Obsah

Články

Řadicí algoritmus 1
Stabilní řazení 4
Řazení haldou 5
Shell sort 6
Merge sort 9
Quicksort 12
Radix sort 15
Bucket sort 16

Reference

Zdroje článků a přispěvatelé 17
Zdroje obrázků, licence a přispěvatelé 18

Licence článků

Licence 19

Řadicí algoritmus

Řadicí algoritmus (často nesprávně **třídicí algoritmus**)[1] je algoritmus zajišťující seřazení daného souboru dat do specifikovaného pořadí. Nejčastěji se řadí podle numerické velikosti čísel, případně abecedně. Řazení je velmi častá úloha, která je také částí mnoha dalších algoritmů; vývoji co možná nejefektivnějších algoritmů řazení se proto věnuje velké úsilí.

Z hlediska řazení se vstupní data chápou jako soubor dvojic klíč–hodnota, přičemž po seřazení je posloupnost klíčů monotónní, zatímco na připojené hodnoty se při řazení nebere zřetel a pouze se přesouvají vždy s odpovídajícím klíčem. Při existenci několika položek se stejným klíčem se však podle pořadí odpovídajících hodnot rozlišují stabilní a nestabilní algoritmy.

Definice problému

Na vstupu je posloupnost $S = (S_1, S_2, \ldots, S_n)$; cílem je najít takovou posloupnost $S' = (S'_1, S'_2, \ldots, S'_n)$, pro kterou platí dvě základní kritéria:

1. Tato posloupnost je seřazená:

   $$S'_1 \leq S'_2 \leq \cdots \leq S'_n.$$
2. Posloupnost S' je permutací původní posloupnosti S (obsahuje tedy stejná data, jen v jiném pořadí).

V definici relace uspořádání ≤ se přitom bere ohled pouze na klíče příslušných hodnot.

Název

Kromě o něco přesnějšího označení úlohy jako *řazení* se velmi často používá také název *třídění*, což je nepřesné. *Třídění* v užším smyslu označuje jednodušší úlohu spočívající v rozdělení vstupní množiny do několika skupin podle zadaného kritéria, bez potřeby určení jejich vzájemného pořadí. Některé řadicí algoritmy však pracují na principu třídění.

Složitost

Pro seřazení množiny n prvků existuje očividná dolní mez časové asymptotické složitosti $\Omega(n)$(každý prvek je potřeba alespoň jednou přečíst). Této dolní meze je ale možno dosáhnout jen při předem známé, omezené množině klíčů (např. interval v přirozených číslech). Lze dokázat, že u řazení, které je založeno na porovnávání dvojic klíčů (což je univerzální metoda použitelná pro libovolná data), je minimální časová složitost $\Omega(n \log n)$.

Klasifikace algoritmů

Podle různých kritérií se algoritmy řazení dají dělit do různých skupin. Dvě základní skupiny algoritmů jsou tzv. *vnitřní* a *vnější řazení*. Vnitřní řazení vyžaduje, aby všechna řazená data byla uložena v operační paměti, kde k nim má algoritmus možnost libovolně přistupovat. Pokud je dat tak velké množství, že v jednu chvíli může být v operační paměti jen nějaká část dat (a zbytek je ve vnější paměti, např. na pevném disku), je třeba použít vnější řazení.

Největší část algoritmů řazení je založena na porovnávání dvojic prvků; jedná se o univerzální metodu, kterou lze seřadit libovolná data v libovolné reprezentaci (stačí příslušná relace uspořádání). Pro některé konkrétní reprezentace nějak vymezené množiny dat lze sestrojit algoritmy, které fungují na jiném principu, např. na základě reprezentace řazených čísel v poziční číselné soustavě.

Kromě samotných řazených dat také algoritmus zpravidla potřebuje nějakou dodatečnou pracovní paměť. Pokud je velikost této paměti konstantní (nezávislá na množství řazených dat, označováno jako $O(1)$), algoritmus se

označuje jako řazení na původním místě (*in situ*), jiné algoritmy však potřebují dodatečnou paměť, například místo o velikosti původních dat (tedy $O(N)$ v asymptotickém vyjádření), ve kterém generují seřazený výsledek.

Vstupní data mohou obsahovat několik prvků se shodným klíčem. Podle vzájemné polohy těchto prvků před a po seřazení (kterou lze detekovat podle přidružených dat, která nejsou součástí klíče) se rozlišují tzv. *stabilní* a *nestabilní* řadicí algoritmy: stabilní algoritmus zachovává vzájemné pořadí položek se stejným klíčem, u nestabilního není vzájemné pořadí prvků se stejným klíčem zaručeno. (Ale z libovolného nestabilního algoritmu lze učinit stabilní tím, že se klíč každé položky vstupních dat rozšíří o pozici položky v původním souboru.)

Podle chování na částečně seřazených souborech dat se rozlišují algoritmy *přirozené* a *nepřirozené*: přirozený algoritmus rychleji zpracuje seřazenou množinu než neseřazenou.

Ukázka řazení

Výběr	**Vkládaní**	**Záměna**
614532		
1•64532	6•14532	**145326**
12•6453	16•4532	14**325**6
123•645	146•532	1**324**56
1234•65	1456•32	1**23**456
12345•6	13456•2	
123456		

Dále lze algoritmy zhruba rozdělit podle základní myšlenky. Existuje několik základních druhů algoritmů univerzální vnitřního řazení, přičemž některé pokročilejší algoritmy kombinují více postupů.

