VYPOČITATELNOST A VÝPOČTOVÁ SLOŽITOST

„computational problem solving“

Problém: řazení

Algoritmus:
- -výběrem
- -vkládáním
- -bublinkové
- -Shellovo
- -haldou
- -slučováním
- -dělením

Datová struktura:
- - pole
- -spojový seznam

Program = Algoritmus + Datová struktura
(Niklaus Wirth)

Analýza algoritmů
- ***-O(N^2)***, ***O($N^{1.25}$), O($N \log_2 N$), ...???***
- -může pro problém řazení existovat asymptoticky lepší algoritmus ???

„Beside merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, algorithm has ...“

(D. Knuth: The Art of Computer Programming, Volume 1, p.4)

- algoritmus řeší problém

- víme co je algoritmus

- existuje pro (nalezení) řešení daného problému algoritmus?
- jaký algoritmus pro (nalezení) řešení daného problému je „nejlepší“?
- jak složitý/těžký je problém

- umíme formalizovat pojem **problém**?

Problém je binární relace nad množinou vstupů a množinou výstupů.

Kartézský součin množin

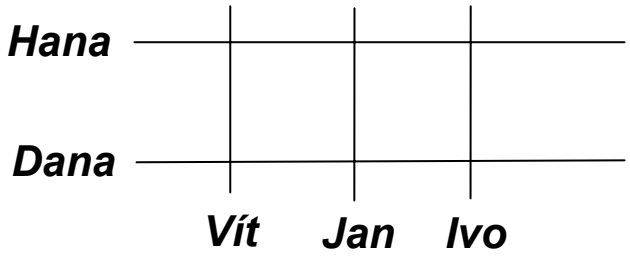
A x B

množina, která obsahuje všechny uspořádané dvojice, ve kterých je první položka prvkem množiny ***A*** a druhá položka je prvkem množiny ***B***

A = { Vít, Jan, Ivo }

B = { Hana, Dana }

A x B

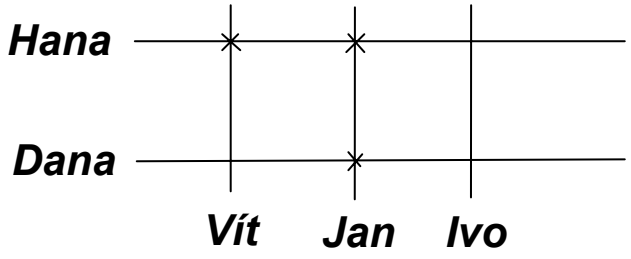


R x R kartézský souřadnicový systém

Binární relace

libovolná podmnožina kartézského součinu dvou množin

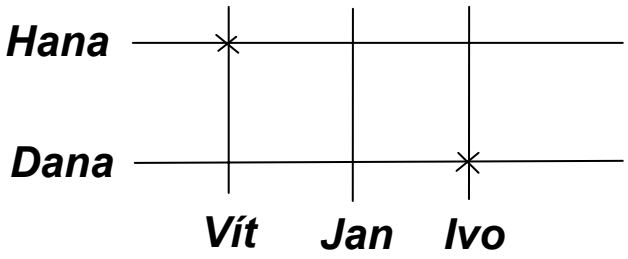
relace *tancoval s*



A	**B**
Vít	***Hana***
Jan	***Dana***
Jan	***Hana***

zobrazení/funkce

relace *on miluje ji*



Abstrakce (formalizace) problému

Abstraktní problém Q definujeme jako binární relaci nad množinou instancí problému I a množinou řešení S.

(Problém je binární relace nad množinou vstupů a množinou výstupů.)

Problém řazení :

vstup (genericky) – posloupnost
vstup (specificky) – konkrétní posloupnost/instance

výstup - řešení

I - všechny (konečné) posloupnosti
S - všechny uspořádané posloupnosti

I	S
1	1
2	2
...	...
1,1	1,1
1,2	1,2
...	...
2,1	1,2
2,2	2,2
...	...
...	...

