

## CURS 12

### TEHNICA PROGRAMĂRII DINAMICE

#### 1. Subșir crescător maximal

Considerăm un șir  $t$  format din  $n$  numere întregi  $t = (t_0, t_1, \dots, t_{n-1})$ . Să se determine un subșir crescător de lungime maximă al șirului dat  $t$ .

Reamintim faptul că un subșir al unui șir este format din elemente ale șirului inițial ai căror indici sunt în ordine strict crescătoare (i.e., un subșir de lungime  $m$  al șirului  $t$  este  $s = (t_{i_0}, t_{i_1}, \dots, t_{i_m})$  cu  $0 \leq i_0 < i_1 < \dots < i_m \leq n - 1$ ) sau, echivalent, este format din elemente ale șirului inițial între care se păstrează ordinea relativă inițială.

De exemplu, în șirul  $t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$  un subșir crescător maximal este **(3, 6, 8, 8, 10)**. Soluția nu este unică, un alt subșir crescător maximal fiind (3, 4, 5, 8, 10). Se observă faptul că problema are întotdeauna soluție, chiar și în cazul în care șirul dat este strict descrescător (orice element al șirului este un subșir crescător maximal de lungime 1)!

Algoritmii de tip Greedy nu rezolvă corect această problemă în orice caz. De exemplu, pentru fiecare element din șirul dat, am putea încerca să construim un subșir crescător maximal selectând, de fiecare dată, cel mai apropiat element mai mare sau egal decât ultimul element din subșirul curent și să reținem subșirul crescător de lungime maximă astfel obținut. Aplicând acest algoritm pentru șirul  $t = (7, 3, 9, 4, 5)$  vom obține subșirurile (7, 9), (3, 9), (9), (4, 5) și (5), deci soluția furnizată de algoritmul Greedy ar fi unul dintre cele 3 subșiruri crescătoare de lungime 2. Evident, soluția ar fi incorectă, deoarece soluția optimă este subșirul (3, 4, 5), de lungime 3!

Un algoritm de tip Backtracking ar trebui să genereze toate submulțimile strict crescătoare de indici (combinări) cu  $1, 2, \dots, n$  elemente, pentru fiecare submulțime de indici să testeze dacă subșirul asociat este crescător și, în caz afirmativ, să rețină subșirul de lungime maximă. Algoritmul este corect, dar ineficient, deoarece numărul submulțimilor generate și testate ar fi egal cu  $C_n^1 + C_n^2 + \dots + C_n^n = 2^n - 1$ , deci algoritmul are avea o complexitate exponențială!

În continuare, vom prezenta un algoritm pentru rezolvarea acestei probleme folosind tehnica programării dinamice (un algoritm de tip Divide et Impera s-ar baza pe aceeași idee, însă fără a utiliza tehnica memoizării).

Pentru a determina un subșir crescător maximal, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subșir crescător care se termină cu  $t[0]$ , apoi lungimea maximă a unui subșir crescător care se termină cu  $t[1]$ , ..., respectiv lungimea maximă a unui subșir crescător care se termină cu  $t[n - 1]$ , iar valorile obținute (optimele locale) le vom păstra într-un tablou  $lmax$  cu  $n$  elemente, respectiv  $lmax[i]$  va memora lungimea maximă a unui subșir crescător care se termină cu  $t[i]$ .

Pentru a calcula lungimea maximă a unui subșir crescător care se termină cu elementul  $t[i]$ , vom lua în considerare, pe rând, toate subșirurile care se termină cu elementele  $t[0], t[1], \dots, t[i - 1]$ , deoarece cunoaștem deja lungimile maxime  $lmax[0], lmax[1], \dots, lmax[i - 1]$  ale subșirurilor crescătoare care se termină cu ele, și vom încerca să alipim elementul  $t[i]$  la fiecare dintre ele. Dacă acest lucru este posibil, respectiv dacă  $t[i] \geq t[j]$  pentru un indice  $j \in \{0, 1, \dots, i - 1\}$ , vom compara lungimea

subșirului care s-ar obține, egală cu  $1 + lmax[j]$ , cu lungimea maximă  $lmax[i]$  găsită până în acel moment și, dacă noua lungime este mai mare, vom actualiza valoarea lui  $lmax[i]$ .