Řazení výběrem

V souboru se vždy najde nejmenší ze zbývajících položek a uloží na konec postupně budovaného seřazeného souboru.

Řazení vkládáním

Ze souboru neseřazených dat se postupně bere položka po položce a vkládá se na správné místo v seřazeném souboru (zpočátku prázdném).

Řazení záměnou

V souboru se vždy nalezne (nějakou metodou závislou na konkrétním algoritmu) nějaká dvojice prvků, která je ve špatném pořadí, a tyto prvky se navzájem zamění.

Řazení slučováním

Vstupní soubor se rozdělí na části, které se (typicky rekurzivně) seřadí; výsledné seřazené části se poté sloučí takovým způsobem, aby i výsledek byl seřazený.

Neexistuje žádný „dokonalý“ řadicí algoritmus, který by byl ideální pro všechna použití. Různé algoritmy mají různé vlastnosti co se týká jejich očekávané časové a paměťové složitosti, náročnosti implementace a dalších vlastností. Pro konkrétní podmínky se tak často navrhují specifické varianty.

Běžné algoritmy

Přehled běžných univerzálních algoritmů řazení

Název		Časová složitost			Dodatečná paměť	Stabilní	Přirozená	Metoda
Anglicky	Česky	Minimum	Průměrně	Maximum				
Bubble sort	Bublinkové řazení	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	záměna
Heapsort	Řazení haldou	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	ne	ne	halda, záměna
Insertion sort	Řazení vkládáním	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	vkládání
Merge sort	Řazení slučováním	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ano	ano	slučování
Quicksort	Rychlé řazení	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	ne	ne	záměna
Selection sort	Řazení výběrem	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	zprav. ne	ne	výběr
Shell sort	Shellovo řazení	$O(n^{1+\frac{c}{\sqrt{m}}})$ [2]		$O(n \log^2 n)$	$O(1)$	ne	ano	vkládání
Comb sort		$O(n)$	$O(n \log n)$	$O(n^2)$	$O(1)$	ne	ano	záměna

Běžné algoritmy řazení založené na jiném principu

Název		Časová složitost	Dodatečná paměť	Stabilní	Stručný popis metody
Anglicky	Česky				
Bucket sort	Přihrádkové řazení	$O(n \cdot k)$	$O(2^k)$	ano	Podle hodnoty klíče se data roztřídí do připravených přihrádek seřazených podle velikosti
Radix sort	Řazení tříděním podle základu	$O(n \cdot 2^k)$	$O(n)$	ano	Postupné třídění po jednotlivých cifrách poziční číselné soustavy
Counting sort	Řazení počítáním četností	$O(n + k)$	$O(k)$	ano	Umístění do správného pořadí po spočítání četností jednotlivých prvků

Reference

- Donald E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching.* Second Edition. Reading, Massachusetts: Addison-Wesley, 1998. ISBN 0-201-89685-0
- Algoritmy řazení ve slovníku algoritmů a datových struktur NIST [3]
- *Třídění* ve výukových materiálech k předmětu Základy algoritmizace [4]

[1] http://ksp.mff.cuni.cz/tasks/16/cook2.html
[2] Robert Sedgewick: *Analysis of Shellsort and Related Algorithms* (http://www.cs.princeton.edu/~rs/shell/), Fourth Annual European Symposium on Algorithms, Barcelona, září 1996
[3] http://www.nist.gov/dads/HTML/sort.html
[4] http://tjn.fjfi.cvut.cz/~virius/jera/binary/trideni.htm

Externí odkazy

- Animace některých algoritmů a datových struktur (http://mifeet.alpaka.cz/algo/algorithms.swf)

Stabilní řazení

Řadicí algoritmus je stabilní tehdy, jestliže po seřazení zachovává vzájemné pořadí prvků se stejným klíčem.

Jinými slovy: Mějme množinu prvků M. Pro každé dva prvky R a S o stejném klíči z této množiny platí, že pokud byl prvek R v neseřazené množině před prvkem S, pak je i v seřazené posloupnosti prvek R před prvkem S. Pokud tato vlastnost platí pro všechny možné množiny M, pak je algoritmus stabilní.

Příklady

Řazení jmen a příjmení

Mějme seznam jmen a příjmení reprezentovaný uspořádanou dvojicí (A, B), kde A je jméno a B je příjmení.
Seznam vypadá takto: (a, z), (b, x), (b, y).
Po seřazení stabilním algoritmem bude výsledek vždy vypadat takto: (a, z), (b, x), (b, y).

Pokud by byl použit nestabilní řadící algoritmus, výsledek by mohl vypadat takto: (a, z), (b, y), (b, x).

