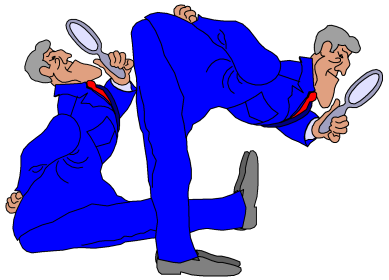
5. Výpočetní složitost

Jan Paseka

Ústav matematiky a statistiky
Masarykova univerzita

23. listopadu 2023

O čem to bude



1. Příklady
   - Úvod
   - Násobení
   - Permanenty
   - Třídění
   - Prvočíselnost
   - Největší společný dělitel a Euklidův algoritmus
2. *P*=polynomiální čas
3. *NP* = nedeterministický polynomiální čas
4. Splnitelnost - SATisfiability
5. Paměťová složitost
6. Náhodné algoritmy

Příklady I

FI Důkaz toho, že existují nerozhodnutelné problémy a nevyčíslitelné funkce, je jedním z největších výdobytků v této oblasti.

Příklady I

FI Důkaz toho, že existují nerozhodnutelné problémy a nevyčíslitelné funkce, je jedním z největších výdobytků v této oblasti.

Šifrovací systém, jehož dešifrování je založeno na výpočtu nevyčíslitelných funkcí, by měl výhodnou pozici. Můžeme však snadno ověřit, že takovéto zbožné přání nemůže být splněno: všechny takovéto systémy jsou konečné a tedy mohou být narušeny prověřením všech možností.

Příklady I

FI Důkaz toho, že existují nerozhodnutelné problémy a nevyčíslitelné funkce, je jedním z největších výdobytků v této oblasti.

Šifrovací systém, jehož dešifrování je založeno na výpočtu nevyčíslitelných funkcí, by měl výhodnou pozici. Můžeme však snadno ověřit, že takovéto zbožné přání nemůže být splněno: všechny takovéto systémy jsou konečné a tedy mohou být narušeny prověřením všech možností.

Teorie výpočetní složitosti se týká třídy problémů, které lze v principu vyřešit: ale vzhledem k této třídě se teorie pokouší klasifikovat problémy podle jejich výpočetní obtížnosti v závislosti na množství času nebo paměti potřebných pro toto řešení.

Příklady II

Porozumění základním pojmům teorie složitosti je podstatné pro kryptografii a v této kapitole se budeme snažit pokrýt podstatu problémů teorie složitosti. Nejprve začněme informálně s několika příklady.

Příklady II

Porozumění základním pojmům teorie složitosti je podstatné pro kryptografii a v této kapitole se budeme snažit pokrýt podstatu problémů teorie složitosti. Nejprve začněme informálně s několika příklady.

Příklad 1.1 (*Násobení přirozených čísel)*

Uvažme problém násobení dvou binárních n-bitových čísel x a y. Je-li $x = x_1 \dots x_n$ *a* $y = y_1 \dots y_n$, *můžeme provést standardní metodu "dlouhého násobení", která je učena na základní škole následovným způsobem:*

Příklady II

Porozumění základním pojmům teorie složitosti je podstatné pro kryptografii a v této kapitole se budeme snažit pokrýt podstatu problémů teorie složitosti. Nejprve začněme informálně s několika příklady.

Příklad 1.1 (*Násobení přirozených čísel)*

Uvažme problém násobení dvou binárních n-bitových čísel x a y. Je-li $x = x_1 \dots x_n$ *a* $y = y_1 \dots y_n$, *můžeme provést standardní metodu "dlouhého násobení", která je učena na základní škole následovným způsobem:*

Postupně násobme x čísly y_1, y_2 *atd., posuňme a pak přičtěme výsledek. Každé násobení x číslem* y_i *nás stojí n jednoduchých bitových operací. Podobně sečtení n součinů nám zabere* $O(n^2)$ *bitových operací. Je tedy celkový počet operací* $O(n^2)$.

Příklady III

Příklad 1.1 (***Násobení přirozených čísel*** - pokračování)

Můžeme výše uvedené ještě zlepšit? Tj., existuje rychlejší algoritmus v tom smyslu, že provede podstatně méně bitových informací? Přesněji, jsou-li dána 2 n-bitová čísla, n sudé, píšeme

$$x = a2^{\frac{n}{2}} + b, \qquad y = c2^{\frac{n}{2}} + d,$$

pak součin z lze získat pomocí tří násobení $\frac{1}{2}$ n-bitových čísel použitím reprezentace

$$z = xy = (ac)2^n + [ac + bd - (a - b)(c - d)]2^{\frac{n}{2}} + bd.$$

Příklady III

Příklad 1.1 (***Násobení přirozených čísel*** - pokračování)

Můžeme výše uvedené ještě zlepšit? Tj., existuje rychlejší algoritmus v tom smyslu, že provede podstatně méně bitových informací? Přesněji, jsou-li dána 2 n-bitová čísla, n sudé, píšeme

$$x = a2^{\frac{n}{2}} + b, \qquad y = c2^{\frac{n}{2}} + d,$$

pak součin z lze získat pomocí tří násobení $\frac{1}{2}$ n-bitových čísel použitím reprezentace

$$z = xy = (ac)2^n + [ac + bd - (a - b)(c - d)]2^{\frac{n}{2}} + bd.$$

Označíme-li $T(n)$ čas, který nám zabere násobení podle této metody, máme, prože násobení 2^n je rychlé, že

$$T(n) \leq 3T(\frac{n}{2}) + O(n).$$

Příklady IV

Příklad 1.1 (***Násobení přirozených čísel*** - pokračování)

Po vyřešení výše uvedené rekurentní nerovnosti obdržíme, že

$$T(n) \leq An^{log3} + Bn,$$

kde A a B jsou konstanty.

Příklady IV

Příklad 1.1 (***Násobení přirozených čísel*** - pokračování)

Po vyřešení výše uvedené rekurentní nerovnosti obdržíme, že

$$T(n) \leq An^{log3} + Bn,$$

kde A a B jsou konstanty.

Protože $log\,3 \simeq 1.59$, *máme k dispozici algoritmus o časové složitosti* $O(n^{1.59})$ *v porovnání se standardním* $O(n^2)$ *algoritmem. V současnosti má jeden z nejlepších známých algoritmů (Schönhagen, Strassen) složitost* $O(n \log n \log\log n)$.

Příklady V

Příklad 1.2 (*Determinanty a permanenty)*

Uvažujme následující dva výpočetní problémy. Pro každý vstup složený z binární matice A typu $n \times n$ chceme vypočítat

1. *determinant matice A, píšeme pak detA,*
2. *permanent matice A, píšeme pak perA, permanenent je definován jako*

$$perA = \sum_{\pi} a_{1\pi(1)} a_{2\pi(2)} \dots a_{n\pi(n)},$$

kde sčítáme přes všechny permutace π na množině $\{1, 2, \dots, n\}$ a a_{ij} značí $(i, j)-$tou komponentu matice A.

Příklady V

Příklad 1.2 (*Determinanty a permanenty)*

Uvažujme následující dva výpočetní problémy. Pro každý vstup složený z binární matice A typu $n \times n$ chceme vypočítat

1. *determinant matice A, píšeme pak detA,*
2. *permanent matice A, píšeme pak perA, permanenent je definován jako*

$$perA = \sum_{\pi} a_{1\pi(1)} a_{2\pi(2)} \dots a_{n\pi(n)},$$

kde sčítáme přes všechny permutace π na množině $\{1, 2, \dots, n\}$ a a_{ij} značí $(i, j)-$tou komponentu matice A.

Zdánlivě je permanent mnohem jednodušší funkce matice A než její determinant; jedná se o součet toho samého systému termů, ale bez toho, že bychom dávali pozor na $\pm$ znaménka jako u determinantu.

Příklady VI

Příklad 1.2 (***Determinanty a permanenty*** - pokračování)

Avšak z hlediska výpočetní složitosti platí opak: zatímco determinant je relativně snadná funkce k výpočtu, u permanentu se prokázalo, že se téměř vždy jedná o neobvykle obtížnou záležitost.

Příklady VI

Příklad 1.2 (***Determinanty a permanenty*** - pokračování)

Avšak z hlediska výpočetní složitosti platí opak: zatímco determinant je relativně snadná funkce k výpočtu, u permanentu se prokázalo, že se téměř vždy jedná o neobvykle obtížnou záležitost.

Upřesněme výše uvedené (za předpokladu ohraničenosti délky vstupů v naší matici): standardní Gaussova eliminační metoda výpočtu determinantu matice $n \times n$ potřebuje $O(n^3)$ bitových operací, přičemž V. Strassen zkonstruoval algoritmus, který potřebuje

$$O(n\log_2 7) = O(n^{2.81\ldots})$$

bitových operací.

Příklady VI

Příklad 1.2 (***Determinanty a permanenty*** - pokračování)

Redukce exponentu pod hranici $\log_2 7$ se prokázalo být velmi obtížné a jeden z nejrychlejších současných algoritmů (Coppersmith, Winograd) potřebuje $O(n^{2.3976\ldots})$ operací.

Příklady VI

Příklad 1.2 (***Determinanty a permanenty*** - pokračování)

Redukce exponentu pod hranici $\log_2 7$ se prokázalo být velmi obtížné a jeden z nejrychlejších současných algoritmů (Coppersmith, Winograd) potřebuje $O(n^{2.3976\ldots})$ operací.

Snadno je vidět, že všechny vstupy matice musí být načteny a tedy

$$n^2 \leq t_{det}(n) \leq Cn^{2.3976\ldots}$$

kde $t_{det}(n)$ označuje časovou složitost problému výpočtu determinantu a C je nějaká konstanta.

Příklady VI

Příklad 1.2 (***Determinanty a permanenty*** - pokračování)

Redukce exponentu pod hranici $\log_2 7$ se prokázalo být velmi obtížné a jeden z nejrychlejších současných algoritmů (Coppersmith, Winograd) potřebuje $O(n^{2.3976\ldots})$ operací.

Snadno je vidět, že všechny vstupy matice musí být načteny a tedy

$$n^2 \leq t_{det}(n) \leq Cn^{2.3976\ldots}$$

kde $t_{det}(n)$ označuje časovou složitost problému výpočtu determinantu a C je nějaká konstanta.

Pro permanent však oproti výše uvedenému žádný takový algoritmus není znám. Nejrychlejší doposud známý algoritmus je pouze o něco lepší než sečtení všech $n!$ termů naší sumy.

