Programación dinámica (y su relación con *backtracking*)

5. En este ejercicio vamos a resolver el problema de suma de subconjuntos usando la técnica de programación dinámica.

a) Sea $n = |C|$ la cantidad de elementos de C. Considerar la siguiente función recursiva $\text{ss'}_C \colon \{0, \dots, n\} \times \{0, \dots, k\} \to \{V, F\}$ (donde V indica verdadero y F falso) tal que:

$$\text{ss'}_C(i,j) = \begin{cases} j = 0 & \text{si } i = 0 \\ \text{ss'}_C(i-1, j) & \text{si } i \neq 0 \wedge C[i] > j \\ \text{ss'}_C(i-1, j) \vee \text{ss'}_C(i-1, j - C[i]) & \text{si no} \end{cases}$$

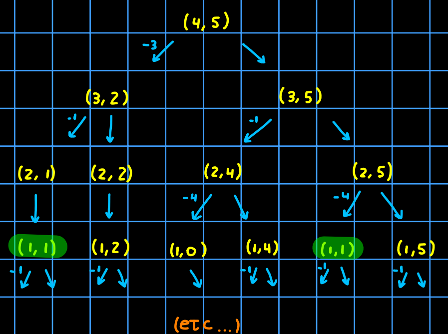
Convencerse de que esta es una definición equivalente de la función ss del inciso *e*) del Ejercicio 1, observando que $\text{ss}(C, k) = \text{ss'}_C(n, k)$. En otras palabras, convencerse de que el algoritmo del inciso *f*) es una implementación por *backtracking* de la función ss'_C. Concluir, pues, que $\mathcal{O}(2^n)$ llamadas recursivas de ss'_C son suficientes para resolver el problema.

En cada llamado recursivo estamos decidiendo si incluir o no al subconjunto i, por lo que hay 2^n soluciones posibles

⇒ Me convence

b) Observar que, como C no cambia entre llamadas recursivas, existen $\mathcal{O}(nk)$ posibles entradas para ss'_C. Concluir que, si $k \ll 2^n/n$, entonces necesariamente algunas instancias de ss'_C son calculadas muchas veces por el algoritmo del inciso *f*). Mostrar un ejemplo donde se calcule varias veces la misma instancia.

C = {3, 1, 4, 1}
k = 5



⇒ La instancia (1, 1) se está calculando 2 veces

c) Considerar la estructura de memoización (i.e., el diccionario) M implementada como una matriz de $(n+1) \times (k+1)$ tal que $M[i, j]$ o bien tiene un valor indefinido $\perp$ o bien tiene el valor $\text{ss'}_C(i, j)$, para todo $0 \leq i \leq n$ y $0 \leq j \leq k$. Convencerse de que el siguiente algoritmo *top-down* mantiene un estado válido para M y computa $M[i, j] = \text{ss'}_C(i, j)$ cuando se invoca $\text{ss'}_C(i, j)$.

1) Inicializar $M[i, j] = \perp$ para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.
2) subset_sum(C, i, j): // implementa $\text{ss}(\{c_1, \dots, c_i\}, j) = \text{ss'}_C(i, j)$ usando memoización
3) Si $j < 0$, retornar **falso**
4) Si $i = 0$, retornar $(j = 0)$
5) Si $M[i, j] = \perp$:
6) Poner $M[i, j] = \text{subset\_sum}(C, i-1, j) \vee \text{subset\_sum}(C, i-1, j - C[i])$
7) Retornar $M[i, j]$

⇒ Me convence

d) Concluir que subset_sum(C, n, k) resuelve el problema. Calcular la complejidad y compararla con el algoritmo subset_sum del inciso *f*) del Ejercicio 1. ¿Cuál algoritmo es mejor cuando $k \ll 2^n$? ¿Y cuándo $k \gg 2^n$?

Con éste algoritmo vamos a resolver únicamente n.k subproblemas

¿Cuál es la complejidad temporal?

- La complejidad espacial es $O(n.k)$
- La complejidad temporal es ???

