Řazení

Problém řazení:

Uspořádat množinu prvků obsahujících klíč podle definovaného kriteria.

Až 30% času běžného počítače.

Příklad:

Mějme zjistit zda jsou v posloupnosti prvků, například celých čísel, duplicitní hodnoty.

Algoritmus:

```
boolean jeDuplikat(int [] a) {
  for(int i = 0; i < a.length; i++)
    for(int j = i + 1; j < a.length; j++)
       if(a[i] == a[j])
         return false;
  return true;
}
```

Časová složitost? $n\text{-}1 + n\text{-}2 + \ldots + 1$

Využití řazení

Myšlenka:

V seřazené posloupnosti stačí porovnat sousedy.

Časová složitost ? *nlogn + n-1*

Než začneme řadit

vnitřní vs. ***vnější*** řazení

algoritmus řadící ***na místě*** (paměť pro řazené prvky + konstantí počet prvků) vs. ***další paměť***

přímé vs. ***nepřímé*** řazení - vhodná je metoda prvku

```
boolean jeMensi(Prvek)
```

```
class Prvek {
  int klic;
  ...
  public boolean jeMensi(Prvek v) {
    return klic < v.klic;
  }
  ...
}
```

více klíčů

Adam	Nový
Ivan	Novák
Jan	Nový
Josef	Nováček
Karel	Nový
Pavel	Novák
Petr	Novák
Roman	Nováček

klíčem křestní jméno

Josef	Nováček
Roman	Nováček
Ivan	Novák
Pavel	Novák
Petr	Novák
Adam	Nový
Jan	Nový
Karel	Nový

klíčem příjmení

Josef	Nováček
Roman	Nováček
Pavel	Novák
Ivan	Novák
Petr	Novák
Karel	Nový
Adam	Nový
Jan	Nový

stabilní řazení

nestabilní řazení

Jak řadíme ?

- porovnáváme hodnoty klíčů (dosavadní algoritmy - ***porovnávací***)

Jsou i jiné !

Počítací algoritmus – klíče řazených prvků mají hodnoty pouze z množiny *{0, 1, ..., k}*

Zjednodušení: klíče všech řazených prvků jsou různé

Myšlenka:

Pořadí řazeného prvku v seřazené posloupnosti je určeno počtem klíčů menších nebo rovných jeho klíči.

k = 9

6, 3, 8, 5, 7, 1

6 bude v seřazené posloupnosti čtvrtý prvek

, , , *6*, , ,

Algoritmus:

prvek pomocného pole s indexem *i* je 1, obsahuje-li řazená polsoupnost klíč *i*, jinak je 0
(indexujeme od *0*, velikost pole je *k+1*)

6, 3, 8, 5, 7, 1

0	1	0	1	0	1	1	1	1	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

vypočteme součet hodnot prvků tohoto pole na indexech *0* až *i,* což je počet prvků menších nebo rovných *i*, postupně pro *i = 0,1,...,9*

0	1	1	2	2	3	4	5	6	6
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

pořadí prvku s klíčem *i* v řazené posloupnosti je v seřazené posloupnosti určeno hodnotou prvku tohoto pole s indexem *i*

6, 3, 8, 5, 7, 1

6 je čtvrtý , , , 6, ,
3 je druhý , 3, , 6, ,
8 je šestý , , , 6, , 8
...

Program:

seřaďme do prvků **b[1]** až **b[n]** prvky **a[1]** až **a[n]**,
pomocné pole **c[0]** až **c[k]**

```
for (i = 0;  i <= k;  i++) //inicializace
  c[i] = 0;                //indexováno od 0


for (j = 1;  j <= n;  j++) //c[a[j]] bude 1
  c[a[j]] = 1;             //na indexech rovných
                           //klíčům v poli a[]

for (i = 1;  i <= k;  i++) //počet prvků menších
  c[i] += c[i-1];          //nebo rovných než i


for (j = 1;  j <= n;  j++) //prvky pole a[] postupně
  b[c[a[j]]]= a[j]         //ukládáme na vypočtené
                           //indexy c[a[j]]
```

Časová složitost ?