Se observă ușor faptul că problema dată îndeplinește atât *condiția de substructură optimală* (calcularea valorii  $lmax[i]$  depinde de valorile  $lmax[0], lmax[1], \dots, lmax[i - 1]$ ), cât și *condiția de superpozabilitate* (valoarea  $lmax[i]$  va fi utilizată în calculul valorilor  $lmax[i + 1], \dots, lmax[n - 1]$ ), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, & \text{pentru } i = 0 \\ 1 + \max_{0 \leq j < i} \{lmax[j] | t[i] \geq t[j]\}, & \text{pentru } 1 \leq i \leq n - 1 \end{cases}$$

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza un tabloul auxiliar unidimensional *pred*, în care un element  $pred[i]$  va conține valoarea  $-1$  dacă elementul  $t[i]$  nu a putut fi alipit la niciunul dintre subșirurile crescătoare maximale care se termină cu  $t[0], t[1], \dots, t[i - 1]$  sau va conține indicele  $j \in \{0, 1, \dots, i - 1\}$  al subșirului crescător maximal  $t[j]$  la care a fost alipit elementul  $t[i]$ , adică indicele  $j$  pentru care s-a obținut  $\max_{0 \leq j < i} \{lmax[j] | t[i] \geq t[j]\}$ .

Considerând tabloul  $t$  din exemplul inițial, vom obține următoarele valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	1	1	1	2	1	3	2	2	3	4	4	5	3
pred	-1	-1	-1	2	-1	3	2	2	6	5	8	9	7

Valorile din tablourile  $lmax$  și  $pred$  au fost calculate astfel:

- $lmax[0] = 1$  și  $pred[0] = -1$ , deoarece este evident faptul că lungimea maximă a unui subșir care se termină cu  $t[0]$  este egală cu 1;
- $lmax[1] = 1$  și  $pred[1] = -1$ , deoarece elementul  $t[1] = 7$  nu poate fi alipit la un subșir crescător maximal care se termină cu  $t[0] = 9 > 7$ ;
- $lmax[2] = 1$  și  $pred[2] = -1$ , deoarece elementul  $t[2] = 3$  nu poate fi alipit nici la un subșir crescător maximal care se termină cu  $t[0] = 9 > 3$  și nici la un subșir crescător maximal care se termină cu  $t[1] = 7 > 3$ ;
- $lmax[3] = 2$  și  $pred[3] = 2$ , deoarece elementul  $t[3] = 6$  poate fi alipit la un subșir crescător maximal care se termină cu  $t[2] = 3 \leq 6$ , deci  $lmax[3] = 1 + lmax[2] = 2$ ;
- $lmax[4] = 1$  și  $pred[4] = -1$ , deoarece elementul  $t[4] = 2$  nu poate fi alipit nici la un subșir crescător maximal care se termină cu  $t[0], t[1], t[2]$  sau  $t[3]$ ;
- $lmax[5] = 3$  și  $pred[5] = 3$ , deoarece elementul  $t[5] = 8$  poate fi alipit la oricare dintre subșirurile crescătoare maximale care se termină cu  $t[1], t[2], t[3]$  sau  $t[4]$ , dar lungimea maximă se obține când îl alipim la subșirul crescător maximal care se termină cu  $t[3]$ , deoarece  $lmax[3]$  este cea mai mare dintre valorile  $lmax[1], lmax[2], lmax[3]$  și  $lmax[4]$ ;
- .....

