Digitální učební materiál

Číslo projektu:	CZ.1.07/1.5.00/34.0548
Název školy:	Gymnázium, Trutnov, Jiráskovo náměstí 325
Název materiálu:	VY_32_INOVACE_152_IVT
Autor:	Ing. Pavel Bezděk
Tematický okruh:	Algoritmy
Datum tvorby:	srpen 2013
Ročník:	4. ročník a oktáva
Anotace:	Algoritmus XII. – Třídění dat IV. - Quick Sort
Metodický pokyn:	Při výuce nutno postupovat individuálně. Části DUM – „ Pro hloubavé“ jsou určeny pro zájemce o studium na technických a matematicko-fyzikálních oborech vysokých škol.

esf evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Autor	**Ing. Pavel Bezděk**		
Vytvořeno dne	**20. 8. 2013**		
Odpilotováno dne	**24. 2. 2014**	**ve třídě**	**8.Y**
Vzdělávací oblast	**Informatika a informační a komunikační technologie**		
Vzdělávací obor	**Informatika a výpočetní technika**		
Tematický okruh	**Algoritmus**		
Téma	**Algoritmus XII. - Třídění dat IV. - Quick Sort**		
Klíčová slova	**Algoritmus, Quick Sort, rekurze**		

Třídění dat

Quick Sort

Rekurzivní metody třídění

Rekurzivní metody

- složitější zápis programu
- technika „**Rozděl a panuj**“ - divide et impera (latinsky)
  - divide and conquer (anglicky)

Metoda rekurzivního návrhu algoritmu (programu)

problém se rozdělí na dva (příp. více) podproblémy stejného typu, ale menší velikosti, z jejich řešení se pak snadno sestaví řešení původního problému. **Každý podproblém je, buď už triviální a vyřeší se přímo, nebo se k jeho řešení použije stejný rekurzivní postup.**

Realizace: rekurzivní procedurou

Podmínkou rozumné (tj. efektivní) použitelnosti metody:

podproblémy vznikající rozkladem jsou na sobě nezávislé
(nevyužívají stejné dílčí podúlohy a jejich řešení)

Jednoduchý program na rekurzi

```
procedure Obratit;
var znak: char;
begin
read(znak);
if znak <>  , ,   then Obratit;          {, ,  mezera}
write(znak)
end;
```

{ Vstup: DNES_ Výstup: _SEND }

**{Obratit (read(D) Obratit (read(N) Obratit (read(E)
Obratit (read(S) Obratit (read(, ,) write(, ,)) write(S))
write(E)) write(N)) write(D))}**

Nevhodné použití rekurze

Ne vždy je použití rekurze výhodné.

1. Dokážeme řešit úlohu lépe bez rekurze, která by zvyšovala časové a paměťové nároky programu.

2. Úloha nevede přirozeným způsobem k rekurzivnímu postupu a umělým použitím rekurze pak docílíme jen komplikované a nepřehledné řešení.

Typickým příkladem nevhodného použití rekurze je Fibonacciho posloupnost.

Rekurze se zde sice nabízí, ale je velmi neefektivní. Má exponenciální časovou složitost, a proto je prakticky nepoužitelná. Jsme schopni vypočítat v reálném čase max. 23. člen posloupnosti (28657).

Bez rekurze lze vymyslet efektivní algoritmus s lineární časovou složitostí. Vypočítáme v reálném čase max. 23601. člen posloupnosti ($9{,}285654 \times 10^{4931}$) .

