

**ALTE CAPITOLE DE ALGORITMI ȘI COMPLEXITATE**  
(note de curs)

Răzvan Andonie

August 12, 2003



# 11

## TEHNICI DE PRELUCRARE A SECVENȚELOR ȘI ȘIRURILOR

### 11.1 Cea mai lungă subsecvență comună

În această secțiune, ne vom referi la secvențe de elemente.

O *subsecvență* a unei secvențe este formată din secvența respectivă minus anumite elemente din ea. O secvență este, în particular, propria sa subsecvență.

Formal, fiind dată secvența  $X = \langle x_1, x_2, \dots, x_m \rangle$ , o altă secvență  $X = \langle z_1, z_2, \dots, z_k \rangle$  este o subsecvență a lui  $X$  dacă există o secvență strict crescătoare  $\langle i_1, \dots, i_k \rangle$  de indici ai lui  $X$ , astfel încât, pentru  $j = 1, 2, \dots, k$ , avem  $x_{i_j} = z_j$ . De exemplu,  $Z = \langle B, C, D, B \rangle$  este o subsecvență a lui  $X = \langle A, B, C, B, D, A, B \rangle$ .

Fiind date două secvențe  $X$  și  $Y$ , secvența  $Z$  este o subsecvență comună a lui  $X$  și  $Y$  dacă este o subsecvență atât a lui  $X$  cât și a lui  $Y$ .

În *problema celei mai lungi subsecvențe comune*, sunt date două secvențe  $X = \langle x_1, x_2, \dots, x_m \rangle$  și  $Y = \langle y_1, y_2, \dots, y_n \rangle$  și se caută cea mai lungă subsecvență comună a lui  $X$  și  $Y$ . În cele ce urmează, vom arăta cum se poate rezolva în mod eficient, prin programare dinamică, problema celei mai lungi subsecvențe comune (CMLSC).

#### 11.1.1 Caracterizarea CMLSC

Algoritmul naiv pentru rezolvarea problemei CMLSC constă în enumerarea tuturor subsecvențelor lui  $X$ , verificarea faptului dacă ele sunt de asemenea subsecvențe ale lui  $Y$ , apoi în păstrarea celei mai lungi subsecvențe găsite. Fiecare subsecvență a lui  $X$  corespunde unei submulțimi de indici  $\{1, \dots, m\}$  din  $X$ . Există  $2^m$  subsecvențe ale lui  $X$ , deci ajungem la un timp exponențial.

După cum vom vedea, problema CMLSC respectă principiul optimalității. Fiind dată secvența  $X = \langle x_1, \dots, x_m \rangle$ , al  $i$ -lea prefix al lui  $X$ , pentru  $i = 0, 1, \dots, m$ , este  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ .  $X_0$  este secvența vidă.

**Proprietatea 11.1** Fie  $X = \langle x_1, x_2, \dots, x_m \rangle$  și  $Y = \langle y_1, y_2, \dots, y_n \rangle$  două secvențe și fie  $Z = \langle z_1, z_2, \dots, z_k \rangle$  o CMLSC a lui  $X$  și  $Y$ .

1. Dacă  $x_m = y_n$ , atunci  $z_k = x_m = y_n$  și  $z_{k-1}$  este o CMLSC a lui  $X_{m-1}$  și  $Y_{n-1}$ .

2. Dacă  $x_m \neq y_n$ , atunci  $z_k \neq x_m$  implică că  $Z$  este o CMLSC a lui  $X_{m-1}$  și  $Y$ .
3. Dacă  $x_m \neq y_n$ , atunci  $z_k \neq y_n$  implică că  $Z$  este o CMLSC a lui  $X$  și  $Y_{n-1}$ .

Din această proprietate rezultă că o CMLSC a două secvențe conține o CMLSC a unor prefixe a celor două secvențe. Deci, este adevărat principiul optimalității.

### 11.1.2 O soluție recursivă pentru subprobleme

Fie  $c[i, j]$  lungimea unei CMLSC a secvențelor  $X_i$  și  $Y_j$ .

Avem:

$$c[i, j] = \begin{cases} 0 & \text{dacă } i = 0 \text{ sau } j = 0 \\ c[i-1, j-1] + 1 & \text{dacă } i, j > 0 \text{ și } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{dacă } i, j > 0 \text{ și } x_i \neq y_j \end{cases}$$

### 11.1.3 Calcularea lungimii unui CMLSC

Pe baza ecuației de mai sus, este ușor de scris un algoritm recursiv, cu timp exponențial, pentru a calcula lungimea unei CMLSC a două șiruri. Deoarece tabloul  $c$  are  $m \cdot n$  elemente, fiecare reprezentând o subproblemă, este clar că multe subprobleme s-ar recalcula inutil.

Completăm tabloul  $c[1..m, 1..n]$  de jos în sus, prin programare dinamică. Completăm tabelul  $b[1..m, 1..n]$  pentru a ne ajuta la construirea soluției optime. Următorul algoritm construiește tablourile globale  $c$  și  $b$ , linie cu linie.

```

procedure lungimea-CMLSC
  { $m = \#X, n = \#Y$ }
  for  $i \leftarrow 1$  to  $m$  do  $c[i, 0] \leftarrow 0$ 
  for  $j \leftarrow 1$  to  $n$  do  $c[0, j] \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $m$  do
    for  $j \leftarrow 1$  to  $n$  do
      if  $x_i = x_j$ 
        then  $c[i, j] \leftarrow c[i-1, j-1] + 1$ 
            $b[i, j] \leftarrow \swarrow$ 
      else if  $c[i-1, j] \geq c[i, j-1]$ 
        then  $c[i, j] \leftarrow c[i-1, j]$ 
            $b[i, j] \leftarrow \uparrow$ 
      else  $c[i, j] \leftarrow c[i, j-1]$ 
            $b[i, j] \leftarrow \leftarrow$ 

```

### 11.1.4 Construirea unei CMLSC

Obținem soluția printr-un apel  $write-CMLSC(m, n)$  al algoritmului recursiv:

```

function write-CMLSC(i, j)
  if i = 0 or j = 0 then return
  if b[i, j] = '↖'
    then write-CMLSC(i - 1, j - 1)
    write xi
  else if b[i, j] = '↑'
    then write-CMLSC(i - 1, j)
    else write-CMLSC(i, j - 1)

```

Timpu pentru *lungimea-CMLSC* este în  $\Theta(m, n)$ , iar pentru *write-CMLSC* este în  $O(m + n)$  (timpul e în  $O(m + n)$ , nu în  $\Theta(m + n)$ , deoarece se poate merge doar pe diagonală sau pe o linie sau coloană). Dacă se cere doar lungimea CMLSC, nu și soluția, putem aloca doar un spațiu auxiliar (pentru *c*) în  $\Theta(n)$ , păstrând linia curentă și cea anterioară. În 1980, Masel și Paterson au găsit un alt algoritm pentru problema CMLSC, cu timpul în  $O(mn / \log n)$ , unde  $n \leq m$ .

