Třídící algoritmy

Ondřej Chwiedziuk, Oktáva

Úvod

K práci s počítačem také patří manipulace se soubory dat. Mezi ně například patří výsledky vědeckých měření, sportovní výsledky, účetnictví, či seznam telefonních čísel. Aby v těchto datech nevládl chaos, potřebujeme je roztřídit podle nějakého klíče. Klíč může být číslo nebo nějaký řetězec. Tyto klíče musíme umět nějak mezi sebou porovnat, k tomu nám slouží komparátor. Komparátor je funkce, která porovná dva klíče a určí, který z nich je větší. Jako příklad můžeme uvést >. Ke třídění užíváme nějaký algoritmus, kterým se budeme během procesu řídit. V utříděném souboru dat můžeme rychleji vyhledávat, např. pomocí binárního vyhledávání v čase $\mathcal{O}(\log n)$.

Záznamy na počátku tvoří nějakou posloupnost $S = (p_1, p_2, ..., p_n)$. Třídící algoritmus musí splňovat dvě podmínky. Pro každé dva prvky utříděné posloupnosti S' platí, že $p_i \leq p_j$, kde $i < j$, a zároveň je S' permutací původní posloupností S. Druhá podmínka říká, že během třídění nám nesmí zmizet, ani se objevit žádný nový člen.

Třídící algoritmy mají určité vlastnosti, které budeme chtít zkoumat. Mezi ně patří např. stabilita, časová a prostorová složitost nebo metoda řazení.

Program je **stabilní**, pokud kdykoliv jsou si prvky p_1 a p_j rovny, tak jejich vzájemné pořadí v nesetříděné posloupnosti je shodné s pořadím v setříděné posloupnosti. Tedy pokud $p_i = p_j$ pro $i < j$, pak p_i bude před p_j i nadále.

Časová složitost vyjadřuje, kolik je nutné provést operací během algoritmu v závislosti na množství dat. Např. se podívat do pole na nějaký index má časovou složitost $\mathcal{O}(1)$, podívat se na každý člen posloupnosti má časovou složitost $\mathcal{O}(n)$. U časové složitosti ignorujeme konstanty, proto $\mathcal{O}(2n) = \mathcal{O}(n)$. Rozlišujeme časovou složitost v nejlepším případě, průměrném případě a nejhorším případě. Některé algoritmy totiž mohou být neefektivní, ale v případě, že bude posloupnost již seřazená, tak jim to zabere méně času, než kdyby byla neseřazená. Naopak může existovat algoritmus, kdy složitost bude větší. Konkrétní příklady, kdy jsou tyto příklady rozdílné, uvedu v průběhu textu. Pokud budeme používat algoritmy založené na komparaci, lze dojít k závěru, že nemůže existovat algoritmus, který by běžel rychleji než v čase $\mathcal{O}(n \log n)$[1].

Prostorová složitost vyjadřuje, jak moc algoritmus využívá paměťových buněk. Pokud je prostorová složitost nějakého algoritmu $\mathcal{O}(1)$, pak počet nově zavedených proměnných a pracovních buněk je konstantní, tj. nezávislý na velikosti souboru dat.

Další věc, která nás může zajímat, je, zda algoritmus třídí prvky **na místě**. Tím rozumíme to, že prvky neopouštějí pozici, kde byly uloženy, s výjimkou konečně mnoha tzv. pracovních

[1] Důkaz k tomuto tvrzení lze nalézt v knize *Průvodce labyrintem algoritmů* od Martina Mareše

buněk, kde můžeme ukládat prvky během výměny. Dále můžeme mít libovolné množství uložených číselných proměnných, kam patří např. indexy, které prohlásíme za *pomocnou paměť* algoritmu. Prostorová složitost takového algoritmu je proto $\mathcal{O}(1)$.

Bogosort

Tento algoritmus je jeden z nejméně efektivních ze všech možných, ale dobře poslouží na praktickou ukázku výše definovaných pojmů.

Algoritmus spočívá v tom, že projde všechny permutace této posloupnosti a zkontroluje, zda jsou seřazené. Časová složitost se počítá následovně: kontrolu, zda je daná permutace seřazená, uděláme v n krocích, počet permutací je $n!$, tudíž výsledná složitost programu by byla $\mathcal{O}(n \times n!)$. Ovšem může nastat případ, kdy je už posloupnost seřazená, tudíž bude složitost $\mathcal{O}(n)$, neboť vlastně jen zkontroluje, zda je posloupnost seřazená. Prostorová složitost je $\mathcal{O}(1)$, jelikož nevyžaduje nic víc, než jednu paměťovou buňku na výměnu prvků a v lepším případě i počítá, kolik permutací bylo provedeno[2]. Algoritmus také není stabilní, jelikož může (obecně) existovat permutace, kde bude prohozeno pořadí členů se stejným klíčem.

Zde uvedu jednoduchý kód v Pythonu, kdyby měl někdo odvahu tento algoritmus vyzkoušet, ale v praxi tohle nikdy neimplementujte!

```
import random

# Definice těla algoritmu
def bogosort(pole):
    while (is_sorted(pole) == False):
        permutace(pole)
    return pole

# Kontrola, zda je posloupnost srovnaná
def is_sorted(pole):
    n = len(pole)
    for i in range(n-1):
        if (pole[i] > pole[i+1] ):
            return False
    return True

# Permutace (náhodná, nekontroluje, zda již taková permutace existovala)
def permutace(pole):
    n = len(pole)
    for i in range (n):
        r = random.randint(0,n-1)
        pole[i], pole[r] = pole[r], pole[i]

pole = [3, 6, 1, 4, 2, 5]
print(bogosort(pole))
```

[2]Kdyby tohle algoritmus neměl, může se stát, že nikdy nenarazí na vhodnou permutaci a běžel by nekonečně dlouho, to opravdu nechcete.

Selectsort

O trochu inteligentnější algoritmus je selectsort. Patří mezi jednodušší algoritmy, co se týče implementace, ale patří mezi pomalejší. Spočívá vlastně pouze v hledání nejmenšího prvku v posloupnosti a jeho zařazení na začátek. Časová složitost je $\mathcal{O}(n^2)$. Není stabilní, jelikož může nastat okamžik, kdy se jeden prvek vymění s minimem, ale ocitne se za prvkem se stejnou hodnotou.

