Estrategias

Estructuras de Datos y Algoritmos I

Universidad Nacional de Rosario

May 29, 2025

Contenido de la presentación

Las estrategias de la presentación suelen estar asociadas a problemas de optimización. Es decir, problemas en los cuales no alcanza con encontrar una de las soluciones correctas sino que hay que encontrar una de las menores/mayores soluciones correctas según algún criterio de orden apropiado.
Por ejemplo,

- dar vuelto con la menor cantidad de monedas,
- establecer la ruta con menos peajes ó
- encontrar la asignación de aulas que maximice la cantidad de clases que se pueden dar.

¿Qué es?

La técnica de programación dinámica (*dynamic programming*) soluciona problemas combinando soluciones a subproblemas. En general, aplica cuando los subproblemas se solapan porque evita recomputar la solución a los subproblemas comunes.

Ejemplo

Un aserradero se encarga de cortar troncos de árboles en partes más pequeñas para su venta. El precio de una sección de tronco depende de su largo y responde a la siguiente tabla

largo	1	2	3	4	5	6	7	8	9	10
precio	1	5	8	9	10	17	17	20	24	30

Dado un tronco de largo n y la tabla de precios queremos saber de qué manera conviene cortarlo (si es que siquiera conviene) para maximizar el precio de venta.

Ejemplo

Un aserradero se encarga de cortar troncos de árboles en partes más pequeñas para su venta. El precio de una sección de tronco depende de su largo y responde a la siguiente tabla

largo	1	2	3	4	5	6	7	8	9	10
precio	1	5	8	9	10	17	17	20	24	30

Dado un tronco de largo n y la tabla de precios queremos saber de qué manera conviene cortarlo (si es que siquiera conviene) para maximizar el precio de venta. ¿De cuántas formas se puede cortar un tronco de largo n?

Para $n = 4$

$$4 = 4 + 0 \qquad (\$9)$$
$$4 = 3 + 1 = 1 + 3 \qquad (\$9)$$
$$4 = 2 + 2 \qquad (\$10)$$
$$4 = 2 + 1 + 1 = 1 + 1 + 2 \qquad (\$7)$$
$$4 = 1 + 2 + 1 = 1 + 1 + 2 \qquad (\$7)$$
$$4 = 1 + 1 + 1 + 1 \qquad (\$4)$$

largo	1	2	3	4	5	6	7	8	9	10
precio	1	5	8	9	10	17	17	20	24	30

Para $n = 4$

$$
\begin{aligned}
4 &= 4 + 0 && (\$9)\\
4 &= 3 + 1 = 1 + 3 && (\$9)\\
4 &= 2 + 2 && (\$10)\\
4 &= 2 + 1 + 1 = 1 + 1 + 2 && (\$7)\\
4 &= 1 + 2 + 1 = 1 + 1 + 2 && (\$7)\\
4 &= 1 + 1 + 1 + 1 && (\$4)
\end{aligned}
$$

largo	1	2	3	4	5	6	7	8	9	10
precio	1	5	8	9	10	17	17	20	24	30

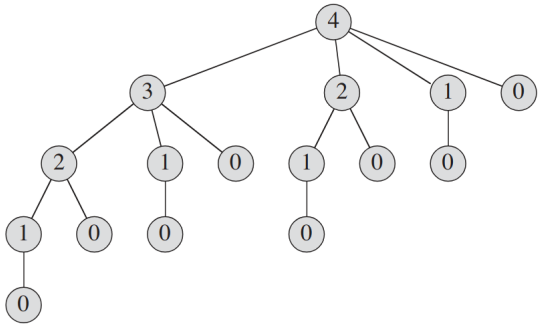
Luego de cada corte continuamos maximizando sobre un problema más chico.

