

# **Съдържание**

- 1. Алгебрични изрази**
- 2. Минимизация на функции**
- 3. Автоматично диференциране**
- 4. Транспортна задача**
- 5. Задача за назначенията**
- 6. Разни задачи**

# Обработка на алгебрични изрази (parsing)

## Алгоритми за пресмятане на стойности

Три начина за представяне на израз:

- *infix*
- *prefix*
- *postfix*

*Инфиксната* нотация е често срещан начин за писане на изрази (това е училищния начин), докато *префиксните* и *постфиксните* нотации се използват предимно в компютърните науки.

Класическият подход за писане на код за пресмятане на аритметичните изрази:

- Преобразуване на инфиксен израз в постфиксен
- Пресмятане на постфиксен израз

## Infix notation

Всеки оператор е поставен между своите операнди, което е възможно само когато един оператор има точно два операнда (както при оператори за събиране, изваждане, умножение, деление и остатък от деление). Използват се скоби.

Syntax: operand\_1 operator operand\_2

Пример:  $(A+B)*C-D/(E+F)$

## Postfix notation

В постфиксната нотация, операторът идва след своите операнди. Постфиксната нотация е известна още като обратна полска нотация *reverse Polish notation* (RPN) и често се използва, защото позволява лесно пресмятане на изразите.

Syntax: operand\_1 operand\_2 operator

Пример :  $AB + C * DEF + / -$

Постфиксната (и префиксна) нотация има три общи характеристики:

- Операндите са в същия ред, в който биха били в еквивалентния инфиксен израз.
- Скоби не са необходими.
- Приоритетът на операторите е без значение.

## Преобразуване от infix към postfix

За да преобразуваме инфиксен израз в постфиксна нотация, трябва да знаем приоритета и асоциативността на операторите.

*Приоритетът* или силата на оператора определя реда на пресмятането; оператор с по-висок приоритет се пресмята преди оператор с по-нисък приоритет.

Ако всички оператори имат еднакъв приоритет, тогава редът на пресмятането зависи от тяхната *асоциативност*. Асоциативността на оператор определя реда, в който операторите със същия приоритет са групирани (отдясно наляво или отляво надясно).

Лява асоциативност:  $A+B+C = (A+B)+C$

Дясна асоциативност:  $A^B^C = A^(B^C)$  тук знакът ^ е за степенуване

За събирането е в сила закон за асоциативността според който няма значение реда на пресмятането:  $(A+B)+C = A+(B+C)$

За степенуването не е в сила закон за асоциативността

Пример: в книгите по математика когато е написано  $2^{3^4}$  се подразбира, че е  $2^{(3^4)}$ , т.е.

$$2^{3^4} = 2^{(3^4)} = 2^{3^4} = 2^{(3^4)} = 2^{81} = 2\,417\,851\,639\,229\,258\,349\,412\,352$$

$$\text{При грешно тълкуване } 2^{3^4} = (2^3)^4 = (2^3)^4 = 8^4 = 4\,096$$

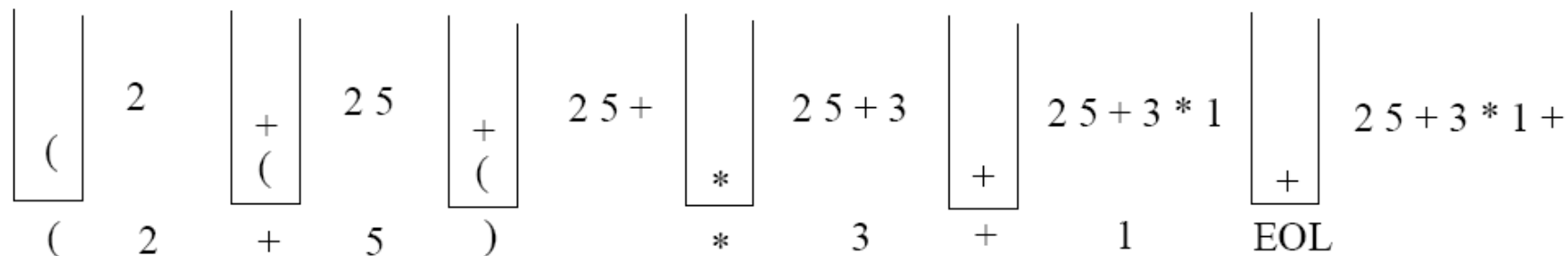
Процесът на преобразуване включва четене на операндите, операторите и скоби на инфиксния израз

1. Инициализираме празен стек  $S$  и празна резултатна низова променлива  $R$ .
2. Четем инфиксния израз отляво надясно по един знак  $C$ .
3. Ако  $C$  е операнд, добавяме (отдясно) към  $R$ .
4. Ако  $C$  е оператор, извадете от  $S$  операторите, докато стигнем до отваряща скоба, оператор с по-нисък приоритет или десен асоциативен символ с равен приоритет. Вкарваме  $C$  в  $S$ .
5. Ако  $C$  е отваряща скоба, вкарваме го в  $S$ .
6. Ако  $C$  е затваряща скоба, изваждаме от  $S$  всички оператори, докато стигнем до отваряща скоба и ги добавяме към  $R$ .
7. Ако бъде намерен край на входния низ, извадете всички оператори от  $S$  и ги добавете към  $R$ .



**Пример:** Infix expression:  $(2 + 5) * 3 + 1$

Стекът съдържа само оператори и скоби. Операндите слагаме направо в изхода.



## Реализация

```
// Return precedence of operators
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}
```

```
// Return associativity of operators
char associativity(char c)
{
    if (c == '^') return 'R';
    return 'L'; // Default to left-associative
}
```

```
void infixToPostfix(string s)
{
    stack<char> st;
    string result;

    for (int i = 0; i < s.length(); i++)
    {
        char c = s[i];
        if ((c >= 'a' && c <= 'z') ||
            (c >= 'A' && c <= 'Z') ||
            (c >= '0' && c <= '9'))
            result += c;
        else if (c == '(') st.push('(');
```

```
else if (c == ')')
{
    while (st.top() != '(')
    {
        result += st.top(); st.pop();
    }
    st.pop(); // Pop '('
}
```

```

else // if c == operator
{
    while (!st.empty() && prec(s[i]) <
           prec(st.top()) ||
           !st.empty() && prec(s[i]) ==
           prec(st.top()) &&
           associativity(s[i]) == 'L')
    {
        result += st.top(); st.pop();
    }
    st.push(c);
}
}

```

```
// Pop all the remaining elements from the stack
while (!st.empty())
{
    result += st.top();
    st.pop();
}

cout << result << endl;
}

int main()
{
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
}
```

## Пресмятане на постфиксен израз.

1. Инициализирайте празен стек  $S$ .
2. Четете отляво надясно по един знак  $C$  от постфиксния израз
3. Ако  $C$  е операнд, вкарайте го в  $S$ .
4. Ако  $C$  е оператор, извадете два операнда от  $S$ , извършете съответната операция и след това вкарайте резултата в  $S$ . Ако не можете да извадите два оператора, синтаксисът на постфиксния израз не е правилен.
5. При край на постфиксния израз извадете резултата от стека  $S$ . Тогава ако постфиксният израз е формиран правилно, стекът трябва да остане празен.



**Пример:** Postfix expression:  $2\ 5\ +\ 3\ *\ 1\ +$

Стекът съдържа само операнди.



## Реализация

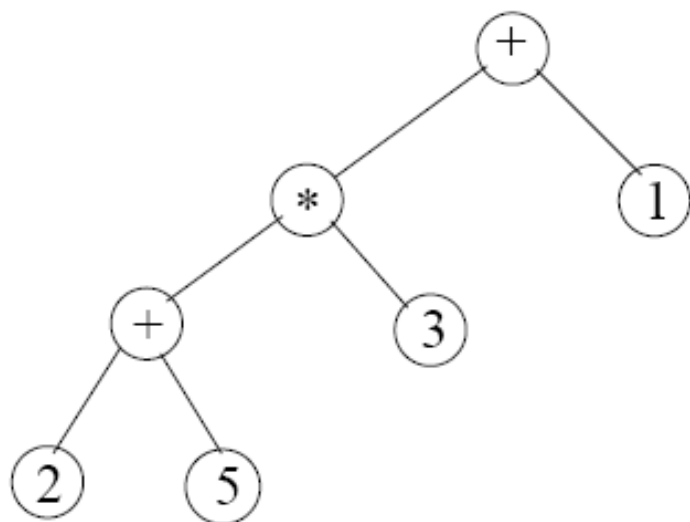
```
int evaluatePostfix(string exp)
{
    stack<int> st;
    for (int i = 0; i < exp.size(); ++i)
    {
        if (isdigit(exp[i])) st.push(exp[i] - '0');
        else // if it is an operator
        {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
```

```
switch (exp[i])
{
    case '+':
        st.push(val2 + val1); break;
    case '-':
        st.push(val2 - val1); break;
    case '*':
        st.push(val2 * val1); break;
    case '/':
        st.push(val2 / val1); break;
}
}
return st.top();
}
```

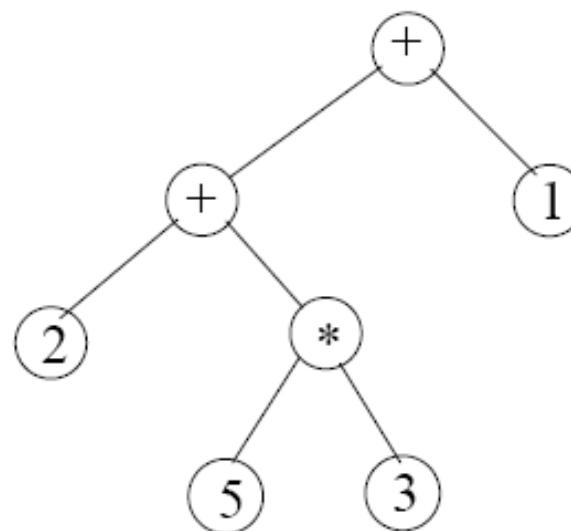
```
int main()  
{  
    string exp = "231*+9-";  
    cout << evaluatePostfix(exp);  
}
```

