СИНТЕЗ И АНАЛИЗ НА АЛГОРИТМИ

Лекции - 30 часа

Лабораторни упражнения - 30 часа

Изпитна процедура - тест

Лектор:

Доц. д-р Елена Рачева

404 ТВ

сл. тел. 383-628

Асистенти:

ас. Антоанета Иванова

ас. Мая Тодорова

ас. Нели Калчева

ас. Павлина Линова

ас. Надежда Цветкова

Семестриален контрол	
форма	***точки***
Задачи за самостоятелна подготовка (5 х 7 т.)	до 35
Активно участие по време на лабораторните упражнения	до 5
Общо	до 40

Сесиен изпит	
форма	***точки***
Компютърен тест	до 60
Всичко	до 100

Литература

1. Робърт Седжуик, *Алгоритми на C, том 1,2*, Софтпрес, С. 2002
2. Лендерт Амерал, *Алгоритми и структури от данни в C++*, *ИК СОФТЕХ*, С., 2001
3. Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, ТроТеат Со., София
   - Уеб сайт на книгата: http://www.programirane.org
   - Facebook група: http://www.facebook.com/groups/168112146541301
4. Е.Рачева. *Лекционни материали по "Синтез и анализ на алгоритми"*, ТУ-Варна, 2005
5. Е.Рачева, Н.Николов, А.Иванова. *Ръководство за лабораторни упражнения по "Синтез и анализ на алгоритми"*, ТУ-Варна
6. Е.Рачева, Н.Николов, Н.Миндов. *Тестове по "Синтез и анализ на алгоритми"*, ТУ-Варна, 2004
7. Н. Уирт. *"Алгоритми + Структури от данни = Програми"*, Техника, 1988

8. http://www.awl.com/cseng/titles/0-201-35088-2 Материали по книгата на Р. Седжуик *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*. Addison-Wesley, Reading, MA, 1999.

9. D. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, Reading, MA, 1997.

10. D. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998.

11. R. Sedgewisk and P.Flajolet. *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.

12. A. Aho, J. Hopcroft, J. Ulman. *Data Structures and Algirithms*. Addison Wesley, 1983.

13. L. Banachowski, A. Kreczmar. *Analysis of Algorithms and Data structures*. Addison Wesley, 1992.

14. Уильям Топп, Уильям Форд. *Структуры данных в C++*. Бином, М., 2000.

АЛГОРИТМИ + СТРУКТУРИ

ОТ ДАННИ = ПРОГРАМИ

(Николаус Уирт)

ИНТУИТИВНА ДЕФИНИЦИЯ:

Алгоритъм е еднозначно трактувана процедура за решаване на задачата

СВОЙСТВА НА АЛГОРИТМИТЕ

- определеност (еднозначност)
- масовост
- дискретност
- резултативност (реализуемост)

Теза на Чърч

Формалните модели на всеки един алгоритъм съответстват на интуитив-ната представа за него

МАШИНА НА ТЮРИНГ

Автомат, обработващ безкрайна лента, разделена на участъци (клетки), по една клетка наведнъж.

Във всяка една клетка може да бъде записан (прочетен) един символ:

$$a_0, a_1, a_2, \ldots a_n$$

- външна азбука на МТ

За предвижване на лентата МТ разполага с прост механизъм, който:

- предвижва лентата с една стъпка напред;
- предвижва лентата с една стъпка назад;
- оставя лентата неподвижна.

Т. е. лентата може да се намира в едно от трите състояния:

d_{-1} d_{+1} d_0

Глава за четене / запис

Логически блок с краен брой вътрешни състояния:

$p_1, p_2, \ldots p_m$

(вътрешна азбука на МТ)

Работа на ЛБ на МТ:

1. Главата чете текущия символ (под нея).
2. В зависимост от прочетения символ (a_i) и своето вътрешно състояние (p_j):
- задейства главата, която пише в клетката под нея нов символ;

- предвижва лентата;
- променя вътрешното си състояние.

3. Ако в резултат на т. 2:
- символът под главата
- положението на лентата
- вътрешното състояние на ЛБ

остават без изменение, МТ спира.
В противен случай - т.1

Функционална таблица на МТ

	p_1	p_2	...	p_j	...	p_m
a_0						
a_1						
...						
a_i				$a_x\ d_y\ p_z$		
...						
a_n						

СЛОЖНОСТ НА АЛГОРИТМИ

- Сложност по време - T(n) (изчислителна сложност, time complexity, growth rate)
- Сложност по памет - M(n)

n - размерност на задачата

Tmin(n)

Tmax(n)

T(n)->Tcp(n)

$$T(n) = \frac{\sum_{i=1}^{k} Ti(n)}{k} \qquad T(n) = \sum_{i=1}^{k} pi * Ti(n)$$

$$O(f(n))$$

$$\lim \frac{f(n)}{T(n)} = const$$

$$n \to \infty$$

- полиномиални алгоритми
- експоненциални алгоритми

Сложност по памет

(функция от големина на входните данни)

(задачи, за които е необходима полиномиална памет)

(изискват експоненциална памет)

NP-задачи
(non-deterministic polynomial time или полиномиално-проверими задачи)

Една задача принадлежи на класа **NP** , ако е възможно с полиномиална сложност да се провери, дали даден кандидат действително е решение, без да се интересуваме, колко време ще отнеме намирането на такъв кандидат.

Например: дадено е естествено число **n.** Да се провери дали съществува делител на **n**, по-малък от дадено естествено число **q:**

$$q>1, q<n$$

Ако разполагаме с кандидат за решение **q'**, лесно можем да проверим дали съществува **p'**, такова че

$$n = p' \times q'$$

Но за да решим формулираната задача, ще трябват по-големи изчислителни ресурси.

P-задачи

(polynomial или задачи, за които съществува решение с полиномиална сложност)

Експоненциални задачи

- могат да се решат с експоненциална сложност

Нерешими задачи

Не могат да бъдат решени независимо с колко време и памет разполагаме.

Например: да се реши уравнението:

$$a^n + b^n = c^n$$

a, b, c, n - естествени числа, n>2

Съгласно скоро доказаната Голяма теорема на Ферма, такава четворка естествени числа не съществува.

Но програмистът (и компилатора) трудно може да прецени, дали решението съществува:

```
int main()
{
   unsigned a, b, c, x, n;
  for (x=3; ;)
  { for (a=1; a<=x; a++)
        for (b=1; b<=x; b++)
            for (c=1; c<=x; c++)
               for (n=3; n<=x; n++)
                  if (pow(a,n) +pow(b,n)==pow(c,n))        exit(0);
      x++;
   }
   return 0;
}
```

NP-пълни задачи (NP-complete)

P↔NP ?

Т.е. ако проверката дали дадено кандидат-решение изисква полиномиално време, то и цялото решение също става за полиномиално време?

NP-пълни задачи е клас от NP-задачи, които могат да се свеждат една към друга с полиномиална сложност, и всяка задача от NP може да се сведе до някоя NP-пълна задача.

Ако само за една NP-пълна задача бъде доказано, че принадлежи или не принадлежи към класа P, то това ще следва и за всички останали задачи от класа NP.

