

Vyčísitelnost a složitost

Co je to algoritmus?

Algoritmus

Algoritmus je mechanický postup skládající se z nějakých jednoduchých elementárních kroků, který pro nějaký zadaný **vstup** vyprodukuje nějaký **výstup**.

Algoritmus může být zadán:

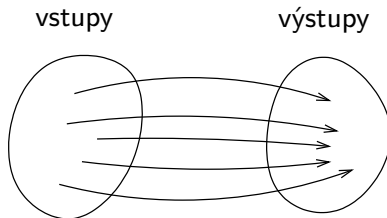
- slovním popisem v přirozeném jazyce
- pseudokódem
- jako počítačový program v nějakém programovacím jazyce
- jako hardwarový obvod
- ...

Algoritmy slouží k řešení různých **problémů**.

Problém

V zadání **problému** musí být určeno:

- co je množinou možných vstupů
- co je množinou možných výstupů
- jaký je vztah mezi vstupy a výstupy



Problém „Třídění“

Vstup: Sekvence prvků a_1, a_2, \dots, a_n .

Výstup: Prvky sekvence a_1, a_2, \dots, a_n seřazené od nejmenšího po největší.

Příklad:

- Vstup: 8, 13, 3, 10, 1, 4
- Výstup: 1, 3, 4, 8, 10, 13

Poznámka: Konkrétní vstup nějakého problému se nazývá **instance** problému.

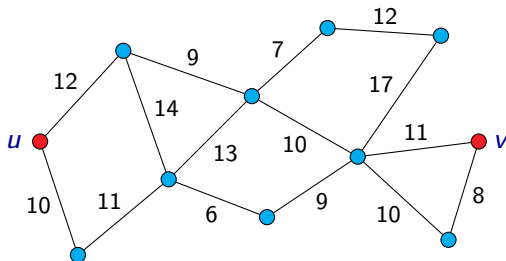
Příklady problémů

Problém „Hledání nejkratší cesty v (neorientovaném) grafu“

Vstup: Neorientovaný graf $G = (V, E)$ s ohodnocením hran, a dvojice vrcholů $u, v \in V$.

Výstup: Nejkratší cesta z vrcholu u do vrcholu v .

Příklad:

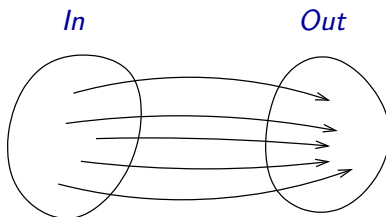


Problém

Formálně tedy můžeme **problém** definovat jako trojici (In, Out, R) , kde:

- In je množina možných vstupů
- Out je množina možných výstupů
- $R \subseteq In \times Out$ je relace přiřazující každému vstupu možné odpovídající výstupy. Tato relace musí splňovat

$$\forall x \in In. \exists y \in Out. R(x, y).$$



Problém „Prvočíselnost“

Vstup: Přirozené číslo n .

Výstup: ANO pokud je n prvočíslo, NE v opačném případě.

Poznámka: Přirozené číslo n je **prvočíslo**, pokud je větší než 1 a je dělitelné beze zbytku pouze čísly 1 a n .

Prvních několik prvočísel: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...

Situace, kdy množina výstupů *Out* je $\{ANO, NE\}$ je poměrně častá. Takovým problémům se říká **rozhodovací problémy**.

Rozhodovací problémy většinou specifikujeme tak, že místo popisu toho, co je výstupem, uvedeme otázku.

Příklad:

Problém „Prvočíselnost“

Vstup: Přirozené číslo n .

Otázka: Je n prvočíslo?

Rozhodovací problém

Jednou z možností, jak formálně definovat **rozhodovací problém**, je definovat ho jako dvojici (In, T) , kde:

- In je množina možných vstupů,
- $T \subseteq In$ je množina vstupů, pro které je odpověď **ANO**.

Optimalizační problémy

Dalším speciálním případem jsou tzv. optimalizační problémy.

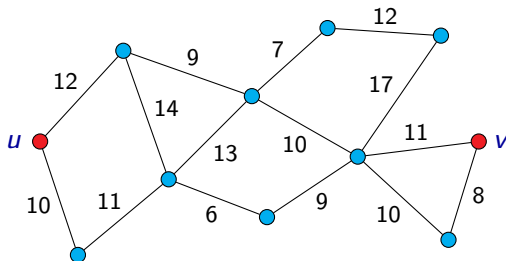
Optimalizační problém je problém, kde je úkolem vybrat z nějaké množiny přípustných řešení takové řešení, které je v nějakém ohledu optimální.

