INFORMATICĂ · CLASA A XI-A · METODE DE PROGRAMARE (INTENSIV) · LECȚIA 16

Programare dinamică: idee, recurențe, probleme clasice

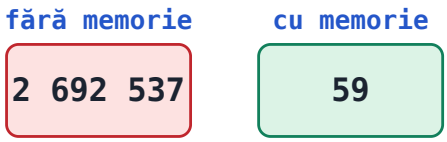
Durată: 60 min Nivel: intensiv informatică / olimpiadă Înainte: lecțiile xi-03, x-09

CE VEI ȘTI DUPĂ LECȚIE

- Înțelegi ideea: soluția problemei mari se construiește din soluțiile subproblemelor, calculate o singură dată.
- Formulezi starea (ce reprezintă d[i] sau d[i][j]) și relația de recurență.
- Rezolvi: scări (moduri de a urca), sumă maximă în triunghi, drumuri în matrice, cel mai lung subșir crescător, rucsac discret.

1. Ideea: nu calcula de două ori același lucru

Fibonacci recursiv, `f(n) = f(n–1) + f(n–2)`, calculează `f(3)` de mai multe ori când cerem `f(5)`. Pentru n = 30 programul face peste **2,6 milioane** de apeluri. Dacă reținem fiecare rezultat într-un vector și îl calculăm **o singură dată**, ajung 59.



memoria reține fiecare rezultat și îl calculează o singură dată

Fără memorie și cu memorie

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

long long memo[100], ap1, ap2;

long long f1(int n)                        // fără memorie
{
    ap1++;
    if (n < 2) return n;
    return f1(n - 1) + f1(n - 2);
}

long long f2(int n)                        // cu memorie (memoizare)
{
    ap2++;
    if (n < 2) return n;
    if (memo[n]) return memo[n];           // l-am calculat deja
    return memo[n] = f2(n - 1) + f2(n - 2);
}

int main()
{
    int n;
    cin >> n;
    cout << f1(n) << ' ' << ap1 << ' ' << (f2(n), ap2);
    return 0;
}
```

DE CE AȘA?

De ce `memo[n] = …` chiar în `return`? Salvăm rezultatul în momentul în care îl aflăm; la următoarea cerere pentru același `n` îl găsim gata. Este programare dinamică „de sus în jos" (memoizare). Varianta „de jos în sus" (tabulare) umple vectorul de la cazurile mici spre cele mari, fără recursivitate; pe aceea o folosim în restul lecției.

Programarea dinamică (PD) = problema mare se împarte în subprobleme care **se repetă**; fiecare se rezolvă o dată, în ordine, iar rezultatele se refolosesc.

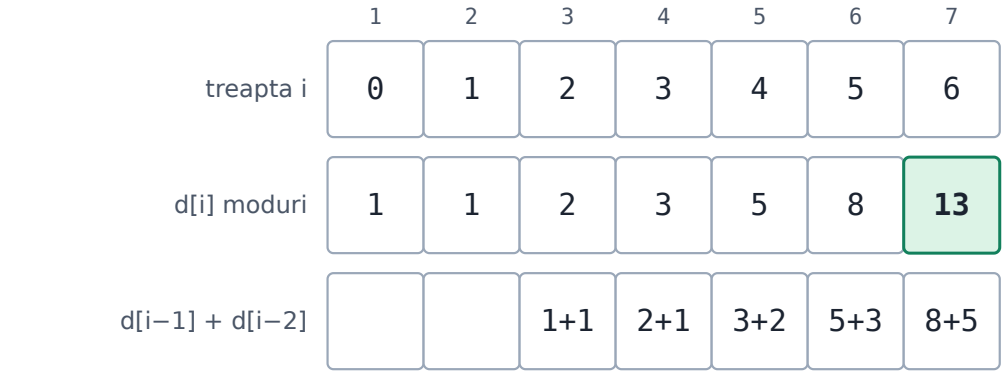
Rețeta în 4 întrebări

1. **Starea:** ce înseamnă `d[i]` (sau `d[i][j]`)? Spune-o în cuvinte.
2. **Recurența:** cum se calculează `d[i]` din valori deja calculate?
3. **Cazurile de bază:** de la ce valori pornești?
4. **Răspunsul:** unde este (de obicei `d[n]` sau maximul din `d`)?

2. Scările

Ai n trepte și urci câte 1 sau 2 trepte o dată. În câte moduri ajungi sus?

- **stare:** `d[i]` = numărul de moduri de a ajunge pe treapta i;
- **recurență:** pe treapta i ajungi de pe i − 1 sau de pe i − 2, deci `d[i] = d[i−1] + d[i−2]`;
- **bază:** `d[0] = 1` (un singur mod: nu urci), `d[1] = 1`.



pe treapta i ajungi de pe i − 1 sau de pe i − 2: adunăm numărul de moduri pentru fiecare

Scări

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

long long d[100];

int main()
{
    int n;
    cin >> n;
    d[0] = d[1] = 1;
    for (int i = 2; i <= n; i++)
        d[i] = d[i - 1] + d[i - 2];
    cout << d[n];
    return 0;
}
```

