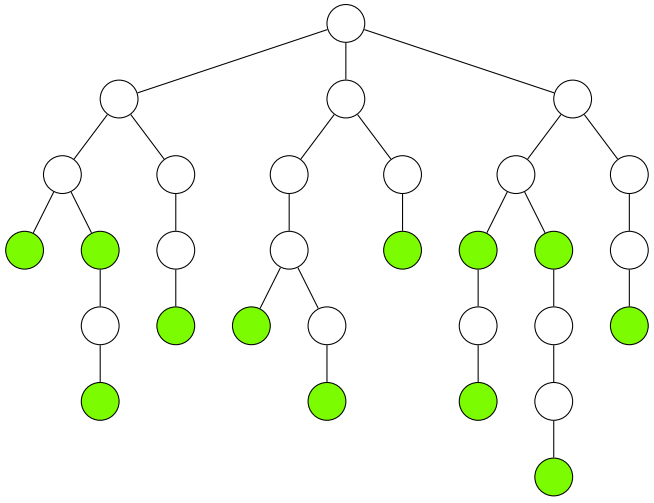
Trie
Algoritmul Knuth–Morris–Pratt
Aplicații pe Stringuri

Iulian Oleniuc



Cuprins

Capitolul 1

Structura de date trie

În acest prim capitol voi prezenta structura de date numită *trie*, precum și câteva aplicații interesante ale acesteia.

Tria este o structură de date arborescentă, ușor de implementat, folosită pentru a stoca un set (dinamic) de cuvinte într-o manieră compactă. Din acest motiv, putem spune că tria este o structură de date de tip *dicționar*.

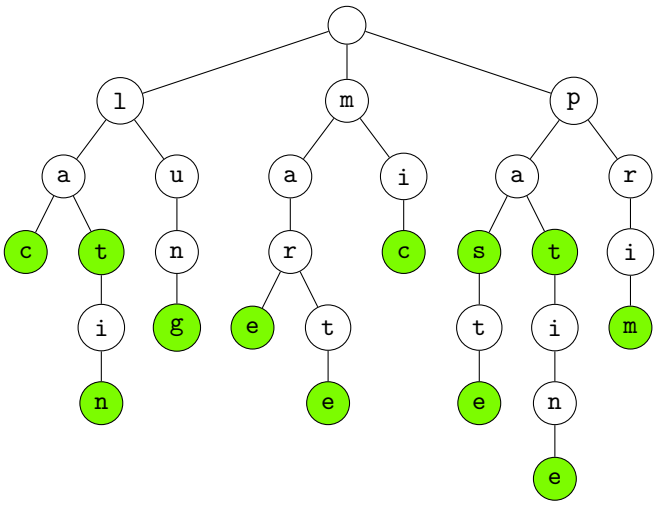
Avantajul principal al acesteia este complexitatea căutării unui cuvânt, care este liniară în lungimea sa, nedepinzând deci de mărimea dicționarului. De aici și numele structurii — *trie* provine de la *retrieval*.

Vom nota cu Σ lungimea alfabetului cu care lucrăm, de obicei 26 (de la alfabetul englez).

1.1 Structura unei trie

O trie este un arbore în care fiecare nod are cel mult Σ fii — câte unul pentru fiecare literă din alfabet. Cuvintele stocate în trie pot fi reconstituite prin citirea caracterelor aflate pe lanțul de la rădăcină la diverse noduri, numite *frunze*.

Iată mai jos o trie construită pe baza cuvintelor `lac`, `lat`, `latin`, `lung`, `mare` etc. Nodurile frunză sunt marcate cu verde.



Am ales să pun literele în *noduri* pentru a desena mai ușor tria, însă conceptual este mai bine să vă gândiți că literele sunt asociate *muchiilor*. Astfel, nu mai este nevoie să-i asociem rădăcinii caracterul nul.

De remarcat că un nod **nu** poate avea doi fii corespunzători aceleiași litere. De asemenea, putem observa cu ușurință cum, în contextul unei trie, conceptul de *frunză* nu este același cu cel de *frunză a unui arbore*.

Astfel, orice frunză a arborelui este și frunză în trie, însă nu și invers. Nodul corespunzător cuvântului s este frunză a triei, dar nu și a arborelui, doar atunci când s este prefixul unui alt cuvânt t din trie.