Optimalizační problémy

Dalším speciálním případem jsou tzv. optimalizační problémy.

Optimalizační problém je problém, kde je úkolem vybrat z nějaké množiny přípustných řešení takové řešení, které je v nějakém ohledu optimální.

Příklad: V problému „Hledání nejkratší cesty v grafu“ je množina všech přípustných řešení tvořena všemi cestami z vrcholu u do vrcholu v . Kritériem, podle kterého cesty hodnotíme, je délka cesty.



Optimalizační problémy

Formálně můžeme **optimalizační problém** definovat jako pěticu (In, Out, f, m, g) , kde:

- In je množina možných vstupů,
- Out je množina **řešení**,
- $f : In \rightarrow \mathcal{P}(Out)$ je funkce přiřazující každému vstupu x odpovídající množinu **přípustných řešení** $f(x)$,
- $m : \bigcup_{x \in In} (\{x\} \times f(x)) \rightarrow \mathbb{R}$ je **optimalizační funkce** (**kriteriální funkce**),
- g je **min** nebo **max**.

Cílem je pro daný vstup $x \in In$ najít nějaké přípustné řešení $y \in f(x)$ takové, že

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\},$$

nebo zjistit, že pro daný vstup x žádné přípustné řešení neexistuje (tj. $f(x) = \emptyset$).

- Optimalizačním problémům, kde g je \min , se říká **minimalizační problémy**.
- Optimalizačním problémům, kde g je \max , se říká **maximalizační problémy**.

Řešení problému

Algoritmus **korektně řeší** daný problém, když:

- 1 Se pro libovolný vstup daného problému (libovolnou vstupní instanci) po konečném počtu kroků zastaví.
- 2 Vyprodukuje výstup z množiny možných výstupů, který vyhovuje podmínkám uvedeným v zadání problému.

Pro jeden problém může existovat celá řada algoritmů, které jej korektně řeší.

Každému algoritmu A můžeme přiřadit funkci

$$f_A : In \rightarrow Out$$

kde:

- In je množina vstupů pro algoritmus A ,
- Out je množina výstupů generovaných algoritmem A ,
- $f_A(x)$ je výstup, který algoritmus A vygeneruje pro vstup $x \in In$.

Funkce f_A nemusí být **totální** (tj. hodnota $f_A(x)$ nemusí být definovaná pro každé $x \in In$), ale může být **častečná** (**parciální**):

- hodnota $f_A(x)$ není definována, pokud se výpočet algoritmu A pro vstup x nikdy nezastaví, pokud během výpočtu dojde k chybě apod.

Pokud tedy máme dán nějaký problém $P = (In, Out, R)$ a nějaký algoritmus A realizující funkci $f_A : In \rightarrow Out$, pak řekneme, že

algoritmus A řeší problém P

jestliže:

- hodnota $f_A(x)$ je definovaná pro každé $x \in In$,
- pro každé $x \in In$ platí $(x, f_A(x)) \in R$

Kódování vstupu a výstupu

Obecně se můžeme omezit na to, že vstupy i výstupy nějakého problému nebo algoritmu jsou slova v nějaké abecedě Σ , tj. $In = Out = \Sigma^*$.

Různé jiné objekty (čísla, posloupnosti čísel, grafy, ...) pak zapisujeme (kódujeme) jako slova v této abecedě.

Příklad: Například u problému „Třídění“ bychom mohli zvolit jako abecedu $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, , \}$.

Vstupem by pak mohlo být například slovo

826,13,3901,101,128,562

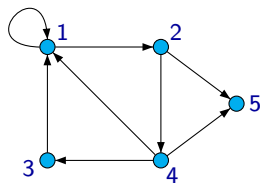
a výstupem slovo

13,101,128,562,826,3901

Poznámka: Ne každé slovo ze Σ^* musí reprezentovat nějaký vstup. Kódování bychom ale měli zvolit tak, abychom byli schopni snadno poznat ta slova, která nějaký vstup reprezentují.

