Algorítmica

Tema 3: Algoritmos voraces

Grado en Ingeniería Informática – ETSINF

Resultados de aprendizaje

- Saber lo que es un problema de optimización y cómo se puede formalizar.
- Conocer los conceptos básicos de la estrategia algorítmica "algoritmos voraces".
- Estudiar 3 algoritmos voraces que se sabe (y demuestra) que proporcionan la solución óptima:
  - El problema de la mochila con fraccionamiento.
  - Selección de actividades.
  - El problema del repostaje.

El material de estos apuntes...

...ha sido extraído del libro de apuntes de algorítmica de:

- Andrés Marzal
- María José Castro
- Pablo Aibar

Introducción

Introducción

- Un problema de **optimización** es uno en el que, además de proporcionar unos requisitos para que una solución sea factible, se proporciona un criterio con el que cuantificar lo buena o mala que es cada solución factible y se busca obtener la *mejor* solución.
- Los algoritmos voraces suelen funcionar muy bien para resolver *algunos* problemas de optimización.
- Los algoritmos voraces trabajan en fases. En cada fase:
  - Toman la mejor decisión local **sin plantearte las consecuencias a largo plazo**. Es una aproximación *miope:*

  - Requieren que la elección de óptimos locales en cada paso dé lugar a un óptimo global.

Introducción: la estrategia voraz

- Muchos problemas de optimización admiten una solución voraz.
- Entre los problemas que no admiten solución óptima voraz, algunos permiten una solución voraz que se aproxima razonablemente a la solución óptima:
  - Si es posible acotar la diferencia entre la solución voraz y la óptima (sea en términos absolutos o en términos relativos) hablamos de algoritmos de **aproximación**.
  - En otro caso, decimos que el algoritmo es **heurístico**.

Mochila con fraccionamiento

Descripción del problema

Tenemos una mochila con capacidad para cargar W unidades de peso y N productos que podemos cargar en ella. Cada producto tiene un peso $w_i \in \mathbb{R}^{\geq 0}$ y un valor $v_i \in \mathbb{R}^{\geq 0}$, para $1 \leq i \leq N$. Podemos fraccionar los productos y guardar en la mochila sólo parte de cada uno. Si r es un real entre 0 y 1, el beneficio que aporta guardar r partes del producto i es $r \cdot v_i$. ¿Cómo cargar la mochila de forma que el beneficio sea máximo sin exceder el límite de carga?

Mochila sin fraccionamiento

Existe una variante de este problema (conocido como *mochila discreta* o *0-1 knapsack problem*) donde **no** es posible fraccionar los objetos.

Formalización

Una solución factible es una lista de valores entre 0 y 1, cada uno de los cuales indica la cantidad de un producto que cargamos en la mochila, tal que no represente una carga superior a la capacidad de la mochila:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in [0,1]^N \mid \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}$$

La función objetivo es:

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \leq i \leq N} x_i v_i$$

Y el criterio de optimización es:

$$\hat{x} = \arg\max_{x \in X} f(x)$$

Ejemplo

- Supongamos una mochila con $W = 20$ kg de capacidad y tres productos ($N = 3$) con pesos $w_1 = 18$ kg, $w_2 = 15$ kg y $w_3 = 10$ kg y valores de $v_1 = 25$ €, $v_2 = 24$ € y $v_3 = 15$ €.
- Una carga factible podría ser $(\frac{1}{2}, \frac{1}{3}, \frac{1}{4})$ con 16.5 kg de peso y un beneficio de 24.25 €.
- Naturalmente, no es la única solución factible: $(1, \frac{2}{15}, 0)$, supone cargar la mochila con 20 kg y obtener un beneficio de 28.20 €.
- No podemos ni enumerar explícitamente el conjunto de soluciones factibles, pues su número es infinito.