Podrobnější popis algoritmu vypadá následovně: vezměme první člen posloupnosti. postupně ho porovnáváme s ostatními. V okamžiku, kdy narazíme na menší, zapamatujeme si ho a následně zbytek posloupnosti porovnáváme s tímto prvkem atd. V okamžiku, kdy zkontrolujeme celou posloupnost, vyměníme nejmenší prvek s prvním prvkem. V případě, že byl první prvek nejmenší, neděláme nic. Postoupíme na druhou pozici a opakujeme. Takto postupujeme, dokud celou posloupnost nesrovnáme.

Kód v Pythonu:

```
def selectsort(pole):
    for i in range(len(pole)):
        pom = i
        for j in range(i+1,len(pole)):
            if pole[pom] > pole[j]:
                pom = j

        pole[i], pole[pom] = pole[pom], pole[i]

    return pole

pole = [3,6,1,4,2,5]

print(selectsort(pole))
```

V případě, že bychom ho chtěli vylepšit, nabízí se, že budeme hledat maximum i minimum najednou, popř. že směrem doprava budeme hledat minimum, následně budeme postupovat doleva a hledat maximum.

Insertionsort

Tento algoritmus patří mezi jedny z nejjednodušších a když se vhodně použije, může být i velice efektivní. Jeho síla spočívá v tom, že velice snadno dokáže srovnat téměř srovnané posloupnosti, přičemž jeho implementace je velice snadná. Mezi další jeho vlastnosti patří stabilita a časová složitost $\mathcal{O}(n^2)$.

Funguje na principu zařazování prvků do srovnané posloupnosti. V prvním kroku si posloupnost rozdělíme na nesrovnanou a srovnanou, která obsahuje právě jeden prvek, tj. ten první. Následně postupně prochází nesetříděnou část a zařazuje její prvky do té setříděné tak, aby i po vložení byla setříděná.

Jeho vylepšenou verzí je shellsort, který představím později. Implementace je velice snadná, jak ukazuje následující kód:

```
def insertionsort(pole):
    for i in range(1, len(pole)):
        pom = pole[i]
        j = i - 1
        while j >= 0 and pom < pole[j]:
            pole[j+1] = pole [j]
            j -= 1
        pole[j+1] = pom
    return pole

pole = [3,6,1,4,2,5]
print(insertionsort(pole))
```

Shellsort

Shellsort je v podstatě vylepšení insertionsortu, které běží v jiném než kvadratickém čase. Byl vymyšlen v roce 1959 mužem jménem Donald Shell. Normální insertionsort vždy zařazuje sousední prvky, zatímco shellsort používá větší mezeru mezi prvky, přičemž ta se postupem času snižuje, až klesne na jedna a tím algoritmus degeneruje na obyčejný insertionsort. V čem je tedy jeho výhoda? Oproti insertionsortu dokáže rychle zařazovat malé prvky na začátek a velké na konec, což ho činí o něco rychlejší.

Co se týče časové složitosti, není možné zde dát nějaké jednoznačné číslo. Složitost totiž velice závisí i na způsobu, jak se bude měnit mezera mezi členy posloupnosti. Donald Shell původně navrhoval mezeru $n/2$ na začátek a s každým průchodem, aby se velikost mezery zmenšila na polovinu. Naneštěstí se ukázalo, že takto zvolená mezera má stále časovou složitost $\mathcal{O}(n^2)$. V následujících letech se objevily jiné řady, kterými by se velikost mezery měnila, které měly složitost např. $\mathcal{O}(n^{\frac{3}{2}})$, $\mathcal{O}(n^{\frac{4}{3}})$, či $\mathcal{O}(n \log^2 n)$. Zatím nejlepšího výsledku dosáhla řada, kterou vytvořil Marcin Ciura. Ta byla vytvořena experimentálně, tudíž není znám žádný předpis, kterým by se měla posloupnost řídit. První členy posloupnosti vypadají následovně: $(1, 4, 10, 23, 57, 132, 301, 701, 1750)$. Je potřeba zmínit, že se bavíme o průměrné složitosti, nejhorší možná je stále $\mathcal{O}(n^2)$.

Jelikož se jedná o algoritmus, který má mezery mezi porovnávanými prvky, tak se řadí mezi nestabilní algoritmy. Jeho implementace bude užívat mezeru stanovenou Donaldem Shellem, tj. $n/2$:

```
def shellsort(pole):
    gap = len(pole)//2 #Zvolení prvotní mezery
    while gap > 0:
        for i in range(gap,len(pole)):
            pom = pole[i]
            j = i
            while  j >= gap and pole[j-gap] > pom:
                pole[j] = pole[j-gap]
                j -= gap
            pole[j] = pom
        gap //= 2 #Zmenšení mezery
    return pole
```

```
pole = [3,6,1,4,2,5]
print(shellsort(pole))
```

Bubblesort

Bubblesort spočívá v tom, že nechává "probublat" největší prvky na konec posloupnosti. Patří mezi stabilní algoritmy, na druhou stranu je velice pomalý a neefektivní, jeho složitost je $\mathcal{O}(n^2)$. Z toho důvodu se příliš nepoužívá. Napříč tomu patří mezi důležité algoritmy, jelikož jeho rychlost lze zvýšit různými modifikacemi, popř. jeho princip zkombinovat s jinými a vytvořit tak velice efektivní algoritmus.

Základní algoritmus spočívá v tom, že procházíme pole neustále dokola a porovnáváme vždy dva sousední prvky. Když je prvek vpravo menší než vlevo, prohodíme je. Důvod, proč se mu říká bublinkový algoritmus je ten, že dříve nebo později narazíme na nejtěžší prvek, který se bude bude posouvat neustále doprava, dokud nedosáhne konce. Tento prvek pak v té chvíli můžeme považovat za pomyslnou bublinku, která se pohybuje polem.

Způsob, jak lze algoritmus vylepšit, může být následující: mějme proměnnou, která obsahuje pravdivostní hodnotu. Na začátku každého procházení polem bude jeho hodnota True. Když v průběhu cyklu prohodíme dva prvky, pak tuto hodnotu nastavíme na False. V okamžiku, kdy neprovedeme žádnou výměnu, tj. hodnota proměnné na konci cyklu bude True, pak je posloupnost již setříděná a my nemusíme dále pokračovat. Následující implementace užívá i tohoto vylepšení:

```
def bubblesort(pole):
    for i in range(len(pole) - 1):
        swap  = False #indikátor srovnané posloupnosti
        for j in range(0,len(pole)-i-1):
            if pole[j] > pole[j+1]:
                pole[j], pole[j+1] = pole[j+1], pole[j]
                swap = True
        if not(swap):
            return pole
    return pole

pole = [3,6,1,4,2,5]
print(bubblesort(pole))
```

Následující algoritmy jsou modifikované verze bubblesortu, tudíž pro ně platí stejné vlastnosti jako pro bubblesort, jen jsou poněkud rychlejší z důvodu nižších konstant.

