AVANSAT · OLIMPIADĂ · LECȚIA 5

Programare dinamică de concurs: LCS, distanța de editare, LIS rapid, rucsac 1D

Durată: 90 min Nivel: olimpiadă / informatică intensiv Înainte: lecțiile xi-16, co-01

CE VEI ȘTI DUPĂ LECȚIE

- Construiești tabele dp pe două șiruri: cel mai lung subșir comun (LCS) și distanța de editare.
- Reconstitui soluția (nu doar valoarea) mergând înapoi prin tabel.
- Scrii LIS în O(n log n) cu lower_bound și explici de ce funcționează.
- Reduci memoria rucsacului la un vector și explici de ce bucla merge „în jos".

1. Rețeta de la clasa a XI-a, pe două șiruri

La lecția XI.16 ai văzut programarea dinamică pe un vector. La concurs apar des probleme cu **două șiruri**, unde starea are doi indici: `dp[i][j]` = răspunsul pentru primele `i` litere din primul șir și primele `j` litere din al doilea. Rețeta rămâne aceeași, în patru pași:

Pas	Întrebarea	Pentru LCS
1	Ce înseamnă `dp[i][j]`?	lungimea celui mai lung subșir comun al prefixelor de lungime i și j
2	Care sunt cazurile de bază?	`dp[0][j] = dp[i][0] = 0` (un prefix gol nu are nimic comun)
3	Cum se calculează din valori mai mici?	privești ultimele litere
4	Ce ordine de calcul?	i și j crescători (valorile mici sunt gata)

2. Cel mai lung subșir comun (LCS)

Un **subșir** se obține ștergând litere (nu neapărat consecutive), fără a schimba ordinea. Pentru ABCBD și BDCB, „BCB" este subșir al amândurora (lungime 3).

Ultimele litere `a[i]` și `b[j]`:

- dacă sunt **egale**, ele pot încheia subșirul comun: `dp[i][j] = dp[i−1][j−1] + 1`;
- dacă sunt **diferite**, cel puțin una nu face parte din subșir: `dp[i][j] = max(dp[i−1][j], dp[i][j−1])`.

	∅	B	D	C	B
∅	0	0	0	0	0
A	0	0	0	0	0
B	0	**1**	1	1	1
C	0	1	1	**2**	2
B	0	1	1	2	**3**
D	0	1	2	2	3

lungimea LCS este în colțul din dreapta-jos: 3 (de exemplu BCB)

Tabelul LCS

```cpp
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int dp[1005][1005];                        // dp[i][j] = LCS al prefixelor a[0..i-1] și b[0..j-1]

int main()
{
    string a, b;
    cin >> a >> b;
    int n = a.size(), m = b.size();
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (a[i - 1] == b[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;       // litere egale: le „lipim”
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);                  // altfel: renunțăm la una
    cout << dp[n][m] << '\n';
    string lcs;                                // reconstituire: de la colțul (n, m) înapoi
    int i = n, j = m;
    while (i > 0 && j > 0)
    {
        if (a[i - 1] == b[j - 1]) { lcs += a[i - 1]; i--; j--; }
        else if (dp[i - 1][j] >= dp[i][j - 1]) i--;
        else j--;
    }
    reverse(lcs.begin(), lcs.end());
    cout << lcs;
    return 0;
}
```

DE CE AȘA?

De ce `+1` doar când literele sunt egale? Dacă `a[i] = b[j]`, nu pierdem nimic potrivindu-le (o potrivire optimă poate fi mereu extinsă cu ele). Dacă sunt diferite, nu pot fi ambele în subșir ca ultime litere, deci cel puțin una „se aruncă": ținem cel mai bun dintre cele două variante. **De ce se reconstituie „înapoi"?** Valoarea din `dp[i][j]` provine dintr-o singură celulă vecină; mergând de la `dp[n][m]` spre `dp[0][0]` și urmărind de unde a venit fiecare valoare, regăsim literele potrivite, în ordine inversă (de aceea `reverse` la final).

DE ȘTIUT

Pot exista **mai multe** subșiruri comune de aceeași lungime (aici și „BCB", dar pe alte teste mai multe variante). Un program care afișează „un subșir comun maxim" poate alege oricare; atenție la ce cere enunțul.

