Algoritmická matematika

1. Definice algoritmu, vlastnosti algoritmů, základní pojmy, možnost implementace algoritmů.
2. Euklidův algoritmus, důkaz správnosti
3. Složitost algoritmů, časová a prostorová, trvání výpočetu, tabulka složitostí
4. Datové struktury (pole,strom)
5. Asymptotika, nejhorší, nejlepší a průměrný případ
6. Řadící algoritmy
7. Příklady

1. Úvod do algoritmů

Algoritmus je předpis, který se skládá z kroků a který zabezpečí, že na základě vstupních dat jsou poskytnuta požadovaná data výstupní.

Každý algoritmus musí splňovat 5 základní vlastností:

A, Konečnost – požadovaný výsledek musí být poskytnout v rozumeném čase. Za rozumný čas považujeme čas, kdy nám výsledek k něčemu bude.
B, Hromadnost – Algoritmus neřeší jeden konkrétní problém, ale širkou škálu možných vstupů
C, Jednoznačnost – každý předpis je složen z kroků, které na sebe navazují. Každý krok lze charakterizovat jako přechod z jednoho stavu algoritmu do jiného, přičemž každý stav je určen zprcovávanými daty.
D, Opakovatelnost – Při použití stejných vstupních údajů musí algoritmus dospět vždy k témuž výsledku
E, Rezultativnost – Algoritmus vede ke správnému výsledku

Programovací jazyk

Jedná se o prostředek vyjádření naších představ o průběhu výpočtu. Program je algoritmus zapsaný v některém z programovacích jazyků. Algoritmus na rozdíl od programu respektuje pouze jeden možný výsledek.

Každý vyšší programovací jazyk poskytuje uživateli 3 základní nástroje.

1, Primitivní výrazy, data (čísla, znky) a procedury (sčítání, násobení, logické operátory)

2, Mechanismy pro sestavení složitějších výrazů

3, Mechanismy pro pojmenování složitějších výrazů (definovaní nových procedur)

Základní pojmy

Instrukce – příkaz k provedení operace, kterou daný jazyk podporuje

Proměnná – hodnota, která za běhu programu mění svůj stav

Program – Sada proměných, které při výpočetním procesu mění své hodnoty

Problém – množina přípustných vstupů a výstupů

2, Euklidův algoritmus

Je algoritmus, kterým lze určit největšího společného dělitele dvou přirozených čísel, tedy největší číslo takové, že beze zbytku dělí obě čísla

Pseudokód

Mějme dána dvě přirozená čísla, uložená v proměnných u a w.

Dokud w není nulové, opakuj:

Do r ulož zbytek po dělení čísla u číslem w

Do u ulož w

Do w ulož r

Konec algoritmu, v u je uložen největší společný dělitel původních čísel.

Důkaz správnosti:

Pro $a > b$ platí : $d \mid a \,\&\, d \mid b \Leftrightarrow d \mid a - b \,\&\, d \mid b$.

Důkaz. 1. ⇒

Nechť d je společný dělitel čísel a i b. Pak existují přirozená číisla x a y, že

$a = d \cdot x$ a $b = d \cdot y$.

Pak ale d je i dělitelem $a - b$, nebot'

$a - b = d \cdot x - d \cdot y = d \cdot (x - y)$.

A triviálně d dělí i b.

Důkaz. 2. ⇐

Nechť d je dělitel $a - b$ i b. Pak existují přirozená čísla x a y, žee

$a - b = d \cdot x$ a $b = d \cdot y$.

Pak d je dělitelem i a, nebot'

$a = a - b + b = d \cdot x + d \cdot y = d \cdot (x + y)$.

3, Složitost

Intuitivní pojem složitosti je svázán s představou množství informace obsažené v daném jevu. Při realizaci výpočetních metod jsme omezeni časem a pamětí, kterou máme k dispozici. Důležitým parametrem každé výpočetní metody je její složitost.

Trvání výpočtu = počet elementárních výpočetních kroků vykonaných od zahájení do skončení výpočetu

Elementárním výpočetním krokem je obvykle instrukce (instrukce pseudokójdu nebo instrukce programovacího jazyka, ve kterém je algoritmus zaspsán

Složitost dělíme na **časovou** – tím rozumíme funkci, která každé množině vstupních dat přiřazuje počet operací vykonaných při výpočetu podle daného algoritmu. Dále na **složitost paměťovou** – defnujeme jako závilost paměťových nároků algoritmu na vstupních datech.

Pro zjednodušení někdy uvažujeme jen počet důležitých (pro algoritmus zákadních instrukcí, tj. těch které se při výpočetu vykonávají často. Jde typicky o instrucke vykonávané opakovaně, např. uvnitř cyklu

Velikost vstupní úlohy	1	100	1000
log *n*	0	2	3
n	1	10	10^3
n^2	1	10^4	10^6
n^3	1	10^6	10^9
2^n	2	10^{30}	10^{301}
n!	1	10^{157}	10^{2567}

Uvažujme algoritmy s časovými složitostmy $T_1(n) = n^2$ a $T_2(n) = 20n \cdot \log_2 n$. Stroj C_1 má rychlost výkonů instrukcí 10^{10} instrukcí / sekundu, C_2 10^7 instrukcí / sekundu.

Pro C_1 platí: $\frac{(10^7)^2}{10^{10}} = 10^4 sec = 166 min 40 sec = 2 hod 46 min 40 sec.$

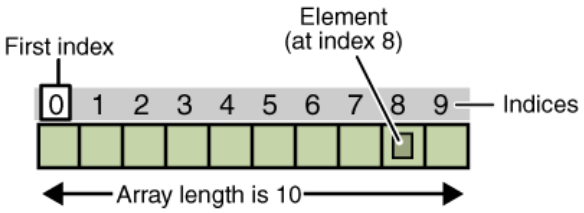
Pro C_2 platí: $\frac{20 \cdot 10^7 \cdot \log 10^7}{10^7} \approx 465 sec = 7 min 35 sec.$

Složitost	Vyjádření	Charakteristika
Konstantní	1	Konstantní doba běhu programu. Nezávisí na vstupních datech.
Logaritmická	log(*n*)	Doba běhu se mírně zvětšuje v závislosti na *N*. Řešení hledáno opakovaným dělení vstupní množiny na menší množiny (hledání v binárním stromu).
Lineární	*n*	Doba běhu programu roste lineárně s *N*. Zpracováván každý prvek, např.cyklus.
n log(*n*)	*n* log(*n*)	Doba běhu roste téměř lineárně. Opakované dělení vstupního problému na menší problémy, které jsou řešeny nezávisle (Divide and Conquer, např. třídění).
Kvadratická	n^2	Doba běhu roste kvadraticky, vhodný pro menší množiny dat. Vnořený cyklus.
Kubická	n^3	Doba běhu roste s třetí mocninou, dvojnásobně vnořený cyklus. V praxi snaha nahrazovat algoritmus předchozími dvěma kategoriemi (Greedy algoritmy)
Exponenciální	2^n	Exponenciální doba běhu. Použitelné pro množiny do *n*=30 Aplikace v kryptografii.

