Složitost

Složitost algoritmů

❖ Základní teoretický přístup vychází z Church-Turingovy teze:

Každý algoritmus je implementovatelný jistým TS.

❖ Zavedení TS nám umožňuje klasifikovat problémy (resp. funkce) do dvou tříd:

1. problémy, jež nejsou algoritmicky ani částečně rozhodnutelné (resp. funkce algoritmicky nevyčíslitelné) a
2. problémy algoritmicky alespoň částečně rozhodnutelné (resp. funkce algoritmicky vyčíslitelné).

❖ Nyní se budeme zabývat třídou algoritmicky (částečně) rozhodnutelných problémů (vyčíslitelných funkcí) v souvislosti s otázkou složitosti jejich rozhodování (vyčíslování).

❖ Analýzu složitosti algoritmu budeme chápat jako analýzu složitosti výpočtů příslušného TS, jejímž cílem je vyjádřit (kvantifikovat) požadované zdroje (čas, prostor) jako funkci závisející na délce vstupního řetězce.

Různé případy při analýze složitosti

❖ Můžeme rozlišit:

1. analýzu složitosti nejhoršího případu,
2. analýzu složitosti nejlepšího případu,
3. analýzu složitosti průměrného případu,
4. amortizovanou analýzu

❖ Průměrná složitost algoritmu je definována následovně:

- Jestliže algoritmus (TS) vede k m různým výpočtům (případům) se složitostí c_1, c_2, ..., c_m, jež nastávají s pravděpodobností p_1, p_2, ..., p_m, pak průměrná složitost algoritmu je dána jako $\Sigma_{i=1}^{n} p_i c_i$.

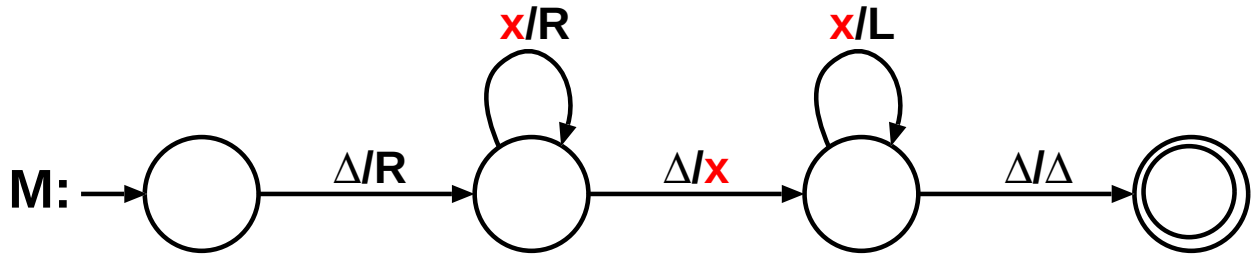
❖ Amortizovaná analýza studuje posloupnost operací jako celek. Tato technika umožňuje, na rozdíl od klasického přístupu mnohem přesnější určení časové složitosti algoritmu.

Složitost výpočtů TS pro daný vstup

❖ Časová složitost – počet kroků (přechodů) TS provedený od počátku do konce výpočtu.

❖ Prostorová (paměťová) složitost – počet „buněk“ pásky TS požadovaný pro daný výpočet.

Příklad 12.1 Uvažme následující TS M:



Pro vstup $\Delta xxx\Delta\Delta$... je:

- časová složitost výpočtu M rovna 10,
- prostorová složitost výpočtu M rovna 5.

Lemma 12.1 Je-li časová složitost výpočtu prováděného TS rovna n, pak prostorová složitost tohoto výpočtu není větší než $n + 1$.

Klíčová definice: Složitost výpočtů TS

Definice 12.1 Řekneme, že k-páskový DTS (resp. NTS) M přijímá jazyk L nad abecedou Σ v čase $T_M : \mathbb{N} \to \mathbb{N}$, jestliže $L = L(M)$ a M přijme (resp. *může* přijmout) každé $w \in L$ v nanejvýš $T_M(|w|)$ krocích.

