C2142 Návrh algoritmů pro přírodovědce

Tomáš Raček

Financováno Evropskou unií
NextGenerationEU
NÁRODNÍ PLÁN OBNOVY
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

1. Od problému k algoritmu

Problém, algoritmus

Problém – nerozřešená, sporná otázka, kterou je třeba řešit.

Příklady problémů:

- Kolik je třetí odmocnina z 27?
- Bude zítra pršet?
- $\hat{H}\Psi = E\Psi$
- Co si vzít na sebe do divadla?

Algoritmus je funkce mezi vstupem a výstupem reprezentována konečným způsobem v nějakém formalismu.

Ne každý problém lze algoritmicky řešit.

Potřeba formalismu

Přirozený jazyk trpí často nejednoznačností – dvojsmysly.

- Návrh putuje k Ústavnímu soudu v době, kdy je neúplný.
- Máme jen velvyslance prezidenta.

Formální jazyk má přesně danou syntaxi (strukturu) a sémantiku (význam).

- matematický zápis (př. $\forall x \in A : x \leq 0$)
- značkovací jazyky (př. HTML, XML,...)
- programovací jazyky (př. C/++, Python, Java, C#,...)

```
def fact(n):
    return n * fact(n - 1) if n > 0 else 1
```

Hledání počátku replikace jednoduchých bakterií

Biologický problém. Nalezněte část genomu, kde začíná replikace DNA (angl. origin of replication, oriC).

Způsoby řešení

- biolog – laboratoř
- informatik – počítač

Vstup: Genom (řetězec znaků A, C, G, T)

Výstup: Pozice počátku replikace v genomu.

Je toto informaticky správně formulovaný problém?

Problémy specifikace

Vstup: Genom (řetězec znaků A, C, G, T)

Výstup: Pozice počátku replikace v genomu.

Počátek replikace (oriC) není užitečně definován → potřeba dalších biologických informací.

Další vlastnosti:

- většinou několik stovek nukleotidů dlouhá oblast genomu
- obsahuje netriviální množství tzv. DnaA box (= krátké oblasti, na které se naváže DnaA protein a spustí tak replikaci)
  - typicky např. 9 nukleotidů
  - liší se pro každý organismus

Transformace problému

Problém nalezení oriC převedeme na hledání DnaA boxes. Oblast s jejich největším výskytem bude pak ukazovat na oriC.

(1) Zjednodušený problém. Nalezněte v textu řetězce délky k s nejvyšším počtem výskytů (DnaA box).

(2) Upravený problém. Nalezněte v textu řetězce délky k (DnaA box) v oblasti délky n ($\approx$ oriC) s minimálním počtem výskytů t.

Je náš zvolený přístup perspektivní? Pravděpodobnost, že existuje řetězec délky 9, který se vyskytuje v oblasti dlouhé 500 nukleotidů alespoň 3krát je asi 1/1300.

(1) Hledání nejčastějších slov v textu – řešení

Užitečným nástrojem pro řešení problému je dekompozice – rozložení problému na menší, lépe zvládnutelné jednotky.

pattern_count(text, pattern)

- pomocná funkce
- vrací počet výskytů řetězce pattern v řetězci text.

frequent_words(text, k)

1. Pro každý podřetězec délky k řetězce text spočítej jeho výskyt pomocí funkce pattern_count
2. Urči nejvyšší nalezenou četnost
3. Vrať řetězce s touto nejvyšší četností

(1) Hledání nejčastějších slov v textu

```
def pattern_count(text, pattern):
    count = 0
    for i in range(0, 1  + len(text) - len(pattern)):
        if text[i : i + len(pattern)] == pattern:
            count += 1

    return count

def frequent_words(text, k):
    counts = dict()
    frequent_patterns = set()

    for i in range(0, len(text) - k + 1):
        pattern = text[i : i + k]
        counts[pattern] = pattern_count(text, pattern)

    max_count = max(counts.values())
    for (pattern, count) in counts.items():
        if count == max_count:
            frequent_patterns.add(pattern)

    return frequent_patterns
```

> ”Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”
>
> John Woods

Pro naše účely je forma uvedených algoritmů dostačující, v praxi je vhodné doplnit zejména:

- komentáře objasňující zvolený postup, případně myšlenkově obtížnější části algoritmu
- kontroly, zda jsou vstupní data v pořádku

(1) Upravená verze hledání nejčastějších slov

```
def frequent_words(text, k):
    """Find the most frequent words of the specified length"""
    counts = dict()
    frequent_patterns = set()

    if k < 1:
        raise Exception('The length of the pattern has to be at least 1')

    # Find the frequency for each pattern in the text
    for i in range(0, len(text) - k + 1):
        pattern = text[i : i + k]
        counts[pattern] = pattern_count(text, pattern)

    # Select the patterns with the highest occurrence
    max_count = max(counts.values())
    for (pattern, count) in counts.items():
        if count == max_count:
            frequent_patterns.add(pattern)

    return frequent_patterns
```

(2) Hledání nejčastějších slov v podřetězci

(2) Upravený problém. Nalezněte v textu řetězce délky k (DnaA box) v oblasti délky n (≈ oriC) s minimálním počtem výskytů t.

```
def frequent_words_within_region(text, k, n, t):
    frequent_patterns = set()

    for i in range(0, len(text) - n + 1):
        text_region = text[i: i + n]
        counts = dict()

        for j in range(0, len(text_region) - k + 1):
            pattern = text_region[j : j + k]
            counts[pattern] = pattern_count(text_region, pattern)

        for (pattern, count) in counts.items():
            if count >= t:
                frequent_patterns.add(pattern)

    return frequent_patterns
```

Korektnost

Korektnost algoritmu. Návrh/implementace odpovídá specifikaci (zadání).

Metody:

- testování *
- matematický důkaz
  - ověření, že korektní vstupní data budou algoritmem transformována na správná výstupní
- formální verifikace
  - ověření, zda model systému splňuje zadanou vlastnost
  - stejná míra jistoty jako u matematického důkazu
  - lze algoritmizovat
  - použitelná pouze pro velmi malé systémy

Testování

Testování. Levný a rychlý nástroj pro ověření základní funkcionality.

Co testovat?

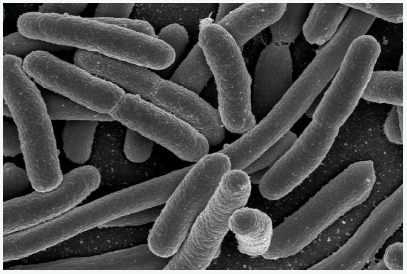
- obvyklý běh
- krajní hodnoty
- zakázané hodnoty

Motto:

> ”Testing shows the presence, not the absence of bugs.”
>
> Edsger W. Dijkstra

Proč tomu tak je?

Escherichia coli



Genom E. coli – asi 4,6 milionu nukleotidů (níže prvních 0,00313 %):

AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAA
AAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGT
AAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATA

Praktický test

(1) Zahřívací problém. Nalezněte nejčastější řetězce délky 9 v rámci prvních 5000 nukleotidů genomu E. coli.

- použijeme funkci frequent_words
- čas výpočtu asi 6 s

ACCATTACC TGGCGATGA CACCATTAC AACTGAAAG TGATGAAGA

(2) Kompletní genom E. coli Zopakujte (1) pro celý genom E. coli.

- téměř 1000krát větší problém než (1)
- odhad času výpočtu $1000 \cdot 6\ s = 6000\ s < 7200\ s = 2\ h$
- realita?

Vylepšení o další biologické poznatky 1

Náš vstupní řetězec představuje pouze jedno ze dvou navzájem reverzně komplementárních vláken genomu.

Komplementární báze:

- A $\leftrightarrow$ T
- C $\leftrightarrow$ G

Např. reverse_complement(ACCCTG) = CAGGGT.

Závěr. Při hledání nejčastějších podřetězců je potřeba vzít do úvahy i druhý řetězec.

Vylepšení o další biologické poznatky 2

V řetězci DNA může dojít k mutacím (např. záměně jednoho nukleotidu za jiný).

Hammingova vzdálenost. Počet pozic, na kterých se dva řetězce stejné délky liší.

Př. Seznam všech řetězců (ze znaků A, C, G, T), které mají Hammingovu vzdálenost od řetězce AAA rovnu nejvýše 1:

GAA ACA CAA AAA AAC ATA AGA TAA AAG AAT

Závěr. Při hledání nejčastějších podřetězců zohledníme i jejich blízké (co do Hammingovy vzdálenosti) sousedy.

2. Úvod do složitosti

Hledání nejčastějších slov – frequent_words

Zadání. Nalezněte v textu řetězce délky k s nejvyšším počtem výskytů.

frequent_words(text, k)

1. Pro každý podřetězec délky k řetězce text spočítej jeho výskyt pomocí funkce pattern_count(text, pattern)
2. Urči nejvyšší nalezenou četnost
3. Vrať řetězce s touto nejvyšší četností

Praktický test.

- krátké řetězce – frequent_words uspokojivě funguje
- dlouhé řetězce – nepoužitelné, čas výpočtu neodpovídá odhadu

frequent_words – doba výpočtu

Pozorování

- doba výpočtu je úměrná velikosti vstupních dat
- závislost není nutně lineární – výpočet na 1000krát větší úloze nemusí trvat 1000krát déle
- na různých strojích/architekturách různé časy výpočtu
  - Thinkpad T480s: 6 s
  - Thinkpad T430s: 7 s
  - Thinkpad X200s: 14 s
  - Desktop (Ryzen 3600): 3 s

Důsledek (1). Nutná hlubší analýza frequent_words.

Důsledek (2). Porovnání náročnosti algoritmů podle času výpočtu není vhodné, potřebujeme aparát nezávislý na konkrétním stroji/architektuře/implementaci.

Složitost

Složitost algoritmu. Zaveďme složitost algoritmu jako funkci $f(n)$, kde n je velikost vstupu.

Návrh. $f(n)$ určuje počet jednoduchých operací daného algoritmu pro vyřešení problému o velikosti n.

- jednoduché operace $\approx$ instrukce CPU (např. sečtení nebo porovnání dvou čísel, AND/OR,...)
- řešení nezávislé na architektuře

Důsledek. Porovnání efektivity algoritmů lze zjednodušeně převést na porovnání jejich složitostí.

Asymptotická složitost – definice

Formální definice

$$\mathcal{O}(g) = \{f \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

$f \in \mathcal{O}(g)$ čteme „f roste asymptoticky nejvýše tak rychle jako g“.

Význam konstant

- c rozdíl pouze v multiplikativní konstantě nepovažujeme za významný, tj. ztotožňujeme např. n^2 a $4n^2$
- n_0 vztah nemusí platit pro prvních n_0 čísel

Poznámka. Analogicky lze definovat další množiny:

- $f \in \Omega(g)$ – f roste asymptoticky alespoň tak rychle jako g
- $f \in \Theta(g)$ – f roste asymptoticky právě tak rychle jako g

Asymptotická složitost – příklady

Rychlost růstu funkcí

$$\log n \ll n \ll n \log n \ll n^2 \ll 2^n \ll n! \qquad \text{pro } n \to \infty$$

Příklady

Funkce	Složitostní třída	Pojmenování
2142	$\mathcal{O}(1)$	konstantní
$2 \log n + 4$	$\mathcal{O}(\log n)$	logaritmická
$0.5n + \log n$	$\mathcal{O}(n)$	lineární
$n^2 - 10n$	$\mathcal{O}(n^2)$	kvadratická
$6n^3$	$\mathcal{O}(n^3)$	kubická
$2^n - 1$	$\mathcal{O}(2^n)$	exponenciální

Poznámka. Ověření, zdali $f \in \mathcal{O}(g)$, lze provést výpočtem limity $\lim_{n\to\infty} f(n)/g(n)$.

Složitost problému

Cíl. Snaha o nalezení efektivních algoritmů pro daný problém.

Otázka. Lze zrychlovat pořád, nebo existuje nějaký dolní limit?

Složitost problému

- minimální počet operací potřebný pro vyřešení libovolné instance problému
- nutno odvodit teoreticky → mnohdy netriviální
- odpovídá složitosti optimálního algoritmu pro daný problém

Jak ale poznám optimální algoritmus? Srovnejme odhady složitosti problému $\mathcal{P}_i$ a složitosti algoritmů $\mathcal{A}_j$ řešící tento problém:

$$\mathcal{P}_1(n) < \ldots < \mathcal{P}_k(n) \leq \mathcal{A}_1(n) < \ldots < \mathcal{A}_m(n)$$

$\mathcal{A}$ je optimální algoritmus, pokud $\mathcal{A}(n) = \mathcal{P}_k(n)$.

Složitost problému – příklady

Nalezení nejmenšího prvku pole

- je nutné projít všechny prvky pole – $\Omega(n)$ operací
- algoritmus se složitostí $\mathcal{O}(n)$ jistě existuje → složitost problému (= složitost optimálního algoritmu) je lineární

Násobení matic

- potřeba $\Omega(n^2)$ operací
- naivní algoritmus – $\mathcal{O}(n^3)$
- Strassenův algoritmus – $\mathcal{O}(n^{\log_2 7}) \doteq \mathcal{O}(n^{2,81})$
- aktuálně nejlepší algoritmus (2014) – $\mathcal{O}(n^{2,372\ldots})$
- nalezení optimálního algoritmu je otevřený problém

Prostorová složitost

Prostorová složitost. Vedle časové náročnosti algoritmů lze určit i množství paměti, které algoritmus potřebuje pro svůj výpočet.

- velikost vstupních (a výstupních) dat neuvažujeme
- vyjadřujeme také $\mathcal{O}$-notací

In situ algoritmus vyžaduje navíc pouze $\mathcal{O}(1)$ paměti.

- výpočet průměrné hodnoty prvků v poli
- naivní násobení matic
- ...

Otázka. Je lepší in situ algoritmus s časovou složitostí $\mathcal{O}(n^2)$ než algoritmus s časovou složitostí $\mathcal{O}(n \log n)$ a prostorovou složitostí $\mathcal{O}(n)$?

Vztah mezi časem a prostorem

Teze. Někdy lze snížit časovou složitost algoritmu zvýšením jeho prostorové složitosti (a naopak).

↑ prostor ↓ čas

- softwarová cache
- předpočítání (mezi)výsledků

↑ čas ↓ prostor

- komprese
- zvýšení abstrakce

Složitost v praxi

Tabulka časů výpočtu algoritmů o složitostech $\log n$, n, n^2, 2^n a pro vstup velikosti 10, 20, 50 a 1000. Předpokládejme, že jedna iterace algoritmu trvá $1\mu s$.

