Rozhledy matematicko-fyzikální

Daniel Kráľ; Martin Mareš; Tomáš Valla

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 2. část

Rozhledy matematicko-fyzikální, Vol. 80 (2005), No. 2, 25–35

Persistent URL: http://dml.cz/dmlcz/146096

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 2. část

Daniel Kráľ, Martin Mareš, Tomáš Valla, MFF UK Praha

Řešitelé *Korespondenčního semináře z programování*, který pro studenty středních škol pořádá MFF UK v Praze, dostávají společně se zadáním úloh též *Recepty z programátorské kuchařky*. Na stránkách *Rozhledů matematicko-fyzikálních* můžete nalézt jejich upravenou verzi. O semináři se lze dovědět více např. z jeho webových stránek na adrese `http://ksp.mff.cuni.cz/`, kde naleznete i původní verzi „receptů“.

V této části „kuchařky“ se budeme věnovat *třídění*, tedy algoritmům umožňujícím co nejrychleji a nejúsporněji seřadit data, se kterými pracujeme. Protože se setříděnými údaji se pracuje mnohem lépe a zejména se v nich lépe vyhledává, bývá třídicí algoritmus součástí téměř každého většího programu. Obvykle třídíme exempláře datové struktury typu pascalského záznamu. V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Pro jednoduchost budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třídit pole celých čísel. Pomocí počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na *vnitřní třídění*, kdy si můžeme dovolit všechna data načíst do (rychlé) paměti počítače, a na *vnější třídění*, kdy třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. V tomto pokračování se omezíme na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 100;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídicí algoritmy patří do skupiny *přímých metod*. Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají většinou časovou složitost $O(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu, pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Stručně popíšeme tři nejznámější algoritmy pro třídění přímými metodami.

Třídění přímým výběrem (SelectSort)

Třídění přímým výběrem (*SelectSort*) je založeno na opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Začneme nalezením nejmenšího čísla v celém poli, které prohodíme s prvkem na začátku pole. Dále postup opakujeme, tentokrát najdeme nejmenší z čísel s indexy $2, \ldots, N$, které prohodíme s druhým prvkem v poli, atd. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i-tém kroku nalezneme i-tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
  begin
    k:=i;
    for j:=i+1 to N do
      if A[j]<A[k] then k:=j;
    if k>i then
    begin
      x:=A[k]; A[k]:=A[i]; A[i]:=x
    end
  end
end;
```

Pro úplnost pár slov o časové složitosti právě popsaného algoritmu: V i-tém kroku musíme nalézt minimum z $N - i + 1$ čísel, na což spotřebujeme čas $O(N)$. Protože algoritmus má celkem N kroků, je jeho celková časová složitost $O(N^2)$.

Třídění přímým vkládáním (InsertSort)

Třídění přímým vkládáním (*InsertSort*) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i-tého kroku je tvořena setříděnými $i-1$ prvky, které byly na začátku algoritmu na prvních $i-1$ místech. V i-tém kroku určíme, na kterou pozici v této utříděné posloupnosti patří prvek s indexem i a zařadíme ho tam („zbytek“ utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $O(N)$. Protože počet kroků algoritmu je N, je jeho časová složitost opět $O\big(N^2\big)$.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
  begin
    x:=A[i];
    j:=i-1;
    while (j>0) and (x<A[j]) do
    begin
      A[j+1]:=A[j];
      j:=j-1;
    end;
    A[j+1]:=x;
  end
end;
```

(Upozornění: Ve všech příkladech předpokládáme, že máme v překladači zapnuto tzv. zkrácené vyhodnocování logických výrazů – třeba v předchozím while-cyklu se při `j=0` hodnoty `x` a `A[0]` již neporovnávají.)

Bublinkové třídění (BubbleSort)

Bublinkové třídění (*BubbleSort*) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublinkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo za větším, tato dvě čísla se

prohodí. Celý postup opakujeme, dokud probíhají nějaké výměny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i,x: integer;
  zmena: boolean;
begin
  zmena:=true;
  while zmena do
  begin
    zmena:=false;
    for i:=1 to N-1 do
      if A[i] > A[i+1] then
      begin
        x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x; zmena:=true
      end
  end
end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech while-cyklem bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (detailní ověření tohoto tvrzení přenecháme čtenáři). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost je v nejhorším případě $O(N^2)$, neboť každý průchod spotřebuje čas $O(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma algoritmům je to, že pokud je pole na začátku setříděné, algoritmus spotřebuje jen lineární čas $O(N)$.

