

Algoritmy a datové struktury II

TIN061

Ondřej Čepek

Sylabus

Vyhledávání v textu

Toky v sítích

Aritmetické algoritmy

Paralelní aritmetické algoritmy

Základní geometrické algoritmy v rovině

Převoditelnost problémů a třídy časové složitosti

Aproximační algoritmy

Pravděpodobnostní algoritmy a kryptografie
(Dynamické programování)

Vyhledávání řetězců v textu

Σ konečná abeceda (množina znaků)

Σ^* množina slov nad abecedou Σ (konečné posloupnosti znaků)

délka slova : $u = x_1x_2 \dots x_m \in \Sigma^*$ \Rightarrow $\text{length}(u) = m$ (počet znaků ve slově)

skládání (konkatenace) slov u a v :

$u = x_1x_2 \dots x_m, v = y_1y_2 \dots y_n \Rightarrow uv = x_1x_2 \dots x_my_1y_2 \dots y_n$

(a samozřejmě $\text{length}(uv) = \text{length}(u) + \text{length}(v)$)

prázdné slovo ε ($\forall u \in \Sigma^*$ platí $u\varepsilon = \varepsilon u = u$)

předpona (prefix): $u \in \Sigma^*$ je předponou $v \in \Sigma^*$ pokud $\exists w \in \Sigma^* : uw = v$

přípona (sufix): $u \in \Sigma^*$ je příponou $v \in \Sigma^*$ pokud $\exists w \in \Sigma^* : wu = v$

pokud $w \neq \varepsilon$ tak se jedná o vlastní předponu (příponu)

Řešená úloha:

vstup: abeceda Σ , prohledávaný text $x = x_1x_2 \dots x_n \in \Sigma^*$ a hledané vzorky $K = \{y_1, y_2, \dots, y_k\}$, kde $y_p = y_{p,1} \dots y_{p,\text{length}(p)} \in \Sigma^*$ pro $p = 1, \dots, k$

výstup: všechny výskyty vzorků v x , tj. všechny dvojice $[i, p]$ takové, že y_p je příponou slova $x_1x_2 \dots x_i$ ($1 \leq i \leq n$ a $1 \leq p \leq k$)

Naivní algoritmus

```
for  $p := 1$  to  $k$  do
  for  $i := 1$  to  $(n - \text{length}(p) + 1)$  do
    begin  $j := 0$ ;
      while  $(j < \text{length}(p))$  and  $(x_{i+j} = y_{p,1+j})$  do  $j := j + 1$ ;
      if  $(j = \text{length}(p))$  then Report( $i, p$ )
    end
  end
```

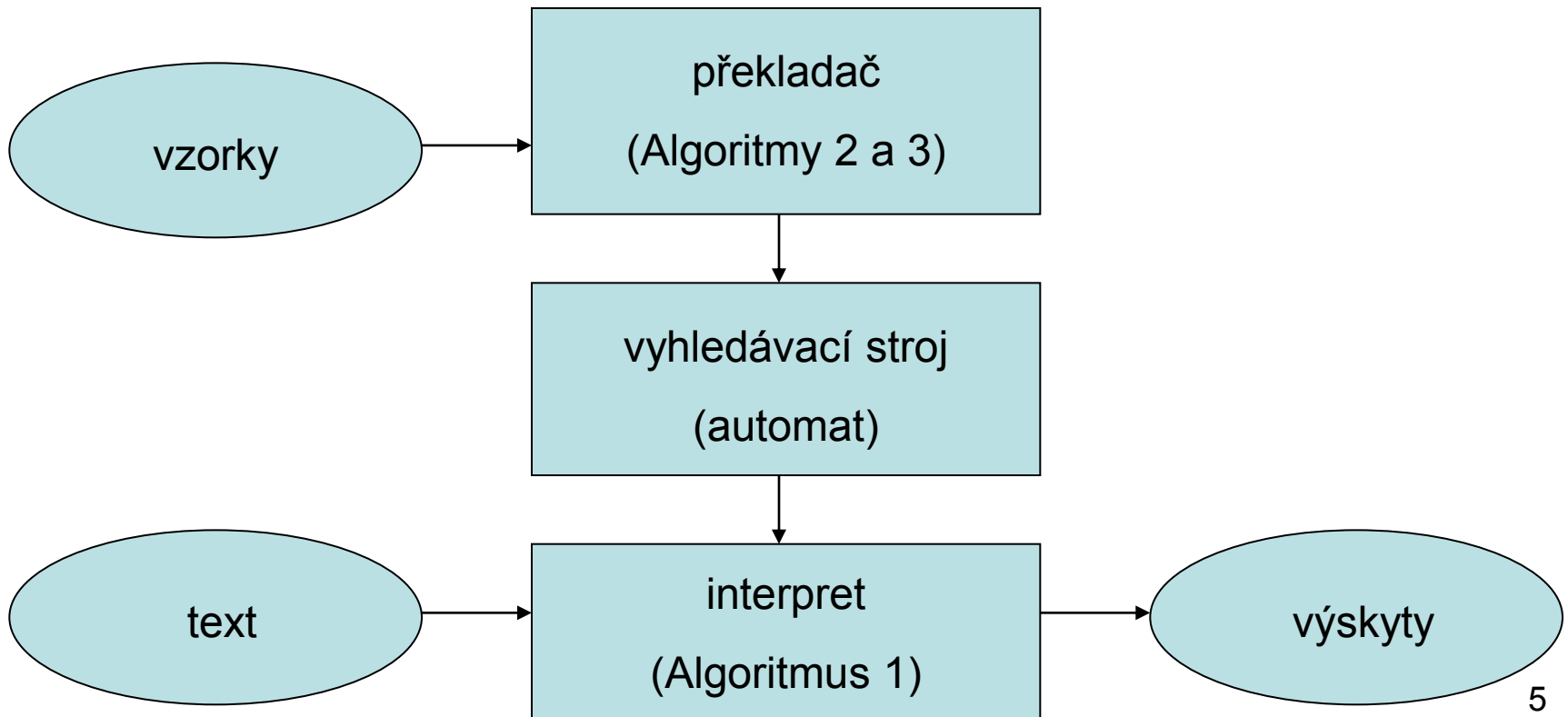
Algoritmus na míru

```
 $c := 0$ ;
for  $i := 1$  to  $n$  do
  begin
    if  $(x_i = a)$  then  $c := c + 1$ 
    else begin
      if  $(c \geq h - 1)$  then Report( $i, 1$ ) ;  $c := 0$ 
    end
  end
end
```

Ukážeme, že algoritmus na míru (= konečný automat) lze vyrobit pro libovolný vzorek nebo množinu vzorků, a to tak, že:

- výroba automatu (vyhledávacího stroje) trvá $\Theta(h \cdot |\Sigma|)$
- vyrobený automat prohlédne text za $\Theta(n)$
- celková práce algoritmu je $\Theta(n + h \cdot |\Sigma|)$

Algoritmus Aho-Corasick(ová) (1975)



Vyhledávací stroj pro konečnou abecedu Σ a množinu vzorků K je čtveřice

$M = (Q, g, f, \text{out})$, kde

1. $Q = \{0, 1, \dots, q\}$ je množina stavů
2. $g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$ je přechodová funkce, pro kterou platí $\forall x \in \Sigma: g(0, x) \in Q$ (symbol \perp znamená „nedefinováno“, přechod ze stavu 0 je definován $\forall x \in \Sigma$)
3. $f : Q \rightarrow Q$ je zpětná funkce, pro kterou platí $f(0) = 0$ (nastupuje pokud g dá \perp)
4. $\text{out} : Q \rightarrow P(K)$ je výstupní funkce (pro daný stav vydá podmnožinu vzorků)

Algoritmus 1 (interpret vyhledávacího stroje)

vstup: $x = x_1 \dots x_n \in \Sigma^*$, $K = \{y_1 \dots y_k\}$, $M = (Q, g, f, \text{out})$

$\text{state} := 0$;

for $i := 1$ to n do

begin

(1) while $(g(\text{state}, x_i) = \perp)$ do $\text{state} := f(\text{state})$;

(2) $\text{state} := g(\text{state}, x_i)$;

(3) for all $y_p \in \text{out}(\text{state})$ do Report (i, p)

end

Klíčové vlastnosti vyhledávacího stroje (konečného automatu):

1. přechodová funkce g

graf funkce g (pro definované dvojice bez smyčky ve stavu 0) je ohodnocený strom, pro který

- stav 0 je kořenem stromu
- každá cesta z kořene je ohodnocena nějakou předponou nějakého vzorku z K
- každá předpona každého vzorku z K ohodnocuje cestu z kořene do nějakého (právě jednoho) stavu $s \Rightarrow$ říkáme, že předpona (slovo) u reprezentuje stav s (speciálně prázdné slovo ϵ reprezentuje stav 0)
- hloubka stavu s reprezentovaného slovem u je definována jako $d(s) = \text{length}(u)$ a pro funkci g (na hranách stromu) platí: $d(g(s, x_i)) = d(s) + 1$

2. zpětná funkce f

pro každý stav s reprezentovaný slovem u platí, že stav $f(s)$ je reprezentován nejdelší vlastní příponou slova u , která je zároveň předponou nějakého vzorku z K

3. výstupní funkce out

pro každý stav s reprezentovaný slovem u a pro každý vzorek $y_p \in K$ platí:

$y_p \in \text{out}(s)$ tehdy a jen tehdy když je y příponou slova u

Algoritmus 2 (konstrukce vyhledávacího stroje – 1.fáze)

vstup: $K = \{y_1 \dots y_k\}$ {množina vzorků}
výstup: $Q = \{0, \dots, q\}$ {množina stavů vyhledávacího stroje}
 $g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$ {výstupní funkce splňující Vlastnost 1}
 $o : Q \rightarrow P(K)$ {„polotovar“ výstupní funkce out}
 procedure Enter($y_{p,1} \dots y_{p,m}$); {připojení slova y_p ke grafu funkce g }
 begin $stav := 0; i := 1;$
 while ($i \leq m$) and ($g(stav, y_{p,i}) \neq \perp$) do
 begin $stav := g(stav, y_{p,i});$ {pohyb po již hotové větvi}
 $i := i + 1$ {posun ve slově y_p }
 end;
 while ($i \leq m$) do
 begin $Q := Q \cup \{q+1\}; q := q+1$ {vytvoření nového stavu}
 for all $x \in \Sigma$ do $g(q, x) := \perp;$
 $g(stav, y_{p,i}) := q;$ {prodloužení větve}
 $stav := q;$ {pokročení do nového stavu}
 $i := i + 1$ {posun ve slově y_p }
 end;
 $o(stav) := \{y_p\}$
 end; {of Enter}
 begin $Q := \{0\};$ for all $x \in \Sigma$ do $g(0, x) := \perp;$ {hlavní program}
 for $p := 1$ to k do Enter(y_p);
 for all $x \in \Sigma$ do if $g(0, x) = \perp$ then $g(0, x) = 0$
 end.