- relace je funkce

Problém: nalezení nejkratší cesty v grafu

Instance: graf a dva jeho vrcholy
Řešení: posloupnost vrcholů na nejkratší cestě

Relace není funkce

Algoritmická řešitelnost problémů

Existuje pro každý problém algoritmus?

- zúžíme množinu všech problémů

- množina řešení: S = {ano,ne}
- relace je funkce
- rozhodovací problémy

obecný problém **vs.** (odpovídající) **rozhodovací** problém

problém řazení - odpovídající rozhodovací problém

I	S
1	ano
2	ano
...	...
1,1	ano
1,2	ano
...	...
2,1	ne
2,2	ano
...	...
...	...

známe-li řešení rozhodovacího problému umíme řešit odpovídající obecný problém (možná neefektivně, ale umíme)

Počítač

Instance: vstup programu, jeho kódování můžeme interpretovat jako celé číslo (možná velmi veliké)

Řešení: výstup program, můžeme kódovat 1=ano, 0=ne

Rozhodovací problém je funkce $f: \mathrm{N} \rightarrow \{0, 1\}$

Problém: *Je zadané přirozené číslo liché?*

I/$n \in$ N	**1**	**2**	**3**	**4**	**5**	**6**	**...**
S/*f(n)*	**1**	**0**	**1**	**0**	**1**	**0**	...

Algoritmy:

```
int jeLiche1(int n) {
  return n%2;
}

int jeLiche2(int n) {
  for( ; n > 1; i-=2)
}
```

Existuje (opět) víc než jeden algoritmus řešení problému!

Ať existuje problém **neřešitelný** JVM.

Je **řešitelný** v jiném jazyku ???

Jiné vyjádření algoritmů:

Alan Turing: Turingovy stroje
Alonzo Church: *lambda*-kalkulus

Churchova – Turingova teze: každý algoritmus může být vykonán Turingovým strojem

Každý program v konvenčních programovacích jazycích může být transformován na **Turingův stroj** a naopak.

Konvenční programovací jazyky, a také Java, jsou dostatečné pro vyjádření jakéhokoliv algoritmu.

Churchova – Turingova teze **nemůže** být dokázána.

Může být vyvrácena, bude-li objevena metoda, akceptována jako algoritmus, který nebude možno vykonat Turingovým strojem.

Existuje-li tedy rozhodovací problém, pro jehož řešení **neexistuje program** například v Javě, potom je tento problém **algoritmicky neřešitelný** a naopak.

Všechny rozhodovací programy v Javě jako celá čísla:
P1, P2, ..., Pn ... .

Vstup: interpretovaný jako celé číslo 1, 2, ..., k, ...

Výstup: Pn*(k)* ∈ {0, 1}.

	1	2	...	k	...
P1	P1(1)	P1(2)	...	P1(k)	...
P2	P2(1)	P2(2)	...	P2(k)	...
...	...	...	...	...	...
Pn	Pn(1)	Pn(2)	...	Pn(k)	...
...	...	...	...	...	...

Existuje-li pro rozhodovací problém *f:* N → *{0, 1}* algoritmus, potom musí existovat řádek Px , Px*(k) = f(k), k = 1, 2, ...*

`jeLiche1()` a **`jeLiche2()`**.

	1	2	...	k	...
P1	P1(1)	P1(2)	...	P1(k)	...
P2	P2(1)	P2(2)	...	P2(k)	...
...	...	...	...	...	...
jeLiche1	1	0	...	...	...
...	...	...	...	...	...
jeLiche2	1	0	...	...	...
...	...	...	...	...	...
Pn	Pn(1)	Pn(2)	...	Pn(k)	...
...	...	...	...	...	...

Ať problém ***D*** je definován:

$\boldsymbol{D}(k) = 1$ je-li $\mathrm{Pk}(k) = 0$
0 je-li $\mathrm{Pk}(k) = 1$

Nemůže být vypočten žádným programem Pk !!!

