Algoritmy řazení

- **byla navržena celá řada metod, výběr metody závisí na množství řazených prvků,**
- **elementární algoritmy jsou použitelné k řazení desítek, popř. stovek prvků.**

<u>Předpoklady pro řazení</u> :

Řadíme neprázdnou posloupnost záznamů (pole záznamů) s následující strukturou :

```
type prvek = record
                Klíč : integer;
                Data : typ_dat;   } datová část
                  …
             end;

type  pole_prvku = array[1..100] of char;
```

Datová část může být jednoduchý nebo strukturovaný typ, v dalších algoritmech nebude tato část uvažována.

U všech uvedených metod budeme předpokládat řazení od nejmenšího do největšího prvku.

Rozdělení algoritmů řazení podle :

- **způsobu uložení řazené posloupnosti**
  - **vnitřní řazení (řazení v poli) - předpokládáme uložení řazené posloupnosti ve vnitřní paměti s přímým přístupem**
  - **vnější řazení - prvky jsou uloženy v paměti se sekvenčním přístupem (řazení v souborech)**

- **způsobu využití klíčů**
  - **adresní řazení (klíče je použito pro výpočet umístění prvku ve výsledné posloupnosti)**
  - **řazení, kdy klíče jsou používány pro porovnání vzájemného pořadí prvků.**

- **časové a paměťové náročnosti**
  - **paměťová složitost je u většiny algoritmů řazení řádu *N*, tj. pro svoji činnost buď nepotřebují žádnou další paměť nebo požadují nejvýše N dalších paměťových míst pro uložení ukazatelů na řazené prvky.**
  - **časová složitost se u většiny uvedených algoritmů pohybuje v rozmezí $N*log\ N$ až N^2.**

- **Stability**
  - **stabilní - nemění původní vzájemné pořadí prvků se stejnými klíči**
  - **nestabilní - připouštějí možnost relativní změny pořadí prvků se stejnými klíči. Nestabilní algoritmy mohou způsobit problémy tehdy, řadíme-li údaje**

podle několika klíčů. Řazení dle sekundárního klíče pak může porušit pořadí určené primárními klíči.

Řazení výběrem mezního prvku

Princip: Nalezne se nejmenší prvek a zamění se s prvním prvkem, pak se vyhledá další minimum, zamění se s druhým prvkem atd.

Př.

44 55 12 42 94 6 18 67

1. průchod **44 55 12 42 94 6 18 67**

2. průchod **6 55 12 42 94 44 18 67**

3. průchod **6 12 55 42 94 44 18 67**

4. průchod **6 12 18 42 94 44 55 67**

atd.

Procedura pro řazení - procedure SelectSort - viz program *razení.pas.*

```
procedure selectsort(var Z: POLE_PRVKU; n : integer);
var i,j, min : integer;
        T  : prvek;
begin
 for i:= 1 to N-1 do
  begin
    min:=i;
    for j:=i+1 to N do
     if Z[j].klic < Z[min].klic then min := j;
     T := Z[min]; Z[min]:=Z[i]; Z[i]:=T;
  end;
end;
```

Řazení přímým vkládáním

Tato metoda připomíná způsob, jakým si karetní hráči řadí karty v ruce : z balíčku rozdaných karet berou jednu kartu po druhé a zařadí ji na správní místo podle velikosti a barvy. Je-li třeba, odsunou doprava karty, které zařadili již dříve.

Algoritmus řazení pole :

- **první prvek pole ponecháme na místě**
- **Vezmeme druhý prvek a porovnáme jej s prvním. Je-li větší ponecháme jej na místě, jinak jej zařadíme na první místo a prvek z prvního místa odsuneme na druhé místo.**
- **Vezmeme třetí prvek a snažíme se ho zařadit na první druhé popř. třetí místo. Ostatní prvky podle potřeby odsuneme**

Př.

Vložení prvku na správné místo : v i-tém kroku porovnáváme prvek s levým sousedem pokud je i-tý prvek menší zaměníme je a pokračujeme směrem vlevo až do dosažení začátku pole - (použití zarážky).

Procedura pro řazení - procedure Insertsort – viz program *Razení.pas*

```
procedure insertsort(var Z: POLE_PRVKU; n : integer);
 var I,J : integer;
      T : prvek;
begin
 for i:=2 to N do begin
  Z[0].klic:=Z[i].klic;
  T:=Z[i];
  j:=i;
  while Z[j-1].klic > T.klic do
   begin
    Z[j]:=Z[j-1];
    j:= j-1;
   end;
  Z[j] := T;
 end;
end;
```

Binární vkládání

Jedná se o vylepšení přímého vkládání. Zvolený prvek ukládáme do posloupnosti, která je seřazená a vkládání prku tedy lze urychlit pokud urychlíme vyhledávání míst, kam prvek patří. Zarážka v tomto případě není potřebná.

Místo pro vkládaný prvek se v tomto případě zjišťujeme metodou binárního vyhledávání (viz dále).

Procedura pro řazení – procedure bininsertsort – viz program *Razeni.pas* .

```
procedure bininsertsort(var Z: POLE_PRVKU; n : integer);
var I,J,K,L,R,M : integer;
            T : prvek;

begin
 for i:=2 to n do begin
  T:=Z[i];
  l:=1; r:=i-1;
  while l<=r do begin
     m:=(l+r) div 2;
     if T.klic < Z[m].klic then r:=m-1
     else l:=m+1;
   end; {while}
  for j:=i-1 downto l do Z[j+1]:=Z[j];
  Z[l]:=T;
  end { for }
 end;
```

Bublinkové třídění
(Bubble sort)

Princip metody: Porovnáváme dva sousední prvky (prvek s indexem *i* a prvek *i+1*) a pokud jsou nesprávně uspořádány, zaměníme je a pokračujeme s porovnáním následujících dvou prvků (*i* se zvětší o 1)

1. průchod

44 55 12 42 94 18 6 67

krok 1. **44 55 12 42 94 18 6 67**

? (44 < 55) - OK

krok 2. **44 55 12 42 94 18 6 67**

? (55 <12) - NE - Záměna

krok 3. **44 12 55 42 94 18 6 67**

? (55 < 42) - NE - Záměna

44 12 42 55 94 18 6 67

… atd.