4, Datové struktury

Datová struktura typu POLE

Pole je posloupnost proměnných stejného datového typu uložených v paměti jako jeden celek. K prvkům pole se přistupuje pomocí indexů. Počet prvků pole může být určen nebo se měnit v době zpracování (mluvíme o statickém a dynamickém poli).



Datová struktura typu STROM

Strom je široce využívanou datovou strukturou, která představuje stromovou strukturu s propojenými uzly. Jedná se o takové uspořádání dat, kdy prvek má nejvýše jednoho předka a může mít více než jednoho následníka.

Základní prvky strom: Kořen stromu, vnitřní uzly, koncové uzly (listy)

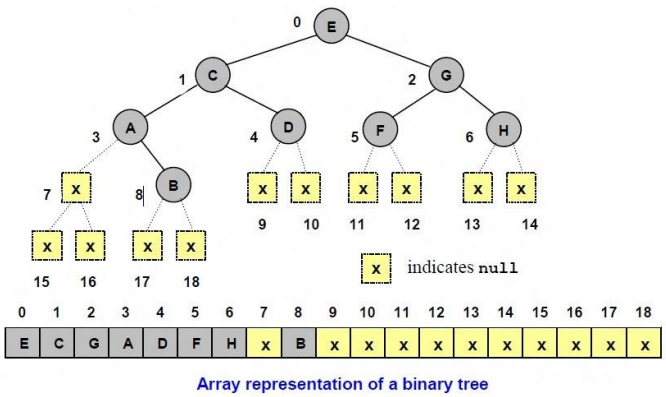
Procházení stromem:

Do šířka – procházení po hladinách (vstvách úrovní) – prohledávání do šířky

Do hloubky – procházení od kořene stromu na potomky daného vrcholu po jednotlivých větvích – prohledávání do hloubky

Binární strom

Binární strom je orientovaný graf s jedním kořenem, z něhož existuje cesta do všech vrcholů grafu. Každý vrchol má maximálně dva následovníky a s vyjímkou kořene právě jednoho předka – kořen předka nemá. Binární strom jako pole – vztahy uzlu na potomky jsou určeny funkcemi 2i+1 a 2i+2



Typy binárních stromů

Binární strom obsahuje uzly které mají nejvýš 2 syny.

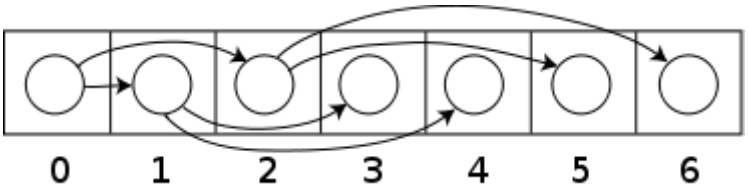
Plný binární strom každý vnitřní uzel má dva syny.

Vyvážený binární strom hloubka podstromů se od sebe liší maximálně o jedna.

Úplný binární strom vyvážený binární strom plněný zleva.

Datová struktura typu Halda

Jedná se o stromovu datovou strukturu, splňující vlasnost haldy. V kořenu stromu se nachází prvek s nejmenším klíčem nebo naopak s největším.



Další důležitou vlastností je, že pokud indexujeme (od čísla 1) prvky haldy od shora dolů, zleva doprava, pak potomci každého vrcholu jsou na indexu 2i a 2i + 1, tato vlastnost je zajištěna tím, že v haldě nevynecháváme mezery. Graficky si tedy haldu můžeme představit jako pyramidu s useknutou základnou.

Vlastnost býti haldou je rekurzivní - všechny podstromy haldy jsou také haldy. Díky této vlastnosti máme zajištěno, že se halda chová jako prioritní fronta - na vrcholek haldy vždy musí vystoupit prvek s nejvyšší prioritou (čehož se využívá u heapsortu – řazení haldou).

Obvyklé operace:

delete-max / delete-min: odstranění kořene z max-/min-heapu

increase-key, decrease-key: aktualizace klíče v max-/min-heapu

insert: vložení nového klíče

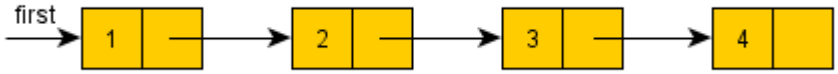
merge: spojení dvou heapů do jednoho nového

Datová struktura typu SEZNAM

Spojový seznam (Lineární seznam, Linked list) je kontejner určený k ukládání dat předem neznámé délky. Základní stavební jednotkou spojového seznamu je uzel, který vždy obsahuje ukládanou hodnotu a ukazatel na následující prvek.

Ekvivalentní definice

Jedná se o datovou strukuturu, která tvoří uspořádanou posloupnost položek. Jednolivé položky nemusí být umístěny zasebou, každá položka obsahuje odkaz na následující položku (popř. více ukazatelů).

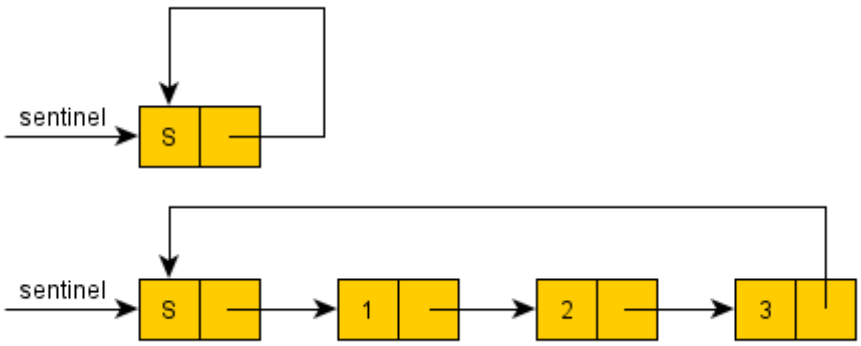


Obousměrný seznam

V obousměrně zřetězeném spojovém seznamu prvky obsahují nejen ukazatel na další prvek, ale také ukazatel na předchozí prvek. Tímto se sice poněkud zkomplikuje implementace struktury, ale toto je vyváženo zvýšenou flexibilitou, protože lze traverzovat v obou směrech.

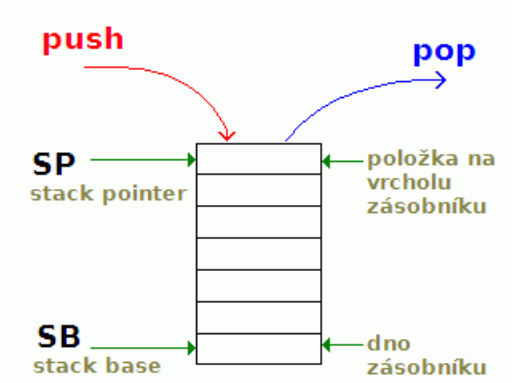
Kruhový spojový seznam

Kruhový seznam je speciálním typem seznamu zřetězeným do kruhu. Poslední uzel seznamu obsahuje místo prázdného odkazu ukazatel na první prvek listu. Kruhový seznam lze vytvořit v obou variantách zřetězení. Obousměrné zřetězení umožňuje tento typ spojového seznamu procházet libovolně v obou směrech od počátku i od konce.



