5.1. Úvod do reťazcov(Strings)

Dáta, ktoré spracovávame, sa nedajú vždy rozložiť na menšie logické celky. Preto sa na ne budeme dívať tvrdohlavým pohľadom a v tejto kapitole sa budeme zaoberať dátami charakterizované len tým, že sa dajú zapísať do reťazca. Reťazec je lineárna, často dlhá, postupnosť znakov – napr. text tejto kapitoly tvorí reťazec.

Reťazce sú zrejme to hlavné na čo sa programy na spracovanie textu zameriavajú. Tieto systémy spracovávajú textové reťazce, voľne definované ako postupnosti písmen, čísel a špeciálnych znakov. Tieto objekty môžu byť naozaj veľké, napr. kniha obsahuje aj milión znakov.

Ďalší typ reťazcov sú binárne reťazce(binary strings), jednoduché postupnosti núl a jednotiek. Možno sa na prvý pohľad až tak nelíšia od všeobecných reťazcov, ale ako neskôr uvidíme má zmysel ich chápať samostatne, pretože na prácu s nimi sa používajú iné algoritmy, a binárne reťazce sa prirodzene objavujú v mnohých aplikáciách – napr. obrázok v počítači charakterizuje nejaký binárny reťazec. Nasleduje príklad binárneho reťazca:

```
00101010011101010101111000
```

Binárne reťazce sa líšia od všeobecných hlavne veľkosťou abecedy z ktorej vyberáme znaky. V klasickom texte sa jeden znak dá, povedzme, charakterizovať 8 bitmi, takže aj obyčajný text sa dá chápať ako binárna postupnosť, ktorú avšak nevnímame po bitoch ale po 8 bitových častiach(textových znakoch). Uvidíme, že veľkosť abecedy je vo viacerých situáciách významných faktorom vo vývoji algoritmov na spracovanie textu.

5.1.1. Vyhľadávanie podreťazca(pattern matching)

Základnou operáciou na reťazcoch je vyhľadávanie podreťazca: daný je textový reťazec dĺžky N a vzorka (podreťazec) dĺžky M, nájdite výskyt vzorky v texte (pojem „text“ chápeme ľubovoľný reťazec, teda aj binárny). Väčšina algoritmov sa dá triviálne rozšíriť, aby našla všetky výskyty vzorky v danom text, keďže algoritmy čítajú text postupne, stačí algoritmus znovu pustiť ďalej od bodu, kde sme našli vzorku.

Algoritmy, riešiace tento problém majú zaujímavú históriu. Existuje zrejmý priamočiari(brute-force) algoritmus, ktorý sa používa bezkonkurenčne najčastejšie. Aj keď jeho najhoršia časová zložitosť je úmerná MN, reťazce, ktoré sa vyskytujú vo väčšine aplikácií, vedú k času, ktorý je skoro vždy úmerný $M+N$. Naviac, tento algoritmus je, vďaka svojej štruktúre, vhodný na efektívnu implementáciu na mnohých systémoch; optimalizovaná metóda poskytuje akýsi štandard, ktorý sa ťažko prekonáva aj so šikovnejšími algoritmami.

V roku 1970 , S.A. Cook dokázal teoretický výsledok o určitom abstraktnom výpočtovom modeli, z ktorého vyplývala existencia algoritmu riešiaceho tento problém v čase úmernom $M+N$ v najhoršom prípade. D.E.Knuth a V.R.Pratt neskôr prepracovali Cookovu konštrukciu na jednoduchý a celkom praktický algoritmus. Toto vyzeralo ako ojedinelý a veľmi uspokojujúci teoretický výsledok s neočakávane jednoduchou praktickou realizáciou. Ukázalo sa však, že J.H.Morris objavil vlastne identický algoritmus ako riešenie znepokojujúceho problému v svojom textovom editore. Skutočnosť, že rovnaký algoritmus vznikol v tak odlišných oblastiach mu dáva dôveru a vážnosť ako základného riešenia tohto problému.