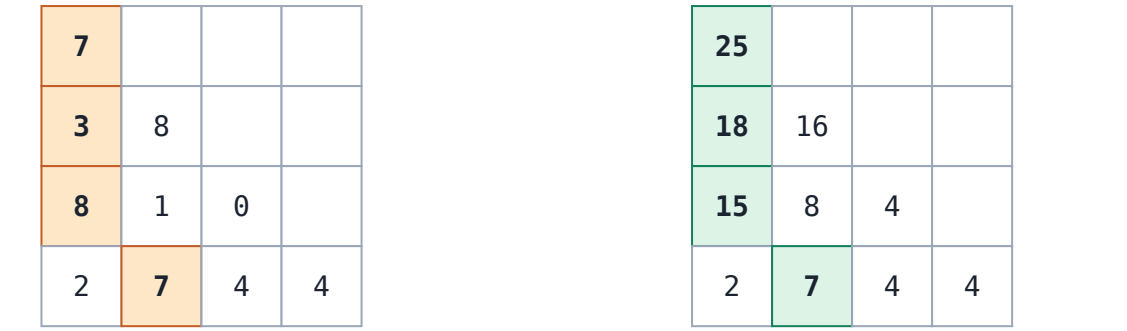
DE CE AȘA?

De ce se adună? Toate modurile de a ajunge pe i se împart în două grupe disjuncte: cele care trec prin i − 1 și cele care trec prin i − 2. Fiecare grupă are `d[...]` moduri, deci în total suma. **De ce `d[0] = 1` și nu 0?** Ca să iasă `d[2] = d[1] + d[0] = 2` (modurile „1+1" și „2"). Dacă `d[0]` ar fi 0, am pierde modul „2".

Dacă poți urca 1, 2 **sau 3** trepte, recurența devine `d[i] = d[i−1] + d[i−2] + d[i−3]` (cu `d[0] = 1`, `d[1] = 1`, `d[2] = 2`); pentru n = 6 se obține 24.

3. Suma maximă în triunghi

Pornești din vârful triunghiului și cobori, la fiecare pas, la una dintre cele două cifre de dedesubt. Care este suma maximă pe un drum?



Triunghi

Dacă pornești de sus, nu știi încotro să o iei. De aceea calculăm **de jos în sus**: `d[i][j]` = suma maximă a unui drum care pornește din celula (i, j) și coboară până jos.

- **recurență:** `d[i][j] = a[i][j] + max(d[i+1][j], d[i+1][j+1])`;
- **bază:** ultima linie: `d[n][j] = a[n][j]`;
- **răspuns:** `d[1][1]`.

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

int a[101][101], n;

int main()
{
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++) cin >> a[i][j];

    for (int i = n - 1; i >= 1; i--)            // de jos în sus
        for (int j = 1; j <= i; j++)
            a[i][j] += max(a[i + 1][j], a[i + 1][j + 1]);