Fibonacciho posloupnost

s použitím rekurze

Pascal

```
Program FibonacciSequence ;
{Fibonacci s rekurzi}
uses CRT;
function FibonacciNumber( n : integer ): integer;
begin
 if n=0 then begin FibonacciNumber:=0; writeln('  ',FibonacciNumber); end
       else if n=1 then begin FibonacciNumber:=1;  writeln('  ',FibonacciNumber); end
             else begin FibonacciNumber:= ( FibonacciNumber( n - 1 ) + FibonacciNumber( n - 2 ) );
                        writeln('  ',FibonacciNumber); end
         end;
var Fib,number:integer;
begin
clrscr;
write('     Zadej poradi (do 23.) v Fibonacciho posloupnosti: ');
Readln(number);
Fib:=FibonacciNumber(number);
Writeln('     Fib(',number,') = ',Fib);
repeat until keypressed;
end.
```

Fibonacciho posloupnost

s použitím rekurze

C++

```
#include <iostream>
using namespace std;
int fib (int n);
int main()
{           int n, odpoved;
            cout << "  Vlozte poradi  (do 23. ) v posloupnosti: ";
            cin >> n;
            cout << "\n\n";
            odpoved = fib(n);
            cout << "  Fib("<<n << ") = "<<odpoved<< endl;
            cout << "  Fibonacciho cislo je " << odpoved << "\n";
            return 0;
}
```

```
int fib (int n)
{
        cout << "  Zpracovani fib(" << n << ")... ";

        if (n < 3 )
        {
                cout << "  Vraceni 1!\n";
                return (1);
        }
        else
        {
                cout << "  Volani fib(" << n-2 << ") ";
                cout << "a fib(" << n-1 << ").\n";
                return( fib(n-2) + fib(n-1));
        }
}
```

Fibonacciho posloupnost

bez rekurze

Pascal

```
Program FibonacciSequence ; {Fibonacci bez rekurze}
uses CRT;
function FibonacciNumber( n : integer ): extended;
var i:integer;
   a,b,c: extended;
begin
 if n=0 then begin FibonacciNumber:=0; writeln('  ',FibonacciNumber); end
          else   begin i:=1; a:=1; b:=0;
               while i<n do
                        begin i:=i+1; c:=b; b:=a; a:=b+c; writeln('  ',a); end;
               FibonacciNumber:=a;
                 end
end;
var number:integer; Fib:extended;
begin
clrscr;
write('     Zadej poradi (do 23601.) v Fibonacciho posloupnosti: ');
      Readln(number); Fib:=FibonacciNumber(number);
Writeln('     Fib(',number,') = ',Fib);
repeat until keypressed;
end.
```

Fibonacciho posloupnost

bez rekurze

C++

```
// Fibonacciho posloupnost bez rekurze do Fib(23601)
#include <iostream>
using namespace std;
 void fib( int n);
int main()
{               int n;
   cout << "  Vlozte poradi (do 23601) v posloupnosti: ";       cin >> n;
   fib(n);    return 0;
}
void fib( int n)
{    int i;
      long double a,b,c;
   if (n <2) { if (n==0) {a=0; cout<<"Fib(0)= "<<a<<endl; }
                    else {a=1; cout<<"Fib(1)= "<<a<<endl;}
               }
                else          {i=1; a=1; b=0; c=0;
                               while (i<n)  {  i++; c=b; b=a; a=b+c;
                                                cout<<"Fib("<<i<<")= "<<a << endl; }
                               }
}
```

Třídící algoritmy používající rekurzi

Patří jsem algoritmy **QuickSort** a **MergeSort**

QuickSort třídí „**na místě**“ (nepotřebuje další pole po ukládání), **MergeSort** potřebuje **další pole velikosti N**.

Oba algoritmy navíc potřebují paměť na realizaci rekurze (tzn. systémový zásobník nebo vlastní zásobník v případě odstranění rekurzivních volání).

Časová složitost

MergeSort vždy **O(N.logN)**

QuickSort **O(N.logN)** **(ale v nejhorším případě** **$O(N^2)$** **)**

Quick Sort - Rozděl a panuj

Rozdělit:

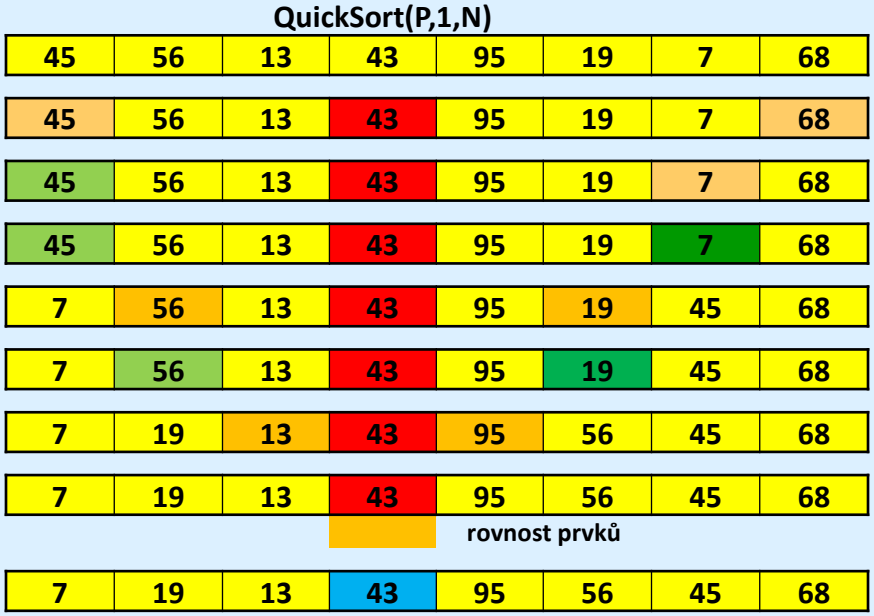
Jestliže má sekvence **S** více než **dva** prvky (pokud má **0** nebo **jeden** prvek, je již seřazená), vyber prvek ***x*** z **S**. Prvek ***x*** se v tomto případě nazývá **pivot** a může to být **jakýkoliv libovolný prvek** z **S**. Odeber všechny prvky z **S** a rozděl je do **3** sekvencí:

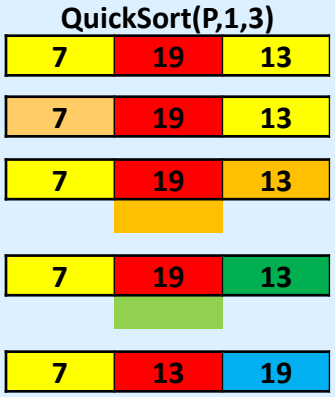
L(less - méně) v níž budou **elementy** z **S** **menší než x**
E(equal - rovný) v níž budou **elementy** z **S** **rovné x**
G(greater - větší) v níž budou **elementy** z **S** **vyšší než x**

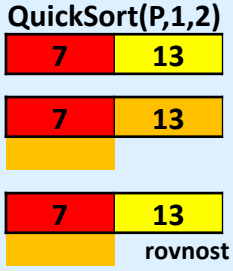
Rekurze: Rekurzivně setřiď sekvence **L** a **G.**

Panovat: Nakonec vrať elementy do sekvence **S** v pořadí, kdy první vlož elementy z **L**, poté elementy z **E** a na konec z **G**.