## Дърво на израз

$$(2 + 5) * 3 + 1$$

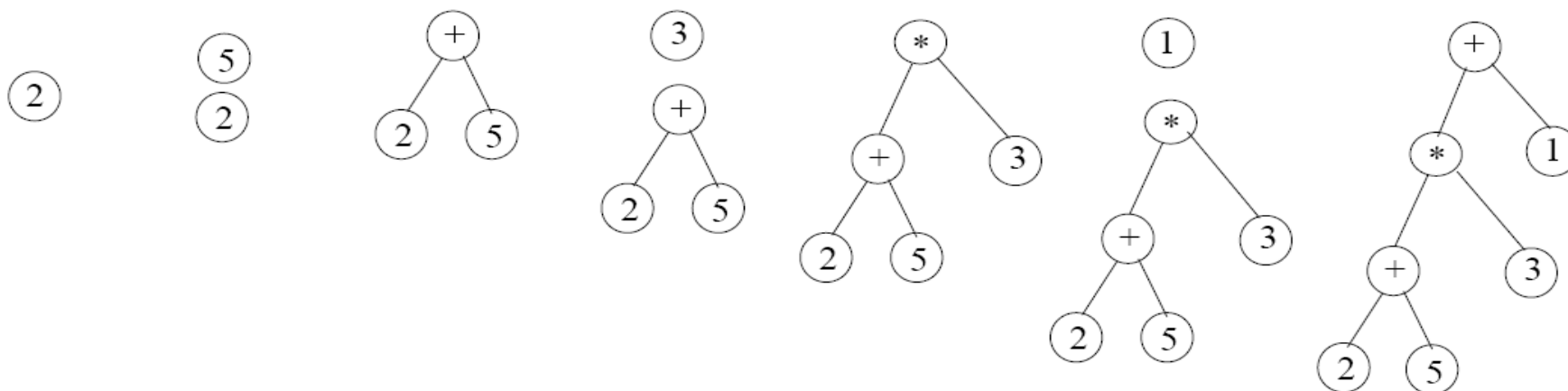


$$2 + 5 * 3 + 1$$



Пример за построяване на дърво за инфиксен израз:  $(2 + 5) * 3 + 1$

(два начина: от ляво надясно и от дясно наляво за фигурата)



Пресмятане по дървото

## Граматики (*Context-Free Grammars*)

Формални правила за продукция на изрази (рекурсивна дефиниция):

$$\begin{aligned} \textit{expression} &::= \textit{term} \{ + \textit{term} \} \\ \textit{term} &::= \textit{factor} \{ * \textit{factor} \} \\ \textit{factor} &::= (\textit{expression}) \mid \textit{operand} \\ \textit{operand} &::= \textit{number} \mid \textit{identifier} \\ \textit{number} &::= 0 \mid 1 \mid 2 \mid \dots \mid 9 \\ \textit{identifier} &::= A \mid B \mid \dots \mid Z \end{aligned}$$

Тази граматика описва изрази като тези, които използваме, напр.  $(A*B+A*C)*D$ . Всеки ред в граматиката се нарича правило за производство или правило за замяна. Продукциите се състоят от терминални символи  $\{ (, ), +, *, 0, 1, \dots, A, B, \dots \}$  и нетерминални  $\{ \textit{expression}, \textit{term}, \textit{factor}, \dots \}$ .

Метасимволи са:  $::=$ ,  $\{$ ,  $\}$ ,  $|$

За да се получат обичайни изрази трябва да добавим знак “ $-$ ” към “ $+$ ”, знак “ $/$ ” към “ $*$ ” и пр.

Например като добавим правила

$expression ::= term \{- term\}$

$term ::= factor \{/ factor\}$



## Програмна реализация

- Прочитане на низ, който съдържа алгебричен израз
- Създаване на дървовидно представяне на алгебричния израз
- Обратно преобразуване в израз на дървовидното представяне
- Използване на дървовидното представяне за пресмятане на числена стойност
- Използване на дървовидното представяне за извършване на алгебрични преобразования (опростяване и пр.)

```
#include<iostream>
using namespace std;
struct node
{ int tn; /* 0,1,2 */ char op;
  double v;
  node *arg1, *arg2;
};
```

```

/***** tree creation *****/
char str[999]; char ch; int spos=0;

void next() {ch=str[spos++];}

void expression(node*& pt);

void factor(node*& pt)
{ if(ch=='(')
    { next(); expression(pt); next(); }
  else
    {pt = new node; pt->tn=0; pt->v=0.0; pt->
    op=ch;
    pt->arg1=NULL; pt->arg2=NULL; next();
    }
}

```

```
void term(node*& pt)
{char op; node *a1, *a2;
  factor(a1); pt=a1;
  while ((ch=='*') || (ch=='/'))
    {op=ch; next(); factor(a2);
     pt=new node; pt->tn=2; pt->v=0.0; pt->op=op;
     pt->arg1=a1; pt->arg2=a2; a1=pt;
    }
}
```

```

void expression(node*& pt)
{ char op; node *a1, *a2;
  term(a1); pt=a1;
  while ((ch=='+') || (ch=='-'))
    {op=ch; next(); term(a2);
     pt = new node; pt->tn=2; pt->v=0.0; pt-
>op=op;
     pt->arg1=a1; pt->arg2=a2; a1=pt;
    }
}

```

```

void create_tree(node*& pt)
{ spos=0; next(); expression(pt); }

```

```
/****** output tree as a string *****/  
void output_tree(node* pt, char ch)  
{char s;  
  switch(pt->tn)  
  {case 0: cout << pt->op; break;  
    case 2:  
    { s=pt->op;
```

```

switch(s)
{case '*':
    if(ch=='/') cout << '(';
    output_tree(pt->arg1,s);
    cout << s;
    output_tree(pt->arg2,s);
    if(ch=='/') cout << ')';
    break;

case '/':
    if((ch=='*') || (ch=='/')) cout << '(';
    output_tree(pt->arg1,s);
    cout << s;
    output_tree(pt->arg2,s);
    if((ch=='*') || (ch=='/')) cout << ')';
    break;

```

```

        case '+': case '-' :
            if((ch=='-') || (ch=='*') || (ch=='/'))
cout << '(';
            if(ch=='+') output_tree(pt->arg1,s);
            else output_tree(pt->arg1,' ');
            cout << s;
            output_tree(pt->arg2,s);
            if((ch=='-') || (ch=='*') || (ch=='/'))
cout << ')';
            break;
    }
}
}

```



```
/****** computing using tree *****/
typedef char varnameType[21];
typedef double varvalType[21];
varnameType varname;
varvalType varval;
int maxvar;

void initval() /** example only **/
{ maxvar=4;
  varname[1]='a'; varname[2]='b';
  varname[3]='c'; varname[4]='d';
  varval[1]=1.0; varval[2]=2.0;
  varval[3]=3.0; varval[4]=4.0;
}
```

```
void argcalc(node* pt)
{ for(int i=1;i<=maxvar;i++)
    if(pt->op==varname[i])
        {pt->v=varval[i]; return;}
}
```

```

void twoargcalc(node* pt)
{if( (pt->arg1->tn==0) && (pt->arg2->tn==0) )
    {switch(pt->op)
        { case '+' : pt->v=pt->arg1->v+pt->arg2->v;
                    break;
          case '-' : pt->v=pt->arg1->v-pt->arg2->v;
                    break;
          case '*' : pt->v=pt->arg1->v*pt->arg2->v;
                    break;
          case '/' : pt->v=pt->arg1->v/pt->arg2->v;
                    break;
        }
    pt->tn=0;
}
}

```

```
void numcalc(node* pt)
{switch(pt->tn)
 {case 0 : argcalc(pt); break;
  case 2 :
    numcalc(pt->arg1);
    numcalc(pt->arg2);
    twoargcalc(pt); break;
 }
}
```

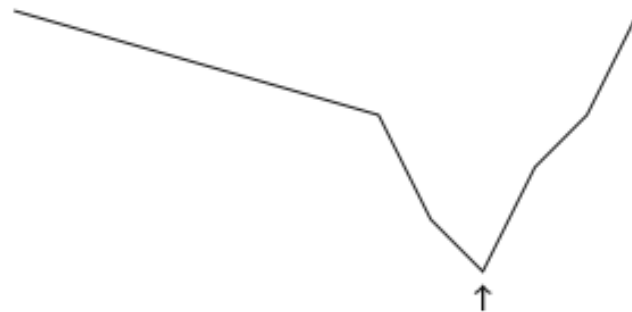
```
double value(node* pt)
{numcalc(pt); return pt->v;}
```

```
int main()
{node* pt;
  cin >> str;
  create_tree(pt);
  output_tree(pt, ' ');
  cout << endl;
  initval();
  cout << value(pt) << endl;
}
```

# Минимизация на функции

## Троично търсене

Разглеждаме функция  $f(x)$  в даден интервал  $[x_L, x_R]$  на изменение на аргумента  $x$  и знаем, че тази функция в началото само намалява, а след това само расте. Търсим точката на минимум:



Разделяме интервала  $[x_L, x_R]$  на три части с равни дължини:

$[x_L, a]$   $[a, b]$   $[b, x_R]$ , където

$$a = x_L + (1/3)(x_R - x_L) \quad \text{и} \quad b = x_L + (2/3)(x_R - x_L)$$

На всяка стъпка пресмятаме  $f(a)$  и  $f(b)$  и в зависимост от релацията

$$f(a) < f(b)$$

Продължаваме или в лявата част  $[x_L, b]$  или в дясната част  $[b, x_R]$ .

