m = points.Length / 2;
    Result r = Closer(
        FindClosestPair(points.Take(m).ToArray())!,
        FindClosestPair(points.Skip(m).ToArray())!);

    Point[] strip = points.Where(p => Math.Abs(p.X
        - points[m].X) < r.Distance).ToArray();
    return Closer(r, Closest(strip));
}
```

Na początku znajduje się warunek bazowy. Przerywamy dalsze wykonywanie, jeśli tablica punktów jest pusta albo zawiera tylko jeden element. Następnie sprawdzamy, czy liczba punktów w tablicy jest równa 3 lub mniejsza. Jeśli tak, po prostu sprawdzamy w kolekcji wszystkie możliwe warianty i wybieramy z niej najbliższą sobie parę punktów. W przeciwnym razie znajdujemy indeks środka i wywołujemy metodę rekurencyjnie dla lewej i prawej połowy. Potem z obu połówek wybieramy punkty położone dostatecznie blisko środka, biorąc pod uwagę wyłącznie ich współrzędne *x*. Następnie obliczamy odległości pomiędzy wszystkimi punktami w tablicy `strip`, aby wybrać z niej najbliższą sobie parę. Na koniec zwracamy z tych dwóch par punktów tę, w której leżą one bliżej siebie.

Jak widać, zasadnicza część algorytmu jest całkiem prosta w implementacji i dosyć zrozumiała. Omówmy więc resztę, począwszy od definicji rekordu `Point`:

```
public record Point(int X, int Y)
{
    public float GetDistanceTo(Point p) =>
        (float)Math.Sqrt(Math.Pow(X - p.X, 2)
            + Math.Pow(Y - p.Y, 2));
};
```

Rekord `Result` wygląda następująco:

```
public record Result(Point P1, Point P2, double Distance);
```

W zasadniczej części kodu stosuje się dwie metody pomocnicze, a mianowicie `Closest` i `Closer`. Pierwsza z nich wyszukuje w tablicy najbliższą parę punktów. Oto kod tej metody:

```
Result Closest(Point[] points)
{
    Result result = new(points[0], points[0], double.MaxValue);
    for (int i = 0; i < points.Length; i++)
    {
        for (int j = i + 1; j < points.Length; j++)
        {
            double distance = points[i].GetDistanceTo(points[j]);
            if (distance < result.Distance)
```

```
                {
                    result = new(points[i], points[j], distance);
                }
            }
        }
        return result;
    }
```

Kod metody `Closer` przedstawia się zaś następująco:

```
Result Closer(Result r1, Result r2) =>
    r1.Distance < r2.Distance ? r1 : r2;
```

Na koniec przyjrzyjmy się wywoływaniu wspomnianych metod:

```
List<Point> points =
[
    new Point(6, 45), //A
    new Point(12, 8), //B
    new Point(14, 31), //C
    new Point(24, 18), //D
    new Point(32, 26), //E
    new Point(40, 41), //F
    new Point(44, 6), //G
    new Point(57, 20), //H
    new Point(60, 35), //I
    new Point(72, 9), //J
    new Point(73, 41), //K
    new Point(85, 25), //L
    new Point(92, 8), //M
    new Point(93, 43) //N
];

points.Sort((a, b) => a.X.CompareTo(b.X));
Result? closestPair = FindClosestPair(points.ToArray());
if (closestPair != null)
{
    Console.WriteLine(
        "Najbliższą parę: ({0}, {1}) i ({2}, {3})
         dzieli odległość: {4:F2}",
        closestPair.P1.X,
        closestPair.P1.Y,
        closestPair.P2.X,
        closestPair.P2.Y,
        closestPair.Distance);
}
```

Dostarczamy kolekcję punktów, sortujemy je po współrzędnych *x* i wywołujemy metodę `FindClosestPair`, przekazując jako parametr całą tablicę. Na końcu w konsoli pojawi się następujący wynik:

```
Najbliższą parę: (24, 18) i (32, 26) dzieli odległość: 11,31
```

Otrzymaliśmy więc taki sam wynik, jak podczas analizy tego przykładu na początku podrozdziału. Dobra robota — gratulacje!

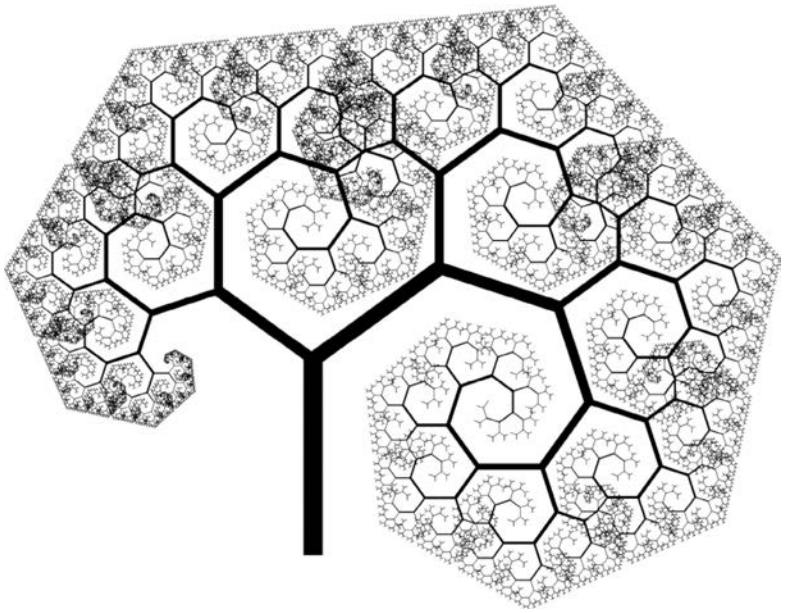
Gdzie można znaleźć więcej informacji?

W tym rozdziale przedstawiłem przykłady reprezentujące różne często spotykane problemy algorytmiczne, stawiane nawet podczas rozmów kwalifikacyjnych na stanowisko programisty. Te zagadnienia często też są poruszane w internecie. Więcej informacji o opisanym podejściu do problemu *najbliższej pary punktów* i implementacji jego rozwiązania można znaleźć na stronie *https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/*. Jako że serwis *GeeksForGeeks* zawiera olbrzymią liczbę różnych artykułów, można tam również znaleźć implementacje rozwiązań innych problemów omówionych w tym rozdziale, między innymi znajdowania wyjścia z labiryntu (na stronie *https://www.geeksforgeeks.org/rat-in-a-maze/*) i łamigłówki sudoku (pod adresem *https://www.geeksforgeeks.org/sudoku-backtracking-7/*).

Moim zdaniem programowanie można traktować jako pewien rodzaj sztuki. Tak jak malarze tworzą piękne obrazy, tak programiści mogą pisać finezyjny kod. Skoro rozmawiamy o sztuce, napiszmy coś pięknego i namalujmy cudne fraktale!

Generowanie fraktali

Rekurencję stosuje się w wielu różnych algorytmach, również tych związanych z grafiką komputerową. Z tego powodu warto przyjrzeć się kolejnemu przykładowi — **generowaniu fraktali** tworzących ciekawe wzory, na przykład takie jak na rysunku 9.4.



Rysunek 9.4. Przykładowy fraktal wygenerowany funkcją rekurencyjną

To naprawdę piękne, nieprawdaż? Czy rozpoznajesz na rysunku wzór drzewa? Jeśli nie, prześledź grubą linię na środku obrazu (*pień* drzewa) i zwróć uwagę, że dzieli się ona na dwie kolejne (*gałęzie*), z których każda jest odchylona o pewien kąt względem pnia. Prześledź dalej jedną z tych linii i zauważ, że rozdziela się ona według tej samej zasady. Proces ten przebiega dalej aż do osiągnięcia określonej liczby poziomów.

Ten algorytm rekurencyjny dosyć łatwo opisać w języku naturalnym, przyjrzyjmy się więc kodowi obliczającemu współrzędne początków i końców kolejnych linii, które razem tworzą ten piękny rysunek. Implementacja metody `AddLine` przedstawia się następująco:

```
void AddLine(int level, float x, float y,
    float length, float angle)
{
    if (level < 0) { return; }

float endX = x + (float)(length * Math.Cos(angle));
float endY = y + (float)(length * Math.Sin(angle));
lines.Add(new(x, y, endX, endY));

AddLine(level - 1, endX, endY, length * 0.8f,
    angle + (float)Math.PI * 0.3f);
AddLine(level - 1, endX, endY, length * 0.6f,
        angle + (float)Math.PI * 1.7f);
}
```

Metoda ta przyjmuje kilka parametrów, a mianowicie:

- poziom wzoru, który na początku jest liczbą nieujemną, a na końcu osiąga wartość 0,
- współrzędne *x* i *y* punktu początkowego,
- długość linii,
- jej kąt, podany w radianach.

Wewnątrz metody sprawdzamy warunek bazowy, a mianowicie to, czy poziom jest mniejszy od 0. Jeżeli nie, obliczamy współrzędne *x* i *y* punktu końcowego i dodajemy linię do kolekcji (`lines`). Na końcu rekurencyjnie wywołujemy metodę `AddLine` z innymi parametrami. Zmniejszamy poziom, obliczone współrzędne punktu końcowego przekazujemy dalej jako współrzędne punktu początkowego następnej linii, zmniejszamy jej długość o 20 lub 40% (w zależności od gałęzi) oraz modyfikujemy kąt.

Warto odnotować, że powyższy kod korzysta z rekordu `Line`, zdefiniowanego następująco:

```
record Line(float X1, float Y1, float X2, float Y2)
{
    public float GetLength() =>
        (float)Math.Sqrt(Math.Pow(X1 - X2, 2)
            + Math.Pow(Y1 - Y2, 2));
}
```

Oto kolejny fragment kodu:

```
using System.Drawing;
using System.Drawing.Drawing2D;
```

```
const int maxSize = 1000;
List<Line> lines = [];
AddLine(14, 0, 0, 500, (float)Math.PI * 1.5f);
```

Określamy tutaj maksymalną szerokość lub wysokość mapy bitowej, na której narysowany zostanie fraktal (`maxSize`). Następnie przygotowujemy pustą listę linii. W ostatnim wierszu wywołujemy metodę `AddLine`. Wskazujemy, że ma być `14` poziomów wzoru.

Wymagany pakiet NuGet

Ponieważ używane są elementy przestrzeni nazw `System.Drawing` i `System.Drawing.Drawing2D`, trzeba zainstalować dodatkowy pakiet NuGet, a mianowicie `System.Drawing.Common`.

Skoro mamy już kolekcję linii, możemy obliczyć minimalne i maksymalne współrzędne *x* i *y*, a także docelową szerokość (*width*) i wysokość (*height*), jak przedstawiono poniżej:

```
float xMin = lines.Min(l => Math.Min(l.X1, l.X2));
float xMax = lines.Max(l => Math.Max(l.X1, l.X2));
float yMin = lines.Min(l => Math.Min(l.Y1, l.Y2));
float yMax = lines.Max(l => Math.Max(l.Y1, l.Y2));
float size = Math.Max(xMax - xMin, yMax - yMin);
float factor = maxSize / size;
int width = (int)((xMax - xMin) * factor);
int height = (int)((yMax - yMin) * factor);
```

Pozostała część kodu jest związana z rysowaniem fraktala na mapie bitowej:

```
using Bitmap bitmap = new(width, height);
using Graphics graphics = Graphics.FromImage(bitmap);
graphics.Clear(Color.White);
graphics.SmoothingMode = SmoothingMode.AntiAlias;
using Pen pen = new(Color.Black, 1);
foreach (Line line in lines)
{
    pen.Width = line.GetLength() / 20;
    float sx = (line.X1 - xMin) * factor;
    float sy = (line.Y1 - yMin) * factor;
    float ex = (line.X2 - xMin) * factor;
    float ey = (line.Y2 - yMin) * factor;
    graphics.DrawLine(pen, sx, sy, ex, ey);
}
bitmap.Save($"{DateTime.Now:HH-mm-ss}.png");
```

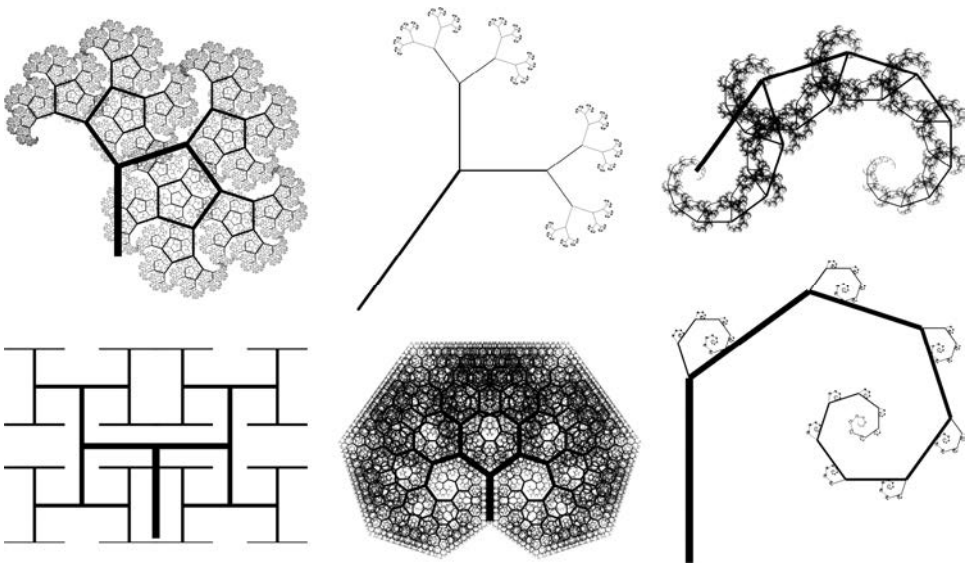
W zaprezentowanym kodzie tworzymy nową instancję klasy `Bitmap` o podanej wielkości, a także przygotowujemy obiekt `Graphics`, służący do rysowania na tej mapie bitowej. Następnie całą tę mapę wypełniamy na biało, ustawiamy wygładzanie krawędzi (ang. *anti-aliasing*) i definiujemy pióro do rysowania czarnym kolorem.

Na powyższym listingu widać pętlę `foreach`. W jej wnętrzu obliczamy grubość linii, a także współrzędne jej początku i końca. Ostatni wiersz pętli po prostu rysuje linię. Na koniec przygotowaną mapę bitową zapisujemy w katalogu roboczym w pliku o nazwie utworzonej na podstawie bieżącego czasu.

Czy widzisz ostrzeżenia?

Przygotowany kod powoduje pojawienie się w środowisku IDE pewnych ostrzeżeń. Informują one, że funkcje związane z grafiką są dostępne tylko na platformie Windows. Ostrzeżenia te można ukryć po dodaniu tuż przed powyższym kodem wiersza `#pragma warning disable CA1416`, a na samym końcu `#pragma warning restore CA1416`. Dodajmy, że jeśli chcesz rysować grafikę również na innych platformach, możesz skorzystać z innych dostępnych pakietów NuGet, na przykład ze `SkiaSharp`. Gorąco zachęcam do opracowania tego przykładu także przy użyciu `SkiaSharp`.

To wszystko! Możesz teraz skorygować poszczególne parametry, by narysować piękne fraktale, jeszcze lepsze niż ten na wcześniejszym rysunku. Kilka innych wyników pokazano na rysunku 9.5.



Rysunek 9.5. Przykładowe fraktale wygenerowane przy użyciu funkcji rekurencyjnej

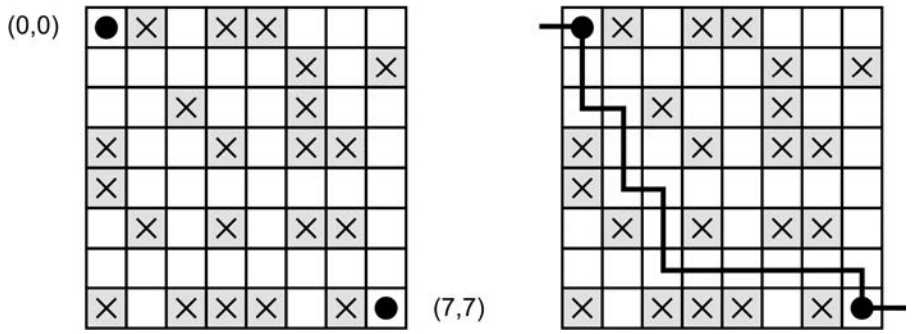
Gdzie można znaleźć więcej informacji?

Wiele materiałów na temat fraktali można znaleźć w internecie, a podejście podobne do przedstawionego tutaj jest opisane na stronie *http://www.csharphelper.com/howtos/howto_curly_tree.html*.

Gdy już wygląd fraktala będzie dla Ciebie satysfakcjonujący, przejdź do następnego podrozdziału, w którym rozwiążemy problem *znajdowania wyjścia z labiryntu*.

Znajdowanie wyjścia z labiryntu

Kontynuując naszą przygodę z przykładami algorytmów, rozwiążemy problem **znajdowania wyjścia z labiryntu** za pomocą **algorytmu z nawrotami**. Diagram jest pokazany na rysunku 9.6.



Rysunek 9.6. Ilustracja przykładu znajdowania wyjścia z labiryntu

Wyobraźmy sobie, że w lewym górnym polu planszy, oznaczonym na rysunku 9.6 współrzędnymi **(0, 0)**, znajduje się szczur, a my musimy mu znaleźć drogę do wyjścia, zlokalizowanego na prawym dolnym polu planszy, o współrzędnych **(7, 7)**. Oczywiście niektóre miejsca są zablokowane (oznaczone na szaro) i szczur nie da rady przez nie przejść. Aby dojść do celu, może poruszać się wyłącznie po dostępnych miejscach w górę, w dół, w lewo i w prawo.

Problem ten da się rozwiązać z zastosowaniem **rekurencji** przez sprawdzanie możliwych dróg wiodących szczura od wejścia do wyjścia. Jeśli właśnie wyznaczana droga nie prowadzi do wyjścia, należy **zawrócić** i spróbować innych wariantów.

Zasadniczą część implementacji stanowi następująca metoda `Go`:

```
bool Go(int row, int col)
{
    if (row == size - 1
        && col == size - 1
        && maze[row, col])
    {
        solution[row, col] = true;
        return true;
    }

if (row >= 0 && row < size
    && col >= 0 && col < size
    && maze[row, col])
{
    if (solution[row, col]) { return false; }
        solution[row, col] = true;
        if (Go(row + 1, col)) { return true; }
        if (Go(row, col + 1)) { return true; }
```

```
            if (Go(row - 1, col)) { return true; }
            if (Go(row, col - 1)) { return true; }
            solution[row, col] = false;
            return false;
        }

        return false;
    }
```

Metoda ta przyjmuje dwa parametry, `row` i `column`. Korzysta też z trzech dodatkowych zmiennych. Pierwsza nazywa się `maze` i jest dwuwymiarową tablicą reprezentującą labirynt z polami, które są dla szczura dostępne (mają wartości `true`) lub niedostępne (`false`). Druga zmienna, o nazwie `size`, przechowuje rozmiar labiryntu, to znaczy liczbę rzędów, która jest równa liczbie kolumn. Kolejna zmienna, `solution`, jest podobna do `maze`, ale zawiera dane obecnie sprawdzanej drogi. Pola tworzące rozwiązanie mają wartości `true`, a pozostałe — `false`.

Na początku metody sprawdzamy, czy szczur dotarł już do wyjścia. Jeśli tak, oznaczamy ostatnie pole jako część rozwiązania i zwracamy wartość wskazującą, że szczur wykonał zadanie i wyszedł z labiryntu. W przeciwnym razie sprawdzamy, czy szczur jest nadal wewnątrz labiryntu i czy nie wszedł na niedostępne pole. Jeżeli oba te warunki są spełnione, sprawdzamy jeszcze, czy bieżące pole jest już częścią ścieżki. Jeśli tak, informujemy, że to rozwiązanie jest niewłaściwe.

Jeżeli szczur jest wewnątrz labiryntu na dostępnym polu, które jeszcze nie było odwiedzone, to pole oznaczamy jako część rozwiązania, a potem próbujemy iść do dołu, w prawo, do góry i w lewo przez rekurencyjne wywoływanie metody `Go`. Jeśli żaden z tych ruchów nie prowadzi do celu (oczywiście przy uwzględnieniu również kolejnych kroków), wskazujemy, że bieżące pole nie jest częścią rozwiązania, i dokonujemy **nawrotu**. Potem zwracamy wartość wskazującą, że cel nie został osiągnięty.

Przyjrzyjmy się teraz kodowi, który po raz pierwszy wywołuje metodę `Go`:

```
int size = 8;
bool t = true;
bool f = false;
bool[,] maze =
{
    { t, f, t, f, f, t, t, t },
    { t, t, t, t, t, f, t, f },
    { t, t, f, t, t, f, t, t },
    { f, t, t, f, t, f, f, t },
    { f, t, t, t, t, t, t, t },
    { t, f, t, f, t, f, f, t },
    { t, t, t, t, t, t, t, t },
    { f, t, f, f, f, t, f, t }
};
bool[,] solution = new bool[size, size];
if (Go(0, 0)) { Print(); }
```

Reszta kodu jest związana z metodą `Print`:

```
void Print()
{
    for (int row = 0; row < size; row++)
    {
        for (int col = 0; col < size; col++)
        {
            Console.Write(solution[row, col] ? "x" : "-");
        }
        Console.WriteLine();
    }
}
```

A tak wygląda wynik:

```
x-------
x-------
xx------
-x------
-xx-----
--x-----
--xxxxxx
-------x
```

Kończąc ten przykład, warto zwrócić uwagę na prostotę i zwięzłość powyższego kodu. Pozwoliło to nam przejrzyście zdefiniować rozwiązanie problemu. Warto jednak pamiętać o tym, że jeśli dróg wyjścia z labiryntu będzie więcej, algorytm pokaże tylko jedną.

Skoro pomogliśmy szczurowi znaleźć drogę w labiryncie, przejdźmy do następnego przykładu, w którym dowiesz się, jak rozwiązać automatycznie łamigłówkę sudoku.

Łamigłówka sudoku

Czy zdarzyło Ci się kiedykolwiek rozwiązywać **sudoku**? To bardzo popularna gra, polegająca na *wypełnianiu pustych komórek na planszy o rozmiarach 9×9 liczbami od 1 do 9. Nie mogą się one jednak powtórzyć w żadnym rzędzie, kolumnie ani polu 3×3*. Przykładowe plansze na początku i po rozwiązaniu przedstawia rysunek 9.7.

	5		4		1			6
1			9	5		8		
9		4		6				1
6	2				5	3		4
	9			7		2		5
5		7					8	9
8			5	1	9			2
2	3				6	5		8
4	1		2		8	6		

7	5	**3**	4	**8**	1	**9**	**2**	6
1	**6**	**2**	9	5	**7**	8	**4**	**3**
9	**8**	4	**3**	6	**2**	**7**	**5**	1
6	2	**1**	**8**	**9**	5	3	**7**	4
3	9	**8**	**1**	7	**4**	2	**6**	5
5	**4**	7	**6**	**2**	**3**	**1**	8	9
8	**7**	**6**	5	1	9	**4**	**3**	2
2	3	**9**	**7**	**4**	6	5	**1**	8
4	1	**5**	2	**3**	8	6	**9**	**7**

Rysunek 9.7. Przykłady nierozwiązanej i rozwiązanej łamigłówki sudoku

Nauczysz się teraz rozwiązywać sudoku nie ołówkiem na kawałku papieru, ale za pomocą algorytmu! Sztuki tej można dokonać w podejściu **z nawrotami**, poprzez próby przydzielania do pustych komórek liczb, o ile oczywiście nie powtarzają się one w żadnym rzędzie, kolumnie ani polu. Jeżeli wprowadzenie liczby nie spowoduje rozwiązania całej łamigłówki, przydzielamy kolejną i znowu dokonujemy sprawdzenia. Przyjrzyjmy się najistotniejszej części kodu:

```
bool Solve()
{
    (int row, int col) = GetEmpty();
    if (row < 0 && col < 0) { return true; }

    for (int i = 1; i <= 9; i++)
    {
        if (IsCorrect(row, col, i))
        {
            board[row, col] = i;
            if (Solve()) { return true; }
            else { board[row, col] = 0; }
        }
    }

    return false;
}
```

Na początku metody `Solve` pobieramy współrzędne pierwszej pustej komórki. Jeżeli takich nie ma, metoda `GetEmpty` zwraca wartości (*–1*, *–1*). Wtedy zwracamy wartość `true`, wskazującą, że gra została rozwiązana.

W przeciwnym razie przy użyciu pętli `for` iterujemy po wszystkich możliwych liczbach (to znaczy od 1 do 9). W każdej iteracji metodą `IsCorrect` sprawdzamy, czy daną liczbę można poprawnie wprowadzić do komórki, co daje nam pewność, że w żadnym rzędzie, kolumnie ani polu nie będzie powtórzeń. Jeśli jest to możliwe, wprowadzamy tę liczbę do komórki i *rekurencyjnie* wywołujemy metodę `Solve`. Jeśli zwróci ona `false`, wskazując, że ten wariant jest niedopuszczalny, dokonujemy nawrotu i czyścimy komórkę z wprowadzonej wartości, co powoduje, że jest ona znowu pusta i możemy spróbować innej opcji. Jeśli żadna z nich nie doprowadzi do rozwiązania, zwracamy `false`.

W przedstawionym kodzie stosowane są dwie metody pomocnicze. Jedna z nich, `GetEmpty`, wyszukuje pierwszą komórkę, która nie jest jeszcze wypełniona. Oto kod tej metody:

```
 (int, int) GetEmpty()
{
    for (int r = 0; r < 9; r++)
    {
        for (int c = 0; c < 9; c++)
        {
            if (board[r, c] == 0) { return (r, c); }
        }
    }
    return (-1, -1);
}
```

Druga metoda pomocnicza, o nazwie `IsCorrect`, gwarantuje, że po wprowadzeniu danej liczby do określonej komórki (o wskazanych rzędzie i kolumnie) plansza nadal będzie spełniać warunki gry sudoku. Oto kod tej metody:

```
bool IsCorrect(int row, int col, int num)
{
    for (int i = 0; i < 9; i++)
    {
        if (board[row, i] == num) { return false; }
        if (board[i, col] == num) { return false; }
    }

    int rs = row - row % 3;
    int cs = col - col % 3;
    for (int r = rs; r < rs + 3; r++)
    {
        for (int c = cs; c < cs + 3; c++)
        {
            if (board[r, c] == num) { return false; }
        }
}
    return true;
}
```

Na początku sprawdzamy, czy wartości w danym rzędzie i kolumnie są unikatowe. W pozostałej części badamy, czy liczby nie powtarzają się w danym polu 3×3.

Przykładowy kod uruchamiający algorytm rozwiązywania sudoku wygląda następująco:

```
int[,] board = new int[,]
{
    { 0, 5, 0, 4, 0, 1, 0, 0, 6 },
    { 1, 0, 0, 9, 5, 0, 8, 0, 0 },
    { 9, 0, 4, 0, 6, 0, 0, 0, 1 },
    { 6, 2, 0, 0, 0, 5, 3, 0, 4 },
    { 0, 9, 0, 0, 7, 0, 2, 0, 5 },
    { 5, 0, 7, 0, 0, 0, 0, 8, 9 },
    { 8, 0, 0, 5, 1, 9, 0, 0, 2 },
    { 2, 3, 0, 0, 0, 6, 5, 0, 8 },
    { 4, 1, 0, 2, 0, 8, 6, 0, 0 }
};
if (Solve()) { Print(); }
```

Na koniec przyjrzyjmy się metodzie `Print`:

```
void Print()
{
    for (int r = 0; r < 9; r++)
    {
        for (int c = 0; c < 9; c++)
        {
            Console.Write($"{board[r, c]} ");
        }
```

```
            Console.WriteLine();
        }
    }
```

Oto wynik:

```
7 5 3 4 8 1 9 2 6
1 6 2 9 5 7 8 4 3
9 8 4 3 6 2 7 5 1
6 2 1 8 9 5 3 7 4
3 9 8 1 7 4 2 6 5
5 4 7 6 2 3 1 8 9
8 7 6 5 1 9 4 3 2
2 3 9 7 4 6 5 1 8
4 1 5 2 3 8 6 9 7
```

Jak widać, algorytmy z nawrotami pozwalają z powodzeniem rozwiązać zarówno problem znajdowania wyjścia z labiryntu, jak i łamigłówkę sudoku. Cel ten można osiągnąć za pomocą krótkiego, przejrzystego, łatwo zrozumiałego kodu. Po tych przykładach przejdźmy więc do następnego podrozdziału, w którym zobaczysz ciekawe zastosowanie algorytmu genetycznego.

Odgadywanie tytułu

Najwyższy czas na zastosowanie algorytmu innego rodzaju, **heurystycznego**. Typ ten również ma liczne zastosowania i dzieli się na wiele podtypów. Tutaj skupimy się wyłącznie na **algorytmach genetycznych**, które są **adaptacyjnymi heurystycznymi algorytmami przeszukiwania**. Mają one związek ze sformułowaną przez Darwina teorią ewolucji i doboru naturalnego. Według niej osobniki tworzące populację konkurują ze sobą, a *populacja ewoluuje tak, by następne pokolenia były lepiej przystosowane do przeżycia*. Algorytmy genetyczne operują na ciągach kodowych, które ewoluują tak, by uzyskać najwyższą wartość **przystosowania** (ang. *fitness*), zgodnie z **regułą przetrwania** (ang. *rule of survival*). *Geny najlepiej przystosowanych rodziców są przekazywane dalej*. Dochodzi również do **losowej wymiany informacji** (ang. *randomized data exchange*). Algorytm kończy działanie po osiągnięciu odpowiedniej wartości przystosowania albo maksymalnej liczby pokoleń.

> **Gdzie można znaleźć więcej informacji?**
>
> Dużo na temat algorytmów genetycznych można znaleźć w internecie, na przykład w artykule opublikowanym pod adresem *https://link.springer.com/article/10.1007/s11042-020-10139-6*. Proste ujęcie algorytmu genetycznego przedstawione w tym rozdziale jest oparte na rozwiązaniu omówionym na stronie *https://www.geeksforgeeks.org/genetic-algorithms/*.

Przyjrzyjmy się przykładowi, w którym wykorzystamy algorytm genetyczny do odgadnięcia tytułu tej książki. Pierwszy fragment kodu wygląda następująco:

```
const string Genes = "aąbcćdeęfghijklłmnńoóprsśtuwyzźż
    #AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ ";
const string Target = "Struktury danych i algorytmy w języku C#";
Random random = new();
int generationNo = 0;
List<Individual> population = [];
for (int i = 0; i < 1000; i++)
{
    string chromosome = GetRandomChromosome();
    population.Add(new(chromosome,
        GetFitness(chromosome)));
}
```

Najpierw tworzymy populację początkową z 1000 osobników. Każdy z nich ma przypadkowy chromosom, reprezentowany przez losowy ciąg znaków o długości równej łańcuchowi docelowemu, którym jest tytuł książki. Przejdźmy dalej:

```
List<Individual> generation = [];
while (true)
{
    population.Sort((a, b) =>
        b.Fitness.CompareTo(a.Fitness));
    if (population[0].Fitness == Target.Length)
    {
        Print();
        break;
    }

    generation.Clear();
    for (int i = 0; i < 200; i++)
    {
        generation.Add(population[i]);
    }

    for (int i = 0; i < 800; i++)
    {
        Individual p1 = population[random.Next(400)];
        Individual p2 = population[random.Next(400)];
        Individual offspring = Mate(p1, p2);
        generation.Add(offspring);
    }

    population.Clear();
    population.AddRange(generation);
    Print();
    generationNo++;
}
```

Najciekawszy fragment znajduje się w nieskończonej pętli `while`. Sortujemy w nim populację, począwszy od osobników najlepiej przystosowanych do przeżycia — to znaczy według ich dopasowania w kolejności malejącej. Mówiąc dokładniej, dopasowanie wynosi 0, jeśli żaden znak w łańcuchu chromosomu nie odpowiada znakowi w ciągu docelowym.

Jeśli zaś ciąg chromosomu będzie identyczny z docelowym, dopasowanie będzie miało wartość 40 (to znaczy tyle, ile znaków ma tytuł książki). Jeśli zatem dopasowanie pierwszego elementu populacji (najlepiej przystosowanego) jest równe długości ciągu docelowego, oznacza to, że znaleźliśmy rozwiązanie, więc wystarczy wypisać je i opuścić pętlę.

W przeciwnym razie czyścimy listę danych nowego pokolenia i dodajemy do niej 200 najlepiej przystosowanych osobników. Tym samym *20% najlepiej przystosowanych osobników przechodzi automatycznie do następnego pokolenia*. Aby zapełnić w nowym pokoleniu pozostałe 800 miejsc, przeprowadzamy **krzyżowanie** (ang. *crossover*) i *spośród 40% najlepiej przystosowanych osobników losujemy rodziców, którzy wygenerują nowe osobniki*. Następnie zastępujemy bieżącą populację nowym pokoleniem i przechodzimy do kolejnej iteracji.

Warto wspomnieć o rekordzie `Individual`, którego kod wygląda następująco:

```
record Individual(string Chromosome, int Fitness);
```

Ma on dwie właściwości, o nazwach `Chromosome` i `Fitness`. Pierwsza z nich przechowuje ciąg znaków korygowany w trakcie ewolucji, a druga jest liczbą wskazującą, w jakim stopniu konkretny osobnik jest przystosowany do przeżycia. Oczywiście, im wyższa wartość, tym lepiej.

Do generowania nowych osobników z dwojga rodziców służy metoda `Mate`:

```
Individual Mate(Individual p1, Individual p2)
{
    string child = string.Empty;
    for (int i = 0; i < Target.Length; i++)
    {
        float r = random.Next(101) / 100.0f;
        if (r < 0.45f) { child += p1.Chromosome[i]; }
        else if (r < 0.9f) { child += p2.Chromosome[i]; }
        else { child += GetRandomGene(); }
    }
    return new Individual(child, GetFitness(child));
}
```

Najciekawszą częścią tej metody jest pętla `for`, w której tworzony jest chromosom dziecka. Odbywa się to według następujących reguł:

- *około 45% genów bierze się z pierwszego rodzica*,
- *około 45% genów bierze się z drugiego*,
- *pozostałe 10% dobiera się losowo*.

A jak uzyskać losowo pojedynczy gen albo wygenerować w ten sposób cały chromosom? Przyjrzyjmy się kodowi:

```
char GetRandomGene() => Genes[random.Next(Genes.Length)];
string GetRandomChromosome()
{
    string chromosome = string.Empty;
    for (int i = 0; i < Target.Length; i++)
    {
        chromosome += GetRandomGene();
```

