

Trie

Să pornim de la următoarea problemă: fie o mulțime A de cuvinte formate din litere mici ale alfabetului latin (în continuare vom numi această mulțime un dicționar) și o serie de interogări de forma: fiind dat un cuvânt B de lungime L, să se spună dacă acest cuvânt apare sau nu în mulțimea dicționar.

Soluția 1

Soluția cea mai evidentă este aceea ca pentru fiecare interogare să comparăm cuvântul B cu fiecare cuvânt din mulțimea A. Complexitatea acestui algoritm ar fi $O(L * |A| * \text{NrInt})$, unde $|A|$ reprezintă cardinalul mulțimii A, iar NrInt numărul de interogări.

Deși această soluție este una corectă (furnizează tot timpul răspunsul corect) dorim să găsim un algoritm de o complexitate mai bună.

Soluția 2 (trie-uri)

Soluția eficientă folosește o structură de date numită în literatura de specialitate **trie**. Trie-ul este de fapt un arbore în care fiecare nod are un număr de fii egal cu dimensiunea alfabetului folosit (fiecare muchie este etichetată cu o literă a alfabetului).

Astfel fiecare nod din trie va reprezenta un sir, format din concatenarea etichetelor muchiilor de la rădăcină până la nodul curent. Definim un nod ca fiind terminal dacă șirul reprezentat de acesta apare în dicționarul A (nu orice nod al trie-ului este nod terminal; șirul asociat unui nod poate fi doar un prefix al unui cuvânt din A).

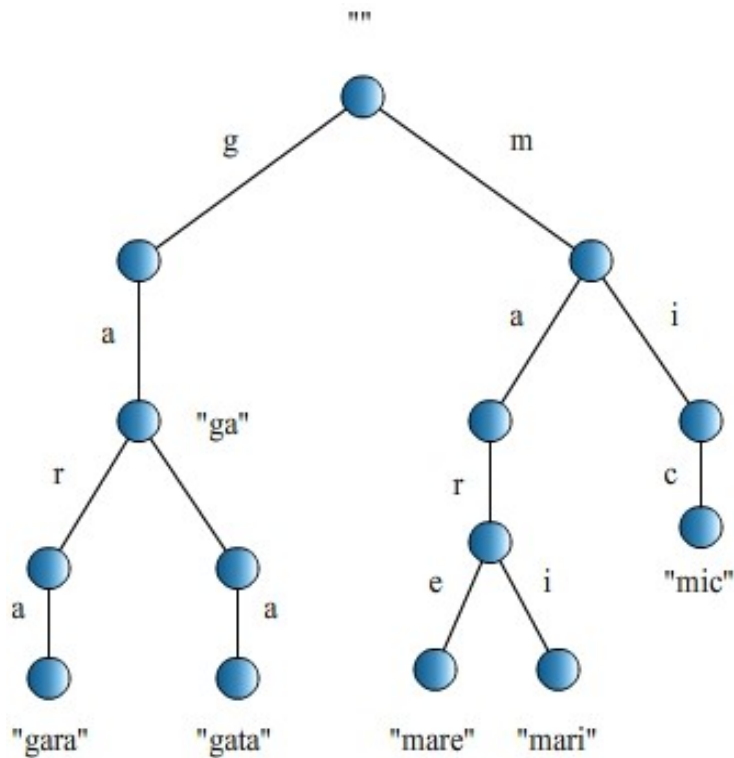
Revenind la problema formulată, aceasta poate fi rezolvată acum în modul următor: citim dicționarul A și formăm un trie cu toate cuvintele acestuia. Acum pentru fiecare interogare vom porni de la rădăcina trie-ului și vom merge în jos (dacă avem un nod cu adâncimea x, vom merge pe muchia etichetată cu caracterul corespunzător poziției $x + 1$ din șirul căutat). Dacă la final (când am ajuns la adâncimea L), nodul curent este un nod terminal, atunci șirul B apare în dicționarul A, altfel nu.

După cum se poate observa pentru fiecare cuvânt interogat vom face L iterații în trie, așadar complexitatea algoritmului este $O(L * \text{NrInt})$, semnificativ mai mică decât cea din soluția precedentă.