Всяка от тези части има дължина  $2/3$  от първоначалната.

Така на всяка стъпка намаляваме интервала в който търсим. След  $n$  стъпки ще имаме интервал с дължина  $(2/3)^n$  от първоначалната, в който се намира минимума.

Например, при начална дължина 1, след 10 стъпки дължината на интервала на неопределеност за минимума е 0.0173..., а след 20 стъпки е 0.0003007...

Този алгоритъм може да се приспособи и за целочислена аритметика.

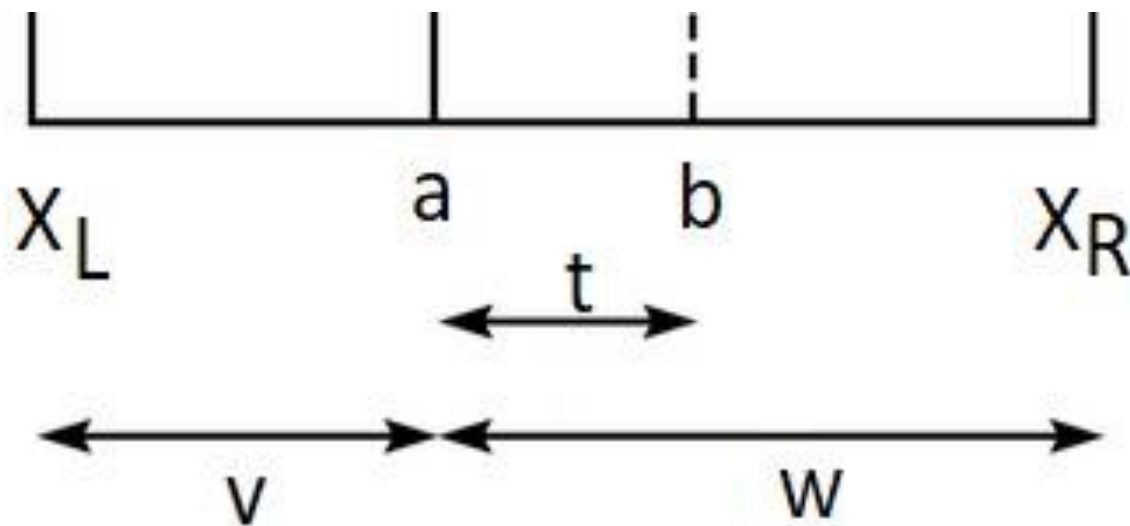


```
const double eps=0.00001;
double findMin(double L, double R)
{
    double d=R-L;
    if(d<eps) return (L+R)/2.;
    double a=L + (1./3.)*d;
    double b=L + (2./3.)*d;
    double fa=f(a);
    double fb=f(b);
    if(fa>=fb) return findMin(a,R);
    else return findMin(L,b);
}
```

На всяко извикване се премята двете стойности  $f(a)$  и  $f(b)$ .

Може ли да пресмятаме само една стойност, което е полезно, когато  $f$  е бавно изчисляема функция.

Да намерим такива точки  $a$  и  $b$ , които разделят интервала  $[x_L, x_R]$ , че едната от точките  $a$  и  $b$ , да е точка на разделяне на следващия интервал.



Точките  $a$  и  $b$  са избрани симетрично спрямо средата.  
Интервалът  $[x_L, x_R]$  се разделя в съотношение  $v:w$   
Искаме същото съотношение да бъде и  
за следващия интервал  $[a, x_R]$ , т.е.

$$v/w = t/v$$

Като имаме предвид, че  $t = w - v$ , поради симетрията, може да  
запишем

$$v/w = (w - v)/v, \text{ т.е. } v/w = w/v - 1.$$

Означаваме  $r = w/v$  и получаваме уравнението  $1/r = r - 1$ ,  
т.е.  $r^2 - r = 1$

Положителното решение на уравнението е числото, известно като  
златното сечение:

$$r = (1/2)(1+\sqrt{5}) \approx 1.618033..$$

Нека  $d$  е дължината на интервала  $[x_L, x_R]$   
т.е.  $d = v+w$

Понеже  $d/v = (w+v)/v = r + 1$ , то  $v = d/(r+1)$

Имаме  $1/(r + 1) \approx 0.3819661...$

Вземаме  $a = x_L + (0.3819661...)d$ ,  
вместо  $a = x_L + (0.33333... )d$ , както беше при обикновеното троично  
търсене

Стойността на  $b$  може да се пресметне от симетрията:

$$b = x_R - (0.3819661...)d$$

## Още за редицата на Фибоначи

Изразяване на редицата на Фибоначи:

$$s[0]=0, s[1]=1, s[n] = s[n-1]+s[n-2] \text{ за } n > 1$$

търсим решение във вида  $s[n]=p^n$

Получаваме уравнение  $p^n = p^{n-1} + p^{n-2}$

$$p^2 = p + 1$$

$$p_1 = (1 + \sqrt{5})/2 \text{ или } p_2 = (1 - \sqrt{5})/2$$

$$p_1 = 1.618033... \text{ или } p_2 = -0.618033...$$

Как да удовлетворим началните условия?

Търсим  $a$  и  $b$  така, че

$$s[n] = a \cdot p_1^n + b \cdot p_2^n$$

Понеже  $s[n] = s[n-1] + s[n-2]$

да намерим  $a$  и  $b$ , като заместим  $n=0$  и  $n=1$ :

$$s[0] = 0 = a + b$$

$$s[1] = 1 = a \cdot p_1 + b \cdot p_2$$

Решение:

$$a = 1/(p_1 - p_2) = 1/\sqrt{5} = 0.44721399..$$

$$b = 1/(p_2 - p_1) = -1/\sqrt{5} = -0.44721399..$$

$$s[n] = 0.4472... * (1.618...) ^n - 0.4472... * (0.618...) ^n$$

Phi=1.618... (Златно сечение)

(0.618...) ^n клони към нула

Пресмятаме:

$$(\text{Phi}^0)/\sqrt{5} = 0.4472135954999579 \text{ rounds to Fib}(0) = 0$$

$$(\text{Phi}^1)/\sqrt{5} = 0.7236067977499789 \text{ rounds to Fib}(1) = 1$$

$$(\text{Phi}^2)/\sqrt{5} = 1.1708203932499368 \text{ rounds to Fib}(2) = 1$$

$$(\text{Phi}^3)/\sqrt{5} = 1.8944271909999157 \text{ rounds to Fib}(3) = 2$$

$$(\text{Phi}^4)/\sqrt{5} = 3.0652475842498528 \text{ rounds to Fib}(4) = 3$$

$$(\text{Phi}^5)/\sqrt{5} = 4.959674775249769 \text{ rounds to Fib}(5) = 5$$

$$(\text{Phi}^{10})/\sqrt{5} = 55.00363612324741 \text{ rounds to Fib}(10) = 55$$

$$(\text{Phi}^{20})/\sqrt{5} = 6765.000029563931 \text{ rounds to Fib}(20) = 6765$$

Имаме формула за пресмятане  $O(1)$ , чрез числа с плаваща точка (т.е. тип `double` в C++)

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Понеже вторият член намалява бързо, приблизително имаме:  
(където  $[]$  означава цяла част)

$$F_n = \left\lfloor \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} \right\rfloor$$



Друг (практичен) начин за пресмятане: *чрез умножение на матрици*:

$$(F_{n-1} \quad F_n) = (F_{n-2} \quad F_{n-1}) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

КОЕТО ВСЪЩНОСТ Е:

$$F_{n-1} = F_{n-2} * 0 + F_{n-1} * 1; \quad F_n = F_{n-2} * 1 + F_{n-1} * 1$$

Означаваме

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Получаваме

$$(F_n \quad F_{n+1}) = (F_0 \quad F_1) \cdot P^n$$

Трябва да пресметнем матричната степен  $P^n$

Това може да стане чрез алгоритъма за бързо повдигане на степен  $O(\log n)$

Рекурсивна формула:

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ (a^{\frac{n}{2}})^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ (a^{\frac{n-1}{2}})^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

**Реализация чрез рекурсия:**

```
long long binpow(long long a, long long b)
{
    if (b == 0) return 1;
    long long res = binpow(a, b / 2);
    if (b % 2) return res * res * a;
    else return res * res;
}
```

**И без рекурсия:**

```
long long binpow(long long a, long long b)
{
    long long res = 1;
    while (b > 0)
    {
        if (b & 1) res = res * a;
        a = a * a; b >>= 1;
    }
    return res;
}
```

Преговор: умножение на две матрици:

```
int mat1[R1][C1], mat2[R2][C2];
```

R1 – брой редове в mat1

C1 – брой стълбове (колони) в mat1

R2 – брой редове в mat2

C2 – брой стълбове (колони) в mat2

трябва броя на елементите в ред на mat1 да е равен на броя на елементите в стълб на mat2, т.е.  $C1=R2$

```
int res[R1][C2];
```

матрицата-резултат има брой редовете от mat1 и брой стълбовете от mat2

```
for(int i=0; i < R1; i++)  
for(int j=0; j < C2; j++)  
{  
    res[i][j]=0;  
    for(int k=0; k < R2; k++) // C1=R2  
        res[i][j] += mat1[i][k]*mat2[k][j];  
}
```

“умножаваме ред по стълб“

Сложност  $O(n^3)$  при квадратни матрици. Има по-бърз метод на Strassen със сложност  $O(n^{\log 7})$ ,  $\log 7 \approx 2.8074\dots$

и прилагане на „разделяй и владей“

Представяме всяка от матриците A и B чрез 4 подматрици с размер  $n/2 \times n/2$ , както е показано по-долу.

