# Základy informatiky
# *Čo počítače (ne)dokážu*
## IB110

Ivana Černá

Fakulta informatiky
Masarykova univerzita

podzim 2010

Technické řešení této výukové pomůcky je spolufinancováno Evropským sociálním fondem a státním rozpočtem České republiky.


INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

# Literatúra


- D. Harel, Y. Feldman:*Algorithmics: The Spirit of Computing*. Third Edition. Addison Wesley, 2004
- D. Harel: *Computers Ltd. : what they really can't do*. Oxford University Press, 2000.
- J. Hromkovič: *Algorithmic Adventures. From Knowledge to Magic*. Springer, 2009.

# Elektronická podpora v IS MU

- Interaktívna osnova
  https://is.muni.cz/auth/el/1433/podzim2010/IB110/index.qwarp
- Diskusné fórum predmetu

Všetky obrázky použité v prezentáciach, pokiaľ nie je explicitne uvedené inak, sú prevzaté z publikácie
D. Harel, Y. Feldman:*Algorithmics: The Spirit of Computing*. Third Edition. Addison Wesley, 2004

# Problémy a algoritmy

# Počítače

- *Computers are amazing machines. They seem to be able to do anything.*
- *It is all the more remarkable, therefore, that the digital computer can be thouhgt of as merely a large collection of switches, or bits.*
- *A computer can directly execute only a small number of extremely trivial operations.*
- *Computers may differ in size, types of elementary operations, speed, physical media, internal organization, external environment.*
- *What is it that transforms such trivial operations on bits into the incredible feats we see computers perform?*
- *What computers can and cannot solve?*

# Historické zastavenia

- cca 400 B.C. Euklidov algoritmus (software)
- 1801 Joseph Jacquard, pletací stroj (hardware)
- 1833 Charles Babbage, *the difference engine* - konkrétne operácie, *the analytical engine* - programovateľný stroj
- Ada Byron - programy
- 1890 Herman Hollerith - stroj založený na diernych štítkoch použirý pri sčítaní obyvateľstva
- 30-te roky: základy teórie algoritmov, pojem nerozhodnuteľnosti
- 50-te, 60-te roky - technologický pokrok

# Čo je informatika (Computer Science)?

## prevládajúce názory

- rozdielne schopnosti populácie v práci s počítačmi
- informatika = ICT skills, informačné technológie
- tendencia posudzovať vedný obor podľa jeho aplikácií

## definícia informatiky ako vedného oboru

- mnohovrstevnosť – matematika až inžniersky prístup
- čo sú základné stavebné kamene vednej disciplíny?
  - jazyk, terminológia
  - definícia, presný význam základných pojmov
  - axiómy, základné tvrdenia

## základné pojmy

problém a algoritmus

# Problémy a algoritmy

# Problém

čo je problém?

# Problém

čo je problém?

## výpočtový problém

- (nekonečná) množina vstupných inštancií
  *prípustné vstupy*
- špecifikácia požadovaných výstupov
  funkcia vstupných inštancií

# Riešenie problému

Metóda riešenia problému popisuje efektívnu cestu, ktorá vedie k nájdeniu požadovaných výstupov. Popis pozostáva z postupnosti inštrukcií, ktoré môže ktokoľvek realizovať.

existencia metódy riešenia problému znamená možnosť vypočítať riešenie *automaticky*

v danom kontexte hovoríme o algoritme pre riešenie problému

# Algoritmus — historický pohľad I

- koniec 19. storočia, priemyslová revolúcia, kauzálne chápanie sveta, zákony umožňujúce úplné pochopenie sveta
- program Davida Hilberta
  (a) celá matematika sa dá vybudovať z konečnej sady vhodných axióm
  (b) matematika vytvorená týmto spôsobom je kompletná v tom zmysle, že každé tvrdenie vyjadriteľné v jazyku matematiky sa dá dokázať alebo vyvrátiť v tejto teórii (z danej sady axiómov)
  (c) existuje *metóda* pre dokázanie resp. vyvrátenie každého tvrdenia.
- kľúčový pojem metódy

# Algoritmus — historický pohľad II

- Kurt Gödel (1931) dokázal, že
  (a) neexistuje žiadna úplná (rozumná) matematická teória;
  v každej korektnej a dostatočne veľkej teórii je možné formulovať tvrdenia, ktoré nie je možné vnútri tejto teórie dokázať. Aby bolo možné dokázať správnosť týchto tvrdení, je nutné k teórii pridať nové axiómy a vybudovať tak novú, väčšiu teóriu
  (b) neexistuje metóda (algoritmus) pre dokazovanie matematických teorém
- pre svoj dôkaz potreboval presnú definíciu pojmu metóda (algoritmus)
  *(nie je možné dokázať neexistenciu algoritmu bez rigoróznej definície toho čo je a čo nie je algoritmus)*
- prvá formálna definícia pojmu algoritmu: Alan Turing (1936)
- ďalšie definície a ich ekvivalencia

# Základné otázky

Existujú problémy, ktoré nie sú riešiteľné automaticky (algoritmicky)?

Ak áno, ktoré problémy sú algoritmicky riešiteľné a ktoré nie?

# Problémy a algoritmy

# Programovacie jazyky I

- algoritmy musia byť zapísané jednoznačne a formálne
- programovací jazyk – jazyk pre špecifikáciu algoritmov
- program – zápis algoritmu v programovacom jazyku
- strojový kód vs. programovací jazyk vyššej úrovne

# Programovacie jazyky II

- ako prezentovať algoritmus reálnemu počítaču?
- ako "prinútiť" počítač, aby realizoval výpočet tak, ako sme to zamýšľali?
- programátori sú ľudia, uvažujú abstraktne a vyjadrujú sa pomocou slov a symbolov
- počítače sú stroje, ktoré dokážu realizovať veľmi jednoduché úlohy
- programovacie jazyky pomáhajú ľuďom komunikovať s počítačmi

# Vývojové diagramy

# Programovací jazyk vyššej úrovne

jazyk pre popis algoritmov

základné stavebné kamene

- riadiace štruktúry
- dátové štruktúry

# Riadiace štruktúry

- postupnosť príkazov
- podmienené vetvenie
- ohraničená iterácia
- podmienená iterácia

kombinácia riadiacich štruktúr - vnorenie

# Dátové typy

- premenné
- vektory, zoznamy
- polia, tabuľky
- zásobníky a fronty
- stromy a hierarchie
- ...

# Vlastnosti programovacích jazykov

- presná syntax
- presná sémantika

# Syntax

- nejednoznačnosti; presný význam pre pojmy, ktoré sa používajú v bežnej komunikácii
- Porovnanie "=" a priradenie "="

Java
```
if (x==y)  z = 3;  else  z = 4;
```

FORTRAN
```
IF (X .EQ. Y)
     THEN Z = 3
     ELSE Z = 4 END FI
```

Pascal
```
if x = y then z := 3 else z:= 4
```

# Syntax - príklad

- hypotetický programovací jazyk PL

## Príklad

```
1 input N;
2 X := 0;
3 for Y from 1 to N do
4       X := X + Y
5 od
6 output X.
```

# Definícia syntaxe - syntaktické diagramy

# Definícia syntaxe — BNF

$$\langle \text{príkaz} \rangle \quad ::= \langle \text{for príkaz} \rangle \mid \langle \text{priraďovací príkaz} \rangle \mid \ldots$$
$$\langle \text{for príkaz} \rangle \quad ::= \langle \text{premenná} \rangle \textbf{ from } \langle \text{hodnota} \rangle \textbf{ to } \langle \textit{hodnota} \rangle$$

. . .

BNF umožňuje generovať (syntakticky správné) programy

# Kontrola syntaktických chýb

- program sa nedá vygenerovať použitím BNF
- automatizovaný proces

# Sémantika

**for** $Y$ **from** 1 **to** $N$ **do**

- význam — odčítanie, príkaz pre tlačiareň, *dnes je pekný deň* ???
- význam podľa použitých slov ???
- čo ak $N = 3.14$ ???
- presná sémantika je nevyhnutná !!!
- manuál (dokumentácia) ako definícia sémantiky ???

# Sémantika - príklad

## Príklad

*procedúra P(s parametron V)*
*(1)* **call** *V(s parametrom V), výsledok ulož do X*
*(2)* **if** $X = 1$ **then return** *0;* **else return** *1*

- procedúra, ktorá má ako svoj parameter názov procedúry
- syntaktická korektnosť
- **call** $P$(s parametrom $P$) – aká je návratová hodnota?
- sémantika musí dať jednoznačnú odpoveď

# Výpočet programu

- od programu v jazyku vyššej úrovne k manipulácii s bitmi
- program je postupne transformovaný na strojovú úroveň
- transformácia
  - kompilácia
  - interpretácia
  - kombinovaný prístup

# Kompilácia

- program v jazyku vyššej úrovne je preložený do jazyka nižšej úrovne (jazyk symbolických adries)

**for** *Y* **from** 1 **to** *N* **do**
    telo cyklu
**od**

```
        MOVE 0, Y    do registra Y ulož 0
LOOP:   CMP N, Y     porovnaj hodnoty uložené v N a Y
        JEQ REST     ak rovnosť, tak pokračuj príkazom REST
        ADD 1, Y     pripočítaj 1 k Y
            telo cyklu
        JMP LOOP
```

- prekladač

# Kompilácia - pokr.

- preklad z jazyka symbolických adries do strojového kódu
- assembler

Optimalizácia kódu počas kompilácie

# Interpretácia

- interpreter postupuje príkaz po príkaze
- každý príkaz je okamžite preložený do strojového kódu a vykonaný

Možnosť lepšie sledovať výpočet.
Typicky jednoduchá tvorba interpretru.
Nemožnosť optimalizácii.

# Programovacie esperanto

- prečo rôzne programovacie jazyky?
- programovací jayzk poskytuje programátorovi istú úroveň abstrakcie
- potreba nových typov abstrakcií
  - vývoj hardwaru (*paralelné počítače a programovanie vo vláknach*)
  - nové aplikačné oblasti (*mulimediálne aplikácie*)
- paradigmata - kategorizácia existujúcich programovacích jazykov

# Imperatívne programovanie

- prístup blízky ľudskému uvažovaniu
- imperatívne programovanie popisuje výpočet pomocou postupnosti príkazov a určuje presný postup, ako daný algoritmický problém riešiť
- pamäť počítača je súborom pamäťových miest organizovaných do rôznych dátových štruktúr
- program buduje, prechádza a modifikuje dátové štruktúry tak, že načíta dáta a mení hodnoty uložené v pamäti
- príkazy, v závislosti na vyhodnotení podmienok, menia stav premenných
- historicky najstaršie imperatívne programovacie jazyky: strojové jazyky
- FORTRAN, ALGOL, COBOL, BASIC, Pascal, C, Ada
- základné typy príkazov - priradenie, cykly, príkazy vetvenia

# Funkcionálne programovanie

- výpočtom funkcionálneho programu je postupnosť vzájomne ekvivalentných výrazov, ktoré sa postupne zjednodušujú
- výsledkom výpočtu je výraz v normálnej forme, ktorý sa nedá ďalej zjednodušiť
- program je chápaný ako jedna funkcia, ktorá obsahuje vstupné parametry. Výpočet je vyhodnotením tejto funkcie.
- imperatívny prístup: ako sa má vypočítať
  funkcionálny prístup: čo sa má vypočítať
- Erlang, Haskell, Lisp, ML, Ozx, Scheme

# Logické programovanie

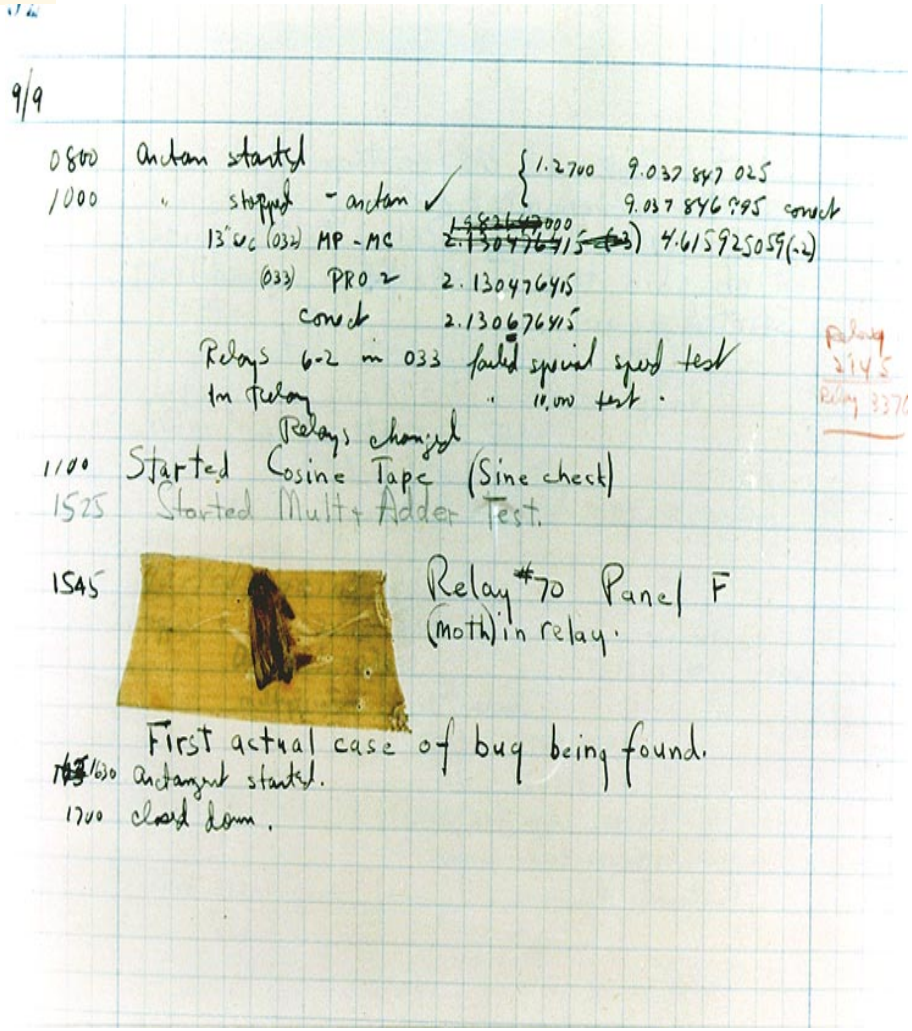
- postavené na použití matematickej logiky
- logický program je tvorený sadou pravidiel a jednoduchých logických tvrdení
- dotaz sa rieši prehľadávaním množiny pravidiel. Hľadá sa dôkaz, ktorý poskytuje odpoveď na zadaný dotaz

# Objektovo orientované programovanie

- pamäť počítača pozostáva z objektov
- s každým objektom sú asociované operácie, ktoré môže objekt poskytovať (prístupné ako metódy volania)
- program je súborom objektov, ktoré si posielajú správy obsahujúce požiadavky na realizáciu operácií a odpovede
- Smalltalk, Java, C++, Python, Ruby. Lisp

# Správnosť algoritmov

# Prečo?




- 1962, Mariner1 - štart rakety
- 1981, Kanada - informácia o volebných preferenciách
- 1985-87, Therac-25 - nesprávne dávky röntgenového žiarenia
- problém Y2K
- http://www.devtopics.com/20-famous-software-disasters/

# Typy chýb

- syntaktické chyby
  - *until / untli*
  - `for (k=0;k<101){ sum = sum + k }` *versus*
    `for (k=0;k<101;k=k+1){ sum = sum + k }`
- sémantické chyby
  - *výsledná hodnota premennej cyklu*
  - `for (k=0;k=k+1;k<101){ sum = sum + k }` *versus*
    `for (k=0;k<101;k=k+1){ sum = sum + k }`
- logické chyby
  *pre daný text zisti, koľko viet obsahuje slovo* kniha
  - *koniec vety indikuje výskyt symbolov „. “ (bodka, medzera)*
  - *koniec vety indikuje výskyt symbolu „.“ (bodka)*

počítače nerobia chyby

# Testovanie a ladenie

- syntaktické chyby, run-time chyby
- testovanie, testovacie sady
- ladenie
- nezaručujú bezchybnosť algoritmu

# Čiastočná a úplná korektnosť

2pecifikácia algoritmického problému
1. určenie množiny vstupných inštancií
2. určenie vzťahu medzi vstupnými inštanciami a požadovaným výstupom.

- čiastočná korektnosť: pre každú vstupnú inštanciu $X$ platí, že ak výpočet algoritmu na $X$ skončí, tak výstup má požadovanú vlastnosť
- konečnosť: výpočet skončí pre každú vstupnú inštanciu
- úplná koreknosť: čiastočná korektnosť + konečnosť

# Dôkaz korektnosti

**invarianty**

- kontrolné body programu
- invariant = tvrdenie, ktoré platí pri každom priechode kontrolným bodom
- čiastočná korektnosť

**konvergencia**

- s kontrolnými bodmi asociujeme kvantitatívnu vlastnosť
- pri každom priechode kontrolným bodom sa hodnota kvantitatívnej vlastnostni znižuje
- hodnota kvantitatívnej vlastnosi nesmie prekročiť dolnú hranicu
- konečnosť výpočtu