1.2 Implementare în linii mari

În această secțiune ne vom axa pe implementarea clasică a triei în C++. Astfel, vom scrie o clasă numită `Trie`, capabilă să efectueze operațiile de inserare, căutare, ștergere și LCP (longest common prefix).

Cu toate acestea, merită menționat că, în majoritatea problemelor, funcțiile `insert` și `count` sunt suficiente. De asemenea, trebuie avut în vedere că uneori putem avea nevoie și de funcții *mai puțin standard*, cum ar fi cea de compresie, care apare în problema CLI.

1.2.1 Definirea clasei `Trie`

Momentan, clasa `Trie` are nevoie de doar doi membri: *cnt* (numărul de cuvinte care se termină în nodul curent) și un vector de pointeri *next*, de lungime Σ, unde $next[i]$ ne spune în ce nod ne ducem dacă urmăm legătura corespunzătoare celei de-a i-a litere din alfabet. Atunci când $next[i]$ este `nullptr`, înțelegem că nu există tranziție către litera i.

```
class Trie {
    int cnt = 0;
    Trie *next[26] = {};
};
```

1.2.2 Funcția `insert`

Această funcție, la fel ca și următoarele, primește ca parametri un string *str* și poziția curentă *pos* din acesta.

Funcția `insert` inserează recursiv cuvântul *str* în trie, astfel: Dacă poziția curentă este egală cu lungimea stringului, înseamnă că ne-am terminat treaba, așa că incrementăm *cnt*-ul din nodul curent și ne oprim.

Altfel, urmăm muchia către litera curentă $str[pos]$ și continuăm inserarea din nodul următor, începând cu poziția $pos + 1$. Înainte de aceasta, avem grijă să *creăm* nodul $next[str[pos]]$, în caz că nu există deja.

Iată și o animație care ilustrează seria de inserări necesare pentru a obține tria de mai devreme: link.

```
void insert(const string& str, int pos = 0) {
    if (pos == int(str.size()))
        cnt++;
    else {
        if (!next[str[pos] - 'a'])
            next[str[pos] - 'a'] = new Trie;
        next[str[pos] - 'a']->insert(str, pos + 1);
    }
}
```

1.2.3 Funcția `count`

Funcția `count` returnează numărul de apariții ale cuvântului *str* în trie. Pentru a face asta, trebuie ca mai întâi să ajungem în nodul în care se termină cuvântul dat, iar apoi să returnăm *cnt*-ul din acesta.

Așadar, funcția `count` va semăna foarte mult cu `insert`, numai că aici, dacă nu există tranziție din nodul curent către litera următoare, returnăm 0 și ne oprim, concluzionând că *str* nu se găsește în trie.

```
int count(const string& str, int pos = 0) {
    if (pos == int(str.size()))
        return cnt;
    if (!next[str[pos] - 'a'])
        return 0;
    return next[str[pos] - 'a']->count(str, pos + 1);
}
```

1.2.4 Funcția `lcp`

Funcția `lcp` trebuie să returneze lungimea celui mai lung prefix comun al lui *str* cu un cuvânt oarecare din trie.

Coborâm în trie până când fie ne blocăm într-un nod fără tranziție către următoarea literă, fie ajungem la finalul stringului. Ce returnăm de fapt este lungimea lanțului de la rădăcină până la nodul în care ne oprim.

```
int lcp(const string& str, int pos = 0) {
    if (pos == int(str.size()))
        return 0;
    if (!next[str[pos] - 'a'])
        return 0;
    return 1 + next[str[pos] - 'a']->lcp(str, pos + 1);
}
```

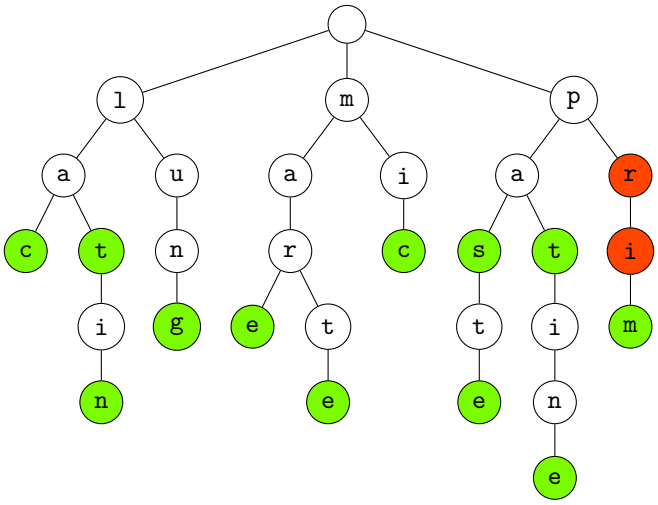
1.2.5 Funcția `erase`

Funcția `erase` șterge din trie o apariție a cuvântului *str*. Se garantează că *str* apare cel puțin o dată în trie înainte să-i dăm `erase`.

