Algoritmy pro práci s vektory a s maticemi

- **Algoritmy pro práci s vektory**
  - Uspořádanou n-tici reálných čísel a = (a_1, a_2, a_3, …, a_n) nazýváme n-rozměrným reálným (aritmetickým) vektorem.

Pořadí	1.	2.	3.	4.	. . .	n.
Hodnota	a_1	a_2	a_3	a_4	. . .	a_n
Index	0	1	2	3	. . .	n-1

  - Hodnota a_i, i= 1,2,3,…,n se nazývá i-tá složka vektoru **a**.
  - **Vztah polí a vektorů**
    - Datový typ pole má v programovacích jazycích obecnější použití nežli jen ve významu vektoru.
    - Skutečný význam použitého pole závisí na povaze úlohy a interpretaci.
    - Pole znaků takto například můžeme považovat za textový řetězec.
    - Pole čísel můžeme interpretovat jako:
      - obecnou posloupnost,
      - vektor popisující vztahy v N-rozměrném prostoru.
    - **Operace nad polem:**
      - přiřazení hodnoty určitému prvku, který je zadán indexem,
      - přiřazení stejné hodnoty všem prvkům,
      - vyhledání prvku s určitou hodnotou,
      - seřazení prvků podle relace uspořádání (sestupně, vzestupně),
      - operace pro jednotlivé prvky pole odpovídající typu prvku,
      - vložení prvku(ů) mezi jiné prvky, vyjmutí prvku + odpovídající posun prvků.
    - **Operace nad vektory:**
      - násobení vektoru konstantou,
      - skalární součin vektorů,
      - vektorový součin vektorů,
      - součet vektorů,
      - rozdíl vektorů atd.
        - In situ násobení vektoru konstantou

Násobení vektoru konstantou.

Pro násobení vektoru konstantou platí vztah: $K\vec{a} = \vec{b}$

Hodnoty jednotlivých prvků nového vektoru pak vypočítáme podle vztahu:

$b_i = K.a_i$, i = 1, 2, …, n

Příklad: varianta s vektorem obecné délky. Výsledkem je modifikovaný vektor.

```
void multConst(int v[], int n, int k)
{
  for(int i = 0; i < n; i++)
  {
    v[i] *= k;
  }
}
```

Příklad: varianta s vektorem obecné délky, který je uložen ve struktuře a alokován dynamicky. Výsledkem bude nově alokovaný vektor.

```
typedef struct tvector
{
  int n;
  int *v;
} TVector;
TVector multConst(const TVector *v, int c)
{
  TVector w = allocVect(v->n); // alokuje potřebnou paměť
  for(int i = 0; i < w.n; i++)
  {
    w.v[i] = v->v[i] * c;
  }
  return w;
}
```

- **Operace nad poli**
  - Hledání prvočísel <2,N>
    - Postupně se berou všechna čísla X počínaje 2 a konče N a tato se dělí všemi čísly od 2 do odmocniny z X a zjišťují se zbytky po dělení.
    - Je- li některý zbytek roven nule, pak číslo není prvočíslo v opačném případě je prvočíslo.
    - Mohou se použít různé optimalizace (např. nedělíme číslem 2, případně dělíme jen prvočísly do odmocniny z X).
    - Efektivnější způsob ➔ **Eratostenovo síto**
    - **Eratostenovo síto**
      - Pomocí pole:
        - 1. vytvoříme (bitové) pole tak, že pro každé přirozené číslo od 2 do N vyhradíme jeden prvek pole (bit), N je horní hranice výpisu prvočísel (N-1).
        - 2. index pole bude uvádět číslo, hodnota prvku pole rovna 1 bude znamenat, že číslo je prvočíslo, 0, že není prvočíslo.
        - 3. na začátku pole inicializujeme tak, jako by všechna čísla byla prvočísly (nastavíme na 1)

