10 ALGORYTMÓW

Które powinien znać, każdy programista

Mateusz Rus
Programowanie z Pasją

www.mateuszrus.pl

Mateusz Rus

Cześć!

Bardzo się cieszę, że dołączyłeś do grona posiadaczy tego wyjątkowego PDF, w którym starałem się opisać najdokładniej jak to możliwe dziesięć przydatnych algorytmów pojawiających się na rozmowach rekrutacyjnych oraz w materiałach szkoleniowych.

Algorytmy, które zostały przedstawione to:

1. NWD
2. Odwrócenie tablicy
3. Równanie kwadratowe
4. Kolejka LIFO
5. Sortowanie Bąbelkowe
6. Szyfr Cezara
7. Silnia
8. Wyszukiwanie binarne
9. Zmodyfikowana StringMerge
10. Naprzemienność dużych i małych liter

Każdy z algorytmów bardzo często pojawia się w toku nauczania na studiach lub podczas rozmów o pracę. Język, jaki został użyty do napisania każdego z nich to C#. Nie stanowi to jednak przeszkody, by w analogiczny sposób przepisać je samodzielnie na języki takie jak Java, C++ czy Python. Możesz to potraktować jako pracę domową!

1. NWD

Pierwszy algorytm, z którym dziś się zapoznamy to matematyczne rozwiązanie problemu **NWD**, czyli największego wspólnego dzielnika. Inaczej mówiąc, jest to liczba naturalna, przez którą możemy podzielić dwie inne liczby całkowite tak, by nie została reszta. Algorytm ten nazywa się też zoptymalizowanym algorytmem Euklidesa. Dla przykładu wezmę parę liczb 20 oraz 30. Są one wspólnie podzielne przez 1, 2, 5 oraz 10. NWD w tym przypadku to 10, ponieważ jak sama nazwa wskazuje, jest to największy wspólny dzielnik.

Poniżej przedstawiam rozwiązanie tego problemu za pomocą rekurencji, która wykonywana jest w prywatnej metodzie typu **int** o nazwie **NWD** (*linie 28-34*). Na wejście podawana jest para liczb, następnie w instrukcji warunkowej sprawdzana jest druga z nich – **liczbaB**. Jeżeli jest ona różna od 0, to następuje wywołanie raz jeszcze **NWD** z innymi danymi wejściowymi – jako **liczbaA** wpisywana jest **liczbaB,** a w drugim parametrze jako **liczbaB** wpisywane jest modulo z dzielenia **liczbaA** przez **liczbaB**. Dzieje się to do momentu osiągnięcia w drugim parametrze wartości zero. Wtedy zwracany jest parametr pierwszy, który przetrzymuje największy wspólny dzielnik.

Metoda **Main** ma cztery zadania, po pierwsze pobiera od użytkownika wartości **liczbyA** i **liczbyB**, następnie uruchamia metodę **NWD** i wypisuje wynik na konsoli. Wszystko to jest objęte blokiem **try…catch** do złapania ewentualnych wyjątków – gdyby pojawił się problem z parsowaniem lub wartością liczby poza zakresem **int**.

Na rysunku poniżej zobrazowane jest działanie programu dla danych wejściowych *20 i 30* oraz *28 i 24*.

Numer iteracji	Liczba A	Liczba B	Modulo
1	20	30	20
2	30	20	10
3	20	10	0
4	10	0	

Numer iteracji	Liczba A	Liczba B	Modulo
1	28	24	4
2	24	4	0
3	4	0	

```
1  using System;
2
3  namespace NWD
4  {
5      class Program
6      {
7          public static void Main()
8          {
9              try
10             {
11                 Console.WriteLine("Podaj pierwszą liczbę");
12                 int pierwsza = int.Parse(Console.ReadLine());
13
14                 Console.WriteLine("Podaj drugą liczbę");
15                 int druga = int.Parse(Console.ReadLine());
16
17                 Console.WriteLine(NWD(pierwsza, druga));
18
19                 Console.ReadLine();
20             }
21             catch (Exception)
22             {
23                 Console.WriteLine("Pojawił się błąd");
24                 throw;
25             }
26         }
27
28         private static int NWD(int liczbaA, int liczbaB)
29         {
30             if (liczbaB != 0)
31                 return NWD(liczbaB, liczbaA % liczbaB);
32
33             return liczbaA;
34         }
35     }
36 }
```

2. Odwrócenie tablicy

W dzisiejszych czasach odwrócenie tablicy z użyciem języków programowania i dostępnych metod w bibliotekach systemowych wywołuje się jedną linią kodu za pomocą, chociażby funkcji typu **Reverse**. Dlaczego więc zaproponowałem algorytm tego typu w tym pliku?