	10	20	50	1000
$\log n$	0,000001 s	0,000001 s	0,000002 s	0,000003 s
n	0,00001 s	0,00002 s	0,00005 s	0,001 s
n^2	0,0001 s	0,0004 s	0,0025 s	1 s
2^n	0,001024 s	1,048576 s	35,7 let	$3,4 \cdot 10^{287}$ let

Poznámka. Stáří vesmíru je odhadováno na $13,8 \cdot 10^9$ let.

pattern_count – analýza

```
def pattern_count(text, pattern):
    count = 0
    for i in range(0, 1  + len(text) - len(pattern)):
        if text[i : i + len(pattern)] == pattern:
            count += 1

    return count
```

Pozorování

- procházíme celkem $|text| - |pattern| + 1$ možných umístění
- každé porovnání dvou řetězců obnáší nejvýše $|pattern|$ porovnání jednotlivých znaků

Závěr. Počet kroků, které vykoná funkce pattern_count, lze vyjádřit jako $O(|pattern| \cdot (|text| - |pattern| + 1))$.

frequent_words – analýza I

```
def frequent_words(text, k):
    counts = dict()
    frequent_patterns = set()

    for i in range(0, len(text) - k + 1):
        pattern = text[i : i + k]
        counts[pattern] = pattern_count(text, pattern)

    max_count = max(counts.values())
    for (pattern, count) in counts.items():
        if count == max_count:
            frequent_patterns.add(pattern)

    return frequent_patterns
```

Pozorování

- počet volání pattern_count je $|text| - k + 1$
- další příkazy nejsou určující pro dobu běhu

frequent_words – analýza II

Složením předchozích informací dostáváme:

frequent_words(text, k)

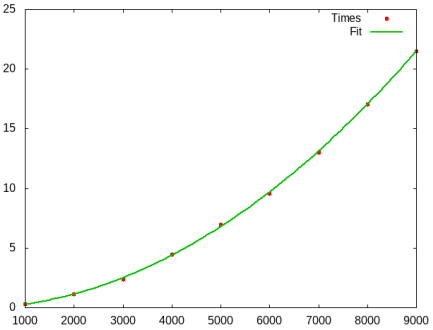
- složitost funkce pattern_count je $O(|pattern| \cdot (|text| - |pattern| + 1))$
- počet volání pattern_count je $|text| - k + 1$
- platí $k = |pattern|$
- počet kroků celkem:

$$k \cdot (|text| - k + 1) \cdot (|text| - k + 1) = k \cdot (|text| - k + 1)^2$$

V praxi platí $k \ll |text|$, asymptotická složitost funkce frequent_words je tedy $O(k \cdot |text|^2)$.

frequent_words – praxe

Měření. Doba výpočtu funkce frequent_words(text, k) pro $k = 9$ a $|text| = \{1000, \ldots, 9000\}$.



Pozorování. Naměřená data lze úspěšně proložit parabolou, což odpovídá odhadnuté složitosti $O(k \cdot |text|^2)$.

Hledání nejčastějších slov v textu

Dosavadní řešení. Jsme schopni navrhnout a implementovat algoritmus se složitostí $O(k \cdot |text|^2)$.

Zásadní otázka. Jde to i lépe?

Alternativní návrh. Počítání četností podřetězců při průchodu textem

1. Procházej vstupní text postupně po podřetězcích délky k
   1.1 Pokud se konkrétní podřetězec vyskytl poprvé, nastav jeho četnost na 1, jinak ji zvyš o 1
2. Urči nejvyšší nalezenou četnost
3. Vrať řetězce s touto nejvyšší četností

faster_frequent_words

Jak ukládat pro každý řetězec jeho četnost?

- počet různých řetězců délky k z písmen A, C, G, T je 4^k
- každému tomuto řetězci lze přiřadit číslo od 0 do $4^k - 1$

Příklad pro $k = 3$:

$$\text{AAA} \to 0, \text{AAC} \to 1, \text{AAG} \to 2, \ldots, \text{TTT} \to 63$$

Příklad převodu řetězce na číslo. A $\to 0$, C $\to 1$, G $\to 2$, T $\to 3$.

$$\begin{aligned} \text{ACCTG} &\to A \cdot 4^4 + C \cdot 4^3 + C \cdot 4^2 + T \cdot 4^1 + G \cdot 4^0 \\ \text{ACCTG} &\to 0 + 64 + 16 + 12 + 2 \\ \text{ACCTG} &\to 94 \end{aligned}$$

Implementace. Vytvořím pole o velikosti 4^k, kde budu ukládat četnosti jednotlivých řetězců.

Převod řetězce na číslo – implementace

```
def pattern2number(pattern):
    characters = "ACGT"

    if pattern == "":
        return 0
    else:
        return 4 * pattern2number(pattern[:-1]) \
                + characters.index(pattern[-1:])


def number2pattern(number, k):
    characters = "ACGT"

    if k == 0:
        return ""
    else:
        divisor = 4 ** (k - 1)
        return characters[number // divisor] \
                + number2pattern(number % divisor, k - 1)
```

faster_frequent_words – implementace

```python
def computing_frequencies(text, k):
    frequency_array = [0] * (4 ** k)

    for i in range(len(text) - k + 1):
        pattern = text[i: i + k]
        frequency_array[pattern2number(pattern)] += 1

    return frequency_array


def faster_frequent_words(text, k):
    frequent_patterns = set()
    frequency_array = computing_frequencies(text, k)
    max_count = max(frequency_array)

    for i in range(0, 4 ** k):
        if frequency_array[i] == max_count:
            frequent_patterns.add(number2pattern(i, k))

    return frequent_patterns
```

Složitost faster_frequent_words

Složitost jednotlivých fází algoritmu

- inicializace pole četností $\mathcal{O}(4^k)$
- převod řetězce na číslo (a naopak) $\mathcal{O}(k)$
- průchod vstupním textem, počítání četností $\mathcal{O}(k \cdot (|text| - k + 1))$
- nalezení nejvyšší četnosti $\mathcal{O}(4^k)$
- výběr řetězců s nejvyšší četností $\mathcal{O}(k \cdot 4^k)$

Celková složitost faster_frequent_words. Po úpravě dostáváme složitost $\mathcal{O}(k \cdot |text| + k \cdot 4^k)$, přičemž paměťová složitost je $\mathcal{O}(4^k)$.

Závěr. Pro $k \ll |text|$ je faster_frequent_words výrazně rychlejší než frequent_words.

A jde to ještě lépe? ;-)

3. Základní datové struktury

C2142 Návrh algoritmů pro přírodovědce

Tomáš Raček

Datové struktury

Datová struktura popisuje uložení dat v paměti počítače.

Výběr datové struktury ovlivňuje:

- množství použité paměti (režie konkrétní struktury)
- složitost operací nad touto strukturou, např.:
  - přidání/odebrání prvku
  - zpřístupnění prvku
  - sekvenční průchod přes všechny prvky
  - nalezení minimálního/maximálního prvku
  - ...

Aktuálně známe:

- jednoduché proměnné (celá čísla, desetinná čísla, znaky,...)
- pole

Pole I

Pole je soubor prvků stejného typu uložených v paměti za sebou.

Vlastnosti:

- zabírá souvislou oblast v paměti
- zpřístupnění prvku přes jméno pole a index (př. A[i + 1])

Indexace v poli

- výpočet adresy konkrétního prvku: adresa prvního prvku (A) + velikost typu (D) * počet předchozích prvků
- 1D: adresa A[i] = $A + D \cdot i$
- 2D: adresa A[i][j] = $A + D \cdot (i \cdot n + j)$
- 3D: adresa A[i][j][k] = $A + D \cdot (i \cdot n^2 + j \cdot n + k)$

Pole II

Výhody pole:

- jednoduchá implementace
- přímočaré použití
- zpřístupnění prvku v konstantním čase
- sekvenční přístup vede často v praxi k vysokému výkonu (díky využití vyrovnávací paměti)

Nevýhody pole:

- problematická změna velikosti – Jak přidám/odeberu prvek?

Statické vs. dynamické datové struktury

Statické datové struktury (např. pole) mají pevně danou velikost.

- neflexibilní přístup
- lze nadhodnotit velikost → plýtvání paměti ve většině případů

Dynamické datové struktury nabízí implicitní způsob změny velikosti.

- složitější na implementaci
- změna velikosti přináší jistou režii

Dynamické pole I

Nedostatky pole. Pole neumožňuje libovolně přidávat/odebírat prvky, změna jeho velikosti je složitá → je potřeba znovu alokovat souvislý kus paměti a kopírovat prvky.

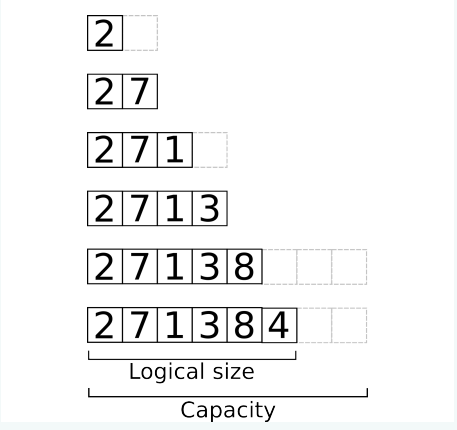
Pozorování

- přidávat prvky na začátek/doprostřed pole je obtížné
- přidání nebo odebrání prvku na konci pole je snazší → můžeme se vyhnout kopírování prvků
- pro přidání prvku na konec pole musí být v paměti místo

Dynamické pole. Rozšíření pole o možnost přidávat a odebírat prvky. Implementací jde o statické pole, které je logicky rozděleno na aktuálně zabranou a volnou oblast.

Dynamické pole II

- prvky lze na konec přidávat až do vyčerpání kapacity
- poté je potřeba realokace celého pole, typicky např. na dvojnásobnou kapacitu
- zjevně má přidání prvku na začátek/doprostřed/na konec pole lineární složitost



Jak je tomu ale v praxi?

Amortizovaná složitost

Pozorování. Při přidávání prvku na konec dynamického pole je většina těchto operací rychlá – mají konstantní složitost. Pokud je ale nutné pole realokovat, složitost je lineární.

- lineární složitost nastává ale velmi zřídka
- někdy může být užitečnější jiný aparát pro popis složitosti

Amortizovaná složitost neuvažuje operace izolovaně, ale v rámci větší skupiny.

- poskytuje reálnější odhad pro dlouhodobé chování datové struktury
- nedává horní odhad → některé operace mohou trvat „překvapivě“ dlouho

Dynamické pole – amortizovaná složitost

Příklad. Uvažme dynamické pole o n prvcích, kapacitě $2n$ a operaci přidání prvku na konec pole.

- prvních n operací přidání prvku na konec má složitost $O(1)$
- následující operace je ale v $O(n) \rightarrow$ způsobí zvětšení pole na dvojnásobek
- dalších $2n$ těchto operací je opět rychlých
- ...

Pozorování. Posloupnost n operací přidání prvku na konec pole má celkem složitost $O(n)$.

Závěr. Přidání prvku na konec dynamického pole má konstantní amortizovanou složitost. (Analogicky pro odebrání posledního prvku.)

Dynamické pole IV – praxe

Praxe. Dynamické pole je implementováno například jako list v Pythonu nebo std::vector v C++.

Měření. Změřte dobu běhu programu, který postupně přidává celkem 100 000 prvků do pole (a) na začátek, (b) doprostřed, (c) na konec.

(a) 8,7 s

(b) 4,5 s

(c) 0,1 s

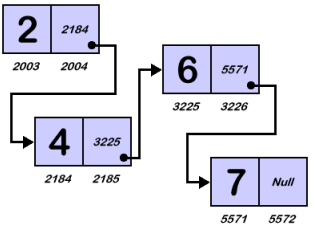
Závěr. Měření potvrzuje amortizovaný odhad složitosti. Dynamické pole je efektivní při přidávání/odebírání posledního prvku.

Spojový seznam I

Nedostatkem dynamického pole je přidávání/odebírání prvku mimo jeho konec.

Spojový seznam představuje datovou strukturu s konstantní složitostí přidání/odebrání prvku.

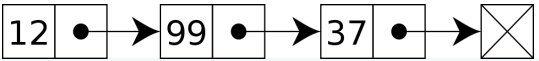
- každý prvek obsahuje ukazatel na prvek následující – next
- na rozdíl od pole nevyžaduje souvislou oblast paměti



Spojový seznam II

Přístup ke spojovému seznamu je realizován přes ukazatel na jeho začátek – HEAD.

Pro zvýšení efektivity přidávání prvků ($\rightarrow O(1)$) na konec seznamu lze využít ukazatel na konec seznamu – TAIL.



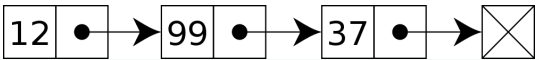
Pozorování. Indexace (= zpřístupnění prvku) je v $O(n)$.

Průchod seznamem:

- sekvenční průchod v $O(n)$ – stejně jako u pole
- v opačném pořadí ovšem $O(n^2)$!

Spojový seznam III – příklad

Příklad operace nad spojovým seznamem



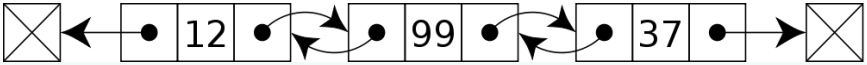
Přidání prvku na začátek seznamu:

1. alokace paměti pro nový prvek N
2. kopie vlastních dat do N
3. nastavení N.next na aktuální HEAD
4. nastavení ukazatele HEAD na nový prvek

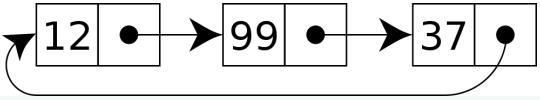
Další typy spojových seznamů

Oboustranně zřetězený spojový seznam. Je rozšířením spojového seznamu o ukazatel na předchozí prvek – prev.

- průchod seznamem v opačném pořadí v $O(n)$
- zvyšuje paměťovou náročnost



Cyklický spojový seznam propojuje poslední prvek s prvním.



Srovnání datových struktur

Složitosti operací nad základními datovými strukturami:

Operace	Pole	Dynamické pole	Spojový seznam
Indexace	$O(1)$	$O(1)$	$O(n)$
Vyhledávání	$O(n)$	$O(n)$	$O(n)$
Přidání prvku	–	$O(n)$ / $O(1)^*$	$O(1)$
Odebrání prvku	–	$O(n)$ / $O(1)^*$	$O(1)$
Paměťová režie	–	$O(n)$	$O(n)$

* amortizovaná složitost pro poslední prvek

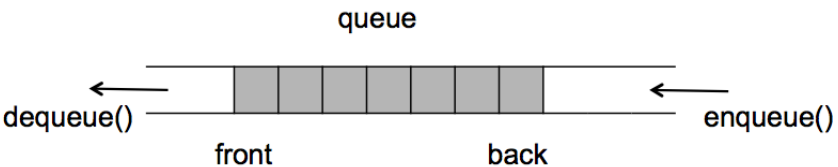
Závěr. Univerzálně nejlepší datová struktura neexistuje.

Fronta I

Fronta je datová struktura typu FIFO (First In First Out). Implementuje tzv. FCFS (First Come First Served) přístup.

Operace:

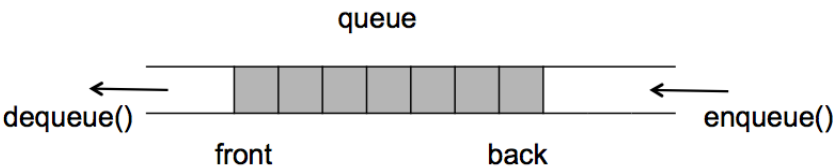
- enqueue – přidá prvek na konec fronty
- dequeue – odebere prvek ze začátku fronty
- obě lze implementovat v $\mathcal{O}(1)$



Fronta II

Implementace fronty

- (dynamické) pole
- spojový seznam

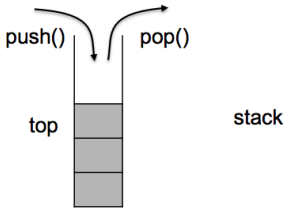


Prioritní fronta

- každému prvku je přiřazeno číslo – priorita
- priorita určuje, kam bude prvek do fronty zařazen
- několik možných implementací

Zásobník

Zásobník je datová struktura typu LIFO (Last In First Out). Všechny ostatní vlastnosti jsou shodné s frontou.