Combsort

Modifikace spočívá ve zvolení větší mezery mezi dvěma prvky. Zatímco bubblesort porovnává dva sousední, combsort porovnává prvky od sebe vzdálené. Zrychlení spočívá v rychlejším odsunu malých hodnot do čela posloupnosti, které bubblesort zbytečně zpomalují.

```
def nextgap(gap):   #Mění velikost mezery
```

```
    gap = int(gap*10/13)
    if gap < 1:
        return 1
    return gap

def combsort(pole):
    gap = len(pole)
    swap = True

    while gap!=1 or swap == 1:
        gap = nextgap(gap)
        swap = False
        for i in range(len(pole) - gap):
            if pole[i] > pole[i + gap]:
                pole[i], pole[i + gap] = pole[i + gap], pole[i]
                swap = True
    return pole

pole = [3,6,1,4,2,5]
print(combsort(pole))
```

Shakersort

Další možností zrychlení bubblesortu je nechat malé prvky probublat na začátek posloupnosti a ty velké na konec zároveň.

```
def shakersort(pole):
    start, end = 0, len(pole) - 1
    swap = True
    while swap:
        swap = False
        for i in range(start,end):
            if pole[i] > pole[i+1]:
                pole[i], pole[i+1] = pole[i+1], pole[i]
                swap = True

        if not(swap):
            return pole
        else:
            swap = False

        end-=1
        for i in range(end-1, start-1, -1):
            if pole[i] > pole[i+1]:
                pole[i], pole[i+1] = pole[i + 1], pole[i]
                swap = True
        start+=1
```

```
    return pole

pole = [3,6,1,4,2,5]
print(shakersort(pole))
```

Navzdory tomu, že je jednou z nejlépe optimalizovaných verzí bubblesortu, stále je poněkud pomalý. Další možnost, jak ho zrychlit, je zkombinovat shakersort a combsort. Tuto verzi již zde uvádět nebudu, ale můžete si ji zkusit naprogramovat sami v rámci procvičení v třídících algoritmech. Tento algoritmus je poslední, který má průměrnou časovou složitost $\mathcal{O}(n^2)$ a horší. Všechny následující algoritmy budou časově efektivnější.

Mergesort

Tento algoritmus je založen na dvou myšlenkách: *divide et impera*, v překladu rozděl a panuj, a slévání dvou posloupností v jednu. Na pochopení, proč, oproti předchozím metodám, běží v čase $\mathcal{O}(n \log n)$, je potřeba pochopit fungování algoritmu samotného.

Nejprve myšlenka slévání. Mějme dvě posloupnosti, které jsou již seřazené. K nim si vytvořme prázdné pole o takové velikosti, aby pojala obě posloupnosti. Na počátku se podíváme na první člen u každé posloupnosti. Porovnáme je, vybereme ten menší z nich a vložíme na začátek pole. V té posloupnosti, ze které jsme vyjmuli první člen, se posuneme na druhý člen. Znovu porovnáme, vybereme menší a zařadíme ho do prázdného pole na druhou pozici. Takto pokračujeme, dokud nezařadíme jednu z posloupností. V tom okamžiku vezmeme zbytek zbylé posloupnosti a zařadíme na konec.

Jenže my nemáme dvě setříděné posloupnosti, ale jednu nesetříděnou. Jak zde aplikovat výše zmíněný postup? Rozdělme si posloupnost na menší o velikosti jeden člen. Taková posloupnost je vždy srovnaná. Zde vezmeme vždy dva sousední prvky, porovnáme a slijeme v jednu posloupnost. Takhle postupujeme, dokud neslijeme všechny do jedné posloupnosti.

Časová složitost algoritmus odvodíme následovně: Nejprve si vezmeme část, která se stará o slévání algoritmu. Ta pouze přesouvá prvky z původních polí do nového. Z toho vyplývá, že běží v $\mathcal{O}(n + m)$, kde n a m je počet prvků v první a druhé posloupnosti. V i-té iteraci spotřebuje čas na slití prvků $\mathcal{O}(2^i)$. V rámci jedné iterace používáme slévání pouze $2^{-i}n$, tudíž v jedné iteraci proběhne algoritmus v čase $\mathcal{O}(n)$. Jelikož je počet iterací závislý na počtu prvků v původním poli, konkrétně $2^i = n$, pak celková složitost algoritmu je $\mathcal{O}(n \log n)$.

Zároveň si pozorný čtenář povšiml, že jsme během algoritmu alokovali dodatečnou paměť o velikosti n, protože jsme netřídili *na místě*, takže správně tuší, že se zde nám časová složitost algoritmu snížila na úkor paměťové, která narostla na $\mathcal{O}(n)$. Existují však metody, jak tento neduh alespoň částečně odstranit. Stabilita algoritmu závisí na tom, jak jej implementujeme. Tento algoritmus také využívá rekurze, což může být výhodou i slabinou.

```
def mergesort(pole):
    #Rozdělení pole
    if len(pole) > 1:
        stred = len(pole) // 2
        left = pole[:stred]
        right = pole[stred:]
        #Rekurze
```

```
        mergesort(left)
        mergesort(right)
        #indexy v polích
        i, j, k = 0, 0, 0
        #Merge
        while i < len(left) and j < len(right):
            if left[i] < right [j]:
                pole[k] = left[i]
                i+=1
            else:
                pole[k] = right[j]
                j+=1
            k+=1
        #Dolití zbývajících prvků
        while i < len(left):
            pole[k] = left[i]
            i+=1
            k+=1
        while j < len(right):
            pole[k] = right[j]
            j+=1
            k+=1
    return pole

pole = [3,6,1,4,2,5]
print(mergesort(pole))
```

Timsort

Tento algoritmus staví na základech mergesortu a insertionsortu. Využívá toho, že insertionsort je velice efektivní na malých polích, zatímco mergesort běží v $\mathcal{O}(n \log n)$, tudíž je rychlejší na polích větších. Timsort tyto vlastnosti kombinuje tím způsobem, že si nejprve pole rozdělí na malé části o předem dané veliosti, ty srovná pomocí insertionsortu a takto srovnané kusy slije pomocí mergesortu.