Lungimea maximă a unui subșir crescător maximal al șirului inițial este dată de valoarea maximă din tabloul  $lmax$ , iar pentru a reconstitui un subșir crescător maximal

vom utiliza informațiile din tabloul *pred* și faptul că un subșir crescător maximal se termină cu elementul  $t[pmax]$ , unde *pmax* este poziția pe care se află valoarea maximă din tabloul *lmax*.

În cazul exemplului de mai sus, valoarea maximă din tabloul *lmax* este  $lmax[11] = 5$ , deci  $pmax = 11$ , ceea ce înseamnă că un subșir crescător maximal se termină cu  $t[11]$ . Pentru afișarea unui subșir crescător maximal vom proceda astfel:

- inițializăm un indice *i* cu *pmax*, deci  $i = 11$ ;
- $i = 11 \neq -1$ , deci afișăm elementul  $t[i] = t[11] = 10$  și indicele *i* devine egal cu  $pred[11] = 9$ ;
- $i = 9 \neq -1$ , deci afișăm elementul  $t[i] = t[9] = 8$  și indicele *i* devine egal cu  $pred[9] = 5$ ;
- $i = 5 \neq -1$ , deci afișăm elementul  $t[i] = t[5] = 8$  și indicele *i* devine egal cu  $pred[5] = 3$ ;
- $i = 3 \neq -1$ , deci afișăm elementul  $t[i] = t[3] = 6$  și indicele *i* devine egal cu  $pred[3] = 2$ ;
- $i = 2 \neq -1$ , deci afișăm elementul  $t[i] = t[2] = 3$  și indicele *i* devine egal cu  $pred[2] = -1$ ;
- $i = -1$ , deci am terminat de afișat un subșir crescător maximal (dar inversat!) și ne oprim.

Pentru a afișa subșirul crescător maximal reconstituit neinvertat, fie îl salvăm într-o structură de date auxiliară și apoi îl afișăm invers, fie utilizăm o funcție recursivă (variantă utilizată în implementarea de mai jos).

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fișierul `sir.txt`, care conține pe prima linie dimensiunea *n* a șirului, iar pe următoarea linie se află cele *n* elemente ale șirului:

```
#include<stdio.h>
#include<stdlib.h>

int n, t[100], lmax[100], pred[100];

int maxim(int a, int b) { return a > b ? a:b; }
void afisare(int i)
{
    if(i != -1)
    {
        afisare(pred[i]);
        printf("%d ", t[i]);
    }
}

int main()
{
    int i, j, pmax;

    FILE* f = fopen("sir.txt", "r");

    fscanf(f, "%d", &n);
    for(i = 0; i < n ; i++)
        fscanf(f, "%d", &t[i]);
    fclose(f);
```

```

lmax[0] = 1;
pred[0] = -1;
for (i = 1; i < n; i++)
{
    lmax[i] = 1;
    pred[i] = -1;
    for (j = 0; j < i; j++)
        if (t[i] >= t[j] && 1 + lmax[j] > lmax[i])
        {
            lmax[i] = 1 + lmax[j];
            pred[i] = j;
        }
}

pmax = 0;
for (i = 1; i < n; i++)
    if (lmax[i] > lmax[pmax])
        pmax = i;

printf("Lungimea maxima a unui subsir crescator: %d\n",
                                             lmax[pmax]);

printf("Un subsir crescator maximal:\n");
afisare(pmax);

return 0;
}

```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice (deoarece pentru a calcula soluția optimă  $lmax[i]$  a subproblemei  $i$  am utilizat soluțiile optime  $lmax[0], lmax[1], \dots, lmax[i-1]$  ale subproblemelor  $0, 1, \dots, i-1$ ), iar complexitatea sa este egală cu  $O(n^2)$ .