Aproximación "mientras quepa"

- Cojo un producto cualquiera y meto todo lo que quepa de él en la mochila.
- Y a por el siguiente producto.

```
def mochila_frac_suboptima(w, v, W):
    B = 0 # beneficio
    for wi,vi in zip(w,v):
        ri = min(1, W/wi)
        W -= ri*wi
        B += ri*vi
    return B

W,v,w = 50,[60, 30, 40, 20, 75],[40, 30, 20, 10, 50]
print('Beneficio:',mochila_frac_suboptima(w,v,W))
```

No da la solución óptima

Considerando los objetos en el orden que nos son dados y metiendo en la mochila la mayor cantidad posible de cada uno de ellos, el resultado obtenido es 70(el objeto 1 y un tercio del objeto 2), pero esta solución **no es la óptima:** es posible obtener un beneficio de 90 cargando totalmente los objetos de índices 2 y 3 más un 40% del último objeto.

Aproximación "mientras quepa" con ordenamiento

Los siguientes criterios de ordenación parecen convenientes:

1. De mayor a menor valor.
2. De menor a mayor peso.
3. De mayor a menor relación valor/peso.

Solamente el último proporciona la solución óptima

```
def coste_unit(wv):
    w,v = wv
    return v/w

def mochila_frac_optima(w, v, W):
    B = 0
    for wi,vi in sorted(zip(w,v), reverse=True, key=coste_unit):
        ri = min(1, W/wi)
        W -= ri*wi
        B += ri*vi
    return B
```

Coste

- La solución óptima de la mochila con fraccionamiento tiene un coste que viene dominado por la ordenación de los objetos por su beneficio unitario. Por tanto, para una entrada con N objetos, el coste del algoritmo es $O(N \log N)$.
- Podemos resaltar que una variante de este problema conocido como *mochila discreta* o *0-1 knapsack problem*) (donde **no** es posible fraccionar los objetos) tiene un coste muchísimo mayor.
- Existe una relación entre ambas variantes: puesto que poder fraccionar un objeto tiene como caso particular dejarlo sin fraccionar, resulta que la mochila con fraccionamiento da una **cota optimista** al problema de la mochila sin fraccionamiento.
- También podemos mencionar que es posible resolver la mochila sin fraccionamiento de manera voraz, pero no se asegura una solución óptima (de hecho, es fácil buscar contraejemplos).

Corrección

- ¿Funciona? ¡Hay que demostrarlo!
- Técnica: reducción a la diferencia.

Reducción a la diferencia

Objetivo: demostrar que toda solución diferente de la que devuelve el algoritmo es igual de buena o peor.

- Las soluciones se suelen expresar como secuencias y se considera una diferente de la que devuelve el algoritmo, pero que comparte con ella un prefijo.
- Entonces se estudia por qué el primer elemento en el que difiere la solución alternativa no puede dar un resultado mejor que la solución voraz.

Corrección: Aplicación de la reducción de la diferencia

- Para simplificar, y sin pérdida de generalidad, supondremos que la lista de productos está ordenada por ratio valor/peso decreciente, o sea, $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \frac{v_N}{w_N}$.
- Sea la solución voraz $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ y sea otra solución cualquiera $(x_1, x_2, \ldots, x_N)$.
- Supongamos que hay un tramo idéntico y sea i el primer índice i en que difieren: $x_i \neq \hat{x}_i$. Está claro que $x_i < \hat{x}_i$, ya que el algoritmo voraz selecciona la mayor cantidad posible del producto i.
- Esto significa que estamos dejando una mayor capacidad de carga libre para el resto de productos.
- Sea $a = (\hat{x}_i - x_i)w_i$ el espacio que dejo libre ¿Cuál es el mayor beneficio que podemos obtener con esa capacidad de carga disponible?

Corrección: Aplicación de la reducción de la diferencia

- ¿Con qué producto (que no sea el i) puedo aprovechar mejor ese "hueco"? Con el $i+1$. En el mejor de los casos puedo poner una cantidad q'_{i+1} tal que $a = q'_{i+1} w_{i+1}$. El beneficio extra será entonces $q'_{i+1} v_{i+1}$ que ha de ser mejor que $(\hat{x}_i - x_i) v_i$, pero ocurre justo lo contrario:

$$\begin{aligned}
\frac{v_i}{w_i} &\geq \frac{v_{i+1}}{w_{i+1}} \Rightarrow \\
a\frac{v_i}{w_i} &\geq a\frac{v_{i+1}}{w_{i+1}} \Rightarrow \\
(\hat{x}_i - x_i) w_i \frac{v_i}{w_i} &\geq q' w_{i+1} \frac{v_{i+1}}{w_{i+1}} \Rightarrow \\
(\hat{x}_i - x_i) v_i &\geq q'_{i+1} v_{i+1}
\end{aligned}$$