Třídění haldou (HeapSort)

Sofistikovanější třídicí algoritmy mají časovou složitost $O(N \log N)$. Minule jsme ukázali datovou strukturu zvanou *halda*. Připomeňme stručně její vlastnosti. Halda je datová struktura, která obsahuje N prvků. V čase $O(\log N)$ lze prvek do haldy přidat či z haldy odebrat a v čase $O(1)$ lze zjistit hodnotu nejmenšího prvku, který halda obsahuje. A právě hledání nejmenšího prvku bylo to, co jsme potřebovali v našem prvním třídicím algoritmu (SelectSort). Což tedy si nejdříve vytvořit haldu obsahující N prvků, které máme setřídit, a pak v N krocích postupně odebírat

z haldy vždy její nejmenší prvek? Prvky budeme odebírat v pořadí od nejmenšího po největší a dávat je postupně do původního pole. Výsledkem bude setříděné pole. Protože každý z N prvků do haldy jednou přidáme a jednou z ní odebereme, bude celková časová složitost algoritmu $O(N \log N)$. Právě popsaný algoritmus se nazývá *třídění haldou* (*HeapSort*) – halda se anglicky řekne *heap*. Popsaný algoritmus lze dokonce implementovat tak, že používá pouze jediné pole (není potřebná pomocná paměť pro haldu). V ukázkové implementaci budeme místo nejmenšího prvku hledat v haldě největší prvek, setříděné pole budeme vytvářet od konce a na začátku algoritmu postavíme haldu v lineárním čase (první for-cyklus).

```
procedure HeapSort(var A: Pole);
var i,x: integer;
  { "zabublání" prvku v haldě }
  procedure bublej(m,i: integer);
  { m je index posledního prvku haldy,
    i je index zabublávaného prvku }
  var j,x: integer;
  begin
    while 2*i<=m do
    begin
      j:=2*i;
      if (j<m) and (A[j+1]>A[j]) then j:=j+1;
      if A[i]>=A[j] then break;
      x:=A[i];
      A[i]:=A[j];
      A[j]:=x;
      i:=j;
    end;
  end;
begin
  { postav haldu }
  for i:=N div 2 downto 1 do bublej(N,i);
  { vybírej největší prvek }
  for i:=N downto 2 do
  begin
    x:=A[1];
    A[1]:=A[i];
```

```
    A[i]:=x;
    bublej(i-1,1);
  end;
end;
```

Třídění sléváním (MergeSort)

Třídění sléváním (*MergeSort*) je založeno na principu slévání (spojování) již setříděných posloupností. Představme si, že máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvky obou posloupností, které jsme dosud do nově vytvářené posloupnosti nedali, a menší z těchto prvků do nové posloupnosti přidat. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

Popíšeme a předvedeme zde modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus pracuje v několika *fázích*. Na začátku první fáze tvoří každý prvek jednoprvkovou setříděnou posloupnost. Na začátku i-té fáze mají setříděné posloupnosti délku 2^{i-1}. V i-té fázi ze dvojic 2^{i-1}-prvkových posloupností vytvoříme vždy jedinou posloupnost délky 2^i. Pokud N není násobkem čísla 2^i, bude délkou poslední posloupnosti zbytek po dělení čísla N číslem 2^i. Zastavíme se, pokud $2^i > N$, tj. po $\lceil \log_2 N \rceil$ fázích.*) Protože v i-té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1}-prvkových posloupností, je časová složitost jedné fáze $O(N)$. Celková časová složitost popsaného algoritmu je $O(N \log N)$.

```
procedure MergeSort(var A: Pole);
var P: Pole; { pomocné pole }
  delka:integer; { délka setříděných posloupností }
  i: integer; { index do vytvářené posloupnosti }
  i1,i2: integer; { index do slévaných posloupností }
  k1,k2: integer; { konce slévaných posloupností }
begin
  delka:=1;
  while delka<N do
  begin
    i1:=1; i2:=delka+1; i:=1;
    k1:=delka; k2:=2*delka;
```

