Potrivirea şirurilor

Capitolul 21

- Algoritmul ineficient
- Algoritmul Rabin-Karp
- Algoritmul Knuth-Morris-Pratt
- Rezumat
- Implementări sugerate
- Probleme propuse
- Soluțiile problemelor

În cadrul acestui capitol vom prezenta modul în care se poate realiza căutarea unui subşir într-un şir.

Vom considera un şir de lungime *N* şi vom verifica dacă acesta conține un subşir de lungime *M* ($M \leq N$). În cazul în care subşirul apare în şir, vom determina poziția la care începe acesta.

Vom începe cu un algoritm simplu, care va determina toate aparițiile subşirului, dar care este ineficient. Vom continua cu doi algoritmi mult mai rapizi şi anume algoritmul *Rabin-Karp* şi algoritmul *Knut-Morris-Pratt*.

21.1. Algoritmul ineficient

Cea mai simplă modalitate de verificare a apariție unui subşir într-un şir constă în parcurgerea şirului şi, pentru fiecare poziție a acestuia, compararea subşirului de lungime *M* care începe la poziția respectivă cu subşirul dat.

21.1.1. Prezentarea algoritmului

Datorită faptului că subşirul are lungimea *M*, acesta va trebui să înceapă la o poziție cuprinsă între 1 şi $N - M + 1$. Începând cu pozițiile mai mari decât $N - M + 1$ şirul inițial nu mai poate conține subşiruri de lungime *M*, deci subşirul nu poate apărea la aceste poziții.

Ca urmare, pentru fiecare poziție cuprinsă între 1 şi $N - M + 1$ vom verifica dacă, începând cu acea poziție, şirul dat conține aceleaşi elemente ca şi subşirul.