Pole po inicializaci 1 - značí je prvočíslo, 0 - značí není prvočíslo

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

        - 4. procházíme prvky pole a vyhledáváme první nenulový bit (v prvním průchodu to bude u indexu 2, dále 3, 5, 7 atd.).
        - 5. nalezené číslo (P) je prvočíslo, proto na jeho místě necháme v poli jedničku.
        - 6. nyní bereme všechny prvočíselné násobky P počínaje P (PxP, Px(první prvočíslo za P z předchozího kroku), Px(druhé prvočíslo za P z předchozího kroku), atd.) a na místa těchto čísel ukládáme nuly (zcela jistě to nejsou prvočísla).

```
#include <stdio.h>
#include <stdbool.h>
#define N 20
int main(void)
{
  bool a[N];
  for(int i=2; i<N; i++)
    a[i] = true;
  for(int i=2; i<N; i++)
  { // hledani prvocisel
    if (a[i])
      for(int j=i*i; j<N; j+=i)
        a[j] = false;
  }
```

- Reverze řetězce

0	1	2	3	4	5	6	7	8	9
'D'	'o'	'b'	'r'	'y'	' '	'd'	'e'	'n'	'\0'

0	1	2	3	4	5	6	7	8	9
'n'	'e'	'd'	' '	'y'	'r'	'b'	'o'	'D'	'\0'

Příklad: reverze řetězce - záměnou prvního a posledního znaku v řetězci.

```
#include <string.h>
inline void swap(char *a, char *b)
{// nonekvivalence a přiřazení - XOR
  *a ^= *b;
  *b ^= *a;
  *a ^= *b;
}
void revert(char *s)
{
  int length = strlen(s);
  for(int i = 0; i < length/2; i++)
    swap(&s[i], &s[length-i-1]);
}
```

a	0	0	1	0	0	0	1	0
b	0	1	0	1	0	1	1	0
a	0	1	1	1	0	1	0	0
b	0	0	1	0	0	0	1	0
a	0	1	0	1	0	1	1	0

Poznámka:
inline - funkční specifikátor paměťové třídy,
Specifikátor inline není příkaz, ale požadavek pro překladač, aby přeložená funkce byla co nejrychlejší.

- **Algoritmy pro práci s maticemi**
  - Soubor čísel uspořádaných do m řádků a n sloupců nazýváme maticí typu (m, n).

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = \left(a_{ij}\right)_{\substack{i=1,2,\dots,m \\ j=1,2,\dots,n}}$$

  - Čísla a_{ij} se nazývají prvky matice.
  - Matice ukládáme do dvourozměrného pole.

Poznámka: nadále budeme pracovat se statickými dvojrozměrnými poli.

```
#define R 10
#define S 20
int matice1[R][S]; // obdelnikova matice
int matice2[R][R]; // ctvercova matice
```

Příklad: tisk prvků na hlavní diagonále.

```
void printDiagonal(int a[R][R])
{
  for(int i = 0; i < R; i++)
    printf("%d ", a[i][i]);
  printf("\n");
}
```

- *Příklad*: tisk prvků na vedlejší diagonále.

```
void printSDiagonal(int a[R][R])
{
  for(int i = 0; i < R; i++)
    printf("%d ", a[i][R-(i+1)]);

  printf("\n");
}
```

- *Příklad*: tisk prvků horní trojúhelníkové matice.

```
void printUpTriangle(int a[R][R])
{
  for(int i = 0; i < R; i++)
  {
    for(int j = i; j < R; j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }
}
```

- *Příklad*: jsou si matice rovné? $j=1,2,\ldots,n$

```
bool areEqual(int a[R][S], int b[R][S])
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      if (a[r][s] != b[r][s])
        return false;

  return true;
}
```

-

Příklad: sečte dvě matice a výsledek uloží do pole a.

```
void addMatrix(int a[R][S], int b[R][S])
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      a[r][s] += b[r][s];
}
```

- *Příklad*: vynásobení matice skalárem.

```
void multConst(int a[R][S], int k)
{
  for(int r = 0; r < R; r++)
    for(int s = 0; s < S; s++)
      a[r][s] *= k;
}
```

- *Příklad*: součin matic.

```
void multMatrix(int a[M][P],int b[P][N],int c[M][N])
{
  for (int i = 0; i < M; i++)
  {
    for (int j = 0; j < N; j++)
    {
      c[i][j] = 0;
      for (int k = 0; k < P; k++)
        c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```