10

Рекурсия

10.1 Рекурсивни функции в математиката

Ако в дефиницията на някаква функция се използва самата функция, дефиницията на функцията се нарича **рекурсивна**.

Примери:

а) Ако n е произволно естествено число, следната дефиниция на функцията факториел

$$n! = \begin{cases} 1, & n = 0 \\ n.(n-1)!, & n > 0 \end{cases}$$

е рекурсивна. Условието при n = 0 не съдържа обръщение към функцията факториел и се нарича **гранично**.

б) Функцията за намиране на най-голям общ делител на две естествени числа a и b може да се дефинира по следния рекурсивен начин:

$$gcd(a, b) = \begin{cases} a, & a = b \\ gcd(a-b, b), & a > b \\ gcd(a, b-a), & a < b. \end{cases}$$

Тук граничното условие е условието при a = b.

в) Ако x е реално, а n – цяло число, функцията за степенуване може да се дефинира рекурсивно по следния начин:

$$x^n = \begin{cases} x.x^{n-1}, & n>0 \\ 1, & n=0 \\ \frac{1}{x^{-n}}, & n<0. \end{cases}$$

В този случай граничното условие е условието при n = 0.

г) Редицата от числата на Фибоначи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

се дефинира рекурсивно по следния начин:

$$fib(n) = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ fib(n-1)+fib(n-2), & n>1. \end{cases}$$

В този случай имаме две гранични условия – при n = 0 и при n = 1.

Рекурсивната дефиниция на функция може да се използва за намиране стойността на функцията за даден допустим аргумент.

Примери:

а) Като се използва рекурсивната дефиниция на функцията факториел може да се намери факториелът на 4. Процесът за намирането му преминава през разширение, при което операцията умножение се отлага, до достигане на граничното условие 0! = 1. Следва свиване, при което се изпълняват отложените операции.

4! =
4.3! =
4.3.2! =
4.3.2.1! =
4.3.2.1.0! =
4.3.2.1.1 =
4.3.2.1 =
4.3.2 =
4.6 =
24

б) Като се използва рекурсивната дефиниция на функцията gcd, може да се намери gcd(35, 14).

gcd(35, 14) =
gcd(21, 14) =
gcd(7, 14) =

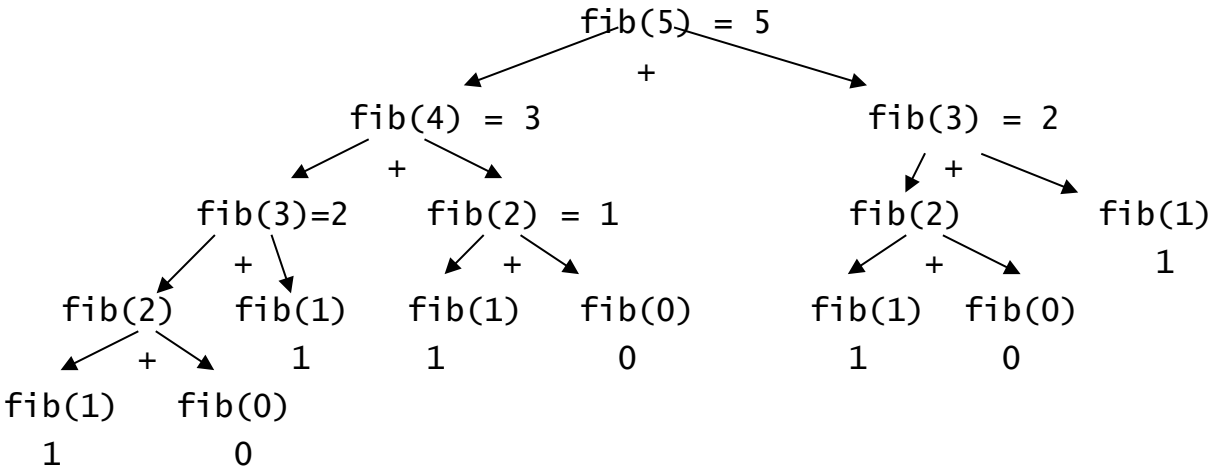
gcd(7, 7) = 7.