P1, protože $D(1) \neq \mathrm{P1}(1)$
P2, protože $D(2) \neq \mathrm{P2}(2)$
...
Pn, protože $D(n) \neq \mathrm{Pn}(n)$
...

Pro rozhodovací problém *D*, neexistuje v Javě žádný program a tedy ani žádný algoritmus.

Existují problémy algoritmicky neřešitelné (nerozhodnutelné).

První algoritmicky nerozhodnutelný problém ukázal v roce 1936 Alan Turing - **halting problem, problém zastavení**

Vstup: Program **P** v Javě reprezentován jako celé číslo a jeho vstup reprezentován také jako celé číslo

Výstup: 1 když se program **P** zastaví , 0 když se program **P** nezastaví

Existuje v Javě program, který vypočítá (řeší) problém zastavení?

J1, J2, ..., Jn, ... **všechny** programy v Javě
1, 2, 3, ... reprezentace vstupů

prvek (Jn, k) = výstup Jn pro vstup k / ? když nezastaví

	1	2	3	4	...
J1	5	4	?	8	...
J2	?	?	7	3	...
J3	4	2	6	?	...
J4	5	9	?	?	...
...	...	...	...	...	...
D	6	0	7	0	...

Nechť existuje algoritmus (program) pro rozhodnutí zda se program Jn pro vstup *k* zastaví nebo ne.

Potom existuje program **D:**

výstup programu D*(k)* = Jk*(k)* + 1, když zastaví
0, když Jk*(k)* nezastaví

```
int D (int Jk, int k) {
  if (zastavi (Jk, k)) return (Jk(k) + 1);
  else return 0;
}
```

- D **musí** být jedním z programů J1, J2, ..., Jn, ...
- **nemůže**, protože jeho výstup D*(k)* je různý od Jk*(k)*

spor

- jediný předpoklad byl existence metody
  `boolean zastavi(int Jk, int k)`

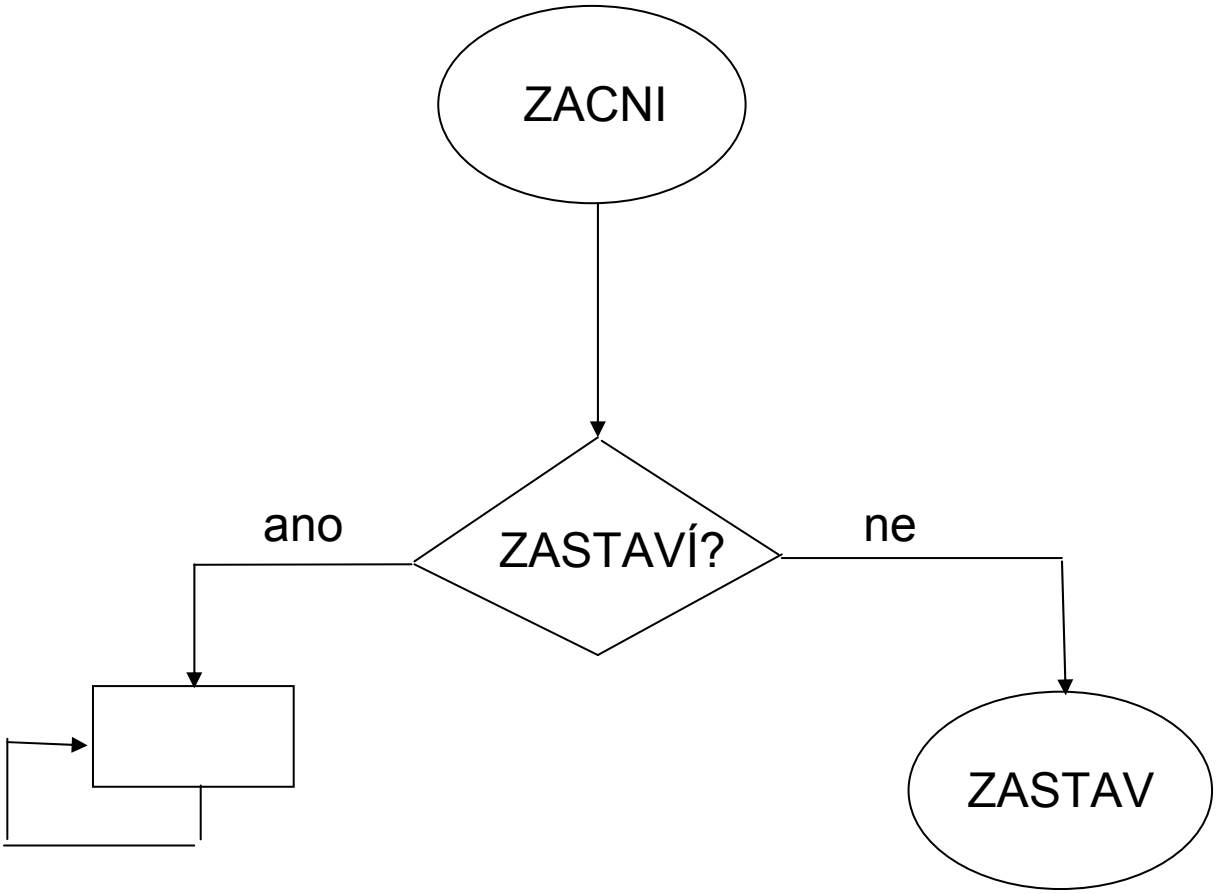
Pro problém zastavení tedy **neexistuje** v Javě program, a tím ani žádný algoritmus a problém zastavení je **algoritmicky nerozhodnutelný**.

