Algoritmus minimaxu

- hra dvou hráčů s úplnou informací
- bílý a černý se pravidelně střídají na tahu
- cíl algoritmu: pro aktuální pozici zvolit nejvýhodnější tah

- **strom hry** (stavový prostor):
  kořen = aktuální pozice
  počet synů = počet možných tahů
  liché hladiny – na tahu je bílý, sudé – na tahu je černý
  list = konec hry, některý z hráčů vyhrál (příp. remíza)

„Malé“ hry (piškvorky na hodně omezené ploše, odebírání zápalek)

→ lze postavit celý strom hry, listy ohodnotit podle výsledku hry (1 = vyhrál bílý, -1 = vyhrál černý, příp. 0 = remíza).

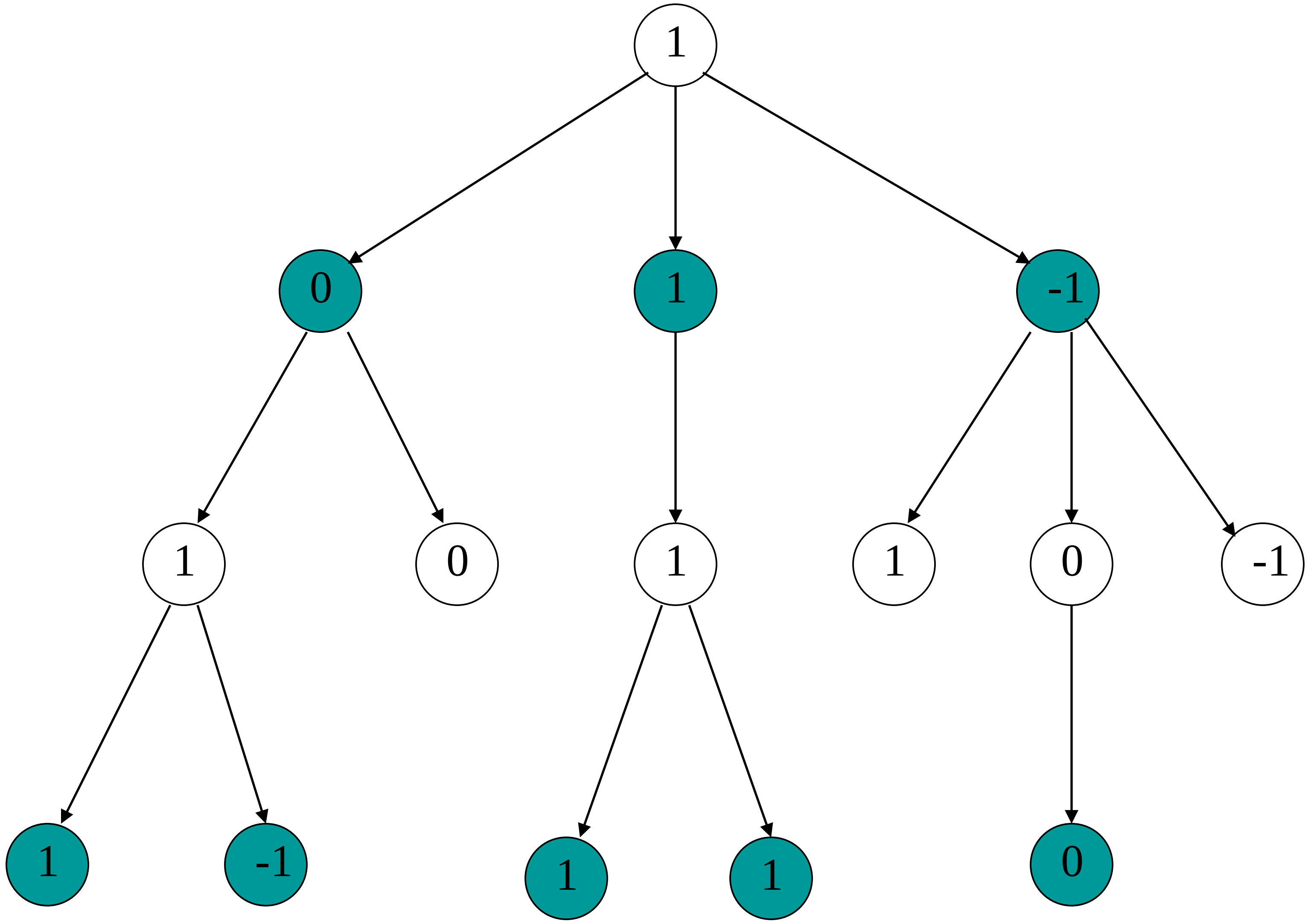
Z hodnot listů se algoritmem minimaxu postupně zdola určí hodnota všech ostatních uzlů, až se získá hodnota kořene
(= nejlepší výsledek, který si může začínající hráč vynutit).