## 11.2 Căutarea unui subșir dat

În această secțiune ne referim la *șiruri de caractere* aparținând unui alfabet finit. Un subșir al unui șir  $S[1], S[2], \dots, S[n]$  este un șir  $S[i], S[i + 1], \dots, S[j]$  de caractere consecutive din  $S$ ,  $i \geq 1, j \leq n$ .

Următoarea problemă apare des în elaborarea editoarelor de text, a macroprocesoarelor și a sistemelor de regăsire a informației (de exemplu programele antivirus).

Fie  $S[1..n]$  un *șir țintă*, constând dintr-un tablou cu  $n$  caractere. Fie  $P[1..m]$  un *pattern*, constând dintr-un tablou cu  $m$  caractere. Dorim să aflăm dacă  $P$  apare în  $S$  și dacă da, unde anume. Putem presupune că  $n \geq m$ . Folosim ca barometru numărul de comparații între perechi de caractere.

Algoritmul naiv este:

```

for i ← 0 to n - m do
  ok ← true
  j ← 1
  while ok and j ≤ m do
    if P[j] ≠ S[i + j]
      then ok ← false
      else j ← j + 1
  if ok
    then return i + 1
return 0

```

Algoritmul returnează  $r$  dacă prima apariție a lui  $P$  în  $S$  începe în poziția  $r$  și returnează 0 dacă  $P$  nu este găsit în  $S$ .

În cel mai nefavorabil caz, adică atunci când bucla **while** efectuează mereu  $m$  comparații, numărul total de comparații efectuate este în  $\Omega(m(n - m))$ , adică în  $\Omega(mn)$  când  $n$  este mult mai mare decât  $m$ .

### 11.2.1 Tehnica amprentelor

Să presupunem că șirul țintă  $S$  poate fi descompus în mod natural în subșiruri:  $S = S_1 S_2 \dots S_t$  și că patternul  $P$ , dacă apare în  $S$ , trebuie să fie inclus complet într-unul dintre aceste subșiruri (de exemplu,  $S_i$  sunt liniile unui fișier text).

Ideea este să folosim o funcție booleană  $T(P, S_i)$  care poate fi calculată rapid într-un test preliminar. Dacă  $T(P, S_i)$  este **false**, atunci  $P$  nu poate fi un subșir al lui  $S_i$ ; dacă  $T(P, S_i)$  este **true**, atunci este posibil ca  $P$  să fie un subșir și trebuie să verificăm detaliat (de exemplu, prin algoritmul naiv) acest lucru. O astfel de funcție booleană poate fi implementată prin *tehnica amprentelor*.

Presupunem că mulțimea caracterelor folosite în  $S$  și  $P$  este  $\{a, b, c, \dots, x, y, z, \text{alte}\}$ , adică alfabetul englez și caractere nealfabetice. De asemenea, presupunem că avem un calculator cu cuvinte pe 32 biți. Definim o amprentă astfel:

1. definim  $val("a") = 0, val("b") = 1, \dots, val("z") = 25, val(\text{celelalte}) = 26$
2. dacă  $c_1$  și  $c_2$  sunt caractere, definim  $B(c_1, c_2) = (27 \cdot val(c_1) + val(c_2)) \bmod 32$
3. definim amprenta  $amp(c)$  a șirului  $C = c_1c_2\dots c_r$  ca un cuvânt pe 32 de biți, unde biții de pe pozițiile  $B(c_1, c_2), B(c_2, c_3), \dots, B(c_{r-1}, c_r)$  sunt 1, iar biții de pe pozițiile celelalte sunt 0.

---

### Exemplul 11.1

Dacă  $C = \text{"computers"}$ , obținem:

$$B("c", "o") = 27 \cdot 2 + 14 \bmod 32 = 4$$

$$B("o", "m") = 27 \cdot 14 + 12 \bmod 32 = 6$$

.

.

$$B("r", "s") = 27 \cdot 17 + 18 \bmod 32 = 29$$

Dacă biții unui cuvânt sunt numărați de la 0 (stânga) la 31 (dreapta),  $amp(C)$  este:

$$0000 \ 1110 \ 0100 \ 0001 \ 0001 \ 0000 \ 0000 \ 0100$$

Numai șapte biți sunt 1, deoarece  $B("e", "r") = B("r", "s") = 29$ .

---

Calculăm amprenta pentru fiecare subșir  $S_i$  și pentru  $P$ . Dacă  $S_i$  conține pattern-ul  $P$ , atunci toți biții care sunt 1 în  $amp(P)$  sunt tot 1 și în  $amp(S_i)$ . Avem atunci funcția  $T$ :

$$T(P, S_i) = [(amp(P) \mathbf{and} amp(S_i)) = amp(P)]$$

unde **and** este operatorul de conjuncție pe bit a două cuvinte.

Pozițiile din  $amp(P)$  care sunt 1 trebuie să fie și în  $amp(S_i)$ , invers însă nu.  $T$  poate fi calculat foarte rapid dacă avem amprentele.

Acesta este un alt exemplu de condiționare. Calcularea amprentelor pentru  $S$  necesită un timp în  $O(n)$ . Pentru a calcula  $amp(P)$  este necesar un timp în  $O(m)$ . De aici încolo, căutarea lui  $P$  se face, în principiu, mai rapid decât dacă nu folosim condiționarea.

Ampretele digitale pot fi calculate în mai multe moduri. De exemplu, luând câte trei caractere consecutive ca argumente în funcția  $B$ .

### 11.2.2 Algoritmul lui Knuth-Morris Pratt (KMP, 1977)

Algoritmul KMP găsește aparițiile lui  $P$  în  $S$  într-un timp în  $O(n + m)$  pentru cazul cel mai nefavorabil și folosește tehnica calculării prealabile a unei funcții (antecalcul). Acest antecalcul nu este o preconditionare, deoarece nu folosește la mai multe cazuri.

