

1.

2.

3.

4.

## 5. *SumaDinámica*

5.a.

$$ss'_C s(i, j) = \begin{cases} j = 0 & \text{si } i = 0 \\ ss'_C(i - 1, j) & \text{si } i \neq 0 \wedge C[i] > j \\ ss'_C(i - 1, j) \vee ss'_C(i - 1, j - C[i]) & \text{c.c.} \end{cases}$$

Es igual al 1.f, nada más que acotar su señoría

$O(2^n)$  llamadas recursivas son suficientes ya que son todas las combinaciones :v

5.b.

En papel

5.c.

Convencido! Es  $O((n + 1)(k + 1)) \in O(nk)$

5.d.

$$k \gg 2^n \implies O(nk) \gg O(2^n)$$

$$k \ll 2^n \implies O(nk) \ll O(2^n)$$

5.e.

Si

## 6. *OptiPago*

6.a.

$$cc(B, c) = \begin{cases} (0, 0) & \text{if } c = 0 \\ (\infty, \infty) & \text{if } B = \{\} \wedge c > 0 \\ (b_n, 1) & \text{if } b_n = c \\ \min_{c,q} (cc(B - \{b_n\}, c - b_n) + (b_n, 1), cc(B - \{b_n\}, c)) & \text{c.c.} \end{cases}$$

Para  $n = |B|$ , solve con  $cc(B, c, 0)$

6.b.

Queda de ejercicio al lector

### 6.c.

$$\text{cc}_B(i, j) = \begin{cases} (0, 0) & \text{if } j = 0 \\ (\infty, \infty) & \text{if } i = 0 \wedge j > 0 \\ (B[i], 1) & \text{if } B[i] = j \\ \min_{i,j}(\text{cc}_B(i-1, j-B[i]) + (B[i], 1), \text{cc}_B(i-1, j)) & \text{c.c.} \end{cases}$$

solve con  $\text{cc}_B(|B|, c)$

### 6.d.

Una matriz  $M$  de  $n \times c$ , se crea en  $O(nc)$

### 6.e.

Lo mismo que el **e.** pero antes de recursionar chequea si existe  $M[i-1, j]$  y  $M[i-1, j-B[i]]$ , si no existe, lo calcula y guarda, luego lo retorna en ambos casos.

### 6.f.

$\text{cc}_B(|B|, c)$ , la complejidad es  $\Omega(nc)$  y  $O(nc)$ , que es peor que  $\Omega(1)$  pero mejor que  $O(2^n)$

### 6.g.

Para  $B, c$  y  $n = |B|$ :

```
cc(i, j):
1   $M[n \times c] \leftarrow \forall i, j :: M[i][j] = (\infty, \infty)$ 
2  let res  $\leftarrow (\infty, \infty)$ 
3  if  $j = 0$ :
4      return  $(0, 0)$ 
5  for  $i \leftarrow 1$  to  $n$ :
6      if  $B[i-1] = c$ :
7          return  $(B[i-1], 1)$ 
8      for  $j \leftarrow 1$  to  $c$ :
9           $M[i][j] = \min(M[i-1][j-B[i]] + (B[i], 1), M[i-1][j])$ 
10 return  $M[n][c]$ 
```

## 7. AstroTrade

### 7.a.

hecho xd

7.b.

$$AT_p(j, c) \begin{cases} -\infty & \text{if } c < 0 \vee c > j \\ 0 & \text{if } j = 0 \wedge c = 0 \\ \max(AT_p(j-1, c-1) - p_j, AT_p(j-1, c+1) + p_j, AT_p(j-1, c)) & \text{c.c.} \end{cases}$$

7.c.  $AT_p(n, 0)$

7.d.

```
memo = [[None for _ in range(n+1)] for _ in range(n+1)]
```

```
def at(j, c):
    if memo[j][c] != None:
        return memo[j][c]

    if c < 0 or c > j:
        return float('-inf')

    if j == 0 and c == 0:
        return 0

    memo[j][c] = max(
        at(j-1, c-1) - p[j],
        at(j-1, c+1) + p[j],
        at(j-1, c)
    )

    return memo[j][c]
```

La complejidad espacial es  $\Theta(n^2)$  ya que en el peor caso,  $c = n$  y debo memoizar para cada  $i \leq n$ , la complejidad temporal será la misma que la espacial, lo que es una mejora a comparación de  $O(3^n)$  en el caso sin memoizar.

