Práctica 2: Técnicas Algorítmicas

Compilado: 2 de septiembre de 2025

Backtracking

SumaSubconjuntosBT

1. En este ejercicio vamos a resolver el problema de suma de subconjuntos con la técnica de *backtracking*. Dado un multiconjunto $C = \{c_1, \ldots, c_n\}$ de números naturales y un natural k, queremos determinar si existe un subconjunto de C cuya sumatoria sea k. Vamos a suponer fuertemente que C está ordenado de alguna forma arbitraria pero conocida (i.e., C está implementado como la secuencia $c_1, \ldots, c_n$ o, análogamente, tenemos un iterador de C). Las *soluciones (candidatas)* son los vectores $a = (a_1, \ldots, a_n)$ de valores binarios; el subconjunto de C representado por a contiene a c_i si y sólo si $a_i = 1$. Luego, a es una solución *válida* cuando $\sum_{i=1}^{n} a_i c_i = k$. Asimismo, una *solución parcial* es un vector $p = (a_1, \ldots, a_i)$ de números binarios con $0 \leq i \leq n$. Si $i < n$, las soluciones *sucesoras* de p son $p \oplus 0$ y $p \oplus 1$, donde $\oplus$ indica la concatenación.

   a) Escribir el conjunto de soluciones candidatas para $C = \{6, 12, 6\}$ y $k = 12$.

   b) Escribir el conjunto de soluciones válidas para $C = \{6, 12, 6\}$ y $k = 12$.

   c) Escribir el conjunto de soluciones parciales para $C = \{6, 12, 6\}$ y $k = 12$.

   d) Dibujar el árbol de *backtracking* correspondiente al algoritmo descrito arriba para $C = \{6, 12, 6\}$ y $k = 12$, indicando claramente la relación entre las distintas componentes del árbol y los conjuntos de los incisos anteriores.

   e) Sea $\mathcal{C}$ la familia de todos los multiconjuntos de números naturales. Considerar la siguiente función recursiva $\text{ss}\colon \mathcal{C} \times \mathbb{N} \to \{V, F\}$ (donde $\mathbb{N} = \{0, 1, 2, \ldots\}$, V indica verdadero y F falso):

$$\text{ss}(\{c_1, \ldots, c_n\}, k) = \begin{cases} k = 0 & \text{si } n = 0 \\ \text{ss}(\{c_1, \ldots, c_{n-1}\}, k) \vee \text{ss}(\{c_1, \ldots, c_{n-1}\}, k - c_n) & \text{si } n > 0 \end{cases}$$

   Convencerse de que $\text{ss}(C, k) = V$ si y sólo si el problema de subconjuntos tiene una solución válida para la entrada C, k. Para ello, observar que hay dos posibilidades para una solución válida $a = (a_1, \ldots, a_n)$ para el caso $n > 0$: o bien $a_n = 0$ o bien $a_n = 1$. En el primer caso, existe un subconjunto de $\{c_1, \ldots, c_{n-1}\}$ que suma k; en el segundo, existe un subconjunto de $\{c_1, \ldots, c_{n-1}\}$ que suma $k - c_n$.

   f) Convencerse de que la siguiente es una implementación recursiva de ss en un lenguaje imperativo y de que retorna la solución para C, k cuando se llama con $C, |C|, k$. ¿Cuál es su complejidad?

   1) subset_sum(C, i, j): // implementa $\text{ss}(\{c_1, \ldots, c_i\}, j)$
   2) Si $i = 0$, retornar $(j = 0)$
   3) Si no, retornar subset_sum$(C, i - 1, j) \vee$ subset_sum$(C, i - 1, j - C[i])$

   g) Dibujar el árbol de llamadas recursivas para la entrada $C = \{6, 12, 6\}$ y $k = 12$, y compararlo con el árbol de *backtracking*.

h) Considerar la siguiente *regla de factibilidad*: $p = (a_1, \ldots, a_i)$ se puede extender a una solución válida sólo si $\sum_{q=1}^{i} a_q c_q \leq k$. Convencerse de que la siguiente implementación incluye la regla de factibilidad.

1) subset_sum(C, i, j): // implementa ss$(\{c_1, \ldots, c_i\}, j)$
2) Si $j < 0$, retornar **falso** // regla de factibilidad
3) Si $i = 0$, retornar $(j = 0)$
4) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

i) Definir otra regla de factibilidad, mostrando que la misma es correcta; no es necesario implementarla.

j) Modificar la implementación para imprimir el subconjunto de C que suma k, si existe. **Ayuda:** mantenga un vector con la solución parcial p al que se le agregan y sacan los elementos en cada llamada recursiva; tenga en cuenta de no suponer que este vector se copia en cada llamada recursiva, porque cambia la complejidad.

MagiCuadrados

2. Un *cuadrado mágico de orden n*, es un cuadrado con los números $\{1, \ldots, n^2\}$, tal que todas sus filas, columnas y las dos diagonales suman lo mismo (ver figura). El número que suma cada fila es llamado *número mágico*.

2	7	6
9	5	1
4	3	8

Existen muchos métodos para generar cuadrados mágicos. El objetivo de este ejercicio es contar cuántos cuadrados mágicos de orden n existen.

a) ¿Cuántos cuadrados habría que generar para encontrar todos los cuadrados mágicos si se utiliza una solución de fuerza bruta?

b) Enunciar un algoritmo que use *backtracking* para resolver este problema que se base en la siguientes ideas:

- La solución parcial tiene los valores de las primeras $i-1$ filas establecidos, al igual que los valores de las primeras j columnas de la fila i.
- Para establecer el valor de la posición $(i, j+1)$ (o $(i+1, 1)$ si $j = n$ e $i \neq n$) se consideran todos los valores que aún no se encuentran en el cuadrado. Para cada valor posible, se establece dicho valor en la posición y se cuentan todos los cuadrados mágicos con esta nueva solución parcial.

Mostrar los primeros dos niveles del árbol de *backtracking* para $n = 3$.

c) Demostrar que el árbol de *backtracking* tiene $\mathcal{O}((n^2)!)$ nodos en peor caso.

d) Considere la siguiente poda al árbol de *backtracking*: al momento de elegir el valor de una nueva posición, verificar que la suma parcial de la fila no supere el número mágico. Verificar también que la suma parcial de los valores de las columnas no supere el número mágico. Introducir estas podas al algoritmo e implementarlo en la computadora. ¿Puede mejorar estas podas?

e) Demostrar que el número mágico de un cuadrado mágico de orden n es siempre $(n^3 + n)/2$. Adaptar la poda del algoritmo del ítem anterior para que tenga en cuenta esta nueva información. Modificar la implementación y comparar los tiempos obtenidos para calcular la cantidad de cuadrados mágicos.

MaxiSubconjunto

3. Dada una matriz simétrica M de $n \times n$ números naturales y un número k, queremos encontrar un subconjunto I de $\{1, \ldots, n\}$ con $|I| = k$ que maximice $\sum_{i,j \in I} M_{ij}$. Por ejemplo, si $k = 3$ y:

$$M = \begin{pmatrix} 0 & 10 & 10 & 1 \\ - & 0 & 5 & 2 \\ - & - & 0 & 1 \\ - & - & - & 0 \end{pmatrix},$$

entonces $I = \{1, 2, 3\}$ es una solución óptima.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

RutaMinima

4. Dada una matriz D de $n \times n$ números naturales, queremos encontrar una permutación π[1] de $\{1, \ldots, n\}$ que minimice $D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)}$. Por ejemplo, si

$$D = \begin{pmatrix} 0 & 1 & 10 & 10 \\ 10 & 0 & 3 & 15 \\ 21 & 17 & 0 & 2 \\ 3 & 22 & 30 & 0 \end{pmatrix},$$

entonces $\pi(i) = i$ es una solución optima.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

[1] Una permutacion de un conjunto finito X es simplemente una función biyectiva de X en X.

Palabras en cadena

5. Dada una cadena de letras sin espacios o puntos queremos analizar si se puede subdividir de forma de obtener palabras. Suponiendo que se tiene una función $palabra\colon [a,z] \to bool$ que verifica si una cadena de letras es una palabra en $O(n)$.

   a) Dar una función recursiva que resuelva el problema.

   b) Calcular una cota superior para la complejidad.

   c) Demostrar que el algoritmo es correcto.

Árboles binarios de búsqueda óptimos

6. Dado un conjunto de elementos de $[n]$, y una función $f\colon [n] \to \mathbb{N}$ que nos da la frecuencia de acceso a dichos elementos, decimos que A es un árbol binario de busqueda óptimo si este minimiza el costo de todos los accesos dados por f.

   a) Escribir una función recursiva que devuelva el costo de acceder a todos los elementos usando f.

   b) Dar una cota superior para la complejidad (Ayuda: pasar de la función recursiva a una recurrencia que solo dependa del tamaño de la entrada).

   c) Probar que el algoritmo es correcto

Dobra

7. Dobra se encuentra con muchas palabras en su vida, como es una persona particular la mayoría de estas no le gustan. Para compensar empezó a inventar palabras más agradables. Dobra crea palabras nuevas escribiendo una cadena de caracteres que considera buena, luego borra los caracteres que peor le caen y los reemplaza con _. Luego para mejorar su vida intenta reemplazar estos guiones bajos con letras más aceptables intentando crear palabras más lindas. Dobra considera una palabra como buena si no contiene 3 vocales consecutivas, 3 consonantes consecutivas y al menos contiene una E.

   a) Mostrar alguna solución candidata posible y alguna solución parcial.

   b) Proponer una función recursiva y estimar su complejidad. Asumir que se tiene una función $verificar$ que toma una cadena y devuelve $True$ si es como Dobra quiere o $False$ en caso contrario. [3]

   c) Probar que la función o programa es correcto.

   d) Proponer al menos una poda por factibilidad.

   e) Si b) no tiene una cota superior $O(3^n)$ para la complejidad, analizar el caso donde se separa la recursión en tener o no una letra E y ver si mejora la misma. [5]

[3] Se puede asumir que $verificar$ funciona correctamente.
[5] La mejor complejidad que conocemos es $O(n2^n)$

Cadenas de Adición

8. Dado un entero n decimos que $C = \{x_1, \ldots, x_k\}$ es una cadena de adición si cumple lo siguiente

   $\sim$ $1 = x_1 < x_2 < \ldots < x_k = n$

   $\sim$ Para cada $2 \leq j \leq n$ existe $k_1, k_2 < j$ tal que $x_{k_1} + x_{k_2} = x_j$

   a) Encontrar un algoritmo de backtracking que encuentre, si existe, la cadena de adición de longitud mínima.

Programación dinámica (y su relación con *backtracking*)

KingArmy

9. El rey Cambyses está interesado en armar ejércitos en una serie de días consecutivos. Mas aun, le interesa que el número de personas de su ejército en el dia d_i sea equivalente a la suma del número de personas del ejército que formó el dia $i-1$ e $i-2$. La excepción para esto es en el día 0 y 1, en cuyo caso la cantidad de personas en esos días va a ser siempre 1. Para el es muy complicado determinar este número, entonces nos pidió que lo ayudemos. Dado un dia N, tenemos que devolver el número de personas de su ejército.

   Pensar un algoritmo $O(N)$ para resolver este problema y demostrar su correctitud y complejidad.

Vacations

10. Tomas tiene N días de vacaciones, donde puede hacer actividades:

    - Se pueden hacer 2 actividades: gimnasio y competencias.
    - Cada día puede tener disponible ninguna, alguna o ambas.

    Tomas cada dia puede

    - Hacer una actividad que este disponible, siempre que no la haya hecho el día anterior.
    - Descansar.

    Tomas quiere minimizar los días de descanso.

    a) Diseñar un algoritmo $O(N)$ que calcule la mínima cantidad de días de descanso.
    b) Probar la correctitud y complejidad del algoritmo.
    c) Indicar cómo se puede reconstruir la solución. Es decir, indicarle a Tomas qué hara cada día.

    Ayuda: Consideren el siguiente ejemplo:

    - $N = 4$
    - Días con gimnasio disponible: $2, 3$
    - Días con competencias disponibles: $1, 2$

    Puede lograr tener solo 2 días de descanso.

SumaDinámica

11. 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.

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) 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 $0 \le i \le n$ y $0 \le j \le k$. 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 $0 \le i \le n$ y $0 \le j \le k$.
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 $M[i,j] =$ subset_sum$(C,i-1,j) \vee$ subset_sum$(C,i-1,j-C[i])$
7) Retornar $M[i,j]$

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 $k \gg 2^n$?

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 $\{c_1,\dots,c_i\}$ 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 $M[i, j] := M[i-1, j] \vee (j - C[i] \geq 0 \wedge M[i-1, j - C[i]])$

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

g) (Opcional) Demostrar que la función recursiva del inciso a) es correcta. **Ayuda:** demostrar por inducción en i que existe algún subconjunto de $\{c_1, \ldots, c_i\}$ que suma j si y solo si $\text{ss'}_C(i, j) = V$.

OptiPago

12. 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 $B = \{b_1, \ldots, b_n\}$. 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.

b) Implementar la función de a) en un lenguaje de programación imperativo utilizando una función recursiva con parámetros B, i, j que compute $\text{cc}(\{b_1, \ldots, b_i\}, j)$. ¿Cuál es la complejidad del algoritmo?

c) Reescribir cc como una función recursiva $\text{cc'}_B(i, j) = \text{cc}(\{b_1, \ldots, b_i\}, j)$ 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*.

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 $0 \leq j \leq k$.

e) Adaptar el algoritmo de b) para incluir la estructura de memoización.

f) Indicar cuál es la llamada recursiva que resuelve nuestro problema y cuál es la complejidad del nuevo algoritmo.

g) (Opcional) Escribir un algoritmo *bottom-up* para calcular todos los valores de la estructura de memoización y discutir cómo se puede reducir la memoria extra consumida por el algoritmo.

h) (Opcional) Formalmente, en este problema de vuelto hay que computar el mínimo $(\sum V, |V|)$, en orden lexicográfico, de entre los conjuntos $V \subseteq B$ tales que $\sum V \geq c$. Demostrar que la función cc' es correcta. **Ayuda:** demostrar por inducción que $\text{cc'}(i, j) = (v, k)$ para el mínimo (v, k) tal que existe un subconjunto V de $\{b_1, \ldots, b_i\}$ con $\sum_V \geq j$.

AstroTrade

13. 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 $p = (3, 2, 5, 6)$ el resultado es 6 y si $p = (3, 6, 10)$ 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 $c < 0$ o $c > j$, 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.

    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.

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

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

    e) (Opcional) Diseñar un algoritmo de PD *bottom-up*, reduciendo la complejidad espacial.

    f) (Opcional) Formalmente, el problema consiste en determinar el máximo $g = \sum_{i=1}^{n} x_i p_i$ para un vector $x = (x_1, \ldots, x_n)$ tal que: $x_i \in \{-1, 0, 1\}$ para todo $1 \leq i \leq n$ y $\sum_{i=1}^{j} x_i \leq 0$ para todo $1 \leq j \leq n$. Demostrar que la formulacion recursiva es correcta. **Ayuda**: primero demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides en el día n. Luego, demostrar por inducción que la función recursiva respeta la semántica, i.e., que computa la m.g.n. al final del día j cuando Astro Void posee c asteroides.

Fire

14. Este problema fue extraido de Codeforces (864E)

    El sabueso de preguntas está en problemas. ¡Su casa se incendia!. Es tiempo de salvar los artículos más valiosos. Cada artículo tiene los parámetros:

    - El tiempo t entero que estima el sabueso que le tomara salvarlo.
    - El tiempo d entero que estima el sabueso a partir del cuál se quema el articulo y no sirve más (inclusive si lo está rescatando en ese momento)
    - El valor p del artículo para el sabueso.

El sabueso queire maximizar la suma de los valores de los artículos que puede salvar, considerando que:

- Salva un árticulo luego de otro, uno a la vez.
- Si un artículo A le toma ta segundos y luego salva el B, entonces le habrá tomado $t_A + t_B$ segundos en total.

Sea D el máximo de los tiempos d de todos los N artículos, diseñar un algoritmo que en $O(D * N + NlogN)$ le índique al sabueso cuanto el máximo valor total qué puede salvar, qué artículos debe salvar para conseguirlo y en qué orden salvarlos.

CortesEconómicos

15. Debemos cortar una vara de madera en varios lugares predeterminados. Sabemos que el costo de realizar un corte en una madera de longitud ℓ es ℓ (y luego de realizar ese corte quedarán 2 varas de longitudes que sumarán ℓ). Por ejemplo, si tenemos una vara de longitud 10 metros que debe ser cortada a los 2, 4 y 7 metros desde un extremo, entonces los cortes se pueden realizar, entre otras maneras, de las siguientes formas:

    - Primero cortar en la posición 2, después en la 4 y después en la 7. Esta resulta en un costo de $10 + 8 + 6 = 24$ porque el primer corte se hizo en una vara de longitud 10 metros, el segundo en una de 8 metros y el último en una de 6 metros.
    - Cortar primero donde dice 4, después donde dice 2, y finalmente donde dice 7, con un costo de $10 + 4 + 6 = 20$, que es menor.

    Queremos encontrar el mínimo costo posible de cortar una vara de longitud ℓ.

    a) Convencerse de que el mínimo costo de cortar una vara que abarca desde i hasta j con el conjunto C de lugares de corte es $j - i$ mas el mínimo, para todo lugar de corte c entre i y j, de la suma entre el mínimo costo desde i hasta c y el mínimo costo desde c hasta j.

    b) Escribir matemáticamente una formulación recursiva basada en *a*). Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

    c) Diseñar un algoritmo de PD y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

    d) Supongamos que se ordenan los elementos de C en un vector *cortes* y se agrega un 0 al principio y un ℓ al final. Luego, se considera que el mínimo costo para cortar desde el i-ésimo punto de corte en *cortes* hasta el j-ésimo punto de corte será el resultado buscado si $i = 1$ y $j = |C| + 2$.

        I) Escribir una formulación recursiva con dos parámetros que esté basada en *d*) y explicar su semántica.

        II) Diseñar un algoritmo de PD, dar su complejidad temporal y espacial auxiliar y compararlas con aquellas de *c*). Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

Travesía Vital

16. 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

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

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).

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.

c) Escribir una formulación recursiva basada en *b*). Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

d) Diseñar un algoritmo de PD y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

e) Dar un algoritmo *bottom-up* cuya complejidad temporal sea $\mathcal{O}(m \cdot n)$ y la espacial auxiliar sea $\mathcal{O}(\text{mín}(m, n))$.

PilaCauta

17. Tenemos cajas numeradas de 1 a N, todas de iguales dimensiones. Queremos encontrar la máxima cantidad de cajas que pueden apilarse en una única pila cumpliendo que:

- sólo puede haber una caja apoyada directamente sobre otra;
- las cajas de la pila deben estar ordenadas crecientemente por número, de abajo para arriba;
- cada caja i tiene un peso w_i y un soporte s_i, y el peso total de las cajas que están arriba de otra no debe exceder el soporte de esa otra.

Si tenemos los pesos $w = [19, 7, 5, 6, 1]$ y los soportes $s = [15, 13, 7, 8, 2]$ (la caja 1 tiene peso 19 y soporte 15, la caja 2 tiene peso 7 y soporte 13, etc.), entonces la respuesta es 4. Por ejemplo, pueden apilarse de la forma 1-2-3-5 o 1-2-4-5 (donde la izquierda es más abajo), entre otras opciones.

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

b) Escribir una formulación recursiva que sea la base de un algoritmo de PD. Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

c) Diseñar un algoritmo de PD y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

d) (Opcional) Formalizar el problema y demostrar que la función recursiva es correcta.

OperacionesSeq

18. Sea $v = (v_1, v_2, \dots v_n)$ un vector de números naturales, y sea $w \in \mathbb{N}$. Se desea intercalar entre los elementos de v las operaciones $+$ (suma), $\times$ (multiplicación) y $\uparrow$ (potenciación) de tal manera que al evaluar la expresión obtenida el resultado sea w. Para evaluar la expresión se opera de izquierda a derecha ignorando la precedencia de los operadores. Por ejemplo, si $v = (3, 1, 5, 2, 1)$, y las operaciones elegidas son $+$, $\times$, $\uparrow$ y $\times$ (en ese orden), la expresión obtenida es $3+1\times5\uparrow2\times1$, que se evalúa como $(((3 + 1) \times 5) \uparrow 2) \times 1 = 400$.

a) Escribir una formulación recursiva que sea la base de un algoritmo de PD que, dados v y w, encuentre una secuencia de operaciones como la deseada, en caso de que tal secuencia exista. Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

b) Diseñar un algoritmo basado en PD con la formulación de *a*) y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

c) (Opcional) Formalizar el problema y demostrar que la función recursiva es correcta.

DadosSuma

19. Se arrojan simultáneamente n dados, cada uno con k caras numeradas de 1 a k. Queremos calcular todas las maneras posibles de conseguir la suma total $s \in \mathbb{N}$ con una sola tirada. Tomamos dos variantes de este problema.

(A) Consideramos que los dados son **distinguibles**, es decir que si $n = 3$ y $k = 4$, entonces existen 10 posibilidades que suman $s = 6$:

1) 4 posibilidades en las que el primer dado vale 1
2) 3 posibilidades en las que el primer dado vale 2
3) 2 posibilidades en las que el primer dado vale 3
4) Una posibilidad en la que el primer dado vale 4

(B) Consideramos que los dados son **indistinguibles**, es decir que si $n = 3$ y $k = 4$, entonces existen 3 posibilidades que suman $s = 6$:

1) Un dado vale 4, los otros dos valen 1
2) Un dado vale 3, otro 2 y otro 1
3) Todos los dados valen 2

a) Definir en forma recursiva la función $f\colon \mathbb{N}^2 \to \mathbb{N}$ tal que $f(n, s)$ devuelve la respuesta para el escenario (A) (fijado k).

b) Definir en forma recursiva la función $g\colon \mathbb{N}^3 \to \mathbb{N}$ tal que $g(n, s, k)$ devuelve la respuesta para el escenario *(B)*.

c) Demostrar que f y g poseen la propiedad de superposición de subproblemas.

d) Definir algoritmos *top-down* para calcular $f(n, s)$ y $g(n, s, k)$ indicando claramente las estructuras de datos utilizadas y la complejidad resultante.

e) Escribir el (pseudo-)código de los algoritmos top-down resultantes.

Nota: Una solución correcta de este ejercicio debería indicar cómo se computa tanto $f(n, s)$ como $g(n, s, k)$ en tiempo $O(nk \min\{s, nk\})$.