Jiný způsob:

```
boolean zastavi(int J, int w) {
  return //program J pro vstup w se zastavi
}

// true – J(w) zastaví
// false - J(w) nezastaví


void cyklujKdyzZastavi(int J, int w) {
  if (zastavi(J, w))
    for ( ; ; );
}
```



Uvažujme program, který volá metodu **`cyklujKdyzZastavi()`** s parametry jenž jsou její celočíselné reprezentace.

```
public static void main(String args[]) {
  cyklujKdyzZastavi((int)cyklujKdyzZastavi,
                    (int)cyklujKdyzZastavi))
}
```

Když volání

`zastavi(cyklujKdyzZastavi, cyklujKdyzZastavi)` v **`cyklujKdyzZastavi()`**

vrátí **`false`** (tedy **`cyklujKdyzZastavi()`** nezastaví), **`cyklujKdyzZastavi()`** zastaví

Když volání

`zastavi(cyklujKdyzZastavi, cyklujKdyzZastavi)` v **`cyklujKdyzZastavi()`**

vrátí **`true`** (tedy **`cyklujKdyzZastavi()`**zastaví), **`cyklujKdyzZastavi()`**nezastaví - cykluje

spor

- jediný předpoklad byl existence metody **`boolean zastavi(int J, int w)`**

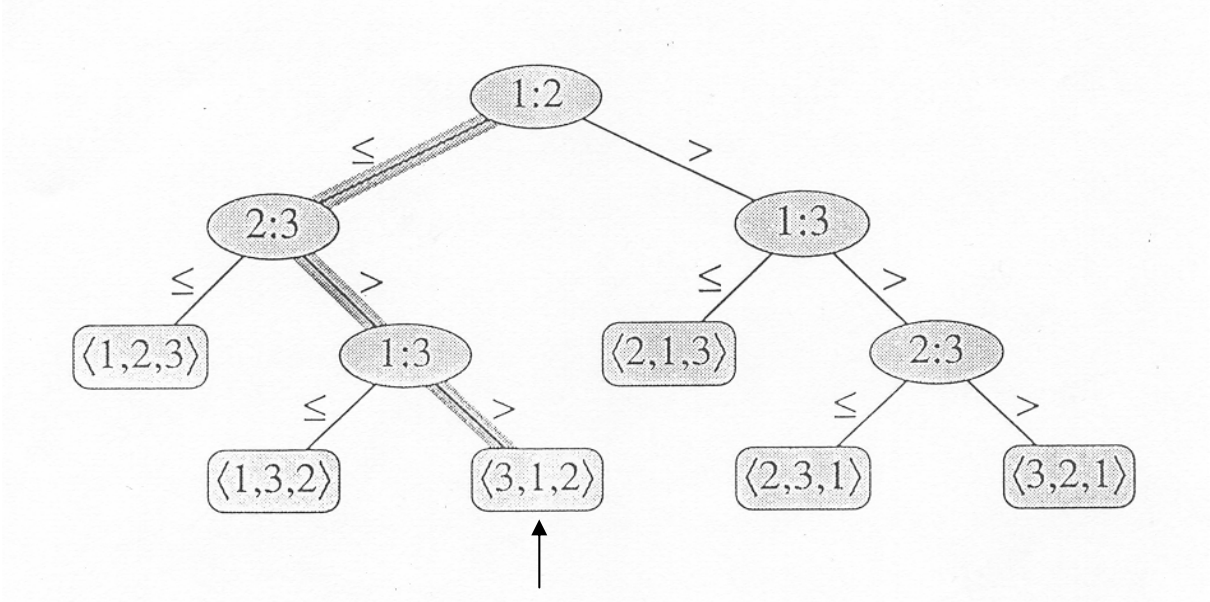
Dolní omezení pro porovnávací řazení

- hledáme dolní omezení *f(n)* počtu porovnání *T(n)* **všech** porovnávacích algoritmů řazení, $f(n) \leq T(n)$

- prvky posloupnosti $a_1, a_2, \ldots, a_n$

rozhodovací strom =
vnitřní vrcholy - porovnávané prvky (indexy)
listy - permutace všech prvků (indexů) původní posloupnosti, která je seřazena

rozhodovací stromu pro řazení a_1, a_2, a_3 vkládáním



4, 7, 2 → 2, 4, 7

- jakýkoliv správný algoritmus musí vytvořit každou z n! permutací prvků původní posloupnosti.

- každá z nich musí být alespoň v jednom listě.

