Práctica 1: Técnicas Algorítmicas

La siguiente guía contiene tres tipos de ejercicios que están claramente identificados.

- Los ejercicios marcados con † complementan lo visto en las clases teóricas y prácticas, mostrando con mucho detalle cómo es el proceso de resolución de un ejercicio tipo. Si bien estos ejercicios son opcionales, se sugiere su resolución previa a la de los ejercicios subsiguientes, especialmente en caso de encontrar dificultades con el resto de la guía.
- Los ejercicios marcados con ⋆ son ejercicios avanzados (no necesariamente difíciles), cuyo objetivo es profundizar en cada una de las técnicas y cómo las mismas están relacionadas.
- Finalmente, los ejercicios que no están marcados forman el corpus de ejercitación mínima que consideramos necesario para aprender los distintos temas.

Backtracking

1. (†) En este ejercicio vamos a resolver el problema de suma de subconjuntos, visto en la teórica, con la técnica de *backtracking*. A diferencia de la clase teórica, vamos a usar la representación binaria. 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. Notar que, a diferencia de la teórica, no necesitamos suponer que los elementos de C son todos distintos. Sí vamos a utilizar 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). Como se discute en la teórica, 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 $\mathrm{BT}\colon \mathcal{C} \times \mathbb{N} \to \{V, F\}$ (donde $\mathbb{N} = \{0, 1, 2, \ldots\}$, V indica verdadero y F, falso):

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

   Convencerse de que $\mathrm{BT}(C, k) = V$ 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 BT 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) BT(C, i, j): // implementa BT($\{c_1, \ldots, c_i\}, j$)
   2) Si $i = 0$, retornar ($j = 0$)
   3) Si no, retornar BT($C, i - 1, j$) $\vee$ BT($C, i - 1, j - C[i]$)

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

i) Definir otra regla de factibilidad, demostrando 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.

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.

Programación dinámica (y su relación con *backtracking*)

3. (†) 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 PD: $\{1, \ldots, n\} \times \{0, \ldots, k\} \to \{V, F\}$ (donde V indica verdadero y F falso) tal que:

   $$\text{PD}(C, i, j) = \begin{cases} j = 0 & \text{si } i = 0 \\ \text{PD}(C, i-1, j) \vee \text{PD}(C, i-1, j - C[i]) & \text{si no} \end{cases}$$

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

   b) Observar que, como C no cambia entre llamadas recursivas, existen $\mathcal{O}(nk)$ posibles entradas para PD. Concluir que, si $k \ll 2^n/n$, entonces necesariamente alguna instancia de PD es calculada más de una vez en 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 $\text{PD}(C, i, j)$, 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 $M[i, j] = \text{PD}(C, i, j)$ cuando se invoca $\text{PD}(C, i, j)$.

   1) Inicializar $M[i, j] = \perp$ para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.
   2) PD(C, i, j): // implementa $\text{BT}(\{c_1, \ldots, c_i\}, j) = \text{PD}(C, i, j)$ usando memoización
   3) Si $j < 0$, retornar **falso**
   4) Si $i = 0$, retornar $(j = 0)$
   5) Si $M[i, j] = \perp$, poner $M[i, j] = \text{PD}(C, i-1, j) \vee \text{PD}(C, i-1, j - C[i])$
   6) Retornar $M[i, j]$

   d) Concluir que $\text{PD}(C, n, k)$ resuelve el problema. Calcular la complejidad y compararla con el algoritmo BT 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

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

   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) PD(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 $M[i, j] := M[i-1, j] \vee (j - C[j] \geq 0 \wedge M[i-1, j - C[j]])$

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

4. (†) 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 $\text{BT}(B, c)$ para resolver el problema, donde $\text{BT}(B, c) = (c', q)$ 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, con la estrategia *top-down*, utilizando una función con parámetros B, i, j que compute $\text{BT}(\{b_1, \ldots, b_i\}, j)$. ¿Cuál es la complejidad del algoritmo?

c) Reescribir BT como una función recursiva $\text{PD}(B, i, j) = \text{BT}(\{b_1, \ldots, b_i\}, j)$ que implemente la idea anterior. A partir de esta función, determinar cuándo PD tiene la propiedad de *superposición de subproblemas*.

d) Definir una estructura de memoización para PD que permita acceder a $\text{PD}(B, i, j)$ 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) ($\star$) 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.

5. 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 para el fin del día j tiene c asteroides, es:
   - si $c = 0$, el máximo entre la m.g.n. finalizando el día $j - 1$ con 0 asteroides y aquella finalizando el día $j - 1$ con 1 asteroide y vendiéndolo en el día j;
   - si $c = j$, la m.g.n. finalizando el día $j - 1$ con $c - 1$ asteroides y comprando un asteroide en el día j (pensar por qué no se compara con la m.g.n. resultante de finalizar el día $j - 1$ con c asteroides);
   - si no valen las anteriores, 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.

   b) Escribir matemáticamente la formulación recursiva enunciada en *a*). Dar los valores de los casos bases 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 que resuelva el problema y explicar su complejidad temporal y espacial auxiliar.

6. 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, supongamos que tenemos una vara de longitud 10 metros que debe ser cortada a los 2, 4 y 7 metros desde un extremo, y escribimos esos números en la vara. Hay varias opciones, comentaremos dos:

   - Una es primero cortar donde dice 2, después donde dice 4 y después donde dice 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.
   - Otra opción es cortar primero donde dice 4, después donde dice 2, y finalmente donde dice 7. Esta opción resulta en 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 el mínimo, para todo lugar de corte c entre i y j, de la suma de la longitud de la vara de i a j con aquella 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*.

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

8. 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*.

9. 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 $3 + 1 \times 5 \uparrow 2 \times 1$, que se evalua 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*.

Golosos (≡ avariciosos ≡ *greedy*)

10. 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 $\{1, 4, 6, 2\}$ y otro con $\{5, 1, 5, 7, 9\}$, 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) Considerar la idea de ordenar crecientemente los elementos de un multiconjunto en un vector v y los del otro multiconjunto en otro vector w y, luego, recorrer alguno de izquierda a derecha hasta lograr emparejar a cada persona.

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

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

11. 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:

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

Ejercicios integradores

12. ($\star$) 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 $\text{BT}\colon \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ tal que, dados S y W compatibles, $\text{BT}(S, W)$ 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 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 $\text{PD}(V, S, i)$ que, en analogía con el inciso *a*) del Ejercicio 3, 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 BT (definida en *a*)) y el inciso anterior, observar que la cantidad posible de instancias es $\Omega(2^n)$. Concluir que la función BT no tiene la propiedad de superposición de subproblemas para el caso general del problema de la fiesta.

13. ($\star$) 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 PD($\mathcal{A}, \mathcal{S}, i$) 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 PD en este caso y no en el inciso *e*) del Ejercicio 12?

   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 optima de $B_1, \ldots, B_{i+1}$.

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