Fie  $\Sigma^*$  mulțimea tuturor șirurilor de lungime finită formate din caractere ale alfabetului  $\Sigma$ . Șirul vid,  $\varepsilon$ , aparține lui  $\Sigma^*$ . Lungimea unui șir  $x$  este  $|x|$ . Concatenarea a două șiruri  $x$  și  $y$  este notată cu  $xy$  și constă din caracterele lui  $x$  urmate de caracterele lui  $y$ .

Șirul  $w$  este un *prefix* al șirului  $x$ ,  $w \sqsubset x$ , dacă  $x = wy$  pt un  $y \in \Sigma^*$ . În acest caz, avem  $|w| \leq |x|$ .

Șirul  $w$  este un *sufix* al șirului  $x$ ,  $w \sqsupset x$ , dacă  $x = yw$  pt un  $y \in \Sigma^*$ . În acest caz, avem  $|w| \leq |x|$ .

Relațiile  $\sqsubset$  și  $\sqsupset$  sunt tranzitive, iar  $\varepsilon$  este un prefix și un sufix al oricărui șir.

Al  $k$ -lea prefix  $P[1..k]$  al lui  $P[1..m]$  îl notăm cu  $P_k$ .  $P_0 = \varepsilon$ , iar  $P_m = P$ .

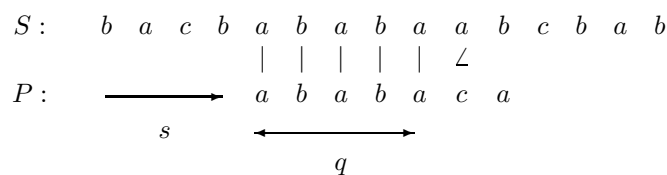
**Proprietatea 11.2** Dacă  $x, y, z$  sunt șiruri astfel încât  $x \sqsubset z$  și  $y \sqsubset z$ , atunci:

1. Dacă  $|x| \leq |y| \Rightarrow x \sqsubset y$
2. Dacă  $|x| \geq |y| \Rightarrow x \sqsupset y$
3. Dacă  $|x| = |y| \Rightarrow x = y$

Demonstrația este imediată.

*Funcția prefix* a unui pattern încapsulează informația referitoare la cum se potrivește pattern-ul cu decalări ale lui. Această informație poate fi folosită pentru a evita testarea unor decalări inutile în algoritmul naiv.

#### Exemplul 11.2



$q = 5$  caractere se potrivesc, al șaselea însă nu.

Cunoscând aceste  $q$  caractere din  $S$ , putem spune că anumite decalări sunt invalide. De exemplu, decalarea  $s + 1$  nu este validă deoarece primul caracter din  $P$ , un  $a$ , s-ar alinia cu un caracter din  $S$  care este egal cu al doilea caracter din  $P$ , un  $b$ .

Decalarea  $s + 2$  aliniază primele trei caractere din  $P$  cu trei caractere cu care știm deja că se potrivesc (deoarece au fost deja comparate).



```

procedure KMP ( $P[1..m], S[1..n]$ )
   $q \leftarrow 0$ 
  calculează-funcția-prefix( $P[1..m]$ )
  for  $i \leftarrow 1$  to  $n$  do
    while  $q > 0$  and  $P[q+1] \neq S[i]$  do  $q \leftarrow \Pi[q]$ 
    if  $P[q+1] = S[i]$  then  $q \leftarrow q+1$ 
    if  $q = m$ 
      then write "pattern-ul apare în poziția"  $i - m$ 
       $q \leftarrow \Pi[q]$ 

procedure calculează-funcția-prefix ( $P[1..m]$ )
   $\Pi[1] \leftarrow 0$ 
   $k \leftarrow 0$ 
  for  $q \leftarrow 2$  to  $m$  do
    while  $k > 0$  and  $P[k+1] \neq P[q]$  do  $k \leftarrow \Pi[k]$ 
    if  $P[k+1] = P[q]$  then  $k \leftarrow k+1$ 
     $\Pi[q] \leftarrow k$ 

```

### Corectitudinea calculării funcției prefix

Vom arăta că, iterând funcția  $\Pi$ , putem enumera toate prefixele  $P_k$  care sunt sufixe ale unui prefix  $P_q$  dat. Fie

$$\Pi^* = \{q, \Pi[q], \Pi^2[q], \dots, \Pi^t\}$$

unde  $\Pi^0[q] = q$ ,  $\Pi^{i+1} = \Pi[\Pi^i[q]]$  pentru  $i > 1$ .

**Proprietatea 11.3** Fie  $P$  un pattern de lungime  $m$  având funcția prefix  $\Pi$ . Atunci, pentru  $q = 1, 2, \dots, m$  avem:  $\Pi^*[q] = \{k | P_k \sqsupseteq P_q\}$

**Demonstrație:** Pentru început, arătăm că  $i \in \Pi^*[q]$  implică  $P_i \sqsupseteq P_q$ . Dacă  $i \in \Pi^*[q]$  atunci  $i = \Pi^u[q]$  pentru un anumit  $u$ . Demonstrăm prin inducție că  $P_i \sqsupseteq P_q$ . Pentru  $u = 0$ , avem  $i = q$  și  $P_q \sqsupseteq P_q$ . Folosind relația  $P_{\Pi[i]} \sqsupseteq P_i$  și tranzitivitatea relației  $\sqsupseteq$ , rezultă că  $P_i \sqsupseteq P_q$  pentru oricare  $i \in \Pi^*[q]$ . Deci,  $P_i^*[q] \subseteq \{k | P_k \sqsupseteq P_q\}$ .