# Zrkadlový obraz

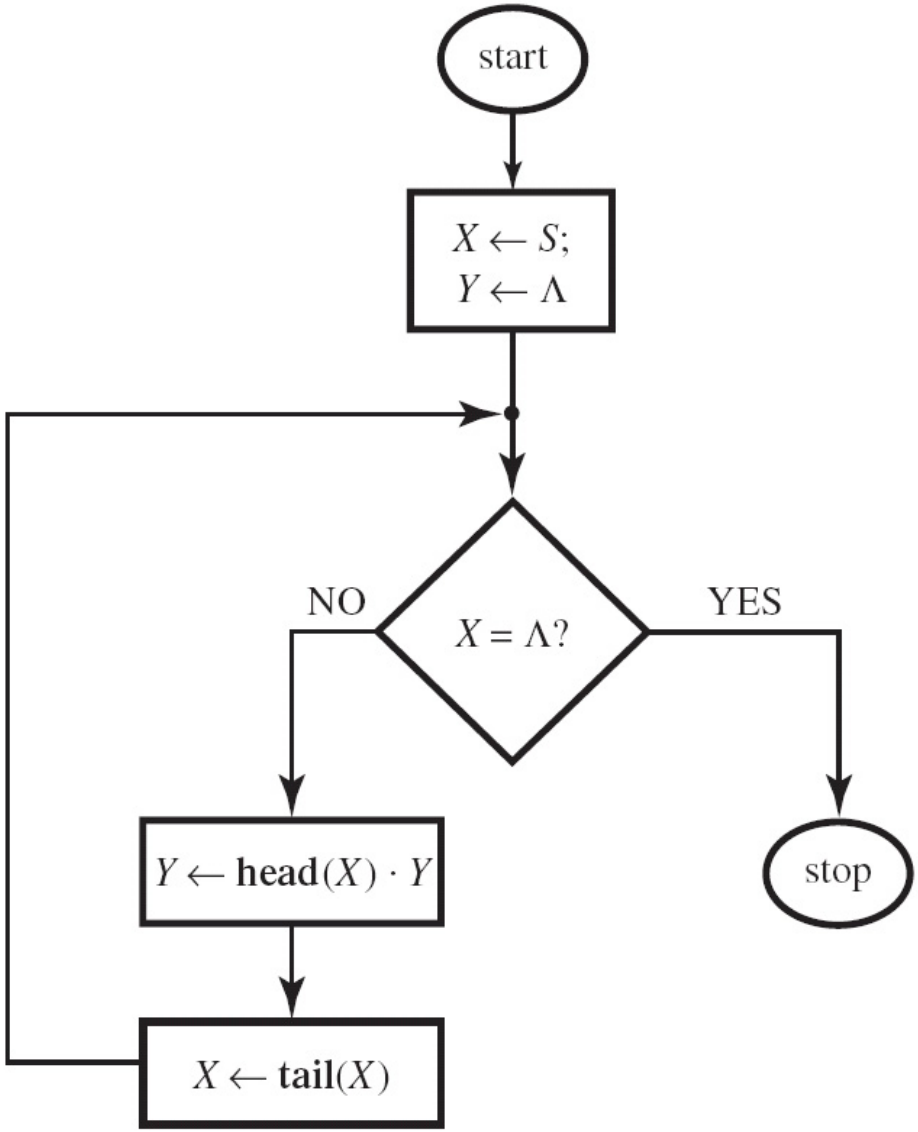
Vstup: reťazec $S$

Výstup: symboly reťazca $S$ v obrátenom poradí

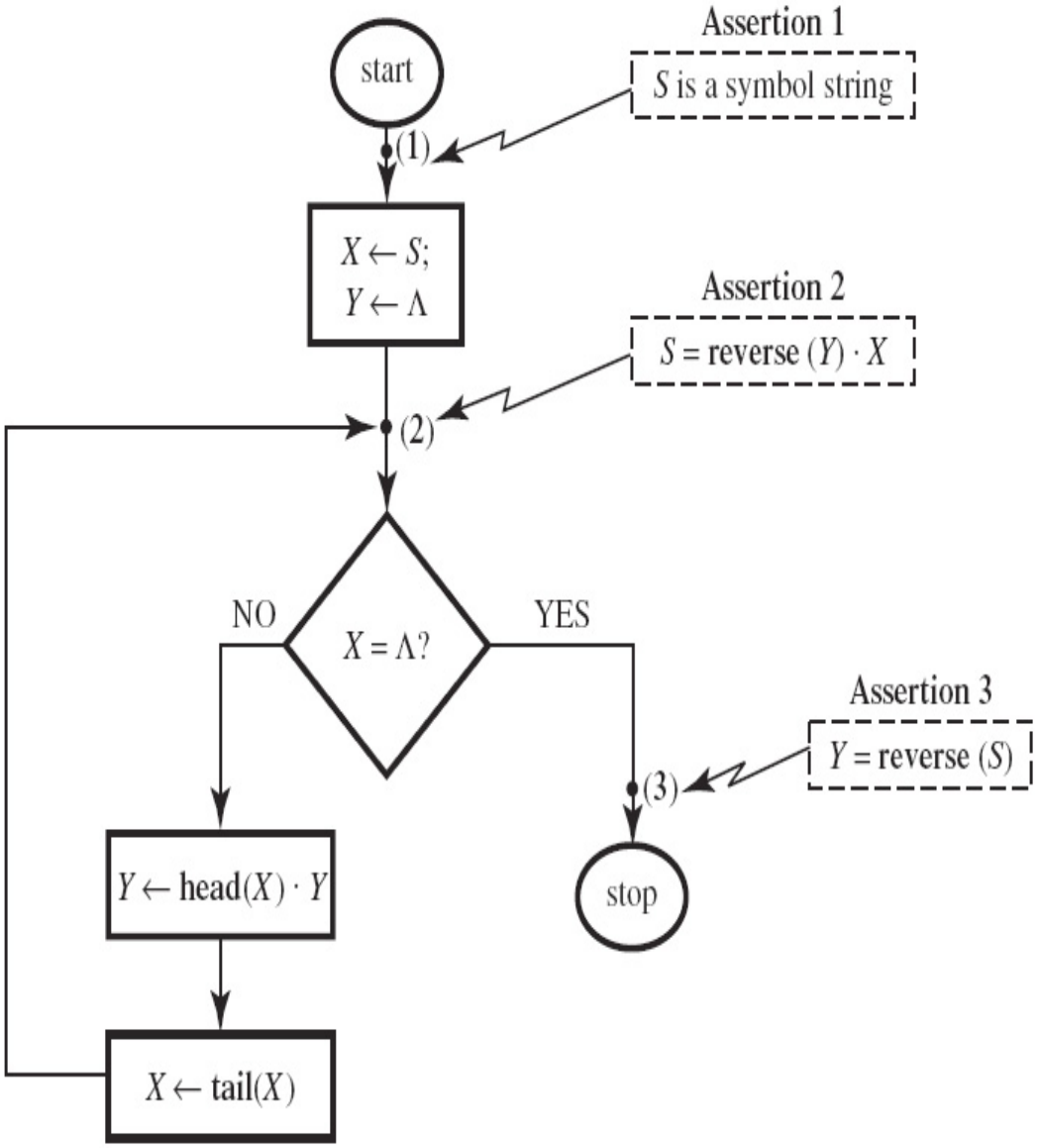
Notácia

- **reverse**(„fakulta“) = ‘„atlukaf“
- **head**(„fakulta“) = „f“
- **tail**(„fakulta“) = „akulta“
- symbol $\Lambda$ označuje prázdny reťazec (reťazec neobsahuje žiaden symbol)
- symbol $\cdot$ označuje zreťazenie (spojenie) dvoch reťazcov

# Algoritmus


start
$X \leftarrow S$;
$Y \leftarrow \Lambda$
$X = \Lambda$?
NO
YES
$Y \leftarrow \mathbf{head}(X) \cdot Y$
stop
$X \leftarrow \mathbf{tail}(X)$

# Kontrolné body a invarianty

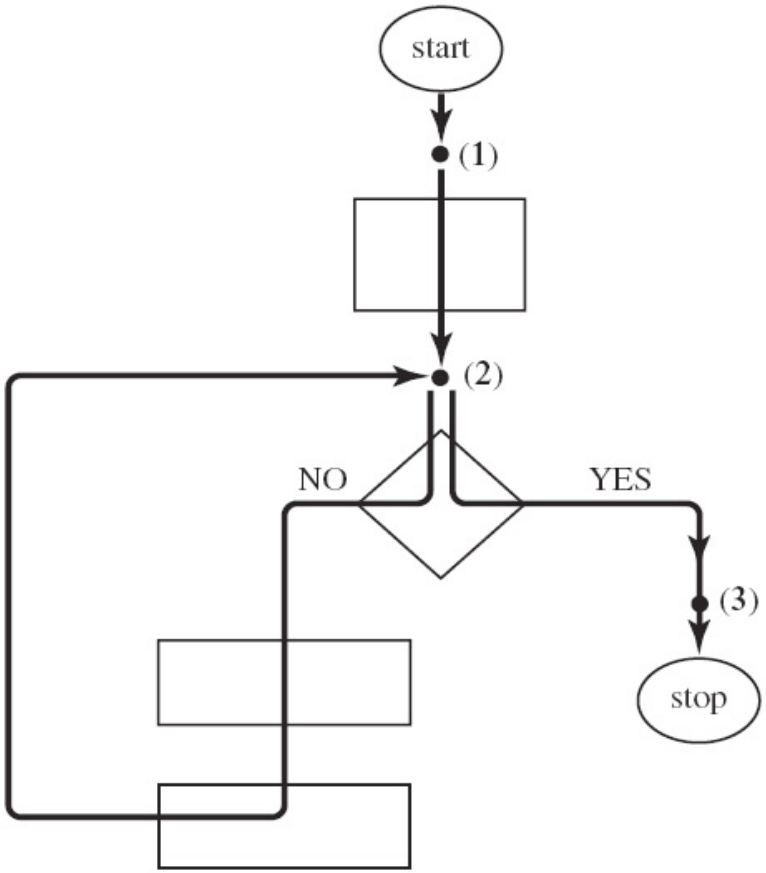



- Invariant 1
  vstupná podmienka
- Invariant 2
  $S = \mathbf{reverse}(Y) \cdot X$
  charakterizuje výpočet
- Invariant 3
  $Y = \mathbf{reverse}(S)$
  požadovaný vzťah medzi vstupom $S$ a výstupom $Y$

# Platnosť invariantov

dokazujeme, že pre každý platný vstup: ak výpočet dosiahne kontrolný bod, tak tvrdenie je pravdivé

v akom poradí sa prechádzajú kontrolné body?




$$1 \longrightarrow 2 \longrightarrow 2 \longrightarrow \cdots 2 \longrightarrow 3$$

# Platnosť invariantov

$1 \longrightarrow 2$ pre každý reťazec $S$ po vykonaní príkazov $X \leftarrow S, Y \leftarrow \Lambda$ platí rovnosť $S = \mathbf{reverse}(Y) \cdot X$

$2 \longrightarrow 3$ ak $S = \mathbf{reverse}(Y) \cdot X$ a $X = \Lambda$, tak $Y = \mathbf{reverse}(S)$

$2 \longrightarrow 2$ ak $S = \mathbf{reverse}(Y) \cdot X$ a $X \neq \Lambda$, tak po vykonaní príkazov $Y \leftarrow \mathbf{head}(X) \cdot Y;\ X \leftarrow \mathbf{tail}(X)$ platí znovu tá istá rovnosť pre nové hodnoty premenných $X$ a $Y$

dokázali sme čiastočnú korektnosť

# Konečnosť

- výpočet algoritmu je nekonečný práve ak prechádza kontrolným bodom 2 nekonečne veľa krát
- s kontrolným bodom 2 asociujeme kvantitatívnu vlastnosť (tzv. *konvergent*) a ukážeme, že jej hodnota klesá a pritom je zdola ohraničená
- konvergentom pre kontrolný bod 2 je dĺžka reťazca $X$
- pri každom priechode kontrolným bodom 2 dĺžka reťazca $X$ klesne o 1
- ak dĺžka $X$ klesne na 0 ($X$ je prázdny reťazec), tak výpočet neprechádza cyklom a nenavštívi kontrolný bod 2

dokázali sme konečnosť

# Korektnosť

$$\text{korektnosť} = \text{čiastočná korektnosť} + \text{konečnosť}$$

# Euklidov algoritmus

Vstup dve kladné celé čísla $X$ a $Y$

Výstup najväčší spoločný deliteľ $Z$ čísel $X$ a $Y$

spoločný deliteľ $Z$ $Z$ delí $X$ a $Z$ delí $Y$ (celočíselne)

najväčší deliteľ pre každé číslo $U > Z$, buď $U$ nedelí $X$ alebo $U$ nedelí $Y$

# Implementácia

**function** $Euclid(X, Y)$
$V \leftarrow X$
$W \leftarrow Y$
**while** $V \neq W$ **do**
    **if** $V > W$ **then** $V \leftarrow V - W$ **fi**
    **if** $V < W$ **then** $V \leftarrow W - V$ **fi**
**od**
**return** $(V)$

Invariant 1 $V$ a $W$ sú násobkom $Z$
Invariant 2 $V \geq Z$ a $W \geq Z$
Invariant 3 neexistuje väčší spoločný deliteľ čísel $V$ a $W$ než číslo $Z$
všetky invariatny platia v každom bode výpočtu

# Čiastočná korektnosť

Invariant 1 $V$ a $W$ sú násobkom $Z$

Invariant 2 $V \geq Z$ a $W \geq Z$

Invariant 3 neexistuje väčší spoločný deliteľ čísel $V$ a $W$ než číslo $Z$

---

Inicializácia $V \leftarrow X$, $W \leftarrow Y$

- invarianty 1, 2, 3 sa priradením neporušia

**IF** príkaz **if** $V > W$ **then** $V \leftarrow V - W$ **fi**

- Fakt Ak $V > W$, tak dvojice čísel $V, W$ a $V - W, W$ majú rovnakých spoločných deliteľov
- ak $Z$ delí $V$, $W$ a $V > W$, tak $V - W > 0$ a $V - W \geq Z$
- invarianty 1, 2, 3 zostávajú zachované

**IF** príkaz **if** $W > V$ **then** $W \leftarrow W - V$ **fi**

- symetricky

# Čiastočná korektnosť

Invariant 1 $V$ a $W$ sú násobkom $Z$
Invariant 2 $V \geq Z$ a $W \geq Z$
Invariant 3 neexistuje väčší spoločný deliteľ čísel $V$ a $W$ než číslo $Z$

---

**while** príkaz

- všetky invarianty zostávajú v platnosti po prevedení jednotlivých príkazov cyklu
- cyklus končí keď $V = W$
- $V$ je najväčším spoločným deliteľom $V, W$
- $V = Z$

čiastočná korektnosť

# Konečnosť

- výpočet je nekonečný práve ak **while** príkaz sa vykoná nekonečne veľa krát
- konvergentom **while** cyklu je súčet $V + W$
- pri každom vstupe do tela cyklu je $V \geq Z > 0$, $W \geq Z > 0$ a $V \neq W$
- pri vykonaní tela cyklu sa odčíta celé kladné číslo buď od $V$ alebo od $W$
- suma $V + W$ sa pri každom priechode cyklom zníži aspoň o 1
- na začiatku je $V + W = X + Y$ a preto sa cyklus vykoná nanajvýš $X + Y$ krát

konečnosť

# Triedenie vkladaním

$Insertion - Sort(A)$
**for** $j \leftarrow 2$ **to** $length[A]$ **do**
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    **while** $i > 0 \wedge A[i] > key$ **do** $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$ **od**
    $A[i+1] \leftarrow key$
**od**

Invariant na začiatku iterácie **for** cyklu obsahuje $A[1 \ldots j-1]$ tie isté prvky, ako obsahovalo na týchto pozíciách pole $A$ na začiatku výpočtu, ale utriedené od najmenšieho po najväčší

# Čiastočná korektnosť a konečnosť

Invariant na začiatku iterácie **for** cyklu obsahuje $A[1 \ldots j-1]$ tie isté prvky, ako obsahovalo na týchto pozíciách pole $A$ na začiatku výpočtu, ale utriedené od najmenšieho po najväčší

---

Inicializácia tvrdenie platí na začiatku výpočtu ($j = 2$, postupnosť $A[1]$ obsahuje jediný prvok a je utriedená)

**FOR** cyklus v tele cyklu sa hodnoty $A[j-1], A[j-2], A[j-3], \ldots$ posúvajú o jednu pozíciu doprava až kým sa nenájde vhodná pozícia pre $A[j]$

Ukončenie **for** cyklus sa ukončí keď $j = n+1$. Substitúciou $n+1$ za $j$ dostávame, že pole $A[+ \ldots n$ obsahuje tie isté prvky, ako na začiatku výpočtu, ale utriedené.

---

Konečnosť **for** cyklus nemení hodnotu riadiacej premennej cyklu

# Správnosť algoritmov

# Formálna verifikácia

- interaktívne dokazovanie
- dokazovanie formálnym odvodením (*theorem proving*)
- overovanie modelu (*model checking*)

# Zložitosť algoritmov

# Zložitosť algoritmov

- koreknosť algoritmu sama o sebe nezaručuje jeho použiteľnosť
- dĺžka výpočtu a jeho pamäťová náročnosť
- časová a priestorová zložitosť
- zložitosť výpočtu závisí na vstupnej inštancii
- zložitosť algoritmu vyjadrujeme ako funkciu dĺžky vstupnej inštancie

# Optimalizácia zložitosti algoritmu

- na úrovni kompilácie
- programátorská optimalizácia

# Príklad 1

Vstup zoznam študentov a počet bodov, ktoré získali v záverečnom teste predmetu IB110
$L(1), \ldots, L(N)$

Výstup normalizované body
(1) Nájdi maximálny počet bodov, *MAX*
(2) každý bodový zisk vynásob hodnotou 100 a vydeľ hodnotou *MAX*

Implementácia (1) štandartne
(2) **for** $I$ **from** 1 **to** $N$ **do** $L(I) \leftarrow L(I) \times 100/MAX$ **od**
*pre každé* $L(I)$ *potrebujeme 1 násobenie a 1 delenie*

Optimalizácia (1) štandartne
(2) $FAKTOR \leftarrow 100/MAX$
(3) **for** $I$ **from** 1 **to** $N$ **do** $L(I) \leftarrow L(I) \times FAKTOR$ **od**
zlepšenie o cca 50%

# Príklad 2

- vyhľadávanie prvku $X$ v neusporiadanom zozname
- implementácia pomocou cyklu, v ktorom sa realizujú dva testy: (1) našli sme $X$? a (2) prehľadali sme celý zoznam?
- optimalizácia: na koniec zoznamu pridáme prvok $X$ a v cykle testujeme len podmienku (1)
- po ukončení cyklu overujeme, či nájdeny prvok $X$ sa nachádza vo vnútri zoznamu alebo na jeho konci
- zlepšenie o cca 50%

# Zložitosť algoritmov

# Otázky

- je zlepšenie o 50% (60%, 90% . . . ) dostačujúce?
- ako charakterizovať zložitosť algoritmu?
- ako porovnať zložitosť dvoch algoritmov?

zložitosť algoritmu ako funkcia dĺžky vstupnej inštancie

zložitosť v najhoršom prípade

asymptotická zložitosť, O-notácia

# Asymptotická zložitosť

- jednoduchá charakterizácia efektivity algoritmu
- umožňuje porovnať relatívnu efektivitu rôznych algoritmov
- charakterizuje, ako rastie zložitosť algoritmu s rastúcou dĺžkou vstupnej inštancie

## O-notácia

Symbolom $\mathcal{O}(g(n))$ označujeme množinu fukcií t.ž.

$$\mathcal{O}(g(n)) = \{f(n) \mid \text{existuje kladná konštanta } c \text{ a } n_0 \text{ také, že } 0 \leq f(n) \leq cg(n) \text{ pre všetky } n \geq n_0\}.$$

Rovnosť $f(n) = \mathcal{O}(g(n))$ vyjadruje, že $f(n)$ je prvkom množiny $\mathcal{O}(g(n))$.

# Robustnosť O-notácie

**Fakt**

O-notácia zakrýva konštantné faktory

- časová zložitosť algoritmu je relatívny pojem
- zložitosť je relatívna voči fixovanej množine elementárnych inštrukcií
- každý programovací jazyk resp. kompilátor môže mať inú množinu elementárnych inštrukcií
- pokiaľ používajú štandartné inštrukcie, tak rozdiel v časovej zložitosti je práve o konštantný faktor
- O-notácia je invariantná voči takýmto implementačným detailom

Nevýhoda O-notácie: skryté konštanty, hraničná hodnota $n_0$

# Binárne vyhľadávanie

Binárne vyhľadávanie položky $Y$ v telefónnom zozname s $N$ položkami $X_1, X_2, \ldots, X_N$ pre $N = 1000000$

lineárne vyhľadávanie až 1 000 000 porovnaní

zložitosť $\mathcal{O}(N)$

binárne vyhľadávanie postup: v prvom kroku porovnaj $Y$ s $X_{500000}$
podľa výsledku porovnaj v druhom kroku $Y$ buď s $X_{250000}$
alebo s $X_{750000}$
v najhoršom prípade 20 porovnaní
zložitosť $\mathcal{O}(\log_2(N))$

**Prečo?**

sme ako zložitosť vyhľadávania sme uvažovali len počet porovnaní?

# Bubblesort

```
1 procedure BUBBLESORT(A, n)
2 for i = 1 to n − 1 do
3     for j = 1 to n − i do
4         if A[j] > A[j + 1] then  vymeň A[j] s A[j + 1] fi
5     od
6 od
```

riadok 4 čas $\mathcal{O}(1)$

for cyklus 3 - 5 čas $\mathcal{O}(n - i)$

for cyklus 2 - 6 čas $\mathcal{O}(\sum_{i=1}^{n-1}(n - i))$

$$= \mathcal{O}(n(n-1) - \sum_{i=1}^{n-1} i) = \mathcal{O}(n^2)$$

celková časová zložitosť je $\mathcal{O}(n^2)$

# Vnorené cykly

```
1 r ← 0
2 for i = 1 to n do
3     for j = 1 to i do
4         for k = j to i + j do
5             r ← r + 1
6         od
7     od
8 od
```

cyklus 4 - 6 $(i+j)-j+1 = i+1$ priradení

cyklus 3 - 7 $i$ opakovaní cyklu 4 - 6, tj. $i(i+1) = i^2 + i$ priradení

cyklus 2 - 8 $n-1$ opakovaní cyklu 3 - 7, tj. $\sum_{i=1}^{n-1} i^2 + i$ priradení

$$\sum_{i=1}^{n-1} i^2 + i = \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} = \frac{n^3 - n}{3} = \mathcal{O}(n^3)$$

# Maximálny a minimálny prvok

Problém nájdenia maximálneho a minimálneho prvku postupnosti $S[1..n]$.
Zložitostné kritérium - počet porovnaní prvkov.

$max \leftarrow S[1]$
**for** $i$ **from** 2 **to** $n$ **do**
    **if** $S[i] > max$ **then** $max \leftarrow S[i]$ **fi**
**od**

Minimum nájdeme medzi zvyšnými $n-1$ prvkami podobne.
Celkove $(n-1)+(n-2)$ porovnaní.

# Maximálny a minimálny prvok

Prístup Rozdeľ a panuj

1. pole rozdeľ na dve (rovnako veľké) podpostunosti
2. nájdi minimum a maximum oboch podpostupností
3. maximálny prvok postupnosti je väčší z maximálnych prvkov podpostupností; podobne minimálny prvok

**function** $\textsc{Maxmin}(x, y)$
**if** $y - x \leq 1$ **then return** $(\max(S[x], S[y]), \min(S[x], S[y]))$
**else** $(max1, min1) \leftarrow \textsc{Maxmin}(x, \lfloor (x + y)/2 \rfloor)$
$(max2, min2) \leftarrow \textsc{Maxmin}(\lfloor (x + y)/2 \rfloor + 1, y)$
**return** $(\max(max1, max2) \min(min1, min2))$
**fi**

# Maximálny a minimálny prvok

Zložitosť: (počet porovnaní)

$$T(n) = \begin{cases} 1 & \text{pre } n = 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 & \text{pre } n > 2 \end{cases}$$

Indukciou k $n$ overíme, že $T(n) \leq \frac{5}{3}n - 2$.