3. Distanța de editare

Problemă. Prin câte operații minime transformi șirul `a` în `b`? Operațiile (fiecare costă 1): **șterge** o literă, **inserezi** o literă, **înlocuiești** o literă.

Stare. `dp[i][j]` = numărul minim de operații pentru a transforma primele `i` litere din `a` în primele `j` din `b`. **Cazuri de bază.** `dp[i][0] = i` (șterge toate cele i litere), `dp[0][j] = j` (inserezi toate cele j litere). **Tranziție.** Ultimele litere pot fi tratate în trei feluri:

- șterge `a[i]`: `dp[i−1][j] + 1`;
- inserezi `b[j]` la sfârșit: `dp[i][j−1] + 1`;
- potrivești `a[i]` cu `b[j]`: `dp[i−1][j−1]` (0 operații dacă literele sunt egale, 1 dacă le înlocuiești).

	∅	p	i	c	a
∅	0	1	2	3	4
p	1	0	1	2	3
i	2	1	0	1	2
s	3	2	1	1	2
i	4	3	2	2	2
c	5	4	3	2	3
a	6	5	4	3	**2**

dp[6][4] = 2 (ștergem „s" și „i")

Distanța de editare

```cpp
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int dp[1005][1005];                        // dp[i][j] = distanța dintre a[0..i-1] și
b[0..j-1]

int main()
{
    string a, b;
    cin >> a >> b;
    int n = a.size(), m = b.size();
    for (int i = 0; i <= n; i++) dp[i][0] = i;     // ștergem i litere
    for (int j = 0; j <= m; j++) dp[0][j] = j;     // inserăm j litere
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
        {
            int cost = (a[i - 1] == b[j - 1]) ? 0 : 1;
            dp[i][j] = min({dp[i - 1][j] + 1,            // ștergere
                            dp[i][j - 1] + 1,            // inserare
                            dp[i - 1][j - 1] + cost});   // potrivire / înlocuire
        }
    cout << dp[n][m];
    return 0;
}
```

DE CE AȘA?

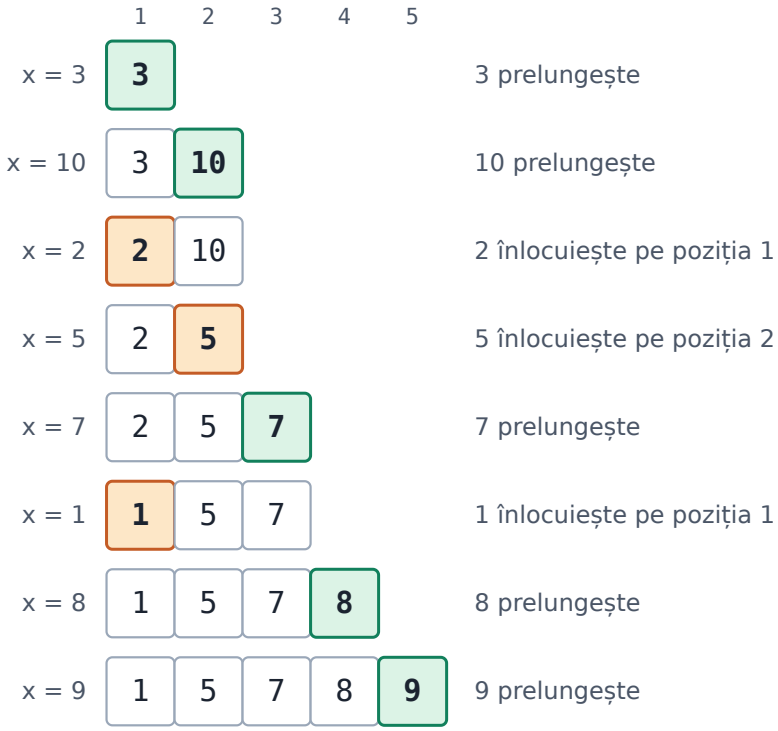
De ce minimul din trei variante este suficient? Orice transformare optimă trebuie să „rezolve” ultima literă a lui `a` și ultima literă a lui `b`: fie șterge una, fie inserează una, fie le potrivește. După una dintre aceste operații, rămâne o problemă mai mică, deja calculată în tabel. **De ce inițializarea `dp[i][0] = i` este esențială?** Fără ea, marginea tabelului ar rămâne 0, iar algoritmul ar crede că transformarea unui prefix nevid în șirul vid costă 0.