Arătăm acum că  $\{k | P_k \sqsupseteq P_q\} \subseteq \Pi^*[q]$ . Presupunem, prin absurd, că există un întreg în mulțimea  $\{k | P_k \sqsupseteq P_q\} - \Pi^*[q]$  și fie  $j$  cel mai mare astfel de întreg. Deoarece  $q \in \{k | P_k \sqsupseteq P_q\} \cap \Pi^*[q]$ , avem  $j < q$ . Fie  $j'$  cel mai mic întreg în  $\Pi^*[q]$  care este mai mare decât  $j$ . (Putem lua  $j' = q$  dacă nu există un alt număr în  $\Pi^*[q]$  mai mare decât  $j$ ). Avem:

$$\begin{aligned}
 P_j \sqsupseteq P_q, \text{ deoarece } j \in \{k | P_k \sqsupseteq P_q\} & \Rightarrow P_j \sqsupseteq P_{j'} & (11.2) \\
 P_{j'} \sqsupseteq P_q, \text{ deoarece } j' \in \Pi^*[q] &
 \end{aligned}$$

Mai mult,  $j$  este cea mai mare valoare cu această proprietate. Trebuie deci să avem  $\Pi[j'] = j$ . Atunci,  $j \in \Pi^*[q]$ , ceea ce duce la o contradicție. *Q.E.D.*

În exemplul nostru, luând  $q = 8$ , avem:

$$\Pi[8] = 6, \quad \Pi[6] = 4, \quad \Pi[4] = 2, \quad \Pi[2] = 0 \Rightarrow \Pi^*[8] = \{8, 6, 4, 2, 0\}.$$

Vom arăta acum că funcția  $\Pi$  este calculată corect prin algoritmul nostru pentru  $q > 1$  (pentru  $q = 1$  e clar).

**Proprietatea 11.4** Fie  $P$  un pattern de lungime  $m$  și fie  $\Pi$  funcția prefix a lui  $P$ . Pentru  $q = 1, 2, \dots, m$ , dacă  $\Pi[q] > 0$ , atunci  $\Pi[q] - 1 \in \Pi^*[q - 1]$ .

**Demonstrație:** Dacă  $k = \Pi[q] > 0$ , atunci  $P_k \sqsupset P_q$  și deci,  $P_{k-1} \sqsupset P_{q-1}$ . Din proprietatea 11.3 rezultă:  $k - 1 \in \Pi^*[q - 1]$ . *Q.E.D.*

Pentru  $q = 2, 3, \dots, m$ , definim submulțimea  $E_{q-1} \subseteq \Pi^*[q - 1]$  astfel:

$$E_{q-1} = \{k | k \in \Pi^*[q - 1] \text{ și } P[k + 1] = P[q]\}.$$

Mulțimea  $E_{q-1}$  constă din toate valorile  $k \in \Pi^*[q - 1]$  pentru care putem extinde  $P_k$  la  $P_{k+1}$ , obținând un sufix al lui  $P_q$ .

**Proprietatea 11.5** Fie  $P$  un pattern de lungime  $m$  și fie  $\Pi$  funcția prefix a lui  $P$ . Pentru  $q = 2, 3, \dots, m$ , avem:

$$\Pi[q] = \begin{cases} 0 & \text{dacă } E_{q-1} = \emptyset \\ 1 + \max \{k \in E_{q-1}\} & \text{dacă } E_{q-1} \neq \emptyset \end{cases}$$

**Demonstrație:** Dacă  $r = \Pi[q]$ , atunci  $P_r \sqsupset P_q$  și deci  $r \geq 1$  implică  $P[r] = P[q]$ . Prin proprietatea 11.4 avem:

$$r = 1 + \max\{k \in \Pi^*[q - 1] | P[k + 1] = P[q]\}$$

adică

$$r = 1 + \max\{k \in E_{q-1}\}, \text{ iar } E_{q-1} \text{ nu este vidă.}$$

Dacă  $r = 0$ , atunci nu există  $k \in \Pi^*[q - 1]$  pentru care să putem extinde pe  $P_k$  la  $P_{k+1}$  obținând un sufix al lui  $P_q$  (altfel am avea  $\Pi[q] > 0$ ). Deci,  $E_{q-1} = \emptyset$ . *Q.E.D.*

Din proprietatea 11.5 rezultă că algoritmul pentru  $\Pi$  este corect.

### Corectitudinea algoritmului KMP

Se poate demonstra similar.

### Eficiența

Se poate demonstra că algoritmul pentru calcularea lui  $\Pi$  necesită un timp în  $O(m)$  în cazul cel mai nefavorabil. Procedura KMP se poate demonstra că necesită, în cazul cel mai nefavorabil, un timp în  $O(n)$ . Deci, în total, timpul e în  $O(m + n)$ , adică în  $O(n)$ , deoarece  $m \leq n$ .

### 11.2.3 Algoritmul lui Boyer-Moore (BM, 1977)

Dacă patternul  $P$  este relativ lung și alfabetul  $\Sigma$  din care fac parte caracterele este rezonabil de mare, următorul algoritm este foarte eficient:

```

procedure BM(P[1..m], S[1..n],  $\Sigma$ )
  s  $\leftarrow$  0
  calculează-funcția-ultimei-apariții(P[1..m],  $\Sigma$ )
  calculează-funcția-sufix-valid(P[1..m])
  while s  $\leq$  n - m do
    j  $\leftarrow$  m
    while j > 0 and P[j] = S[s + j] do j  $\leftarrow$  j - 1
    if j = 0
      then write "pattern-ul apare pe poziția" s
      s  $\leftarrow$   $\gamma$ [0]
    else s  $\leftarrow$  s + max{ $\gamma$ [j], j -  $\lambda$ [S[s + j]]}

```

Exceptând funcțiile  $\lambda$  și  $\gamma$ , algoritmul seamănă cu algoritmul naiv. Dacă înlocuim ultimele 2 linii cu:

```

  s  $\leftarrow$  s + 1
else s  $\leftarrow$  s + 1

```

obținem în esență algoritmul naiv. Singura deosebire importantă este că algoritmul BM compară pe  $P$  cu  $S$  de la dreapta spre stânga.

Algoritmul BM încorporează două euristici care permit decalări mai mari de un caracter. Aceste euristici sunt: "caracter invalid" și "sufix valid".

#### Exemplul 11.4

```

. . .                               n o t i c e - t h a t
                                   < | |
  _____> r e m i n i s c e n c e
      s

```

Se compară de la dreapta la stânga. Sufixul "ce" este valid iar caracterul "i" este invalid.

```

. . .                               n o t i c e - t h a t
                                   |
  _____> r e m i n i s c e n c e
      s + 4

```

Euristica "sufix valid" decalează pattern-ul spre dreapta până la prima apariție a lui "ce" în  $P$ :

```

. . .                               n o t i c e - t h a t
                                   | |
  _____> r e m i n i s c e n c e
      s + 3

```

Algoritmul BM alege maximul dintre cele două decalaje, deci 4.