Příklady VII

Příklad 1.3 (*Třídění)*

Předpokládejme, že chceme sestrojit algoritmus, který, obdrží-li na vstupu n celých čísel $a_1, \ldots, a_n$, setřídí tyto v rostoucím pořadí. Snadný, téměř instinktivní přístup je následující algoritmus, známý jako **Bubblesort**.

Příklady VII

Příklad 1.3 (*Třídění)*

Předpokládejme, že chceme sestrojit algoritmus, který, obdrží-li na vstupu n celých čísel $a_1, \ldots, a_n$, setřídí tyto v rostoucím pořadí. Snadný, téměř instinktivní přístup je následující algoritmus, známý jako **Bubblesort**.

Postupně porovnávejme a_1 s každým s prvků $a_2, \ldots, a_n$. Pro maximální index i takový, že $a_i < a_1$ umístěme a_1 za a_i a obdržíme pak nové uspořádání. Po $n - 1$ porovnáních obdržíme uspořádání $b_1, \ldots, b_n$; je okamžitě vidět, že se jedná o vzestupně uspořádaný seznam. Jednoduchý výpočet ukazuje, že k výše uvedenému je třeba $O(n^2)$ porovnání.

Příklady VIII

Příklad 1.3 (*Třídění* - pokračování)

Poznamenejme, že máme několik rekurzivních algoritmů založených na principu rozděl a panuj tím, že třídíme 2 polovice množiny a pak spojíme setříděné seznamy k sobě, což nám zabere pouze $O(n \log n)$ srovnání. Pro názornost uveďme následující tabulku

Příklady VIII

Příklad 1.3 (*Třídění* - pokračování)

Poznamenejme, že máme několik rekurzivních algoritmů založených na principu rozděl a panuj tím, že třídíme 2 polovice množiny a pak spojíme setříděné seznamy k sobě, což nám zabere pouze $O(n \log n)$ srovnání. Pro názornost uveďme následující tabulku

n	$n \log_2 n$	n^2
50	$\simeq 300$	2500
500	$\simeq 4500$	250000

.

Příklady VIII

Příklad 1.3 (*Třídění* - pokračování)

Poznamenejme, že máme několik rekurzivních algoritmů založených na principu rozděl a panuj tím, že třídíme 2 polovice množiny a pak spojíme setříděné seznamy k sobě, což nám zabere pouze $O(n \log n)$ srovnání. Pro názornost uveďme následující tabulku

n	$n \log_2 n$	n^2
50	$\simeq 300$	2500
500	$\simeq 4500$	250000

.

Ovšem, výraz $O(n \log n)$ může skrýt velké konstantní výrazy, ale tak jako tak se jedná o velký rozdíl, obzvláště proto, že třídění je často používaný algoritmus a velikost seznamů je často velmi velká.

Příklady IX

Příklad 1.3 (*Třídění* - pokračování)

Jiný fascinující pohled na třídění je ten, že existuje spodní mez stejného řádu pro každý algoritmus založený na třídění.

Příklady IX

Příklad 1.3 (*Třídění* - pokračování)

Jiný fascinující pohled na třídění je ten, že existuje spodní mez stejného řádu pro každý algoritmus založený na třídění.

Často mluvíme o informačně-teoretické spodní mezi, ale nejedná se o nic jiného, než o přímý důsledek pozorování, že každý algoritmus založený na srovnání lze reprezentovat pomocí binání stromové struktury a protože musíme pokrýt všech $n!$ možných uspořádání, každý takovýto strom musí mít alespoň $n!$ listů.

Příklady IX

Příklad 1.4 (*Test prvočíselnosti)*

Bezprostředně se zdá, že testovat, zda přirozené číslo N je prvočíslo, lze vyřešit velmi rychlým a snadným algoritmem: testujeme, zda je N dělitelené 2 nebo nějakým lichým číslem z intervalu $[3, N^{\frac{1}{2}}]$.

Příklady IX

Příklad 1.4 (*Test prvočíselnosti)*

Bezprostředně se zdá, že testovat, zda přirozené číslo N je prvočíslo, lze vyřešit velmi rychlým a snadným algoritmem: testujeme, zda je N dělitelené 2 nebo nějakým lichým číslem z intervalu $[3, N^{\frac{1}{2}}]$.

Protože se jedná pouze o $\frac{1}{2}N^{\frac{1}{2}}$ *dělení, jedná se o polynomiální algoritmus v proměnné N a tudíž rychlý algoritmus.*

Příklady IX

Příklad 1.4 (*Test prvočíselnosti)*

Bezprostředně se zdá, že testovat, zda přirozené číslo N je prvočíslo, lze vyřešit velmi rychlým a snadným algoritmem: testujeme, zda je N dělitelené 2 nebo nějakým lichým číslem z intervalu $[3, N^{\frac{1}{2}}]$.

Protože se jedná pouze o $\frac{1}{2}N^{\frac{1}{2}}$ *dělení, jedná se o polynomiální algoritmus v proměnné N a tudíž rychlý algoritmus.*

Avšak další úvahy ukazují, že reprezentace čísla N v počítači by byl binární řetězec délky $\lceil \log N \rceil$ *a tudíž, abychom mohli algoritmus na testování prvočíselnosti považovat za rychlý, jeho výpočetní složitost musí být polynomiální v* $n = \lceil \log N \rceil$.

Příklady X

Příklad 1.4 (***Test prvočíselnosti*** - pokračování)

V současnosti jsou k dispozici algoritmy se složitostí

$$t(N) = O(\ln N)^{c \ln \ln \ln N},$$

kde c je kladná konstanta.

Příklady X

Příklad 1.4 (***Test prvočíselnosti*** - pokračování)

V současnosti jsou k dispozici algoritmy se složitostí

$$t(N) = O(\ln N)^{c \ln \ln \ln N},$$

kde c je kladná konstanta.

V roce 2002 byl poprvé nalezen M. Agrawalam, K. Neerajem a N. Saxenou deterministický test na prvočíselnost. Jejich test na prvočíselnost měl složitost $O((\log n)^{12})$. Následně Lenstra a Pomerance představili verzi testu, která běží v čase $O((\log n)^6)$.

Příklady XI

Příklad 1.5 (*Největší společný dělitel a Euklidův algoritmus*)

Uvažujme problém nalezení největšího společného dělitele (nsd) dvou přirozených čísel u a v.

Příklady XI

Příklad 1.5 (*Největší společný dělitel a Euklidův algoritmus*)

Uvažujme problém nalezení největšího společného dělitele (nsd) dvou přirozených čísel u a v.

Zřejmá metoda je faktorizace obou čísel na prvočísla

$$u = 2^{u_1}3^{u_2}5^{u_3}\ldots, \qquad v = 2^{v_1}3^{v_2}5^{v_3}\ldots,$$

pak lze zjistit jejich největší společný dělitel následovně

$$w = nsd(u, v) = 2^{w_1}3^{w_2}5^{w_3}\ldots,$$

kde $w_i = min\{u_i, v_i\}$.

Příklady XI

Příklad 1.5 (*Největší společný dělitel a Euklidův algoritmus*)

Uvažujme problém nalezení největšího společného dělitele (nsd) dvou přirozených čísel u a v.

Zřejmá metoda je faktorizace obou čísel na prvočísla

$$u = 2^{u_1}3^{u_2}5^{u_3}\ldots, \qquad v = 2^{v_1}3^{v_2}5^{v_3}\ldots,$$

pak lze zjistit jejich největší společný dělitel následovně

$$w = nsd(u, v) = 2^{w_1}3^{w_2}5^{w_3}\ldots,$$

kde $w_i = min\{u_i, v_i\}$.

Jedná se však o velmi neefektivní postup. Potřebujeme totiž faktorizovat obě čísla a tento postup nelze rychle provést pro velká přirozená čísla.

Příklady XII

Příklad 1.5 (***Euklidův algoritmus*** - pokračování)

Metoda, jíž tento postup můžeme obejít, je známá jako **Euklidův algoritmus**.

Příklady XII

Příklad 1.5 (***Euklidův algoritmus*** - pokračování)

Metoda, jíž tento postup můžeme obejít, je známá jako **Euklidův algoritmus**.
Předpokládejme, že $u > v > 0$. *Pak obdržíme posloupnost dělení:*

$$\begin{aligned} u &= a_1 v + b_1, & 0 \leq b_1 < v, \\ v &= a_2 b_1 + b_2, & 0 \leq b_2 < b_1, \\ b_1 &= a_3 b_2 + b_3, & 0 \leq b_3 < b_2, \\ &\vdots \\ b_{k-2} &= a_k b_{k-1} + b_k, & 0 \leq b_3 < b_2, \end{aligned}$$

která skončí buď $b_k = 0$ *nebo* $b_k = 1$. *Pokud* $b_k = 1$, *jsou čísla* u *a* v *nesoudělná; pokud* $b_k = 0$, *je* $nsd(u, v) = b_{k-1}$.

Příklady XIII

Příklad 1.5 (***Euklidův algoritmus*** - pokračování)

O tomto algoritmu lze snadno dokázat, že je korektní a detailní analýza jeho účinnosti ukáže, že nejhorší případ (měřeno počtem dělení) nastane, jsou-li u a v za sebou následující Fibonacciho čísla F_{n+2} a F_{n+1}. Pak v tomto případě

$$F_{k+2} = F_{k+1} + F_k,$$

a to vede k následujícímu Lamého výsledku (1845)

Příklady XIII

Příklad 1.5 (***Euklidův algoritmus*** - pokračování)

O tomto algoritmu lze snadno dokázat, že je korektní a detailní analýza jeho účinnosti ukáže, že nejhorší případ (měřeno počtem dělení) nastane, jsou-li u a v za sebou následující Fibonacciho čísla F_{n+2} a F_{n+1}. Pak v tomto případě

$$F_{k+2} = F_{k+1} + F_k,$$

a to vede k následujícímu Lamého výsledku (1845)

Věta 1.6

Je-li $0 \leq u, v < N$, pak počet dělení při použití Euklidova algoritmu na u a v je nejvýše

$$\lceil log_{\varphi}(\sqrt{5}N) \rceil - 2,$$

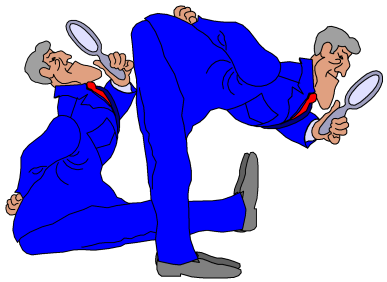
kde φ je zlatý řez $\frac{1}{2}(1 + \sqrt{5})$.