Definice 12.2 Řekneme, že k-páskový DTS (resp. NTS) M přijímá jazyk L nad abecedou Σ v prostoru $S_M : \mathbb{N} \to \mathbb{N}$, jestliže $L = L(M)$ a M přijme (resp. *může* přijmout) každé $w \in L$ při použití nanejvýš $S_M(|w|)$ buněk pásky – *nepočítáme zde buňky pásky, na nichž je zapsán vstup, ale nikdy na nich nespočine hlava stroje*.

❖ Zcela analogicky můžeme definovat vyčíslování určité funkce daným TS v určitém čase, resp. prostoru.

Analýza složitosti mimo prostředí TS

❖ V jiném výpočetním prostředí, než jsou TS, nemusí mít každá primitivní operace stejnou cenu.

❖ Velmi často se užívá tzv. uniformní cenové kritérium, kdy každé operaci přiřadíme stejnou cenu.

❖ Používá se ale např. také tzv. logaritmické cenové kritérium, kdy operaci manipulující operand o velikosti i, $i > 0$, přiřadíme cenu $\lfloor lg\ i \rfloor + 1$:

- Zohledňujeme to, že s rostoucí velikostí operandů roste cena operací – logaritmus odráží růst velikosti s ohledem na binární kódování (v n bitech zakódujeme 2^n hodnot).

❖ Analýza složitosti za takových předpokladů není zcela přesná, důležité ale obvykle je to, aby se zachovala informace o tom, jak rychle roste čas/prostor potřebný k výpočtu v závislosti na velikosti vstupu.[a]

[a] Algoritmus A_1, jehož nároky rostou pomaleji než u jiného algoritmu A_2, nemusí být výhodnější než A_2 pro řešení malých instancí.

❖ Význam srovnávání rychlosti růstu složitosti výpočtů si snad nejlépe ilustrujeme na příkladu:

Příklad 12.2 Srovnání polynomiální (přesněji kvadratické – n^2) a exponenciální (2^n) časové složitosti:

délka vstupu n	časová složitost n^2	časová složitost 2^n
10	0.0001 s	0.0001 s
20	0.0004 s	0.1024 s
30	0.0009 s	1.75 min
40	0.0016 s	1.24 dne
50	0.0025 s	3.48 roku
60	0.0036 s	35.68 století
70	0.0049 s	3.65 mil. roků

Složitost výpočtů na TS a v jiných prostředích

❖ Při použití vhodného cenového kritéria, je složitost pro výpočetní modely blízké běžným počítačům (RAM, RASP stroje) polynomiálně vázaná se složitostí výpočtu na TS. Tudíž složitost výpočtu na TS není „příliš“ rozdílná oproti výpočtům na běžných počítačích.

- RAM stroje mají paměť s náhodným přístupem (jedna buňka obsahuje libovolné přirozené číslo), instrukce typu LOAD, STORE, ADD, SUB, MULT, DIV, vstup/výstup, skok atd. U RAM stroje je program součástí řízení stroje (okamžitě dostupný), u RASP je uložen v paměti stejně jako operandy.
- Funkce $f_1(n), f_2(n) : \mathbb{N} \to \mathbb{N}$ jsou polynomiálně vázané, existují-li polynomy $p_1(x)$ a $p_2(x)$ takové, že pro všechny hodnoty n je $f_1(n) \leq p_1(f_2(n))$ a $f_2(n) \leq p_2(f_1(n))$.
- Logaritmické cenové kritérium se uplatní, uvažujeme-li možnost násobení dvou čísel. Jednoduchým cyklem typu $A_{i+1} = A_i * A_i$ ($A_0 = 2$) jsme schopni počítat 2^{2^n}, což TS s omezenou abecedou není schopen provést v polynomiálním čase (pro uložení/načtení potřebuje projít 2^n buněk při použití binárního kódování).