Knuth, Morris a Pratt publikovali svoj výsledok až v roku 1976. Medzičasom ale R.S.Boyer a J.S.Moore (nezávisle aj R.W.Gosper!) vynašli algoritmus, ktorý je v mnohých situáciách neporovnateľne rýchlejší, pretože často sa pozrie len na zlomok znakov textu. Mnohé textové editori ho využívajú na znateľné zníženie

odozvy pri hľadaniach. Oba tieto algoritmy *Knuth-Morris-Prattov* a *Boyer-Mooreov* vyžadujú komplikované predpočítavanie na danej vzorke, ktoré je náročné na pochopenie a obmedzuje širšie rozšírenie týchto efektívnych algoritmov.

V roku 1980, R.M.Karp a M.O.Rabin si všimli, že tento problém nie je až tak odlišný od klasického vyhľadávacieho problému a vyvinuli algoritmus, ktorý beží skoro vždy v čase *M+N*. Naviac sa ľahko rozširuje do viacerých dimenzií, čo je vhodné napríklad pre vyhľadávanie vzoriek v dvoj-rozmernej mriežke.

5.1.2. Priamočiari algoritmus

Zrejmá metóda riešenia tohto problému, ktorá nás hneď napadne je, že pre každú pozíciu v texte, na ktorej by sa mohla daná vzorka vyskytovať, jednoducho vyskúšame, či sa tam nachádza, alebo nie. Nasledujúci program priamo implementuje túto myšlienku:

```
function search(p,t:string):integer;
var i,j,k,n,m:integer;
begin
  n:=Length(t); m:=Length(p);
  search:=-1; {zatiaľ sme nič nenašli}
  for i:=1 to n-m do
  begin
    k:=1; {zatiaľ dobré}
    for j:=i to i+m-1 do
      if t[j]<>p[j-i+1] then
      begin
        k:=0; {zlý začiatok}
        break;
      end;
    if k=1 then {našli sme výskyt}
    begin
      search:=i; exit;
    end;
  end;
end;
```

```
int search(char *p, char *t)
{
  int i,j,m,n;

  m=strlen(p); /* vzorka */
  n=strlen(t); /* text */

  for(i=0;i<n-m+1;i++)
  {
    /* vyskúšame začať na i-tom znaku */
    for(j=i;j<i+m;j++)
      if(t[i]!=p[j-i])
        break;  /* zlý začiatok */

    if(j==i+m)  /* našli sme výskyt */
      return i;
  }

  return -1;
}
```

V programe si pre každý možný začiatok výskytu vzorky v texte *i* naozaj overíme, či všetkých *M* nasledujúcich znakov textu súhlasí so znakmi vzorky, ak áno našli sme prvý výskyt vzorky. Uvedený algoritmus pracuje v čase *O(NM)*. V obyčajnom texte (veľká abeceda znakov) algoritmus vnútornú slučku iteruje len veľmi zriedka, takže čas sa blíži *M+N*. Problém je, že v binárnych reťazcoch to nemusí byť veľmi také jednoduché.

5.1.3. Rozšírenia problému

Vyhľadávací problém nájdenia všetkých výskytov danej vzorky ešte rozšírime. Prvé rozšírenie spočíva vo vyhľadávaní viacerých vzoriek v tom istom texte naraz, teda nás zaujíma nájsť také miesta v danom texte, že sa tam začína nejaká z daných vzoriek. Druhé rozšírenie bude prechod do vyšších dimenzií. Predstavme si, že nevyhľadávame lineárny reťazec v lineárnom reťazci, ale že napr. vyhľadávame v dvojrozmernej mriežke dvojrozmerný vzor. Tieto problémy tiež zaraďujeme do kapitoly venovanej reťazcom, pretože základ používaných algoritmov leží práve v riešeniach pre lineárne reťazce..

Všimnime si, že pre každé toto rozšírenie znovu existuje jednoduchý priamočiari postup, riešiaci daný problém, ktorý nie je zrovna časovo efektívny, čo nám necháva priestor pre lepšie algoritmy.