Пресмятаме рекурсивно стойностите (матриците):

$ae + bg$ ,  $af + bh$ ,  $ce + dg$  и  $cf + dh$ .

има 8 умножения при стандартния подход:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Но това не подобрява скоростта

Може да направим конструкция със 7 умножения:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

$$p1 = a(f - h)$$

$$p3 = (c + d)e$$

$$p5 = (a + d)(e + h)$$

$$p7 = (a - c)(e + f)$$

$$p2 = (a + b)h$$

$$p4 = d(g - e)$$

$$p6 = (b - d)(g + h)$$

Това води до алгоритъма на Strassen

Връщаме се към Фибоначи: друг начин за пресмятане числата на Фибоначи е да използваме формулите:

$$F_{2k} = F_k (2F_{k+1} - F_k).$$

$$F_{2k+1} = F_{k+1}^2 + F_k^2.$$



## Програмна реализация:

```
pair<int, int> fib (int n)
{
    if (n == 0 return {0, 1};
    auto p = fib(n >> 1);
    int c = p.first * (2 * p.second - p.first);
    int d =
        p.first * p.first + p.second * p.second;
    if (n & 1) return {d, c + d};
    else return {c, d};
}
```

Формулите, които използваме тук може да се изведат от някои други формули:

$$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$$

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$$

при  $k = n$ :  $F_{2n} = F_n(F_{n+1} + F_{n-1})$

за всяко  $k$ ,  $F_{nk}$  е кратно на  $F_n$

и вярно е обратното: ако  $F_m$  е кратно на  $F_n$ , тогава  $m$  е кратно на  $n$

И интересното свойство:  $GCD(F_m, F_n) = F_{GCD(m,n)}$

Числата на Фибоначи са най-лошите възможни входни данни за Евклидовия алгоритъм

## Периодичност по модул на редицата на Фибоначи

Ще докажем, че последователността е периодична за всеки модул  $p > 1$  и че периодът започва от  $F_0$  (разглеждаме редицата  $F_0=0, F_1=1, F_2=1, ..$ )

при  $p=2$ :

0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1  
0 1 1 0 1 1 0 1 1 ...

при  $p=5$ :

0 1 1 2 3 0 3 3 1 4 0 4 4 3 2 0 2 2 4 1 0 1 1 2 3 0 3 3 1 4 0 4 4 3 2 0 2 2 4  
1 0 1 1 2 3 0 3 3 1...

Тока може да пресмятаме по модул  $p$  числата на Фибоначи с много големи номера.

Доказателство с допускане на противното. Разглеждаме  $p^2 + 1$  двойки числа на Фибоначи, взети по модул  $p$ :

$$(F_0, F_1), (F_1, F_2), \dots, (F_{p^2}, F_{p^2+1})$$

Може да има само  $p$  различни остатъци по модул  $p$  и най-много  $p^2$  различни двойки остатъци, така че измежду тях има поне две еднакви двойки. Това е достатъчно, за да докаже, че последователността е периодична, тъй като число на Фибоначи се определя само от двете предишни. Следователно, ако две двойки числа се повтарят, това би означавало, че числата след двойката ще се повтарят също.

## Фибоначиева бройна система (Фибоначиево кодиране)

Можем да използваме последователността на Фибоначи, за да кодираме положителни цели числа в двоични кодови думи.

Доказва се (теоремата на Зекендорф), че всяко естествено число  $n$  може да бъде представено по единствен начин като сума от числа на Фибоначи:

$$N = F_{k_1} + F_{k_2} + \dots + F_{k_r}$$

като числата на Фибоначи в представянето **не са последователни числа на Фибоначи**

Примери:

$$\begin{aligned}1 &= 1 &= F_2 &= (11)_F \\2 &= 2 &= F_3 &= (011)_F \\6 &= 5 + 1 &= F_5 + F_2 &= (10011)_F \\8 &= 8 &= F_6 &= (000011)_F \\9 &= 8 + 1 &= F_6 + F_2 &= (100011)_F \\19 &= 13 + 5 + 1 = F_7 + F_5 + F_2 = (1001011)_F\end{aligned}$$

Към кода се долепя 1, за да се означаи края на кодовата дума.  
Забележете, че само там се появяват два последователни бита 1.  
Водещите нулеви битове тук са важни.  
Кодът е  $d_0d_1d_2 \dots d_s1$ , където  $d_i$  е 1, ако  $F_{i+2}$  участва в представянето

Алгоритъмът за кодиране е:

1. Търсим в числата на Фибоначи от най-голямото до най-малкото, докато намерим число по-малко или равно на  $n$ . Нека това число е  $F_i$ .
2. Изваждаме  $F_i$  от  $n$  и записваме 1 в  $i-2$  позиция на кодовата дума (индексиране от 0 от най-левия към най-десния бит).
3. Повтаряме, докато няма остатък.
4. Добавяме финална единица.

Алгоритъм за декодиране:

Полагаме  $R=0$ . Премахваме крайния десен бит, който е равен на 1 .  
След това движейки се (от ляво надясно), ако  $i$ -тият бит е 1  
(индексиране от 0 от най-левия до най-десния бит), натрупваме в  $R$   
стойността  $F_{i+2}$ .



Още за бързото степенуване:

при  $n=15$  според описания алгоритъм имаме пресмятанията:

$$15 = 2*7 + 1; \quad 7 = 2*3 + 1; \quad 3 = 2*1 + 1$$

$$a \rightarrow a^2 \rightarrow a^3 \rightarrow a^6 \rightarrow a^7 \rightarrow a^{14} \rightarrow a^{15}$$

така се използват 6 умножения (всяка стрелка е едно умножение)

$a^{15}$  може да се пресметне с по-малко умножения, използвайки вече пресметнати резултати:

$$a \rightarrow a^2 \rightarrow a^3 \rightarrow a^5 \rightarrow a^{10} \rightarrow a^{15} \quad (5 \text{ умножения})$$

Кой е оптималният алгоритъм за бързо степенуване? Няма бърз начин за намирането му – прилага се backtracking

Задача за *най-къса адитивна редица*: трябва да се конструира редица  $a_1=1, a_2, a_3, \dots, a_k=n$ , така че  $k$  да е минимално и за всяко  $i > 1$ , да е вярно че  $a_i = a_{i_1} + a_{i_2}$ , за някои индекси  $i_1 < i$  и  $i_2 < i$ .

За фиксирано  $n$ , намирайки най-къса адитивна редица, ще имаме най-бърз алгоритъм за степенуване, защото при степенуване събирането на степенните показатели се превръща в умножение,

Пример за  $n=15$ , най-късата адитивна редица е

1, 2, 3, 5, 10, 15 или 1 2 4 5 10 15.

## Приложение на идеята за бързо степенуване

### Пресмятане на $x^n \bmod m$

Използваме същия код, поради свойствата на операцията mod

$$ab \bmod m = (a \bmod m)(b \bmod m) \bmod m$$

```
long long binpow (long long a, long long b, long long m)
{
    a %= m;
    long long res = 1;
    while (b > 0)
    {
        if (b & 1) res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

## Умножение по модул $m$

Задача: Умножете две числа  $a$  и  $b$  по модул  $m$ . Числата  $a$  и  $b$  се побират във вградените типове данни, но тяхното произведение е твърде голямо, за да се побере в 64-битово цяло число. Идеята е да се изчисли  $a \cdot b \pmod{m}$  без да се използва bigint аритметика.

Решение: Прилагаме двоичната конструкция на алгоритъма за бързо степенуване, като извършваме събирания вместо умножения. Така правим умножението на две числа за  $O(\log m)$  операции събиране и умножение по две.

$$a \cdot b = \begin{cases} 0 & \text{if } a = 0 \\ 2 \cdot \frac{a}{2} \cdot b & \text{if } a > 0 \text{ and } a \text{ even} \\ 2 \cdot \frac{a-1}{2} \cdot b + b & \text{if } a > 0 \text{ and } a \text{ odd} \end{cases}$$

## Пресмятане на пермутация k пъти

Задача: Дадена е последователност с дължина  $n$ . Приложете към последователността дадената пермутация  $k$  пъти.

Решение: Повдигнете пермутацията на  $k$ -та степен с помощта на двоично степенуване и след това приложете към редицата. Това дава времева сложност от  $O(n \log k)$ .

```
vector<int> applyPermutation
(vector<int> sequence, vector<int> permutation)
{
    vector<int> newSequence(sequence.size());
    for(int i = 0; i < sequence.size(); i++)
        newSequence[i] = sequence[permutation[i]]
    return newSequence;
}
```

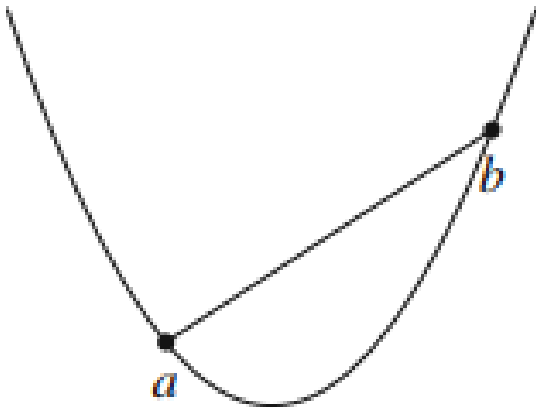
```