Algoritmus 3 (konstrukce vyhledávacího stroje – 2.fáze)

```

vstup:     $Q = \{0, \dots, q\}$                                 {množina stavů vyhledávacího stroje}
             $g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$                 {výstupní funkce splňující Vlastnost 1}
             $o : Q \rightarrow P(K)$                                 {„polotovary“ výstupní funkce out}
výstup:    $f : Q \rightarrow Q$                                 {zpětná funkce splňující Vlastnost 2}
             $out : Q \rightarrow P(K)$                             {výstupní funkce splňující Vlastnost 3}

vytvoř prázdnou frontu stavů;
 $f(0) := 0$ ;  $out(0) := \emptyset$ ;
for all  $x \in \Sigma$  do begin                                     {zpracuje potomky kořene}
     $s := g(0, x)$ ;
    if  $s \neq 0$  then
        begin     $f(s) := 0$ ;  $out(s) := o(s)$ ;
                  zařaď s na konec fronty
        end
    end;
while fronta není prázdná do
begin     $r :=$  první prvek z fronty (a vyřaď r z fronty);
    for all  $x \in \Sigma$  do if  $g(r, x) \neq \perp$  then    {zpracuje potomky uzlu r}
    begin     $s := g(r, x)$ ;  $t := f(r)$ ;
            while  $g(t, x) = \perp$  then  $t := f(t)$ ;
             $f(s) := g(t, x)$ ;  $out(s) := o(s) \cup out(f(s))$ ;
            zařaď s na konec fronty
    end
end
end

```

Algoritmus Knuth-Morris-Pratt

- zjednodušená verze algoritmu Aho-Corasick(ová) pro vyhledávání jediného vzorku
- kratší a snadněji pochopitelný popis
- (mírně) lepší asymptotická složitost ($\Theta(n + h)$ místo $\Theta(n + h \cdot |\Sigma|)$)
- graf přechodové funkce g není strom ale řetězec, což umožňuje g explicitně vůbec nepoužívat (zde je ta úspora ve složitosti preprocessingu, protože g má $h \cdot |\Sigma|$ přechodů), funkce g je používána pouze implicitně
- zpětná funkce f se zde nazývá prefixová funkce a protože v případě jediného vzorku odpovídá číslu stavu délce prefixu daného vzorku, který je daným stavem reprezentován, tak má f jednodušší definici:
 $f(s)$ je délka nejdelší vlastní přípony slova reprezentovaného stavem s (toto slovo je prostě předpona délky s daného vzorku), která je zároveň předponou (daného vzorku)
- výstupní funkce je triviální, ve stavu h hlásí výskyt (jediného) vzorku, jinde nic

procedura Prefix (nahrazuje Algoritmy 2 a 3)

vstup: $K = \{y\}$ {jediný vzorek}
výstup: $f: Q \rightarrow Q$ {prefixová funkce}

```
f(1) := 0;  
t := 0;  
for q := 2 to h do  
begin   while (t > 0) and ( $y_{t+1} \neq y_q$ ) do t := f(t);  
        if ( $y_{t+1} = y_q$ ) then t := t + 1;  
        f(q) := t  
end
```

Algoritmus KMP (nahrazuje Algoritmus 1)

vstup: $x = x_1 \dots x_n \in \Sigma^*$, $K = \{y\}$, prefixová funkce f

```
state := 0;  
for i := 1 to n do  
begin  
  (1) while (state > 0) and ( $y_{state+1} \neq x_i$ ) do state := f(state);  
  (2) if  $y_{state+1} = x_i$  then state := state + 1;  
  (3) if (state = h) then begin Report (i);  
                          state := f(state)  
end  
end
```

Toky v sítích

Síť: orientovaný graf $G = (V, E)$ se dvěma vybranými vrcholy s, t (zdroj a stok) a kladnou kapacitou $c(u, v)$ na každé hraně $(u, v) \in E$. Kapacita je dodefinována i pro ostatní dvojice vrcholů: $c(u, v) = 0$ pokud $(u, v) \notin E$.

Tok: je funkce $f : V \times V \rightarrow \mathbb{R}$ splňující následující tři vlastnosti:

1. (Symetrie) $\forall u, v \in V : f(u, v) = -f(v, u)$
2. (Kapacita) $\forall u, v \in V : f(u, v) \leq c(u, v)$
3. (Zachování toku) $\forall u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0$ (jako Kirchhoffův zákon pro el.proud)

Pokud $f(u, v) > 0$ tak říkáme, že máme (nenulový) tok z u do v . Pokud $f(u, v) = c(u, v)$ tak říkáme, že je hrana (u, v) **saturovaná**. Velikost toku f značíme $|f|$ a je to celkový čistý tok ze zdroje, tj. $|f| = \sum_{v \in V} f(s, v)$.

Problém: hledání **maximálního toku**, tj. toku maximální velikosti.

Řez: v kontextu toků je **řez** dvojice množin vrcholů X, Y taková, že $X \cup Y = V$, $s \in X$, $t \in Y$.
Kapacita řezu X, Y je součet kapacit hran jdoucích přes řez, tj. $c(X, Y) = \sum_{u \in X, v \in Y} c(u, v)$.
Minimální řez je řez s minimální kapacitou. **Tok přes řez** X, Y je součet toků po hranách jdoucích přes řez, tj. $f(X, Y) = \sum_{u \in X, v \in Y} f(u, v)$.

Lemma 1: Pro každý tok f a řez X, Y platí, že tok přes řez X, Y je roven velikosti toku, tj. $f(X, Y) = |f|$.

Díky Lemma 1 a triviálnímu faktu, že $f(X,Y) \leq c(X,Y)$ pro každý řez X,Y (díky kapacitnímu omezení) platí, že velikost maximálního toku je nejvýše rovna kapacitě minimálního řezu. Ukážeme, že zde vždy platí rovnost.

Reziduální kapacita toku f je funkce $r : V \times V \rightarrow \mathbb{R}$ definovaná předpisem

$$r(u,v) = c(u,v) - f(u,v)$$

Reziduální graf pro f : orientovaný graf $R = (V, E')$, kde $(u,v) \in E'$ tehdy a jen tehdy, když platí $r(u,v) > 0$ a hodnotu $r(u,v)$ pak nazýváme kapacitou hrany (u,v) v reziduálním grafu.

Zlepšující cesta pro f : libovolná cesta p z s do t v grafu R . Reziduální kapacita $r(p)$ zlepšující cesty p je rovna minimálnímu $r(u,v)$ z hran (u,v) na cestě p . Velikost toku můžeme zvýšit až o $r(p)$ zvýšením toku (o stejné množství) na všech hranách cesty p . (Poznámka: při zvýšení $f(u,v)$ je nutné proporčně snížit $f(v,u)$ pro zachování symetrie.)

Věta (o max.toku a min.řezu): Následující podmínky jsou ekvivalentní

1. tok f je maximální tok
2. pro f neexistuje zlepšující cesta
3. platí $|f| = c(X,Y)$ pro nějaký řez X,Y

Věta o max.toku a min.řezu dává návod jak zkonstruovat maximální tok postupným zlepšováním.

Metoda zlepšující cesty: Začni s nulovým tokem a opakuj následující krok, dokud není dosaženo toku, pro který neexistuje zlepšující cesta.

Zlepšující krok: Pro aktuální tok f najdi zlepšující cestu p a zvyš velikost toku pomocí zvýšení toku na cestě p o $r(p)$.

Jednoduchá implementace (Ford a Fulkerson): Najdi libovolnou zlepšující cestu pomocí prohledání grafu R (libovolným algoritmem na prohledávání orientovaných grafů).

Poznámky:

- ze znalosti maximálního toku můžeme v čase $O(m)$ zkonstruovat minimální řez (viz důkaz Věty o max.toku a min.řezu)
- pokud jsou kapacity iracionální čísla, tak nemusí jednoduchá implementace metody zlepšující cesty skončit po konečném počtu kroků, velikost toku sice vždy konverguje ale nemusí konvergovat k velikosti maximálního toku
- pokud jsou kapacity racionální čísla, tak lze úlohu převést na ekvivalentní úlohu s celočíselnými kapacitami
- pokud jsou kapacity celočíselné, tak každý zlepšující krok zvýší velikost toku alespoň o jedna, a tudíž metoda končí po nejvýše $|f^*|$ zlepšujících krocích, navíc je zkonstruovaný maximální tok celočíselný (na každé hraně)

„Ideální verze“ metody zlepšující cesty:

Lemma 2: Vždy existuje posloupnost nejvýše m zlepšujících cest, pomocí nichž lze zkonstruovat maximální tok.

Implementace s maximálním zlepšením (Edmonds a Karp): V každém kroku najdi mezi všemi zlepšujícími cestami tu s maximální reziduální kapacitou.

Lemma 3: Necht' f je libovolný tok a necht' f^* je maximální tok na G . Potom velikost maximálního toku na reziduálním grafu R pro tok f je $|f^*| - |f|$.

Věta: Počet zlepšujících kroků při implementaci s maximálním zlepšením je $O(m \log c)$ kde c je maximální kapacita nějaké hrany.

Poznámky:

- toto už je polynomiální počet kroků vzhledem k velikosti dat
- odhad platí jen když jsou kapacity celočíselné (a metoda v tom případě samozřejmě opět konverguje k maximálnímu toku)
- zlepšující cestu s maximální reziduální kapacitou lze najít v polynomiálním čase pomocí modifikovaného Dijkstrova algoritmu (pro tzv. bottleneck problém), kde délku cesty neměříme součtem délek hran ale délkou nejkratší hrany (details nebudeme zkoumat, další probírané algoritmy jsou lepší)
- nyní bude cílem implementace jejíž časová složitost bude záviset jen na n a m

Implementace s nejkratším zlepšením (Edmonds a Karp): V každém kroku najdi mezi všemi zlepšujícími cestami tu nejkratší, tj. zlepšující cestu s minimálním počtem hran.

Věta: Počet zlepšujících kroků při implementaci s nejkratším zlepšením je $O(nm)$ a metoda zlepšující cesty běží v tomto případě v čase $O(nm^2)$.

Definice: Necht' f je tok a R je reziduální graf pro f . Úroveň $u(x)$ vrcholu x v R je délka nejkratší cesty z s do x v R . Úrovňový graf U pro tok f je podgraf grafu R , který obsahuje pouze vrcholy dosažitelné z s a pouze hrany (x,y) pro které $u(y) = u(x) + 1$.

Pozorování: Graf U lze zkonstruovat v čase $O(m)$ pomocí BFS a pokud existuje zlepšující cesta, tak U obsahuje všechny nejkratší zlepšující cesty.

Dalšího zlepšení lze dosáhnout tím, že místo zvyšování toku po jednotlivých nejkratších zlepšujících cestách použijeme všechny nejkratší zlepšující cesty „najednou“.

