Algoritmos golosos

TDA

Facultad de ciencias Exactas

September 22, 2024

Algoritmos golosos

Llamamos algoritmos golosos (greedy) a los algoritmos que siempre hacen la mejor elección posible en el momento (sin calcular todas las opciones y ver cual es mejor).

Muchos algoritmos greedy no logran encontrar la solución óptima, ya que no siempre es posible.

Nosotros vamos a centrarnos en los algoritmos greedy que sí puedan encontrar la solución óptima, en casos en los que solo podamos aproximarnos usando este método vamos a utilizar otras técnicas.

Propiedades necesarias para que un problema admita una solución Greedy

1. Propiedad de elección Greedy (greedy choice): puedo generar una solución óptima global tomando elecciones óptimas locales (osea elecciones greedy).
2. Subestructura óptima: Un problema tiene subestructura óptima si una solución óptima al problema contiene dentro soluciones óptimas a sus subproblemas.

Suma

Enunciado

Dado un conjunto de números A un natural $k \leq |A|$ se quiere calcular $\max_{S \subseteq A, |S|=k} \sum_{a \in S} a$

Suma

Primero veamos si se cumplen las propiedades que necesitamos para usar greedy

Subestructura óptima: Sí

Greedy Choice: Sí

Suma

- "Es obvio, agarro los más grandes" (Pero por qué?)
- "Si agarras los más grandes vas a sumar más" (No es muy formal eso)
- "La suma de n elementos más grande es la suma de los n elementos más grandes" (Ahora hay que probarlo).

Función recursiva

$$\text{mejor\_suma(A, k)} = \begin{cases} \| k = 0 \implies \emptyset \\ \| k > 0 \implies max_{a \in A}\{a\} \cup mejor\_suma(A \setminus \{a\}, k-1) \end{cases}$$

Esta función recursiva podría interpretarse como "hacer backtracking eligiendo una sola rama del árbol" (la que elige Max(A) en este caso).

Tenemos en este caso una función recursiva que hace una unión entre el paso actual y el paso recursivo, estas funciones pueden expresarse de manera iterativa fácilmente.

Pseudocódigo

Algorithm 1 maxSuma(X: [Int], k: Int) → Int

```
1: X.sortAscendente()                          ▷ O(n log n)
2: val ← 0                                     ▷ O(1)
3: while k > 0 do                              ▷ O(k)
4:     val ← val + X[X.length() − k]           ▷ O(1)
5:     k − −                                   ▷ O(1)
6: end while
7: return val                                  ▷ O(1)
```

Complejidad: ▷ $\mathcal{O}(n \log n) + \mathcal{O}(k) = \mathcal{O}(n \log n)$

Demostración

- ¿Que forma tenemos de probarlo?
- Inducción.
- Extensión de la solución a una óptima. (Muy útil)
- Reducción al absurdo. (podría funcionar)

Inducción directa

- Caso base: Un sólo elemento.
- Caso recursivo: Agrego un elemento.
- En ambos casos se puede probar por absurdo.

Otra opción

- Mi algoritmo toma en cada paso el elemento más grande.
- Quiero ver que la solución de mi algoritmo puede extenderse a una óptima en 0 o más pasos.
- Si eso se cumple, cuando mi algoritmo termine va a poder extenderse a una óptima en 0 casos $\iff$ Es una solución óptima
- Ahora vamos a probarlo

Extender a una solución óptima

- Tengo una solución parcial sin elementos, puede extenderse a una óptima (trivial)
- Supongo que a una solución que puede extenderse a una óptima le agrego el elemento más grande que queda en el conjunto.
- Si no puede extenderse a una solución óptima es porque ninguna solución óptima usa ese elemento.
- Si fuera ese el caso, cualquier solución óptima ("la solución óptima"), elije otro elemento del conjunto en el paso i-ésimo.