*) $\lceil x \rceil$ je tzv. horní celá část čísla x, tj. nejmenší celé číslo větší nebo rovné číslu x.

```
    while i<=N do
    begin
      while (i1<=k1) or (i2<=k2) do
      begin
        if (i1>k1) or ((i2<=k2) and (A[i1]<=A[i2])) then
        begin
          P[i]:=A[i1]; i:=i+1; i1:=i1+1
        end
        else
        begin
          P[i]:=A[i2]; i:=i+1; i2:=i2+1
        end
      end;
      i1:=k2+1; i2:=k2+delka;
      k1:=k2+delka;
      k2:=k2+2*delka;
      if k1>N then k1:=N;
      if k2>N then k2:=N;
    end;
    A:=P;
    delka:=2*delka
  end;
end;
```

Pro úplnost poznamenejme, že popsaný algoritmus lze při zachování časové složitosti implementovat i bez pomocného pole. Existuje též modifikace tohoto algoritmu, která má v nejhorším případě počet fází $O(\log N)$, ale pokud je pole na začátku již setříděné, proběhne pouze jediná fáze – v takovém případě má algoritmus časovou složitost $O(N)$.

QuickSort

Jako poslední algoritmus, který pracuje v čase $O(N \log N)$, předvedeme algoritmus zvaný *QuickSort*. Je založen na metodě *Rozděl a panuj*. Nejprve zvolíme nějaké číslo, kterému budeme říkat *pivot*. Poté pole přeuspořádáme a rozdělíme na dvě části tak, že žádný prvek první části nebude větší než pivot a žádný prvek druhé části nebude menší než pivot. Prvky v obou částech pak setřídíme rekurzivním zavoláním našeho algoritmu. Musíme dát ale pozor, aby v každém kroku byly obě části

neprázdné (a rekurze tedy byla konečná). Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků byly levá a pravá část pole přibližně stejně velké. Nejlepší volbou pivota by tedy byl *medián* tříděného úseku, tj. takový prvek, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání zvládneme v lineárním čase a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $O(\log N)$ a celková časová složitost $O(N \log N)$ (na každé úrovni rekurze je součet délek tříděných posloupností nejvýše N). Ačkoliv existuje algoritmus, který medián pole nalezne v čase $O(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N *) je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud nesetříděného úseku – zkrátka se „sáhne“ někam do pole a příslušný prvek se prohlásí za pivota. Dá se ukázat, že takovýto algoritmus poběží s velmi vysokou pravděpodobností v čase $O(N \log N)$. Je však třeba si uvědomit, že pokud se pivot volí náhodně, může rekurze dosáhnout až hloubky N a časová složitost algoritmu je pak $O(N^2)$ – představme si, že se pivot v každém rekurzivním volání nešťastně zvolí jako největší prvek z tříděného úseku. V naší implementaci QuickSortu nebudeme pivota volit náhodně, ale vždy za něj zvolíme prostřední prvek tříděného úseku.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
  k:=A[(i+j) div 2];
  repeat
    while A[i]<k do i:=i+1;
    while A[j]>k do j:=j-1;
    if i<=j then
    begin
      x:=A[i]; A[i]:=A[j]; A[j]:=x;
      i:=i+1;
      j:=j-1;
    end
```

*) Jde o hodnotu konstanty c z definice symbolu „velké O“ (str. 27 v 1. čísle tohoto ročníku časopisu).