În continuare, vom prezenta un algoritm care utilizează varianta înainte a tehnicii programării dinamice, respectiv vom calcula soluția optimă  $lmax[i]$  a subproblemei  $i$  utilizând soluțiile optime  $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$  ale subproblemelor  $i+1, i+2, \dots, n-1$ . În acest scop, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subșir crescător care începe cu  $t[n-1]$ , apoi lungimea maximă a unui subșir crescător care începe cu  $t[n-2]$ , ..., respectiv lungimea maximă a unui subșir crescător care începe cu  $t[n-1]$ , iar valorile obținute (optimele locale) le vom păstra în tabloul  $lmax$  cu  $n$  elemente, respectiv  $lmax[i]$  va memora lungimea maximă a unui subșir crescător care începe cu  $t[i]$ .

Pentru a calcula lungimea maximă a unui subșir crescător care începe cu elementul  $t[i]$ , vom lua în considerare, pe rând, toate subșirurile care încep cu elementele  $t[i+1], t[i+2], \dots, t[n-1]$ , deoarece cunoaștem deja lungimile maxime  $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$  ale subșirurilor crescătoare care încep cu ele, și vom încerca să adăugăm elementul  $t[i]$  înaintea fiecăruia dintre ele. Dacă acest lucru este posibil, respectiv dacă  $t[i] \leq t[j]$  pentru un indice  $j \in \{i+1, i+2, \dots, n-1\}$ , vom compara lungimea subșirului care s-ar obține, egală cu  $1 + lmax[j]$ , cu lungimea maximă

$lmax[i]$  găsită până în acel moment și, dacă noua lungime este mai mare, vom actualiza valoarea lui  $lmax[i]$ .

Se observă ușor faptul că problema dată îndeplinește atât *condiția de substructură optimală* (calcularea valorii  $lmax[i]$  depinde de valorile  $lmax[i + 1]$ ,  $lmax[i + 2]$ , ...,  $lmax[n - 1]$ ), cât și *condiția de superpozabilitate* (valoarea  $lmax[i]$  va fi utilizată în calculul valorilor  $lmax[0]$ , ...,  $lmax[i - 1]$ ), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, & \text{pentru } i = n - 1 \\ 1 + \max_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}, & \text{pentru } 0 \leq i < n - 1 \end{cases}$$

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza un tabloul auxiliar unidimensional *succ*, în care un element  $succ[i]$  va conține valoarea  $-1$  dacă elementul  $t[i]$  nu a putut fi adăugat înaintea niciunui dintre subșirurile crescătoare maximale care încep cu  $t[i + 1]$ ,  $t[i + 2]$ , ...,  $t[n - 1]$  sau va conține indicele  $j \in \{i + 1, i + 2, \dots, n - 1\}$  al subșirului crescător maximal  $t[j]$  înaintea căruia a fost adăugat elementul  $t[i]$ , adică indicele  $j$  pentru care s-a obținut  $\max_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}$ .

Considerând tabloul  $t$  din exemplul inițial, vom obține următoarele valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	2	4	5	4	5	3	4	4	3	2	2	1	1
succ	11	5	3	5	6	9	8	8	9	11	11	-1	-1

Valorile din tablourile  $lmax$  și  $succ$  au fost calculate astfel:

- $lmax[12] = 1$  și  $succ[12] = -1$ , deoarece este evident faptul că lungimea maximă a unui subșir care începe cu  $t[12]$  este egală cu 1;
- $lmax[11] = 1$  și  $succ[11] = -1$ , deoarece elementul  $t[11] = 10$  nu poate fi adăugat înaintea unui subșir crescător maximal care începe cu  $t[12] = 4 < 10$ ;
- $lmax[10] = 2$  și  $succ[10] = 11$ , deoarece elementul  $t[10] = 7$  poate fi adăugat înaintea unui subșir crescător maximal care începe cu  $t[11] = 10 \geq 7$ , deci  $lmax[10] = 1 + lmax[11] = 2$ ;
- $lmax[9] = 2$  și  $succ[9] = 11$ , deoarece elementul  $t[9] = 8$  poate fi adăugat înaintea unui subșir crescător maximal care începe cu  $t[11] = 10 \geq 8$ , deci  $lmax[9] = 1 + lmax[11] = 2$ ;
- $lmax[8] = 3$  și  $succ[8] = 9$ , deoarece elementul  $t[8] = 5$  poate fi adăugat înaintea oricăruia dintre subșirurile crescătoare maximale care încep cu  $t[9]$ ,  $t[10]$  sau  $t[11]$ , dar lungimea maximă se obține când îl alipim la subșirul crescător maximal care începe cu  $t[9]$ , deoarece  $lmax[9]$  este cea mai mare dintre valorile  $lmax[9]$ ,  $lmax[10]$  și  $lmax[11]$ ;
- .....

