11. přednáška

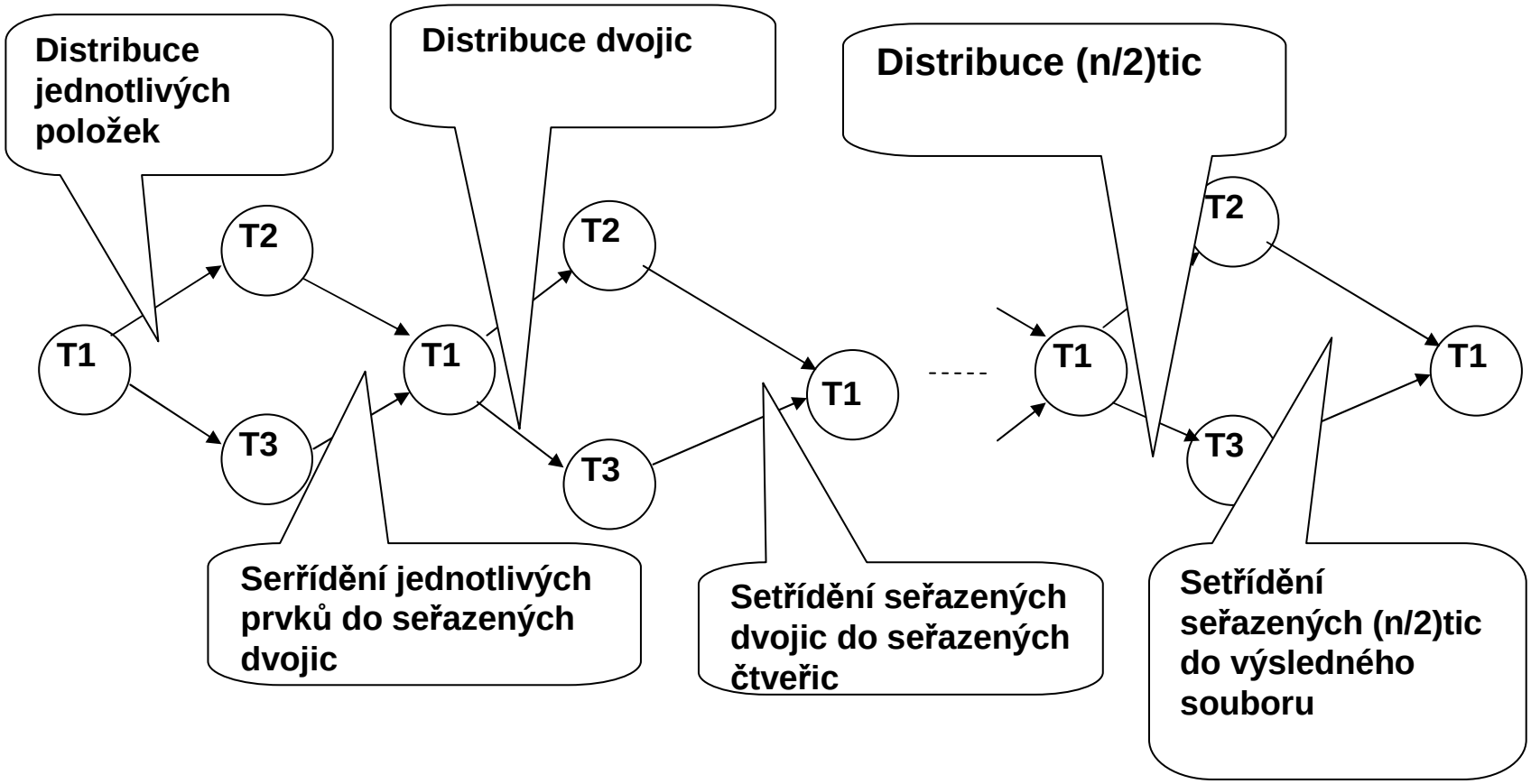
Řazení II.

- Demonstrační ukázka řazení
- Hodnocení metod řazení polí.
- Princip řazení sekvenčních souborů.
- Tří a čtyř pásková metoda řazení sekvenčních souborů - přímá a přirozená
- Mnohacestné vyvážené setřiďování
- Polyfázové setřiďování
- Hodnocení metod řazení sekvenčních souborů
- Zdroj: přednáška, skripta VKPT kap 7.

- Vyhledávání podřetězců v řetězcích.
- Knuth-Morris-Prattův algoritmus,
- Boyer-Mooreův algoritmus
- Zdroj: webová stránka předmětu –> Private ->Další texty-> Upravené texty ze skript -> Přednášky kurzu Algoritmy a datové struktury -> Hledání podřetězců
- nebo také: web ADS -> Public-> Prednask