1. Pre $n = 2$ platí $\frac{5}{3} \cdot 2 - 2 > 1 = T(2)$.
2. Predpokladajme platnosť nerovnosti pre všetky hodnoty $2 \leq i < n$, dokážeme jej platnosť pre $n$.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 && \text{indukčný predp.} \\ &\leq \frac{5}{3}\lfloor n/2 \rfloor - 2 + \frac{5}{3}\lceil n/2 \rceil - 2 + 2 = \frac{5}{3}n - 2 \end{aligned}$$

# Priemerná a očakávaná zložitosť

Priemerná zložitosť priemer zložitostí výpočtov na všetkých vstupoch danej dĺžky
*Quicksort - zložitosť v najhoršom prípade je* $\mathcal{O}(n^2)$, *priemerná zložitosť je* $\mathcal{O}(n \log n)$

Očakávaná zložitosť zložitosť jednotlivých výpočtov je vážená frekvenciou výskytu príslušných vstupných inštancií

Výhody: presnejšia informácia o efektivite algoritmu
*v prípade očakávanej zložitosti je relevancia voči aplikačnej oblasti*
Nevýhody: obtiažna analýza
*v prípade očakávanej zložitosti nutnosť poznať presnú frekvenciu vstupných inštancií*

# Horné a dolné odhady zložitosti

**Zložitosť algoritmu**
- v najlepšom prípade
- v najhoršom prípade
- priemerná zložitosť
- očakávaná zložitosť

**Zložitosť problému**
- dolný odhad zložitosti problému
- horný odhad zložitosti problému — zložitosť konkrétneho algoritmu pre problém
- zložitosť problému

# Techniky

**Informačná metóda** riešenie problému v sebe obsahuje isté množstvo informácie a v každom kroku výpočtu sme schopní určiť len časť tejto informácie (*násobenie matíc, cesta v grafe, triedenie*)

**Metóda sporu** *Varianta A:* za predpokladu, že algoritmus má zložitosť menšiu než uvažovanú hranicu, vieme skonštruovať vstup, na ktorom nedá korektné riešenie.
*Varianta B:* za predpokladu, že algoritmus nájde vždy korektné riešenie, vieme skonštruovať vstup, pre ktorý zložitosť výpočtu presiahne uvažovanú hranicu.

**Redukcia**

# Príklad: maximálny prvok postupnosti

## Dolný odhad

*Nech algoritmus $\mathcal{A}$ je algoritmus založený na porovnávaní prvkov a nech $\mathcal{A}$ rieši problém maximálneho prvku. Potom $\mathcal{A}$ musí na každom vstupe vykonať aspoň $n-1$ porovnaní.*

## Dôkaz

Nech $x = (x_1, \ldots, x_n)$ je vstup dĺžky $n$, na ktorom $\mathcal{A}$ vykoná menej než $n-1$ porovnaní a nech $x_r$ je maximálny prvok v $x$.
Potom v $x$ musí existovať prvok $x_p$ taký, že $p \neq r$ a v priebehu výpočtu $x_p$ nebol porovnávaný so žiadnym prvkom väčším než on sám. Existencia takého prvku plynie z počtu vykonaných porovnaní.
Ak v $x$ zmeníme hodnotu prvku $x_p$ na $x_r + 1$, tak $\mathcal{A}$ určí ako maximálny prvok $x_r$ – spor.

# Príklad: vyhľadávanie v telefónnom zozname

binárny strom a jeho hĺbka

# Výzkum v oblasti zložitosti problémov

- optimalizácia dátových štruktúr
- dolné odhady zložitosti a dôkaz optimality
- priestorová zložitosť
- vzťah medzi priestorovou a časovou zložitosťou

# Prakticky riešiteľné problémy

# Praktická použiteľnosť algoritmov

Je každý algoritmus prakticky použiteľný?

$N = 1\ 000\ 000$

vyhľadávanie v utriedenom zozname $\mathcal{O}(\log N) \ldots 20$

vyhľadávanie v zozname $\mathcal{O}(N) \ldots 1\ 000\ 000$

triedenie zoznamu $\mathcal{O}(N \log N) \ldots 20\ 000\ 000$

Hanojské veže $\mathcal{O}(2^N) \ldots$ milión presunov za minútu $\Rightarrow$ 500 000 rokov

Je riešením výkonnejší hardware a väčšia trpezlivosť?

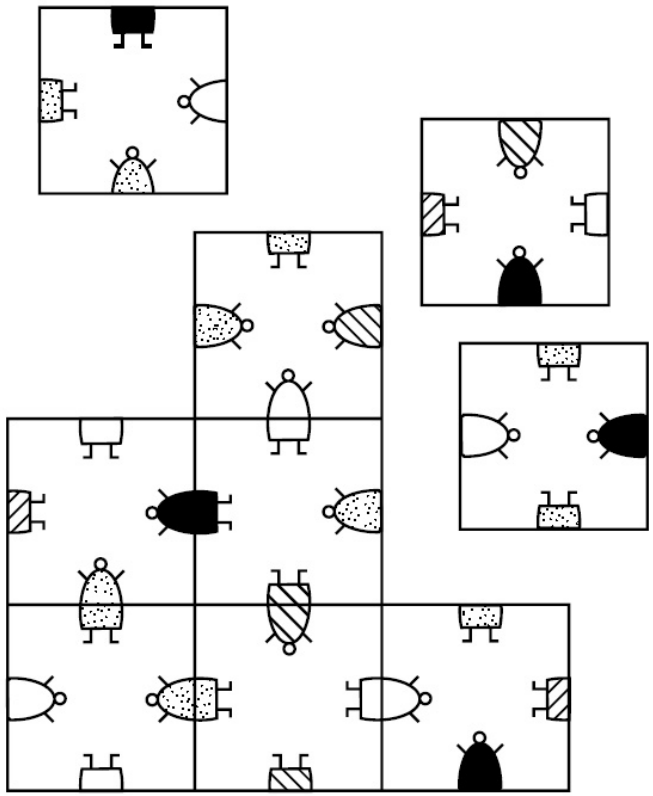
# Monkey Puzzle Problem

http://www.keithharingart.com/

# MP hlavolam

Vstup $N$ kariet ($N = M^2$)

Otázka Dajú sa karty usporiadať do štvorca $M \times M$ tak, aby sa susediace hrany zhodovali?



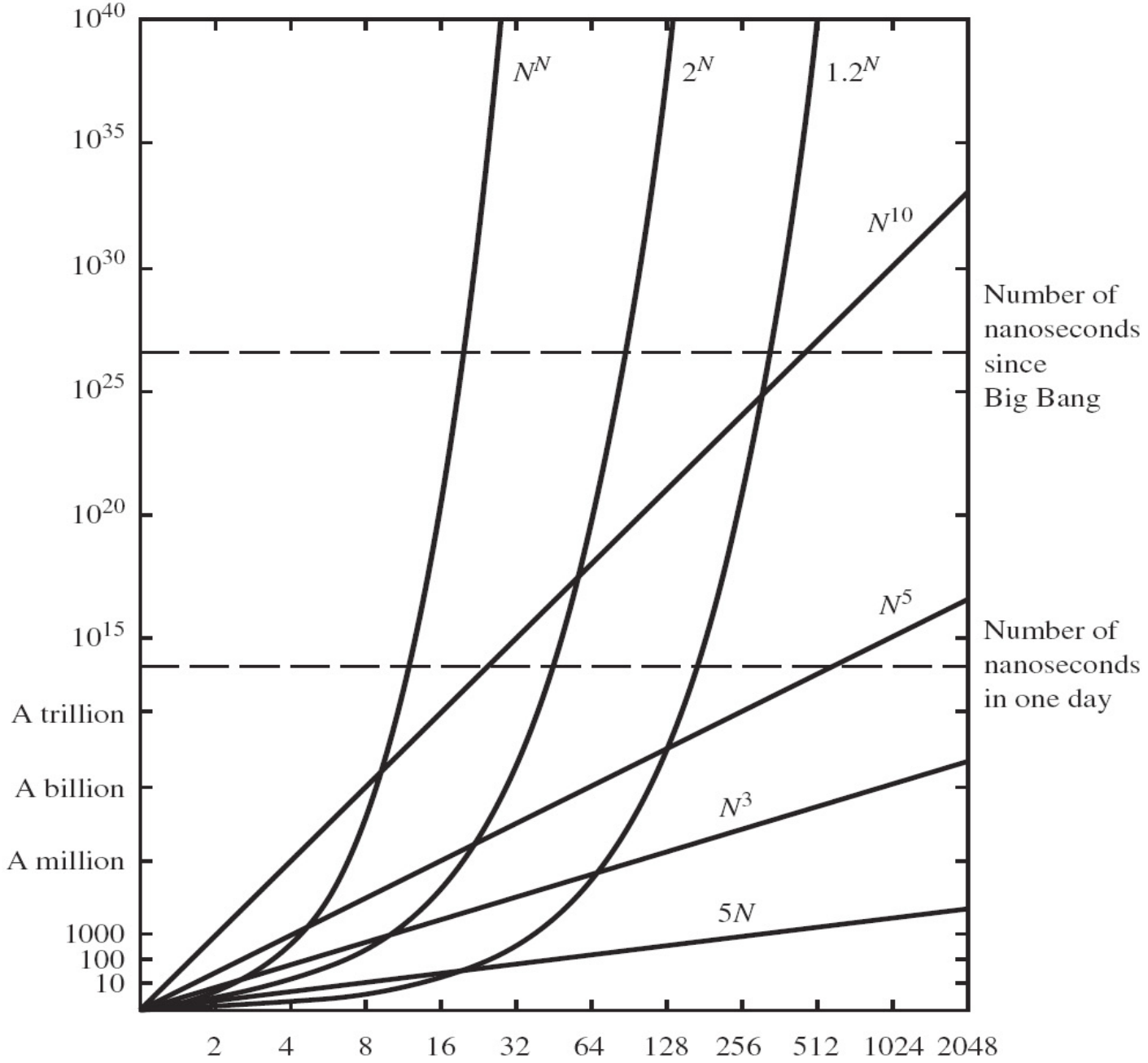
Karty majú fixnú orientáciu (nemôžeme ich otáčať)
Zaujíma nás len existencia riešenia (nepotrebujeme poznať usporiadanie vyhovujúce podmienkam)

# MP hlavolam - riešenie

- je daný konečný počet kariet
- každú kartu môžeme umiestniťna konečný počet pozícií
- môžeme vyskúšať všetky možnosti
- $N = 25 \times 25$, počet možností $25 \times 24 \cdots \times 3 \times 2 \times 1$
- $10^9$ možností za sekundu $\Rightarrow$ 490 miliónov rokov

# Hranice praktickej použiteľnosti

| | Function \ $N$ | 20 | 60 | 100 | 300 | 1000 |
|---|---|---|---|---|---|---|
| Polynomial | $5N$ | 100 | 300 | 500 | 1500 | 5000 |
| | $N \times \log_2 N$ | 86 | 354 | 665 | 2469 | 9966 |
| | $N^2$ | 400 | 3600 | 10,000 | 90,000 | 1 million (7 digits) |
| | $N^3$ | 8000 | 216,000 | 1 million (7 digits) | 27 million (8 digits) | 1 billion (10 digits) |
| Exponential | $2^N$ | 1,048,576 | a 19-digit number | a 31-digit number | a 91-digit number | a 302-digit number |
| | $N!$ | a 19-digit number | an 82-digit number | a 161-digit number | a 623-digit number | unimaginably large |
| | $N^N$ | a 27-digit number | a 107-digit number | a 201-digit number | a 744-digit number | unimaginably large |


$10^{40}$
$10^{35}$
$10^{30}$
$10^{25}$
$10^{20}$
$10^{15}$
A trillion
A billion
A million
1000
100
10
$N^N$
$2^N$
$1.2^N$
$N^{10}$
$N^5$
$N^3$
$5N$
Number of nanoseconds since Big Bang
Number of nanoseconds in one day
2
4
8
16
32
64
128
256
512
1024
2048

# Hranice praktickej použiteľnosti

| | Function \ N | 20 | 40 | 60 | 100 | 300 |
|---|---|---|---|---|---|---|
| Polynomial | $N^2$ | 1/2500 millisecond | 1/625 millisecond | 1/278 millisecond | 1/100 millisecond | 1/11 millisecond |
| Polynomial | $N^5$ | 1/300 second | 1/10 second | 78/100 second | 10 seconds | 40.5 minutes |
| Exponential | $2^N$ | 1/1000 second | 18.3 minutes | 36.5 years | 400 billion centuries | a 72-digit number of centuries |
| Exponential | $N^N$ | 3.3 billion years | a 46-digit number of centuries | an 89-digit number of centuries | a 182-digit number of centuries | a 725-digit number of centuries |

hranica: polynomiálna zložitosť
prakticky riešiteľné vs prakticky neriešiteľné problémy

# Prakticky neriešiteľné problémy

- použiť výkonnejší počítač?
  *pre algoritmus zložitosti $2^N$: ak dnes vyriešime inštancie veľkosti max. $C$, tak 1000 krát rýchlejším počítačom vyriešime inštancie veľkosti max. $C + 9.97$.*
- zintenzívniť výzkum a nájsť efektívnejší algoritmus?
- dokázať, že neexistuje efektívnejší algoritmus?
  *Millenium Prize Problem, 1 000 000*
  `http://www.claymath.org/millennium/`
- je otázka dôležitá?
  *existujú aj iné (dôležité) problémy podobného charakteru?*

# Prakticky riešiteľné problémy

# NP-úplné problémy

## NP-úplné problémy

problémy, pre ktoré majú lineárny dolný odhad zložitosti a exponenciálny horný odhad zložitosti

*nepoznáme lepší než exponenciálny algoritmus a zároveň nevieme dokázať, či existuje alebo neexistuje asymptoticky efektívnejší algoritmus*

# NP-úplné problémy - príklady

dvojrozmerné pokrytie daných je $N$ štvoruholníkov; je možné pokryť nimi štvorcovú plochu?

Hamiltonovská cesta daný je neorientovaný graf; existuje v grafe cesta, ktorá navštívi každý vrchol práve jeden krát?

obchodný cestujúci daný je neorientovaný graf s ohodnotenými hranami a konštanta $K$; existuje v grafe Hamiltonovský cyklus dĺžky nanajvýš $K$?

problém rozvrhu daných je $M$ miestností a $N$ prednášok, každá prednáška má určený začiatok a koniec; je možné rozdeliť prednášky do daných miestností?

splniteľnosť daná je logická formula; existuje priradenie hodôt jej premenným, pre ktoré je formula splnená?

. . . a tisíce ďalších

# NP-úplné problémy - spoločná charakteristika

- rozhodovacie problémy (*odpoveď je „Áno“ alebo „Nie“*)
- existencia čiastočných riešení
- hľadanie riešenia problému pomocou zpaätného vyhľadávania (*backtracking*); exponenciálny algoritmus

- je extrémne ťažké rozhodnúť, či reišením vstupnej inštancie je „Áno“ aelbo „Nie“
- ak riešením inštancie je „Áno“, tak je veľmi jednoduché dokázať to — pomocou tzv. certifikátu
- obvykle je certifikát krátky reťazec (lineárny voči $N$) a jeho overenie je možné v polynomiálnom čase

# Alternatívna charakterizácia NP-úplných problémov

- predpokladajme, že máme magickú micu, ktorú budeme používať pri zpätnom vyhľadávaní (*backtrackovaní*) riešenia inštancie
- vždy, keď sa máme rozhodnúť, ako rozšíriť čiastočné riešenie, rozhodnutie urobíme tak, že si hodíme mincou
- „magično“ — minca vždy vyberie možnosť, ktorá vedie k riešeniu „Áno“ (samozrejme, len ak existuje)
- pojem nedeterminizmu
- pre NP-úplné problémy máme nedeterministické polynomiálne algoritmy
- NP v názve NP-úplný je skratka pre pre nedeterministický polynomiálny

# Pojem úplnosti

buď všetky NP-úplné problémy sú prakticky riešiteľné
alebo žiaden z nich nie je prakticky riešiteľný

- ak pre jeden NP-úplný problém skonštruujeme polynomiálny algoritmus, tak máme polynomiálne algoritmy pre všetky NP-úplné problémy
- ak pre niektorý NP-úplný problém dokážeme neexistenciu polynomiálneho algoritmu, tak polynomiálny algoritmus neexistuje pre žiaden NP-úplný problém

# Dôkaz úplnosti

## Polynomiálna časová redukcia

Pre dané dva rozhodovacie problémy $P_1$ a $P_2$; polynomiálna časová redukcia je algoritmus $\mathcal{A}$ taký, že

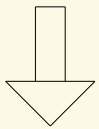
- $\mathcal{A}$ má polynomiálnu časovú zložitosť a
- vstupnú inštanciu $X$ problému $P_1$ transformuje na vstupnú inštanciu $Y$ problému $P_2$ takú, že riešením $X$ je „Áno“ vtedy a len vtedy, ak riešení $Y$ je tiež „Áno“.

# Polynomiálna redukcia - príklad

redukcia problému Hamiltonovskej cesty na problém obchodného cestujúceho

## Transformácia

graf $G = (V, H)$
(inštancia problému Hamiltonovskej cesty)



graf $\overline{G} = (\overline{V}, \overline{H})$ s ohodnotením $w : \overline{H} \to \mathbb{N}$ a konštanta $K$, kde
$\overline{V} = V$, $\overline{H} = V \times V$, $w(h) = 1$ pre všetky hrany $h \in H$,
$w(h) = 2$ pre všetky hrany $h \in \overline{H} \setminus H$, a $K = |V| + 1$
(inštancia problému obchodného cestujúceho)

- transformácia sa dá vypočítať v polynomiálnom čase
- v $G$ Hamiltonovská cesta existuje vtedy a len vtedy ak riešením inštancie $(\overline{G}, w, K)$ problému obchodného cestujúceho je „Áno“

# Polynomiálna redukcia a existencia algoritmu

Predpokladajme, že máme polynomiálny algoritmus $\mathcal{O}$ pre problém obchodného cestujúceho.
Ukážeme, ako za tohto predpokladu skonštruujeme polynomiálny algoritmus $\mathcal{H}$ pre problém Hamiltonovskej cesty.

Algoritmus $\mathcal{H}$

1. vstup $G$ transformuj na $(\overline{G}, w, K)$
2. aplikuj $\mathcal{O}$ na $(\overline{G}, w, K)$
3. ak riešením $(\overline{G}, w, K)$ je „Áno“, tak vráť odpoveď „Áno“
4. v opačnom prípade vráť odpoveď „Nie“

# Polynomiálna reduckia a úplnosť

**Fakt**

Všetky NP-úplné problémy sú vzájomne polynomiálne redukovateľné

- ak chceme o probléme $R$ dokázať, že je NP-úplný, nemusíme ukazovať redukciu medzi $R$ a vetkými ostatnými NP-úplnými problémami
- stačí ukázať polynomiálnu redukciu problému $R$ na jeden konkrétny NP-úplný problém
- redukcia je tranzitívna

**Cookova veta**

Problém splniteľnosti je NP-úplný.

# Polynomiálna redukcia - príklad 2

Redukcia 3-zafarbenia mapy na problém splniteľnosti

# P=NP? problém

P je trieda prakticky riešiteľných problémov (tj. problémov, pre ktoré existujú polynomiálne algoritmy)

**Fakt**

$$P \subseteq NP$$

**Otvorený problém**

$$P = NP\ ?$$

Dôsledky riešenia problému.

# Čiastočné riešenie NP-úplných problémov

- pseudopolynomiálne algoritmy
- aproximatívne algoritmy
- náhodnostné algoritmy
- kvantové algoritmy
- genetické algoritmy

# Prakticky riešiteľné problémy

# Ešte ťažšie problémy

- NP-úplné problémy môžu mať polynomiálne algoritmy
- existujú problémy, ktoré dokázateľne nemôžu mať efektívnejšie než exponenciálne (prípadne ešte zložitejšie) algoritmy

*Príklady: pravdivosť formule v bohatšej logike*

# Priestorová zložitosť

- časová zložitosť $\geq$ priestorová zložitosť
- polynomiálny priestor (PSPACE)
- PSPACE-úplné problémy

# Teória výpočtovej zložitosti

- pojem zložitostnej triedy
- vzťahy medzi zložitostnými triedami
- LOGTIME $\subseteq$ LOGSPACE$\subseteq$ PTIME $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ EXPSPACE . . .
- analogicky pre nedeterminizmus
- kľúčová otázka presného vzťahu

$$P \subseteq NP \subseteq PSPACE$$

# Nerozhodnuteľnosť

*Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software.* *Time Magazin, April 1984*

# Otázka

- existujú algoritmické problémy, ktoré sú prakticky neriešiteľné?
- je to len otázka dostatočne dlhého času, výkonného hardwaru resp. sofistikovaných algoritmov ???
- alebo existujú problémy, ktoré sú principiálne neriešiteľné ???