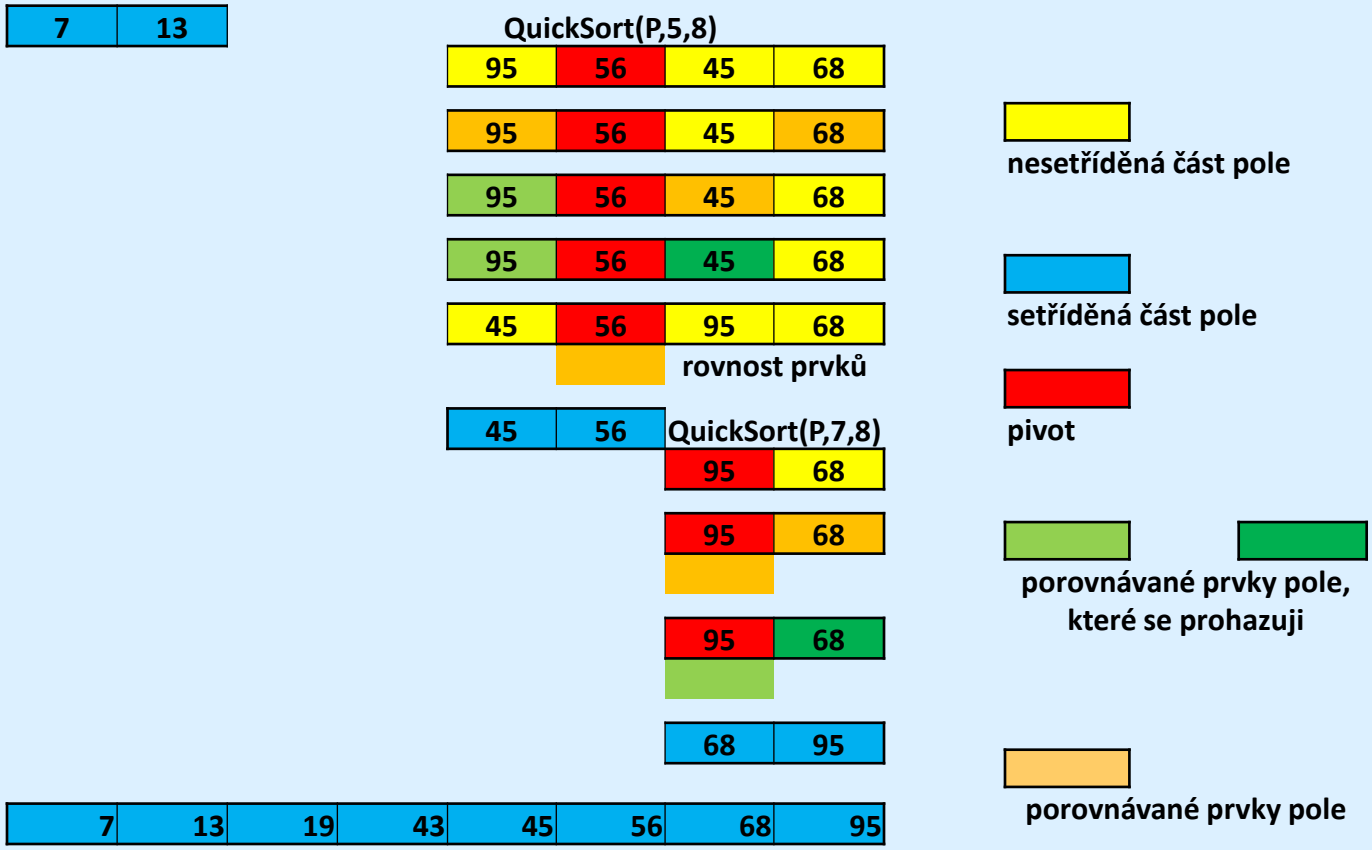
Quick Sort


QuickSort(P,1,N)
45 56 13 43 95 19 7 68
45 56 13 43 95 19 7 68
45 56 13 43 95 19 7 68
45 56 13 43 95 19 7 68
7 56 13 43 95 19 45 68
7 56 13 43 95 19 45 68
7 19 13 43 95 56 45 68
7 19 13 43 95 56 45 68
rovnost prvků
7 19 13 43 95 56 45 68


QuickSort(P,1,3)
7 19 13
7 19 13
7 19 13
7 19 13
7 13 19


QuickSort(P,1,2)
7 13
7 13
7 13
rovnost prvků


nesetříděná část pole
setříděná část pole
pivot
porovnávané prvky pole, které se prohazuji
porovnávané prvky pole


7
13
QuickSort(P,5,8)
95
56
45
68
95
56
45
68
95
56
45
68
95
56
45
68
45
56
95
68
rovnost prvků
45
56
QuickSort(P,7,8)
95
68
95
68
95
68
68
95
7
13
19
43
45
56
68
95
nesetříděná část pole
setříděná část pole
pivot
porovnávané prvky pole,
které se prohazuji
porovnávané prvky pole

