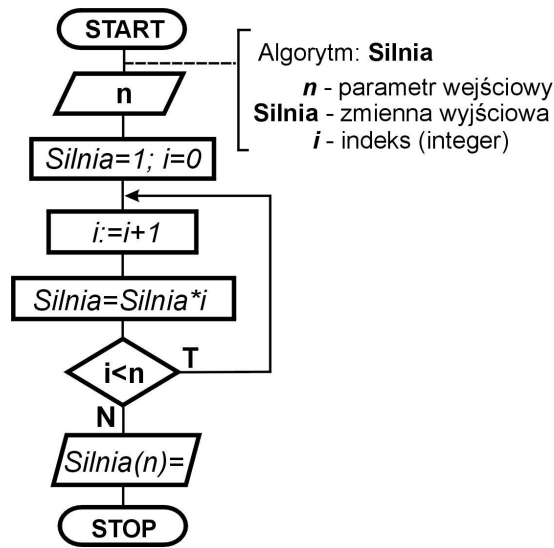


C9

Projektowanie algorytmów rekurencyjnych

Ćwiczenie 1.

Przeanalizować działanie poniższego algorytmu dla parametru wejściowego $n = 4$ (rysunek 9.1):



n	i	$i < n$	$Silnia(n)$
0	1	N	1
1	1	N	1
2	1	T	1
	2	N	2
3	1	T	1
	2	T	2
	3	N	6
4			

Rys. 9.1.. Algorytm iteracyjny do obliczania *silni* z n .

Zastanówmy się teraz, w jaki sposób algorytm iteracyjny zastąpić algorytmem rekurencyjnym do obliczania *silni* z n według definicji:

$$\begin{cases} 0! = 1 \\ n! = n*(n-1)!, \text{ gdzie } n \geq 1 \end{cases}$$

Przeanalizujemy mechanizm obliczeń silni z n . Z definicji wynika, że aby obliczyć np. $4!$, należy wykonać kolejne kroki:

START

$$4! = 4 * (4-1)! = 4 * 3!$$

$$3! = 3 * (3-1)! = 3 * 2!$$

$$2! = 2 * (2-1)! = 2 * 1!$$

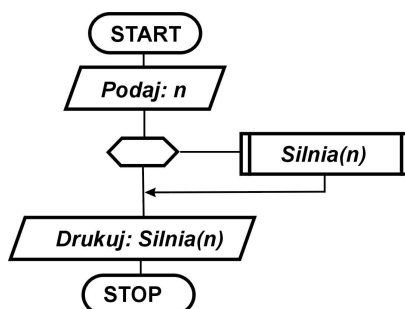
$$1! = 1 * (1-1)! = 1 * 0!$$

$$0! = 1 \text{ (def.)}$$

STOP

Zbudujmy algorytm rekurencyjny obliczający wartość funkcji *Silnia(n)*, którą na podstawie definicji możemy zapisać jako: $Silnia(0) = 1$, $Silnia(1) = 1$, $Silnia(n) = n * Silnia(n-1)$.

Algorytm główny (rysunek 9.2) ograniczy się jedynie do wczytania *n*, wywołania funkcji rekurencyjnej *Silnia(n)* i wyprowadzenia wyniku *Silnia(n)*.



Rys. 9.2. Algorytm główny obliczania silni.

Nowym elementem jest tutaj wywołanie funkcji *Silnia(n)* zwracającej wynik obliczeń do miejsca wywołania.

Algorytm: Funkcja *Silnia(n)*

Krok 1. Jeśli $n = 0$ lub $n = 1$, to wydrukuj wynik funkcji *Silnia(n)* = 1 i zakończ działanie algorytmu.

Krok 2. W przeciwnym razie znajdź wynik funkcji $Silnia(n) = n * Silnia(n-1)$

Krok 3. Drukuj wynik *Silnia(n)* i zakończ działanie algorytmu.

