1.

2.

3.

4.

5. *SumaDinámica*

5.a.

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

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.

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

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

6.b.

Queda de ejercicio al lector

6.c.

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

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

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.

$\mathrm{cc}_B(|B|, c)$, 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 $n = |B|$:

<u>CC</u>(i, j):

$M[n \times c] \leftarrow \forall i,j :: M[i][j] = (\infty,\infty)$
let res $\leftarrow (\infty,\infty)$
if $j = 0$:
    return $(0,0)$
for $i \leftarrow 1$ **to** n:
    if $B[i-1] = c$:
        return $(B[i-1], 1)$
    for $j \leftarrow 1$ **to** c:
        $M[i][j] = \min(M[i-1][j-B[i]] + (B[i],1), M[i-1][j])$
return M[n][c]

7. AstroTrade

7.a.

hecho xd

7.b.

$$\mathrm{AT}_p(j,c)\begin{cases}-\infty & \text{if } c<0 \lor c>j\\ 0 & \text{if } j=0 \land c=0\\ \max\big(\mathrm{AT}_p(j-1,c-1)-p_j, \mathrm{AT}_p(j-1,c+1)+p_j, \mathrm{AT}_p(j-1,c)\big) & \text{c.c.}\end{cases}$$

7.c. $\mathrm{AT}_p(n,0)$

7.d.

```python
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.

```python
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]
```