Datová struktura typu ZÁSOBNÍK

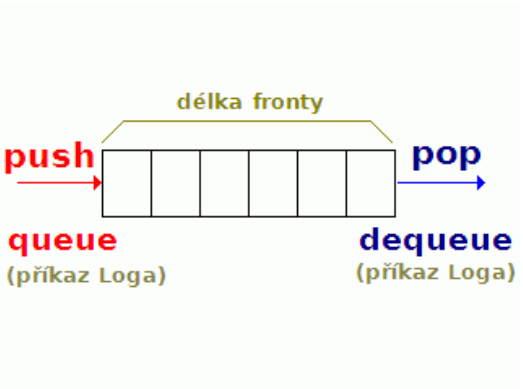
Představuje jednoduchý typ množiny, u které jepřesně určen způsob vkládání a mazání prvků. U zásobníku je uplatněn princip last-in, first out – LIFO tj. prvek, který byl poslední vložen, je jako první ze zásobníku vyzvednut. Zásobník lze přirovnat k zásobníku nábojů v pistoli1. Náboje jsou přesouvány do nábojové komory v opačném pořadí, než byly do zásobníku vloženy. V jednom okamžiku máme k dispozici pouze horní náboj nebo je zásobník prázdný. Ke spodním nábojům se lze dostat jen vyjmutím předchozích nábojů.



Ukazatel na aktuální prvek v zásobníku (posledně vložený) se nazývá vrchol zásobníku (angl. stack pointer). Opakem je dno zásobníku. Operace vložení do zásobníku se tradičně nazývá Push a vyjmutí se nazývá Pop. Jako třetí se u zásobníku implementuje dotaz Empty, který indikuje prázdnost zásobníku. Navíc se někdy přidává dotaz Top, který vrací prvek na vrcholu zásobníku, aniž by ho vyjmul (nedestruktivní varianta Pop).

Datová struktura typu FRONTA

Fronta uplatňuje mechanismus přístupu **FIFO** – first in, first out – jako první je z fronty odebrán prvek, který byl do fronty první vložen. Jde tudíž o obdobu fronty, jak ji známe z každodenního života. (V tomto okamžiku neuvažujeme prvky, které se mohou „předbíhat. Potom bychom hovořili o frontě s prioritou). Operace vložení prvku se tradičně nazývá Put, operace odebrání potom Get. Obdobně jako u zásobníku je definován dotaz Empty, který indikuje prázdnost fronty. Pokud provedeme operaci Get nad prázdnou frontou, nastane chyba podtečení. U velikostně omezené fronty může nastat i přetečení, překročíme-li při vkládání přidělený prostor.



5, Asymptotická složitost

Θ-Značení

Pro každou funkci $g(n)$, označíme zápisem $\Theta(g(n))$ množinu funkcí $\Theta(g(n)) = \{f(n)$: takových, že existují kladné konstanty c_1, c_2 a n_0 tak, že $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ pro všechna $n \geq n_0\}$.

Funkce $f(n)$ patří do množiny $\Theta(g(n))$ jestliže existují kladné konstanty c_1 a c_2 takové, že tato funkce nabývá hodnot mezi $c_1 g(n)$ a $c_2 g(n)$. Skutečnost, že $f(n)$ splňuje předcházející vlastnost zapisujeme „$f(n) = \Theta(g(n))$“. Tento zápis znamená $f(n) \in \Theta(g(n))$.

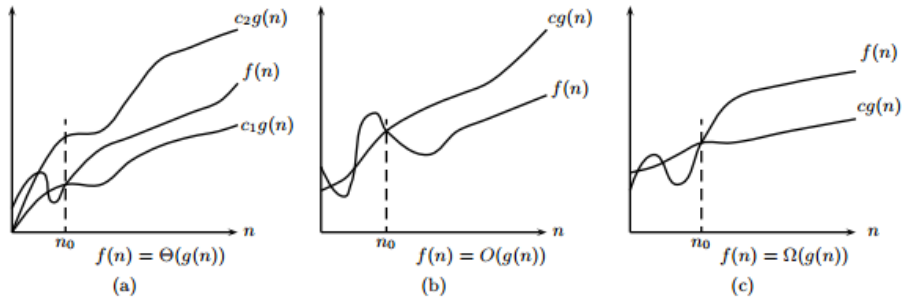
O-Značení

Θ-značení omezuje asymptoticky funkci zdola a shora. Jestliže budeme chtít omezit funkci jen shora použijeme O-značení.

Pro každou funkci $g(n)$, označíme zápisem $O(g(n))$ množinu funkcí $O(g(n)) = \{f(n)$: takových, že existují kladné konstanty c a n_0 tak, že $0 \leq f(n) \leq cg(n)$ pro všechna $n \geq n_0\}$.

Ω-Značení

Pro každou funkci $g(n)$, označíme zápisem $\Omega(g(n))$ množinu funkcí $\Omega(g(n)) = \{f(n)$: takových, že existují kladné konstanty c a n_0 tak, že $0 \leq cg(n) \leq f(n)$ pro všechna $n \geq n_0\}$.



Obrázek 3.1: Grafické vyjádření Θ, O a Ω značení

n_0 představuje nejmenší možnou hodnotu vyhovující kladeným požadavkům; každá vyšší hodnota samozřejmě také vyhovuje. **(a)** Θ notace ohraničuje funkci mezi dva konstantní faktory. Píšeme, že $f(n) = \Theta(g(n))$, jestliže existují kladné konstanty n_0, c_1 a c_2 takové, že počínaje n_0, hodnoty funkce $f(n)$ vždy leží mezi $c_1 g(n)$ a $c_1 g(n)$ včetně. **(b)** O značení shora ohraničuje funkci nějakým konstantním faktorem. Píšeme, že $f(n) = O(g(n))$, jestliže existují kladné konstanty n_0 a c takové, že počínaje n_0, hodnoty funkce $f(n)$ jsou vždy menší nebo rovny hodnotě $cg(n)$. **(c)** Ω notace určuje dolní hranici funkce $f(n)$. Píšeme, že $f(n) = \Omega(g(n))$, jestliže existují kladné konstanty n_0 a c takové, že počínaje n_0, hodnoty funkce $f(n)$ jsou vždy větší nebo rovny hodnotě $cg(n)$.