Varianta în pseudocod a acestui algoritm este următoarea:

```
Algoritm Potrivire(şir,subşir):
                                  { şir – şirul în care se caută subşirul }
                                         { subşir – subşirul căutat }
  N ← lungime(şir)
  M ← lungime(subşir)
  pentru i ← 1, N - M + 1 execută:
    ok ← adevărat
    j ← 0
    cât timp ok şi (j < M) execută:
      dacă şir[i+j] ≠ subşir[j+1] atunci
        ok ← fals
      sfârşit dacă
    sfârşit cât timp
    dacă ok atunci
      scrie 'Subşirul apare la poziţia ', i, '.'
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

Acest algoritm va funcţiona corect, indiferent de tipul şirului şi al subşirului. Aşadar, el poate fi aplicat indiferent de tipul elementelor conţinute de şir şi subşir, atâta timp cât şirul şi subşirul conţin acelaşi tip de elemente. Ca urmare, vom putea folosi acest algoritm pentru şiruri de caractere, şiruri de numere etc.

21.1.2. Analiza complexităţii

Pentru fiecare dintre cele $N - M + 1$ poziţii vom compara cel mult M elemente (de obicei mult mai puţine deoarece, în majoritatea cazurilor, vom detecta relativ repede faptul că subşirul nu apare la poziţia respectivă). Aşadar, ordinul de complexitate al unei verificări (pentru o anumită poziţie) este $O(M)$.

Datorită faptului că efectuăm $N - M + 1$ astfel de comparaţii, ordinul de complexitate al acestui algoritm este $O((N - M + 1) \cdot M) = O(M \cdot N - M^2 - M) = O(M \cdot N - M^2)$.

Se observă că algoritmul va fi foarte rapid dacă subşirul are lungimea relativ mică (conţine puţine elemente) sau relativ mare (conţine un număr de elemente apropiat de numărul de elemente al şirului).

Durata timpului de execuţie creşte pe măsură ce creşte lungimea subşirului, dar numai până la un moment dat, după care începe să scadă.

Cazurile nefavorabile apar atunci când lungimea subşirului este aproximativ egală cu jumătate din lungimea şirului. În această situaţie ordinul de complexitate ar deveni:

$$O\left(N \cdot \frac{N}{2} - \frac{N^2}{4}\right) = O\left(\frac{N^2}{2} - \frac{N^2}{4}\right) = O\left(\frac{N^2}{4}\right) = O\left(N^2\right).$$

Ca urmare, pentru cazul cel mai defavorabil, avem un algoritm pătratic.

21.2. Algoritmul Rabin-Karp

Algoritmul propus de *Rabin* şi *Karp* pentru potrivirea şirurilor foloseşte anumite noțiuni de teoria numerelor, cum ar fi egalitatea a două numere, modulo un al treilea.

Deşi, pentru cel mai nefavorabil caz algoritmul are acelaşi ordin de complexitate ca şi algoritmul prezentat anterior, pentru cazul mediu algoritmul *Rabin-Karp* este mult mai rapid.

În cadrul acestei secțiuni vom prezenta noțiunile teoretice care stau la baza algoritmului, vom descrie algoritmul şi îi vom analiza complexitatea.

21.2.1. Reprezentarea şirurilor

Algoritmul *Rabin-Karp* consideră fiecare element al şirului ca fiind o cifră într-o anumită bază. În cazul în care şirul conține cifre zecimale, vom lucra cu baza 10; dacă avem caractere ASCII, vom utiliza baza 256.

Se observă imediat o limitare importantă a acestui algoritm şi anume, faptul că numărul elementelor distincte ale şirului trebuie să fie relativ mic, pentru a putea alege o bază şi a codifica elementele în baza aleasă. Aşadar, dacă avem n elemente distincte, va trebui să lucrăm cu o bază $b \geq n$.

În cele ce urmează, pentru simplitate (dar fără a reduce generalitatea) vom presupune că şirurile conțin cifre zecimale, deci vom lucra cu baza 10.

Aşadar, fiecare şir care conține k caractere poate fi considerat a fi un număr zecimal cu k cifre (ignorăm situația în care primele caractere reprezintă cifra 0). De exemplu, şirul "2457" poate fi codificat prin numărul 2.457 (două mii patru sute cincizeci şi şapte).

21.2.2. Verificarea potrivirilor

Fiind dat un subşir P cu m elemente, vom nota prin p valoarea zecimală corespunzătoare subşirului. De asemenea, vom nota prin t_s valoarea zecimală corespunzătoare subşirului de lungime m din T care începe la poziția $s + 1$. Este evident faptul că vom avea $t_s = p$ dacă şi numai dacă subşirul P se regăseşte în şirul T la poziția $s + 1$.

Valoarea p poate fi calculată într-un timp $O(m)$ folosindu-se schema lui *Horner*:

$$p = P_m + 10 \cdot (P_{m\text{-}1} + 10 \cdot (P_{m\text{-}2} + \ldots + 10 \cdot (P_2 + 10 \cdot P_1)\ldots)).$$

Valoarea t_0 poate fi calculată în mod analog pe baza primelor m elemente ale şirului T:

$$t_0 = T_m + 10 \cdot (T_{m\text{-}1} + 10 \cdot (T_{m\text{-}2} + \ldots + 10 \cdot (T_2 + 10 \cdot T_1)\ldots)).$$

Trebuie remarcat faptul că aceste formule sunt valabile doar pentru baza 10. Pentru o bază oarecare b, valoarea p va fi calculată astfel:

$$p = P_m + b \cdot (P_{m\text{-}1} + b \cdot (P_{m\text{-}2} + \ldots + b \cdot (P_2 + b \cdot P_1)\ldots)).$$

De exemplu, dacă lucrăm cu şiruri de caractere ASCII, vom avea $b = 256$ şi formula corectă va fi:

$$p = P_m + 256 \cdot (P_{m\text{-}1} + 256 \cdot (P_{m\text{-}2} + \ldots + 256 \cdot (P_2 + 256 \cdot P_1)\ldots)).$$

Se observă acum că o valoare t_{s+1} poate fi calculată pe baza valorii t_s foarte simplu, folosind formula:

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} \cdot T_{s+1}) + T_{s+m-1}.$$

Prin scăderea valorii $10^{m-1} \cdot T_{s+1}$ se elimină prima cifră a numărului, prin înmulţirea cu zece se adaugă cifra 0 la sfârşitul numărului, iar prin adunarea valorii T_{s+m-1} cifra 0 este înlocuită cu cifra corectă.

Să considerăm şirul 2457 şi subşirul 13. Valoarea p va fi 13, iar valoarea t_0 va fi 24. Pe baza formulei anterioare vom obţine:

$$t_1 = 10 \cdot (24 - 10^{2-1} \cdot 2) + 5 = 10 \cdot (24 - 20) + 5 = 10 \cdot 4 + 5 = 40 + 5 = 45.$$

Aşadar, putem calcula în timp constant fiecare valoare t_s şi apoi o putem compara cu valoarea p. Ca urmare, ordinul de complexitate al algoritmului va deveni $O(n - m + m + m) = O(m + n)$. Iniţial vom calcula valorile t_0 şi p, ambele într-un timp de ordinul $O(m)$. Ulterior, vom determina cele $n - m$ valori t_s, într-un timp total de ordinul $O(n - m)$.

Din nefericire, acest algoritm are un inconvenient şi anume faptul că valorile p şi t_s sunt numere foarte mari. Din acest motiv nu vom putea lucra efectiv cu astfel de valori decât dacă simulăm operaţii cu numere mari. Datorită faptului că aceste valori au m cifre, apare un factor suplimentar $O(m)$, deci obţinem un ordin de complexitate $O(m \cdot n)$.

21.2.3. Utilizarea aritmeticii modulare

Din fericire, există o posibilitate de a elimina numerele mari. Practic, valorile p şi t_s vor fi calculate modulo o valoare q. De obicei, pentru valoarea q este ales un număr prim astfel încât valoarea $b \cdot q$ ($10 \cdot q$ pentru baza 10) să poată fi reprezentată în memorie.

Folosind aritmetica modulară vom putea calcula valorile p şi t_s modulo q într-un timp total de ordinul $O(m + n)$. Pentru a determina valoarea t_{s+1} pe baza valorii t_s vom folosi formula:

$$t_{s+1} = \text{rest}[10 \cdot (t_s - x \cdot T_{s+1}) + T_{s+m-1} / q],$$

unde prin x am notat restul împărţirii întregi a valorii 10^{m-1} la q.

Formula generală de calcul (pentru o bază oarecare b) este:

$$t_{s+1} = \text{rest}[b \cdot (t_s - x \cdot T_{s+1}) + T_{s+m-1} / q],$$

unde x reprezintă restul împărţirii întregi a valorii b^{m-1} la q.

Folosind aritmetica modulară eliminăm inconvenientul numerelor mari, dar apare o nouă problemă: în cazul în care avem $p = t_s$ (modulo q), nu este obligatoriu să avem şi $p = t_s$. Putem deduce doar faptul că există o valoare întreagă d astfel încât $p = t_s + d \cdot q$.

Totuşi, avantajul principal îl constituie faptul că dacă avem $p \neq t_s$ (modulo q), atunci, cu siguranţă, vom avea $p \neq t_s$.

Aşadar, putem folosi comparaţia modulară ca un test euristic. În cazul în care testul eşuează, deducem imediat că subşirul nu poate apărea în şir la poziţia $s + 1$. Dacă testul nu eşuează, va trebui să verificăm dacă nu avem o aşa numită falsă potrivire. Pentru aceasta vom compara efectiv subşirul P cu subşirul din T care începe la poziția $s + 1$.

Teoretic, testul respectiv ar putea fi necesar la fiecare pas. Ordinul de complexitate al testului este $O(m)$ deci, în cazul cel mai nefavorabil, ajungem la ordinul de complexitate $O((n - m) \cdot (m + n)) = O(n^2 - m^2)$.

Practic, dacă valoarea q este bine aleasă (un număr prim cât mai mare) şansa apariției unei false potriviri este foarte redusă ($1 / q$). De aceea, numărul comparațiilor efectuate inutil datorită unor false potriviri va fi de aproximativ $(n - m + 1) / q$.

Aşadar, pentru cazul mediu, algoritmul va funcționa în timp liniar.

21.2.4. Prezentarea algoritmului

În continuare vom prezenta versiunea în pseudocod a algoritmului pentru cazul general în care se utilizează baza de numerație b.