Algoritmus minimaxu:
- hodnota uzlu, kde je na tahu bílý = *maximum* z hodnot jeho synů (bílý si vybere ten tah, který je pro něj nejlepší)
- hodnota uzlu, kde je na tahu černý = *minimum* z hodnot jeho synů (také černý si vybere ten tah, který je zase pro něj nejlepší)

Strom hry se ohodnocuje zdola od listů po vrstvách, v jednotlivých vrstvách stromu se počítají střídavě minima a maxima z hodnot synů, dokud se nezíská hodnota kořene.

Pokud je v kořeni na tahu bílý a kořen bude mí hodnotu 1, může si bílý vynutit vítězství. Pokud získá kořen hodnotu 0, může si začínající bílý vynutit aspoň remízu. Bude-li mít kořen hodnotu –1, bílý si vítězství ani remízu vynutit nemůže (což neznamená, že nemůže vyhrát – ale jedině při chybě černého).

Zvolený tah: ten, který vede z kořene do toho uzlu, kde je ohodnocení stejné jako v kořeni.


1
0
1
-1
1
0
1
1
0
-1
1
-1
1
1
0

Programová realizace
- strom hry je rozsáhlý, není vhodné (nebo ani nelze) vygenerovat ho najednou celý, uložit do paměti a pak zdola po vrstvách procházet
- algoritmus je realizován prohledáváním (tzn. zároveň i vytvářením) stromu hry **do hloubky**, vždy při návratu z podstromu se přepočítá hodnota uzlu, do něhož se vracíme

Minimax – na jednotlivých hladinách stromu hry se hodnoty uzlu počítají střídavě jako minimum a maximum z hodnot synů (musíme vědět, který hráč je na tahu).

Negamax (jiná realizace téhož algoritmu) – hodnota každého uzlu se počítá jako maximum z hodnot synů, před předáním výsledné hodnoty z uzlu nahoru se změní její znaménko.

$$\min(a,b) = -\max(-a,-b)$$

„Velké“ hry (šachy)
→ lze postavit jenom část stromu (zvolený počet hladin),
listy ohodnotit podle statické ohodnocovací funkce
(ocenit materiál každého hráče, příp. vhodné bonusy za pozici):

kladná hodnota = pozice výhodnější pro bílého
(čím vyšší hodnota, tím výhodnější pozice)
záporná hodnota = pozice výhodnější pro černého
(čím vyšší absolutní hodnota, tím výhodnější pozice)
příp. 0 = vyrovnaná pozice

