20 Przykłady implementacji wybranych algorytmów

Podstawy algorytmiki, w tym zasady opisu algorytmów w postaci schematu blokowego i pseudokodu, zostały przedstawione w rozdziale 1. podręcznika. W niniejszym rozdziale zaś zaprezentowano przykłady implementacji wybranych, popularnych algorytmów. W szczególności dotyczy to:

- wyznaczania największego wspólnego dzielnika dwóch liczb całkowitych na podstawie algorytmu Euklidesa,
- algorytmów sortowania tablic za pomocą metody bąbelkowej i przez proste wstawianie,
- algorytmu wyszukiwania binarnego zadanego elementu (liczby) w określonym zbiorze liczb.

Ponadto przedstawiono wykorzystanie wbudowanych — predefiniowanych — funkcji standardowych języka C++ zawartych w bibliotece *algorithm*, które pozwalają efektywnie zrealizować wymienione powyżej zadania.

20.1. Wyznaczenie największego wspólnego dzielnika

Największy wspólny dzielnik (ang. *greatest common divisor*) dwóch liczb całkowitych, w skrócie NWD, to największa liczba naturalna, która dzieli obie te liczby bez reszty. Przy tym wspomniane liczby mają dowolny znak.

UWAGA

W przypadku ogólnym NWD można wyznaczać dla dowolnej ilości liczb całkowitych.

Wyznaczenie NWD dwóch zadanych liczb całkowitych jest realizowane najczęściej z zastosowaniem **algorytmu Euklidesa** (ang. *Euclidean algorithm*).

Wspomniany powyżej algorytm jest oparty na założeniu, że wykonanie operacji odejmowania mniejszej liczby od większej — skutkujące zmniejszeniem wartości większej z liczb — nie powoduje zmiany wartości NWD. Tym samym odjęcie liczby mniejszej od większej większą liczbę razy (co można zrealizować w pętli) również nie spowoduje zmiany NWD. Algorytm kończy działanie, jeśli spełniony jest warunek, że bieżące (zmniejszone) wartości obu liczb są identyczne.

Algorytm wyznaczania NWD można wykorzystać w praktyce np. do skracania ułamków.

Przykład 20.1

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

// Definicja funkcji globalnej obliczNWD(), pozwalającej wyznaczyć NWD:
int obliczNWD(int l1, int l2) {
    while (l1 != l2) { // Iteracje są wykonywane, dopóki liczby l1 i l2 są różne.
        // Ustalenie nowej wartości (zmniejszenie) większej z liczb przez odjęcie od niej liczby mniejszej:
        if (l1 > l2) // Jeśli wartość l1 jest większa od wartości l2.
            l1 = l1 - l2;
        else if (l2 > l1) // Jeśli wartość l2 jest większa od wartości l1.
            l2 = l2 - l1;
    }
    /* UWAGA
     * Po wykonanych iteracjach wartość l1 jest równa wartości l2.
     * Pętla (i funkcja) kończy działanie, jeśli wartości obu liczb są takie same.
     */
    return l1;   // jeśli liczby l1 i l2 są równe, funkcja zwraca wartość l1
}

int main() {
    int liczba1, liczba2; // liczby, dla których wyznaczany jest NWD
    int nwd; // NWD
    // Pobranie wartości dwóch liczb całkowitych z klawiatury:
    cout << "Dane wejściowe " << endl;
    cout << "Podaj wartość pierwszej liczby: ";
    cin >> liczba1;
```

```
    cout << "Podaj wartość drugiej liczby: ";
    cin >> liczba2;
    // Wyznaczenie NWD:
    nwd = obliczNWD(liczba1, liczba2);
    // Wyświetlenie wyniku — wyznaczonej wartości NWD — na ekranie monitora:
    cout << "Wynik" << endl;
    cout << "NWD = " << nwd << endl;

    return 0;
}
```

Przedstawiony program pozwala wyznaczyć największy wspólny dzielnik (NWD) dwóch liczb całkowitych. Przy czym rozpatrywane są wyłącznie liczby dodatnie — wartość *0* oraz liczby ujemne nie są tutaj uwzględniane. Obliczenie NWD jest realizowane na podstawie algorytmu Euklidesa z użyciem funkcji globalnej `obliczNWD()`.

Algorytm Euklidesa użyty w zaprezentowanym programie w definicji funkcji `obliczNWD()` można zaimplementować również w inny sposób — z zastosowaniem operatora *modulo* (`%`). Definicja funkcji `obliczNWD()` mogłaby mieć wtedy postać:

```
int obliczNWD(int l1, int l2) {
    int reszta;
    while ((l1 % l2) > 0) {
        reszta = l1 % l2;
        l1 = l2;
        l2 = reszta;
    }
    return l2;
}
```

Do obliczenia NWD można wykorzystać również wbudowaną funkcję standardową `__gcd()`, dostępną w standardzie C++17 lub wyższym. Wyrażenie zawierające wywołanie tej funkcji mogłoby mieć w omawianym programie postać: `nwd = __gcd(liczba1, liczba2);`. Użycie funkcji `__gcd()` wymaga dołączenia do programu zasobów biblioteki *algorithm*: `#include <algorithm>`.

Ćwiczenie 20.1

Zmodyfikuj program z przykładu 20.1 — uzupełnij/zmień implementację funkcji `obliczNWD()` w taki sposób, aby przy wyznaczaniu NWD uwzględniane były liczby całkowite o dowolnym znaku oraz liczba *0*.

Przykład 20.2

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
// Definicja funkcji globalnej pozwalającej wyznaczyć NWD:
int obliczNWD(int l1, int l2) {
    // Wyznaczenie wartości bezwzględnych obu liczb reprezentowanych przez parametry formalne funkcji:
    l1 = abs(l1); l2 = abs(l2);
    /* UWAGA
     * Wyznaczenie wartości bezwzględnych liczb l1 i l2 ma na celu uwzględnienie w obliczeniach liczb całkowitych
     * o dowolnym znaku — co wynika z definicji NWD. Wartość NWD z definicji jest nieujemna.
     */
    if (l1 == 0) // Jeśli wartość liczby l1 wynosi 0.
        return l2; // NWD jest równe wartości bezwzględnej liczby l2.

    if (l2 == 0) // Jeśli wartość liczby l2 wynosi 0.
        return l1; // NWD jest równe wartości bezwzględnej liczby l1.

    if (l1 == l2) // Jeśli wartości bezwzględne obu liczb są równe.
        return l1; // NWD jest równe wartości bezwzględnej liczby l1, która jest równa wartości bezwzględnej l2.

    if (l1 > l2) // Jeśli wartość bezwzględna l1 jest większa od wartości bezwzględnej l2.
        // Wywołanie rekurencyjne funkcji obliczNWD():
        return obliczNWD(l1 - l2, l2);
        /* UWAGA
         * Pierwszym argumentem wywołania funkcji obliczNWD() jest zmniejszona wartość l1, drugim — l2.
         */
        if (l2 > l1) // Jeśli wartość bezwzględna l2 jest większa od wartości bezwzględnej l1.
            // Wywołanie rekurencyjne funkcji obliczNWD():
            return obliczNWD(l1, l2 - l1);
            /* UWAGA
             * Pierwszym argumentem wywołania funkcji jest zmniejszona wartość l2, drugim — l1.
             */
}
int main() {
    int liczba1, liczba2, nwd;
    cout << "Dane wejściowe " << endl;
    cout << "Podaj wartość pierwszej liczby: ";
    cin >> liczba1;
    cout << "Podaj wartość drugiej liczby: ";
    cin >> liczba2;
    // Wyznaczenie NWD:
    nwd = obliczNWD(liczba1, liczba2);
    cout << "Wynik" << endl;
    cout << "NWD = " << nwd << endl;

    return 0;
}
```

Funkcjonalność przedstawionego programu została rozszerzona względem funkcjonalności programu zawartego w przykładzie 20.1. Wspomniane rozszerzenie polega na uwzględnieniu w wyznaczeniu NWD liczb całkowitych o dowolnym znaku wraz z liczbą *0*, a nie jedynie liczb dodatnich — jak w przykładzie 20.1.