# Pravidlá

- uvažujeme algoritmické problémy (tj. problémy určené svojou množinou vstupných inštancií a požadovaným vzťahom medzi vstupom a výstupom)
- problémy s nekonečnou množinou vstupných inštancií (v opačnom prípade pre problém vždy existuje algoritmus založený na vymenovaní všetkých vstupov a k ním príslušných výstupov)
- algoritmus = program zapísaný v programovacom jazyku vyššej úrovne

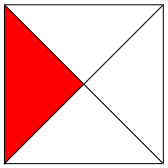
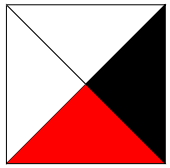
# Príklad - domino

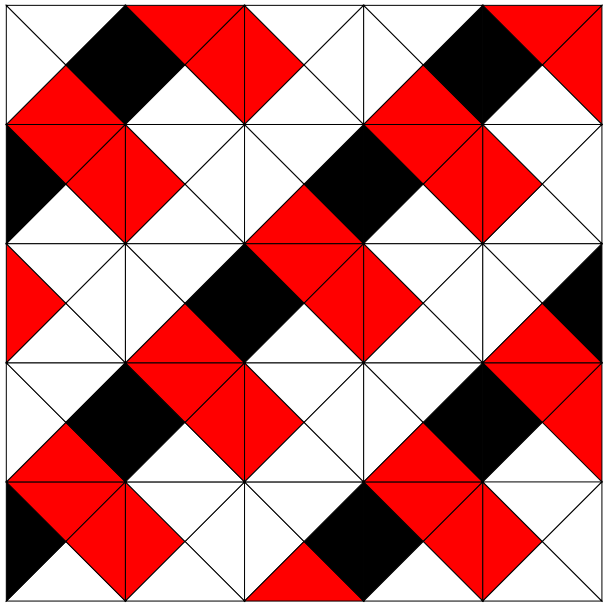
Vstup (konečná) množina $T$ typov dlaždíc s farebnými hranami

Otázka je možné dlaždicami pokryť ľubovoľne veľkú plochu tak, aby sa dlaždice vždy dotýkali hranami rovnakej farby?

*z každého typu je k dispozícii neobmedzený počet dlaždíc*

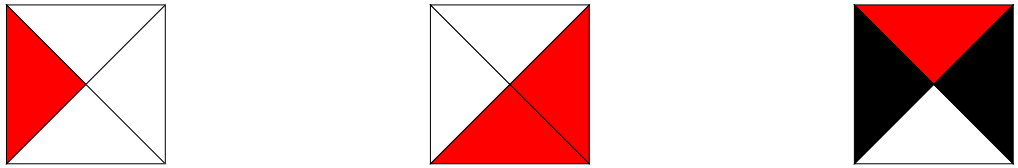
# Príklad - domino, inštancia 1

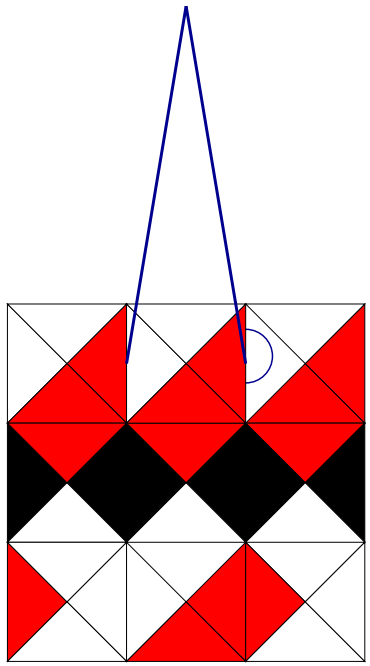



riešením inštancie 1 je „Áno“

# Príklad - domino, inštancia 2





riešením inštancie 2 je „Nie“

# Nerozhodnuteľné problémy

**Definícia**

Problém, pre ktorý neexistuje žiaden algoritmus, sa nazýva nerozhodnuteľný (algoritmicky neriešiteľný).

prakticky riešiteľné problémy

prakticky neriešiteľné problémy

nerozhodnuteľné problémy

# Príklad - domino

**Fakt**

Problém domina je nerozhodnuteľný

- „zdroj“ nerozhodnuteľnosti
- ekvivalencia s problémom pokrytia nekonečnej plochy
- periodicita riešenia
- je dôvodom potenciálne nekonečný počet možností?
- dominový had a jeho varianty

# Príklad - Postov korešpondečný problém (PKP)

Vstup dva zoznamy slov $X = (x_1, \ldots x_n)$ a $Y = (y_1, \ldots y_n)$

Otázka existuje konečná postupnosť indexov taká, že spojením príslušných slov zoznamu $X$ vznikne rovnaké slovo ako spojením príslušných slov zoznamu $Y$?

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $X$ | $abb$ | $a$ | $bab$ | $baba$ | $aba$ |
| $Y$ | $bbab$ | $aa$ | $ab$ | $aa$ | $a$ |

*riešením inštancie je „Áno“, príkladom postupnosti je 2, 1, 1, 4, 1, 5*

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $X$ | $bb$ | $a$ | $bab$ | $baba$ | $aba$ |
| $Y$ | $bab$ | $aa$ | $ab$ | $aa$ | $a$ |

*riešením ištancie je „nie“*

# Príklad - ekvivalencia a verifikácia programov

## Ekvivalencia

**Vstup** dva programovacie jazyky vyššej úrovne, ktorých syntax je daná syntaktickým diagramom alebo v BNF

**Otázka** sú množiny syntakticky správnych programov pre oba jazyky zhodné?

## Verifikácia

**Vstup** popis algoritmického problému a program v programovacom jazyku vyššej úrovne

**Otázka** rieši program daný problém? (tj. odpoveď je „Áno“ ak pre každú vstupnú inštanciu problému sa výpočet programu zastaví a dá správnu odpoveď)

*Pre oba problémy závisí nerozhodnuteľnosť na voľbe programovacieho jazyka resp. na voľbe jazyka pre popis algoritmického problému. Pre bežné jazyku sú oba problémy nerozhodnuteľné.*

# Príklad - problém zastavenia

Uvažujme algoritmy $A$ a $B$ s množinou vstupných inštancií $\mathbb{N}$

Algoritmus $A$

**while** $X \neq 1$ **do** $X \leftarrow X - 2$ **od**

**return** $X$

Algoritmus $B$

**while** $X \neq 1$ **do**

    **if** $X$ je sudé **do** $X \leftarrow X/2$

    **if** $X$ je liché **do** $X \leftarrow 3X + 1$ **od**

**return** $X$

Algoritmus $A$ pre sudé čísla neskončí. O algoritme $B$ nie je známe, či skončí pre všetky vstupné inštancie.

# Príklad - problém zastavenia

| | |
|---|---|
| Vstup | program $R$ v programovacom jazyku $L$ vyššej úrovne a množina vstupných inštancií programu $R$ |
| Otázka | zastaví sa výpočet $R$ pre každú vstupnú inštanciu? |

Problém zastavenia je nerozhodnuteľný.

*Notácia: $R(x)\downarrow$ označuje, že výpočet $R$ na $x$ skončí; symbol $R(x)\uparrow$ označuje, že neskončí.*

# Riceova veta

Rozhodnuteľnosť je výnimka, ktorá potvrdzuje pravidlo

Zaujíma nás netriviálna vlastnosť programu, ktorá je
(1) pravdivá pre niektoré a nepravdivá pre ostané programy
(2) nie je viazaná na syntax programu, ale vzťahuje sa k problému, ktorý program rieši.

**Riceova veta**

Všetky netriviálne vlastnosti programov sú nerozhodnuteľné.

Príklady: je výstupom programu vždy „Áno“?, zastaví program pre každý vstup?

# Nerozhodnuteľnosť

# Dôkaz nerozhodnuteľnosti

Analógia s dôkazom NP-úplnosti
(1) dokážeme nerozhodnuteľnosť nejakého problému
(2) nerozhodnuteľnosť ďalších problémov dokazujeme metódou redukcie

V prípade nerozhodnuteľnosti použijeme redukciu, ktorá nemusí byť polynomiálne časovo ohraničená.

# Redukcia

**Redukcia**

Pre dané dva rozhodovacie problémy $P_1$ a $P_2$; redukcia je algoritmus $\mathcal{A}$ ktorý vstupnú inštanciu $X$ problému $P_1$ transformuje na vstupnú inštanciu $Y$ problému $P_2$ takú, že riešením $X$ je „Áno“ vtedy a len vtedy, ak riešením $Y$ je tiež „Áno“.

# Redukcia - príklad

redukcia problému zastavenia na problém verifikácie

# Redukcia a dôkaz nerozhodnuteľnosti

**Fakt**

Ak $P_1$ sa redukuje $P_2$ a $P_1$ je nerozhodnuteľný, tak aj $P_2$ je nerozhodnuteľný.

Predpokladajme, že $P_2$ je rozhodnuteľný a že $B$ je algoritmus, ktorý ho rieši.
Pomocou $B$ skonštruujeme algoritmus pre $P_1$. Konrétne, vstupnú inštanciu $X$ problému $P_1$ prevedieme pomcou algoritmu redukcie na vstupnú inštanciu $Y$ problému $P_2$. Použijeme algoritmus $B$ a nájdeme riešenie $Y$. Ak riešením $Y$ je „Áno“ tak riešením $X$ je tiež „Áno“. Ak riešením $Y$ je „Nie“ tak riešením $X$ je tiež „Nie“.
To je ale spor s predpokladom, že $P_1$ je nerozhodnuteľný.

# Nerozhodnuteľnosť problému zastavenia

**Tvrdenie**

Neexistuje program v $L$, ktorý pre ľubovoľnú dvojicu $\langle R, X \rangle$ ($R$ je syntakticky správny program v $L$ a $X$ je symbol reťazcov), dá na výstup „Áno“ práve ak výpočet $R$ pre vstup $X$ skončí po konečnom počte krokov a dá na výstup „Nie“ v opačnom prípade.

Neexistenciu programu požadovaných vlastností dokážeme sporom.

# Nerozhodnuteľnosť problému zastavenia

Predpokladajme, že existuje program požadovaných vlastností, nazvime ho $Q$.

Skonštruujeme nový program v jazyku $L$, nazvime ho $S$, nasledovne:

1. vstupom programu $S$ je syntakticky správny program $W$ v jazyku $L$
2. program $S$ prečíta $W$ a vytvorí kópiu $W$
3. $S$ aktivuje program $Q$ so vstupom $\langle W, W \rangle$ (volaním $Q$ ako procedúry)
4. výpočet $Q$ na vstupe $\langle W, W \rangle$ skončí (z predpokladu o vlastnostiach $Q$) a vráti $S$ odpoveď („Áno“ alebo „Nie“).
5. ak $Q$ skončí s odpoveďou „Áno“, tak $S$ vstúpi do nekonečného cyklu (jeho výpočet sa nezastaví)
6. ak $Q$ skončí s odpoveďou „Nie“, tak $S$ dá na výstup „Áno“.

# Nerozhodnuteľnosť problému zastavenia - spor

Z konštruckie programu $S$ je zrejmé, že $S$ sa pre každý vstup $W$ zastaví (s odpoveďou „Áno“) alebo jeho výpočet cyklí donekonečna.

Uvážme výpočet $S$ na vstupe $W = S$. $S$ aktivuje program $Q$ na vstupe $\langle S, S \rangle$ (bod 3). Podľa predpokladu $Q$ zastaví. Sú dve možnosti:

1. $Q$ skončí s odpoveďou „Áno“ (tj. áno, program $S$ sa na vstupe $S$ zastaví); v takomto prípade ale $S$ vstúpi do nekonečného cyklu. Dostávame spor.
2. $Q$ skončí s odpoveďou „Nie“ (tj. nie, program $S$ sa na vstupe $S$ nezastaví ); v takomto prípade ale $S$ dá odpveď „Áno“. Opäť dostávame spor.

# Metóda diagonalizácie

- Georg Cantor
- obecná metóda, postavená na princípe rozdielnych kardinalít
- dôkaz, že reálnych čísel je viac ako racionálnych; dolné odhady zložitosti problémov, ...

# Konečné certifikáty pre nerozhodnuteľné problémy

Analógia s NP-úplnými problémami. V prípade nerozhodnuteľných problémov požadujeme od certifikátov len konečnosť a existenciu algoritmu ktorý overí, či daný reťazec je certifikátom.

**PKP** certifikátom odpovede „Áno“ je konkrétna, konečná postupnosť indexov; ľahko overíme, či daná postupnosť indexov má požadovanú vlastnosť

**problém zastavenia** certifikátom je samotný konečný výpočet; jednoducho overíme, či daná postupnosť krokov je korektným výpočtom programu na vstupe.

**hadové domino** certifikátom je postupnosť dlaždíc a spôsob ich skladania

**domino** certikát existuje pre odpoveď nie. Certifikátom je konečná plocha $E$. Pretože $E$ je konečný a počet typov dlaždíc je tiež konečný, dokážeme overiť, že $E$ sa nedá pokryť (preveríme všetky možnosti).

# Čiastočne rozhodnuteľné problémy

Všetky predchádzajúce problémy mali certifikát pre jeden typ odpovede; hovoríme o tzv. jednosmernom cerfikáte.

**Definícia**

Nerozhodnuteľné problémy, ktoré majú jednosmerný certifikát, sa nazývajú čiastočne rozhodnuteľné.

Čiastočne rozhodnuteľné problémy majú algoritmus, ktorý je korektný pre jeden typ vstupných inštancií (tj. buď pre vstupy s odpoveďou „Áno“, alebo pre vstupy s odpoveďou „Nie“). Algoritmus systematicky overuje všetky konečné reťazce, či sú certifikátom daného vstupu.

# Dvojsmerné certifikáty

- môže nastať situácia, keď pre daný problém máme certifikát ako pre odpoveď „Áno“, tak aj (iný) certifikát pre odpoveď „Nie“ (hovoríme o tzv. dvojcestnom certifikáte)?
- príklad: problém Hamiltonovského cyklu. Certifikátom pre „Áno“ vstup je permutácia vcholov (jednoducho overíme, či tvorí Hamiltonovský cyklus). Certifikátom pre „Nie“ vstup sú všetky permutácie vrcholov (jednoducho overíme, že žiadna permutácia netvorí Hamiltonovský cyklus).

**Fakt**

Ak problém má dvojcestný certifikát, tak je rozhodnuteľný.

Algoritmus systematicky a striedavo overuje všetky konečné reťazce či sú certifikátom pre daný vstup. Konečnosť je zaručená, pretože každý vstup má svoj certifikát.

# Ešte ťažšie problémy?

- všetky čiastočne rozhodnuteľné problémy sú vzájomne redukovateľné (*analógia s NP-úplnými problémami, tkroé sú polynomiálne ekvivalentné*)
- existujú problémy, ktoré sú ťažšie než NP-úplné; platí analógia aj pre čiastočne rozhodnuteľné problémy?

problém verifikácie existuje redukcia problému zastavenia na problém verifikácie; opačná redukcia ???

úplný problém zastavenia (je výpočet programu konečný pre všetky vstupné inštancie?) Existuje redukcia problému zastavenia na úplný problém zastavenia; opačná redukcia ???

Problém verifikácie ani úplný problém zastavenia nemajú certifikáty

# Stupne nerozhodnuteľnosti

- analogicky ako rozhodnuteľné problémy môžeme zoskupiť do rôznych zložitostných tried, tak aj nerozhodnuteľné problémy tvoria úrovne podľa stupňa nerozhodnuteľnosti
- každá úroveň obsahuje problémy, ktoré sú ešte ťažšie než problémy predchádzajúcej úrovne
- prvé dve úrovne hierarchie tvoria čiastočne rozhodnuteľné a nerozhodnuteľné problémy
- ďalšie úrovne označujeme súhrnne ako vysoko nerozhodnuteľné problémy

# Vysoko nerozhodnuteľné problémy - príklad

modifikácia problému domina kde požadujeme, aby v nekonečnom pokrytí bola vopred špecifikovaná dlaždica použitá nekonečne veľa krát

# Univerzalita a robustnosť

# Jednoduchý výpočtový model

Hľadáme čo najjednoduchší počítač (výpočtový model), ktorý je schopný realizovať všetky algoritmické výpočty.

Prečo?

- aký najjednoduchší model je schopný realizovať všetky výpočty?
- obecnosť výsledkov o praktickej neriešiteľnosti a nerozhodnuteľnosti
- presná formulácia a formálne dôkazy tvrdení týkajúcich sa praktickej neriešiteľnosti a nerozhodnuteľnosti

# Jednoduchý výpočtový model - dáta

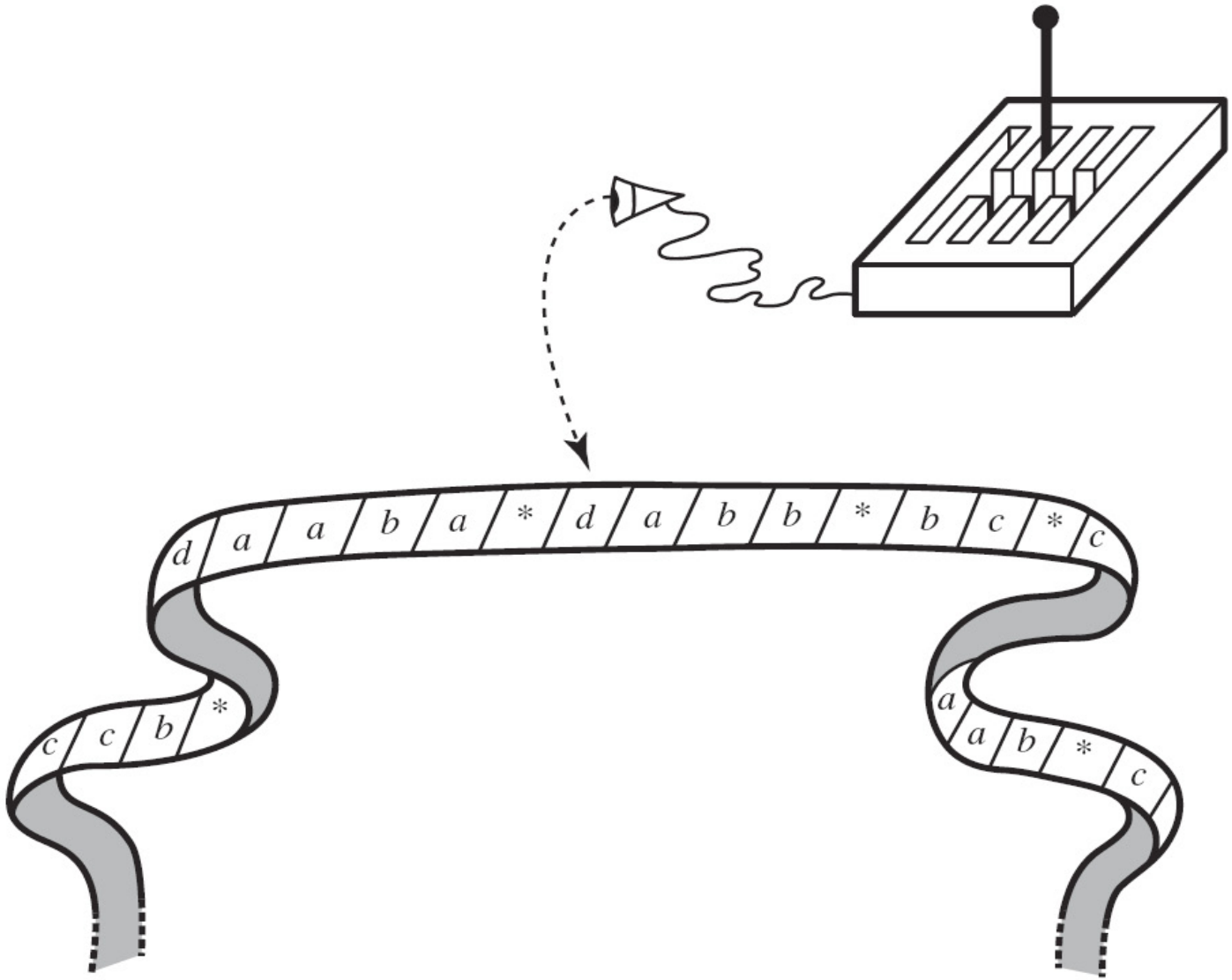
- dáta = reťazce symbolov
- počet rôznych symbolov potrebných na zakódovanie reťazcov je konečný (*podobne ako na zakódovanie všetkých čísel nám stačí v desiatkovej resp. binárnej sústave 10 resp. 2 číslice*)
- dáta môžeme zapisovať na jednorozmernú pásku, ktorá obsahuje políčka; na každom políčku je zapísaný jeden symbol, ktorý je prvkom vstupnej abecedy

## Linearizácia dátových štruktúr

- linearizácia zoznamu
- linearizácia dvojrozmerného poľa, matice
- linearizácia stromu

## Dynamické dátové štruktúry a neohraničenosť jednosmernej pásky

# Jednoduchý výpočtový model - riadiaca jednotka



# Jednoduchý výpočtový model - riadiaca jednotka

- výpočet je realizovaný riadiacou jednotkou
- riadiaca jednotka je vždy v jednom z konečne veľa rôznych stavov (*stav zodpovedá inštrukcii algoritmu*)
- riadiaca jednotka vždy snímá práve jedno políčko jednorozmernej pásky
  (*hodnota, s ktorou manipuluje inštrukcia algoritmu*)
- atomické akcie
  - prečítanie symbolu z políčka pásky
  - zápis symbolu na políčko pásky
  - posun o jedno políčko na páske
  - zmena stavu riadiacej jednotky
- závislosť zmeny na aktuálnych hodnotách

# Jednoduchý výpočtový model - základné operácie

jeden krok výpočtu

- prečítanie symbolu
- podľa aktuálneho stavu riadiacej jednotky a prečítaného symbolu sa vykoná
  - zmena stavu
  - zápis symbolu
  - posun o jedno políčko vpravo alebo vľavo