Solución recursiva top-down

```
cortar_tronco(p, n):
  if (n == 0)
    return 0

  q = -\inf
  for i = 1 to n
    q = max(q, p[i] +
          cortar_tronco(p,n-i))
  return q
```

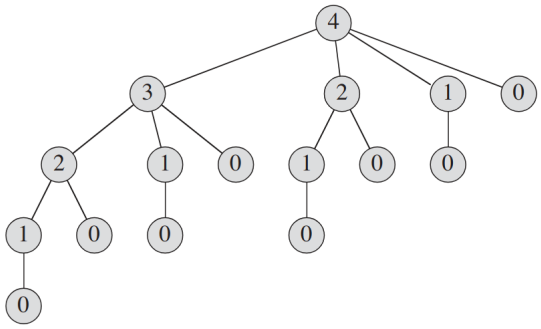
Solución recursiva top-down

```
cortar_tronco(p, n):
  if (n == 0)
    return 0

  q = -\inf
  for i = 1 to n
    q = max(q, p[i] +
          cortar_tronco(p,n-i))
  return q
```



Solución recursiva top-down

```
cortar_tronco(p, n):
  if (n == 0)
    return 0

  q = -\inf
  for i = 1 to n
    q = max(q, p[i] +
          cortar_tronco(p,n-i))
  return q
```



Costo : $O(2^n)$ $(C(n) = C(n-1) + C(n-2) + ... + C(0) = \sum_{i=0}^{n} C(i))$

Solución interativa top-down (con memoization)

```
cortar_tronco_memo(p, n):
  let r[0..n]
  for i = 0 to n
    r[i] = -\inf
  return
    cortar_tronco_memo_aux(p,n,r)
```

```
cortar_tronco_memo_aux(p,n,r):
  if (r[n] ≥ 0)
    return r[n]

  if (n == 0)
    q = 0
  else {
  q = -\inf
  for i = 1 to n
    q = max(q, p[i] +
      cortar_tronco_memo_aux(p, n - i, r))
  }

  r[n] = q
  return q
```

Solución interativa top-down (con memoization)

```
cortar_tronco_memo(p, n):
  let r[0..n]
  for i = 0 to n
    r[i] = -\inf
  return
    cortar_tronco_memo_aux(p,n,r)
```

```
cortar_tronco_memo_aux(p,n,r):
  if (r[n] ≥ 0)
    return r[n]

  if (n == 0)
    q = 0
  else {
  q = -\inf
  for i = 1 to n
    q = max(q, p[i] +
      cortar_tronco_memo_aux(p, n - i, r))
  }

  r[n] = q
  return q
```

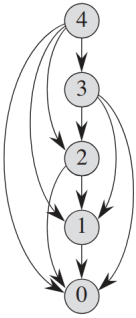
Costo temporal: $O(n^2)$, Costo espacial: $O(n)$

Solución interativa bottom-up

```
cortar_tronco_bottom-up(p, n):
  let r[0..n]
  r[0] = 0
  for j = 1 to n
    q = -\inf
    for i = 1 to j
      q = max(q, p[i] + r[j-i])
    r[j] = q
  return r[n]
```

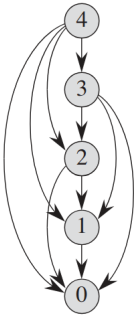
Solución interativa bottom-up

```
cortar_tronco_bottom-up(p, n):
  let r[0..n]
  r[0] = 0
  for j = 1 to n
    q = -\inf
    for i = 1 to j
      q = max(q, p[i] + r[j-i])
    r[j] = q
  return r[n]
```



Solución interativa bottom-up

```
cortar_tronco_bottom-up(p, n):
  let r[0..n]
  r[0] = 0
  for j = 1 to n
    q = -\inf
    for i = 1 to j
      q = max(q, p[i] + r[j-i])
    r[j] = q
  return r[n]
```



Costo temporal: $O(n^2)$, Costo espacial: $O(n)$

Otro ejemplo

Consideremos ahora el problema de la multiplicación de matrices. Dada una cadena de Matrices $A_1, A_2, ..., A_n$ queremos determinar en qué orden conviene multiplicarlas para realizar la mínima cantidad de operaciones.
Para $n = 3$ y dimensiones 10×100, 100×5 y 5×50 realizamos

- $((A_1A_2)A_3)$ $10.100.5 + 10.5.50 = 7500$
- $(A_1(A_2A_3))$ $100.5.50 + 10.100.50 = 75000$

Otro ejemplo

Consideremos ahora el problema de la multiplicación de matrices. Dada una cadena de Matrices $A_1, A_2, ..., A_n$ queremos determinar en qué orden conviene multiplicarlas para realizar la mínima cantidad de operaciones.
Para $n = 3$ y dimensiones 10×100, 100×5 y 5×50 realizamos

- $((A_1A_2)A_3)$ $10.100.5 + 10.5.50 = 7500$
- $(A_1(A_2A_3))$ $100.5.50 + 10.100.50 = 75000$

¿Podemos aplicar programación dinámica?