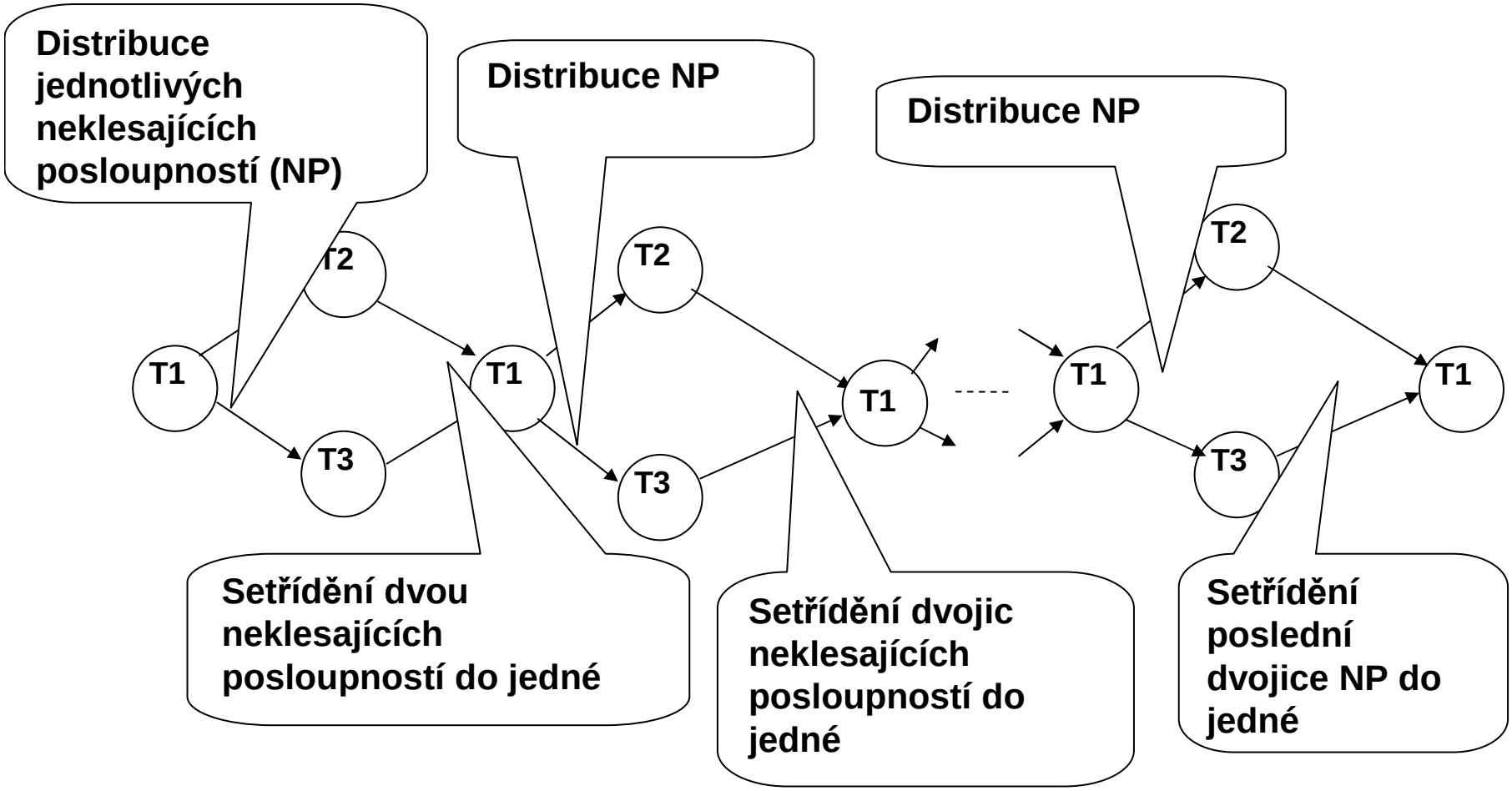
-> PrednRTF-> PTSearst.rtf

Přímé setřiďování souborů



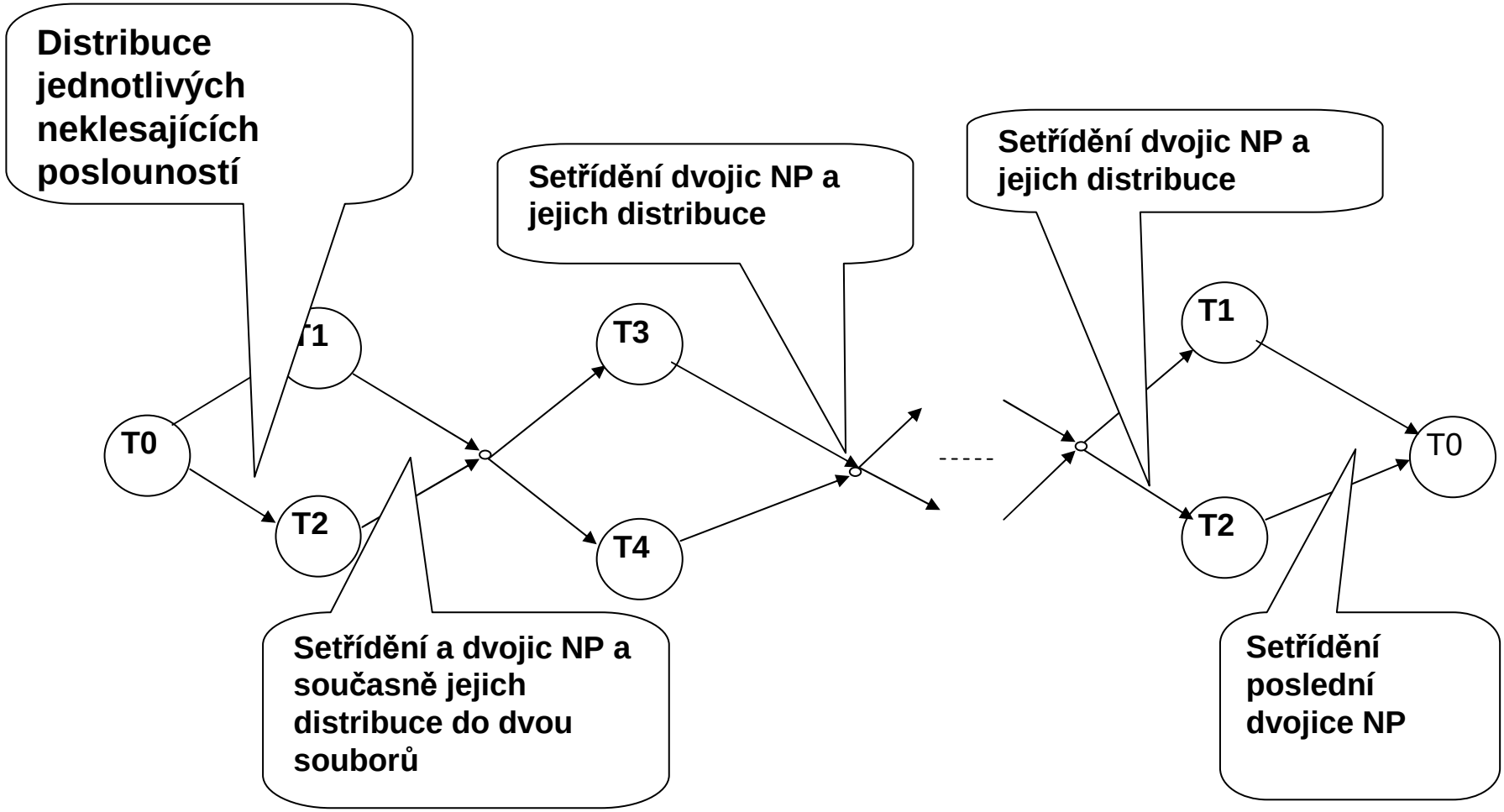
Počet distribučních a setřiďovacích fází je lg_2N, kde N je počet prvků. Metoda se nechová přirozeně.