**Turingov stroj**

Alan Turing, 1936

# Univerzalita a robustnosť

# Turingov stroj

Turingov stroj (TS) pozostáva z

- (konečnej) množiny stavov
- (konečnej) abecedy symbolov
- nekonečnej pásky rozdelenej na políčka
- čítacej a zapisovacej hlavy, ktorá sa pohybuje po páske a sníma vždy 1 políčko pásky
- prechodového diagramu

# Turingov stroj - prechodový diagram

Prechodový diagram

- orientovaný graf
- vrcholy grafu sú stavy TS
- hrana z vrcholu $s$ do vrcholu $t$ reprezentuje prechod a je označená dvojicou tvaru $\langle a/b, L\rangle$ alebo $\langle a/b, R\rangle$;
  - $a$ je symbol, ktorý hlava TS z pásky číta (tzv. spínač)
  - $b$ je symbol, ktorý na pásku zapisuje
  - $L$ resp. $R$ určuje smer pohyb hlavy doľava resp. doprava
- požadujeme, aby diagram bol jednoznačný (deterministický Turingov stroj), tj. zo stavu nesmú vychádzať dve hrany s rovnakým spínačom
- jeden zo stavov je označený ako štartovný (počiatočný) stav (označený šípkou)
- niektoré zo stavov sú označene ako koncové stavy (označené výrazným ohraničením)

# Turingov stroj - výpočet

## Krok výpočtu

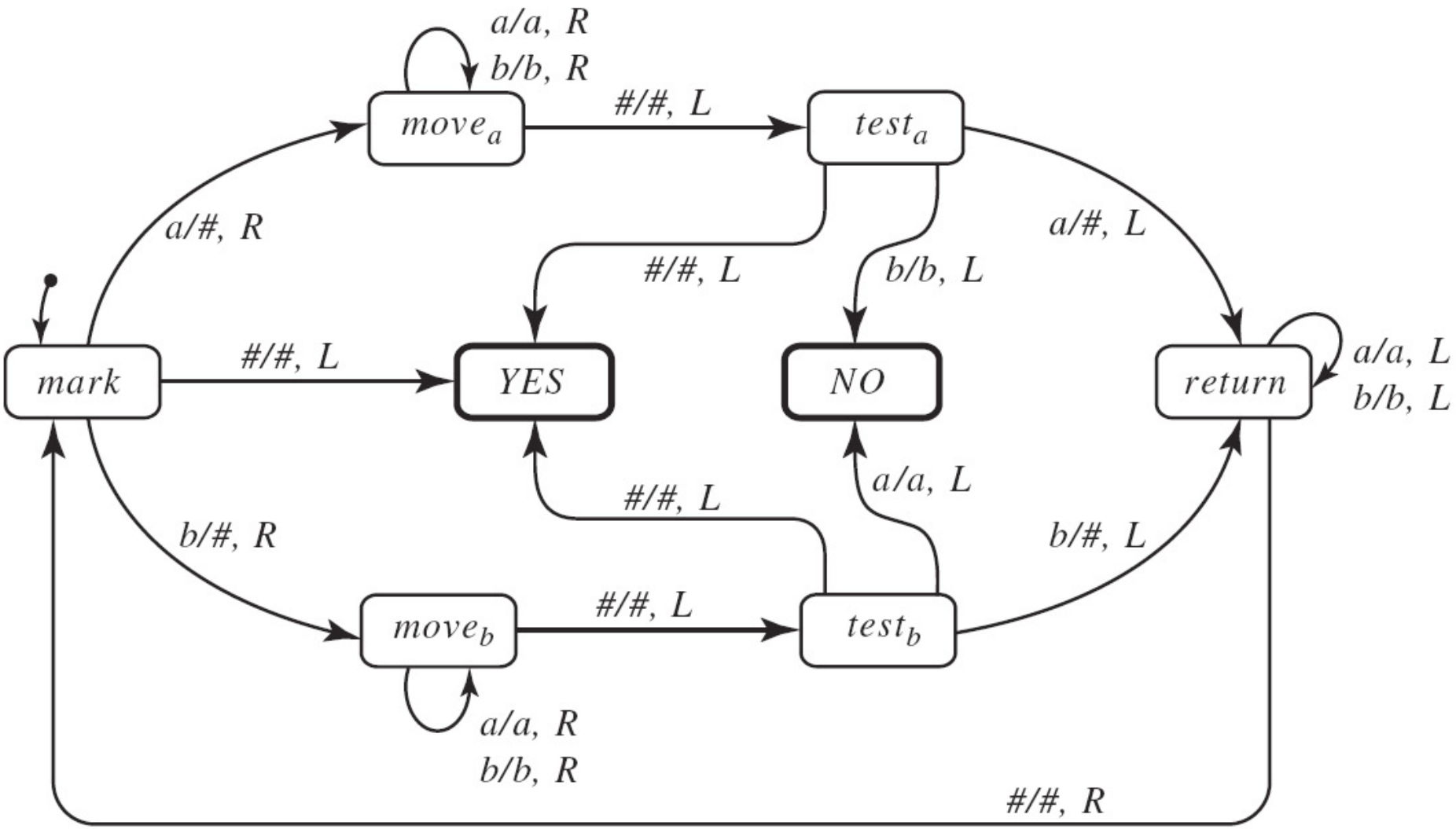
prechod z $s$ do $t$ označený $\langle a/b, L\rangle$ v prechodovom diagrame

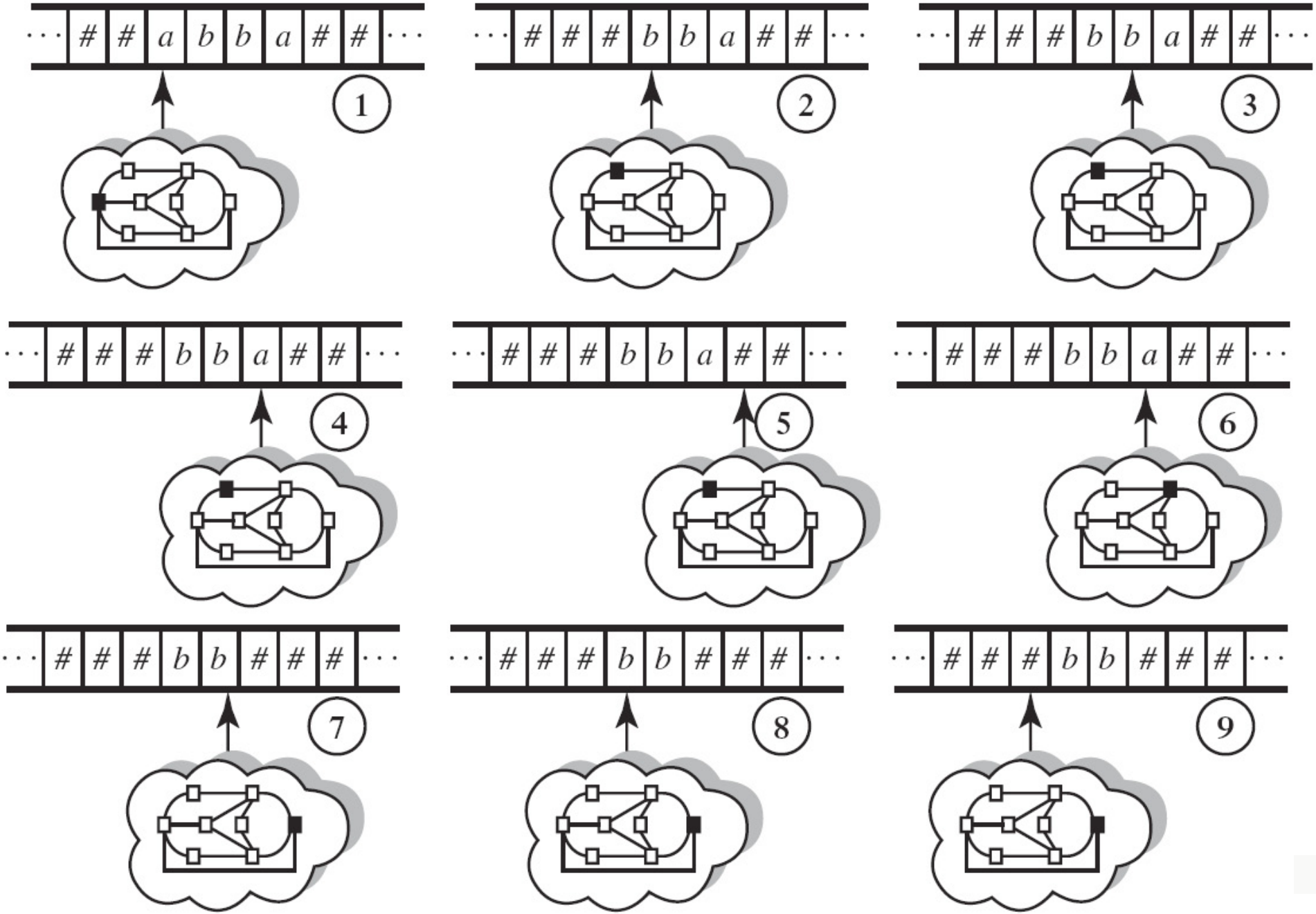
ak riadiaca jednotka TS je v stave $s$ a hlava číta symbol $a$, tak hlava prepíše symbol $a$ symbolom $b$, posunie sa o 1 políčko doľava a stav riadiacej jednotky sa zmení na $t$
(*analogicky pre* $\langle a/b, R\rangle$ *a pohyb vpravo*)

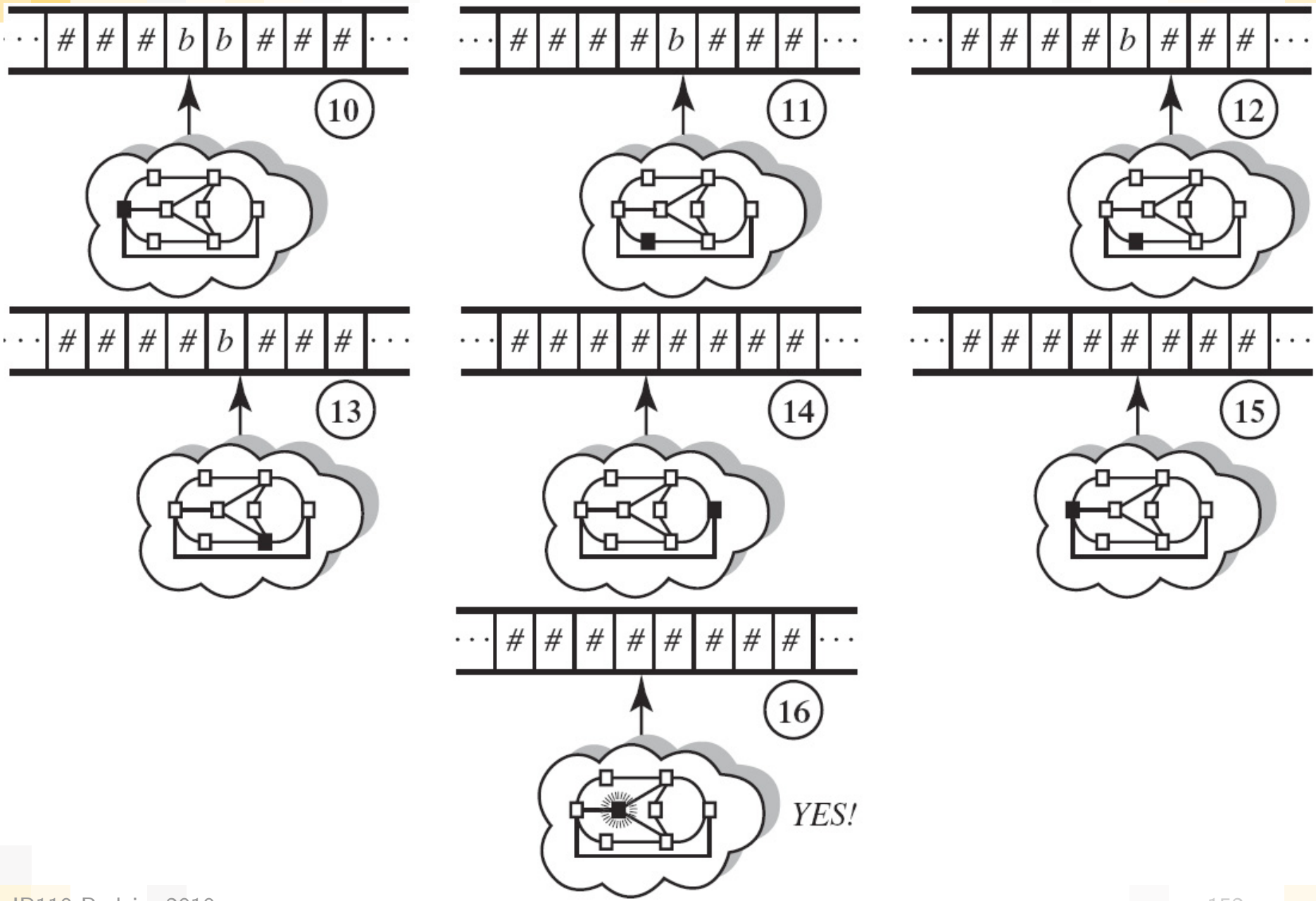
## Výpočet

Výpočet začína v počiatočnom stave na najľavejšom neprázdnom políčku pásky.
Výpočet prebieha krok po kroku tak, ako predpisuje prechodový diagram.
Výpočet sa zastaví keď dosiahne niektorý z koncových stavov.

# Turingov stroj pre palindrómy




··· # # a b b a # # ···
1
··· # # # b b a # # ···
2
··· # # # b b a # # ···
3
··· # # # b b a # # ···
4
··· # # # b b a # # ···
5
··· # # # b b a # # ···
6
··· # # # b b # # # ···
7
··· # # # b b # # # ···
8
··· # # # b b # # # ···
9


10
11
12
13
14
15
16
YES!

# Simulátor Turingových strojov

http://www.fi.muni.cz/ xbarnat/tafj/turing

# Turingov stroj ako algoritmus

- Turingov stroj môžeme chápať ako počítač s jedným, fixovaným programom
- softwarom je prechodový diagram; hardwarom je riadiaca jednotka a páska
- jednotlivé TS sa líšia iba svojim softwarom, preto často hovoríme o *programovaní* Turingovho stroja

# Turingov stroj ako algoritmus

- Turingov stroj môžeme naprogramovať tak, aby riešil rozhodovací problém
- pre rozhodovací problém $P$, ktorého množina vstupných inštancií je kódovaná ako množina linearizovaných reťazcov, konštruujeme Turingov stroj $M$ s počiatočným stavom $s$ a dvoma končovými stavmi $YES$ a $NO$

pre každý vstup $X$, ak $M$ začne výpočet v stave $s$ na najľavejšom symbole reťazca $X$, tak $M$ skončí výpočet v stave $YES$ a $NO$ v závislosti na tom, či výstupom $P$ pre $X$ je „Áno“ alebo „Nie“

# Turingov stroj ako algoritmus

Turingove stroje môžu byť naprogramované aj pre riešenie iných než rozhodovacích problémov.

V takomto prípade predpokladáme, že keď sa TS zastaví (prejde do koncového stavu), tak výstupom je reťazec zapísaný na páske medzi dvoma špeciálnymi znakmi (napr. !)

Ak sa výpočet zastaví a na páske je iný počet symbolov ! ako 2, chápeme výpočet ako neukončený (tj. ako výpočet, ktorý cyklí donekonečna).

# Univerzalita a robustnosť

# Churchova Turingova hypotéza

Aké problémy sú riešiteľné pomocou vhodne naprogramovaného TS?

## Churchova Turingova hypotéza

Každý algoritmický problém, pre ktorý existuje program v nejakom programovacom jazyku vyššej úrovne a je riešiteľný na nejakom hardwaru, je riešiteľný aj na Turingovom stroji.

### Prečo hypotéza?

CT hypotéza formuluje vzťah medzi dvoma konceptami:

- matematicky presný pojem riešiteľnosti na Turingovom stroji a
- neformálny koncept algoritmickej riešiteľnosti, ktorý je postavený na pojmoch „programovací jazyk vyššej úrovne“, „program v programovacom jazyku“

# Argumenty pre Churchovovu Turingovu hypotézu

CT hypotézu formulovali v 30-tych rokoch nezávisle Alonso Church a Alan Turing.
Od tej doby bolo navrhnutých množstvo „univerzálnych“ modelov (*absolútnych, schopných riešiť všetky mechanicky riešiteľné problémy*)

- Turingove stroje (*Alan Turing*)
- lambda kalkulus (*Alonso Church*)
- produkčné systémy (*Emil Post*)
- rekurzívne funkcie (*Stephen Kleene*)
- kvantové počítače
- ...

**Fakt**

O všetkých navrhnutých formalizmoch je dokázané, že sú ekvivalentné v tom zmysle, že určujú zhodnú triedu algoritmicky riešiteľných problémov.

# Dôsledky Churchovej Turingovej hypotézy

- extrémne výkonné superpočítače nie sú silnejšie než malé počítače s jednoduchým programovacím jazykom; za predpokladu neohraničeného času a veľkosti pamäte dokážu obidva riešiť tie isté algoritmické problémy
- pojem algoritmicky riešiteľného (rozhodnuteľného) problému je robustný, tj. je nezávislý na konkrétnej voľbe výpočtového modelu resp. programovacieho jazyka
- CT hypotéza podporuje správnosť definície nerozhodnuteľných problémov

# Univerzalita a robustnosť

# Modifikácie Turingovho stroja

Argumentom podporujúcim CT hypotézu je aj robustnosť TS.

Modifikácie

- TS, ktorý má po skončení výpočtu zapísaný na páske len vstupný a výstupný reťazec
- TS s jednosmerne nekonečnou páskou
- TS s dvojrozmernou páskou
- TS s konečným počtom pások (každá más svoju čítaciu/zapisovaciu hlavu)

**Fakt**

Všetky uvedené modifikácie sú vzájomne ekvivalentné

**Dôkaz**

technikou simulácie: ukážeme, že výpočet jedného zariadenia sa dá simulovať na druhom zariadení a naopak

# Programy s počítadlami (Counter Programs, CP)

- programy manipulujú s prirodzenými číslami uloženými v premenných
- tri elementárne operácie

  $X \leftarrow 0$ priradí premennej hodnotu 0

  $X \leftarrow Y + 1$

  $X \leftarrow Y - 1$ ak hodnota $Y$ je 0, tak $X$ priradí hodnotu 0
- jeden elementárny riadiaci príkaz

  $$\textbf{if}\, X = 0\ \textbf{goto}\ G,$$

  kde $X$ je premenná a $G$ je návestie pripojené k príkazu

# Programy s počítadlami - príklad

$U \leftarrow 0$
$Z \leftarrow 0$
A : **if** $X = 0$ **goto** G
$X \leftarrow X - 1$
$V \leftarrow Y + 1$
$V \leftarrow V - 1$
B : **if** $V = 0$ **goto** A
$V \leftarrow V - 1$
$Z \leftarrow Z + 1$
**if** $U = 0$ **goto** B

vykonanie **goto** G je ekvivalentom úspešného ukončenia výpočtu

# Turingove stroje a programy s počítadlami

TS manipulujú so symbolmi, PS s číslami

je možná ich vzájomná simulácia?

# Simulácia TS na CP

obsah pásky $\longrightarrow$ čísla

pre jednoduchosť predpokladajme, že abeceda TS má desať znakov
znaky očíslujeme a reťazce znakov prevedieme na čísla

*Príklad*

$$\begin{array}{lllll} \# - 0 & ! - 1 & * - 2 & a - 3 & b - 4 \\ c - 5 & d - 6 & e - 7 & f - 8 & g - 9 \end{array}$$

*reťazec*

$$\ldots \# \# \; a \; b \; * \; e \; b \; ! \; \# \; a \; g \; a \; \# \; \# \ldots$$

*v ktorom je snímaný symbol b, prevedieme na dvojicu čísel*

$$3427 \; a \; 393014$$

# Simulácia TS na CP

zmena symbolu na páske

zmena poslednej číslice v druhom čísle

*Príklad ak symbol b prepíšeme symbolom g, tak číslo* 393014 *sa zmení na* 393019, *PC pripočíta 5 krát hodnotu 1*

posun hlavy doprava

prvé číslo vynásobíme 10 a pripočítame k nemu poslednú číslicu druhého čísla

druhé číslo vydelíme 10 (celočíselne)

analogicky pre posun hlavy doľava

zmena stavu

stavu TS zodpovedá skupina inštrukcií CP; zmena stavu je simulovaná skokom na novú skupinu inštrukcií (príkaz $\mathrm{goto}$)

CP, ktorý simuluje TS, má len dve počítadlá!

# Simulácia CP na TS

čísla ⟵ symboly

hodnota každej premennej je zapísaná ako postupnosť symbolov = číslic; jednotlivé hodnoty sú vzájomne oddelené špeciálnym symbolom (napr. *)

inštrukcie ⟵ stavy

každej inštrukcii programu zodpovedá skupina stav TS, vykonenie inštrukcie je simulované prechodom do príslušného stavu a realizácia postupnosti krokov, ktoré potrebným spôsobom upravia obsah premennej

# Simulácie ako redukcie

Ak model A simuluje model B, tak máme redukciu medzi týmito modelmi.