Powód jest jeden, ale jakże ważny. Jest to zadanie, które dwukrotnie otrzymałem na rozmowie o prace. Oczywiście rozwiązanie typu zastosowanie gotowej funkcji odwracającej nie wchodziło w grę.

Zadanie jak widać po objętości kodu jest proste, jednak trzeba sobie uświadomić, w jaki sposób podejść do niego. Założenie przyjąłem następujące. Na wejście użytkownik proszony jest o podanie ciągu znaków (*mogą to być na przykład liczby bądź litery alfabetu*) oddzielone spacjami. Następnie ciąg znaków dzielony jest po wspomnianej spacji i powstałe podciągi wstawiane są do tablicy typu **string**.

Cała rzecz dzieje się w pętli **for**, która iteruje po tablicy **splitTable** od ostatniego elementu do pierwszego.

Tak dla przykładu mamy ciąg podany na wejście: **1 2 3 4 5**.

Ma on pięć elementów oddzielonych spacjami, czyli tablica **splitTable** ma rozmiar 5. Iterację zaczynamy od ostatniego elementu tablicy.

- splitTable[4] = 5,
- splitTable[3] = 4,
- splitTable[2] = 3,
- splitTable[1] = 2,
- splitTable[0] = 1

Kolejność od 4 elementu do 0 umożliwia pętla **for**, która zaczyna swoje działanie od iteratora, który jest równy wielkości tabeli **splitTable** pomniejszonego o 1 (*pomniejszam o jeden, ponieważ numeracja tabeli zaczyna się od 0*). Następnie z każdym kolejnym obrotem pętli obniża się iterator o 1 aż do osiągniecia wartości 0, a w ciele pętli dopisuję do pola tekstowego **returnTable** wartość elementu tablicy, na którą wskazuje iterator.

Na zakończenie przed prezentacją danych uruchamiam jeszcze metodę z biblioteki System o nazwie **EndTrim**, która to metoda usunie wszystkie białe znaki z końca tekstu. Jest to zabieg czysto kosmetyczny i może pomóc na

```
using System;

namespace OdwrocenieTablicy
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("Podaj ciąg znaków oddzielony spacjami: ");

            string data = Console.ReadLine();
            string[] splitTable = data.Split(' ');

            string returnTable = "";

            for (int i = splitTable.Length - 1; i >= 0; i--)
            {
                returnTable += splitTable[i] + " ";
            }

            Console.WriteLine(returnTable.TrimEnd());

            Console.ReadLine();
        }
    }
}
```

rozmowie rekrutacyjnej zaplusować przed rekruterem. Pamiętaj, że detale mają znaczenie. Na koniec dodam, że problem ten miałem do rozwiązania na kartce papieru, więc detale takie, o których przypomina kompilator, będą jeszcze bardziej brane pod uwagę niż zwykle.

3. Funkcja kwadratowa

Kto z nas nie miał w szkole funkcji kwadratowej. Algorytm numer trzy rozwiązuje problem wyznaczania miejsc zerowych. Nie będę w tym miejscu zgłębiał tematu funkcji kwadratowej – odsyłam ciekawskich lub niezorientowanych do Google. Skupię się na trzech aspektach funkcji kwadratowej. Trzeba w niej obliczyć deltę, sprawdzić, czy jest w jednym z trzech stanów (*dodatnia, zerowa lub ujemna*), a następnie obliczyć ewentualne miejsca zerowe. Dla dodatniej mamy ich dwa, dla zerowej jedno a dla ujemnej nie ma takiego miejsca, ponieważ parabola nie przecina na wykresie osi OX.

Program standardowo jest konsolowy. Na wejście użytkownik zmuszony jest podać trzy wartości liczbowe oddzielone spacjami. Pierwsza wartość będzie

reprezentować **A**, druga **B** a trzecia **C**. Poniżej przedstawiony jest wzór opisujący funkcję kwadratową:

$$f(x) = ax^2 + bx + c$$

W liniach 10-13 wykonuję parsowanie do wartości zmiennoprzecinkowej. Zakładamy, że użytkownik poda poprawne liczby na wejście. Kolejnym zadaniem programu jest obliczenie delty. Deltę liczymy z poniższego wzoru:

$$\Delta = b^2 - 4ac$$

Gdy program obliczy deltę, możemy przejść do najważniejszego punktu programu. W instrukcji warunkowej sprawdzamy, czy delta jest ujemna, zerowa lub dodatnia. Gdy wartość delty jest poniżej zera, wtedy sytuacja jest klarowna i pozostaje nam wypisać komunikat na ekran, że funkcja nie posiada miejsc zerowych.