- nejhorším případem počtu porovnaní vykonaných algoritmem řazení je nejdelší cesta od kořene stromu k listu, je tedy roven výšce stromu

$T(n) = h$

- ať rozhodovací strom o výšce h má l (el) listů, potom $l \leq 2^h$

- listů musí být alespoň tolik kolik je permutací, $n! \leq l$

$n! \leq l \leq 2^h$

$log_2 (n!) \leq h = T(n)$

$n! \approx (n/e)^n$

$log_2(n!)=n*log_2 n– n* log_2 e,$

- dolní omezení pro nejhorší případ: $\Omega(n*log_2 n).$

- pro řazení haldou a slučováním je horní omezení času výpočtu $O(n*log_2 n)$

- tato řazení jsou **asymptoticky optimální**

Klasifikace problémů

- algoritmicky řešitelné a algoritmicky neřešitelné

- většina našich algoritmů měla časovou náročnost $O(n^k)$ pro nějakou konstantu k, tj. polynomiální

- jsou i řešitelné problémy, o nichž víme, že nemají algoritmus s polynomiálním časem výpočtu; problém Hanojských věží vyžaduje minimální počet přesunů disků je $2^n - 1$

- problémy, které jsou řešitelné v polynomiálním čase považujeme za zvládnutelné, nebo-li lehké

- problémy, které vyžadují superpolynomiální čas považujeme za nezvládnutelné, nebo-li těžké

- jako při zkoumání řešitelnosti problémů se omezíme na **rozhodovací** problémy

- mnoho prakticky důležitých problémů jsou **optimalizační** problémy, kdy výstupem je hodnota, která nejlépe splňuje zadaná kritéria

Problém **NEJKRATŠÍ_CESTA**:

V grafu G je úkolem najít mezi jeho vrcholy u a v cestu s nejmenším počtem hran.

optimalizační problém vs. **rozhodovací** problém

NEJKRATŠÍ_CESTA **CESTA**

Problém **CESTA**:

Existuje v daném grafu G mezi vrcholy u a v cesta mající délku nejvíce k hran.