(Например: търсене на прости пътища с дължина K е граф)

Най-често срещани стойности на f(n) за O(f(n))

- 1
- log n $\log_2 n$ $\log_{10} n$
- n
- n*log n
- n^2
- n^3
- 2^n
- n!

Основни правила за определяне на O(f(n))

- Всеки прост оператор, който не зависи от n, има сложност **1**
- Два последователни програмни фрагмента с оценки на сложност $O(f_1(n))$ и $O(f_2(n))$ имат обща сложност **$O(max(f_1(n), f_2(n)))$**
- Ако фрагмент със сложност $O(f_1(n))$ е вложен в друг със сложност $O(f_2(n))$, то общата им сложност е **$O(f_1(n)*f_2(n))$**
- Ако има K цикли, вложени един в друг, със сложност O(n) всеки, общата им оценка е **$O(n^k)$.**

МЕТОДИ ЗА РАЗРАБОТВАНЕ НА АЛГОРИТМИ И ПРОГРАМИ

- “поредово” програмиране
- използване на специални методи и подходи с цел повишаване на ефективността на процеса на проектиране и реализация на сложни проекти

МЕТОДИ ЗА РАЗРАБОТВАНЕ НА АЛГОРИТМИ И ПРОГРАМИ

- Структурно програмиране
- Обектно-ориентирано програмиране
- Метод на частните цели
- Метод "Разделяй и владей"
- Метод "Връщане назад" (обратно проследяване или backtracking)
- Евристични алгоритми
- Рекурсия и други

Структурно програмиране

Програмиране без goto

Произволно сложна програма може да бъде реализирана при използване на следния минимален базис:
- последователност
- while (итеративна структура)
- case (избор)

Всяка неструктурирана програма може да бъде структурирана допълнително.

Цели на структурното програмиране:

1. Да се осигури дисциплина в програмирането
2. Да се подобри читаемостта на програмите
3. Да се повиши ефективността на програмите
4. Да се повиши надеждността на програмите
5. Да се намали времетраенето и стойността на разработването на програмите

Основни принципи на структурното програмиране:

1. Принцип на абстракциите
2. Принцип на формалността
3. Принцип “Разделяй и владей”
4. Принцип на йерархичното подреждане

Търсене с връщане назад (backtracking)

Започва с с целта или решението и се предприема движение в обратната посока – к началната постановка на задачата. След това пак от постановката към решението и т. н.

Задача за джипа:
Да се пресече пустиня с дължина 1000 мили с MIN разход на гориво.

Обемът на резервоара – 500 л, разходът на гориво е 1 л на 1 миля (за улесняване на сметките).

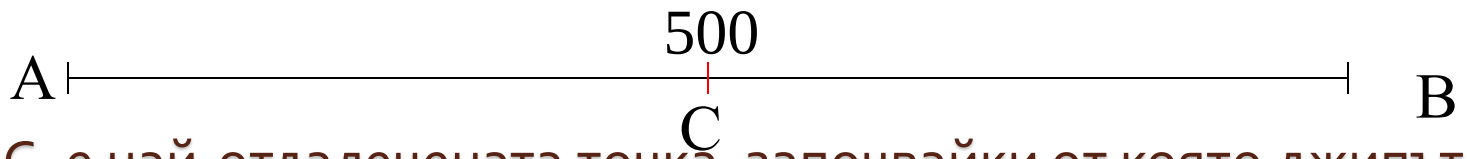
В началния пункт има неограничени количества гориво, в пустинята – няма.

Очевидно е, че в пустинята трябва да се създадат складове. Къде да ги направим и колко гориво да съхраняваме, за да пресечем пустинята с MIN разход на гориво?

Ако имаме склад с K резервоара гориво, какво разстояние можем да изминем? Къде трябва да се намира този склад, че да стигнем до края на пустинята?

Ще търсим отговор на тази задача за K=1, 2, 3, … N, докато N ще се окаже достатъчно за 1000 мили.

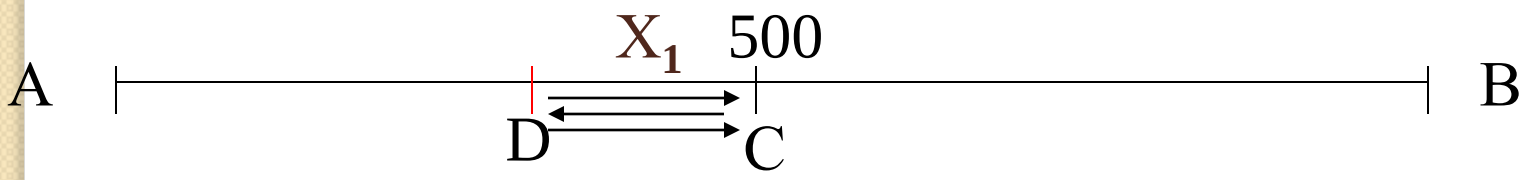
Нека K=1. Очевидно е, че могат да се изминат 500 мили:



C е най-отдалечената точка, започвайки от която джипът може да пресече пустинята, разполагайки с 500 л гориво.

Т. е. в С трябва да осигурим склад с 1 резервоар (500 л) гориво. Как?

Нека К=2, т.е. в следващия склад ще има два пълни резервоара гориво (1000 л). Къде да се разположи този склад, така че от него да се пренесат в С 500 л (К=1)?



Т.е. каква е Max стойност на X_1, че тръгвайки с 1000 л (К=2) от т. (500- X_1) могат да се занесат достатъчни количества в т. С.
Един от начините е следния:

Зареждаме колата в т. (500- X_1) , стигаме до т. С (изминавайки X_1 мили), преливаме в склада цялото гориво с изключение на X_1 литра, необходими за връщане в т. (500- X_1). Тук резервоарът вече е празен. Пълним го за втори път, изминаваме X_1 мили до т. С, зареждаме запазеното там гориво и тръгваме за т. В с пълен резервоар.

Общото разстояние, което се изминава, се състои от 3 отсечки по X_1 мили и СВ или в литри:

$$3\,X_1 \text{ литра} + 500 \text{ литра} = 1000 \text{ литра}$$

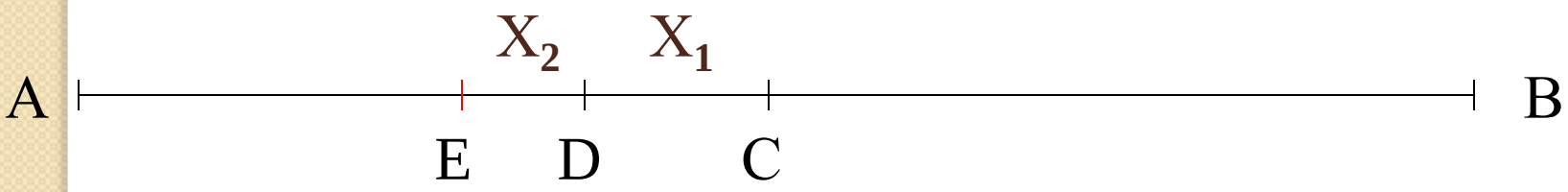
Решението на уравнението е $X_1 = 500/3$.