vector<int> permute
    (vector<int> sequence, vector<int> permutation,
     long long b)
{
    while (b > 0)
    {if (b & 1) sequence =
        applyPermutation(sequence, permutation);
        permutation
        = applyPermutation(permutation, permutation);
        b >>= 1;
    }
    return sequence;
}

```

# Изпъкнали (convex) функции

Една функция е изпъкнала, ако отсечка между всякакви две точки на графиката на функция винаги лежи над или върху графиката. Например  $f(x) = x^2$  е изпъкнала функция.



Ако знаем, че минималната стойност на изпъкнала функция е в диапазона  $[x_L, x_R]$ , можем да използваме троично търсене, за да го намерим. При някои изпъкнали функцията може да има минималната стойност в повече от една точка. Например  $f(x) = 0$ .

Свойство на изпъкналите функции:

Ако  $f(x)$  и  $g(x)$  са изпъкнали функции, тогава  $f(x) + g(x)$  и  $\max(f(x), g(x))$  са също изпъкнали функции. Това е вярно и за повече от две функции.



## Пример за минимизиране на сума

При дадени  $n$  числа  $a_1, a_2, \dots, a_n$ , разглеждаме задачата за намиране на стойност  $x$ , която минимизира сумата от абсолютните стойности:

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

Например, ако числата са  $[1, 2, 9, 2, 6]$ , оптималното решение е да изберем  $x = 2$ , което дава сумата

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

Тъй като всяка функция  $|a_k - x|$  е изпъкнала, сумата също е изпъкнала и използваме троично търсене.

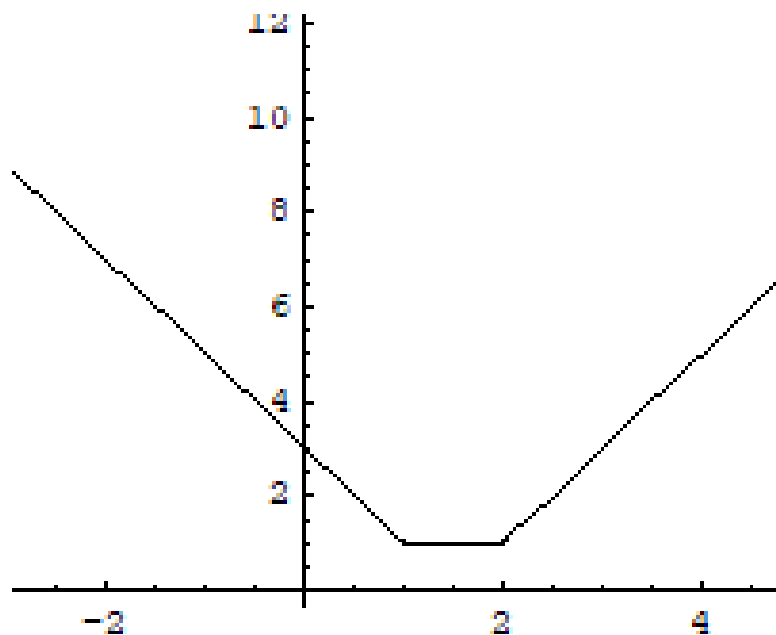
Има обаче и по-лесно решение. Оказва се, че оптималният избор за  $x$  винаги е медианата (средния елемент след сортиране) на дадените числа.

Например списъкът  $[1, 2, 9, 2, 6]$  става  $[1, 2, 2, 6, 9]$  след сортиране, така че медианата е 2.

Медианата винаги е оптимална, защото ако  $x$  е по-малко от медианата, сумата става по-малка чрез увеличаване на  $x$  и ако  $x$  е по-голямо от медианата, сумата става по-малка чрез намаляване на  $x$ .