Města a okresy

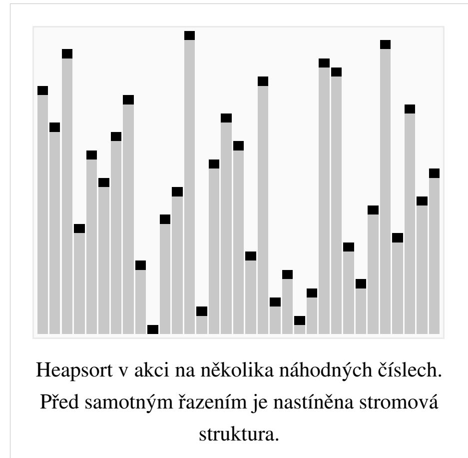
Máme-li seznam českých měst seřazený abecedně dle názvu a necháme-li ho seřadit stabilním řadicím algoritmem dle okresů, budou v seznamu města seřazena dle okresů, ale v rámci každého okresu zůstane zachováno abecední řazení dle názvu. Pokud bychom použili algoritmus, který není stabilní, tak toto zaručeno nemáme.

Příklady stabilních algoritmů

- Bubble sort
- Insertion sort
- Merge sort
- Bucket sort
- Radix sort
- Counting sort

Řazení haldou

Heapsort neboli **řazení haldou** je jeden z nejlepších obecných algoritmů řazení, založených na porovnávání prvků. Byť je v průměru o něco pomalejší než dobře napsaný quicksort, je jeho zaručená časová náročnost $O(N \log N)$ a dokáže řadit data na původním místě (má pouze konstantní nároky na paměť). Heapsort není stabilní řadící algoritmus.

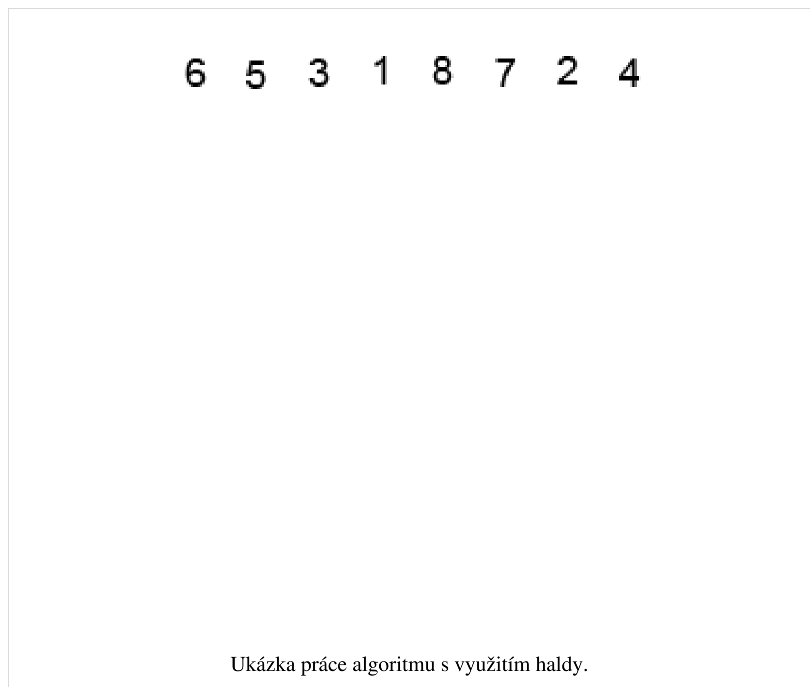


Heapsort v akci na několika náhodných číslech. Před samotným řazením je nastíněna stromová struktura.

Popis algoritmu

Základní myšlenkou tohoto algoritmu je využití datové struktury označované jako *halda* (angl. *heap*). Tato struktura umí velmi efektivně provést operaci vložení prvku a operaci výběr největšího prvku. Proto lze pomocí haldy seřadit dodaná data od největšího k nejmenšímu prostě pomocí jejich vložení do haldy a následného postupného vybírání největšího prvku.

V praxi lze haldu vystavět přímo ve vstupním poli tím způsobem, že jsou následovníci prvku n uloženi do prvků $2n$ a $2n$+1 (při indexování od jedničky), a také následné vybírání prvků lze provádět pouhým přeuspořádáváním dat v tomto poli.



Ukázka práce algoritmu s využitím haldy.

Pomocný algoritmus

Uvažujme binární haldu pro nalezení "nejvyššího" prvku. Tato halda bude umístěna v poli od indexu 1 do indexu N. "Pravidlem haldy" je, že prvek umístěný na indexu i je vyšší než prvky umístěné na indexu 2*i a 2*i+1. U koncových uzlů na indexech (N/2+1 až N) je "pravidlo haldy" automaticky splněné - není je s čím porovnávat.

Pro oba kroky řazení je využit pomocný algoritmus, který v čase O(log(n)) dokáže prodloužit částečně vytvořenou haldu zepředu o jeden prvek. Přesněji řečeno - pokud všechny prvky s indexy L+1 až R (včetně krajních prvků) splňují "pravidlo haldy", po provedení pomocného algoritmu budou splňovat pravidlo haldy prvky s indexy L až R. Pokud prvek na indexu L nevyhovuje pravidlu haldy, vyměníme ho s větším z prvků na indexech 2*L a 2*L+1 a postup zopakujeme pro index s kterým jsme měnili.

Stavba haldy

První krok haldového řazení spočívá v postupném prodlužování haldy z rozsahu (N/2 až N) na (1 až N). Po provedení N/2 kroků je halda vytvořena.