**Euristica “caracter invalid”**

În cel mai favorabil caz, o neconcordanță apare deja la prima comparație:  $P[m] \neq S[s+m]$ , iar caracterul invalid  $S[s+m]$  nici nu apare în  $P$ . În acest caz, putem efectua o decalare cu  $m$ . Aici se vede avantajul comparării de la dreapta la stânga față de compararea de la stânga spre dreapta, unde am efectua o decalare cu un singur caracter.

În general, euristica “caracterului invalid” lucrează astfel: presupunând că  $P[j] \neq S[s+j]$  pentru un  $j$ ,  $1 \leq j \leq m$ . Fie  $k$  cel mai mare index,  $1 \leq k \leq m$ , astfel încât  $S[s+j] = P[k]$  (dacă un astfel de index nu există, se ia  $k = 0$ ). Vom demonstra că putem incrementa pe  $s$  cu  $j - k$ . Pentru aceasta, considerăm trei cazuri:

i)  $k = 0$

. . .	t h e - t r e a t m e n t - o f
	∠
→	r e m i n i s c e n c e
s	

Caracterul invalid  $h$  nu apare în  $P$  și deci putem incrementa pe  $s$  cu  $j - k$ .

ii)  $k < j$

. . .	n o t i c e - t h a t
	∠
→	r e m i n i s c e n c e
s	

Caracterul invalid  $i$  apare cel mai în dreapta în  $P$ , într-o poziție la stânga lui  $j$ . Deci putem decala  $P$  cu  $j - k$  caractere la dreapta.

iii)  $k > j$

. . .	f l e e c e - o f
	∠
→	r e m i n i s c e n c e
s	

În acest caz,  $j - k < 0$ . Putem deci incrementa pe  $s$  cu  $j - k$ . Algoritmul BM va ignora acest caz deoarece se ia

$$\begin{array}{ccc} \max\{\gamma[j], & & j - \lambda[S[s+j]]\} \\ \uparrow & & \uparrow \\ \text{euristica} & & \text{euristica} \\ \text{“sufix valid”} & & \text{“caracter invalid”} \\ \text{(mereu pozitivă)} & & \end{array}$$

Următorul algoritm calculează  $\lambda[a]$ , indexul cel mai mare din  $P$  la care apare caracterul  $a$ ,  $a \in \Sigma$ .

**procedure** calculează-funcția-ultimei-apariții( $P[1..m], \Sigma$ )  
**for** fiecare  $a \in \Sigma$  **do**  $\lambda[a] \leftarrow 0$   
**for**  $j \leftarrow 1$  **to**  $m$  **do**  $\lambda[P[j]] \leftarrow j$

Timpul este în  $O(\#\Sigma + m)$ .

**Euristica “sufix valid”**

Definim relația  $Q \sim R$  (“Q este similar cu R”) astfel:

$$Q \sim R \iff Q \sqsupset R \text{ sau } R \sqsupset Q$$

Relația “ $\sim$ ” este simetrică. De asemenea, datorită proprietății 11.1 avem:

$$Q \sqsupset R \text{ si } S \sqsupset R \Rightarrow Q \sim S$$

Două șiruri similare se pot alinia la dreapta, fiecare pereche aliniată potrivitându-se.

Dacă  $P[j] \neq D[s + j]$ ,  $j < m$ , atunci, euristica “sufix valid” afirmă că putem incrementa pe  $s$  cu următoarea funcție sufix valid:

$$\gamma[j] = m - \max\{k \mid [0 \leq k < m][P[j + 1..m] \sim P_k]\}$$

**Observație:** Condiția  $P_k \sqsupset P[j + 1..m]$  este evidentă. Condiția  $P[j + 1..m] \sqsupset P_k$  apare când  $P_k$  este mai scurt decât  $P[j + 1..m]$ .

Definim  $P'$  ca inversul lui  $P$ :  $P'[i] = P[m - i + 1]$  pentru  $i = 1, 2, \dots, m$ . Fie  $\Pi'$  funcția prefix a lui  $P'$ . Se poate arăta că:

$$\gamma[j] = \min(\{m - \Pi[m]\} \cup \{l - \Pi'[l] \mid [1 \leq l \leq m][j = m - \Pi'[l]]\}).$$

```

procedure calculează-funcția-sufix-valid( $P[1..m]$ )
  calculează-funcția-prefix( $P[1..m]$ )
   $P' \leftarrow inversa(P)$ 
  calculează-funcția-prefix( $P'[1..m]$ )
  for  $j \leftarrow 0$  to  $m$  do  $\gamma[j] \leftarrow m - \Pi[m]$ 
  for  $l \leftarrow 1$  to  $m$  do
     $j \leftarrow m - \Pi'[l]$ 
    if  $\gamma[j] > l - \Pi'[l]$ 
      then  $\gamma[j] \leftarrow l - \Pi'[l]$ 

```

Timpul este în  $O(m)$ . Deci, în cazul cel mai nefavorabil, algoritmul BM necesită un timp în  $O((n - m + 1) \cdot m + \# \Sigma)$ . În practică, acest algoritm este cel mai bun.

**Observații:**

Spre deosebire de algoritmul KMP, nu sunt examinate în mod necesar toate caracterele din  $S$ . (De exemplu, în cazul cel mai favorabil, când apare o decalare cu  $m$ ). Algoritmul BM este în multe cazuri subliniar, spre deosebire de algoritmul KMP care este în mod inevitabil liniar.

Algoritmul KMP are avantajul că citește secvențial pe  $S$ , fiind deci util când  $S$  este un fișier secvențial.

**Exemplul 11.5****Algoritmul Knuth-Morris-Pratt**


---

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
        $\angle$   
*P*    *a b c a b c a c a b*

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
       | | |  $\angle$   
*P*    *a b c a b c a c a b*

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
           | | | | | |  $\angle$   
*P*           *a b c a b c a c a b*

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
           | | | |  $\angle$   
*P*           *a b c a b c a c a b*

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
           | | | | | | |  $\angle$   
*P*           *a b c a b c a c a b*

*S*    *b a b c b a b c a b c a a b c a b c a b c a c a b c*  
           | | | | | | | | |  
*P*           *a b c a b c a c a b*

Se execută 28 de comparații.

---



**11.3** Calculați funcția prefix  $\Pi$  pentru patternul *ababbaba*.

**11.4** Să presupunem că aveți calculată funcția prefix  $\Pi$  pentru patternul *PS*. Cum puteți folosi acest lucru când aveți de determinat aparițiile lui *P* în *S*?