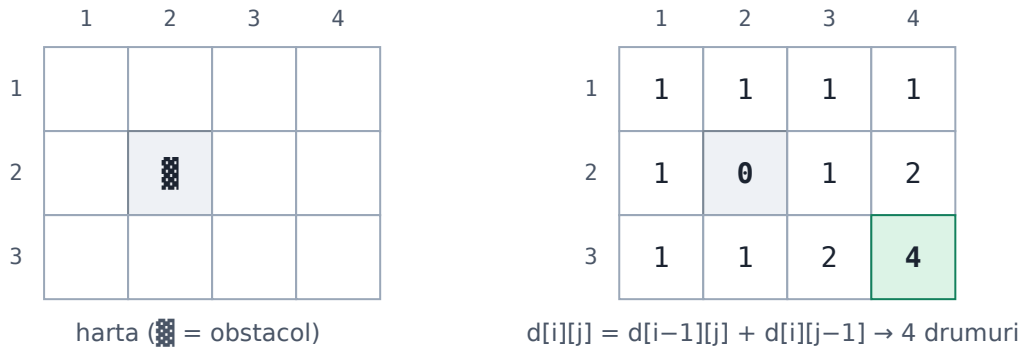
    cout << a[1][1];
    return 0;
}
```

DE CE AȘA?

De ce putem folosi aceeași matrice `a`? Când calculăm linia i, avem nevoie doar de linia i + 1, care e deja finală; valoarea originală `a[i][j]` se adună o singură dată. **De ce nu merge greedy (alege mereu cel mai mare dedesubt)?** Din 7 alege 8 (nu 3), apoi 1 (nu 0), apoi 7: suma 7 + 8 + 1 + 7 = 23, dar drumul 7, 3, 8, 7 are 25. Un pas care pare slab acum poate deschide numere mari mai jos; PD verifică toate drumurile eficient.

4. Drumuri într-o matrice

Câte drumuri sunt de la colțul stânga-sus la colțul dreapta-jos, mergând doar **la dreapta** sau **în jos**, ocolind obstacolele?



Drumuri

- **stare:** `d[i][j]` = numărul de drumuri până în (i, j);
- **recurență:** în (i, j) ajungi din sus sau din stânga: `d[i][j] = d[i−1][j] + d[i][j−1]`; pe obstacol `d = 0`;
- **bază:** `d[1][1] = 1`.

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

int a[101][101], n, m;
long long d[101][101];

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) cin >> a[i][j];      // 1 = obstacol

    d[1][1] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (a[i][j] == 1) d[i][j] = 0;
            else if (i > 1 || j > 1) d[i][j] = d[i - 1][j] + d[i][j - 1];

    cout << d[n][m];
    return 0;
}
```

DE CE AȘA?

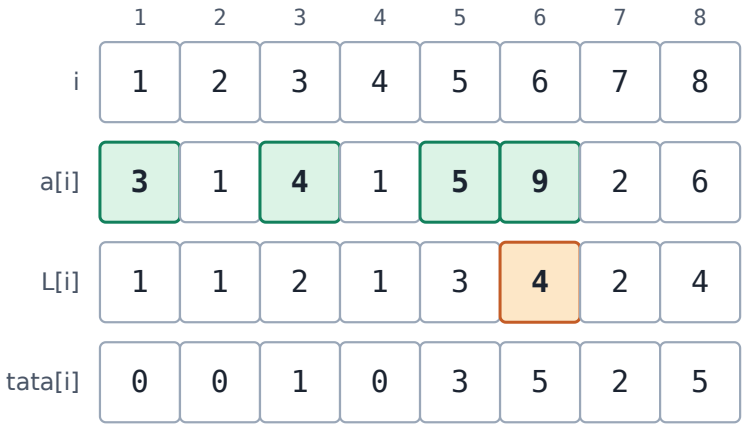
De ce `d[0][j]` și `d[i][0]` nu strică? Sunt zero (global), deci pe prima linie / coloană adunăm doar vecinul existent. **De ce `long long`?** Numărul de drumuri crește exponențial (pentru o matrice 20 × 20 fără obstacole sunt 35 345 263 800).

Varianta „sumă maximă a valorilor întâlnite pe drum”: `d[i][j] = a[i][j] + max(d[i−1][j], d[i][j−1])`.

5. Cel mai lung subșir crescător

Din șirul `3 1 4 1 5 9 2 6` păstrezi elemente (în ordinea lor) astfel încât să crească strict. Care este cel mai lung subșir crescător?

- **stare:** `L[i]` = lungimea celui mai lung subșir crescător care se **termină** în `a[i]`;
- **recurență:** `L[i] = 1 + max { L[j] : j < i, a[j] < a[i] }` (sau 1 dacă nu există niciun j);
- **răspuns:** `max L[i]` (nu neapărat `L[n]`!).



maxim 4: subșirul 3, 4, 5, 9 (verde), reconstruit din tata[] de la poziția 6

SCLM

Ca să afișăm și subșirul, reținem `tata[i]` = poziția anterioară din subșirul optim care se termină în i.