Příklady XIII

Příklad 1.5 (*Euklidův algoritmus* - pokračování)

Tedy složitost bude tvaru $O(log(u+v))$ za předpokladu konstantních algebraických operacích, pokud budeme uvažovat velká celá čísla, bude nutně $O((log(u+v))^2)$, protože každá algebraická operace bude provedena se složitostí $O(log(u+v))$.

O čem to bude



1. Příklady
2. *P*=polynomiální čas
   - Úvod
   - Polynomiálnost
   - Třída *P*
   - Složitost
3. *NP* = nedeterministický polynomiální čas
4. Splnitelnost - SATisfiability
5. Paměťová složitost
6. Náhodné algoritmy

Motivace I

Základní mírou obtížnosti výpočtu je množství doby, které nám výpočet zabere. Zformulování přesné definice, co to je čas, je netriviální záležitost; přesná formulace požaduje velmi precizní definici strojového modelu, jednotky času atd.

Motivace I

Základní mírou obtížnosti výpočtu je množství doby, které nám výpočet zabere. Zformulování přesné definice, co to je čas, je netriviální záležitost; přesná formulace požaduje velmi precizní definici strojového modelu, jednotky času atd.

V příkladech z předchozího paragrafu jsme měřili ***složitost*** výpočtu v pojmech počtu základních operací, které byly prováděny. Mohlo se jednat o součet bitů, srovnání nebo cokoliv jiného.

Motivace I

Základní mírou obtížnosti výpočtu je množství doby, které nám výpočet zabere. Zformulování přesné definice, co to je čas, je netriviální záležitost; přesná formulace požaduje velmi precizní definici strojového modelu, jednotky času atd.

V příkladech z předchozího paragrafu jsme měřili ***složitost*** výpočtu v pojmech počtu základních operací, které byly prováděny. Mohlo se jednat o součet bitů, srovnání nebo cokoliv jiného.

Klíčové pojmy jsou následující:

1. složitost je funkce ***velikosti vstupu*** (obvykle ji značíme jako n),
2. pro danou velikost vstupu n je složitost doba ***nejhoršího možného případu*** běhu algoritmu.

Motivace II

Připomeňme si, že při testování prvočíselnosti přirozeného čísla N jsme obdrželi jinou složitost v případě, že jsme považovali vstup velikosti N nebo vhodněji pomocí reprezentace $n = \lceil \log N \rceil$ binárních číslic.

Motivace II

Připomeňme si, že při testování prvočíselnosti přirozeného čísla N jsme obdrželi jinou složitost v případě, že jsme považovali vstup velikosti N nebo vhodněji pomocí reprezentace $n = \lceil \log N \rceil$ binárních číslic.

Na základě tohoto důsledku bude ***velikost vstupu*** vždy považována za "***přirozenou***" délku ekonomického vstupu.

Motivace II

Připomeňme si, že při testování prvočíselnosti přirozeného čísla N jsme obdrželi jinou složitost v případě, že jsme považovali vstup velikosti N nebo vhodněji pomocí reprezentace $n = \lceil \log N \rceil$ binárních číslic.

Na základě tohoto důsledku bude ***velikost vstupu*** vždy považována za "***přirozenou***" délku ekonomického vstupu.

Co se týče definice složitosti jako nejhoršího možného případu, jiná možnost – "***průměrný případ***" – se potýká s obtížemi, a to jak teoretickými tak praktickými.

Motivace II

Připomeňme si, že při testování prvočíselnosti přirozeného čísla N jsme obdrželi jinou složitost v případě, že jsme považovali vstup velikosti N nebo vhodněji pomocí reprezentace $n = \lceil \log N \rceil$ binárních číslic.

Na základě tohoto důsledku bude ***velikost vstupu*** vždy považována za "***přirozenou***" délku ekonomického vstupu.

Co se týče definice složitosti jako nejhoršího možného případu, jiná možnost – "***průměrný případ***" – se potýká s obtížemi, a to jak teoretickými tak praktickými.

Ne poslední obtížnost je rozhodnout citlivě o pravdivostním rozdělení na vstupu.

Polynomiálnost I

Nejprve budeme postupovat ***neformálně***. Řekneme, že algoritmus $\mathcal{A}$ má ***polynomiální složitost***, jestliže existuje polynom $p(x)$ tak, že

$$t_{\mathcal{A}}(n) \leq p(n),$$

pro všechna přirozená čísla n, přičemž $t_{\mathcal{A}}(n)$ je maximální doba potřebná algoritmem k výpočtu přes všechny vstupy velikosti n.

Polynomiálnost I

Nejprve budeme postupovat ***neformálně***. Řekneme, že algoritmus $\mathcal{A}$ má ***polynomiální složitost***, jestliže existuje polynom $p(x)$ tak, že

$$t_{\mathcal{A}}(n) \leq p(n),$$

pro všechna přirozená čísla n, přičemž $t_{\mathcal{A}}(n)$ je maximální doba potřebná algoritmem k výpočtu přes všechny vstupy velikosti n.

Problém lze provést v ***polynomiálním čase***, pokud existuje nějaký algoritmus, který ho řeší a má polynomiální složitost, v tomto případě tvrdíme, že ***problém leží v třídě*** P.

Polynomiálnost I

Nejprve budeme postupovat ***neformálně***. Řekneme, že algoritmus $\mathcal{A}$ má ***polynomiální složitost***, jestliže existuje polynom $p(x)$ tak, že

$$t_{\mathcal{A}}(n) \leq p(n),$$

pro všechna přirozená čísla n, přičemž $t_{\mathcal{A}}(n)$ je maximální doba potřebná algoritmem k výpočtu přes všechny vstupy velikosti n.

Problém lze provést v ***polynomiálním čase***, pokud existuje nějaký algoritmus, který ho řeší a má polynomiální složitost, v tomto případě tvrdíme, že ***problém leží v třídě*** P.

Porovnejme naši definici s příklady 1-5 z předchozího paragrafu.

Polynomiálnost II

Příklad 2.1 (Polynomiální složitost násobení - Příklad 1.1)

Násobení přirozených čísel *je operace prováděná v polynomiální době.*

Polynomiálnost II

Příklad 2.1 (Polynomiální složitost násobení - Příklad 1.1)

Násobení přirozených čísel *je operace prováděná v polynomiální době.*

Příklad 2.2 (Determinanty a permanenty - Příklad 1.2)

Výpočet determinantu *je problém, který určitě leží v P. U výpočtu permanentu nevíme, zda tento problém leží v P.*

Polynomiálnost II

Příklad 2.1 (Polynomiální složitost násobení - Příklad 1.1)

Násobení přirozených čísel *je operace prováděná v polynomiální době.*

Příklad 2.2 (Determinanty a permanenty - Příklad 1.2)

Výpočet determinantu *je problém, který určitě leží v P. U výpočtu permanentu nevíme, zda tento problém leží v P.*

Předpokládá se, že zde neleží, a důkaz jakékoliv implikace by měl velkou důležitost v teorii složitosti. je operace prováděná v polynomiální době.

Polynomiálnost III

Příklad 2.3 (Polynomiální složitost třídění - Příklad 1.3)

Třídění *lze provést pomocí* $O(n \log n)$ *srovnání a protože srovnání lze provést v polynomiálním čase, leží třídění v P.*

Polynomiálnost III

Příklad 2.3 (Polynomiální složitost třídění - Příklad 1.3)

Třídění *lze provést pomocí* $O(n \log n)$ *srovnání a protože srovnání lze provést v polynomiálním čase, leží třídění v P.*

Příklad 2.4 (Testu prvočíselnosti - Příklad 1.4)

O **testu prvočíselnosti** *od roku 2002 víme, že leží v P.*

Polynomiálnost III

Příklad 2.3 (Polynomiální složitost třídění - Příklad 1.3)

Třídění *lze provést pomocí* $O(n \log n)$ *srovnání a protože srovnání lze provést v polynomiálním čase, leží třídění v P.*

Příklad 2.4 (Testu prvočíselnosti - Příklad 1.4)

O **testu prvočíselnosti** *od roku 2002 víme, že leží v P.*

Tento vynikající výsledek prof. Manindry Agrawala spolu s jeho dvěma studenty (Neeraj Kayal a Nitin Saxena) dává polynomiální algoritmus pracující v čase $O(n^{6,5})$ *(při konstantní složitosti aritmetických operací).*

Polynomiálnost III

Příklad 2.3 (Polynomiální složitost třídění - Příklad 1.3)

Třídění *lze provést pomocí* $O(n \log n)$ *srovnání a protože srovnání lze provést v polynomiálním čase, leží třídění v P.*

Příklad 2.4 (Testu prvočíselnosti - Příklad 1.4)

O **testu prvočíselnosti** *od roku 2002 víme, že leží v P.*

Tento vynikající výsledek prof. Manindry Agrawala spolu s jeho dvěma studenty (Neeraj Kayal a Nitin Saxena) dává polynomiální algoritmus pracující v čase $O(n^{6,5})$ *(při konstantní složitosti aritmetických operací).*

Tento algoritmus je poměrně komplikovaný a odhad jeho složitosti vyžaduje dosti netriviální věty z teorie čísel.

Polynomiálnost IV

Příklad 2.5 (Euklidův algoritmus - Příklad 1.5)

Nalezení **největšího společného dělitele** *dvou přirozených čísel velikosti* $\leq N$ *a proto velikosti vstupu* $\log N$ *bitů lze provést v čase* $O(\log N)$ *a proto tento problém leží v* P*.*

Třída P je v současnosti nejdůležitější třídou v matematice a computer science. To, že nějaký problém leží v P, lze obvykle považovat za to, že se jedná o výpočetně dobrý problém.

Polynomiálnost IV

Příklad 2.5 (Euklidův algoritmus - Příklad 1.5)

Nalezení **největšího společného dělitele** *dvou přirozených čísel velikosti* $\leq N$ *a proto velikosti vstupu* $\log N$ *bitů lze provést v čase* $O(\log N)$ *a proto tento problém leží v* P*.*

Třída P je v současnosti nejdůležitější třídou v matematice a computer science. To, že nějaký problém leží v P, lze obvykle považovat za to, že se jedná o výpočetně dobrý problém.

Ačkoliv poslední uvedené obecně zcela neplatí (problém nalezení klik v grafu), následující tvrzení nám ukazují, že se jedná o atraktivní a efektivní pojem.

Třída *P* I

1. Třída *P* je robustní vzhledem k různým reprezentacím vstupních hodnot za předpokladu, že tyto změny jsou vůči sobě polynomiálně korelovány.[1] Například to, zda uvažujeme vstup matice typu $n \times n$ velikosti n nebo n^2, nedělá žádný rozdíl.