Sytuacja komplikuje się, gdy delta równa się zero (*linie 22-23*). Wtedy parabola w jednym miejscu przecina oś OX. Aby to obliczyć, korzystamy ze wzoru na $\mathbf{x_0}$:

$$x_0 = -b / 2a$$

Jeszcze bardziej skomplikowany jest przypadek dodatniej delty (*linie 27-30*). Wtedy musimy obliczyć dwa miejsca zerowe $\mathbf{x_1}$ oraz $\mathbf{x_2}$. Wyliczymy je z następującego wzoru:

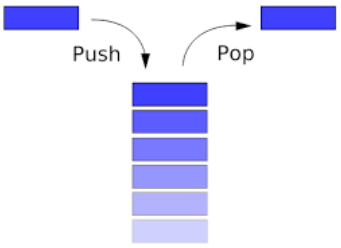
$$x_1 = -b - \sqrt{\Delta} / 2a$$

$$x_2 = -b + \sqrt{\Delta} / 2a$$

```
1  using System;
2
3  public class Test
4  {
5      public static void Main(string[] args)
6      {
7          Console.WriteLine("Podaj wartości A, B oraz C oddzielone spacjami:");
8          string wejscie = Console.ReadLine();
9
10         double A = double.Parse(wejscie.Split(' ')[0]);
11         double B = double.Parse(wejscie.Split(' ')[1]);
12         double C = double.Parse(wejscie.Split(' ')[2]);
13         double delta = B * B - 4 * A * C;
14         double x0, x1, x2;
15
16         if (delta < 0)
17         {
18             Console.WriteLine("Delta wyszła ujemna i równa się: " + delta.ToString() + ". Funkcja nie posiada miejsc zerowych.");
19         }
20         else if (delta == 0)
21         {
22             x0 = -B / (2 * A);
23             Console.WriteLine("Delta równa się zero. Funkcja posiada jedno miejsce zerowe, które równe jest: " + x0.ToString());
24         }
25         else
26         {
27             x1 = (-B - Math.Sqrt(delta)) / (2 * A);
28             x2 = (-B + Math.Sqrt(delta)) / (2 * A);
29             Console.WriteLine("Delta wyszła dodatnia i równa się: " + delta.ToString() +
30             ". Funkcja posiada dwa miejsca zerowe. X1 = " + x1.ToString() + ", X2 = " + x2.ToString());
31         }
32
33         Console.Read();
34     }
35 }
```

4. Kolejka LIFO

Przed nami czwarty algorytm. Tym razem pokaże rozwiązanie jednego z problemów, który było mi dane rozwiązać na studiach. Chodzi mianowicie o kolejkę LIFO (Last In First Out). LIFO to inaczej stos, który możemy porównać do stosu książek – odkładamy na górę książkę (push) i pobieramy ją później z góry (pop).



Push i Pop to metody do wkładania i zdejmowania ze stosu. Takie też zaimplementuję w moim przykładzie. Dodatkowo dodam metodę sortującą **SortLifo**. Metoda **ShowLifo** wypisze w konsoli wszystkie elementu od dołu do góry.

Omówienie rozpocznę od głównej metody, którą jest metoda układania wartości na kolejce – **Push**. Na wejście przyjmuje parametry typu **int**, w który będziemy wkładać wartość od -5 do 5. Takie było wymaganie dotyczące zadania. Jedynym zadaniem funkcji jest wpisanie elementu do tablicy i podbicie licznika **tableSize**.

Następną metodą jest metoda zdejmująca wartość ze stosu – **Pop**. Sprawdzane jest, czy wartość indeksu tablicy jest większa od zera. Gdy tak jest, wtedy wyświetlana jest informacja o pobraniu elementu ostatniego w tablicy, a następnie obniżenie pozycji **tableSize** o jeden.

Metoda **ShowLifo** to nic innego jak zwykła pętla, która wypisuje na ekran elementy zapisane w tablicy od tego położonego najniżej stosu do tego najwyżej.