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

int a[1001], L[1001], tata[1001], n;

void afisSubsir(int i)
{
    if (i == 0) return;
    afisSubsir(tata[i]);
    cout << a[i] << ' ';
}

int main()
{
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];

    int best = 0, poz = 0;
    for (int i = 1; i <= n; i++)
    {
        L[i] = 1;
        for (int j = 1; j < i; j++)
            if (a[j] < a[i] && L[j] + 1 > L[i])
            {
                L[i] = L[j] + 1;
                tata[i] = j;
            }
        if (L[i] > best) { best = L[i]; poz = i; }
    }
    cout << best << '\n';
    afisSubsir(poz);
    return 0;
}
```

DE CE AȘA?

De ce „se termină în a[i]" și nu „în primele i elemente"? Pentru a decide dacă `a[i]` poate prelungi un subșir, trebuie să știm cu ce valoare se termină acesta. **Complexitate:** $O(n^2)$; se poate coborî la O(n log n), dar la Bac și la clasă n ≤ 1000 este suficient.

Trace pentru poziția 5 (`a[5] = 5`), cu `a = 3 1 4 1 5 …` :

j	a[j]	a[j] < 5?	L[j] + 1	L[5] devine
(început)				1
1	3	da	2	2 (tata = 1)
2	1	da	2	2 (nu e strict mai mare)
3	4	da	3	3 (tata = 3)
4	1	da	2	3

6. Rucsacul discret (0/1)

Ai n obiecte (greutate `g` , valoare `v`) și un rucsac de capacitate G. Fiecare obiect se ia **întreg sau deloc**. Valoarea maximă? (Cu aceleași date, greedy după V/G greșea: lecția XI.15.)

- **stare:** `d[c]` = valoarea maximă pe care o poți pune într-un rucsac de capacitate c, folosind obiectele văzute până acum;
- **recurență:** pentru obiectul i: `d[c] = max(d[c], d[c – g[i]] + v[i])` (nu-l iau / îl iau);
- **răspuns:** `d[G]` .



răspuns d[5] = 7 (obiectele de 2 kg și 3 kg)

Rucsac

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

int g[101], v[101], d[10001], n, G;

int main()
{
    cin >> n >> G;
    for (int i = 1; i <= n; i++) cin >> g[i] >> v[i];
    for (int i = 1; i <= n; i++)
        for (int c = G; c >= g[i]; c--)                    // DE LA MARE LA MIC
            d[c] = max(d[c], d[c - g[i]] + v[i]);
    cout << d[G];
    return 0;
}
```

ATENȚIE — GREȘELI FRECVENTE

Bucla după `c` merge **descrescător**. Dacă ar merge crescător, `d[c − g[i]]` ar conține deja obiectul i și l-ai putea lua de mai multe ori: ai rezolva alt tip de problemă (rucsac cu repetiție).

DE CE AȘA?

De ce funcționează un singur vector `d` ? `d[c]` se calculează din valori `d[c − g[i]]` cu indice **mai mic**; mergând de la mare la mic, acele valori încă nu au fost modificate de obiectul i, deci reprezintă situația „fără obiectul i".

7. Număr minim de monede

Ai monede de valori $m_1, m_2, \ldots$ (oricâte din fiecare). Care este numărul minim de monede cu care plătești suma S? Pentru {1, 3, 4} greedy greșea (6 = 4+1+1, optim 3+3).

- **stare:** `d[s]` = numărul minim de monede pentru suma s;
- **recurență:** `d[s] = 1 + min(d[s − mₖ])` pentru monedele cu $m_k \le s$;
- **bază:** `d[0] = 0`; o sumă imposibilă rămâne „infinit".

	1	2	3	4	5	6	7
suma s	0	1	2	3	4	5	6
d[s] monede	0	1	2	1	1	2	**2**
din ce monedă		1	1	3	4	1 sau 4	3

d[6] = 2: 3 + 3 (greedy dădea 4 + 1 + 1 = 3 monede)

Monede

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

int mon[101], d[10001], m, S;