**11.5** Găsiți un algoritm liniar care determină dacă un șir *T* este o rotație ciclică a unui alt șir *T'*. De exemplu, "arc" este o rotație completă a lui "car" (și vice-versa).

Indicație:

a) Fie  $T = t_1, \dots, t_n$  și fie  $T' = t'_1, \dots, t'_n$ . Calculăm  $a = \Pi(t_n)$  în  $TT'$  și  $b = \Pi(t'_n)$  în  $TT'$ . *T* este o rotație ciclică a lui *T'* dacă  $a + b = n$ .

b) Altă metodă este să modificăm algoritmul de calcul al lui  $\Pi$  astfel încât să se caute cel mai mare prefix al lui *T* care este sufix al lui *T'*.

**11.6** Fie  $S = "abaabcababb"$ ,  $P = "abb"$ . Câte comparații între elemente ale acestor șiruri efectuează algoritmul lui i)KMP ii)BM pentru a găsi patternul *P* în șirul *S*?

Soluție: i) 14, ii) 8.

**11.7** Elaborați un algoritm cu timpul în  $O(n^2)$  care găsește cea mai lungă subsecvență monoton crescătoare a unei secvențe de *n* numere.

**11.8** Pentru problema anterioară, găsiți un algoritm cu timpul în  $O(n \log n)$ .

Indicație: Observați că ultimul element al unei subsecvențe candidat de lungime *i* este mai mică sau egală cu ultimul element al unei subsecvențe candidat de lungime *i* - 1. Mențineți subsecvențele candidat, legându-le în secvențe de intrare.

# 12

## Introducere în NP-Completitudine

### 12.1 Probleme ușoare - probleme dificile

Algoritmii studiați în acest curs sunt utilizați în general pentru a rezolva probleme practice și de aceea necesită resurse rezonabile (timp și memorie). Din păcate, multe probleme practice nu admit soluții eficiente ba mai mult, pentru unele nici nu știm dacă există soluții eficiente.

Ce este însă un algoritm eficient? Depinde de problema respectivă. Un algoritm de sortare care necesită timp în  $\Theta(n^2)$  este ineficient, în timp ce un algoritm pentru înmulțirea matricilor cu timpul în  $O(n^2 \log n)$  ar fi extraordinar de bun. Uneori este dificil să determinăm o problemă “ușoară” de una “dificilă”. De exemplu, am dat un algoritm care rezolvă problema găsirii celui mai scurt drum de la un vârf  $x$  la un vârf  $y$  într-un graf. Dar dacă ne întrebăm care este cel mai lung drum (fără cicluri) de la  $x$  la  $y$ , avem o problemă pentru care nu se știe altă soluție decât să verificăm toate drumurile posibile. Iată un exemplu:

- Problemă ușoară: Există un drum de la  $x$  la  $y$  de lungime  $\leq M$ ?
- Problemă dificilă: Există un drum de la  $x$  la  $y$  de lungime  $\geq M$ ?

Căutarea *breadth – first* ne dă în timp liniar răspunsul la prima problemă, în timp ce pentru a soluționa a doua problemă, toți algoritmii cunoscuți au timp exponențial.

Într-o primă aproximație, vom accepta că o problemă este *ușor rezolvabilă* numai dacă s-a elaborat un algoritm polinomial pentru rezolvarea ei. O problemă pentru care nu există un algoritm polinomial se numește *dificilă*. Algoritmii exponențiali sunt dificili și, în general, constituie variante ale unei enumerări totale a căilor de identificare a soluțiilor unei probleme. Algoritmii polinomiali reprezintă rezultatul unei cunoașteri detaliate a problemei studiate. O clasă de probleme ce nu vor fi studiate, sunt cele *dificile în sensul cel mai tare*, adică problemele pentru care sa demonstrat că *nu există* algoritmi care să le rezolve.

Problemele dificile pot fi împărțite în două clase:

1. Problemele pentru care se poate demonstra că nu pot fi rezolvate nici cu algoritmi nedeterminiști polinomiali. În această clasă intră și problemele pentru care încă nu există algoritmi.
2. Problemele pentru care există algoritmi polinomiali nedeterminiști. Această clasă este numită NP.

În ordinea descrescătoare a dificultății, avem:

- Probleme dificile în sensul cel mai tare.

- Probleme dificile propriu-zise.
- Probleme din clasa NP.
- Probleme ușoare (pentru care există un algoritm determinist polinomial). Acestea sunt problemele din clasa P.

## 12.2 Algoritmi nedeterminiști

Prin *determinist* înțelegem că în orice moment, indiferent ce face algoritmul respectiv, există un singur lucru pe care sa-l poată face în continuare. Toți algoritmi discutați până acum sunt determiniști și acesta este modul “clasic” de lucru pe calculator.

Un algoritm *nedeterminist* este un algoritm în care este permisă trecerea (necondiționată) dintr-o stare dată în mai multe stări următoare și care poate efectua *simultan* mai multe calcule independente. Un algoritm nedeterminist realizează - ori de câte ori ajunge în situația de a alege între mai multe alternative - oricât de multe copii ale sale, astfel încât să fie posibilă parcurgerea independentă a tuturor alternativelor. Dacă o copie corespunde unei alegeri ce nu furnizează un rezultat, atunci execuția ei se întrerupe. Dacă o copie depistează o soluție a problemei, atunci rezultatele sunt memorate și se întrerupe parcurgerea *tuturor* copiilor. Dacă s-a ajuns în situația că nici o copie generată anterior nu furnizează un rezultat și nici nu mai pot fi generate alte copii, atunci algoritmul semnalizează că problema studiată nu are soluție. Denumirea de algoritm nedeterminist trebuie înțeleasă în sensul că algoritmul se poate afla în mai multe stări *independente* ce nu sunt alese după un anumit criteriu sau generate aleator. Algoritmii nedeterminiști constituie o noțiune abstractă care permite să se ignore anumite detalii.