¿Cuándo aplicar programación dinámica?

- Subestructura óptima: la solución óptima del problema contiene la solución óptima de subproblemas.
  Cortar un tronco de manera óptima involucra cortar las secciones obtenidas de manera óptima.
  Encontrar una parentización óptima para el producto de matrices involucra encontrar una parentización óptima para las cadenas resultantes.
- Subproblemas solapados: el espacio de solución es "pequeño" en el sentido de que para resolver un problema se recurre a resolver los mismos subproblemas y no nuevos en cada toma de decisión.

Más problemas

- Encontrar un camino de peso mínimo en un grafo.
- Determinar un conjunto de objetos de valor máximo para meter en una mochila de tamaño acotado.

Más problemas

- Encontrar un camino de peso mínimo en un grafo.
- Determinar un conjunto de objetos de valor máximo para meter en una mochila de tamaño acotado.

¿Tienen subestructura óptima? ¿Se solapan los subproblemas?

Estrategia Greedy

Son una *clase* de algoritmos que funcionan tomando decisiones *localmente óptimas*. (En general, aplican a problemas de optimización.)
Es decir, hacen lo que parece ser la mejor opción en el momento, sin que necesariamente sea una buena opción a largo plazo.

[0] a.k.a.: Voraces, Golosos, Avaros.

Estrategia Greedy

Son una *clase* de algoritmos que funcionan tomando decisiones *localmente óptimas*. (En general, aplican a problemas de optimización.)
Es decir, hacen lo que parece ser la mejor opción en el momento, sin que necesariamente sea una buena opción a largo plazo.
En algunos problemas... resulta que esto de todas formas nos da el óptimo.
Caracterizar esos problemas es lo importante.

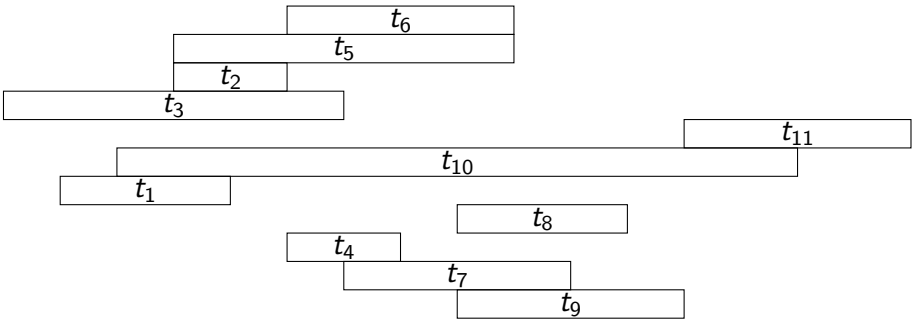
[0] a.k.a.: Voraces, Golosos, Avaros.

Ejemplo: Selección de Actividades

- Tenemos n actividades, cada una con un tiempo de comienzo c_i y finalización f_i. La actividad nos "ocupa" durante $[c_i, f_i)$.
- No podemos tomar todas las actividades (algunas se solapan).

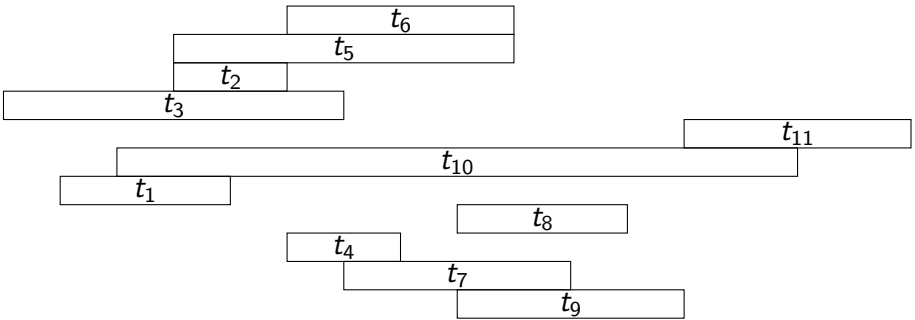
Ejemplo: Selección de Actividades

- Tenemos n actividades, cada una con un tiempo de comienzo c_i y finalización f_i. La actividad nos "ocupa" durante $[c_i, f_i)$.
- No podemos tomar todas las actividades (algunas se solapan).