Ostatnia metoda to **SortLifo,** dzięki której sortujemy nasz stos od najwyższej cyfry do najniższej. Sortowanie zbudowane jest na dwóch pętlach, które nazywa się sortowaniem bąbelkowym.

Na koniec w funkcji głównej **Main** utworzona została pętla inicjalizująca z funkcją **Random**, która to funkcja losuje cyfry z przedziału -5 do 5 i wpisuje

do tablicy. Następnie wykonywane jest kilka operacji w oparciu o opisane metody.

```
using System;

namespace Kolejka_LIFO
{
    class Program
    {
        static int tableSize = 0;
        static int[] table = new int[10];

        static void Main(string[] args)
        {
            Random rnd = new Random();
            for (int i = 0; i < 10; i++)
            {
                Push(rnd.Next(-5, 5));
            }

            ShowLifo();
            Pop();
            ShowLifo();

            SortLifo();
            ShowLifo();

            Console.ReadLine();
        }

        static void Push(int element)
        {
            table[tableSize] = element;
            tableSize++;

            Console.WriteLine("Dodałeś " + element.ToString());
        }

        static void Pop()
        {
            if (tableSize > 0)
            {
                Console.WriteLine("Pobrany element to: " + table[tableSize - 1]);
                tableSize--;
            }
            else
            {
                Console.WriteLine("Stos nie zawiera elementów!");
            }
        }

        static void ShowLifo()
        {
            Console.WriteLine("Wszystkie elementy stosu: ");

            if (tableSize > 0)
            {
                for (int i = 0; i < tableSize; i++)
                {
                    Console.WriteLine("Element numer " + (i + 1).ToString() + ": " + table[i]);
                }
            }
            else
            {
                Console.WriteLine("Brak elementów na stosie");
            }
        }

        static void SortLifo()
        {
            if (tableSize > 0)
            {
                int temp;
                for (int i = 0; i < table.Length - 1; i++)
                {
                    for (int j = i + 1; j < table.Length; j++)
                    {
                        if (table[i] < table[j])
                        {
                            temp = table[i];
                            table[i] = table[j];
                            table[j] = temp;
                        }

                    }
                }

                Console.WriteLine("Posortowano stos!");
            }
            else
            {
                Console.WriteLine("Stos nie zawiera elementow do posortowania ");
            }
        }
    }
}
```

5. Sortowanie Bąbelkowe

Sortowania tego użyliśmy w poprzednim zadaniu przy kolejce. Poniżej zaimplementowałem przykładowe rozwiązanie z tablicą 20 elementową liczb, które zostaną wygenerowane za pomocą funkcji **Random** (linie 9-16)**.**

Mając przygotowaną tablicę losowych liczb z zakresu od 0 do 99999, możemy zająć się główną funkcją **Sort**, która na wejście przyjmuje tablicę **int.** Na początku do zmiennej **tableLen** pobierany jest jej rozmiar. Następnie dzieje się cała magia sortowania bąbelkowego. W pętli **while** nastepuje iteracja po wszystkich elementach tablicy. Pętla **for** wewnątrz również iteruje po elementach tablicy, „przepychając" na górę tabeli najmniejszy przetrzymywany w **tmp** element.

Działa to tak jak na załączonym poniżej obrazku.

A	B	C	D	E	Opis
9	8	2	4	13	A > B. Prawda, więc zamiana 9 z 8
8	9	2	4	13	B > C. Prawda, więc zamiana 9 z 2
8	2	9	4	13	C > D. Prawda, więc zamiana 9 z 4
8	2	4	9	13	D > E. Fałsz, nie robimy nic i wracamy do początku zmniejszając o 1 iterator
8	2	4	9	13	A > B. Prawda, więc zamiana 8 z 2
2	8	4	9	13	B > C. Prawda, więc zamiana 8 z 4
2	4	8	9	13	C > D. Fałsz, nie robimy nic i wracamy do początku zmniejszając o 1 iterator
2	4	8	9	13	A > B. Fałsz, nie robimy nic i idziemy dalej
2	4	8	9	13	B > C. Fałsz, nie robimy nic i wracamy do początku zmniejszając o 1 iterator
2	4	8	9	13	A > B. Fałsz, nie robimy nic i wracamy do początku zmniejszając o 1 iterator
2	4	8	9	13	Koniec sortowania. Iterator doszedł do najmniejszej wartości

Jak widać pętla **do while** idzie po wszystkich elementach tablicy (od góry do dołu powyższego rysunku). Pętla for natomiast iteruje od A do E, zamienia ewentualnie miejscami liczby i zmniejszając o jeden iterator, dochodząc do końca. Jak widzimy, sortowanie bąbelkowe może wykonywać kilka niepotrzebnych ruchów, tak jak wszystkie te, które są fałszem na rysunku powyżej.