Řazení haldou (Heapsort)

`RazeniPoleHaldou1`

1. v poli vytvoříme haldu o dvou, třech, ... prvcích - $N \log_2 N$
2. největší prvek vyměníme s posledním a obnovíme haldu o jeden prvek menší - $N \log_2 N$

Haldu můžeme vytvořit v lineárním čase!

`RazeniPoleHaldou2`

Myšlenka:

- metoda **`dolu()`** vytvoří haldu, pro podstrom s kořenem v kterémkoliv prvku stromu, pokud levý a pravý podstrom jsou haldy

- listy stromu jsou jednoprvkové haldy

- metodou **`dolu()`** postupně vytvoříme haldy od prvku **`a[pocet/2]`** až ke kořenu stromu, zespoda nahoru vytvoříme haldu nad polem s *N* prvky

```
void razeniHaldou2() {
  // vytvoříme haldu
  for (int i = pocet/2; i >= 1; i--)
    dolu(i,pocet);
  // seřadíme
  for (int i = pocet; i > 1;) {
    vymen(1,i);
    dolu(1,--i);
  }
}
```

Invariant vytvoření haldy v poli (prvního cyklus):

Na začátku každé iterace, všechny vrcholy s indexy *i + 1, i + 2, ..., pocet* jsou kořeny hald.

Inicializace: Před první iterací je `i == pocet/2`, tedy *i* je předchůdcem posledního listu - žádný z následujících vrcholů již nemá následníky, jsou tedy listy a tím kořeny jednoprvkových hald.

Udržování: Následníci vrcholu *i* mají indexy větší než *i*, jsou tedy haldami a metoda **dolu()** vytvoří haldu s kořenem i, přičemž zachová haldy s kořeny *i + 1, i + 2, ..., pocet.* Dekrementováním *i* obnovíme invariant před další iterací.

Skončení: Po skončení je `i == 0`, a tedy každý vrchol *i = 1, 2, ..., pocet* je kořenem haldy a specielně je ním vrchol s indexem 1.

- časová složitost metody `dolu()` je $O(\log_2 N)$
- počet volání je $O(N)$
- časová složitost vytvoření haldy je omezena $O(N \log_2 N)$.

Je lepší omezení!

$N = 2^n - 1$

metoda **dolu()**bude volána:

2^{n-2} krát na stromy o výšce 1
2^{n-3} krát na stromy o výšce 2
...
jednou pro kořen stromu s výškou $n - 1$

$$\sum_{k=1}^{n-1} k.2^{n-k-1} = 2^n - n - 1 < N$$

Důkaz pro $N \neq 2^n - 1$ je podobný.

Vzhledem na druhý cyklus, kde je $n - 1$ krát volána metoda **`dolu()`**, která je $O(\log_2 N)$, časová složitosost **razeniHaldou2()** je $O(N \log_2 N)$.

Shellovo řazení

Řazení vkládáním

Příklad:

Mějme posloupnost

4 2 7 6 3 9 8 5 1 0
↑

vložme postupně prvky 2, 7, ... až 5 do uspořádaných podposloupností před nimi

2 3 4 5 6 7 8 9 1 0

Prvky *0* a *1* nyní pro dosažení správné pozice musí „projít“ téměř celou posloupnost – *N* prvků.

Poznámka:
Řazení haldou – „procházíme“ $log_2 N$ prvků.

Myšlenka (Donalda Shella z roku 1959) :

Cestu prvku ke správné pozici skrátíme, seřadíme-li vkládáním prvky vzdálené *h* – prvky se posouvají ne o jedno nýbrž o *h* míst

- posloupnost je nyní organizována jako *h* podposloupností začínajících prvky na pozicích *0, 1,* ..., *h-1*, každá obsahující prvky vzdálené *h*

Pro *h* = 4

4	2	7	6	***3***	9	8	5	***1***	0
4	***2***	7	6	3	***9***	8	5	1	***0***
4	2	***7***	6	3	9	***8***	5	1	0
4	2	7	***6***	3	9	8	***5***	1	0