4. LIS în O(n log n)

La lecția XI.16 ai calculat cel mai lung subșir **crescător** cu `dp[i]` în $O(n^2)$. Pentru $n = 10^5$ este prea lent. Trucul: ținem un vector `coada` în care `coada[k]` este **cea mai mică valoare** cu care se poate termina un subșir crescător de lungime `k + 1`. Vectorul este mereu **crescător**, deci putem căuta binar în el.

Pentru fiecare număr `x`:

- dacă `x` este mai mare decât toate, **prelungește** cel mai lung subșir: îl adăugi la sfârșit;
- altfel, găsești primul element `≥ x` (cu `lower_bound`) și îl **înlocuiești** cu `x` (un subșir de aceeași lungime, dar cu final mai mic: mai ușor de prelungit).



lungimea șirului la final = lungimea celui mai lung subșir crescător: 5

LIS rapid

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int n;
    cin >> n;
    vector<int> coada;                    // coada[k] = cea mai mică valoare finală a unui subșir crescător de lungime k+1
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        auto it = lower_bound(coada.begin(), coada.end(), x);   // primul element >= x
        if (it == coada.end()) coada.push_back(x);              // x prelungește cel mai lung subșir
        else *it = x;                                           // x îmbunătățește un subșir existent
    }
    cout << coada.size();
    return 0;
}
```

DE CE AȘA?

De ce înlocuirea nu strică nimic? Înlocuind `coada[k]` cu un `x` mai mic sau egal, subșirul de lungime `k + 1` se termină acum într-o valoare mai mică: orice număr care urma să prelungească vechiul subșir îl prelungește și pe cel nou. Nu pierdem nimic, doar câștigăm. **Atenție:** vectorul `coada` nu este, în general, un subșir real din șir (pentru 3 10 2 5 7 1 8 9, după citirea lui 1 vectorul este „1 5 7", deși 1 apare în șir după 5 și 7); doar **lungimea** lui este corectă. **Strict sau nestrict?** Pentru subșir **strict** crescător se folosește `lower_bound` (primul `≥ x`); pentru **nedescrescător** (egalitățile permise) se folosește `upper_bound` (primul `> x`).

5. Rucsacul cu un singur vector

La rucsacul 0/1 din XI.16 ai folosit `dp[i][g]`. Observă că linia `i` depinde doar de linia `i − 1`: putem folosi **un singur vector** `dp[g]` = valoarea maximă cu greutate cel mult `g`, **dacă** parcurgem greutățile **în jos**.

```
#include <iostream>
using namespace std;

int w[1005], val[1005];
long long dp[10005];                    // dp[g] = valoarea maximă cu greutate cel mult g