Implementacja funkcji `obliczNWD()` zdefiniowanej na listingu 20.2 różni się znacząco od implementacji funkcji o tej samej nazwie zawartej w przykładzie 20.1. Mianowicie funkcja `obliczNWD()` zdefiniowana tutaj jest funkcją rekurencyjną. Funkcja ta wywołuje w swoim ciele „samą siebie". Mamy zatem do czynienia z rekurencją bezpośrednią (ang. *direct recursion*).

Algorytm Euklidesa w funkcji rekurencyjnej `obliczNWD()` zdefiniowanej w przedstawionym programie można zaimplementować również przy użyciu operatora *modulo* (`%`). Definicja rozpatrywanej funkcji mogłaby mieć wtedy postać:

```
int obliczNWD(int l1, int l2) {
    l1 = abs(l1); l2 = abs(l2);
    if (l2 == 0)
        return l1;
    return obliczNWD(l2, l1 % l2);
}
```

Ćwiczenie 20.2

Zmodyfikuj program zawarty w przykładzie 20.2 — zamiast funkcji globalnej `obliczNWD()` wykorzystaj zdefiniowaną samodzielnie klasę `NWD`. Załóż, że klasa `NWD` zawiera dwie zmienne członkowskie: `liczba1` i `liczba2`, reprezentujące liczby całkowite, dla których wyznaczany jest NWD, oraz metodę `obliczNWD()` o funkcjonalności analogicznej do funkcjonalności funkcji `obliczNWD()` z przykładu 20.2.

20.2. Sortowanie tablic

W ogólności **sortowanie** (ang. *sorting*) może dotyczyć różnych struktur (rodzajów) danych, np. tablic, łańcuchów znaków, plików. Polega ono na uporządkowaniu — ustaleniu kolejności — danych w określonej strukturze według zadanego klucza (reguły).

Przykładami struktur danych, których elementy składowe można poddać sortowaniu, są tablica 1-wymiarowa złożona z liczb rzeczywistych czy też tablica zawierająca łańcuchy znaków.

Istnieje wiele algorytmów sortowania tablic 1-wymiarowych: **sortowanie bąbelkowe** (ang. *bubble sort*), **sortowanie przez wstawianie** (ang. *insertion sort*), **sortowanie przez wybór** (ang. *selection sort*) itd.

W przykładach przedstawionych w tym punkcie zaprezentowano przykładowe implementacje algorytmów sortowania bąbelkowego (przykład 20.3) i sortowania przez wstawianie (przykład 20.4) wykonywane na tablicach 1-wymiarowych, w których elementami składowymi są liczby.

Przykład 20.3

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

// Definicja funkcji pozwalającej posortować elementy tablicy liczbowej metodą bąbelkową:
void sortowanieBabelkowe(float t[], int n) {
    int i, j;
    float temp;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (t[j] < t[i]) {
                temp = t[i];
                t[i] = t[j];
                t[j] = temp;
            }
        }
    }
}
/* UWAGA
 * Funkcja sortowanieBabelkowe() pozwala posortować elementy 1-wymiarowej tablicy liczbowej
 * w porządku rosnącym — od elementu najmniejszego do największego.
 */
// Definicja funkcji pozwalającej wprowadzić wartości elementów tablicy z klawiatury:
void tablicaWejscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> t[i];
    }
}
// Definicja funkcji pozwalającej wyświetlić wartości elementów tablicy na ekranie:
void tablicaWyjscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}

int main() {
const int n = 5; // rozmiar tablicy
float tablica[n]; // 1-wymiarowa tablica liczbowa o rozmiarze n

cout << "Wprowadź wartości elementów tablicy:" << endl;
tablicaWejscie(tablica, n);

cout << "Zawartość tablicy wejściowej: " << endl;
tablicaWyjscie(tablica, n);
```