[1] Dvě funkce f a g jsou vůči sobě polynomiálně korelovány, pokud existují polynomy p_1 a p_2 tak, že $f(n) \leq p_1(g(n))$ a $g(n) \leq p_2(f(n))$ pro všechna dostatečně velká n.

Třída *P* I

1. Třída *P* je robustní vzhledem k různým reprezentacím vstupních hodnot za předpokladu, že tyto změny jsou vůči sobě polynomiálně korelovány.[1] Například to, zda uvažujeme vstup matice typu $n \times n$ velikosti n nebo n^2, nedělá žádný rozdíl.
2. Třída *P* je robustní vzhledem k použitému modelu výpočetního stroje. Jinak řečeno, zda použijeme Pentium nebo stroj s náhodným přístupem nebo Turingův stroj, naše třída zůstane nezměněna. Toto lze celkem snadno dokázat. Je pouze nutno ověřit, že doby pro simulaci základních operací na obou strojích, jsou polynomiálně korelovány.

[1] Dvě funkce f a g jsou vůči sobě polynomiálně korelovány, pokud existují polynomy p_1 a p_2 tak, že $f(n) \leq p_1(g(n))$ a $g(n) \leq p_2(f(n))$ pro všechna dostatečně velká n.

Třída *P* II

Připomeňme stručně formální definici třídy *P*.

Turingovy stroje – formální definice třídy *P*

Turingův stroj sestává z 2-směrné nekonečné pásky rozdělené do čtverců. Každý čtverec může obsahovat symbol z konečné abecedy Σ obsahující prázdný symbol $*$. Až na konečně mnoho čtverců všechny obsahují prázdný symbol $*$.

Třída *P* II

Připomeňme stručně formální definici třídy *P*.

Turingovy stroje – formální definice třídy *P*

Turingův stroj sestává z 2-směrné nekonečné pásky rozdělené do čtverců. Každý čtverec může obsahovat symbol z konečné abecedy Σ obsahující prázdný symbol $*$. Až na konečně mnoho čtverců všechny obsahují prázdný symbol $*$.

Páska je snímána rychlostí 1 čtverec za jednotku času tzv. ***čtecí i zapisovací*** hlavicí.

Třída *P* II

Připomeňme stručně formální definici třídy P.

Turingovy stroje – formální definice třídy P

Turingův stroj sestává z 2-směrné nekonečné pásky rozdělené do čtverců. Každý čtverec může obsahovat symbol z konečné abecedy Σ obsahující prázdný symbol $*$. Až na konečně mnoho čtverců všechny obsahují prázdný symbol $*$.

Páska je snímána rychlostí 1 čtverec za jednotku času tzv. ***čtecí i zapisovací*** hlavicí.

Stroj může být v jednom z konečné množiny Γ stavů, $\Gamma = \{q_0, q_1, \ldots, q_m\}$ a příslušná akce stroje v danám čase je jednoznačně určena jeho vnitřním stavem a symbolem v současně snímaném čtverci.

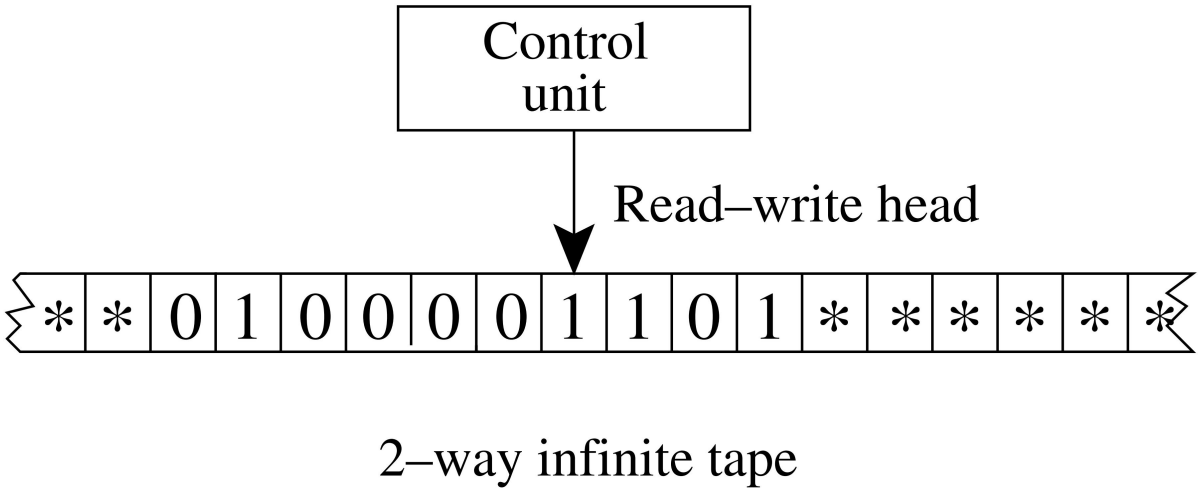
Třída *P* III

Akce provede libovolnou z následujících operací:

1. změní (přepíše) snímaný symbol na jiný symbol ze Σ,
2. posune čtecí hlavici o jeden čtverec doprava ($\longrightarrow$) či doleva ($\longleftarrow$),
3. změní svůj současný stav z q_i na q_j.

Výpočet Turingova stroje je tedy řízen přechodovou funkcí

$$\delta : \Gamma \times \Sigma \to \Gamma \times \Sigma \times \{\longleftarrow, \longrightarrow\}.$$



Třída *P* IV

Výpočet Turingova stroje sestává z následujících kroků:

1. reprezentace výpočtu konečným řetězcem $x \in \Sigma_0^*$, kde $\Sigma_0 = \Sigma \setminus \{*\}$, který je umístěn ve čtvercích $1 - n$, kde n je počet symbolů obsažených v x,

Třída *P* IV

Výpočet Turingova stroje sestává z následujících kroků:

1. reprezentace výpočtu konečným řetězcem $x \in \Sigma_0^*$, kde $\Sigma_0 = \Sigma \setminus \{*\}$, který je umístěn ve čtvercích $1 - n$, kde n je počet symbolů obsažených v x,
2. odstartování činnosti Turingova stroje z jeho počátečního stavu (obvykle q_0) s přepisovací hlavou na čtverci 1 a jeho pokračování se základními operacemi (četní, zapsání a změny stavu) až do doby, kdy stroj skončí v koncovém stavu (q_f).

Třída *P* IV

Výpočet Turingova stroje sestává z následujících kroků:

1. reprezentace výpočtu konečným řetězcem $x \in \Sigma_0^*$, kde $\Sigma_0 = \Sigma \setminus \{*\}$, který je umístěn ve čtvercích $1 - n$, kde n je počet symbolů obsažených v x,
2. odstartování činnosti Turingova stroje z jeho počátečního stavu (obvykle q_0) s přepisovací hlavou na čtverci 1 a jeho pokračování se základními operacemi (četní, zapsání a změny stavu) až do doby, kdy stroj skončí v koncovém stavu (q_f).

Výstup stroje M po jeho aplikování na stav x je obsah pásky dosažený v koncovém stavu.

Třída P IV

Výpočet Turingova stroje sestává z následujících kroků:

1. reprezentace výpočtu konečným řetězcem $x \in \Sigma_0^*$, kde $\Sigma_0 = \Sigma \setminus \{*\}$, který je umístěn ve čtvercích $1 - n$, kde n je počet symbolů obsažených v x,
2. odstartování činnosti Turingova stroje z jeho počátečního stavu (obvykle q_0) s přepisovací hlavou na čtverci 1 a jeho pokračování se základními operacemi (četní, zapsání a změny stavu) až do doby, kdy stroj skončí v koncovém stavu (q_f).

Výstup stroje M po jeho aplikování na stav x je obsah pásky dosažený v koncovém stavu. Jeden ***krok*** výpočtu stroje sestává z jedné akce 1-3 uvedených výše a ***délka*** neboli ***čas použitý*** při výpočtu je počet takovýchto kroků. Pokud M označuje nějaký Turingův stroj , označíme tuto dobu $t_M(x)$.

Třída *P* V

Funkce $f : \Sigma_0^* \to \Sigma_0^*$ je ***vyčíslitelná pomocí Turingova stroje*** M, jestliže pro všechna $x \in \Sigma_0^*$, x je vstup pro M, v případě ukončení výpočtu stroj zastaví s hodnotou $f(x)$ na jeho výstupní pásce.

Třída *P* V

Funkce $f : \Sigma_0^* \to \Sigma_0^*$ je ***vyčíslitelná pomocí Turingova stroje*** M, jestliže pro všechna $x \in \Sigma_0^*$, x je vstup pro M, v případě ukončení výpočtu stroj zastaví s hodnotou $f(x)$ na jeho výstupní pásce.

Tedy Turingův stroj je přesná analogie počítače – každý výpočet, který lze vykonat moderním počítačem, lze provést i Turingovým strojem.

Třída *P* V

Funkce $f : \Sigma_0^* \to \Sigma_0^*$ je ***vyčíslitelná pomocí Turingova stroje*** M, jestliže pro všechna $x \in \Sigma_0^*$, x je vstup pro M, v případě ukončení výpočtu stroj zastaví s hodnotou $f(x)$ na jeho výstupní pásce.

Tedy Turingův stroj je přesná analogie počítače – každý výpočet, který lze vykonat moderním počítačem, lze provést i Turingovým strojem.

Ovšem v praxi je konstrukce Turingova stroje schopného i pouze jednoduchých výpočtů velmi časově náročná. Proto byl vyvinut soubor základních Turingových strojů, které provádí odpovídající úlohy.

Třída *P* V

Funkce $f : \Sigma_0^* \to \Sigma_0^*$ je ***vyčíslitelná pomocí Turingova stroje*** M, jestliže pro všechna $x \in \Sigma_0^*$, x je vstup pro M, v případě ukončení výpočtu stroj zastaví s hodnotou $f(x)$ na jeho výstupní pásce.

Tedy Turingův stroj je přesná analogie počítače – každý výpočet, který lze vykonat moderním počítačem, lze provést i Turingovým strojem.

Ovšem v praxi je konstrukce Turingova stroje schopného i pouze jednoduchých výpočtů velmi časově náročná. Proto byl vyvinut soubor základních Turingových strojů, které provádí odpovídající úlohy.

Konstruujeme-li pak složitý Turingův stroj, používáme strojů již dříve zkonstruovaných, podobně jako když používáme subrutiny v obvyklých počítačových programech.