W przypadku mniejszej ilości elementów nie jest to problemem, jednak przy pracy z dużym zbiorem danych czas wykonania sortowania może być uciążliwe. Złożoność czasowa tego algorytmu to **O(n^2)** a pamięciowa **O(1)**. O tym, czym jest złożoność obliczeniowa algorytmu, dowiesz się z 7 wpisu na blogu. Zerknij już teraz!

```
using System;

namespace SortowanieBabelkowe
{
    class Program
    {
        static void Main(string[] args)
        {
            Random rnd = new Random();

            int[] table = new int[20];

            for (int i = 0; i < 20; i++)
            {
                table[i] = rnd.Next(0, 99999);
            }

            Console.WriteLine("Przed posortowaniem:" );
            for (int i = 0; i < table.Length; i++)
            {
                Console.Write(table[i] + ", ");
            }

            Sort(table);

            Console.WriteLine(Environment.NewLine + Environment.NewLine  + "Po posortowaniu: ");
            for (int i = 0; i < table.Length; i++)
            {
                Console.Write(table[i] + ", ");
            }

            Console.ReadKey();
        }

        static void Sort(int[] table)
        {
            int tableLen = table.Length;
            do
            {
                for (int i = 0; i < tableLen - 1; i++)
                {
                    if (table[i] > table[i + 1])
                    {
                        int tmp = table[i];
                        table[i] = table[i + 1];
                        table[i + 1] = tmp;
                    }
                }
                tableLen--;
            }
            while (tableLen > 1);
        }
    }
}
```

6. Szyfr Cezara

Kolejny algorytm, z którym mierzą się studenci podczas zajęć. Algorytm szyfrowania za pomocą przesunięcia jest jednym z najłatwiejszych tego typu algorytmów. Polega to na tym, by każdy znak tekstu jawnego zastąpić innym znakiem, który będzie przesunięty o stałą x. Dla przykładu postarajmy się zaszyfrować imię *MATEUSZ*, gdzie stałą będzie $x = 10$. Mając przed sobą tablicę ASCII, wystarczy znaleźć miejsce wystąpienia każdej z liter tekstu jawnego i dodać do niego 10 – w przypadku dojścia do końca alfabetu (brak polskich znaków, więc ostatnią literą będzie Z) wracamy na początek i zaczynamy od litery A. Podstawowe 26 dużych liter alfabetu mieszczą się w zakresie od 65 do 90 włącznie. Poniżej przykład wartości dziesiętnych odpowiadających literom.

Tekst jawny	Tekst zaszyfrowany
M = 77	W = 8
A = 65	K =75
T = 84	D = 68
E = 69	O = 79
U = 85	E = 69
S = 83	C = 67
Z = 90	J = 74

Szyfrowaniem ciągów znaków pisanych dużymi literami zajmuje się program, którego kod przedstawiłem poniżej. Składa się on z głównej metody typu **void** o nazwie **Szyfruj**, która przyjmuje jako parametry klucz (*w przypadku powyżej będzie to dodania liczba 10*) oraz tablicę znaków (*tablica liter MATEUSZ*). Przyjęte założenie jest takie, by klucz nie przekraczał wartości z zakresu od -26 do 26, ponieważ bazujemy tylko na literach drukowanych, a takich w tablicy ASCII jest właśnie 26 (*osobny artykuł na temat ASCII dostępny jest na blogu*). Jeżeli klucz jest dodatni to następuje sprawdzenie od A w górę, pod warunkiem, że nie następuje przekroczenie litery Z. W przypadku ujemnego klucza zaczynamy analizę wstecz od Z w dół, chyba że dojdzie do litery A.

Teraz w metodzie głównej prosimy użytkownika o wpisanie ciągu znaków oraz klucza z wymaganego zakresu. Dzięki tak napisanej metodzie **Szyfruj**

możemy zarówno szyfrować ciągi znaków, jak i odszyfrowywać podając przeciwny klucz do wejściowego.