Un algoritm se numește *nedeterminist polinomial* (NP) în cazul în care complexitatea calculelor efectuate de orice copie a sa (deci pe orice drum al arborelui ce descrie ramificările procesului de căutare a soluției) este polinomială. În mod asemănător se definește algoritmul *determinist polinomial* (P). Evident,  $P \subseteq NP$ . Pentru scrierea algoritmilor nedeterminiști, vom adăuga trei pseudoinstrucțiuni:

- **choice**( $S$ ) - alege arbitrar un element din mulțimea  $S$
- **failure**( $S$ ) - semnalizează încheierea fără succes
- **success**( $S$ ) - semnalizează încheierea cu succes

O atribuire  $X \leftarrow \mathbf{choice}(1:n)$  înseamnă că lui  $X$  i se atribuie unul din întregii între 1 și  $n$ . Nu se specifică nici o regulă conform căreia se face însă selecția. Semnalizările **success** și **failure** sunt folosite pentru a defini un calcul în algoritm. Ele sunt echivalente unui **stop** și nu unui **return**.

Timpul de execuție pentru **choice**, **failure**, **success** se presupune că este în  $O(1)$ . Nu există un calculator care să corespundă unui astfel de mod de lucru, este doar o abstractizare care ne ajută să înțelegem anumite lucruri.

### Exemplul 12.1

Fie problema căutării unui element  $x$  într-un tabel  $A[1], \dots, A[n]$ . Trebuie să determinăm un index  $j$  astfel încât  $A[j] = x$ , sau  $j = 0$  dacă  $x \notin A$ . Un algoritm nedeterminist este:

```

j ← choice(1 : n)
if A[j] = x then print(j); success
print('0'); failure

```

Algoritmul are complexitatea nedeterministă  $O(1)$ . Pe de altă parte, orice algoritm determinist echivalent este în  $\Omega(n)$ , presupunând că  $A$  este neordonat.

### Exemplul 12.2

Fie  $A[1], \dots, A[n]$  un tablou neordonat de întregi pozitivi. Să se sorteze crescător, iar rezultatul să fie afișat:

```

procedure NSort( $A, n$ )
  array  $B[1..n]$ 
   $B \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $n$  do
     $j \leftarrow \text{choice}(1 : n)$ 
    if  $B[j] \neq 0$  then failure
     $B[j] \leftarrow A[i]$ 
  for  $i \leftarrow 1$  to  $n - 1$  do
    if  $B[i] > B[i + 1]$  then failure
  print  $B$ 
success

```

Dacă există o mulțime de alegeri prin **choice** care conduce la rezolvarea cu succes a problemei, atunci algoritmul se termină cu succes. În cazul nostru, deoarece o astfel de permutare există, algoritmul *NSort* este un algoritm de sortare.

Complexitatea este în  $O(n)$ , în timp ce pentru orice algoritm determinist de sortare timpul este în  $\Omega(n \log n)$ .

De fapt, un algoritm nedeterminist are capacitatea de a selecta un element corect prin **choice** (dacă acesta există). Deoarece este un calculator fictiv, nu ne interesează cum face această selecție. De observat că algoritmul se oprește la prima soluție găsită. Investigarea completă are loc doar dacă algoritmul nu are soluție.

Probleme complexe, pentru care un algoritm determinist ar fi foarte complicat, de multe ori pot fi rezolvate ușor printr-un algoritm nedeterminist. Este foarte ușor să obținem un algoritm NP pentru multe probleme care altfel s-ar rezolva prin algoritmi determinați polinomiali.

## 12.3 Probleme NP-dificile și probleme NP-complete

Pentru identificarea faptului că o problemă aparține clasei algoritmilor NP este necesar să definim echivalența dintre două probleme. Presupunând apoi că se cunoaște măcar o problemă  $P_1 \in \text{NP}$ , din echivalența unei probleme  $P_2$  cu  $P_1$ , rezultă  $P_2 \in \text{NP}$ .

### Definiția 12.1

O problemă  $P_2$  se *reduce polinomial* la o problemă  $P_1$  dacă orice caz particular al problemei  $P_2$  se poate transforma în timp polinomial într-un caz particular al problemei  $P_1$  și dacă soluția problemei  $P_2$  se poate obține în timp polinomial din soluția corespunzătoare a problemei  $P_1$ .

Folosim notația  $P_2 \rightarrow P_1$ . Este o relație tranzitivă. Ne referim în continuare doar la reducerea polinomială.

**Definitia 12.2**

O problemă este *NP-dificilă* dacă orice problemă din clasa NP se reduce la ea.

O problemă este *NP-completă* dacă este NP-dificilă și aparține clasei NP.

**Definitia 12.3**

Două probleme  $P_1$  și  $P_2$  sunt *polinomial echivalente* dacă  $P_2 \rightarrow P_1$  și  $P_1 \rightarrow P_2$ .

Ținând seama că relația ' $\rightarrow$ ' este reflexivă și tranzitivă, rezultă că problemele echivalente în sensul definiției 12.3 formează o clasă de echivalență.

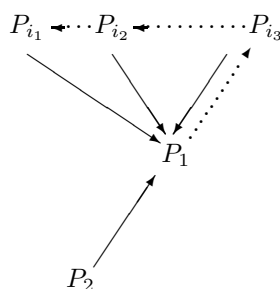
Presupunând că se cunoaște o problemă NP-completă  $P_1$ , pentru a se demonstra că o problemă  $P_2$  este și ea NP-completă, este suficient să se demonstreze că

1.  $P_2 \in \text{NP}$
2.  $P_1 \rightarrow P_2$

Putem folosi un șir de reduceri de forma

$$P_1 \rightarrow P_{i_1} \rightarrow P_{i_2} \rightarrow \dots \rightarrow P_{i_n} \rightarrow P_2$$

dacă demonstrațiile din acest șir sunt mai ușoare.



Problemele  $P_{i_1}, \dots, P_{i_n}, P_1$  sunt toate NP-complete și echivalente. Pentru a demonstra că și  $P_2$  este NP-completă, este suficient să se demonstreze că măcar una dintre problemele  $P_{i_1}, \dots, P_{i_n}, P_1$  se reduce direct sau prin tranzitivitate la  $P_2$ .

Pentru construirea clasei problemelor NP-complete, care este inclusă în clasa problemelor NP, este necesară demonstrarea a două tipuri de teoreme:

1. O singură teoremă care să identifice o problemă NP-completă.
2. Teoreme în care se demonstrează direct sau prin tranzitivitate că  $P_1 \rightarrow P_2$ , unde  $P_2 \in \text{NP}$  este o problemă oarecare pe care dorim să arătăm că este NP-completă.

Cook (1971) a demonstrat primul o teoremă de tipul 1. Există mai multe teoreme de tipul 2.