Acabamos de demostrar que el beneficio adicional que se podría obtener con esa solución $(x_1, x_2, \ldots, x_N)$ sería $q'_{i+1} v_{i+1}$; pero que ese valor siempre será inferior o igual a $(\hat{x}_i - x_i) v_i$. Así queda demostrado que en cada paso conviene cargar la mayor cantidad posible del producto de mayor ratio valor/peso.

EJERCICIOS

- Demuestra que si los productos se seleccionan por orden de mayor a menor valor, no siempre se encuentra una solución óptima aplicando una estrategia voraz.
- Demuestra que si los productos se seleccionan por orden de menor a mayor peso, no siempre se encuentra una solución óptima aplicando una estrategia voraz.
- Diseña un algoritmo voraz para el problema de la mochila discreta. ¿Funciona correctamente el algoritmo si no podemos fraccionar los objetos, es decir, si sólo podemos incluir o no incluir completamente cada objeto? Demuéstralo.
- ¿Funciona correctamente el algoritmo si puedes poner tanta cantidad como desees de cualquiera de los productos (es decir, x_i puede ser mayor que 1)? ¿Sigue funcionando un algoritmo voraz? ¿Puedes diseñar una versión más eficiente (y sencilla)?

Ejercicios

Para estar seguros de que algo no funciona $\rightarrow$ **CONTRAEJEMPLO**

- Aproximación voraz "mientras quepa", pero seleccionando de mayor a menor valor: NO FUNCIONA. Contraejemplo:
  - Instancia: $W = 50$, $w = [40, 30, 20, 10, 50]$, $v = [60, 30, 40, 20, 75]$
  - Ordenamos de mayor a menor valor: $w' = [50, 40, 20, 30, 10]$, $v' = [75, 60, 40, 30, 20]$
  - Solución: $(1, 0, 0, 0, 0)$, con valor total 75. Pero esta no es la óptima, hay una mejor, por ejemplo: $(0, 1/2, 1, 0, 1)$, con valor 90.
- Aproximación voraz "mientras quepa", pero seleccionando de menor a mayor peso: NO FUNCIONA. Contraejemplo:
  - Instancia: $W = 50$, $w = [40, 30, 20, 10, 50]$, $v = [60, 30, 40, 20, 75]$
  - Ordenamos de menor a mayor peso: $w' = [10, 20, 30, 40, 50]$, $v' = [20, 40, 30, 60, 75]$
  - Solución: $(1, 1, 2/3, 0, 0)$, con valor total 80. Hay una mejor, por ejemplo: $(1, 1, 0, 1/2, 0)$, con valor 90.

Ejercicios

- Diseña un algoritmo voraz para el problema de la mochila discreta. ¿Funciona correctamente el algoritmo si no podemos fraccionar los objetos, es decir, si sólo podemos incluir o no incluir completamente cada objeto? Demuéstralo.

El algoritmo ordenaría por valor unitario e iría seleccionado objetos completos en ese orden hasta que se llenara completamente la mochila o se revisaran todos. La misma instancia nos sirve como contraejemplo:

- Instancia: $W = 50$, $w = [40, 30, 20, 10, 50]$, $v = [60, 30, 40, 20, 75]$
- Ordenamos de mayor a menor valor unitario: $w' = [10, 20, 40, 50, 30]$, $v' = [20, 40, 60, 75, 30]$, $v'/w' = [20/10, 40/20, 60/40, 75/50, 30/30] = [2, 2, 1.5, 1.5, 1]$
- Solución: $(1, 1, 0, 0, 0)$, con valor total 60. Hay una mejor, por ejemplo: $(1, 0, 1, 0, 0)$, con valor 80.