```
Algoritm Rabin_Karp(şir,subşir,b,q):
                                    { şir – şirul în care se caută subşirul }
                                         { subşir – subşirul căutat }
                                      { b – baza de numerație utilizată }
                               { q – valoarea folosită pentru calculele în }
                                               { aritmetica modulară }
  N ← lungime(şir)
  M ← lungime(subşir)
  x ← Ridicare_la_putere_modulo_n_eficient(b,M - 1,q)
                       { se utilizează algoritmul prezentat în capitolul 20 }
  p ← 0
  t0 ← 0
  pentru i ← 1, M execută:
    p ← rest[(b · p + subşir_i) / q]
    t0 ← rest[(b · t0 + şir_i) / q]
  sfârşit pentru
  pentru s ← 0, N - M execută:
    dacă p = t_s atunci
      ok ← adevărat
      j ← 0
      cât timp ok şi j < M execută:
        dacă şir_(s+j+1) ≠ subşir_(j+1) atunci
          ok ← fals
        sfârşit dacă
      sfârşit cât timp
      dacă ok atunci
        scrie 'Subşirul apare la poziția ', s + 1, '.'
```

```
      sfârşit dacă
    sfârşit dacă
    ts+1 ← rest[(b · (ts – şirs+1 · x) + şirs+M+1) / q]
  sfârşit pentru
sfârşit algoritm
```

21.2.5. Timpul de execuţie estimat

Aşa cum am afirmat anterior, ordinul de complexitate al algoritmului *Rabin-Karp* este pătratic. Totuşi, pentru cazul mediu, ordinul de complexitate este liniar. Vom prezenta în continuare modul în care poate fi determinat timpul de execuţie estimat al algoritmului.

Vom nota cu v numărul potrivirilor corecte şi prin q valoarea aleasă pentru efectuarea operaţiilor în aritmetica modulară.

Numărul comparărilor efective va fi de cel puţin v, deoarece o poziţie corectă se determină doar pe baza unei astfel de comparări. Numărul falselor potriviri va fi $O(n / q)$ deoarece pentru fiecare poziţie şansa apariţiei unei false potriviri este $1 / q$.

Aşadar, numărul comparărilor efectuate va avea ordinul $O(v + n / q)$. Calculul valorilor t_s şi p are ordinul de complexitate $O(m + n)$, iar calculul valorii x are ordinul de complexitate $O(\log m)$.

În concluzie, ordinul de complexitate va fi $O(m + n) + O(\log m) + O(v + n / q) = O(m + n) + O(v + n / q)$.

21.3. Algoritmul Knuth-Morris-Pratt

Acest algoritm realizează potrivirea şirurilor folosindu-se o funcţie prefix. În cadrul acestei secţiuni vom prezenta această funcţie şi vom descrie modul în care aceasta poate fi utilizată pentru a determina eficient apariţiile unui subşir într-un şir.

21.3.1. Funcţia prefix

Această funcţie se calculează pentru subşirul ale cărui apariţii sunt căutate. Ea păstrează informaţii referitoare la potrivirea subşirului cu deplasamente ale acestuia.

În cazul în care ştim că primele q caractere ale subşirului se potrivesc cu q caractere ale şirului la o anumită poziţie $s + 1$, funcţia prefix va arăta, pentru o poziţie s, care este cea mai mică poziţie s', astfel încât primele k caractere ale subşirului se pot potrivi cu k caractere ale şirului la poziţia $s + 1$.

Cu alte cuvinte, funcţia va determina poziţia s' pentru care are sens să căutăm potriviri având în vedere structura subşirului căutat. În cel mai bun caz vom sări direct la poziţia $s + q$ şi vom elimina toate poziţiile cuprinse între $s + 1$ şi $s + q - 1$. În orice caz, indiferent care este valoarea determinată, datorită semnificaţiei funcţiei prefix, vom şti cu siguranţă că primele k caractere ale subşirului se vor potrivi la poziţia $s' + 1$.

Valorile funcției prefix pot fi precalculate, folosindu-se comparări ale subșirului cu el însuși. Această funcție este notată, de obicei, prin π.

Valoarea π_q reprezintă lungimea celui mai lung prefix al subșirului care reprezintă un sufix pentru șirul format din primele q caractere ale subșirului.

Modul de calcul al funcției π este descris în cele ce urmează:

```
Algoritm FuncțiePrefix(subșir):
                              { subșirul pentru care se calculează funcția prefix }
  M ← lungime(subșir)
  π1 ← 0
  k ← 0
  pentru q ← 2, M execută:
    cât timp (k > 0) și (subșirk+1 ≠ subșirq) execută:
      k ← πk
    sfârșit cât timp
    dacă subșirk+1 = subșirq atunci
      k ← k + 1
    sfârșit dacă
    πq ← k
  sfârșit pentru
  returnează π
sfârșit algoritm
```

Există o demonstrație a faptului că timpul de execuție al acestei funcții, în cazul în care este implementată în modul prezentat, are ordinul de complexitate $O(m)$. Demonstrația acestei afirmații implică operații matematice relativ complicate, motiv pentru care nu o vom prezenta. Este importantă doar concluzia, și anume faptul că valorile funcției prefix sunt calculate în timp liniar în funcție de lungimea subșirului.

21.3.2. Prezentarea algoritmului

Folosind valorile prefix vom ști, la fiecare pas, numărul de poziții peste care putem "sări" în siguranță, fiind siguri că subșirul nu se poate potrivi la pozițiile respective. Vom prezenta în continuare versiunea în pseudocod a algoritmului:

```
Algoritm KnuthMorrisPratt(șir,subșir)
                                   { șir – șirul în care se caută subșirul }
                                             { subșir – subșirul căutat }
  N ← lungime(șir)
  M ← lungime(subșir)
  π ← FuncțiePrefix(subșir)
  q ← 0
```