int main()
{
    cin >> m;
    for (int i = 1; i <= m; i++) cin >> mon[i];
    cin >> S;
    const int INF = 1000000000;
    for (int s = 1; s <= S; s++) d[s] = INF;              // d[0] = 0
    for (int s = 1; s <= S; s++)
        for (int k = 1; k <= m; k++)
            if (mon[k] <= s && d[s - mon[k]] != INF && d[s - mon[k]] + 1 < d[s])
                d[s] = d[s - mon[k]] + 1;
    cout << (d[S] == INF ? -1 : d[S]);
    return 0;
}
```

DE CE AȘA?

De ce `INF` și nu 0 pentru „imposibil”? 0 înseamnă „zero monede” (suma 0). Un `INF` clar separat ne arată că suma nu se poate forma, iar condiția `d[...] != INF` previne adunarea lui 1 la „infinit”.

8. Greedy sau PD?

	Greedy	Programare dinamică
ideea	alegi local, nu te mai întorci	calculezi toate subproblemele, o singură dată
corectitudine	trebuie demonstrată / contraexemple	decurge din recurență
viteză	foarte rapid	mai lent ($O(n^2)$, $O(n \cdot G)$, ...)
când	criteriu clar (spectacole, rucsac fracționar)	alegerea de acum depinde de viitor (rucsac 0/1, monede oarecare)

9. Greșeli frecvente

Greșeala	Efect	Corect
`d[0] = 0` la scări	rezultate prea mici	`d[0] = 1`
bucla rucsacului crescătoare	obiecte luate de mai multe ori	`c` descrescător
răspunsul la SCLM = `L[n]`	greșit dacă subșirul optim nu se termină în n	`max L[i]`

Greșeala	Efect	Corect
`int` pentru număr de drumuri	depășire	`long long`
`d[s] + 1` cu `d[s] = INF`	depășire sau valori false	testezi `!= INF`
definire neclară a stării	recurență imposibil de scris	scrii în cuvinte ce înseamnă `d[i]`

10. La BAC și la concursuri

> **LA BAC**
>
> Programarea dinamică **nu** este în programa de Bac (Subiectul III cere de obicei simulare, șiruri, matrice, recursivitate sau backtracking), dar apare la olimpiade, la admitere (facultăți de informatică) și la interviuri. Subproblemele de „număr de moduri" (scări, drumuri) sau „maxim/minim" (triunghi, monede) se rezolvă cu aceeași rețetă: stare, recurență, bază, răspuns.

11. Exersează pe hârtie

1. Calculează pe hârtie suma maximă pentru triunghiul din secțiunea 3 și scrie drumul.

2. Pe scări poți urca 1, 2 sau 3 trepte. Scrie recurența și calculează `d[6]`.

3. Monede 1, 3, 4: calculează `d[6]` și arată ce monede folosești.

4. Câte drumuri sunt într-o matrice 3 × 3 fără obstacole (doar dreapta / jos)?

5. Pentru șirul `5 2 8 6 3 6 9 7`, care este lungimea celui mai lung subșir crescător?

6. Rucsac 0/1 de capacitate 6, obiecte (g, v): (2, 3), (3, 4), (4, 5), (5, 6). Valoarea maximă?

7. Dă un exemplu de date pentru care greedy (după V/G) nu dă soluția optimă la rucsacul 0/1.

8. Câte șiruri binare de lungime 5 nu au doi de 1 alăturați? (Hint: ultima cifră este 0 sau 1.)

9. În câte moduri se poate plăti suma 5 cu monede de 1, 2 și 5 (ordinea monedelor nu contează)?

10. Explică, în două propoziții, de ce la rucsacul 0/1 bucla după capacitate merge descrescător.

12. Mai departe (pentru cine vrea mai mult)

Cel mai lung subșir comun (LCS). Pentru două șiruri x și y: `d[i][j]` = lungimea celui mai lung subșir comun al prefixelor `x[1..i]` și `y[1..j]`. Dacă `x[i] == y[j]`, `d[i][j] = d[i−1][j−1] + 1`; altfel `d[i][j] = max(d[i−1][j], d[i][j−1])`.

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

char x[1005], y[1005];
int d[1005][1005];

int main()
{
    cin >> x + 1 >> y + 1;                  // citim de la poziția 1
    int n = strlen(x + 1), m = strlen(y + 1);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (x[i] == y[j]) d[i][j] = d[i - 1][j - 1] + 1;
            else d[i][j] = max(d[i - 1][j], d[i][j - 1]);
    cout << d[n][m];
    return 0;
}
```

Distanța de editare. Câte operații (inserare, ștergere, înlocuire) transformă un cuvânt în altul? Aceeași formă de tabel `d[i][j]`, cu trei variante în recurență.

Optimizarea memoriei. La rucsac se folosește un singur vector, nu o matrice. La drumuri în matrice, dacă ai nevoie doar de răspuns, ajung două linii.

Numărarea modurilor de plată. Pentru „în câte moduri" cu monedele luate în orice ordine, bucla după monede este cea **exterioară**: `for k … for s = mon[k]..S: d[s] += d[s − mon[k]]`.