Lungimea maximă a unui subșir crescător maximal al șirului inițial este dată de valoarea maximă din tabloul  $lmax$ , iar pentru a reconstitui un subșir crescător maximal

vom utiliza informațiile din tabloul *succ* și faptul că un subșir crescător maximal începe cu elementul  $t[pmax]$ , unde *pmax* este poziția pe care se află valoarea maximă din tabloul *lmax*.

În cazul exemplului de mai sus, valoarea maximă din tabloul *lmax* este  $lmax[2] = 5$ , deci  $pmax = 2$ , ceea ce înseamnă că un subșir crescător maximal începe cu  $t[2]$ . Pentru afișarea unui subșir crescător maximal vom proceda astfel:

- inițializăm un indice *i* cu *pmax*, deci  $i = 2$ ;
- $i = 2 \neq -1$ , deci afișăm elementul  $t[i] = t[2] = 3$  și indicele *i* devine egal cu  $succ[2] = 3$ ;
- $i = 3 \neq -1$ , deci afișăm elementul  $t[i] = t[3] = 6$  și indicele *i* devine egal cu  $succ[3] = 5$ ;
- $i = 5 \neq -1$ , deci afișăm elementul  $t[i] = t[5] = 8$  și indicele *i* devine egal cu  $succ[5] = 9$ ;
- $i = 9 \neq -1$ , deci afișăm elementul  $t[i] = t[9] = 8$  și indicele *i* devine egal cu  $succ[9] = 11$ ;
- $i = 11 \neq -1$ , deci afișăm elementul  $t[i] = t[11] = 10$  și indicele *i* devine egal cu  $succ[11] = -1$ ;
- $i = -1$ , deci am terminat de afișat un subșir crescător maximal și ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fișierul text *sir.txt*, care conține pe prima linie dimensiunea *n* a șirului, iar pe următoarea linie se află cele *n* elemente ale șirului:

```
#include<stdio.h>

int maxim(int a, int b) { return a > b ? a:b; }

int main()
{
    int i, j, n, pmax, t[100], lmax[100], succ[100];

    FILE* f = fopen("sir.txt", "r");
    fscanf(f, "%d", &n);
    for(i = 0; i < n ; i++)
        fscanf(f, "%d", &t[i]);
    fclose(f);

    lmax[n-1] = 1;
    succ[n-1] = -1;
    for (i = n-2; i >= 0; i--)
    {
        lmax[i] = 1;
        succ[i] = -1;
        for (j = i+1; j < n; j++)
            if (t[i] <= t[j] && 1 + lmax[j] > lmax[i])
            {
                lmax[i] = 1 + lmax[j];
                succ[i] = j;
            }
    }
}
```

```

pmax = 0;
for (i = 1; i < n; i++)
    if (lmax[i] > lmax[pmax])
        pmax = i;

printf("Lungimea maxima a unui subsir crescator: %d\n",
                                             lmax[pmax]);

printf("Un subsir crescator maximal:\n");
for (i = pmax; i != -1; i = succ[i])
    printf("%d ", t[i]);

return 0;
}

```

Algoritmul prezentat are complexitatea egală tot cu  $\mathcal{O}(n^2)$ , aceasta nefiind însă minimă. O rezolvare cu complexitatea  $\mathcal{O}(n \log_2 n)$  se poate obține utilizând căutarea binară: <https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/>.