В този случай няма разширяване, нито пък свиване. Двата аргумента на gcd играят ролята на натрупващи параметри. Те се променят докато се достигне граничното условие.

в) Като се използва рекурсивната дефиниция на функцията за степенуване може да се намери стойността на 2^{-4}.

$$2^{-4} =$$
$$\frac{1}{2^4} =$$
$$\frac{1}{2.2^3} =$$
$$\frac{1}{2.2.2^2} =$$
$$\frac{1}{2.2.2.2.2^0} =$$
$$\frac{1}{2.2.2.2.1} =$$
$$\frac{1}{2.2.2.2} =$$
$$\frac{1}{2.2.4} =$$
$$\frac{1}{2.8} = \frac{1}{16}.$$

г) Чрез рекурсивната дефиниция на функцията, генерираща редицата на Фибоначи, може да се намери fib(5).



В този случай намирането на fib(5) генерира дървовиден процес. Операцията + се отлага. Забелязваме много повтарящи се изчисления, например fib(0) се пресмята 3 пъти, fib(1) – 5 пъти, fib(2) - 3 пъти, fib(3) – 2 пъти, което илюстрира неефективността на този начин за пресмятане.

10.2 Рекурсивни функции в C++

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или е декларирана до момента на извикването й. Освен това, в C++ е вграден т. нар. механизъм на рекурсия – разрешено е функция да вика в тялото си самата себе си.

Функция, която се обръща пряко или косвено към себе си, се нарича рекурсивна. Програма, съдържаща рекурсивна функция е **рекурсивна**.

Чрез примери ще илюстрираме описанието, обръщението и изпълнението на рекурсивна функция.

Задача 85. Да се напише рекурсивна програма за намиране на m! (m е дадено естествено число).

Програма Zad85.cpp решава задачата.

```
// Program Zad85.cpp
#include <iostream.h>
int fact(int);
int main()
{cout << "m= ";
 int m;
 cin >> m;
 if (!cin || m < 0)
 {cout << “Error! \n”;
  return 1;
 }
 cout << m << "!= " << fact(m) << '\n';
 return 0;
}
int fact(int n)
```

```
{if (n == 0) return 1;
 else return n * fact(n-1);
}
```

В тази програма е описана рекурсивната функция fact, която приложена към естествено число връща факториела на това число. Стойността на функцията се определя посредством обръщение към самата функция в оператора return n * fact(n-1);. Запазената дума else в оператора

```
if (n == 0) return 1;
else return n * fact(n-1);
```

е излишна заради оператора return 1; пред нея. Използвана е с цел увеличаване на читаемостта на функцията.

Изпълнение на програмата

Изпълнението започва с изпълнение на главната функция. Фрагментът

```
cout << "m= ";
int m;
cin >> m;
```

въвежда стойност на променливата m. Нека за стойност на m е въведено 4. В резултат в стековата рамка на main, отделените 4В за променливата m се инициализират с 4. След това се изпълнява операторът

```
cout << m << "!= " << fact(m) << '\n';
```

За целта трябва да се пресметне стойността на функцията fact(m) за m равно на 4, след което получената стойност да се изведе. Обръщението към функцията fact е илюстрирано по-долу:

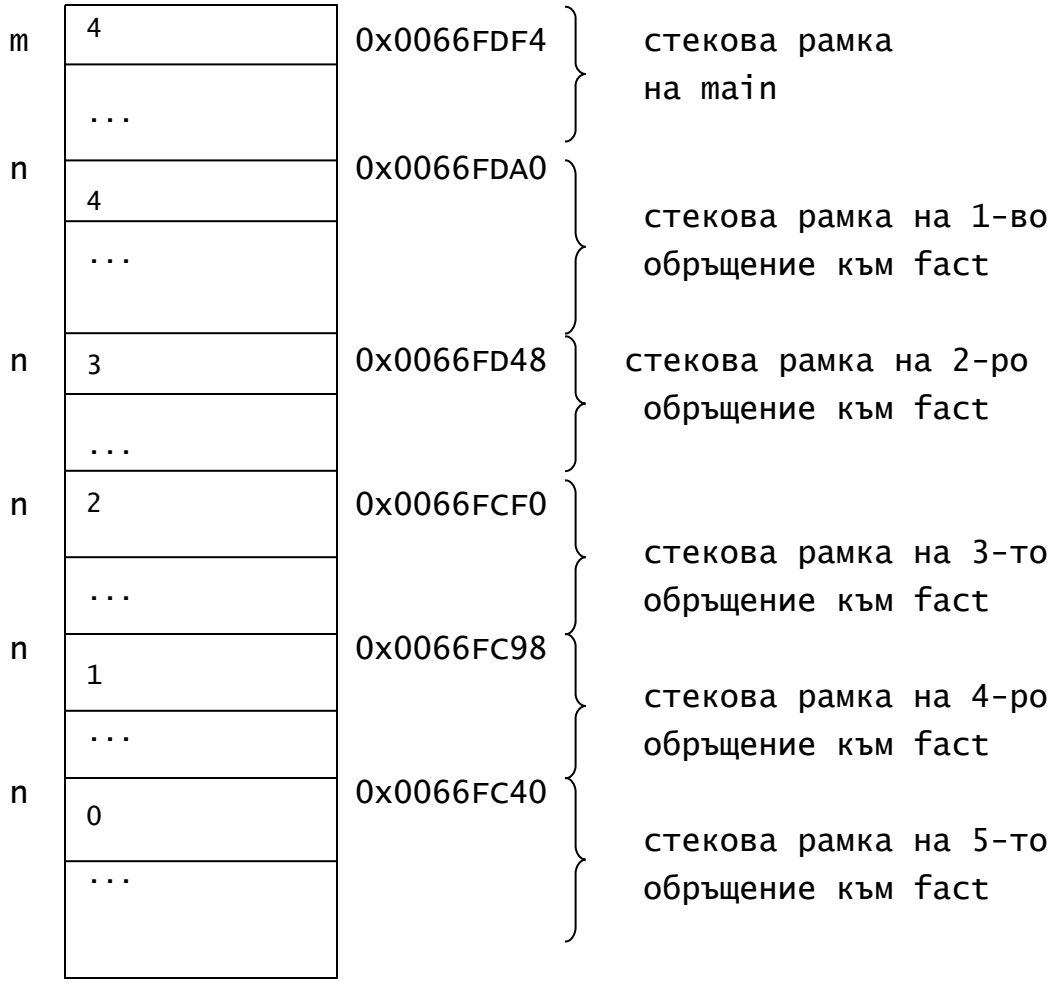
Генерира се стекова рамка за това обръщение към функцията fact. В нея се отделят 4В ОП за фактическия параметър n, в която памет се откопирва стойността на фактическия параметър m и започва изпълнението на тялото на функцията. Тъй като n е различно от 0, изпълнява се операторът

```
return n * fact(n-1);
```

при което трябва да се намери fact(n-1), т.е. fact(3). По такъв начин преди завършването на първото обръщение към fact се прави второ обръщение към тази функция. За целта се генерира нова стекова рамка на функцията fact, в която за формалния параметър n се откопирва стойност 3. Тялото на функцията fact започва да се изпълнява за втори

път (Временно спира изпълнението на тялото на функцията, предизвикано от първото обръщение към нея).

По аналогичен начин възникват още обръщения към функцията fact. При последното от тях, стойността на формалния параметър n е равна на 0. Получава се:



При петото обръщение към fact стойността на n е равна на 0. В резултат, изпълнението на това обръщение завършва и за стойност на fact се получава 1. След това последователно завършват изпълненията на останалите обръщения към тялото на функцията. При всяко изпълнение на тялото на функцията се определя съответната стойност на функцията fact. След завършването на всяко изпълнение на функцията fact, отделената за fact стекова рамка се освобождава. В крайна сметка в главната програма се връща 24 - стойността на 4!, която се извежда върху екрана.

В този случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение.