Ако  $n$  е четно и има две медиани, тогава и двете медиани и всички стойности между тях са оптимален избор.

Пример за  $f(x) = |x-1| + |x-2|$



**Задача.** Минимизиране на функцията

$$f(x) = (a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2.$$

Например, ако числата са [1, 2, 9, 2, 6], най-доброто решение е да изберете  $x = 4$ , което дава

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

Тази функция е изпъкнала и може да използваме троично търсене.

По-просто решение: оптималният избор за  $x$  е средната стойност на числата. В примера средната стойност е  $(1 + 2 + 9 + 2 + 6) / 5 = 4$ .

Доказателство. Представяме

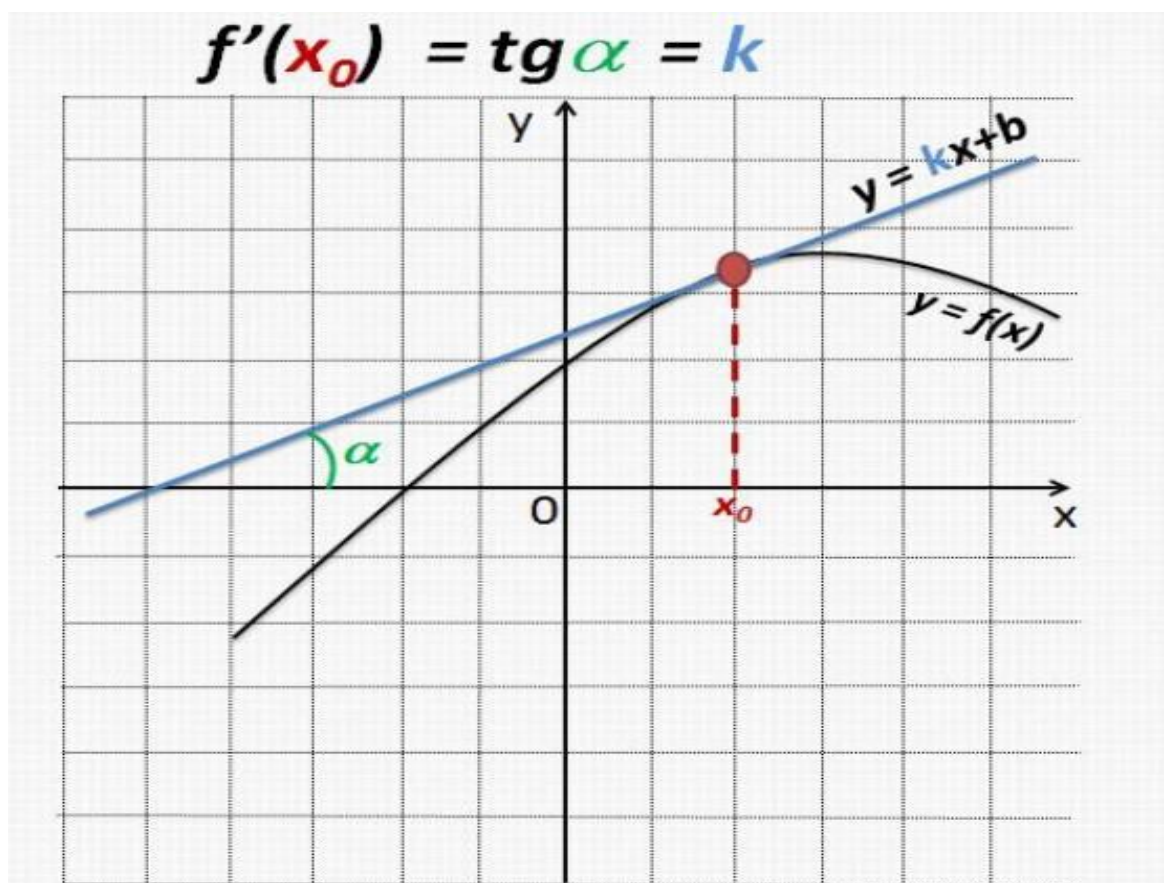
$$f(x) = nx^2 - 2x(a_1 + a_2 + \dots + a_n) + (a_1^2 + a_2^2 + \dots + a_n^2)$$

Последната част не зависи от  $x$ , така че можем да я игнорираме.

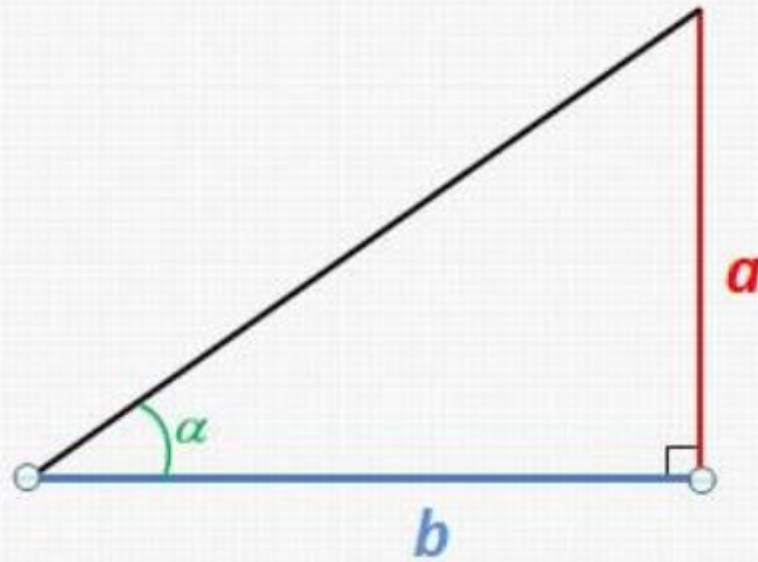
Остава  $nx^2 - 2xs$ , където  $s = a_1 + a_2 + \dots + a_n$ . Това е парабола (с отвор нагоре) с корени  $x = 0$  и  $x = 2s/n$ , а минималната стойност е средната стойност на корените  $x = s/n$ , т.е. средната стойност на числата  $a_1, a_2, \dots, a_n$ .

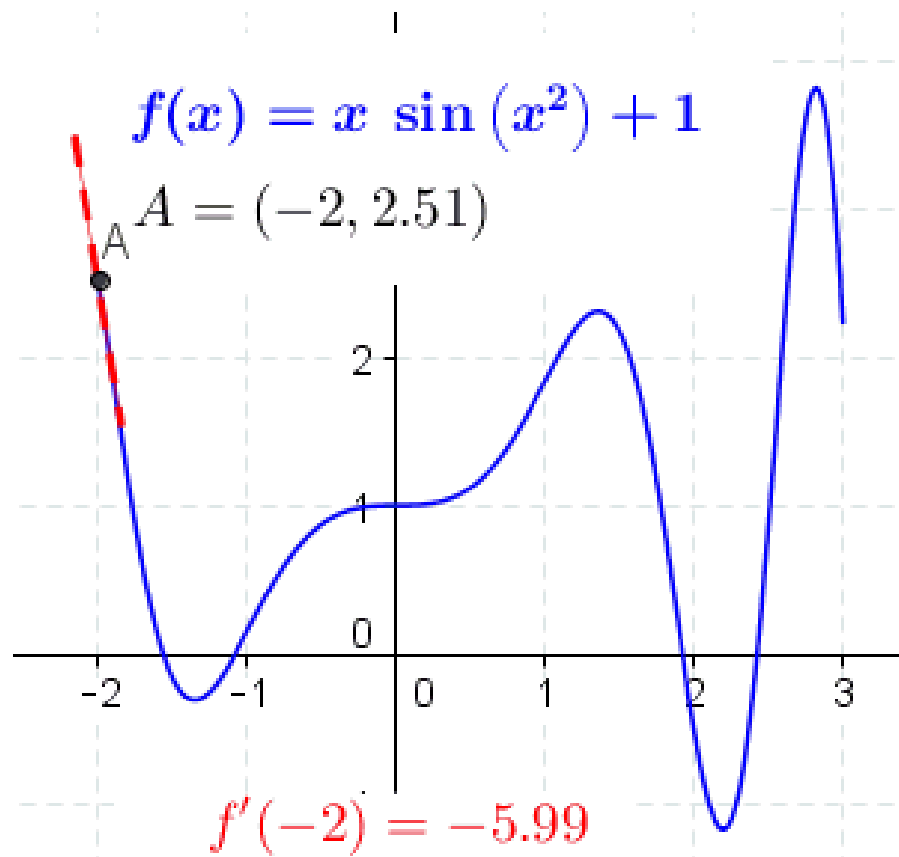
# Автоматично диференциране и числени методи

## Производна на функция на една променлива – нагледно изложение



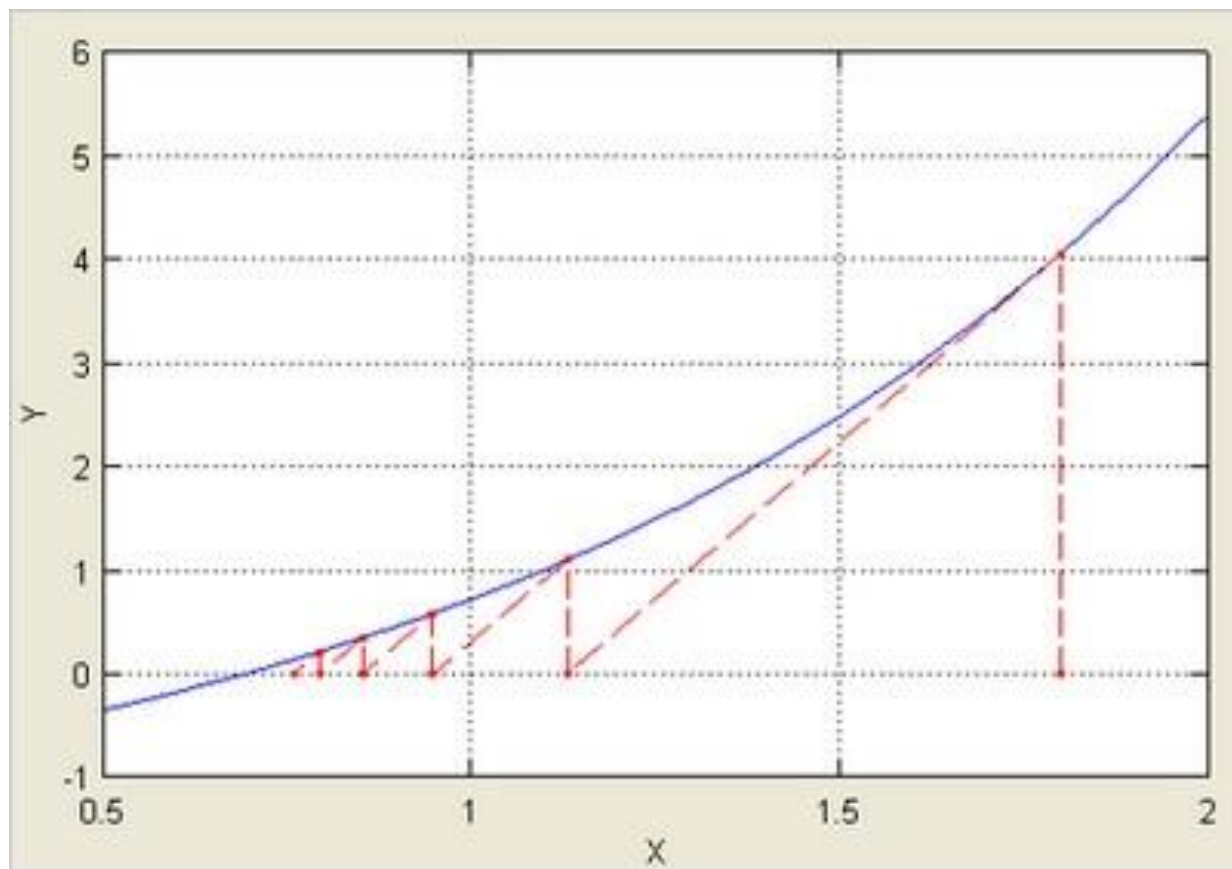
$$\operatorname{tg} \alpha = \frac{a}{b}$$

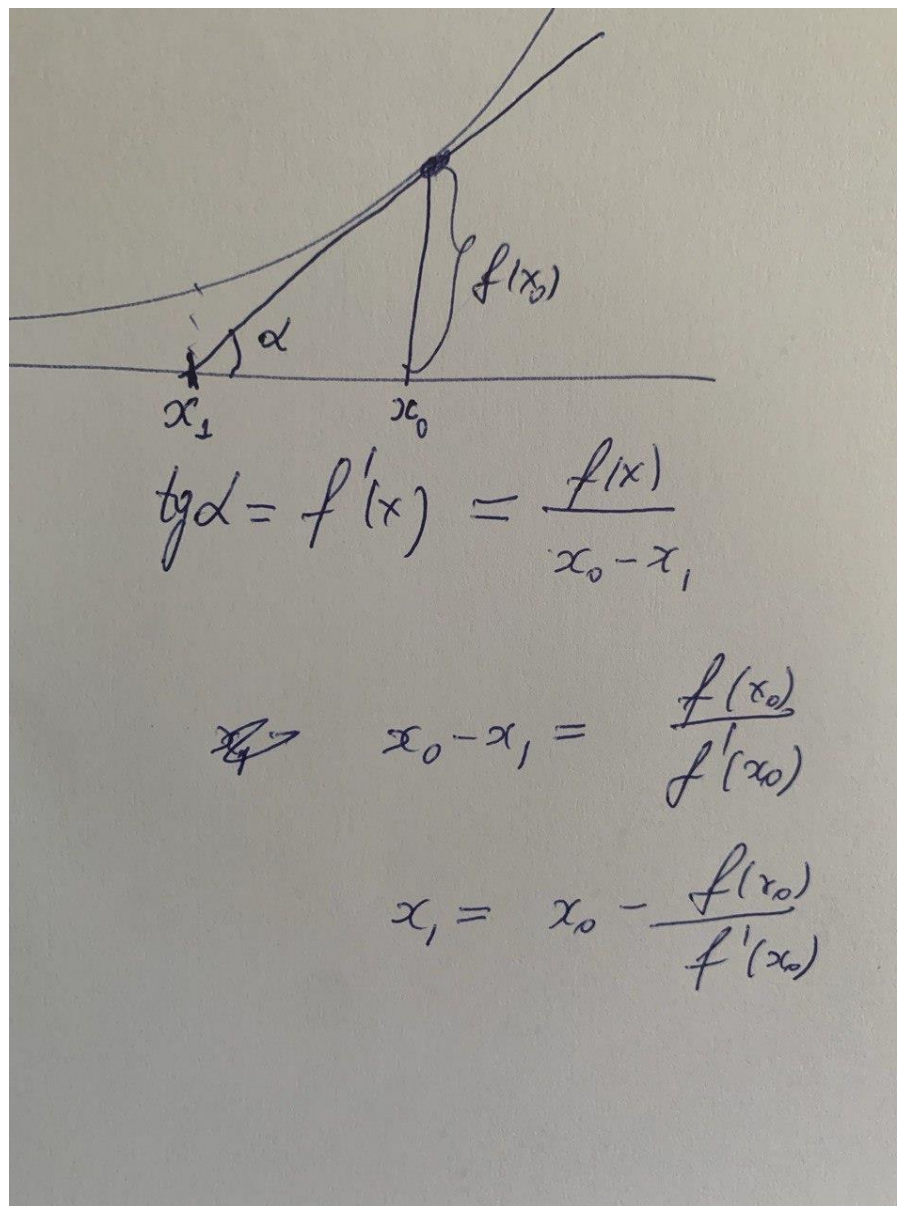






# Итеративен числен метод





## Пресмятане на производна в точка $x$

$$f(x) = c \rightarrow f'(x) = 0$$

$$f(x) = x \rightarrow f'(x) = 1$$

$$(f(x) + g(x))' = f'(x) + g'(x)$$

$$(f(x) - g(x))' = f'(x) - g'(x)$$

$$(f(x) * g(x))' = f'(x) g(x) + f(x) g'(x)$$

$$(f(x) / g(x))' = (f'(x) g(x) - f(x) g'(x)) / g^2(x)$$

$$f(x) = \cos(x) \rightarrow f'(x) = -\sin(x)$$

$$f(g(x))' = f'(g(x)) * g'(x)$$

## Реализация