Operace:

- push – přidá prvek na vrchol zásobníku
- pop – odebere prvek z vrcholu zásobníku
- obě lze implementovat v $O(1)$

Abstraktní datový typ

Abstraktní datový typ (ADT) je matematický model pro určité datové struktury definované svými operacemi a omezeními na nich.

- teoretický koncept zjednodušující analýzu chování
- někdy je součástí specifikace i složitost operací
- konkrétní implementaci lze zvolit

Příklady: seznam, fronta, zásobník, množina,...

Ukázka pro zásobník

- zásobník S, prvek k, proměnná X
- S.push(k); X ← S.pop() je ekvivalentní X ← k
- operace push() a pop() mají konstantní složitost

Zajímavé odkazy

- Vizualizace operací nad polem: https://visualgo.net/en/array
- Vizualizace operací nad spojovým seznamem: https://visualgo.net/en/list
- Vizualizace datových struktur v Pythonu: https://pythontutor.com/visualize.html

4. Řazení

Sekvenční vyhledávání

Problém. Uvažme problém nalezení prvku v poli. Tento lze zřejmě řešit s lineární složitostí.

```
def search(array, k):
    for i in range(len(array)):
        if array[i] == k:
            return i

    return None
```

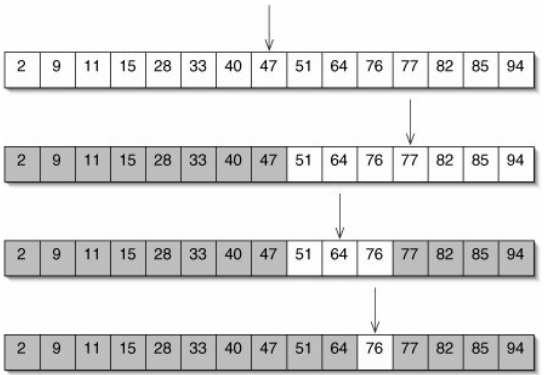
Zamyšlení. Pokud by bylo pole seřazeno, vyhledávání v něm by mohlo být snazší.

- Jak náročné je vyhledávání v seřazeném poli?
- Jak náročné je seřadit pole?

Binární vyhledávání I

Myšlenka. Pole je seřazeno ⇔ pro každý prvek pole platí, že hodnoty vlevo od něj jsou menší nebo rovny tomuto prvku a vpravo od něj větší nebo rovny.

Příklad. Vyhledávání čísla 76.



Binární vyhledávání II

Rekurzivní implementace binárního vyhledávání:

```
def binary_search(A, k, i_min, i_max):
    if i_max < i_min:
        return None

    mid = (i_min + i_max) // 2
    if k == A[mid]:
        return mid
    elif k < A[mid]:
        return binary_search(A, k, i_min, mid - 1)
    else:
        return binary_search(A, k, mid + 1, i_max)
```

Složitost. V každé iteraci se velikost prohledávané oblasti zmenší na polovinu. Složitost binárního vyhledávání je tedy $O(\log n)$.

Problém řazení

Neformální definice:

Vstup: Posloupnost prvků $a_1, \ldots, a_n$ délky n

Výstup: Neklesající permutace vstupní posloupnosti, tj.

$$\forall i \in \{1, \ldots, n-1\} : a_i \leq a_{i+1}$$

Poznámka k definici. Formální definice pracuje s prvky tvaru (a_i, D_i), kde a_i jsou klíče (podle kterých se řadí) a D_i pak další data, která mají význam v praktických aplikacích. Při analýze algoritmů je většinou zanedbáme ($D_i = \emptyset$).

Poznámka k terminologii. Někdy se nesprávně používá pro řazení výraz třídění, které značí ale spíše seskupování objektů podle daných vlastností.

Selection sort

Myšlenka. Najdi v poli nejmenší prvek a vyměň jej s prvkem na první pozici. Opakuj postup pro zbytek pole (bez prvního prvku).

```
def selection_sort(A):
    for i in range(len(A)):
        min_idx = i
        for j in range(i, len(A)):
            if A[min_idx] > A[j]:
                min_idx = j
        A[min_idx], A[i] = A[i], A[min_idx]
```

Složitost. Vnější for cyklus zjevně $\mathcal{O}(n)$, vnitřní má však různé délky (n, n - 1,..., 1). Celkem tedy 1 + 2 + ...+ n $\in \mathcal{O}(n^2)$.

Bubble sort

Myšlenka. Porovnávám postupně prvky na sousedních pozicích. Pokud jsou vůči sobě dva v nesprávném pořadí, prohodím je.

Důsledek. Po první iteraci „probublá“ největší prvek na konec pole.

```
def bubble_sort(A):
    for i in range(len(A)):
        for j in range(len(A) - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
```

Složitost. $O(n^2)$ podle stejné úvahy jako pro selection sort. Složitost lze vylepšit na příznivých datech. Jak?

Insertion sort

Myšlenka. Rozdělme (virtuálně) pole na dvě části: (1) už seřazenou a (2) dosud neseřazenou. Vždy první prvek z (2) zařazujeme korektně do (1).

```
def insertion_sort(A):
    for i in range(len(A)):
        item = A[i]
        j = i
        while j > 0 and item < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = item
```

Složitost. Pro nejhorší případ (pole seřazeno v opačném pořadí) je složitost $O(n^2)$.

Složitost problému řazení

Idea důkazu. Uvažme posloupnosti složené pouze z čísel 1,...,n, kde se každé číslo vyskytuje nejvýše jednou (= permutace této množiny). Takových je zjevně n!.

Asociativní řadicí algoritmy mohou provádět pouze porovnání dvojice prvků.

Korektní řadicí algoritmus musí pro každou takovouto posloupnost provést jiný výpočet.

Libovolný asociativní řadicí algoritmus musí tyto výpočty odlišit, tj. provést alespoň $\log_2(n!)$ binárních testů.

Důsledek. Dolní odhad složitosti řazení je $\Omega(n \log n)$.

Quick sort

Myšlenka. Vyberu prvek pole. Vlevo od něj přesunu všechny menší prvky, vpravo od něj větší. Obě tyto části pak řadím dále rekurzivně stejným postupem.

```
def quick_sort(array):
    if len(array) <= 1:
        return array
    else:
        pivot = array[0]
        smaller = [x for x in array[1:] if x < pivot]
        bigger = [x for x in array[1:] if x >= pivot]
        return quick_sort(smaller) + [pivot] + quick_sort(bigger)
```

Složitost. V nejhorším případě (libovolně seřazené pole) bude hloubka rekurzivního volání $O(n)$, v každé úrovni je potřeba rozdělit prvky do dvou částí se složitostí $O(n)$. Celkem tedy $O(n^2)$.

Merge sort

Idea. Uvažme dvě již seřazené posloupnosti. Jak z nich vytvořit jednu seřazenou?

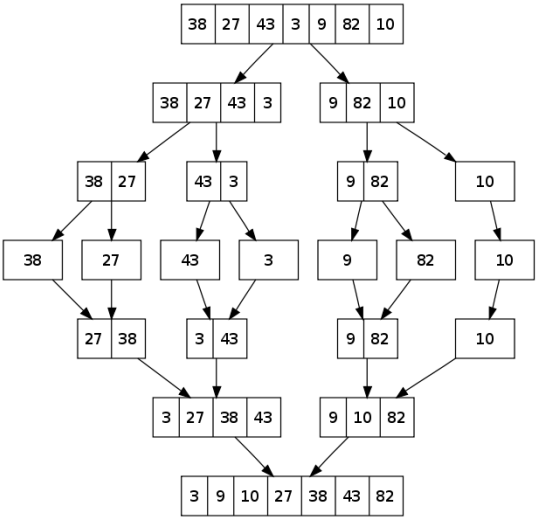
Funkce merge(A, B)

- porovnává vždy první prvky obou posloupností, menší z nich přesune do výstupní posloupnosti

Merge sort

- seřazenou posloupnost délky n získám ze dvou seřazených posloupností délky $n/2$ aplikací funkce merge
- analogicky posloupnosti velikosti $n/2$ „slévám“ ze dvou seřazených posloupností o velikosti $n/4$
- posloupnosti délky 1 jsou implicitně seřazeny

Merge sort – ukázka



Merge sort – implementace

```
def merge(A, B):
    if len(A) == 0:
        return B
    if len(B) == 0:
        return A

    if A[0] < B[0]:
        return [A[0]] + merge(A[1:], B)
    else:
        return [B[0]] + merge(A, B[1:])

def merge_sort(A):
    if len(A) <= 1:
        return A

    mid = len(A) // 2
    return merge(merge_sort(A[:mid]), merge_sort(A[mid:]))
```

Merge sort – složitost

Složitost merge sortu

- funkce merge má složitost $O(n)$
- počet úrovní volání funkce funkce merge je $O(\log n)$
- složitost merge sortu je $O(n \log n)$

Pozorování

- dolní odhad složitosti problému řazení je $\Omega(n \log n)$
- složitost merge sortu je $O(n \log n)$

Závěr. Merge sort je optimální algoritmus pro problém řazení se složitostí $O(n \log n)$.

Stabilita řadicích algoritmů

Definice. Řadicí algoritmus je stabilní, pokud neprohazuje prvky se stejným klíčem.

Příklad. Uvažme seznam lidí seřazených podle jména. Pokud jej sežadíme dále stabilním algoritmem podle příjmení, získáme seznam seřazený podle příjmení a jména.

	A	B
1	**Jméno**	**Příjmení**
2	Marek	Dostál
3	Petr	Klíč
4	Petr	Dostál
5	Prokop	Buben
6	Tomáš	Fuk

Přirozenost řadicích algoritmů

Definice. Řadicí algoritmus je přirozený, pokud dokáže využít předuspořádání vstupní posloupnosti.

Důsledek. Přirozené řadicí algoritmy řadí částečně seřazené posloupnosti v lepším čase.

Příklad. Modifikace algoritmu bubble sort:

```
def bubble_sort(A):
    for i in range(len(A)):
        swapped = False
        for j in range(len(A) - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                swapped = True

        if not swapped:
            return
```

Součet čísel

Experiment. Určete součet zadaného souboru reálných čísel.

Řešení. V Pythonu například funkce sum nebo explicitně:

```
total = 0
for item in array:
    total += item
```

Pozorování. Uvažme stejnou sadu čísel, jen v jiném (zcela náhodně zvoleném) pořadí.

```
array2 = sorted(array, key = lambda k : random.random())
```

Zřejmě platí:

```
sum(array) == sum(array2)
```

Nebo ne?

Asociativita sčítání reálných čísel

Příklad. Uvažme výpočet 1,11 + 0,001 s přesností na 3 platné číslice.

Poznámka. Počet platných desítkových číslic pro typ float je průměrně 7, pro typ double pak 16.

- 1,11 + 0,001 = 1,11
- korektní výsledek v rámci zvolené přesnosti

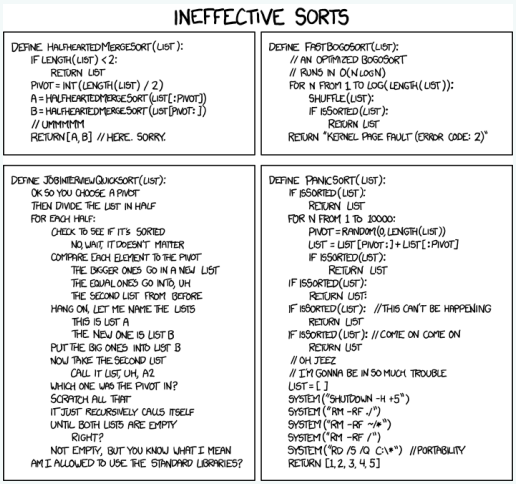
Srovnejme

- 1,11 + 0,001 + ...+ 0,001
- 1,11 + (0,001 + ...+ 0,001)

Závěr. Sčítání reálných čísel není asociativní.

Doporučení. Pokud je přesnost důležitá, sčítám čísla v pořadí podle jejich (absolutní) velikosti.

xkcd.com/1185/



StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.

Zajímavé odkazy

- Přehled řadicích algoritmů: https://visualgo.net/en/sorting
- Vizuální srovnání řadicích algoritmů:
  https://www.toptal.com/developers/sorting-algorithms
- Ukázka merge sortu (taneční soubor):
  https://www.youtube.com/watch?v=dENca26N6V4
- Obama a bubble sort: https://www.youtube.com/watch?v=k4RRi_ntQc8

5. Haldy, vyhledávací stromy

Optimální algoritmy pro problém řazení

Připomenutí. Aktuálně známe několik algoritmů se složtostí $O(n^2)$ a jeden optimální se složitostí $O(n \log n)$ – merge sort.

Nevýhoda merge sortu je dodatečná paměť, kterou potřebuje pro svůj výpočet, typicky $O(n)$ na poli.

Zamyšlení. Lze navrhnout optimální algoritmus pro problém řazení s konstatní extrasekvenční prostorovou složitostí?

Idea pro nový řadicí algoritmus. Uvažme datovou strukturu, která poskytuje efektivní operaci pro odebrání minimálního prvku. Jak lze pomocí ní implementovat řazení?

Požadavky na datovou strukturu

Pozorování. Strukturu, která by poskytovala operaci odebrání minimálního prvku v nižším než $O(n)$, nepostavím na základě pole nebo spojového seznamu.

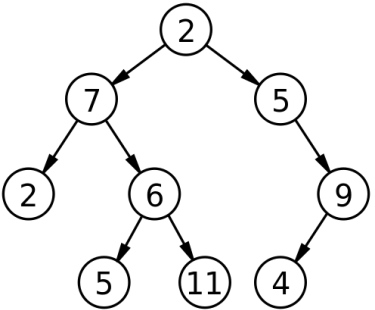
Poznámka. S lineární složitostí vyhledání minima nad polem nebo seznamem dostávám de facto selection sort.

Nápad. Uvažme strukturu, která již v sobě bude obsahovat vhodné uspořádání prvků, které nám umožní provádět operace nad prvky v nejvýše logaritmickém čase. Celkem pro n prvků tedy dostanu nejhůře $O(n \log n)$.

- Toho by mohlo jít dosáhnout, pokud všechny prvky budou „vzdáleny“ od výchozího nejvýše $O(\log n)$.
- Jak takovou strukturu navrhnout?

Binární strom I

Nápad. Uvažme rozšíření spojového seznamu, kdy každý prvek bude obsahovat dva ukazatele na další prvky. Cykly mezi prvky nepovolíme.

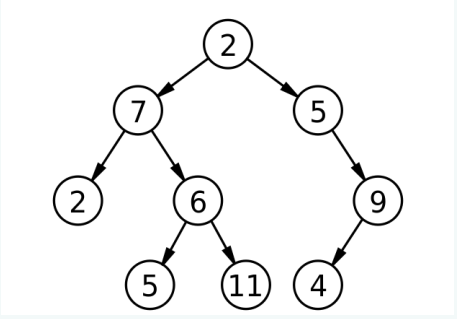


Takovou strukturu nazveme binární strom. Každý uzel (prvek) binárního stromu má nejvýše dva potomky.