Situace na konci 1. Průchod

44 12 42 55 18 6 | **67**

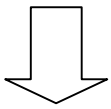
prvek s max. hodnotou probublal na konec

Při dalším průchodu pole zkrátíme a postupujeme stejným způsobem od začátku pole. Po N-1 průchodech dostaneme seřazené pole.

Procedura pro řazení – procedure bubble_sort1 – viz program *Razeni.pas* .

```
procedure bubble_sort1(var Z: POLE_PRVKU; n : integer);
var   i,j : integer;
      T : prvek;
begin
for i:=N downto 2 do
 for j:=1 to i-1 do
  if Z[j].klic>Z[j+1].klic then begin
                                t:=z[j];
                                z[j]:=z[j+1];
                                z[j+1] := t;
                               end;
end;
```

Nevýhoda tohoto postupu – procedura provádí N-1 průchodů polem i v případě, že je pole seřazené

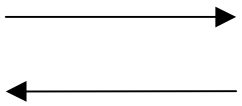


Modifikace algoritmu - doplnění testů, které ukončí proceduru v případě, že pole už je uspořádané – modifikovaná procedura – viz procedure bubble_sort2

```
procedure bubble_sort2(var Z: POLE_PRVKU; n : integer);
var   i,j : integer;
       T : prvek;
   zamena: boolean;
begin
i:=N;
repeat
 zamena:=false;
 for j:=1 to i-1 do
  if Z[j].klic>Z[j+1].klic then begin
                                t:=z[j];
                                z[j]:=z[j+1];
                                z[j+1] := t;
                                zamena:=true;
                               end;
i:=i-1;
until not  zamena;
end;
```

Třídění přetřásáním
(Shake sort)

5 7 12 18 45 1



5 průchodů

1 průchod

Pokud na předchozí pole použijeme metodu bubble sort a postupujeme od nižších indexů k vyšším je k seřazení pole potřeba maximální počet průchodů (tj. v tomto případě 5). Pokud však postupujeme od vyšších indexů k nižším postačí jediný průchod.

Na tomto principu je založena metoda *shake sort* – tj. střídavě s prochází polem od nižších indexů k vyšším a od vyšších indexů k nižším. Po každém průchodu se pole zkrátí vždy o jeden prvek (o prvek který „probublá“ na správné místo).

Procedura pro řazení – procedure shake_sort – viz program *Razeni.pas* .

```
procedure shake_sort(var Z: POLE_PRVKU; n : integer);
 var i,j,k,l,r: integer;
     t: prvek;
begin
 l:=2; r:=n; k:=n;
 repeat
  for j:=r downto l do { Prohlizi od vyssich indexu k nizsim }
   if Z[j-1].klic>Z[j].klic then begin
                                t:=z[j-1];
                                z[j-1]:=z[j];
                                z[j] := t;
                                k:=j;
                                end;
  l:=k+1; { Uschovani indexu posledni zameny }
  for i:=l to r do { Prohlizi od nizsich indexu k vyssim }
   if Z[i-1].klic>Z[i].klic then begin
                                t:=z[i-1];
                                z[i-1]:=z[i];
                                z[i] := t;
                                k:=i;
                                end;
   r:= k-1;      { Uschovani indexu posledni zameny }
 until l>r;
end;
```

Rychlé třídění (quicksort)

Myšlenka algoritmu vychází ze skutečnosti, že nejefektivnější jsou výměny prvků v poli na velké vzdálenosti. Např. vezmeme-li pole s N prvky, seřazenými v obráceném pořadí, můžeme jej setřídit pomocí N/2 výměn. (Porovnáme prvky na obou koncích pole a zaměníme; pak postoupíme o jeden prvek směrem ke středu a opakujeme porovnání) . Na podobném principu pracuje i algoritmus *quicksort.*

Princip algoritmu:

- **Zvolíme náhodně prvek *x* a uspořádáme pole tak, aby vlevo od *x* ležely prvky s klíči menšími než je klíč *x* a vpravo prvky s klíči většími než je klíč *x.* Výsledkem je pole které se skládá ze tří useků *a[1], … , a[s-1], a[s] , a[s+1], … a[n]*, kde a[s]=x**
- **Proces rozdělení opakujeme pro úseky *a[1], … , a[s]*, a *a[s+1], … , a[n]* a pro jejich části až dospějeme k úsekům délky o délce 1.**

Procedura pro řazení – procedure quicksort – viz program *Razeni.pas* .

```
procedure  quicksort(var Z: POLE_PRVKU; l,r : integer);
 var i,j : integer;
     x,w : PRVEK;
 begin
  i:=l;
  j:=r;
  x:=z[(l+r) div 2 ];
    repeat
     while z[i].klic < x.klic do i:= i+1;
     while z[j].klic > x.klic do j:= j-1;
     if i<=j then begin
      w:=z[i]; z[i]:=z[j]; z[j]:=w;
      end;
    until i >= j;
  if l<j then quicksort(z,l,j-1);
  if i<r then quicksort(z,i+1,r);
  end;
```