Aplicaţii ale programării dinamice

1 Calculul distanţei de editare

O problemă întâlnită frecvent în informatică este calculul **distanţei de editare** dintre 2 şiruri de caractere. Distanţa de editare este o măsură de similaritate între 2 şiruri, care reprezintă costul minim de operaţii necesare de eliminare/inserare/substituţie, care transformă un şir în alt şir. Fie c_e, c_i şi c_s costurile operaţiilor de eliminare, inserare şi substituţie. Vom presupune că $c_i > 0, c_e > 0$ şi $c_s > 0$.

De exemplu, numărul minim de operaţii necesare pentru a transforma $b =$ "innfornaticant" în $a =$ "informatician" este 4:

1. eliminare: se elimină a treia literă din b,
2. substituţie: se înlocuieşte 'n' (a 7-a literă din b) cu 'm',
3. inserare: se inserează 'i' după poziţia 11 (a lui 'c') în b,
4. eliminare: se elimină ultima literă din b.

Dacă c_e este costul unei operaţii de eliminare, c_i este costul unei operaţii de inserare, şi c_s costul unei operaţii de substituţie, atunci distanţa de editare pentru cele 4 operaţii este $2 \cdot c_e + c_i + c_s$.

1.1 Algoritmul lui Levenshtein

Se presupun date două şiruri de caractere s şi t : $s = a_1 a_2 \dots a_m$ şi $t = b_1 b_2 \cdots b_n$. Pentru fiecare $0 \leq i \leq m$ şi $0 \leq j \leq n$, fie $d(i, j)$ distanţa de editare dintre prefixul $s_i = a_1 a_2 \cdots a_i$ al lui s, şi prefixul $t_j = b_1 b_2 \cdots b_j$ al lui t:

> $d(m, n)$ reprezintă valoarea minimă a costului secvenţelor de operaţii care transformă t în s.

Algoritmul lui Levenshtein se bazează pe observaţiile următoare:

1. Dacă $j = 0$ atunci $t_j = \epsilon$ este şirul vid. În acest caz, secvenţa cu cost minim care transformă $t_j = \epsilon$ în $s_i = a_1 \dots a_i$ constă din i inserări. Deci $d(i, 0) = i \cdot c_i$.
   Dacă $i = 0$ atunci $s_i = \epsilon$ este şirul vid. În acest caz, secvenţa cu cost minim care transformă $t_j = b_1 \dots b_j$ în $s_i = \epsilon$ constă din j eliminări. Deci $d(0, j) = j \cdot c_e$.
2. Dacă $i > 0$, $j > 0$ şi $a_i = b_j$ atunci $d(i, j) = d(i - 1, j - 1)$.

3. Dacă $i > 0$, $j > 0$ şi $a_i \neq b_j$ atunci putem transforma $t_j = b_1 \dots b_j$ în $s_i = a_1 \dots a_i$ în 3 feluri:

   3.1) Transformăm $t_{j-1} = b_1 \dots b_{j-1}$ în $s_{i-1} = a_1 \dots a_{i-1}$, şi înlocuim b_j cu a_i. Costul este $d(i-1, j-1) + c_s$.

   3.2) Transformăm $t_j = b_1 \dots b_j$ în $s_{i-1} = a_1 \dots a_{i-1}$, şi inserăm a_i la sfârşitul lui t_j. Costul este $d(i-1, j) + c_i$.

   3.3) Transformăm $t_{j-1} = b_1 \dots b_{j-1}$ în $s_{i-1} = a_1 \dots a_i$, şi eliminăm b_j de la sfârşitul lui t_j. Costul este $d(i, j-1) + c_e$.