Definice: Tok f na grafu U se nazývá blokující tok, pokud každá cesta z s do t v grafu U (v původním U , tj. ne pozměněném tokem f) obsahuje saturovanou hranu.

Algoritmus (Dinic): Začni s nulovým tokem a opakuj následující (blokující) krok dokud existuje zlepšující cesta, tj. dokud je vrchol t dosažitelný v aktuálním úrovňovém grafu:

(Blokující) krok: Najdi blokující tok f' na úrovňovém grafu U definovaném pomocí aktuálního toku f . Nahradej tok f tokem $f + f'$ který je definován předpisem

$$(f + f')(x,y) = f(x,y) + f'(x,y)$$

Lemma 4: Dinicův algoritmus zastaví po nejvýše $n - 1$ blokujících krocích.

Jak hledat blokuující tok: Existuje několik metod, my probereme tu původní Dinicovu, která je nejprimitivnější.

Idea: Najdi cestu z **s** do **t** v **U** (pomocí DFS), zvyš po ní tok tak, aby nějaká hrana na nalezené cestě byla saturována. Z **U** vyhod' všechny nově saturované hrany a postup opakuj dokud je **t** dosažitelné z **s** v grafu **U**.

Formální popis: Začni s nulovým tokem a jdi na **Inicializaci**. Aktuálně prohlížený vrchol budeme označovat **x** a **p** bude zlepšující cesta z **s** do **x**.

Inicializace: Definuj $p = [s]$ a $x = s$. Jdi na Vpřed.

Vpřed: Pokud z x nevede v U žádná hrana jdi na **Vzad**. Jinak vezmi libovolnou hranu (x,y) , prodluž p o vrchol y a do x dosad' y (posuň aktuální vrchol). Pokud platí $y \neq t$ tak opakuji **Vpřed**, pokud $y = t$ jdi na **Zlepši**.

Vzad: Pokud $x = s$ tak zastav (neexistuje zlepšující cesta do t). Pokud $x \neq s$ tak nechť (v, x) je poslední hrana na p . Zkrat' p o vrchol x a hranu (v, x) odstraň z U . Do x dosad' y (posuň aktuální vrchol) a jdi na **Vpřed**.

Zlepší: Necht' $d = \min\{c(x,y) - f(x,y) \mid (x,y) \text{ je hrana v } p\}$. Přidej d k toku na všech hranách cesty p , odstraň z U všechny nově saturované hrany a jdi na **Inicializace**.

Věta: Dinicův algoritmus nalezne blokující tok v úrovňovém grafu U v čase $O(nm)$ a maximální tok ve vstupním grafu G v čase $O(n^2m)$.

Metoda „preflow-push“

Definice: **Předtok** (preflow) je funkce $f : V \times V \rightarrow R$ splňující stejně jako tok podmínku symetrie a kapacity na každé hraně, ale která místo podmínky zachování toku má pro každý vrchol u (kromě zdroje s) podmínku $e(u) = \sum_{v \in V} f(u,v) \geq 0$, kde $e(u)$ je **přebytek** (exces) ve vrcholu u . Vrchol u různý od s a t se nazývá **aktivní** pokud má kladný přebytek ($e(u) > 0$).

Definice: Necht' f je předtok a R reziduální graf pro f . Pak funkce $h : V \rightarrow N$ je **výšková funkce** vzhledem k f pokud

- $h(s) = |V|$
- $h(t) = 0$
- $\forall (u,v) \in R : h(u) \leq h(v) + 1$

Pokud pro hranu $(u,v) \in R$ platí $h(u) = h(v) + 1$, tak se (u,v) nazývá **přípustná**.

Inicializace (pro generický preflow-push algoritmus):

1. Vynuluj všechna $h(u)$, $e(u)$ a $f(u,v)$.
2. Dosad' $h(s) := |V|$.
3. Pro všechny sousedy u zdroje s dosad'
 - $f(s,u) := c(s,u)$ a $f(u,s) := -c(s,u)$
 - $e(u) := c(s,u)$

Platí po inicializaci: f je předtok a h je výšková funkce vzhledem k f .

Algoritmus používá dvě základní akce :

ZATLAČ(u) (neboli **PUSH(u)**)

- **POUŽITÍ** : pokud je u aktivní ($e(u) > 0$) a existuje přípustná hrana (u,v) v R , tj. hrana splňující $r(u,v) > 0$ a $h(u) = h(v) + 1$.
- **AKCE** : pošli $d(u,v) = \min \{e(u), r(u,v)\}$ jednotek toku z u do v a příslušně aktualizuj $f(u,v)$, $f(v,u)$, $e(u)$ a $e(v)$.

Definice: **ZATLAČ(u)** je **saturující**, pokud se hrana (u,v) , po které je posílán tok, stane saturovanou (což nastane pokud $d(u,v) = r(u,v)$) a **nesaturující** v opačném případě (tj. pokud $d(u,v) = e(u) < r(u,v)$).

ZVEDNI(u) (neboli **LIFT(u)**)

- **POUŽITÍ** : pokud je u aktivní ($e(u) > 0$) a neexistuje přípustná hrana (u,v) v R .
- **AKCE** : $h(u) := 1 + \min \{h(v) \mid (u,v) \in R\}$

Algoritmus (generický preflow-push algoritmus):

- Inicializace
- Dokud existuje aktivní vrchol tak nějaký vyber (necht' je to u) a uplatni buď **ZATLAČ(u)** nebo **ZVEDNI(u)**.

Lemma 1: Pro každý vrchol u během celého běhu algoritmu platí:

- $h(u)$ nikdy neklesne
- každé $ZVEDNI(u)$ zvýší $h(u)$ alespoň o jedna

Lemma 2: Funkce h zůstává během celého běhu algoritmu výškovou funkcí vzhledem k aktuálnímu (před)toku f .

Lemma 3: Během celého běhu algoritmu platí, že v R neexistuje cesta z s do t .

Věta (korektnost algoritmu):

Pokud algoritmus skončí, tak f (v tom okamžiku) reprezentuje maximální tok.

Časová složitost: Za celou dobu běhu algoritmus udělá:

- $O(|V|^2)$ zvednutí
- $O(|V||E|)$ saturujících zatlačení
- $O(|V|^2|E|)$ nesaturujících zatlačení

což při implementaci potřebující $O(|V|)$ na každé zvednutí a $O(1)$ na každé zatlačení dává celkovou složitost algoritmu $O(|V|^2|E|)$.

Poznámka: „chytrým“ výběrem aktivních vrcholů lze časovou složitost snížit až na $O(|V||E| \log(|V|^2 / |E|))$, což je asymptoticky nejrychlejší známý algoritmus [Goldberg, Tarjan 88].

Násobení polynomů

Dva způsoby reprezentace polynomů:

1. Pomocí vektoru koeficientů (zde vektor komplexních čísel)

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x^1 + a_0x^0$$

$A(x)$ je polynom s **mezí stupně** n , $a_0 \dots a_{n-1}$ jsou jeho **koeficienty** a $A(x)$ je polynom **stupně** k pokud a_k je jeho nejvyšší nenulový koeficient

- Součet dvou polynomů: $\Theta(n)$
- Výpočet hodnoty v bodě: $\Theta(n)$ pomocí Hornerova schématu
- Součin dvou polynomů: $\Theta(n^2)$ (konvoluce příslušných vektorů koeficientů)

2. Pomocí funkčních hodnot v bodech

$$\{ (x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1}) \}$$

Takto lze jednoznačně reprezentovat libovolný polynom $A(x)$ s mezí stupně n , a to libovolnou n -ticí ve které jsou všechny x_i po dvou různé a $y_i = A(x_i)$ pro každé i .

- Výpočet hodnoty v bodě: mimo zadané body nelze bez konverze k reprezentaci pomocí vektoru koeficientů
- Součet dvou polynomů: $\Theta(n)$
- Součin dvou polynomů: $\Theta(n)$ ale **POZOR** potřebujeme $2n$ bodů

Konverze mezi oběma reprezentacemi

1. Koeficienty \rightarrow Dvojice (**evaluate**): $\Theta(n^2)$ pomocí Hornerova schématu (i pro $2n$ bodů), ale lze v $\Theta(n \log n)$ když body pro výpočet funkčních hodnot vybrány „chytře“
2. Dvojice \rightarrow Koeficienty (**interpolate**): obecně $\Theta(n^3)$ Gaussovou eliminací nebo $\Theta(n^2)$ pomocí Lagrangeovy formule (nebudeme probírat), ale lze v $\Theta(n \log n)$ když známe funkční hodnoty v „chytře“ vybraných bodech

Násobení polynomů v $\Theta(n \log n)$

Chytře vybranými body budou $2n$ -té komplexní odmocniny čísla 1, které budeme značit

$$w_{2n}^0, w_{2n}^1, \dots, w_{2n}^{2n-1}$$

Násobení pro vstup $a_0 \dots a_{n-1}$ a $b_0 \dots b_{n-1}$ (dva vektory koeficientů polynomů s mezí stupně n) pak bude probíhat následovně:

1. Doplníme oba vektory n nulami na posloupnosti délky $2n-1$ (to je nová mez stupně)
2. Evaluate: spočítáme funkční hodnoty $A(x)$ i $B(x)$ ve všech $2n$ odmocninách čísla 1
3. Násobení: bod po bodu $C(x) = A(x)B(x)$ ve všech $2n$ odmocninách čísla 1
4. Interpolace: spočítáme vektor koeficientů polynomu $C(x)$ zadaného $2n$ funkčními hodnotami (ve všech odmocninách čísla 1)

Kroky 1 a 3 trvají $\Theta(n)$ a kroky 2 a 4 trvají $\Theta(n \log n)$ pokud jsou implementovány
Rychlou Fourierovou Transformací (FFT)

Vlastnosti n-tých odmocnin čísla 1 v komplexním oboru

Kraticí lemma: Pokud $n \geq 0$, $k \geq 0$ a $d > 0$ potom $w_{dn}^{dk} = w_n^k$.

Důsledek: Pro $n > 0$ sudé platí $w_n^{n/2} = w_2^1 = -1$.

Půlící lemma: Pro $n > 0$ sudé platí, že druhé mocniny n komplexních n -tých odmocnin čísla 1 jsou rovny $n/2$ komplexním $n/2$ -tým odmocninám čísla 1 (každá zastoupena 2x).

Součtové lemma: Pokud $n \geq 0$ a $k > 0$ a k není dělitelné n potom $\sum_{j=0}^{n-1} (w_n^k)^j = 0$.

Diskrétní Fourierova Transformace (DFT)

Polynom $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x^1 + a_0x^0$ s mezí stupně n zadaný vektorem koeficientů chceme reprezentovat pomocí funkčních hodnot v n komplexních n -tých odmocninách čísla 1, tj. chceme spočítat hodnoty

$$y_k = A(w_n^k) \text{ pro } k = 0, 1, \dots, n-1.$$

Vektor y (funkčních hodnot) se nazývá **DFT** vektoru a (koeficientů), píšeme $y = \text{DFT}_n(a)$.