Využití haldy

Halda, která splňuje popsané podmínky, má na vrcholu (index 1) prvek s největší hodnotou. Ve druhém kroku haldového řazení se tento prvek vymění za poslední prvek pole. Tak se na konec pole dostane největší prvek (tím je zařazen na správné místo) ale prvkem, přesunutým z konce pole dojde k porušení pravidel haldy. Je třeba spustit pomocný algoritmus, tentokrát pro prvky s indexy (1 až N-1).

Výše zmíněný postup se opakuje pro stále se zmenšující haldu. Při každém kroku je na vrcholu haldy největší ze zbývajících prvků, a ten je výměnou s posledním prvkem této menší haldy zařazen na správné pořadí v poli.

Shell sort

Další významy jsou uvedeny v článku Shell.

Shellovo řazení (anglicky **shell sort**, nebo též řazení se snižujícím se přírůstkem) je řadicí algoritmus podobný insert sortu, který objevil a v roce 1959 publikoval Donald Shell.

Shell sort je nestabilní řadicí metoda (tj. nezachovává původní pořadí dvou prvků se stejným klíčem - mají-li prvky X a Y stejný klíč a v původním neseřazeném poli se prvek X vyskytuje před prvkem Y, ve výsledném seřazeném poli tomu tak po použití nestabilního řazení nebude).

Asymptotická složitost je $O(n^2)$. Přesto je shell sort z kvadratických řadicích algoritmů nejvýkonnější. Časová složitost shell sortu je přibližně rovna $n^{3/2}$.

Princip

Shell sort funguje podobně jako insert sort. Ovšem shell sort neřadí prvky umístěné vedle sebe, nýbrž prvky mezi nimiž je určitá mezera. V každém následujícím kroku je mezera zmenšena. V okamžiku, kdy je mezera zmenšena na velikost 1 tak je principiálně řazeno pomocí insert sortu. Výhoda tohoto přístupu (řazení se snižujícím se přírůstkem) je rychlé přemístění nízkých a vysokých hodnot. Poslední iterace totiž přesune jen minimum prvků.

Velikost mezery

Aby byla zajištěna nejvyšší efektivita algoritmu je potřeba zvolit ideální velikost mezery. Marcin Ciura zjistil, že ideální mezerou je 2,2. Jako počáteční mezera je tedy v algoritmu použita největší možná z řady čísel: 1, 4, 10, 23, 57, 132, 301, 701, ... (celočíselné násobky 2,2).

Implementace

Pseudokód s vnitřním vkládáním (sekvence od Marcin Ciura):

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

foreach (gap in gaps)
{
    # Do an insertion sort for each gap size.
    for (i = gap; i < n; i += 1)
```

```
    {
        temp = a[i]
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap]
        }
        a[j] = temp
    }

}
```

Shell sort v jazyce C/C++

Následující implementace shell sortu v jazyce C setřídí pole celočíselných typů (*int*ů).

```
void shell_sort(int A[], int size) {
  int i, j, increment, temp;
  increment = size / 2;

  while (increment > 0) {
    for (i = increment; i < size; i++) {
      j = i;
      temp = A[i];
      while ((j >= increment) && (A[j-increment] > temp)) {
        A[j] = A[j - increment];
        j = j - increment;
      }
      A[j] = temp;
    }

    if (increment == 2)
       increment = 1;
    else
       increment = (int) (increment / 2.2);
  }
}
```

Shell sort v Javě

Implementace v Javě je následující:

```
public static void shellSort(int[] a) {
    for (int increment = a.length / 2; increment > 0;
          increment = (increment == 2 ? 1 : (int) Math.round(increment
/ 2.2))) {
        for (int i = increment; i < a.length; i++) {
            int temp = a[i];
            for (int j = i; j >= increment && a[j - increment] > temp; j
-= increment){
                a[j] = a[j - increment];
```

```
                a[j - increment] = temp;
            }
        }
    }
}
```

Shell sort v Pythonu

```
def shellsort(a):
    def increment_generator(a):
        h = len(a)
        while h != 1:
            if h == 2:
                h = 1
            else:
                h = 5*h//11
            yield h

    for increment in increment_generator(a):
        for i in xrange(increment, len(a)):
            for j in xrange(i, increment-1, -increment):
                if a[j - increment] < a[j]:
                    break
                a[j], a[j - increment] = a[j - increment], a[j]
    return a
```

Reference

- http://dudka.cz/studyIAL
- http://www.algoritmy.net/article/154/Shell-sort

Merge sort

Merge sort je řadicí algoritmus, jehož průměrná i nejhorší možná časová složitost je ($O(N \log N)$). Algoritmus je velmi dobrým příkladem programátorské metody rozděl a panuj.

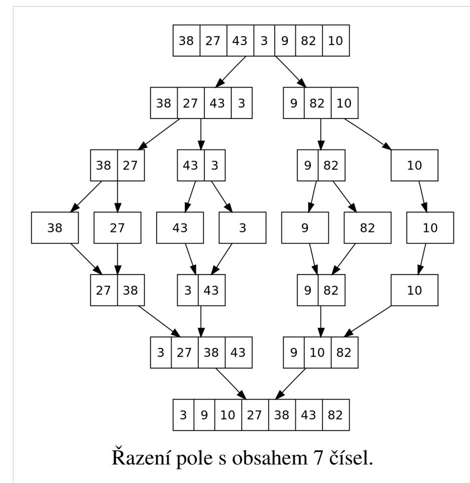
Algoritmus vytvořil v roce 1945 John von Neumann.

Merge sort v akci na několika náhodných číslech

Algoritmus

Algoritmus pracuje následovně:

1. Rozdělí neseřazenou množinu dat na dvě podmnožiny o přibližně stejné velikosti
2. Seřadí obě podmnožiny
3. Spojí seřazené podmnožiny do jedné seřazené množiny



Řazení pole s obsahem 7 čísel.

Implementace v pseudokódu