Препоръка: Ако за решаването на някаква задача може да се използва итеративен алгоритъм, реализирайте го. Не се препоръчва винаги използването на рекурсия, тъй като това води до загуба на памет и време.

В тази глава няма да спазим препоръката, тъй като целта ни е придобиване на умения за рекурсивно дефиниране на функции.

Задача 86. Да се напише програма, която въвежда от клавиатурата записана без грешка формула от вида

```
<формула> ::=  <цифра> |
               (<формула><знак><формула>)
<знак> ::= + | - | *
<цифра> ::= 0 | 1 | 2 | ...| 9.
```

Програмата да намира и извежда стойността на въведената формула (Например 8 --> 8; ((2-6)*4) --> -16).

Програма Zad86.cpp решава задачата. В нея е дефинирана рекурсивната функция formula, реализираща рекурсивната дефиниция на <формула>. Ще отбележим, че случаите на оператора switch не завършват с оператора break;. Това е така заради използването на оператора return в края на всеки случай.

```
// Program Zad86.cpp
#include <iostream.h>
int formula();
int main()
{cout << formula() << "\n";
 return 0;
}
int formula()
{char c, op;
 int x, y;
 cin >> c;                  // c е ‘(‘ или цифра
 // <формула> ::= <цифра>
```

```
 if (c >= '0' && c <= '9') return (int)c - (int)'0';
 else
 // <формула> ::= (<формула><знак><формула>)
 {x = formula();
  cin >> op;
  y = formula();
  cin >> c;                // прескачане на ‘)’
  switch (op)
  {case '+': return x + y;
   case '-': return x - y;
   case '*': return x * y;
   default: cout << "Error! \n"; return 111;
  }
 }
}
```

Забелязваме простотата и компактността на записа на рекурсивните функции. Това проличава особено при работа с динамичните структури: свързан списък, стек, опашка, дърво и граф.

Основен недостатък е намаляването на бързодействието поради загуба на време за копиране на параметрите им в стека. Освен това се изразходва повече памет, особено при дълбока степен на вложеност на рекурсията.

Задачи върху рекурсия

Задача 87. Да се напише рекурсивна програма, която намира най-големия общ делител на две естествени числа.

Програма Zad87.cpp решава задачата.

```
// Program Zad87.cpp
#include <iostream.h>
int gcd(int, int);
int main()
{cout << "a, b= ";
 int a, b;
 cin >> a >> b;
```

```
 if (!cin || a < 1 || b < 1)
 {cout << "Error! \n";
  return 1;
 }
 cout << "gcd{" << a << ", " << b << "}= " <<  gcd(a, b) << "\n";
 return 0;
}
int gcd(int a, int b)
{if (a == b) return a;
 if (a > b) return gcd(a-b, b);
 return gcd(a, b-a);
}
```

Задача 88. Като се използва рекурсивната дефиниция на функцията за степенуване да се напише програма, която по дадени a реално и k – цяло число, намира стойността на a^k.

Програма Zad88.cpp решава задачата.

```
// Program Zad88.cpp
#include <iostream.h>
double pow(double, int);
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << "k= ";
 int k;
 cin >> k;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
```

```
 cout << "pow{" << a << ", " << k << "}= " << pow(a, k) << "\n";
 return 0;
}
double pow(double x, int n)
{if (n == 0) return 1;
 if (n > 0) return x * pow(x, n-1);
 return 1.0/pow(x, -n);
}
```

Задача 89. Квадратна мрежа с k реда и k стълба (1 ≤ k ≤ 20) има два вида квадратчета - бели и черни. В черно квадратче може да се влезе, но не може да се излезе. От бяло квадратче може да се премине във всяко от осемте му съседни, като се прекоси общата им страна или връх. Да се напише програма, която ако са дадени произволна мрежа с бели и черни квадратчета и две произволни квадратчета - начално и крайно, определя дали от началното квадратче може да се премине в крайното.

Анализ на задачата:

а) Ако началното квадратче не е в мрежата, приемаме, че не може да се премине от началното до крайното квадратче.