Transformace opačným směrem se nazývá **inverzní DFT** a píšeme $a = \text{DFT}_n^{-1}(y)$.

Poznámka: pokud n je mezí stupně obou vstupních polynomů (které chceme vynásobit), tak zde vlastně pracujeme s $n' = 2n$, ale pro jednoduchost značení budeme používat n (navíc na asymptotickou složitost nemá vliv zda používáme n nebo n').

Rychlá Fourierova Transformace (FFT)

Algoritmus využívající strategii „rozděl a panuj“ pro spočítání $\text{DFT}_n(a)$, rozdělením $A(x)$ pomocí koeficientů lichého a sudého stupně na dva polynomy s mezí stupně $n/2$:

$$A^s(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$

$$A^l(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

Nyní zjevně platí $A(x) = A^s(x^2) + x A^l(x^2)$ (vztah označme (α)), takže úloha spočítat funkční hodnoty polynomu $A(x)$ v bodech $w_n^0, w_n^1, \dots, w_n^{n-1}$ je převedena na úlohu

- spočítat funkční hodnoty polynomů $A^s(x)$ a $A^l(x)$ (s mezí stupně $n/2$) v bodech $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n-1})^2$ (což je ale jen $n/2$ různých bodů díky půlícímu lemmatu), tj. místo původní úlohy vyřešit dvakrát zcela stejnou úlohu na polovičních datech
- zkombinovat výsledky podle vztahu (α) což zabere čas $\Theta(n)$

Celkový čas $T(n)$ pro FFT délky n je $T(n) = 2T(n/2) + \Theta(n)$ a tedy $T(n) = \Theta(n \log n)$.

Lemma Nechť V_n je Vandermondeho matice pro $w_n^0, w_n^1, \dots, w_n^{n-1}$. Pak prvek na pozici (j,k) v inverzní matici V_n^{-1} je roven w_n^{-kj} / n .

Tím pádem lze i inverzní DFT počítat pomocí FFT, protože spočítání koeficientů polynomu zadaného funkčními hodnotami y_0, y_1, \dots, y_{n-1} je to samé jako spočítání funkčních hodnot polynomu $Y(x)/n$ s koeficienty $y_0/n, y_1/n, \dots, y_{n-1}/n$ a sice v bodech $w_n^0, w_n^{-1}, \dots, w_n^{-(n-1)}$

Implementace FFT

1. Rekurzivní implementace (přímý přepis algoritmu)

```
REC-FFT(a);  
n := length(a);  
if n=1 then return a;  
p := e2πi / n;  
w := 1;  
as := (a0, a2, ..., an-2);  
al := (a1, a3, ..., an-1);  
ys := REC-FFT(a0);  
yl := REC-FFT(a1);  
for k := 0 to (n/2 - 1) do  
    yk := yks + w ykl;  
    yk+n/2 := yks - w ykl;  
    w := w p;  
return y
```

{n je mocnina dvojky}
{dno rekurze}
{principiální kořen w_n^1 = generátor ostatních kořenů}
{aktuálně zpracováváný kořen, začíná se s $w = w_n^0$ }

{příprava vstupních dat pro rekurzi}

{vlastní rekurze}

{spočítání hodnoty $A(w)$ v aktuálním kořeni $w = w_n^k$ }
{spočítání hodnoty $A(w)$ v protilehlém kořeni $w_n^{k+n/2}$ }
{posun na další kořen, tj. na w_n^{k+1} }

Časová složitost: je vidět, že $T(n) = 2T(n/2) + \Theta(n)$ a tedy $T(n) = \Theta(n \log n)$, protože práce mimo rekurzi je zjevně $\Theta(n)$.

2. Iterativní implementace (hlavní idea)

vznikne následujícími úpravami rekurzivní implementace

- tělo for cyklu z rekurzivní implementace nahradíme „butterfly“ operací
- strom rekurze procházíme po patrech odspodu
- iniciální pořadí listů stromu pro start algoritmu získáme bitovou reverzí

Časová složitost: na každém patře stromu je voláno právě $n/2$ butterfly operací, tj. na každém patře je celková práce $\Theta(n)$, což dohromady s faktem, že hloubka stromu je přesně $\log n$ dává časovou složitost algoritmu $\Theta(n \log n)$.

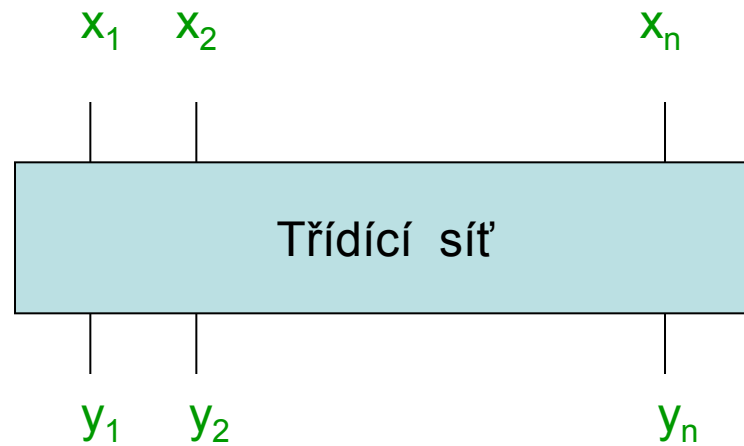
3. Paralelní implementace (hlavní idea)

- butterfly operace na stejném patře stromu se provádějí paralelně
- algoritmus lze realizovat hardwarově vhodným zřetězením butterfly obvodů do sítě, která má „šířku“ n a „hloubku“ $\log n$

Časová složitost: na každém patře nyní trvá (s použitím $n/2$ „procesorů“) veškerá práce $O(1)$, takže celková časová složitost je $\Theta(\log n)$.

Třídící síť

Třídící síť je obvod který má n vstupů z hodnotami z nějakého lineárně uspořádaného typu (tj.každé dvě hodnoty jsou porovnatelné) a n výstupů, na kterém jsou vstupní hodnoty setříděné (bez ohledu na to v jakém pořadí přišly na vstup).



Tento obvod obsahuje jediný typ hradla a sice **komparátor**, což je hradlo se dvěma vstupy x_1 a x_2 a dvěma výstupy y_1 a y_2 , pro které platí $y_1 = \min\{x_1, x_2\}$ a $y_2 = \max\{x_1, x_2\}$.

Formální definice třídící sítě:

- $K = \{K_1, K_2, \dots, K_s\}$ je množina komparátorů, s se pak nazývá **velikost** sítě
- $O = \{ (k,i) \mid 1 \leq k \leq s, 1 \leq i \leq 2 \}$ je množina výstupů (k je číslo komparátoru a i výstupu)
- $I = \{ (k,i) \mid 1 \leq k \leq s, 1 \leq i \leq 2 \}$ je množina vstupů
- $C = (K, f)$ je třídící síť, kde $f : O \rightarrow I$ je částečné prosté zobrazení

Podmínka acyklicity sítě:

Požadujeme aby orientovaný graf $G = (K, E)$ kde $(K_u, K_v) \in E$ pokud existují i a j takové, že $f(u, i) = (v, j)$, byl **acyklický**.

Rozdělení komparátorů do hladin:

- Definujme $L_1 = \{ K_i \mid K_i \text{ má v } G \text{ vstupní stupeň nula} \}$ (L_1 je neprázdná díky acyklicitě)
- Nechť jsou definovány L_1, L_2, \dots, L_h , kde $L = L_1 \cup L_2 \cup \dots \cup L_h \subsetneq K$. Pak definujme $L_{h+1} = \{ K_i \mid K_i \text{ má v } G \setminus L \text{ vstupní stupeň nula} \}$ (L_{h+1} je neprázdná díky acyklicitě)
- Počet hladin značíme d a nazýváme **hloubkou** sítě

Práce sítě:

- čas 0** : definovány vstupy sítě (kam patří vstupy všech komparátorů v L_1)
pracují komparátory v L_1
- čas 1** : definovány vstupy všech komparátorů v L_2
pracují komparátory v L_2
...
- čas d-1** : definovány vstupy všech komparátorů v L_d
pracují komparátory v L_d
- čas d** : definovány všechny výstupy sítě

Pozorování: časová složitost třídění odpovídá hloubce sítě (to je tedy klíčový parametr)

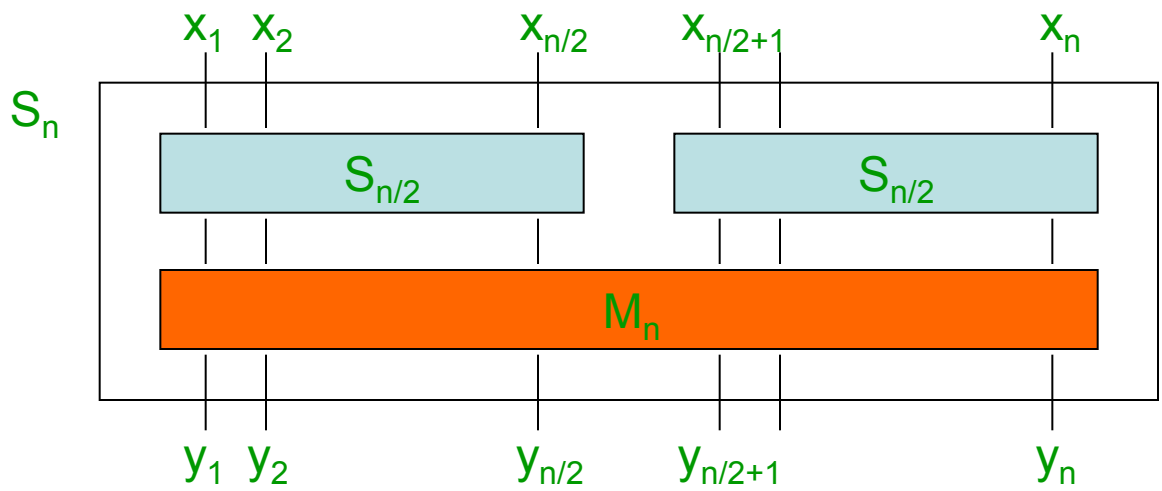
Topologicky jiná reprezentace sítě:

- „dráty“ ze vstupu x_i do výstupu y_i nakresleny jako přímky
- jednotlivé komparátory „roztaženy“ mezi příslušné „dráty“
- každá síť jde takto překreslit
- počtu vstupů/výstupů (drátů) říkáme **šířka** sítě

