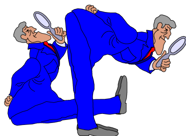
5. Výpočetní složitost

Jan Paseka

Ústav matematiky a statistiky
Masarykova univerzita

18. října 2021

O čem to bude



1 Příklady
- Úvod
- Násobení
- Permanenty
- Třídění
- Prvočíselnost
- Největší společný dělitel a Euklidův algoritmus

2 P=polynomiální čas

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 $x = x_1 \ldots x_n$ a $y = y_1 \ldots y_n$, 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

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

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

$$T(n) \leq An^{\log 3} + Bn,$$

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

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, který potřebuje

$$O(nlog_2 7) = O(n^{2.81\cdots})$$

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

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

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 $a_1, \dots, a_n$, 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 $O((logn)^{12})$. *Následně Lenstra a Pomerance představili verzi testu, která běží v čase* $O((logn)^6)$.

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

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

pak lze zjistit jejich největší společný dělitel následovně

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

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

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

pak lze zjistit jejich největší společný dělitel následovně

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

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:*

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

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

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

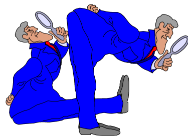
kde φ je zlatý řez $\frac{1}{2}(1+\sqrt{5})$.

Příklady XIII

Příklad 1.5 (*Euklidův algoritmus* - pokračování)

Ted 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

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.

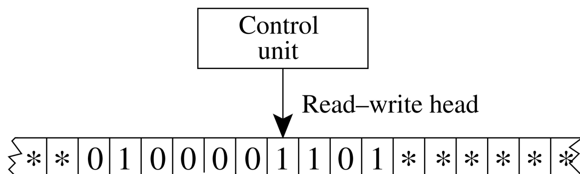
Třída P III

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

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



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 (čtení, 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 $f : \Sigma_0^* \rightarrow \Sigma_0^*$ 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

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

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.