Programare dinamică

- Soluția problemei de dimensiune mare este exprimată în funcție de soluțiile unor subprobleme de dimensiune mai mică (soluție recursivă)
- Comportament exponențial din cauza recalculării subproblemelor
  => Se reține un **tabel cu soluțiile tuturor subproblemelor care au fost deja rezolvate**
  - acest tabel **se completează bottom-up** (de la subprobleme mici la subprobleme mari)
  - prin convenție, vom numi mereu acest tabel dp (de la dynamic programming)

SSM (Subsecvența de sumă maximă)

v[1..n] de numere întregi

Care e suma maximă care se poate obține pentru o subsecvență nevidă (sij = v[i] + v[i+1] + … + v[j]) ?

Exemplu:

-1 3 2 -4 5 -3 -6 2

Suma maximă: 6, pentru subsecvența 3 2 -4 5

Raționament:

- Căutăm un tipar de genul: „soluția pentru vectorul mai mare v[1..i] depinde de soluțiile pentru vectorii mai mici v[1..1], v[1..2] ... v[1..i-1]”
- Încercăm: dp[i] = soluția (suma maximă) pentru v[1..i]
- Rezultă că dp[1] = v[1] (nu avem de ales, avem o singură subsecvență)
- Este util? Se poate exprima dp[i] doar în funcție de subprobleme deja calculate? Cum?
  - dp[i] = ...

- Avem o problemă! Subsecvențele se bazează pe elemente consecutive în vector; degeaba știm cea mai bună soluție din v[1..i-1] dacă nu știm cea mai bună soluție din v[1..i-1] pe care v[i] ar putea-o îmbunătăți, adică **cea mai bună soluție din v[1..i-1] care se termină pe poziția i-1**

- Redefinim: dp[i] = soluția (suma maximă) pentru v[1..i] care îl conține pe v[i]
  dp[1] = v[1] (nu avem de ales, avem o singură subsecvență)

Să vedem împreună dacă acum reușim să construim dp:

v	-1	3	2	-4	5	-3	-6	2
dp	-1	3	5	1	6	3	-3	2
unde_incepe	1	2	2	2	2	2	2	8

Recurența este:

dp[1] = v[1] (caz de bază)

dp[i] = max(v[i], v[i] + dp[i-1])

Rezultatul final al problemei se va găsi în: max(dp[k]), k = 1:n

SCMAX (Subșirul crescător de lungime maximă)

v[1..n] de numere întregi

Care e lungimea maximă care se poate obține pentru un subșir ordonat strict crescător?

Exemplu:

-3	3	-2	-1	5	-3	6	4	5

Lungimea maximă: 5, pentru (de exemplu) subșirul -3 -2 -1 4 5

Raționament:

- Încercăm un tipar similar celui de la problema anterioară: întrucât există din nou o relație între numerele care trebuie selectate în soluție (la problema anterioară ele trebuia să fie consecutive în v, aici ele trebuie să fie ordonate strict crescător), vom prefera din nou varianta în care știm cu ce număr se termină soluția
- Așadar încercăm: dp[i] = soluția (lungimea maximă) pentru v[1..i] care îl conține pe v[i]
- Rezultă că dp[1] = 1 (avem un singur subșir de 1 element)

Să vedem împreună dacă reușim să construim dp:

v	-3	3	-2	-1	5	-3	6	4	5
dp	1	2	2	3	4	1	5	4	5
pred	---	1	1	3	4	---	5	4	8

Recurența este:

dp[1] = 1 (caz de bază)

dp[i] = 1 + max(dp[k]), k = 1:i-1, v[i] > v[k]

Rezultatul final al problemei se va găsi în: max(dp[k]), k = 1:n

Rucsac (Prețul maxim care se poate obține pentru o serie de obiecte care încap în rucsac)

v[1..n] de obiecte caracterizate prin *weight* și *price*

rucsac care suportă o greutate maximă W

Care e prețul maxim care se poate obține pentru o submulțime de obiecte care încap în rucsac?

Exemplu (fiecare obiect e reprezentat prin perechea (weight, price)):

(3,6) (3,3) (1,2) (1,8) (2,5)

W = 3

Prețul maxim: 13, pentru submulțimea (1,8) (2,5)

Raționament:

- Aici nu mai există nicio relație între diversele obiecte pe care le selectăm în soluție, motiv pentru care încercăm dp[i] = soluția (prețul maxim) pentru v[1..i]
- dp[0] = 0 (nu există profit cât timp nu adăugăm niciun obiect în rucsac)
- Este util? Se poate exprima dp[i] doar în funcție de subprobleme deja calculate? Cum?
  - dp[i] = ...

- Avem o problemă! Degeaba știm cea mai bună soluție din v[1..i-1] dacă nu știm cea mai bună soluție din v[1..i-1] pe care v[i] ar putea-o îmbunătăți, adică **cea mai bună soluție din v[1..i-1] care îi permite obiectului v[i] să fie adăugat (să încapă) în rucsac**
- Pentru a avea această informație, avem nevoie de o nouă dimensiune: nu ajunge să știm cea mai bună soluție din v[1..i], ci ne trebuie să știm care este cea mai bună soluție din v[1..i] care se încadrează într-o anumită greutate w, și avem nevoie să știm asta pentru toate greutățile intermediare!