5.2. Knuth-Morris-Prattov algoritmus

Prirodzený algoritmus, ktorý sa priamočiaro rozširuje do vyšších dimenzií, vznikol nezávisle na viacerých miestach, dosahuje optimálny lineárny čas na predpočítanie na vzorke a aj na vyhľadávanie v texte samotné.

5.2.1. Princíp

Hlavná myšlienka algoritmu spočíva v tom, že pri neúspešnom začiatku, sa nevrátime v texte dozadu, ako v priamočiarom algoritme. Namiesto toho využijeme informáciu, ktorú nám „ponúka“ už prečítaný text, a v každom kroku radšej určíme najdlhšiu časť vzorky, ktorá končí v danom mieste textu, potom vieme nájsť miesta výskytu vzorky tak, že v danom mieste končí celá vzorka.

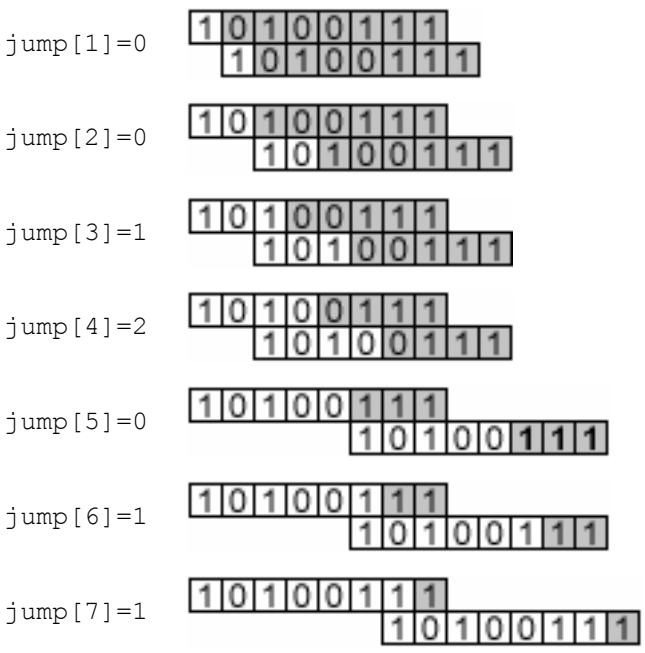
Predstavme si, že hľadáme vzorku, ktorej prvý znak sa nenachádza na žiadnom inom mieste vo vzorke (napr. `10000000`). Uvažujme, že sme začali vzorku porovnávať od pozície *i* v texte a našli sme chybu po prečítaní *j* znakov – teda *j* znakov vzorky súhlasí s textom, prvá chyba nastane na *j+1* znaku. Pri vyhľadávaní takejto vzorky vieme, že nemá zmysel začať porovnávať vzorku znovu od *i+1* znaku textu, ale až od *i+j* znaku textu, pretože znaky *i+1, ..., i+j-1* textu sú 0 (keďže súhlasili s vyhľadávanou vzorkou) a prvý znak vzorky je 1.

Knuth-Morris-Prattov (skrátene KMP) algoritmus je zovšeobecnením tohto pozorovania. Ukážeme si, že všetko môžeme zariadiť tak, aby sme znak textu po jeho spracovaní už nikdy viac nepotrebovali!

5.2.2. Ako to funguje?

V prvom rade si uvedomíme, že po nájdení chyby nám preskočenie celej vzorky a začatie od znova v mieste chyby, nie vždy pomôže, pretože vzorka sa môže rovnať v mieste chyby. Uvažujme príklad vzorky `10100111` vyhľadávanej v texte `10`**`10100111`**. Prvú chybu objavíme na piatom znaku textu, ale mali by sme sa vrátiť a začať znovu od tretieho znaku textu, pretože inak by sme mohli prehliadnuť výskyt vzorky!

Našťastie, všetko závisí len na vyhľadávanej vzorke! Dopredu si problémové situácie rozmyslíme a pri vyhľadávaní v texte už nenastanú ťažkosti. Nech *M* je dĺžka vzorky, definujme si pole `jump[1…M]`, ktoré nám v prípade chyby na *j*-tom znaku vzorky určí, koľko veľa znakov v texte sa treba vrátiť, aby sme mohli nájsť najbližší možný ďalší výskyt vzorky v texte. Príklad poľa `jump` sa nachádza povedľa.