Legat de memoria ocupată de trie, aceasta de variabilă, ea depinzând de cuvintele din dicționar (mai precis de prefixele comune ale acestora). Totuși aceasta nu poate depăși $O(\text{LungTot} * |\text{sigma}|)$, unde LungTot reprezintă lungimea tuturor cuvintelor din A, iar sigma dimensiunea alfabetului.

Exemplu

În următoarea figură este reprezentat trie-ul asociat dicționarului {"ga", "gara", "gata", "mare", "mari", "mic"}.



Iar aici se pot vedea operațiile de inserare și căutare (scrise în C++), cât și modul de declarare al unui trie:

```
struct trie{
    int end; //aici retinem daca nodul curent este terminal sau nu
    /* aici se poate retine orice alta informatie care se doreste sa se stocheze intr-un nod*/

    trie *fui[26]; //pointeri catre cei 26 de fii

    trie(){//initializari
        end = 0;
        memset(fui, 0, sizeof(fui));
    }
};

void insert(trie * q, int poz){
    if (poz == L){
        q → end ++; //incrementam numarul de cuvinte care se termina in nodul curent
```

```

        return ;
    }

    int next = sir[poz + 1] - 'a';
    if (!(q → fiu[next]))
        q → fiu[next] = new trie; //daca nu exista un fiu cu muchia etichetata cu
        caracterul next il cream

    insert(q → fiu[next], poz + 1);
}
bool find(trie *q, int poz){
    if (poz == L){
        if (q → end > 0)
            return 1; //nodul curent este nod terminal (se termina cel putin un cuvant)
        return 0; //nu este nod terminal
    }

    next = B[poz + 1] - 'a';
    if (!(q → fiu[next]))
        return 0; //nu mai putem sa ne urmam drumul => cuvantul cautat nu se gaseste

    return find(q → fiu[next], poz + 1);
}

```

Aplicații în concursuri de informatică

Această structură de date se întâlnește destul de des în concursurile de informatică, atât în România, cât și în alte țări.

În continuare sunt prezentate câteva probleme ce se pot rezolva cu trie.

Pentru antrenament, dacă până acum nu s-a mai implementat niciodată această structură de date se recomandă rezolvarea problemei Trie din arhiva educațională de pe infoarena.ro (www.infoarena.ro/problema/trie).

1. Toponyms (Balcaniada de Informatica 2007)

Enunț: [http://docs.google.com/Doc?](http://docs.google.com/Doc?docid=0AacqIqq39W_WZGczOTRzOG1fMTB4amhwemJkaA&hl=en)

[docid=0AacqIqq39W_WZGczOTRzOG1fMTB4amhwemJkaA&hl=en](http://docs.google.com/Doc?docid=0AacqIqq39W_WZGczOTRzOG1fMTB4amhwemJkaA&hl=en)

Dat fiind că problema se leagă de prefixele cuvintelor din fișierul de intrare vom construi un trie pornind de la acestea.

Acum când ne aflăm într-un nod în trie putem afirma că șirul asociat celui nod (cel

format de muchiile de la rădăcină până la el) este cu siguranță prefix pentru toate nodurile terminale din subarborele nodului curent.

Astfel, după ce am terminat de construit trie-ul, îl parcurgem, iar pentru fiecare nod vom calcula adâncimea curentă * numărul de noduri terminale din subarbore. Răspunsul la problema noastră este maximul dintre toate aceste valori (adică valorile calculate pentru toate nodurile trie-ului).

Un exemplu de cod (funcția de căutare):

```
void caut(trie *q, int ad){

    for (int i = 0; i < 53; i ++){
        if (q -> fiu[i]){
            caut(q -> fiu[i], ad + 1);

            q -> x += (q -> fiu[i]) -> x; //calculez numarul de noduri terminale din
subarborele curent
        }

        rsp = max(rsp, (lint)ad * (lint)(q -> x)); //iau maximul dintre toate valorile
    }
}
```

2. Ratina (ONI 2006)

Enunț: www.infoarena.ro/problema/ratina

Din nou, fiind vorba despre prefixele mai multor cuvinte, mai întâi le introducem pe toate într-un trie.