Т.е. два резервоара бензин (или 1000 литра) позволяват ни да изминем:

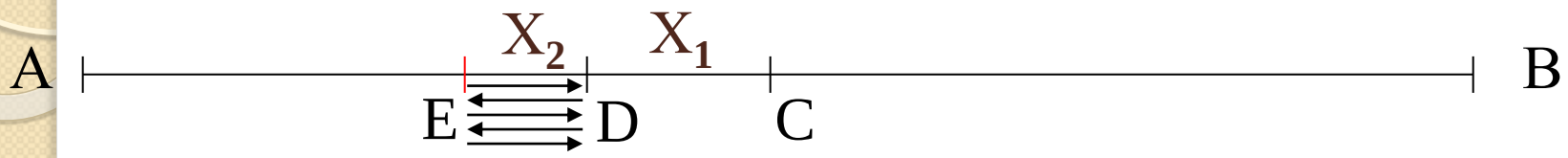
$$DIST_2 = 500 + X_1 = 500\,(1 + 1/3) \text{ мили}$$

Постановката е следната: като разполагаме с K резервоара гориво, ние се стремим да се придвижим колкото може повече назад от точката, в която се намират K-1 резервоара, т. е. търсим Max X.

K=3. От тази точка трябва да тръгнем с 1500 л гориво, за да доставим 1000 л в т. (500- X_1). Търсим Max X_2, което да ни осигури това условие.



Тръгваме с 1500 л гориво от т. (500 - X_1 - X_2) (или т. E) и доставяме 1000 л в т. (500- X_1) (или т. D).

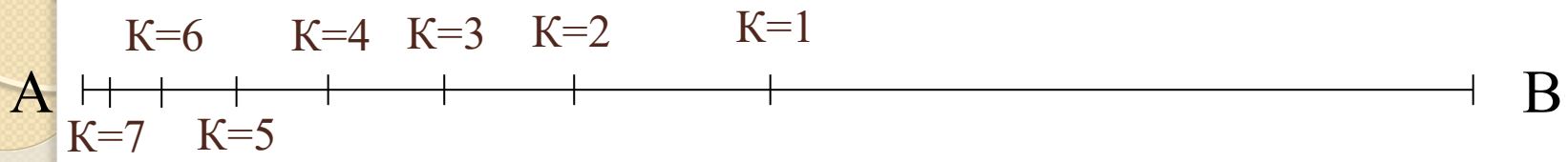


Това може да стане по следния начин: тръгваме от т. (500 - X_1 - X_2), стигаме до т. (500- X_1), преливаме горивото, като запазваме само необходимото X_2 количество. Връщаме се в т. (500 - X_1 - X_2) или т. E с празен резервоар. Ако повторим процедурата, ние ще изразходваме $4X_2$ литра гориво и ще оставим (1000 – $4X_2$) литра в т. (500- X_1) или т. D. Сега в т. E остават само 500 л. Зареждаме ги и тръгваме за т. D, като изразходваме по пътя X_2 литра:

1500 - $5X_2$ = 1000 (толкова остават в т. E) т. е. X_2 = 500/5

Или с три резервоара можем да се придвижим напред с:

$DIST_3 = 500 + X_1 + X_2 = 500\ (1 + 1/3 + 1/5)$ мили



Ако продължим този процес, ще получим следната закономерност:

$$DIST_N = 500\ (1 + 1/3 + 1/5 + \ldots + 1/(2N+1))$$

Задачата се решава с $N = 7$

Рекурсия

Рекурсията е метод в математиката и програмирането за дефиниране на функция или езикова конструкция чрез самата себе си.

Пример за рекурсивно дефиниране на функция е функцията факториел:

f(n) = n!=n*(n-1)*(n-2)*(n-3)* … *2*1, като се приеме 0! = 1

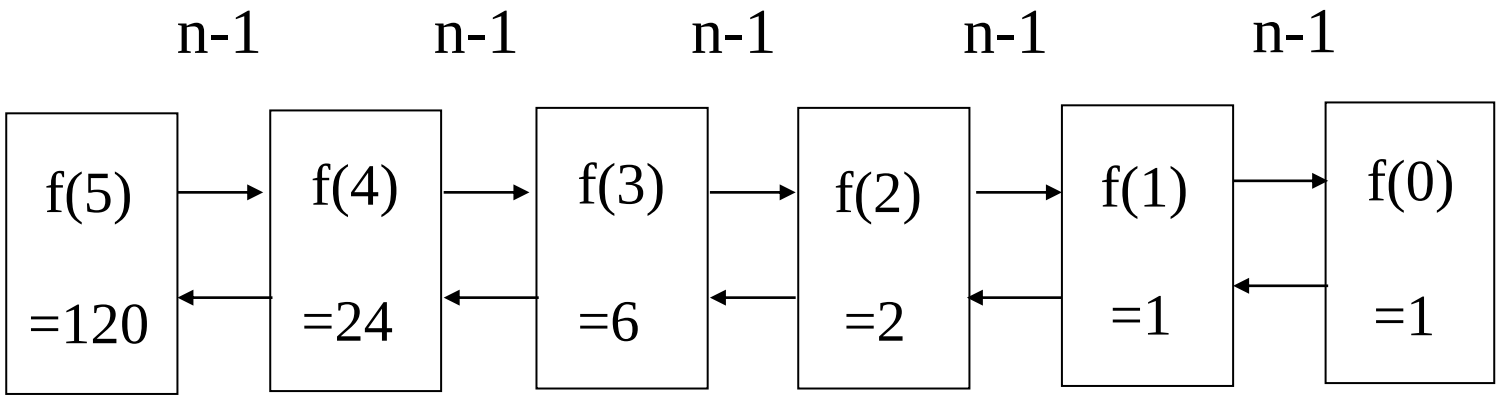
или

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

$$n! = n*(n-1)*(n-2)*\ldots*2*1$$

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

За n=5:



Рекурсивен вариант на функцията F(n) (factorial(n)):

```
#include <iostream>
using namespace std;
long factorial (long int n);//prototype
int main()
{
    long k;
    cout<<"\nВъведете цяло число:";
    cin>>k;
    cout<<"\n"<<k<<"! = "<<factorial(k);
    return 0;
}
long int factorial (long n)
{
if (n>1)
    return (n*factorial(n-1));
  else
    return(1);
}
```

Итеративен вариант на функцията F(n) (factorial(n)):

```
#include<iostream>
using namespace std;
int factor(int n);//prototype
void main()
 {
   int n;
   char c;
   do
   {
     cout<<"\nВъведете цяло число: ";
     cin>>n;
     factor(n);
     cout<<"\nДруга стойност (Y/N)? ";
     cin>>c;
   }
   while ((c!='N')&&(c!='n'));
 }
```

```
int factor(int n)
{
      int rez=1;
      for(int i=1;i<=n;i++)
       rez*=i;
      cout<<"\nn!="<<rez;
     return 0;
}
```