Ejemplo: Selección de Actividades

- Tenemos n actividades, cada una con un tiempo de comienzo c_i y finalización f_i. La actividad nos "ocupa" durante $[c_i, f_i)$.
- No podemos tomar todas las actividades (algunas se solapan).



- ¿Cómo encontrar un conjunto *máximo* de actividades?

Selección de Actividades - Solución con DP

Para encontrar un conjunto máximo entre T_0 y T_1, hacemos:

Selección de Actividades - Solución con DP

Para encontrar un conjunto máximo entre T_0 y T_1, hacemos:

1. Elegimos una actividad i que "entre" en $[T_0, T_1)$. Supongamos que es $[c_i, f_i)$.

Selección de Actividades - Solución con DP

Para encontrar un conjunto máximo entre T_0 y T_1, hacemos:

1. Elegimos una actividad i que "entre" en $[T_0, T_1)$. Supongamos que es $[c_i, f_i)$.
2. Resolvemos optimamente los subproblemas $[T_0, c_i)$ ("antes") y $[f_i, T_1)$ ("después").

Selección de Actividades - Solución con DP

Para encontrar un conjunto máximo entre T_0 y T_1, hacemos:

1. Elegimos una actividad i que "entre" en $[T_0, T_1)$. Supongamos que es $[c_i, f_i)$.
2. Resolvemos optimamente los subproblemas $[T_0, c_i)$ ("antes") y $[f_i, T_1)$ ("después").
3. No sabemos cuál i elegir... así que repetimos para todo i posible (y memoizamos subproblemas).

Selección de Actividades - Solución con DP

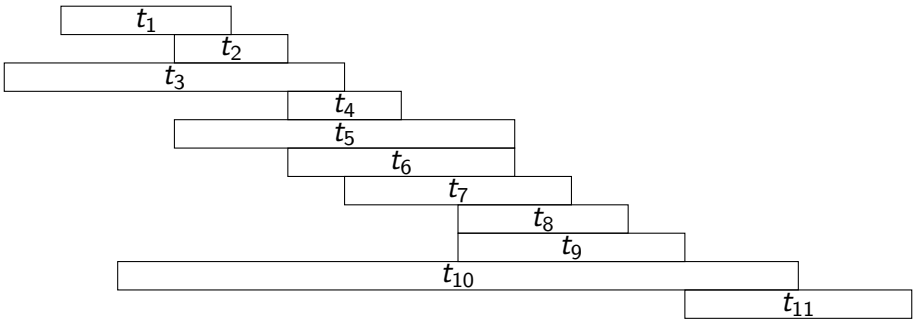
Para encontrar un conjunto máximo entre T_0 y T_1, hacemos:

1. Elegimos una actividad i que "entre" en $[T_0, T_1)$. Supongamos que es $[c_i, f_i)$.
2. Resolvemos optimamente los subproblemas $[T_0, c_i)$ ("antes") y $[f_i, T_1)$ ("después").
3. No sabemos cuál i elegir... así que repetimos para todo i posible (y memoizamos subproblemas).

Con esto sale un algoritmo de programación dinámica en $O(n^3)$. Algunas optimizaciones lo pueden mejorar bastante, pero siempre con memoria (mínimo) $O(n)$.