## 2. Subșir comun maximal

Considerăm două șiruri de caractere  $s$  și  $t$  formate din  $m$ , respectiv  $n$  litere mari ale alfabetului englez. Să se determine un subșir comun de lungime maximă al celor două șiruri.

De exemplu, pentru șirurile  $s = \text{"ALGORITMICA"}$  și  $t = \text{"PROGRAMARE"}$ , un subșir comun maximal este **"ORMA"**. Soluția nu este unică, un alt subșir comun maximal al celor două șiruri fiind **"GRMA"**.

O idee de rezolvare care folosește paradigma programării dinamice constă în determinarea, într-o manieră ascendentă, a lungimilor subșirurilor comune maxime, astfel:

- determinăm lungimile subșirurilor comune maxime care se pot obține folosind prima literă din șirul  $s$  și prima literă din șirul  $t$ , primele două litere din  $t, \dots$ , primele  $n$  litere din  $t$  (toate literele, de fapt);
- determinăm lungimile subșirurilor comune maxime care se pot obține folosind primele două litere din șirul  $s$  și prima literă din șirul  $t$ , primele două litere din  $t, \dots$ , primele  $n$  litere din  $t$ ;
- .....;
- determinăm lungimile subșirurilor comune maxime care se pot obține folosind primele  $i$  litere din șirul  $s$  și prima literă din șirul  $t$ , primele două litere din  $t, \dots$ , primele  $j$  litere din  $t, \dots$ , primele  $n$  litere din  $t$ ;
- .....;
- determinăm lungimile subșirurilor comune maxime care se pot obține folosind primele  $m$  litere din șirul  $s$  și prima literă din șirul  $t$ , primele două litere din  $t, \dots$ , primele  $n$  litere din  $t$ .

Pentru a calcula lungimea maximă a subșirului comun maxime care se poate obține folosind primele  $i$  litere  $s_0, s_1, \dots, s_{i-1}$  din șirul  $s$  și primele  $j$  litere  $t_0, t_1, \dots, t_{j-1}$  din  $t$  vom proceda astfel:

- dacă litera  $s_{i-1}$  este egală cu litera  $t_{j-1}$ , atunci lungimea subșirului comun maximal care se poate obține folosind primele  $i$  litere  $s_0, s_1, \dots, s_{i-1}$  din șirul  $s$  și primele  $j$  litere  $t_0, t_1, \dots, t_{j-1}$  din șirul  $t$  se obține adăugând 1 la lungimea subșirului comun maximal care se poate obține folosind primele  $i - 1$  litere din  $s$  și primele  $j - 1$  litere din șirul  $t$ ;
- dacă litera  $s_{i-1}$  este diferită de litera  $t_{j-1}$ , atunci lungimea subșirului comun maximal care se poate obține folosind primele  $i$  litere  $s_0, s_1, \dots, s_{i-1}$  din șirul  $s$  și primele  $j$  litere  $t_0, t_1, \dots, t_{j-1}$  din șirul  $t$  se obține alegând maximul dintre:
  - lungimea subșirului comun maximal care se poate obține folosind primele  $i$  litere din șirul  $s$  și primele  $j - 1$  litere din șirul  $t$ ;
  - lungimea subșirului comun maximal care se poate obține folosind primele  $i - 1$  litere din șirul  $s$  și primele  $j$  litere din șirul  $t$ .