Merge-Sort implementovaný třídící sítí

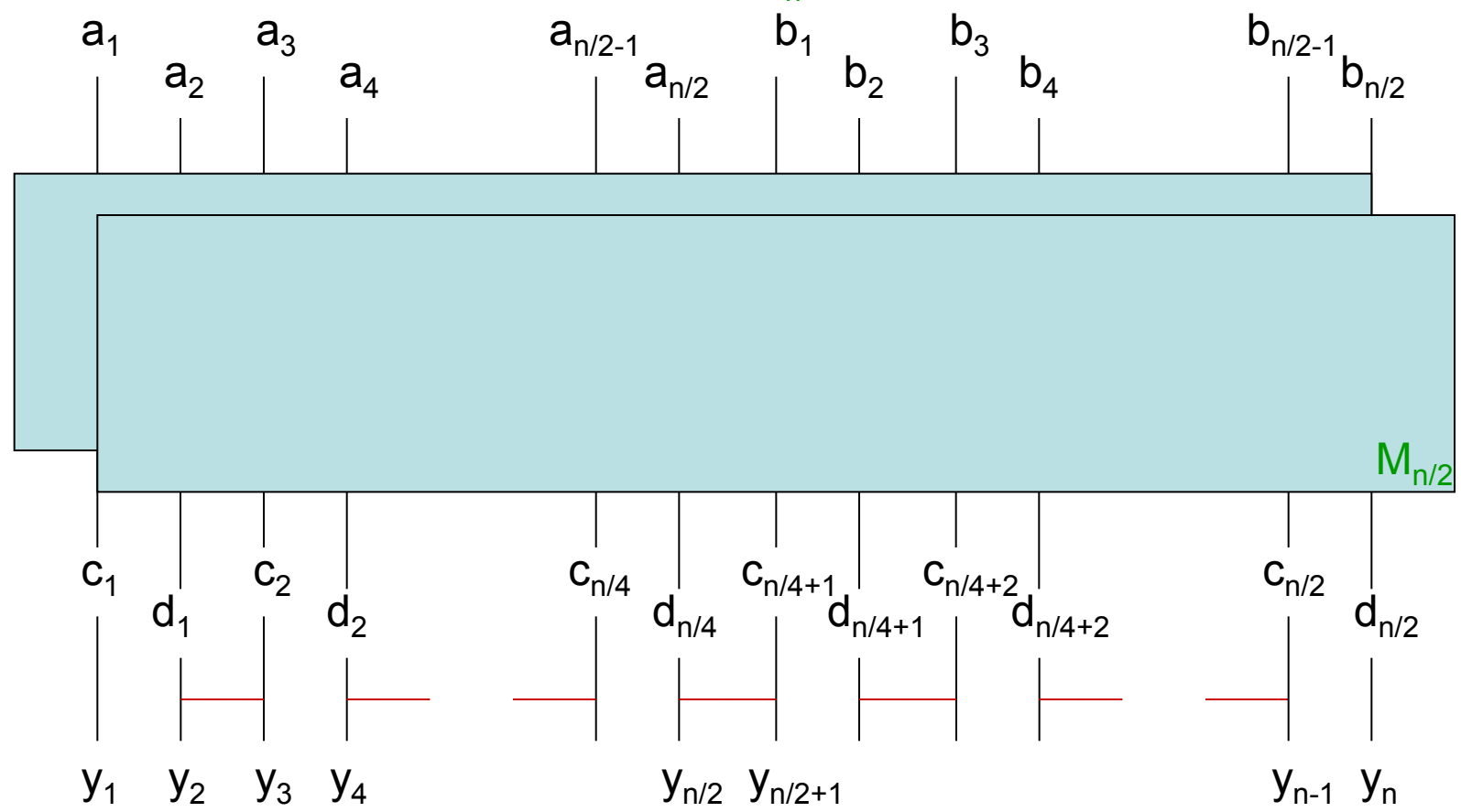
Chceme setřídít x_1, x_2, \dots, x_n (předpokládáme že n je mocnina dvojky)

Realizujeme to sítí S_n , která je rekurzivně definována následujícím obrázkem



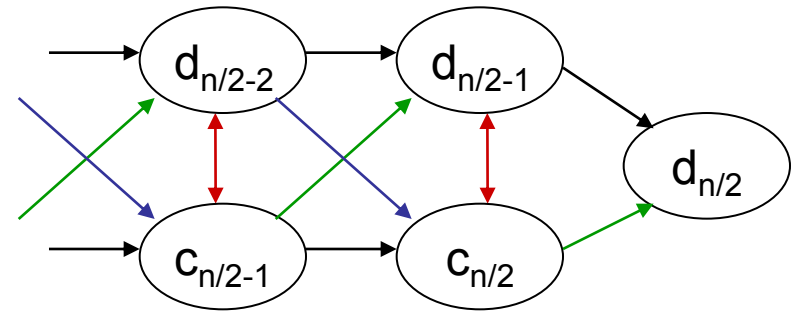
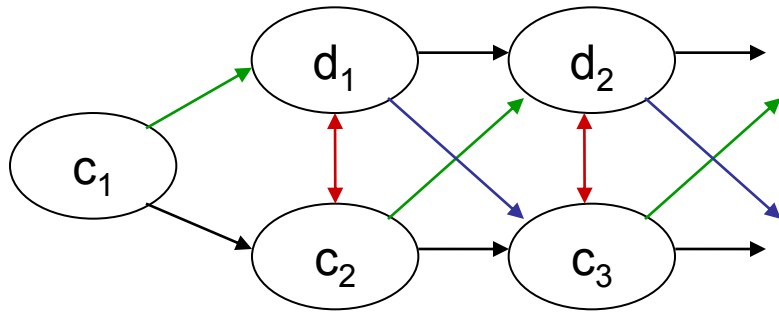
kde M_n je **slučovací (slévací)** síť šířky n (rekurze se zastaví pro $n=2$)

Zbývá ukázat jak zkonstruovat slučovací síť M_n (opět jde o rekursivní konstrukci) :



Liché členy obou seříděných posloupností jsou vstupem jedné kopie $M_{n/2}$ a sudé členy jsou vstupem druhé kopie $M_{n/2}$. Navíc jsou výstupy obou sítí propojeny jednou hladinou komparátorů dle obrázku (červené komparátory). Rekurze se opět zastaví pro $n=2$.

Pro vstup platí: $a_1 \leq a_2 \leq \dots \leq a_{n/2}$ a $b_1 \leq b_2 \leq \dots \leq b_{n/2}$
 Indukční předpoklad: $c_1 \leq c_2 \leq \dots \leq c_{n/2}$ a $d_1 \leq d_2 \leq \dots \leq d_{n/2}$
 Dokážeme, že: $y_1 \leq y_2 \leq \dots \leq y_n$



Černé nerovnosti (šipky) víme, **zelené nerovnosti** a **modré nerovnosti** (šipky) dokážeme. Bez ohledu to, jak dopadne porovnání jednotlivými **červenými komparátory**, budou šipky generovat lineární uspořádání, které bude správným uspořádáním výstupních hodnot.

Hloubka a velikost třídící sítě šířky $n = 2^k$

1. Slučovací síť M_n

má hloubku (počet hladin)
a velikost (počet komparátorů)

$$d(M_n) = \log_2 n$$

$$s(M_n) = n/2 \log_2(n/2) + 1$$

2. Třídící síť S_n

má hloubku (počet hladin)
a velikost (počet komparátorů)

$$d(S_n) = 1/2 \log_2 n (\log_2 n + 1)$$

$$s(S_n) = 1/4 n \log_2 n (\log_2 n - 1) + (n - 1)$$

Dolní odhad složitosti třídění pomocí transpozičních sítí

Transpoziční síť = síť složená pouze z komparátorů (zcela libovolně umístěných)

⇒ třídící síť je tedy speciálním případem transpoziční sítě

Lemma 1 Každá transpoziční síť dává pro vstup x_1, x_2, \dots, x_n na výstupu nějakou permutaci Π vstupních hodnot, tedy dává na výstupu $x_{\Pi(1)}, x_{\Pi(2)}, \dots, x_{\Pi(n)}$.

Důkaz Zcela zřejmý, síť nemůže udělat nic jiného než nějak zpřeházet vstupy.

Definice Necht' C je transpoziční síť šířky n . Permutace $\Pi : \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ je **dosažitelná** pro C , pokud existuje vstupní posloupnost x_1, x_2, \dots, x_n taková, že C vydá na výstupu posloupnost $x_{\Pi(1)}, x_{\Pi(2)}, \dots, x_{\Pi(n)}$.

Lemma 2 Necht' C je třídící síť šířky n . Pak je pro C dosažitelných všech $n!$ permutací.

Lemma 3 Necht' C je transpoziční síť šířky n a necht' p je počet dosažitelných permutací pro C . Potom platí

$$p \leq 2^{s(C)}.$$

Důsledek Pokud je C třídící síť tak

$$n! \leq 2^{s(C)}$$

a tím pádem má C velikost $s(C) = \Omega(n \log n)$ a hloubku $d(C) = \Omega(\log n)$.

Aritmetické sítě

- definice je podobná jako u třídících sítí, zcela stejně je definována **šířka** (počet vstupů, tj. vstupních drátů), **velikost** (počet hradel) a **hloubka** (počet hladin) sítě
- **časová složitost** výpočtu opět odpovídá hloubce sítě
- dráty zde přenášejí jen hodnoty **0** a **1** (tj. jeden bit informace, ne více bitů jako v případě třídících sítí)
- hradla jsou **logická hradla**, typicky unární hradlo NOT a binární hradla AND, OR a XOR
- drát z jednoho výstupu může jít do více vstupů („rozvětvit se“), musí být ovšem stále zachována **acyklicita sítě**

Sčítač (full adder)

Sčítač je obvod mající tři vstupy **x,y,z** a dva výstupy **c,s**, jehož funkci lze chápat tak, že **x** a **y** jsou bity dvou binárních čísel ve stejném řádu, **z** je přenos z nižšího řádu, **s** příslušný bit součtu **x+y** a **c** je přenos do vyššího řádu

z tabulky pro výpočet **s** a **c** plyne, že **s = parita(x,y,z)** a **c = majorita(x,y,z)**

Sčítání dvou n-bitových binárních čísel

- **n** bude mocnina dvojky, sčítat budeme čísla **a = (a_{n-1}, ..., a₀)** a **b = (b_{n-1}, ..., b₀)**
- ukážeme dva obvody: pro klasické sčítání a pro carry-lookahead sčítání

Obvod pro klasické sčítání

- sčítání je realizováno kaskádou n sčítačů, kde sčítač v řádu i čeká na přenos (carry bit) od sčítače řádu $i - 1$ a posílá svůj carry bit do sčítače řádu $i + 1$
- síť má velikost $\Theta(n)$ a hloubku také $\Theta(n)$

Obvod pro carry-lookahead sčítání

zrychlení je založeno na myšlence, že v některých případech lze přenos z i -tého řádu určit jen na základě a_i a b_i (které jsou známy od začátku) a není potřeba čekat na přenos z nižšího řádu (jehož spočítání může trvat dlouho)