```
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
```

Existuje několik variant pro funkci merge(), toto je nejjednodušší varianta:

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    while length(left) > 0
        append left to result
        left = rest(left)
    while length(right) > 0
        append right to result
        right = rest(right)
    return result
```

Příklad

Zde je názornější ukázka za pomocí STL algoritmu *std::inplace_merge*.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(void)
{
        std::vector<unsigned> data;

        for(unsigned i = 0; i < 10; i++)
                data.push_back(i);

        std::random_shuffle(data.begin(), data.end());

        std::cout << "Initial: ";

        for(unsigned i = 0; i < 9; i++)
                std::cout << data.at(i) << " ";

        std::cout << data.at(9) << "." << std::endl;

        for(unsigned m = 1; m <= data.size(); m *= 2)
        {
                for(unsigned i = 0; i < data.size() - m; i += m * 2)
                {
                        std::inplace_merge(
                                data.begin() + i,
```

```
                                        data.begin() + i + m,
                                        data.begin() + std::min<unsigned>(i + m * 2,
(unsigned)data.size()));
                    }
          }

          std::cout << "Sorted:  ";

          for(unsigned i = 0; i < 9; i++)
                    std::cout << data.at(i) << " ";

          std::cout << data.at(9) << "." << std::endl;

          return 0;
}
```

Pro porovnání, zde je funkcionální varianta programu zapsaná v jazyce Haskell:

```
mergeSort []  = []
mergeSort [x] = [x]
mergeSort s   = merge (mergeSort u) (mergeSort v)
                where (u,v) = splitAt (n `div` 2) s
                      n     = length s