```
    // Posortowanie elementów tablicy tablica:
    sortowanieBabelkowe(tablica, n);

    cout << "Zawartość tablicy posortowanej: " << endl;
    tablicaWyjscie(tablica, n);

    return 0;
}
```

Przedstawiony program pozwala posortować elementy składowe 1-wymiarowej tablicy `tablica`, którymi są liczby rzeczywiste należące do typu `float`. Rozmiar tablicy `tablica` jest określony za pomocą stałej `n` o wartości ustalonej w programie na `5`.

Sortowanie jest realizowane metodą bąbelkową z zastosowaniem funkcji `sortowanieBabelkowe()`. Funkcje `tablicaWejscie()` i `tablicaWyjscie()` mają rolę pomocniczą. Są elementami składowymi interfejsu programu. Wszystkie z wymienionych funkcji są funkcjami globalnymi.

Ćwiczenie 20.3

Zmodyfikuj program z przykładu 20.3 — zamiast funkcji `sortowanieBabelkowe()`, `tablicaWejscie()` i `tablicaWyjscie()` wykorzystaj zdefiniowaną samodzielnie klasę `Tablica`. Wspomniana klasa powinna zawierać zmienne członkowskie reprezentujące 1-wymiarową tablicę liczbową i jej rozmiar oraz metody członkowskie odpowiadające wymienionym powyżej funkcjom globalnym.

Przykład 20.4

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

// Definicja funkcji pozwalającej posortować elementy tablicy liczbowej przez wstawianie:
void sortowaniePrzezWstawianie(float t[], int n) {
    int i, j;
    float temp;
    for (i = 1; i < n; i++) {
        temp = t[i];
        j = i - 1;
        while ((j >= 0) && (t[j] > temp)) {
            t[j+1] = t[j];
            j = j - 1;
        }
        t[j+1] = temp;
    }
}
```

```
/* UWAGA
 * Funkcja sortowaniePrzezWstawianie() pozwala posortować elementy 1-wymiarowej tablicy liczbowej
 * w porządku rosnącym — od elementu najmniejszego do największego.
 */
void tablicaWejscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = ";
        cin >> t[i];
    }
}
void tablicaWyjscie(float t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}

int main() {
    const int n = 5; // rozmiar tablicy
    float tablica[n]; // 1-wymiarowa tablica liczbowa o rozmiarze n
    cout << "Wprowadź wartości elementów tablicy:" << endl;
    tablicaWejscie(tablica, n);
    cout << "Zawartość tablicy wejściowej: " << endl;
    tablicaWyjscie(tablica, n);
    // Posortowanie elementów tablicy tablica w porządku rosnącym:
    sortowaniePrzezWstawianie(tablica, n);
    cout << "Zawartość tablicy posortowanej: " << endl;
    tablicaWyjscie(tablica, n);

    return 0;
}
```

Funkcjonalność zaprezentowanego programu jest analogiczna do funkcjonalności programu z przykładu 20.3. Wartości elementów składowych tablicy `tablica` są wprowadzane z klawiatury, wyświetlane kontrolnie na ekranie, sortowane, a na końcu ponownie wyświetlane na ekranie.

Zastosowano tutaj **sortowanie przez proste wstawianie** (ang. *simple insertion sort*). Jest ono realizowane z użyciem funkcji globalnej `sortowaniePrzezWstawianie()`.

Ćwiczenie 20.4

Zmodyfikuj program z przykładu 20.4 — w funkcji `sortowaniePrzezWstawianie()` zamiast sortowania przez wstawianie proste wykorzystaj algorytm **sortowania przez wstawianie binarne** (ang. *binary insertion sort algorithm*), który jest nazywany również sortowaniem przez wstawianie z wyszukiwaniem binarnym (ang. *binary search*).

UWAGA

Zamiast samodzielnie definiować funkcje mające za zadanie posortowanie elementów tablicy za pomocą wybranego algorytmu, można wykorzystać wbudowaną funkcję standardową `sort()`. Funkcja ta jest dostępna w standardzie C++17 lub wyższym. W ogólności funkcję `sort()` można stosować zarówno do tablic, jak i do wektorów należących do typu `vector`.

