Meklēšana I

1. Meklēšana simbolu virknēs

Uzdevuma nostādne: Dots teksts

$$T = T[0] \ldots T[n-1].$$

Vai šajā virknē ir atrodama apakšvirkne

$$P = P[0] \ldots P[m-1]?$$

Naivais algoritms

Ņemam katru iespējamo vietu tekstā un pārbaudām, vai virkne tur patiešām ir.

```
for i = 0 to n - m
{
      j = 0
      while (T[i + j] = P[j] and j < m) j = j + 1
      if j = m print(i);
}
```

Noliekam blakus un sākam salīdzināt
```
T[i]T[i + 1] ...
P[0]P[1] ...
```
Ja šādā veidā sakrīt m elementi, tad virkni esam atraduši.

Ātrdarbība

O (n · m) sliktākajā gadījumā.
Ir n - m + 1 iespējamās vērtības priekš i. Katrai no tām var gadīties salīdzināt m simbolus.

Laiks

$$(n - m + 1) \cdot m \approx n \cdot m$$

Piemērs

$$T = \underbrace{aa \ldots a}_{n}$$

„sliktākais" gadījums

$$P = \underbrace{aa \ldots a}_{m}$$

Algoritmu var modificēt, lai tas apstātos, kad ir atradis pirmo rezultātu. Bet arī šajā gadījumā darbības laiks var būt tikpat liels, piemēram, šīm divām virknēm:

$$T = \underbrace{aa \ldots a}_{n}$$

$$P = \underbrace{aa \ldots ab}_{m}$$

<u>Knuta - Morisa - Prata algoritms</u>

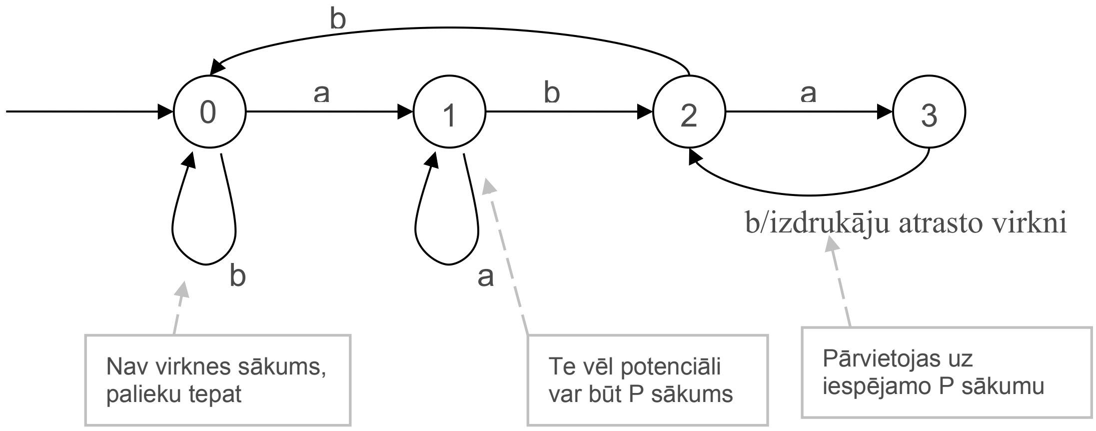
Ja mēs zinām, ka

$$T[i] = P[0], \dots T[i + j - 1] = P[j - 1],$$
bet
$$T[i + j] \neq P[j],$$

tad to var izmantot, lai izvēlētos nākošo i.

<u>Meklēšana ar galīgu automātu</u>

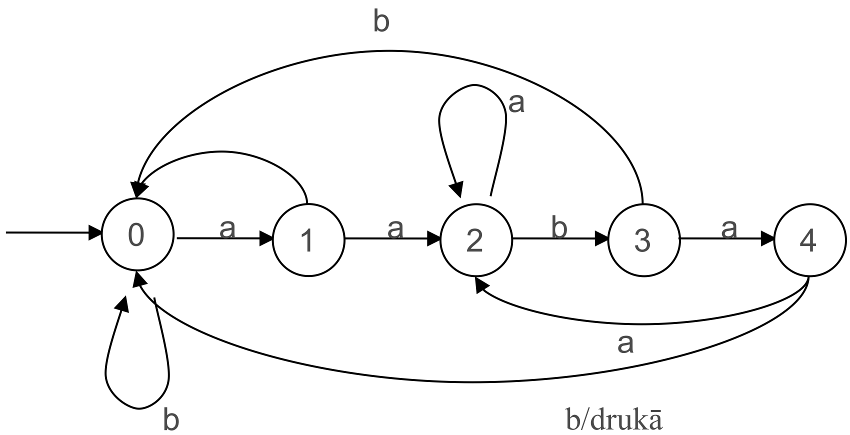
Stāvokļi 0, 1 ... m -1
i - „pēdējie i simboli no T sakrīt ar pirmajiem i simboliem no P”.
P = abab



ab	ab
	ab

<u>Uzdevums</u>

Uzzīmēt galīgu automātu virknei aabab.