❖ Ilustrujme si nyní určení složitosti v prostředí mimo TS:

Příklad 12.3 Uvažme následující implementaci porovnávání řetězců.

```
int str_cmp (int n, string a, string b) {
    int i;

    i = 0;
    while (i<n) {
      if (a[i] != b[i]) break;
      i++;
    }

    return (i==n);
}
```

- Pro určení složitosti aplikujme uniformní cenové kritérium např. tak, že budeme považovat složitost každého řádku programu (mimo deklarace) za jednotku. (Předpokládáme, že žádný cyklus není zapsán na jediném řádku.)
- Složitost nejhoršího případu

❖ Ilustrujme si nyní určení složitosti v prostředí mimo TS:

Příklad 12.4 Uvažme následující implementaci porovnávání řetězců.

```
int str_cmp (int n, string a, string b) {
    int i;

    i = 0;
    while (i<n) {
      if (a[i] != b[i]) break;
      i++;
    }

    return (i==n);
}
```

- Pro určení složitosti aplikujme uniformní cenové kritérium např. tak, že budeme považovat složitost každého řádku programu (mimo deklarace) za jednotku. (Předpokládáme, že žádný cyklus není zapsán na jediném řádku.)
- Složitost nejhoršího případu lze pak snadno určit jako $3n + 3$:
  - cyklus má 3 kroky, provede se n krát, tj. $3n$ kroků,
  - tělo funkce má 3 kroky (včetně testu ukončujícího cyklus).

Příklad 12.5 Uvažme následující implementaci řazení metodou insert-sort.

```
void insertsort(int n, int a[]) {
    int i, j, value;
    for (i=0; i<n; i++) {
        value = a[i];
        j = i - 1;
        while ((j >= 0) && (a[j] > value)) {
                a[j+1] = a[j];
                j = j - 1;
        }
        a[j+1] = value;
    }
}
```

- Pro určení složitosti aplikujme uniformní cenové kritérium např. tak, že budeme považovat složitost každého řádku programu (mimo deklarace) za jednotku. (Předpokládáme, že žádný cyklus není zapsán na jediném řádku.)
- Složitost nejhoršího případu

Příklad 12.6 Uvažme následující implementaci řazení metodou insert-sort.

```
void insertsort(int n, int a[]) {
    int i, j, value;
    for (i=0; i<n; i++) {
        value = a[i];
        j = i - 1;
        while ((j >= 0) && (a[j] > value)) {
                a[j+1] = a[j];
                j = j - 1;
        }
        a[j+1] = value;
    }
}
```

- Pro určení složitosti aplikujme uniformní cenové kritérium např. tak, že budeme považovat složitost každého řádku programu (mimo deklarace) za jednotku. (Předpokládáme, že žádný cyklus není zapsán na jediném řádku.)
- Složitost lze pak určit jako $1.5n^2 + 3.5n + 1$:
  - vnitřní cyklus má 3 kroky, provede se 0, 1, ..., $n-1$ krát, tj. $3(0 + 1 + ... + n - 1) = 3\frac{n}{2}(0 + n - 1) = 1.5n^2 - 1.5n$ kroků,
  - vnější cyklus má mimo vnitřní cyklus 5 kroků (včetně testu ukončujícího vnitřní cyklus) a provede se n krát, tj. $5n$ kroků,
  - jeden krok připadá na ukončení vnějšího cyklu.