Příklad: Pokud je vstupem nějakého problému například graf, můžeme ho reprezentovat jako seznam vrcholů a hran:

Například následující graf



můžeme reprezentovat jako slovo

$(1, 2, 3, 4, 5), ((1, 2), (2, 4), (4, 3), (3, 1), (1, 1), (2, 5), (4, 5), (4, 1))$

v abecedě $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ,, (,)\}$.

Kódování vstupu a výstupu

Můžeme se omezit na případ, kdy jsou vstupy i výstupy kódovány jako slova v abecedě $\{0, 1\}$ (tj. jako sekvence bitů).

Symbole jakékoli jiné abecedy lze reprezentovat jako sekvence bitů.

Příklad: Abeceda $\{a, b, c, d, e, f, g\}$

a \leftrightarrow 001

b \leftrightarrow 010

c \leftrightarrow 011

d \leftrightarrow 100

e \leftrightarrow 101

f \leftrightarrow 110

g \leftrightarrow 111

Slovo 'defb' můžeme reprezentovat jako '100101110010'.

Na slova v abecedě $\Sigma = \{0, 1\}^*$ (tj. sekvence bitů) se můžeme dívat jako na zápisy čísel v binární soustavě.

Alternativně bychom se tedy mohli omezit na případ, kdy

$$In = Out = \mathbb{N},$$

kde $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ je množina přirozených čísel.

Pokud se omezíme na to, že vstupy jsou slova z nějaké abecedy Σ , můžeme na rozhodovací problémy pohlížet jako na jazyky.

Jazyk odpovídající danému rozhodovacímu problému je množina těch slov ze Σ^* , která reprezentují ty vstupy, pro něž je odpověď **ANO**.

Příklad: Jazyk L tvořený těmi slovy ze $\{0, 1\}^*$, která jsou binárním zápisem nějakého prvočísla.

Například $101 \in L$, ale $110 \notin L$.

Často se při zkoumání algoritmů a problému omezujeme jen na rozhodovací problémy.

Není to však na úkor obecnosti, neboť libovolný obecný problém je možné vhodným způsobem přeformulovat jako rozhodovací problém, s tím, že když najdeme algoritmus, který by řešil tento rozhodovací problém, tak bychom snadno sestrojili algoritmus, který by řešil původní problém, a naopak.

Pokud například máme problém P , kde:

- vstupy jsou prvky z nějaké množiny In
- výstupy jsou slova z $\{0, 1\}^*$

můžeme tento problém přeformulovat jako následující rozhodovací problém:

Problém

Vstup: Prvek $x \in In$ a číslo k .

Otázka: Když z je výstup, který odpovídá vstupu x problému P , má k -tý bit slova z hodnotu 1?

Předpokládejme, že máme dán nějaký problém P .

Jestliže existuje nějaký algoritmus, který řeší problém P , pak říkáme, že problém P je **algoritmicky řešitelný**.

Jestliže P je rozhodovací problém a jestliže existuje nějaký algoritmus, který problém P řeší, pak říkáme, že problém P je **rozhodnutelný**.

Když chceme ukázat, že problém P je algoritmicky řešitelný, stačí ukázat nějaký algoritmus, který ho řeší (a případně ukázat, že daný algoritmus problém P skutečně řeší).

U mnohých problémů je na první pohled zřejmé, že jsou algoritmicky řešitelné, jako třeba:

- Třídění
- Hledání nejkratší cesty v grafu
- Prvočíselnost

kde stačí probrat všechny možnosti (kterých je ve všech těchto případech konečně mnoho), i když takový triviální algoritmus založený na řešení **hrubou silou** nemusí být zrovna efektivní.

Na druhou stranu existuje celá řada problémů, u kterých to tak jasné není.

- Najít algoritmus a dokázat, že řeší daný problém, může být velmi netriviální úkol.
- Algoritmus, který by řešil daný problém, nemusí vůbec existovat.

Dosavadní definice pojmu algorithmus byla poněkud vágní.

Pokud bychom pro nějaký problém chtěli ukázat, že neexistuje algorithmus, který by daný problém řešil, tak by to s takovouto neurčitou definicí pojmu algorithmus asi nešlo.