Binární strom II

Názvosloví:

- kořen – výchozí prvek stomu, nevedou na něj žádné ukazatele
- list – uzel, který nemá žádné potomky
- větev – poslopnost uzlů od kořene k listu
- výška stromu – délka nejdelší větve

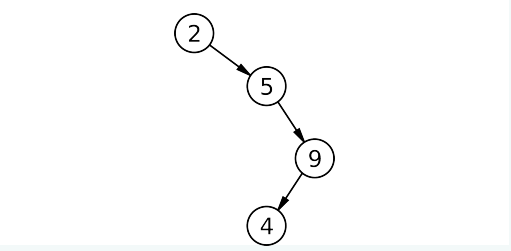


Pozorování. Složitost přístupu k jednotlivým prvkům od kořene bude nejvýše $O(h)$, kde h je výška stromu.

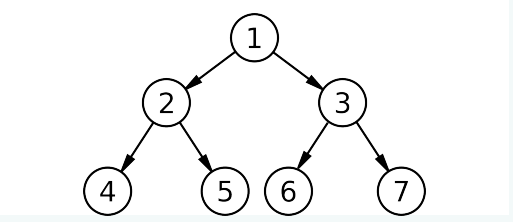
Otázka. Jaká je minimální a maximální výška binárního stromu o n uzlech?

Binární strom III

Nejhorší případ. Degenerovaný strom odpovídající de facto spojovému seznamu. Jeho výška pro n prvků je $O(n)$.



Nejlepší případ. Úplný binární stom má na h úrovních $2^h - 1$ uzlů. Obráceně, úplný binární strom o n uzlech má výšku nejvýše $O(\log_2 n)$.



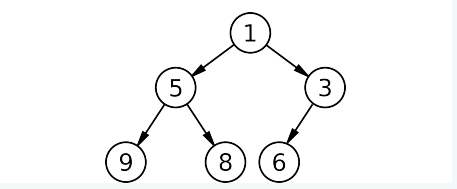
Závěr. Cílem při návrhu pro nás vhodné datové struktury je přiblížit se co nejvíce úplnému binárnímu stromu.

Halda

Halda je datová struktura poskytující efektivní operace pro přidání prvku a odebrání minima.

Binární halda má jako základ binární strom se dvěma vlastnostmi:

1. Pro každý uzel platí, že jeho potomci mají stejnou nebo větší hodnotu.
2. Na všech úrovních s výjimkou poslední je binární strom zcela zaplněn. V poslední úrovni jsou listy zaplňovany zleva doprava.



Poznámka. Druhá vlastnost zaručuje, že výška haldy je logaritmická vzhledem k počtu prvků.

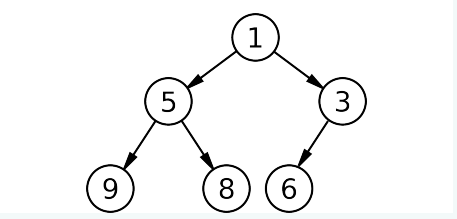
Poznámka. Uvedená definice platí pro tzv. minimovou haldu, která má v kořeni minimum. Analogicky lze definovat maximovou haldu.

Operace nad haldou

Zjištění minima. Minimum haldy je v jejím kořeni, složitost operace je zjevně $\mathcal{O}(1)$.

Přidání prvku. Nový prvek přidám na první volné místo. Musím ovšem zajistit, že bude zachováno uspořádání na větvích.

- pokud je prvek menší než jeho rodič, dojde k jejich prohození
- těchto prohození může být nejvýše $\mathcal{O}(\log n)$, kdy se nový prvek dostane do kořene haldy



Odstranění minima. Vyměním hodnotu kořene a posledního prvku, který pak mohu snadno odstranit.

- pokud je nový kořen větší než jeho potomci, je potřeba menšího z nich s kořenem prohodit a postupovat obdobně níže směrem k listům → celkově až $\mathcal{O}(\log n)$

Heap sort

Heap sort je řadicí algoritmus, který je postaven nad operacemi haldy. Má dvě fáze:

1. Přidání všech prvků do haldy.
2. Opakované odebírání minima.

Složitost Heap sortu

- první fáze má složitost $\mathcal{O}(n)$, druhá pak $\mathcal{O}(n \log n)$
- celkově tedy $\mathcal{O}(n \log n)$
- Heap sort je optimální algoritmus

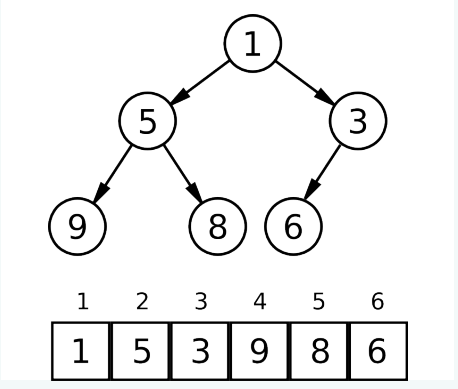
Zamyšlení. Ačkoliv asymptoticky optimální, navržená implementace vyžaduje stále lineární množství paměti pro vytvoření haldy. Jde to udělat lépe?

Implementace haldy v poli

Nápad. Každou binární haldu (obecně i každý zleva zarovnaný binární strom) lze reprezentovat v poli.

Pro každý uzel platí:

- je-li uzel uložen na indexu i, jeho levý potomek je na indexu $2i$
- analogicky pravý potomek pak na indexu $2i + 1$



Heap sort pak spočívá ve vytvoření maximové haldy přeuspořádáním prvků v poli (1. fáze) a následně odebrání všech prvků (2. fáze), které budou tvořit od konce seřazenou posloupnost.

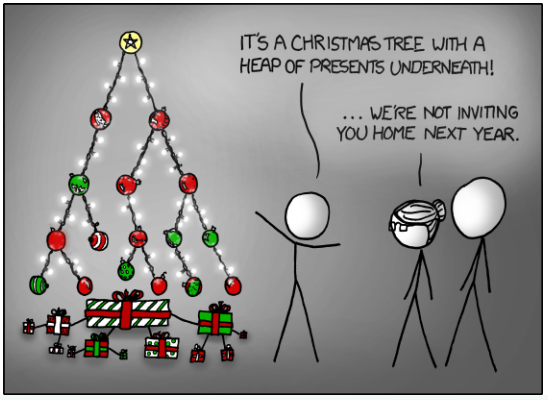
Heap sort – implementace

```
def sift_down(A, start, end):
    root = start
    while True:
        child = 2 * root + 1
        if child >= end:
            break
        if child + 1 < end and A[child] < A[child + 1]:
            child += 1
        if A[root] < A[child]:
            A[root], A[child] = A[child], A[root]
            root = child
        else:
            break

def heap_sort(A):
    for i in range(len(A) // 2, -1,  -1):
        sift_down(A, i, len(A))

    for i in range(len(A)):
        A[0], A[len(A) - i - 1] =  A[len(A) - i - 1], A[0]
        sift_down(A, 0, len(A) - i - 1)
```

Haldy v životě informatika (http://xkcd.com/835/)



Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt.

Odbočka: Prioritní fronta

Opakování. Aktuálně umíme implementovat jednoduchou frontu pomocí spojového seznamu nebo pole pevné délky.

Zobecnění. Přiřaďme každému prvku prioritu, která bude určovat v jakém pořadí bude z fronty odstraněn.

Pozorování. Stávající implementace v tomto případě neposkytují lepší než lineární složitost pro alespoň jednu z operací fronty, tedy přidání nebo odebrání prvku.

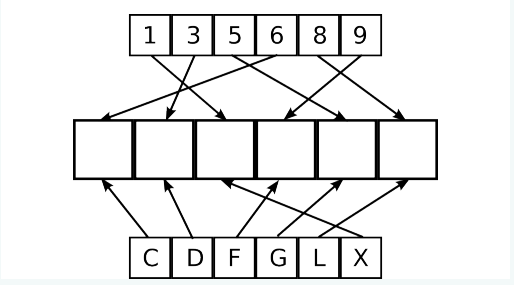
Řešení. Přímočarou implemetací prioritní fronty je binární halda.

- přidání prvku – $\mathcal{O}(\log n)$
- odebrání prvku (odpovídá odebrání minima) – $\mathcal{O}(\log n)$
- pokud je implentována v poli, k přidávání nebo odebírání prvků dochází jen na jeho konci $\rightarrow$ lze využít dynamického pole

Indexy

Shrnutí. V současnosti umíme seřadit data podle zvoleného klíče. Vyhledávat v seřazeném poli lze efektivně pomocí binárního vyhledávání v logaritmickém čase.

- není nutné řadit vlastní (často objemná) data, ale stačí seřadit klíče
- takových uspořádání, tzv. indexů, je možné udržovat více najednou



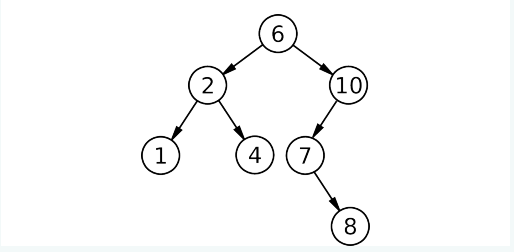
Nevýhoda současného řešení spočívá v obtížné změně velikosti indexů. Přidání nebo odebrání prvku má lineární složitost.

Nápad. Využijme místo pole jako základ indexu binární strom.

Binární vyhledávací strom

Binární vyhledávací strom (BST) je datová struktura založena na binárním stromě, kde pro každý uzel platí, že:

- všechny uzly v jeho levém podstromu mají menší ohodnocení než on sám
- analogicky všechny uzly v pravém podstromu mají ohodnocení větší

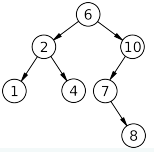


Základní operace nad BST:

- přidání prvku
- odebrání prvku
- vyhledání prvku

Operace nad BST

Vyhledání prvku. Analogie binárního vyhledávání – porovnáváme hledanou hodnotu s ohodnocením aktuálního prvku. Je-li menší, vyhledávání pokračuje v jeho levém podstromu, je-li větší, pak v pravém.



Přidání/odebrání prvku. Nejprve je potřeba nalézt místo, kde k vlastnímu přidání/odebrání dojde. Na konci pak zajistit, že změněný strom je stále BST.

Pozorování. Složitost těchto operací bude záviset na výšce stromu, v nejhorším případě bude tedy lineární (uvažme např. vytvoření BST ze seřazené posloupnosti klíčů).

Závěr. Pro dosažení efektivity je potřeba implementovat operace přidání a odebrání prvku tak, aby udržovaly logaritmickou výšku BST.

Zajímavé odkazy

- Vizualizace operací nad binární haldou: https://visualgo.net/en/heap
- Vizualizace operací nad binárním vyhledavacím stromem: https://visualgo.net/en/bst

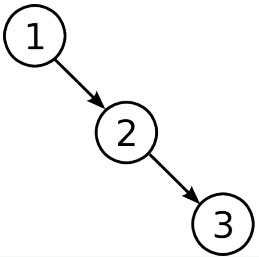
6. Vyhledávací stromy, hashovací tabulky, trie.

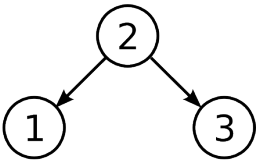
Nevýhody BST

Opakování. Binární vyhledávací strom představuje koncepčně jednoduchou formu indexu nad daty.

Jeho hlavní nevýhoda je až lineární výška.

Myšlenka. Modifikujme operace přidání a odebrání prvku tak, aby zbytečně nezvyšovaly výšku stromu.





AVL stromy

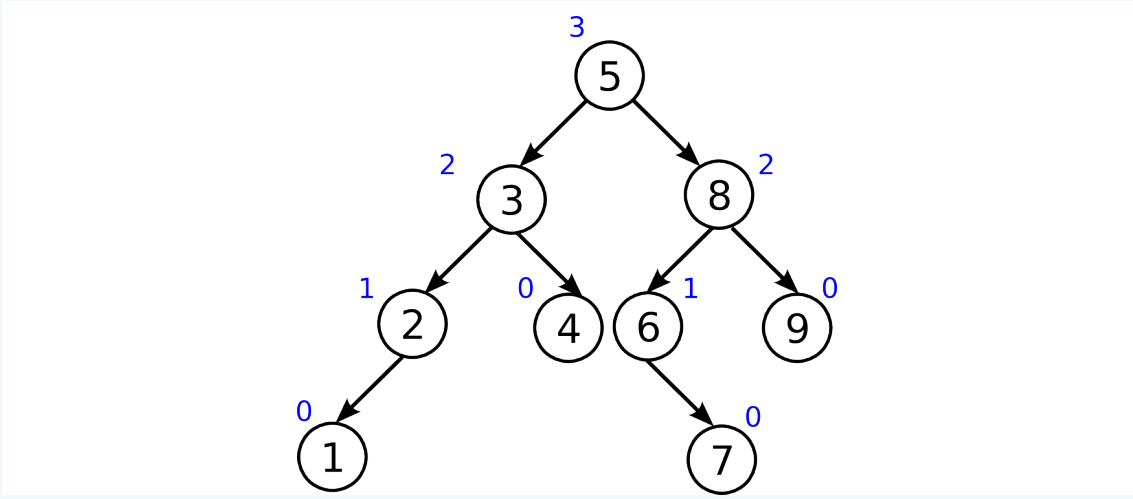
AVL strom je binární vyhledávací strom, který navíc splňuje následující podmínku:

Pro každý uzel AVL stromu platí, že výška jeho levého a pravého podstromu se liší nejvýše o 1.

Poznámka. Definiční podmínka zaručuje, že výška AVL stromu je nejvýše asi $1{,}5 \cdot \log(n)$, kde n je počet jeho prvků.

Implementace. Každý uzel AVL stromu bude mít u sebe navíc i informaci o své výšce. V případě porušení definiční podmínky při některé z operací bude nutné strom upravit.

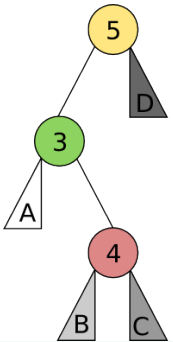
AVL strom – ukázka

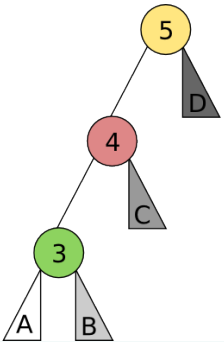


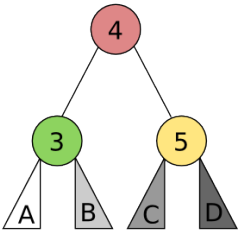
Operace nad AVL stromy

Vyhledání prvku. Úplně stejná operace jako v obyčejném BST. Díky zaručené výšce AVL stromu ovšem nyní nejvýše $O(\log n)$.

Přidání/odebrání prvku. Probíhá obdobně jako u BST. Pokud je narušena vyváženost AVL stromu (porušení definiční podmínky), probíhá vyvažování pomocí tzv. rotací.

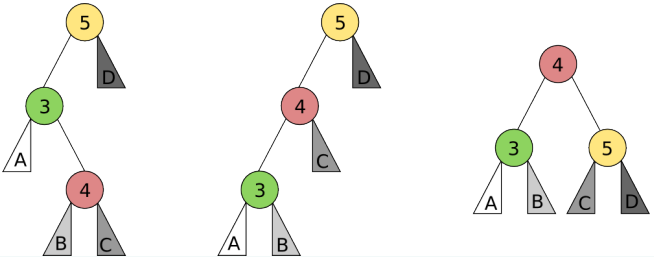






Operace nad AVL stromy II

Rotace



- pouze lokální změna datové struktury
- konstantní složitost
- při přidání prvku stačí nejvýše 1
- při odebrání prvku je jejich počet omezen výškou stromu

Složitost přidání/odebrání prvku v AVL stromu je $O(\log n)$.