20.3. Wyszukiwanie binarne

W programowaniu często się zdarza, że trzeba wyszukać zadaną liczbę w zbiorze liczb przechowywanych w tablicy. Jeżeli wspomniana tablica jest posortowana, zadanie to można zrealizować na podstawie **algorytmu wyszukiwania binarnego** (ang. *binary search algorithm*).

Przykład 20.5

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

// Definicja funkcji pozwalającej wyszukać zadaną liczbę w tablicy:
int wyszukiwanieBinarne(int t[], int n, int liczba) {
                                    // t — posortowana tablica liczbowa
                                    // n — liczba elementów (rozmiar) tablicy
                                    // liczba — zadana liczba do wyszukania w tablicy
    int poczatek = 0;
    int koniec = n - 1;
    int srodek = (poczatek + koniec) / 2;
    int wynik;
    /* UWAGA
     * Jeśli w tablicy znaleziono element składowy o wartości liczba, to w zmiennej wynik zapisywany
     * jest indeks tego elementu.
     * Jeśli tablica nie zawiera elementu składowego o wartości liczba, to do zmiennej wynik podstawiana
     * jest wartość -1.
     */
    while (poczatek <= koniec) {
        if (t[srodek] < liczba) {
            poczatek = srodek + 1;
        }
        else {
            if (t[srodek] == liczba) {
                wynik = srodek;
                break;
            }
```

```
            else {
                koniec = srodek - 1;
            }
        }
        srodek = (poczatek + koniec) / 2;

    }
    if (poczatek > koniec) {
        wynik = -1;
    }
    return wynik;
}
// Funkcja pomocnicza:
void tablicaWyjscie(int t[], int n) {
    for (int i = 0; i < n; i++) {
        cout << "element[" << i << "] = " << t[i] << endl;
    }
}
int main() {
    // Deklaracja i inicjalizacja tablicy:
    int tablica[] {10, 21, 32, 43, 54, 65, 76, 87, 98, 109};
    /* UWAGA
     * 1-wymiarowa tablica liczbowa powinna być posortowana.
     */
    int n = sizeof(tablica) / sizeof(int); // liczba elementów składowych tablicy

    cout << "Tablica wejściowa" << endl;
    // Prezentacja tablicy wejściowej na ekranie monitora:
    tablicaWyjscie(tablica, n);
    cout << endl;

    int liczba = 65; // szukana liczba
    cout << "Szukana liczba: " << liczba << endl;
    cout << endl;

    // Wyszukanie zadanej liczby w tablicy:
    int indeks = wyszukiwanieBinarne(tablica, n, liczba);

    // Prezentacja wyników na ekranie monitora:
    cout << "Wyniki" << endl;
    if (indeks != -1) {
        cout << "Pozycja (indeks) szukanej liczby w tablicy: "
             << indeks << endl;
```

```
        cout << "Wartość elementu tablicy o indeksie " << indeks
             << ": " << tablica[indeks] << endl;
    }
    else {
        cout << "Zadanej liczby nie znaleziono!" << endl;
    }