Uvažujme teraz, že sme našli chybu na *i*-tom znaku textu a *j*-tom znaku vzorky (*i*-ty znak textu sa nezhodoval s *j*-tym znakom vzorky), potom podľa definície tabuľky `jump` sa ďalší možný výskyt vzorky bude začínať na (*i*-`jump[`*j*`]`)-tom znaku textu. Avšak o prvých `jump[`*i*`]` znakoch vzorky už vieme, že sa budú zhodovať, takže jednoducho postačuje zmeniť *j*`:=jump[`*j*`]` a *i* ponecháme nezmenené!

Predpokladajme, že pole `jump` už máme zo vstupnej vzorky vypočítané, samotná realizácia algoritmu vyhľadávania potom bude vyzerať nasledovne:

```
procedure kmp_search(p,text:string);
var i,j,m,n:integer;
begin
  m:=Length(p); n:=Length(text);
  i:=0;j:=0;
  kmp_init(p);
  repeat
    while((j>=0)and(text[i+1]<>p[j+1]))
      do j:=jump[j];
    if j=M-1 then
    begin
      { výskyt zač.: text[i-M+1+1] }
      j:=jump[j];
    end;
    inc(i); inc(j);
  until i=n;
end;
```

```
void kmp_search(char *p,char *text)
{
  int i,j,m,n;
  m = strlen(p), N = strlen(text);

  kmp_init(p);
  for (i=0,j=0; j<M && i<N; i++,j++)
  {
    while ((j>=0) && (text[i]!=p[j]))
      j = jump[j];
    if (j==M-1)
    {
      /* výskyt zač.: text[i-M+1] */
      j=jump[j];
    }
  }
}
```

V uvedenej implementácií v jazyku Pascal predpokladáme, že znaky vzorky sú `p[1..M]`, znaky textu sú `text[1..N]`. V jazyku C to je vzorka `p[0..M-1]` a text sú znaky `text[0..N-1]`. Pole jump predpokladáme vyplnené podľa vyššie uvedenej definície, pričom `jump[0]=-1`. Pred iteráciou cyklu vieme, že posledných *j-1* znakov textu súhlasí z prvými *j-1* znakmi vzorky. Ak súhlasí aj *j*-ty znak, postupujeme ďalej. Ak nesúhlasí, tak vieme, že najväčšia časť (začiatku) vzorky, ktorá súhlasí s poslednými znakmi textu je dlhá `jump[j]` znakov. Takto pokračujeme až do vyčerpania celého textu. V prípade, že narazíme na koniec vzorky, vyhlásime fiktívny nesúlad písmeniek – vyhlásime nález a vzorku posunieme na najbližší možný ďalší výskyt.

Zamyslime sa nad náročnosťou uvedeného vyhľadávania. V každom znaku textu sa buď posunieme vo vzorke ďalej alebo sa vrátime. Problém je ten, že sa môžeme v jednom znaku textu vracať aj viac razy. Stačí si ale uvedomiť, že sa za celý beh vyhľadávania spolu nevrátime viac, ako sme šli dopredu a dopredu sme šli rádovo *N* krát a teda celý výpočet si vyžaduje rádovo *N* operácií.

Vymysleli sme funkčný algoritmus, ku ktorému avšak zostáva zostrojiť pole `jump`. Všimnime si, že na skonštruovanie poľa `jump` nám postačuje algoritmus podobný priamočiaremu hľadaniu vzorky v texte. Nech je vzorka dlhá *M* a text v ktorom ju hľadáme *N*, potom už teraz vieme zostrojiť algoritmus bežiaci v čase $O(N+M^2)$.