Ejercicios

- ¿Funciona correctamente el algoritmo si puedes poner tanta cantidad como desees de cualquiera de los productos (es decir, x_i puede ser mayor que 1)? ¿Sigue funcionando un algoritmo voraz? ¿Puedes diseñar una versión más eficiente (y sencilla)?

Sí que funcionaría, porque elegiría tantas unidades (o parte fraccional) que cupiesen del mejor producto. Peero se puede diseñar una versión más sencilla y eficiente que no seleccione producto a producto, sino seleccionar directamente W/w_1 unidades del objeto 1, siendo este el de mejor relación valor/peso. La selección de este producto solo requiere un coste $O(N)$.

Selección de actividades

Selección de actividades

Queremos obtener el mayor beneficio posible alquilando una sala para la realización de una serie de actividades:

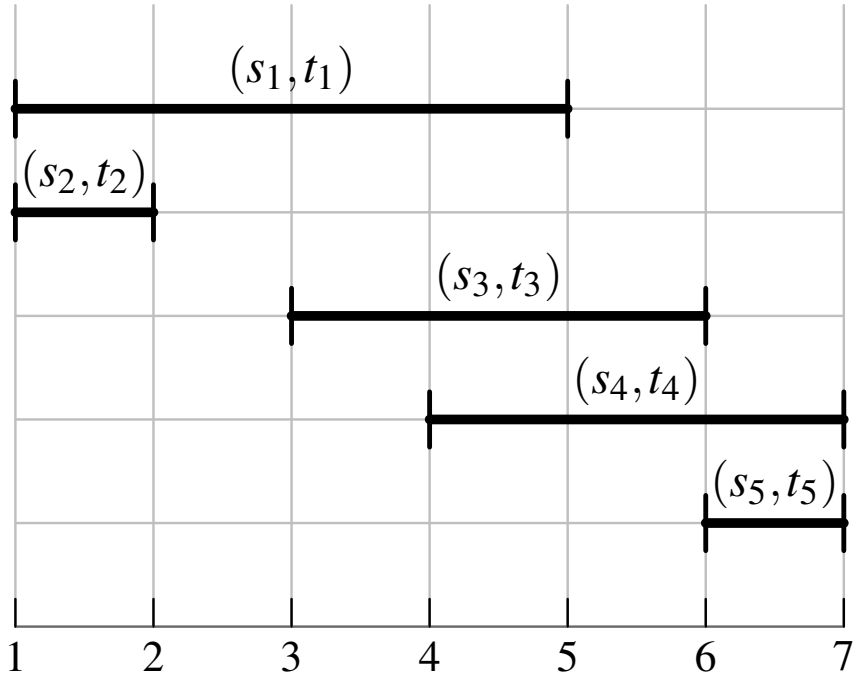
- Cada actividad ocupa un intervalo de tiempo con una hora de inicio s y una hora de terminación t.
- Sólo puede llevarse a cabo una actividad en cada instante, pero podemos empezar una actividad que empiece en el mismo instante en que termina la anterior.
- El beneficio que nos reporta cada actividad **es el mismo**, independientemente de su duración.

Selección de actividades

Dadas unas actividades descritas por sus instantes inicial y final:

$$C = \{(s_1, t_1), (s_2, t_2), \ldots, (s_N, t_N)\}$$

tales que $s_i < t_i$ para $1 \leq i \leq N$, queremos seleccionar el mayor número de actividades sin solapamiento.



Formalización

- El conjunto de soluciones factibles X está formado por los subconjuntos de C tales que si (s_i, t_i) y (s_j, t_j) son elementos del mismo subconjunto, entonces **no se solapan:**

$$s_i < t_i \leq s_j < t_j \quad \text{o bien} \quad s_j < t_j \leq s_i < t_i$$

- La función objetivo, cuyo valor queremos maximizar, es la cardinalidad del conjunto que suministramos como argumento:

$$f(x) = |x|$$

- Queremos conocer, por tanto, $\hat{x} = \arg\max_{x \in X} |x|$.

Solución voraz

¿En qué orden seleccionar las actividades?