Rezumat

- PD: subprobleme care se repetă, rezolvate o singură dată; patru întrebări: stare, recurență, bază, răspuns.
- Scări: `d[i] = d[i−1] + d[i−2]`. Triunghi: de jos în sus. Drumuri: `d[i−1][j] + d[i][j−1]`.
- SCLM: `L[i] = 1 + max L[j]` cu `a[j] < a[i]`; răspunsul este `max L[i]`.
- Rucsac 0/1: `c` descrescător; monede minime: `d[s] = 1 + min d[s − mₖ]`.

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] **#1991** **trepte2** — ușoară
  https://www.pbinfo.ro/probleme/1991/trepte2
- [ ] **#385** **sumtri** — ușoară
  https://www.pbinfo.ro/probleme/385/sumtri
- [ ] **#396** **sclm** — ușoară
  https://www.pbinfo.ro/probleme/396/sclm

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] #386 **sumtri1** — ușoară
  https://www.pbinfo.ro/probleme/386/sumtri1
- [ ] #392 **cladire** — ușoară
  https://www.pbinfo.ro/probleme/392/cladire
- [ ] #395 **comori** — ușoară
  https://www.pbinfo.ro/probleme/395/comori
- [ ] #4186 **monede** — ușoară
  https://www.pbinfo.ro/probleme/4186/monede
- [ ] #1886 **rucsac1** — medie
  https://www.pbinfo.ro/probleme/1886/rucsac1
- [ ] #2528 **lungimesubsircomunmaximal** — ușoară
  https://www.pbinfo.ro/probleme/2528/lungimesubsircomunmaximal

Pentru cei care vor mai mult

Probleme mai grele sau de concurs, pentru cine vrea să meargă dincolo de BAC.

- [ ] #3655 **distanta-de-editare** — ușoară
  https://www.pbinfo.ro/probleme/3655/distanta-de-editare

Citește și exersează în plus

- Programare dinamică — introducere https://www.pbinfo.ro/articole/17951/programare-dinamica-introducere
- Numărul de drumuri în matrice https://www.pbinfo.ro/articole/20522/numarul-de-drumuri-in-matrice
- Subșir crescător de lungime maximă https://www.pbinfo.ro/articole/20677/subsir-crescator-de-lungime-maxima
- USACO Guide — Introduction to Dynamic Programming — aceleași probleme clasice, cu explicații și exerciții de concurs (engleză) https://usaco.guide/gold/intro-dp
- CSES — Dynamic Programming (listă de probleme) — Dice Combinations, Coin Combinations, Minimizing Coins, Grid Paths: aceleași idei ca în lecție (engleză) https://cses.fi/problemset/list/
- Programare dinamică — Wikipedia — origine, principiul optimalității, exemple https://ro.wikipedia.org/wiki/Programare_dinamică

Răspunsuri

Exersează pe hârtie

1. De jos în sus: 15, 8, 4; apoi 18, 16; apoi 25. Drumul: 7, 3, 8, 7. Suma maximă este **25**.

2. `d[i] = d[i−1] + d[i−2] + d[i−3]`, `d[0] = 1, d[1] = 1, d[2] = 2`: d[3] = 4, d[4] = 7, d[5] = 13, d[6] = **24**.

3. `d[0..6] = 0, 1, 2, 1, 1, 2, 2`: **d[6] = 2**, monedele 3 + 3.

4. Fără obstacole: linia 1: 1 1 1; linia 2: 1 2 3; linia 3: 1 3 6. Răspuns **6** (C(4,2)).

5. `L = 1 1 2 2 2 3 4 4`, deci **4** (de exemplu 2, 3, 6, 9).

6. Cele mai bune combinații: obiectele (2, 3) și (4, 5), cu greutate 6 și valoare **8**.

7. Capacitate 50, obiecte (10, 60), (20, 100), (30, 120): greedy după V/G ia primele două (160); optimul ia ultimele două: 220.

8. Fie `d[i]` numărul de șiruri de lungime i. Dacă șirul se termină în 0, înainte poate fi orice șir corect de lungime i − 1; dacă se termină în 1, penultima cifră este 0, deci înainte este un șir corect de lungime i − 2. `d[i] = d[i−1] + d[i−2]`, cu `d[1] = 2`, `d[2] = 3`: d[3] = 5, d[4] = 8, d[5] = **13**.

9. Moduri: 5; 2+2+1; 2+1+1+1; 1+1+1+1+1 → **4**.

10. `d[c]` se calculează din `d[c − g[i]]`, care trebuie să reprezinte situația **fără** obiectul i. Mergând de la capacități mari la mici, acele valori nu au fost încă actualizate cu obiectul i, deci obiectul se folosește cel mult o dată.