б) Ако началното квадратче съвпада с крайното, приемаме, че може да се премине от началното до крайното квадратче.

в) Ако началното и крайното квадратчета са различни и началното квадратче е черно, не може да се премине от него до крайното квадратче.

г) Във всички останали случаи, от началното квадратче може да се премине до крайното тогава и само тогава, когато от някое от съседните му квадратчета (в хоризонтално, във вертикално или диагонално направление), може да се премине до крайното квадратче.

Програма Zad89.cpp решава задачата.

Представяне на данните:

Мрежата ще представим чрез квадратна матрица от цели числа mr[k x k] (1 ≤ k ≤ 20) и ще реализираме чрез съответен двумерен масив. При това mr[i][j] е равно на 0, ако квадратче (i, j) е бяло и е равно на

1, ако е квадратче (i, j) е черно (0 ≤ i ≤ k-1, 0 ≤ j ≤ k-1). Нека началното квадратче е от ред x и стълб y, а крайното квадратче е от ред m и стълб n.

В програмата Zad89.cpp е дефинирана рекурсивната функция way. Тя връща стойност true, ако от квадратче (x, y) може да се премине до квадратче (m, n) и false - в противен случай. За да се избегне зацикляне (връщане в началното квадратче от всяко съседно), се налага преди рекурсивните обръщения към way, да се промени стойността на mr[x][y] като квадратчето (x, y) се направи черно.

```
// Program Zad89.cpp
#include <iostream.h>
int mr[20][20];
int k, m, n;
bool way(int, int);
void writemr(int, int [][20]);
int main()
{cout << "k from [1, 20] = ";
 do
 {cin >> k;
 }while (k < 1 || k > 20);
 int x, y;
 do
 {cout << "x, y = ";
  cin >> x >> y;
 } while (x < 0 || x > k-1 || y < 0 || y > k-1);
 do
 {cout << "m, n = ";
  cin >> m >> n;
 } while (m < 0 || m > k-1 || n < 0 || n > k-1);
 for (int i = 0; i <= k-1; i++)
 for (int j = 0; j <= k-1; j++)
   cin >> mr[i][j];
 cout << "\n";
 writemr(k, mr);
 if (way(x, y)) cout << "yes \n";
 else cout << "no \n";
```

```
 return 0;
}
bool way(int x, int y)
{if (x < 0 || x > k-1 || y < 0 || y > k-1) return false;
 if (x == m && y == n) return true;
 if (mr[x][y] == 1) return false;
 mr[x][y] = 1;
 bool b = way(x+1,  y) || way(x-1, y)  ||
          way(x,  y+1) || way(x, y-1)  ||
          way(x-1,y-1) || way(x-1, y+1)||
          way(x+1,y-1) || way(x+1, y+1);
 mr[x][y] = 0;
 return b;
}
void writemr(int k, int mr[][20])
{for (int i = 0; i <= k-1; i++)
  {for (int j = 0; j <= k-1; j++)
     cout << mr[i][j];
   cout << '\n';
  }
}
```

Задача 90. Да се напише рекурсивен вариант на функцията от по-висок ред accumulate.

Функцията accumulate, дефинирана по-долу, решава задачата. Тя използва дефинициите на типовете:

```
typedef double (*type1) (double, double);
typedef double (*type2) (double);

double accumulate(type1 op, double null_val, double a, double b,
                  type2 f, type2 next)
{if (a > b + 1e-14) return null_val;
 return op(f(a),
               accumulate(op, null_val, next(a), b, f, next));
}
```

Задача 91. да се напише рекурсивна функция double min(int n, double* x), която намира минималния елемент на редицата x_0, x_1, ..., x_{n-1}.

първо решение:

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0];
 b =  min(n-1, x);
 if (b < x[n-1]) return b;
 return x[n-1];
 }
```

второ решение:

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0];
 b =  min(n-1, x+1);
 if (b < x[0]) return b;
 return x[0];
 }
```

Задача 92. да се напише рекурсивна функция, която проверява дали елементът x принадлежи на редицата a_0, a_1, ..., a_{n-1}.