1. De mayor a menor duración.
2. De menor a mayor duración.
3. De menor a mayor instante de inicio.
4. De menor a mayor instante de finalización.

La única estrategia que proporciona la solución óptima es la 4ª

```
def seleccion_actividades(C):
    x,t_prev = set(), min(s for (s,t) in C)
    for s,t in sorted(C, key=lambda x:x[1]):
        if t_prev <= s:
            x.add((s,t))
            t_prev = t
    return x
```

- No hace falta comprobar que `(s,t)` solapa con cualquier otra actividad ya seleccionada, basta con comprobar si solapa con la *última* que hemos seleccionado.
- Coste $O(N \log N)$

Corrección

Estrategia de demostración basada en:

Encontrar una subestructura óptima:

1. Considerar qué conviene hacer ante la primera decisión que hemos de tomar.
2. Resolver un problema de naturaleza similar pero de menor talla.
3. Demostrar que la solución óptima del subproblema, tras tomar la decisión voraz, es mejor o igual que si hubiésemos tomado una decisión distinta a la voraz.
4. Demostrar que la solución óptima del subproblema, combinada con la decisión local, conduce a una solución óptima globalmente.

Corrección

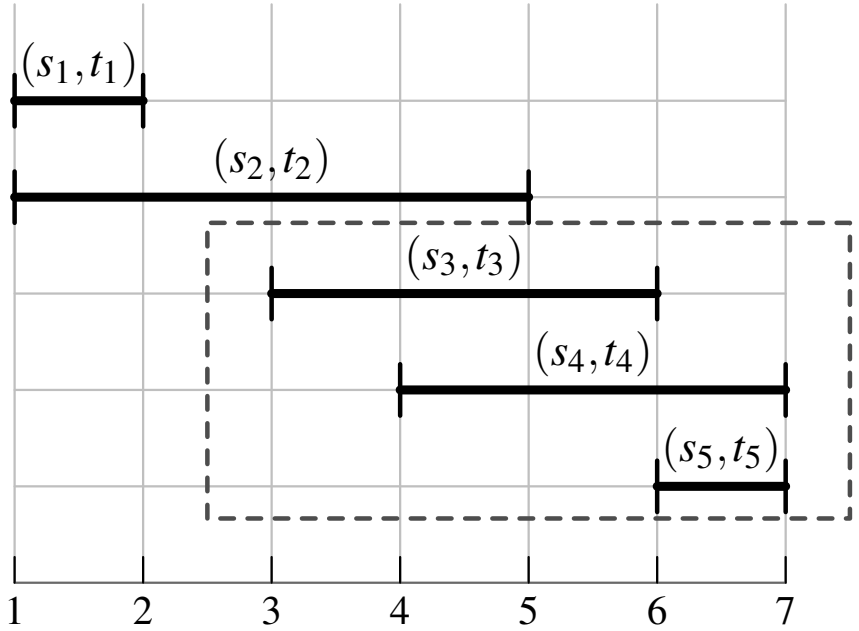
Suponemos las tareas están ordenadas tal que $t_i \leq t_{i+1}$, $\forall 1 \leq i \leq N$.
¿Seleccionamos (s_1, t_1)? Consideremos 2 posibilidades:

- Si la seleccionamos, generamos un subproblema definido por un nuevo conjunto de actividades $C' = \{(s_i, t_i) \in C \mid s_i \geq t_1\}$ que queremos resolver maximizando el nº tareas seleccionadas. Sea m' el mayor nº tareas que podemos seleccionar de C'.
  Seleccionar (s_1, t_1) proporciona una solución con $1 + m'$ tareas.
- Si no seleccionamos la primera tarea, acabaremos seleccionando en primer lugar alguna otra, (s_k, t_k) tal que $t_k > t_1$. Esto nos dejará un subproblema definido por $C'' = \{(s_i, t_i) \in C \mid s_i \geq t_k\}$. Sea m'' el mayor nº tareas que podemos seleccionar de C''.
  Seleccionar (s_k, t_k) proporciona una solución con $1 + m''$ tareas.

Como $C'' \subseteq C'$, necesariamente $m'' \leq m'$.

