Práctica Algoritmos Golosos

Compilado: 14 de mayo de 2025

Golosos (≡ avariciosos ≡ *greedy*)

ParejasdeBaile

1. 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

2. 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

3. 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

4. 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 $0 = x_1 \leq x_2 \leq \dots x_n \leq M$.

   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

5. 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

6. 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

7. 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 $R = \{r_1, r_2, \ldots, r_n\}$[1] 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.

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*.

SeleccióndeActividades

8. 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?

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

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 $\{A_i, \ldots, A_n\}$.

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 $\mathcal{O}(n)$.