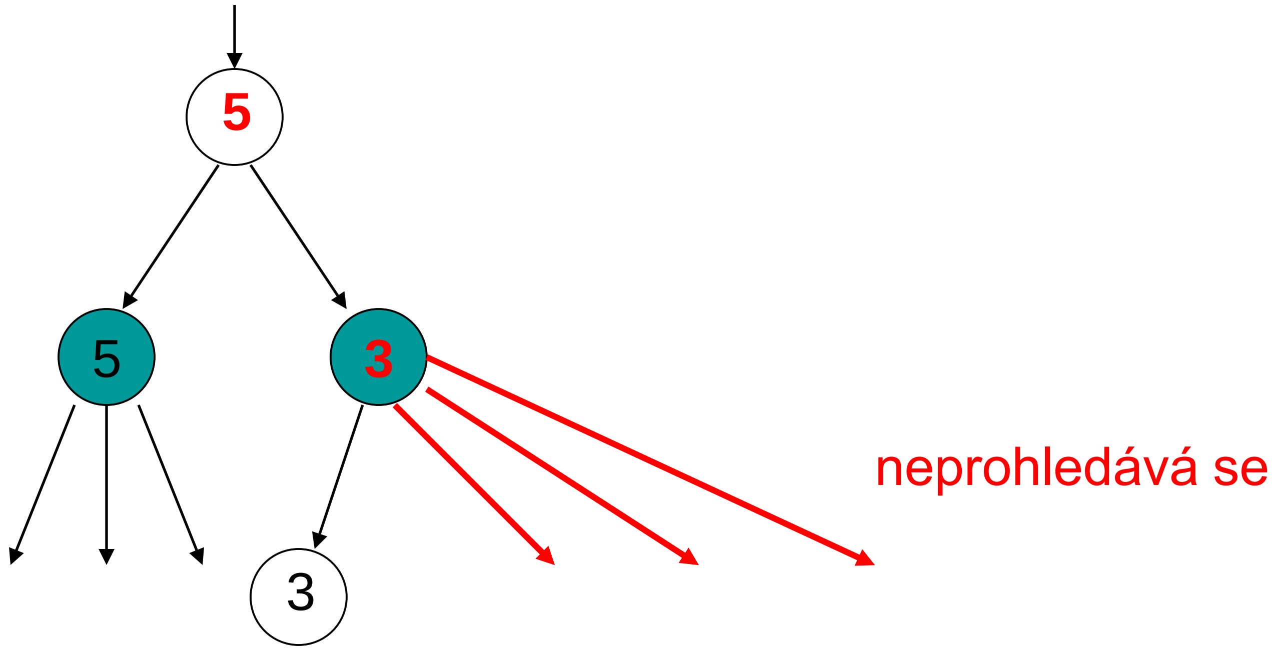
Dále použijeme algoritmus minimaxu stejně jako v předchozím případě.

Vylepšení ohodnocení: pokud by se stala listem „živá“ pozice (tah do ní vedoucí zásadně mění situaci na hracím plánu, např. braní figury v šachu), rozvíjí se zde lokálně strom hry dále do hloubky.

Zrychlení výpočtu – ořezávání stromu hry

- ztrátové: po několika málo vrstvách provést statické ohodnocení pozic a část nejhorších pozic odmítnout (tedy dále nerozvíjet), podrobněji do větší hloubky analyzovat jen nadějnější pozice
$\rightarrow$ kaskádové vyhodnocování