```
        }
        return chromosome;
    }
```

Potrzebna będzie jeszcze jedna metoda, `GetFitness`, która po prostu zwróci liczbę znaków pasujących do docelowego tytułu książki. Oto kod tej metody:

```
int GetFitness(string chromosome)
{
    int fitness = 0;
    for (int i = 0; i < Target.Length; i++)
    {
        if (chromosome[i] == Target[i]) { fitness++; }
    }
    return fitness;
}
```

Na koniec przyjrzyjmy się metodzie `Print`:

```
void Print() => Console.WriteLine(
    $"Pokolenie {generationNo:D2}:
    {population[0].Chromosome} / {population[0].Fitness}");
```

Po uruchomieniu kodu wyświetlone zostaną najlepiej dostosowane osobniki z każdej generacji, co widać na poniższych danych wyjściowych:

```
Pokolenie 00: PŻmKGzPŹREŚżŃbayEJkólJEKyóaNty jŹśIęJŃOc / 4
Pokolenie 01: uąuMwfzWy aaWźICNPńŹąćeHćŻą RAyCzbbÓą / 5
Pokolenie 02: SfrÓiućżUhoG oh ŁF iFśNAJmuWsoycFJŃŹ fa / 6
Pokolenie 03: IDenńuućŚĘaÓjlSeUNPleodhĄIA m#OękRąZU# / 7
Pokolenie 04: IęAnZduraYĘaKŃlSeDŁWlgożhLIA mFŹękyRhÓś# / 10
Pokolenie 05: SĄUUbHurYHeŁEONhŚpoazjArĈtmł#  NDzyam CZ / 13
Pokolenie 10: SĄrUTburYHPaKOąhŚŁoaFgźrĈtmf UTóDzyam C# / 17 (...)
Pokolenie 15: hlrWkŁ ry óanych n hpĘoryĈm  B #ęz kŹ B# / 22 (...)
Pokolenie 20: StYuFturcńdanuch i algĄrĄtmZ N językOkŃ Ck / 28 (...)
Pokolenie 30: Strukłury daUDch i algorymmT w językR C# / 34 (...)
Pokolenie 40: StruktZry danyco i algorytmy w językC C# / 37 (...)
Pokolenie 50: Struktury Łanych i algorytmy w języku ł# / 38 (...)
Pokolenie 60: Struktury danych i aĄgorytmy w języku C# / 39 (...)
Pokolenie 67: Struktury danych i algorytmy w języku C# / 40
```

Jakim cudem tak się dzieje? Po prostu napisany przez nas algorytm steruje kolejnymi generacjami i doprowadza do ewolucji osobników, dzięki czemu daje oczekiwany wynik.

Odgadywanie hasła

Jako przykład **algorytmu siłowego** utwórzmy program *generujący wszystkie możliwe hasła i próbujący odgadnąć to właściwe*, przy czym hasła będą składać się wyłącznie z małych liter i cyfr. Program zaczyna od haseł o długości dwóch znaków i dochodzi do ośmiu.

Oto jak przedstawia się pierwszy fragment kodu:

```
using System.Diagnostics;
using System.Text;

const string secretPassword = "csharp";
int charsCount = 0;
char[] chars = new char[36];

for (char c = 'a'; c <= 'z'; c++)
{
    chars[charsCount++] = c;
}

for (char c = '0'; c <= '9'; c++)
{
    chars[charsCount++] = c;
}
```

Najpierw definiujemy tajne hasło, które będziemy próbowali odgadnąć w dalszej części kodu. Następnie tworzymy tablicę dostępnych znaków, to znaczy małych liter i cyfr. Na końcu tego fragmentu kodu zmienna `charsCount` zawiera liczbę dostępnych znaków.

Najciekawszym fragmentem tego kodu jest pętla `for`, której poszczególne iteracje odpowiadają konkretnej długości hasła, od dwóch do ośmiu znaków. Oto jej kod:

```
for (int length = 2; length <= 8; length++)
{
    Stopwatch sw = Stopwatch.StartNew();
    int[] indices = new int[length];
    for (int i = 0; i < length; i++) { indices[i] = 0; }

    bool isCompleted = false;
    StringBuilder builder = new();
    long count = 0;
    while (!isCompleted)
    {
        builder.Clear();
        for (int i = 0; i < length; i++)
        {
            builder.Append(chars[indices[i]]);
        }

        string guess = builder.ToString();
        if (guess == secretPassword)
        {
            Console.WriteLine("Znaleziono.");
        }

        count++;

        if (count % 10000000 == 0)
        {
            Console.WriteLine($" > Sprawdzono: {count}.");
```

```
            }

            indices[length - 1]++;
            if (indices[length - 1] >= charsCount)
            {
                for (int i = length - 1; i >= 0; i--)
                {
                    indices[i] = 0;
                    indices[i - 1]++;
                    if (indices[i - 1] < charsCount) { break; }
                    if (i - 1 == 0 && indices[0] >= charsCount)
                    {
                        isCompleted = true;
                        break;
                    }
                }
            }
        }