Logaritmická výška I

Shrnutí. AVL stromy poskytují z pohledu asymptotické složitosti optimální rozšíření BST.

Poznámka. Na BST jsou také založeny např. červeno-černé stromy, které mají logaritmickou výšku (i když větší než AVL), nicméně poskytují v praxi rychlejší vyvažování.

Příklad. Uvažme data o velikosti $n = 22\,000\,000$ (přibližný počet sloučenin v databázi PubChem). Vybudujeme nad těmito daty index na bázi binárního stromu.

- výška úplného binárního stromu by byla 25
- v rámci teorie ideální, v praxi může být i to příliš

Logaritmická výška II

Nápad. Uvažme vyvážený strom (= s logaritmickou výškou), jehož arita k je větší než 2.

- zvolme např. $k = 100$
- pak výška tohoto stromu pro stejnou velikost dat bude 4

Realizace. Na této myšlence jsou založeny tzv. B stromy.

- logaritmická složitost operací
- typicky pro velké objemy dat
- existují různé varianty (B/B+/B*)

Využití B stromů

- souborové systémy (NTFS, Ext4, btrfs)
- indexy pro databáze

Hashovací tabulka

Shrnutí. Vyvážené vyhledávací stromy poskytují základní operace v logaritmickém čase. Jde dosáhnout i konstantní složitosti?

Nápad. Uvažme pole o velikosti M a tzv. hashovací funkci, která každé hodnotě klíče přiřadí index i do tohoto pole, kde se daná položka bude nacházet.

Hashovací funkce – příklad

- uvažme klíče k jako přirozená čísla
- pak např. $i = H(k) = k \mod M$
- konstantní složitost výpočtu indexu

Hashovací funkce

Ideální případ. Uvažme hashovací tabulku o velikosti M s následujícími vlastnostmi:

- počet vkládaných prvků $n \leq M$
- hashovací funkce má konstantní složitost
- hashovací funkce je prostá, tj. $\forall k_1, k_2 : H(k_1) = H(k_2) \rightarrow k_1 = k_2$
- pak složitost přidání, vyhledání a odebrání prvku je konstantní

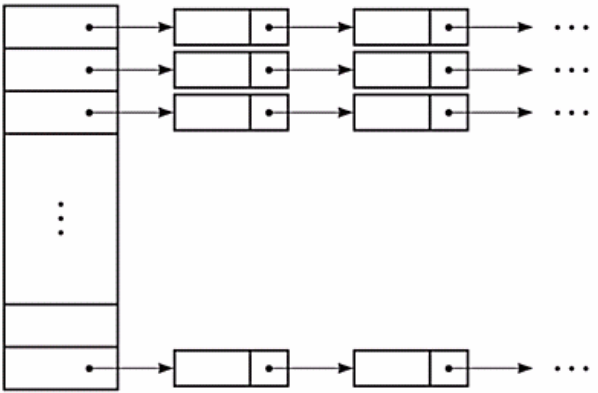
Kolize. V praxi ovšem tohoto není často možné dosáhnout, hashovací funkce nebývá prostá → pro dva různé klíče získáme stejnou hodnotu hashovací funkce.

Příklady řešení pro $n \leq M$

- lineární hashování – je-li index i obsazen, zkusím $i + 1$ atd.
- dvojité hashování – při kolizi volím jinou hashovací funkci

Řešení kolizí I

Řešení kolizí v obecném případě spočívá v možnosti vytvoření spojového seznamu pro každý index pole.



Řešení kolizí II

Vlastnosti

- ideální hashovací funkce rozděluje klíče rovnoměrně
- délka každého seznamu je pak průměrně n/M
- přidání prvku v $\mathcal{O}(1)$
- vyhledání/odebrání prvku ale v $\mathcal{O}(n)$ – stejné jako u spojového seznamu

V praxi je ale často výrazně lepší než obyčejný spojový seznam. Srovnejme několik příkladů s ideální hashovací funkcí a $n = 5000$:

- $M = 1000$
- $M = 5000$
- $M = 10000$

Závěr. Hashovací tabulka je velmi efektivní, pokud známe rozložení prvků (→ hashovací funkce) a jejich počet (→ velikost tabulky).

Hashovací tabulka – poznámky

Rozšíření. V případě, kdy není počet prvků znám dopředu, lze měnit velikost i vlastní tabulky.

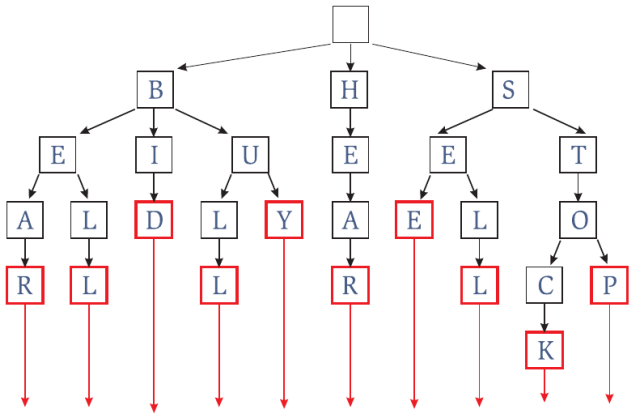
- základem je dynamické pole
- je potřeba sledovat zaplnění tabulky, tedy poměr n/M
- vysoká zaplněnost přináší nižší výkon
- změna velikosti nastává při dosažení zvolené hranice zaplněnosti (např. 2/3 nebo 3/4)

Použití

- implementuje ADT asociativní pole (slovník), které uchovává dvojice typu (klíč, hodnota)
- v Pythonu typ dict, v Javě HashMap, v C++ std::unordered_map

Trie

Trie (prefixový strom) je stromová datová struktura zejména vhodná pro uložení klíčů, které jsou řetězci.



Trie – příklad implementace

Příklad. Uvažme řetězce složené ze znaků anglické abecedy

- každý uzel trie obsahuje pole 26 ukazatelů na další prvky
- v každém uzlu uložíme příznak, který říká, zdali je tento uzel validním klíčem (listy jsou implicitně)

Složitost operací

- nechť h je délka nejdelšího řetězce
- pak trie má výšku h
- operace přidání, vyhledání a odebrání prvku mají všechny složitost $O(h)$
- neúspěšné hledání může skončit v kterékoliv úrovni (srovnejme s BST)

Zajímavé odkazy

- Vizualizace operací nad AVL: https://visualgo.net/en/bst
- Vizualizace operací nad hashovací tabulkou: https://visualgo.net/en/hashtable
- Vizualizace operací nad trií:
  https://www.cs.usfca.edu/~galles/visualization/Trie.html

7. Grafy.

Vyhledávání v databázích I

Opakování. Umíme efektivně vyhledávat a řadit objekty podle různých klíčů v případě, že je na těchto klíčích definováno uspořádání ($\leq$).

Vyhledávání molekul. Mějme molekulu a chtějme zjistit, zdali se již vyskytuje v dané sadě sloučenin (= databázi).

- Záznamy o molekulách často obsahují jednoznačné identifikátory (řetězce znaků) $\rightarrow$ umíme.

Problém. Tyto informace ale nemusí být dostupné. K dispozici máme však minimálně:

- údaje o atomech (pozice, typy)
- vazby mezi atomy

Příklad molekuly – formát MOL

```
702
 -OEChem-03301510303D

  9  8  0     0  0  0  0  0  0999 V2000
   -1.1712    0.2997    0.0000 O   0  0  0  0  0  0
   -0.0463   -0.5665    0.0000 C   0  0  0  0  0  0
    1.2175    0.2668    0.0000 C   0  0  0  0  0  0
   -0.0958   -1.2120    0.8819 H   0  0  0  0  0  0
   -0.0952   -1.1938   -0.8946 H   0  0  0  0  0  0
    2.1050   -0.3720   -0.0177 H   0  0  0  0  0  0
    1.2426    0.9307   -0.8704 H   0  0  0  0  0  0
    1.2616    0.9052    0.8886 H   0  0  0  0  0  0
   -1.1291    0.8364    0.8099 H   0  0  0  0  0  0
  1  2  1  0  0  0  0
  1  9  1  0  0  0  0
  2  3  1  0  0  0  0
  2  4  1  0  0  0  0
  2  5  1  0  0  0  0
  3  6  1  0  0  0  0
  3  7  1  0  0  0  0
  3  8  1  0  0  0  0
```

Vyhledávání v databázích II

Intuice. Porovnání na základě pozic jednotlivých atomů a vazeb představuje výpočetně netriviální problém. Současné databáze navíc obsahují stovky tisíc až miliony struktur.

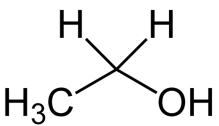
Návrh řešení. Proveďme vyhledávání v několika fázích, které budou postupně omezovat množinu přípustných struktur. Postupujme od nejjednodušších metod po složitější.

1. jednoduché deskriptory (př. sumární vzorec)
2. využití znalosti topologie, podstruktur
3. porovnání pozic atomů v prostoru

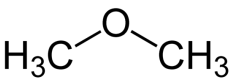
Příklad. Porovnání molekul podle sumárních vzorců v rozumném čase výrazně redukuje množinu kandidátů. Nicméně samo o sobě nestačí.

Vyhledávání v databázích III

Omezení. Pomocí sumárního vzorce nelze rozlišit izomery.



Ethanol (Alcohol)



Dimethylether

Topologie. Je nutné přidat další informace o struktuře sloučenin – propojení vazbami.

Problém. Potřebujeme nalézt vhodnou datovou strukturu pro reprezentaci molekuly.

3. fáze. Ani toto rozlišení obecně nestačí (stereoizomery), ale získané výsledky lze použít jako výchozí bod pro další algoritmy.

Graf

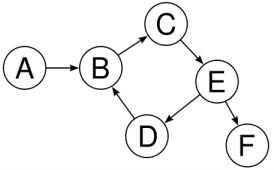
Definice. Graf $G = (V, E)$, kde V je množina uzlů (vrcholů) a E je množina hran.

Typy grafů

- orientovaný – hrany jsou uspořádané dvojice (u, v)
- neorientovaný – hrany jsou dvouprvkové podmnožiny $\{u, v\}$

Příklad orientovaného grafu

- $G = (V, E)$
- $V = \{A, B, C, D, E, F\}$
- $E = \{(A, B), (B, C), (C, E), (D, B), (E, D), (E, F)\}$



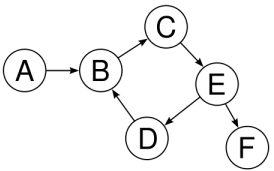
Reprezentace grafu

Minimální požadavky na datovou strukturu

- dotaz na existenci hrany v grafu
- sousedé daného vrcholu

Triviální řešení představuje obyčejný seznam (pole) hran. Nicméně výše zmíněné operace pak nelze implementovat efektivně.

- $G = (V, E)$
- $V = \{A, B, C, D, E, F\}$
- $E = \{(A, B), (B, C), (C, E), (D, B), (E, D), (E, F)\}$



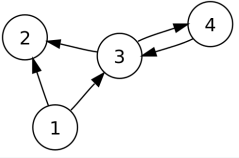
Matice sousednosti

Matice sousednosti. Vytvořme pro graf $G = (V, E)$ matici A o rozměrech $|V| \times |V|$ s vlastností:

$$A_{i,j} = 1 \leftrightarrow (i, j) \in E$$

Příklad

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



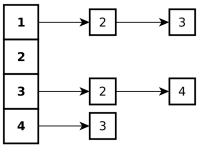
Vlastnosti

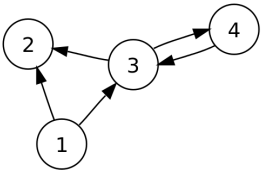
- dotaz na přítomnost hrany je konstatní operace
- seznam následníků daného vrcholu v lineárním čase
- potřeba $|V|^2$ paměti $\rightarrow$ vhodné pro husté grafy ($|E| \approx |V|^2$)

Seznam následníků

Seznam následníků. Uvažme pole ukazatelů na seznamy následníků daných vrcholů.

Příklad





Vlastnosti

- dotaz na přítomnost hrany je lineární operace
- seznam následníků v lineárním čase
- pouze $|V| + |E|$ paměti → vhodné pro řídké grafy ($|E| \approx |V|$)

Procházení grafu

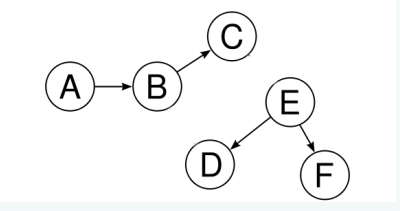
Cíl. Projít všechny vrcholy grafu dostupné ze zvoleného výchozího.

Naivní řešení. Projít postupně seznam vrcholů od začátku do konce (podobně jako u obyčejného pole).

- zjevně lineární operace
- nerespektuje strukturu grafu
- graf nemusí být souvislý → projdeme i jeho nedosažitelné části

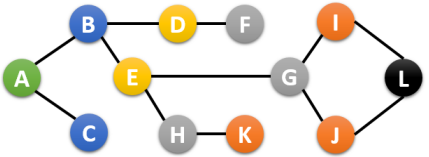
Ideální řešení

- zachová lineární složitost
- každý vrchol projde právě jednou
- odstraní výše uvedené nedostatky



Procházení do šířky

Breadth First Search (BFS) prochází graf po jednotlivých úrovních – než projde vrcholy vzdálené (co do počtu hran) n od výchozího, projde předtím všechny vrcholy vzdálené $n-1$.



Breadth-First Search (BFS)

Vlastnosti

- procházíme nejdříve všechny přímé následníky vrcholů
- pro uložení pořadí, ve kterém vrcholy prohledáváme, používáme frontu
- lineární složitost vzhledem k velikosti grafu – $\mathcal{O}(|V| + |E|)$

Procházení do šířky – pseudokód

```
1:  function BFS(G, u) is
2:      Nechť Q je prázdná fronta
3:      Enqueue(Q, u)
4:      Označ u jako navštívený
5:      while Q není prázdná do
6:          v ← Dequeue(Q)
7:          for all (v, w) ∈ E do
8:              if w není navštívený then
9:                  Označ w jako navštívený
10:                 Enqueue(Q, w)
11:             fi
12:         done
13:     done
14: end
```

Procházení do hloubky

Depth First Search (DFS) prochází graf „dokud to jde“, pak se vrací do posledního místa, kde existuje neprozkoumaná cesta, kterou pak pokračuje dále (= obvyklé prohledávání bludiště).

Vlastnosti

- lineární algoritmus – $O(|V| + |E|)$
- často v rekurzivní podobě, iterativní využívá zásobník

```
1: function DFS(G, u) is
2:     Označ u jako navštívený
3:     for all (u, v) ∈ E do
4:         if v není navštívený then
5:             DFS(G, v)
6:         fi
7:     done
8: end
```

Procházení do hloubky (iterativně) – pseudokód

```
 1: function DFS(G, u) is
 2:     Nechť S je prázdný zásobník
 3:     Push(S, u)
 4:     Označ u jako navštívený
 5:     while S není prázdný do
 6:         v ← Pop(S)
 7:         for all (v, w) ∈ E do
 8:             if w není navštívený then
 9:                 Označ w jako navštívený
10:                 Push(S, w)
11:             fi
12:         done
13:     done
14: end
```

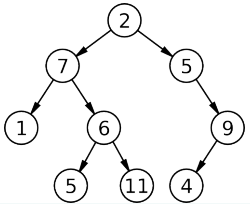
Otázka. Čím se liší pseudokód pro BFS a DFS?