Přirozené setřiďování – třípásková metoda



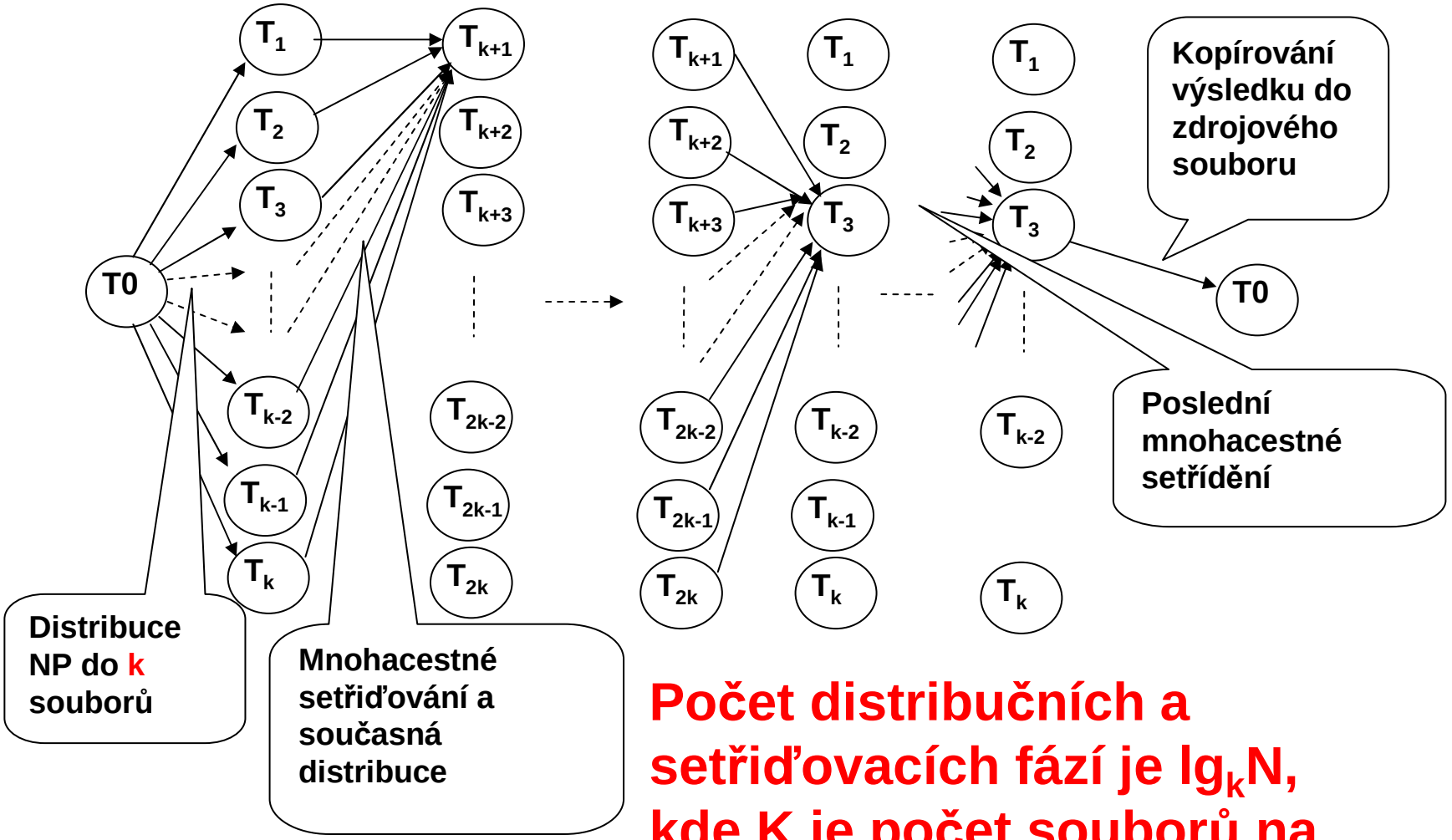
Počet distribučních a setřiďovacích fází je $\lg_2 n$, kde n je počet neklesajících posloupností! Metoda se chová přirozeně!

Přirozená čtyřpásková metoda



Metoda ušetří opakovanou fázi distribuce za cenu dalšího souboru (pásky).

Mnohacestné vyvážené setřiďování (Multiway Balanced merging)



Počet distribučních a setřiďovacích fází je $\lg_k N$, kde K je počet souborů na jedné straně a N je počet neklesajících posloupností

Polyfázové setřiďování

Polyfázové seřazování je založeno na snaze „ušetřit“ počet potřebných souborů (kdysi každý soubor představoval jednu magnetopáskovou mechaniku!!) při zachování rychlosti dané 2K soubory. Polyfázové setřiďování má s použitím K+1 souborů řádově shodnou složitost jako mnohafázové setřiďování s 2K soubory. Vykazuje tedy „úsporu“ K-1 souborů (mechanik).

Metoda je založena na principu postupného setřiďování K neklesajících posloupností ze zdrojových souborů do jednoho cílového souboru tak dlouho, dokud se jeden ze zdrojových souborů nevyprázdní (neobsahuje již žádnou neklesající posloupnost). Pak se tento prázdný soubor zamění s cílovým souborem a proces setřiďování pokračuje tak dlouho, až se vytvoří jediný výsledný cílový seřazený soubor.