   Ne interesează costul minim, deci

$$d(i,j) = \min\{d(i-1,j-1) + c_s, d(i-1,j) + c_i, d(i,j-1) + c_e\}.$$

În concluzie:

$$d(i,j) = \begin{cases} i \cdot c_i & \text{dacă } j = 0, \\ j \cdot c_e & \text{dacă } i = 0, \\ d(i-1,j-1) & \text{dacă } i > 0, j > 0, a_i = b_j, \\ \min \begin{cases} d(i-1,j-1) + c_s, \\ d(i-1,j) + c_i, \\ d(i,j-1) + c_e \end{cases} & \text{dacă } i > 0, j > 0, a_i \neq b_j. \end{cases}$$

Observăm că pentru calculul lui $d(i,j)$, când $i > 0$ şi $j > 0$, avem nevoie de valorile lui $d(i-1,j-1), d(i-1,j), d(i,j-1)$. Deci există următoarea diagramă de dependenţe între valorile lui $d(i,j)$:

$$\begin{array}{ccccccccc} d(0,0) & & d(0,1) & & d(0,2) & \dots & & d(0,n) \\ & \searrow & \downarrow & \dots & \downarrow & & \searrow & \downarrow \\ d(1,0) & \rightarrow & d(1,1) & \rightarrow & d(1,2) & \dots & \rightarrow & d(1,n) \\ \vdots & \dots & \vdots & \dots & \vdots & \dots & \rightarrow & \vdots \\ & \searrow & \downarrow & \dots & \downarrow & & \searrow & \downarrow \\ d(m,0) & \rightarrow & d(m,1) & \dots & d(m,2) & \dots & \rightarrow & d(m,n) \end{array}$$

Programarea dinamică evită calculul repetat al lui $d(i,j)$, memorând într-un tablou $dc[i,j]$ valorile deja calculate ale lui $d(i,j)$.

1.2 Calculul unei secvenţe crescătoare maximale

Se consideră dat un şir de numere $d[0], \dots, d[n]$. Se doreşte să se determine cea mai mare valoare a lui k pentru care există $0 \leq i_1 < i_2 < \dots < i_k < n$ şi $d[i_1] \leq d[i_2] \leq \dots \leq d[i_k]$. O astfel de secvenţă de indeşi $(i_1, \dots, i_k)$ se numeşte **secvenţă crescătoare maximală** a lui $d[0..n-1]$.

De exemplu, dacă $d[0..n]$ este secvenţa

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0,	8,	4,	12,	**2**,	10,	**6**,	14,	1,	**9**,	5,	13,	3,	**11**,	7,	**15**

atunci (0,4,6,9,13,15) este o secvenţă crescătoare maximală a lui $d[0..n]$.

O soluţie cu complexitate pătratică

Pentru fiecare poziţie $0 \leq i \leq n-1$ calculăm progresiv (de la dreapta la stânga) şi reţinem două informaţii:

1. $\ell[i]$: lungimea celei mai lungi subsecvenţe cere începe cu indexul i
2. $secv[i]$: a listă care conţine o secvenţă maximală a lui $d[0..n-1]$ ce începe cu indexul i. (De fapt, $secv[i]$ poate fi un pointer la primul element al unei astfel de liste)

Deasemenea, reţinem în idx indexul de la care porneşte cea mai lungă secvenţă crescătoare găsită până la momentul respectiv. Pseudocodul acestui algoritm este:

$idx := n-1$
`for` $i := n-1$ `downto` 0 `do`
    $\ell[i] := 1$
    $secv[i] := [i]$
    `for` $j := i+1$ to n `do`
        `if` $d[i] \leq d[j]$ şi $\ell[i] < 1 + \ell[j]$ `then`
            $\ell[i] := 1 + \ell[j]$
            $secv[i]$:=adaugă i ca prim element al lui $secv[j]$
        `if` $\ell[idx] < \ell[i]$ `then` $idx := i$
`return` $secv[idx]$

Acest algoritm are complexitate pătratică $O(n^2)$.

O soluţie cu complexitate $O(n \log n)$

A se vedea `https://en.wikipedia.org/wiki/Longest_increasing_subsequence`

Probleme

1. (Distanţa de editare)

   Să se implementeze un algoritm care calculează distanţa de editare dintre două şiruri citite de la terminal, presupunând că $c_i = c_e = c_s = 1$.

   Programul va trebui să afişeze mesajul

   `Distanta de editare este` nn

   unde nn este valoarea distanţei de editare dintre şirurile citite de la terminal.