Předpoklad 1: Zanedbání multiplikativní konstanty c. Nechť f(n) je libovolna funkce a c libovolná konstanta, c > 0. Pak funkce f(n) a c · f(n) jsou označovány jako (asymptoticky) stejně rychle rostoucí funkce

Předpoklad 2: Zanedbání aditivní konstanty d. Nechť f(n) je libovolná funkce a d libovolná konstanta, d > 0. Pak funkce f(n) a f(n) + d jsou označovány jako (asymptoticky) stejně rychle rostoucí funkce

Shrnutí

Složitost algoritmu udává, jak je daný algoritmus rychlý (kolik provede elementárních operací) vzhledem k množině vstupních dat. Ke klasifikaci algoritmů se obvykle používá tzv. asymtotická složitost, což je rozdělení algoritmů do tříd složitostí, u kterých platí, že od určité velikosti dat, je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí, bez ohledu na to, jestli je některý z počítačů c-násobně výkonnější (c je konstanta).

$$1 \ll \log(n) \ll n \ll n \cdot \log(n) \ll n^k \ll k^n \ll n! \ll n^n$$

Pokud máme dva algoritmy o srovnatelné složitosti, první O(n) a druhý O(2n), tak nám stačí ten druhý pustit na 2x rychlejším stroji a nepoznáme rozdíl. Pokud ovšem nejsou ve stejné třídě složitosti, například jeden O(n) a druhý O(n^{2}), tak nám na srovnání výkonu nepomůže libovolně výkonný počítač, protože dvojnásobný objem dat bude druhému algoritmu trvat 4x tolik času, desetinásobný 100x tolik času.

Jednoduše řečeno: pokud spadají dva algoritmy do různých tříd asymptotické složitosti, pak vždy existuje takové množství dat, od kterého je asymptoticky lepší algoritmus vždy rychlejší, bez ohledu na to, kolikrát je některý z počítačů výkonnější.

U většiny algoritmů nelze říci, že jejich složitost odpovídá přesně jedné třídě, protože rychlost algoritmu závisí také na povaze dat. Z tohoto důvodu se používáme řád růstu funkcí, který zohledňuje nejhorší i nejlepší možný běh algoritmu. O algoritmu tak například řekneme, že je v Omega(n) a $O(n^2)$, což znamená, že nikdy nedoběhne rychleji než v lineárním čase, ale na druhou stranu jeho složitost není pro žádná data horší než kvadratická.

6. Řadící algoritmy

Insertion sort

Řazení vkládáním (anglicky insertion sort) je jednoduchý řadicí algoritmus založený na porovnávání.

Vlastnosti IS

A – Je to jeden z nejrychlejších algoritmů s kvadratickou časovou složitostí. Je asymptoticky pomalejší než pokročilé algoritmy jako třeba quicksort nebo mergesort, ale má jiné výhody.

B – Je efektivnější než většina ostatních $O(N^2)$ algoritmů (selection sort, bubble sort), **průměrný čas je $N^2/4$** a v nejlepším případě je dokonce lineární

C – Řadí stabilně (nemění vzájemné pořadí prvků se stejnými klíči) vyžaduje pouze O(1) paměti (kromě vlastního vstupu). Je online algoritmem, tzn. dokáže řadit data tak, jak přicházejí na vstup

D – Jednoduchá implementace a efektivita na malých množinách

Princip

1 – Posloupnost rozdělíme na seřazenou a neseřazenou tak, že seřazená obsahuje první prvek posloupnosti

2 – Z neseřazené části vybereme první prvek a zařadíme jej na správné místo v seřazené posloupnosti

3 – Prvky v seřazené posloupnosti posuneme o jednu pozici doprava

4 – Seřazenou část zvětšíme o jeden prvek. Naopak neseřazenou část o jeden prvek zleva zmenšíme

5 – Kroky 2–5 aplikujeme až do úplného seřazení neseřazené části

Pseudokód

```
INSERTION-SORT (A[ ])
1  for j ← 2 to length[A]
2       do key ← A[j]
3        Insert A[j] into the sorted sequence A[1 . . j - 1].
4       i ←  j - 1
5       while i > 0 and A[i] > key
6          do A[i + 1] ← A[i]
7             i ← i - 1
8       A[i + 1] ← key
```

Selection sort

Je implementačně jednoduchý řadicí algoritmus s časovou složitostí $O(N^2)$ na principu řazení výběrem.

Vlastnosti SS

A – Algoritmus je univerzální (pracuje na základě porovnávání dvojic prvků),

B – Pracuje lokálně (nevyžaduje pomocnou paměť)

C – Není stabilním (prvkům se stejným klíčem může změnit vzájemnou polohu)

D – Nepatří mezi přirozené řadicí algoritmy (částečně seřazený seznam se bude zpracovávat stejně dlouho jako neseřazený).

Princip

1 – Rozdělíme si posloupnost na seřazenou a neseřazenou část

2 – Najdeme prvek s nejmenší hodnotou v neseřazené části posloupnosti

3 – Zaměníme ho s prvkem na první pozici neseřazené části

4 – První prvek neseřazené části zahrneme do seřazené části a zároveň neseřazenou část zmenšíme o 1 prvek zleva

5 – Zbytek posloupnosti se uspořádá opakováním kroků 2 až 5 pro zbylou neseřazenou část

Pseudokód

```
Selection-Sort(A[0..n − 1])
1 for j ← 0 to n − 2
2 do iMin ← j
3 for i ← j + 1 to n − 1
4 do if A[i] < A[iMin] then iMin ← i
5 t ← A[j]; A[j] ← A[iMin]; A[iMin] ← t
```

Bubble sort

Je implementačně jednoduchý řadicí algoritmus. Algoritmus opakovaně prochází seznam, přičemž porovnává každé dva sousedící prvky, a pokud nejsou ve správném pořadí, prohodí je. Pro praktické účely je neefektivní

Vlastnosti BS

A – Algoritmus je univerzální (pracuje na základě porovnávání dvojic prvků)

B – Pracuje lokálně (nevyžaduje pomocnou paměť),

C – Je stabilní (prvkům se stejným klíčem nemění vzájemnou polohu)

D – Patří mezi přirozené řadicí algoritmy (částečně seřazený seznam zpracuje rychleji než neseřazený).

E - Průměrná i nejhorší asymptotická složitost bublinkového řazení je $O(n^2)$.

Princip

Algoritmus postupně prochází pole a pokud 2 prvky vedle sebe nejsou ve správném pořadí, tak je prohodí. Tímto způsobem postupně probublá nejmenší (nevětší) prvek a dochází k postupnému setřídění.

Optimalizace algoritmu

Optimalizací algoritmu je detekce prohození prvků v průchodu seznamu. V případě, že algoritmus v průchodu neprohodil žádné dva prvky, tak žádné další prvky již nikdy neprohodí. Tudíž řazení můžeme ukončit s tím, že seznam je seřazen.

Pseudokód

Bubble-Sort(A[0..n − 1])

1 for j ← 0 to n − 2

2 do for i ← n − 1 downto j + 1

3 do if A[i] < A[i − 1]

4 then temp ← A[i]; A[i] ← A[i − 1]; A[i − 1] ← temp

Coctail sort

Je implementačně jednoduchý řadicí algoritmus, vycházející z algoritmu bublinkového řazení. Od bublinkového řazení se liší tím, že prochází seznam v obou směrech. Tedy od začátku seznamu ke konci a poté zpět. Tímto postupem se předejde nevýhodě bublinkového řazení, tzv. problému želv a zajíců. Problém spočívá v tom, že vysoké hodnoty probublají na konec pole rychle, ale ty nízké postupují na začátek velmi pomalu. Porovnávání prvků běží do té doby, dokud není seznam seřazený.