merge s []        = s
merge [] t        = t
merge (x:u) (y:v) = if x <= y then x : merge u (y:v)
                              else y : merge (x:u) v
```

Srovnání s ostatními řadicími algoritmy

Velkou nevýhodou oproti algoritmům stejné rychlostní třídy (např. heapsort) je, že Mergesort pro svou práci potřebuje navíc pole o velikosti N. Existuje sice i modifikace Mergesortu, která toto pole nepotřebuje, ale její implementace je velmi složitá a kvůli vysoké režii i pomalá. Kromě toho je Mergesort ve většině případů pomalejší než quicksort nebo heapsort.

Na druhou stranu je Mergesort stabilní řadicí algoritmus, lépe se paralelizuje a má vyšší výkon na sekvenčních médiích s nižší přístupovou dobou. Velkou výhodou proti quicksortu je, že čas potřebný pro třídění je téměř nezávislý na počátečním řazení tříděné posloupnosti. Vyšší spotřeba paměti není tak velkým problémem jak se může na první pohled zdát, protože při třídění nemusíme manipulovat přímo s položkami tříděného pole, ale pouze s polem indexů, které v paměti většinou zabírá mnohem méně místa. Při použití více polí indexů můžeme mít pole setříděné "současně" podle více kritérií. V mnoha implementacích programovacích jazyků je Mergesort implicitním řadicím algoritmem (v Perlu 5.8, v Javě nebo v GNU C Library).

Externí odkazy

- Jednoduchá implementace Merge sortu v C# [1]
- Merge sort v C++ [2]

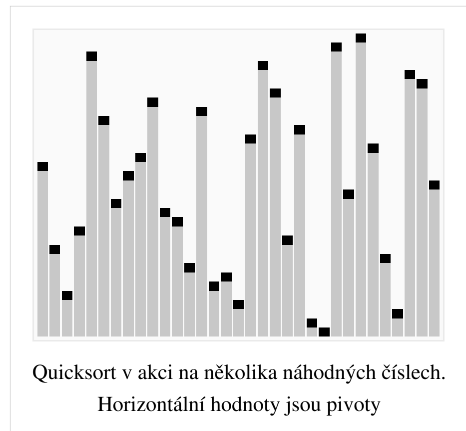
Reference

[1] http://www.martinvseticka.eu/index.php?sekce=browse&page=54
[2] http://www.24bytes.com/Merge-Sort.html

Quicksort

Quicksort (česky „rychlé řazení“) je jeden z nejrychlejších běžných algoritmů řazení založených na porovnávání prvků. Jeho průměrná časová složitost je pro algoritmy této skupiny nejlepší možná ($O(N \log N)$), v nejhorším případě (kterému se ale v praxi jde obvykle vyhnout) je však jeho časová náročnost $O(N^2)$. Další výhodou algoritmu je jeho jednoduchost.

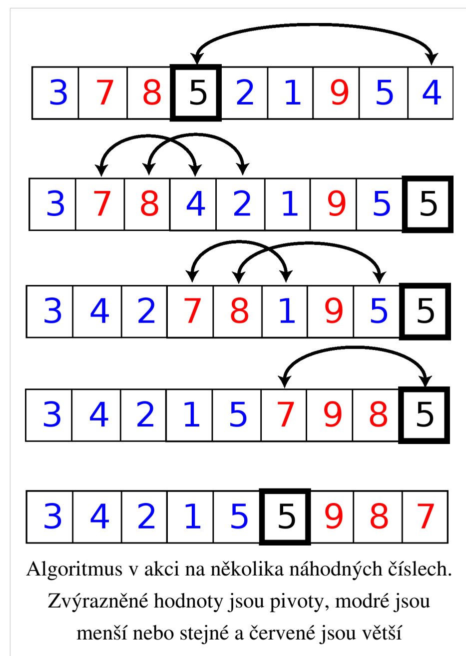
Vymyslel jej Sir Charles Antony Richard Hoare v roce 1962.



Quicksort v akci na několika náhodných číslech. Horizontální hodnoty jsou pivoty

Algoritmus

Základní myšlenkou quicksortu je rozdělení řazené posloupnosti čísel na dvě přibližně stejné části (quicksort patří mezi algoritmy typu rozděl a panuj). V jedné části jsou čísla větší a ve druhé menší, než nějaká zvolená hodnota (nazývaná *pivot* – anglicky „střed otáčení“). Pokud je tato hodnota zvolena dobře, jsou obě části přibližně stejně velké. Pokud budou obě části samostatně seřazeny, je seřazené i celé pole. Obě části se pak rekurzivně řadí stejným postupem.



Algoritmus v akci na několika náhodných číslech. Zvýrazněné hodnoty jsou pivoty, modré jsou menší nebo stejné a červené jsou větší

Volba pivotu

Největším problémem celého algoritmu je volba pivotu. Pokud se daří volit číslo blízké mediánu řazené části pole, je algoritmus skutečně velmi rychlý. V opačném případě se jeho časová složitost blíží $O(N^2)$. Přirozenou metodou na získání pivotu se pak jeví volit za pivot medián. Hledání mediánu (a obecně k-tého prvku) v posloupnosti běží v lineárním čase vzhledem k počtu prvků, tím dostaneme složitost $O(N \log N)$ v nejhorším případě. Nicméně tato implementace není příliš rychlá z důvodu vysokých konstant schovaných v O notaci. Proto existuje velké množství alternativních způsobů, které se snaží efektivně vybrat pivot co nejbližší mediánu. Zde je seznam některých metod:

- První prvek – popřípadě kterákoli jiná fixní pozice. (Fixní volba prvního prvku je velmi nevýhodná na částečně seřazených množinách.)
- Náhodný prvek – často používaná metoda. Lze dokázat, že pokud je pozice pivotu skutečně náhodná, algoritmus poběží v O($N \log N$). Skutečně náhodná čísla generují ale pouze hardwarové generátory, které nemusí dodávat data dostatečně rychle. V praxi se používá spíše pseudonáhodný výběr.
- Metoda mediánu tří – případně pěti či libovolné jiné konstanty. Pomocí pseudonáhodného algoritmu (používají se i fixní pozice, typicky první, prostřední a poslední) se vybere několik prvků z množiny, ze kterých se použitím některého primitivního řadicího algoritmu najde medián a ten je zvolen za pivot.

Průměrná časová náročnost Quicksortu pro náhodná data je řádu O($N \log N$). Testy ukazují, že na pseudonáhodných datech je nejrychlejší ze všech obecných řadicích algoritmů (tedy i rychlejší než Heapsort a Mergesort, které jsou formálně rychlejší). Praktické zkušenosti prvenství Quicksortu rovněž potvrzují. Rychlost Quicksortu však není zaručena pro všechny vstupy. Maximální časová náročnost $O(N^2)$ a použití rekurze Quicksort diskvalifikují v kritických aplikacích.

Kód algoritmu v jazyce Pascal

```
procedure quicksort(l, r: integer);
var
    i, j, pivot, pom: integer;
begin
    i := l; j := r;
    pivot := akt[(l + r) div 2];
    repeat
        while (i < r) and (akt[i] < pivot) do i := i + 1;
        while (j > l) and (pivot < akt[j]) do j := j - 1;
        if i <= j then
        begin
            pom := akt[i];
            akt[i] := akt[j];
            akt[j] := pom;
            i := i + 1;
            j := j - 1;
      end;
    until i > j;
    if j > l then quicksort(l, j);
    if i < r then quicksort(i, r)
end;
```

Kód algoritmu v jazyce C

```
void quicksort(int array[], int left_begin, int right_begin)
{
    int pivot = array[(left_begin + right_begin) / 2];
    int left_index, right_index, pom;
    left_index = left_begin;
    right_index = right_begin;
    do {
        while (array[left_index] < pivot && left_index < right_begin)
```

