Temat: Przykłady problemów rozwiązywanych rekurencyjnie

1. Funkcję, która oblicza N-tą potęgę liczby X, gdzie N jest nieujemną liczbą całkowitą.

Zauważmy że

$$X^N = X * X^{N-1}$$

Taka postać to definicja rekursyjna (postać rekurencyjna) problemu – przedstawienie go jako zależności między rozwiązaniem problemu dla danego kroku przetwarzania danych i wynikiem z kroku poprzedniego (ta zależność niekoniecznie dana jest jako prosta zależność arytmetyczna, jak w powyższym przykładzie). Aby móc obliczyć X^{N-1}, musimy znaleźć X^{N-2}, X^{N-3} ... – czyli rozwiązanie problemu „mniejszego o jeden krok przetwarzania". W rekursji problem jest redukowany do prostszej (dla danego problemu) postaci - aż do momentu, gdy możemy znaleźć rozwiązanie bez odwoływania się do postaci rekurencyjnej (w naszym przypadku [$X^N = X * X^{N-1}$]). Taka postać problemu to wspomniany powyżej warunek zatrzymania – w naszym przypadku dane, dla których nie jest wymagane odwołanie się do definicji rekurencyjnej problemu to N = 0 (zastanów się, czy warunkiem zatrzymania mogłoby być również N=1, czy dla X=0 opłaci się dodawać odrębny warunek?).

Rozważany problem można zapisać przez następującą zależność rekurencyjną:

$$X^N = \begin{cases} X * X^{N-1} & dla \quad N > 0 \\ 1 \quad dla & N = 0 \end{cases}$$

```
long double potega(long double X,int N)
{
      if (N==0)
      return 1;
      else
            return X*potega(X,N-1);
}
```

Analizując powyższą funkcję możemy zauważyć, że wywołując funkcję `potega` powodujemy, że będzie ona wywoływała samą siebie, do momentu, gdy parametr N wywołania nie będzie równy 0. Funkcja nie może zwrócić wartości, do momentu, gdy wszystkie wywołania wewnętrzne nie zwrócą wartości**. Uproszczony schemat zawartości stosu przedstawia tabela:**

wywołanie	stos
potega(2,0)	Rezerwacja pamięci na zmienne lokalne - brak
	Rezerwacja pamięci na argumenty wywołania
	Rezerwacja pamięci na wartość zwracaną
potega(2,1)	Rezerwacja pamięci na zmienne lokalne - brak
	Rezerwacja pamięci na argumenty wywołania
	Rezerwacja pamięci na wartość zwracaną
potega(2,2)	Rezerwacja pamięci na zmienne lokalne - brak
	Rezerwacja pamięci na argumenty wywołania
	Rezerwacja pamięci na wartość zwracaną
potega(2,3)	Rezerwacja pamięci na zmienne lokalne - brak
	Rezerwacja pamięci na argumenty wywołania
	Rezerwacja pamięci na wartość zwracaną

Omówmy krótko sposób działania stosu: funkcja w momencie wywołania rezerwuje na stosie ilość pamięci odpowiadającą typowi zwracanej wartości. Ten obszar– zawiera śmieci do momentu zwrócenia wartości przez funkcję. Następnie funkcja rezerwuje obszar pamięci na parametry wywołania, zmienne lokalne i wywołuje się ponownie – o ile parametry wywołania na to pozwolą. W momencie, gdy parametry wywołania odpowiadają warunkowi zatrzymania – funkcja zwraca wartość, obszar pamięci zarezerwowany na

zmienne lokalne i parametry wywołania zostaje zwolniony. Zwracana wartość jest przypisywana jako wartość funkcji w miejscu wywołania funkcji (w naszym przypadku `return X*potega(X,N-1);`), obszar pamięci zarezerwowany na zwróconą właśnie wartość zostaje zwolniony.

Teraz przedostatnie wywołanie funkcji może obliczyć wartość formuły i zwrócić wartość- itd., aż do momentu, gdy można obliczyć wartość pierwszego wywołania funkcji (dla przykładu z tabeli powyżej – to wywołanie funkcji `potega(2, 3).`

Wnioski:

Ponieważ obszar pamięci zarezerwowany na stos jest ograniczony – deklarowanie zbyt wielu zmiennych lokalnych prowadzi dla złożonych problemów do szybkiego jego przepełnienia – należy unikać deklarowania zmiennych lokalnych w funkcjach rekurencyjnych, o ile to tylko możliwe. Ponadto rezerwacja miejsca na stosie na zmienne lokalne i parametry wywołania jest kosztowna, jeśli chodzi o czas działania programu – w przypadkach, gdy funkcja rekurencyjna wywołuje samą siebie kilkukrotnie (np. funkcja rekurencyjna obliczająca wartość dwumianu Newtona dla (n,k) lub funkcja rekurencyjna obliczająca wartość n-tego wyrazu ciągu Fibonacciego) – czas ten bardzo szybko wzrasta wraz z wartością parametrów wywołania.

2. **Napisz funkcję, która wypisze na ekranie n znaków *.**
   Problem posiada banalne rozwiązanie iteracyjne, rozwiązanie rekurencyjne (w tym przypadku nieco „na siłę") uzyskujemy w następujący sposób:
   - **funkcja nie zwraca wartości, natomiast wypisuje znak ***
   - **dla n=0 nie wypisuje znaku i kończy działanie**
   - **dla n>0 wypisuje znak i rozwiązuje problem wypisania n-1 znaków.**

   Program będzie miał postać:

```
void stars(int n)
{
 if (n==0)
         return;
 else  {
   cout<<'*';
         return stars(n-1);
 }
}
```

3. **Napisz funkcję, która wyświetli liczby całkowite od n do 1**
   Problem analizujemy analogicznie jak poprzedni, podobnie jak poprzedni – nie wymaga stosowania rekurencji:
   - **funkcja nie zwraca wartości, natomiast wypisuje liczbę całkowitą**
   - **dla n=1 wypisuje 1 i kończy działanie**
   - **dla n>1 wypisuje n i rozwiązuje problem wypisania liczb od n-1 do 1**

   Program będzie miał postać:

```
void reverse(int n)
{
 if (n==1)
   cout<<n;
else  {
         cout<<n;
         return reverse(n-1);
 }
}
```

4. **Napisz funkcję, która wyświetli liczby całkowite od 1 do n Problem wymaga niewielkich korekt kodu z poprzedniego punktu:**
   - **funkcja nie zwraca wartości, natomiast wypisuje liczbę całkowitą**
   - **dla n=1 wypisuje 1 i kończy działanie**
   - **dla n>1 rozwiązuje problem wypisania liczb od 1 do n-1 i wypisuje n**

   Program będzie miał postać:

```
void integers(int n)
{  if (n==1)
   cout<<n;  else  {
   integers(n-1);
   cout<<n;    return;
 }
}
```

5. **Napisz funkcję rekurencyjną, która zwróci wartość typu string złożoną ze znaków wprowadzonych na klawiaturze. Znaki są wprowadzane do momentu wystąpienia symbolu *. Zwracana wartość typu string składa się ze znaków wprowadzonych przez użytkownika (łącznie z symbolem *), uporządkowanych w kolejności odwrotnej do kolejności wprowadzania.**
   Problem wymaga wykorzystania przeciążonego operatora + klasy string, umożliwiającego scalanie łańcuchów znaków.
   - **niech zmienna** `s` **typu** `string` **służy do przechowywania wprowadzanych znaków**
   - **jeśli s== '*' funkcja zwraca s**
   - **jeśli s!='*' funkcja dołącza do znaków wprowadzonych po** `s` **znak przechowywany w zmiennej** `s`

```
string odkonca()
{  string s;
 cout<<"Wprowadź znak i wciśnij ENTER";
 cin>>s;
 if(s=="*")  //uwaga - operator == nie może porównywać typu char z
               //typem string - dlatego nie '*' lecz "*"
   return s;  else
         return odkonca()+s;
}
```

Jeśli wprowadzimy każdorazowo kilka znaków, powyższy kod zwróci wprowadzone ciągi znaków w kolejności odwrotnej do kolejności ich wprowadzania.

6. **Napisz funkcję rekurencyjną, która wyświetli „choinkę" o wysokości n:**

```
    *
   ***
  *****
 *******
```

Aby wyświetlić choinkę o rozmiarze N, należy wyświetlić przesuniętą o jeden znak w prawo **choinkę o wymiarze N-1, a następnie wyświetlić linię 2N-1 gwiazdek, jeśli N>1, lub niczego nie wyświetlić, jeśli N==0.**

```
void choinka(int n, int przesuniecie=0)//jeśli nie zostanie podany 2-gi
//parametr wywołania, zostanie użyta wartość domyślna drugiego
//parametru, tu ustawiona na 0
{  if (n==0)
```

```
   return;  else
 {
    choinka(n-1,++przesuniecie); //wyświetlamy choinkę przesuniętą o 1
 }
 cout<<setw(przesuniecie)<<setfill(' ')<<"*"
        <<setw(2*n-1)<<setfill('*')<<'\n';//to jest powtórzenie
                     //materiału o formatowaniu UWAGA! '\n' to też ZNAK!
}
```

Spróbuj zastosować funkcje analogiczne do tych z punktu 1 do rozwiązania tego problemu (zastąp nimi manipulatory). Spróbuj napisać program wyświetlający „odwróconą choinkę"

7. **Napisz funkcję rekurencyjną, która obliczy, wykorzystując algorytm Hornera wartość wielomianu postaci**

   $a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n$

   dla podanej jako argument wartości x i danego wektora współczynników t[i]= a_i
   Ponieważ wielomian stopnia n możemy zapisać w następującej postaci,

   $a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n = (...((a_0 x + a_1)*x + a_2)*x + ... + a_{n-1})x + a_n$,

   korzystając ze schematu Hornera, możemy zauważyć, że

   - **funkcja zwraca t[0] gdy i==0**
   - **funkcja zwraca sumę: t[i]+x*(wynik funkcji dla i-1) gdy i!=0 W rezultacie rozwiązanie problemu ma postać:**

```
long double horner(long double t[], long double x,int i)
{  if (i==0)
   return t[0];
 else
   return x*horner(t,x,i-1)+t[i]; }
```

---

8. **Napisz rekurencyjną funkcję, znajdującą największy wspólny dzielnik 2 liczb całkowitych dodatnich. Algorytm Euklidesa opiera się na spostrzeżeniu, że NWD 2 liczb całkowitych m i n, m>n, jest równy NWD liczb y i x mod y (reszta z dzielenia x przez y). Liczba x dzieli zarówno liczbę m jak i n ⇔ x dzieli n i resztę z dzielenia m przez n, ponieważ m jest równe sumie m mod n i wielokrotności n.**

```
int NWD(int m, int n)
{  if (n==0)
   return m;  else
         return NWD(n,m%n);
}
```