2. (Problema elefanţilor inteligenţi)

   Unele persoane cred că un elefant, cu cât este mai greu, este mai inteligent. Pentru a contrazice această presupunere, vom efectua experimentul următor:

   - Presupunem dat un fişier text `elefanti.txt` care conţine informaţii despre un grup de elefanţi: pentru fiecare elefant din grup, există o linie în fişier care conţine două numere întregi separate cu spaţiu: greutatea lui în kilograme şi coeficientul

lui de inteligenţă. Prima linie din fişier conţine numărul `n` de elefanţi din grup, şi este urmată de `n` linii de forma

$$g_i \quad IQ_i$$

unde g_i este greutatea elefantului i, iar IQ_i este coeficientul lui de inteligenţă.

- Vom scrie un program care citeşte acest fişier şi calculează o secvenţă maximală de numere $1 \leq a_1 < a_2 < \ldots < a_k \leq$ `n` astfel încât

$$w_{a_1} < w_{a_2} < \ldots < w_{a_k} \quad \text{şi} \quad IQ_{a_1} > IQ_{a_2} > \ldots > IQ_{a_k}.$$

Altfel spus, vom găsi o cea mai lungă secvenţă de elefanţi care cresc în greutate şi descresc în nivel de inteligenţă.

Pentru scrierea acestui program, vă puteţi folosi de noţiunile descrise în acest curs.

3. (Problema economiei de consum de benzină)

   Se presupune dată situaţia următoare: Se dau 2 localităţi A şi B legate cu un drum de-a lungul căruia sunt $n > 0$ benzinării. Fiecare benzinărie $1 \leq i \leq n$ vinde benzină cu $pret_i$ lei/litru. Un şofer doreşte să plece din A în B cheltuind cât mai puţini bani pe benzină. Se ştie că (1) rezervorul maşinii este de 200 litri, (1) la plecare din localitatea A, rezervorul maşinii este pe jumătate plin, (3) maşina consumă 6 litri/km; şi (4) când ajunge în B, rezervorul maşinii trebuie să fie pe jumătate plin. Se doreşte să se scrie un program care să calculeze costul minim pentru benzină, astfel încât condiţiile (1)-(4) să aibe loc, presupunând că programul citeşte la început:

   (a) distanţa în km dintre localităţile A şi B. Se presupune că distanţa de la A la B este ≤ 5000 km.

   (b) Distanţele d_i de la A la fiecare benzinărie dintre A şi B,

   (c) Preţurile $pret_i$ de lei/litru la fiecare benzinărie dintre A şi B.

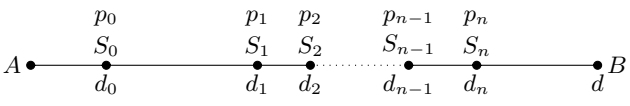
   Dacă nu este posibil ca şoferul să ajungă de la A la B satisfăcând condiţiile de mai sus, programul va trebui să afişeze mesajul `IMPOSIBIL`.

4. Să se determine, folosind programare dinamică, un cel mai lung palindrom care apare ca o subsecvenţă a unui şir de caractere. De exemplu, cel mai lung palindrom care apare ca subsecvenţă a lui `character` este `carac`.

   Programul va trebui să citească şirul de la terminal şi să afişeze pe linia următoare palindromul găsit.

Sugestie de rezolvare a problemei 3

Să presupunem că $S_0, S_1, \ldots, S_n$ sunt staţiile de benzină dintre A şi B, în această ordine.



Pentru a rezolva această problemă vom calcula cu programare dinamică vectorii

$$\texttt{dp}[i] = \langle \texttt{dp}[i][0], \texttt{dp}[i][1], \ldots, \texttt{dp}[i][200] \rangle \quad (0 \leq i \leq n)$$

unde $\texttt{dp}[i][j]$ este preţul minim care trebuie dat pe gaz pentru a ajunge la staţia S_i cu j litri rămaşi în rezervor. Pentru situaţiile în care nu se poate ajunge la staţia S_i cu j litri rămaşi în rezervor, vom defini $\texttt{dp}[i][j] := \texttt{MAXINT}$, unde `MAXINT` este valoarea maximă a unui întreg.