Podposloupnosti řadíme vkládáním

3	2	7	6	***4***	9	8	5	***1***	0
1	2	7	6	***3***	9	8	5	***4***	0
1	***2***	7	6	3	***9***	8	5	4	***0***
1	***0***	7	6	3	***2***	8	5	4	***9***
1	0	***7***	6	3	2	***8***	5	4	9
1	0	7	***5***	3	2	8	***6***	4	9

- získané posloupnosti říkáme, že je *h* (4) seřazena
- *0* a *1* jsou téměř na svých místech

- velké h, posuneme prvky na velkou vzdálenost.

- postup opakujeme pro klesající posloupnost hodnot *h* až k *1*

- pro h = 1, půjde o řazení vkládáním, ale prvky budou blízko svým konečným místům

Jaká má být posloupnost hodnot *h?*

- Shell navrhnul pro posloupnost s *N* prvky začít s *N/2* a postupně tuto hodnotu půlit dokud nedosáhne 1
- prvky na sudých pozicích nejsou porovnány s prvky na lichých pozicích až do posledního průchodu

- Knuth navrhnul zvolit začáteční vzdálenost přibližně třetinovou a postupně ji dělit třemi, přesněji, čísla tvaru $3i + 1, i = 0, 1, ...$.

1 4 13 40 121 364 1093 3280 ...

Implementace:

- pro danou vzdálenost *h* jsou jednotlivé podposloupnosti navzájem nezávislé

- v první podposloupnosti vložíme druhý prvek (s indexem *h*) na správné místo

- ve druhé podposloupnosti vložíme druhý prvek (s indexem *h+1*) na správné místo
...
- v *h*-té podposloupnosti vložíme druhý prvek na správné místo

- v první podposloupnosti vložíme třetí prvek (s indexem *2h*) na správné místo
...

4	2	7	6	3	9	8	5	1	0
3	2	7	6	**4**	9	8	5	**1**	0
3	**2**	7	6	4	**9**	8	5	1	**0**
3	2	**7**	6	4	9	**8**	5	1	0
3	2	7	**5**	4	9	8	**6**	1	0
1	2	7	5	**3**	9	8	6	**4**	0
1	**0**	7	5	3	**2**	8	6	4	**9**
1	0	7	**5**	3	2	8	**6**	4	9

- dostali jsme stejnou 4 seřazenou posloupnost

```
class RazeniShell {

  private int[] a;
  private int pocet;
  final int maxN=10;

  RazeniShell() {
    a = new int[maxN];
    pocet = 0;
  }

  void nactiPrvek(int klic) {
    a[pocet++] = klic;
  }

  void tiskPole() {
    for(int i = 0; i < pocet; i++)
      System.out.print(a[i]+" ");
    System.out.println(" ");
  }

  void razeniShell() {
    int h = 1;
    while(h <= pocet/3)
      h = 3*h + 1;
    for( ; h > 0; h /= 3)
      for (int i = h; i < pocet; i++) {
        int j = i;
        int v = a[i];
        while(j >= h && v < a[j-h]) {
          a[j] = a[j-h];
          j -= h;
        }
        a[j] = v;
      }
   }
}
```

Složitost:

- obecné řešení není dosud známo

- není ani známa dokazatelně nejlepší posloupnost vzdáleností i když některé posloupnosti v praktických aplikacích se dobře osvědčují

- Knuth zjistil, že $N^{1.25}$ je dobré omezení

Shellovo řazení je nejjednodušší efektivní řazení (i když ne nejefektivnější) a může být proto první volbou v praktických aplikacích, zejména pro rozsah řazených posloupností v řádu do desítek tisíc prvků.

Shellův algoritmus je hezkým příkladem jednoduchého algoritmu, jehož analýza je velice složitá.