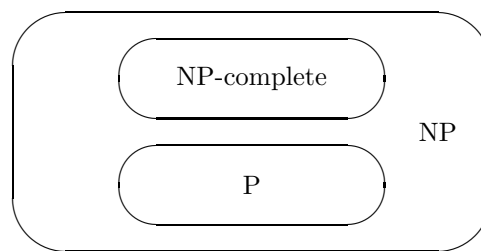
Teoria NP-completitudinii nu dă o metodă pentru a obține algoritmi polinomiali pentru probleme nepolinomiale. Nici nu se afirmă că o problemă nepolinomială este de fapt polinomială sau, mai mult, că toate problemele pot fi rezolvate prin algoritmi polinomiali. Ceea ce ne spune această teorie este că multe probleme pentru care nu

se cunosc algoritmi polinomiali sunt computațional înrudite (echivalente). Am stabilit două clase de probleme NP. O problemă NP-completă are proprietatea că este în P dacă și numai dacă toate problemele NP-complete sunt în P. Dacă o problemă NP-dificilă este în P, atunci toate problemele NP-complete sunt în P. O problemă NP-completă este NP-dificilă, dar invers nu.

Relația dintre aceste două clase de probleme NP și algoritmi nedeterminiști ne face să credem (deocamdată este nedemonstrat) că nici o problemă NP-completă sau PN-dificilă nu este în P. Căci dacă s-ar întâmpla acest lucru, atunci toate problemele NP ar fi în P ceea ce este greu de presupus, deoarece clasele problemelor NP-complete sau PN-dificile sunt foarte bogate și variate.

Se știe că  $P \subseteq NP$ . Nu se știe încă și aceasta este cea mai celebră problemă nerezolvată în informatică, dacă  $P = NP$  sau  $P \neq NP$ . Cel mai probabil este ca  $P \neq NP$ .

În ipoteza că  $P \neq NP$ , avem:



Cel mai important rezultat până în prezent este cel al lui Cook, care și-a pus următoarea problemă: Există o problemă NP care, dacă arătăm că este în P, atunci  $P = NP$ ? Răspunsul este afirmativ. Cook a găsit o problemă NP-completă. Orice problemă NP se poate reduce deci la această problemă.

## 12.4 Teorema lui Cook

Fiind date variabilele booleene  $x_1, x_2, \dots, x_n$  se numește *formă canonică conjunctivă* o expresie de forma:

$$C(x_1, x_2, \dots, x_n) = c_1 \wedge c_2 \wedge \dots \wedge c_p \quad \text{cu} \quad c_j = \tilde{x}_{j1} \vee \tilde{x}_{j2} \vee \dots \vee \tilde{x}_{jm_j}$$

unde  $\tilde{x}_i$  reprezintă variabila  $x_i$  negată sau negația ei, iar disjuncțiile  $c_j$  conțin câte  $m_j$  literale, cu  $1 \leq m_j \leq n$ .

*Problema satisfacerii (SATI)* constă în identificarea unui sistem de valori ale variabilelor  $x_1, x_2, \dots, x_n$  pentru care  $C = 1$ , deci pentru care fiecare conjuncție  $c_j$  ia valoarea 1. Este instructiv de a interpreta aceasta problemă în contextul circuitelor booleene.

*SATI* este în NP, deoarece ea este rezolvabilă prin următorul algoritm nedeterminist polinomial:

```

procedure SATI
  for  $i \leftarrow 1$  to  $n$  do
     $x_i \leftarrow$  choice(0 : 1)
  if  $C(x_1, x_2, \dots, x_n = 1)$  then success
  else failure

```

Teorema Cook poate fi enunțată în două moduri, echivalente.

### Primul enunț

*SATI*  $\in$  P dacă și numai dacă  $P = NP$ .

Demonstrație:

Deoarece  $SATI \in NP$ , din  $P = NP$  se deduce că  $SATI \in P$ .

Rămâne să arătăm că, dacă  $SATI \in P$ , atunci  $P = NP$ . Pentru aceasta este suficient să arătăm că  $SATI$  este NP-dificilă. Ținând cont că  $SATI \in NP$ , asta este echivalent cu a demonstra teorema:

### Al doilea enunț

$SATI$  este NP-completă.

Demonstrație: Trebuie demonstrat că orice problemă din NP se reduce la  $SATI$ . Nu vom demonstra aici acest lucru deoarece este mult mai laborios.

O teoremă similară a fost descoperită independent de Leonid Levin.

## 12.5 Câteva probleme NP-complete

Pentru a demonstra că o problemă  $P_2$  este NP-completă, se va demonstra că  $P_2 \in NP$  și că există o problemă  $P_1$ , NP-completă, reductibilă la  $P_2$ . Iată câteva probleme NP-complete:

1. Problema găsirii unui ciclu (numit *ciclu hamiltonian*) care trece exact o singură dată prin fiecare vârf al unui graf orientat.
2. Problema partiționării: fiind dată o mulțime de întregi, pot fi aceștia împărțiți în două submulțimi cu sume egale?
3. Există o soluție întreagă pentru o problemă de programare linară dată?

Există mii de probleme NP-complete, cu aplicații practice extrem de importante. Faptul că nici unul din algoritmi respectivi nu este polinomial este un argument că  $P \neq NP$ . Ceea ce este însă sigur este faptul că pentru toate aceste probleme nu avem algoritmi eficienți.

Există probleme NP-complete în aplicații numerice, în sortare și căutare, în geometrie, în teoria grafurilor.

Cea mai importantă contribuție practică a teoriei NP-completitudinii este că dă un mecanism pentru a descoperi dacă o nouă problemă este "ușoară" sau "dificilă". Dacă cineva găsește un algoritm eficient pentru a rezolva o nouă problemă atunci problema este "ușoară". În caz contrar, o demonstrare a faptului că problema este NP-completă cel puțin ne spune că obținerea unui algoritm eficient ar fi foarte dificilă (un eveniment în informatică) și că trebuie probabil să simplificăm problema inițială, mulțumindu-ne cu o soluție aproximativă.

Cursul ne arată că am învățat multe pornind de la algoritmul lui Euclid, dar teoria NP-completitudinii ne arată că încă mai este mult de învățat și de descoperit în acest domeniu.

## 12.6 Exerciții

**12.1** Există un algoritm polinomial pentru a determina dacă un circuit boolean oarecare produce mereu 0? (Dacă da, acest circuit se poate înlocui cu un circuit mai simplu care omite toate porțile logice, producând doar 0 la ieșire).