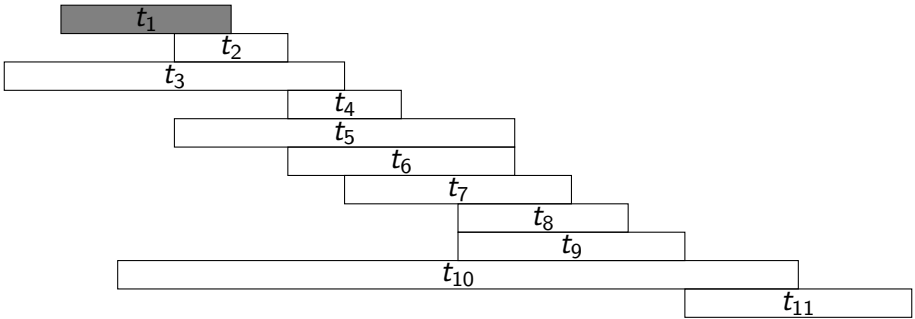
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



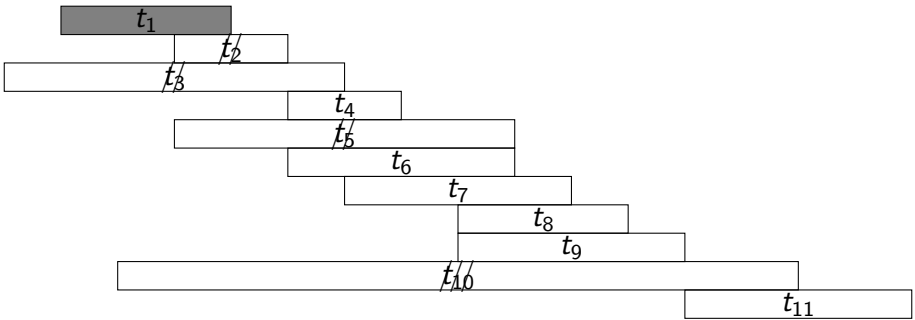
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



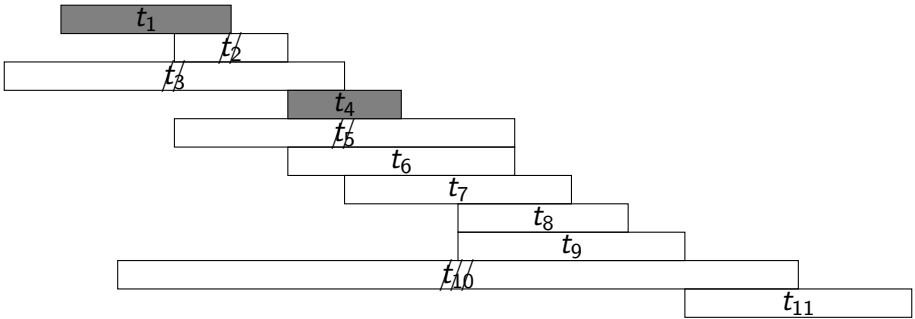
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



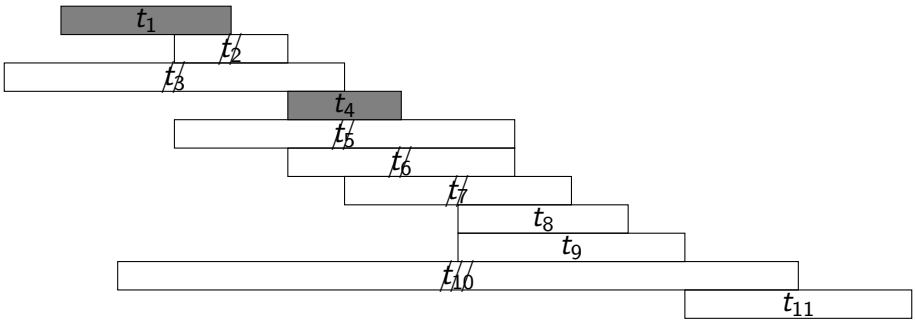
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



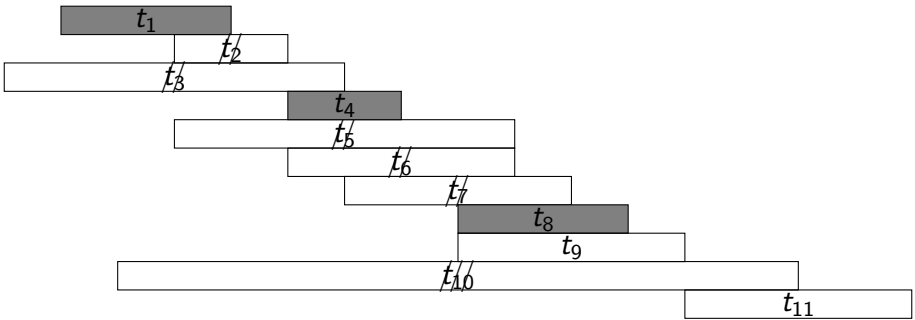
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