Potíží metody je situace, kdy se ve stejném okamžiku vyprázdní více, než jeden soubor. Lze najít počáteční rozdělení posloupností tak, aby se v každém následujícím kroku vyprázdnila vždy jen jeden soubor?

Tento problém lze vyřešit pomocí Fibonacciho posloupností. Fibonacciho posloupnost k-tého řádu je definována k+1 počátečními hodnotami. Následující prvek má hodnotu součtu aktuálního prvku a k předchozích prvků.

Fibonacciho posloupnot 1. řádu s inicializačními konstantami 0 a 1 má další členy: 1,2,3,5,8,13,21,34,55,...

Fibonacciho posloupnost 4. řádu s inicializačními hodnotami 0,0,0,0,1 má další členy: 1,2,4,8,16,31,61,120,236...

Neklesající posloupnosti pro třípáskovou polyfázovou metodu, která pracuje ze dvou zdrojových souborů do jednoho cílového souboru, lze na počátku rozdělit s využitím Fibonacciho např. posloupnosti takto:

f1	f2	f3	∑
13	8	0	21
5	0	8	13
0	5	3	8
3	2	0	5
1	0	2	3
0	1	1	2
1	0	0	1

Počáteční rozložení pro 21 posloupností je:
5 a 5+8 .

Pro soustavu 6 pásek (souborů), se budou neklesající posloupnosti z 5 zdrojových souborů setřiďovat do jednoho cílového souboru. Pro inicializační rozdělení se použije Fibonacciho posloupnost 4. řádu, která má 5 počátečních hodnot a má tvar: 0,0,0,0,1,1,2,4,8,16,31,61,120,236,...

f1	f2	f3	f4	f5	f6	∑
16	15	14	12	8	0	65
8	7	6	4	0	8	33
4	3	2	0	4	4	17
2	1	0	2	2	2	9
1	0	1	1	1	1	5
0	1	0	0	0	0	1

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
f_i	0	0	0	0	1	1	2	4	8	16	31	61	120	236

65 posloupností je dáno počátečním součtem inicializačního rozdělení 16+15+14+12+8.

kde 16=1+1+2+4+8

15=1+2+4+8

14=2+4+8

12=4+8

8=8

Podobně pro 129 posloupností bychom dostali počáteční rozdělení: 31,30,28,24,16

kde 31= 1+2+4+8+16

30=2+4+8+16 atd.

Inicializační hodnoty jsou tedy dány součtem postupně zleva se snižujícího počtu předchozích hodnot.

Z toho vyplývá, že vyhovující počáteční rozdělení je možné jen pro určité hodnoty celkového počtu posloupností. Otázka, jak řešit případy libovolného počtu je otázkou vhodné distribuce „prázdných“ posloupností, které „zaslepí“ použití některých souborů.
Ukázka tabulky „vhodných“ čísel pro metody s různými počty souborů je uvedena ve skriptech.

Pozn. Ve skriptech je formálně nesprávný popis získání inicializačních hodnot.

Vyhledávání podřetězců v řetězcích

V dalším textu zavedeme značení:

P..... Vyhledávaný vzorek (pattern):
P[i] ..... i-tý znak vzorku
m=length(P) nebo PL .....délka vzorku

T..... Prohledávaný text
T[i]..... i-tý znak prohledávaného textu
n=length(T) nebo TL ..... délka prohledávaného textu

Klasický algoritmus:

```
function Match(P,T:String; PL,TL:index):index;
(* V případě neúspěchu vrátí hodnotu TL+1 *)
var PomZacT, BezT,BezP:index;
begin
  PomZacT:=1; BezT:=1; BezP:=1; (* inicializace *)
 while (BezT<=TL) and (BezP <=PL) do
    if T[BezT]=P[PezP]
    then begin (* Posun po vzorku v řetězci *)
   inc(BezT); inc(BezP)
    end else begin (* posun zač. řetězce a nové porovn.*)
      inc(PomZacT); BezT:=PomZacT; BezP:=1;
    end; (* while, if *)
  if BezP>PL
  then Match:=PomZacT (* Našel *)
 else Match:=BezT (* t.j. Nenašel a vrátil hodnotu TL+1 *)
end. (* function *)
```

Analýza algoritmu

- Je-li vzorek na počátku řetězce, provede se PL porovnání (nejlepší případ)
- Není-li P[i] v řetězci T obsažen, provede se TL srovnání.

V nejhorším případě dojde na každé startovací pozici k (PL-1) shodám. Pak se provede mn srovnání a algoritmus má složitost O(mn). Proto musíme ukázat, že takový případ může nastat. (V některých algoritmech může být jeden krok časově náročný, zatím co jiný nikoliv). Takový případ je např. P='AAA...AB' a T='AAA...AAA'.

V přirozených jazycích jsou takové případy řídké. Uvedený algoritmus v nich pracuje průměrně dobře. (Statistiky ukazují, cca. 1.1 porovnání na jeden znak řetězce T). Pro některé aplikace je však nepřijatelný návrat v řetězci (t.j. BezT:=PomZacT v cyklu). Následující algoritmus byl vytvořen pro odstranění návratu a zlepšuje jeho nejhorší případ.

- Řídící
- Standartní
- Nejoptimálnější - nesmysl
- Potencionální nebo raději potenciální
- Funkcionalita versus funkce?