- v případě, že $a_i = b_i = 0$ tak $c_{i+1} = 0$ (bez ohledu na hodnotu c_i) \rightarrow **kill** carry bit c_{i+1}
- v případě, že $a_i = b_i = 1$ tak $c_{i+1} = 1$ (bez ohledu na hodnotu c_i) \rightarrow **generate** carry bit c_{i+1}
- v případě $a_i \neq b_i$ platí, že $c_{i+1} = c_i \rightarrow$ **propagate** carry bit c_i do carry bitu c_{i+1}

tím je definován **carry status** každého řádu $i \in \{1, \dots, n\}$, který označíme proměnnou x_i , jejíž hodnotu z množiny $\{k, p, g\}$ lze snadno spočítat z a_{i-1} a b_{i-1} podle uvedených pravidel

uvažujme dva za sebou napojené sčítače jako jeden obvod s jedním vstupním a jedním výstupním carry bitem: **carry status** kombinovaného obvodu lze spočítat z (k, p, g) obou zúčastněných sčítačů

tím je definován binární **carry status operátor** \otimes na množině $\{k, p, g\}$, který je **asociativní** (bez důkazu, stačí rutinní rozbor všech 27 případů)

pokud dodefinujeme $x_0 = y_0 = k$, pak pro každý řád $i \in \{1, \dots, n\}$ lze definovat proměnnou

$$y_i = y_{i-1} \otimes x_i = x_0 \otimes x_1 \otimes \dots \otimes x_i$$

kterou lze chápat jako i -tý prefix součinu $x_0 \otimes x_1 \otimes \dots \otimes x_n$

Lemma Pro $i = 0, 1, \dots, n$ platí

1. pokud $y_i = k$ tak $c_i = 0$
2. pokud $y_i = g$ tak $c_i = 1$
3. případ $y_i = p$ nenastane

Důsledek Pokud jsou známa všechna y_i tak už lze součet spočítat v $O(1)$ pomocí n paralelně běžících sčítačů (pokud ztotožníme $k = 0$ a $g = 1$, což lze protože p nemůže nastat), takže problém sčítání se tímto převádí na problém výpočtu všech prefixů.

Sčítací obvod sestává z n **KPG obvodů** a **prefixového obvodu**. Každý KPG obvod je využit dvakrát. Při prvním průchodu má i -tý KPG obvod na vstupu a_i a b_i a na výstupu x_i , které pošle do prefixového obvodu. Při druhém průchodu funguje i -tý KPG obvod jako sčítač (respektive jako jeho část počítající paritu), na vstupu má a_i , b_i a y_i , které přijde z prefixového obvodu (s tím že hodnoty k a g jsou interpretovány jako 0 a 1) a na výstupu má $s_i = \text{parita}(a_i, b_i, y_i)$.

Čili zbývá dodělat paralelní prefixový obvod ...

Paralelní prefixový obvod

jako vstupy dostane carry statusy jednotlivých řádů, tj. x_0, x_1, \dots, x_n a spočítá z nich hodnoty všech prefixů y_0, y_1, \dots, y_n

označme $[i, j] = x_i \otimes x_{i+1} \otimes \dots \otimes x_j$ (a tedy $[i, i] = x_i$)

díky asociativitě platí $[i, k] = [i, j-1] \otimes [j, k]$ pro libovolné j takové, že $i < j \leq k$

chceme spočítat $y_i = [0, i]$ pro všechna $i \in \{0, 1, \dots, n\}$

paralelní prefixový obvod sestává pouze z obvodů realizujících carry status operátor \otimes

obvod má topologii úplného binárního stromu (zde potřebujeme aby n byla mocnina 2):

- v listech stromu jsou vstupy x_1 až x_n a v kořeni je vstup x_0
- při průchodu od listů ke kořeni se počítá pro každý vnitřní uzel (reprezentující interval od i do k), jehož levý syn reprezentuje interval od i do $j-1$ (a tedy má hodnotu $[i, j-1]$) a pravý syn interval od j do k (a tedy má hodnotu $[j, k]$) hodnota $[i, k] = [i, j-1] \otimes [j, k]$
- při průchodu od kořene k listům přijde do každého vnitřního uzlu (reprezentujícího interval od i do k) shora hodnota $[0, i-1]$, kterou uzel pošle beze změny levému synovi (reprezentujícímu interval od i do $j-1$) a pravému synovi (reprezentujícímu interval od j do k) spočítá hodnotu $[0, j-1] = [0, i-1] \otimes [i, j-1]$
- výstupy y_0 až y_{n-1} jsou spočítány v listech a výstup y_n v kořeni
- velikost obvodu je $\Theta(n)$ a hloubka $\Theta(\log n)$

Třídy P a NP, převoditelnost problémů, NP úplnost

Úloha: pro dané zadání najít strukturu s danými vlastnostmi

Příklady:

- v daném orientovaném grafu najdi cyklus
- vynásob dvě dané čtvercové matice

Optimalizační úloha: pro dané zadání najít optimální (většinou nejmenší nebo největší) strukturu s danými vlastnostmi

Příklady:

- v daném neorientovaném grafu najdi největší (počtem vrcholů) úplný podgraf (kliku)
- pro danou množinu úkolů najdi nejkratší rozvrh

Rozhodovací problém: pro dané zadání odpovědět **ANO/NE**

Příklady:

- existuje v daném neorientovaném grafu Hamiltonovská kružnice?
- je daná čtvercová matice regulární?

My se v následujícím omezíme jen na rozhodovací problémy, což lze (více méně) udělat bez újmy na obecnosti - v tom smyslu, že k většině (optimalizačních) úloh existuje „stejně těžký“ rozhodovací problém.

Definice (vágní): Třída **P** je třída rozhodovacích problémů, pro které existuje (deterministický sekvenční) algoritmus běžící v **polynomiálním** čase (vzhledem k velikosti zadání), který správně rozhodne ANO/NE (který řeší daný problém).

- je daný orientovaný graf silně souvislý?
- obsahuje daný neorientovaný graf trojúhelník? (speciální případ „kliky“)
- je daná matice regulární?

Nedeterministický algoritmus = algoritmus, který v každém svém kroku může volit z několika možností

Nedeterministický algoritmus **řeší** daný rozhodovací problém \Leftrightarrow pro každé kladné zadání problému (odpověď ANO) existuje posloupnost voleb vedoucí k tomu, že algoritmus odpoví ANO, pro žádné záporné zadání taková posloupnost voleb neexistuje.

Definice (vágní): Třída **NP** je třída rozhodovacích problémů, pro které existuje **nedeterministický sekvenční** algoritmus běžící v **polynomiálním** čase (vzhledem k velikosti zadání), který řeší daný problém.

Jiný model nedeterministického algoritmu: dopředu provede volby (do paměti zapíše vektor čísel) a pak už provádí jednotlivé kroky původního algoritmu deterministicky.

Alternativní definice (opět vágní): Rozhodovací problém patří do třídy **NP**, pokud pro každé jeho kladné zadání existuje (polynomiálně velký) **certifikát**, pomocí něhož lze v polynomiálním čase (deterministicky) ověřit, že zadání je skutečně kladné (že odpověď na dané zadání je skutečně ANO).

Příklady problémů ze třídy NP:

- **KLIKA** (úplný podgraf): Je dán neorientovaný graf G a číslo k .

Otázka: Existuje v G úplný podgraf velikosti alespoň k ?

- **HK** (Hamiltonovská kružnice): Je dán neorientovaný graf G .

Otázka: Existuje v G Hamiltonovská kružnice?

- **TSP** (obchodní cestující): Je dán ohodnocený úplný neorientovaný graf G a číslo k .

Otázka: Existuje v G Hamiltonovská kružnice celkové délky nejvýše k ?

- **SP** (součet podmnožiny): Jsou dána přirozená čísla a_1, a_2, \dots, a_n, b .

Otázka: Existuje podmnožina čísel a_1, a_2, \dots, a_n , jejíž součet je přesně b ?

- **ROZ** (rozvr. na paralel. strojích): Je dán počet úkolů, jejich délky, počet strojů a číslo k .

Otázka: Existuje přípustný rozvrh délky nejvýše k ?

- **SAT** (splnitelnost Booleovských formulí): Je dána formule na n 0-1 proměnných v KNF.

Otázka: Existuje (pravdivostní) ohodnocení proměnných pro které má daná formule hodnotu 1?

Ukážeme, že $HK \rightarrow TSP$, $SP \rightarrow ROZ$ a $SAT \rightarrow KLIKA$, kde $A \rightarrow B$ znamená, že pokud existuje polynomiální algoritmus řešící B potom také existuje polynomiální algoritmus řešící A , neboli vyřešit B je alespoň tak „těžké“ jako vyřešit A .

Převody (redukce) mezi rozhodovacími problémy

Nechť A, B jsou dva rozhodovací problémy. Říkáme, že A je **polynomiálně redukovatelný** na B , pokud existuje zobrazení f z množiny zadání problému A do množiny zadání problému B s následujícími vlastnostmi:

1. Nechť X je zadání problému A a Y zadání problému B takové, že $Y = f(X)$. Potom je X kladné zadání problému A tehdy a jen tehdy, když je Y kladné zadání problému B .
2. Nechť X je zadání problému A . Potom je zadání $f(X)$ problému B (deterministicky sekvenčně) zkonstruovatelné v polynomiálním čase vzhledem k velikosti X .

Poznámka: Z 2. také vyplývá, že velikost $f(X)$ je polynomiální vzhledem k velikosti X .

NP-úplnost

Definice: Problém B je **NP-těžký** pokud pro libovolný problém A ze třídy **NP** platí, že A je polynomiálně redukovatelný na B .

Definice: Problém B je **NP-úplný** pokud 1) patří do třídy **NP** a 2) je **NP-těžký**.

Důsledek 1: Pokud je A **NP-těžký** a navíc je polynomiálně redukovatelný na B , tak je B také **NP-těžký**.

Důsledek 2: Pokud existuje polynomiální algoritmus pro nějaký **NP-těžký** problém, pak existují polynomiální algoritmy pro všechny problémy ve třídě **NP**.

Věta (Cook-Levin 1971): **SAT** je **NP-úplný**.

Aproximační algoritmy

Aprox. algoritmy jsou vhodné tam, kde je nalezení optimálního řešení „beznadějné“ (časově příliš náročné), typicky u NP-těžkých optimalizačních úloh (optimalizačních verzí NP-úplných rozhodovacích problémů). Mají následující tři vlastnosti:

1. konstruují suboptimální řešení
2. poskytují odhad kvality zkonstruovaného řešení vzhledem k optimu
3. běží v polynomiálním čase (jinak nejsou zajímavé)

Příklad maximalizační úlohy (optimalizační verze **KLIKY**):

Pro daný neorientovaný graf najdi **největší** (počtem vrcholů) kliku (úplný podgraf).

Po aproximačním algoritmu chceme garanci typu $f(\text{APROX}) \geq \frac{3}{4} f(\text{OPT})$, kde $f(X)$ je v tomto případě počet vrcholů (tj. velikost kliky) v řešení X , OPT je optimální řešení a APROX je řešení vydané aproximačním algoritmem.