Corrección

La siguiente figura ilustra el razonamiento con el ejemplo de la explicación visto anteriormente:

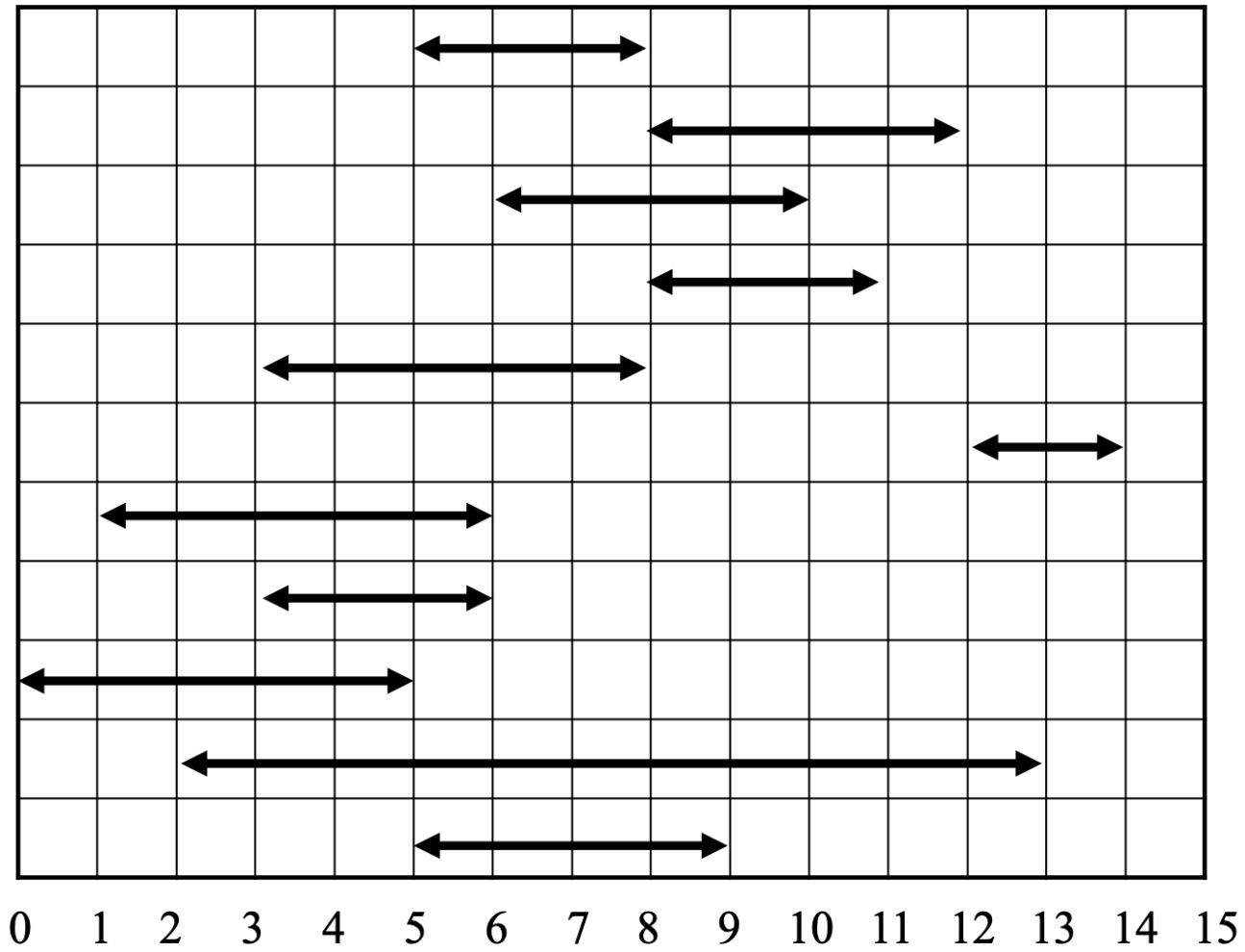


El marco de trazo discontinuo muestra el subproblema que resulta de seleccionar la primera tarea como parte de la solución. El subproblema que resulta de seleccionar cualquier otra en primer lugar es un subconjunto de este subproblema.