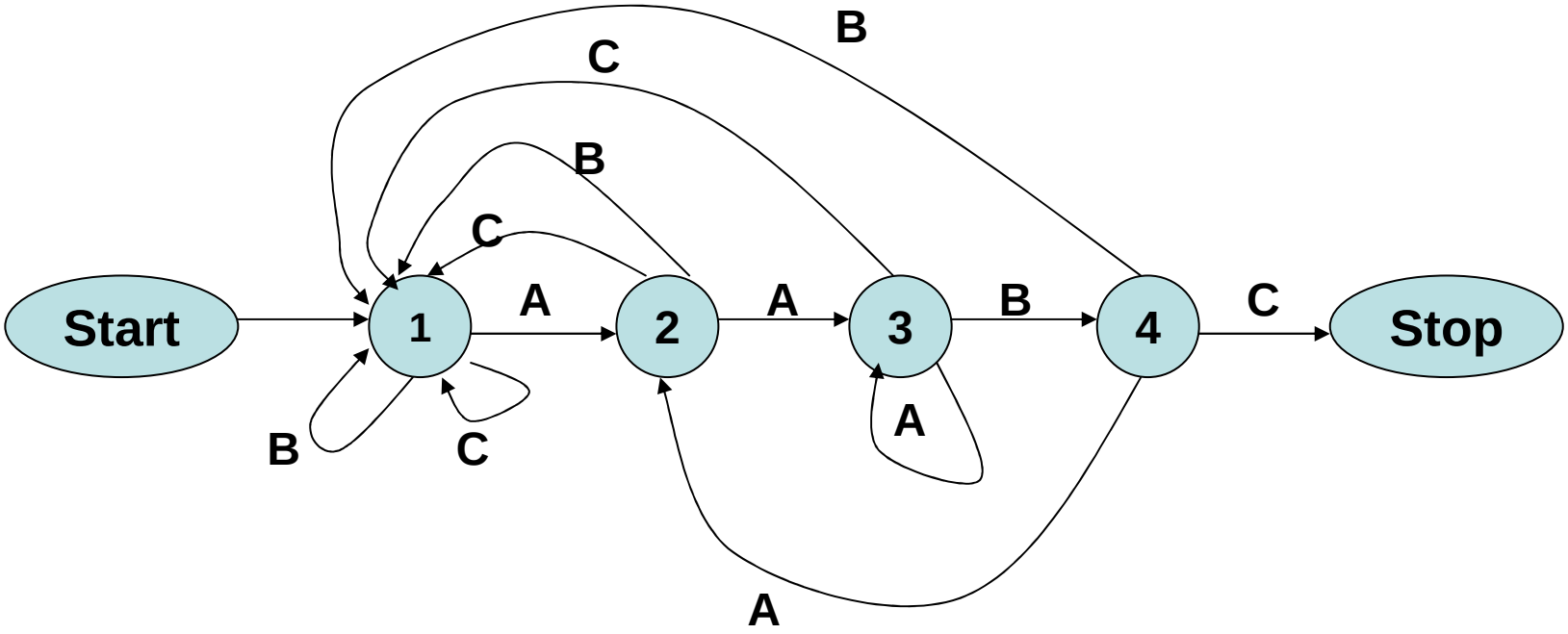
Knuth-Morris-Prattův algoritmus (KMP)

Tento algoritmus využívá tzv. konečný automat. Konečný automat můžeme interpretovat jako zvláštní druh stroje - procesoru, nebo vývojového (postupového) diagramu či grafu a lze ho zde použít bez znalosti teorie automatů.

Použitý konečný automat v podobě grafu má tři typy uzlů:

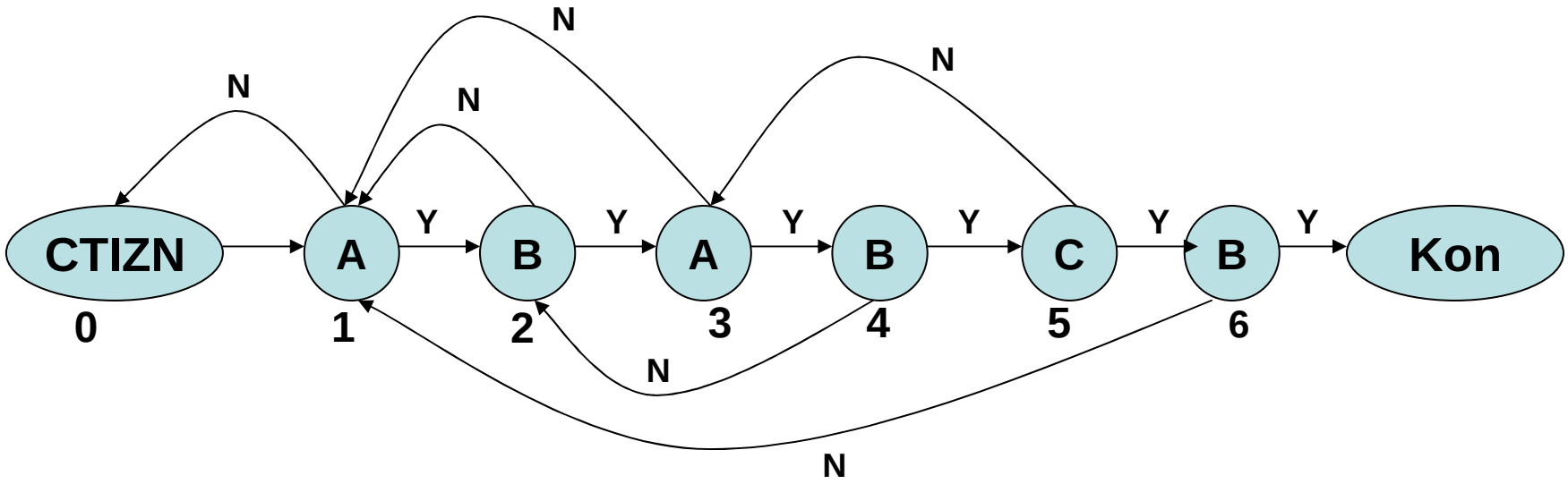
- Startovací uzel START
- Uzel STOP s významem "úspěšný konec"
- Uzel READ s významem "čti další znak". Je-li řetězec na konci a není k dispozici další znak, jde o neúspěšný konec

Nechť A je množina znaků (abeceda), které mohou být v řetězci, a o=|A| je kardinalita abecedy. Pak z každého uzlu vychází o orientovaných hran, oceněných jednotlivými znaky abecedy. Pro vzorek 'AABC' a abecedu {A,B,C} dostaneme automat:



Nevýhodou je, že z každého uzlu vychází tolik hran, kolik je znaků abecedy.