Fie *node* nodul în care se termină cuvântul *str*. Pentru ca *node* să poată fi șters, trebuie ca numărul de cuvinte care se termină în subarborele cu rădăcina în el să fie exact 1, adică doar cel pe care vrem să-l ștergem — chiar *str*.

De exemplu, în tria de mai jos **nu** putem șterge nodul în care se termină cuvântul `lat`, pentru că sub el se află frunza corespunzătoare lui `latin`. Altfel spus, `lat` este un prefix al lui `latin`.

În schimb, putem șterge nodul în care se termină `prim`. De fapt, putem șterge și din strămoșii lui `prim`, mai exact pe `pri` și pe `pr`. Nodul `p` în schimb nu poate fi șters, deoarece este prefix al mai multor cuvinte stocate în trie, nu doar al lui `prim`.



Să mai adăugăm un membru privat la clasa `Trie`, notat cu *lvs* (de la *leaves*), care să reprezinte suma valorilor *cnt* ale nodurilor din subarborele nodului curent.

Această valoare poate fi menținută fără mari bătăi de cap, incrementând-o când vizităm nodul curent în cadrul unei inserări și decrementând-o când efectuăm o ștergere.

Acum că avem calculată această informație, ne este mai ușor să rezumăm ce trebuie să facem concret în funcția `erase`: Coborâm în trie până în nodul în care se termină *str*, decrementând valorile *lvs* întâlnite pe parcurs. După ce ajungm în *node*, îl decrementăm pe *cnt* și ne întoarcem în rădăcină pe același traseu, ștergând nodurile al căror *lvs* a devenit 0.

```
void erase(const string& str, int pos = 0) {
    lvs--;
    if (pos == int(str.size()))
        cnt--;
    else {
        next[str[pos] - 'a']->erase(str, pos + 1);
        if (!next[str[pos] - 'a']->lvs) {
            delete next[str[pos] - 'a'];
            next[str[pos] - 'a'] = nullptr;
        }
    }
}
```

1.2.6 Implementare alternativă

Există și o implementare mai puțin populară, în care nodurile sunt ținute într-un vector alocat dinamic, pointerii sunt de fapt indici în acest vector, iar funcțiile sunt implementate iterativ.

Dezavantajul este că această metodă de stocare a nodurilor nu ne permite să eliberăm *efectiv* memorie în urma unui `erase`. Nodurile sunt eliminate doar conceptual, prin ștergerea muchiei către primul nod dintre rădăcină și *node* în care $lvs = 0$.

1.3 Complexitate

În cazul ambelor implementări, funcția `insert` (la fel ca și celelalte trei de mai sus) are complexitatea $\mathcal{O}(n)$, unde n este lungimea stringului dat.

Partea mai puțin bună este consumul relativ mare de memorie — pentru fiecare nod reținem Σ pointeri (sau indici), dintre care majoritatea nu sunt folosiți. Iar asta se reflectă în complexitatea unui DFS, care are constanta Σ.

Putem reduce această constantă la 1 înlocuind vectorul *next* cu un `map`. Însă, dacă facem asta, complexitatea lui `insert` devine $\mathcal{O}(n \log \Sigma)$. Evident, putem scăpa de log dacă recurgem la un `unordered_map`, însă pierdem posibilitatea de a itera fiii unui nod în ordine alfabetică — lucru uneori necesar. Alegerea depinde așadar de problema dată.

1.4 Aplicații clasice

În cele ce urmează, vom discuta despre trei aplicații simple ale triei, precum și despre structurile de date inspirate de aceasta.

1.4.1 Sortarea unei liste de cuvinte

Mai întâi, construim o trie din cuvintele date, reținând în fiecare frunză o listă cu indicii celor care se termină în nodul respectiv. Apoi, efectuăm o parcurgere preordine a triei, afișând indicii întâlniți pe parcurs.