Přestože se zmenšil počet zbytečných porovnání (viz. výše zmíněný problém želv a zajíců), tak asymptotická složitost zůstává stejná. Průměrná i nejhorší asymptotická složitost koktejlového řazení je O(n^2).

Vlastnosti CS

A – Algoritmus je univerzální (pracuje na základě porovnávání dvojic prvků)

B – Pracuje lokálně (nevyžaduje pomocnou paměť)

C – Je stabilní (prvkům se stejným klíčem nemění vzájemnou polohu)

D – Patří mezi přirozené řadicí algoritmy (částečně seřazený seznam zpracuje rychleji než neseřazený).

Pseudokód

```
while (bylo_serazeno !=1)
    bylo_seřazeno = ano;
    for i od 1 to (počet_prvků - 1)
      pokud seznam[i] > seznam[i + 1]
        zaměň(seznam[i], seznam[i + 1])
        bylo_seřazeno = ne;
    for i od počet_prvků downto 2
      pokud seznam[i - 1] > seznam[i]
        zaměň(seznam[i], seznam[i - 1])
        bylo_seřazeno = ne;
```

Quick sort

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 $O(N^2)$.

Vlastnosti

A – Při správné implementaci prakticky nepotřebuje dodatečnou paměť, řadí prvky přímo v poli

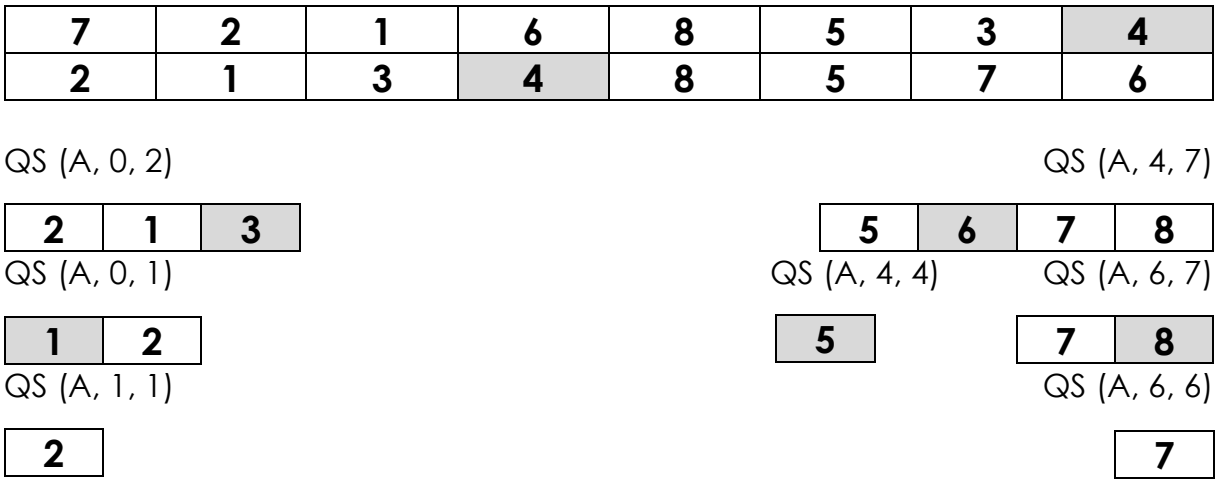
B – Jde o nestabilní algoritmus, způsob volby pivota může mít vliv na výsledek řazení

C - V průměru jde o nejrychlejší známý univerzální algoritmus pro řazení polí v operační paměti počítače

D – Omezení pravděpodobnosti nejhoršího případu slouží různé postupy volby pivota

Princip

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, což ale neznamená, že implementace musí taky použít rekurzi.



Pseudokód

Quick-Sort(A, p, r)

1 if p < r

2 then q ← Partition(A, p,r)

3 Quick-Sort(A, p, q − 1)

4 Quick-Sort(A, q + 1, r)

Partition(A,p,r)

1 x ← A[r]

2 i ← p − 1

3 for j ← p to r − 1

4 do if A[j] ≤ x

5 then i ← i + 1

6 swap(A[i], A[j])

7 swap(A[i + 1],A[r])

8 return i + 1

Praktické testy ukázaly, že QuickSort je velice rychlý při třídění rozsáhlých polí, ale zaostává oproti přirozeným algoritmům třídění (InsertSort, SelectSort) při třídění malých polí. Ukazuje se, že složitost QuickSortu má jistou minimální hodnotu, pod kterou i při třídění malého počtu prvků neklesne. Kdežto složitost přirozených algoritmů roste úměrně s počtem třídě - ných prvků. Jinými slovy do jistého počtu prvků má QuickSort větší režii (třídí pomaleji) než např. InsertSort. Tato hranice byla experimentálně stanovena na asi 12 prvků. Vyplatilo by se tedy QuickSortem třídit rozsáhlé pole, ale jakmile úseky na než se pole dělí budou kratší než zvolená mez (řekněme zmíněných 12 prvků), tento krátký úsek dotřídit InsertSortem.

Nejlepší případ QuickSortu nastane, pokud se každým dělením tříděná posloupnost rozdělí přesně na poloviny. Potom počet porovnání lze vypo- čítat podle formule (stejnou formuli splňují i ostatní algoritmy založené na strategii divide-et-impera)

Cn = 2Cn/2 + n

Výraz 2Cn/2 pokrývá třídění dvou podposloupností, n je počet porovnání všech prvků v průběhu rozdělování posloupnosti. Výše uvedenou rekurzivní formuli lze vyřešit a dostáváme

Cn ≈ n log n

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.

Vlastnosti

A - Velkou nevýhodou oproti algoritmům stejné rychlostní třídy (např. heapsort) je, že Mergesort pro svou práci potřebuje navíc pole o velikosti N.

B – Mergesort je ve většině případů pomalejší než quicksort nebo heapsort.

Mergesort je stabilní řadicí algoritmus

C – Velkou výhodou proti quicksortu je, že čas potřebný pro třídění je téměř nezávislý na počátečním řazení tříděné posloupnosti.

D – Merge sort pracuje na bázi slévání již seřazených částí pole za pomoci dodatečného pole velikosti n.

Princip

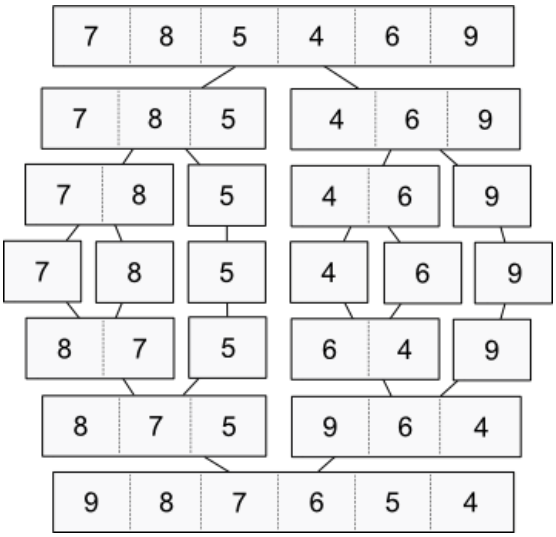
Dělicí část merge sortu má na svém vstupu celé pole. Pokud je pole sudé délky, tak jej rozdělí na dvě stejně velké části. Má-li pole lichou délku, tak bude jedna část obsahovat o prvek více než druhá. V každém případě pak algoritmus nově vzniklé části dále rekurzivně dělí. V okamžiku, kdy rekurze narazí na seznamy jednotkové velikosti, tak se zastaví. Nyní má algorimus v každé větvi k dispozici dva sousední seznamy, které obsahují jeden prvek a jsou tedy triviálně seřazeny.