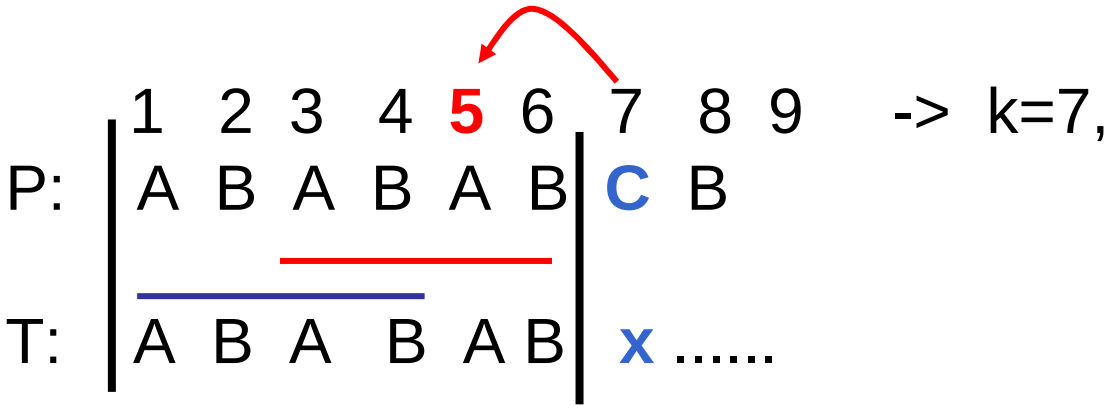
Graf automatu KMP používá pouze dvě hrany:
ANO – souhlas – Y
NE – nesouhlas – N
Pro vzorek ‘ABABCB’ má automat KMP tvar:



1. Na hraně N se porovná starý znak s uzlem
2. Z uzlu CTIZN hrana Y čte nový znak
3. Na hraně Y se čte nový znak a porovná se s uzlem
4. Dosažení uzlu Kon znamená nalezení vzorku

Konstrukce automatu KMP:

Automat je reprezentován vzorkem P a vektorem FAIL, který má prvky typu integer, reprezentující cílový index zpětné šipky.
Příklad tvorby vektoru pro P='ABABABCB'
FAIL[1]=0;



Předpokládejme, že **x<>C**; pak další možné místo, na kterém může vzorek v textu začínat je třetí pozice Protože došlo k nesouhlasu na 7 pozici a protože platí:
(P**1**...P**4**)=(P**3**...P**6**) může nové porovnání začít na indexu **5** a tedy FAIL[**7**]=5
Platí tedy: FAIL: 0 1 1 2 3 4 **5** 1
FAIL[k]= max r{(r<k) and (P_1..P_{r-1})=(P_{k-r+1}...P_{k-1})}

Vyjádřeno jinak:
FAIL[k]= max r{(r<k) and ($P_1..P_{r-1}$)=($P_{k-r+1}...P_{k-1}$)}

```
procedure KMPGraf(T,P:String; TL,PL:index; var
Fail:TFail);
var
  k,r: index;
begin
  Fail[1]:=0;
  for k:=2 to PL do begin
    r:=Fail[k-1];
    while (r>0) and (P[r]<>P[k-1]) do r:=Fail[r];
                    (* použit zkratový booleovský výraz ! *)
    Fail[k]:=r+1
  end; (* for *)
end; (* procedure *)
```

Celkový počet porovnání je (2m-3). To představuje lineární časovou složitost. (Odvození složitosti je v pom. textu).

Algoritmus pro KMP vyhledání vzorku v řetězci

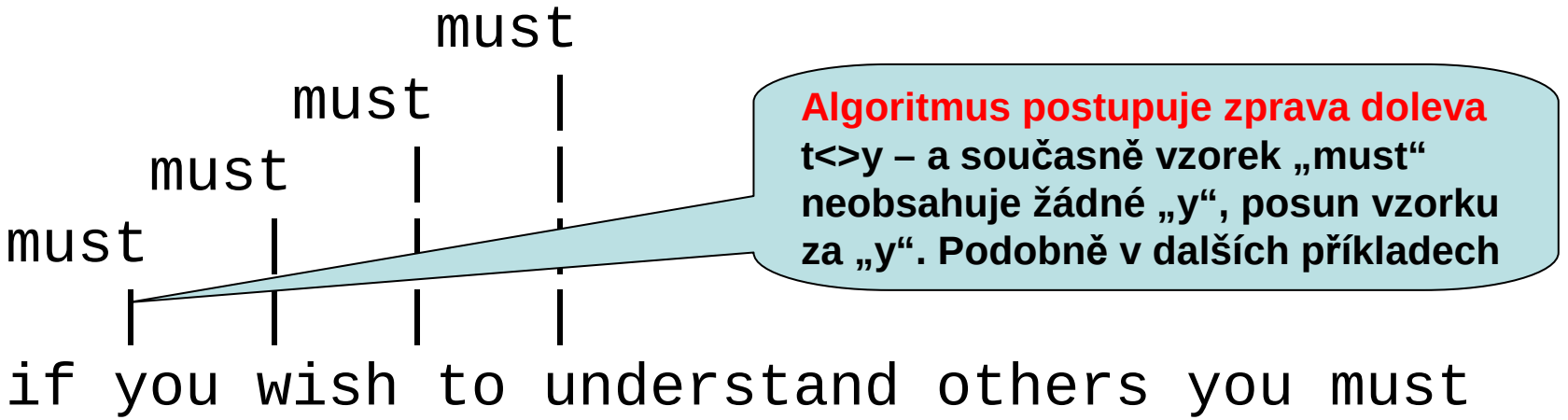
```
function KMPMatch(T,P:string; TL,PL:index;
Fail:TFail):index;
var
  TInd,PInd:index;
begin
  TInd:=1; PInd:=1;
  while (TInd<=TL) and (PInd<=PL) do
    if (PInd=0) or (T[TInd]=P[PInd]) (* ZkratovýBool. výr.! *)
    then begin
      inc(TInd);
      inc(PInd)
    end else (* jdi po hraně "N" *)
      PInd:=Fail[Pind];
  if PInd>PL
  then KMPMatch=TInd-PL (* Našel vzorek na ind. TInd-PL *)
  else KMPMatch:=TInd (* Nenašel, vrací hodnotu TL+1 *)
end; (* function *)
```

Tento algoritmus provede maximálně 2n porovnání (n=TL), a celkový algoritmus vyhledávání, který je složen z konstrukce automatu a z vyhledání má tedy složitost Θ(n+m), což je lepší než Θ(nm).