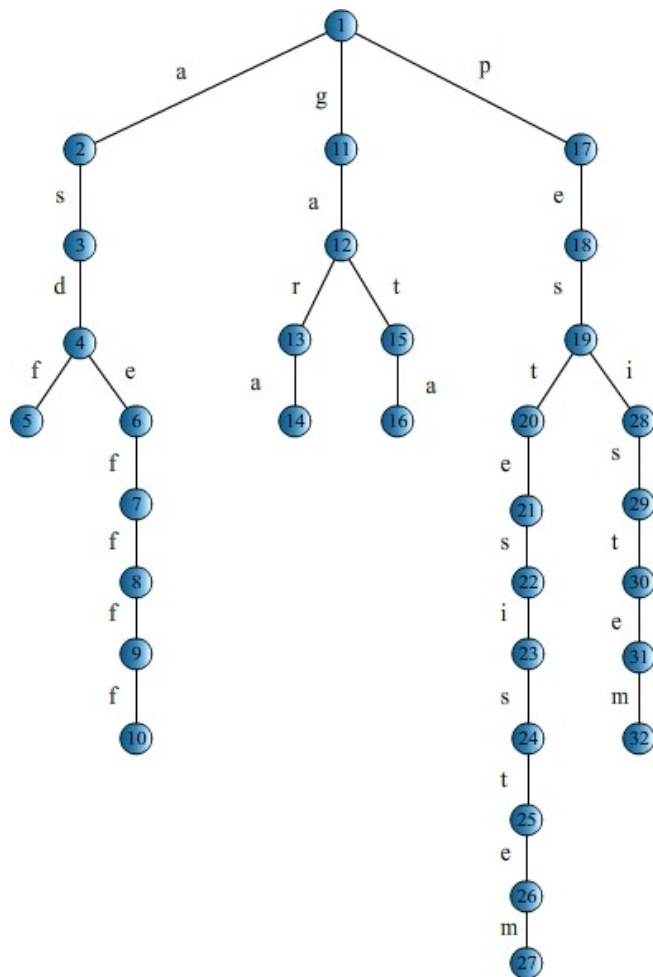
Se observă destul de ușor că pentru două cuvinte din trie cel mai lung prefix comun al lor este șirul asociat nodului care este lca-ul (lowest common ancestor) nodurilor corespunzătoare celor două cuvinte.

Generalizând, pentru n noduri din trie, cel mai lung prefix comun al acestora este lca-ul tuturor acestor noduri.

Așadar, pentru a răspunde la o întrebare este suficient să aflăm lca-ul nodurilor din trie corespunzătoare cuvintelor. Bineînțeles, cea mai directă metoda ar fi, pentru n noduri, să facem $n - 1$ operații de aflare al lca-ului (facem lca între primele 2, apoi între lca-ul rezultat și al treilea, ș.a.m.d). O optimizare a metodei de mai sus ar fi următoarea: se parcurge trie-ul și pentru fiecare nod se reține timpul de intrare în parcurgerea în adâncime. Se observă că lca-ul dintre cele n noduri este același cu lca-ul dintre nodul cu timpul de intrare minim și cel cu timpul de intrare maxim (figurativ vorbind, cel mai din stânga și cel mai din dreapta nod). Astfel este necesar doar să aflăm nodul minim și cel maxim și să afișăm lca-ul dintre cele două.

Un exemplu de cod (doar răspunsul la interogări):

```
for (i = 1; i <= m; i++){  
    scanf("%d", &nr); mn = 0x3f3f3f3f, mx = 0;  
  
    for (j = 1; j <= nr; j++){  
        scanf("%d", &x);  
  
        mn = min(mn, poz[x]); mx = max(mx, poz[x]); //poz[x] retine timpul de intrare  
        in parcurgerea in adancime a nodului x  
    }  
  
    printf("%d\n", lca(mn, mx));  
}
```



De asemenea, în figura alăturată este prezentat trie-ul asociat cuvintelor din exemplu (valoarea din fiecare nod reprezintă timpul de intrare al parcurgerii în adâncime).