Cuando $k \ll 2^n$ conviene el algoritmo con memoization, mientras que cuando $k \gg 2^n$ conviene el algoritmo sin.

e) Supongamos que queremos computar todos los valores de M. Una vez computados, por definición, obtenemos que

$$M[i,j] \overset{\text{def}}{=} \text{ss'}_C(i,j) \overset{\text{ss'}}{=} \text{ss'}_C(i-1,j) \vee \text{ss'}_C(i-1,j-C[i]) \overset{\text{def}}{=} M[i-1,j] \vee M[i-1,j-C[i]]$$

cuando $i > 0$, asumiendo que $M[i-1, j-C[i]]$ es falso cuando $j - C[i] < 0$. Por otra parte, $M[0,0]$ es verdadero, mientras que $M[0,j]$ es falso para $j > 0$. A partir de esta observación, concluir que el siguiente algoritmo *bottom-up* computa M correctamente y, por lo tanto, $M[i,j]$ contiene la respuesta al problema de la suma para todo $\{c_1, \ldots, c_i\}$ y j.

1) subset_sum(C, k): // computa $M[i,j]$ para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.
2) Inicializar $M[0,j] := (j = 0)$ para todo $0 \leq j \leq k$.
3) Para $i = 1, \ldots, n$ y para $j = 0, \ldots, k$:
4) Poner $M[i,j] := M[i-1,j] \vee (j - C[i] \geq 0 \wedge M[i-1, j-C[i]])$

$$M = \begin{bmatrix} V & F & F & \cdots & F \\ \perp & \perp & \perp & \cdots & \perp \\ & & & \vdots & \\ \perp & \perp & \perp & \perp & \perp \end{bmatrix}$$

(Estado inicial)

$\Rightarrow$ Me convence

f) (Opcional) Modificar el algoritmo *bottom-up* anterior para mejorar su complejidad espacial a $O(k)$.

Obs: La fila i es accedida únicamente para formar la fila i+1, entonces para generar la instancia a_{i+1} sólo necesito la info de a_i

Sin embargo, yo no puedo trabajar únicamente con una sola fila xq mi objetivo es acceder a c/elem de la matriz en $O(1)$. → Nop, creo queremos computar M aunque no se guarde en ningún lado

Voy a usar 2 arreglos, uno para describir la instancia a_i y el otro para a_{i-1}

El algoritmo: subset_sum(C, k)

Inicializar $A_{i-1}[j] := (j == 0)$ para todo $0 \leq j \leq k$

Para $i = 1, \ldots, n$ y para $j = 0, \ldots, k$

Poner $A_i := A_{i-1}[j] \vee (j - C[i] \geq 0 \wedge A_{i-1}[j - C[i]])$

$A_{i-1} := A_i$

Si fuera intercambiando los arrays en cada iteración (por ejemplo, para i par el array A representa A_i y B representa A_{i-1}, y para i impar al revés, me ahorraría de copiar el arreglo en c/ iteración.

6. Tenemos un multiconjunto B de valores de billetes y queremos comprar un producto de costo c de una máquina que no da vuelto. Para poder adquirir el producto debemos cubrir su costo usando un subconjunto de nuestros billetes. El objetivo es pagar con el mínimo exceso posible a fin de minimizar nuestra pérdida. Más aún, queremos gastar el menor tiempo posible poniendo billetes en la máquina. Por lo tanto, entre las opciones de mínimo exceso posible, queremos una con la menor cantidad de billetes. Por ejemplo, si $c = 14$ y $B = \{2, 3, 5, 10, 20, 20\}$, la solución es pagar 15, con exceso 1, insertando sólo dos billetes: uno de 10 y otro de 5.

a) Considerar la siguiente estrategia por *backtracking* para el problema, donde $B = \{b_1, \dots, b_n\}$. Tenemos dos posibilidades: o agregamos el billete b_n, gastando un billete y quedando por pagar $c - b_n$, o no agregamos el billete b_n, gastando 0 billetes y quedando por pagar c. Escribir una función recursiva $\text{cc}(B, c)$ para resolver el problema, donde $\text{cc}(B, c) = (c', q)$ cuando el mínimo costo mayor o igual a c que es posible pagar con los billetes de B es c' y la cantidad de billetes mínima es q.