Redukcia prevedie program modelu A a jeho vstup $X$ na program modelu $B$ a jeho vstup $Y$.

CT hypotéza ukazuje, že naše úvahy pri konštrukcii redukcií boli korektné.

# Simulácie ako redukcie

Ak model A simuluje model B, tak máme redukciu medzi týmito modelmi.

Redukcia prevedie program modelu A a jeho vstup $X$ na program modelu $B$ a jeho vstup $Y$.

CT hypotéza ukazuje, že naše úvahy pri konštrukcii redukcií boli korektné.

*Príklad nerozhodnuteľnosť problému zastavenia*

1. *formálne dokážeme nerozhodnuteľnosť problému pre TS*
2. *podľa CT hypotézy problém zastavenia nemôže byť rozhodnuteľný ani pre žiaden iný programovací jazyk vyššetj úrovne (je ekvivalentý TS!)*

**Fenomén**

algoritmus, ktorého vstupom je iný algoritmus

# Univerzalita a robustnosť

# Univerzálny algoritmus

jeden z najdôležitejších dôsledkov CT hypotézy

Existencia univerzálneho algoritmu, ktorý má schopnosť chovať sa ako akýkoľvek iný algoritmus.

- vstupom pre univerzálny algoritmus je popis akéhokoľvek algoritmu $A$ a akéhokoľvek jeho vstupu $X$
- univerzálny algoritmus simuluje výpočet $A$ na $X$
- výpočet univerzálneho algoritmu sa zastaví práve ak výpočet $A$ na $X$ sa zastaví; ako výstup poskytne univerzálny algoritmus presne tú istú odpoveď ako poskytne $A$ na $X$

*ak fixujeme algoritmus $A$ a meníme $X$, tak univerzálny algoritmus sa chová presne ako algoritmus $A$*

# Univerzálny algoritmus

- môže byť vstupom univerzálneho algoritmu program v akomkoľvek programovacom jazyku?
- využijeme CT hypotézu a poznatok o ekvivalencii všetkých známych formalizmov pre popis algoritmov
- ku konštrukcii univerzálneho algoritmu potrebujeme len jazyk $L_1$, v ktorom napíšeme program $U$ pre univerzálny algoritmus; program akceptuje ako vstup ľubovoľný program napísaný vo fixovanom konkrétnom jazyku $L_2$
- program $U$ je nezávislý na výbere modelu, pretože podľa CT hypotézy
  1. môže byť napísaný v akomkoľvek jazyku
  2. dokáže simulovať akýkoľvek algoritmus popísaný v akomkoľvek jazyku

*Vhodným kandidátom pre jazyky $L_1$ a $L_2$ sú Turingove stroje.*

# Univerzálny Turingov stroj

- potrebujeme popísať Turingov stroj ako lineárny reťazec nad konečnou abecedou symbolov
- stačí linearizovať prechodový diagram
- *mark ** mark YES* $\langle \#/\#, L \rangle$ ** mark* $move_a$ $\langle a/\#, R \rangle$ **  $move_a$ $move_a$* $\langle a/a/, R \rangle$ **. . .*
- linearizovaný prechodový diagram prevedieme štandartným spôsobom na reťazec nad fixovanou abecedou (napr. binárnou)
- podobne linearizujeme a kódujeme aj vstup simulovaného TS
- samotný program univerzálneho TS je jednoduchý svojim princípom: uchováva si aktuálny stav simulovaného TS, obsah jeho pásky a čítaný symbol; z linearizovaného popisu simulovaného TS odvodí, aké akcie sa majú realizovať v ďalšom kroku výpočtu simulovaného TS

# Univerzálny program s počítadlami

- vstupom je dvojica čísel; prvé číslo je kódom nejakého programu s počítadlami, druhé číslo je kódom jeho vstupu
- univerzálny program je možné skonštruovať tak, aby využíval len dve počítadlá

# Univerzalita a robustnosť

# Modifikované programy s počítadlami

## Motivácia

- TS manipuluje jednom kroku výpočtu s jedným symbolom pásky (*s jednou číslicou čísla*)
- CP mení jednou inštrukciou hodnotu premennej o 1 (*exponenciálne menej efektívne v porovnaní s TS*)
- narovnanie diskrepancie
- CP musí mať možnosť k číslu pridať alebo odobrať číslicu v konštantnom čase

## Modifikácia

množinu inštrukcií CP rozšírime o 2 nové inštruckie

$X \leftarrow X \times 10$

$X \leftarrow X/10$ celočíselné delenie

# Polynomiálna redukcia

Existencia redukcií medzi programovacími jazykmi vyššej úrovne (dostatočne silnými výpočtovými modelmi) ukazuje, že trieda rozhodnuteľných problémov je invariantná voči voľbe jazyka (modelu).

**Otázka**

Aká je zložitosť redukcie?

**Fakt**

Ak oba modely manipulujú s číslami v inej než unárnej sústave, tak redukcia má polynomiálnu časovú zložitosť .

# Zložitosť redukcií medzi TS a modifikovanými CP

Zložitosť výpočtu

TS počet krokov výpočtu

CP počet vykonaných inštrukcií

Zložitosť výpočtu je funkciou dĺžky vstupu; hodnota funkcie pre argument $N$ zhora ohraničuje zložitosť výpočtov na všetkých vstupoch dĺžky $N$. Dĺžkou vstupu pre TS je počet znakov vstupného reťazca, dĺžkou vstupu pre CP je počet číslic počiatočných hodôt premenných.

Redukcia TS $\longrightarrow$ modifikované CP

krok výpočtu je simulovaný zmenou hodnoty každého počítadla; zmena je realizovateľná konštantným počtom inštrukcií

Redukcia modifikované CP $\longrightarrow$ TS

každá inštrukcia je simulovaná konštantným počtom krokov

TS a modifikované CP sú polynomiálne ekvivalentné

# Polynomiálna redukcia - dôsledky

Nech výpočtové modely $A$ a $B$ sú polynomiálne ekvivalentné.

Ak algoritmický problém $P$ je riešiteľný na $A$ s časovou zložitosťou $\mathcal{O}(f(N))$ ( $f$ je funkcia dĺžky vstupu), tak existuje program pre $B$, ktorý rieši problém $P$ a jeho časová zložitosť je $\mathcal{O}(p(f(N)))$, pričom $p$ je nejaká (fixovaná) polynomiálna funkcia.

Naopak, ak $P$ je riešiteľný na $B$ v čase $\mathcal{O}(g(N))$, tak existuje program pre $A$, ktorý rieši $P$ s časovou zložitosťou $\mathcal{O}(q(f(N)))$, pričom $q$ je nejaká (fixovaná) polynomiálna funkcia.

*Ak TS rieši problém v polynomiálnom čase, tak aj modifikový CP rieši tento problém v polynomiálnom čase (a naopak).*
*Ak neexistuje polynomiálny TS pre daný problém, tak neexistuje ani polynomiálny modifikoaný CP pre tento problém.*

# Robustnosť triedy prakticky riešiteľných problémov

CT hypotéza ukazuje robustnosť pojmu rozhodnuteľný problém. Polynomiálna ekvivalencia zjemňuje toto pozorovanie na prakticky riešiteľné problémy.

## Sekvenčná výpočtová hypotéza

Pojem prakticky riešiteľného problému je robustný, tj. je nezávislý na konkrétnej voľbe výpočtového modelu resp. programovacieho jazyka.

*Hypotéza sa nevzťahuje na modely s neohraničeným zdrojom paralelizmu, preto sa označuje ako „sekvenčná“.*

Triedy P, NP, PSPACE, EXPTIME sú robustné

*Triedy s lineárnou časovou zložitosťou nie sú robustné.*

# Nedeterministické Turingove stroje

pre rozhodovacie problémy

- v prechodovom diagrame je povolené, aby s jedného stavu vychádzal ľubovoľný počet hrán označených zhodým spínačom (*symbolom, ktorý sa číta*)
- stroj má možnosť výberu, ktorý z prechodov použije
- pre vstup $X$ dá TS odpoveď „Áno“ (*akceptuje*) práve ak existuje taká postupnosť výberu prechodov, pre ktorú výpočet skončí v koncovom stave *YES*
  (*stroj uváži všetky možné výpočty na $X$ a akceptuje $X$ práve ak aspoň jeden z výpočtov skončí v stave YES*)
- v opačnom prípade, tj. ak žiaden výpočet neskončí v stave *YES*, dá odpoveď „Nie“

Nedeterministické TS sú ekvivalentné (deterministickým) TS.

# P=NP? problém - revízia

## Formálna definícia tried P a NP

Trieda P (NP) obsahuje rozhodovacie problémy, ktoré sú riešiteľné Turingovymi strojmi (nedeterministickými TS) s polynomiálnou časovou zložitosťou.

## P=NP? problém

Sú deterministické a nedeterministické Turingove stroje polynomiálne ekvivalentné?

# P=NP? problém - revízia

## Definícia NP-ťažkého a NP-úplného probému

Rozhodovací problém sa nazýva NP-ťažký ak každý problém z triedy P je na neho polynomiálne redukovateľný.

Rozhodovací problém sa nazýva NP-úplný ak je NP-ťažký a naviac patrí do triedy NP.

## Fakty

Ak nejaký NP-úplný problém patrí do triedy P, tak P $=$ NP.

Ak P $\neq$ NP, tak žiaden NP-úplný problém nie je riešiteľný algoritmom polynomiálnej zložitosti.

# Turingove stroje a dolné odhady zložitosti problémov

- dôkaz nerozhodnuteľnosti problému
  redukcia problému o ktorom je už dokázané, že je nerozhodnuteľný, na problém o ktorom chceme dokázať, že je nerozhodnuteľný
  príklad redukcia problému zastavenia na problém domina
- dôkaz vzťahu medzi zložitostnými triedami
  metóda diagonalizácie
  príklady $P \subset EXPTIME$, $PSPACE \subset EXPSPACE$
- dôkaz, že problém nepatrí do zložitostnej triedy
  dôsledok úplnosti
  príklad žiaden EXPTIME-úplný problém nepatrí do triedy P

*pre dôkaz horných odhadov zložitosti problémov nie sú TS vhodné; naopak je vhodné použiť programovací jazyk vyššej úrovne*

# Redukcia problému zastavenie na problém domina

problém domina úlohou je pokryť hornú polovicu nekonečnej plochy s podmienkou, že prvá dlaždica v $T$ (nazveme ju $t$) je umiestnená niekde v spodnom riadku

problém zastavenia odpoveď „Áno“ pre vstup $\langle M, X \rangle$ taký, že výpočet $M$ na $X$ sa nezastaví

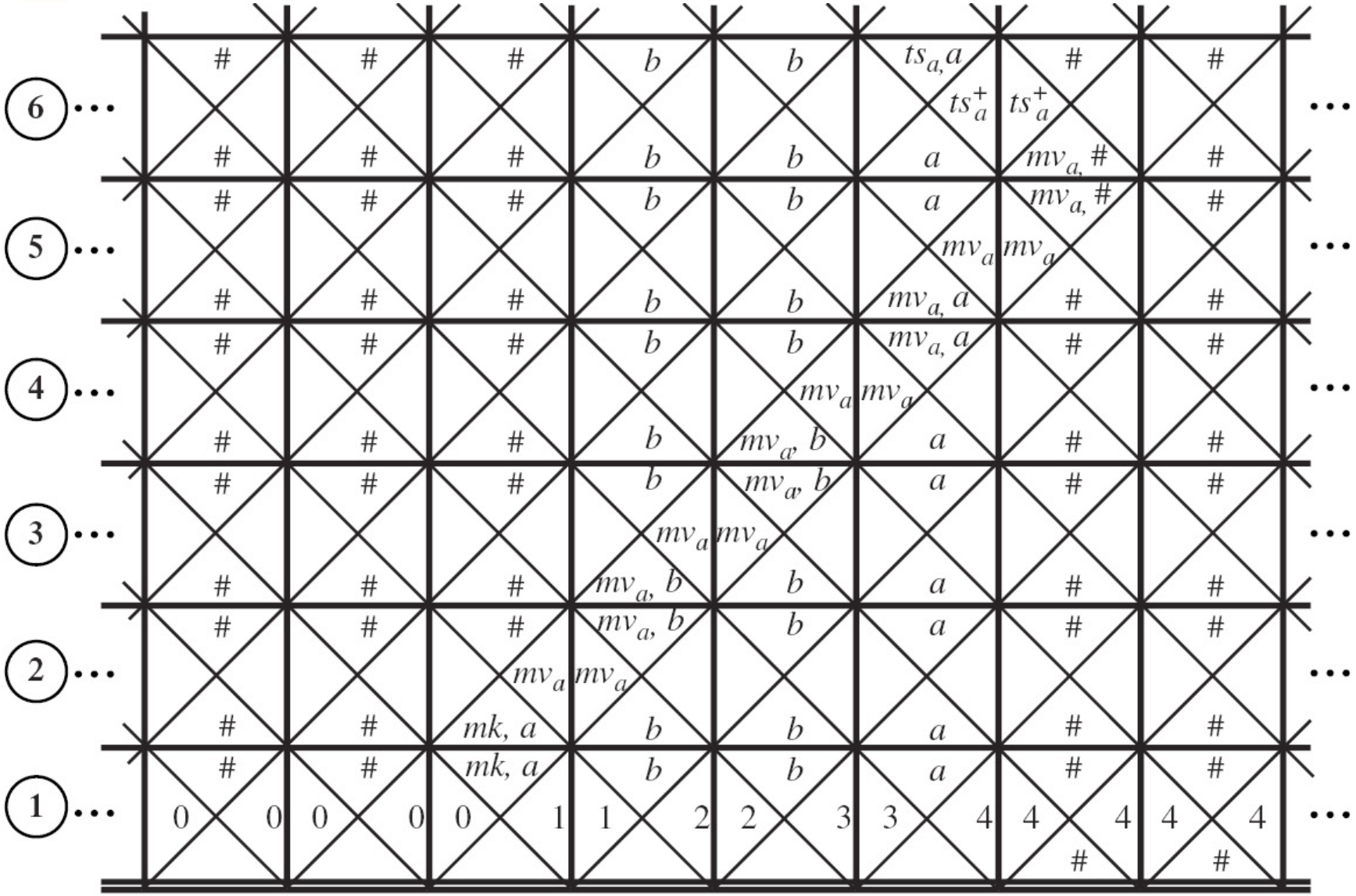
# Redukcia problému zastavenie na problém domina

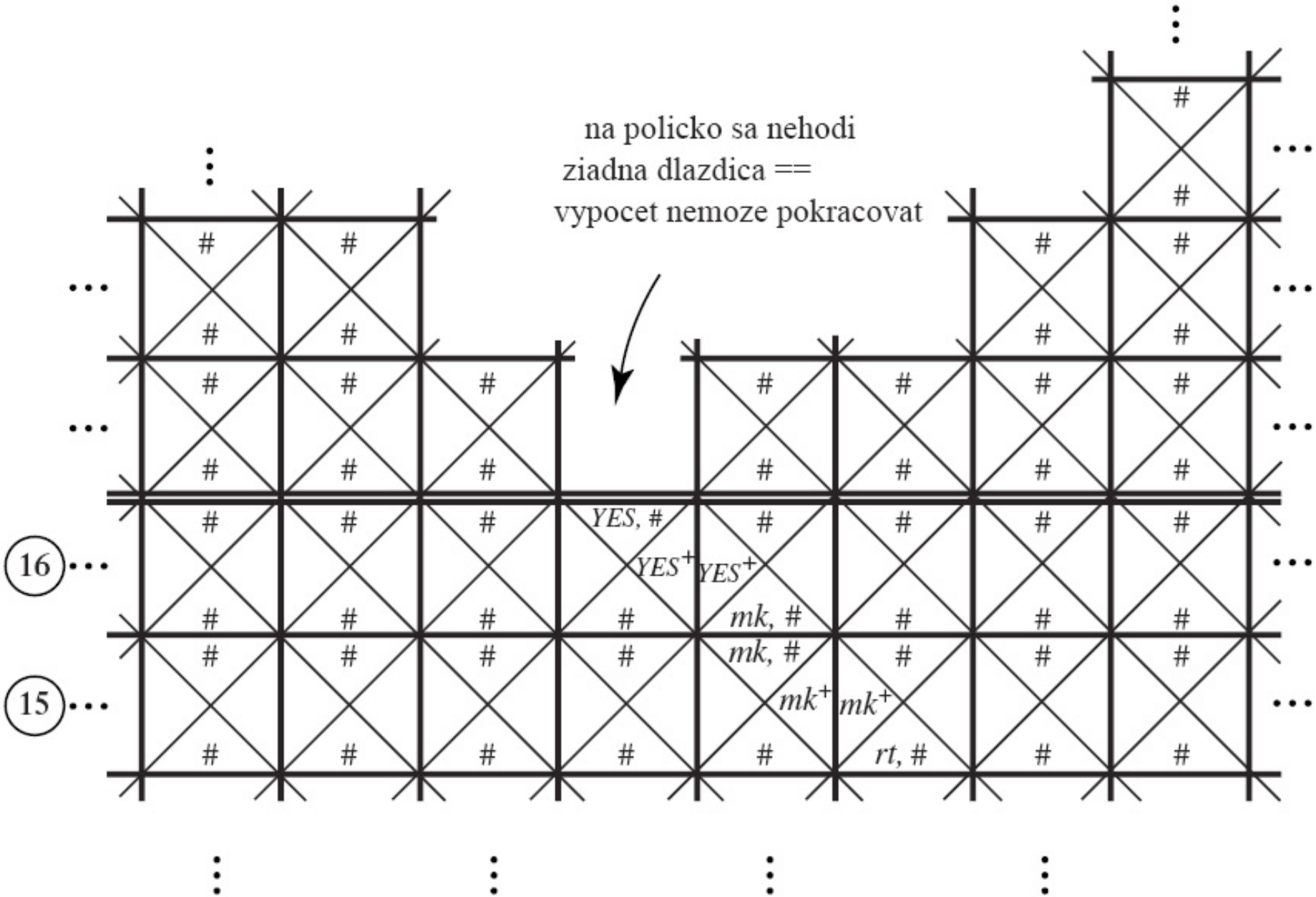
Redukcia

Vstup dvojica $\langle M, X \rangle$

Výstup množina typov dlaždíc $T$ a dlaždica $t$

Princíp konštrukcie $\langle T, t \rangle$ pokrytie dlaždicami korešponduje s výpočtom; pokrytie nekonečnej plochy je možné len v prípade existencie nekonečného výpočtu


6
5
4
3
2
1
0 0 0 0 0 1 1 2 2 3 3 4 4 4 4 4
mk, a
mv_a
ts_a,a
ts_a^+
mv_a, #
mv_a, a
mv_a, b
#
b
a


na policko sa nehodi
ziadna dlazdica ==
vypocet nemoze pokracovat
16
15
YES, #
YES+
YES+
mk, #
mk, #
mk+
mk+
rt, #

# Formálne jazyky

# Jednosmerné TS alebo konečné automaty

- TS sú robustné voči modifikáciam
- existuje modifikácia, ktorá zmení (zmenší) výpočtovú silu TS?
- áno, modifikácia ale musí ohraničiť výpočtový zdroj Turingovho stroja

  polynomiálny čas trieda P

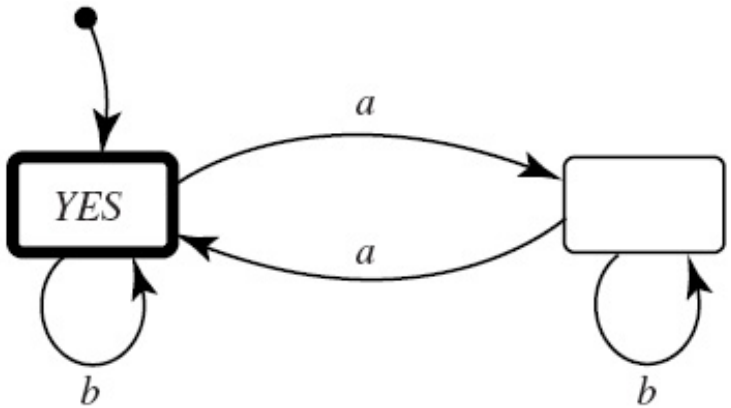
  polynomiálny priestor trieda PSPACE

  (triedy P a PSPACE sú menšie než trieda rozhodnuteľných problémov)

  jeden smer pohybu na páske ohraničenie spočíva v tom, že TS nemá možnosť vrátiť sa k informácii, ktorú už raz prečítal a ani nemá možnosť si o prečítanom úseku uchovať kompletnú informáciu (*riadiaca jednotka má len konečný počet stavov*) = konečné automaty

# Konečné automaty

- vstupný reťazec sa číta zľava doprava, symbol po symbole
- prečítaný symbol sa neprepisuje
- výpočet sa zastaví po prečítaní posledného symbolu alebo v situácii, keď prechodový diagram neumožňuje žiadny ďalší krok
- ak sa výpočet zastaví po prečítaní celého vstupu v stave *YES*, znamená to odpoveď „Áno“ (konečný automat akceptuje vstup); ak sa výpočet zastaví v inom stave alebo sa zastaví a nie je prečítaný celý vstup, znamená to odpoveď „Nie“ (automat zamieta vstup)