Program Quick Sort
Pascal

```
program RychleTrideniQuickSort;
uses CRT;{ Unita pro praci s obrazovkou}
const MaxN = 100; {maximalni pocet tridenych cisel}
type Pole = array[1..MaxN] of integer; {tridena cisla}
var P: Pole;     {ulozeni tridenych udaju}
      i:integer;
     N: 1..MaxN;  {pocet prvku v poli P}
```

```
procedure Quicksort(var P: Pole; Zac, Kon: integer);
      {setridi v poli P usek od indexu Zac do indexu Kon}
       var k: integer;     {hodnota pro rozdeleni na useky}
       x: integer;     {pomocne pro vymenu prvku v poli}
       i, j: integer;  {posouvane pracovni indexy v poli}
   begin
    i:=Zac; j:=Kon;  k:=P[(Zac+Kon) div 2];
 {za hodnotu X vezmeme pro jednoduchost prostredni prvek ve zkoumanem useku}
        Repeat
          while P[i] < k do i:=i+1;
          while P[j] > k do j:=j-1;
          if   i < j then {vymenit prvky s indexy I a J}
                    begin
                     x:=P[i]; P[i]:=P[j]; P[j]:=x;
                     i:=i+1; j:=j-1;   {posun indexu na dalsi prvky}
                    end
                  else if i = j then {indexy i, j se sesly, oba dva ukazuji na hodnotu k}
                              begin
                               i:=i+1; j:=j-1 {posun indexu na dalsi prvky, nutne kvuli
ukonceni cyklu}
                              end
        until i > j;
{usek <Zac,Kon> je rozdelen na useky <Zac,j> a <i,Kon>, ktere zpracujeme
rekurzivnim volanim procedury:}
        if Zac < j then QuickSort(P, Zac, j);
        if i < Kon then QuickSort(P, i, Kon);
  end;  {end procedure QuickSort}
```

```
begin {hlavni program}
clrscr;  {Vymazani obrazovky}
writeln('Zadej N prvku pole, ktere chces setridit:');
writeln('Maximalne vsak ',MaxN, ' prvku.');
write('Pocet prvku:');Readln (N);
writeln('Zadej jednotlive prvky pole:');
for i:=1 to N do begin write(i,'. prvek pole:'); readln(P[i]);end;
writeln;
writeln;

QuickSort(P, 1, N);                  { Volani procedury}
writeln('Setridene pole:');
writeln;
for i:=1 to N do   write('  ',P[i]);
repeat until keypressed;

end.
```

Quick Sort s rekurzí C++

```
include <iostream> //Quick Sort rekurze
using namespace std;
const int POCET=10;
void quick(int p[], int n); //prototyp quick
int main()
{
        int pole[POCET];                    // vstupní pole pro trideni
   int pocet=POCET;                         // vstupni parametr - pocet prvku
        int k;
        cout<<"  Zadej prvky pole ke trideni: "<<endl;
        for (k=0; k<pocet; k++)     {cout<<"   pole["<<k<<"]= "; cin>>pole[k];}
   cout<<endl;
        quick(pole,pocet);
        cout<<"  Setridene pole (quicksort): "<<endl;
        for (k=0; k<pocet; k++) {cout<<"  "<<pole[k];}
   cout<<endl;
                return 0;
}
```

```
void quick(int p[], int n)      // trideni rozdelovanim – quicksort
//   prvek x je vybran ze stredu tridene posloupnosti,  kterou rozdelime
// na dve podmnoziny;   v prvni jsou  prvky mensi nez x, ve druhe vetsi;
// vznikle  useky tridime rekurzivne stejnym zpusobem.
{
            void quickc(int p[], int l, int r);
   /* tridi usek od indexu l po r */
            quickc(p,0,n-1);
}

void quickc(int p[], int l, int r)
{
int i,j,x,y;

i=l; j=r;
x=p[(l+r)/2];
```

```
do
{
  while(p[i]<x && i<r)i++;
  while(p[j]>x && j>l)j--;
  if(i<=j)
  {
    if(i<j)
                {
      y=p[i]; p[i]=p[j]; p[j]=y;
                }
    i++; j--;
  }
}
while(i<=j);
if(l<j) quickc(p,l,j);
if(i<r) quickc(p,i,r);
}
```