7.e.

```
memo = [[None for _ in range(n+1)] for _ in range(n+1)]
```

```
def atbt(p):
    memo[0][0] = 0

    for j in range(1, n+1):
        for c in range(0, j+1):
            if c < 0 or c > j:
                memo[j][c] = float('-inf')
            else:
                v1 = memo[j-1][c-1] if c-1 >= 0 else float('-inf')
                v2 = memo[j-1][c+1] if c+1 <= j-1 else float('-inf')
                v3 = memo[j-1][c] if memo[j-1][c] else float('-inf')

                memo[j][c] = max(v1-p[j], v2+p[j], v3)

    return memo[n][0]
```

## 8. Cortes Económicos

### 8.a.

convencido(?)

### 8.b.

$$CE(i, j, C) = \begin{cases} 0 & \text{if } C = \{\} \\ \min(CE(i, c, C - \{c\}), CE(c, j, C - \{c\}) \mid \forall c \in C, i \leq c < j) + (j - i) & \text{c.c} \end{cases}$$

Se explica por si misma, se resuelve con  $CE(0, \ell)$  donde  $C$  son los cortes posibles

### 8.c.

memo = {}

```
def ce(i, j, C):
    if (i, j) in memo:
        return memo[(i, j)]

    res = []
    if not C:
        return 0

    for c in C:
        s1 = ce(i, c, [k for k in C if i <= k < c])
        s2 = ce(c, j, [k for k in C if c < k < j])
        res.append(s1+s2+(j-i))

    memo[(i, j)] = min(res)
    return memo[(i, j)]
```

Será  $O(n^3)$  temporal y  $O(n^3)$  espacial. Un bottom-up costaría lo mismo temporal y  $(n^2)$  espacial ya que no memoizamos cada k

## 9. Travesía Vital

### 9.a.

$$TV_A(i, j) = \begin{cases} 1 & \text{if } TV'_A(i, j) < 1 \\ TV'_A(i, j) & \text{c.c.} \end{cases}$$
$$TV'_A(i, j) = \begin{cases} A_{i, j} & \text{if } i = n \wedge j = m \\ A_{i+1, j} + TV'_A(i+1, j) & \text{if } (A_{i+1, j} < A_{i, j+1} \wedge i < n) \vee j = m \\ A_{j, i+1} + TV'_A(i, j+1) & \text{if } (A_{i+1, j} \geq A_{i, j+1} \wedge j < m) \vee i = n \end{cases}$$

está mal pero bueno, a corregir por el lector!

### 9.b.

si

### 9.c.

$$TV_A(i, j) \begin{cases} TV_{\max_A}(1 - A_{i,j}) & \text{if } i = n \wedge j = m \\ TV_{\max_A}(TV_A(i, j+1) - A_{i,j}) & \text{if } i = n \\ TV_{\max_A}(TV_A(i+1, j) - A_{i,j}) & \text{if } j = m \\ TV_{\max_A}(\min(TV_A(i+1, j), TV_A(i, j+1)) - A_{i,j}) & \text{c.c.} \end{cases}$$

$$TV_{\max_A}(f) = \max(1, f)$$

Se resuelve con  $TV_A(1, 1)$

### 9.d.

```
A=[
    [0,0,0,0],
    [0,-2,-3,3],
    [0,-5,-10,1],
    [0,10,30,-5]]

n = len(A[0])-1
m = len(A)-1

memo = {}

def TV(i, j):
    if (i,j) in memo:
        return memo[(i,j)]

    if i == n and j == m:
        memo[(i,j)] = TVmax(1-A[i][j])
        return memo[(i,j)]

    if i == n:
        memo[(i,j)] = TVmax(TV(i,j+1) - A[i][j])
        return memo[(i,j)]

    if j == m:
        memo[(i,j)] = TVmax(TV(i+1,j) - A[i][j])
        return memo[(i,j)]

    memo[(i,j)] = TVmax(min(TV(i+1,j), TV(i,j+1)) - A[i][j])
    return memo[(i,j)]

def TVmax(f):
    return max(1,f)

print(TV(1,1))
```

La complejidad temporal será  $O(nm)$  y la espacial  $O(nm)$ , frente a un bottom-up que probablemente lo unico que cambiaría sería la recursión por dos bucles anidados hasta n y m respectivamente.