```
class VD
{
    double val, der;
public:
    VD(double _val, double _der)
        {val=_val; der=_der;}
    VD(double c) {val=c; der=0;}
    static VD IndependentVar(double x)
        {return VD(x,1);}

    double getVal() const {return val;}
    double getDer() const {return der;}
```

```
friend VD operator+(const VD& f1, const VD& f2);  
friend VD operator-(const VD& f1, const VD& f2);  
friend VD operator*(const VD& f1, const VD& f2);  
friend VD operator/(const VD& f1, const VD& f2);  
friend VD cos(VD f);  
};
```

```
VD operator+(const VD& f1, const VD& f2) {  
    return VD(f1.val + f2.val, f1.der + f2.der);  
}
```

```
VD operator-(const VD& f1, const VD& f2) {  
    return VD(f1.val - f2.val, f1.der - f2.der);  
}
```

```
VD operator*(const VD& f1, const VD& f2) {  
    return VD(f1.val * f2.val,  
              f1.der * f2.val + f1.val * f2.der);  
}
```

```
VD operator/(const VD& f1, const VD& f2) {  
    return VD(f1.val / f2.val, (f1.der*f2.val -  
                                f1.val*f2.der) / (f2.val*f2.val));  
}
```

```
VD cos(VD f) {  
    return VD(cos(f.val), -sin(f.val)*f.der);  
}
```

```

VD f(double x, double a)
{
    VD xd = VD::IndependentVar(x);
    return a*xd*xd*xd - cos(xd/2);
}