Extender a una solución óptima - continuación

- Supongo que le cambio a la (una) solución óptima el i-ésimo elemento "i_o" por el de mi solución i_m.
- Si la solución óptima sumaba un total de $\mathcal{O}$, la solución óptima modificada suma:

$$\mathcal{O} + i_m - i_o$$

Extender a una solución óptima - Continuación

- Como sabemos que i_m era el mayor elemento en el i-ésimo paso, entonces:

$$i_m - i_o > 0 \implies \mathcal{O} + i_m - i_o > \mathcal{O}$$

- Pero $\mathcal{O}$ es una solución óptima! (No se puede haber una más óptima).
- Absurdo por suponer que mi solución no se puede extender a una solución óptima.

Por reducción al absurdo.

- El argumento acá sería el mismo que en el último paso.
- Supongo que mi solución no es óptima, entonces, la solución óptima contiene al menos un elemento que no pertenece a los k más grandes.
- Cambio uno de esos elementos por uno de los k más grandes que no tenía la solución óptima.
- Obtuve una solución mejor → Absurdo por suponer que mi solución no era óptima

Selección de actividades

Enunciado

Para un conjunto de actividades se conocen los deadlines $d_i \in N$ para completar cada una de ellas. Asumiendo que para completar cualquiera de ellas se tarda 1 unidad de tiempo, y que solo se puede trabajar en una a la vez, indicar cuál es la máxima cantidad de tareas que se pueden terminar antes de su deadline.

Selección de actividades

- Primero veamos si se cumplen las propiedades que necesitamos para usar greedy.
- Subestructura óptima: Sí
- Greedy Choice: Sí

Selección de actividades

- El algoritmo greedy para este problema consiste en elegir la actividad con la deadline más temprana
- Para demostrar esto podemos pensar en cómo va a ser la solución óptima
- Si tenemos un conjunto máximo de actividades que podemos realizar, vamos a poder ordenarlas por deadline.

Selección de actividades

- Al tenerlas ordenadas por deadline vamos a poder seguir completando todas, ya que si terminamos la primera actividad, como la segunda va a tener una deadline igual o posterior, todavía voy a poder hacerla (partiendo de una solución válida).
- Si una actividad termina en la hora 1, la voy a poder efectuar en la hora 1, si una termina en la 2 la voy a poder hacer en la 2, etc
- Si varias son parte de una solución óptima y tienen la misma deadline, voy a tener esa misma cantidad de horas libres antes de esa deadline para poder llevar a cabo todas

Pseudocódigo

Algorithm 2 maxValor(deadlines: [Int]) → [Int]

```
1: deadlines.sortAscendente()                    ▷ O(n log n)
2: B ← []                                        ▷ O(1)
3: for deadline ∈ deadlines do                   ▷ O(n)
4:     if deadline > B.length() then             ▷ O(1)
5:         B.append(deadline)                    ▷ O(1) amortizado
6:     end if
7: end for
8: return B                                      ▷ O(1)
```

Complejidad: ▷ $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$

Demostración

- Extensión de nuestra solución a la óptima
- En 0 hs no podemos seleccionar ninguna actividad
- Si tenemos una solución parcial de i actividades y le sumamos la actividad A_m (siendo A_m la actividad que todavía podemos realizar cuya deadline es más próxima), quiero ver que podemos extenderla a una solución óptima.

Demostración - continuación

- Supongamos que tenemos una solución óptima $\mathcal{O}$.
- O bien $\mathcal{O}$ incluye a A_m, o bien no la incluye.
- Si la incluye entonces puede ser extendida a una solución óptima.
- Luego, si no la incluye puedo tomar la actividad A_{i+1} de $\mathcal{O}$ y cambiarla por A_m. Esta solución también va a ser óptima ya que tiene la misma cantidad de actividades. También puedo extender a una solución óptima.
- En caso que en la solución óptima ya se encuentre A_m pero en otra posición, podemos cambiar A_m por el elemento A_{i+1}. Si la posición en la que se encuentra A_m es válida para A_m también lo será para A_{i+1}, ya que A_m era la actividad con deadline más próxima. En este caso sigo teniendo también la misma cantidad de actividades y puedo extender la solución a una óptima.