Řazení dělením (Quicksort)

- algoritmus řazení dělením je pravděpodobně nejpoužívanější algoritmus řazení
- objeven C.A.R. Hoarem v roce 1960 a má mnoho verzí

```
void razeniDelenimR(int l, int p) {
    if(p <= l) return;
    int i = deleni(l,p);
    razeniDelenimR(l, i-1);
    razeniDelenimR(i+1, p);
  }
```

Myšlenka dělení:

Rozdělit prvky v poli tak, aby vlevo byly hodnoty menší a vpravo větší než dělící hodnota – **pivot**.

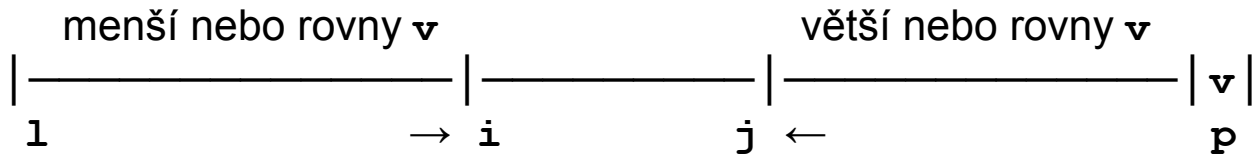
Myšlenka řazení:

1. Prvek **a[i]** je na své správné konečné pozici.
2. Seřadíme prvky **a[l], ..., a[i-1],** které jsou menší nebo rovny **a[i].**
3. Seřadíme prvky **a[i+1], ..., a[p],** které jsou větší nebo rovny **a[i].**

Poznámka: **l je písmeno (levý), 1 je jednička**

- algoritmus seřadí prvky pole, ať zvolíme jako dělící prvek kterýkoliv prvek řazeného pole.

- zvolme dělícím prvkem **a[p]**



- index **i** prochází prvky pole zleva a index **j** zprava

- procházení zastavíme, když není splněna podmínka dělení

- prvky vyměníme a pokračujeme v procházení

- když se indexy procházení zleva a zprava zkříží, vyměníme **a[p]** s prvkem, jenž je v části s většími prvky nejvíce vlevo, tj. **a[i]**

- pří nalezení prvku stejného jako dělící prvek, můžeme zastavit procházení a tento použít na výměnu, protože tím neporušíme invariant naznačený v diagramu

- tato strategie vede k vyváženému dělení pole

```
private int deleni(int l, int p) {
  int i = l - 1;
  int j = p;
  int v = a[p];
  for (;;) {
    while(a[++i] < v);
    while(a[--j] > v)
      if (j == l) break;
    if(i >= j) break;
    vymen(i, j);
  }
  vymen(i, p);
  return i;
}
```

- je-li dělící prvek největším prvkem, potom se na něm, pokud nemá duplikáty, zastaví procházení zleva (**`a[++i] < v)`**.

- je-li dělící prvek nejmenším prvkem, procházení zprava je na levém konci zastaveno explicitně (**`if (j == l) break)`**

A SORTING EXAMPLE (R. Sedgewick)

Příklad dělení

A S O R T I N G E X A M P L E

A S A M P L E

A A O E X S M P L E

A A E R T I N G O X S M P L E

A A E E T I N G O X S M P L R

Příklad řazení

A A E **E** T I N G O X S M P L R
A A **E**
A A

L I N G O P M **R** X T S
L I G **M** O P N
G I L
I **L**

N P O
O P

S T X
T **X**

```
class RazeniDelenim {
  private void vymen(int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
  }
  private int deleni(int l, int p) {
    ...
  }

  private int[] a;
  private int pocet;
  final int maxN=10;

  RazeniDelenim() {
    a = new int[maxN];
    pocet = 0;
  }

  void nactiPrvek(int klic) {
    a[pocet++] = klic;
  }

  void tiskPole () {
    for(int i = 0; i < pocet; i++)
      System.out.print(a[i]+" ");
    System.out.println(" ");
  }

  private void razeniDelenimR(int l, int p) {
    ...
  }

  void razeniDelenim() {
    razeniDelenimR(0, pocet-1);
  }
}
```

Analýza řazení dělením

Efektivnost řazení dělením závisí na tom jak vyvážené je rozdělení.