Putem adapta algoritmul și pentru sortarea unui vector de *întregi*, inserând în trie reprezentările lor în baza 2 de exemplu. Chiar am obține o complexitate bună: $\mathcal{O}(n \log \max_i\{v_i\})$. Însă, din cauza consumului mare de memorie auxiliară și a faptului că aceasta nu este o *sortare prin comparare*, poate fi considerată impractică.

1.4.2 Funcția de autocomplete

Aici mă refer la căutarea de cuvinte ce încep cu un prefix dat, adică ceea ce face Google atunci când ne oferă sugestii de căutare. Navigăm în trie până la nodul în care se termină prefixul acela, după care enumerăm frunzele din subarborele acelui nod.

1.4.3 Cel mai lung prefix comun a două cuvinte

Să zicem că avem dat un set de cuvinte și trebuie să răspundem la multe întrebări de forma: "Care este lungimea LCP-ului cuvintelor x și y?" Mai întâi, construim o trie peste cuvintele date. Apoi, problema se reduce la a determina adâncimea LCA-ului nodurilor în care se termină x și respectiv y.

1.4.4 Structuri de date similare

Tria este esențială în algoritmul Aho–Corasick, care rezolvă următoarea problemă: "Dându-se un string s și un dicționar d, să se determine, pentru fiecare cuvânt $x \in d$, numărul său de apariții în șirul s."

Totodată, tria stă la baza a două structuri de date mai avansate, Suffix Tree și Suffix Automaton, care ne ajută să rezolvăm în timp liniar probleme de genul: "Care este cea mai lungă subsecvență comună a stringurilor $s_1, s_2, \ldots, s_n$?"

1.5 Probleme

Ușor	Trie, SETI, Ratina
Mediu	Sub, A Lot of Games
Dificil	Xor Max, CLI

1.5.1 Problema Xor Max

Se dă un vector a de n numere naturale. Să se determine o subsecvență a sa cu suma xor maximă. Suma xor a unei subsecvențe $[l, r]$ este $a_l \oplus a_{l+1} \oplus \cdots \oplus a_r$, unde $\oplus$ reprezintă desigur operația xor. În caz că există mai multe soluții, se va alege secvența cu r-ul minim. Dacă în continuare există mai multe soluții, se va alege secvența de lungime minimă.

Fie $ps[i] \triangleq ps[1] \oplus ps[2] \oplus \cdots \oplus ps[i]$ a i-a sumă xor parțială a șirului a. În particular, cum elementul neutru al operației xor este 0, vom considera că $ps[0] \triangleq 0$. Din fericire, eleganta relație

$$a_{l+1} \oplus a_{l+2} \oplus \cdots \oplus a_r = ps[r] \oplus ps[l]$$

are loc și sub operația xor, deoarece $ps[l] \oplus ps[l] = 0$.

Acum, trebuie să vedem ce presupune să găsim cel mai bun indice l pentru un capăt-dreapta fixat r. Ne uităm la biții lui $ps[j]$. Dacă cel mai semnificativ bit al său este x, atunci vom alege, dacă se poate, ca primul bit al lui $ps[i]$ să fie $\neg x$, deoarece $x \oplus \neg x = 1$, iar asta clar ne garantează o sumă mai mare decât dacă am alege ca bitul curent să fie x, căci $x \oplus x = 0$.

Apoi, trecem la al doilea cel mai semnificativ bit y, iar dintre candidații de la pasul precedent îi vom da la o parte, dacă putem, pe cei cu al doilea bit egal cu y. Și așa mai departe.

Acest algoritm se pretează perfect pe o trie. Mai exact, o trie în care reținem reprezentările binare ale sumelor xor parțiale calculate până la pasul curent. Un detaliu de implementare ar fi că de fiecare dată când inserăm o nouă sumă în trie, va trebui ca după ultimul bit să punem și poziția din vector a sumei respective.

1.5.2 Problema CLI

Se dă un șir de n cuvinte și un întreg $k \leq n$. Avem un terminal în care putem tasta diverse cuvinte; fiecare operație (tastarea unei noi litere sau ștergerea literei curente — *backspace*) se efectuează la finalul stringului curent. Să se determine, pentru fiecare i de la 1 la k, costul minim necesar pentru a tasta i cuvinte oarecare dintre cele n date. Un cuvânt se consideră tastat dacă, la un moment dat, stringul curent din terminal a fost identic cu acesta. La finalul operațiilor, terminalul trebuie să fie gol.