Merge sort se tedy začne navracet z rekurze a při každém návratu sleje dva seznamy pomocí výše zmíněné procedury slévání. Algoritmus má jistotu, že buď slévá triviálně seřazené prvky nebo seznamy, které již byly slity. V okamžiku, kdy se merge sort plně navrátí z rekurze, tak terminuje. Pole je seřazeno od nevyšší hodnoty.

Ve skutečnosti slévá merge sort vždy dvě sousední části pole. Drží si dva ukazatele, každý na první prvek seznamu a po každém porovnání přesune jeden z prvků do pomocného pole a posune příslušný ukazatel o jedno místo (címž se dostane na nový nejvyšší prvek příslušného seznamu). Poté, co zkopíruje všechny prvky obou seznamů do pomocného pole, tak celé původní pole přepíše seřazeným seznamem (pomocným polem).

Merge – sort (1 , 6)

MS_1 (1, 3)			MS_2 (4 , 6)		MS_3 (1, 3, 5)
$MS_{1.1}$ (1 , 2)	$MS_{1.2}$ (3, 3)	m_1 (1, 2, 3)	$MS_{2.1}$ (4 , 5)	$MS_{2.2}$ (6 , 6)	m_2 (4 , 5 , 6)
$MS_{1.1A}$ (1 , 1)	$MS_{1.1B}$ (2 , 2)	$m_{1.1}$ (1, 1, 2)	$MS_{2.1A}$ (4, 4)	$MS_{2.1B}$ (5, 5)	$m_{2.1}$ (4, 4, 5)



Princip slučování

Základní idea třídění pomocí slučováním spočívá v dělení původní posloupnosti na dvě části (nejlépe o polovičním počtu prvků), jejich setřídění a poté použití metody slučování. Výsledkem je setříděná posloupnost o stejném počtu prvků jako byl v původní posloupnosti. Jak dojde k setřídění dvou rozdělených částí? Opětovným rozdělením na dvě části - v ideálním případě to budou části se čtvrtinovým počtem prvků. Z nich pak pomocí slučování dostaneme setříděné části s polovič- ním počtem prvků a následně setříděnou celou posloupnost. Rekurentně takto můžeme postupovat dál - z rozdělených „osminÿ obdržíme slučová - ním setříděné „čtvrtiny, z těch dalším slučováním „poloviny a posledním slučováním celou setříděnou posloupnost. Kdy bude dělení posloupnosti na menší a menší části končit? Až dojdeme k takovému počtu prvků, které již budou setříděny. Nejmenší setříděnou posloupností je posloupnost jednoprvková , tím jsme tedy nalezli i podmínku pro ukončení rekurentního dělení posloupnosti na menší a menší části. Každé rekurentní volání znamená rozdělení posloupnosti na dvě části a návrat zpět znamená slučování rozdělených (již setříděných) částí do jedné pomocí metody slučování.

Pseudokód

Merge-Sort (A, p, r)

```
1       if p < r
2               then q ← b(p + r)/2c
3               Merge-Sort(A, p, q)
4               Merge-Sort(A, q + 1, r)
5               Merge(A, p, q, r)
```

Merge (A,p, q, r)

```
01      n1 ← q − p + 1
02      n2 ← r − q
03      L[0..n1] a R[0..n2]
04              for i ← 0 to n1 − 1
05                      do L[i] ← A[p + i]
06              for j ← 0 to n2 − 1
07                      do R[j] ← A[q + 1 + j]
08      L[n1] ← ∞
09      R[n2] ← ∞
10      i ← 0
11      j ← 0
12              for k ← p to r
13                      do if L[i] ≤ R[j]
14                              then A[k] ← L[i]
15                              i ← i + 1
16                      else A[k] ← R[j]
17                      j ← j + 1
```

Heap sort

Heapsort (řazení haldou) je jedním z nejefektivnějších řadících algoritmů založených na porovnávání prvků s asymptotickou složitostí $O(n \cdot \log n)$. Jelikož je tato složitost zaručená, tak je heapsort vhodnější pro použití v real-time systémech než v průměrném případě rychlejší quicksort, jenž však může dosahovat složitosti v nejhorším případě až $O(n^2)$.

Vlastnosti

A – Založen na porovnání prvků

B – Zaručená časová složitost $O(n \cdot \log n)$

C – Konstantní nároky na paměť

D – Nejedná se o stabilní algoritmus

Princip

1 – Postavme haldu nad zadaným polem.

2 – Utrhněme vrchol haldy (prvek s nejvyšší prioritou - nejvyšší nebo nejnižší prvek dle způsobu řazení).

3 – Prohoďme utržený prvek s posledním prvkem haldy.

4 – Zmenšeme haldu o 1 (prvky řazené dle priority na konci pole jsou již seřazené).

5 – Opravme haldu tak, aby splňovala požadované vlastnosti (přestaly platit v momentě prohození prvků).

6 – Dokud má halda prvky opakuj krok 2 - 5.

7 – Pole je seřazené v opačném pořadí, než je priorita prvků.

Z indexu „i“ prvku lze snadno určit index Parent (i) jeho rodiče

Parent(i)

1 return $(i - 1)/2$ (zaokrouhleno dolů)

Left(i)

1 return $2i + 1$

Right(i)

1 return $2i + 2$

Pseudokód

Max-Heapify (A, i)

```
1    l ← Left(i)
2    r ← Right(i)
3    if l ≤ heap-size(A) and A[l] > A[i]
4          then largest ← l
5    else largest ← i
6    if r ≤ heap-size(A) and A[r] > A[largest]
7          then largest ← r
8    if largest 6= i
9          then swap(A[i],A[largest])
10   Max-Heapify(A, largest)
```

Build-Max-Heap(A[0..n − 1])

```
1    heap-size(A) ← n
2    for i ← n/2 − 1 downto 0
3    do Max-Heapify(A, i)
```

Heap-Sort(A[0..n − 1])

1 **Build-Max-Heap**(A)

2 for i ← n − 1 downto 1

3 do swap(A[0],A[i])

4 heapsize(A) ← heapsize(A) − 1

5 **Max-Heapify**(A, 0)

Analýza:

Všechny základní operace s haldou – vložení, zrušení, výměna (naše pomocná funkce DownHeap) prvků – vyžadují méně než 2 log n porovnání, za předpokladu, že halda má n prvků. Důkaz. Všechny tyto operace vyžadují průchod haldou od jejího kořene k listům, což představuje ne více než log n uzlů pro haldu s n prvky. Násobící faktor 2 pochází právě od funkce DownHeap, která ve svém cyklu provádí dvě porovnání. Konstrukci haldy zdola nahoru lze provést v lineárním čase.