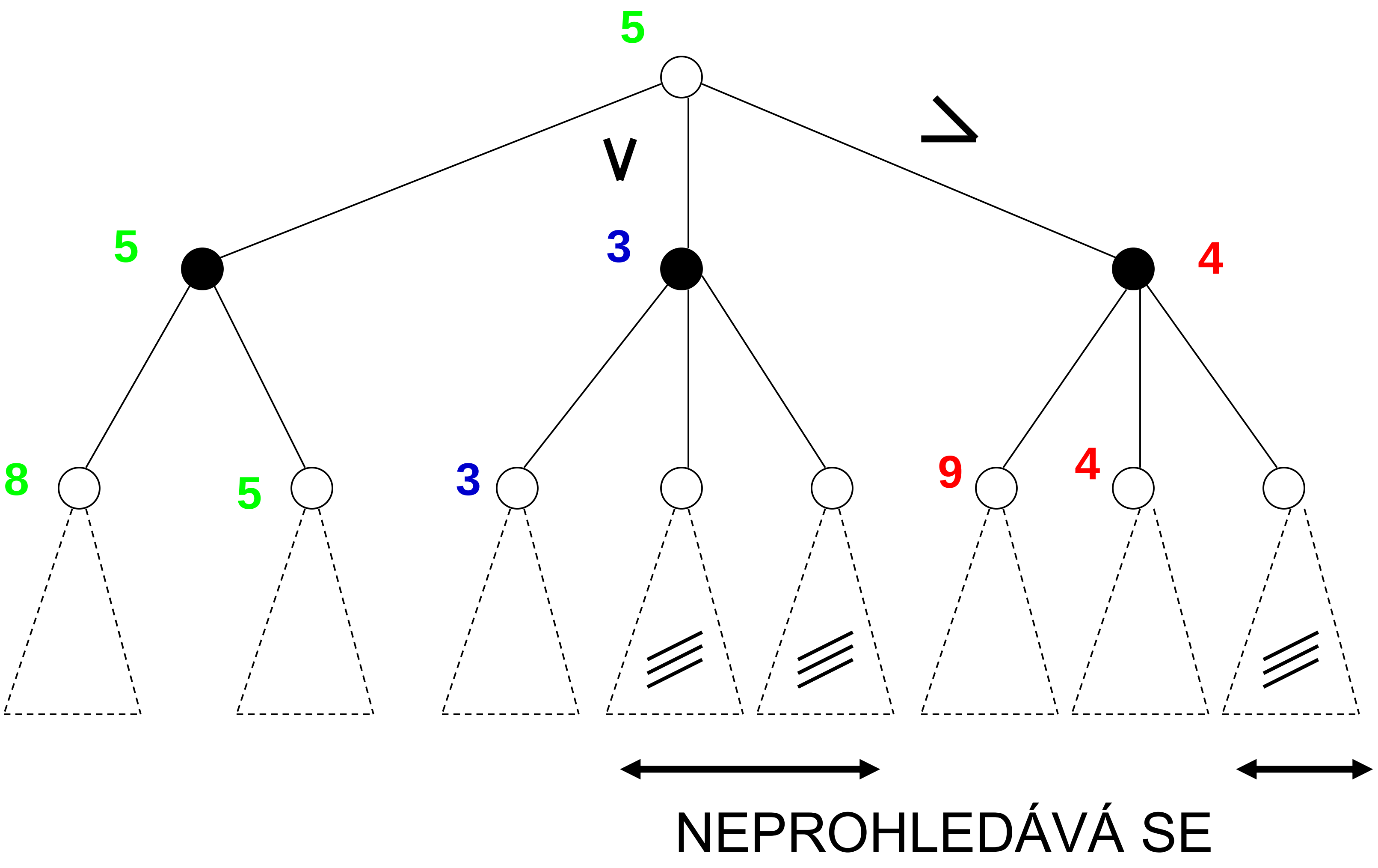
- bezztrátové: *alfa-beta-prořezávání*



- analogické prořezávání se provádí při jednom průchodu stromem hry ve všech vrstvách stromu, pro bílého i pro černého

- pokud hráč v každé pozici zkouší přednostně ty tahy, které jsou pro něj výhodnější, alfa-beta-prořezávání je výrazně účinnější (prořeže se více větví, prochází se mnohem menší část stromu)

$\rightarrow$ uspořádat možné pokračovací tahy v každé pozici podle statické ohodnocovací funkce nebo podle nějaké heuristiky


5
5
3
4
8
5
3
9
4
NEPROHLEDÁVÁ SE

„Rozděl a panuj“
- metoda rekurzivního návrhu algoritmu (programu)

Problém se rozdělí na dva podproblémy stejného typu, ale menší velikosti, z jejichž řešení lze snadno získat řešení původního problému. Každý podproblém je buď už triviální a vyřešíme ho přímo, nebo k jeho řešení použijeme stejný rekurzivní postup.

Realizace: nejčastěji rekurzivní funkcí

Podmínka rozumné (tj. efektivní) použitelnosti metody:
podproblémy vznikající rozkladem jsou na sobě nezávislé (nevyužívají stejné dílčí podúlohy a jejich řešení)

Protipříklad: Fibonacciho čísla rekurzivně
- řešení úlohy se sice skládalo z řešení podobných menších podúloh, ale ty nebyly na sobě nezávislé $\rightarrow$ opakované výpočty, velmi neefektivní řešení (exponenciální časová složitost)

Vyhodnocení aritmetického výrazu

Příklad: 5 * (3 + 4 * 6 – 8)

Postup řešení:

1. ve výrazu najdeme znaménko, které se vyhodnocuje až jako poslední
(je to znaménko zcela mimo závorky, z nich to s nejnižší prioritou,
z nich to nacházející se ve výrazu co nejvíce vpravo)