Optimizamos

Si nos damos cuenta, vamos a poder hacer como máximo n actividades.

Esto nos permite ordenar las actividades desde el principio con counting sort (todas las de n o más de deadline van al mismo bucket) y aplicar el algoritmo desde ahí.

En el mejor casi vamos a poder realizar las n actividades de deadline más pronto.

Selección de actividades (Reloaded)

Enunciado

Ahora cada actividad tiene también un valor v_i, y el objetivo es en realidad maximizar el valor de las actividades completadas antes de sus deadlines. Indicar el máximo valor que se puede obtener.

Selección de actividades (Reloaded)

- La idea esta vez es priorizar las actividades que dan más valor, ya que todas las tareas llevan la misma cantidad de tiempo.
- Pero tenemos que intentar que no nos impidan hacer otras actividades valiosas
- Ejemplo → 1: [Deadline=2, Valor=3], (2)[Deadline=1,Valor=2].
- En este caso si elegimos de forma greedy hacer primero la actividad más valiosa (1) vamos a desperdiciar la oportunidad de hacer otra actividad (2) y tener un valor total mayor.

Selección de actividades (Reloaded)

- Vamos a tomar las actividades más valiosas y hacerlas lo más tarde posible
- Con esto nos aseguramos de no desperdiciar los primeros horarios disponibles, ya que en estos podemos hacer tareas con deadlines cercanas.

Pseudocódigo

Algorithm 3 maximoValor(actividades: [(Int, Int)]) → [(Int, Int)]

```
1:  actividades.sortPorValorDescendente()
2:  actividades.sortPorDeadlineAscendente()
3:  B ← []                          ▷ Inicializo lista para devolver la respuesta
4:  T ← Array[n]                    ▷ Inicializo estructura para guardar tiempos ocupados
5:  for actividad ∈ actividades do                                          ▷ O(N)
6:      if puedoUbicar(actividad, T) then
7:          B.append(actividad)
8:      end if
9:  end for
10: return B
```

Pseudocódigo

Algorithm 4 puedoUbicar(actividad: (Int, Int), T: Array) → Bool

```
1: iterator = actividad.first          ▷ Inicializo un iterador en la deadline
2: while iterator ≥ 0 do                                              ▷ O(N)
3:     if T[iterator] == 0 then                          ▷ (0 representa libre)
4:         T[iterator] = 1                                ▷ Marco como ocupada
5:         return True
6:     end if
7: end while
8: return False                                          ▷ No lo pude ubicar
```

Complejidad

- Los algoritmos de búsqueda pueden hacerse en $\mathcal{O}(N * log(N))$.
- El ciclo principal realiza N iteraciones.
- En cada iteracion la función "puedoUbicar" realiza otras N (en el peor caso).
- Nos queda $\mathcal{O}(N) * \mathcal{O}(N) = \mathcal{O}(N^2)$.
- Se puede optimizar la estructura de tiempos ocupados para mejorar la complejidad a $\mathcal{O}(N * log(N))$.

Definición de la solución

- Para poder demostrar debemos tener en cuenta tanto el valor de las actividades como su posición (en que momento son realizadas), para eso defino la solución óptima como una lista de tuplas "(valor, posición) ordenadas por valor.
- En este caso la solución hasta el i-ésimo elemento se define como una solución con los i elementos de mayor valor en sus posiciones correspondientes.

Demostración

- Nuevamente buscamos que nuestra solución sea extensible a una óptima.
- Dada una solución con 0 elementos podemos extenderla a una óptima.
- Dada una solución con i elementos que se puede extender a una óptima, quiero ver que al agregarle el elemento (g_{i+1}, h_{i+1}) puedo seguir extendiendo la solución a una óptima. (siendo g_{i+1} actividad con más valor y menor deadline de las que quedan disponibles después de colocar los primeros i elementos en sus posiciones correspondientes y h_{i+1} la posición más tardía en la que puedo colocar g_{i+1}.)