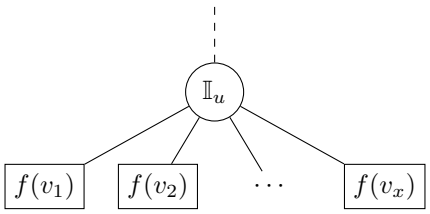
Intuitiv, cuvintele pe care le alegem trebuie să aibă cât mai multe prefixe comune, pentru a retasta (și șterge) cât mai puține cuvinte. Cum soluția este atât de strâns legată, în mod intrinsec, de ideea de prefix comun, ne gândim ca mai întâi să construim o trie peste cuvintele date.

În continuare, faptul că trebuie să rezolvăm problema pentru fiecare i din range-ul dat ne indică ideea că, dacă alegem să folosim programare dinamică, unul dintre parametri poate fi numărul de cuvinte tastate. Cum deja avem tria la dispoziție, ne gândim că celălalt parametru poate fi un nod din aceasta.

Acum putem defini o recurență cât de cât coerentă. În consecință, notăm prin $dp[node][i]$ costul necesar tastării a i cuvinte din primul moment în care terminalul conține cuvântul corespunzător lui *node* până în ultimul astfel de moment.

Pentru a oferi mai mult context, menționez că, odată ce avem selectată o *subtrie* cu rădăcina în tria principală și care conține i frunze, procedeul pentru tastarea celor i cuvinte aferente constă pur și simplu într-o parcurgere Euler a subtriei.

Relația de recurență în sine constă în aplicarea problemei rucsacului așa cum este ilustrat și mai jos. Nodul curent este u, iar fiii săi sunt $v_1, v_2, \ldots, v_x$. Am notat cu $f(v)$ numărul de cuvinte alese din subtria cu rădăcina în v, iar $\mathbb{I}_u \in \{0, 1\}$ ne indică dacă nodul u este o frunză sau nu. Așadar, relația asupra căreia trebuie să ne concentrăm în scrierea dinamicii este $f(u) = \mathbb{I}_u + \sum_i f(v_i)$.



Complexitatea acestei soluții este $\mathcal{O}(nk^2)$, însă nu ne garantează punctajul maxim. Pentru acesta, trebuie să observăm că putem calcula dinamica în mult mai puține noduri. Mai precis, doar în frunze. Astfel, fiecare nou cuvânt adăugat (conceptual) la soluție contribuie la costul final cu distanța de la el către prima frunză de pe drumul de la el la rădăcină.

Cu alte cuvinte, ce trebuie să facem este să *compresăm* tria (din nou, eventual, doar conceptual), astfel încât să conțină doar noduri la care se produc bifurcații. Acest lucru poate fi rezolvat ușor printr-un DFS pe tria principală sau chiar în timpul calculării dinamicii.

Capitolul 2

Algoritmul KMP

În acest capitol voi prezenta algoritmul Knuth–Morris–Pratt, din perspectiva automatelor finite. Mulți profesori aleg să nu menționeze legătura fundamentală între KMP și automate, însă aceasta va fi foarte utilă atunci când vom învăța despre algoritmi mai avansați, cum ar fi Aho–Corasick. În plus, automatele ne oferă o perspectivă vizuală mult mai clară asupra a ceea ce se întâmplă.

2.1 Automate finite deterministe

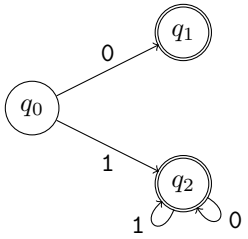
Pentru a înțelege cu adevărat ideea din spatele algoritmului KMP, trebuie mai întâi să înțelegem conceptul de *automat finit determinist* — un mecanism pentru recunoașterea limbajelor de diverse tipuri.

2.1.1 Definiție

Formal, un automat finit determinist este un tuplu $(Q, \Sigma, \delta, q_0, F)$, unde Q este mulțimea de stări, Σ este alfabetul de intrare, $\delta : Q \times \Sigma \rightsquigarrow Q$ este funcția de tranziție, q_0 este starea inițială, iar $F \subseteq Q$ este mulțimea stărilor finale.