int main()
{
    int n, G;
    cin >> n >> G;
    for (int i = 1; i <= n; i++) cin >> w[i] >> val[i];
    for (int i = 1; i <= n; i++)
        for (int g = G; g >= w[i]; g--)                  // în jos: fiecare obiect se
folosește o singură dată
            dp[g] = max(dp[g], dp[g - w[i]] + val[i]);
    cout << dp[G];
    return 0;
}
```

obiect (greutate, valoare)	dp[0..7] după procesare
început	0 0 0 0 0 0 0 0
(3, 4)	0 0 0 4 4 4 4 4
(4, 5)	0 0 0 4 5 5 5 9
(2, 3)	0 0 3 4 5 7 8 9
(5, 6)	0 0 3 4 5 7 8 9

Răspunsul este `dp[7] = 9` (de exemplu obiectele (3,4) și (4,5)).

DE CE AȘA?

De ce bucla pe greutate merge în jos? `dp[g − w[i]]` trebuie să fie valoarea **dinainte** de a lua obiectul `i`. Mergând de la G spre w[i], celulele cu indice mai mic nu au fost încă modificate în pasul curent. Dacă ai merge în sus, `dp[g − w[i]]` ar include deja obiectul `i` și l-ai lua de mai multe ori: ar ieși **rucsacul cu obiecte nelimitate** (un alt algoritm, valid și el, dar altă problemă). **Câștigul:** memoria scade de la n·G la G.

6. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
tabel LCS fără linia/coloana 0	`dp[i−1][j−1]` iese din tablou	indici de la 1, `dp[0][*] = dp[*][0] = 0`
editare: uiți `dp[i][0] = i`	marginea rămâne 0	inițializează marginile
LIS cu `upper_bound` pentru „strict crescător”	acceptă și valori egale	`lower_bound`
se afișează `coada` ca subșir	nu este un subșir real	doar lungimea e corectă (pentru subșir real: păstrezi predecesori)
rucsac 1D cu bucla în sus	obiecte folosite de mai multe ori	`g` de la G în jos
`dp[5000][5000]` de `int`	100 MB: depășește memoria	două linii alternative
tie-break diferit la reconstituire	alt subșir (la fel de bun)	citește ce cere enunțul

7. La olimpiadă

Subiectele clasice: LCS și variante (subșir comun a trei șiruri, cel mai lung palindrom ca subșir), editare, LIS cu reconstituire, rucsac cu constrângeri suplimentare. Reflexul: *„starea mea trebuie să rețină doar atât cât să iau decizia următoare”*. Pentru n până la ~5 000 un tabel pătratic merge; la 10^5 cauți O(n log n) (LIS) sau altă idee.

8. Exersează pe hârtie

1. Calculează LCS pentru ABCD și ACBD (lungimea și un subșir).

2. Care este distanța de editare dintre abc și yabd? Descrie operațiile.

3. Pentru șirul 5 1 6 2 7 3 8, simulează vectorul `coada` și spune lungimea LIS.

4. Rucsac cu obiectele (2, 3), (3, 4), (4, 5) și G = 5: completează vectorul `dp[0..5]` după fiecare obiect.

5. De ce, la rucsacul 1D, dacă am parcurge `g` crescător am obține alt rezultat? Dă un exemplu cu un singur obiect (2, 3) și G = 6.

6. Câte celule are tabelul LCS pentru două șiruri de lungime 3 000? Ce memorie (în MB) ocupă dacă folosești `int`?

7. Cum afli cel mai lung subșir **palindrom** al unui șir s folosind LCS? Aplică pe agbdba.

8. Cum ai modifica LIS pentru subșir **nedescrescător**? Care este lungimea pentru 2 2 1 3 3?

9. În tabelul distanței de editare pentru „pisica" → „pica", ce reprezintă valoarea din celula (3, 2)?

10. Estimează numărul de operații pentru LCS cu n = m = 3 000 și pentru LIS cu n = 10^5.

9. Mini-grile

1. Complexitatea timp pentru LCS pe șiruri de lungimi n și m: a) O(n + m) b) O(n · m) c) O(n log m) d) O(2^n)

2. Distanța de editare dintre „ab" și „" (șirul vid) este: a) 0 b) 1 c) 2 d) nedefinită

3. La rucsacul 1D, parcurgem greutățile în jos pentru că: a) merge mai repede b) fiecare obiect să fie folosit cel mult o dată c) ca vectorul să rămână sortat d) nu contează

4. LIS strict crescător în O(n log n) folosește: a) `upper_bound` b) `lower_bound` c) sortare d) `binary_search`

10. Mai departe (pentru cine vrea mai mult)

- **Memorie O(min(n, m))** pentru LCS și editare: păstrezi doar două linii (cea curentă și cea precedentă).