Asymptotická omezení složitosti

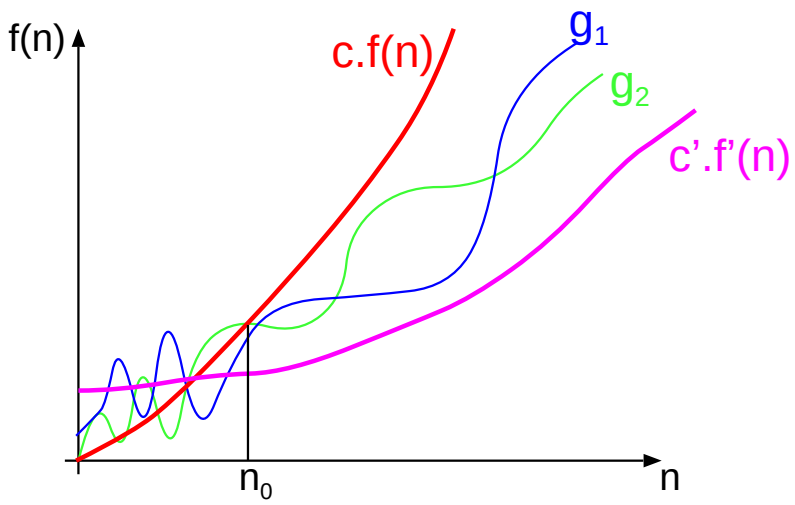
❖ Při popisu složitosti algoritmů (výpočtů TS), chceme často vyloučit vliv aditivních a multiplikativních konstant:

- různé aditivní a multiplikativní konstanty vzniknou velmi snadno „drobnými“ úpravami uvažovaných algoritmů,
- např. srovnávání dvou řetězců je možné donekonečna zrychlovat tak, že budeme porovnávat současně 2, 3, 4, ... za sebou jdoucích znaků,
- lineární komprese prostoru (rozšíření abecedy) a zrychlení výpočtu (před-vypočítání výsledků)
- tyto úpravy ovšem nemají zásadní vliv na rychlost nárůstu časové složitosti (ve výše uvedeném případě nárůst zůstane kvadratický),
- navíc při analýze složitosti mimo prostředí TS se dopouštíme jisté nepřesnosti již zavedením různých cenových kritérií.

❖ Proto se často složitost popisuje pomocí tzv. asymptotických odhadů složitosti

Definice 12.3 Nechť $\mathcal{F}$ je množina funkcí $f : \mathbb{N} \to \mathbb{N}$. Pro danou funkci $f \in \mathcal{F}$ definujeme množiny funkcí $O(f(n))$, $\Omega(f(n))$ a $\Theta(f(n))$ takto:

- Asymptotické horní omezení funkce $f(n)$ je množina
  $O(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq g(n) \leq c.f(n)\}$.
- Asymptotické dolní omezení funkce $f(n)$ je množina
  $\Omega(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c.f(n) \leq g(n)\}$.
- Asymptotické oboustranné omezení funkce $f(n)$ je množina $\Theta(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c_1.f(n) \leq g(n) \leq c_2.f(n)\}$.



Příklad 12.7 S využitím asymptotických odhadů složitosti můžeme říci, že složitost našeho srovnání řetězců patří do $O(n)$ a složitost insert-sort do $O(n^2)$.

Definice 12.4 Nechť $\mathcal{F}$ je množina funkcí $f : \mathbb{N} \to \mathbb{N}$. Pro danou funkci $f \in \mathcal{F}$ definujeme množiny funkcí $O(f(n))$, $\Omega(f(n))$ a $\Theta(f(n))$ takto:

- Asymptotické horní omezení funkce $f(n)$ je množina
  $O(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq g(n) \leq c.f(n)\}$.
- Asymptotické dolní omezení funkce $f(n)$ je množina
  $\Omega(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c.f(n) \leq g(n)\}$.
- Asymptotické oboustranné omezení funkce $f(n)$ je množina $\Theta(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow 0 \leq c_1.f(n) \leq g(n) \leq c_2.f(n)\}$.