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{\mathtt{0}, \mathtt{1}\} \\
\delta &= \{(q_0, \mathtt{0}) \to q_1, (q_0, \mathtt{1}) \to q_2\} \\
&\quad \cup \{(q_2, \mathtt{0}) \to q_2, (q_2, \mathtt{1}) \to q_2\} \\
q_0 &= q_0 \\
F &= \{q_1, q_2\}
\end{aligned}
$$

Pentru o intuiție mai bună, unui automat îi putem asocia un (multi)graf orientat. Mulțimea sa de noduri este Q, iar pentru fiecare relație $\delta(q_i, c) = q_j$ putem construi o muchie (q_i, q_j) cu eticheta c.



2.1.2 Algoritmul de recunoaștere

Rolul unui automat este să ne spună dacă un cuvânt peste alfabetul Σ, primit ca input, urmează un anumit pattern. Algoritmul pentru recunoașterea unui cuvânt s este foarte simplu: Se pornește din q_0 și se citește pe rând câte un caracter c din s. La fiecare pas, ne deplasăm din starea curentă urmând tranziția cu caracterul c. Adică, din q_i mergem în $\delta(q_i, c)$.

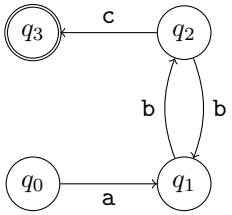
Dacă funcția δ nu este definită în (q_i, c), înseamnă că ne-am blocat în starea q_i înainte să procesăm tot inputul, așa că *rejectăm* cuvântul. În schimb, dacă am ajuns la sfârșitul inputului și ne aflăm într-o stare finală, *acceptăm* cuvântul și ne oprim.

Accept	`10010`	$q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \xrightarrow{0} q_2 \xrightarrow{1} q_2 \xrightarrow{0} q_2 \in F$
Reject	`011`	$q_0 \xrightarrow{0} q_1 \xrightarrow{1}$

Dacă un automat conține două tranziții din aceeași stare, cu aceeași literă, atunci acesta devine *nedeterminist*, pentru că, în timpul evaluării unui anumit cuvânt, nu va ști ce tranziție să urmeze din acea stare.

2.1.3 Exemple

Automatul de mai sus recunoaște cuvintele ce reprezintă numere scrise în baza 2. Întâi verifică cu ce cifră încep acestea. Dacă încep cu 0, avem grijă ca lungimea lor să nu depășească 1. În caz contrar, pot fi urmate de zero, una sau mai multe cifre, indiferent de valoarea lor.



Un alt exemplu este automatul de mai sus. Acesta recunoaște cuvintele de forma $\texttt{ab}^{2k-1}\texttt{c}$, cu $k \geq 1$, deoarece orice drum care duce la acceptarea inputului are forma

$$q_0 \xrightarrow{\texttt{a}} \underbrace{q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{b}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{b}} \cdots \xrightarrow{\texttt{b}} q_1 \xrightarrow{\texttt{b}} q_2}_{\text{de } k \geq 1 \text{ ori } q_1 \xrightarrow{\texttt{b}} q_2} \xrightarrow{\texttt{c}} q_3.$$

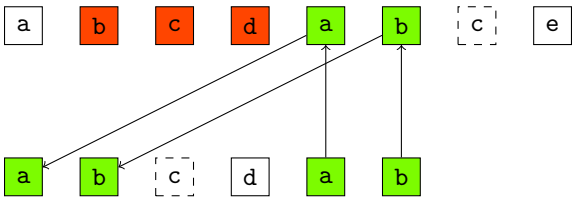
2.2 Motivație

Problema pe care dorim să o rezolvăm este cea în care primim două stringuri s (numit *text*) și t (numit *pattern*) și ni se cere să determinăm toate aparițiile lui t în s. Algoritmul KMP ne oferă o soluție în timp liniar, mai precis $\mathcal{O}(|s| + |t|)$, dar mai întâi vom analiza cum putem ajunge aici pornind de la soluția naivă.

Aceasta presupune, evident, să luăm fiecare poziție de la 0 la $|s| - |t|$ și să verificăm dacă aceasta reprezintă startul vreunui *match* cu t. Complexitatea este, desigur, $\mathcal{O}(|s| \cdot |t|)$.

Testarea unei singure poziții ca început de match este foarte costisitoare, însă, la o analiză mai atentă, ne dăm seama că multe comparații efectuate în cadrul ei ne pot ajuta și mai târziu.