Mezi jeho vlastnosti patří časová složitost časová složitost $\mathcal{O}(n \log n)$, prostorová složitost záleží na způsobu implementace, v nejjednodušším případě je $\mathcal{O}(n)$. Algoritmus je také stabilní a velice dobře běží i na částečně srovnaných polích.

Timsort je velice často používaný na srovnávání reálných dat a je zakomponován do metody sort() v Pythonu, či Arrays.sort() v Javě. Z toho důvodu je velice užitečné zde předvést jeho implementaci. Nevypadá úplně snadně, ale většina kódu je jen modifikace částí kódu mergesortu a insertionsortu.

```
#Modifikace insertionsortu na rovnání úseků
def insertionsort(pole, left, right):
    for i in range(left + 1, right + 1):
        pom = pole[i]
        j = i - 1
```

```
        while j >= left and pom < pole[j]:
            pole[j+1] = pole [j]
            j -= 1
        pole[j+1] = pom
    return pole
#Merge slévá dvě pole v jedno
def merge(left,right):
    pole = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            pole.append(left[i])
            i+=1
        else:
            pole.append(right[j])
            j+=1
    #Dolití zbývajících prvků
    while i < len(left):
        pole.append(left[i])
        i+=1
    while j < len(right):
        pole.append(right[j])
        j+=1
    return pole

def timsort(pole):
    min_run = 32 #Doporučené rozmezí je mezi 32 a 48
    #Insertionsort
    for i in range(0, len(pole), min_run):
        insertionsort(pole, i, min((i + min_run - 1), len(pole) - 1))
    size = min_run
    #Mergesort úseků
    while size < len(pole):
        for start in range(0, len(pole), size * 2):
            mid = start + size - 1
            end = min((start + size*2 - 1), (len(pole) - 1))
            pom = merge(pole[start:mid + 1],pole[mid + 1:end + 1])
            pole[start:start + len(pom)] = pom
        size *= 2
    return pole

#Pole potřebuje velké množství dat
import random
pole = [random.randint(0,1000) for _ in range(10000)]

print(timsort(pole))
```

Heapsort

Heapsort, jak již název napovídá, používá ke svému fungování haldu, anglicky *heap*. Halda je takový binární strom, která má následující vlastnosti:

všechna patra jsou zaplněna prvky, s výjimkou posledního, které nemusí,

poslední patro je zaplněno zleva,

pro každý prvek platí, že synové jsou menší nebo rovni otci.

Z toho vyplývá, že kořen, tj. prvek, který nemá otce, je maximem. Algoritmus heapsortu vybírá maxima a zařazuje je na konec pole, takže funguje podobně jako selectsort. Od něj se liší tak, že hledání maxima v haldě má časovou složitost $\mathcal{O}(1)$, jelikož se stačí podívat na kořen.

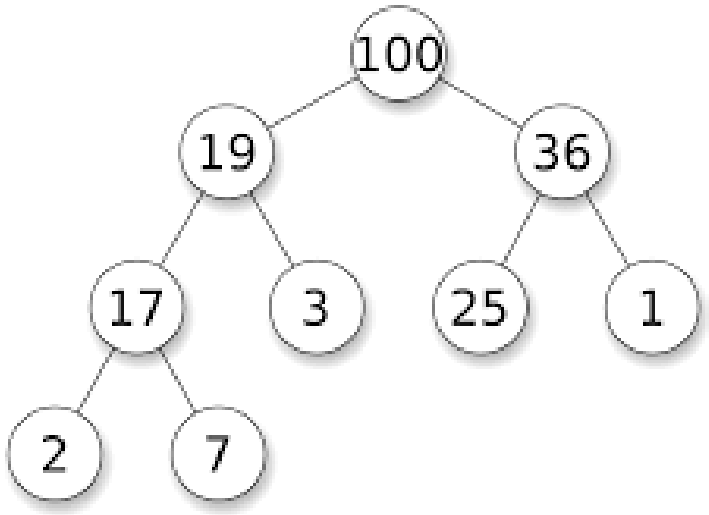

Figure 1: Halda

Důležité je zmínit, že halda se indexuje následovně: první je kořen, od něj se postupuje o patro níže, přičemž se čísluje zleva doprava. Díky tomu, že je předem dané, jaké bude číslování, nemusíme pro haldu alokovat paměť, ale můžeme si ji trikem definovat na poli.

Abychom mohli haldy při třídění využít, musíme ji nejprve sestavit. Nejprve roztřiďme naše pole do binárního stromu. Do kořene vložíme první prvek. Nyní si definujme funkci *Up*. Ta kontroluje, zda jsou synové menší než otec, jestliže ne, dojde k prohození otce s větším ze synů. Nyní přiřadíme druhý prvek a aplikujeme funkci *Up*, abychom opravili haldu. Postupně budeme přiřazovat nové a nové prvky z původní posloupnosti. Někdy sem může stát, že prohození otce a syna nevytvoří haldu, pouze posune problém s velikostí otce a syna o patro výš, proto je potřeba *Up* aplikovat tak dlouho, dokud nebude halda opravená.

V okamžiku, kdy bude halda sestavená, odtrhneme kořen, zařadíme ho na konec a místo něj dáme poslední prvek. Jelikož je poslední prvek menší než některý z jeho synů, budeme aplikovat funkci *Up*. Zdejší oprava bude podobná té při sestavování haldy, ale nebudeme postupovat odspodu, ale shora.

Opravovat haldu budeme muset $2n$-krát, jednou při sestavování, podruhé při rozebírání, přičemž při opravě haldy se díváme nanejvýš tolikrát, kolik je pater. Počet pater je $\log_2 n$,

neboť má každý otec 2 syny. Jak jsme si řekli, odtrhnout kořen zabere časovou složitost $\mathcal{O}(1)$. Proto můžeme dojít k závěru, že časová složitost celého algoritmu je $\mathcal{O}(n \log n)$. Paměťová složitost je $\mathcal{O}(1)$ za předpokladu, že halda bude pouze myšlená. Tento algoritmus má jednu slabinu, je nestabilní.

```
#Definice funkce kontrolující, zda je nějaký syn větší než otec
def up(pole, n, i):
    largest = i
    l = 2 * i + 1     # index levého syna
    r = 2 * i + 2     # index pravého syna
    #Kontrola velikosti synů
    if l < n and pole[i] < pole[l]:
        largest = l

    if r < n and pole[largest] < pole[r]:
        largest = r
    # Případná výměna syna a otce
    if largest != i:
        pole[i],pole[largest] = pole[largest],pole[i]
        up(pole, n, largest) #Rekurze, kontroluje další změny

def heapsort(pole):
    # Konstrukce haldy
    for i in range(len(pole) // 2 - 1, -1, -1):
        up(pole, len(pole), i)
    # Rozklad haldy
    for i in range(len(pole)-1, 0, -1):
        pole[i], pole[0] = pole[0], pole[i]  #Odtržení maxima
        up(pole, i, 0)
    return pole

pole = [3, 6, 1, 4, 2, 5]
print(heapsort(pole))
```

Quicksort

Velice populární algoritmus na řazení čísel je quicksort. Tento algoritmus je velice rychlý, běží v čase $\mathcal{O}(n \log n)$, může být při vhodně zvolené implementace stabilní, prvky řadí na místě a je snadno implementovatelný. Spočívá v tom, že roztřídí prvky vůči prvnímu na větší a menší.