Temat szyfrowania jest bardzo obszerny i na pewno poruszę go dokładniej na blogu.

```
using System;

namespace Szyfr_Cezara
{
    class Program
    {
        static void Main(string[] args)
        {
            int klucz;

            Console.WriteLine("Podaj wyraz składający się z dużych liter: ");
            string tabStr = Console.ReadLine();

            char[] tab = new char[tabStr.Length];
            tab = tabStr.ToCharArray();

            Console.WriteLine("Podaj klucz z przedziału [-26..26]: ");
            klucz = int.Parse(Console.ReadLine().ToString());

            Szyfruj(klucz, tab);

            Console.WriteLine("Po zaszyfrowaniu: " + new string(tab));

            Szyfruj(-klucz, tab);

            Console.WriteLine("Po odszyfrowaniu: " + new string(tab));
            Console.ReadLine();

        }

        static void Szyfruj(int klucz, char[] tab)
        {
            int dl = tab.Length;

            if (!(klucz >= -26 && klucz <= 26))
                return;

            if (klucz >= 0)
            {
                for (int i = 0; i < dl; i++)
                {
                    if (tab[i] + klucz <= 'Z')
                    {
                        tab[i] = Convert.ToChar(tab[i] + klucz);
                    }
                    else
                    {
                        tab[i] = Convert.ToChar(tab[i] + klucz - 26);
                    }
                }
            }
            else
            {
                for (int i = 0; i < dl; i++)
                {
                    if (tab[i] + klucz >= 'A')
                    {
                        tab[i] = Convert.ToChar(tab[i] + klucz);
                    }
                    else
                    {
                        tab[i] = Convert.ToChar(tab[i] + klucz + 26);
                    }
                }
            }

        }
    }
```

7. Silnia

Silnia jest prostym działaniem matematycznym, którego uczymy się już na etapie szkoły średniej. Jest on też bardzo ciekawym przypadkiem do stworzenia algorytmu. Zarówno rekurencyjnego, jak i iteracyjnego. Wróćmy jednak do początku. Silnia jest to iloczyn wszystkich liczb naturalnych aż do n. Dla przykładu silnia 5 jest równa **1 * 2 * 3 * 4 * 5**, czyli 120.
Taka konstrukcja matematyczna powoduje, że silnia z każdą kolejną liczbą naturalną bardzo szybko rośnie i **już 20!** jest równa *2 432 902 008 176 640 000.*
Poniżej kod implementujący dwie funkcje – **silniaRecursion** oraz **silniaIteration**. Pierwsza z nich to typowy przykład związany z rekurencją. Na wejście podajemy liczbę, dla której chcemy obliczyć silnię, a następnie w ciele funkcji implementujemy prosty warunek i sprawdzamy, czy wartość na wejściu jest mniejsza od 1. Gdy warunek nie jest prawdą, następuje wywołanie funkcji przez samą siebie z elementem o jeden mniejszym oraz przemnożenie przez element podany na wejściu.
Mniej elegancki algorytm zastosowany został w **silniaIteration**. Co prawda na wejście również podawana jest liczba, dla której silnia będzie liczona, jednak cała operacja odbywa się w pętli **for**, która z każdą iteracją mnoży rezultat przez wartość iteratora.
Warto na koniec zaznaczyć, że dla dużego **n** silnia nie będzie możliwa do policzenia ze względu na ograniczenia wielkości typu **long**.

```
using System;

namespace Silnia
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Podaj dla jakiej liczby obliczyc silnie: ");
            int n = int.Parse(Console.ReadLine());

            Console.WriteLine("Silnia obliczona rekurencyjnie. " + n + "! = " + silniaRecursion(n));
            Console.WriteLine("Silnia obliczona iteracyjnie. " + n + "! = " + silniaIteration(n));
            Console.ReadLine();
        }

        private static long silniaRecursion(int i)
        {
            if (i < 1)
            {
                return 1;
            }
            else
            {
                return i * silniaRecursion(i - 1);
            }
        }

        private static long silniaIteration(int n)
        {
            long result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            return result;
        }
    }
}
```

8. Wyszukiwanie binarne

Jest to jeden z najpopularniejszych algorytmów szukających. Polega on na wyszukiwaniu zadanego elementu w tablicy. Wyszukiwanie binarne zostało oparte na algorytmie „*dziel i zwyciężaj*”. Jest to bardzo wydajna metoda o logarytmicznej złożoności obliczeniowej. Do wyszukania elementu w milionowym zbiorze danych potrzebuje jedynie 20 cykli. Jest jednak jeden haczyk. Aby wszystko zadziałało,musimy wcześniej posortować tablicę rosnąco (*lub malejąco, ale wtedy odwrotnie będzie działał algorytm „dziel i zwyciężaj”*).