първо решение:

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
```

второ решение:

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[n-1] || member(x, n-1, a);
}
```

Задача 93. Да се напише рекурсивна функция, която проверява дали редицата x_0, x_1, ..., x_{n-1} е монотонно растяща.

първо решение:

```
bool monincr(int n, double* x)
{if (n == 1) return true;
 return x[n-2] <= x[n-1] && monincr(n-1, x);
}
```

второ решение:

```
bool monincr(int n, double* x)
{if (n == 1) return true;
 return x[0] <= x[1] && monincr(n-1, x+1);
}
```

Задача 94. Да се напише рекурсивна функция, която проверява дали редицата a_0, a_1, ..., a_{n-1} се състои от различни елементи.

първо решение:

```
bool differ(int n, int* a)
{if (n == 1) return true;
 return !member(a[0], n-1, a+1) && differ(n-1, a+1);
}
```

второ решение:

```
bool differ(int n, int* a)
{if (n == 1) return true;
 return !member(a[n-1], n-1, a) && differ(n-1, a);
}
```

Задача 95. Дадена е квадратна мрежа от клетки, всяка от които е празна или запълнена. Запълнените клетки, които са свързани, т.е. имат съседни в хоризонтално, вертикално или диагонално направление, образуват област. Да се напише програма, която намира броя на областите и размера (в брой клетки) на всяка област.

Програма Zad95.cpp решава задачата.

Представяне на данните:

Мрежата ще представим чрез квадратна матрица от цели числа mr[n x n] (1 ≤ n ≤ 20) и ще реализираме чрез съответен двумерен масив. При това mr[i][j] е 1 ако квадратче (i, j) е запълнено и 0 – в противен случай (0 ≤ i ≤ n-1, 0 ≤ j ≤ n-1).

Анализ на задачата:

Ще дефинираме функция broy, която преброява клетките в областта, съдържаща дадена клетка (x, y). Функцията има два параметъра x и y - координатите на точката и реализира следния алгоритъм:

а) Ако клетката с координати (x, y) е извън мрежата, приемаме, че броят на клетките в областта е равен на 0.

б) В противен случай, ако клетката с координати (x, y) е празна, приемаме, че броят е равен на 0.

в) В останалите случаи, броят на клетките в областта е равен на сумата от 1 и броя на клетките на всяка област, на която принадлежат осемте съседни клетки на клетката (x, y).

От подточка в) следва, че функцията broy е рекурсивна. За да избегнем зацикляне и многократно преброяване, трябва преди рекурсивното обръщение на направим клетката (x, y) празна.

```
// Program Zad95.cpp;
#include <iostream.h>
int mr[20][20];
int n;
int broy(int x, int y)
{if (x < 0 || x > n-1 || y < 0 || y > n-1) return 0;
 if (mr[x][y] == 0) return 0;
 mr[x][y] = 0;
 return 1 + broy(x-1, y+1) + broy(x, y+1)
          + broy(x+1, y+1) + broy(x+1,y)
          + broy(x+1, y-1) + broy(x, y-1)
          + broy(x-1, y-1) + broy(x-1, y);
}
int main()
{cout << "mreja: \n”;
 do
 {cout << "n= ";
  cin >> n;
```

```
 } while (n < 1 && n > 20);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
    cin >> mr[i][j];
int br = 0;
for (i = 0; i <= n-1; i++)
 for (j = 0; j <= n-1; j++)
  if (mr[i][j] == 1)
  {br++;
   cout << "size of the " <<  br << " th location is equal to "
        << broy(i, j) << endl;
  }
return 0;
}
```

Задача 96. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която установява дали съществува път между два произволно зададени града (Приемаме, че ако от град i до град j има път, то има път и от град j до град i).

Анализ на задачата:

Ще дефинираме рекурсивната булева функция way, която зависи от два параметъра i и j, показващи номерата на градовете, между които се проверява дали съществува път. Функцията реализира следния алгоритъм:

а) Ако $i = j$, съществува път от град i до град j.