3. T9 (Olimpiada Internationala Zhautykov 2010)

Enunț: http://docs.google.com/Doc?docid=0AacqIqq39W_WZGczOTRzOG1fMTJxZm5tanJmNw&hl=en

Pentru început vom citi toate cuvintele din dicționar, le vom converti în șiruri de cifre (corespunzătoare tastelor unui telefon), și le vom introduce într-un trie. Când va veni rândul să citim mesajul, pentru fiecare șir format din cifre vom efectua o căutare în trie (vom nota nodul găsit cu N).

Problema acum revine în a găsi al k -lea cuvânt ca număr de apariții din cele care se termină în nodul N , dar și de a putea modifica aceste valori. O soluție care să execute rapid aceste operații ar fi să ținem în fiecare nod din trie câte un arbore echilibrat (se poate folosi containerul *set* din STL), în care se va reține pentru fiecare cuvânt numărul de apariții ale sale.

Când căutăm cuvântul dorit este suficient să facem o simplă parcurgere a set-ului, iar pentru modificare vom șterge vechea valoare din set și vom introduce una nouă, cu numărul de apariții incrementat.

Mai jos sunt operațiile de introducere, căutare, cât și modul de declarare al trie-ului.

```
struct trie{

    trie *fiu[20];

    set< pair<int, int> > q; //set-ul din fiecare nod; retine doua componente: numarul de
    aparitii si indicele cuvintului

    trie(){
        memset(fiu, 0, sizeof(fiu));
    }

};

void baga(trie * tt, int nrm){
    if (nrm > lng){
        tt -> q.insert(mp(-val, mp(0, i)));
        return ;
    }

    int cur = caract[ (int)smic[i][nrm] ]; //caut cifra asociata caracterului cu indice nrm din
    cuvantul i din dictionar
    if (!(tt -> fiu[cur]))
        tt -> fiu[cur] = new trie;

    baga(tt -> fiu[cur], nrm + 1);
}
```

```

}
void find(trie * tt, int nrm){
    if (nrm > ind){
        gasit = tt;

        return ;
    }

    find(tt -> fiu[ sir[nrm] - '0' ], nrm + 1);
}

//citirea sirului, cautare in trie si modificarea valorilor

for (j = i, ind = 0; cit[j] != ' ' && cit[j] != '1' && cit[j] != '*' && cit[j] != '\n' && j <= m; j ++ )
    sir[++ ind] = cit[j];

if (ind > 0){
    find(tt, 1);
    it = gasit -> q.begin();

    for (; cit[j] == '*'; j ++ )
        it ++;

    printf("%s", smic[ it -> y.y ] + 1);

    x = it -> x, z = it -> y.y;
    gasit -> q.erase(it);

    gasit -> q.insert(mp(x - 1, mp(-i, z)));
}

```

4. Prefixes and suffixes (acm.sgu.ru)

Enunț: <http://acm.sgu.ru/problem.php?contest=0&problem=505>

Fiecare cuvânt din cele inițiale îl vom insera în două trie-uri, într-unul șirul inițial, iar în celălalt șirul inversat (scris de la dreapta la stânga). Vom numi cele două trie-uri pref și suf (unul reține prefixele, iar celălalt sufixele).

Când vom citi câte o interogare vom face câte o căutare în cele două trie-uri, o dată

pentru sufix și o dată pentru prefix (să presupunem că cele două noduri găsite sunt X și Y).

Este clar că toate nodurile terminale din subarborele lui X îl au ca prefix pe primul șir, la fel și pentru nodurile terminale din subarborele Y, care îl au ca sufix pe al doilea șir. Problema acum revine în a număra câte noduri terminale din primul subarbor se regăsesc în al doilea (mai precis, câte cuvinte inițiale citite de la stânga la dreapta se termină în subarborele lui X, și citite de la dreapta la stânga se termină în subarborele lui Y).