Intuitivně chápeme, co by měl mít algorithmus za vlastnosti:

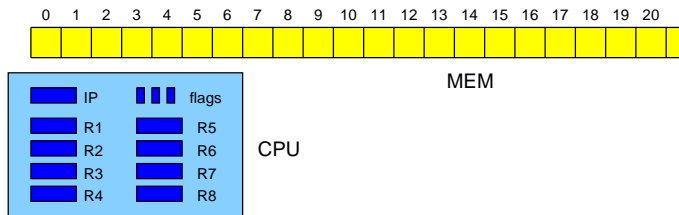
- Měl by se skládat z jednoduchých kroků, které je možno vykonávat „mechanicky“, bez porozumění problému.
- Objekty, se kterými algorithmus pracuje, i prováděné operace by měly být konečné.

Například program v libovolném programovacím jazyce dané vlastnosti zcela jistě má.

Ať už je program napsán v jakémkoliv programovacím jazyce, jsou jeho instrukce nakonec prováděny na hardwaru nějakého konkrétního počítače na úrovni instrukcí procesoru.

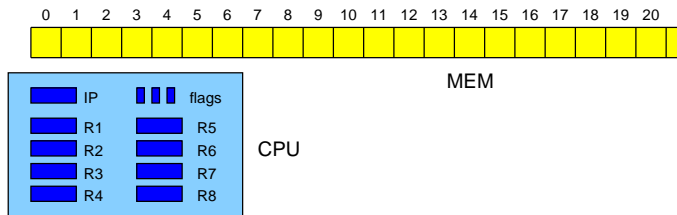
Je tedy jasné, že **každý** program v **každém** programovacím jazyce bychom mohli zapsat jako program tvořený pouze instrukcemi strojového kódu nějakého procesoru.

Typická architektura naprosté většiny počítačů vypadá následovně:



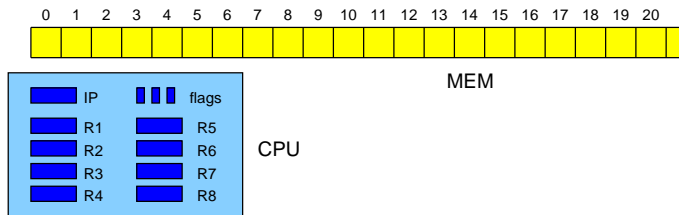
- Počítač má **paměť** skládající se z velkého množství paměťových buněk.
- Každá buňka může obsahovat číslo určité velikosti, typicky 1 byte (8 bitů), tj. číslo v rozsahu **0 . . 255**.
- Buňky jsou očíslovány. Číslo buňky se nazývá její **adresa**.

Typická architektura naprosté většiny počítačů vypadá následovně:



- Instrukce jsou uloženy v paměti (každá instrukce má svůj číselný **kód**) a jsou sekvenčně vykonávány **procesorem**.
- Procesor udržuje tzv. **čítač instrukcí IP**, který obsahuje adresu aktuálně prováděné instrukce.
- Procesor načte instrukci z adresy určené **IP**, zvětší **IP** o délku načtené instrukce a provede danou instrukci.

Typická architektura naprosté většiny počítačů vypadá následovně:



- Procesor obsahuje několik **registru** pevné délky (např. 32 nebo 64 bitů).
- Většina operací je prováděna na registrech.
- Procesor obsahuje **příznaky (flags)**, které umožňují testovat výsledek poslední operace (např. přetečení, jestli je výsledek nula apod.).

Typické instrukce:

- Načtení obsahu paměťové buňky (resp. několika po sobě jdoucích buněk) do některého registru (**LOAD**).
- Uložení obsahu registru do některé paměťové buňky (resp. několika po sobě jdoucích buněk) (**STORE**).

Poznámka: Adresa buňky je buď přímá (tj. je přímo součástí instrukce) nebo nepřímá (uložená v některém registru, případně spočítaná z obsahu jednoho nebo několika registrů).

- Načtení obsahu jednoho registru do jiného registru (**MOV**).
- Aritmetické instrukce (**ADD, SUB, MUL, DIV, NEG, CMP, INC, DEC, ...**).
- Logické instrukce (**AND, OR, XOR, NOT, ...**).
- Bitové posuny a rotace (**SHL, SHR, ...**)

Typické instrukce (pokračování):

- Nepodmíněný skok (**JMP**).

Poznámka: Cílová adresa může být přímá nebo nepřímá.

- Podmíněné skoky (**JZERO**, **JGTZ**, ...).
- Volání podprogramů (**CALL**, **RET**).
- Různé speciální instrukce – práce se vstupem a výstupem, obsluha přerušení, mody činnosti procesoru, řízení přístupu do paměti, stránkování apod.