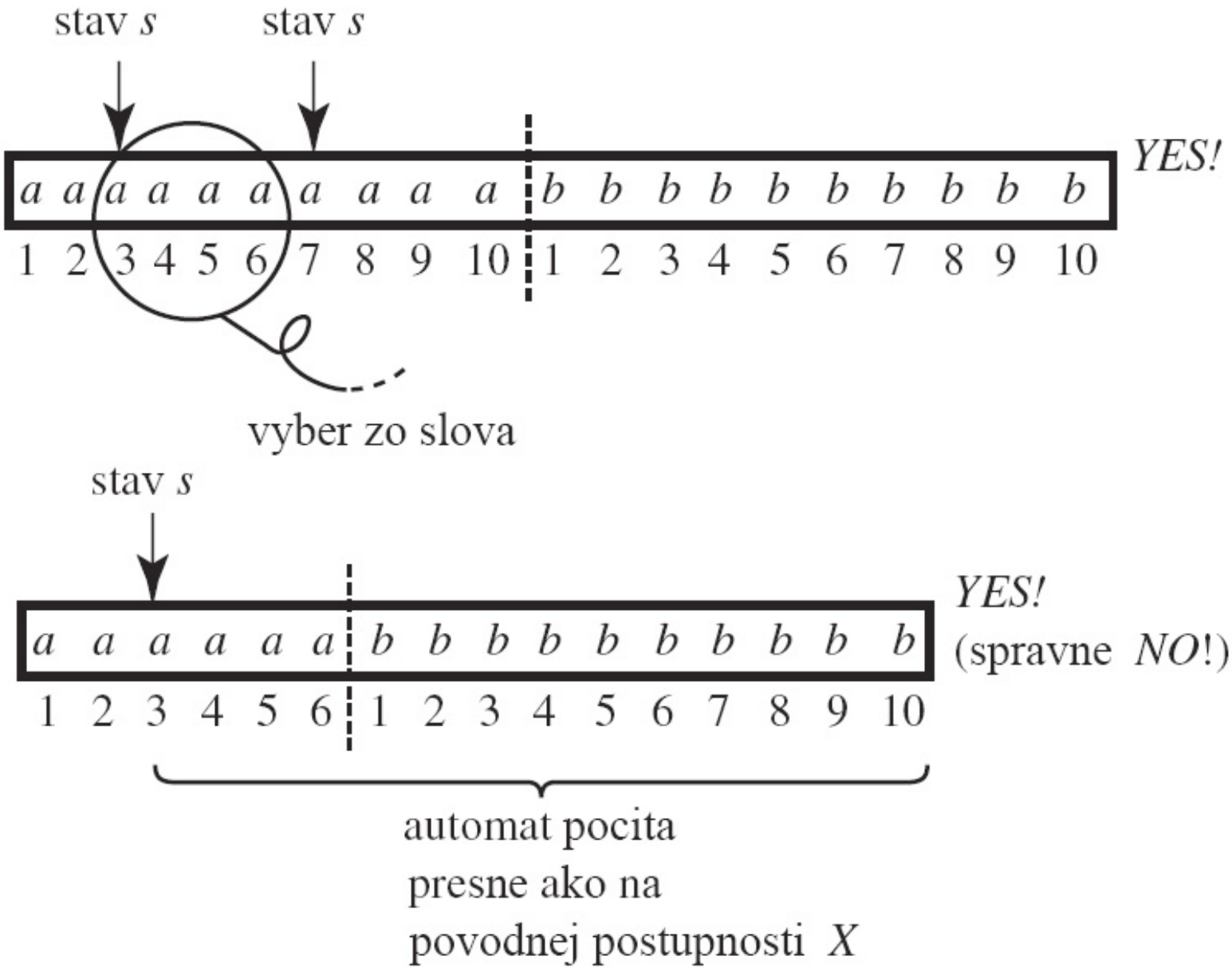


# Konečné automaty - dolné odhady

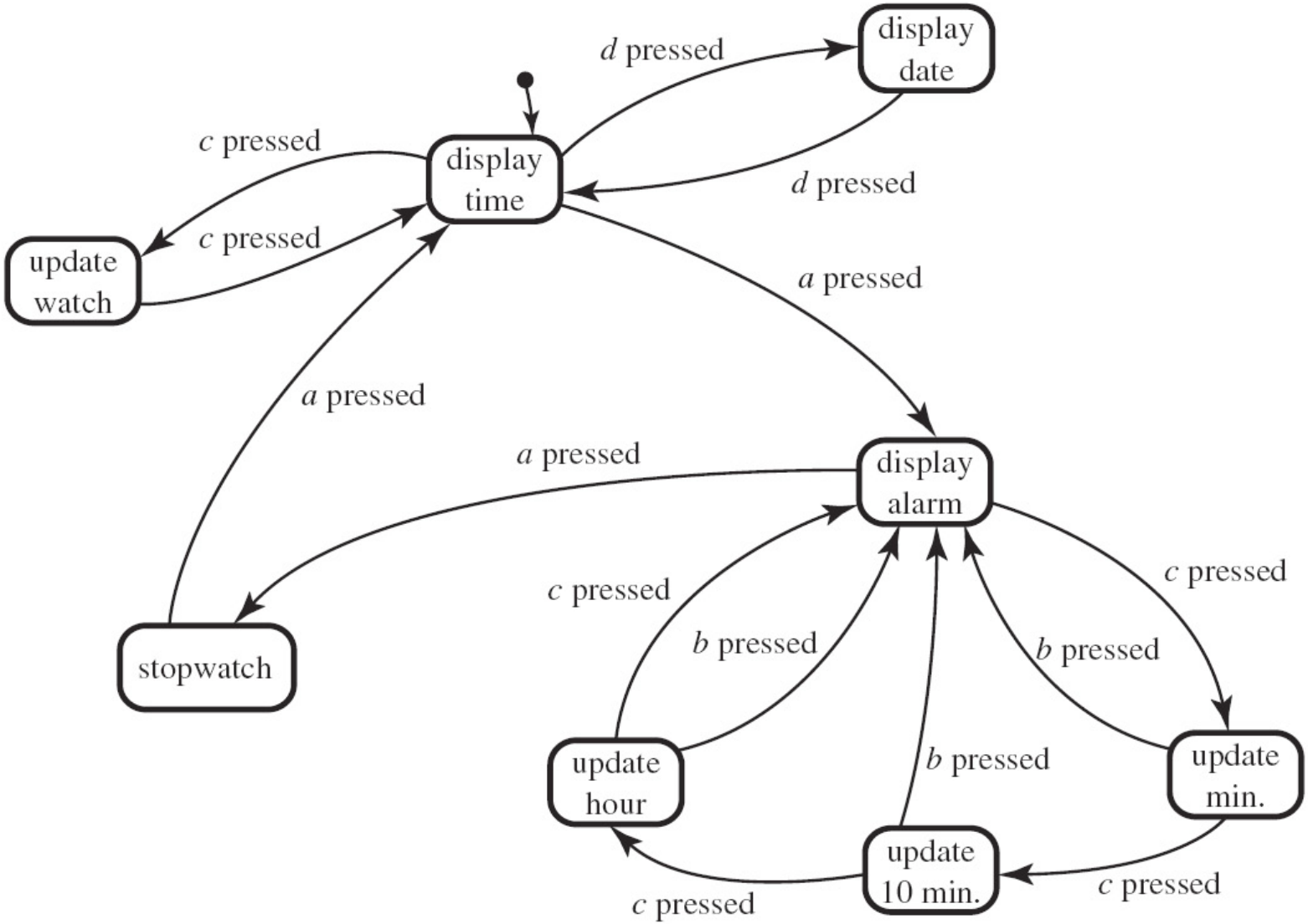
Problém rozhodnúť, či daný reťazec obsahuje rovnaký počet symbolov $a$, $b$

Tvrdenie Neexistuje konečný automat, ktorý rieši tento problém.

Dôkaz Sporom.
Predpokladajme, že existuje automat $F$, ktorý problém rieši. Označme $N$ počet stavov automatu $F$. Uvážme vstupný reťazec, ktorý obsahuje presne $N+1$ symbolov $a$, za ktorými nasleduje presne $N+1$ symbolov $b$. Pri čítaní úvodnej sekvencie $a$-čok musia byť dve políčka, ktoré automat číta v tom istom stave (*počet políčok je* $N+1$, *počet stavov je* $N$). Vytvorme nový vstupný reťazec tak, že odstránime všetky symboly medzi týmito dvoma $a$-čkami (viz obrázok).
Výpočet na novom vstupnom reťazci skončí v tom istom stave a v tej istej pozícii ako výpočet na pôvodnom vstupnom reťazci.
Ak automat akceptuje obidva reťazce dostávame spor (*modifikovaný reťazec nemá požadovanú vlastnosť*). Naopak, ak automat obidva reťazce zamieta – spor (*pôvodný reťazec má požadovanú vlastnosť*).


stav s
stav s
a a a a a a a a a a b b b b b b b b b b
YES!
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
vyber zo slova
stav s
a a a a a a b b b b b b b b b b
YES!
(spravne NO!)
1 2 3 4 5 6 1 2 3 4 5 6 7 8 9 10
automat pocita
presne ako na
povodnej postupnosti X

# Konečné automaty ako model systému s udalosťami



# Konečné automaty - terminológia

pojem jazyka ako ekvivalentu pojmu rozhodovací problém
regularny jazyk
regularne vyrazy

# Formálne jazyky

# Generatívne výpočtové modely

Fixujme rozhodovací problém $P$ (*resp. jazyk* $P_L$)

rozhodovanie určiť, či pre daný vstup $X$ je odpoveď „Áno“ alebo „Nie“
(*resp. určiť, či* $X$ *patrí do jazyka* $L_X$)

generovanie vymenovať všetky vstupy, pre ktoré je odpoveď „Áno“
(*resp. vymenovať všetky slová jazyka* $P_L$)

## Motivácia

návod, ako vytvoriť „správny“ reťazec

formálna gramatika

# Formálne gramatiky

príklad

# Formálne gramatiky - vlastnosti

**Fakt**

Trieda jazykov generovaných formálnymi gramatikami je práve trieda rozhodnuteľných problémov.

Formálne gramatiky s obmedzeniami

kontextové gramatiky reťazec na ľavej strane pravidla je nie je dlhší než reťazec na pravej strane pravidla

bezkontextové gramatiky na ľavej strane každého pravidla je práve jeden neterminálny symbol

regulárne gramatiky

# Formálne gramatiky - problém syntaktickej analýzy

Pre danú formálnu gramatiku a reťazec rozhodnúť, či je slovo sa gramatikou vygenerovať.

*rozhodnúť, či program v programovacom jazyku (definovanom gramatikou) je syntakticky správny*

Problém syntaktickej analýzy je

čiastočne rozhodnuteľný pre formálne gramatiky

rozhodnuteľný pre kontextové gramatiky

polynomiálne riešiteľný pre bezkontextové gramatiky

*je dôležité, aby sme pre definíciu syntaxe programovacieho jazyka použili čo najjednoduchší typ gramatiky*

# Alternatívne výpočtové modely

## Motivácia

existencia veľkej triedy prakticky neriešiteľných (*ale rozhodnuteľných*) problémov, ktoré potrebujeme prakticky riešiť!

## Idea

využiť principiálne iné spôsoby počítania

- paralelné počítanie
- súbežnosť
- kvantové počítanie
- molekulárne počítanie
- náhodnosť

pomôže to ???

# Alternatívne výpočtové modely

# Princíp paralelizmu

Praktický príklad

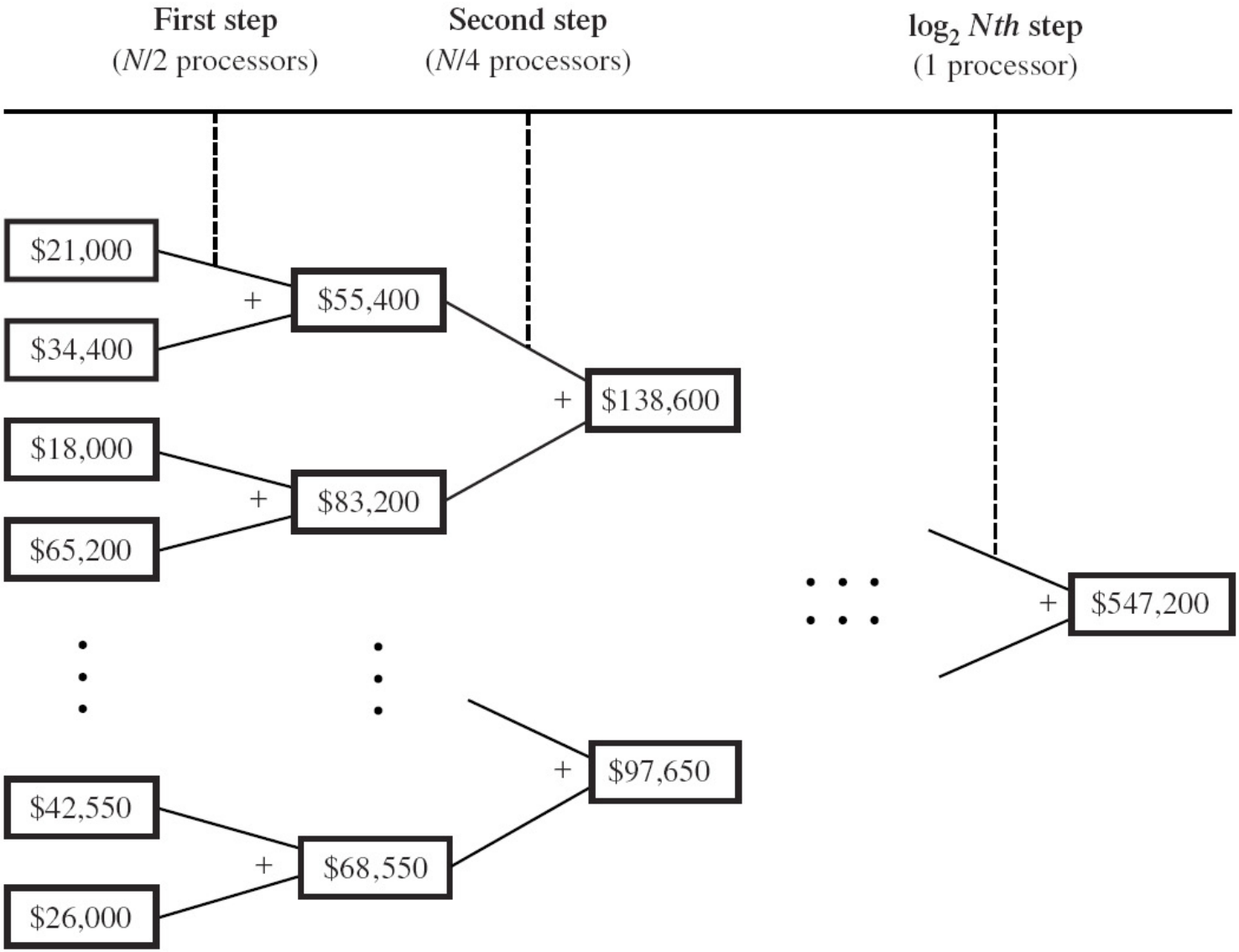
- Varianta A veža o základe 1 m $\times$ 10 m a výšky 1 m; 1 murár vs 10 murárov
- Varianta B veža o základe 1 m $\times$ 1 m a výšky 10 m; 1 murár vs 10 murárov

Jednoduchý program

- Varianta A $X \leftarrow 3;\ Y \leftarrow 4$
- Varianta B $X \leftarrow 3;\ Y \leftarrow X$

*paralelizovateľné problémy* vs *vnútorne sekvenčné problémy*

# Paralelné sčitovanie



# Paralelné sčitovanie

pre sčítanie 1 000 čísel potrebujeme

- v 1. kroku 500 procesorov
- v 2. kroku 250 procesorov
- v 3. kroku 125 procesorov
- ... ...

pri vhodne zvolenej dátovej štruktúre a organizácii komunikácie mezdi procesormi stačí na realizáciu celého výpočtu práve 500 procesorov

pre sčítanie $N$ čísel potrebujeme $N/2$ procesorov a počet (paralelných) výpočtových krokov je $\mathcal{O}(\log N)$

# Paralelizmus - počet procesorov

- počet procesorov potrebných k realizácii paralelného výpočtu ako funkcia veľkosti vstupnej inštancie
  ($N/2$ *pre sčítanie* $N$ *čísel*)
- je to realistické?

indikátor, aké veľké vstupy môžeme riešiť s počtom procesorov, ktoré máme k dispozícii
(*viz analógia s časovou a priestorovou zložitosťou*)

ak počet procesorov, ktoré máme k dispozícii, je menší, môžeme kombinovať paralelný a sekvenčný prístup

# Paralelené triedenie

sekvenčné triedenie zoznamu $L$ spájaním (*mergesort*)

procedúra **sort**-$L$
(1) ak $L$ má len 1 prvok, je utriedený
(2) inak
    (2.1) rozdeľ $L$ na dve polovičky $L_1$ a $L_2$
    (2.2) **sort**-$L_1$
    (2.3) **sort**-$L_2$
    (2.4) spoj dva utriedené zoznamy do jedného utriedeného zoznamu

počet vykonaných porovnaní je $\mathcal{O}(N \log N)$

# Paralelné triedenie

paralelné triedenie zoznamu $L$ spájaním

procedúra **parallel-sort-**$L$

(1) ak $L$ má len 1 prvok, je utriedený

(2) inak

(2.1) rozdeľ $L$ na dve polovičky $L_1$ a $L_2$

(2.2) súbežne volaj **parallel-sort-**$L_1$ a **parallel-sort-**$L_2$

(2.3) spoj dva utriedené zoznamy do jedného utriedeného zoznamu

počet (paralelných) porovnaní je (*predpokladáme, že $N$ je mocninou 2*)

v 1. kroku $N$ postupností dĺžky 1; $N/2$ procesorov;
spojenie dvoch postupností = 1 porovnanie

v 2. kroku $N/2$ postupností dĺžky 2; $N/4$ procesorov;
spojenie dvoch postupností = 3 porovnania

v 3. kroku $N/4$ postupností dĺžky 4; $N/8$ procesorov;
spojenie dvoch postupností = 7 porovnaní

spolu $1 + 3 + 7 + 15 + \cdots + (N - 1) \leq 2N$ porovnaní

# Paralelizmus - čas × priestor

konvencia: v kontexte paralelných výpočtov sa pod priestorovou zložitosťou rozumie počet procesorov potrebných k realizácii výpočtu

časová a priestorová zložitosť paralelných výpočtov sú spolu tesne zviazané

*zníženie jednej obvykle znamená zvýšenie druhej a naopak*

# Paralelizmus - čas × priestor

| | Name | Size (no. of processors) | Time (worst case) | Product (time × size) |
|---|---|---|---|---|
| SEQUENTIAL | Bubblesort | 1 | $O(N^2)$ | $O(N^2)$ |
| SEQUENTIAL | Mergesort | 1 | $O(N \times \log N)$ | $O(N \times \log N)$ (optimal) |
| PARALLEL | Parallelized mergesort | $O(N)$ | $O(N)$ | $O(N^2)$ |
| PARALLEL | Odd-even sorting network | $O(N \times (\log N)^2)$ | $O((\log N)^2)$ | $O(N \times (\log N)^4)$ |
| PARALLEL | "Optimal" sorting network | $O(N)$ | $O(\log N)$ | $O(N \times \log N)$ (optimal) |