Nejhorší případ

Pokud při každém dělení vznikne jedna část prázdná a druhá se zbývajícími prvky, kromě dělícího prvku, vznikne nejvíc nevyvážené dělení.

$$T(N) = T(N-1) + N$$

řešením je

$$T(N) = N(N + 1)/2 = O(N^2)$$

- například když algoritmus řazení dělením aplikujeme na již seřazenou posloupnost

- hloubka zásobníku bude také odpovídat *N* rekurzivním voláním

Nejlepší případ

Vzniknou-li každým dělení dvě poloviční posloupnosti, je

$$T(N) = 2\,T(N/2) + N$$

a

$$T(N) \approx N \log_2 N$$

Průměrný případ

- předpoklad, že všechny permutace jsou stejně pravděpodobné

- vyjádření času výpočtu počtem porovnání, s uvážením zkřížení indexů při procházení pole je

$$T(N) = N + 1 + 1/N \sum_{k=1}^{N} (T(k-1) + T(N-k))$$

přičemž $T(0) = T(1) = 0$

- řešením uvedené rekurentní rovnice

$$T(N) \approx 2 \ln N \approx 1.39 N \log_2 N$$

$$T(N) = O(N \log_2 N)$$

Velikost zásobníku

V nerekurzivní verzi se můžeme, podle velikosti úseků vzniklých dělením, rozhodnout, kterou vložíme do zásobníku jako první.

Vložením většího z nich jako prvního, pokračujeme menším z nich.

Tímto způsobem zajistíme, že největší možná velikost bude úměrná $log_2 N$.

Tuto velikost zásobník dosáhne, když posloupnost bude dělena uprostřed.

```
void razeniDelenim() {
  IntZasobnik Z = new IntZasobnik();
  Z.push(0);
  Z.push(pocet-1);
  while (!Z.jePrazdny()) {
    int p = Z.pop();
    int l = Z.pop();
    if (p <= l) continue;
    int i = deleni(l, p);
    if (i-l > p-i) {
      Z.push(l);
      Z.push(i-1);
    }
    Z.push(i+1);
    Z.push(p);
    if (p-i >= i-l) {
      Z.push(l);
      Z.push(i-1);
    }
  }
}
```

Příklad nerekurzivního řazení dělením

A	A	E	**E**	T	I	N	G	O	X	S	M	P	L	R
A	A	**E**												
A	A													
				L	I	N	G	O	P	M	**R**	X	T	S
												S	T	X
													T	**X**
				L	I	G	**M**	O	P	N				
				G	I	L								
					I	**L**								
								N	P	O				
									O	P				

Řazení slučováním (Mergesort)

Algoritmus řazení slučováním je založen na slučování seřazených podposloupností.

```
void slouceniAB(int[] c, int cl,
                int[] a, int al, int ap,
                int[] b, int bl, int bp) {
  int i = al;
  int j = bl;
  for (int k = cl; k < cl + ap - al + bp - bl +  1;
           k++) {
    if (i > ap) {
      c[k] = b[j++];
      continue;
    }
    if (j > bp) {
      c[k] = a[i++];
      continue;
    }
    c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
  }
}
```

- prvky uspořádaných polí **a** a **b** seřadíme sloučením do pole **c** jejich postupným procházením od začátku a vybráním menšího prvku

- přijdeme-li na konec jednoho z nich překopírujeme zbývající prvky druhého

3 5 6 **//a**

2 4 7 8 **//b**

2 3 4 5 6 7 8 **//c**

- úkolem je seřadit prvky jednoho pole

- předpokládejme, že v poli **a** jsou dva seřazené úseky, seřadit je algoritmem řazení slučováním můžeme tak, že vytvoříme pole **pom** s bitonickou posloupností prvků pole **a**, tj. druhý úsek bude seřazen sestupně