Algoritmus funguje následovně. Vybereme jeden prvek, říkejme mu pivot. Následně projdeme celou posloupnost a všechny prvky, jež jsou větší menší než pivot, zařadíme před něj, zbytek zařadíme za něj. Tímto způsobem nesetřídíme posloupnost, pouze rozdělíme na dvě poloviny a během toho vyřadíme pivot. Následně rekurzivně aplikujeme algoritmus na zbylé dvě poloviny posloupnosti. Stejně jako heapsort a mergesort, i zde využíváme toho, že jednoprvkové pole je setříděné. Program v Pythonu vypadá následovně:

```
def quicksort(pole):
    #Když je pole jednoprvkové, nebo prázdné
```

```
    if len(pole) < 2:
        return pole

    pivot = 0 #Pozice pivotu
    for i in range(1, len(pole)): #Rozdělení pole
         if pole[i] <= pole[0]:
              pivot += 1
              pole[i], pole[pivot] = pole[pivot], pole[i]

    pole[pivot], pole[0] = pole[0], pole[pivot]
    #Rekurze (příkaz rozdělen na dva řádky)
    pole = quicksort(pole[0:pivot]) + [pole[pivot]
    ] + quicksort(pole[pivot+1:len(pole)])
    return pole

#Pole potřebuje velké množství dat
import random
pole = [random.randint(0,100) for _ in range(20)]

print(quicksort(pole))
```

Podobně jako mergesort, tak i quicksort dosahuje časové složitosti $\mathcal{O}(n \log n)$ pomocí toho, že rozdělí pole na menší úseky s poloviční velikostí. Jen je díky vyřazování pivotu rychlejší. Jenže abychom dosáhli maximální rychlosti, je potřeba zvolit prvek, který bude mediánem daného pole. Hledání mediánu je časově náročné a algoritmus zbytečně zpomaluje. Z toho důvodu se volí většinou ten první. Dá se dokázat, že průměrné časové složitosti $\mathcal{O}(n \log n)$ algoritmus dosahuje i v okamžiku, kdy prvek není mediánem. Problém nastává, když je pole již seřazené. V tom okamžiku je pivot minimem (či maximem), tudíž rozdělení na větší a menší dopadne tak, že pole si vůbec nesnížíme. V tom okamžiku algoritmus dosahuje časové složitosti $\mathcal{O}(n^2)$, mluvíme tedy o časové složitosti v nejhorším případě. Využití téhle slabiny může být fatální. Představme si server, jehož účelem je řadit velká pole pomocí quicksortu. Za normálních okolností by dosahoval veliké rychlosti. V okamžiku, kdyby ho chtěl někdo napadnout, tak na server pošle velkou spoustu již setříděných polí. Rychlost serveru se rázem propadne na $\mathcal{O}(n^2)$, server nápor nezvládne a spadne. Nehledě na to, že některé jazyky mají omezenou paměť na rekurzi, kterou když program přesáhne, tak spadne. Tohle můžeme opatřit jiným výběrem pivotu, jenže třeba výběrem posledního prvku si příliš nepomůžeme. Můžeme vybírat třeba prvek, co leží v jedné polovině, nebo v jedné třetině, jenže když bude útočník znát naše kódy, není problém, aby pole setřídil tak, aby byl quicksort co nejpomalejší. Proto bychom chtěli vybírat prvek náhodně. Tomu se říká *randomizovaný quicksort*, který se opravdu používá. Jeho slabinou je fakt, že čísla generovaná procesorem ve skutečnosti nejsou náhodná, ale generují se podle předem daného algoritmu. V okamžiku, kdy znáte seed programu, dokážete generovat stejnou posloupnost pseudonáhodných čísel. Náhodná čísla ale můžete generovat i jinak, třeba pomocí teploty na procesoru, nebo v armádě pomocí rozpadů atomů.

Introsort

Introsort je algoritmus, který řeší problém quicksortu při řazení srovnaných polí. Nejprve pole třídí podle quicksortu, přičemž počítá, kolik rekurzí bylo provedeno. V okamžiku, kdy rekurze

přesáhne předem danou mez, dojde ke změně algoritmu, zpravidla na heapsort, ale může jím být třeba timsort, nebo mergesort, který zbylé nesrovnané části dotřídí. Doporučená mez je přibližně $2 \log n$.

Tento algoritmus dosahuje složitosti $\mathcal{O}(n \log n)$, může třídit na místě a není stabilní. Jeho hlavní předností je rychlost a aplikovatelnost na předtříděná pole bez významné ztráty rychlosti. Jeho implementace vypadá následovně:

```
from math import *
#Definice funkce kontrolující, zda je nějaký syn větší než otec
def up(pole, n, i):
    largest = i
    l = 2 * i + 1     # index levého syna
    r = 2 * i + 2     # index pravého syna
    #Kontrola velikosti synů
    if l < n and pole[i] < pole[l]:
        largest = l

    if r < n and pole[largest] < pole[r]:
        largest = r
    # Případná výměna syna a otce
    if largest != i:
        pole[i],pole[largest] = pole[largest],pole[i]
        up(pole, n, largest) #Rekurze, kontroluje další změny

def heapsort(pole):
    # Konstrukce haldy
    for i in range(len(pole) // 2 - 1, -1, -1):
        up(pole, len(pole), i)
    # Rozklad haldy
    for i in range(len(pole)-1, 0, -1):
        pole[i], pole[0] = pole[0], pole[i]  #Odtržení maxima
        up(pole, i, 0)
    return pole

def introsort(pole, count=0,n=None):
    #Když je pole jednoprvkové, nebo prázdné
    if len(pole) < 2:
        return pole
    if n is None:
        n = len(pole)
    pivot = 0 #Pozice pivotu
    count+=1 #Počítání vnoření
    for i in range(1, len(pole)): #Rozdělení pole
         if pole[i] <= pole[0]:
              pivot += 1
              pole[i], pole[pivot] = pole[pivot], pole[i]

    pole[pivot], pole[0] = pole[0], pole[pivot]
```

```
        #Když program přesáhne počet vnoření, přepne se do heapsortu
        if count >= round(2 * log(n)):
                pole = heapsort(pole[0:pivot]) + [pole[pivot]
        ] + heapsort(pole[pivot+1:len(pole)])
        #Rekurze (příkaz rozdělen na dva řádky)
        else:
            pole = introsort(pole[0:pivot],count,n) + [pole[pivot]
            ] + introsort(pole[pivot+1:len(pole)],count,n)
        return pole

#Pole potřebuje velké množství dat
import random
pole = [random.randint(0,100) for _ in range(20)]

print(introsort(pole))
```