```
            left_index++;
        while (array[right_index] > pivot && right_index > left_begin)
            right_index--;

        if (left_index <= right_index) {
            pom = array[left_index];
            array[left_index++] = array[right_index];
            array[right_index--] = pom;
        }
    } while (left_index < right_index);

    if (right_index > left_begin) quicksort(array, left_begin,
right_index);
    if (left_index < right_begin) quicksort(array, left_index, right_begin);
}
```

Vlastnosti a využití

Jak již bylo zmíněno, tento algoritmus je ve většině běžných případů nejrychlejší, což může být při řazení rozsáhlých posloupností hlavním požadavkem. Obecně je nestabilní. Může být upraven tak, aby byl stabilní, avšak na to je potřeba dodatečná paměť.

Rychlost výpočtu je většinou vynikající, avšak tento algoritmus vyžaduje více než jiné pečlivou implementaci. Zde uvedená základní rekurzivní verze algoritmu může být pro nasazení v praxi naprosto nevhodná. Vhodné je použít sofistikovanější výběr pivota a řadit po pivotizaci větší část sekvence nerekurzivně a pouze pro menší část použit rekurzi. Základní quicksort je nejpomalejší a nejnáročnější na zásobník při řazení již seřazených nebo převážně seřazených polí. Vzhledem k tomu, že nejde o stabilní řadicí algoritmus, a vzhledem k nutnosti obcházet jeho nedostatky mohou být knihovní funkce pro quicksort na různých systémech a v různých knihovnách implementovány různým způsobem. To znamená, že při zavolání knihovní funkce nebude pole setříděno vždy stejně, což je potenciálním zdrojem problémů s přenositelností software.

Navíc na slabším hardware (například u jednoduchých vestavěných systémů) nebo při řazení velkých polí může prostoduchá implementace quicksortu vést dokonce i k přeplnění zásobníku a pádu programu. Je třeba si uvědomit, že zde uvedený algoritmus pouze demonstruje funkční princip quicksortu. Jde o rekurzivní algoritmus a každé rekurzivní volání rychle spotřebovává paměť zásobníku. Např. průmyslový standard MISRA C použití rekurze zakazuje. Paměť zásobníku bývá zvl. u menších vestavěných systémů velmi omezená. Problém se přitom při ladění nemusí vůbec projevit, protože množství spotřebované paměti závisí na řazených datech.

Navzdory výše zmíněným problémům jde v drtivé většině případů o nejrychlejší známý univerzální algoritmus pro řazení polí v operační paměti počítače. Ne však nejsnadněji použitelný.

Externí odkazy

- Vícerozměrný quicksort v Javě [1]
- Tutorial Quicksortu s průběhem, diagramy a zdrojovými kódy v Javě [2] (Anglicky)

Literatura

Wirth, N.: Algoritmy a štruktúry údajov, Alfa, Bratislava, 1989

Reference

[1] http://fiehnlab.ucdavis.edu/staff/wohlgemuth/java/quicksort
[2] http://web.archive.org/web/20080409034028/http://www.mycsresource.net/articles/programming/sorting_algos/quicksort

Radix sort

Radix sort (česky Číslicové třídění) je řadicí algoritmus, který řadí celá čísla postupným procházením všech číslic (často se vstupní čísla převádějí do soustavy o jiném základu, odtud tedy název). Jelikož celočíselné hodnoty mohou reprezentovat řetězce (jména, data apod.), a dokonce i vhodně formátovaná čísla s plovoucí desetinnou čárkou, radix sort není omezen pouze na řazení celých čísel.

Většina digitálních počítačů vnitřně reprezentuje všechna data jako binární čísla, nejpřirozenější je pro něj tedy řazení podle skupin bitů (tj. podle číslic o základu 8, 16, 32, 256 apod.).

V některých případech lze dosáhnout asymptotické časové složitosti až O(n) (dolní hranice). Obecně je časová složitost Radix sortu $O((z+n)*\log_z u)$, kde z je základ zvolené číselné soustavy, n počet čísel na vstupu a u je maximální rozmezí čísel na vstupu. Tedy pokud zvolíme za základ soustavy počet čísel na vstupu (z=n), dostáváme složitost $O(n*\log_n u)$. Pokud je rozmezí čísel polynomiálně velké k velikosti vstupu, lze tedy dosáhnout časové složitosti O(n). Pro neomezeně velký rozsah vstupních čísel se Radixsort nehodí.

Související články

- Count sort

Externí odkazy

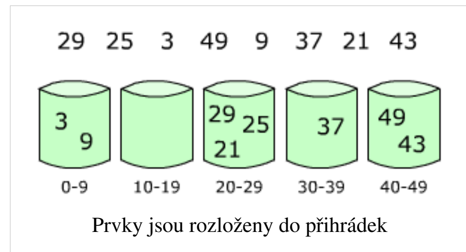
- Bakalářská práce popisující Algoritmy číslicového třídění [1], součástí práce je i měření výkonu algoritmů, kódy algoritmů v jazyce C# a studijní aplikace.
- Demonstrace a srovnání [2] Radix sortu s Bubble sortem, Merge sortem a Quicksortem implementovaný v JavaScriptu
- Stránka s vizuální demonstrací řadicích algoritmů [3]

Reference

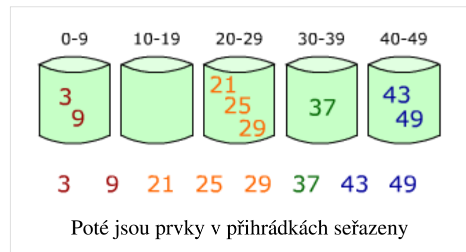
[1] http://toodle.info/radixsort/
[2] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix/
[3] http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

Bucket sort