Algoritma darbības laiks

Automātam O(n), katram burtam viena operācija.
Automāta izveidošanai:

- Jāizrēķina nākošais stāvoklis q' jebkurai pašreizējā stāvokļa q un pašreizējā burta kombinācijai.
- Mums ir m stāvokļi. Ar A apzīmējam burtu skaitu.
- Tad sanāk tabula ar $m \cdot A$ elementiem.
- $\geq m \cdot A$ operācijas.

KMP algoritms

Ja

$$T[i + j] \neq P[j],$$

ignorē P[j].

Priekšapstrāde:

Katram j (j = 1 ... m) izrēķina maksimālo k, kur k < j un izpildās šādi nosacījumi:

$$P[0] = P[j - k]$$
$$P[1] = P[j - k + 1]$$
$$\dots$$
$$P[k - 1] = P[j - 1]$$

To noglabā masīvā k, kā vērtību k[j].
(Ja nosacījumi neizpildās nevienam k<j, tad k[j]=0.)

Gadījumā, ja T pašreiz salīdzināmais simbols nesakritīs ar P[j], mums būs šāda situācija.

T[i]	T[i+1]		T[i+k-1]
=	=		=
P[j-k]	P[j-k+1]	...	P[j - 1]
=	=		=
P[0]	P[1]		P[k - 1]

Tad mēs zināsim, ka nākošā pozīcija tekstā T, kur var parādīties apakšvirkne P, ir sākot ar pēdējiem k burtiem no jau nolasītā T gabala.

Izmantojot mūsu iepriekš izveidoto masīvu k, KMP algoritms strādā šādi:

```
i = 0; j = 0;
while(i <= n - m)
{
      while(T[i + j] = P[j] and j < m)
      {
            j = j + 1;
      }
      if(j = m) print(i);
      if(j = 0)
```

```
            i = i + 1;
      else
      {
            i = i + j - k[j];
            j = k[j];
      }
}
```

Ātrdarbības novērtējums

Kad masīvs k jau ir uzbūvēts.
Var ievērot, ka uz katru salīdzināšanu:

- ja salīdzināmie simboli ir vienādi – tiek palielināts (*i*+*j*) (un *i* atstāts tāds pats)
- ja salīdzināmie simboli atšķiras – tiek palielināts *i* (un (*i*+*j*) atstāts tāds pats)

Tā kā *i* un *i*+*j* ir veseli skaitļi, abi sākumā ir 0 un nevar pārsniegt *n*, tad katru var palielināt kopā būs ne vairāk kā 2*n* salīdzināšanas. Tātad ātrdarbība O(*n*).

Kāpēc algoritms strādā pareizi?

Viegli redzēt, ka, ja tiek izdrukāts i, tad tik tiešām T[i], T[i+1], ..., T[i+m] sakrīt ar apakšvirkni P[0], P[1], ..., P[m-1].
Algoritms darbojas, mēģinot „pielikt" apakšvirkni P jebkurā teksta T pozīcijā. Ja ir nesakritība, tad apakšvirknes sākums tiek pārbīdīts uz priekšu uz tuvāko nākamo pozīciju, kur tas potenciāli varētu atrasties.
...

Piemērs:

Ja mēs meklējam apakšvirkni abab, tabula k ir šāda:

j	k[j]
1	0
2	0
3	1
4	2

j = 1 vai j= 2 => nosacījumi neizpildās;

j = 3 => $\begin{array}{r|l} aba & \\ \hline a & ba \end{array}$

izpildās nosacījumi pie k = 1;

j = 4 => $\begin{array}{r|l} ab & ab \\ \hline & ab \end{array}$

ir sakritība garumā 2.

Kad kaut kas nesakritīs, vārda sākumu i pabīdīsim uz priekšu par j-k[j] un sāksim salīdzināšanu no P[k[j]].

Tabula apakšvirknei aabaab:

j	k[j]
1	0
2	1
3	0
4	1
5	2
6	3

j=1 => k[1] = 0

a	
	a

j=2 => k[2] = 1

a	**a**	
	a	a

j=3 => k[3] = 0

a	a	b			
			a	a	b

j=4 => k[4] = 1

a	a	b	**a**			
			a	a	b	a

j=5 => k[5] = 2

a	a	b	**a**	**a**			
			a	**a**	b	a	a

j=6 => k[6] = 3

a	a	b	**a**	**a**	**b**			
			a	**a**	**b**	a	a	b

Tabulas k būvēšana

```
k[0] = -1;
i = 0;
while (i < m) {
      k[i + 1] = k[i] + 1;
      while (k[i + 1] > 0 && P[i] != P[k[i + 1] - 1])
            k[i + 1] = k[k[i + 1] - 1] + 1;
      i = i + 1;
}
```

Ātrdarbības novērtējums

Ārējais cikls izpildās *m* reizes. Katrā iekšējā cikla iterācijā *k*[*i*+1] vērtība tiek samazināta. Tā kā šī vērtība tiek palielināta tikai katrā ārējā cikla iterācijā par 1, tad tā var sasniegt ne vairāk kā *m*. Tā kā tā nevar būt negatīva, tad samazināta tā var tikt arī ne vairāk kā *m* reizes, tātad iekšējais cikls kopumā izpildās ne vairāk kā m reizes. Tātad masīva būvēšanai ātrdarbība O(m).