Se observă cu ușurință faptul că sunt îndeplinite condiția de substructură optimală și condiția de superpozabilitate, deci problema poate fi rezolvată utilizând tehnica programării dinamice. În acest sens, vom aplica direct tehnica memoizării, utilizând o matrice  $lmax$  cu  $m + 1$  linii și  $n + 1$  coloane în care un element  $lmax[i][j]$  va memora lungimea maximă a unui subșir comun care se poate obține folosind primele  $i$  elemente  $s_0, s_1, \dots, s_{i-1}$  din șirul  $s$  și primele  $j$  elemente  $t_0, t_1, \dots, t_{j-1}$  din șirul  $t$ . Plecând de la condiția de substructură optimală, obținem următoarea relație de recurență:

$$lmax[i][j] = \begin{cases} 0, & \text{dacă } i = 0 \text{ sau } j = 0 \\ 1 + lmax[i-1][j-1], & \text{dacă } s[i-1] = t[j-1] \\ \max\{lmax[i][j-1], lmax[i-1][j]\}, & \text{dacă } s[i-1] \neq t[j-1] \end{cases}$$

pentru fiecare  $i \in \{0, 1, \dots, m\}$  și fiecare  $j \in \{0, 1, \dots, n\}$ . În plus față de modalitatea de calcul a elementului  $lmax[i][j]$  descrisă mai sus, am adăugat cazurile particulare  $lmax[0][j] = lmax[i][0] = 0$  (evident, lungimea maximă a unui subșir comun care se poate obține folosind 0 litere din șirul  $s$  și primele  $j$  litere din șirul  $t$  este 0).

Pentru șirurile  $s = \text{"ALGORITMICA"}$  și  $t = \text{"PROGRAMARE"}$ , elementele matricei  $lmax$  vor fi următoarele (atenție,  $i$  și  $j$  nu sunt indicii literelor din șirurile  $s$  și  $t$ , ci au semnificația descrisă mai sus – primele  $i$  litere din șirul  $s$ /primele  $j$  litere din șirul  $t$ ):

		Șirul $t$										
		P	R	(		A	R	E				
$lmax$		0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0
Șirul $s$	A	1	0	0	0	0	0	1	1	1	1	1
	L	2	0	0	0	0	0	1	1	1	1	1
	G	3	0	0	0	0	1	1	1	1	1	1
	O	4	0	0	0	1	1	1	1	1	1	1
	R	5	0	0	1	1	1	2	2	2	2	2
	I	6	0	0	1	1	1	2	2	2	2	2
	T	7	0	0	1	1	1	2	2	2	2	2
	M	8	0	0	1	1	1	2	2	3	3	3
	I	9	0	0	1	1	1	2	2	3	3	3
	C	10	0	0	1	1	1	2	2	3	3	3
	A	11	0	0	1	1	1	2	3	3	4	4

$j$

8

$i$



Elementele evidențiate în matricea  $lmax$  au fost calculate astfel:

- $lmax[1][4] = 0$ , deoarece lungimea maximă a unui șir comun format din prima literă din șirul  $s$  și primele 4 litere din șirul  $t$  este 0, deoarece nu au nicio literă în comun, sau, utilizând relația de recurență,  $lmax[1][4] = \max\{lmax[1][3], lmax[0][4]\} = 0$ , deoarece  $s[0] \neq t[3]$ ;
- $lmax[1][6] = 1$ , deoarece lungimea maximă a unui șir comun format din prima literă din șirul  $s$  și primele 6 litere din șirul  $t$  este 1 (au în comun doar litera 'A'), sau, utilizând relația de recurență,  $lmax[1][6] = 1 + lmax[0][5] = 1$ , deoarece  $s[0] = t[5]$ ;
- $lmax[11][10] = 4$ , deoarece lungimea maximă a unui șir comun format din primele 11 litere din șirul  $s$  și primele 10 litere din șirul  $t$  este 4 (au în comun subșirul maximal "ORMA"), sau, utilizând relația de recurență,  $lmax[11][10] = \max\{lmax[10][10], lmax[11][9]\} = \max\{3, 4\}$ , deoarece  $s[11] \neq t[10]$ ;