Příklad minimalizační úlohy (optimalizační verze **ROZ**):

Pro dané úkoly a daný počet strojů najdi **nejkratší** rozvrh.

Po aproximačním algoritmu chceme garanci typu $f(\text{APROX}) \leq 2 f(\text{OPT})$.

Definice: **Poměrová chyba** aproximačního algoritmu je definována jako poměr (podíl) $f(\text{APROX}) / f(\text{OPT})$ pro minimalizační úlohy a $f(\text{OPT}) / f(\text{APROX})$ pro maximalizační úlohy. **Relativní chyba** je pak definována jako $|f(\text{APROX}) - f(\text{OPT})| / f(\text{OPT})$.

Naivní aproximační algoritmus FRONTA pro optimalizační verzi ROZ: bere úkoly postupně podle jejich čísel a každý úkol vždy umístí na stroj, který je volný nejdříve.

Značení: OPT = optimální rozvrh, Q = rozvrh zkonstruovaný algoritmem FRONTA,
délka(OPT) = o, délka(Q) = q

Věta: Pokud m je počet strojů, tak $q \leq ((2m - 1) / m)o$ a tento odhad již nelze zlepšit.

Důsledek: Aproximační algoritmus FRONTA má poměrovou chybu 2.

Důkaz:

1. Těsnost odhadu: Pro každé m zkonstruujeme zadání, pro které platí v dokazované nerovnosti rovnost, a to následujícím způsobem

$$x_1 = x_2 = \dots = x_{m-1} = m-1 \quad (m-1 \text{ úkolů délky } m-1)$$

$$x_m = x_{m+1} = \dots = x_{2m-2} = 1 \quad (m-1 \text{ úkolů délky } 1)$$

$$x_{2m-1} = m \quad (1 \text{ úkol délky } m)$$

2. Platnost nerovnosti: Nechť j je úkol končící jako poslední v rozvrhu Q (končící v čase q) a nechť t je okamžik zahájení úkolu j. Potom žádný procesor nemá prostoj před časem t a platí $mq \leq (2m - 1)o$.

Lepší aproximační algoritmus **USPOŘÁDANÁ FRONTA** pro optimalizační verzi **ROZ**: pracuje stejně jako FRONTA, ale na začátku úkoly setřídí do nerostoucí posloupnosti podle jejich délek.

Značení: **OPT** = optimální rozvrh,
U = rozvrh zkonstruovaný algoritmem **USPOŘÁDANÁ FRONTA**,
 $\text{délka}(\text{OPT}) = o$, $\text{délka}(\text{U}) = u$

Věta: Pokud m je počet strojů, tak $u \leq ((4m - 1) / 3m)o$ a tento odhad již nelze zlepšit.

Důsledek: Aproximační algoritmus **USPOŘÁDANÁ FRONTA** má poměrovou chybu $4/3$.

Důkaz: Těsnost odhadu: Pro každé liché m zkonstruujeme zadání, pro které platí v dokazované nerovnosti rovnost, a to následujícím způsobem

$$x_1 = x_2 = 2m-1 \quad (2 \text{ úkoly délky } 2m-1)$$

$$x_3 = x_4 = 2m-2 \quad (2 \text{ úkoly délky } 2m-2)$$

$$x_{2m-3} = x_{2m-2} = m+1 \quad (2 \text{ úkoly délky } m+1)$$

$$x_{2m-1} = x_{2m} = x_{2m+1} = m \quad (3 \text{ úkoly délky } m)$$

Lemma: Pokud pro všechny úkoly platí $x_i \geq 1/3o$ pak $u = o$.

Dokončení důkazu: Necht' j je úkol končící jako poslední v rozvrhu **U** (končící v čase u). Pokud $x_j > 1/3o$ tak použijeme Lemma, v opačném případě je důkaz velmi podobný jako pro algoritmus **FRONTA**.

Pravděpodobnostní (randomizované) algoritmy

pravděpodobnostní algoritmus dělá (na rozdíl od **deterministického** algoritmu) **náhodné** kroky, např. k některým krokům používá hodnoty získané z generátoru náhodných čísel tím pádem dvě různá spuštění téhož pravděpodobnostního algoritmu na stejných datech mají (s velkou pravděpodobností) různý průběh

pravděpodobnostních algoritmů je mnoho typů, zde zmíníme jen dva a to algoritmy typu Las Vegas a typu Monte Carlo

Algoritmy typu Las Vegas

výsledek je vždy správný, náhodnost ovlivňuje pouze dobu běhu algoritmu, tj. po jaké cestě se algoritmus ke správnému výsledku dobere

Příklad: randomizovaný QuickSort – od deterministické verze se liší náhodnými výběry pivotu při každém dělení posloupnosti, což poskytuje následující výhody

- dává dobrý průměrný čas (tj. $O(n \log n)$) i v případě, že data na vstupu nejsou náhodné permutace – žádný vstup není apriori špatný (pro každý deterministický výběr pivotu existují apriori špatné vstupy)
- může být spuštěn paralelně v několika kopiích, výsledek je získán z kopie, kde výpočet skončí nejdříve (pro deterministickou verzi nemá takový postup žádný smysl)

Algoritmy typu Monte Carlo

náhodnost ovlivňuje jak dobu běhu, tak správnost výsledku: algoritmus může udělat chybu, ale pouze jednostranně (u odpovědí ANO/NE) a s omezenou pravděpodobností

Příklad: Rabin-Millerův algoritmus na testování prvočíselnosti

Úloha: pro zadané přirozené číslo n (rychle) rozhodnout zda je n prvočíslo

Trocha teorie (Malá) Fermatova věta (bez důkazu):

Nechť p je prvočíslo. Potom $\forall k \in \{1, 2, \dots, p-1\}$ platí $k^{p-1} \equiv 1 \pmod{p}$

Myšlenka: pokud n není prvočíslo, tak zkusíme (náhodně) najít „svědka“ k , porušujícího $k^{n-1} \equiv 1 \pmod{n}$, který „dovádí“, že n je opravdu číslo složené (není to prvočíslo)

Problém: pro některá složená čísla je svědků příliš málo, takže je „příliš malá pravděpodobnost, že nějakého svědka (náhodně) vybereme.“

Definice: Nechť T je množina dvojic přirozených čísel, kde $(k, n) \in T$ pokud $0 < k < n$ a je splněna alespoň jedna z následujících dvou podmínek:

1. neplatí $k^{n-1} \equiv 1 \pmod{n}$,
2. existuje i takové, že $m = (n-1) / 2^i$ je přirozené číslo a platí $1 < \text{NSD}(k^{m-1}-1, n) < n$

Věta 1: Číslo n je složené tehdy a jen tehdy, když existuje k takové, že $(k, n) \in T$.

Věta 2: Nechť n je složené číslo. Pak existuje alespoň $(n-1)/2$ takových čísel k , pro které platí $(k, n) \in T$.

```

Rabin-Miller(n);
for i:=1 to počet do
   $k_i := \text{náhodné přirozené číslo z intervalu } [1, n-1];$ 
  if  $(k_i, n) \in T$  then Report (n je složené);
  Abort;
Report (n je prvočíslo)

```

Pokud Rabin-Miller(n) rozhodne, že n je složené, tak je to zaručeně správný výsledek (byl nalezen „svědek“), pokud Rabin-Miller(n) rozhodne, že n je prvočíslo, tak se může jednat o chybu, ale pouze v případě, že všechna vybraná k_i byli „ne-svědci“ pro složené číslo n, což ale může (díky Větě 2) nastat nejvýše s pravděpodobností

$$P(\text{chyba}) \leq (1/2)^{\text{počet}}$$

pokud jsou výběry jednotlivých k_i vzájemně nezávislé

Vlastnosti algoritmu:

- zvyšováním počtu iterací (počtu testovaných k_i) lze dostat libovolně malou (předem zvolenou) pravděpodobnost chyby
- jednotlivé iterace (testy pro různá k_i) lze provádět paralelně

Časová složitost:

každá iterace trvá jen polynomiálně vzhledem k délce zápisu čísla n (tj. k délce vstupu), k tomu je ovšem potřeba ukázat, že test zda $(k_i, n) \in T$ je možno provést v čase polynomiálním v $\log n$, což není triviální (je nutné mít další znalosti z teorie čísel)

Kryptografie s veřejným klíčem (asymetrickou šifrou)

- každý účastník X má svůj veřejný klíč PX a soukromý klíč SX
- SX je znám pouze X , veřejný klíč PX může X sdělit všem s kterými komunikuje, nebo může být dokonce zveřejněn ve veřejně dostupném seznamu klíčů (třeba na webu)
- oba klíče specifikují funkce, které lze aplikovat na jakoukoli zprávu: tedy pokud D je množina všech konečných posloupností bitů (množina všech možných zpráv), tak obě funkce musí být prosté funkce zobrazující D na D (tj. jsou to permutace množiny D)
- funkci specifikovanou soukromým klíčem SX značíme $SX()$ a funkci specifikovanou veřejným klíčem PX značíme $PX()$, přičemž předpokládáme, že každá z těchto funkcí je efektivně vyčíslitelná pokud známe příslušný klíč
- funkce $SX()$ a $PX()$ musí tvořit vzájemně inverzní pár funkcí – pro každou zprávu (konečnou posloupnost bitů) M tedy musí platit $SX(PX(M)) = M$ a $PX(SX(M)) = M$.
- bezpečnost šifry stojí a padá s tím, že nikdo kromě účastníka X není schopen v „rozumném“ čase spočítat $SX(M)$ pro jakoukoli zprávu M , což znamená, že
 1. účastník X musí držet klíč SX v absolutním bezpečí před vyzrazením
 2. funkce $SX()$ nesmí být efektivně vyčíslitelná na základě znalosti PX (a schopnosti efektivně vyčíslit funkci $PX()$), což je hlavní obtíž při návrhu systému šifrování s veřejným klíčem

Předpokládejme, že máme 2 účastníce: **A** (Alici) a **B** (Barboru) s klíči **SA**, **PA**, **SB** a **PB**

Posílání zašifrované zprávy a její rozšifrování

Barbora chce poslat Alici zašifrovanou zprávu **M**:

- Barbora si opatří Alicin veřejný klíč **PA** (přímo od Alice či z veřejného seznamu klíčů)
- Barbora spočítá **zašifrovaný text** $C = PA(M)$ a pošle ho Alici
- Alice na **C** aplikuje svůj soukromý klíč **SA**, tedy spočítá $SA(C) = SA(PA(M)) = M$
- Pokud **C** zachytí někdo jiný než Alice, nemá šanci získat **M**, protože neumí efektivně spočítat **SA(C)**.