Složitost I

Můžeme pak formálně definovat časovou složitost. Je-li M Turingův stroj, který zastaví pro všechny vstupy $x \in \Sigma_0^*$, ***časová složitost*** Turingova stroje M je funkce $t_M : \mathbf{Z}^+ \to \mathbf{Z}^+$ určená vztahem

$$t_M(n) = \max\{t : \text{existuje } x \in \Sigma_0^* \text{ tak, že } |x| = n \text{ a čas uběhlý strojem } M \text{ při vstupu } x \text{ je } t\}.$$

Složitost I

Můžeme pak formálně definovat časovou složitost. Je-li M Turingův stroj, který zastaví pro všechny vstupy $x \in \Sigma_0^*$, ***časová složitost*** Turingova stroje M je funkce $t_M : \mathbf{Z}^+ \to \mathbf{Z}^+$ určená vztahem

$$t_M(n) = \max\{t : \text{existuje } x \in \Sigma_0^* \text{ tak, že } |x| = n \text{ a čas uběhlý strojem } M \text{ při vstupu } x \text{ je } t\}.$$

Funkce f je vyčíslitelná v ***polynomiálním čase*** nebo má ***polynomiální složitost***, pokud existuje nějaký Turingův stroj M, který vypočte f a jistý polynom p tak, že $t_M(n) \leq p(n)$ pro všechna n.

Složitost I

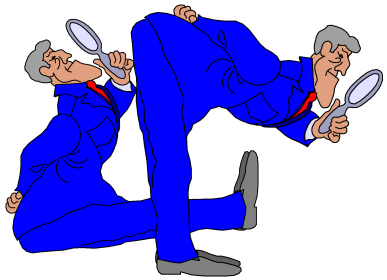
Můžeme pak formálně definovat časovou složitost. Je-li M Turingův stroj, který zastaví pro všechny vstupy $x \in \Sigma_0^*$, ***časová složitost*** Turingova stroje M je funkce $t_M : \mathbf{Z}^+ \to \mathbf{Z}^+$ určená vztahem

$$t_M(n) = \max\{t : \text{existuje } x \in \Sigma_0^* \text{ tak, že } |x| = n \text{ a čas uběhlý strojem } M \text{ při vstupu } x \text{ je } t\}.$$

Funkce f je vyčíslitelná v ***polynomiálním čase*** nebo má ***polynomiální složitost***, pokud existuje nějaký Turingův stroj M, který vypočte f a jistý polynom p tak, že $t_M(n) \leq p(n)$ pro všechna n.

V praxi většinou neuvažujeme s Turingovým modelem, ale pracujeme na mnohem vyšší úrovni.

O čem to bude



1. Příklady
2. *P*=polynomiální čas
3. *NP* = nedeterministický polynomiální čas
   - Úvod
   - Definice
   - NP-úplné problémy
   - NP-těžké problémy
   - VLSI obvody
   - Malé obvody
4. Splnitelnost - SATisfiability
5. Paměťová složitost
6. Náhodné algoritmy

Třída *NP* - I

Popišme si neformální ideu třídy *NP*.

Předpokládejme, že máte za úkol prodat velká složená čísla opravdu hodně zaměstnaným nákupčím.

S pomocí otroků pracujících neomezený počet hodin můžete sestavit seznam složených čísel $c_1, c_2, \ldots$ a abychom byli schopni prodávat tato čísla rychle, budete mít k dispozici odpovídající seznam faktorů $y_1, y_2, \ldots$ tak, že pokud má dojít k prodeji, vše, co musíte udělat, je dát číslo c_i dohromady s faktorem y_i a ověřit, že složené číslo c_i je dělitelné číslem y_i.

Pak y_i nazýváme ***certifikátem*** neprvočíselnosti čísla c_i, protože pak existuje při jeho použití algoritmus pracující v polynomiální době pro ověření, že c_i je složené.

Definice třídy *NP* - I

Neformálně můžeme tedy říci, že vlastnost ***náleží*** do *NP*, jestliže splnění této vlastnosti lze ověřit v polynomiální době s pomocí vhodného certifikátu.

Podejme nyní formální definici:

Buď Σ_0 nějaká konečná abeceda. Libovolnou podmnožinu L množiny Σ_0^* nazveme ***vlastností*** (jazykem). Řekneme pak, že $L \in NP$, jestliže existuje funkce $f : \Sigma^* \times \Sigma^* \to \{0, 1\}$ tak, že

1. $x \in L$ právě tehdy, když existuje $y \in \Sigma_0^*$ tak, že $f(x, y) = 1$,
2. výpočtová časová náročnost f je omezena polynomem v proměnné x.

Definice třídy *NP* - II

To lze přesněji přeformulovat následovně:

Pro $x, y \in \Sigma_0^*$ označme $x\ y$ řetězec začínající x, následovaný prázdným symbolem a poté následovaný y.

Jazyk $L \subseteq \Sigma_0^*$ bude v *NP*, pokud existuje Turingův stroj M a polynom $p(n)$ tak, že $T_M(n) \leq p(n)$ a pro každý vstup $x \in \Sigma_0^*$:

1. Pokud $x \in L$, pak existuje ***certifikát*** $y \in \Sigma_0^*$ tak, že $|y| \leq p(|x|)$ a M akceptuje vstupní řetězec $x\ y$.
2. Pokud $x \notin L$, pak, pro každý řetězec $y \in \Sigma_0^*$, M zamítne vstupní řetězec $x\ y$.

$P = NP$? - I

V pojmech teorie Turingových strojů, považujem y sdružené se vstupem x za ***certifikát*** relace patřit prvku x v L, a necháváme Turingův stroj pracovat v polynomiální době na vstupu sestávajícímu ze vstupu x a certikátu y.

Speciálně pak

$$P \subseteq NP,$$

považujeme-li P za soubor vlastností.

Otázka, zda $P = NP$ je pravděpodobně nejdůležitější otázkou v teoretické computer science.

NP-úplné problémy - I

Důležitou vlastností třídy *NP* je, že obsahuje "**nejtěžší vlastnosti**" tak, že kdybychom byli schopni vyřešit některou z těchto vlastností, byli bychom schopni rozhodnout každou z vlastností v *NP*.

Precizněji, řekneme, že vlastnost π_1 je ***polynomiálně redukovatelná*** na vlastnost π_2, pokud existuje funkce f z P tak, že x má vlastnost π_1 právě tehdy, když $f(x)$ má vlastnost π_2. Píšeme pak

$$\pi_1 \leq_p \pi_2.$$

Je snadno vidět, že pokud $\pi_1 \leq_p \pi_2$, implikuje existence polynomiálního algoritmu pro π_2 existenci polynomiálního algoritmu pro π_1.

NP-úplné problémy - II

Totiž, nechť $\mathcal{A}$ je algoritmus pro π_2, který pracuje v čase $t(n)$.

Je-li x nějaký vstup pro π_1 tak, že $|x| = n$, pak transformujme x na $f(x)$ a aplikujme algoritmus $\mathcal{A}$.

Protože transformace pracuje v polynomiálním čase, je $|f(x)|$ ohraničené nějakým polynomem $g(n)$. Tedy transformace f a algoritmus $\mathcal{A}$ pracují v čase ohraničeny nějakým polynomem.

Připomeňme následující tvrzení:

Věta 3.1

Existuje vlastnost $\pi \in NP$ tak, že libovolná jiná vlastnost $\pi' \in NP$ je polynomiálně redukovatelná na π.

Takovéto π se nazývá ***NP-úplný problém***.

NP-úplné problémy - III

Připomeňme, že máme k dispozici seznam více než 3000 *NP*-úplných problémů.

Přitom téměř každá vlastnost z *NP*, o které se neví, zda leží v *P*, je *NP*-úplná, ačkoliv máme k dispozici tvrzení, které říká, že pokud $NP \neq P$, pak $NP - P$ obsahuje problémy, jež nejsou *NP*-úplné.

Věta 3.2

$SAT \in$ NP-úplné problémy.

NP-těžké problémy - I

Řekneme, že problém π je *NP*-těžký, pokud existuje nějaká *NP*-úplná vlastnost π_0 tak, že pokud existuje polynomiální algoritmus pro π, existuje i polynomiální algoritmus pro π_0.

Ekvivalentně, funkce *f* se nazývá *NP*-těžká, pokud existuje *NP*-úplný jazyk *L* takový, že $L \leq_p f$, přičemž zde identifikujeme *L* s funkcí f_L a f_L je charakteristická funkce množiny *L*.

Tedy *NP*-těžká je minimálně stejně obtížná jakožto libovolný jazyk (vlastnost) z *NP* v tom smyslu, že polynomiální algoritmus pro výpočet takovéto funkce by nám garantoval polynomiální algoritmus pro libovolný jazyk (vlastnost) z *NP*.

NP-těžké problémy - II

Například rozhodovací problém klikovosti grafu, který se ptá, zda v daném grafu existuje klika alespoň o k vrcholech, je *NP*-těžký problém. Připomeňme, že klika grafu je každý maximální úplný podgraf grafu G (tj. každý úplný podgraf pro který platí, že žádný z jeho nadgrafů není úplný). Maximální číslo k, pro které existuje v G klika o k vrcholech nazýváme ***klikovostí grafu***.

Triviální důsledky této definice jsou následující tvrzení:

1. Pokud existuje polynomiální algoritmus pro každý *NP*-problém, je pak $NP = P$.
2. Je-li π_1 *NP*-těžký a $\pi_1 \leq_p \pi_2$, je i π_2 *NP*-těžký.
3. Každý *NP*-úplný problém je *NP*-těžký.

Obrácené tvrzení k 3 není pravdivé.

VLSI obvody - I

Věnujme se na chvíli zcela odlišnému modelu výpočtu, který zhruba odpovídá realizaci čipu nebo VLSI obvodu (very large scale integration – velmi vysoká integrace).
Kombinační obvod (***logický obvod***, ***hradlo***) je zařízení pro výpočet ***booleovských funkcí***.
Obvykle je reprezentován jakožto konečný orientovaný acyklický graf, jehož množina vrcholů se dělí na ***vstupní vrcholy***, ***vnitřní vrcholy*** neboli ***brány*** a jediný ***výstupní vrchol***.
Každý vstupní vrchol patří k právě jednomu z argumentů počítané booleovské funkce.
Každý z vnitřních vrcholů odpovídá vzájemně jednoznačně příslušné booleovské funkci dvou proměnných použité během výpočtu. Výstupní vrchol pak obsahuje výsledek výpočtu booleovské funkce na vstupních vrcholech.

VLSI obvody - II

Řekneme, že ***kombinační obvod*** C s n vstupními vrcholy *vypočte* booleovskou funkci $f(x_1, \ldots, x_n)$, pokud, v případě, že vstupním vrcholům je přiřazena po řadě posloupnost x_1, x_2, $\ldots$, x_n a tyto vstupní hodnoty jsou zpracovány vnitřními branami dle zřejmého pořadí indukovaného acyklickým uspořádáním grafu C, je hodnota výstupního vrcholu rovna $f(x_1, \ldots, x_n)$.