```
  pentru i ← 1, N execută:
    cât timp (q > 0) şi (subşir_{q+1} ≠ şir_i) execută:
      q ← π_q
    sfârşit cât timp
    dacă subşir_{q+1} = şir_i atunci
      q ← q + 1
    sfârşit dacă
    dacă q = M atunci
      scrie 'Subşirul apare la poziţia ', i - m, '.'
      q ← π_q
    sfârşit dacă
  sfârşit pentru
sfârşit algoritm
```

21.3.3. Analiza complexităţii

Aşa cum am afirmat anterior, ordinul de complexitate al operaţiei de determinare a valorilor funcţiei prefix este $O(m)$.

După determinarea acestei funcţii, algoritmul *KMP* (Knuth-Morris-Pratt) efectuează n paşi (câte unul pentru fiecare poziţie a şirului). Cu excepţia structurii repetitive în cadrul căreia se modifică valoarea q, operaţiile pentru fiecare pas se efectuează în timp constant.

Folosindu-se aceeaşi metodă ca şi în cazul algoritmului de determinare al funcţiei prefix, se poate arăta că ordinul de complexitate total al acestor operaţii (pentru toţi cei n paşi) va fi $O(n)$.

În concluzie, ordinul de complexitate al algoritmului *KMP* este $O(m + n)$.

21.4. Rezumat

În cadrul acestui capitol am prezentat trei modalităţi prin care pot fi determinate apariţiile unui subşir într-un şir. Toţi cei trei algoritmi determină toate apariţiile, dar pot fi uşor modificaţi pentru a determina prima sau ultima apariţie.

Am început cu un algoritm simplu, dar ineficient care poate fi utilizat pentru orice tip de şiruri. Am continuat cu algoritmul *Rabin-Karp*, un algoritm care impune anumite limitări pentru structura şirurilor. Cu această ocazie am arătat modul în care se utilizează aritmetica modulară pentru îmbunătăţirea timpului de execuţie. Am arătat că, deşi pentru cel mai nefavorabil caz timpul de execuţie al algoritmului este pătratic, pentru cazul mediu acesta devine liniar, motiv pentru care acest algoritm poate fi utilizat cu succes în majoritatea cazurilor.

În final, am prezentat algoritmul *KMP* care rulează în timp liniar în orice situaţie. Am introdus noţiunea de funcţie prefix şi am arătat modul în care aceasta se poate utiliza pentru a verifica apariţiile unui subşir într-un şir.

21.5. Implementări sugerate

Pentru a vă însuşi noţiunile prezentate în cadrul acestui capitol vă sugerăm să realizaţi implementări pentru:

1. determinarea tuturor apariţiilor unui subşir într-un şir de numere, folosind cel mai uşor de implementat algoritm;
2. determinarea tuturor apariţiilor unui subşir într-un şir de caractere, folosind cel mai uşor de implementat algoritm;
3. determinarea primei apariţii a unui subşir într-un şir de caractere, folosind o variantă a aceluiaşi algoritm;
4. determinarea cât mai rapidă a ultimei apariţii a unui subşir într-un şir de caractere, folosind o variantă a aceluiaşi algoritm;
5. determinarea tuturor apariţiilor unui subşir într-un şir de caractere, folosind algoritmul *Rabin-Karp*;
6. determinarea primei apariţii a unui subşir într-un şir de caractere, folosind algoritmul *Rabin-Karp*;
7. determinarea ultimei apariţii unui subşir într-un şir de caractere, folosind algoritmul *Rabin-Karp*;
8. determinarea tuturor apariţiilor unui subşir într-un şir de numere, folosind algoritmul *KMP*;
9. determinarea tuturor apariţiilor unui subşir într-un şir de caractere, folosind algoritmul *KMP*;
10. determinarea primei apariţii a unui subşir într-un şir de caractere, folosind algoritmul *KMP*;
11. determinarea ultimei apariţii unui subşir într-un şir de caractere, folosind algoritmul *KMP*.

21.6. Probleme propuse

În continuare vom prezenta enunţurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informaţiile prezentate în cadrul acestui capitol. Cunoştinţele suplimentare necesare sunt minime.

21.6.1. Scooby Doo

Descrierea problemei