Příklad: Instrukce zapsaná ve vyšším programovacím jazyce jako

$$x = y + 2$$

může být realizována následující sekvencí instrukcí určitého (hypotetického) procesoru:

```
LOAD    R5,[0x001b7c44]
ADD     R5,2
STORE   [0x001b7c38],R5
```

Poznámka: Předpokládáme, že proměnná x je uložena na adrese `0x001b7c38` a proměnná y na adrese `0x001b7c44`.

Stroj RAM (Random Access Machine) je idealizovaný model počítače.

Rozdíly oproti skutečnému počítači:

- Velikost paměti není omezena (adresa může být libovolné přirozené číslo).
- Velikost obsahu jednotlivých buněk není omezena (buňka může obsahovat libovolné celé číslo).
- Čte data sekvenčně ze vstupu, který je tvořen sekvencí celých čísel. Ze vstupu lze pouze číst.
- Zapisuje data sekvenčně na výstup, který je tvořen sekvencí celých čísel. Na výstup je možné pouze zapisovat.

RAM (Random Access Machine) neboli **počítač s libovolným přístupem** se skládá z těchto částí:

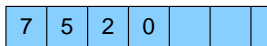
- **Programová jednotka** – obsahuje program stroje RAM a ukazatel na právě prováděnou instrukci
- **Pracovní paměť** tvořená buňkami očíslovanými $0, 1, 2, \dots$; obsah buněk je možno číst i do nich zapisovat
- **Vstupní páska** – je z ní možné pouze číst
- **Výstupní páska** – je na ni možno pouze zapisovat

Stroj RAM

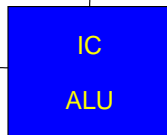
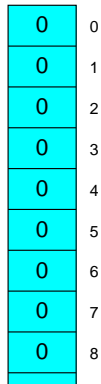
programová
jednotka

1	READ
2	JZERO 10
3	STORE *3
4	ADD 2
5	STORE 2
6	LOAD 1
7	ADD =1
8	STORE 1
9	JUMP 1
10	LOAD 2
11	DIV 1
12	STORE 2
13	LOAD =0
14	STORE 1

vstup



pracovní
paměť



výstup



Buňky 0 a 1 mají speciální význam a slouží jako „registry“ stroje RAM:

- **Buňka 0** – **pracovní registr** (akumulátor) – registr, který je jedním z operandů většiny instrukcí a do kterého se ukládá výsledek většiny operací.
- **Buňka 1** – **indexový registr** – je použit při nepřímém adresování.

Tvary **operandů** instrukcí ($i \in \mathbb{N}$):

tvar	hodnota operandu
$=i$	přímo číslo udané zápisem i
i	číslo obsažené v buňce s adresou i
$*i$	číslo v buňce s adresou $i + j$, kde j je aktuální obsah indexového registru

Příklad:

LOAD <op>

načte obsah operandu <op> do pracovního registru (tj. do buňky číslo 0).

- LOAD =5 – uloží do pracovního registru hodnotu 5
- LOAD 5 – uloží do pracovního registru obsah buňky číslo 5
- LOAD *5 – uloží do pracovního registru obsah buňky číslo $5 + j$, kde j je aktuální obsah indexového registru

Instrukce vstupu a výstupu (jsou bez operandu):

- READ** – do pracovního registru se uloží číslo, které je v políčku snímaném vstupní hlavou, a vstupní hlava se posune o jedno políčko doprava
- WRITE** – výstupní hlava zapíše do snímaného políčka výstupní pásky obsah pracovního registru a posune se o jedno políčko doprava

Instrukce přesunu v paměti:

- LOAD <op>** – do pracovního registru se načte hodnota operandu
- STORE <op>** – hodnota operandu se přepíše obsahem pracovního registru (zde se nepřipouští operand tvaru $=i$)

Instrukce aritmetických operací:

- ADD <op>** – číslo v pracovním registru se zvýší o hodnotu operandu (tedy přičte se k němu hodnota operandu)
- SUB <op>** – od čísla v pracovním registru se odečte hodnota operandu
- MUL <op>** – číslo v pracovním registru se vynásobí hodnotou operandu
- DIV <op>** – číslo v pracovním registru se celočíselně vydělí hodnotou operandu (do pracovního registru se uloží výsledek příslušného celočíselného dělení)

Instrukce skoku:

- JUMP** <návěští> – výpočet bude pokračovat instrukcí určenou návěštím
- JZERO** <návěští> – je-li obsahem pracovního registru číslo 0, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí
- JGTZ** <návěští> – je-li číslo v pracovním registru kladné, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí

Instrukce zastavení:

- HALT** – výpočet je ukončen („regulérně“ zastaven)

Příklad programu pro stroj RAM, který pro posloupnost čísel na vstupu spočítá aritmetický průměr (zaokrouhlený dolů) a na výstup vypíše odchylky jednotlivých čísel ze vstupu od tohoto průměru:

1	READ	13	LOAD =0
2	JZERO 10	14	STORE 1
3	STORE *3	15	LOAD *3
4	ADD 2	16	JZERO 23
5	STORE 2	17	SUB 2
6	LOAD 1	18	WRITE
7	ADD =1	19	LOAD 1
8	STORE 1	20	ADD =1
9	JUMP 1	21	STORE 1
10	LOAD 2	22	JUMP 15
11	DIV 1	23	HALT
12	STORE 2		

Každý program v každém programovacím jazyce by mohl být realizován jako program stroje RAM.

Není složité (i když je to trochu pracné) si rozmyslet, že libovolný algoritmus prováděný strojem RAM je možné realizovat také Turingovým strojem.

Turingův stroj je schopen realizovat libovolný algoritmus, který by bylo možné zapsat jako program v nějakém programovacím jazyce.

Stroj RAM a Turingův stroj

Turingův stroj pracuje se slovy nad nějakou abecedou, zatímco stroj RAM s čísly. Číslo ale můžeme zapisovat jako sekvence symbolů a naopak symboly nějaké abecedy můžeme zapisovat jako čísla.

Například následující vstup stroje RAM

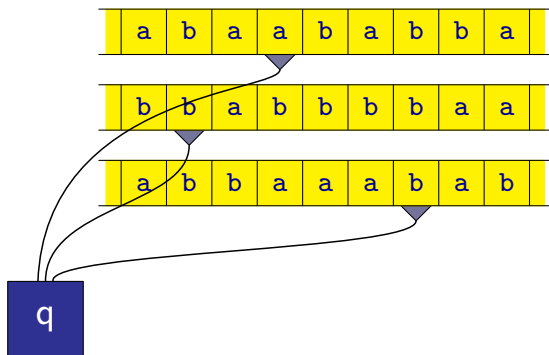
5	13	-3	0	6	
---	----	----	---	---	--

může být v případě Turingova stroje reprezentován jako

#	1	0	1	#	1	1	0	1	#	-	1	1	#	0	#	1	1	0	#
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

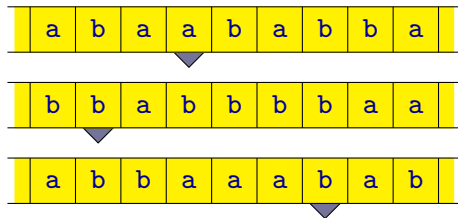
Vícepáskový Turingův stroj

Při konstrukci Turingova stroje k danému stroji RAM může být jednodušší zkonstruovat nejprve **vícepáskový Turingův stroj**:

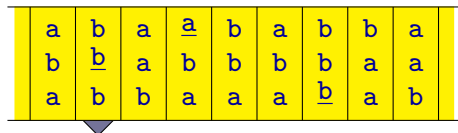


Vícepáskový Turingův stroj

Činnost vícepáskového Turingova stroje

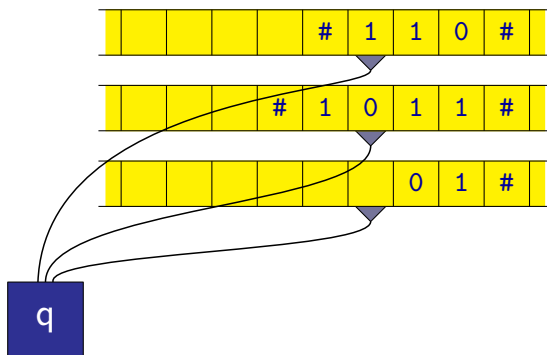


je možné simulovat jednopáskovým Turingovým strojem



Vícepáskový Turingův stroj

Příklad: Stroj, který dostane jako vstup dvě přirozená čísla zapsaná binárně a oddělená znaky # (např. čísla 6 a 11 budou zapsaná jako slovo “#110#1011#”) a spočítá jejich součet.