Fie `minCost` costul minim care ne interesează. După ce am calculat vectorul $\texttt{dp}[n]$, putem raţiona astfel:

> Rezervorul trebuie să conţină 100 litri când se ajunge în B, iar $d - d_n$ litri sunt consumaţi pe distanţa de la S_n la B. Deci, la plecare din S_n rezervorul trebuie să conţină $100 + (d - d_n)$ litri, ceea ce este posibil doar dacă $100 + (d - d_n) \leq 200$. Aşadar
>
> - Dacă $100 + d - d_n > 200$ atunci $\texttt{minCost} := \texttt{MAXINT}$ (ceea ce indică `IMPOSIBIL`).
> - Dacă $100 + d - d_n \leq 200$ atunci şoferul trebuie să plece din S_n cu $100 + d - d_n$ litri în rezervor. Dacă şoferul ajunge în S_n cu k litri atunci trebuie ca
>   - $k \leq 100 + d - d_n$.
>   - să cheltuie minim, adică $d[n][k]$ lei pentru a ajunge în sta tia S_n,
>   - să mai cumpere $100 + d - d_n - k$ litri de benzină din S_n cu preţul de p_n lei/litru. Adică, trebuie să mai plătească $p_n \cdot (100 + d - d_n - k)$ lei.
>
> Deci, în total l-ar costa $d[n][k] + p_n \cdot (100 + d - d_n - k)$ lei.
>
> Rezultă că
>
> $\texttt{minCost} := \min(\{\texttt{MAXINT}\} \cup \{\texttt{dp}[n][k] + p_n \cdot (100 + d - d_n - k) \mid k \in A_n\})$
> unde $A_n = \{k \mid 0 \leq k \leq 100 + d - d_n$ şi $\texttt{dp}[n][k] \neq \texttt{MAXINT}\}$.
>
> A_n conţine numărul de litri de benzină cu care poate ajunge şoferul la staţia S_n.

Pentru calculul dinamic al lui $\texttt{dp}[n]$, observăm că

- $\texttt{dp}[0]$ are toate elementele `MAXINT`, cu excepţia cazului când $100 - d_0 \geq 0$, iar în acest caz $\texttt{dp}[0][100 - d_0] := 0$.
- Pentru $1 \leq i \leq n$ putem calcula $\texttt{dp}[i]$ din $\texttt{dp}[i-1]$ calculând $\texttt{dp}[i][j]$ pentru $0 \leq j \leq 200$ în felul următor:
  - De la S_{i-1} la S_i se consumă $d_i - d_{i-1}$ litri iar j este numărul de litri de benzină cu care ajunge în S_i. Deci maşina trebuie să plece din S_{i-1} cu $j + d_i - d_{i-1}$ litri. Acest lucru este posibil doar dacă şoferul ajunge în S_{i-1} cu k litri în rezervor, iar şoferul mai cumpără $j + d_i - d_{i-1} - k$ litri de benzină din S_{i-1}. În această situaţie, costul minim este $\texttt{dp}[i-1][k] + p_{i-1} \cdot (j + d_i - d_{i-1} - k)$. Rezultă că

$\texttt{dp}[i][j] := \min(\{\texttt{MAXINT}\} \cup \{\texttt{dp}[i-1][k] + p_{i-1} \cdot (j + d_i - d_{i-1} - k) \mid k \in A_{i-1}\})$
unde $A_{i-1} = \{k \mid 0 \leq k \leq \min(j + d_i - d_{i-1}, 200)$ şi $\texttt{dp}[i-1][k] \neq \texttt{MAXINT}\}$.

Dacă $\texttt{dp}[i]$ are toate elementele egale cu `MAXINT` (adică nu se poate ajunge în S_i) atunci $\texttt{dp}[k] = \texttt{dp}[i]$ pentru toţi $i < k \leq n$ (nu se poate ajunge nici în S_k) şi $\texttt{minCost} = \texttt{MAXINT}$ (ceea ce indică `IMPOSIBIL`)