Procházení binárního stromu

BFS → procházení po úrovních

Varianty DFS

- pre-order → 1. uzel, 2. levý podstrom, 3. pravý podstrom
- in-order → 1. levý podstrom, 2. uzel, 3. pravý podstrom
- post-order → 1. levý podstrom, 2. pravý podstrom, 3. uzel



Pořadí procházení vrcholů

- BFS: 2, 7, 5, 1, 6, 9, 5, 11, 4
- DFS pre-order: 2, 7, 1, 6, 5, 11, 5, 9, 4
- DFS in-order: 1, 7, 5, 6, 11, 2, 5, 4, 9
- DFS post-oder: 1, 5, 11, 6, 7, 4, 9, 5, 2

Otázka. Co kdybychom použili DFS in-order na BST?

Zajímavé odkazy

- Vizualizace různých reprezentací grafů: https://visualgo.net/en/graphds
- Vizualizace průchodů grafem (BFS/DFS) nad AVL: https://visualgo.net/en/dfsbfs
- Zajímavé aplikace teorie grafů: https://www.fi.muni.cz/~xpelanek/ucitele/data/Prezentace%20-%20zajimave%20aplikace%20teorie%20grafu.pdf

8. Nejkratší vzdálenosti.

Nejkratší vzdálenosti v grafu

Problém. Určete nejkratší vzdálenost mezi vrcholy u a v v grafu.

Pozorování. V tuto chvíli můžeme posuzovat vzdálenost pouze jako počet hran na cestě mezi u a v.

Takový výpočet realizuje prohledávání do šířky (BFS) v lineárním čase vůči velikosti grafu.

Rozšíření. Uvažme případ, kdy bychom chtěli přiřadit hranám na cestě různou váhu.

Graf $G = (V, E, w_e)$ nazveme hranově ohodnocený, w_e je funkce, která každé hraně přiřazuje její ohodnocení – reálné číslo.

Nejkratší vzdálenost mezi dvěma vrcholy v grafu je minimální součet ohodnocení hran některé cesty mezi těmito vrcholy.

Matice vzdáleností

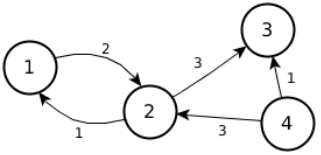
Poznámka. Na hranově neohodnocený graf se lze dívat jako na speciální případ ohodnoceného, kde $\forall (u, v) \in E : w_e(u, v) = 1$.

Matice vzdáleností W je rozšíření matice sousednosti:

$$W_{i,j} = \begin{cases} 0 & \text{pro } i = j \\ w_e(i, j) & \text{pro } (i, j) \in E \\ \infty & \text{pro } (i, j) \notin E \end{cases}$$

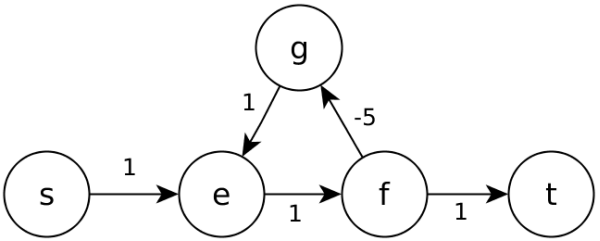
Příklad

$$W = \begin{pmatrix} 0 & 2 & \infty & \infty \\ 1 & 0 & 3 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 3 & 1 & 0 \end{pmatrix}$$



Záporný cyklus

Příklad. Určete nejkratší vzdálenost mezi vrcholy s a t v grafu:



Pozorování. Graf obsahuje cyklus záporné délky (vrcholy e, f, g), nejkratší vzdálenost mezi s a t není definována.

Poznámka. Uvažme graf bez cyklů záporné délky. Každá nejkratší cesta mezi dvěma vrcholy obsahuje libovolný vrchol nejvýše jednou.

Bellman-Fordův algoritmus

Pozorování. Z předchozí poznámky vyplývá, že nejkratší cesta mezi dvěma vrcholy v grafu obsahuje nejvýše $|V| - 1$ hran.

Relaxace hrany. Nechť (u, v) je hrana v grafu G s ohodnocením $w_e(u, v)$ a hodnoty $u.d$ a $v.d$ jsou v daném okamžiku nejkratší nalezené vzdálenosti do u, resp. do v, z výchozího vrcholu. Zjevně pak platí, že:

$$v.d = \min(v.d, u.d + w_e(u, v))$$

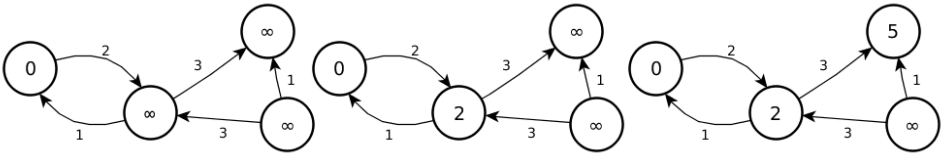
Tato (případná) změna ohodnocení $v.d$ se nazývá relaxací.

Bellman-Fordův algoritmus je založen na těchto dvou principech.

- počítá nejkratší vzdálenosti z výchozího vrcholu do všech ostatních (1:N)
- $(|V| - 1)$krát relaxuje všechny hrany

Bellman-Fordův algoritmus – poznámky

Složitost Bellman-Fordova algoritmu je $\mathcal{O}(|V| \cdot |E|)$, relaxace hrany má totiž zřejmě konstatní složitost.



Pozorování

- Pokud při některé z iterací nedojde ke změně hodnoty $v.d$ pro žádný vrchol v, pak je možné výpočet ukončit.
- Provedením jedné iterace výpočtu navíc lze v grafu detekovat záporné cykly.
- Pokud dojde k relaxaci hrany, lze u koncového vrcholu nastavit ukazatel na jeho předchůdce → rekonstrukce cesty.

Bellman-Fordův algoritmus – pseudokód

```
1:  function Bellman-Ford(G = (V, E, w_e), s) is
2:      ∀v ∈ V : v.d ← ∞; s.d ← 0
3:      for i ← 1 to |V| − 1 do
4:          for all (u, v) ∈ E do
5:              if v.d > u.d + w_e(u, v) then
6:                  v.d ← u.d + w_e(u, v)
7:              fi
8:          done
9:      done
10:     for all (u, v) ∈ E do
11:         if v.d > u.d + w_e(u, v) then
12:             Error : Negative cycle detected
13:         fi
14:     done
15: end
```

Dijkstrův algoritmus

Dijkstrův algoritmus představuje odlišný způsob řešení problému nejkratších vzdáleností v grafu.

- Opět řeší problém typu 1:N.
- Vyžaduje graf s nezáporným ohodnocením všech hran.

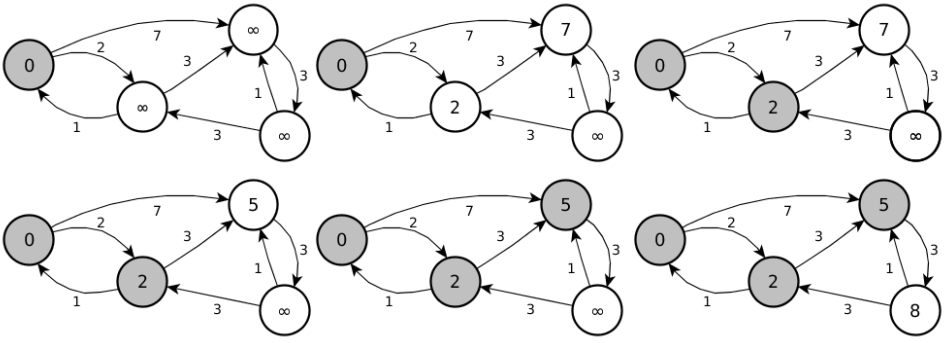
Myšlenka. Pokud je posloupnost $u, u_1, \ldots, u_n, v$ nejkratší cesta z u do v, pak posloupnost $u, u_1, \ldots u_k$, kde $k \leq n$ je nejkratší cesta z u do u_k.

Výpočet algoritmu. V každé iteraci rozšiřujeme množinu vrcholů, do kterých již známe nejkratší vzdálenost z výchozího.

Dijkstrův algoritmus

Příklad

1. Pokud je Q množina vrcholů s dosud neurčenou nejkratší vzdáleností, tak z ní nejprve vybereme vrchol u, jehož ohodnocení $u.d$ je nejnižší.
2. Poté přepočítáme všechny vzdálenosti do vrcholů z Q, do kterých vede z u hrana.



Dijkstrův algoritmus – pseudokód

```
1:  function Dijkstra(G = (V, E, w_e), s) is
2:      ∀v ∈ V : v.d ← ∞
3:      s.d ← 0
4:      Q ← V
5:      while Q není prázdná do
6:          u ← t ∈ Q s minimální t.d
7:          Odstraň u z Q
8:          for all v : (u, v) ∈ E do
9:              if v.d > u.d + w_e(u, v) then
10:                 v.d ← u.d + w_e(u, v)
11:             fi
12:         done
13:     done
14: end
```

Dijkstra – volba datové struktury

Složitost Dijkstrova algoritmu je dána volbou datové struktury pro Q. Zajímají nás dvě operace:

- f_{ext} – extrakce prvku s minimálním klíčem ($|V|$ operací)
- f_{dec} – snížení klíče $v.d$ (nejvýše $|E|$ operací)

Seznam vrcholů

- snížení klíče – $O(1)$
- extrakce minimálního prvku – $O(|V|)$
- celkem $O(|E| + |V|^2) = O(|V|^2)$

Binární halda

- snížení klíče i extrakce minima – $O(\log |V|)$
- celkem $O(\log |V| \cdot (|V| + |E|))$

Nejkratší vzdálenosti mezi všemi dvojicemi vrcholů

Pozorování. Určit nejkratší vzdálenost mezi dvěma vrcholy (1:1) má stejnou složitost jako určit nejkratší vzdálenost z jednoho vrcholu do všech ostatních (1:N).

Nejkratší mezi všemi dvojicemi vrcholů lze spočítat např. využitím $|V|$ volání Dijkstrova nebo Bellman-Fordova algoritmu, kdy v každém výpočtu volíme jiný výchozí vrchol. Jde to i lépe?

Floyd-Warshallův algoritmus počítá nejkratší vzdálenosti mezi všemi dvojicemi vrcholů v čase $O(|V|^3)$. Navíc oproti Dijkstrovu algoritmu umí pracovat se zápornými hranami.

Floyd-Warshallův algoritmus

Myšlenka. Označme $d_{i,j}^{(k)}$ délku nejkratší cesty mezi vrcholy i a j, kde na této cestě jsou vrcholy pouze z množiny $\{1, \ldots, k\}$. Zjevně $d_{i,j}^{(0)} = w_e(i, j)$ a požadovaný výsledek odpovídá $d_{i,j}^{(|V|)}$.

Mohou nastat dvě možnosti pro $d_{i,j}^{(k)}$:

1. k není součástí nejkratší cesty $\rightarrow d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$
2. k je součástí nejkratší cesty $\rightarrow d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$

Odtud tedy:

$$d_{i,j}^{(k)} = \min \left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\}$$

Poznámka. Pokud $d_{i,i}^{(|V|)} < 0$ pro nějaké i, pak graf obsahuje cyklus záporné délky.

Floyd-Warshallův algoritmus – pseudokód

```
1:  function Floyd-Warshall(G = (V, E, w_e)) is
2:      ∀(u, v) ∈ {1, ..., |V|} × {1, ..., |V|} : dist(u, v) ← ∞
3:      ∀v ∈ V : dist(v, v) ← 0
4:      ∀(u, v) ∈ E : dist(u, v) ← w_e(u, v)
5:      for k ← 1 to |V| do
6:          for i ← 1 to |V| do
7:              for j ← 1 to |V| do
8:                  if dist(i, j) > dist(i, k) + dist(k, j) then
9:                      dist(i, j) ← dist(i, k) + dist(k, j)
10:                 fi
11:             done
12:         done
13:     done
14: end
```

Zajímavé odkazy

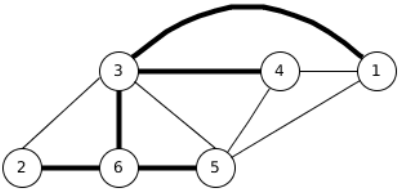
- Vizualizace algoritmů pro výpočet nejkratších vzdáleností:
  https://visualgo.net/en/sssp

9. Minimální kostry.

Kostra grafu

Kostra neorientovaného grafu G je podgraf, který obsahuje všechny vrcholy G a je stromem.

Opakování. Každá kostra grafu má $|V|$ vrcholů a $|V| - 1$ hran.



Poznámka. Počet různých koster v úplném grafu je n^{n-2}.

Minimální kostra grafu

Minimální kostra hranově ohodnoceného neorientovaného grafu G je taková kostra G, která má součet ohodnocení hran nejnižší.

Poznámka. Minimální kostra nemusí být určena jednoznačně. Uvažme např. graf, pro který platí $\forall \{u, v\} \in E : w_e(u, v) = 1$.

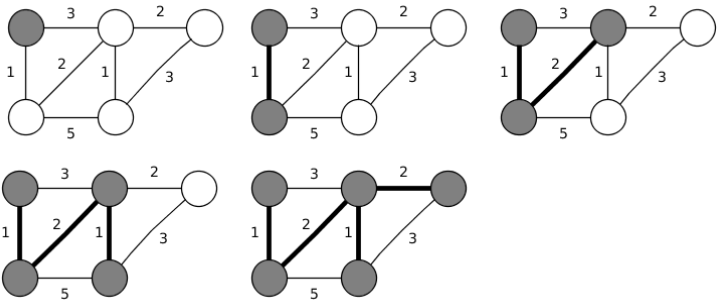
Zajímavost. Motivací pro řešení problému minimální kostry byla elektrifikace jižní Moravy (Otakar Borůvka, 1925). Popsáno v článcích:

- O jistém problému minimálním
- Příspěvek k řešení otázky ekonomické stavby elektrovodných sítí

Primův algoritmus

Princip

- budování kostry začíná v libovolném vrcholu grafu
- v každém kroce algoritmu je do kostry přidána minimální hrana sousedící s některým již v kostře obsaženým vrcholem tak, aby nebyl utvořen cyklus



Primův algoritmus – poznámky

Zajímavost. Tento algoritmus nejprve popsal český matematik Vojtěch Jarník (1930), později (1957, 1959) byl znovuobjeven nezávisle na sobě Robertem Primem a Edsgerem Dijkstrou.

Implementace

- potřeba udržovat seznam hran, které sousedí s aktuálně zpracovanými vrcholy kostry → rozdělení vrcholů na dvě množiny
- výběr minimální hrany – struktura podobná jako u Dijkstrova algoritmu

Primův algoritmus – pseudokód

```
1:  function Prim(G = (V, E, w_e), s) is
2:      ∀v ∈ V : v.key ← ∞
3:      s.key ← 0, s.p ← NULL
4:      Q ← V
5:      while |Q| ≠ 0 do
6:          u ← t ∈ Q s minimálním .key
7:          Q ← Q \ {u}
8:          for all {u, v} ∈ E do
9:              if v ∈ Q ∧ w_e(u, v) < v.key then
10:                 v.key ← w_e(u, v)
11:                 v.p ← u
12:             fi
13:         done
14:     done
15: end
```

Primův algoritmus – volba datové struktury

Pozorování. Složitost Primova algoritmu je dána volbou datové struktury pro Q.