        sw.Stop();
        int seconds = (int)sw.ElapsedMilliseconds / 1000;
        Console.ForegroundColor = ConsoleColor.White;
        Console.WriteLine($"Znaków: {length}, czas: {seconds}s");
        Console.ResetColor();
    }
```

Tablica `indices` ma długość równą wartości zmiennej `length`. Każdy jej element przechowuje pewien indeks tablicy `chars`, wskazujący znak, który obecnie znajduje się na *i*-tej pozycji ciągu. W każdej iteracji pętli `while` zmieniamy wartości tablicy `indices` dopóty, dopóki nie wykorzystamy wszystkich możliwych kombinacji indeksów.

Odgadywane hasło zapisujemy poza tym w zmiennej `guess`, którą możemy teraz albo wypisać na konsoli, albo po użyciu haszowania porównywać ze skrótem hasła, które chcemy odgadnąć. Ponieważ program jest tylko demonstracją algorytmu siłowego, jego działanie nie zatrzymuje się po odgadnięciu hasła. Pozwala to uzyskać więcej wyników wydajnościowych i zaobserwować, jak długość hasła wpływa na czas jego odgadnięcia.

Jak widać, podejście siłowe jest bardzo proste, ale co z wydajnością? W powyższym kodzie użyto klasy `Stopwatch`, co pozwoliło uzyskać pewne wyniki. Generowanie wszystkich możliwych wariantów haseł składających się z dwóch znaków nie zajmuje nawet milisekundy. W przypadku haseł trzy- i czteroznakowych czas również jest bardzo krótki, znacznie poniżej 100 milisekund. W przypadku haseł pięcioznakowych czas wzrasta do około 2 sekund, a generowanie haseł o długości sześciu znaków trwa prawie minutę. Jeśli dodamy do tego mechanizm haszowania hasła i będziemy porównywać jego skrót ze skrótem hasła docelowego oraz uwzględnimy to, że hasła mogą zawierać także wielkie litery i wiele innych znaków, algorytm siłowy w odgadywaniu dłuższych haseł wydaje się po prostu niepraktyczny.

Muszę zaznaczyć, że przedstawione wyniki wydajnościowe pochodzą z mojego komputera i będą się różnić w zależności od urządzenia. Pokazałem je tylko po to, by wskazać, że przy wzroście długości hasła z każdym kolejnym znakiem czas potrzebny do

jego odgadnięcia znacznie się wydłuża. Płynie z tego morał. żeby zawsze stosować skomplikowane hasła zawierające małe i wielkie litery, cyfry oraz znaki specjalne. Oczywiście długość hasła też jest ważna.

Podsumowanie

Masz już za sobą dziewiąty rozdział tej książki, w której badamy struktury danych i algorytmy w kontekście języka C#. Tym razem skupiliśmy się na praktycznych przykładach algorytmów, z fragmentami kodu, dokładnymi opisami i zwięzłymi informacjami o tym, jakiego typu algorytm został użyty w danym przykładzie.

Najpierw pokazałem trzy sposoby implementacji prostego algorytmu obliczania liczby z **ciągu Fibonacciego**. Zaprezentowałem proste podejście rekurencyjne, a także metodę zstępującą i wstępującą, czyli techniki programowania dynamicznego.

W następnym przykładzie przedstawiłem rozwiązywanie problemu **wydawania reszty** metodą zachłanną. Następny w kolejności był algorytm „dziel i zwyciężaj", pozwalający znaleźć **najbliższą parę punktów** na dwuwymiarowej powierzchni. W czwartym przykładzie pokazałem rekurencyjne **generowanie fraktali** i rysowanie ich na mapie bitowej.

Następne dwa przykłady, ilustrujące algorytmy z nawrotami, dotyczyły rozwiązywania problemu **znajdowania wyjścia z labiryntu** i łamigłówki **sudoku**. W tych przykładach również wykorzystana została rekurencja.

Kolejne interesujące zagadnienie było związane z algorytmem genetycznym, należącym do podtypu algorytmów heurystycznych. Został on zaprzęgnięty do **odgadnięcia tytułu książki** według reguł sformułowanej przez Darwina teorii ewolucji i doboru naturalnego.

W ostatnim przykładzie przy użyciu algorytmu siłowego **odgadywaliśmy tajne hasło** przez sprawdzanie wszystkich możliwych jego wariantów. Zobaczyliśmy, że wraz z wydłużaniem się hasła znacznie wzrasta czas potrzebny do jego odgadnięcia.

Najwyższy czas, by przejść do podsumowania całości i przyjrzeć się wszystkim strukturom danych zaprezentowanym do tej pory w książce. Odwróć kartkę i rozpocznij lekturę ostatniego rozdziału!


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Kilkusetkrotne zwiększenie wydajności kodu na wyciągnięcie ręki!

Projektowanie aplikacji jest wymagającym zadaniem, zwłaszcza jeśli trzeba rozwiązywać złożone problemy. W takich przypadkach należy mieć na uwadze również wydajność kodu, aby program działał płynnie na urządzeniach o ograniczonych zasobach. Takie zadania bywają naprawdę trudne i wymagają wiedzy, w tym dotyczącej struktur danych i algorytmów.

Tę praktyczną książkę docenią programiści C#. Zaczniesz od zapoznania się z zasadami działania algorytmów, aby później przejść do różnych struktur danych: tablic, list, stosów, kolejek, słowników i zbiorów. Poszczególne przykłady zostały zilustrowane fragmentami kodu i rysunkami. Opanujesz także sortowanie tablic przy użyciu rozmaitych algorytmów, co solidnie ugruntuje Twoje umiejętności. Następnie poznasz bardziej złożone struktury danych i algorytmy służące do różnych zadań, jak wyznaczanie najkrótszej ścieżki w grafie czy rozwiązywanie łamigłówek. W ten sposób nauczysz się budować w języku C# komponenty algorytmiczne, które bez problemu zastosujesz w rozmaitych aplikacjach, również internetowych i na platformy mobilne.

Z tą książką nauczysz się:

- podstaw algorytmów i ich klasyfikacji
- przechowywać dane w ustrukturyzowany sposób
- budować aplikacje wzbogacone o stosy, kolejki, tablice z haszowaniem, słowniki i zbiory
- tworzyć wydajne aplikacje z użyciem algorytmów związanych z drzewami
- podnosić wydajność swoich rozwiązań przy użyciu grafów
- implementować algorytmy pozwalające rozwiązywać łamigłówki i generować fraktale

Dr hab. inż. Marcin Jamro jest przedsiębiorcą, ekspertem i programistą z międzynarodowym doświadczeniem. Służy wiedzą ekspercką w projektach międzynarodowych i inwestuje w nowoczesne rozwiązania. Jest autorem książek i artykułów dotyczących inżynierii oprogramowania. Posiada certyfikaty: MCPD, MCTS, MCP i CAE. Ma bogate doświadczenie w zarządzaniu rozbudowanymi projektami informatycznymi.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1889-4
9 788328 918894
Cena: 89,00 zł