- **LIS cu reconstituirea** subșirului: reții pentru fiecare element poziția în `coada` și predecesorul.
- **Cel mai lung subșir comun al trei șiruri:** `dp[i][j][k]`.
- **PD pe intervale** (de exemplu cel mai lung subșir palindrom direct, `dp[st][dr]`).
- **Rucsac cu obiecte nelimitate**, **rucsac cu două restricții** și **PD pe submulțimi (bitmask)**, lecția 11.

Rezumat

- Pe două șiruri: `dp[i][j]` pentru prefixe; cazuri de bază pe margini; tranziția privește ultimele litere.
- **LCS:** egale → `dp[i−1][j−1] + 1`, diferite → maximul dintre `dp[i−1][j]` și `dp[i][j−1]`; reconstituirea merge înapoi.
- **Editare:** minimul dintre ștergere, inserare, înlocuire/potrivire; marginile `dp[i][0] = i`, `dp[0][j] = j`.
- **LIS rapid:** vectorul `coada` + `lower_bound`: O(n log n); doar lungimea este valabilă.
- **Rucsac 1D:** greutățile în jos, ca fiecare obiect să se folosească o singură dată.

Probleme de lucru (concurs)

Probleme de pe infoarena, CSES, Kilonova sau pbinfo, în ordinea dificultății. Trimite soluțiile pe platformă și bifează ce ai terminat.

- [ ] CSES **Edit Distance** — ușoară
  exact distanța de editare din lecție
  https://cses.fi/problemset/task/1639

- [ ] CSES **Longest Common Subsequence** — medie
  LCS cu reconstituirea șirului
  https://cses.fi/problemset/task/3403
- [ ] CSES **Increasing Subsequence** — medie
  LIS în O(n log n)
  https://cses.fi/problemset/task/1145
- [ ] CSES **Book Shop** — medie
  rucsac 0/1 cu vector
  https://cses.fi/problemset/task/1158
- [ ] infoarena **scmax** — medie
  cel mai lung subșir crescător, cu afișarea subșirului
  https://www.infoarena.ro/problema/scmax

Temă (concurs)

Mai grele: încearcă-le după ce le-ai terminat pe cele de mai sus.

- [ ] infoarena **cmlsc** — medie
  LCS cu afișarea unui subșir comun maxim
  https://www.infoarena.ro/problema/cmlsc
- [ ] infoarena **rucsac** — medie
  rucsac 0/1 cu memorie O(G)
  https://www.infoarena.ro/problema/rucsac

Citește și exersează în plus

- CP-Algorithms — Dynamic programming (introducere și probleme) — idei și rețete pentru stări și tranziții (engleză) https://cp-algorithms.com/dynamic_programming/intro-to-dp.html
- USACO Guide — Dynamic Programming — de la rucsac la PD pe șiruri, cu probleme graduate (engleză) https://usaco.guide/gold/intro-dp
- VisuAlgo — Longest Common Subsequence / Edit Distance — animații pentru algoritmi; caută „DP” în meniul principal (engleză) https://visualgo.net/en

Răspunsuri

Exersează pe hârtie

1. Lungimea este **3**; subșiruri comune posibile: ABD și ACD.

2. Distanța este **2**: inserezi „y" la început (yabc), apoi înlocuiești „c" cu „d" (yabd).

3. coada: 5 → 1 → 1 6 → 1 2 → 1 2 7 → 1 2 3 → 1 2 3 8. Lungimea LIS este **4** (de exemplu 1 2 3 8).

4. Începem cu 0 0 0 0 0 0. După (2,3): 0 0 3 3 3 3. După (3,4): 0 0 3 4 4 7. După (4,5): 0 0 3 4 5 7. Răspunsul `dp[5] = 7` (obiectele (2,3) și (3,4)).

5. Cu `g` crescător: dp[2] = 3, apoi dp[4] = dp[2] + 3 = 6, dp[6] = 9: obiectul (2, 3) este luat de trei ori (9). Corect, cu `g` în jos: dp[6] = 3.

6. 3 001 · 3 001 ≈ $9 \cdot 10^6$ celule; cu `int` (4 octeți) ≈ 36 MB.

7. LCS între s și s inversat. Pentru agbdba, inversul este abdbga; LCS are lungimea 5 („abdba"): cel mai lung palindrom ca subșir are 5 litere.

8. Înlocuiești `lower_bound` cu `upper_bound`. Pentru 2 2 1 3 3: coada devine 2 → 2 2 → 1 2 → 1 2 3 → 1 2 3 3, lungimea este **4** (2 2 3 3).

9. `dp[3][2]` este distanța dintre „pis" și „pi": 1 (șterge „s").

10. LCS: 3 000 · 3 000 = $9 \cdot 10^6$ operații. LIS: $10^5 \cdot \log_2 10^5 \approx 1{,}7 \cdot 10^6$. Ambele sunt sub o secundă.

Mini-grile

1. b) O(n · m). **2.** c) 2 (două ștergeri). **3.** b). **4.** b) `lower_bound`.