Demostración - continuación

Supongo que tengo una solución $\mathcal{O}$ óptima que es una extensión de la solución con i elementos. Esta solución o bien contiene a (g_{i+1}, h_{i+1}), o bien no la contiene.
Si no la contiene:

- Puedo ubicar g_{i+1} en la(g_{i+1}, h_{i+1}).
- Si la posición h_{i+1} estaba libre obtendría una solución más óptima que la óptima. (Absurdo)
- Si contiene otra actividad en h_{i+1} la reemplazo, y como g_{i+1} era la actividad con mayor valor disponible, voy a obtener un valor mayor o igual. (Acá podemos dividir en dos casos, si es mayor es absurdo y si es igual se puede extender a una solución óptima).

Demostración - continuación

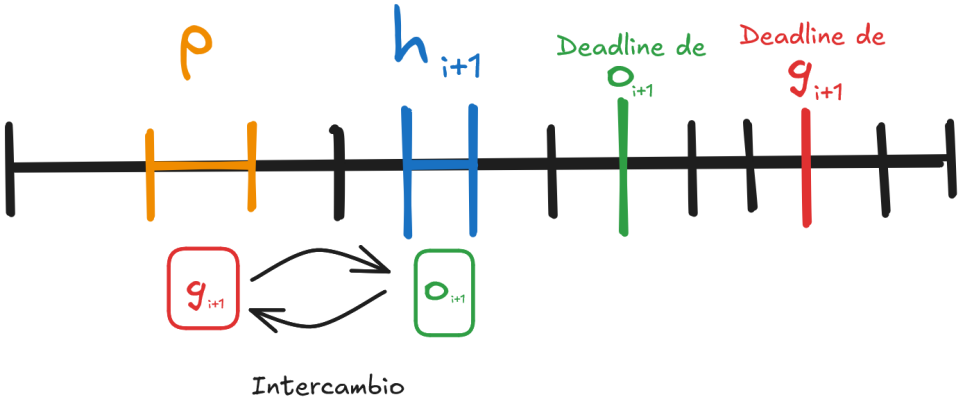
Si la contiene:

- Si se encuentra en la posición h_{i+1} se puede extender a una solución óptima.
- Si se encuentra en otra posición la intercambio. Pero tengo que demostrar que el resultado es una solución válida. Para hacerlo llamo o_{i+1} a la actividad de la solución óptima que se encuentre en la posición h_{i+1} y p a la posición de g_{i+1} en $\mathcal{O}$.

Demostración - continuación

- Sabemos que la deadline de o_{i+1} es mayor a h_{i+1} ya que en O se encuentra en esa posición.
- Sabemos que p es una posición válida para g_{i+1} porque en ella está g_{i+1} en $\mathcal{O}$.
- Finalmente, como h_{i+1} era la posición más tardía en la que podíamos ubicar a o_{i+1}, p se encuentra antes de h_{i+1}.
- Luego, si cambiamos la posición de o_{i+1} y g_{i+1} obtenemos una solución válida. (A o_{i+1} lo hacemos mas temprano, por lo que no hay problema y a g_{i+1} lo hacemos en la posición que eligió nuestro algoritmo, por lo que es previa a su deadline. Tampoco hay problemas.)

Demostración - imagen



Demostración - final

- Con esto queda demosrado que el intercambio nos lleva a una solución válida, y como no modificamos el valor total de $\mathcal{O}$, sigue siendo óptima.
- Ya cubrimos todos los casos y demostramos que podemos extender cualquier solucion parcial a la que se le agregue el elemento seleccionado por nuestro algoritmo a una solución óptima en 0 o más pasos.
- Por lo tanto, cuando nuestro algoritmo termine, la solución obtenida será una solución válida y óptima.

Containers[1]

Enunciado

Dado un conjunto de containers, se los quiere apilar en un barco de forma tal que el desapilado se pueda hacer sin tener que reubicar los containers. Más formalmente, cada container i tiene una etiqueta e_i. Si $e_i < e_j$ entonces el container i se debe desapilar antes que el j, por lo que no puede ser colocado por debajo del container j en una pila.

[1] https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3503

Containers[2]

Enunciado

Se sabe el orden en que llegan los containers, y lo que se quiere descubrir es cuál es la mínima cantidad de stacks necesaria para poder llevar a cabo la tarea.

Ejemplo (donde las etiquetas son letras): Si los containers llegan como `CCBBAA` podemos usar una única pila, ya que los de etiqueta `A` nos quedan arriba, y luego están los de etiqueta `B`.
Otro ejemplo: Si llegan como `CBACBACBA` entonces necesitamos 3 pilas como mínimo.

[2] `https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3503`

Containers

- Intentemos proponer alguna estrategia para apilar los containers a medida que llega que suene razonable.
- Idea: cuando llega el container i con etiqueta e_i lo apilamos en la pila p_k que sea menos restrictiva, en el siguiente sentido: si podemos apilar i tanto en p_{k_1} como en p_{k_2}, entonces usamos la que minimice $top(p_k)$.
- Si no lo podemos apilar en ninguna, creamos una nueva vacía.
- ¿Esto anda en los ejemplos?

Containers

- Intentemos probar que esta estrategia es óptima. Para eso necesitamos formalizar qué es una solución a este problema.
- Una solución son n números $s_1, \ldots, s_n$ donde s_i indica en qué pila debemos colocar el container i.
- Esta asignación debe cumplir la siguiente propiedad: si nos dicen de colocar el container i en s_i entonces debe pasar que el último container puesto en s_i tenga una etiqueta mayor o igual a la del container i. Más formalmente, si $k = \max\{j : j < i \wedge s_j = s_i\}$ entonces $e_i \leq e_k$.
- El valor de una solución es la cantidad distinta de containers que usa. Una solución óptima debería minimizar esto.

Containers

- Probemos que nuestra estrategia es óptima. Para eso, llamemos $g_1, \ldots, g_n$ a la asignación que genera nuestra estrategia golosa. Vamos a probar, por inducción, que cada prefijo $g_1, \ldots, g_i$ cumple estar incluida en alguna solución óptima, para todo i. Naturalmente, con probar esto concluímos que $g_1, \ldots, g_n$ es óptima.
- Para $i = 0$ esto vale trivialmente (la solución vacía está incluida en todas).
- Para el caso inductivo, supongamos que $g_1, \ldots, g_i$ está incluida en alguna solución óptima $O = g_1, \ldots g_i, s_{i+1}, \ldots s_n$. Queremos ver que $g_1, \ldots, g_{i+1}$ también está incluida en una óptima.
- Si $s_{i+1} = g_{i+1}$ ya terminamos (¿No?), asi que imaginemos que $s_{i+1} \neq g_{i+1}$.

Containers

- Intuivamente (o, por lo menos, según nuestra intuición) g_{i+1} era mejor que s_{i+1}, debido a qué condicionaba menos las siguientes jugadas. ¿Cómo podemos formalizar esto?
- Agarremos la solución óptima $g_1, \ldots, g_i, s_{i+1}, \ldots, s_n$ y cambiémosla por $g_1, \ldots, g_i, g_{i+1}, s_{i+2}, \ldots, s_n$. ¿Es esta solución válida?
- A lo mejor no, porque capaz después venía un container $j > i+1$ con $e_j > e_{i+1}$ y $s_j = g_{i+1}$. ¿Podemos redireccionar esta jugada?
- Podemos decir lo siguiente: todas las siguientes jugadas s_j que iban a g_{i+1} las podemos mandar a s_{i+1}, y las que iban a s_{i+1} las mandamos a g_{i+1}. Mostremos que este *swap* hace que la solución vuelva a ser válida.

Containers

- Para convencernos de esto, pensemos en el siguiente container que la solución óptima O mandaba a s_{i+1}. Ahora va a g_{i+1}, y esto no puede ser inválido porque al final del día se está apilando sobre el mismo container, con misma etiqueta (aunque en otra pila).
- Por otro lado, pensemos en el siguiente container k que la solución óptima O mandaba a g_{i+1}. Ahora va a s_{i+1}. Como antes era válida, tenemos que el top de la pila g_{i+1} hasta este momento es mayor o igual a e_k. Aparte, por como funciona la estrategia golosa debe pasar que el top de s_{i+1} era todavía mayor o igual al de g_{i+1}, dado que estamos eligiendo el de menor etiqueta de los válidos. Luego, por transitividad, el container k se va a colocar de forma correcta.

Containers

- Aparte, esta solución nueva O' que definimos no usa más containers que O. Esto es inmediato observando que el único número nuevo que pudo haber aparecido en O es g_{i+1}, pero g_{i+1} es un container nuevo únicamente si no había forma de colocar al container $i+1$ en la solución parcial hasta el paso i, en cuyo caso s_{i+1} también era un número nuevo (pila nueva).
- Con esto mostramos que $g_1, \ldots g_i, g_{i+1}$ está contenida en alguna solución óptima. Luego, por inducción, concluimos que la solución golosa es óptima.
- ¿En qué complejidad podemos encontrar esta solución?
- Se puede hacer en $O(n \log R)$ donde R es la cantidad de pilas óptima (R de *Respuesta*), usando un ABB para buscar la siguiente pila. Sino, es fácil hacerlo en $O(nR)$.

Impuestos

Enunciado

Juan es actor de hollywood, y como le va bastante bien ya tiene varias ofertas de trabajo para este año. De cada oferta i conoce la fecha f_i en la que inicia el contrato, el cual puede aceptar o no. Si lo acepta, entonces recibe d_i dólares.

Impuestos

Enunciado

Juan está interesado, naturalmente, en maximizar la cantidad de dólares que recibe. Sin embargo, si acepta múltiples trabajos la AFIP podría inspeccionarlo, cosa que preferiría evitar por motivos varios. Él conoce las fechas a_i en las cuales la AFIP puede iniciar una inspección. Para cada fecha conoce también un threshold t_i: si para la fecha a_i Juan inició t_i o más contratos, entonces le harán una inspección.
Juan quiere saber cuál es la máxima cantidad de dólares que puede ganar sin que AFIP le haga una inspección.

Impuestos

- Consideremos la primera fecha a_1 donde puede haber una inspección. A esa altura, Juan solo puede estar involucrado en a lo sumo $t_1 - 1$ contratos. ¿Cuáles son los mejores que puede elegir? Los $t_1 - 1$ más caros de los que empiezan en el intervalo $[0, a_i]$. De ahora en más, notemos como $C_{[a,b]}$ al conjunto de contratos que inician en el intervalo $[a, b]$.
- Luego, al momento de llegar a a_2 debe tener a lo sumo $t_2 - 1$ contratos. ¿Cuáles son los mejores ahora? No son exactamente los $t_2 - 1$ mas caros de $C_{[0,a_2]}$, ya que podría no ser posible tomarlos sin generar la inspección en a_1.

Impuestos

- Intuitivamente, al momento de pasar a_1 podemos olvidarnos de todos los contratos que no estuvieran entre los $t_1 - 1$ más caros. O sea, si O_1 es el conjunto de contratos óptimos para maximizar la ganancia sin tener una inspección en a_1, luego los mejores contratos al momento de llegar a a_2 se obtienen tomando los $t_2 - 1$ mejores contratos en la unión $O_1 \cup C_{[a_1+1,a_2]}$.
- Más formalmente, el conjunto de contratos óptimos O_k que se puede hacer hasta llega a a_k se obtiene tomando los $t_k - 1$ contratos más caros de $O_{k-1} \cup C_{[a_{k-1}+1,a_k]}$.

Impuestos

- ¿En qué complejidad podemos implementar el cómputo de los conjuntos O_k?
- Se puede hacer en $O(n \log n)$, donde n es la cantidad de fechas relevantes (i.e. inicios de contratos más inspecciones de AFIP) usando un heap.