- Redefinim: dp[i][w] = soluția (prețul maxim) pentru v[1..i] care încape în greutatea w
  dp[0][w] = 0, pt w = 0 : W (0 obiecte înseamnă 0 profit)
  dp[i][0] = 0, pt i = 1 : n (0 greutate înseamnă că niciun obiect nu va încăpea)

Să vedem împreună dacă reușim să construim matricea dp:

v\w	0	1	2	3
---	0	0	0	0
(3,6)	0	0	0	6
(3,3)	0	0	0	6
(1,2)	0	2	2	6
(1,8)	0	8	10	10
(2,5)	0	8	10	13 = max(5+8, 10)

<u>Recurența</u> este:

dp[][] = ... (caz de bază)

dp[i][w] = (maximul dintre cazul in care aleg v[i] si cazul in care nu aleg v[i])

= max(v[i].price + dp[i-1][w – v[i].weight], dp[i-1][w])

<u>Rezultatul final</u> al problemei se va găsi în: dp[n][W]

Monede (numărul minim de monede cu care pot obține suma S (sau -1 dacă S nu se poate obține))

v[1..n] tipuri de monede

suma S trebuie obținută folosind (de oricâte ori) doar monede din v

Care e numărul minim de monede din v cu care se poate atinge exact suma S?

Exemplu:

1 2 5

S = 11

Numărul minim: 3, pentru monedele 1 5 5 (1 + 5 + 5 = 11)

Raționament:

- La problemele anterioare aveam câte un vector de elemente și pentru fiecare nou element ne puneam problema dacă selectarea lui ar putea îmbunătăți soluția
- Problema aici este că nu ajunge să ne hotărâm dacă selectăm un element sau nu, ci și de câte ori ar fi necesar elementul în soluția finală
- Altă observație este că nu încercăm să optimizăm suma la care ajungem, ci avem de atins o sumă fixă (ceea ce optimizăm este calea către această sumă fixă)
- Când avem de atins o țintă fixă, este util să ne întrebăm care sunt drumurile care ne pot duce la această țintă
    - Cum ar putea arăta pasul imediat anterior ajungerii la destinație?
    - Dar pasul anterior acestuia?

- Pe exemplul nostru, când avem monede cu valoarea 1, 2, respectiv 5, observăm că nu putem ajunge la S = 11 decât dacă putem ajunge la una din următoarele:
    - S = 10, după care luăm o monedă de valoare 1
    - S = 9, după care luăm o monedă de valoare 2
    - S = 6, după care luăm o monedă de valoare 5

- Așadar încercăm: dp[s] = soluția (numărul minim/-1) pentru a atinge suma s
- Rezultă că dp[0] = 0 (suma 0 se obține nefolosind nicio monedă)

Să vedem împreună dacă reușim să construim dp:

s	0	1	2	3	4	5	6	7	8	9	10	11
dp	0	1	1	2	2	1	2	2	3	3	2	3

Recurența este:

dp[0] = 0 (caz de bază)

dp[s] = 1 + min(dp[s-v[k]]), k = 1:n (atentie la cazul când toate întorc -1)

Rezultatul final al problemei se va găsi în: dp[S]

CMLSC (Cel mai lung subșir comun)

v[1..n] de numere întregi

w[1..m] de numere întregi

Care e cel mai lung subșir comun care apare în cei 2 vectori?

(Pentru aceasta, se va începe prin a determina care e lungimea maximă a CMLSC)

Exemplu:

1	2	3	2	
2	4	3	1	2

CMLSC: 2 3 2, de lungime 3

Raționament:

- Observăm că nu există o relație între numerele din v (sau din w) care trebuie selectate în soluție
- Încercăm o recurență de tipul dp[i] = soluția pentru v[1..i] (ca în cazul SSM sau SCMAX), extinsă pentru 2 vectori (întrucât acum avem de făcut alegeri din 2 vectori)
- Așadar încercăm: dp[i][j] = soluția (lungimea maximă) pentru v[1..i] și w[1..j]
- Rezultă că
    - dp[0][j] = 0, j = 0:m (dacă v este vid, nu poate avea un subșir nevid comun cu w)
    - dp[i][0] = 0, i = 0:n (dacă w este vid, nu poate avea un subșir nevid comun cu v)

Să vedem împreună dacă reușim să construim matricea dp:

v\w	---	2	4	3	1	2
---	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	1	1	1	1	2
3	0	1	1	2	2	2
2	0	1	1	2	2	3

Cum reconstituim CMLSC pe baza dp?

2 3 2

Recurența este:

dp[0][j] = 0, j = 0:m (caz de bază)

dp[i][0] = 0, i = 0:n (caz de bază)

dp[i][j] = 1 + dp[i-1][j-1], dacă v[i] = w[j]

max(dp[i-1][j], dp[i][j-1]), altfel

Rezultatul final al problemei se va găsi în: dp[n][m]