Ogólnie najważniejszą częścią zadania będzie w naszym przypadku dzielenie tablicy na połowy i szukanie elementu w danej połowie poprzez kolejne dzielenia aż pozostanie jeden szukany element.

Zadanie rozpoczniemy od pobrania informacji od użytkownika o wielkości tablicy i zadeklarowanie jej. Następnie w pętli generujemy losowo liczby z przedziału od -10000 do 10000 i wstawienie je w odpowiednie miejsca. Mamy więc już tablicę elementów. Następnie sortujemy za pomocą **Linq** i metody **OrderBy**. W kolejnym kroku postanowiłem, że wypisze posortowane elementy tablicy w konsoli, tak by użytkownik wiedział, jakie elementy zostały wyszukane oraz na którym miejscu w tabeli się znajdują. Będzie to potrzebne w kolejnym kroku, gdzie poprosimy użytkownika o podanie szukanej liczby.

Ostatnim etapem jest wywołanie przygotowanej specjalnie do tego celu metody o nazwie **FindPlace**.

Metoda na wejście przyjmuje tablicę elementów i szukaną liczbę. Do zmiennych **left** oraz **right** wstawiamy zakres tablicy, czyli od 0 do podanej na początku programu wielkości tablicy. Pętla **while** rozpoczyna swe działanie i po każdej iteracji tablica jest dzielona na połowę. Porównywany jest ostatni element przedzielonej tablicy z szukanym przez nas elementem. Jeżeli wartość z tablicy równa się szukanej liczbie, to kończymy pętlę i zwracamy wynik. W innych przypadkach sprawdzamy, czy wartość szukana jest mniejsza czy większa od ostatniego elementu podzielonej tablicy. Wtedy albo zmieniamy parametr **left** albo **right** i tniemy znów na pół pierwszą lub drugą połowę.

```cs
using System;
using System.Linq;

namespace SzukanieBinarne
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Podaj liczbe elementow tablicy:");
            int countElements = int.Parse(Console.ReadLine());

            int[] elements = new int[countElements];

            Random rnd = new Random();
            for (int i = 0; i < countElements; i++)
            {
                elements[i] = rnd.Next(-10000, 10000);
            }

            elements = elements.OrderBy(c => c).ToArray();

            string sortingValues = "";
            for (int i = 0; i < elements.Length; i++)
            {
                sortingValues += elements[i].ToString() + ", ";
            }
            Console.WriteLine("Wylosowane i posortowane elementy: " + sortingValues.TrimEnd());
            Console.WriteLine("Podaj element do wyszukania:");
            int searchingElement = int.Parse(Console.ReadLine());
            Console.WriteLine("Element " + searchingElement.ToString() + " znajduje się na " + FindPlace(elements, searchingElement).ToString() + " w tablicy.");

            Console.ReadLine();
        }

        static int FindPlace(int[] vector, int search)
        {
            int left = 0;
            int right = vector.Length - 1;

            while (left <= right)
            {
                int currentPosition = left + (right - left) / 2;

                if (vector[currentPosition] == search)
                {
                    return currentPosition + 1;
                }

                if (vector[currentPosition] < search)
                {
                    left = currentPosition + 1;
                }

                if (vector[currentPosition] > search)
                {
                    right = currentPosition - 1;
                }
            }

            return -1;
        }
    }
}
```

9. Zmodyfikowana StringMerge

Ciekawy problem znalazłem podczas rozwiązywania zadań na SPOJ. Temat zadania to **StringMerge**, czyli w zasadzie funkcja łącząca dwa zestawy znaków w jeden. Autor zadania poszedł o poziom dalej i zaproponował napisanie funkcji, która będzie łączyła ciąg znaków na przemian, raz jeden

znak z ciągu A, raz z ciągu B aż do momentu osiągnięcia końca krótszego ze stringów.

Link do zadania: https://pl.spoj.com/problems/PP0504B/

Na wejście musimy pobrać ilość zestawów danych. Następnie w pętli for pobieramy za każdym razem parę ciągów znaków oddzieloną spacjami od siebie. Kolejnym etapem jest sprawdzenie, który z ciągów jest krótszy. Po wyznaczeniu tej wartości pozostaje już tylko napisanie pętli, która pobierze po jednym znaku z lewego i prawego stringa i wstawi je połączone ze sobą. Pętla będzie iterować do momentu, aż nie zostanie osiągnięty ostatni znak krótszego ciągu.