Counting sort

Je algoritmus řazení, který je vhodný pro řazení velkého pole prvků, nabývajících jen malý počet různých diskrétních hodnot. Jedná se o výkonný stabilní řadící algoritmus se složitostí O(n + k)

Princip

Algoritmus nejprve zleva (či zprava) projde vstupní pole a pro každý prvek, na který narazí, zvýší v pomocném poli četnost výskytu tohoto prvku o 1. Poté, co má v pomocném poli zaznamenán počet výskytů, poupraví toto pole následujícím způsobem: ke každé položce přičte počet výskytů všech předchozích položek. Tím u každé položky v pomocném poli získá přesnou pozici hranice, na které bude v seřazeném poli. Následuje vlastní řazení. Algoritmus začne zprava procházet neseřazené pole a pro každý prvek, na který narazí, se podívá do pomocného pole na horní hranici pro umístění. Na tuto hranici ho umístí a zároveň ji sníží o jedna. Takto postupuje, dokud neprojde celé pole. Tím je řazení skončeno.

Časová náročnost je lineární k počtu prvků a počtu různých prvků (O(N+M)), protože musíme pro každý prvek zvýšit údaj o četnosti v pomocném poli, a pak každý prvek znovu vypsat do výsledného setříděného pole.

Paměťová náročnost je (O(M)), protože si potřebujeme pamatovat četnosti pro všechny hodnoty prvků. Celé pole, které se třídí, není potřeba mít v paměti, proto se tento algoritmus dá použít i na pole, která jsou tak velká, že se nemohou do paměti celá vejít.

Příklad: V příkladu ukážeme, jak setřídit čísel 1, 4, 2, 4, 1, 3, 1.

```
1 - 3x , 2 - 1x , 3 - 1x , 4 - 2x
1 - 3 , 2 - 4 , 3 - 5 , 4 - 7
```

```
[ ][ ][1][ ][3][ ][ ]
[ ][1][1][ ][3][ ][ ]
[ ][1][1][ ][3][ ][4]
[ ][1][1][2][3][ ][4]
[ ][1][1][2][3][4][4]
[1][1][1][2][3][4][4]
```

Příklad:

Mějme vstupní pole:

7	0	5	2	1

Potom pole posčítaných četností

1	2	3	3	3	4	4	5

B[C[A[n − 1]] − 1] ← A[n − 1], tj. B[C[A[4]] − 1] ← A[4], tj. B[1] ← 1,

B[C[A[n − 2]] − 1] ← A[n − 2], tj. B[C[A[3]] − 1] ← A[3], tj. B[2] ← 2,

B[C[A[2]] − 1] ← A[2], tj. B[3] ← 5,

B[C[A[1]] − 1] ← A[1], tj. B[0] ← 0,

B[C[A[0]] − 1] ← A[0], tj. B[4] ← 7,

Pseudokód

Counting-Sort(A, B, k)

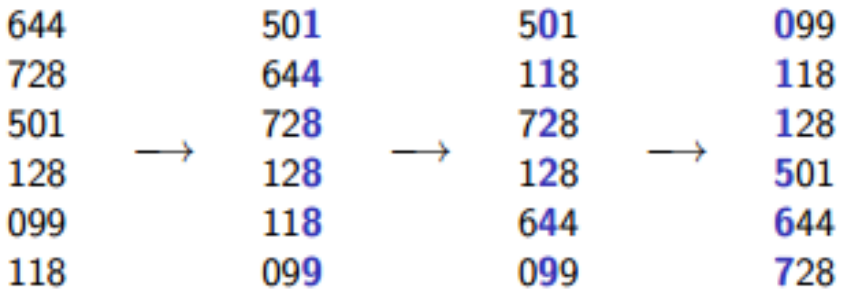
```
1     for i ← 0 to k
2           do C[i] ← 0
3     for j ← 0 to n − 1
4           do C[A[j]] ← C[A[j]] + 1
            //komentář  C[i] obsahuje počet prvků v A rovných i
5     for i ← 1 to k
6           do C[i] ← C[i] + C[i − 1]
            //komentář C[i] obsahuje počet prvků v A ≤ i
7     for j ← n − 1 downto 0
8           do B[C[A[j]] − 1] ← A[j]
9           C[A[j]] ← C[A[j]] − 1
```

Radix sort

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.

Asymptotická složitost radix sortu je $O(c \cdot C(n))$, kde m je počet znaků řazených řetězců, n je velikost dat a C(n) je složitost vnitřního stabilního řadícího algoritmu (za tímto účelem je často použit counting sort).

Příklad



Pseudokód

Radix-Sort(A, d)

```
1       for i ← 1 to d
2               do Stable-Sort(A, i)
```

```c
#include<stdio.h>

int getMax(int arr[], int n) {
    int mx = arr[0];
    int i;
    for (i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int arr[], int n) {
    int m = getMax(arr, n);

    int exp;
    for (exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void print(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

int main() {
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = sizeof(arr) / sizeof(arr[0]);
    radixsort(arr, n);
    print(arr, n);
    return 0;
}}
```

Bucket sort

Bucket sort (bin sort) je stabilní řadicí algoritmus založení na rozdělení vstupního pole do několika částí – takzvaných bucketů (přihrádek) – a seřazení těchto částí pomocí jiného stabilního řadicího algoritmu.

Princip

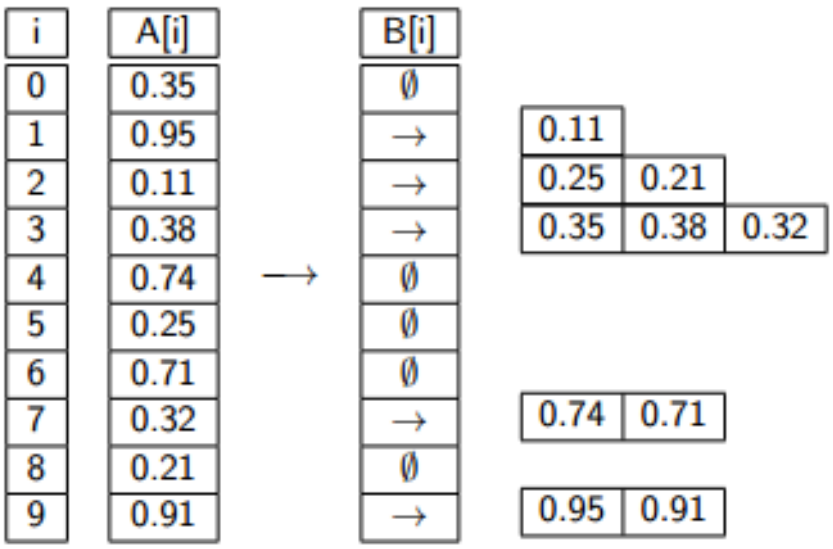
Bucket sort nejprve rozdělí vstupní pole do několika (disjunktních) přihrádek. Každá přihrádka reprezentuje určitý rozsah vstupních dat – data by měla být v optimálním případě rovnoměrně rozdělena, aby nedocházelo k situaci, kdy je jedna přihrádka přeplněna a další je prázdná. V druhé fázi pak bucket sort zavolá na každou z přihrádek stabilní řadicí algoritmus, případně rekurzivně sám sebe (bucket sort degeneruje na counting sort v případě, že je počet přihrádek stejný jako rozsah dat). Nakonec algoritmus nakopíruje postupně všechny seřazené přihrádky do výstupního pole. Připomeňme si, že pole je seřazeno, protože každá přihrádka pokrývala určitý rozsah a rozsahy se nepřekrývaly.