2. výraz rozdělíme na dva podvýrazy – vlevo a vpravo od nalezeného znaménka

3. oba tyto podvýrazy vyhodnotíme

4. s výsledky obou podvýrazů vykonáme poslední operaci určenou zvoleným znaménkem

Realizace:

1. lineární průchod výrazem zleva doprava
(pokud žádné znaménko mimo závorky neexistuje,
odstranit z výrazu vnější závorky a průchod zopakovat)
2. jednoduchá akce (konstantní časová složitosti)
3. buď je podvýraz triviální (pouze konstanta),
nebo se vyhodnotí ***rekurzivním voláním*** téhož algoritmu
4. jednoduché akce (konstantní časová složitosti)

Příklad:

5 * (3 + 4 * 6 – 8)		**95**
5	(3 + 4 * 6 – 8)	
	3 + 4 * 6 – 8	19
	3 + 4 * 6 8	27
	3 4 * 6	24
	4 6	

```python
def znamenko(s):
    plus, krat, zavorky = 0, 0, 0
    for i in range(len(s)):
        c = s[i]
        if c == '(': zavorky += 1
        if c == ')': zavorky -= 1
        if c == '+' or c == '-':
            if zavorky == 0: plus = i
        if c == '*' or c == '/':
            if zavorky == 0: krat = i
    if plus > 0: return s, plus
    if krat > 0: return s, krat
    if s[0] == '(':
        s = s[1:-1]
        s, z = znamenko(s)
        return s, z
    return s, None
```

```
def hodnota(s):
    s, z = znamenko(s)
    if z == None: return int(s)
    s1 = s[:z]
    s2 = s[z+1:]
    if s[z] == '+': return hodnota(s1) + hodnota(s2)
    if s[z] == '-': return hodnota(s1) - hodnota(s2)
    if s[z] == '*': return hodnota(s1) * hodnota(s2)
    if s[z] == '/': return hodnota(s1) // hodnota(s2)

print(hodnota(input()))
```

Časová složitost:
N – délka výrazu (počet čísel ve výrazu)

nejhorší případ
- vždy zvolíme první nebo poslední znaménko v aktuálním úseku
- postupně procházíme úseky délky N, N-1, N-2, …
  celkem tedy práce $N + (N-1) + (N-2) + \ldots + 1 = N.(N+1)/2$ **O(N^2)**

nejlepší případ
- vždy zvolíme prostřední znaménko ve výrazu
- procházíme 1 úsek délky N, 2 úseky délky $N/2$, 4 úseky délky $N/4$, …
  celkem hloubka rekurze log N
  na každé hladině rekurze se vykoná práce o celkovém rozsahu N
  (= součet délek K úseků, každý dlouhý N/K prvků) **O(*N*.log *N*)**

průměrný případ - lze ukázat, že časová složitost v průměrném případě je **O(*N*.log *N*)** jako v nejlepším případě

Hanojské věže

- 3 kolíky A, B, C
- na A je *N* disků různé velikosti, seřazené od největšího (dole) k nejmenšímu (nahoře)
- kolíky B a C jsou prázdné
- úkol: přenést všechny disky z A na B, mohou se odkládat na C
- podmínka: nikdy nesmí ležet větší disk na menším

Řešení:

PŘENESVĚŽ (N, A, B, C) = přenes *N* disků z A na B pomocí C

$\rightarrow$ provedeme jedině takto:

1. PŘENESVĚŽ (N-1, A, C, B) - rekurze
2. přenes disk z A na B - triviální akce
3. PŘENESVĚŽ (N-1, C, B, A) - rekurze

Časová složitost: O(2^N) – dána charakterem problému, k vyřešení úkolu je nutné provést tolik přesunů.

```python
def hanoj(n, a, b, c):
    """"
      řešení úlohy o Hanojských věžích:
      přenášíme "n" kotoučů
      z kolíku "a" na kolík "b";
      třetí kolík "c" je pomocný
    """

    if n > 0:
        hanoj(n-1, a, c, b)
        print(str(a) + " -> " + str(b))
        hanoj(n-1, c, b, a)


hanoj(10, 1, 2, 3)
```