javi dame fuerza pls

$$\text{cc}(B, c, q, n) = \begin{cases} (c, q) & \text{cuando } n = 0 \wedge c \geq 0 \\ (+\infty, +\infty) & \text{cuando } n = 0 \wedge c < 0 \\ \text{mejor\_solución}(\text{cc}(B, c, q, n-1),\ \text{cc}(B, c - b_{n-1}, q+1, n-1)) & \text{En caso contrario} \end{cases}$$

```
mejor_solución (tupla a , tupla b){
    if ¿mismo_exceso?(a,b)
        return la_de_menos_billetes(a,b)
    else
        return la_de_menos_exceso(a,b)
    fi
}
```

• Le quito q

$$\text{cc}(B, c, n) = \begin{cases} (c, 0) & \text{cuando } n = 0 \wedge c \geq 0 \\ (+\infty, +\infty) & \text{cuando } n = 0 \wedge c < 0 \\ \text{mejor\_solución}(\text{cc}(B, c, n-1),\ \text{cc}(B, c - b_{n-1}, n-1) + (0, 1)) & \text{En caso contrario} \end{cases}$$

c) Reescribir cc como una función recursiva $\text{cc'}_B(i, j) = \text{cc}(\{b_1, \dots, b_i\}, j)$ que implemente la idea anterior **dejando fijo el parámetro** B. A partir de esta función, determinar cuándo cc'_B tiene la propiedad de *superposición de subproblemas*.

$$\text{cc'}_B(\underset{n}{i}, \underset{c}{j}) = \begin{cases} (c, 0) & \text{si } i = 0 \wedge c \geq 0 \\ (+\infty, +\infty) & \text{si } i = 0 \wedge c < 0 \\ \text{mejor\_solución}(\text{cc'}_B(i-1, j),\ \text{cc'}_B(i-1, j - B[i-1]) + (0, 1)) & \text{en caso contrario} \end{cases}$$

Hay superposición de problemas cuando se llama múltiples veces a un mismo caso recursivo

$\text{cc'}_B(i, j)$

→ $0 \leq j < c$

→ $0 \leq i \leq n+1$

⇒ En total hay $(c+1).(n+2)$ subproblemas y 2^n llamados recursivos, dado que $2^n \gg c.n$, puedo afirmar que se cumple la propiedad de superposición de suproblemas.

d) Definir una estructura de memoización para cc'_B que permita acceder a $\text{cc'}_B(i, j)$ en $\mathcal{O}(1)$ tiempo para todo $0 \leq i \leq n$ y $0 \leq j \leq k$. κ

Voy a usar una matriz (un arreglo de arreglos) de tamaño n.c.

	0	1	2	3	...	c
0	0	1	2	3	...	c
1	0					
2	0					
3	0					
⋮	⋮					
n	0					

devuelvo el exceso

• Tengo que cambiar mi función:

Antes devolvía una tupla (exceso, #billetes), pero #billetes puede ser reconstruido después de encontrar el resultado

$$cc'B(\overset{n}{i}, \overset{c}{j}) = \begin{cases} j & \text{si } i=0 \wedge j \geq 0 \\ +\infty & \text{si } i=0 \wedge j<0 \\ \text{mínimo}\{(cc'B(i-1, j), \; cc'B(i-1, j-B[i-1])\} & \text{en caso contrario} \end{cases}$$

$$cc'B(i, j) = \begin{cases} j & \text{si } i=0 \\ (cc'B(i-1, j) & \text{si } j-B[i-1]<0 \\ \text{mínimo}\{(cc'B(i-1, j), \; cc'B(i-1, j-B[i-1])\} & \text{en caso contrario} \end{cases}$$

• ¿como reconstruyo mi q?

Yo se cuánto es $cc'B(n,c)$ y c, y también se que $cc'B(n,c) = cc'B(n-1, c) \vee cc'B(n,c) = cc'B(n-1, c-B[n-1]) + B[n-1]$

???

⇒ Después lo sigo.