Příklad

1. sloupec = indexy

2. sloupec = vstupn´ı pole

3. sloupec = prvky za˘razen´e v seznamech B[i]



Bucket-Sort(A[0..n − 1])

```
1    for i ← 0 to n − 1
2        do vlož A[i] do seznamu B[bn · A[i]c]
3    for i ← 0 to n − 1
4        do Sort(B[i])
5        vlož postupně prvky z B[0], . . . , B[n − 1] do pole A
```

Bucket sort se dá využít pro řazení obrovských dat, která by nemohl načíst klasický O(n · log(n)) algoritmus najednou. Algoritmus rozdělí vstupní data do dostatečně malých přihrádek a tyto postupně řadí v paměti, zatímco neaktivní přihrádky nechává uložené ve vnější paměti (např. pevný disk).

Program v jazyce C

```c
/*
 * C Program to Sort Array using Bucket Sort
 */
#include <stdio.h>

/* Function for bucket sort */
void Bucket_Sort(int array[], int n)
{
    int i, j;
    int count[n];
    for (i = 0; i < n; i++)
        count[i] = 0;

    for (i = 0; i < n; i++)
        (count[array[i]])++;

    for (i = 0, j = 0; i < n; i++)
        for(; count[i] > 0; (count[i])--)
            array[j++] = i;
}
/* End of Bucket_Sort() */

/* The main() begins */
int main()
{
    int array[100], i, num;

    printf("Enter the size of array : ");
    scanf("%d", &num);
    printf("Enter the %d elements to be sorted:\n",num);
    for (i = 0; i < num; i++)
        scanf("%d", &array[i]);
    printf("\nThe array of elements before sorting : \n");
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);
    printf("\nThe array of elements after sorting : \n");
    Bucket_Sort(array, num);
    for (i = 0; i < num; i++)
        printf("%d ", array[i]);
    printf("\n");
    return 0;
}
```

Pořádková statistika

Statistika se zabývá jevy, které se vyznačují velkými počty – hromadností, takže se mohou projevit a odhalit zákonitosti velkého počtu pozorování.

Výběrový soubor (často jen výběr) – vybrané prvky ze základního souboru, které sledujeme místo celého souboru

Rozsah souboru — počet prvků souboru (statistických jednotek), obvykle značíme n.

Výsledkem statistického šetření, pozorování či měření je většinou nepřehledný soubor dat - souhrn čísel, která nám v uvedeném stavu nic neříkají. Snažíme se tedy charakterizovat tato data menším počtem čísel, která jsou pro nás přehlednější a lze z nich vyčíst hledané obecné vztahy a zákonitosti. K tomu setřídíme prvky souboru podle hodnot sledovaného znaku (proměnné). Vytváříme skupiny prvků souboru se stejnou hodnotou znaku a určujeme počet hodnot v dané skupině, tzv. četnost, neboli frekvenci. Při třídění získáme rozdělení četností, které udává kolikrát se jednotlivé hodnoty znaku (varianty) v souboru opakují.

Modus

V tabulce četností nebo na diagramu můžeme snadno najít nejčetnější hodnotu proměnné tzv. modus (hodnota, která má nejvyšší počet opakování); značívá se např. (písmeno se stříškou se častěji používá jako symbol odhadované hodnoty

První pořádková statistika, x(1), je nejnižší hodnota, poslední pořádková statistika je hodnota nejvyšší, x(n). (Hodnota značená x1 je první naměřená hodnota, kdežto xn je poslední naměřená hodnota.)

Kvantil, přesněji řečeno p-procentní kvantil (značíme x̃p) je hodnota, která dělí neklesající řadu pořádkových statistik na dvě části tak, že jedna obsahuje p% hodnot menších než kvantil nebo právě stejných a druhá obsahuje 100-p% (zbytek %) větších nebo právě stejných.

Medián je padesátiprocentní kvantil, x̃50% - hodnota, která dělí pořádkovou statistiku na dvě poloviny (soubor má 50% hodnot menších a 50% hodnot větších než je medián). Je-li počet prvků souboru lichý, určíme medián jako prostřední hodnotu z řady výsledků pozorování uspořádané tak, aby v ní hodnoty neklesaly. Je-li počet sudý, určíme medián jako průměr dvou prostředních hodnot.

Dolní kvartil x̃ 25% – ten jsme popsali v úvodu (dělí hodnoty seřazené v neklesající řadu na 25% hodnot pod ním, 75% hodnot nad ním). **Horní kvartil** x̃75% – (75% hodnot pod ním, 25% hodnot nad ním). Prostřední kvartil je vlastně **medián** (2x25%=50% hodnot pod ním, 50% nad ním).

Příklady

Scitani-Cisel-Desitkove(a[0..n − 1], b[0..n − 1])

1 t ← 0

2 for i ← 0 to n − 1

3 do c[i] ← a[i] + b[i] + t mod 10

4 t ← b(a[i] + b[i] + t)/10c

5 c[n] ← t

gcd-Euclid(m, n)

1 while n 6= 0

2 do r ← m mod n

3 m ← n

4 n ← r

5 return m

Set-To-Zero(A)

```
1       for i ← 0 to n − 1
2               do A[i] ← 0
```

Set-To-Zero(A)

```
1       for i ← 0 to length(A) − 1
2               do A[i] ← 0
```

Set-To-Zero(A)

```
1       for i ← 1 to n
2               do A[i] ← 0
```

```
Swap-Arrays(A[0..n − 1],B[0..n − 1])
1       for i ← 0 to n − 1
2               do temp ← A[i]
3       A[i] ← B[i]
4       B[i] ← temp
```

```
Swap-Arrays(A[0..n − 1],B[0..n − 1])
1       for i ← 0 to n − 1
2               do swap(A[i], B[i])
```

```
Randomized-Select(A, p, r, i)
1       if p = r
2               then return A[p]
3       q ← Randomized-Partition(A, p,r)
4       k ← q − p + 1
5       if i = k
6               then return A[q]
7       else if i < k
8               then return Randomized-Select(A, p, q − 1, i)
9       else return Randomized-Select(A, q + 1, r, i − q + p − 1)
```

```
Randomized-Partition(A, p, r)
1       k ← Random(p,r)
2       swap(A[k], A[r])
3       return Partition(A, p,r)
```

Generátor permutací

```
void permute(int k,int size){
   int i;
   ccc++;

   if (k==0)
      printArray(size);
   else{
      for (i=k-1;i>=0;i--){
         swap(i,k-1);
         permute(k-1,size);
         swap(i,k-1);
      }
   }

}
```