MergeSort (třídění sléváním)

- princip již známe z iterativní implementace,
  nyní algoritmus implementujeme rekurzívně

- rozdělíme seznam čísel na dvě stejně velké části (plus/minus 1)
- každou z nich setřídíme
  (buď je triviální, nebo rekurzivním voláním téhož algoritmu)
- obě setříděné části pole slijeme dohromady = *merge*
  (lineární časová složitost vzhledem k délce slévaných úseků)

```python
def merge(x, y):
    """slévání dvou setříděných posloupností"""
    i = j = 0
    out = []

    while i < len(x) and j < len(y):
        if x[i] < y[j]:
            out.append(x[i])
            i += 1
        else:
            out.append(y[j])
            j += 1

    if i < len(x):
        out.extend(x[i:])
    if j < len(y):
        out.extend(y[j:])
    return out
```

```python
def mergesort(s):
    if len(s) <= 1: return s
    mid = len(s) // 2
    return merge(mergesort(s[:mid]),
                 mergesort(s[mid:]))
```

Časová složitost:

- sléváme postupně (odzadu)

  2 úseky délky *N*/2,

  4 úseky délky *N*/4,

  8 úseků délky *N*/8,

  …,

- celkem hloubka rekurze log *N*,
  na každé hladině rekurze se vykoná práce *N*
  (= součet délek *K* úseků, každý dlouhý *N/K* prvků) → **O(*N*.log *N*)**

Paměťová složitost:

- algoritmus potřebuje pomocnou paměť pro slévání
  velikosti *N* na každé hladině rekurze, takže celkem **O(*N*.log *N*)**
- navíc je třeba paměť na realizaci rekurzivních volání
  (zásobník – systémový nebo příp. vlastní)

Jiná implementace – řadí se data v původním seznamu:

```
def mergesort(s, zac, kon, kopie):
    """seřadí prvky v seznamu s v úseku zac – kon
       pomocný seznam kopie se používá pro slévání
    """

    stred = (zac + kon)//2
    if zac < stred:
        mergesort(s, zac, stred, kopie)
    if stred+1 < kon:
        mergesort(s, stred+1, kon, kopie)

    for _ in range(zac, kon+1):
        kopie[_] = s[_]  # kopie tříděného úseku
    i = zac              # začátek prvního úseku
    j = stred+1          # začátek druhého úseku
    k = zac              # začátek výsledného seznamu
```

```
while i <= stred and j <= kon:
    if kopie[i] <= kopie[j]:
        s[k] = kopie[i]
        i += 1
    else:
        s[k] = kopie[j]
        j += 1
    k += 1

while i <= stred:
    s[k] = kopie[i]
    i += 1
    k += 1

while j <= kon:
    s[k] = kopie[j]
    j += 1
    k += 1
```

Volání funkce (řadíme čísla v seznamu *p*):

```
mergesort(p, 0, len(p)-1, [0]*len(p))
```

Druhá implementace algoritmu má sice delší kód, celý výpočet se ale provádí jen se dvěma seznamy délky *N*, neprovádí se ani žádná průběžná alokace paměti (žádné realokace při prodlužování seznamů).

Paměťovou složitost jsme tím snížili na **O(*N*)**, algoritmus ovšem nadále potřebuje druhý pomocný seznam délky *N* pro slévání a paměť na realizaci rekurzivních volání.

QuickSort (třídění rozdělováním)

- v průměru nejrychlejší známý třídicí algoritmus

Základní idea:

- v seznamu zvolíme jeden prvek (my třeba ten, co leží uprostřed)
  → tzv. *pivot*
- prvky seznamu rozdělíme na menší, rovné, větší než pivot
- zvlášť ty menší a zvlášť ty větší setřídíme rekurzivním voláním
  téhož algoritmu, setříděné úseky spojíme za sebe

```
def quicksort(s):
    if len(s) <= 1: return s
    x = s[len(s) // 2]                      # pivot
    vlevo = [ a for a in s if a < x ]
    stred = [ a for a in s if a == x ]
    vpravo = [ a for a in s if a > x ]
    return quicksort(vlevo) + stred + quicksort(vpravo)
```