Scooby Doo a primit de ziua lui o brăţară mai ciudată. Aceasta este formată din *N* mărgele, dispuse circular. Pe fiecare dintre mărgele este desenată o literă a alfabetului englez. *Scooby* a numerotat mărgelele de la 1 la *N*, iar acum le cere prietenilor să spună cuvinte şi încearcă să găsească mărgeaua de la care începe cuvântul spus de prieteni.

Date de intrare
Prima linie a fişierului de intrare **SCOOBY.IN** conţine un şir de litere ale alfabetului englezesc, reprezentând literele desentate pe mărgele. Cea de-a doua linie conţine un alt şir de litere ale alfabetului englezesc care reprezintă un cuvânt pe care *Scooby* îl caută pe brăţară.

Date de ieşire
Fişierul de ieşire **SCOOBY.OUT** va conţine o singură linie pe care se va afla un singur număr, reprezentând numărul de ordine al mărgelei de la care începe cuvântul căutat. În cazul în care cuvântul nu se află pe brăţară, valoarea acestui număr va fi –1.

Restricţii şi precizări
- $1 \leq N \leq 5000$;
- cuvântul căutat va conţine cel mult N litere;
- cuvântul căutat poate apărea pe brăţară de mai multe ori; în acest caz se poate alege oricare dintre poziţiile la care începe cuvântul;
- se va face distincţie între literele mici şi literele mari.

Exemple

SCOOBY.IN	SCOOBY.OUT
ScoobyDoobyDoo Doo	7

SCOOBY.IN	SCOOBY.OUT
abcd dabc	4

SCOOBY.IN	SCOOBY.OUT
AlphaBetaGamma Beta	-1

Timp de execuţie: 1 secundă/test

21.6.2. Venus

Descrierea problemei
În timpul tratativelor, în urma cărora se va stabili cine are dreptul de a coloniza satelitul Titan, ambasadorul venusian a primit un mesaj codificat care părea să fi fost trimis de către *Guvernul Planetar* de pe *Venus*.

Evident, el trebuie să verifice dacă mesajul este autentic şi după aceea să aplice un algoritm de decodificare foarte simplu. Dacă mesajul este autentic, atunci el va conține, într-un anumit loc, o semnătură pe care venusianul o cunoaşte.

După identificarea semnăturii, ea va fi eliminată din mesaj, iar restul mesajului va fi destul de uşor de citit. Litera `'a'` va fi înlocuită de litera `'z'`, litera `'b'` va fi înlocuită de litera `'y'` şi aşa mai departe.

Va trebui să verificați dacă mesajul este autentic şi, în caz afirmativ, să determinați mesajul scris de venusieni.

Date de intrare

Prima linie a fişierului de intrare **`VENUS.IN`** conține mesajul care pare a fi sosit de la guvernul venusian. Cea de-a doua linie a fişierului va conține semnătura care trebuie să apară în mesaj.

Date de ieşire

Fişierul de ieşire **`VENUS.OUT`** va conține mesajul decodificat, dacă acesta este autentic sau doar caracterul `'*'` în caz contrar.

Restricții şi precizări

- $1 \leq N \leq 5000$;
- semnătura va conține cel mult $N - 2$ litere;
- semnătura poate apărea în mesaj o singură dată;
- mesajul conține doar litere mici ale alfabetului englezesc.

Exemple

VENUS.IN	**VENUS.OUT**
wvenusz venus	da

VENUS.IN	**VENUS.OUT**
titanevmfh titan	venus

VENUS.IN	**VENUS.OUT**
renunta venus	*

Timp de execuție: 1 secundă/test

21.6.3. Parole

Descrierea problemei
Se consideră un şir de caractere ASCII de lungime *N*, şi mai multe şiruri care reprezintă parole. O parlă este validă dacă şi numai dacă ea apare ca subsecvenţă a şirului dat. Va trebui să determinaţi numărul parolelor valide.

Date de intrare
Prima linie a fişierului de intrare **`PAROLE.IN`** conţine şirul de caractere ASCII. Cea de-a doua linie a fişierului va conţine numărul *K* al parolelor care trebuie verificate. Fiecare dintre următoarele *K* linii va conţine câte o parolă.

Date de ieşire
Fişierul de ieşire **`PAROLE.OUT`** va conţine o singură linie pe care se va afla numărul parolelor valide.

Restricţii şi precizări
- $1 \leq K \leq 1000$;
- $1 \leq N \leq 500$;
- o parolă va conţine cel mult *N* caractere.

Exemplu

PAROLE.IN	**PAROLE.OUT**

`AlphaBetaGamma`
`10`
`Alpha`
`Beta`
`Gamma`
`Delta`
`aBe`
`aGa`
`alpha`
`beta`
`gamma`
`delta`
	`5`

Timp de execuţie: 1 secundă/test

21.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare şi va fi analizată complexitatea algoritmului prezentat.