Zostáva dokončiť celú myšlienku tým, že pole `jump` sa dá predpočítavať prakticky rovnakým algoritmom ako je samotné vyhľadávanie vzorky v texte. Stačí si uvedomiť, že v momente keď chceme vypočítať hodnotu `jump[j]` pre $j=1,...,N$ už hodnoty `jump[i]` pre $i<j$ poznáme a naviac tieto hodnoty nám plne postačujú. Program je vlastne rovnaký s hlavnou slučkou KMP, na rozdiel od nej ale porovnáva vzorku so sebou samou.

Nasledujúca implementácia ukončuje realizáciu Knuth-Morris-Prattovho algoritmu:

```
var jump:array[0..Max] of integer;

procedure kmp_init(p:string);
var i,j,m:integer;
begin
  m:=Length(p);
  jump[0] := -1;
  i:=0; j:=-1;
  repeat
    while((j>=0)and(p[i+1]<>p[j+1])) do
      j:=jump[j];
    inc(i); inc(j);
    jump[i]:=j;
  until i=m;
end;
```

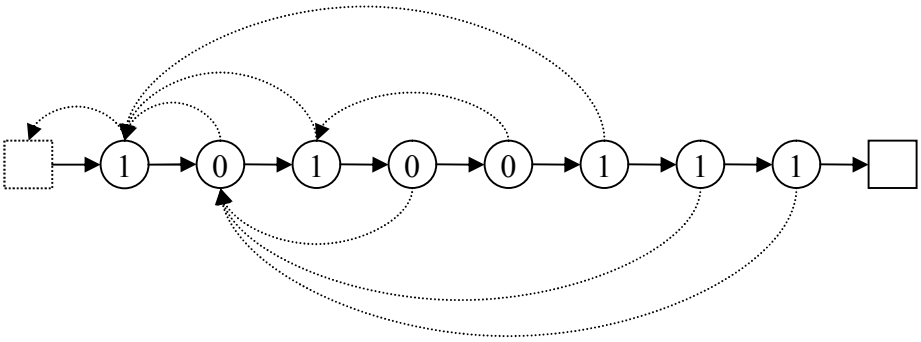
```
int jump[Max+1];

void kmp_init(char *p)
{
  int i,j,m;
  m = strlen(p);
  jump[0] = -1;
  for (i=0,j=-1; i<M; i++,j++,jump[i]=j)
  {
    while ((j >= 0) && (p[i] != p[j]))
      j = jump[j];
  }
}
```

5.2.3. Predstava konečných automatov

Každý problém vyhľadávania vzorky môžeme charakterizovať modelom, ktorý nazývame konečné automaty. Konečný automat pozostáva zo stavov (zobrazených ako čísla v krúžku) a prechodoch (znázornených ako šípky). Z každého stavu vedú dva prechody: plná čiara – znak textu súhlasí so znakom vzorky; prerušovaná čiara – znak textu nesúhlasí so vzorkou.

Stavy reprezentujú, ako veľa vzorky sa nachádza medzi poslednými znakmi zatiaľ preskúmaného textu. Automat je vždy v práve jednom stave. Na základe zhody alebo nezhody znakov sa automat buď posunie do nasledujúceho stavu, alebo sa vráti naspäť.



Táto predstava automatov, nám pomáha k lepšej predstave takýchto problémov, dokážeme si jasnejšie predstaviť, čo z informácie ktorú už vieme využiť, aby sme zasa nezačínali odznovu – napr. automat na obrázku sa dá pre našu vzorku a abecedu $\sum = \{0,1\}$ ešte zlepšiť. Pre danú pevnú vzorku sa dá zostrojiť takýto automat, z ktorého pomocou sa jednoducho zostrojí asymptoticky optimálny vyhľadávací „stroj“ – program – pre danú vzorku (v ľubovoľnom texte).

5.2.4. Náročnosť

Knuth-Morris-Prattov algoritmus nám poskytuje ideálny nástroj na realizáciu vyhľadávania vzorky v texte. Algoritmus sa dá rozumne rozšíriť na vyhľadávanie množiny vzoriek, ako aj vzoriek vyššej dimenzie. Ukázali sme si, že základný KMP na vyhľadanie jedného lineárneho reťazca dosahuje optimálnu časovú aj pamäťovú zložitosť.