Niewątpliwie problem nie jest typowym algorytmem, który stosujemy do rozwiązania konkretnej grupy zagadnień, ale pokazuje, jak prosto można rozwiązać problem na pierwszy rzut oka skomplikowany w zaledwie 35 liniach kodu. O SPOJ i zadaniach tam dostępnych będę czasami pisał na blogu.

```
using System;

namespace StringMerge
{
    class Program
    {
        public static void Main()
        {
            int l = int.Parse(Console.ReadLine());

            for (int i = 0; i < l; i++)
            {
                string we = Console.ReadLine();
                string[] s = we.Split(' ');

                int ile = s[0].Length;

                if (s[0].Length > s[1].Length)
                {
                    ile = s[1].Length;
                }

                string wynik = "";

                for (int j = 0; j < ile; j++)
                {
                    wynik += (s[0][j].ToString() + s[1][j].ToString());
                }
                Console.WriteLine(wynik);
            }
        }
    }
}
```

10. Naprzemienność dużych i małych liter

Ostatni algorytm to zadanie, z którym pewnego razu zmierzył się mój znajomy podczas swoich studiów technicznych. Treść zadania brzmiała mniej więcej tak „*Tekst podany na wejście zapisz na wyjściu jako naprzemienną sekwencję małej i dużej litery*”.
Treść zadania zmieściła się w jednej linii, ale zadanie wydaje się klarowne, mimo że można się zastanawiać nad takimi aspektami jak spacje, znaki specjalne, cyfry, czy to, od której litery rozpoczniemy – dużej czy małej.
Na wejście prosimy użytkownika, by podał ciąg znaków. Następnie w pętli **foreach** przechodzimy po każdym elemencie tablicy (*w C# typ string jest tak naprawdę tablicą typu char*). W banalny sposób sprawdzamy, czy jesteśmy w miejscu tablicy, której indeks jest parzysty czy nie. Dla indeksu parzystego litera będzie duża, a dla indeksu nieparzystego będzie mała.
Jak widać, kolejny teoretycznie skomplikowany problem można rozwiązać za pomocą kilku linii kodu i pewnym sprytnym rozwiązaniom.

```
using System;

namespace Naprzemienność_dużych_i_małych_liter
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Podaj tekst, który ma zostać zmodyfikowany.");
            string str = Console.ReadLine().ToString();
            string result = "";

            int count = 0;
            foreach (char c in str)
            {
                if (count % 2 == 0)
                {
                    result += c.ToString().ToUpper();
                }
                else
                {
                    result += c.ToString().ToLower();
                }
                count++;
            }

            Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}

```

Podsumowanie

Bardzo Ci dziękuję, że wspierasz moje działania. Prezent w formie tego PDF to tak naprawdę kropla w morzu algorytmów i jedyne drobne wprowadzenie. Złapanie zajawki to dopiero początek. Dalej czeka nas już ciężka praca i nauka, która jednak w dalszej perspektywie przynosi wiele korzyści!

W przypadku problemów, pytań lub chęci uzyskania pomocy zapraszam do zakładki Skontaktuj się na blogu i pozostawieniu tam wiadomości. Odpiszę na pewno!

Oczywiście zapraszam na bloga, a szczególnie do zakładki Książki!

https://mateuszrus.pl/ksiazki/

Zachęcam też do odwiedzenia mojego profilu na **Social Media**:

- **Facebook**

  Mateusz Rus - Programowanie z pasją
  https://www.facebook.com/mateusz.rus.blog
- **Facebook grupa**

  Programowanie z pasją
  https://www.facebook.com/groups/612557692819603
- **YouTube**

  Mateusz Rus - Programowanie z pasją
  https://www.youtube.com/channel/UCGsUdxhm4m3a7CpANQTqIUw
- **Twitter**

  @MateuszDawidRus
  https://twitter.com/MateuszDawidRus
- **LinkedIn**

  Mateusz Rus
  https://www.linkedin.com/in/mateusz-rus-90a26394/
- **Instagram**

  @mateusz.rus
  https://www.instagram.com/mateusz.rus/

Mateusz Rus

Pozdrawiam, Mateusz Rus.