Posílání autentizované a podepsané (nešifrované) zprávy

Alice chce odpovědět Barboře tak, aby Barbora měla jistotu, že odpověď **Q** přichází od Alice a že text odpovědi nebyl pozměněn:

- Alice spočítá svůj **digitální podpis** **q** pro zprávu **Q** pomocí svého soukromého klíče, tj. spočítá $q = SA(Q)$ a pošle Barboře dvojici **(Q,q)** – tj. zpráva **Q** odchází nešifrovaně
- Barbora spočítá $PA(q) = PA(SA(Q)) = Q$ a porovná to s došlou zprávou **Q**
- Pokud se obě zprávy zcela shodují, má Barbora jistotu, že zpráva přichází od Alice a nebyla cestou pozměněna
- Pokud se zprávy liší, tak buď je podpis **q** falešný (nebyl vytvořen funkcí **SA()**) nebo je podpis pravý ale nezašifrovaná zpráva **Q** byla cestou pozměněna

Posílání autentizované a podepsané zašifrované zprávy

Alice chce poslat Barboře zprávu M tak, aby Barbora měla jistotu, že M přichází od Alice a že text M nebyl pozměněn. Navíc Alice chce, aby si M mohla přičíst pouze Barbora a nikdo jiný.

- Alice spočítá svůj digitální podpis pro M , tedy spočítá $m = SA(M)$
- Alice zašifruje dvojici (M, m) pomocí Barbořina veřejného klíče, tedy spočítá zašifrovaný text $C = PB(M, m)$ a pošle C Barboře
- Barbora rozšifruje C pomocí svého soukromého klíče, tedy spočítá $SB(C) = (M, m)$
- Barbora ověří platnost Alicina podpisu a autenticitu M pomocí Alicina veřejného klíče, tj. spočítá $PA(m)$ a porovná to s M – při neshodě Barbora ví, že buď bylo C cestou změněno (úmyslně či přenosovou chybou) nebo C nepřichází od Alice.

Hybridní šifrování

Pokud je zpráva M , kterou chce Barbora poslat Alici, velmi dlouhá a výpočet $C = PA(M)$ a následně $M = SA(C)$ by trval příliš dlouho, je možné použít šifrování s veřejným klíčem v kombinaci s nějakou symetrickou šifrou K , která šifruje zprávy rychle:

- Barbora spočítá $C = K(M)$, což je opět dlouhá posloupnost bitů, k tomu spočítá $PA(K)$, což je krátká posloupnost bitů (ve srovnání s M a $PA(M)$) a pošle $(C, PA(K))$ Alici
- Alice rozšifruje $PA(K)$ pomocí svého SA , takže dostane K , pomocí kterého rozšifruje C a tak získá M

Hybridní autentizace a podepisování

Pro dlouhou zprávu M je také časově náročné počítat digitální podpis $m = SA(M)$. Zde si vypomůžeme (veřejně známou) **hashovací funkcí** h , která má následující dvě vlastnosti:

1. I pro dlouhé M lze $h(M)$ spočítat velmi rychle, typicky je $h(M)$ krátký (např. 128 bitový) **otisk** (fingerprint) zprávy M .
2. Je výpočetně velmi obtížné (v rozumném čase nemožné) najít k M jinou zprávu Q takovou, aby platilo $h(M) = h(Q)$

Pokud chce Alice podepsat dlouhou zprávu posílanou Barboře, může postupovat takto:

- Alice spočítá otisk $h(M)$ zprávy M , udělá z něj digitální podpis $m = SA(h(M))$ a pošle Barboře dvojici (M, m)
- Barbora obdrží M a také spočítá otisk $h(M)$ který poté porovná s rozšifrovaným Aliciným digitálním podpisem $PA(m) = PA(SA(h(M)))$. Pokud byla M cestou změněna, tak dojde k neshodě, protože díky vlastnosti 2 je těžké pozměnit M tak, aby se její otisk nezměnil.

Certifikační authority

Pokud si Alice pořizuje Barbořin veřejný klíč z veřejně dostupného seznamu (nebo jí ho Barbora posílá po síti), jak může mít jistotu, že nejde o podvrh? Pokud by byl klíč podvržen a následné zprávy modifikovány nebo podvrhovány stejným člověkem, který podvrhl svůj klíč jako Barbořin, tak jejich nepravost nelze zjistit (protože daný člověk bude mít k podvrženému veřejnému klíči i odpovídající soukromý klíč). Řešení:

- Existuje **certifikační autorita Z**, jejíž veřejný klíč **PZ** má každý účastník (tedy i Alice) nainstalován u sebe (například přišel na CD s šifrovacím softwarem).
- Barbora pak má od autority **Z** vydán certifikát ve tvaru **C** = „Barbořin klíč je **PB**“ podepsaný autoritou **Z**, tedy dvojici **(C, SZ(C))** – toto může mít Barbora také již od koupě šifrovacího softwaru, nebo certifikát získá jinou bezpečnou cestou
- Tuto dvojici **(C, SZ(C))** připojí Barbora ke každé podepisované zprávě, takže Alice (i kdokoli jiný) zjistí pomocí veřejného klíče **PZ**, že **C** bylo opravdu vydáno autoritou **Z**, a že tedy **PB** opravdu je Barbořin veřejný klíč

RSA (Rivest, Shamir, Adelman) šifra

pro vysvětlení RSA potřebujeme řadu pojmů a tvrzení z teorie čísel

Věta: Necht' **a, b** jsou přirozená čísla. Pak **NSD(a, b)** je nejmenší kladný prvek množiny

$$L = \{ax + by \mid x, y \in \mathbb{Z}\}$$

Důsledek: Necht' **a, b** jsou přirozená čísla. Pokud **d** je přirozené číslo, které dělí **a** i **b**, tak **d** dělí také **NSD(a, b)**.

Věta: Necht' **a, b** jsou přirozená čísla, kde **b > 0**. Pak **NSD(a, b) = NSD(b, a mod b)**.

EUCLID(a, b)

if **b=0** then **Return(a)**

else **Return(EUCLID(b, a mod b))**

Lemma: Necht' $a > b \geq 0$ a $\text{EUCLID}(a,b)$ udělá $k \geq 1$ rekurzivních kroků. Pak $a \geq F(k+2)$ a $b \geq F(k+1)$, kde $F(i)$ je i -té Fibonacciho číslo.

Důsledek (Lamého věta): Necht' $a > b \geq 0$ a $F(k) \leq b < F(k+1)$. Pak $\text{EUCLID}(a,b)$ udělá nejvýše $k - 1$ rekurzivních kroků.

Věta (bez Dk): $F(k) = \Theta(\varphi^k)$, kde $\varphi = (1+\sqrt{5})/2$ (což je tzv. „zlatý řez“).

Důsledek: Necht' $a > b \geq 0$ a $F(k) \leq b < F(k+1)$. Pak $\text{EUCLID}(a,b)$ udělá nejvýše $O(\log b)$ rekurzivních kroků.

Pozorování: Pokud a, b jsou dvě nejvýše t -bitová binární čísla, tak $\text{EUCLID}(a,b)$ provede $O(t)$ rekurzivních kroků a v každém z nich $O(1)$ aritmetických operací na (nejvýše) t -bitových číslech, tj. $O(t^3)$ bitových operací, pokud předpokládáme, že každá aritmetická operace na t -bitových číslech potřebuje $O(t^2)$ bitových operací (což je snadné ukázat). EUCLID je tedy polynomiální algoritmus vzhledem k velikosti vstupu.

Euklidův algoritmus lze snadno rozšířit tak, aby počítal také koeficienty x, y , pro které $\text{NSD}(a,b) = ax + by$.

$\text{EXTENDED-EUCLID}(a,b)$

if $b=0$ then Return($a, 1, 0$)

else $(d', x', y') := \text{EXTENDED-EUCLID}(b, a \bmod b);$

$(d, x, y) := (d', y', x' - \lfloor a/b \rfloor y');$

Return(d, x, y)

Věta (bez Dk): Necht' $n > 1$ a $a < n$ jsou dvě nesoudělná přirozená čísla. Pak má rovnice $ax \equiv 1 \pmod{n}$, právě jedno řešení $0 < x < n$ (a pokud jsou a, n soudělná, tak nemá žádné řešení).

Definice: Řešení rovnice $ax \equiv 1 \pmod{n}$ značíme $(a^{-1} \pmod{n})$ a nazýváme **multiplikativní inverz** čísla a modulo n (aby existoval, tak musí být a, n nesoudělná).

Pozorování: $(a^{-1} \pmod{n})$ snadno získáme pomocí rozšířeného Euklidova algoritmu.

Věta (speciální důsledek tzv. „čínské věty o zbytcích“ – bez Dk): Necht' a, b jsou nesoudělná přirozená čísla. Pak pro každá přirozená čísla x, y platí: $x \equiv y \pmod{ab}$ tehdy a jen tehdy, když $x \equiv y \pmod{a}$ a zároveň $x \equiv y \pmod{b}$.

Věta (malá Fermatova): Necht' p je prvočíslo. Pak $\forall k \in \{1, 2, \dots, p-1\}$ platí $k^{p-1} \equiv 1 \pmod{p}$.

Nyní máme vše co potřebujeme k definici a vysvětlení **RSA**:

1. Náhodně vyber dvě velká prvočísla p a q (např. každé s 200 binárními ciframi).
2. Spočítej $n = pq$ (v uvedeném případě má n cca 400 binárních cifer).
3. Vyber malé liché číslo e , které je nesoudělné s číslem $(p-1)(q-1)$.
4. Spočítej multiplikativní inverz d čísla e modulo $(p-1)(q-1)$.
5. Zveřejni (e, n) jako **veřejný RSA klíč** a uschovej (d, n) jako **soukromý RSA klíč**.

Věta (korektnost RSA): Funkce $P(M) = M^e \pmod{n}$ a $S(M) = M^d \pmod{n}$ definují dvojici inverzních funkcí na množině všech zpráv, tj. na množině všech čísel v $Z_n = \{0, 1, \dots, n-1\}$.

Proč je RSA bezpečná?

Na základě (e, n) není (zatím) nikdo schopen spočítat d aniž by znal rozklad $n = pq$ a tím pádem také číslo $(p-1)(q-1)$. A faktorizace velkých čísel je výpočetně těžký problém.

Jak je RSA rychlá?

To, jak rychle lze spočítat $P(M)$ a $S(M)$ závisí na tom, jak rychle umíme počítat zbytek modulo n při umocňování, tj. jak rychle lze spočítat $a^b \bmod n$.

UMOCNI (a, b, n) {kde binární zápis čísla b je $\langle b_k, \dots, b_0 \rangle$ }

$c := 1; d := a \bmod n;$

for $i := k-1$ downto 0 do

$c := 2 \cdot c;$

$d := (d \cdot d) \bmod n;$

if $b_i = 1$ then $c := c + 1;$

$d := (d \cdot a) \bmod n;$

Return(d)

Časová složitost: Pokud a, b jsou nejvýše t -bitová binární čísla, tak **UMOCNI** provede $O(t)$ aritmetických operací na (nejvýše) t -bitových číslech, tj. $O(t^3)$ bitových operací, pokud předpokládáme, že každá aritmetická operace na t -bitových číslech potřebuje $O(t^2)$ bitových operací (což je snadné ukázat).