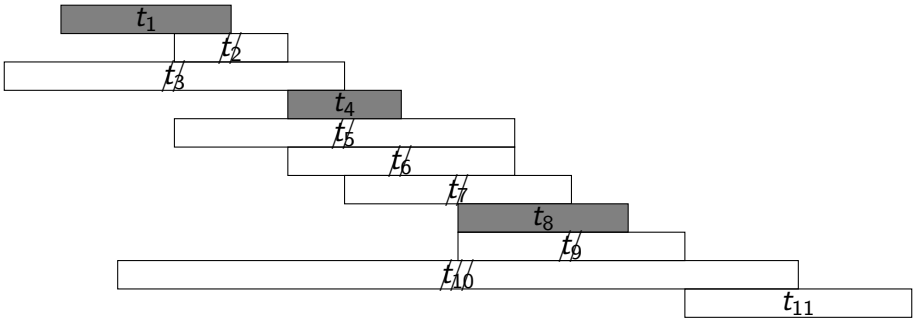
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



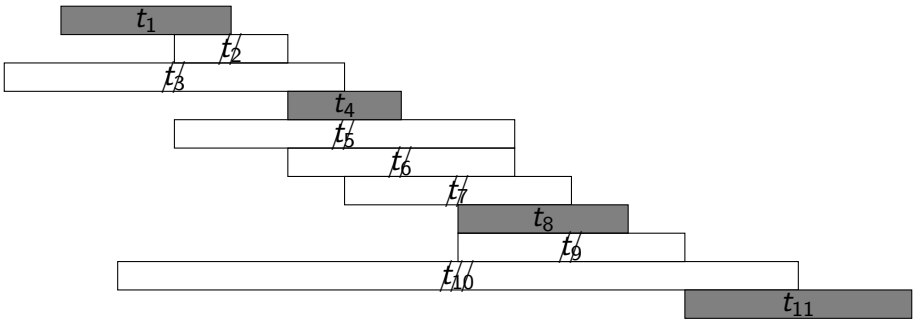
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



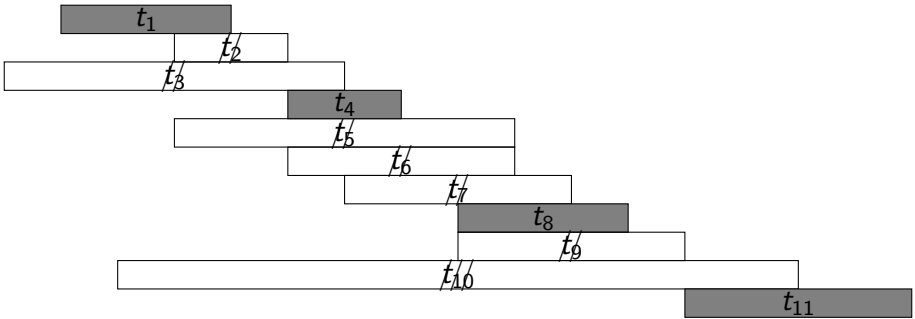
Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



Selección de Actividades - Solución Greedy

Tomemos otro enfoque. ¿Cuál es la primera tarea que deberíamos ejecutar? Una idea es tomar la que *termine* primero, de manera de "liberarnos" lo antes posible.



Resulta ser un conjunto máximo... ¿pasará siempre?

¿Es óptimo?

Para este problema, se puede *demostrar* que la elección greedy siempre lleva a *una* solución óptima.

¿Es óptimo?

Para este problema, se puede *demostrar* que la elección greedy siempre lleva a *una* solución óptima.

Lemma

Si $C \subseteq S$ es un conjunto de actividades máximo cuya primera actividad es t_1, que no es la primera en terminar de S ($= t_0$), entonces $C - \{t_1\} \cup \{t_0\}$ es también un conjunto válido (y de tamaño máximo).

¿Es óptimo?

Para este problema, se puede *demostrar* que la elección greedy siempre lleva a *una* solución óptima.

Lemma

Si $C \subseteq S$ es un conjunto de actividades máximo cuya primera actividad es t_1, que no es la primera en terminar de S ($= t_0$), entonces $C - \{t_1\} \cup \{t_0\}$ es también un conjunto válido (y de tamaño máximo).

Proof.

La primera actividad t_1 tiene comienzo c_1 y final f_1. Claramente toda otra actividad de C tiene comienzo $\geq f_1$ o se solaparía. Si removemos t_1 y agregamos t_0, no puede haber ningún solapamiento, porque:

- $f_0 \leq f_1$, entonces no puede haber solapamiento hacia la derecha
- t_1 es la primer actividad en comenzar en C, entonces tampoco hay solapamientos hacia la izquierda

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.
2. Eliminar las incompatibles, no pueden estar en la solución.

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.
2. Eliminar las incompatibles, no pueden estar en la solución.
3. Si quedan actividades, tenemos un subproblema. Repetimos con la misma lógica.

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.
2. Eliminar las incompatibles, no pueden estar en la solución.
3. Si quedan actividades, tenemos un subproblema. Repetimos con la misma lógica.

Esto lleva a un algoritmo lineal con uso de memoria $O(1)$.

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.
2. Eliminar las incompatibles, no pueden estar en la solución.
3. Si quedan actividades, tenemos un subproblema. Repetimos con la misma lógica.

Esto lleva a un algoritmo lineal con uso de memoria $O(1)$. (Asumiendo que ya estaban ordenadas por tiempo de finalización...

El algoritmo

Dado que tenemos el lema anterior, sabemos que existe una solución óptima que incluye a la primer actividad en terminar. El algoritmo resulta:

1. Tomar la primer actividad en terminar y agregarla al conjunto– tiene que haber una solución óptima con ella.
2. Eliminar las incompatibles, no pueden estar en la solución.
3. Si quedan actividades, tenemos un subproblema. Repetimos con la misma lógica.

Esto lleva a un algoritmo lineal con uso de memoria $O(1)$. (Asumiendo que ya estaban ordenadas por tiempo de finalización... si no, es $O(n \lg n)$).

Forma genérica

```
greedy(C : conjunto) : conjunto =
  S ← vacio
  while (¬solucion (S) ∧ ¬vacio(C))
    x ← menor_elemento(C) // el x que minimiza el costo
    C ← C \ {x}
    if (factible(S ∪ {x}))
      S ← S ∪ {x}

  if solucion(S)
    return S
  else
    no hay solucion!
```

Orden de algoritmo Greedy

Notar como estamos *primero* tomando la decisión (de manera *local*) y luego recién tenemos un subproblema que resolver.

En Divide&Conquer y Programación Dinámica, primero resolvíamos los subproblemas para *luego* tomar una decisión óptima con esa información.

Selección de Actividades - Otras Ideas

Hay otras opciones para la elección greedy.

- La que comienza primero
- La más corta
- La que comienza último
- La que se solapa con el menor número posible

Ejercicios: ¿cuáles funcionan y cuáles no?

Otro ejemplo: Dar Cambio

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.
- Dado una cantidad de dinero N, queremos dar la *mínima* cantidad de monedas para la misma.
- Algoritmo Greedy:

Otro ejemplo: Dar Cambio

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.
- Dado una cantidad de dinero N, queremos dar la *mínima* cantidad de monedas para la misma.
- Algoritmo Greedy: tomar las monedas más grandes posibles mientras no nos pasemos.

Dar Cambio - Ejemplo

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.

Para 74:

Dar Cambio - Ejemplo

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.

Para 74:

- Tomamos 3 monedas de 20, $\rightsquigarrow$ restan 14

Dar Cambio - Ejemplo

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.

Para 74:

- Tomamos 3 monedas de 20, $\rightsquigarrow$ restan 14
- Tomamos 1 moneda de 10, $\rightsquigarrow$ restan 4

Dar Cambio - Ejemplo

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.

Para 74:

- Tomamos 3 monedas de 20, $\rightsquigarrow$ restan 14
- Tomamos 1 moneda de 10, $\rightsquigarrow$ restan 4
- No tomamos monedas de 5.

Dar Cambio - Ejemplo

- Tenemos monedas de alguna denominación, ej: 1, 5, 10, 20.

Para 74:

- Tomamos 3 monedas de 20, $\rightsquigarrow$ restan 14
- Tomamos 1 moneda de 10, $\rightsquigarrow$ restan 4
- No tomamos monedas de 5.
- Tomamos 4 monedas de 1, listo.

Dar Cambio - Optimalidad

¿Es óptimo? Supongamos que dimos cambio y en algún momento *no* tomamos la elección greedy.

Dar Cambio - Optimalidad

¿Es óptimo? Supongamos que dimos cambio y en algún momento *no* tomamos la elección greedy.
Por ejemplo, estábamos dando cambio a una cantidad $C \geq 20$, y no elegimos la moneda de 20.

Dar Cambio - Optimalidad

¿Es óptimo? Supongamos que dimos cambio y en algún momento *no* tomamos la elección greedy.
Por ejemplo, estábamos dando cambio a una cantidad $C \geq 20$, y no elegimos la moneda de 20.
Tenemos a, b, c tales que $a + 5b + 10c = C \geq 20$. Convencerse de que tiene que haber un "subconjunto" (a', b', c') tal que $a' + 5b' + c' = 20$, y por lo tanto la solución no puede ser óptima. (Se puede demostrar por inducción en C).

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.
- Greedy: 1x17 + 7x1 (8 monedas)

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.
- Greedy: 1x17 + 7x1 (8 monedas)
- Óptima: 3x8 (3 monedas)

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.
- Greedy: 1x17 + 7x1 (8 monedas)
- Óptima: 3x8 (3 monedas)

Demostrar que algoritmo greedy da el óptimo para un conjunto S de monedas no es trivial (hay algoritmos para decidirlo!).

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.
- Greedy: 1x17 + 7x1 (8 monedas)
- Óptima: 3x8 (3 monedas)

Demostrar que algoritmo greedy da el óptimo para un conjunto S de monedas no es trivial (hay algoritmos para decidirlo!).
Pero sí hay casos especiales.. ej: $1\ c\ c^2\ c^3\ \ldots\ c^n$. Ejercicio: demostrarlo.

Dar Cambio - Otras denominaciones

¿Funciona siempre el algoritmo Greedy? Es fácil ver que no:

- Monedas: 1, 8, 17. $N = 24$.
- Greedy: 1x17 + 7x1 (8 monedas)
- Óptima: 3x8 (3 monedas)

Demostrar que algoritmo greedy da el óptimo para un conjunto S de monedas no es trivial (hay algoritmos para decidirlo!).
Pero sí hay casos especiales.. ej: $1\ c\ c^2\ c^3\ \ldots\ c^n$. Ejercicio: demostrarlo.
En el caso genérico: se resuelve con programación dinámica (es un caso especial del problema de la mochila).

Otros algoritmos Greedy

- Varios problemas de scheduling y optimización: ver práctica 3.
- *Mínimo arbol recubridor (minimum spanning tree - MST)*: algoritmos de Prim & Kruskal
- Distancia en grafos: Algoritmo de Dijkstra
- A veces se usan como aproximación o cota en problemas difíciles (ej: en problema del viajante, MST+acomodamiento da una solución a lo sumo 50% peor).

Otros algoritmos Greedy

- Varios problemas de scheduling y optimización: ver práctica 3.
- *Mínimo arbol recubridor (minimum spanning tree - MST)*: algoritmos de Prim & Kruskal
- Distancia en grafos: Algoritmo de Dijkstra
- A veces se usan como aproximación o cota en problemas difíciles (ej: en problema del viajante, MST+acomodamiento da una solución a lo sumo 50% peor).
- ¿Algún otro...?

DP vs. Greedy

Mochila 0-1 vs. Mochila fraccional.

DP vs. Greedy

Mochila 0-1 vs. Mochila fraccional.

- Subestructura óptima: la solución óptima del problema contiene la solución óptima de subproblemas.

DP vs. Greedy

Mochila 0-1 vs. Mochila fraccional.

- Subestructura óptima: la solución óptima del problema contiene la solución óptima de subproblemas.
- Subproblemas solapados: si los subproblemas se solapan (son dependientes entre sí) se debe usar dp.
  Si por el contrario son independientes, alcanza con la elección greedy.