7. Astro Void se dedica a la compra de asteroides. Sea $p \in \mathbb{N}^n$ tal que p_i es el precio de un asteroide el i-ésimo día en una secuencia de n días. Astro Void quiere comprar y vender asteroides durante esos n días de manera tal de obtener la mayor ganancia neta posible. Debido a las dificultades que existen en el transporte y almacenamiento de asteroides, Astro Void puede comprar a lo sumo un asteroide cada día, puede vender a lo sumo un asteroide cada día y comienza sin asteroides. Además, el Ente Regulador Asteroidal impide que Astro Void venda un asteroide que no haya comprado. Queremos encontrar la máxima ganancia neta que puede obtener Astro Void respetando las restricciones indicadas. Por ejemplo, si $p = (\underset{c}{3}, \underset{c}{2}, \underset{v}{5}, \underset{v}{6})$ el resultado es 6 y si $p = (\underset{c}{3}, 6, \underset{v}{10})$ el resultado es 7. Notar que en una solución óptima, Astro Void debe terminar sin asteroides.

a) Convencerse de que la máxima ganancia neta (m.g.n.), si Astro Void tiene c asteroides al fin del día j, es:

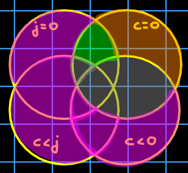
- indefinido (i.e., $-\infty$) si $\underbrace{c < 0}_{\text{vendí más de los que tengo}}$ o $\underbrace{c > j}_{\text{no llego a vender todos}}$, o
- el máximo entre:
  - la m.g.n. de finalizar el día $j-1$ con $c-1$ asteroides y comprar uno en el día j,
  - la m.g.n. de finalizar el día $j-1$ con $c+1$ asteroides y vender uno en el día j,
  - la m.g.n. de finalizar el día $j-1$ con c asteroides y no operar el día j.

Me convence

b) Escribir matemáticamente la formulación recursiva enunciada en *a*). Dar los valores de los casos base en función de la restricción de que comienza sin asteroides.

→ día 0 es antes de empezar, los días hábiles son $[1, n]$

$$mgn(\underbrace{j}_{0 \le j \le n}, \underbrace{c}_{0 \le c \le \frac{n}{2}}) = \begin{cases} 0 & j=0 \wedge c=0 \\ \text{máx}\{mgn(j-1,c-1) - p_j,\ mgn(j-1,c+1) + p_j,\ mgn(j-1,c)\} & \neg(j=0 \wedge c=0) \wedge j \ge c \ge 0 \\ -\infty & \neg(j=0 \wedge c=0) \wedge \underbrace{\neg(j \ge c \ge 0)}_{(j<c \ \vee \ c<0)} \end{cases}$$



casos malos

casos buenos

casos recursivos

casos inexistentes

c) Indicar qué dato es la respuesta al problema con esa formulación recursiva.

La respuesta es la ganancia máxima obtenida

d) Diseñar un algoritmo de PD *top-down* que resuelva el problema y explicar su complejidad temporal y espacial auxiliar.

variables globales: array $P \in \mathbb{N}^n$

matriz $M \in \mathbb{N}^{n \times \frac{n}{2}}$ → día x cdad de asteroides

.sin PD:

```
mgn(int j, int c){
    if (j==0 ∧ c==0){
        return 0;
    } else if (j<c ∨ c<0){
        return -∞;
    } else {
        return máx{mgn(j-1, c-1) - Pj, mgn(j-1, c+1) + Pj, mgn(j-1, c)}
    }
}
```

• Con PD

```
mgn ( int j , int c){
    if (j==0 ∧ c==0){
        if(M[0][0] == -1) M[0][0] = 0
        return M[0][0];
    } else if ( j<c  v  c<0){
        return -∞;
    } else {
        if (M[j-1][c-1] == -1) M[j-1][c-1] = mgn(j-1, c-1);
        if (M[j-1][c+1] == -1) M[j-1][c+1] = mgn (j-1, c+1);
        if (M[j-1][c] == -1)  M[j-1][c] = mgn (j-1, c);
        return máx{ M[j-1][c-1] - Pj,  M[j-1][c+1] + Pj,  M[j-1][c]}
    }
}
```

▷ Complejidad espacial : $O(n^2)$ (por usar la matriz)

▷ Complejidad temporal: #Llamadas recursivas: $\Omega(3^n)$? | #Subproblemas: $O(n^2)$

$n^2 << 3^n$, por lo que se cumple que haya superposición de subproblemas.