Алгоритъм на Евклид на намиране на най-големия общ делител на две положителни числа

$$NGOD(M,N)=\begin{cases} NGOD(N,M), & N>M \\ M & N=0 \\ NGOD(N,M\%N), & N\le M \end{cases}$$

Рекурсивен вариант:

```
#include <iostream>
using namespace std;
int NGOD (int m,int n); //Prototype
void main()
{
  int m, n;
  char r;
  do
   {
    do
    {
      cout<<"\nm=";
      cin>>m;
      cout<<"\nn=";
      cin>>n;
    }
    while ((n<0)||(m<0));
    cout<<"\nNGOD("<<m<<","<<n<<")="<<NGOD(m,n);
    cout<<"\nNext arg (y/n):";
    cin>>r;
   }
  while ((r=='y')||(r=='Y'));
 }
```

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

По-ефективна реализация:

```
int NGOD (int m, int n)
{ if (n>m)
    return NGOD(n,m);
     else  return n?NGOD(n,m%n):m;
}
```

Итеративна реализация:

```
int NGOD (int n, int m)
{
        int k, nn,mm;
        if (n<m)
        {nn=n; mm=m;}
            else
                    {nn=m; mm=n;}
          while (nn)
            {
                    k=nn;
                    nn=mm%nn;
                    mm=k;
            }
         return mm;
}
```

Интересен е случаят на т. н. “двойна” рекурсия, представляваща по-сложна рекурсивна зависимост. Като такъв пример може да послужи функцията на Акерман:

$$A(M,N)=\begin{cases} N+1, & M=0 \\ A(M-1,1) & N=0 \\ A(M-1,A(M,N-1)), & M,N\neq 0 \end{cases}$$

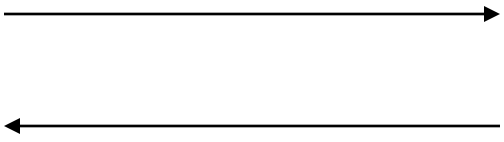
Рекурсивна реализация:

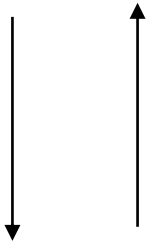
```
#include<iostream>
using namespace std;
int ack(int m, int n);//prototype
void main()
{
  int n, m;
  cout<<"n=";
  cin>>n;
  cout<<"m=";
  cin>>m;
  cout<<"Funcition ACK: "<<ack(m,n);
}

int ack(int m, int n)
{
   if (m==0)
            return n+1;
     else
              if (n==0)
                     return ack(m-1,1);
              else
                     return ack(m-1, ack(m,n-1));
}
```

Форми на рекурсивни функции

Рекурсивните обръщения могат да се разглеждат като движение:

напред/назад 

или рекурсивно спускане/завръщане 

Всяка рекурсивна функция се състои от определен брой оператори (S - тяло или действие), конструкция if (условие) (задължително!) и едно или няколко рекурсивни обръщения.

Форма 1

S се изпълнява преди рекурсивно обръщение (т. е. при рекурсивно спускане):

```
int Rec (параметри)
{
        S;
        if (условие)
                Rec (параметри);
}
```

Форма 2

S се изпълнява след рекурсивно обръщение (т. е. при рекурсивно връщане):

```
int Rec (параметри)
{
        if (условие)
                Rec (параметри);
    S;
}
```

Форма 3

Действието се извършва както преди, така и след рекурсивното обръщение (при спускане и при връщане:

```
int Rec (параметри)
{   S1;
      if (условие)
            Rec (параметри);
      S2;
}
```

Рекурсивна програма за формиране на аритметична прогресия:

$$a_i = a_{i-1} + b, \quad a_i <= c$$

```
#include <iostream>
using namespace std;
void prog (int n);              //prototype
int a, b, c;
void main()
{           cout <<"\nВъведете a:";   cin>>a;        // Например  a=1
            cout <<"\nВъведете b:";   cin>>b;        //   b=2
             cout <<"\nВъведете c:";   cin>>c;       //   c=7
             prog(a);
}
void prog (int n)
{
            cout<<n<<"   ";                          // 1   3    5   7
            if (n+b<=c)
                        prog(n+b);
            cout<<n<<"   ";                          // 7   5    3    1
}
```

Демонстрационен пример обръщане на символен низ: Hello! - !olleH

```
void main()
{…
  Reverse(c);
…
}
void  Reverse (char f)
{if (f!='!')
   {cin>>f;
     Reverse (f);
     cout<<f;
    }
 }
```

Текущо ниво	Рекурсивно спускане	Рекурсивно връщане
0 (main())	Reverse	
1	c='H' (false)	H'
2	c='e' (false)	e'
3	c='l' (false)	l'
4	c='l' (false)	l'
5	c='o' (false)	o'
6	c='!' (true)	!'

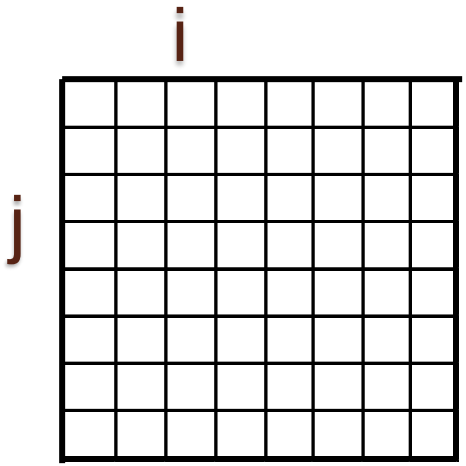
Задача за разполагане на царици върху шахматна дъска (или задача за 8-те царици)

Задачата представлява пример за използване на метода “на пробите и грешките”, а също така метода на обратно търсене (или връщане назад, или *backtracking*). Известно е, че още през 1850 върху тази задача е работил Гаус, но така и не успял да получи пълното й решение. В това няма нищо чудно, подобни задачи обикновено нямат аналитическо решение, те изискват голям обем неприятни сметки, търпение и точност. Затова подобни задачи се решават предимно от компютри.