Lungimea maximă a unui subșir comun al celor două șiruri este dată de valoarea elementului  $lmax[m][n]$ , iar pentru a reconstitui un subșir comun maximal vom utiliza informațiile din matricea  $lmax$ , astfel:

- considerăm doi indici  $i = m$  și  $j = n$ ;
- dacă  $s[i] = t[j]$ , atunci adăugăm litera  $s[i - 1]$  (sau  $t[j - 1]$ ) în soluție și decrementăm ambii indici  $i$  și  $j$ , deoarece valoarea  $lmax[i][j]$  provine din  $1 + lmax[i - 1][j - 1]$ ;
- dacă  $s[i] \neq t[j]$ , atunci verificăm de unde provine  $lmax[i][j]$ , respectiv din  $lmax[i - 1][j]$  sau  $lmax[i][j - 1]$ , și decrementăm doar indicele corespunzător.

Deoarece soluția va fi reconstituită în sens invers, vom utiliza un șir de caractere pe care îl vom completa de la sfârșit spre început!

În cazul exemplului de mai sus, avem  $lmax[11][10] = 4$ , ceea ce înseamnă că un subșir comun maximal are lungimea 4, iar pentru reconstituirea unui astfel de subșir vom urma traseul marcat cu verde în matricea  $lmax$ , elementele care sunt și încadrate cu un pătrat corespunzând literelor care vor fi adăugate în subșirul comun maximal "ORMA":

		Șirul $t$										
			P	R	O	G	R	A	M	A	R	E
$lmax$		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	0	1	1	1	1	1
L	2	0	0	0	0	0	0	1	1	1	1	1
G	3	0	0	0	0	1	1	1	1	1	1	1
O	4	0	0	0	1	1	1	1	1	1	1	1
R	5	0	0	1	1	1	2	2	2	2	2	2
I	6	0	0	1	1	1	2	2	2	2	2	2
T	7	0	0	1	1	1	2	2	2	2	2	2
M	8	0	0	1	1	1	2	2	3	3	3	3
I	9	0	0	1	1	1	2	2	3	3	3	3
C	10	0	0	1	1	1	2	2	3	3	3	3
A	11	0	0	1	1	1	2	3	3	4	4	4

$j=n$

În continuare, vom prezenta implementarea acestui algoritm în limbajul C:

```
#include<string.h>
#include<stdio.h>

int max(int x, int y) { return x > y ? x : y; }

int main()
{
    char s[101], t[101], sol[101];
    int i, j, k, m, n, lmax[101][101];

    printf("Primul sir: "); scanf("%s", s);
    printf("Al doilea sir: "); scanf("%s", t);

    m = strlen(s);
    n = strlen(t);

    for(i = 0; i <= m; i++)
        for(j = 0; j <= n; j++)
            if(i == 0 || j == 0)
                lmax[i][j] = 0;
            else
                if(s[i-1] == t[j-1])
                    lmax[i][j] = lmax[i-1][j-1] + 1;
                else
                    lmax[i][j] = max(lmax[i-1][j], lmax[i][j-1]);

    k = lmax[m][n];
    if(k == 0)
    {
        printf("\nSubsirul comun maximal este vid!\n");
        exit(0);
    }

    sol[k] = '\0';
    i = m;
    j = n;
    while(i > 0 && j > 0)
        if(s[i-1] == t[j-1])
        {
            sol[k-1] = s[i-1];
            i--;
            j--;
            k--;
        }
        else
            if(lmax[i-1][j] > lmax[i][j-1]) i--;
            else j--;

    printf("\nSubsirul comun maximal este %s!\n", sol);

    return 0;
}
```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice, iar complexitatea sa este egală cu  $\mathcal{O}(mn)$ . Dacă aplicăm algoritmul pentru un șir  $s$  și șirul obținut prin sortarea sa crescătoare, ce vom obține?