Paměťová a časová složitost Quick Sort

Paměťová složitost: O(N) třídění probíhá na místě v poli, navíc je ale třeba paměť na realizaci rekurzivních volání (zásobník)

Časová složitost: nejhorší případ - za X vždy vybereme nejmenší nebo největší prvek v úseku, - postupně procházíme úseky délky N, N-1, N-2, …,2
celkem tedy práce N + (N-1) + (N-2) + … + 1 =N.(N+1)/2 … **$O(N^2)$**

nejlepší případ - za X vždy vybereme prostřední hodnotu (medián) v úseku
- procházíme 1 úsek délky N, 2 úseky délky N/2, 4 úseky délky N/4, …,
celkem hloubka rekurze logN a na každé hladině rekurze práce N
(součet délek K úseků, každý dlouhý N/K prvků) … **O(N.logN)**

průměrný případ - lze dokázat, že **O(N.logN)** jako v nejlepším případě

Pro hloubavé Quick Sort bez rekurze!

QuickSort - implementace bez rekurzivní procedury

- **použití rekurzivní procedury lze nahradit vlastním zásobníkem** - zásobník „dluhů“ = seznam úseků, které je ještě třeba dotřídit - místo rekurzivního volání → vložení úseku do zásobníku
- v cyklu se postupně vybírají ze zásobníku jednotlivé dluhy a řeší se (čímž zase vznikají nové, menší dluhy)
- pro programátora více práce při psaní programu - výsledný kód může být úspornější
  - na čas (úspora za režii rekurzivních volání)
  - na paměť (při dobré organizaci práce stačí zásobník logaritmické výšky)

```
Program QuikSort_bez_rekurze;
const MaxN = 100;        {maximální počet tříděných čísel}
MaxNdiv2 = 50;  {= MaxN div 2 (velikost zásobníku)}
type Pole = array[1..MaxN] of integer;   {tříděná čísla}
var P: Pole;        {uložení tříděných údajů}
N: 1..MaxN;    {počet prvků v poli P}
Zasob: array[1..MaxNdiv2] of
record Zac, Kon: integer end;   {zásobník úseků  čekajících na zpracování}
Vrchol: 0..MaxN;                    {vrchol zásobníku}
I: integer;
```

```
procedure Quicksort2(var P:Pole; Zac,Kon:integer);
                                {setridi v poli P usek od indexu Zac do indexu Kon}
var X: integer;                  {hodnota pro rozdeleni na useky}
Q: integer;                      {pomocne pro vymenu prvku v poli}
 Z,K: integer;                   {zacatek a konec zkoumaneho useku}
 I,J: integer;                   {posouvane pracovni indexy v poli}
begin  Zasob[1].Zac:=Zac;   Zasob[1].Kon:=Kon;  Vrchol:=1;
                                   {cely trideny usek vlozen do zasobniku}
 while Vrchol>0 do                  {zasobnik je neprazdny}
 begin   Z:=Zasob[Vrchol].Zac;    K:=Zasob[Vrchol].Kon;   Vrchol:=Vrchol-1;
                                   {odebran jeden usek ze zasobniku}
 I:=Z; J:=K;    X:=P[(I+J) div 2];    {za hodnotu X vezmeme pro jednoduchost
prostredni prvek ve zkoumanem useku}
repeat
  while P[I] < X do I:=I+1;
  while P[J] > X do J:=J-1;
  if I < J then
            begin  Q:=P[I]; P[I]:=P[J]; P[J]:=Q; {vymenit prvky s indexy I a J}
                   I:=I+1; J:=J-1;               {posun indexu na dalsi prvky}
            end
         else if I = J then   {indexy I a J se sesly, oba dva ukazuji na hodnotu X}
                    begin       I:=I+1; J:=J-1
                   {posun indexu na dalsi prvky - nutne kvuli ukonceni cyklu}
                    end
 until I > J;
```

```
{usek <Z,K> je rozdelen na useky <Z,J> a <I,K>,
                    ktere vlozime do zasobniku k dalsimu zpracovani:}
  if Z < J then
      begin
      Vrchol:=Vrchol+1;      Zasob[Vrchol].Zac:=Z;      Zasob[Vrchol].Kon:=J
      end;
  if I < K then
     begin
     Vrchol:=Vrchol+1;      Zasob[Vrchol].Zac:=I;      Zasob[Vrchol].Kon:=K
     end
end        {end while Vrchol>0 }
end;       {procedure Quicksort2}


begin  {hlavni program}
write('Pocet tridenych cisel: ');
readln(N);writeln('Zadani posloupnosti tridenych cisel:');
for I:=1 to N do read(P[I]);
Quicksort2(P,1,N);
writeln('Setridena cisla:');
for I:=1 to N do write(P[I]:5);
writeln
end.
```