Některé empirické studie ukazují, že přímý i KMP algoritmus udělají v řetězci vytvořeném přirozeným jazykem přibližně stejný počet porovnání, ale KMP nejde v textu zpět, což lze považovat za jistou výhodu.

Boyer-Mooreův algoritmus

Boyer-Mooreův algoritmus (BMA) vychází z úvahy, že při porovnávání vzorku s řetězcem, lze některé znaky, o nichž lze prohlásit, že se nemohou rovnat, přímo přeskočit. Čím delší je vyhledávaný vzorek (a čím více se poskytuje informace) tím větší počet znaků v prohledávaném řetězci může dobrý algoritmus přeskočit.



Dále porovnání vypadá takto:

```
                                   must
                               must   |
                           must   |   |
                        must  |   |   |
                    must  ||  |   |   |
                  must |  ||  |   |   |
              must ||| |  ||  |   |   |
                 | ||| |  ||  |   |   |
if you wish to understand others you must
```

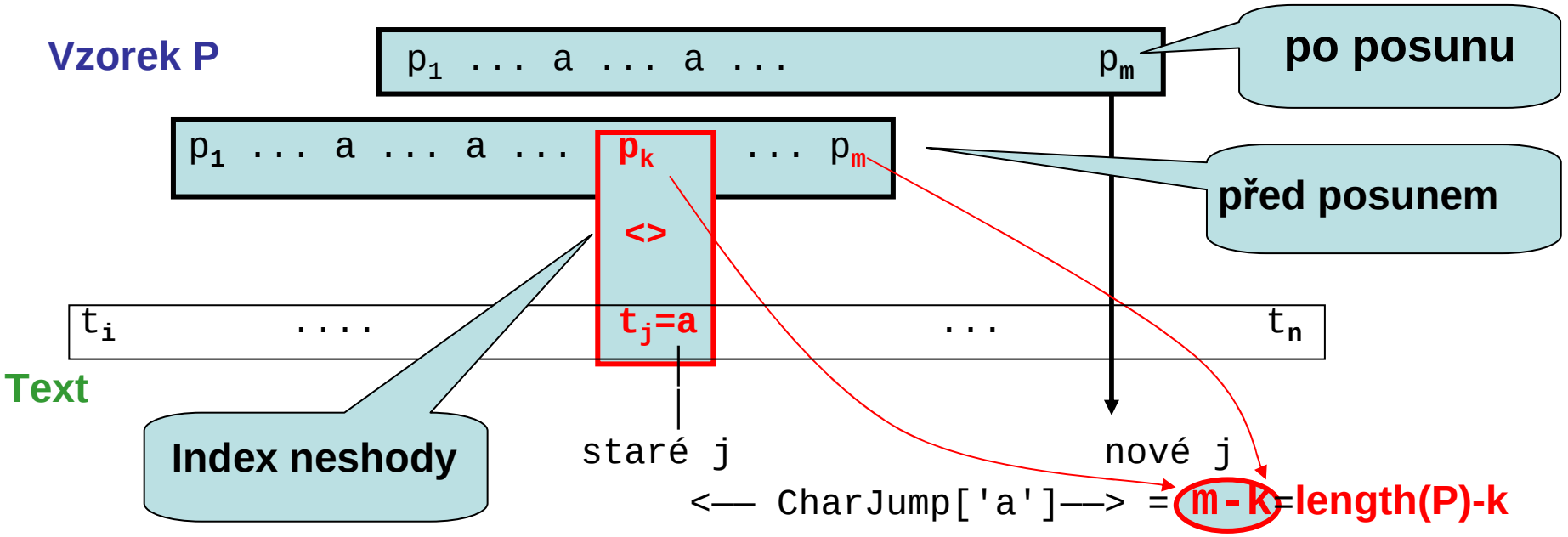
u<>r ⇨ posun vzorku právě za **r**

s<>o ⇨ posun vzorku právě za **o**

Provede se jen 18 porovnání pro nalezení 18 pozice

Počet pozic, o které lze při nesouhlasu porovnávaného vzorku "skočit" dopředu, závisí na znaku, který označme t_j. Hodnoty těchto počtů (skoků) můžeme uložit do pole CharJump, které bude indexováno typem "znak" (bude mít počet prvků shodný s počtem prvků použité abecedy). Pro řízení prohlížecího algoritmu je ale pohodlnější uchovávat hodnotu, o kterou se má zvýšit index j, od něhož se zahájí testování ve směru zprava-doleva, než počet pozic, o který se vzorek posouvá podél prohledávaného textu. Z předchozího příkladu je vidět, že když se t_j nevyskytne ve vzorku P, lze poskočit o "m" pozic, což je výhodné.

Na následujícím obrázku je vidět, jak lze vypočítat skok v případě, že se t_j ve vzorku nachází. Ve skutečnosti se t_j (v příkladu znak 'a') může ve vzorku vyskytnout vícekrát. V tom případě provedeme nejmenší možný skok, odvozený od nejpravějšího výskytu znaku t_j ('a') ve vzorku. (Pro algoritmus je typické, že vzorek se nikdy neposunuje podél textu ve zpětném směru; je-li aktuální pozice v PP vlevo od nejpravějšího výskytu t_j v P, nepřináší již CharJump[t_j] žádný užitek).