21.7.1. Scooby Doo

La prima vedere problema se reduce la determinarea poziției la care apare un subşir într-un şir. Totuşi, se observă imediat că există posibilitatea ca subşirul să înceapă spre sfârşitul şirului şi să continue la începutul acestuia. Să presupunem că lungimea subşirului căutat este M. Pentru a evita problema descrisă este suficient ca, la sfârşitul şirului, să adăugăm primele $M - 1$ caractere ale şirului. În aceste condiții problema se reduce la determinarea poziției la care apare un şir format din M elemente într-un şir format din $M + N - 1$ elemente.

Pentru a rezolva problema este suficient să utilizăm un algoritm rapid de potrivire a şirurilor.

Analiza complexității

Datele de intrare constau în două şiruri de caractere formate din N, respectiv M, elemente. Ca urmare, ordinul de complexitate al operației de citire a datelor este **O(N + M)**.

În continuare va trebui să adăugăm $M - 1$ caractere la sfârşitul primului şir, operație al cărei ordin de complexitate este **O(M)**.

Vom aplica acum un algoritm de potrivre a şirurilor; dacă se foloseşte algoritmul *KMP*, ordinul de complexitate al operației este $O(N + M - 1 + M)$ = **O(N + M)**, deoarece căutăm un subşir format din M elemente, într-un şir format din $N + M - 1$ elemente.

Datele de ieşire constau într-un singur număr, ordinul de complexitate al operației de scriere a acestora fiind **O(1)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N + M) + O(M) + O(N + M) + O(1)$ = **O(N + M)**.

20.7.2. Venus

Pentru a vedea dacă mesajul este autentic, va trebui doar să verificăm apariția unui subşir într-un şir. Dacă subşirul apare, îl vom elimina şi apoi vom realiza decodificarea mesajului pe baza regulii descrise în enunț.

Practic, nu va trebui să eliminăm subşirul, ci doar să memorăm poziția la care începe acesta şi să ignorăm secvența corespunzătoare în momentul în care realizăm transformările.

Analiza complexităţii

Datele de intrare constau în două şiruri de caractere formate din N, respectiv M elemente. Ca urmare, ordinul de complexitate al operaţiei de citire a datelor este **O(N + M)**.

Vom folosi un algoritm de potrivre a şirurilor; dacă utilizăm algoritmul eficient (*KMP*), ordinul de complexitate al operaţiei este **O(N + M)**.

Pentru a scrie datele de ieşire va trebui doar să parcurgem şirul, să verificăm dacă poziţia curentă face parte din semnătură şi, dacă nu, să scriem în fişierul de ieşire caracterul decodificat. Această operaţie are ordinul de complexitate **O(N)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N + M) + O(N + M) + O(N)$ = **O(N + M)**.

21.7.3. Parole

Problema se reduce la verificarea existenţei mai multor subşiruri într-un şir dat. Vom lua în considerare fiecare subşir şi vom aplica un algoritm de potrivire a şirurilor pentru a verifica dacă subşirul face sau nu parte din şir. Pe parcursul verificărilor vom număra subşirurile care fac parte din şir (parolele valide) şi în final vom scrie acest număr în fişierul de ieşire.

Analiza complexităţii

Datele de intrare constau într-un şir format din N caractere şi alte K şiruri de dimensiuni diferite. Dacă notăm prin S suma totală a dimensiunilor celor K şiruri, atunci ordinul de complexitate al operaţiei de citire a datelor este **O(N + S)**.

Vom aplica algoritmul *KMP* pentru fiecare dintre cele K şiruri. Dacă lungimea unui subşir este s_i, atunci pentru un subşir vom avea ordinul de complexitate $O(N + s_i)$. Ordinul de complexitate al întregii operaţii de verificare are forma:

$$\sum_{i=1}^{K} O(N + s_i) = O(K \cdot N + S),$$

deoarece avem:

$$\sum_{i=1}^{K} s_i = S.$$

Datele de ieşire constau într-un singur număr, ordinul de complexitate al operaţiei de scriere a acestora fiind **O(1)**.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N + M) + O(M) + O(N + M) + O(1)$ = **O(N + M)**.