Seznam vrcholů

- odstranění minima – $\mathcal{O}(|V|)$
- snížení klíče – $\mathcal{O}(1)$
- celkem $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$

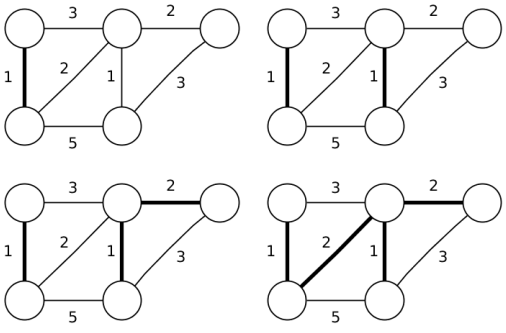
Binární halda

- odstranění minima – $\mathcal{O}(\log |V|)$
- snížení klíče – $\mathcal{O}(\log |V|)$
- celkem $\mathcal{O}(|V| \log |V| + |E| \log |V|) = \mathcal{O}(|E| \log |V|)$

Kruskalův algoritmus

Princip

- každý vrchol v grafu představuje jednu komponentu kostry
- v každém kroku jsou dvě komponenty spojeny minimální hranou



Kruskalův algoritmus – poznámky

Myšlenka. Seřadím hrany podle jejich ohodnocení. V tomto pořadí je budu uvažovat pro zařazení do kostry.

Pozorování. Algoritmus musí udržovat informaci o tom, v jaké komponentě se který vrchol nachází. Nelze totiž vybrat do kostry hranu, která spojuje vrcholy v rámci stejné komponenty.

Implemetace využívá tří pomocných funkcí:

- MakeSet(u) vytvoří jednoprvkovou množinu obsahující vrchol u
- FindSet(u) vrátí identifikátor množiny obsahující vrchol u
- Union(u, v) sloučí množiny obsahující vrcholy u a v

Kruskalův algoritmus – pseudokód

```
 1: function Kruskal(G = (V, E, w_e)) is
 2:     mst ← ∅
 3:     for all v ∈ V do
 4:         MakeSet(v)
 5:     done
 6:     for all {u, v} ∈ E od nejmenší podle w_e do
 7:         if FindSet(u) ≠ FindSet(v) then
 8:             mst ← mst ∪ {{u, v}}
 9:             Union(u, v)
10:         fi
11:     done
12:     Vrať mst
13: end
```

Jednoduchá struktura Union-Find

Myšlenka. Každá množina bude reprezentována spojovým seznamem.

- MakeSet(u) vytvoří jednoprvkový spojový seznam obsahující vrchol u
- FindSet(u) vrátí první prvek seznamu obsahující vrchol u
- Union(u, v) sloučí dva seznamy obsahující vrcholy u a v

Pozorování. Operace FindSet má lineární složitost, ale je v rámci Kruskalova algoritmu volána nejčastěji.

Vylepšení. U každého prvku seznamu přidáme navíc ukazatel na hlavu seznamu. FindSet bude nyní konstantní, Union bude muset přepisovat všechny tyto ukazatele.

Optimální verze Union-Find

Myšlenka. Každou množinu reprezentujeme jako strom. Složitosti operací budou závislé na jejich výšce.

- MakeSet(u) vytvoří jednoprvkový strom s kořenem u
- FindSet(u) vrátí kořen stromu obsahující vrchol u
- Union(u, v) sloučí dva stromy obsahující vrcholy u a v

Vylepšení snižující složitost operací

- Union napojuje vždy strom s nižší výškou pod kořen druhého
- FindSet navíc snižuje výšku prohledávaného stromu napojením vrcholů přímo pod kořen

Složitost Kruskalova algoritmu při využití této struktury je určena nutností seřazení hran podle jejich ohodnocení, tedy $O(|E| \log |E|) = O(|E| \log |V|)$.

Zajímavé odkazy

- O jistém problému minimálním: https://dml.cz/bitstream/handle/10338.dmlcz/500114/Boruvka_01-0000-6_1.pdf
- Příspěvek k otázce ekonomické stavby elektrovodných sítí : https://dml.cz/bitstream/handle/10338.dmlcz/500188/Boruvka_02-0000-1_1.pdf
- Vizualizace algoritmů pro výpočet minimální kostry: https://visualgo.net/en/mst
- Vizualizace operací nad strukturou Union-Find: https://www.cs.usfca.edu/~galles/visualization/DisjointSets.html

10. Přístupy k řešení problémů I.

Hrubá síla

Hrubá síla (brute force) představuje přímočarý přístup založený na definici problému.

Vlastnosti

- obecně výpočetně nejnáročnější postup → použitelné pouze pro velmi malé instance problémů
- v případě optimalizačních úloh se jedná o zkoušení všech potenciálních řešení
- pro některé typu problémů jediná možnost

Příklady

- Selection sort
- naivní násobení matic
- jednoduché hledání podřetězce v textu

Rekurzivní algoritmy

Myšlenka rekurzivních algoritmů spočívá v znovupoužití algoritmu na problém menší velikosti. Speciální (triviální) případy řešíme přímo.

```
def fact_recursive(n):
    return n * fact_recursive(n - 1) if n > 0 else 1


def fact_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i

    return result
```

Poznámka. Volání funkce je ve srovnání s iterací cyklu drahá operace. Vysoká úroveň rekurzivního zanoření navíc může narazit na limity velikosti zásobníku.

Hanoiské věže

Hanoiské věže jsou hlavolam, v rámci kterého je potřeba přemístit všechny disky z jednoho kolíku na jiný za dodržení dalších pravidel.

Poznámka. Existuje velmi jednoduché rekurzivní řešení.

Složitost řešení problému Hanoiských věží je exponenciální vůči počtu disků (přesně $2^n - 1$ operací).

Rozděl a panuj

Rozděl a panuj (divide and conquer, D & C) je přístup vycházející z rekurzivních algoritmů založený na dělení problému na menší, snadněji zvládnutelné, podproblémy.

Princip

1. Rozděl problém na menší podproblémy
   - stejného typu
   - nepřekrývající se
2. Rekurzivně vyřeš každý podproblém
3. Zkombinuj řešení podproblémů v řešení původního problému

Příklady

- Mergesort
- Quicksort

Master theorem

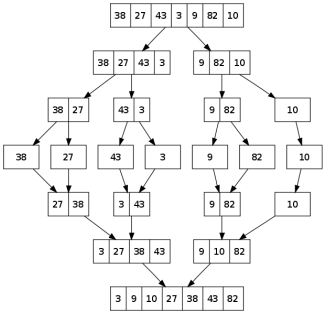
Master theorem je nástroj pro určování složitosti algoritmů založených na přístupu rozděl a panuj. Pokud je velikost podproblémů shodná, lze využít následujícího vztahu:

$$T(n) = aT(n/b) + O(n^d)$$

- $T(n)$ – počet kroků nutných pro vyřešení problému o velikosti n
- a – počet podproblémů
- b – faktor určující velikost podproblémů
- n^d – rozdělení na podproblémy | sloučení řešení jednotlivých podproblémů

$$T(n) = \begin{cases} O(n^d) & \text{pro } d > \log_b a \\ O(n^d \log n) & \text{pro } d = \log_b a \\ O(n^{\log_b a}) & \text{pro } d < \log_b a \end{cases}$$

Master theorem – Mergesort



Složitost Mergesortu. Každé pole je rozděleno na dvě části o poloviční velikosti ($a = b = 2$), slévání seřazených podposloupností je lineární operace ($d = 1$). Platí $d = \log_b a \rightarrow T(n) = O(n^d \log n)$.

$$T(n) = 2T(n/2) + O(n) \rightarrow T(n) = O(n \log n)$$

Násobení matic

Úkol. Popište algoritmus pro násobení dvou matic ($Z = XY$) o rozměrech $n \times n$.

Naivní násobení matic představuje učebnicový způsob řešení se složitostí $O(n^3)$.

$$\forall (i, j) \in \{1..n\} \times \{1..n\} : Z_{i,j} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}$$

Blokové násobení matic je přístup, kdy každou z matic rozdělíme na menší a násobíme dle následujícího schématu:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \qquad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

$$Z = XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Násobení matic

Složitost blokového násobení matic je však zřejmě stále shodná s naivním přístupem, tedy $O(n^3)$.

Poznámka. V reálném běhu je dekompozice na bloky výrazně rychlejší řešení díky lepšímu využití vyrovnávací paměti.

Rekurzivní algoritmus násobení matic

$$Z = XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

- aplikujme stejný (D & C) přístup pro výpočet součinů $AE, BG, \ldots$
- aplikací MT získáváme

$$T(n) = 8T(n/2) + O(n^2) \rightarrow T(n) = O(n^3)$$

Strassenův algoritmus

Myšlenka. Využijeme rekurzivní algoritmus pro násobení matic, avšak pomocí algebraických transformací snížíme počet nutných násobení. Počet sčítání není příliš významný.

$$
\begin{aligned}
P_1 &= A(F-H) & P_5 &= (A+D)(E+H)\\
P_2 &= (A+B)H & P_6 &= (B-D)(G+H)\\
P_3 &= (C+D)E & P_7 &= (A-C)(E+F)\\
P_4 &= D(G-E)
\end{aligned}
$$

$$
Z = XY = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}
$$

Složitost Strassenova algoritmu

$$
T(n) = 7T(n/2) + O(n^2) \rightarrow T(n) = O(n^{\log_2 7}) \approx O(n^{2,81})
$$

Hladové algoritmy

Hladový algoritmus (greedy algorithm) v každém kroce výpočtu výbírá aktuálně nejlepší možnost, přičemž spoléhá, že tato posloupnost voleb vede ke globálně nejlepšímu řešení.

Vlastnosti

- ne vždy lze s úspěchem použít – pouze některé problémy mají tuto strukturu
- časově efektivní

Příklady

- Kruskalův a Primův algoritmus
- Dijkstrův algoritmus

Minimální počet mincí

Problém. Vyplaťte zadanou částku pomocí minimálního počtu mincí.

Příklad. Částka 79, hodnoty mincí 1, 2, 5, 10,...

- hladový přístup – volím vždy minci nejvyšší hodnoty menší než zbývající částka
- 79 = 50 + 20 + 5 + 2 + 2

Příklad. Částka 6, hodnoty mincí 1, 3 a 4.

- hladový přístup – 6 = 4 + 1 + 1
- optimální řešení – 6 = 3 + 3

Příklad. Částka 14, hodnoty mincí 3, 7 a 10.

- optimální řešení – 14 = 7 + 7
- hladový přístup – 14 = 10 + ?

Zajímavé odkazy

- Hanoiské věže: https://www.mathsisfun.com/games/towerofhanoi.html

11. Přístupy k řešení problémů II.

Sudoku

Úkol. Vyřešte následující zadání Sudoku.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Zamyšlení. Řešení je poměrně snadné pro člověka, ale jak jej algoritmizovat?

Backtracking

Backtracking je rekurzivní přístup, v rámci něhož hledám řešení následujícím způsobem:

1. Zkontroluji, zdali jsem nalezl řešení.
2. Pokud ne, zkusím pokračovat v hledání některou z možností, kterou v danou chvíli mám a ještě jsem nevyzkoušel.
3. Pokud žádné možnosti nezbývají, vracím se do posledního místa, kde jsem měl ještě na výběr.

Vlastnosti:

- garance nalezení nejlepšího/všech řešení
- potenciálně vysoká složitost

Známé příklady:

- DFS
- minimální počet mincí

Sudoku – jednoduché řešení

1. Pokud jsou obsazena všechna pole, vracím TRUE.
2. Najdu první prázdné pole.
3. Pro všechny přípustné číslice pro toto pole:
   3.1 Zapíšu zvolenou číslici.
   3.2 Celý algoritmus rekurzivně opakuji.
   3.3 Pokud je výsledkem rekurzivního volání TRUE, vracím jej, v opačném případě zkouším další číslici.
4. Všechny možnosti pro dané pole jsou neúspěšně vyčerpány, vracím FALSE.

Zamyšlení. Uvedený postup lze vylepšit, pokud budou volná pole vybírána v pořadí podle počtu možných číslic (od nejmenšího).

Branch and bound

Branch and bound je jednoduché vylepšení backtrackingu pro optimalizační problémy, kdy si během výpočtu pamatuju aktuálně nejlepší nalezené řešení (resp. jeho cenu).

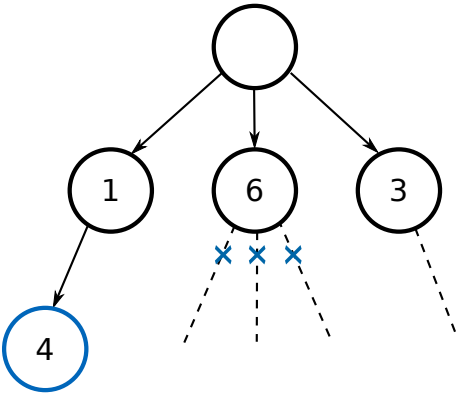
- eliminuji cesty, které už nemohou vést k lepšímu řešení (= jejich ohodnocení je větší než ohodnocení aktuálně nejlepšího nalezeného řešení)

Vlastnosti:

- vede k omezení větvení → „prořezávání větví“
- nezaručuje, že se vyhneme exponenciální složitosti
- užitečné, pokud brzy najdeme dobré řešení

Branch and bound – příklad

Ukázka. Po nalezení řešení s ohodnocením 4 neprohledávám dále neperspektivní cesty.



Dynamické programování

Dynamické programování je metoda podobná rozděj a panuj použitelná pro optimalizační úlohy.

Princip

1. Rozděl problém na menší podproblémy.
   - stejného typu
   - musí se překrývat (optimální řešení problému v sobě zahrnuje optimální řešení podproblému)
2. Vyřeš jednotlivé podproblémy v pořadí od nejmenších.
3. Zkombinuj řešení podproblémů na řešení původního problému.

Příklady

- Dijkstrův algoritmus
- Floyd-Warshallův algoritmus

Minimální počet mincí

Opakování. Pro správně zvolené hodnoty mincí je hladový přístup optimální.
V některých případech však nenalezne (nejlepší) řešení.

Optimální řešení lze nalézt pomocí dynamického programování.

- označme $C[j]$ minimální počet mincí na zaplacení částky j
- pokud známe optimální řešení pro $C[j]$ a použili jsme minci hodnoty h_i, pak máme:

$$C[j] = 1 + C[j - h_i]$$

Příklad. Pokud je $C[46]$ optimální a použili jsme minci hodnoty 20, pak $C[46] = 1 + C[26]$.

Minimální počet mincí

Zobecnění. Mějme k různých mincí hodnot h_i, kde $1 \leq i \leq k$. Pak optimální řešení pro částku j je dáno:

$$C[j] = \begin{cases} \infty & \text{pro } j < 0 \\ 0 & \text{pro } j = 0 \\ 1 + \min\limits_{1 \leq i \leq k} \{C[j - h_i]\} & \text{pro } j \geq 1 \end{cases}$$

Příklad pro částku 6 a hodnoty mincí 1, 3 a 4.