Přihrádkové řazení (anglicky **bucket sort**) je stabilní řadicí algoritmus. Algoritmus rozděluje vstupní data na několik částí (přihrádek, anglicky bucketů). Tyto části jsou následně seřazeny.



Prvky jsou rozloženy do přihrádek



Poté jsou prvky v přihrádkách seřazeny

Předpoklady

- Algoritmus je vhodný pro rovnoměrně rozložené hodnoty vstupních dat.
- Algoritmus řazení použitý pro seřazení přihrádek musí být stabilní. Pokud stabilním není, tak ani výsledný bucket sort neřadí stabilně.

Princip

- V prvním kroku jsou vstupní data rozdělena do předem definovaného počtu přihrádek.
- V druhém kroku je na každou přihrádku volán stabilní řadicí algoritmus.
- V třetím kroku jsou jednotlivé přihrádky postupně kopírovány do výstupního pole.

Složitost

Asymptotická složitost algoritmu je $O(n * k)$, kde $k = n/m$. Vstupní data mají velikost n. Počet přihrádek je m.

Výhody

Bucket sort lze využít pro distribuované řazení. Každá přihrádka může být řazena v jiném vlákně.

Algoritmus lze použít pro řazení vstupních množin které nelze najednou načíst do paměti. Jednotlivé přihrádky mohou být řazeny ve vnitřní paměti a neaktivní přihrádky mohou být dočasně uloženy na vnější paměti.

Zdroje článků a přispěvatelé

Řadicí algoritmus *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11419761 *Přispěvatelé*: Adam Zivner, Dehet, Honza Záruba, Jj14, Mercy, Mifeet, Milan Keršláger, Mormegil, Ondratra, Slady, 22 anonymní úpravy

Stabilní řazení *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11113606 *Přispěvatelé*: Adam Zivner, Bilboq, Mercy, MiroslavJosef, Mormegil, RocketRanger, Vaclav.Makes, 3 anonymní úpravy

Řazení haldou *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11328012 *Přispěvatelé*: Adam Zivner, Iwbrowse, Josef Plch, Mormegil, Opiczka, Postrach, Pteryx, Tchoř, 8 anonymní úpravy

Shell sort *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11081791 *Přispěvatelé*: Ka04pkri, Kibitzer, Milan Keršláger, MiroslavJosef, Utar, Vaclav.Makes, Zacatecnik, 7 anonymní úpravy

Merge sort *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11368357 *Přispěvatelé*: Adam Zivner, Hobr, Jakub Onderka, Josef Plch, LiMr, MartyIX, Mercy, Miďonek, Petr Kopač, Reaperman, Riemann'sZeta, RocketRanger, Uacs451, Vaclav.Makes, Zacatecnik, 14 anonymní úpravy

Quicksort *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11370356 *Přispěvatelé*: Adam Zivner, Beretux, Bruce Shorty, Chatoooo, DaBler, Esprit, Gully, Honza889, Horst, Hypertornado, Ioannes Pragensis, Jan Špička, Limojoe, Magdulka, Melebius, Mercy, Mormegil, Nick519, PaD, Petr Kopač, Postrach, Pteryx, ToOb, Vaclav.Makes, Xyzchmelmilos, 32 anonymní úpravy

Radix sort *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11081619 *Přispěvatelé*: Adam Zivner, Honza Záruba, Milan Keršláger, Peci1, 1 anonymní úpravy

Bucket sort *Zdroj*: https://cs.wikipedia.org/w/index.php?oldid=11125655 *Přispěvatelé*: Harold, Vaclav.Makes

Zdroje obrázků, licence a přispěvatelé

Soubor:Sorting_heapsort_anim.gif *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Sorting_heapsort_anim.gif *Licence*: Creative Commons Attribution-Sharealike 2.0 *Přispěvatelé*: de:User:RolandH

Soubor:Heapsort-example.gif *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Heapsort-example.gif *Licence*: Creative Commons Attribution-Sharealike 3.0 *Přispěvatelé*: Swfung8

Image:Merge sort animation2.gif *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Merge_sort_animation2.gif *Licence*: Creative Commons Attribution-Sharealike 2.5 *Přispěvatelé*: CobaltBlue

Image:merge sort algorithm diagram.svg *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Merge_sort_algorithm_diagram.svg *Licence*: Public Domain *Přispěvatelé*: Original uploader was VineetKumar at en.wikipedia

Soubor:Sorting_quicksort_anim.gif *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Sorting_quicksort_anim.gif *Licence*: Creative Commons Attribution-ShareAlike 3.0 Unported *Přispěvatelé*: Wikipedia:en:User:RolandH

Soubor:Partition example.svg *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Partition_example.svg *Licence*: Public Domain *Přispěvatelé*: User:Dcoetzee

Image:Bucket sort 1.png *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Bucket_sort_1.png *Licence*: není známo *Přispěvatelé*: Original uploader was Booyabazooka at en.wikipedia

Image:Bucket sort 2.png *Zdroj*: https://cs.wikipedia.org/w/index.php?title=Soubor:Bucket_sort_2.png *Licence*: není známo *Přispěvatelé*: Original uploader was Booyabazooka at en.wikipedia

Licence

Creative Commons Attribution-Share Alike 3.0
//creativecommons.org/licenses/by-sa/3.0/