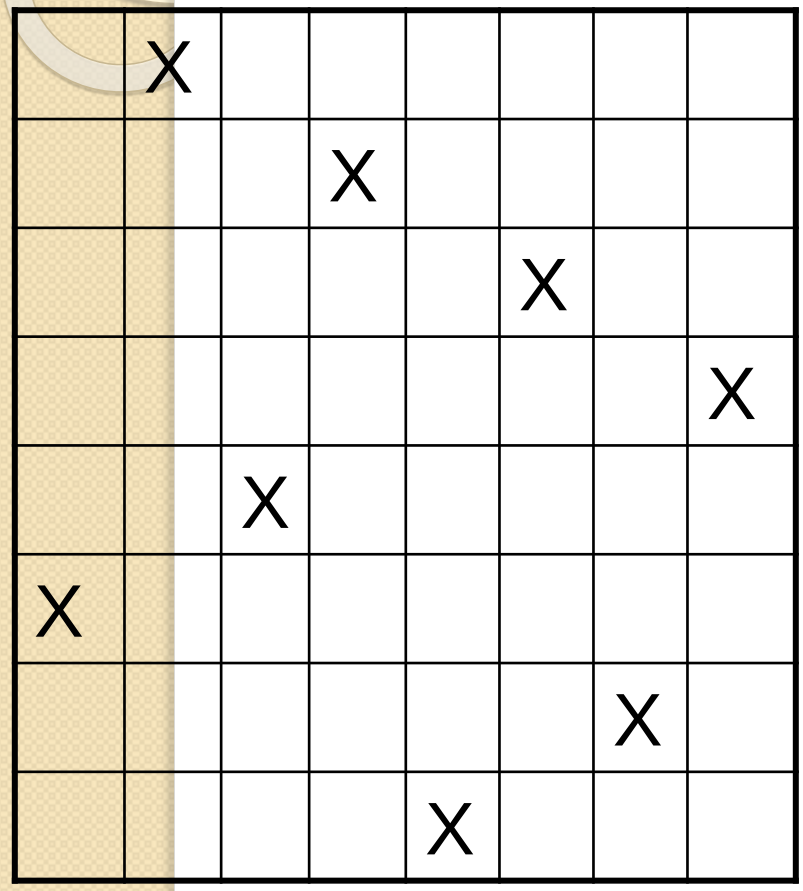
Условието на задачата е: Да се състави програма, която намира всички възможни начини за разполагане на **N** царици (максимално възможната стойност на **N** е 8) върху шахматна дъска по такъв начин, че никоя царица да не заплашва друга царица.

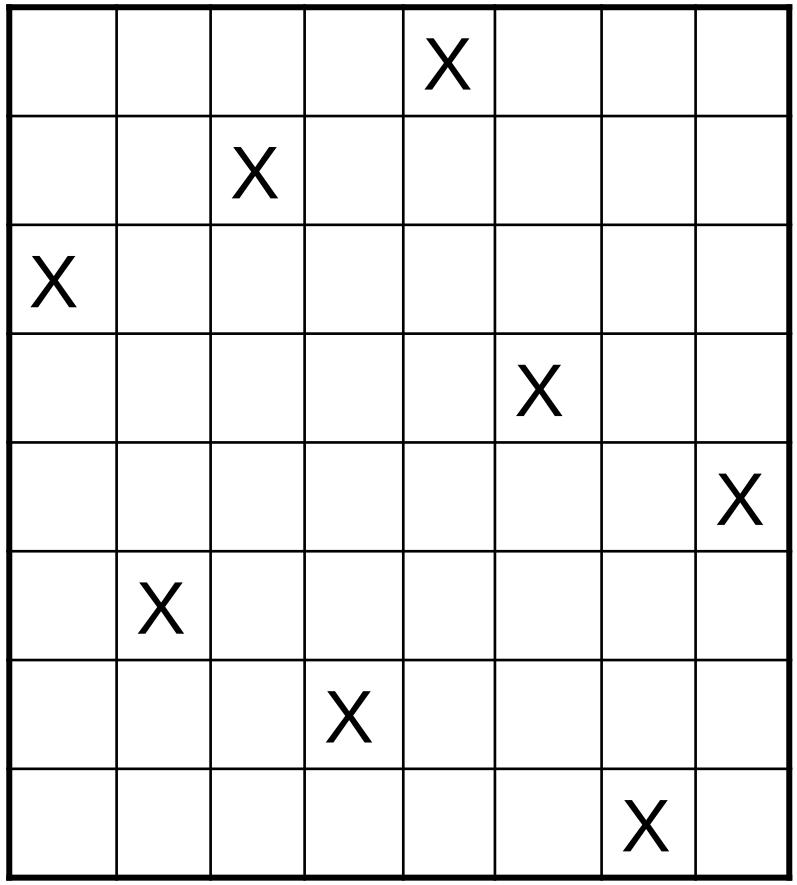
Естественият подход за намиране на решение на задачата е рекурсивен: ако сме разположили удачно **i** царици, проблемът се свежда до намиране на задоволително положение на **i+1**-вата царица. Задачата е решена, когато **i=N.** Остава да изведем намереното решение.

Известно е, че царицата заплашва всяка фигура в реда, колоната и диагонала, в които се намира, т.е. всеки ред и всяка колона може да съдържа само една царица. Царициите можем да представим с номерата на съответните колони(**i**). Тогава изборът на позиция за **i**-та царица се ограничава в **i**-та колона и се свежда до определяне на номера **j** на реда, в който тя може да се разположи в тази колона:



Две примерни решения на задачата:





Полуформалното описание на алгоритъма за разполагане на **i**-та царица е:

```
 PLACE(int i)
{
        for (j=0; j< n; j++)
          { if допустимо i-та царица в j-я ред
              {
                Регистрация на i-та царица;
                if  (i<n)   {PLACE(i+1)}
                            else PRINT;
                Премахване на регистрацията на i-та царица;
              }
           }
 } {PLACE}
```

За да разположим **i**-та царица задоволително, е необходимо да знаем, в кои редове и диагонали са поставени царици до момента. Как да представим тази информация?

Ако разгледаме шахматната дъска, то можем да открием следната закономерност: за “червените” диагонали сумата от координатите им i+j е константа:

i+j=const; 0 <= i+j <=2(N-1)

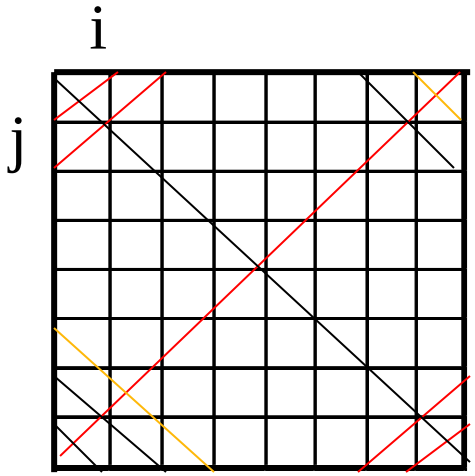
За “сините”диагонали i-j=const; 1-N <= i-j <= N-1

Така можем да дефинираме три булеви масива, които да съхраняват нужната ни информация:

a[j]= 1, ако в j ред няма царица, и 0, ако има;

b[i+j]= 1, ако в този диагонал няма царица, и 0, ако има;

c[i-j]= 1, ако в този диагонал няма царица, и 0, ако има;



Проверката на тези три масива ще ни позволява да определим, дали **i**-та царица може да бъде поставена в **j**-я ред. При положителен отговор **i**-та царица се поставя в клетката с координати **(i,j),** след което тази позиция се регистрира – съответните елементи на масивите **a**, **b** и **c** се установяват в 0. Продължаваме рекурсивно да търсим място на следващата, **i+1**-та царица. Ако това не е възможно при **i<N**, връщаме се обратно към **i**-та царица и се опитваме да я преместим в друга клетка с надеждата, че това ще доведе до друго решение. Непосредствено преди връщането е необходимо да премахнем регистрацията на текущата позиция на **i**-та царица, т.е. да присвоим на съответните елементи на масивите **a**,**b** и **c** стойност 1.

Ядрото на програмата е рекурсивната функция PLACE. В тялото на функцията е организиран цикъл за изчерпване на всички възможни варианти за разполагане на дадената царица.

```
#include <iostream>
using namespace std;
void print();
void true_false(int i, int j, int k);
void place(int i);

const int N=8;
int a[N]={1,1,1,1,1,1,1,1};
int b[2*N-1]={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
int c[2*N-1]={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
int pos[N]={0,0,0,0,0,0,0,0};
```

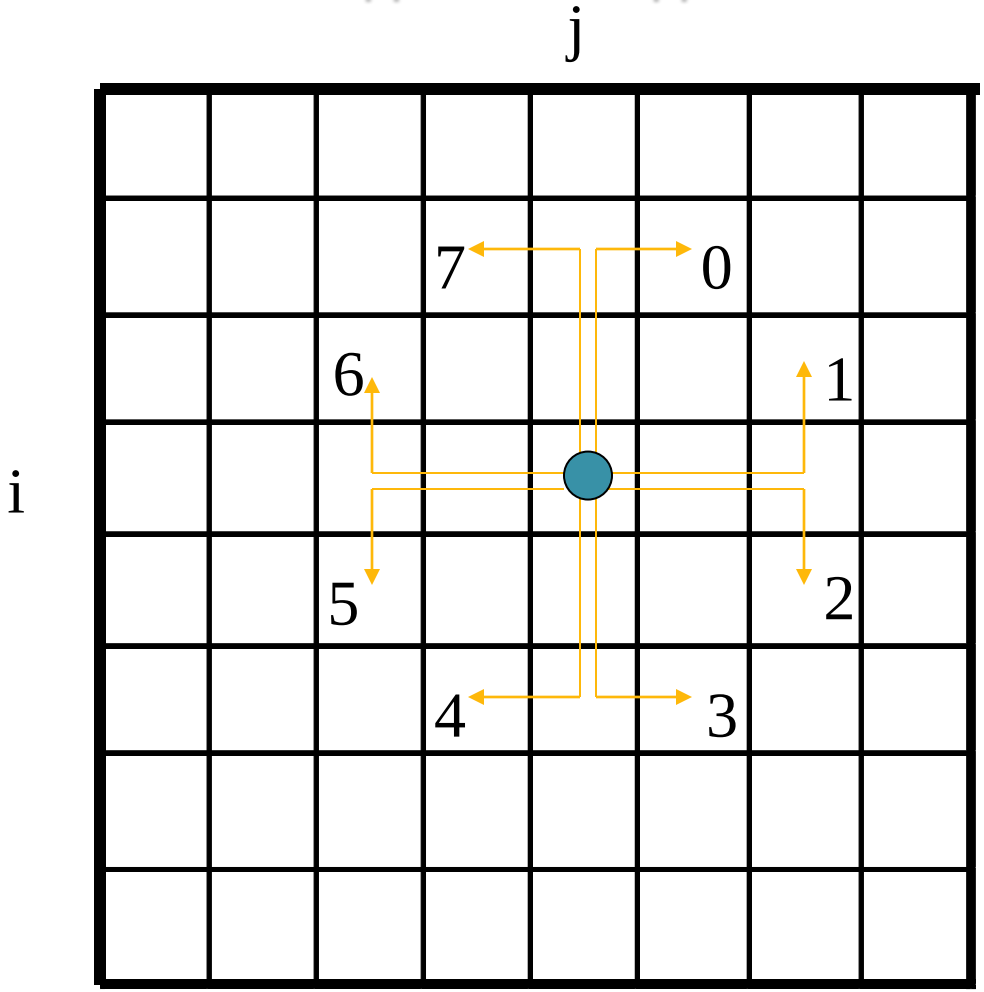
```
void main()
{
  place(1);
}

void print()
{
      for (int m=0; m<N; m++)
            cout<< "    " <<pos[m];
      cout<<endl;
}

void true_false(int i, int j, int k)
{
  a[j-1]=k;
  b[i+j-2]=k;
  c[i-j+7]=k;
}
```

```
void place(int i)
{
 int j;
 for (j=1; j<=N;j++)
  {
  if((a[j-1]==1)&&(b[i+j-2]==1)&&(c[i-j+7]==1))
            {
                  pos[i-1]=j;
                  true_false(i,j,0);
                  if (i<N)
                        place(i+1);
                  else
                        print();
                  true_false(i,j,1);
            }
  }
}
```

Задача за коня
(Обхождане на шахматна дъска с ходове на коня)



Реализация:

```
#include <iostream>
using namespace std;
#include <iomanip.h>
void print();
bool opit(int ibr,int x,int y);
const int N = 5, NSQ = N*N;
bool qq, q=false;
int a[8]={2, 1, -1, -2, -2, -1,  1,  2};
int b[8]={1, 2,  2,  1, -1, -2, -2, -1};
int h[N][N];
int main()
{       int i,j,y_start,x_start;
        for (i=0;i<=N-1;i++)
                for (j=0;j<=N-1;j++)
                        h[i][j]=0;
```

```
// N=5
// x_start=4; y_start=0 - OK!;
// x_start=0; y_start=4 - OK!;
// x_start=4; y_start=4 - OK!
// N=8
// x_start=0; y_start=4 - OK!;

cout<<"Enter start posiotion x_start= ";    cin>>x_start;
cout<<"Enter start posiotion y_start= ";    cin>>y_start;
h[y_start][x_start]=1;
print();
qq=opit(2, y_start,x_start);
if (qq)     print();
           else      cout<<"No solution!\n";
return 0;
}
```

```
void print()
{
        int i,j;
        for (i=1; i<=5; i++)
                cout<<'\n';
        for (i=0;i<=N-1;i++)
        {
                for (j=0;j<=N-1;j++)
                        cout<<setw(3)<<h[i][j]<<"  ";
                cout<<'\n';
        }
}
```

```
bool opit(int ibr,int x,int y)
{   int k,u,v;
    bool q1;
    k=0;
    do
        {  k++;  q1=false;
           u=x+a[k-1];   v=y+b[k-1];
          if ((u>=0 && u<=N-1) && (v>=0 && v<=N-1))
                if (h[u][v] == 0)
                 { h[u][v]=ibr;
                   if (ibr<NSQ)
                                {q1=opit(ibr+1,u,v);
                                 if (!q1)    h[u][v]=0;
                                 }    else     q1=true;
                  }
        } while (!q1 && (k!=7));
     q=q1;
     return q1;
}
```

Друг класически пример за използване на рекурсия е формирането на редицата на Фибоначи. Числата на Фибоначи представляват безкрайна поредица от цели числа, в която елементът с индекс **N (N>2)** е равен на сумата на двата непосредствено предхождащи го елемента - с индекси **N-1** и **N-2**. Първите два елемента на поредицата са **0** и **1**:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,…

$$FIB(N) = \begin{cases} 0, & N = 1 \\ 1, & N = 2 \\ FIB(N-1) + FIB(N-2), & N > 2 \end{cases}$$

Рекурсивна реализация на програма за изчисляване числата на Фибоначи:

```
#include<iostream>
using namespace std;
int fib (int i);//prototype
int main()
{
  int nn;
  cout<<"Въведете броя на числата (не повече
от 24): ";
  cin>>nn;
  for (int j=0; j<nn; j++)
          cout<<fib(j)<<"  ";
  return 0;
}
```

```
int fib (int i)
{
  if (i<1)
     return 0;
  if (i==1)
     return 1;
  return fib(i-1)+fib(i-2);
}
```

Реализацията е неефективна!

Итеративна реализация на програма за изчисляване числата на Фибоначи:

```
#include”stafx.h”
#include<iostream>
const int n=24;// max за int
int main()
{
 int f[n];
 int nn;
 cout<<"Въведете броя на числата (не повече от 24:";
 cin>>nn;
 f[0]=0;
 f[1]=1;
 for (int j=2; j<nn; j++)
      f[j]=f[j-1]+f[j-2];
 cout<<"\nЧислата на Фибоначи са:\n ";
 for (j=0; j<nn; j++)
            cout<<f[j]<<"  ";
 return 0;
}
```

ДИНАМИЧНО ПРОГРАМИРАНЕ

Метод, при който се извършва запомняне на междинните резултати от решенията на вече решени подзадачи с цел избягване на повторни пресмятания, се нарича *динамично програмиране (dynamic programming)* или *динамично оптимиране*. Основното предназначение на този метод е решаването на различни оптимизационни задачи.

За да разгледаме метода, нека да използваме известния вече пример за изчисляване на числата на Фибоначи (рекурсивна и итеративна реализация). На практика рекурсивната реализация изисква много излишни обръщения към функцията. За да се изчисли **n**-то число всеки път се изчисляват **n-1**-то **и n-2**-то числа, което е изключително неефективно. Броят на извършваните рекурсивни обръщения е ненужно голям. Алгоритмът е експоненциален и няма да работи при големи **n**.

При динамичната реализация изчислените вече числа се съхраняват в статичен масив *known[ ]* (или друг външен за рекурсивната функция масив) и се използват при изчисляване на следващите стойности. Алгоритмът има полиномиална сложност.

Динамична реализация на програма за изчисляване на числата на Фибоначи:

```
#include<iostream>
using namespace std;
int fib(int i); //prototype
int main()
{
  int nn;
  cout<<"nn=";
  cin>>nn;
  for (int j=0; j<nn; j++)
     cout<<fib(j)<<"  ";
  return 0;
}
```

```
int fib (int i)
{
 static int knownf[n];
 if (knownf[i]!=0)
        return knownf[i];
 int t=i;
 if (i<0)
    return 0;
 if (i>1)
    t=fib(i-1)+fib(i-2);
 return knownf[i]=t;
}
```

Основни стъпки при създаване на алгоритми и програми, реализиращи динамично програмиране (оптимиране)

- Описание на структурата на решението
- Рекурсивно дефиниране на резултата
- Изчисляване на стойностите-кандидати за оптимално решение
- Конструиране на оптималното решение

Още една класическа оптимизационна задача, която се решава чрез динамично оптимиране:

ЗАДАЧА ЗА РАНИЦАТА

Дадена е раница с вместимост M килограма и N предмета, всеки от които се характеризира с две числа: $\mathbf{m_i}$ (тегло) и $\mathbf{c_i}$ (стойност). Да се избере такова множество предмети, чиято сумарна стойност е максимална, а сумата от теглата им не надвишава M. Числата M, N, $\mathbf{m_i}$, и $\mathbf{c_i}$ са естествени числа (1<=i<=N).

Задачата се свежда до намиране на максимума на сумата:

$$\sum_{i=1}^{N} x_i c_i \qquad \sum_{i=1}^{N} x_i m_i \le M$$

$$c_i > 0,\ \mathrm{m_i} > 0,\ \mathrm{x_i} \in \{0,1\},\ \mathrm{i} = 1, 2, ..., \mathrm{N}$$

Решение: Ще дефинираме рекурентна целева функция *F(i)*, представляваща решение за раница с вместимост *i*.
Тогава

$$F(i) = \begin{cases} 0 & i = 0 \\ \max\limits_{j=1,2,\ldots,N;\, m_j \le i} [c_i + F(i - m_j)] & i > 0 \end{cases}$$

Рекурентната формула трябва да бъде реализирана като рекурсивна функция. За да се извърши динамично оптимиране, за всяка *F(i)*, ще се поддържа множество *s[i]*, съдържащо конкретните предмети, чието вземане води до тази минимална стойност. Освен това *s[i]* ще ни предпази от повторното включване на един и същи предмет.
Реализация - **[3]**

Задача за раницата

```
#include<iostream>
using namespace std;
const int N=10  // Брой предмети
void tr (int i, int tw, int c); // Рекурсивна функция

struct object
{
        int w;
        int c;
}
a[N]={10, 18, 11, 20, 12, 17, 13, 19, 14, 25, 15, 21, 16, 27, 17, 23, 18, 25,
19, 24};
int limW, w1, wh, w2, totalC, maxC;
int S[N], optS[N];          // текуща извадка и оптимална извадка
char flag[2]={' ', '*'};
```

```
void main()
{    int i;
     for (i=0; i<N; i++)        totalC+=a[i].c;
     cout<<"\nInput limW: ";   cin>>limW;
     cout<<"\nWeights: ";
     for (i=0; i<N; i++)
                    cout<<a[i].w<<' ';
     cout<<"\nCosts:   ";
     for (i=0; i<N; i++)
                    cout<<a[i].c<<' ';
     for (i=0; i<N; i++)
                    S[i]=optS[i]=0;
     tr(0, 0, totalC);
     cout<<"\n          ";
     for(i=0; i<N; i++)
                    cout<<flag[optS[i]]<<"  ";
}
```

```
void tr (int i, int tw, int c)
{int c1, k;
  if(tw+a[i].w<limW)  //включване на i-я предмет
          {S[i]=1; if (i<N-1)
                              tr(i+1, tw+a[i].w, c);
                    else      if (c>maxC)
                                        {maxC=c;
                                         for(k=0; k<N; k++)
                                                optS[k]=S[k]; }
           S[i]=0;
          }
          c1=c-a[i].c;
          if (c1>maxC)
                    if(i<N-1)  tr(i+1, tw, c1);
                      else
                    { maxC=c1;
                      for (k=0; k<N; k++)    optS[k]=S[k];
                    }
}
```

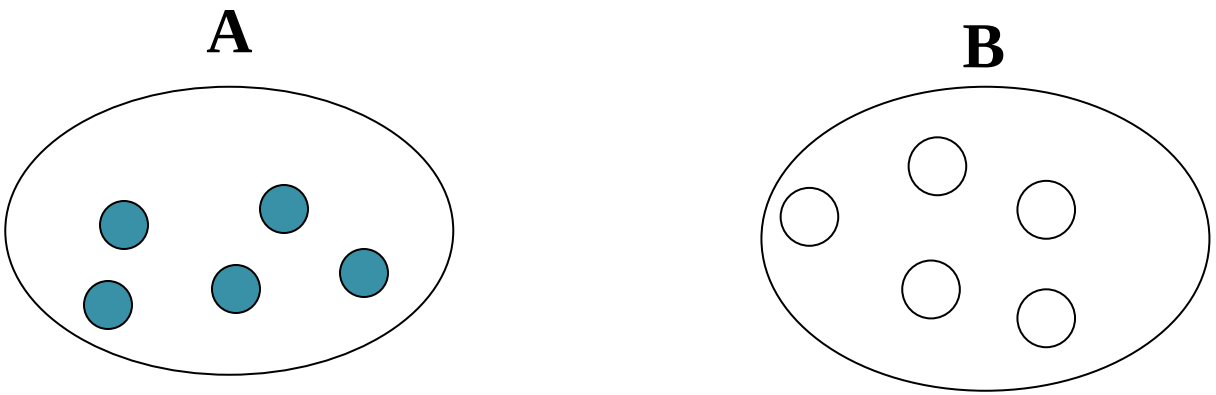
Задача за стабилните бракове (по Н. Уирт)

В какво се състои задачата?

Дадени са две непресичащите се множества **A** и **B** с по **n** елемента. Необходимо е да се намери множество двойки **[a, b]** от **n** елементи, такива, че **a in A, b in B** да отговарят на определени условия (критерии).

Критериите могат да бъдат най-различни. Единият от тях се нарича “Правило на стабилните бракове”.

Нека **A** да е множество, съдържащо **n** мъже, а **B** – множество, съдържащо **n** жени. Всички те желаят да сключат брак и си имат определени предпочитания.



Необходимо е да се състави алгоритъм, който да формира **n** двойки на базата на тези предпочитания. Ако двойките са съставени така, че съществуват мъж и жена, които не образуват по между си двойка (т.е. участват в различни двойки), но предпочитат един друг, то множеството бракове се нарича “нестабилно”. Ако не съществува такава ситуация (т.е. такава “неправилна” двойка), множеството се нарича “стабилно”.

Тази задача има значително по-общ характер и се решава много често: класиране по ВУЗ-ове и специалности след кандидат-студентски изпити, разпределение на студенти по групи за свободно избираеми дисциплини и т.н.

Примерът с браковете – е само една абстракция, която позволява да формулираме и да решим задачата.

Решението ще се търси чрез последователно получаване на двойки елементи от двете множества, до изчерпване на множествата. Нека да намерим **всички** стабилни комбинации, като за целта ще използваме подхода от задачата за 8-те царици. Нека **try(m)** е алгоритъм за търсене на партньорка за мъжа **m** и нека търсенето става съгласно списъка на предпочитания на този мъж.

Първата версия на алгоритъма е:

```
int  man, woman; //1..n

 try(int man)
 int       rank;              //1..n;
{          for (r=1; r<= n; r++)
             { От списъка на предпочитания на m-я мъж да
               се избере   r-та жена;
               if  (изборът е приемлив)
                    {
                      фиксиране на брака;
                        if  (m<>n)   try(succ(m));
                           else  да се запише стабилното множество;
                        да се отмени брака;
                     }
               }
}
```

Рекурсия и операции с променливите

а) променливи-параметри
б) локални променливи
Глобални променливи?

```
// Изчисляване на n!
// аргументът на функцията е заменен с глобална
// променлива
// Реализацията работи в средата на Borland C за DOS
// и не работи в средата MS Visual C++ за Windows

#include <iostream>
using namespace std;
const int n=6;
unsigned i;
unsigned long fact (void)
{
        if (i==1) return 1;
        return --i*fact();
}
int main (void)
{
        i=n+1;
        cout<<"n!="<<fact();
        return 0;
}
```

Още един пример: за дадено естествено число ***n*** да се отпечатат в нарастващ и намаляващ ред числата 10^k (1<=***k***<=***n***).
Например, за ***n*** =3: 10 100 1000 1000 100 10

Вариант 1: Всички променливи (освен входа ***n***) са параметри на рекурсивната функция:

```
#include <iostream>
using namespace std;
const  int n=5;

void printLine (unsigned k, unsigned long result)
{
  cout<<result<<' ';
  if (k<n) printLine(k+1, result*10);
  cout<<result<<' ';
}

void main()
{
 printLine(1,10);
  cout<<'\n';
}
```

Сега ще заместваме променливи-аргументи с глобални променливи:

Вариант 2: **k** става глобална променлива

```
#include <iostream.h>
const  int n=5;
unsigned k=0;
void printLine (unsigned long result)
{
  k++;
  cout<<result<<' ';
  if (k<n) printLine(result*10);
  cout<<result<<' ';
}
void main()
{
 printLine(10);
 cout<<'\n';
}
```

Вариант 3: ***result*** след подходящо модифициране преди и след рекурсивното обръщение също може да се изнесе извън функцията като глобална променлива

```
#include <iostream>
using namespace std;
const  int n=5;
unsigned k=0;
unsigned long result=1;
void printLine ()
{
  k++;
  result*=10;
  cout<<result<<' ';
  if (k<n) printLine();
  cout<<result<<' ';
  result/=10;
}
void main()
{
 printLine();
 cout<<'\n';
}
```

Рекурсия и ефективност

При всяко рекурсивно обръщение към подпрограма (функция) се заделят клетки от ОП за параметрите и за локалните променливи. Възможно е системният стек да се окаже недостатъчен за изпълнението на програмата, в която има много рекурсивни обръщения (т.е. дълбочината на рекурсията е голяма). Подобен проблем възниква също така и при неправилно рекурсивно обръщение – стекът се препълва.

В случая на голяма дълбочина на рекурсията и невъзможност програмата да се реализира по този начин необходимо е да се премине към другия подход – итеративен.

По-принцип рекурсивните програми са по-бавни и по-трудно се тестват. Използването на рекурсия изисква по-голям обем ОП.

Но, от друга страна, рекурсията води до повишаване на ефективността на програмисткия труд. Много са задачите (и ние видяхме някои от тях), които се програмират трудно и бавно без рекурсия и просто и бързо – с рекурсия.

Рекурсивните програми са по-четливи, по-ясни и разбираеми от съответните нерекурсивни програми. Затова трудно можем да намерим средно-сложна програма на някой от известните езици за програмиране, в която да няма рекурсивни обръщения. Но трябва да се има предвид, че не всички езици допускат рекурсия (например, Фортран, Бейсик, Кобол). Но и чрез тях може да се моделира механизма на рекурсията.

Съвременните компютри по-ефективно се справят с нерекурсивните програми, отколкото с рекурсивните.

Кога НЕ ТРЯБВА да прилагаме рекурсия:

- когато тя е неефективна в степен, по-голяма за допустимата в конкретния случай (например, бързодействието пада така, че това пречи на работата, или дълбочината на рекурсията е такава, че процесът на обръщенията става нереализуем поради препълване на системния стек);

- когато нерекурсивното решение е по-просто, по-естествено и много по-ефективно от рекурсивното (като задачите за събиране и умножение на цели числа).