Velikost kombinačního obvodu C je počet vnitřních vrcholů a značí se c(C).

Je-li f booleovská funkce, je pak ***obvodová složitost*** funkce f definována jakožto minimální velikost kombinačního obvodu, který realizuje funkci f a označuje se jakožto c(f).

Příklad - I

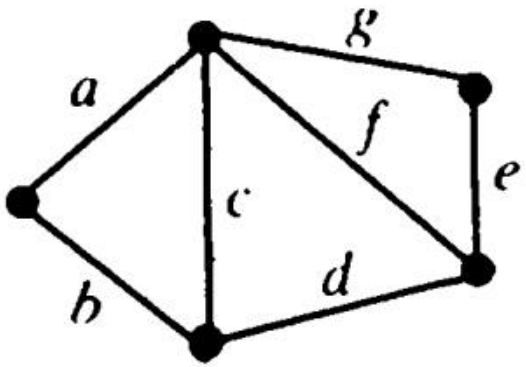
Příklad 3.3

(**Hamiltonovská kružnice**) *Uvažme následující otázku: Určete, zda graf G obsahuje Hamiltonovskou kružnici. Pro každou hodnotu n můžeme najít kombinační obvod C_n, jenž má $O(n^2)$ vstupních vrcholů (jeden pro každou možnou hranu), která dává na výstupu výsledek* TRUE *tehdy a jen tehdy, když vstupní graf G má hamiltonovskou kružnici. V současné době, bohužel, všechny známé takovéto kombinační obvody C_n mají exponenciální počet vrcholů.*

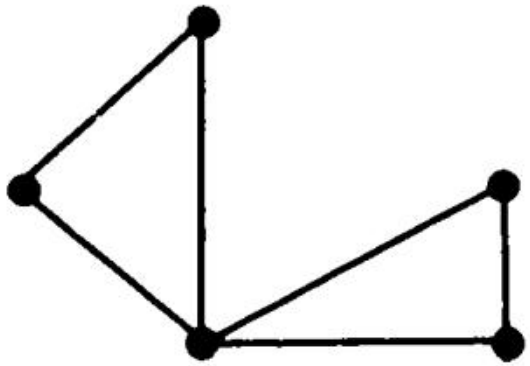
Hamiltonovský graf je graf, který lze projít takovou cestou, že každý jeho uzel je navštíven právě jednou s výjimkou uzlu výchozího, který je zároveň uzlem cílovým. Neboli – graf je hamiltonovský, právě když obsahuje kružnici, která prochází všemi jeho uzly (tzv. hamiltonovská kružnice).

Příklad - II

Příklad 3.3



G_1



G_2

Graf G_1 má Hamiltonovskou kružnici $(a, b, d, e, g))$, ale graf G_2 nemá žádnou Hamiltonovskou kružnici.

Malé obvody - I

Řekneme tedy, že vlastnost π má ***polynomiální obvodovou velikost*** neboli obsahuje pouze ***malé obvody***, pokud existuje posloupnost ***kombinačních obvodů*** $(C_n : 1 \leq n < \infty)$ a polynom p tak, že C_n rozhodne π na všech možných vstupech velikosti n a velikost $\mathrm{c}(C_n)$ splňuje

$$\mathrm{c}(C_n) \leq p(n) \qquad (1 \leq n < \infty).$$

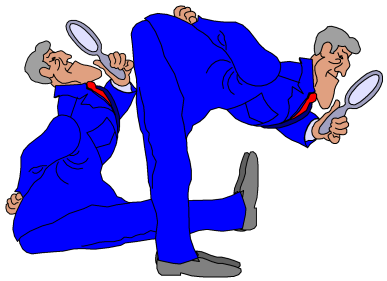
Poznamenejme, že platí:

- Je-li výpočet turingovsky vypočítatelný v polynomiálním čase, pak má malé kružnice.
- Obrácené tvrzení není pravdivé: existují nerekurzivní funkce, které nejsou vypočítatelné žádným turingovským strojem, ale mají malé kružnice.
- Má-li každý NP-těžký problém malé kružnice, má i každý NP-problém malé kružnice.

Malé obvody - II

V dalším budeme každý problém s malými kružnicemi považovat za "snadný" a šifrovací systém na něm založený nebude považován za bezpečný.

O čem to bude



1. Příklady
2. *P*=polynomiální čas
3. *NP* = nedeterministický polynomiální čas
4. Splnitelnost - SATisfiability
5. Paměťová složitost
6. Náhodné algoritmy

Splnitelnost - I

Klasickým případem rozhodovacího problému je booleovská splnitelnost.

Splnitelnost - I

Klasickým případem rozhodovacího problému je booleovská splnitelnost.

Booleovská funkce je funkce $f : \{0,1\}^n \to \{0,1\}$. Budeme interpretovat '1' jakožto pravda a '0' jakožto nepravda. Základní booleovské funkce jsou negace (NOT), konjunkce (AND) a disjunkce (OR).

Splnitelnost - I

Klasickým případem rozhodovacího problému je booleovská splnitelnost.

Booleovská funkce je funkce $f : \{0,1\}^n \to \{0,1\}$. Budeme interpretovat '1' jakožto pravda a '0' jakožto nepravda. Základní booleovské funkce jsou negace (NOT), konjunkce (AND) a disjunkce (OR).
Je-li x booleovská proměnná, pak ***negace*** x je

$$\neg x \quad = \quad \begin{cases} 1, & \text{pokud je } x \text{ nepravda,} \\ 0, & \text{jinak.} \end{cases}$$

Splnitelnost - I

Klasickým případem rozhodovacího problému je booleovská splnitelnost.

Booleovská funkce je funkce $f : \{0,1\}^n \to \{0,1\}$. Budeme interpretovat '1' jakožto pravda a '0' jakožto nepravda. Základní booleovské funkce jsou negace (NOT), konjunkce (AND) a disjunkce (OR).
Je-li x booleovská proměnná, pak ***negace*** x je

$$\neg x \quad = \quad \begin{cases} 1, & \text{pokud je } x \text{ nepravda,} \\ 0, & \text{jinak.} \end{cases}$$

Literál je booleovská proměnná nebo její negace. ***Konjunkce*** posloupnosti literálů $x_1, \ldots, x_n$ je

$$x_1 \wedge x_2 \wedge \cdots \wedge x_n \quad = \quad \begin{cases} 1, & \text{pokud všechna } x_i \text{ jsou pravdivá,} \\ 0, & \text{jinak.} \end{cases}$$

Splnitelnost - II

Disjunkce posloupnosti literálů $x_1, \ldots, x_n$ je

$$x_1 \vee x_2 \vee \cdots \vee x_n \quad = \quad \begin{cases} 1, & \text{pokud některé } x_i \text{ je pravdivé,} \\ 0, & \text{jinak.} \end{cases}$$

Splnitelnost - II

Disjunkce posloupnosti literálů $x_1, \ldots, x_n$ je

$$x_1 \vee x_2 \vee \cdots \vee x_n \quad = \quad \begin{cases} 1, & \text{pokud některé } x_i \text{ je pravdivé,} \\ 0, & \text{jinak.} \end{cases}$$

Booleovská funkce f je v ***konjunktivní normální formě*** (zkráceně CNF), pokud

$$f(x_1, \ldots, x_n) \quad = \quad \bigwedge_{k=1}^{m} C_k,$$

kde každá klauzule C_k je disjunkce literálů.

Splnitelnost - II

Disjunkce posloupnosti literálů $x_1, \ldots, x_n$ je

$$x_1 \vee x_2 \vee \cdots \vee x_n \quad = \quad \begin{cases} 1, & \text{pokud některé } x_i \text{ je pravdivé,} \\ 0, & \text{jinak.} \end{cases}$$

Booleovská funkce f je v ***konjunktivní normální formě*** (zkráceně CNF), pokud

$$f(x_1, \ldots, x_n) \quad = \quad \bigwedge_{k=1}^{m} C_k,$$

kde každá klauzule C_k je disjunkce literálů.

Pravdivostní přiřazení pro booleovskou funkci $f(x_1, \ldots, x_n)$ je výběr hodnot $\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$ pro její proměnné.

Splnitelnost - III

Splněné pravdivostní přiřazení je výběr hodnot $\mathbf{x} = (x_1, \ldots, x_n) \in \{0, 1\}^n$ tak, že $f(\mathbf{x}) = 1$.

Splnitelnost - III

Splněné pravdivostní přiřazení je výběr hodnot $\mathbf{x} = (x_1, \ldots, x_n) \in \{0, 1\}^n$ tak, že $f(\mathbf{x}) = 1$.

Pokud takové pravdivostní přiřazení existuje, řekneme, že funkce f je ***splnitelná***.

Splnitelnost - III

Splněné pravdivostní přiřazení je výběr hodnot $\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$ tak, že $f(\mathbf{x}) = 1$.

Pokud takové pravdivostní přiřazení existuje, řekneme, že funkce f je ***splnitelná***.

Booleovská splnitelnost (SAT) je následující rozhodovací problém.

SAT
Vstup: booleovská funkce $f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m} C_k$ v CNF.
Otázka: je $f(x_1, \ldots, x_n)$ splnitelná?

Splnitelnost - IV

Uvažme přirozené zakódování tohoto problému. Budeme pracovat nad abecedou $\Sigma = \{*, 0, 1, \vee, \wedge, \neg\}$, přičemž zakódujem proměnnou x_i pomocí binární reprezentace i. Literál $\overline{x_i}$ budeme kódovat přidáním symbolu $\neg$ na začátek. Evidentně pak můžeme zakódovat CNF formuli, $f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m}$ přirozeným způsobem nad abecedou Σ.

Splnitelnost - IV

Uvažme přirozené zakódování tohoto problému. Budeme pracovat nad abecedou $\Sigma = \{*, 0, 1, \vee, \wedge, \neg\}$, přičemž zakódujem proměnnou x_i pomocí binární reprezentace i. Literál $\overline{x_i}$ budeme kódovat přidáním symbolu $\neg$ na začátek. Evidentně pak můžeme zakódovat CNF formuli, $f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m}$ přirozeným způsobem nad abecedou Σ.

Např. formuli

$$f(x_1, \ldots, x_5) = (x_1 \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_2) \wedge (\overline{x_3} \vee x_5),$$

zakódujeme jako

‘$1 \vee 100 \wedge 11 \vee \neg 101 \vee 10 \wedge \neg 11 \vee 101$’.