б) Ако $i \ne j$ и има пряк път от град i до град j, има път между двата града.

в) В останалите случаи има път от град i до град j, тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път.

Програма Zad96.cpp решава задачата.

```
// Program Zad96.cpp;
#include <iostream.h>
```

```
int arr[10][10];
int n;
bool way(int i, int j)
{if (i == j) return true;
 if (arr[i][j] == 1) return true;
 bool b = false;
 int k = -1;
 do
 {k++;
  if (arr[i][k] == 1)
  {arr[i][k] = 0; arr[k][i] = 0;
   b = way(k, j);
   arr[i][k] = 1; arr[k][i] = 1;
   }
 } while (!b && k <= n-2);
 return b;
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i
         << " and " << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
```

```
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (way(i, j)) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

Задача 97. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която намира пътя между два произволно зададени града в случай, че път между тях съществува.

Анализ на задачата:

Процедурата foundway намира един път от град i до град j в случай, че път съществува, т.е. ако way(i, j) има стойност true. Пътят се записва в едномерния масив int x[100], а дължината му - в променливата s.

Програма Zad97.cpp решава задачата. В нея е пропусната дефиницията на функцията way. При обръщение към функцията foundway параметърът-псевдоним s трябва да се свърже с параметър, инициализиран с –1.

```
// Program Zad97.cpp
#include <iostream.h>
int arr[10][10];
int n;
bool way(int i, int j)
...
void foundway(int i, int j, int& s, int x[])
{s++;
 x[s] = i;
 if (i != j)
  if (arr[i][j] == 1)
  {s++;
   x[s] = j;
  }
  else
```

```
  {bool b = false;
   int k = -1;
   do
   {k++;
    if (arr[i][k] == 1) b = way(k, j);
   } while (!b);
   arr[i][k] = 0; arr[k][i] = 0;
   foundway(k, j, s, x);
   arr[i][k] = 1; arr[k][i] = 1;
  }
}
int main()
{int p = -1;
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
 int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
   cin >> arr[i][j];
   arr[j][i] = arr[i][j];
   }
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (way(i, j))
 {foundway(i, j, p, a);
  p++;
```

```
     for (int l = 0; l <= p-1; l++) cout << a[l] << "  " ;
     cout << endl;
    }
    else cout << "no \n";
    return 0;
   }
```

Задача 98. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише булева функция fixway(int i, int j, int p), която установява дали съществува път от град i до град j с дължина p ($p \ge 1$).

Анализ на задачата:

Булевата функция fixway реализира следния алгоритъм:

а) Ако p = 1, има път от град i до град j с дължина p ако има пряк път между двата града.

б) Ако p > 1, има път от град i до град j с дължина p тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път с дължина p-1.

Следващият програмен фрагмент дефинира само функцията fixway.

```
bool fixway(int i, int j, int p)
{if (p == 1) return arr[i][j] == 1; else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  } while (!b && k <= n-2);
  return b;
 }
}
```

Тази функция извършва проверка за съществуване на цикличен път от един до друг връх с указана дължина.

Пример: Ако имаме само два града, означени с 0 и 1 и те са свързни с пряк път, fixway(0, 1, 3) ще отговори с true.

Ако търсим съществуването само на ациклични пътища, ще използваме модификацията на горната функция, дадена по-долу.

```
bool fixway(int i, int j, int p)
{bool b;
 int k;
 if (p == 1) return arr[i][j] == 1;
 b = false;
 k = -1;
 do
 {k++;
  if (arr[i][k] == 1)
  {arr[i][k] = 0; arr[k][i] = 0;
   b = fixway(k, j, p-1);
   arr[i][k] = 1; arr[k][i] = 1;
  }
 } while (!b && k <= n-2);
 return b;
}
```

Задача 99. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише процедура foundfixway(int i, int j, int p, int s&, int* x), която намира пътя от град i до град j с дължина p в случай, че такъв съществува.

Програма Zad99.cpp решава задачата. За краткост, дефиницията на функцията fixway е пропусната. Пътят е запомнен в масива x, а параметърът s съдържа текущата му дължина.

```
// Program Zad99.cpp
#include <iostream.h>
int arr[10][10];
int n;
bool fixway(int i, int j, int p)
...
void foundfixway(int i, int j, int p, int& s, int* x)
```

```
{s++;
 x[s] = i;
 if (p == 1)
 {s++;
  x[s] = j;
 }
 else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  } while (!b);
  foundfixway(k, j, p-1, s, x);
 }
}
int main()
{int p = -1;
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 10);
int i, j;
 for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 int l;
 do
```

```
 {cout << "start and final towns, and len between them: ";
  cin >> i >> j >> l;
 } while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (fixway (i, j, l))
 {foundfixway(i, j, l, p, a);
  for (int m = 0; m <= l; m++) cout << a[m] << "  " ;
  cout << endl;
 }
 else cout << "no \n";
 return 0;
}
```

Задачи

Задача 1. Да се напише рекурсивна функция, която намира стойността на функцията на Акерман Ack(m, n), дефинирана за $m \geq 0$ и $n \geq 0$ по следния начин:

Ack(0, n) = n+1

Ack(m, 0) = Ack(m-1, 1), m > 0

Ack(m, n) = Ack(m-1, Ack(m, n-1)), m > 0, n > 0.

Задача 2. Да се напише рекурсивна функция, която установява, дали в записа на естественото числа n се съдържа цифрата k.

Задача 3. Да се напише рекурсивна програма, която намира стойността на полинома

$$P_n(x) = a_0x^{n-1} + a_1x^{n-2} + ... + a_{n-2}x + a_{n-1}$$

където x и a_i ($0 \leq i \leq n$) са дадени реални числа.

Задача 4. Да се напише рекурсивна функция, която пресмята корен квадратен от x, $x \geq 0$, по метода на Нютон.

Задача 5. Да се напише рекурсивна функция, която добавя елемент в сортиран масив, като запазва наредбата на елементите.

Задача 6. Да се напише рекурсивна функция, която изключва елемент от сортиран масив, като запазва наредбата на елементите.

Задача 7. Дадена е мрежа от m x n квадратчета, като за всяко квадратче е определен цвят - бял или черен. Път ще наричаме редица от

съседни във вертикално или хоризонтално направление квадратчета с един и същ цвят. Област ще наричаме множество от квадратчета с един и същ цвят между всеки две, от които има път. Дадено е квадратче. Да се определи:

а) броят на квадратчетата от областта, в която се съдържа даденото квадратче.

б) броят на областите с цвят, съвпадащ с цвета на даденото квадратче.

в) броят на областите с цвят, различен от цвета на даденото квадратче.

г) броят на квадратчетата с цвят, съвпадащ с цвета на даденото квадратче, които не са в една област с него.

Задача 8. Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове ще наричаме пълно, ако всеки два различни града, принадлежащи на множеството, са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички пълни множества, състоящи се от k на брой града.

Задача 9. Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове ще наричаме независимо, ако всеки два различни града, принадлежащи на множеството, не са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички независими множества, състоящи се от k на брой града.

Задача 10. Да се напише програма, която намира стойността на произволен израз от вида:

<израз> ::= <цяло_число>|
(<израз> * <израз>).

Задача 11. Да се напише програма, която намира стойността на произволен израз от вида:

<израз> ::= <цяло_число>|
(<израз> ^ <израз>),

където с ^ е означена операцията степенуване.

Задача 12. Да се напише програма, която въвежда от клавиатурата без грешка булев израз от вида

<булев_израз> ::= t|f|

<операция>(<операнди>)
<операция> ::= n | a | o
<операнди> ::= <операнд>|
<операнд>, <операнди>
<операнд> ::= <булев_израз>,

където t и f означават истина и лъжа съответно, **n** има само един операнд, а **a** и **o** могат да имат произволен брой операнди и означават съответно логическо отрицание, конюнкция и дизюнкция. Програмата намира и извежда стойността на булевия израз.

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
3. М. Тодорова, Програмиране на Паскал, Полипринт Враца, София, 1993.
4. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, София, 1998.