- vyřešením problému NEJKRATŠÍ_CESTA (optimalizační) a porovnáním nalezené hodnoty s k máme řešení problému CESTA (rozhodovací)

- rozhodovací problém je **„lehčí“** nebo alespoň **není „těžší“** než problém optimalizační

- je-li rozhodovací problém těžký, je i optimalizační problém je těžký

Kódování

- pro řešení **abstraktního** problému na počítači jeho instance musí být **kódována** do binárních řetězců

- problém v tomto tvaru nazýváme **konkrétní**

Konkrétní problém je řešitelný v polynomiálním čase, existuje-li algoritmus, který ho řeší v čase $O(n^k)$.

Třída problémů složitosti *P* je množina konkrétních problémů řešitelných v polynomiálním čase.

Kódování a čas výpočtu

1. Zjištění hrany mezi dvěma vrcholy grafu:

Je-li graf zakódován maticí sousednosti je čas výpočtu $\Theta(1)$.

Je-li graf zakódován seznamy sousednosti je čas výpočtu $\Theta(n)$, kde $n = |V|$.

2. Mějme algoritmus, kterého vstup je celé číslo *k* a časová náročnost je *Θ(k):*

- velikost vstupu ***n*** takových algoritmů vyjadřujeme počtem znaků pro jeho reprezentaci

- v unární reprezentaci (Římané ji použili pro k=1,2,3) to bude řetězec ***k*** jedniček, ***k=n***
- čas výpočtu bude polynomiální – ***Θ(n)***.

- v kódování v dvojkové soustavě, pro číslo ***k*** délka vstupu bude ***n=***$\lfloor \log_2 k + 1 \rfloor$***, k≈2^n***
- čas výpočtu bude exponenciální - ***Θ(2^n)***

- budeme uvažovat standardní kódovaní, tj. čísla v dvojkové soustavě a znaky v ASCII kódu (Unicode).

Pro některé rozhodovací problémy neumíme nalézt **řešení** v polynomiálním čase, ale existuje pro ně polynomiální **verifikační algoritmus.**

Verifikační algoritmus ověří řešení pro instanci problému na základě poskytnutých dat, která nazveme **certifikát.**

Je zadaná výroková formule splnitelná?
Instance problému: zadaná formule $P \wedge \neg Q \vee R$
Certifikát: *P* = true, *Q* = false, *R* = false
Verifikace – vyhodnocení
(algoritmus: vyhodnocení 2^n pravdivostních přiřazení *n* proměnným)

Je v zadané množině celých čísel podmnožina, která má součet prvků rovný nule?
Instance problému: zadaná množina {-7, -2, -3, 5, 6}
Certifikát { -2, -3, 5}
Verifikace – sčítání
(algoritmus: výpočet součtů prvků 2^n podmnožin zadané množiny *n* prvků)

Problémy, pro které existuje polynomiální verifikační algoritmus *A(instance problému, certifikát),* tvoří třídu složitosti NP – nedeterministicky polynomiální.

Vztah třídy P a třídy NP

Je-li nějaký problém ve třídě **P,** verifikační algoritmus ***A(instance problému, certifikát)*** ignoruje certifikát a jeho výstup je dán polynomiálním algoritmem řešení pro instatnci problému.

Příklad:

Problém: Je v grafu G cesta z u do v ?
Instance problému: G, u, v
Verifikační algoritmus: polynomiální BFS(G, u) - nalezení dosažitelných vrcholů, je v prvkem množiny dosažitelných vrcholů?

$$\mathbf{P} \subseteq \mathbf{NP}$$

Je **P** vlastní podmnožinou **NP** ?

Tato otázka se označuje **P ≠ NP**.

Není známo jestli **P = NP**.