Splnitelnost - V

Protože žádná klauzule nemůže obsahovat více než $2n$ literálů, bude velikost vstupu CNF formule o n proměnných a m klauzulích $O(mn \log n)$.

Splnitelnost - V

Protože žádná klauzule nemůže obsahovat více než $2n$ literálů, bude velikost vstupu CNF formule o n proměnných a m klauzulích $O(mn \log n)$.

Fundamentální věta teorie složitosti říká, že SAT $\in$ NP-úplný.

Důležitým podproblémem problému SAT je tzv. k-SAT, for $k \geq 1$.

Splnitelnost - V

Protože žádná klauzule nemůže obsahovat více než $2n$ literálů, bude velikost vstupu CNF formule o n proměnných a m klauzulích $O(mn \log n)$.

Fundamentální věta teorie složitosti říká, že SAT $\in$ NP-úplný.

Důležitým podproblémem problému SAT je tzv. k-SAT, for $k \geq 1$.

k-SAT
Vstup: booleovská funkce f v CNF s nejvýše k literály v každé klauzuli.
Otázka: je $f(x_1, \ldots, x_n)$ splnitelná?

Splnitelnost - VI

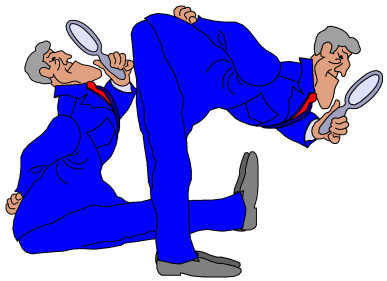
Evidentně je problém 1-SAT snadný. Každé splněné pravdivostní přiřazení pro f v tomto případě musí zajistit, že každý literál obsažený v f musí být pravda.

Tedy f je splnitelné právě tehdy, když neobsahuje zároveň literál a jeho negaci. To samozřejmě snadno ověříme v polynomiální době a tedy 1-SAT $\in P$.

Podstatně náročnější je důkaz, že 2-SAT $\in P$.

Ale už 3-SAT $\in NP$-úplný.

O čem to bude



1. Příklady
2. *P*=polynomiální čas
3. *NP* = nedeterministický polynomiální čas
4. Splnitelnost - SATisfiability
5. Paměťová složitost
   - Polynomiální paměť
   - Doplňky jazyků
6. Náhodné algoritmy

Paměťová složitost - I

Doposud jsme uvažovali jakožto jediný výpočetní zdroj čas. Jiným zdrojem, který omezuje naši schopnost provádět výpočty je paměť. Zavedeme nyní potřebné definici, abychom se mohli krátce věnovat paměťové složitosti deterministického Turingova stroje.

Zastaví-li Turingův stroj při vstupu $x \in \Sigma_0^*$, pak ***paměť použitá*** při vstupu x je počet různých čtverců pásky, které byly použity čtecí-zapisovací hlavicí Turingova stroje během jeho výpočtu. Toto číslo označíme jako $s_M(x)$.

Zastaví-li Turingův stroj M pro každý vstup $x \in \Sigma_0^*$, pak ***paměťová složitost*** Turingova stroje M je funkce $S_M : \mathbf{N} \to \mathbf{N}$ definovaná jakožto

$$S_M(n) = max\{s \mid \text{existuje } x \in \Sigma_0^n \text{ tak, že } s_M(x) = s\}.$$

Paměťová složitost - II

Nejdůležitější třídou paměťové složitosti je třída jazyků, které mohou být rozhodnuty v ***polynomiální paměti***

$$PSPACE = \{L \subseteq \Sigma_0^* \quad | \quad \text{existuje Turingův stroj } M, \text{ který rozhodne } L, \text{ a polynom } p(n) \text{ tak, že } S_M(n) \leq p(n) \text{ pro všechna } n \geq 1\}.$$

Je zřejmé, že paměť je hodnotnější zdroj než čas v tom smyslu, že množství paměti použité při výpočtu je vždy ohraničeno shora množstvím času, který výpočet zabere.

Paměťová složitost - III

Tvrzení 5.1

Je-li jazyk L rozhodnutelný v čase $f(n)$, pak je L rozhodnutelný v paměti $f(n)$.

Důkaz.

Počet různých čtverců pásky, které byly použity čtecí-zapisovací hlavicí libovolného Turingova stroje během jeho výpočtu nemůže převýšit počet kroků, které Turingův stroj provede. ■

Důsledek 5.2

$P \subseteq PSPACE$.

Paměťová složitost - IV

Další důležitou třídou paměťové složitosti je třída jazyků, které mohou být rozhodnuty v ***exponenciálním čase***

$$EXP = \{L \subseteq \Sigma_0^* \mid \text{existuje Turingův stroj } M, \text{ který rozhodne } L, \text{ a polynom } p(n) \text{ tak, že } T_M(n) \leq 2^{p(n)} \text{ pro všechna } n \geq 1\}.$$

Platí ale, že při exponenciálním množství času můžeme spočítat cokoliv, co lze spočítat pomocí polynomiální paměti.

Věta 5.3

$P \subseteq PSPACE \subseteq EXP$

Paměťová složitost - V

Důkaz.

Stačí ověřit $PSPACE \subseteq EXP$.

Předpokládejme, že jazyk $L \in PSPACE$. Pak existuje polynom $p(n)$ a Turingův stroj M takový, že M rozhodne L a zastaví po nejvýše $p(|x|)$ čtvercích pásky při vstupu $x \in \Sigma_0^n$.

Základní myšlenkou důkazu je, že protože M zastaví, nemůže nikdy opakovat stejnou konfiguraci dvakrát (přičemž konfigurace sestává ze stavu Turingova stroje, pozice čtecí-zapisovací hlavice a obsahu pásky). V opačném případě by vznikla nekonečná smyčka a tedy by Turingův stroj nikdy nezastavil. ▮

Paměťová složitost - VI

Pokračování.

Přesněji, uvažujme vstup $x \in \Sigma_0^n$. Pokud $|\Sigma| = m$ a $|\Gamma| = k$, pak v každém bodě výpočtu může být momentální konfigurace Turingova stroje popsána následovně:

(i) současným stavem Turingova stroje,

(ii) pozicí čtecí-zapisovací hlavice,

(iii) obsahem pásky.

Máme tedy k možností pro (i) a, protože výpočet použije nejvýše $p(n)$ různých čtverců pásky, máme nejvýše $p(n)$ možností pro (ii).

Protože každý čtverec pásky obsahuje nějaký symbol z abecedy Σ a obsah čtverce pásky, který nebyl použit čtecí-zapisovací hlavicí, se během výpočtu nezmění, máme pak $m^{p(n)}$ možností pro (iii). ∎

Paměťová složitost - VII

Pokračování.

Celkem tedy máme $kp(n)m^{p(n)}$ konfigurací pro Turingův stroj M během jeho výpočtu při vstupu x délky n.
Může se některá z těchto konfigurací opakovat? Evidentně nikoliv, protože kdyby se opakovala, Turingův stroj by se dostal do smyčky a nikdy by nezastavil. Tudíž

$$t_M(x) \leq kp(n)m^{p(n)}.$$

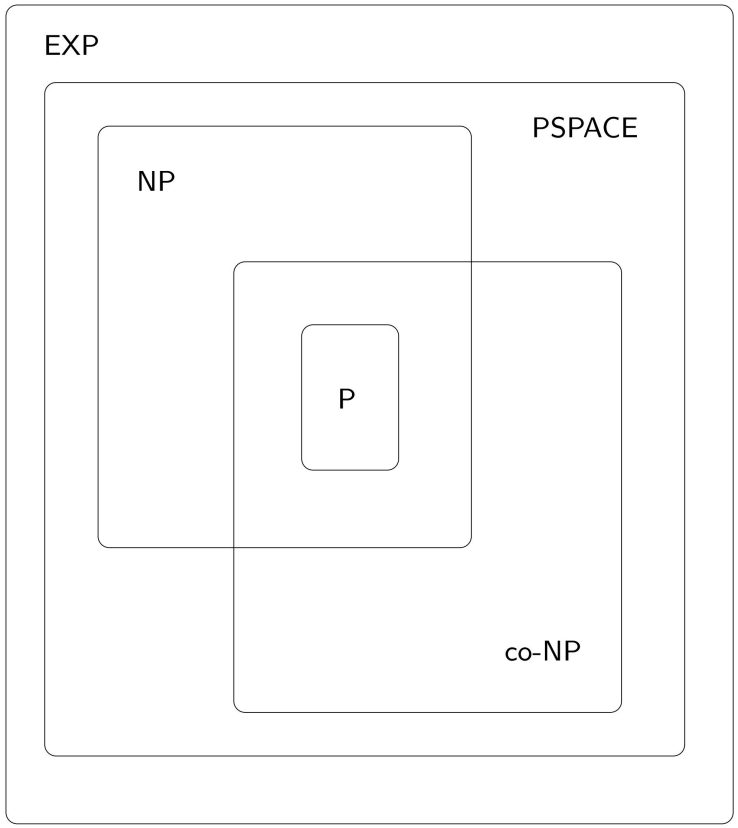
Uvažme polynom $q(n)$ splňující

$$\log k + \log p(n) + p(n)\log m \leq q(n).$$

Odtud $t_M(x) \leq 2^{q(n)}$. Pak L je rozhodnutelný v čase $2^{q(n)}$ a tedy $L \in EXP$. ■

Paměťová složitost - VII

Je známo, že $P \neq EXP$. Ale zda platí, že $PSPACE = EXP$, je jedním z hlavních problémů teorie složitosti. Kdyby výše uvedené platilo, bylo by $P \neq PSPACE$, což se neví.



Doplňky jazyků - I

Je-li $L \subseteq \Sigma_0^*$ jazyk, pak ***komplement*** jazyka L je množina

$$L^c = \{x \in \Sigma_0^* \mid x \notin L\}.$$

Je-li C třída složitosti, pak třída ***komplementů jazyků*** z C se označuje jako

$$co - C = \{L \subseteq \Sigma_0^* \mid L^c \in C\}.$$

Nejdůležitějším příkladem takovéto třídy je $co - NP$, tj. soubor komplementů jazyků z NP.

Doplňky jazyků - II

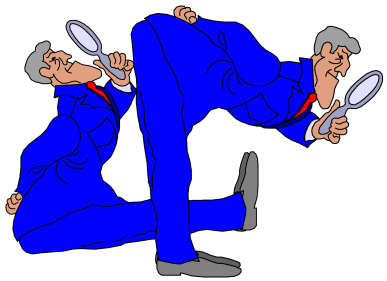
Podle naší definice jazyk $L \subseteq \Sigma_0^*$ leží v $co - NP$ tehdy a jen tehdy, když existuje Turingův stroj M a polynom $p(n)$ tak, že $T_M(n) \leq p(n)$ a pro každý vstup $x \in \Sigma_0^*$ máme:

(i) pokud $x \notin L$, pak existuje ***certifikát*** $y \in \Sigma_0^*$ takový, že $|y| \leq p(|x|)$ a M akceptuje vstupní řetězec xy,

(ii) pokud $x \in L$, pak pro každý řetězec $y \in \Sigma_0^*$ Turingův stroj M zamítne vstupní řetězec xy.

V případě jazyka ležícího v P máme Turingův stroj M, který v polynomiálním čase rozhodne L. Znegujeme-li výstup našeho Turingova stroje, obdržíme Turingův stroj M_1, který v polynomiálním čase rozhodne L^c. Tedy $P = co - P$. Pro NP už výše uvedené neplatí. Otázka, zda $NP = co - NP$ je pravděpodobně druhým nejdůležitějším problémem teorie složitosti, po otázce, zda $P = NP$.

O čem to bude



1. Příklady
2. *P*=polynomiální čas
3. *NP* = nedeterministický polynomiální čas
4. Splnitelnost - SATisfiability
5. Paměťová složitost
6. Náhodné algoritmy

Náhodné algoritmy - I

Algoritmus, který má pravděpodobnost chyby méně, než 2^{-100} a který během minuty je schopen identifikovat číslo $2^{400} - 593$ jakožto největší prvočíslo menší než 2^{400}, má bezprostřední praktický a estetický důsledek. Takovými jsou například algoritmy pro testování prvočíselnosti od Rabina a dále od Solovaye a Strassena.

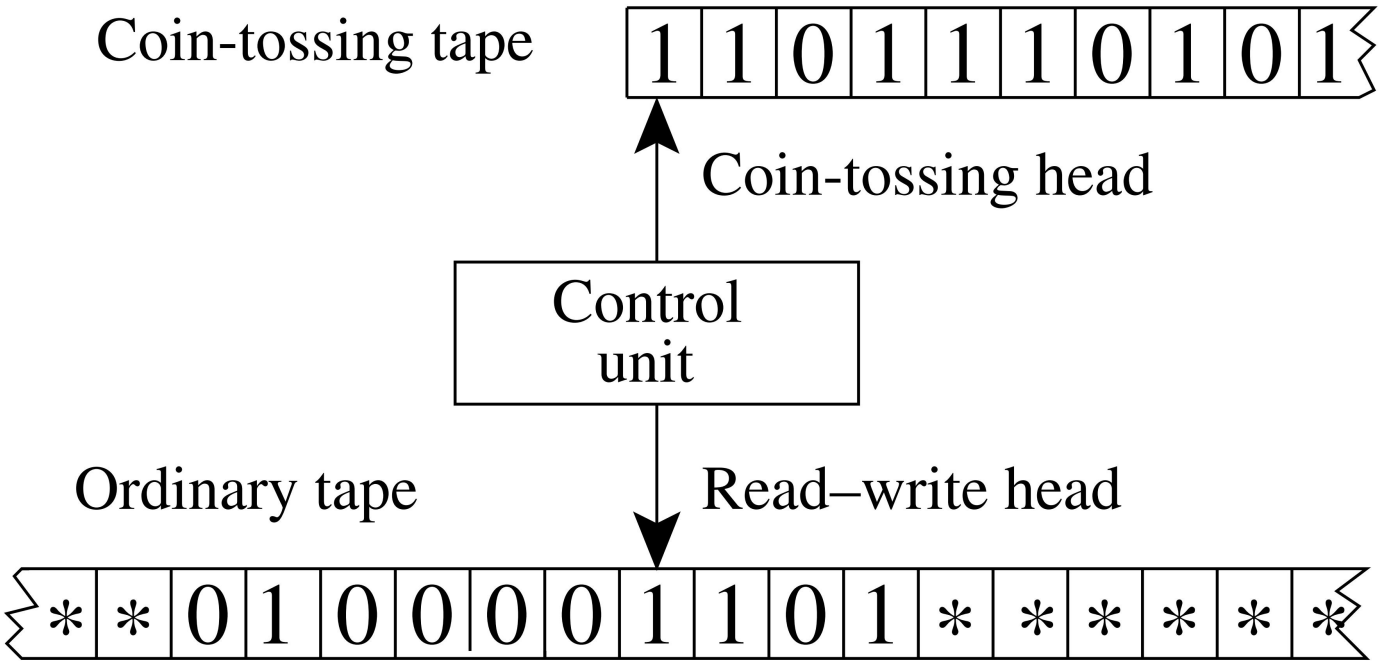
Náhodné algoritmy - II

Nejprve ilustrujme ideu náhodného algoritmu na příkladě: Předpokládejme, že máme polynomiální výraz v n proměnných, řekněme $f(x_1, \ldots, x_n)$ a že si přejeme ověřit, zda je polynom f identicky nulový. Ověřit to analyticky je neskutečné počítání.

Předpokládejme místo toho, že jsme vygenerovali náhodný vektor $(r_1, \ldots, r_n)$ a vyčíslili $f(r_1, \ldots, r_n)$. Pokud $f(r_1, \ldots, r_n) \neq 0$, víme že f je nenulový. Pokud $f(r_1, \ldots, r_n) = 0$, je buď f identicky nulové nebo jsme měli obrovské štěstí s výběrem $(r_1, \ldots, r_n)$.

Proveďme tyto kroky několikrát a pokud vždy obdržíme nulu, můžeme tvrdit, že f je identicky nula. Pravděpodobnost toho, že jsme udělali chybu, je zanedbatelná.

Náhodné algoritmy - III



Náhodné algoritmy - IV

Je-li π výpočetní problém a x je vstup nebo instance π, řekneme, že ***náhodný algoritmus*** pro řešení π postupuje následovně. V jistých okamžicích pro řešení instance x problému π algoritmus provede náhodné rozhodnutí.
S vyjímkou těchto náhodných rozhodnutí je algoritmus čistě deterministický.

Jazyk L je ***náhodně rozhodnutelný*** v polynomiálním čase neboli leží ve třídě RP, pokud existuje polynom f a polynomiální algoritmus, který pro každý vstup x a každý možný certifikát y délky $f(|y|)$ vypočte hodnotu $\nu(x, y) \in \{0, 1\}$ tak, že

1. $x \notin L$ implikuje $\nu(x, y) = 0$ pro všechna y.
2. $x \in L$ implikuje $\nu(x, y) = 1$ pro alespoň polovici všech možných certifikátů y.

Náhodné algoritmy - V

Evidentně

$$RP \subseteq NP$$

a

$$P \subseteq RP.$$

Abychom podali formální definici pravděpodobnostního polynomiálního časového algoritmu, zavedeme nový typ Turingova stroje.

Pravděpodobnostní Turingův stroj (zkráceně PTM) je deterministický Turingův stroj, který navíc obsahuje další pásku nazvanou ***páska házející mince***, která obsahuje nekonečnou posloupnost rovnoměrně rozdělených nezávislých náhodných bitů. Tato páska má hlavu jen pro čtení nazvanou ***hlava pro házení mincí***.

Náhodné algoritmy - VI

Stroj provádí výpočty podobně jako deterministický Turingův stroj kromě toho, že hlava pro házení mincí může číst bit z pásky házející mince v jediném kroku.
Přechodová funce nyní závisí nejen na aktuálním stavu a symbolu v aktuálním čtverci běžné pásky, ale také na náhodném bitu v současné době snímanén čtverci pomocí hlava pro házení mincí.
Přechodová funce nyní řekne pravděpodobnostnímu Turingovu stroji čtyři věci: nový stav; nový symbol, který má být zapsán do současnáho čtverce běžné pásky; pohyb vlevo nebo vpravo hlavy na čtení i zápis a pohyb vlevo nebo vpravo hlavy pro házení mincí. (Všimněme si, že páska házející mince je nekonečná pouze v jednom směru, hlava pro házení mincí se nesmí posunout z konce pásky).

Náhodné algoritmy - VII

Protože výpočet PTM při vstupu $x \in \Sigma_0^*$ nezáleží nejen na x, ale také na náhodných bitech použitých při jeho výpočtu, je doba běhu PTM M je náhodná proměnná: $t_M(x)$. Ve skutečnosti, zda se PTM zastaví na určitém vstupu, je samo o sobě náhodná proměnná.
Řekneme, že PTM ***je zastavující***, pokud zastaví po konečně mnoha krocích na každém vstup $x \in \Sigma_0^*$ bez ohledu na náhodné bity použitých při jeho výpočtu.
Časová složitost zastavujícího PTM M je $T_M : \mathbb{N} \to \mathbb{N}$ a je definována takto,

$$T_M(n) = max\{t \mid \text{ existuje } x \in \Sigma_0^* \text{ tak, že } Pr[t_M(x) = t] > 0\}.$$

V podstatě to odpovídá maximální době zastavení, která nastane s nenulovou pravděpodobností.

Náhodné algoritmy - VIII

Budeme říkat, že PTM M má ***polynomiální dobu běhu***, pokud existuje polynom $p(n)$ tak, že $T_M(n) \leq p(n)$, pro každé $n \in \mathbb{N}$. Takže dle definice je jakýkoli PTM s polynomiální dobou běhu zastavující.

Nyní můžeme korektně definovat třídu jazyků ***rozhodnutelných v náhodném polynomiálním čase neboli RP***. Jazyk L patří do RP tehdy a jen tehdy, pokud existuje PTM M s polynomiální dobou běhu tak, že při jakémkoliv vstupu $x \in \Sigma_0^*$ platí:

(I) je-li $x \in L$, pak platí $Pr[M \text{ akceptuje } x] \geq 1/2$;

(II) jestliže $x \notin L$, pak platí $Pr[M \text{ akceptuje } x] = 0$.

Náhodné algoritmy - IX

Pokud jazyk patří do RP, můžeme snížit pravděpodobnost mylného odmítnutí správného vstup opakováním výpočtu. Náš další výsledek ukazuje, že opakováním výpočtu můžeme polynomiálně snížit pravděpodobnost chyby podstatným způsobem.

Tvrzení 6.1

Když L patří do RP a $p(n) \geq 1$ je polynom, pak existuje PTM M s polynomiální dobou běhu tak, že při jakémkoliv vstupu $x \in \Sigma_0^*$ platí:

(I) *je-li $x \in L$, pak platí $Pr[M \text{ akceptuje } x] \geq 1 - 2^{-p(n)}$;*

(II) *jestliže $x \notin L$, pak platí $Pr[M \text{ akceptuje } x] = 0$.*