int main()
{
    const double EPS = 1e-8;
    double x=5; double a=3;
    while (1)
    { VD F=f(x,a);
      if (fabs(F.getVal()) < EPS) break;
      x = x - F.getVal() / F.getDer();
      cout << x << endl;
    }
}

```

## Класическа транспортна задача

Еднороден продукт е произведен в пунктовете  $A_1, \dots, A_m$ , съответно в количества  $a_1, \dots, a_m$ . Пунктовете  $B_1, \dots, B_n$  се нуждаят от този продукт в количества съответно  $b_1, \dots, b_n$ . Считаме, че има условие за баланс:

$$\sum_{i=1}^m a_i = \sum_{j=1}^n b_j.$$

Транспортните разходи за превоз на единица продукт от пункта  $A_i$  до пункта  $B_j$  са съответно  $c_{ij}$ ,  $i = 1, \dots, m$ ,  $j = 1, \dots, n$ .



Да се направи план за транспортиране, така че потребностите да бъдат изцяло задоволени и общите транспортни разходи да бъдат минимални.

Означаваме план за снабдяване  $x=(x_{11},\dots,x_{mn})$

Искаме да минимизираме

$$z(x_{11}, \dots, x_{mn}) = \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} \rightarrow \min,$$

при следните ограничения:

$$\sum_{j=1}^n x_{ij} = a_i, \quad i = 1, \dots, m,$$

$$\sum_{i=1}^m x_{ij} = b_j, \quad j = 1, \dots, n,$$

$$x_{ij} \geq 0, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

Когато няма баланс, обикновено се въвежда фиктивен потребител (когато производството е по-голямо от потреблението) или фиктивен производител (когато потреблението е по-голямо от производството).

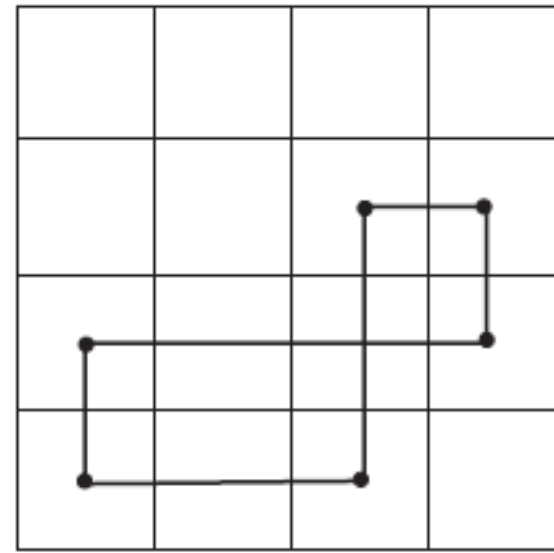
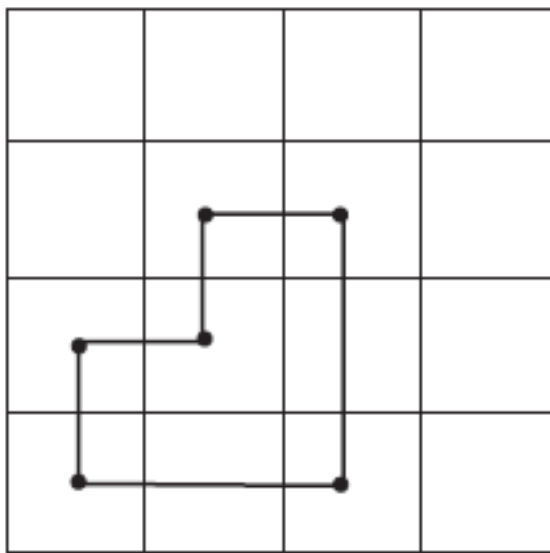
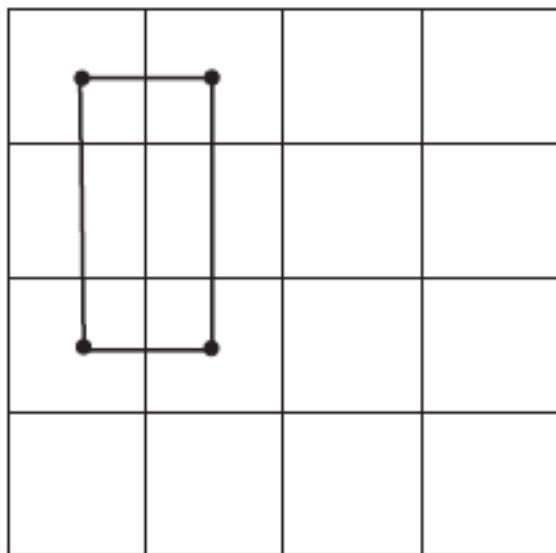
# Транспортна таблица

	$c_{11}$		$c_{12}$			$c_{1n}$	
$x_{11}$		$x_{12}$		.....	$x_{1n}$		$a_1$
	$c_{21}$		$c_{22}$			$c_{2n}$	
$x_{21}$		$x_{22}$		.....	$x_{2n}$		$a_2$
$\vdots$		$\vdots$		.....	$\vdots$		$\vdots$
	$c_{m1}$		$c_{m2}$			$c_{mn}$	
$x_{m1}$		$x_{m2}$		.....	$x_{mn}$		$a_m$
$b_1$		$b_2$		.....	$b_n$		$a$ $b$

По-долу ще разглеждаме цикли в тази таблица.

Съвкупност от клетки на транспортната таблица се нарича *цикъл*, ако начупената линия, образувана от отсечки с върхове в тези клетки, е затворена и от всеки две отсечки с общ връх едната лежи в ред, а другата — в стълб на транспортната таблица.

Съвкупност от клетки в транспортната таблица, която съдържа *цикъл*, се нарича *циклична*. В противен случай се нарича *ациклична*.



Всеки цикъл се състои от четен брой клетки. Във всеки цикъл две последователни (съседни) клетки имат равни първи или втори индекси, т. е. съседните клетки лежат в един и същи ред или стълб на транспортната таблица.

Един план  $x=(x_{11},\dots,x_{mn})$  се нарича *опорен*, когато съвкупността от клетките  $(i,j)$  в таблицата за които  $x_{ij} > 0$  е *ациклична*. Ние ще търсим оптимален план измежду опорните планове.

Свойство: За всяка празна клетка  $(i,j)$  съществува точно един цикъл, който я свързва с базисните (пълните) клетки на опорния план  $x$ .

Задачата се решава чрез итерации с последователно намиране на опорни планове.

## Намиране на начален опорен план (метод на северозападния ъгъл)

Пример (тук за показани цените  $c_{ij}$ ,  $a_i$  - отдясно и  $b_j$  - отдолу):

	3	5	7	11	
	1	4	6	3	100
	5	8	12	7	130
150		120	80	50	170



Започваме от северозападния ъгъл и записваме по-малкото от двете стойности  $a_1$  и  $b_1$  и продължаваме или в същия ред или в същия стълб:

	II	IV	V	VI	
I	100				100
III	50	80			130 80
VI		40	80	50	170 130 50
	150	120	80	50	
	50	40			

Транспортните разходи са 2300.

Метод на минималния елемент – вместо са започваме от северозападния ъгъл, започваме от клетката с най-малко  $C_{ij}$ .

Това дава в повечето случай по-добър начален план:

	II	V	VI	IV	
III	20	80			100
I	130				130
VI		40	80	50	170
	150	120	80	50	

20 40

Транспортните разходи са 2220.

Всеки опорен план има  $m + n - 1$  пълни клетки.

Намиране на следващи планове (чрез „разпределителния метод“)

Нека  $(k,l)$  е празна клетка. Образуваме единствения ЦИКЪЛ

$$\gamma_{kl} : (k, l)(k, j_1) \dots (i_s, j_s)(i_s, l).$$

който я свързва с клетките от текущия опорен план.

Клетките от цикъла маркираме алтернативно със знаци  $+$  и  $-$ , започвайки от клетката  $(k, l)$  със знак  $+$ . Тогава относителната оценка на променливата  $x_{kl}$  е

$$\bar{c}_{kl} = \sum_{(i,j) \in \gamma_{kl}^+} c_{ij} - \sum_{(i,j) \in \gamma_{kl}^-} c_{ij},$$

**Теорема:** Опорен план е оптимален т.с.т.к. оценките за всички празни клетки са неотрицателни.

Алгоритъм:

1. Построяваме начален опорен план.
2. За всяка празна клетка построяваме цикъла ѝ, свързващ я с клетките на опорния план. Ако критерият за оптималност е изпълнен – спираме.
3. Намираме най-малката оценка за всяка празна клетка по нейния цикъл:

$$\bar{c}_{i_0 j_0} = \min\{\bar{c}_{ij} : \bar{c}_{ij} < 0\}.$$

и променяме плана по този цикъл като намираме най-малкото  $x_{ij}$  от отрицателните клетки и го изваждаме от отрицателните клетки и го прибавяме към положителните клетки. Повтаряме с новия план т. 2.

*Забележка:* алгоритмът работи когато вземем първата отрицателна оценка, вместо да търсим горния минимум.

Пример. Тръгваме от вече намерения опорен план по метода на северозападния ъгъл от предишния пример. Първата празна клетка е (1,2) и нейният цикъл е (1,2)(2,2)(2,1)(1,1). Оценката за клетка (1,2) е  $5 - 3 + 1 - 4 = -1 < 0$ , следователно планът не е оптимален:

	3		5		7		11
100							
	1		4		6		3
50		80					
	5		8		12		7
		40		80		50	

Правим итерация като променяме плана по цикъла (изваждаме 80 от отриц. клетки и го добавяме към

полож., т.е. (1,2) става непразна клетка, а (2,2) става празна.

	3		5		7		11
20		80	-		+		
	1		4		6		3
130							
	5		8		12		7
		40	+		80	-	50

Първата празна клетка е (1,3) и е с цикъл (1,3)(3,3)(3,2)(1,2). Оценката е  $7 - 12 + 8 - 5 = -2$ , т.е. планът не оптимален и правим следваща итерация. Стойността 80 трябва да се извади от отриц. клетки и да се добави към положителните. Обаче на две места се получават



нули и едната от тези клетки (с по-малкото  $c_{ij}$ ) става „базисна нула“, т.е. не става празна.

	3		5		7		11
20	-		0	+		80	
	1		4		6		3
130							
	5		8		12		7
	+		120	-		50	

Сега пресмятаме оценката на първата празна клетка (1,4) и се получава положителна. Трябва да проверим оценките на всички празни клетки. Намираме, че за клетка (3,1) оценката е отрицателна и правим следваща итерация.

	3		5		7		11
		20		80			
	1		4		6		3
130							
	5		8		12		7
20		100				50	

Сега може да се провери, че оценките на всички празни клетки са неотрицателни. Следователно намерили сме оптималното решение. Оптималната стойност е 2040.

Забележка: Сега има някои празни клетки с оценка 0.  
Това означава, че задачата има и други оптимални  
планове.

Забележка: При целочислени стойности на данните,  
транспортната задача винаги има целочислено решение.

## Задача за назначенията (Assignment problem)

### Пример

Трите деца Джон, Карън и Тери на мистър Смит искат да изкарат малко пари за екскурзия. Мистър Смит е избрал три вида работи, които децата могат да изпълнят срещу определено заплащане: косене на ливадата, чистене на гаража и измиване на колата. За да избегне ненужни спорове между децата, той попитал тайно всяко дете колко иска за всеки вид работа. Резултатите от допитването (в долари) са представени в таблицата:

	Косене на ливадата (1)	Чистене на гаража (2)	Миене на колата (3)
Джон (1)	15	10	9
Карън (2)	9	15	10
Тери (3)	10	12	8

Как мистър Смит да разпредели работата между децата, така че разходите му да бъдат минимални? Всъщност трябва да отбележим 3 клетки в таблицата, така че да няма повече от една отбелязана клетки във всеки ред и във всеки стълб, и сумата от числата в отбелязаните клетки да е минимална.

Решението е за клетките (1,2) (2,1) (3,3) и минималните разходи са  $10 + 9 + 8 = 27$ .

# Унгарски метод

Да се намери

$$\min z = \sum_{i=1}^m \sum_{j=1}^m c_{ij} x_{ij},$$

при ограничения

$$\sum_{j=1}^m x_{ij} = 1,$$

$$i = 1, \dots, m,$$

$$\sum_{i=1}^m x_{ij} = 1,$$

$$j = 1, \dots, m,$$

$$x_{ij} \in \{0, 1\},$$

$$i = 1, \dots, m, \quad j = 1, \dots, m.$$

Да отбележи, че в матрицата на решението  $x_{ij}$  във всеки ред и всеки стълб може да има само по една единица и всички останали стойности са 0.

Задачата за назначенията е частен случай на транспортната задача, но за нея има по-бърз метод, наречен “Унгарски метод”, чийто по-нови модификации достигат сложност  $O(n^3)$ .

Интересно е, че Якоби е решавал тази задача и има публикация на латински език от 1890 г. – една от последните научни публикации на латински език.

## Алгоритъм

**Стъпка 1.** Намираме минималния елемент във всеки ред на матрицата на цените. Конструираме нова матрица, като изваждаме от всеки елемент на матрицата минималния елемент за съответния ред.

За тази нова матрица намираме минималния елемент във всеки стълб. Конструираме нова матрица (наречена матрица с *редуцираните цени*), като изваждаме от всеки елемент на втората матрица минималния елемент за съответния стълб.



**Стъпка 2.** Зачертаваме с минималния възможен брой линии (хоризонтални, вертикални или и двете) всички нули в матрицата с редуцираните цени. Ако броят на тези линии е  $m$ , намерено е оптимално решение, чийто единици се намират точно там, където са нулите в матрицата с редуцираните цени. **КРАЙ.**

Ако броят на линиите е по-малък от  $m$ , преминаваме към Стъпка 3.

**Стъпка 3.** Намираме най-малкия ненулев елемент (нека стойността му е  $k$ ) в матрицата с редуцираните цени, който не е зачертан от линиите в Стъпка 2. Сега изваждаме  $k$  от всеки *незачертан* елемент на матрицата с редуцираните цени и прибавяме  $k$  към всеки елемент на матрицата с редуцираните цени, който е зачертан от две линии.

Връщаме се към Стъпка 2 с така модифицираната матрица с редуцираните цени.

Забележка: В задача с големи размери трябва да имаме бърз метод за намирането на минималния брой линии, с които да зачертаем всички нулеви елементи на матрицата с редуцираните цени.

Пример. Дадена е следната матрица на цените:

$$\begin{bmatrix} 14 & 5 & 8 & 7 \\ 2 & 12 & 6 & 5 \\ 7 & 8 & 3 & 9 \\ 2 & 4 & 6 & 10 \end{bmatrix}$$

Стъпка 1. Определяме минималния елемент във всеки ред и го изваждаме от всички елементи на матрицата, които се намират в съответния ред:

$$\begin{bmatrix} 14 & 5 & 8 & 7 \\ 2 & 12 & 6 & 5 \\ 7 & 8 & 3 & 9 \\ 2 & 4 & 6 & 10 \end{bmatrix} \begin{matrix} 5 \\ 2 \\ 3 \\ 2 \end{matrix} \longrightarrow \begin{bmatrix} 9 & 0 & 3 & 2 \\ 0 & 10 & 4 & 3 \\ 4 & 5 & 0 & 6 \\ 0 & 2 & 4 & 8 \end{bmatrix}$$

Определяме минималния елемент във всеки стълб и го изваждаме от всички елементи на матрицата, които се намират в съответния стълб:

$$\begin{bmatrix} 9 & 0 & 3 & 2 \\ 0 & 10 & 4 & 3 \\ 4 & 5 & 0 & 6 \\ 0 & 2 & 4 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 9 & 0 & 3 & 0 \\ 0 & 10 & 4 & 1 \\ 4 & 5 & 0 & 4 \\ 0 & 2 & 4 & 6 \end{bmatrix}$$

$0 \quad 0 \quad 0 \quad 2$

Стъпка 2. В матрицата с редуцираните цени първият ред и първият стълб съдържат по две нули. Като зачертаем първия ред и първия стълб с две линии, остава незачертана само нулата в третия ред и третия стълб. Избираме линията, зачертаваща третия ред:

$$\begin{bmatrix} \cancel{9} & \cancel{0} & \cancel{3} & \cancel{0} \\ 0 & 10 & 4 & 1 \\ \cancel{4} & \cancel{5} & \cancel{0} & \cancel{4} \\ 0 & 2 & 4 & 6 \end{bmatrix}$$

Стъпка 3. Най-малкият незачертан елемент е равен на 1. Изваждаме 1 от всеки незачертан елемент и прибавяме 1 към всеки елемент на матрицата с редуцираните цени, който е зачертан два пъти. Получаваме матрица, в която с 4 линии са зачертани всички редове и стълбове с нули:

$$\begin{bmatrix} 1 & 0 & 0 & 3 & 0 \\ 0 & 9 & 3 & 0 & \\ 5 & 5 & 0 & 4 & \\ 0 & 1 & 3 & 5 & \end{bmatrix}$$

Понеже броят на линиите е 4 (равен на размера на матрицата), намерено е оптимално решение. Кое е решението?

Във втория и третия стълб има само по една нула. Това дава  $x_{12} = 1$  и  $x_{33} = 1$ . Така втората нула в първия ред (и четвъртия стълб) не може да се използва. Сега вече остава  $x_{24} = 1$ , което прави нулата във втория ред и първия стълб неизползваема. Остава  $x_{41} = 1$  (това е единствената нула в четвъртия ред).

Решението е  $x_{12} = x_{24} = x_{33} = x_{41} = 1$ .  
Всички други  $x_{ij}$  са 0.