Prohledávání stavového prostoru do šířky

= breadth-first search (BFS)
algoritmus „vlny“

- budeme ***souběžně*** zkoušet všechny možné varianty pokračování výpočtu, dokud nenajdeme řešení úlohy
  → průchod stromem všech možných cest výpočtu do šířky, po vrstvách
  (v každé vrstvě zkoušíme všechny možnosti zleva doprava)

- je nutné pamatovat si všechny stavy v jedné vrstvě stromu
  → při větší hloubce to může být paměťově velmi náročné (exponenciálně), až nepoužitelné

- použijeme, máme-li nalézt jedno (nejkratší) řešení
  → vždy najde nejkratší možné řešení co do počtu kroků

- výhodné, existují-li v rozsáhlém prohledávacím stromě nějaká řešení vzdálená od startu jen na málo kroků – prohledá se jen několik horních vrstev stromu (významná časová úspora oproti prohledávání do hloubky)

- nevhodné při úloze nalézt všechna řešení – musí se projít celý strom, a to je při stejné časové složitosti výhodnější provádět do hloubky (paměťově mnohem méně náročné)

- v některých úlohách je nutné prohledávat do hloubky (prohledávání do šířky by se paměťově nezvládlo), v některých je nutné prohledávat do šířky (prohledávání do hloubky by se časově nezvládlo), v některých je to jedno (obě varianty prohledávání jsou paměťově i časově stejně náročné)

Implementace prohledávání stavového prostoru do hloubky a do šířky

- již známe z prohledávání binárního stromu,
  zde použijeme v principu stejný postup

- prohledávání *do hloubky* se zpravidla realizuje rekurzivní funkcí,
  totéž lze naprogramovat pomocí **zásobníku** a cyklu

vlož do zásobníku výchozí stav
dokud (není zásobník prázdný) a (není nalezeno řešení) opakuj
    vyjmi ze zásobníku vrchní stav a zpracuj ho
    vlož do zásobníku všechna jeho možná pokračování
        (pouze dosud nenavštívené stavy!)

- prohledávání *do šířky* se realizuje podobně pomocí **fronty** a cyklu

- do fronty ukládáme navštívené stavy,
  které je třeba ještě zpracovat

- musíme evidovat všechny již navštívené stavy
  a nezařazovat je do fronty opakovaně

vlož do fronty výchozí stav
dokud (není fronta prázdná) a (není nalezeno řešení) opakuj
    vyjmi z fronty první stav a zpracuj ho
    vlož do fronty všechna jeho možná pokračování
        (pouze dosud nenavštívené stavy!)

Příklady:

Proskákání šachovnice koněm *(bylo minule)*
- každé možné řešení úlohy má délku 63 kroků,
  prohledávání do šířky je paměťově nereálné

Nejkratší cesta koněm na šachovnici
- dána výchozí a cílová pozice
- úkol: dojít šachovým koněm co nejmenším počtem tahů
  z výchozí do cílové pozice
- prohledávání do hloubky je velmi nevhodné – backtracking
  do hloubky až 63
  (přitom vždy existuje cesta délky maximálně 6)

Postup řešení:

- prohledáváme do šířky jako při hledání nejkratší cesty
  v neohodnoceném grafu – *algoritmus „vlny“*

  * počáteční pozici označíme 0,
  * sousední políčka označíme 1, jejich sousedy označíme 2, atd.,
    dokud nedojdeme na cílové políčko
    (číslo = délka nejkratší cesty ze startu na toto políčko)

- k nalezení vlastní cesty je nutný ještě *„zpětný chod“*
  (cíl $\rightarrow$ sousední políčko s hodnotou o 1 menší $\rightarrow$ … $\rightarrow$ start)

Možnosti implementace (možná vylepšení):

1. Políčka šachovnice z předchozího kroku vlny (tzn. políčka s číslem o 1 menším) nevyhledávat vždy procházením celé šachovnice, ale ukládat si jejich souřadnice do fronty
→ je potřeba paměť navíc na uložení fronty, ale urychlí to výpočet ve fázi algoritmu „vlny“

2. U každého políčka na šachovnici si pamatovat nejen minimální počet kroků, které na něj vedou ze startu, ale i souřadnice předchozího políčka
→ snadné nalezení předchůdce při zpětném chodu (ale zase je to paměťově i časově náročnější při provádění „vlny“)

Obě uvedená vylepšení implementace známe již z algoritmu hledání nejkratší cesty v grafu.

```
n = 8                  # rozměr šachovnice
s = [[-1]*(n+1) for _ in range(n+1)
                       # vzdálenost pole od startu
pred = [[None]*(n+1) for _ in range(n+1)]
                       # předchůdce na nejkratší cestě
tah = ((1,2), (2,1), (2,-1), (1,-2),
       (-1,-2), (-2,-1), (-2,1), (-1,2))
                       # přírůstky souřadnic při tahu koněm

start1, start2 =
    [int(_) for _ in input('Startovní pole: ').split()]
cil1, cil2 =
    [int(_) for _ in input('Cílové pole: ').split()]
s[start1][start2] = 0
fronta = [(start1, start2)]
                       # fronta řídí průchod do šířky
```

```
while s[cil1][cil2] == -1: # dokud nejsme v cíli
    i1, i2 = fronta.pop(0) # vyzvedneme z fronty (i1,i2)
    krok = s[i1][i2] + 1
    for smer in range(len(tah)):   # zkusíme všemi směry
        j1 = i1 + tah[smer][0]
        j2 = i2 + tah[smer][1]
        if j1 >= 1 and j1 <= n and j2 >= 1 and j2 <= n:
            if s[j1][j2] == -1:    # pole nenavštíveno
                s[j1][j2] = krok   # půjdeme na (j1,j2)
                pred[j1][j2] = (i1, i2)
                fronta.append((j1, j2))
                                   # vložíme do fronty
```

```
print('Nejkratší cesta v pořadí od cíle ke startu:')
print((cil1, cil2), end=' ')
i1, i2 = cil1, cil2
while (i1, i2) != (start1, start2):
    # rekonstrukce cesty - zpětný chod
    j1, j2 = pred[i1][i2]
    i1, i2 = j1, j2
    print((i1, i2), end=' ')
print()
```

„Rozděl a panuj“ (divide et impera)

- 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	

```
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) + … + 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

→ 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.

```
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)

```
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
```

```
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*)**

Prostorová 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(a, zac, kon, temp):
    """
      třídění sléváním - rekurzivní verze
      třídí data v původním poli
      seřadí seznam a[zac..kon] pomocí temp[zac..kon]
    """

    stred = (zac + kon) // 2
    if zac < stred:
        mergesort(a, zac, stred, temp)
    if stred+1 < kon:
        mergesort(a, stred+1, kon, temp)

    i = zac                 # začátek prvního úseku
    j = stred+1             # začátek druhého úseku
    k = zac                 # začátek výsledného seznamu
```

```
# sleje a[zac..stred] s a[stred+1..kon]
# do temp[zac..kon]
while i <= stred and j <= kon:
    if a[i] <= a[j]:
        temp[k] = a[i]
        i += 1
    else:
        temp[k] = a[j]
        j += 1
    k += 1

if i <= stred:          # zbytek prvního úseku
    temp[k:kon+1] = a[i:stred+1]
else:                   # zbytek druhého úseku
    temp[k:kon+1] = a[j:kon+1]

# výsledek zkopíruje do seznamu a
a[zac:kon+1] = temp[zac:kon+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).

Prostorovou 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

- rekurze končí u úseků délky 1

```
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 $\leq x$ a vpravo prvky $\geq x$ (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)
```

Prostorová 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 algoritmu na 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 funkce

- 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)