Spre exemplu, dacă $s = \texttt{abcdabce}$ și $t = \texttt{abcdab}$, după ce am testat prima poziție și am găsit un match, putem observa două lucruri. În primul rând, este clar că putem sări peste următoarele trei poziții, pentru că nu ne oferă match nici măcar la prima literă. În al doilea rând, putem spune din prima că primele două litere de la următoarea poziție fac match, deoarece le-am comparat deja, la început. Așadar, am sărit peste cinci comparații!



Problema constă în formalizarea acestei idei. Deci, dacă ne-am blocat după ce am făcut match la primele i litere din pattern, ne interesează să știm care este cel mai lung sufix propriu (diferit de el însuși) al lui $t[0, i)$ care este totodată prefix al lui t. Vom considera că acest sufix din s este noul început al match-ului curent. În cazul de mai sus, sufixul respectiv este $\texttt{ab}$.

2.3 Funcția π

Așadar, avem nevoie ca pentru fiecare $i \in \{0, 1, \ldots, |t| - 1\}$ să cunoaștem lungimea maximă a unui sufix propriu al lui $t[0, i]$ care este și sufix al său (și implicit al lui t). Această valoare se notează de obicei cu $\pi(i)$, iar funcția π se mai numește funcție *prefix* sau *fail*. Iată un exemplu:

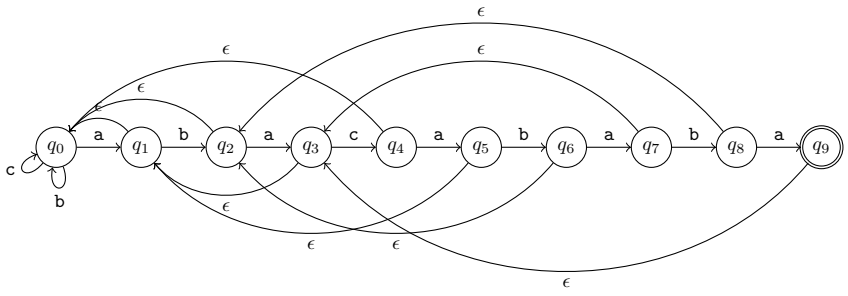
i	0	1	2	3	4	5	6	7	8
t	a	b	a	c	a	b	a	b	a
π	0	0	1	0	1	2	3	2	3

2.4 Automatul KMP

Înainte să ajungem la calculul eficient al funcției π, vom discuta puțin despre structura automatului asociat acesteia, care ne va ajuta să deducem mai ușor algoritmul.

Avem câte o stare q_i pentru fiecare poziție i din pattern, cu semnificația că, dacă ne aflăm în starea q_i, atunci am făcut match la primele i caractere din t. Tranzițiile inițiale sunt de forma $\delta(q_i, t[i]) = q_{i+1}$. Desigur, singura stare finală este $q_{|t|}$.

Restul tranzițiilor sunt generate de funcția π. Astfel, $\pi(i-1) = j$ se traduce prin $\delta(q_i, \epsilon) = q_j$, unde ϵ reprezintă stringul vid. În timpul parsării inputului, tranzițiile cu ϵ vor fi urmate doar atunci când nu avem tranziție cu caracterul curent din s! Evident, ϵ **nu** consumă niciun caracter din s.



Acum că fiecare stare q_i, cu $i > 0$, are câte o ϵ-tranziție, nu ne vom putea bloca niciodată în q_0 atunci când parsăm inputul. Însă ne vom bloca în q_0 atunci când $s[i] \neq t[0]$. Ca să rezolvăm asta, adăugăm tranzițiile $\delta(q_0, c) = q_0, \forall c \neq t[0]$.

Ideea de bază a ϵ-tranzițiilor, și implicit a funcției π, este că acestea ne duc în starea care face match cu cel mai mare prefix al lui t. Astfel, chiar dacă trebuie să abandonăm un match aproape complet, măcar suntem siguri că nu vom rata niciun alt match care s-ar fi suprapus cu acesta.

2.5 Calculul lui π în $\mathcal{O}(|t|)$

Cheia algoritmului optim pentru calculul funcției π este faptul că automatul poate fi construit în mod incremental folosindu-ne de el însuși (de ce am construit până la momentul curent).