Bitonicsort

Bitonicsort, nebo také bitonic mergesort, je svým způsobem velmi zvláštní algoritmus. Od předchozích algoritmů se liší v tom, že se jedná o *paralelní algoritmus*. Takový algoritmus provádí více operací v jeden okamžik.

Pro pochopení algoritmu je třeba vědět, co je *bitonická posloupnost*. Posloupnost je bitonická, pokud po spojení do cyklu obsahuje dva monotónní úseky[3]. Pro lepší představu, bitonická posloupnost může vypadat následovně:

$$(3, 4, 7, 8, 6, 5, 2, 1)$$

Samotný algoritmus funguje následovně: Nejprve si rozdělíme posloupnost na dvojice. Liché dvojice seřadíme vzestupně, sudé sestupně. Každá dvojice je teď bitonická posloupnost.

$$(3, 4), (8, 7), (5, 6), (2, 1)$$

Nyní spojíme dvě sousední posloupnosti. Porovnáme prvky, které mezi sebou mají rozestup poloviny délky posloupnosti, poté čtvrtiny, atd. Dohromady je potřeba $n \log n$ porovnání, aby se posloupnost srovnala. Tento postup aplikujeme rekurzivně, dokud nedojde ke srovnání posloupnosti celé.

Pokud si spočítáme, kolik potřebujeme porovnání, dojdeme ke složitosti $\mathcal{O}(n \log^2 n)$, což je pomalejší, než mergesort, nebo quicksort. Jak jsem již řekl, tento algoritmus je paralelní, což umožňuje provádět několik na sobě nezávislých úkonů naráz. Pokud použijeme paralelní implementaci, tak dokážeme provádět n porovnání naráz. Díky tomu se naše časová složitost dokáže snížit na $\mathcal{O}(\log^2 n)$. Jeho slabinou je nestabilita a prostorová složitost $\mathcal{O}(n \log^2 n)$. Následující implementace navíc třídí pouze posloupnosti mocniny dvou.

[3] Viz `https://kam.mff.cuni.cz/~koncicky/statnice/bc/out/Obecna_informatika.pdf` na straně 39.

```
#Funguje jen pro mocniny 2 (potřeba vylepšit)
#Vyměňuje prvky v závislosti na paritě posloupnosti
def swap(pole,i,j,d):
    if (d == 1 and pole[i] > pole[j]) or (d == 0 and pole[i] < pole[j]):
        pole[i], pole[j] = pole[j], pole[i]
    return pole
#Spojuje dvě bitonické posloupnosti (rekurze)
def merge(pole,l,count,d):
    if count > 1:
        k = count // 2
        for i in range(l,l + k):
            swap(pole,i,i+k,d)
        merge(pole,l,k,d)
        merge(pole,l + k,k,d)
    return pole
#Tělo algoritmu (rekurze)
def bitonicsort(pole, l=0, count=None, d=1):
    if count is None:
        count = len(pole)
    if count > 1:
        k = count // 2
        bitonicsort(pole,l,k,1)
        bitonicsort(pole,l+k,k,0)
        merge(pole,l,count,d)
    return pole

pole = [3,4,7,8,6,5,2,1]
print(bitonicsort(pole))
```

Countingsort

Všechny předchozí algoritmy používaly k řazení prvků jejich vzájemné porovnávání. Jak jsme si řekli již v úvodu, žádný komparační algoritmus nemůže běžet v čase rychlejším než $\mathcal{O}(n \log n)$. Pokud náš klíč má specifickou vlastnost, např. je to celé číslo v pevně daném intervalu, dokážeme vymyslet algoritmus, který dokáže běžet v čase rychlejším.

Countingsort používá pro řazení prvků přídavné pole, přičemž prochází posloupnost a prvky zařazuje na příslušné místo v přídavném poli. Tohle zařazení využívá toho, že velikost členu je zároveň indexem v přídavném poli. Na příslušném místě proto zvýší hodnotu o jedna. Countingsort se říká algoritmu proto, že počítá četnost členů posloupnosti. Jakmile zařadí všechny prvky původního pole, projde přídavné pole a vytvoří novou posloupnost.

U tohoto algoritmu je důležité znovu zmínit jednu vlastnost, kterou má mít řadící algoritmus, tj. že nová posloupnost je permutací té původní. V případě špatné implementace algoritmu se může stát, že nějaké prvky ztratí.

Výhodou algoritmu je jeho vysoká rychlost a jednoduchost. Běží v čase $\mathcal{O}(n + r)$, kde r značí velikost přídavného pole. Slabinou může být, že je určitě nestabilní a aplikovatelný jen na specifické klíče. Prostorová složitost bude $\mathcal{O}(r)$. Jeho implementace je velice snadná, jak si

můžete povšimnout sami.

```
    def countingsort(pole, r):
        pom = [0 for _ in range(r)] #Definice přídavného pole
        for i in pole: #Naplnění pole
            pom[i]+=1
        count = 0
        for i in range(r):
            while pom[i] !=0:
                pole[count] = i
                count+=1
                pom[i]-=1
        return pole

    pole = [3,6,1,4,2,5]
    print(countingsort(pole,7))
```

Bucketsort

Další nezmíněná nevýhoda countingsortu je, že nedokáže uchovat dodatečné informace, což může být problém, pokud máte databázi a algoritmus používáte na setřídění dat podle klíče. Countingsort vám totiž jenom srovná čísla, ale neřekne vám, který číslo patří ke které informaci. Tento problém řeší bucketsort. Jak již název napovídá, tento algoritmus nebude počítat četnosti, ale bude prvky zařazovat do přihrádek, v češtině se mu také říká přihrádkové řazení.