În continuare ne vine ideea să renumerotăm nodurile din cele două trie-uri în funcție de timpul de intrare și de ieșire din parcurgerea în adâncime a acestora. Astfel toate nodurile terminale din subarborele lui X vor avea indicele asociat între valorile $\text{first}[X][0]$ și $\text{first}[X][1]$ (unde $\text{first}[X][0]$ și $\text{first}[X][1]$ vor reprezenta timpii de intrare, respectiv de ieșire). De asemenea același lucru se aplică și pentru subarborele lui Y, adică nodurile terminale se vor afla între valorile $\text{second}[Y][0]$ și $\text{second}[Y][1]$.

Să presupunem că pentru fiecare cuvânt i din dicționarul inițial nodurile sale terminale din cele două trie-uri sunt memorate în vectorii *poz1* și *poz2*. Acum să ne imaginăm că fiecare cuvânt i este introdus într-un nou șir, pe poziția $\text{first}[\text{poz1}[i]][0]$ cu valoarea $\text{second}[\text{poz2}[i]][0]$. Cu alte cuvinte indicele din șir va fi timpul de intrare din trie-ul pref, valoarea din șir va fi timpul de intrare din trie-ul suf.

Acum când avem de făcut un query, trebuie să spunem câte elemente din sirul nou creat cu indicii între $\text{first}[X][0]$ și $\text{first}[X][1]$ au valorile asociate între $\text{second}[Y][0]$ și $\text{second}[Y][1]$. Aceasta problemă se poate rezolva relativ ușor cu un arbore de intervale într-o complexitate de $O(\log^2 N)$.

În continuare un exemplu de cod:

```
//am citit dictionarul si am introdus cuvintele in pref si suf
```

```
void fix(trie * q){ //in aceasta functie imi calculez timpii de intrare si iesire din fiecare nod
    ind++;
    q->x = ind;

    for (int i = 0; i < q->v.size(); i++)
        if (!s)
            first[ q->v[i] ] = ind;
        else{
            last[ q->v[i] ] = ind;

            scmb[ first[ q->v[i] ] ].pb(ind);
        }

    for (int i = 1; i <= 26; i++)
        if (q->fiu[i])
            fix(q->fiu[i]);

    q->y = ind;
}
```



```

void find(trie * q, int lun){//caut sirurile date pentru a imi stabili pozitiile pe care sa fac query
    if (lun > m){
        if (!s)
            p1 = q -> x, p2 = q -> y;
        else
            x = q -> x, y = q -> y;

        return ;
    }

    if (q -> fiu[ sir[lun] - 'a' + 1])
        find(q -> fiu[ sir[lun] - 'a' + 1], lun + 1);
    else
        stop = 1;
}

fix(pref);
s = 1; fix(suf);

scanf("%d\n", &teste);

for (tst = 1; tst <= teste; tst++){
    stop = 0;

    scanf("%s ", sir + 1); m = strlen(sir + 1);

    s = 0;
    find(pref, 1);

    scanf("%s\n", sir + 1); m = strlen(sir + 1);

    for (x1 = 1, x2 = m; x1 < x2; x1 ++, x2 --)
        swap(sir[x1], sir[x2]);

    s = 1;
    find(suf, 1);

    if (stop == 1)
        printf("0\n");
    else
        printf("%d\n", query(1, ind, 1));
}

```

Spre a se înțelege mai bine, în continuare se găsesc două ilustrații ale celor două trie-uri (cel albastru este de prefixe și cel roșu de sufixe). Perechile de valori de lângă fiecare nod sunt timpii de intrare și de ieșire din parcurgerea în adâncime, iar valorile din nodurile terminale ale trie-ului roșu reprezintă timpul de intrare din trie-ul albastru al șirului corespunzător nodului respectiv (evident, reprezentat în primul trie în ordinea de la stânga la dreapta).

Dicționarul folosit este {"aastr", "bactr", "actr", "batrr", "aapr", "bamt"}. Se poate încerca simularea algoritmului pe interogările ("aa", "tr"), ("a", "tr"), ("b", "r"), ("b", "tr"), ("aa", "r"), ("ba", "t").