Quick Sort bez rekurze C++

```
#include <iostream> //Quick Sort bez rekurze
using namespace std;
const int POCET=10;
void quicksnr(int p[], int n);
int main()
{
        int pole[POCET];                        // vstupní pole pro trideni
        int pocet=POCET;                        // vstupni parametr - pocet prvku
        int k;
        cout<<"  Zadej prvky pole ke trideni: "<<endl;
        for (k=0; k<pocet; k++)         {cout<<"   pole["<<k<<"]= "; cin>>pole[k];}
   cout<<endl;
        quicksnr(pole,pocet);
        cout<<"  Setridene pole (quicksort nerekurzivni): "<<endl;
        for (k=0; k<pocet; k++){cout<<"   "<<pole[k];}
   cout<<endl;
        return 0;
}
```

```
void quicksnr(int p[], int n)  // quicksort - nerekurzivni verze
// velikost zasobniku pro uchovavani useku je rovna dvojkovému  logaritmu
// z velikosti trideneho pole. Je ovsem nutno uchovávat  vetsi useky.
{
int zac[12],kon[12];   // zasobniky pro oznaceni zacatku a koncu jednotl. useku
int i,j,l,r;
int x,y;
int uk;                    // ukazovatko v zasobnicich
uk=0; zac[uk]=0; kon[uk]=n-1;
do {
        l=zac[uk]; r=kon[uk];
        uk--;
        i=l; j=r; x=p[(l+r)/2];
```

```
do {
                    while(p[i]<x && i<r)i++;
                    while(p[j]>x && j>l)j--;
                    if(i<=j)
                    {
                              if(i<j)
                              {y=p[i]; p[i]=p[j]; p[j]=y;}
                              i++; j--;
                    }
          } while(i<=j);
                    if ((j-l)<(r-i))
                    {   // levy usek je mensi
                    if(i<r)
                              { // uloz pravy usek do zasobniku
                                        uk++; zac[uk]=i; kon[uk]=r;
                                        r=j;
                              }
                    }
```

```
else
                        {           // pravy usek je mensi
                                    if (j>l)
                                    { // uloz levy usek do zasobniku
                                                uk++; zac[uk]=l; kon[uk]=j;
                                                l=i;
                                    }
                        }
}while (uk>=0);
}
```

Doporučené video

Quick Sort:

http://www.youtube.com/watch?v=ywWBy6J5gz8

Použité zdroje

WIRTH, Niklaus. *Algoritmy a štruktúry údajov*. 2.vyd. Bratislava: Alfa, 1989, 481 s. ISBN 80-050-0153-3.

BÖHM, Martin. *Programátorská kuchařka: Recepty z programátorské kuchařky* [online]. Praha: KSP MFF UK Praha, 2011/2012 [cit. 2013-08-20]. KSP, Korespondenční seminář z programování: Programátorské kuchařky, 24. ročník KSP. Dostupné z: http://ksp.mff.cuni.cz/tasks/24/cook1.html