Jiná implementace – řadí se data v původním seznamu:

- inicializace: tříděným úsekem je celý seznam čísel délky *N*
- v tříděném úseku zvolíme jeden prvek (*pivot*, označme *x*)
- prvky přerovnat tak, aby vlevo byly prvky ≤ *x* a vpravo prvky ≥ *x* (lineární časová složitost vzhledem k délce tříděného úseku)
- tím se původní seznam rozdělí na dvě části
- oba vzniklé úseky setřídíme rekurzivním voláním téhož algoritmu (pokud nejsou triviální, tzn. délky 1, příp. 2)
- po skončení všech rekurzivních volání je celý seznam seřazen

Programová realizace:
- rekurzivní funkce, poprvé bude zavolána s parametry 0, *N*-1
- parametry = indexy v seznamu vymezující aktuální tříděný úsek

```
def quicksort(s, zac, kon):
    """seřadí prvky v seznamu s v úseku zac - kon"""
    i = zac
    j = kon
    x = s[(i + j)//2]
    while i <= j:
        while s[i] < x:
            i += 1
        while s[j] > x:
            j -= 1
        if i < j:
            s[i], s[j] = s[j], s[i]
            i += 1
            j -= 1
        elif i == j:
            i += 1
            j -= 1
```

```
    if zac < j:
        quicksort(s, zac, j)
    if i < kon:
        quicksort(s, i, kon)
```

```
p = [5, 22, -4, 0, 5, 77, -14, 0, 0, 1, -80]
quicksort(p, 0, len(p)-1)
print(p)
```

Paměťová složitost:

- třídění dat probíhá na místě v seznamu

- navíc je ale třeba paměť na realizaci rekurzivních volání
  (zásobník – systémový nebo vlastní)
  * v nejhorším případě O(N)
  * při dobré implementaci s vlastním zásobníkem
    lze snížit na O(log N)

Časová složitost:
stejná jako u vyhodnocení aritmetického výrazu

nejhorší případ
- za pivota vybereme vždy nejmenší nebo největší prvek v úseku
- postupně procházíme úseky délky N, N-1, N-2, …, **O(N^2)**

nejlepší případ
- za pivota vybereme vždy prostřední hodnotu (medián) v úseku
- procházíme 1 úsek délky N, 2 úseky délky N/2, 4 úseky délky N/4, …,
  celkem hloubka rekurze log N,
  na každé hladině rekurze se vykoná práce o celkovém rozsahu N
  O(N.log N)

průměrný případ
- lze dokázat, že časová složitost v průměrném případě je **O(N.log N)**
  jako v nejlepším případě

Jak snížit pravděpodobnost nepříznivého nejhoršího případu s kvadratickou časovou složitostí?

Volba pivota:

- jeden náhodně vybraný prvek z tříděného úseku

- vzorkování – medián ze tří náhodně zvolených prvků

- náhodný výběr
  + ověření, zda je alespoň ¼ prvků menších a ¼ prvků větších než on, pokud ne, opakovaný výběr (třeba i vícekrát)

- nalezení mediánu tříděného úseku
  umíme provést v lineárním čase, zajistí to časovou složitost quicksortu O(N.log N) i v nejhorším případě, ale v průměru bude pomalejší (vzroste multiplikativní konstanta)

QuickSort – implementace bez rekurzivní procedury

- použití rekurzivní funkce lze nahradit vlastním zásobníkem
  a cyklem (podobně jako u prohledávání do hloubky)
- zásobník „dluhů“ = seznam úseků, které je ještě třeba dotřídit
- místo rekurzivního volání → vložení úseku do zásobníku
- v cyklu se postupně vybírají ze zásobníku jednotlivé dluhy
  a řeší se (čímž zase vznikají nové, menší dluhy)

- pro programátora více práce při psaní programu
- výsledný kód ovšem může být úspornější
    * na čas (úspora za režii rekurzivních volání)
    * na paměť (při dobré organizaci práce stačí zásobník
      logaritmické výšky)