Przykładowa implementacja tego algorytmu w języku Java może być następująca:

```
public class SilniaRek {
    public static void main(String[] args) {
        long x;
        for (int n = 0; n <= 6; n++) {
            x = silnia(n);
            System.out.println(n + "! = " + x);
        }
    }
    static long silnia (int n) {
        long wynik;
        if (n > 1) wynik = silnia(n-1) * n;
        else     wynik = 1;
        return wynik;
    }
}
```

Często zachodzi potrzeba wielokrotnego obliczania silni, co z oczywistych względów jest kosztowne czasowo. Weźmy dla przykładu znany nam symbol Newtona, wykorzystywany do obliczania liczby k -elementowych kombinacji bez powtórzeń ze zbioru n -elementowego, wyrażony wzorem:

$$C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Wyznaczenie algorytmu obliczania Symbolu Newtona wiąże się z koniecznością uwzględnienia trzykrotnego obliczania silni, dla: $n!$, $k!$ i $(n-k)!$

Poniżej przedstawiono przykładowe rozwiązania tego problemu w języku Java. Istotą tych propozycji jest to, że wszystkie potrzebne wartości funkcji *silnia* będą obliczane tylko raz i zapisane, a później tylko odczytywane. Taki sposób rozwiązania będziemy nazywać **tablicowaniem funkcji**.

```
public class Silnia {
    private final int MAX_SILNIA = 100;
    private int[] silniaTab = new int[MAX_SILNIA];

    public static void main(String[] args) {
        Silnia silnia = new Silnia();
        for(int i=0; i<6; i++){
            try {
                System.out.println("n="+i+" : "+
                    silnia.silnia(i));
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
        for(int i=0; i<6; i++){
            try {
                System.out.println("n="+i+" : "+
                    silnia.silniaTablicowana(i));
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public int silnia(int n) throws Exception{
        if (n<0){
            throw new Exception("Liczba nie może być ujemna");
        }
        if (n==0 || n==1){
```

```

        return 1;
    }else{
        return n*this.silnia(n-1);
    }
}
public int silniaTablicowana(int n) throws Exception{
    if (n<0){
        throw new Exception("Liczba nie może być ujemna");
    }
    if (n >= MAX_SILNIA){
        throw new Exception("Zbyt duża liczba");
    }
    if (silniaTab[n] == 0) {
        return silniaWypelnianie(n);
    }else{
        return silniaTab[n];
    }
}
private int silniaWypelnianie(int n) throws Exception {
    if (n<0){
        throw new Exception("Liczba nie może być ujemna");
    }
    if (n >= MAX_SILNIA){
        throw new Exception("Zbyt duża liczba");
    }
    int wynik = 0;
    if (n==0 || n==1){
        wynik = 1;
    }else{
        wynik = n*this.silnia(n-1);
    }
    silniaTab[n] = wynik;
    return wynik;
}
}

```

ZADANIA DO SAMODZIELNEGO WYKONANIA

ZADANIE 1.

Skonstruować schematy blokowe, opracować algorytmy rekurencyjne w postaci listy kroków, schematów blokowych oraz kodu do obliczania:

- Liczby kombinacji k -elementowych ze zbioru n -elementowego.
- Liczby wariacji k -elementowych ze zbioru n -elementowego bez powtórzeń i z powtórzeniami.
- n -tego wyrazu ciągu arytmetycznego i geometrycznego, jeśli dane są a_1 i odpowiednio r i q .

$$\begin{cases} a_1 = a \\ a_{n+1} = a_n + r, \quad a, r \in R \end{cases}$$

$$\begin{cases} a_1 = a \\ a_{n+1} = q * a_n, \quad a, q \in R \end{cases}$$

Wskazówka.

Funkcja powinna posiadać trzy parametry formalne: n , a_1 oraz r (bądź q).

- Zapisać algorytmy z zadania 1 w języku C^{++} i przetestować ich działanie dla różnych n .
- Narysować i opisać schematy realizacji poszczególnych poziomów rekurencji (analogicznie do rysunku 4.9¹).

ZADANIE 2.

Skonstruować schematy blokowe, opracować algorytmy rekurencyjne wraz z implementacją w języku C^{++} procedury, która pobiera dowolną liczbę naturalną n i odlicza od tej liczby aż do zera. Procedura powinna spowodować wypisanie kolejno liczb: $n, n-1, n-2, \dots, 0$.

¹ Ochodek B., Ochodek M.: *Algorytmy i struktury danych*, Wyd. PWSZ w Pile, Piła 2003, s. 102.