### 9.e.

No cumple con la complejidad pero una solución B-U. Una posible solución es usar un while hasta n,m e ir incrementando i o j en base a la decisión

```

A = [
    [0, 0, 0, 0],
    [0, -2, -3, 3],
    [0, -5, -10, 1],
    [0, 10, 30, -5]
]

n = len(A)-1
m = len(A[0])-1

memo = [[float('inf')] * (m+1) for _ in range(n+1)]

def TV(i, j):
    memo[n][m] = TVmax(1-A[n][m])

    for i in range(n-1, -1, -1):
        memo[i][m] = TVmax(memo[i+1][m] - A[i][m])

    for j in range(m-1, -1, -1):
        memo[n][j] = TVmax(memo[n][j+1] - A[n][j])

    for i in range(n-1, -1, -1):
        for j in range(m-1, -1, -1):
            memo[i][j] = TVmax(min(memo[i+1][j], memo[i][j+1]) - A[i][j])

    return memo[1][1]

def TVmax(f):
    return max(1, f)

print(TV(1, 1))

```

## 10. PilaCauta

qvq  $s_k - s_i \geq 0 \wedge s_k - s_i \geq w_i$  cada paso, partiendo de  $i = n \wedge C = \{\}$  para cada  $k \neq i$  y en cada paso hacer  $C = C \cup \{i\}$

Primero defino  $n = |S|$

- Parto de  $i = n \wedge C = \{\}$
- Devolver 1 si  $|C| = n$
- $\forall c \neg \in C \mid 1 \leq c \leq n :: 1 + \max(C - \{c\})$  tal que valga  $s_c - s_i \geq 0 \wedge s_c - s_i \geq w_i$

$$pc(i, C) = \begin{cases} 0 & \text{if } |C| = n \\ \max(1 + pc(c, C \cup \{c\}) \mid c \neg \in C \wedge s_c - s_i \geq 0 \wedge s_c - s_i \geq w_i) & \text{if } |C| < n \end{cases}$$

### 10.a. LA POSTA! (al final fracasé en el intento)

#### 10.b.

$$pc_{W,S}(i, p) = \begin{cases} 0 & \text{if } i < n \\ \max(pc(i+1, p)) & \text{if } s_i \geq p \wedge s_i - p \geq w_i \\ pc(i+1, p) & \text{if } p > s_i \end{cases}$$

Para  $i = 1, p = 0$

La posta nada..

## 11. OperacionesSeq

### 11.a.

$$\text{os}_{\text{res}}(i, w, C) = \text{reverse}(\text{os}_v(i, w, C))$$

$$\text{os}_v(i, w, C) = \begin{cases} \emptyset & \text{if } |C| = n \wedge v_i \neq w \\ C & \text{if } |C| = n \wedge v_i = w \\ \text{any}(\alpha, \beta, \gamma \mid \text{if } C \neq \emptyset \text{ else } \emptyset) & \text{if } |C| \neq n \end{cases}$$

$$\alpha = \text{os}(i-1, w - v_i, C \oplus [+])$$

$$\beta = \text{os}\left(i-1, \frac{w}{v_i}, C \oplus [x]\right)$$

$$\gamma = \text{os}(i-1, \sqrt[n]{w}, C \oplus [\uparrow])$$

Esto se resuelve para  $\text{os}_{\text{res}}(n, w, \emptyset)$

### 11.b.

```
v = [0,3,1,5,2,1]
n = len(v)-1
C = ""
memo = {}
def os(i,w,C):
    if (i,w) in memo:
        return memo[(i,w)]

    if len(C) == n-1 and w != v[i]:
        return ""

    if len(C) == n-1 and w == v[i]:
        return C

    res1 = os(i-1,w-v[i],C + "+")
    res2 = os(i-1,w/v[i],C + "x")
    res3 = os(i-1,w*(1/v[i]),C + "↑")

    if res1:
        memo[(i,w)] = res1
    elif res2:
        memo[(i,w)] = res2
    elif res3:
        memo[(i,w)] = res3
    else:
        memo[(i,w)] = ""

    return memo[(i,w)]

def os_res(i,w,C):
    return os(i,w,C)[::-1] #sale al revés porque obviamente es recursivo

print(os_res(n,400,C))
```

Tenemos 3 recursiones, es claramente  $O(3^n)$  tanto temporal como espacial sin memo,  $O(nw)$  tanto espacial como temporal con memo, esto es porque hacemos  $O(nw)$  llamadas recursivas

Si fuese bottom-up sería iterando de 1 a n y para cada uno, de 1 a w, con complejidad tanto espacial como temporal idéntica a la top-down

**11.c.**