- postupujeme od nejnižších částek až po výslednou dle předchozího výrazu
- zjevně $C[0] = 0$

Minimální počet mincí

$$C[1] = \min \begin{cases} 1 + C[1-4] = \infty \\ 1 + C[1-3] = \infty \\ 1 + C[1-1] = 1 \end{cases}$$

$$C[2] = \min \begin{cases} 1 + C[2-4] = \infty \\ 1 + C[2-3] = \infty \\ 1 + C[2-1] = 2 \end{cases}$$

$$C[3] = \min \begin{cases} 1 + C[3-4] = \infty \\ 1 + C[3-3] = 1 \\ 1 + C[3-1] = 3 \end{cases}$$

$$C[4] = \min \begin{cases} 1 + C[4-4] = 1 \\ 1 + C[4-3] = 2 \\ 1 + C[4-1] = 2 \end{cases}$$

$$C[5] = \min \begin{cases} 1 + C[5-4] = 2 \\ 1 + C[5-3] = 3 \\ 1 + C[5-1] = 2 \end{cases}$$

$$C[6] = \min \begin{cases} 1 + C[6-4] = 3 \\ 1 + C[6-3] = 2 \\ 1 + C[6-1] = 3 \end{cases}$$

Optimální pořadí násobení matic

Problém. Chceme vynásobit $A_1 \cdots A_n$ s nejmenším počtem operací.

Pozorování

- násobení matic je asociativní, tj. $A(BC) = (AB)C$
- vhodným uzávorkováním lze snížit množství nutných operací

Příklad s maticemi $A_{10\times30}$, $B_{30\times5}$, $C_{5\times60}$:

- $(A_{10\times30} \cdot B_{30\times5}) \cdot C_{5\times60} = X_{10\times5} \cdot C_{5\times60}$
- $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 4\,500$ operací
- $A_{10\times30} \cdot (B_{30\times5} \cdot C_{5\times60}) = A_{10\times30} \cdot X_{30\times60}$
- $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 27\,000$ operací

Závěr. Složitost řešení pomocí přístupu rozděl a panuj je $O(3^n)$, užitím dynamického programování pak $O(n^3)$.

Heuristiky

Pozorování. Některé instance problémů mohou být z hlediska složitosti exaktně velmi těžko řešitelné.

Myšlenka

- suboptimální řešení lze nalézt často výrazně rychleji
- často není potřeba určit všechna řešení
- heuristika – forma odhadu jak vypadá/co obsahuje řešení (př. v 95 % případů platí...)

Příklad. Určete nejkratší vzdálenost mezi vrcholy s a t v grafu G.

- pro výpočet uvažujeme pouze hrany s ohodnocením $\leq k$
- zřejmě nemusí vést k (nejlepšímu) řešení

Redukce

Redukce je metoda převodu jednoho problému na jiný. Využívá se zejména v rámci teoretického porovnávání složitosti algoritmů.

Princip

1. Zadání problému A transformuji na zadání pro problém B.
2. Vyřeším zadání problému B.
3. Řešení problému B převedu zpátky na řešení původního problému A.

Příklad. Nejkratší vzdálenost v neohodnoceném grafu.

- lze převést na nejkratší vzdálenost v ohodnoceném grafu
- $\forall (u, v) \in E : w_e(u, v) = 1$
- nejkratší cesta je v novém i původním grafu stejná

Zajímavé odkazy

- Řesitel sudoku: https://www.sudoku-solutions.com/
- Backtracking a průchod bludištěm: https://www.youtube.com/watch?v=h0aXgiL-lws

12. Těžké problémy.

Typy problémů

V rámci teoretické analýzy nejčastěji rozlišujeme dva typy problémů:

Rozhodovací problém

- ověření, zdali něco platí, nebo ne
- př. Existuje v grafu G cesta mezi vrcholy s a t délky nejvýše 10?
- odpověď: ANO $\times$ NE

Optimalizační problém

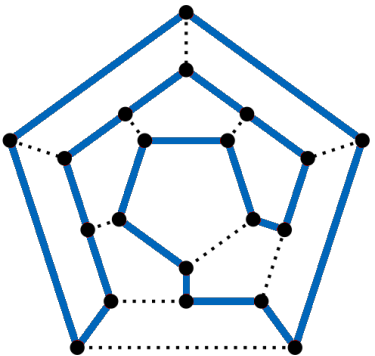
- cílem je nalezení nejlepšího řešení z množiny přípustných řešení
- př. Jaká je nejkratší cesta v grafu G mezi vrcholy s a t?
- odpověd: konkrétní nejkratší cesta $\times$ cesta neexistuje

Poznámka. Pokud existuje polynomiální algoritmus pro rozhodovací problém, existuje i pro jeho optimalizační variantu (a naopak).

Problém obchodního cestujícího (TSP)

Problém. Nalezněte nejkratší cestu, která prochází všemi zadanými městy a začíná a končí ve stejném městě.

Alternativní definice. Nalezněte v hranově ohodnoceném grafu Hamiltonovskou kružnici (= obsahující všechny vrcholy) minimální délky.



TSP – možnosti řešení

Hrubá síla. Vygeneruji a ověřím délky všech možných cest.

- složitost přístupu $O(n!)$
- v praxi nepoužitelné

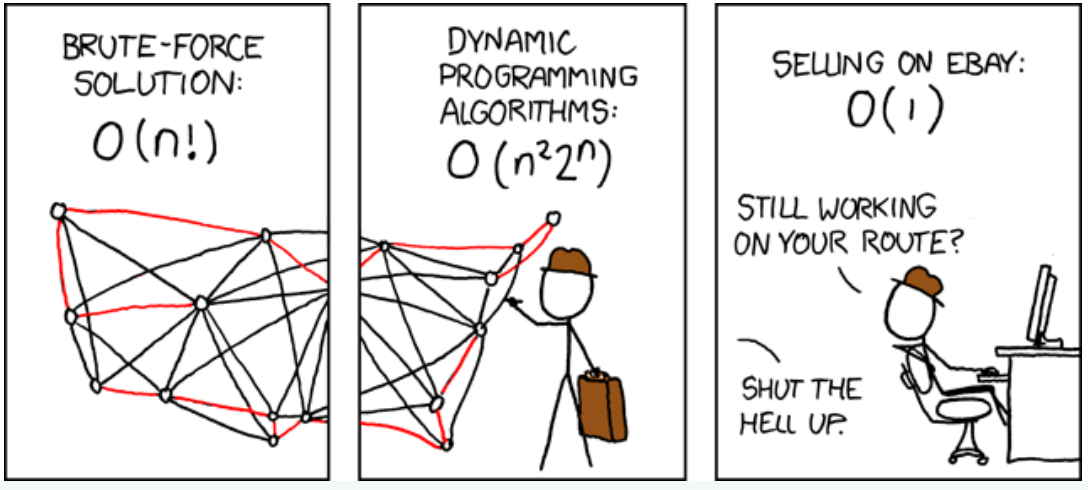
Dynamické programování

- výrazně netriviální
- složitost $O(n^2 2^n)$

Aktuální stav řešení TSP.

- nevíme, zdali existuje algoritmus se složitostí nižší než $O(2^n)$
- v roce 2006 se podařilo najít řešení pro instanci problému o velikosti 85 900 měst → 136 CPU let výpočtů

xkcd.com/399


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.

Třídy problémů

Pozorování. Velká část dosud prezentovaných problémů byla bez větších problémů prakticky řešitelná. Opakem je například TSP.

V rámci teorie pak můžeme přemýšlet, zdali lze problémy dělit do kategorií podle složitosti jejich řešení.

Nejčastěji rozlišujeme dvě třídy problémů:

- P třída problémů řešitelných v polynomiálním čase
- NP třída problémů, pro které lze ověřit řešení v polynomiálním čase

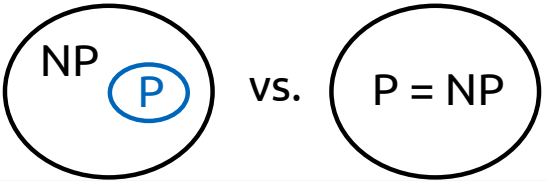
Poznámka. V rámci zařazování problémů do těchto tříd vždy uvažujeme jejich rozhodovací varianty.

Příklad. TSP je ve třídě NP, nejkratší vzdálenost v grafu je v P i v NP.

P vs. NP

Zamyšlení. Zjevně platí, že každý problém ve třídě P patří i do třídy NP, tedy $\mathrm{P} \subseteq \mathrm{NP}$.

Otázka. Platí to ale i naopak ($\mathrm{NP} \subseteq \mathrm{P}$)? Pokud ano, pak $\mathrm{P} = \mathrm{NP}$.



P vs. NP

- otevřený problém, jeden z největších v matematice a informatice
- jeden ze sedmi problémů milénia (Millennium Prize Problem → odměna 1 milion dolarů)

P = NP

If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps,' no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss...

Scott Aaronson, MIT

NP-úplné problémy

Pozorování. I v rámci třídy NP jsou problémy, které jsou různě těžké.

NP-úplné problémy jsou nejtěžší problémy ve třídě NP.

- každý problém v NP lze převést na NP-úplný problém v polynomiálním čase (existuje polynomiální redukce)
- rozhodovací varianta TSP je NP-úplný problém
- pro žádný NP-úplný problém není znám polynomiální algoritmus

Možnosti řešení P vs. NP

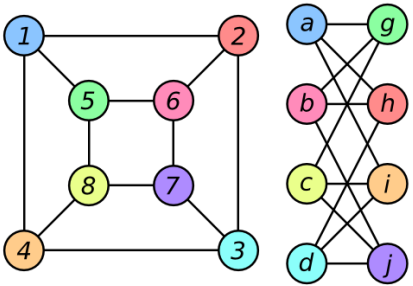
1. Ukázat, že pro některý NP-úplný problém nelze zkonstruovat polynomiální algoritmus. Pak $\mathrm{P} \neq \mathrm{NP}$.
2. Nalézt polynomiální algoritmus pro libovolný NP-úplný problém. Pak $\mathrm{P} = \mathrm{NP}$.

Problémy v NP – příklady

Zamyšlení. Předpokládejme $\mathrm{P} \neq \mathrm{NP}$. Existují problémy, které jsou v NP, ale nejsou NP-úplné?

Pravděpodobně následující:

- prvočíselný rozklad
- izomorfismus grafů



Prvočíselný rozklad

Úkol. Rozložte následující číslo na prvočísla:

135066410865995223349603216278805969938881475605
667027524485143851526510604859533833940287150571
909441798207282164471551373680419703964191743046
496589274256239341020864383202110372958725762358
509643110564073501508187510676594629205563685529
475213500852879416377328533906109750544334999811
150056977236890927563

- ekvivalentní rozluštění RSA-1024
- odměna 100 000 dolarů
- soutěž skončila v roce 2007

Travelling salesman (2012)

Travelling Salesman

Drama / Mysteriózní / Thriller / Sci-Fi
USA, 2012, 80 min

Hrají: Steve West

Obsah

Čtveřice geniálních matematiků objeví v průběhu úspěšného výzkumu problému P versus NP algoritmus rapidně zrychlující výpočetní operace. Jejich objev může mít obrovské důsledky. Jak pozitivní, v podobě mohutné akcelerace biologického a medicínského vývoje, tak negativní, neboť nový algoritmus mj. umožňuje překonat moderní šifrování během několika vteřin. Poté, co vláda Spojených států nabídne každému z nich 10 milion dolarů za exkluzivní přístup k jejich části algoritmu, musí se čtveřice vypořádat s morálními i praktickými problémy, které jejich rozhodnutí přináší. *(Slaboproud)*

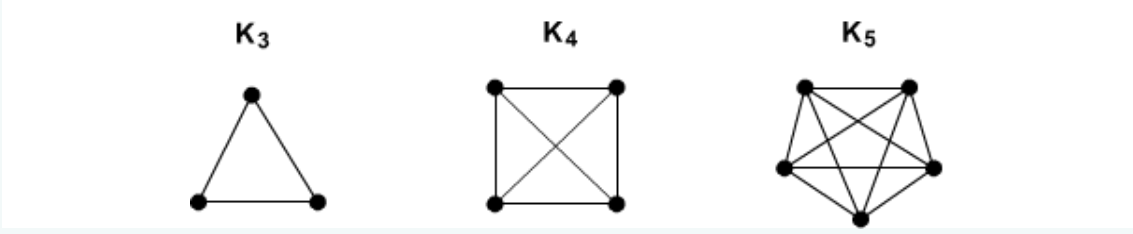
Vybrané příklady NP-úplných problémů I

Problém splnitelnosti výrokových formulí

- formule výrokové logiky s proměnnými $A_1, \ldots, A_n$
- Existuje přiřazení proměnných takové, že se zadaná formule vyhodnotí na TRUE?
- Příklad: $(\neg A_1 \vee A_2) \wedge A_3 \wedge \neg A_1$ je splnitelná např. pro $A_1 = 0$, $A_2 = 0$, $A_3 = 1$,

Klika

- Existuje v grafu klika (= podgraf, který je úplným grafem) o k vrcholech?



Vybrané příklady NP-úplných problémů II

Problém dvou loupežníků

- Lze rozdělit multimnožinu nezáporných čísel na dvě tak, že v obou bude součet obsažených čísel stejný?

Izomorfismus podgrafu

- Je graf H izomorfní nějakému podgrafu grafu G?

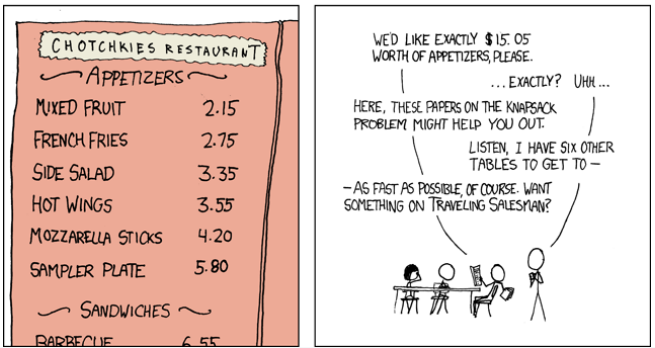
Problém batohu

- mějme batoh o nosnosti W a n předmětů, každý o hmotnosti w_i a hodnotě v_i
- Lze do batohu umístit předměty o celkové hodnotě alespoň V?

Součet podmnožiny

- Lze najít podmnožinu zadané množiny celých čísel takovou, že součet jejích prvků je nula?

xkcd.com/287



General solutions get you a 50% tip.

P vs. NP – poznámky

Možné výsledky

- $\mathrm{P} = \mathrm{NP}$, ale nejlepší algoritmus pro TSP se složitostí $\Omega(n^{100})$
- $\mathrm{P} \neq \mathrm{NP}$, ale algoritmus pro TSP se složitostí $O(2^{0{,}00\ldots01 \cdot n})$

Možnosti řešení

1. přijmout exponenciální algoritmus
2. omezit se na speciální případy (př. izomorfismus stromů je v P)
3. přijmout suboptimální řešení (užitím hladových algoritmů, heuristik)

Zajímavé odkazy

- Netypická varianta TSP: https://www.math.uwaterloo.ca/tsp/pubs/index.html
- Přehled důkazů P vs. NP:
  https://www.win.tue.nl/~wscor/woeginger/P-versus-NP.htm
- Redukce mezi NP-úplnými problémy:
  https://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf