Práctica 1: Técnicas Algorítmicas

Compilado: 15 de septiembre de 2024

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.

Programación dinámica (y su relación con *backtracking*)

SumaDinámica

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 $\text{ss'}_C \colon \{0, \ldots, n\} \times \{0, \ldots, k\} \to \{V, F\}$ (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 $0 \leq i \leq n$ 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 $\text{ss}(\{c_1, \ldots, c_i\}, j) = \text{ss'}_C(i, j)$ 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 $\{c_1, \ldots, c_i\}$ 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 $M[i, j] := M[i-1, j] \vee (j - C[i] \geq 0 \wedge M[i-1, j - C[i]])$

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

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

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 $p = (3, 2, 5, 6)$ 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.

CortesEconómicos

8. 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

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

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

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

10. 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

11. Sea $v = (v_1, v_2, \ldots v_n)$ 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

12. 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 $f(n, s, k)$ 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 $O(nk \min\{s, nk\})$.

Golosos (≡ avariciosos ≡ *greedy*)

ParejasdeBaile

13. Tenemos dos conjuntos de personas y para cada persona sabemos su habilidad de baile. Queremos armar la máxima cantidad de parejas de baile, sabiendo que para cada pareja debemos elegir exactamente una persona de cada conjunto de modo que la diferencia de habilidad sea menor o igual a 1 (en módulo). Además, cada persona puede pertenecer a lo sumo a una pareja de baile. Por ejemplo, si tenemos un multiconjunto con habilidades $\{1, 2, 4, 6\}$ y otro con $\{1, 5, 5, 7, 9\}$, la máxima cantidad de parejas es 3. Si los multiconjuntos de habilidades son $\{1, 1, 1, 1, 1\}$ y $\{1, 2, 3\}$, la máxima cantidad es 2.

    a) Considerando que ambos multiconjuntos de habilidades estan ordenados en forma creciente, observar que la solución se puede obtener recorriendo los multiconjuntos en orden para realizar los emparejamientos.

b) Diseñar un algoritmo goloso basado en *a*) que recorra una única vez cada multiconjunto. Explicitar la complejidad temporal y espacial auxiliar.

c) Demostrar que el algoritmo dado en *b*) es correcto.

SumaSelectiva

14. Dado un conjunto X con $|X| = n$ y un entero $k \leq n$ queremos encontrar el máximo valor que pueden sumar los elementos de un subconjunto S de X de tamaño k. Más formalmente, queremos calcular $\max_{S \subseteq X, |S|=k} \sum_{s \in S} s$.

a) Proponer un algoritmo *greedy* que resuelva el problema, demostrando su correctitud. Extender el algoritmo para que también devuelva uno de los subconjuntos S que maximiza la suma.

b) Dar una implementación del algoritmo del inciso *a*) con complejidad temporal $O(n \log n)$.

c) Dar una implementación del algoritmo del inciso *a*) con complejidad temporal $O(n \log k)$.

SumaGolosa

15. Queremos encontrar la suma de los elementos de un multiconjunto de números naturales. Cada suma se realiza exactamente entre dos números x e y y tiene costo $x + y$.

Por ejemplo, si queremos encontrar la suma de $\{1, 2, 5\}$ tenemos 3 opciones:

- $1 + 2$ (con costo 3) y luego $3 + 5$ (con costo 8), resultando en un costo total de 11;
- $1 + 5$ (con costo 6) y luego $6 + 2$ (con costo 8), resultando en un costo total de 14;
- $2 + 5$ (con costo 7) y luego $7 + 1$ (con costo 8), resultando en un costo total de 15.

Queremos encontrar la forma de sumar que tenga costo mínimo, por lo que en nuestro ejemplo la mejor forma sería la primera.

a) Explicitar una estrategia golosa para resolver el problema.

b) Demostrar que la estrategia propuesta resuelve el problema.

c) Implementar esta estrategia en un algoritmo iterativo. **Nota:** el mejor algoritmo simple que conocemos tiene complejidad $\mathcal{O}(n \log n)$ y utiliza una estructura de datos que implementa una secuencia ordenada.

RutaEficiente

16. Tomás quiere viajar de Buenos Aires a Mar del Plata en su flamante Renault 12. Como está preocupado por la autonomía de su vehículo, se tomó el tiempo de anotar las distintas estaciones de servicio que se encuentran en el camino. Modeló el mismo como un segmento de 0 a M, donde Buenos aires está en el kilómetro 0, Mar del Plata en el M, y las distintas estaciones de servicio están ubicadas en los kilómetros $0 = x_1 \leq x_2 \leq \ldots x_n \leq M$.

Razonablemente, Tomás quiere minimizar la cantidad de paradas para cargar nafta. Él sabe que su auto es capaz de hacer hasta C kilómetros con el tanque lleno, y que al comenzar el viaje este está vacío.

a) Proponer un algoritmo *greedy* que indique cuál es la cantidad mínima de paradas para cargar nafta que debe hacer Tomás, y que aparte devuelva el conjunto de estaciones en las que hay que detenerse. Probar su correctitud.

b) Dar una implementación de complejidad temporal $O(n)$ del algoritmo del inciso a).

División Pandémica

17. En medio de una pandemia, la Escuela de Aulas Grandes y Ventiladas quiere implementar un protocolo especial de distanciamiento social que tenga en cuenta que la escuela no tienen restricciones de espacio. El objetivo es separar a cada curso en dos subcursos a fin de reducir la cantidad de pares de estudiantes que sean muy cercanos, dado que se estima que estos estudiantes tienen dificultades para respetar tan buscado distanciamiento. Para este fin, en el protocolo se estableció que cada curso que tenga c parejas de estudiantes cercanos tiene que dividirse en dos subcursos, cada uno de los cuales puede tener a lo sumo $c/2$ parejas de estudiantes cercanos. Notar que no importa si un subcurso queda con más estudiantes que otro.

    Formalmente, para cada curso contamos con un conjunto de estudiantes E y su conjunto C de pares de estudiantes cercanos. Luego, una partición (A, B) de E es una *solución factible para* (E, C) cuando $|(A \times A) \cap C| \leq |C|/2$ y $|(B \times B) \cap C| \leq |C/2|$. Por ejemplo, si $E = \{1, 2, 3, 4\}$ y $C = \{1\text{-}2, 2\text{-}3, 3\text{-}4\}$, entonces $(\{1, 3, 4\}, \{2\})$ y $(\{2, 4\}, \{1, 3\})$ son soluciones factibles.

    a) Especificar el problema descrito definiendo cuál es la instancia (i.e. cuáles son los datos de entrada y qué condiciones satisfacen) y cuál es el resultado esperado (i.e., cuáles son los datos de salida y qué condiciones satisfacen).

    b) Demostrar que para toda instancia existe un resultado esperado que satisface las condiciones definidas por el protocolo. **Ayuda:** hacer inducción en la cantidad de estudiantes. Para el paso inductivo, considerar que si les estudiantes se asignan iterativamente a los subcursos, entonces conviene enviar a cada estudiante al subcurso que tenga la menor cantidad de estudiantes cercanos a elle.

    c) A partir de la demostración del inciso anterior, diseñar un algoritmo que encuentre una solución factible en tiempo lineal en función del tamaño de la entrada definido en el inciso a).

MaxMex

18. Se define la función $mex : \mathcal{P}(\mathbb{N}) \to \mathbb{N}$ como

$$mex(X) = \text{mín}\{j : j \in \mathbb{N} \wedge j \notin X\}$$

    Intuitivamente, mex devuelve, dado un conjunto X, el menor número natural que no está en x. Por ejemplo, $mex(\{0, 1, 2\}) = 3$, $mex(\{0, 1, 3\}) = 2$ y $mex(\{1, 2, 3, \ldots\}) = 0$.

    Dado un vector de número $a_1 \ldots a_n$ queremos encontrar la permutación $b_1 \ldots b_n$ de los mismos que maximize

$$\sum_{i=1}^{n} mex(\{b_1 \ldots b_i\})$$

Por ejemplo, si el vector es $\{3, 0, 1\}$ podemos ver que la mejor permutación es $\{0, 1, 3\}$, que alcanza un valor de

$$mex(\{0\}) + mex(\{0, 1\}) + mex(\{0, 1, 3\}) = 1 + 2 + 2 = 5$$

a) Proponer un algoritmo *greedy* que resuelva el problema y demostrar su correctitud. **Ayuda**: ¿Cuál el máximo valor que puede tomar $mex(X)$ si X tiene n elementos? Si $X \subseteq Y$, ¿Qué pasa con los valores $mex(X)$ y $mex(Y)$?

b) Dar una implementación del algoritmo del inciso anterior con complejidad temporal $O(n)$.

CacheOpt

19. Como los accesos a memoria RAM son lentos en comparación al trabajo propio del CPU, es común que se coloquen memorias intermedias más diminutas y de alta velocidad entre ambas unidades, las cuales llamamos *cachés*. Cuando un programa se ejecuta y hace una consulta a la memoria por cierta posición r primero se verifica si la posición r está cargada en la caché, en cuyo caso el CPU la puede obtener sin tener que hacer el acceso a la RAM. Cuando esto ocurre, decimos que ocurre un *caché hit*. En cambio, si la posición r no está en la caché, esta se busca a memoria, se carga en la caché, y luego se la informa al CPU. A este evento se lo conoce como *caché miss*.

    Como la caché es más chica que la memoria RAM es inevitable que eventualmente ocurra un *caché miss* y que la caché este llena. En ese caso, la caché debe decidir qué información va a desechar para darle lugar a la nueva entrada. Naturalmente, se busca minimizar la cantidad de *misses* de los siguientes accesos.

    El problema de *Off-line* caching consiste en determinar, dada una caché C de tamaño k y una lista de n requests $R = \{r_1, r_2, \ldots, r_n\}$[2] a posiciones de memoria, qué decisión debe tomar en cada paso la caché para minimizar la cantidad de *misses*. Por ejemplo, si $k = 2$ y $R = \{1, 2, 3, 1\}$ entonces:

    - La primera consulta es un *miss*, pero como hay lugar en la caché (empieza vacía) se carga la posición 1 a C ($C = \{1\}$).
    - Con la segunda consulta pasa lo mismo, por lo que la caché queda en el estado $C = \{1, 2\}$.
    - En la tercera consulta la caché esta llena, por lo que se debe desechar alguna entrada. Notemos que si se desecha 1 entonces la cuarta consulta dará otro *miss*, mientras que si se desecha 2 entonces habrá un *hit*.

    Una política posible para decidir qué elemento desechar es la *furthest-in-future*: se desecha aquella posición r cuyo siguiente acceso es el más lejano (o bien, que no tiene un siguiente acceso).

    a) **Opcional**: Definir una función recursiva $f(i, mem)$ que tome un índice y un estado de la memoria y devuelva la mínima cantidad de *caché misses* que deben ocurrir para procesar todas las consultas $\{r_i, r_{i+1}, \ldots, r_n\}$ si el estado actual de la memoria es mem ¿Con qué llamado se resuelve el problema? Estudiar la superposición de subproblemas y explicar en qué casos vale la pena memorizar.

[2] Sin pérdida de generalidad respecto al problema, podemos asumir que $1 \leq r_i \leq n$ para todo i.

b) Probar que la política *furthest-in-future* es óptima (es decir, que minimiza la cantidad de *misses*). **Ayuda**: Dada una serie de decisiones, probar que si en un paso no se sigue la política *furthest* entonces podemos alterar ese paso para que sí la siga sin afectar la cantidad de *misses*.

c) Dar un algoritmo con complejidad temporal $O(n \log(k))$ que informe qué decisión debe tomar la caché en cada paso para minimizar la cantidad de *misses*.

Ejercicios integradores

Invitación Estratégica

20. El problema de la fiesta consiste en determinar un conjunto de invitados que no tengan conflictos entre sí y que sea de cardinalidad máxima. Formalmente, dado un conjunto V de posibles invitados y un conjunto E de conflictos, formados por pares no ordenados de V, queremos encontrar un subconjunto $S \subseteq V$ de cardinalidad máxima entre aquellos que cumplen que $\{v, w\} \notin E$ para todo par $v, w \in S$. Por ejemplo, si $S = \{1, 2, 3, 4, 5\}$ y $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}\}$, entonces una solución es $S = \{1, 3, 5\}$, ya que no se puede invitar a ningún conjunto de 4 personas. Vamos a suponer que los posibles invitados se representan con el conjunto $V = \{1, \ldots, n\}$ para algún $n \geq 0$ (el caso $n = 0$ es válido y representa el conjunto $V = \emptyset$).

a) Decimos que $S \subseteq \mathbb{N}$ y $W \subseteq \mathbb{N}$ son *compatibles* cuando $S \subseteq V$ es un conjunto posible de invitados y ningún elemento de $W \subseteq V \setminus S$ tiene un conflicto con algún elemento de S. En el ejemplo anterior, $S = \{1\}$ y $W = \{4, 5\}$ son compatibles pero $S = \{1, 4\}$ y $W = \{2\}$ no lo son. Sea $\mathcal{V}$ el conjunto de subconjuntos de V. Escribir una función recursiva $\text{fs}\colon \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ tal que, dados S y W compatibles, $\text{fs}(S, W)$ retorne un conjunto de invitados de máxima cardinalidad que contenga a S. (Notar que la llamada recursiva debe garantizar la compatibilidad). **Ayuda:** considerar dos posibilidades: no invitar a $w \in W$, o invitar a $w \in W$ y no invitar a nadie que tenga un conflicto con w.

b) En base a *a*), implementar un algoritmo recursivo de *backtracking* para resolver el problema de la fiesta basado en las siguientes ideas:

- cada solución parcial es un conjunto $S \subseteq V$ que no contiene invitados con conflictos.
- a cada nodo del árbol de *backtracking* se le asocia un conjunto $W \subseteq V$ compatible con S de posibles invitados.
- para la extensión, se consideran dos posibilidades: o bien no se invita a $w \in W$ o bien se invita a w y se eliminan de W todos los otros elementos que estén en conflicto con w.

c) Escribir los tres primeros niveles del árbol de *backtracking* resultante de la implementación anterior.

d) Describir una regla de optimalidad para poder podar el árbol e incluirla en la implementación de *b*).

e) ¿Se le ocurre una forma de escribir una función recursiva $\text{fs}(V, S, i)$ que, en analogía con el inciso *a*) del Ejercicio 5, determine el conjunto de invitados óptimo que incluya a $S \subseteq \{1, \ldots, i-1\}$ y que se obtenga agregando sólo invitados de $\{i, \ldots, n\}$? ¿Cuál es el problema? ¿Se le ocurre alguna manera de escapar a este problema?

f) Considerando la función fs (definida en *a*)) y el inciso anterior, observar que la cantidad posible de instancias es $\Omega(2^n)$. Concluir que la función fs no tiene la propiedad de superposición de subproblemas para el caso general del problema de la fiesta.

SeleccióndeActividades

21. Dado un conjunto de actividades $\mathcal{A} = \{A_1, \ldots, A_n\}$, el problema de selección de actividades consiste en encontrar un subconjunto de actividades $\mathcal{S}$ de cardinalidad máxima, tal que ningún par de actividades de $\mathcal{S}$ se solapen en el tiempo. Cada actividad A_i se realiza en algún intervalo de tiempo (s_i, t_i), siendo $s_i \in \mathbb{N}$ su momento inicial y $t_i \in \mathbb{N}$ su momento final. Suponemos que $1 \leq s_i < t_i \leq 2n$ para todo $1 \leq i \leq n$.

   a) Considerar la siguiente analogía con el problema de la fiesta: cada posible actividad es un invitado y dos actividades pueden "invitarse" a la fiesta cuando no se solapan en el tiempo. A partir de esta analogía, proponga un algoritmo de *backtracking* para resolver el problema de selección de actividades. ¿Cuál es la complejidad del algoritmo?

   b) Supongamos que $\mathcal{A}$ está ordenado por orden de comienzo de la actividad, i.e., $s_i \leq s_{i+1}$ para todo $1 \leq i < n$. Escribir una función recursiva $\text{act}(\mathcal{A}, \mathcal{S}, i)$ que encuentre el conjunto máximo de actividades seleccionables que contenga a $\mathcal{S} \subseteq \{A_1, \ldots, A_{i-1}\}$ y que se obtenga agregando únicamente actividades de $\{A_i, \ldots, A_n\}$. **Para reflexionar:** ¿por qué se puede definir act en este caso y no en el inciso *e*) del Ejercicio 20?

   c) Implementar un algoritmo de programación dinámica para el problema de selección de actividades que se base en la función del inciso *b*). ¿Cuál es su complejidad temporal y cuál es el espacio extra requerido?

   d) Considerar la siguiente estrategia golosa para resolver el problema de selección de actividades: elegir la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapen con las actividades ya elegidas. Demostrar que un algoritmo goloso que implementa la estrategia anterior es correcto. **Ayuda:** demostrar por inducción que la solución parcial $B_1, \ldots, B_i$ que brinda el algoritmo goloso en el paso i se puede extender a una solución óptima. Para ello, suponga en el paso inductivo que $B_1, \ldots, B_i, B_{i+1}$ es la solución golosa y que $B_1, \ldots, B_i, C_{i+1}, \ldots, C_j$ es la extensión óptima que existe por inducción y muestre que $B_1, \ldots, B_{i+1}, C_{i+2}, \ldots, C_j$ es una extensión óptima de $B_1, \ldots, B_{i+1}$.

   e) Mostrar una implementación del algoritmo cuya complejidad temporal sea $\mathcal{O}(n)$.