    return 0;
}
```

W zaprezentowanym programie przedstawiono przykładową implementację algorytmu wyszukiwania binarnego liczby całkowitej `liczba` w tablicy liczbowej `tablica`. Rozmiar (liczba elementów składowych) tej tablicy jest przechowywany w zmiennej `n`.

Wyszukiwanie binarne jest realizowane z wykorzystaniem funkcji `wyszukiwanieBinarne()`. Parametr `t` omawianej funkcji reprezentuje tablicę liczbową o rozmiarze `n`, która jest posortowana. Poszukiwana liczba odpowiada parametrowi `liczba`. Omawianą funkcję można zdefiniować również jako funkcję rekurencyjną.

Funkcja `wyszukiwanieBinarne()` zdefiniowana tutaj pozwala znaleźć zadaną liczbę w tablicy oraz indeks elementu, w którym liczba ta jest przechowywana. Implementacja tej funkcji nie uwzględnia jednak przypadku, w którym w tablicy zapisanych jest więcej elementów (liczb) o wartościach odpowiadających szukanej liczbie — co oczywiście można zmienić.

Wyszukiwanie binarne można również zrealizować z użyciem wbudowanej funkcji standardowej `binary_search()` z biblioteki *algorithm*. Funkcja ta jest dostępna w standardzie C++20 i nowszych. Wyrażenie zawierające wywołanie tej funkcji w omawianym programie mogłoby mieć postać: `bool wynik = binary_search(tablica, tablica + n, 65);`, gdzie:

- zmienna logiczna `wynik` reprezentuje wynik wyszukiwania: `true` — zadaną liczbę znaleziono, `false` — przeciwnie,
- `tablica` (a faktycznie `tablica + 0`) to początek zakresu, w którym jest prowadzone wyszukiwanie, a `tablica + n` to koniec tego zakresu,
- `65` odpowiada poszukiwanej liczbie.

Funkcję biblioteczną `binary_search()` można również stosować do zmiennych (wektorów) należących do typu `vector`.

Ćwiczenie 20.5

Zmodyfikuj kod źródłowy programu z przykładu 20.5 — zaimplementuj funkcję `wyszukiwanieBinarne()` jako funkcję rekurencyjną.

20.4. Pytania i zadania kontrolne

20.4.1. Pytania

1. Omów i opisz za pomocą pseudokodu wyznaczanie największego wspólnego dzielnika (NWD) dwóch liczb całkowitych na podstawie algorytmu Euklidesa bez stosowania rekurencji.
2. Zapisz algorytm Euklidesa do wyznaczania NWD dwóch liczb całkowitych za pomocą schematu blokowego. Wykorzystaj rekurencję.
3. Przedstaw zalety i wady sortowania tablic za pomocą metody bąbelkowej.
4. Przedstaw algorytm sortowania tablic metodą bąbelkową za pomocą pseudokodu i schematu blokowego.
5. Na czym polega **efektywność algorytmu** (ang. *algorithm efficiency*)?
6. Omów pojęcie **złożoności obliczeniowej** (ang. *computational complexity*) algorytmu, w tym jego **złożoność czasową** (ang. *time complexity*) i **złożoność pamięciową** (ang. *memory complexity*).

20.4.2. Zadania

1. Napisz program pozwalający wyznaczyć najmniejszą wspólną wielokrotność (NWW) dwóch zadanych liczb całkowitych. Wykonaj program w dwóch wariantach:
   - bez stosowania rekurencji,
   - z wykorzystaniem rekurencji.
2. Napisz program pozwalający posortować elementy składowe tablicy liczbowej w porządku malejącym. Wykorzystaj metodę **sortowania szybkiego** (ang. *quick sort*, *quicksort*). Omów założenia i ideę algorytmu **„dziel i zwyciężaj”** (ang. *divide and conquer*) oraz jego wykorzystanie w praktyce.
3. Napisz program pozwalający posortować w porządku rosnącym elementy tablicy 1-wymiarowej złożonej z łańcuchów znaków. Załóż, że wspomniane łańcuchy to nazwiska pracowników działu księgowości w zakładzie pracy.
4. Napisz program umożliwiający wyszukanie zadanego nazwiska w tablicy, w której elementy składowe to łańcuchy znaków reprezentujące nazwiska uczniów w grupie klasowej. Załóż, że dane nazwisko może występować w tablicy wielokrotnie. Określ liczbę wystąpień szukanego nazwiska w tablicy.
5. Napisz program pozwalający posortować elementy składowe wektora liczbowego należącego do typu `vector` za pomocą funkcji wbudowanej `sort()` z biblioteki *algorithm*.
6. Napisz program pozwalający wyszukać zadaną liczbę w zestawie liczb zapisanych w wektorze należącym do typu `vector`. Wykorzystaj wbudowaną funkcję biblioteczną `binary_search()`.