Příklad:

a = {1,2,3,**8,9**}

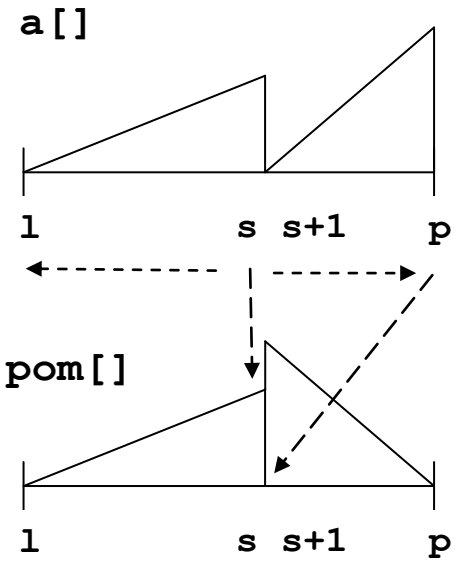
pom = {1,2,3,**9,8**}

- v poli **a** z pole **pom** vytvoříme seřazenou posloupnost **sloučením** původních úseků následovně:

```
i = 0;
j = pocet - 1;
for(k = 0; k < pocet; k++)
  if(pom[j] < pom[i])
    a[k] = pom[j--];
  else
    a[k] = pom[i++];
```

Jak vytvořit bitonickou posloupnost v poli **pom**?

nechť je v poli **a**
- první úsek v prvcích **a[l]** až **a[s]** a
- druhý úsek v prvcích **a[s+1]** až **a[p]**




```
private void slouceni(int l, int s, int p) {
  int i,j;
  // vytvoření bitonické posloupnosti
  for(i = s+1; i > l; i--)
    pom[i-1] = a[i-1];
  for(j = s; j < p; j++)
    pom[p+s-j] = a[j+1];
  // index i - nejmenší prvek levé posloupnosti
  // index j - nejmenší prvek pravé posloupnosti
  for(int k = l; k <= p; k++)
    if (pom[j] < pom[i])
      a[k] = pom[j--];
    else
      a[k] = pom[i++];
}
```

/ / opravit eKnihu – pole **pom** je vytvořeno vně metody **slouceni**
Vysvětlení chyby:
http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa2/portal/cviceni/merge.pdf

Nyní můžeme libovolné pole seřadit tak, že ho rozdělíme na půlky seřadíme jej a potom je sloučíme – základní algoritmus řazení slučováním

```
private void razeniSlucovanimR(int l, int p) {
    if (p <= l) return;
    int s = (p+l)/2;
    razeniSlucovanimR(l, s);
    razeniSlucovanimR(s+1, p);
    slouceni(l, s, p);
}
```

```
class RazeniSlucovanim {

  private void slouceni(int l, int s, int p) {
    ...
  }
  private void razeniSlucovanimR(int l, int p) {
    ...
  }
  private int[] a;
  private int pocet;
  final int maxN=10;

  RazeniSlucovanim() {
    a = new int[maxN];
    pocet = 0;
  }
  void nactiPrvek(int klic) {
    a[pocet++] = klic;
  }
  void tiskPole() {
    for(int i = 0; i < pocet; i++)
      System.out.print(a[i]+" ");
    System.out.println(" ");
  }
  void razeniSlucovanim() {
    int[] pom = new int[pocet];
    razeniSlucovanimR(0, pocet-1);
  }
}
```

Příklad:

A S O R T I N G E X A M P L E
A S
O R
A O R S
I T
G N
G I N T
A G I N O R S T
E X
A M
A E M X
L P
E L P
A E E L M P X

A A E E G I L M N O P R S T X

Analýza algoritmu řazení slučováním

- velikost řazeného pole je $N = 2^n$

- seřazení vznikne sloučením dvou polí o velikosti $N/2$, což vyžaduje N porovnání

- každé z nich vzniklo sloučením dvou polí o velikosti $N/4$, což vyžadovalo $2.(N/2) = N$ porovnání

- postupujeme až k jednoprvkovým polím, což je $log_2 N$ kroků

- počet porovnání je $N \, log_2 N$

- tento výsledek platí obecně

- jako pro řazení haldou, algoritmus řazení slučováním je složitost $N \, log_2 N$ pro libovolnou posloupnost N prvků.

- uvedený algoritmus potřebuje navíc paměť úměrnou N