```
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;
```

Přihrádkové třídění (RadixSort)

Na závěr předvedeme dva třídicí algoritmy, které jsou vhodné, pokud tříděné objekty mají některé další speciální vlastnosti. Prvním z nich je *přihrádkové třídění* (*RadixSort*). Algoritmus je založen na jednoduché myšlence, že pokud jsou hodnoty klíče tříděných objektů z nějaké malé množiny, řekněme s počtem prvků K, stačí rozdělit tříděné objekty do K skupin podle hodnoty jejich klíče. Do výsledného pole se pak postupně vypíší objekty z jednotlivých skupin v pořadí dle rostoucí hodnoty klíče. Jak rozdělení do množin, tak vytvoření výsledného pole lze provést v čase $O(N + K)$, tedy $O(N)$, pokud je K konstanta.

Popíšeme zde *víceprůchodovou* variantu, která je vhodnější pro větší hodnoty K. V první fázi rozdělíme čísla do přihrádek (skupin) podle nejméně významné cifry a spojíme do jedné posloupnosti, v druhé fázi roztřídíme čísla podle druhé nejméně významné cifry a opět spojíme do jedné posloupnosti atd. Je důležité, aby se uvnitř každé přihrádky zachovalo pořadí čísel v posloupnosti na začátku fáze, tj. posloupnost uložená v každé přihrádce je vybranou posloupností z posloupnosti ze začátku fáze. Tvrdíme, že na konci i-té fáze obsahuje výsledná posloupnost čísla utříděná podle i nejméně významných cifer. Zřejmě i-té nejméně významné cifry tvoří neklesající posloupnost, neboť podle nich jsme právě v této fázi rozdělovali čísla do přihrádek, a pokud dvě čísla mají tuto cifru stejnou, jsou uložena v pořadí dle jejich $i - 1$ nejméně významných cifer, neboť v každé přihrádce jsme zachovali pořadí čísel z konce minulé fáze. Na závěr poznamenejme, že místo čísel podle cifer lze do přihrádek rozdělovat textové řetězce podle jejich znaků apod.

Časová složitost této varianty RadixSortu, pokud třídíme celá čísla od 1 do K a v každém kroku je rozdělujeme do ℓ přihrádek, je $O((N + \ell) \log_\ell K)$, tedy $O(N)$, pokud K a ℓ jsou konstanty. Předvedeme implemenatci algoritmu pro $K = 255$ a $\ell = 2$ (čísla budeme roztřiďovat podle bitů v jejich binárním zápisu).

```
const K=255;
procedure RadixSort(var A: Pole);
var P0,P1: Pole;
  k1,k2: integer;
  i: integer;
  bit: integer;
begin
  bit:=1;
  while bit<=K do
  begin
    k1:=0; k2:=0;
    for i:=1 to N do
      if (A[i] and bit)=0 then
      begin
        k1:=k1+1; P0[k1]:=A[i]
      end
      else
      begin
        k2:=k2+1; P1[k2]:=A[i]
      end;
    for i:=1 to k1 do A[i]:=P0[i];
    for i:=1 to k2 do A[k1+i]:=P1[i];
    bit:=bit shl 1;
  end
end;
```

CountSort

Posledním třídicím algoritmem, o kterém se zmíníme, je *CountSort*. Pokud tříděné objekty obsahují pouze klíč a možných hodnot klíče je málo, je dobré spočítat objekty se stejnými hodnotami klíče a na základě toho vytvořit setříděné pole. Tento algoritmus předvedeme na příkladu třídění pole celých čísel z intervalu $\langle D, H\rangle = \langle 1, 10\rangle$. Časová složitost takového algoritmu je lineární v N, tedy $O(N)$, ale nesmíme zapomínat, že v tomto případě „velké O“ obsahuje jako aditivní konstantu počet prvků množiny, ze které mohou být objekty v tříděném poli (v našem případě $H - D + 1$).

```
const D = 1;
      H = 10;
procedure CountSort(var A: Pole);
var C: array[D..H] of integer;
  i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]]:=C[A[i]] + 1;
  k:=1;
  for i:=D to H do
    for j:=1 to C[i] do
    begin
      A[k]:=i;
      k:=k+1;
    end
end;
```

* * * * *

TEKUTÁ MEDAILE

Za své návštěvy v Rusku vyprávěl Bohr historku o tekuté nobelovské medaili. Německý fyzik Max von Laue, který se proslavil myšlenkou zkoumat krystaly metodou difrakce rentgenového záření, pomohl před druhou světovou válkou řadě svých židovských kolegů ohrožených nacisty a byl v Německu ve složité situaci. Protože v té době nacisté zabavovali drahé kovy pro válečné účely, hrozilo Lauemu, že by mohl přijít i o zlatou medaili, kterou dostal při udělení Nobelovy ceny v roce 1914. Poslal ji proto Bohrovi do Dánska, aby mu ji uschoval. Když pak Němci obsadili Dánsko, byl Laue ve dvojím nebezpečí – na medaili bylo jeho jméno. Bohr proto uvažoval, jak s medailí naložit. Chemik Hevesy navrhl, že by ji mohli rozpustit v lučavce královské. To také udělali a medaile se ocitla v láhvi v tekutém stavu. Po válce pak zlato ze žlutě zbarveného roztoku opět vyloučili a poslali do Stockholmu, aby na medaili byl zhotoven nový nápis.

Ivan Štoll *)

*) Z publikace *Historky o slavných matematicích a fyzicích*, Praha, Prometheus 2005