CaesarsLegions

20. Este problema fue extraido de *Codeforces (118D)*

Al famoso general Caesar le gusta poner en línea sus tropas, que son patos y dodos.

- Tiene un ejército de P patos y D dodos.
- No le gusta que la forma de poner en línea sus tropas incumpla que:
  - No puede haber más de MP patos consecutivos.
  - No puede haber más de MD dodos consecutivos.
- Los patos son indistinguibles entre ellos, al igual que los dodos.

Le interesa contar la cantidad de formas posibles de formar esta línea.

a) Definir una función recursiva $f(t, n_P, n_D, k_P, k_D)$ (asumiendo MP y MD fijos, dados en el entorno) que calcule la cantidad de formas de poner los patos y dodos restantes, dado que:
- Quedan t lugares disponibles en la fila.
- Quedan n_P patos y n_D dodos disponibles.
- Al final de la fila hay, actualmente, k_P patos y k_D dodos formados de forma consecutiva.

Pensar qué elección tomamos en cada paso.

b) Pensar cómo modificar la función para que su firma sea $f(n_P, n_D, k, ultTropa)$, donde $ultTropa$ nos indica cuál es la última tropa puesta y k la cantidad consecutiva de la misma.

c) Pensar cómo modificar la funcion para que su firma sea $f(n_P, n_D, ultTropa)$.

d) Calcular la complejidad computacional de cada una de las versiones de la función.

e) Demostrar la correctitud de cada una de las versiones de la función.

Farmer

21. Este problema fue extraido de Codeforces (41D)

Un granjero tiene un terreno de N metros de largo y M de ancho, dividido en celdas de $1x1m$. en algunas celdas hay una cantidad arbitraría de arvejas. El granjero tiene como objetivo recolectar la mayor cantidad posible de arvejas, respetando que:

- Empieza desde alguna celda en el comienzo del terreno ($y = 0$).
- Se puede mover en 2 sentidos: **Adelante e Izquierda** y **Adelante y Derecha**.
- Tiene que llegar al final del terreno ($y = N$) con una cantidad de arvejas recolectadas divisible por $K + 1$, con K un número fijo dado.

Diseñar un algoritmo que en $O(N * M * K)$ calcule cuanta es la máxima cantidad de arvejas que puede recolectar el granjero. Demostrar su correctitud y complejidad.

Indicar también como es posible reconstruir un posible cámino para recolectar esa cantidad de arvejas.

Problemas Anteriores

22. Utilizando Programación Dinámica iterativa, repensar los siguientes ejercicios de las clases teóricas y prácticas anteriores para alcanzar las complejidades *espaciales* propuestas:

    - 9: $O(1)$
    - 10:$O(1)$
    - 13: $O(|p|)$
    - Cambio de Monedas (devolver un vuelto K): $O(K)$
    - Número combinatorio $C(N, K)$: $O(K)$
    - Longest Common Subsequence entre S y T: $O(min(|S|, |T|))$
    - 19: $O(1)$ (ambas versiones)
    - 21: $O(M * K)$

ABB óptimo

23. Revisitar el ejercicio 6 de la práctica 1 y diseñar un algoritmo de PD *bottom-up* que resuelva el problema en $\mathcal{O}(n^3)$ tiempo y $\mathcal{O}(n^2)$ espacio auxiliar. Demostrar su correctitud y complejidad.

    Opcional: La *Optimización de Knuth* nos permite mejorar la complejidad temporal a $O(N^2)$. La observación clave que realiza es que la raíz optima para el rango $[i, j)$ es $opt(i, j)$, se cumple que $opt(i, j-1) \leq opt(i, j) \leq opt(i+1, j)$. Pensar como pueden utilizar Programación Dinámica Iterativa para computar los estados en un orden que permita aprovechar esta optimización.