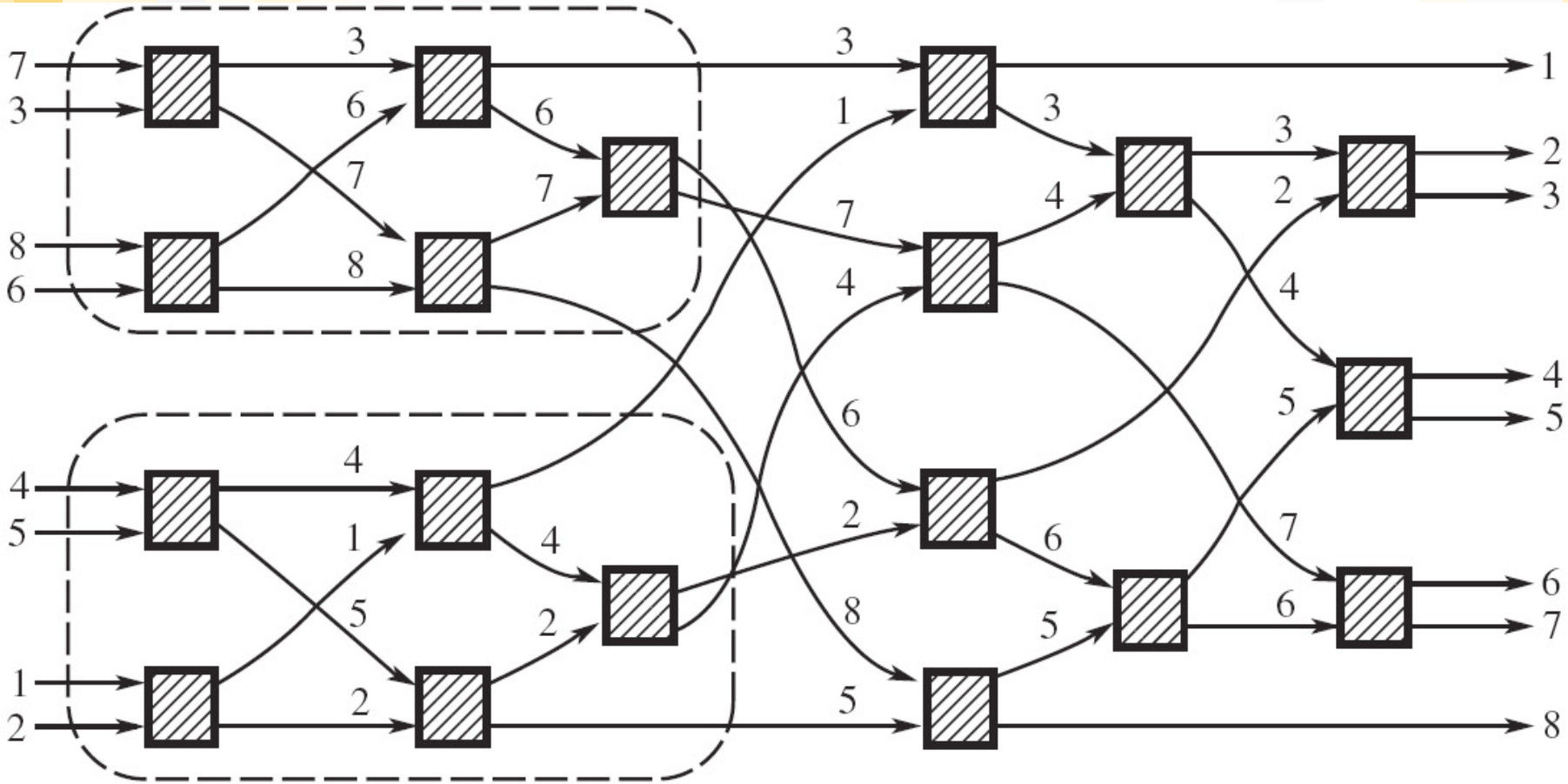
# Paralelné siete

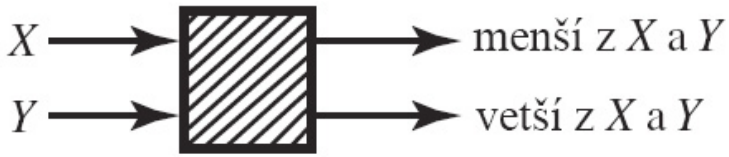
Spôsob komunikácie medzi procesormi

zdieľaná pamäť súčasný zápis resp. čítanie z pamäťového miesta ???
v prípade súčasného zápisu nutnosť stratégie riešenia konfliktov

sieť s fixovaným prepojením každý procesor je prepojený (môže komunikovať) len s ohraničením počtom *susediacich* procesorov;
často ako špecializovaný hardware

# Triediaca sieť



# Môže paralelizmus riešiť neriešiteľné?

**Fakt**

Každý paralelný algoritmus sa dá transformovať na sekvenčný algoritmus.

(*každý paralelný krok nahradíme postupnosťou sekvenčných krokov; každý sekvenčný krok vykoná prácu jedného procesoru*)

**Dôsledok**

Neexistuej paralelný algoritmus pre nerozhodnuteľný problém.

(*CT hypotéza sa vzťahuje aj na paralelné výpočtové modely*)

# Paralelizmu a prakticky neriešiteľné problémy

- existujú problémy, ktoré sú sekvenčne prakticky neriešiteľná a pritom sú prakticky riešiteľné paralelnými algoritmami?
- špecifikácia pojmu „efektívneho“ paralelného algoritmu ???

# Paralelizmu a prakticky neriešiteľné problémy

- existujú problémy, ktoré sú sekvenčne prakticky neriešiteľná a pritom sú prakticky riešiteľné paralelnými algoritmami?
- špecifikácia pojmu „efektívneho“ paralelného algoritmu ???
- pozorovanie: pre problém z triedy NP máme nedeterministický algoritmus polynomiálnej časovej zložitosti;
  ak nedeterministický výber nahradíme paralelizmom, tak okamžite dostávama polynomiálne časovo ohraničený paralelný algoritmus pre tento problém
- je to prijateľné riešenie?

# Paralelizmu a prakticky neriešiteľné problémy

- existujú problémy, ktoré sú sekvenčne prakticky neriešiteľná a pritom sú prakticky riešiteľné paralelnými algoritmami?
- špecifikácia pojmu „efektívneho“ paralelného algoritmu ???
- pozorovanie: pre problém z triedy NP máme nedeterministický algoritmus polynomiálnej časovej zložitosti;
  ak nedeterministický výber nahradíme paralelizmom, tak okamžite dostávama polynomiálne časovo ohraničený paralelný algoritmus pre tento problém
- je to prijateľné riešenie?
  - exponenciálny počet procesorov

# Paralelizmu a prakticky neriešiteľné problémy

- existujú problémy, ktoré sú sekvenčne prakticky neriešiteľná a pritom sú prakticky riešiteľné paralelnými algoritmami?
- špecifikácia pojmu „efektívneho“ paralelného algoritmu ???
- pozorovanie: pre problém z triedy NP máme nedeterministický algoritmus polynomiálnej časovej zložitosti;
  ak nedeterministický výber nahradíme paralelizmom, tak okamžite dostávama polynomiálne časovo ohraničený paralelný algoritmus pre tento problém
- je to prijateľné riešenie?
  - exponenciálny počet procesorov
  - čo ak prakticky neriešiteľný problém nepatrí do NP?

# Paralelizmu a prakticky neriešiteľné problémy

- existujú problémy, ktoré sú sekvenčne prakticky neriešiteľná a pritom sú prakticky riešiteľné paralelnými algoritmami?
- špecifikácia pojmu „efektívneho“ paralelného algoritmu ???
- pozorovanie: pre problém z triedy NP máme nedeterministický algoritmus polynomiálnej časovej zložitosti;
  ak nedeterministický výber nahradíme paralelizmom, tak okamžite dostávama polynomiálne časovo ohraničený paralelný algoritmus pre tento problém
- je to prijateľné riešenie?
  - exponenciálny počet procesorov
  - čo ak prakticky neriešiteľný problém nepatrí do NP?
  - otázka praktickej implementácie paralelného algoritmu, ktorý má síce polynomiálnu zložitosť, ale potrebuje exponenciálny počet procesorov (*napr. otázka komunikácie*)

# Paralelná výpočtová téza

## Časť A

Všetky „rozumné“ paralelné výpočtové modely sú polynomiálne ekvivalentné.

## Časť B

Paralelný čas je polynomiálne ekvivalentý sekvenčnému času.

$$\textit{Sekvenčný}\text{-PSPACE} = \textit{Paralelný}\text{-PTIME}$$

# NC - Nickova trieda

- polynomiálne časovo ohraničené paralelné algoritmy nemôžeme považovať za prakticky použiteľné
- zmyslom zavedenia paralelizmu je výrazne redukovať výpočtový čas

sublineárne algoritmy

**Trieda NC**

Trieda problémov riešiteľných paralelnými algoritmami s polylogaritmickou časovou zložitosťou a polynomiálnym počtom procesorov.

Trieda NC je robustná

# Vzťah sekvenčných a paralelných zložitostných tried

$$NC \subseteq P \subseteq NP \subseteq PSPACE$$

Otvorené problémy: o žiadnej z inklúzií nie je známe, či je ostrá, alebo predstavuje rovnosť

**Predpoklady**

1. existujú problémy, ktoré sú prakticky (sekvenčne) riešiteľné, ale nie sú riešiteľné veľmi rýchlo paralelne s použitím rozumne veľkého hardwaru
2. existujú problémy, ktoré sa dajú riešiť (sekvenčne) v polynomiálnom čase s využitím nedeterminizmu, ale nie bez neho
3. existujú problémy, ktoré sa dajú riešiť v rozmunom (tj. polynomiálnom) sekvenčnom priestore (tj. aj v rozumnom paralelnom čase), ale nie sú riešiteľné v rozumnom sekvenčnom čase bez využitia nedeterminizmu.

# Vzťah sekvenčných a paralelných zložitostných tried

$$NC \subseteq P \subseteq NP \subseteq PSPACE$$

Príklady problémov

| | |
|---|---|
| NC | sčítať $N$ čísel |
| | rozhodnúť, či v grafe existuje cesta z vrcholu $s$ do vrcholu |
| $P \setminus NC$ | rozhodnúť, či $c$ je najväčším spoločným deliteľom čísel $a$, |
| $NP \setminus P$ | problém Hamiltonovského cyklu; splniteľnosť logickej form |
| $PSPACE \setminus PN$ | slovný futbal |

# Alternatívne výpočtové modely

# Súbežnosť

situácie, keď paralelizmus nevyužívame k tomu, aby sme zefektívnili výpočty, ale keď paralelizmus vzniká v reálnych aplikáciach

reaktívne a zapúzdrené systémy

**Úloha**

navrhnúť komunikačné protokoly pre komunikujúce objekty tak, aby spĺňali požadované vlastnosti

**Špecifikum**

úlohou systémov nie je nájsť konkrétne riešenie, ale počítať (byť funkčný) „donekonečna“

# Problém hotelovej sprchy

- na poschodí je len jedna sprcha, na veľmi studenej chodbe
- každý hosť sa chce osprchovať, ale nemôže čakať na voľnú sprchu na chodbe
- ak by z času na čas kontroloval, či je sprcha voľná, môže nastať situácia, že sa nikdy neosprchuje

možné riešenie

- tabuľa pri vstupe do sprchy
- hosť pri odchode zo sprchy zmaže z tabule číslo svojej izby a napíše na ňu číslo nasledujúcej izby (v nejakom fixovanom poradí)
- každý hosť z času na čas kontroluje, či je na tabuli napísané číslo jeho izby a ak áno, osprchuje sa

*je to dobré riešenie?*

(*prázdna izba, práve jedno sprchovanie v cykle, . . .*)

# Distribuované súbežné systémy

- (súbežné) komponenty systému sú fyzicky oddelené
- komponenty vzájomne komunikujú
- typicky od systémov nepožadujeme vstupno - výstupné chovanie, ale kontinuálnu (neprerušenú, neukončenú) funkcionalitu
- dôležité je chovanie systému v čase
- okrem požiadavkov na jednotlivé komponenty, kladieme na systém globálne obmedzujúce podmienky
  - zdieľanie spločných zdrojov
  - prevencia uviaznutia (vzájomné čakanie, *deadlock*)
  - prevencia starnutia procesov (*starvation*)

*(algoritmické) protokoly*

# Problém vzájomného vylúčenia

zobecnenie problému hotelovej sprchy

Problém vzájomného vylúčenia: je daných $N$ procesov, každý proces opakovane (v nekonečnom cykle) vykonáva

- súkromné aktivity (proces ich môže vykonávať nezávislé na ostatných procesoch) a
- kritické aktivity (žiadne dva procesy nemôžu súčasne vykonávať svoje kritické aktivity)

# Protokol pre problém vzájomného vylúčenia

pre dva procesy $P_1$ a $P_2$

protokol využíva 3 premenné

- $Z$ globálna premenná (*všetky procesy môžu meniť jej hodnotu*); nadobúda hodnoty 1 a 2
- $X_1$ lokálna premenná procesu $P_1$; nadobúda hodnoty *yes* a *no*
- $X_2$ lokálna premenná procesu $P_2$; nadobúda hodnoty *yes* a *no*

počiatočná hodnota premenných $X_1$ a $X_2$ je *no*, počiatočná hodnota $Z$ je buď 1 alebo 2

# Protokol pre problém vzájomného vylúčenia (pre dva procesy)

protokol pre proces $P_1$

(1) opakuj v nekonečnom cykle
  (1.1) vykonaj súkromné aktivity
  (1.2) $X_1 \leftarrow yes$
  (1.3) $Z \leftarrow 1$
  (1.4) čakaj kým buď
    $X_2$ nenadobudne hodnotu *no* alebo
    $Z$ nenadobudne hodnotu 2
  (1.5) vykonaj kritické aktivity
  (1.6) $X_1 \leftarrow no$

protokol pre proces $P_2$

(1) opakuj v nekonečnom cykle
  (1.1) vykonaj súkromné aktivity
  (1.2) $X_2 \leftarrow yes$
  (1.3) $Z \leftarrow 2$
  (1.4) čakaj kým buď
    $X_1$ nenadobudne hodnotu *no* alebo
    $Z$ nenadobudne hodnotu 1
  (1.5) vykonaj kritické aktivity
  (1.6) $X_2 \leftarrow no$

korektnosť protokolu

| | |
|---|---|
| vzájomné vylúčenie | *OK* |
| starnutie procesu | *OK* |
| uviaznutie | *OK* |

# Protokol pre problém vzájomného vylúčenia (pre $N$ procesov)

protokol pre $I$-ty proces $P_I$

(1) opakuj v nekonečnom cykle
  (1.1) vykonaj súkromné aktivity
  (1.2) pre každé $J$ od 1 do $N-1$ urob
    (1.2.1) $X[I] \leftarrow J$
    (1.2.2) $Z[I] \leftarrow I$
    (1.2.3) čakaj kým buď
      $X[K] < J$ pre nejaké $K \neq I$ alebo
      $Z[J] \neq I$
  (1.5) vykonaj kritické aktivity
  (1.6) $X[I] \leftarrow 0$

# Vlastnosti distribuovaných súbežných systémov

distribuované súbežné systémy sa odlišujú od sekvenčných systémov

špecifikácia problému nemôžeme použiť špecifikáciu problému prostredníctvom množiny vstupných inštancií a fuknčnej závislosti medzi vstupom a požadovaným výstupom

korektnosť nepostačuje dôkaz konečnosti výpočtu a správnosti vstupno-výstupného vzťahu

efektívnosť nepostačuje vyjadrenie efektivity cez časovú (*počet krokov od začiatku do ukončenia*) a priestorovú zložitosť

*aké vlastnosti požadujeme od súbežných procesov?*

# Vlastnosti živosti a bezpečnosti

Vlastnosti, ktoré požadujeme od protokolov pre súbežné procesy (najčastejšie) spadajú do dvoch kategórií: bezpečnosť a živosť.

bezpečnosť nikdy nenastanú „špatné“ udalosti resp. vždy sa stávajú len „dobré“ udalosti

živosť určité „dobré“ udalosti sa nakoniec stanú

aby sme ukázali, že protokol porušuje vlastnosť bezpečnosti, stačí ukázať konkrétnu postupnosť akcií, ktoré vedú k zakázanej udalosti

aby sme ukázali, že protokol porušuje vlastnosť živosti, musíme ukázať existenciu nekonečnej postupnosti akcií, ktoré nevedú k požadovanej dobrej udalosti

# Overovanie vlastností živosti a bezpečnosti

**testovanie a simulácia** môžu odhaliť chybu, ale nemôžu dokázať platnosť požadovanej vlastnosti;
techniky sú jednoduché

**formálna verifikácia** matematická metóda, ktorá *dokáže* platnosť požadovanej vlastnosti;
metóda overovania modelov (*model checking*)
náročné na implementáciu a samotné overenie vlastnosti;
pre niektoré systémy je problém nerozhodnuteľný

# Temporálna logika

jazyk pre popis vlastností súbežných systémov

formule logiky sa vyjadrujú o pravdivosti základných tvrdení v čase (*v priebehu výpočtu*)

základné konštrukcie: globálna platnosť (globally, G) a platnosť v budúcnosti (future, F)

príklady - protokol vzájomného vylúčenia pre dva procesy

$$P_1\text{-je-v-}(1.4) \Longrightarrow \mathrm{F}\ (P_1\text{-je-v-}(1.5))$$

$$\mathrm{G}\ (\neg[P_1\text{-je-v-}(1.5) \wedge P_2\text{-je-v-}(1.5)\ ])$$

# Alternatívne výpočtové modely

# Kvantové počítanie

- založené na princípoch kvantovej mechaniky
- teoretický model: základnou jednotkou reprezentácie informácie je qubit, ktorý (zjednodušene) môže nadobúdať ľuboboľnú hodnotu z intervalu $[0, 1]$
- kvantové algoritmy
- otázka konštrukcie kvantového počítača

# Alternatívne výpočtové modely

# Molekulárne (DNA) počítanie

- DNA == reťazce nad abededou A, C, T, G
- vývoj organizmu == manipulácia s reťazcacmi: kopírovanie, rozdeľovanie, spájanie
- 1994: experiment, v ktorom pomocou manipulácie s molekulami bola vyriešená inštancia problému Hamiltonovského cyklu veľkosti 7; experiment predstavoval niekoľko dní laboratórnej práce
- pozitívum: veľká miera paralelizmu
- negatívum: veľký objem biologického materiálu potrebného k riešeniu rozsiahlejších inštancií
- budúcnosť: ???

# Alternatívne výpočtové modely

náhodnostný protokol pre večerajúcich filozofov

# Porovnávanie reťazcov

Na počítačoch A a B sú uložené databázy $x$ a $y$; nech $x$ a $y$ sú binárne reťazce dĺžky $n$. Úlohou je rozhodnúť, či $x$ a $y$ sú zhodné. Zaujíma nás, koľko bitov si musia počítače A a B vymeniť, aby dokázali vyriešiť problém rovnosti.

Dá sa dokázať, že neexistuje deterministický komunikačný protokol, ktorý by riešil problém rovnosti a pritom si A a B vymenili nanajvýš $n-1$ bitov. T.j. protokol, v ktorom A pošle celý reťazec $x$ počítaču B je optimálny.

# Porovnávanie reťazcov

## Randomizovaný protokol pre problém rovnosti

*Vstup* $x = x_1x_2 \dots x_n$, $y = y_1y_2 \dots y_n$

*Krok 1* A vyberie náhodne prvočíslo $p$ z intervalu $[2, n^2]$.

*Krok 2* A vypočíta číslo $s = x \mod p$ a pošle čísla $s$, $p$ počítaču B.

*Krok 3* B vypočíta číslo $q = y \mod p$.
Ak $q \neq s$, tak B vráti odpoveď $x \neq y$.
Ak $q = s$, tak B vráti odpoveď $x = y$.

počet bitov, ktoré si počítače pošlú je $2 \cdot \lceil \log_2 n^2 \rceil \leq 4 \cdot \lceil \log_2 n \rceil$
pravdepodobnosť, že protokol vráti nesprávnu odpoveď je $\leq \frac{\ln n^2}{n}$

*Ak $n = 10^{16}$, tak zložitosť deterministického protokolu je $10^{16}$, zatiaľ čo zložitosť randomizovaného protokolu je 256.*
*Pravdepodobnosť, že randomizovaný protokol vráti nesprávnu odopoveď je $\leq 0.36892 \cdot 10^{-14}$.*

# Randomizovaný Quicksort

*Rand-Quicksort(A)*

| | |
|---|---|
| *Vstup* | zoznam prvkov $A$ |
| *Krok 1* | ak $A$ má jeden prvok, je utriedený<br>ak $A$ má viac prvkov, tak náhodne vyber prvok $x$ z $A$ |
| *Krok 2* | vytvor zoznam $A_<$ obsahujúci prvky z $A$ menšie než $x$<br>vytvor zoznam $A_>$ obsahujúci prvky z $A$ väčšie než $x$ |
| *Krok 3* | výstup je *Rand-Quicksort*($A_<$), $x$, *Rand-Quicksort*($A_>$) |

očakávaná zložitosť algoritmu je $\mathcal{O}(N \log N)$

# Typy náhodnostných algoritmov

**Monte Carlo** s ohraničenou pravdepodobnosťou je odpoveď nesprávna
*príklad*: randomizovaný protokol pre problém rovnosti

**Las Vegas** odpoveď je vždy správna;
*cieľ*: očakávaná zložitosť Las Vegas algoritmu pre problém je lepšia než zložitosť (deterministického) algorimtu
*príklad*: randomizovaný Quicksort

# Náhodnostné zložitostné triedy

## Pravdepodobnostný Turingov stroj

pracuje ako nedeterministický TS s tým rozdielom, že nedeterministický výber kroku výpočtu interpretujeme ako náhodnostnú voľbu

## Trieda RP

obsahuje rozhodovacie problémy, pre ktoré existuje polynomiálne časovo ohraničený pravdepodobnostný Turingov stroj s vlastnosťou:
ak odpoveďou pre vstup $X$ je „Nie“, tak s pravdepodobnosťou 1 stroj dá správnu odpoveď
ak odpoveďou je „Áno“, tak stroj s pravdepodobnosťou $\geq 1/2$ dá *yes*.

$$\mathrm{P} \subseteq \mathrm{RP} \subseteq \mathrm{NP}$$

*náhodnostné algoritmy nemôžu efektívne riešiť problémy mimo NP; problémy z NP ale dokážu (často) riešiť s väčšou efektivitou*