Dá se usoudit, že běží ve stejné časové složitosti jako countingsort, prostorová bude $\mathcal{O}(n + r)$. Pozorný čtenář si také všimne, že algoritmus je, při zařazování prvků v přihrádce na konec, stabilní, což z něj činí velice efektivní a použitelný algoritmus.

```
def countingsort(pole, r):
    #Definice přídavného pole
    bucket = []
    for _ in range(r):
        bucket.append([])

    for i in pole: #Naplnění pole
        bucket[i].append(i)

    pole = []
    for i in bucket:
        pole+=i
    return pole

pole = [3,6,1,4,2,5]
print(countingsort(pole,7))
```

Bucketsort se dá také využít na zmenšení procházených polí. Dejme tomu, že se klíč pohybuje v rozmezí od 0 do 1, ale pole je příliš obrovské, než aby se dalo srovnat v rozumném čase i s rychlými algoritmy. V tu chvíli se aplikuje bucketsort, který původní posloupnost naporcuje v

lineárním čase na spoustu menších posloupností, které můžeme již rozumně roztřídit. Výše uvedená implementace lze velice snadno na tenhle druh práce upravit.

Další možností, jak algoritmus modifikovat, je tzv. *Lexikografický bucketsort*. Představme si, že klíčem není nějaké konkrétní číslo, nýbrž na k-tice čísel. V tom případě budeme používat bucketsort rekurzivně. Nejprve ho aplikujeme na poslední číslo (číslici nejnižšího řádu), následně postupujeme až na první (nejvyšší řád). Časová složitost bude $\mathcal{O}(k(n+r))$ a paměťová $\mathcal{O}(kn+r)$. Např. program, který by řadil k-tice cifer, vypadá následovně:

```
from math import log10, ceil

def lex(pole, rekurze=0):
    bucket = []
    for _ in range(10): #Vytvoření přihrádek
        bucket.append([])

    pom = []
    for i in pole: #Naplnění pole
        l = ceil(log10(i))
        if l > rekurze:
            bucket[l - (i // 10**rekurze % 10)].append(i)
        else: #Vyřadí prvky z rekurze
            pom.append(i)

    #Zpětně naplní pole
    for i in bucket:
        if i != []:
            i = lex(i, rekurze+1)
        pom+=i
    return pom

pole  = [5421,5389,6854,2536,2486]
print(lex(pole))
```

Radixsort

Rozviňme myšlenku řazení pomocí bucketsortu. Radixsort, česky také číslicové řazení, nejprve rozdělí čísla podle řádu a následně je roztřídí pomocí lexikografického bucketsortu. Výhoda radixsortu vůči bucketsortu nastává, kdy počet přihrádek je řádově větší než počet klíčů. Bucketsort totiž musí procházet ohromnou spoustu prázdných přihrádek, které zabírají místo paměti a zbytečně zpomalují program.

Matematicky přesnější popis algoritmu vypadá následovně: nejprve si napišme čísla v soustavě o vhodném základu z. Z každého čísla se tak stane k-tice cifer z rozsahu $0, 1, ..., z-1$, kde $k = \text{ceil}(\log_z r)$, kde funkce *ceil* vrací nejbližší vyšší celé číslo. Tyto k-tice čísel poté střídíme lexikograficky. Takové třídění proběhne v čase $\mathcal{O}((\log_z r)\cdot(n+z)) = \mathcal{O}(\frac{\log r}{\log z}(n+z))$.

Jaký si tedy zvolit základ? Pokud si vybereme nějaký konstantní, třeba $z = 10$, jak jsem učinil já v implementaci níže, pak by vyšla časová složitost $\mathcal{O}(n \log r)$. Sice to není špatná složitost,

ale jelikož $r > n$, pak je algoritmus pomalejší než algoritmy komparativní.

Praktičtější se zdá zvolit z takové, aby bylo závislé na n. Pak jsme schopni dosáhnout složitosti $\mathcal{O}(\frac{\log r}{\log n} n)$. Pokud by čísla byla polynomiálně velká vzhledem k n, tedy $r \geq n^{\alpha}$, kde α je pevně stanoveno, pak by $\log r \leq \alpha \log n$, což implikuje, že by časová složitost byla lineární. Polynomiálně velká čísla lze tudíž třídit v lineárním čase. Lze také dokázat, že i v lineárním prostoru.

```
from math import log10, ceil

def lex(pole, rekurze=0):
    bucket = []
    for _ in range(10): #Vytvoření přihrádek
        bucket.append([])

    pom = []
    for i in pole: #Naplnění pole
        l = ceil(log10(i))
        if l > rekurze:
            bucket[l - (i // 10**rekurze % 10)].append(i)
        else: #Vyřadí prvky z rekurze
            pom.append(i)

    #Zpětně naplní pole
    for i in bucket:
        if i != []:
            i = lex(i, rekurze+1)
        pom+=i
    return pom

def radixsort(pole):
    #Definice přídavného pole
    l = ceil(log10(max(pole))) #Nejvyšší řád

    bucket = []
    for _ in range(l):
        bucket.append([])

    for i in pole: #Naplnění pole
        bucket[ceil(log10(i))-1].append(i)

    #Aplikace lexikologického bucketsortu na jednotlivé přihrádky
    pole = []
    for i in bucket:
        if i != []:
            i = lex(i)
        pole+=i
    return pole
```

```
pole  = [1278,5235,1248,5267,12,24,775,123]
print(radixsort(pole))
```

Třídění řetězců

Pomocí lexikografického bucketsortu můžeme třídit nejenom čísla, nýbrž i řetězce znaků. Modifikace musí splňovat novou podmínku, tj. že když budeme mít dva řetězce, z nich je jeden kratší, přičemž jsou na délce kratšího z nich stejné, pak ten kratší je menší. Příklad: máme řetězce "kolovrat" a "kolo". V případě seřazení musí být "kolo" před "kolovrat".

Na vstupu dostaneme n retězců $r_1, ..., r_n$ délek $l_1, ..., l_n$. Nechť $l = max(l_1, ..., 1_n)$, a $s = \sum_{i=1}^{n}(l_i)$, což bude celková délka řetězce. Znaky abecedy si očíslujeme od 1 do r. Kdyby byly všechny řetězce stejně dlouhé, pak bude složitost $\mathcal{O}(ln) = \mathcal{O}(s)$, tedy lineární velikosti vstupu.