❖ Asymptotické horní omezení exponenciálních funkcí

$$2^{O(f(n))} = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq 2^{c.f(n)}\}$$

- relaxuje aditivní a multiplikativní konstanty v exponentu
- dovoluje dále zjednodušit analýzu složitosti

Amortizovaná složitost

❖ Technika dovolující přesnější analýzu složitosti v nejhorším případě pro danou posloupnost operací.

❖ Klasický přístup analyzuje složitost jednotlivých operací. Výsledná složitost je součtem jednotlivých operací.

❖ Technika amortizace analyzuje posloupnost jako celek. Stojí zejména na pozorování, že drahé operace mohou nastat pouze jednou za delší dobu a tudíž se jejich cena amortizuje.

❖ Existují různé metody dovolující analyzovat amortizovanou složitost: **seskupování**, **metoda účtů**, potenciálové funkce

Amortizovaná složitost – příklad

❖ Uvažme zásobník S a operace PUSH(S, x), POP(S) a MULTIPOP(S, k) – odebere k prvků případně vyprázdní zásobník pokud $|S| \leq k$).

❖ Uvažujme libovolnou posloupnost n operací.

❖ Každá operace PUSH(S, x) a POP(S) má složitost 1. V posloupnosti n operací má MULTIPOP(S, k) v nejhorším případě složitost $O(n)$.

❖ Naivní analýza časové složitosti (v nejhorším případě) pro n operací je $\mathbf{O(n^2)}$.

❖ Existuje posloupnost operací, která má tuto složitost? Můžeme tuto analýzu zpřesnit?

Amortizovaná složitost – příklad

❖ Uvažme zásobník S a operace PUSH(S, x), POP(S) a MULTIPOP(S, k) – odebere k prvků případně vyprázdní zásobník pokud $|S| \leq k$.

❖ **Metoda seskupování:** Rozdělíme operace do skupin a analyzujeme jejich složitost. Celková složitost je menší nebo rovna součtu složitostí jednotlivých skupin.

❖ Skupina 1: Posloupnost n operací PUSH(S, x) má složitost n

❖ Skupina 2: Složitost posloupnosti operací POP(S) a MULTIPOP(S, k) je daná počtem prvků odebraných z S a tudíž nemůže být větší než počet provedených operací PUSH(S, x). Složitost celé skupiny je tedy menší než n.

❖ Celková složitost libovolných n operací je menší než $\mathbf{2n}$.

Amortizovaná složitost – metoda účtů

❖ Každé operaci přiřadíme kredit. Při realizaci operace zaplatíme její cenu dle následujících pravidel:

- pokud cena operace $\leq$ kredit operace, tak operaci zaplatíme kredity a zbylé kredity vložíme na účet
- pokud cena operace $\geq$ kredit operace, tak scházející kredity odebereme z účtu

❖ Na začátku je na účtě 0 kreditů. Pokud během celého výpočtu je počet kreditů na účtě **nezáporný**, tak platí, že součet kreditů vykonaných operací $\geq$ složitost těchto operací.

Amortizovaná složitost – metoda účtů

❖ Každé operaci přiřadíme cenu a kredit. Při realizaci operace zaplatíme její cenu dle následujících pravidel:

- pokud cena operace $\leq$ kredit operace, tak operaci zaplatíme kredity a zbylé kredity vložíme na účet
- pokud cena operace $\geq$ kredit operace, tak scházející kredity odebereme z účtu

❖ Na začátku je na účtě 0 kreditů. Pokud během celého výpočtu je počet kreditů na účtě **nezáporný**, tak platí, že součet kreditů vykonaných operací $\geq$ složitost těchto operací.

❖ Zpět k našemu příkladu

operace	cena	kredit
PUSH(S, x)	1	2
POP(S)	$\min\{1, \lvert S\rvert\}$	0
MULTIPOP(S, k)	$\min\{k, \lvert S\rvert\}$	0

Při operaci PUSH(S, x) se předplatíme jeden kredit na případné odstranění vloženého prvku

❖ Zřejmě platí invariant: Počet kreditů na účtu je rovný počtu prvkou v S. Tudíž skutečně platí, že počet kreditů na účtě je vždy nezáporný. Celková cena n operací je $\leq 2n$.

Amortizovaná složitost – ukázka

- Příklad: dynamicky alokovaná tabulka T (odebraní záznamu zneplatní řádek)
  - neznámá velikost – dynamická realokace mění kapacitu
  - při naplnění tabulky alokujeme větší tabulku a záznamy do ní přesuneme
  - při vyprázdnění na určitou mez alokujeme menší tabulku a záznamy do ní přesuneme
  - $\alpha(T)$ je poměr platných záznamů (velikost) ku kapacitě tabulky
  - pokud je $\alpha(T) = 1$ vložení záznamů vede k alokaci 2-krát větší tabulky a přesunu
  - pokud je $\alpha(T) < 0.5$ odebrání záznamů vede k alokaci 2-krát menší tabulky a přesunu
- Asymptotická složitost nejhoršího případu
  - složitost jedné operace vložení i odebrání je rovna n (velikost tabulky)
  - složitost n operací je v nejhorším případě $O(n^2)$

Amortizovaná složitost – ukázka

- Amortizovaná složitost – seskupování
  - n operací vložení:
    - c_i – cena i-té operace vložení
    - $c_i = i$ pokud $i - 1$ je mocninou 2 , jinak $c_i = 1$
    - $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$

- Amortizovaná složitost – metoda účtů
  - každé operaci vložení dáme 3 kredity
    - 1 kredit zaplatí vložení,
    - 1 kredit zůstane na účtě pro přesun tohoto záznamu
    - 1 kredit zůstane na účtě pro přesun záznamu, který je v tabulce, ale nemá žádný kredit
  - nechť byla alokována tabulka o velikosti m a bylo sem přesunuto $m/2$ záznamů
    - těchto $m/2$ záznamů nemá na účtě žádný kredit
    - než se tabulka znovu zaplní, bude na účtě m kreditů, které zaplatí přesun
  - celková cena n operací vložení je $\leq 3n$

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

- zabráníme tomu, že kapacita může oscilovat mezi dvěma hodnotami při malém počtu operací (viz výše)
- pokud je $\alpha(T) < 0.25$ odebrání záznamu vede k alokaci 2-krát menšího pole a přesunu
- amortizovaná složitost libovolných n operací (seskupením, hlavní myšlenka)
  - $2^k + 1$ operací vložení, kapacita 2^{k+1}, cena $< 3(2^k + 1)$
  - realokaci vyvolá $2^{k-1} + 1$ operací odebrání, kapacita je 2^k, cena $2^{k-1} + 1 + 2^{k-1}$
  - další realokaci vyvolá $2^{k-1} + 1$ operací vložení, kapacita 2^{k+1}, cena $2^{k-1} + 1 + 2^k$
  - celkově operací: $2^k + 1 + 2 * 2^{k-1} + 2 = 2^{k+1} + 3$
  - celkově cena: $3 * 2^{k-1} + 4 * 2^k + 5 \leq 6 * 2^k + 5 = 3 * 2^{k+1} + 5$
  - ukázali jsem, že n operací má cenu $O(n)$

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

- zabráníme tomu, že kapacita může oscilovat mezi dvěma hodnotami při malém počtu operací (viz výše)
- pokud je $\alpha(T) < 0.25$ odebrání záznamu vede k alokaci 2-krát menšího pole a přesunu
- amortizovaná složitost libovolných n operací (metodou účtů, hlavní myšlenka)
  - každé operaci vložení dáme 3 kredity (viz výše)
  - každé operaci odebraní dáme 2 kredity: 1 kredit na účet pro realokaci při odebírání
  - nechť byla alokována tabulka o velikosti m a bylo sem přesunuto $m/2$ záznamů bez kreditů
  - realokace při vkládání (viz výše)
  - než se tabulka zmenší na $m/4$, bude na účtě $m/4$ kreditů, které zaplatí přesun
  - celková cena n libovolných operací je $\leq 3n$

Třídy složitosti

Složitost problémů

❖ Přejdeme nyní od složitosti konkrétních algoritmů (Turingových strojů) ke složitosti problémů.

❖ Třídy složitosti zavádíme jako prostředek ke kategorizaci (vytvoření hierarchie) problémů dle jejich složitosti, tedy dle toho, jak dalece efektivní algoritmy můžeme navrhnout pro jejich rozhodování (u funkcí můžeme analogicky mluvit o složitosti jejich vyčíslování).

❖ Podobně jako u určování typu jazyka se budeme snažit zařadit problém do co nejnižší třídy složitosti – tedy určit složitost problému jako složitost jeho nejlepšího možného řešení.

❖ Zařazení různých problémů do téže třídy složitosti může odhalit různé vnitřní podobnosti těchto problémů a může umožnit řešení jednoho převodem na druhý (a pragmatické využití např. různých již vyvinutých nástrojů).

Třídy složitosti

Definice 12.5 Mějme dány funkce $t, s : \mathbb{N} \to \mathbb{N}$ a nechť T_M, resp. S_M, značí časovou, resp. prostorovou, složitost TS M. Definujeme následující časové a prostorové třídy složitosti deterministických a nedeterministických TS:

- $DTime[t(n)] = \{L \mid \exists\ k$-páskový DTS $M : L = L(M)$ a $T_M \in O(t(n))\}$.
- $NTime[t(n)] = \{L \mid \exists\ k$-páskový NTS $M : L = L(M)$ a $T_M \in O(t(n))\}$.
- $DSpace[s(n)] = \{L \mid \exists\ k$-páskový DTS $M : L = L(M)$ a $S_M \in O(s(n))\}$.
- $NSpace[s(n)] = \{L \mid \exists\ k$-páskový NTS $M : L = L(M)$ a $S_M \in O(s(n))\}$.

❖ Definici tříd složitosti pak přímočaře zobecňujeme tak, aby mohly být založeny na množině funkcí, nejen na jedné konkrétní funkci.

❖ *Poznámka*: Dále ukážeme, že použití více pásek přináší jen polynomiální zrychlení. Na druhou stranu ukážeme, že zatímco nedeterminismus nepřináší nic podstatného z hlediska vyčíslitelnosti, může přinášet mnoho z hlediska složitosti.

Časově/prostorově zkonstruovatelné funkce

❖ Třídy složitosti obvykle budujeme nad tzv. časově/prostorově zkonstruovatelnými funkcemi:

- Důvodem zavedení časově/prostorově zkonstruovatelných funkcí je dosáhnout intuitivní hierarchické struktury tříd složitosti – např. odlišení tříd $f(n)$ a $2^{f(n)}$, což, jak uvidíme, pro třídy založené na obecných rekurzívních funkcích nelze.

Definice 12.6 Funkci $t : \mathbb{N} \to \mathbb{N}$ nazveme časově zkonstruovatelnou (time constructible), jestliže existuje vícepáskový TS, jenž pro libovolný vstup w zastaví po přesně $t(|w|)$ krocích.

Definice 12.7 Funkci $s : \mathbb{N} \to \mathbb{N}$ nazveme prostorově zkonstruovatelnou (space constructible), jestliže existuje vícepáskový TS, jenž pro libovolný vstup w zastaví s využitím přesně $s(|w|)$ buněk pásky.

❖ *Poznámka*: Pokud je jazyk L nad Σ přijímán strojem v čase/prostoru omezeném časově/prostorově zkonstruovatelnou funkcí, pak je také přijímán strojem, který pro každé $w \in \Sigma^*$ vždy zastaví:

- U časového omezení $t(n)$ si stačí předem spočíst, jaký je potřebný počet kroků a zastavit po jeho vyčerpání.
- U prostorového omezení spočteme z $s(n)$, $|Q|$, $|\Delta|$ maximální počet konfigurací, které můžeme vidět a z toho také plyne maximální počet kroků, které můžeme udělat, aniž bychom cyklili.

Nejběžněji užívané třídy složitosti

❖ Deterministický/nedeterministický polynomiální čas:

$$\mathbf{P} = \bigcup_{k=0}^{\infty} DTime(n^k) \qquad \mathbf{NP} = \bigcup_{k=0}^{\infty} NTime(n^k)$$

❖ Deterministický/nedeterministický polynomiální prostor:

$$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} DSpace(n^k) \quad \equiv \quad \mathbf{NPSPACE} = \bigcup_{k=0}^{\infty} NSpace(n^k)$$

❖ *Poznámka*: Problémy ze třídy $\mathbf{PSPACE}$ se často reálně neřeší v polynomiálním prostoru – zvyšují se prostorové nároky výměnou za alespoň částečné snížení časových nároků (např. z $O(2^{n^2})$ na $O(2^n)$ u LTL model checkingu apod.). Intuitivně: je možno si pamatovat více mezivýsledků a není třeba je znovu spočíst.

Třídy pod a nad polynomiální složitostí

- Deterministický/nedeterministický logaritmický prostor:

$$\mathbf{LOGSPACE} = \bigcup_{k=0}^{\infty} DSpace(k \lg n) \qquad \mathbf{NLOGSPACE} = \bigcup_{k=0}^{\infty} NSpace(k \lg n)$$

- Deterministický/nedeterministický exponenciální čas:

$$\mathbf{EXP} = \bigcup_{k=0}^{\infty} DTime(2^{n^k}) \qquad \mathbf{NEXP} = \bigcup_{k=0}^{\infty} NTime(2^{n^k})$$

- Deterministický/nedeterministický exponenciální prostor:

$$\mathbf{EXPSPACE} = \bigcup_{k=0}^{\infty} DSpace(2^{n^k}) \quad \equiv \quad \mathbf{NEXPSPACE} = \bigcup_{k=0}^{\infty} NSpace(2^{n^k})$$

Třídy nad exponenciální složitostí

- Det./nedet. k-exponenciální čas/prostor založený na věži exponenciál $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ o výšce k:

$$\textbf{k-EXP} = \bigcup_{l=0}^{\infty} DTime(2^{2^{\cdot^{\cdot^{\cdot^{2^{n^l}}}}}})$$

$$\textbf{k-NEXP} = \bigcup_{l=0}^{\infty} NTime(2^{2^{\cdot^{\cdot^{\cdot^{2^{n^l}}}}}})$$

$$\textbf{k-EXPSPACE} = \bigcup_{l=0}^{\infty} DSpace(2^{2^{\cdot^{\cdot^{\cdot^{2^{n^l}}}}}}) \equiv \textbf{k-NEXPSPACE} = \bigcup_{l=0}^{\infty} NSpace(2^{2^{\cdot^{\cdot^{\cdot^{2^{n^l}}}}}})$$

$$\textbf{ELEMENTARY} = \bigcup_{k=0}^{\infty} \textbf{k-EXP}$$

Vrchol hierarchie tříd složitosti

❖ Na vrcholu hierarchie tříd složitosti se pak hovoří o obecných třídách jazyků (funkcí), se kterými jsme se již setkali:

- třída primitivně-rekurzívních funkcí **PR** (implementovatelných pomocí zanořených cyklů s pevným počtem opakování – `for i=... to ...`),
- třída $\mu-$rekurzívních funkcí (rekurzívních jazyků) **R** (implementovatelných pomocí cyklů s předem neurčeným počtem opakování – `while ...`) a
- třída rekurzívně vyčíslitelných funkcí (rekurzívně vyčíslitelných jazyků) **RE**.