La pasul i, mai întâi, creăm starea q_{i+1} și setăm $\delta(q_i, t[i]) = q_{i+1}$. Apoi, pentru a determina unde ne va duce tranziția cu ϵ, vrem să calculăm sufixul propriu maxim al lui $t[0, i]$ care este și prefix al lui t. Ei bine, acest string este cu siguranță un sufix propriu al lui $t[0, i)$, urmat de litera $t[i]$. În consecință, urmăm tranziția $q_j \triangleq \delta(q_{i-1}, \epsilon)$ și încercăm să efectuăm o tranziție cu litera $t[i]$. Dacă aceasta nu există, repetăm recursiv procedeul mergând în $\delta(q_{j-1}, \epsilon)$ și așa mai departe.

```
auto getPi(const string& str) {
    vector<int> pi(str.size());
    int k = 0;
    for (int i = 1; i < int(str.size()); i++) {
        while (k && str[i] != str[k])
            k = pi[k - 1];
        if (str[i] == str[k])
            k++;
        pi[i] = k;
    }
    return pi;
}
```

Complexitatea acestui algoritm se amortizează la $\mathcal{O}(|t|)$, însă demonstrația depășește cadrul cursului.

2.6 Aplicații clasice

În cele ce urmează, vom discuta despre trei aplicații simple ale funcției π, una dintre ele fiind chiar algoritmul KMP. La acestea se adaugă, din nou, algoritmul Aho–Corasick — extinderea KMP-ului la mai multe patternuri.

2.6.1 String matching

Să ne întoarcem la problema inițială — găsirea tuturor aparițiilor patternului t în textul s. Ei bine, odată calculată funcția π, deci implicit și automatul KMP, nu ne rămâne decât să trecem stringul s prin acesta, adică să-l dăm ca input algoritmului de recunoaștere.

De fiecare dată când ajungem în starea finală contorizăm o nouă apariție a lui t în s (care se termină pe poziția curentă) și **continuăm** căutarea. Algoritmul este aproape identic cu cel al calculării lui π.

```
auto kmp(const string& str, const string& pat) {
    auto pi = getPi(pat);
    vector<int> occ;
    int k = 0;
    for (int i = 1; i < int(str.size()); i++) {
        while (k && str[i] != pat[k])
            k = pi[k - 1];
        if (str[i] == pat[k])
            k++;
        pi[i] = k;
        if (k == int(pat.size()))
            occ.push_back(i - k + 1);
    }
    return occ;
}
```

Există însă o soluție mai elegantă, care ne scapă de nevoia de a duplica codul. Aceasta constă în a calcula funcția π pe stringul $t \parallel \# \parallel s$, unde # este un caracter *delimitator*, care nu apare nici în s și nici în t. Astfel, limităm valoarea maximă a lui π în pozițiile corespunzătoare lui s la $|t|$. Ei bine, pozițiile unde întâlnim această valoare sunt chiar cele care fac match complet.

```
auto kmp(const string& str, const string& pat) {
    const string concat = pat + '#' + str;
    auto pi = getPi(concat);
    vector<int> occ;
    for (int i = pat.size() + 1; i < int(concat.size()); i++)
        if (pi[i] == int(pat.size()))
            occ.push_back(i - 2 * int(pat.size()));
    return occ;
}
```

2.6.2 Frecvența fiecărui prefix

Să se afle, pentru fiecare prefix al lui t, care este numărul său de apariții în t.

Fiecare poziție i reprezintă sfârșitul unui match chiar cu $t[0, i]$, dar asta înseamnă implicit și un match cu prefixul $\pi(i)$, unul cu $\pi(\pi(i))$ etc.

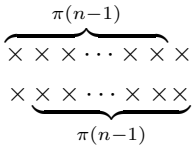
Putem calcula eficient aceste frecvențe ca mai jos. Rezultatele se vor afla pe pozițiile $1, 2, \ldots, n$.

```
vector<int> ans(n + 1);
for (int i = 1; i <= n; i++)
    ans[i]++;
for (int i = n - 1; i > 0; i--)
    ans[pi[i]] += ans[i];
```

2.6.3 Perioada minimă a unui string

Un string t' se numește *perioadă* a lui t dacă t poate fi obținut prin concatenarea lui t' cu el însuși de un anumit număr de ori. Să se determine perioada minimă a lui t.

Observăm că, dacă t' este o perioadă a lui t, atunci fie $t' = t$, fie $\pi(n-1) \geq n/2$. Dacă, pe deasupra, $n - \pi(n-1) \mid n$, atunci avem garanția că $t[0, n-\pi(n-1))$ chiar este perioada minimă a lui t.



2.7 Probleme

- Potrivirea Șirurilor
- Sufixe
- Prefixes and Suffixes
- Password
- Zlego