V případě různě dlouhých řetězců nastává problém, jelikož je nelze porovnávat jako k-tice čísel. Řešení může být je doplnit prázdnými mezerami. Problém nastává v tom, že časová složitost tímto krokem naroste na $\mathcal{O}(s^2)$.

Dalším řešením bude program vytvořit tak, že při každém vnoření do rekurze vyřadí ty řetězce, které jsou kratší než minimální délka v daném vnoření, vyřadí z řazení a zařadí je do čela příslušné přihrádky[4]. Časová složitost bude při konstantním počtu znaků v abecedě $\mathcal{O}(s)$. Paměťová složitost bude také $\mathcal{O}(s)$. Z toho důvodu lze řazení považovat za velice efektivní.

Implementace níže řadí slova složená z 26 znaků abecedy. Není problém modifikovat algoritmus na řazení znaků podle celého ASCII.

```
def letter(slovo,pozice): #ASCII, velká a malá písmena
    for i in range(26):
        if ord(slovo[pozice]) in [i+65,i+97]:
            return i

def lex(pole, rekurze=0):
    bucket = []
    for _ in range(26): #Vytvoření přihrádek
        bucket.append([])

    pom = []
    for i in pole: #Naplnění pole
        if len(i) > rekurze:
            bucket[letter(i,rekurze)].append(i)
        else: #Vyřadí prvky z rekurze
            pom.append(i)

    #Zpětně naplní pole
    for i in bucket:
        if i != []:
            i = lex(i, rekurze+1)
        pom+=i
```

[4]Tuto metodu jsem již implementoval do lexigrafického bucketsortu.

```
        return pom

pole  = ["ahoj","trideni","prihradka","algoritmus"]
print(lex(pole))
```

Porovnání algoritmů

Na závěr bych chtěl porovnat jednotlivé algoritmy z hlediska stability, časové a prostorové složitosti. Je potřeba říct, že ne vždy znamená nižší průměrná složitost také nižší rychlost. Vhodné použití algoritmu se totiž váže i k velikosti srovnávaného pole, typu klíče, či konstantám, které Landauovo značení nezohledňuje.

Algoritmus	*Čas*	*Pomocná paměť*	*Stabilita*
Bogosort	$\mathcal{O}(n \times n!)$	$\mathcal{O}(1)$	Ne
Selectsort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Ne
Insertionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Ano
Shellsort	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(1)$	Ne
Bubblesort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Ano
Combsort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Ne
Shakersort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Ano
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Ano
Timsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Ano
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Ne
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Ano
Introsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Ne
Bitonicsort	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log^2 n)$	Ne
Countingsort	$\mathcal{O}(n + r)$	$\mathcal{O}(r)$	Ne
Bucketsort	$\mathcal{O}(n + r)$	$\mathcal{O}(n + r)$	Ano
Lexikografický bucketsort	$\mathcal{O}(k(n + r))$	$\mathcal{O}(kn + r)$	Ano
Radixsort	$\mathcal{O}(s)$	$\mathcal{O}(s)$	Ano
Řetězcový bucketsort	$\mathcal{O}(s)$	$\mathcal{O}(s)$	Ano

Mezi nejrychlejší komparativní algoritmy patří quicksort, ale jelikož je jeho časová složitost na setříděných polích $\mathcal{O}(n^2)$, je vhodné ho zkombinovat s jiným algoritmem, např. mergesortem, nebo heapsortem. Heapsort, ač patří mezi algoritmy s $\mathcal{O}(n \log n)$, ale má veliké konstanty, což ho velice zpomaluje. Mergesort je také rychlý, ale netřídí na místě. Lze sice opravit, aby tak činil, ale jeho rychlost se sníží tak moc, že se to nevyplatí. Z jednodušších algoritmů stojí za použití pouze insertionsort. Bubblesort má vedle kvadratické složitosti i veliké konstanty.

Kombinací základních algoritmů vznikají tzv. hybridní algoritmy, které používají myšlenky různých algoritmů a skládají je do jedné metody. Mezi takové algoritmy můžeme řadit timsort či introsort. Další, zde neuvedený, je např. blocksort, který běží v čase $\mathcal{O}(n \log n)$ a kombinuje vlastnosti mergesortu a insertionsortu jiným způsobem než timsort. Tento algoritmus je ovšem již velice obtížné naprogramovat.

Bucketsort a radixsort jsou velice efektivní algoritmy, ale jejich rychlost není závislá jen na množství porovnávaných klíčů, nýbrž i na jejich délce a rozmezí, ve kterém se tyto klíče pohybují. Navíc je nelze aplikovat univerzálně.

Zdroje

MAREŠ, Martin a Tomáš VALLA. *Průvodce labyrintem algoritmů.* Praha: CZ.NIC, z.s.p.o., 2017. CZ.NIC. ISBN 978-80-88168-19-5.

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-selection-sort-razeni-cisel-podle-velikosti

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-bubblesort-probublavani-trideni-cisel

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-insertion-sort-trideni-cisel-podle-velikosti

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-heap-sort-trideni-cisel-podle-velikosti

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-merge-sort-trideni-cisel-podle-velikosti

https://www.itnetwork.cz/navrh/algoritmy/algoritmy-razeni/algoritmus-quick-sort-razeni-cisel-podle-velikosti

https://www.algoritmy.net/article/154/Shell-sort

https://www.algoritmy.net/article/44152/Bogosort

https://www.algoritmy.net/article/51222/Block-Merge-Sort

https://en.wikipedia.org/wiki/Sorting_algorithm

https://en.wikipedia.org/wiki/Quicksort

https://en.wikipedia.org/wiki/Merge_sort

https://en.wikipedia.org/wiki/Introsort

https://en.wikipedia.org/wiki/Heapsort

https://en.wikipedia.org/wiki/Insertion_sort

https://en.wikipedia.org/wiki/Timsort

https://en.wikipedia.org/wiki/Selection_sort

https://en.wikipedia.org/wiki/Shellsort

https://en.wikipedia.org/wiki/Bubble_sort

https://en.wikipedia.org/wiki/Comb_sort

https://en.wikipedia.org/wiki/Cocktail_shaker_sort

https://en.wikipedia.org/wiki/Bogosort

https://www.pythonpool.com/bitonic-sort-python/

https://en.wikipedia.org/wiki/Bitonic_sorter

https://kam.mff.cuni.cz/~koncicky/statnice/bc/out/Obecna_informatika.pdf