Znaky vzorku se porovnávají se znaky textu zprava doleva.
Z indexu neshody (k) se spočítá hodnota posunu vzorku doprava – vpřed (m-k)

Algoritmus pro výpočet hodnot pole CharJump:

```
procedure ComputeJumps(P:TString; var CharJump:TCharJump);
(* stanovení hodnot pole CharJump, které určují posuv vzorku *)
var
  ch:char;
  k:integer;
begin
  for ch:=chr(0) to chr(255) do begin
    CharJump[ch]:=length(P);
  end; (* for *)
  for k:=1 to length(P) do begin
    CharJump[P[k]]:= length(P)-k
(* viz (m-k)na předch. str. *)
  end; (* for*)
end; (* procedure *)
```

Druhá heuristika BMA

Již použití pole CharJump pro postup vzorku podél textu činí BMA podstatně rychlejší než algoritmus KMP. Přesto lze použít další myšlenky:

Předpokládejme, že nejpravější úsek vzorku P souhlasí s odpovídajícím úsekem v textu a teprve vlevo od tohoto úseku dochází k nesouhlasu:

```
P:          batsandc|ats|
                    |   |
                    |   |
T:  ...           d|ats| ...
                   ^
                   j
```

Kombinace ‘ats’ se ve vzorku ale vyskytuje dvakrát!

Můžeme posunout vzorek P tak, aby kombinace 'ats' stály proti sobě.

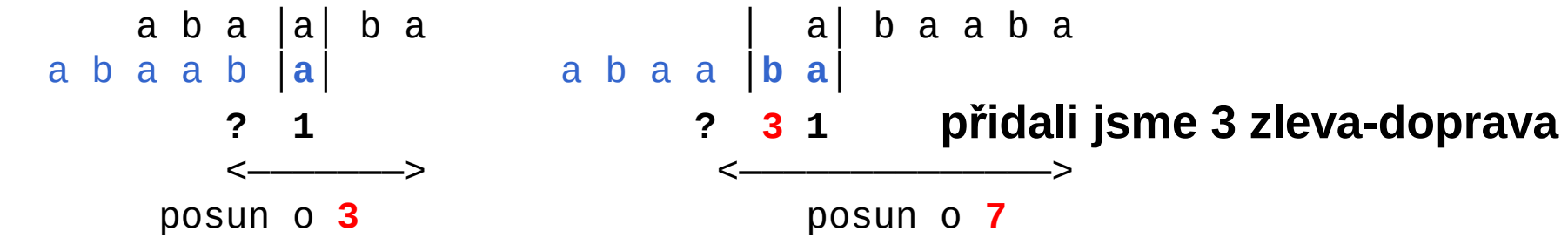
```
P:              b|ats|andcats
                 |   |     ^
T:        ...   d|ats|...  nové j
```

Je-li r nejlevější index, pro který platí:
$(p_r .. p_{r+m-k-1}) = (p_{k+1} .. p_m)$ a současně $p_{r-1} <> p_k$

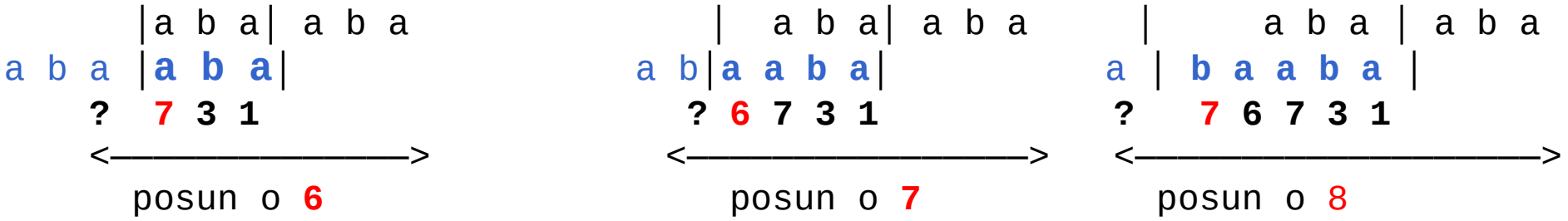
pak pomocné pole MatchJump[k]=m-r+1.

Pole MatchJump je definované pro daný vzorek P a určí se algoritmem popsaným ve studijní opoře.

Příklad pro výpočet MatchJump ilustrujme ukázkou pro vzorek 'abaaba, Ve třetím řádku je vznikající vektor MatchJump. Nová hodnota je červená. Nad otazníkem je neshoda.



Všimněme si, že první 'ba' a druhé 'ba' ve druhém kroku není použito, protože obě předchází 'a' a nedochází tedy k nesouhlasu na pozici před příponou. Dojde-li k nesouhlasu na 4. pozici vzorku, neexistuje žádná poloha pro zarovnání s jiným 'a' vzorku, než s prvním a posun je o 7.



Výsledkem pro řetězec P: a b a a b a
je pole MatchJump: 8 7 6 7 3 1

```
function BMA(P,T:string; CharJump:TCharJump;
MatchJump:TMatchJump):index;
(* funkce pro stanovení indexu polohy vzorku nalezeného v daném textu *)
var
  j,  (* j je index do textu *)  k (* k je index do vzorku *)  :index;
begin
  j:=length(T);  k:=length(P);
  while (j<=length(T)) and (k>0) do begin
    if T[j]=P[k]
    then begin
      j:=j-1;   k:=k-1
    end else begin
      j:=j+max(CharJump[T[j]], MatchJump[k]);
      k:=length(P)
    end; (* if *)
 end; (* while *)
 if k=0
  then BMA:=j+1 (* našla se shoda *)
 else BMA:=length(T)+1; (* shoda se nenašla *)
end; (* function *)
```

Závěrečné poznámky

Chování BMA závisí na kardinalitě abecedy a na opakování podřetězců ve vzorku. Empirické studie s hovorovým jazykem ukázaly, že pro délku vzorku m>5 provádí algoritmus přibližně 0.24 až 0.3 porovnání z počtu znaků v prohledávaném textu. Jinými slovy, porovnává asi jednu čtvrtinu až jednu třetinu znaků prohledávaného textu.