Turingův stroj simulující činnost stroje RAM

Turingův stroj simulující činnost stroje RAM bude mít několik pásek:

- Pásku reprezentující vstupní pásku stroje RAM.
- Pásku reprezentující výstupní pásku stroje RAM.
- Pásku, na které bude uložen obsah pracovního registru.
- Pásku, na které bude uložen obsah indexového registru.
- Pásku, na které bude uložen obsah ostatních buněk paměti stroje RAM.
- Pásku, na které bude uložena adresa buňky paměti, se kterou se aktuálně pracuje.
- Několik dalších pomocných pásek (pro uložení mezivýsledků operací apod.)

Turingův stroj simulující činnost stroje RAM

Turingův stroj si bude v řídicí jednotce pamatovat, která instrukce stroje RAM se právě provádí.

Provedení většiny instrukcí není složité:

- Instrukce **READ** – zkopírování hodnoty (ohraničené znaky “#”) ze vstupní pásky na pásku reprezentující pracovní registr.
- Instrukce **WRITE** – zkopírování hodnoty pracovní registru na výstupní pásku.
- Instrukce **JUMP** – změní se jen stav řídicí jednotky Turingova stroje.
- Instrukce **JZERO** a **JGTZ** – snadno otestujeme obsah pracovního registru a podle výsledku změníme stav řídicí jednotky Turingova stroje.

Aritmetické instrukce (**ADD**, **SUB**, **MUL**, **DIV**) jsou také relativně jednoduché:

- Hodnotu druhého operandu zapíšeme na pomocnou pásku.
- Provedeme operaci (např. sčítání) bit po bitu, výsledek ukládáme na další pomocnou pásku.
- Výsledek zkopírujeme na pásku s obsahem pracovního registru.

Poznámka: Násobení je možné realizovat pomocí série sčítání a bitových posunů.

Turingův stroj simulující činnost stroje RAM

Asi nejsložitější je realizace paměti stroje RAM.

Jednou z možností je pamatovat si jen obsah těch buněk, se kterými stroj RAM v průběhu své činnosti někdy pracoval (víme, že všechny ostatní obsahují hodnotu 0).

Příklad: Stroj RAM zatím pracoval jen s buňkami 2, 3 a 6:

- Buňka 2 obsahuje hodnotu 11.
- Buňka 3 obsahuje hodnotu -1.
- Buňka 6 obsahuje hodnotu 2.

Obsah pásky Turingova stroje reprezentující buňky paměti stroje RAM bude následující:

\$	#	1	0	:	1	0	1	1	#	1	1	:	-	1	#	1	1	0	:	1	0	#	\$
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

Instrukce **LOAD**:

- Hledanou adresu uložíme na příslušnou pásku obsahující adresu, se kterou se aktuálně pracuje.
- Budeme hledat příslušnou adresu na pásce reprezentující obsah paměti stroje RAM.
(Pokud ji nenajdeme, přidáme ji na konec, s tím, že obsahuje hodnotu 0.)
- Příslušnou hodnotu zkopírujeme na pásku reprezentující pracovní registr.

Instrukce **STORE**:

- Podobně jako u instrukce **LOAD** najdeme příslušné místo na pásce.
- Zbytek pásky s obsahem paměti stroje RAM zkopírujeme na pomocnou pásku.
- Na příslušné místo zkopírujeme hodnotu z pracovního registru.
- Zbytek pásky, který jsme zkopírovali na pomocnou pásku, zkopírujeme zpět (za nově zapsanou hodnotu).

Churchova-Turingova teze

Každý algoritmus je možné realizovat nějakým Turingovým strojem.

Není to věta, kterou by bylo možno dokázat v matematickém smyslu – není formálně definováno, co je to algoritmus.

Tezi formulovali nezávisle na sobě v polovině 30. let 20. století Alan Turing a Alonzo Church.

Ve stejné době bylo navrženo několik různých formalismů zachycujících pojem algoritmus:

- Turingovy stroje (Alan Turing)
- lambda kalkulus (Alonzo Church)
- rekurzivní funkce (Stephan Kleene)
- produkční systémy (Emil Post)
- ...

Dále můžeme uvést:

- Libovolný (obecný) programovací jazyk (jako např. C, Java, Lisp, Haskell, Prolog apod.).

Všechny tyto modely jsou ekvivalentní z hlediska algoritmů, které jsou schopny realizovat.