⇒ La complejidad del algoritmo es $O(n^2)$

9. Hay un terreno, que podemos pensarlo como una grilla de m filas y n columnas, con trampas y pociones. Queremos llegar de la esquina superior izquierda hasta la inferior derecha, y desde cada casilla sólo podemos movernos a la casilla de la derecha o a la de abajo. Cada casilla i,j tiene un número entero $A_{i,j}$ que nos modificará el nivel de vida sumándonos el número $A_{i,j}$ (si es negativo, nos va a restar $|A_{i,j}|$ de vida). Queremos saber el mínimo nivel de vida con el que debemos comenzar tal que haya un camino posible de modo que en todo momento nuestro nivel de vida sea al menos 1. Por ejemplo, si tenemos la grilla

$$A = \begin{array}{c} \begin{array}{ccc} \xrightarrow{-2} & \xrightarrow{-5} & \xrightarrow{-2} \end{array} \\ \begin{bmatrix} -2 & -3 & 3 \\ -5 & -10 & 1 \\ 10 & 30 & -5 \end{bmatrix} \end{array} \begin{array}{l} \downarrow \\ \downarrow -1 \\ \downarrow -6 \end{array}$$

↳ min_HP = 7

el mínimo nivel de vida con el que podemos comenzar es 7 porque podemos realizar el camino que va todo a la derecha y todo abajo.

a) Pensar la idea de un algoritmo de *backtracking* (no hace falta escribirlo).

Quiero que mi recorrido me provoque el menor daño posible.

Puedo probar todos los recorridos, anotar la vida final y :

- Si existe un camino con vida final positiva, la respuesta es 1.
- Sino, elijo mi camino con menos daño y la respuesta es 1+(daño recibido)

b) Convencerse de que, excepto que estemos en los límites del terreno, la mínima vida necesaria al llegar a la posición i,j es el resultado de restar al mínimo entre la mínima vida necesaria en $i+1,j$ y aquella en $i,j+1$ el valor $A_{i,j}$, salvo que eso fuera menor o igual que 0, en cuyo caso sería 1.

HP_min(i,j) = min(HP_min(i+1, j), HP_min(i, j+1)) - Aij ?

$$\text{HP\_min}(i,j) = \begin{cases} 1 & j=n \wedge i=m \wedge A_{ij} \geq 0 \ ✓ \\ -A_{ij}+1 & j=n \wedge i=m \wedge A_{ij} < 0 \ ✓ \\ +\infty & j>n \vee i>n \\ \min(\text{HP\_min}(i+1,j),\ \text{HP\_min}(i,j+1) - A_{ij} & \text{en caso contrario} \ ✓ \end{cases}$$

$$A = \begin{pmatrix} -2 & -3 & 3 \\ -5 & -10 & 1 \\ 10 & 30 & -5 \end{pmatrix}$$

$\vec{-2}$ $\vec{-5}$ $\vec{-2}$

$-1 \downarrow$

$-6 \downarrow$

quiero encontrar éste número

$$A' = \begin{pmatrix} -2 & -3 & 3 \\ -5 & -10 & -5 \\ 10 & 30 & 1 \end{pmatrix}$$

$\vec{-2}$ $\vec{-5}$ $\vec{-2}$

$-7 \downarrow$ quiero encontrar éste número

$-6 \downarrow$

$\downarrow -2$

$\downarrow -7$ → quiero encontrar éste número

$\downarrow 3$ $\vec{33}$ $\vec{34}$

• mini debuging:

$$\begin{array}{cccc} & 0 & 1 & \\ & & 5 & \\ & -2 & -3 & 0 \\ & & 2 & \\ B = & 4 & -1 & 1 \end{array}$$

$$HP\_min(0,0) = \min(HP\_min(1,0),\ HP\_min(0,1)) - A_{00}$$

$$\begin{aligned} HP\_min(1,0) &= \min(HP\_min(1,1),\ HP\_min(2,0)) - A_{10} \\ &= \min(1-(A_{11}),\ +\infty) - 1 \\ &= \min(2, +\infty) - 1 \\ &= 2-1 \\ &= 1 \end{aligned}$$

$$\begin{aligned} HP\_min(0,1) &= \min\{HP\_min(1,1),\ HP\_min(0,2)\} - A_{01} \\ &= \min\{2,\ +\infty\} - (-3) \\ &= 2+3 \\ &= 5 \end{aligned}$$