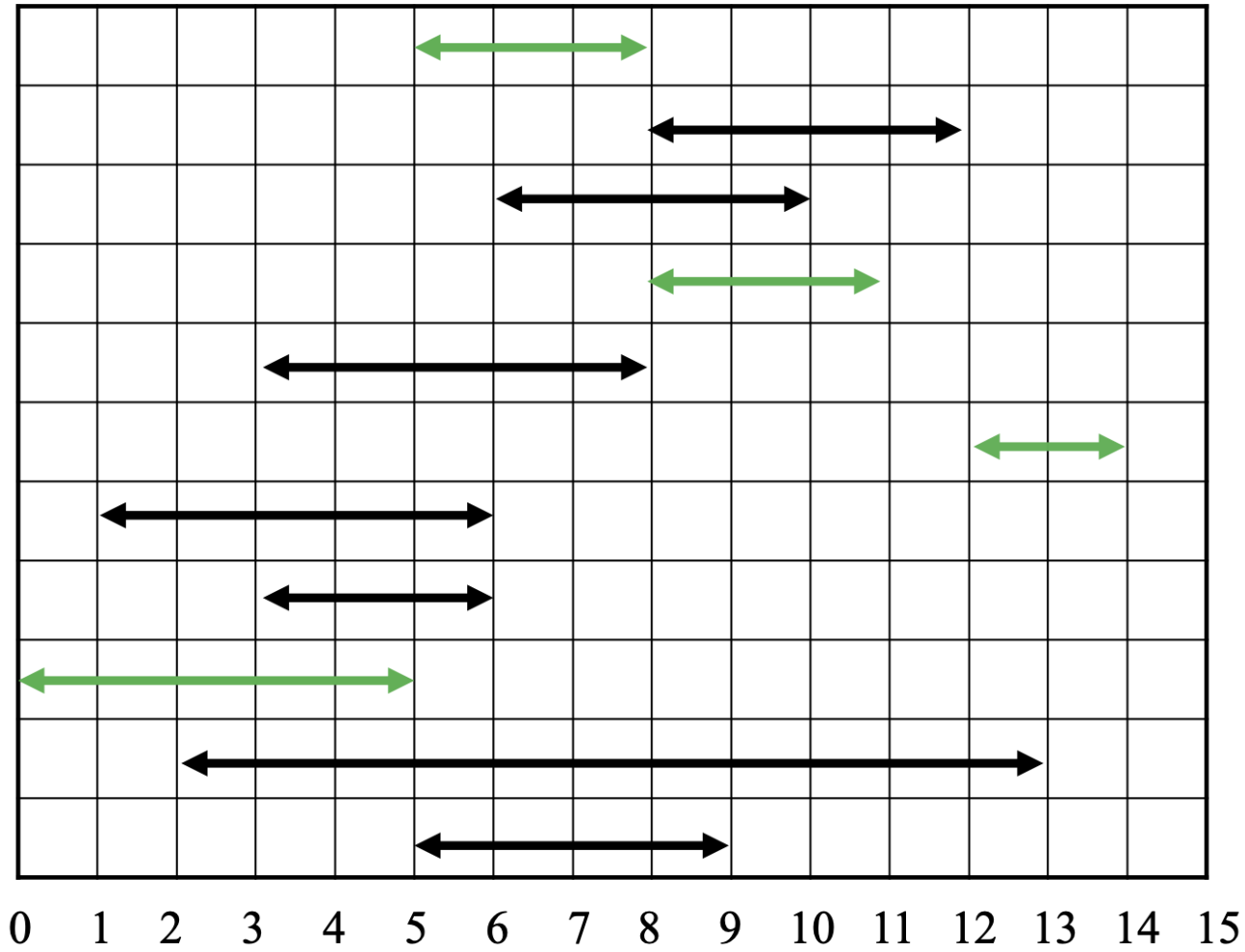
EJERCICIOS

- Demuestra que los otros tres criterios conducen a soluciones no óptimas.

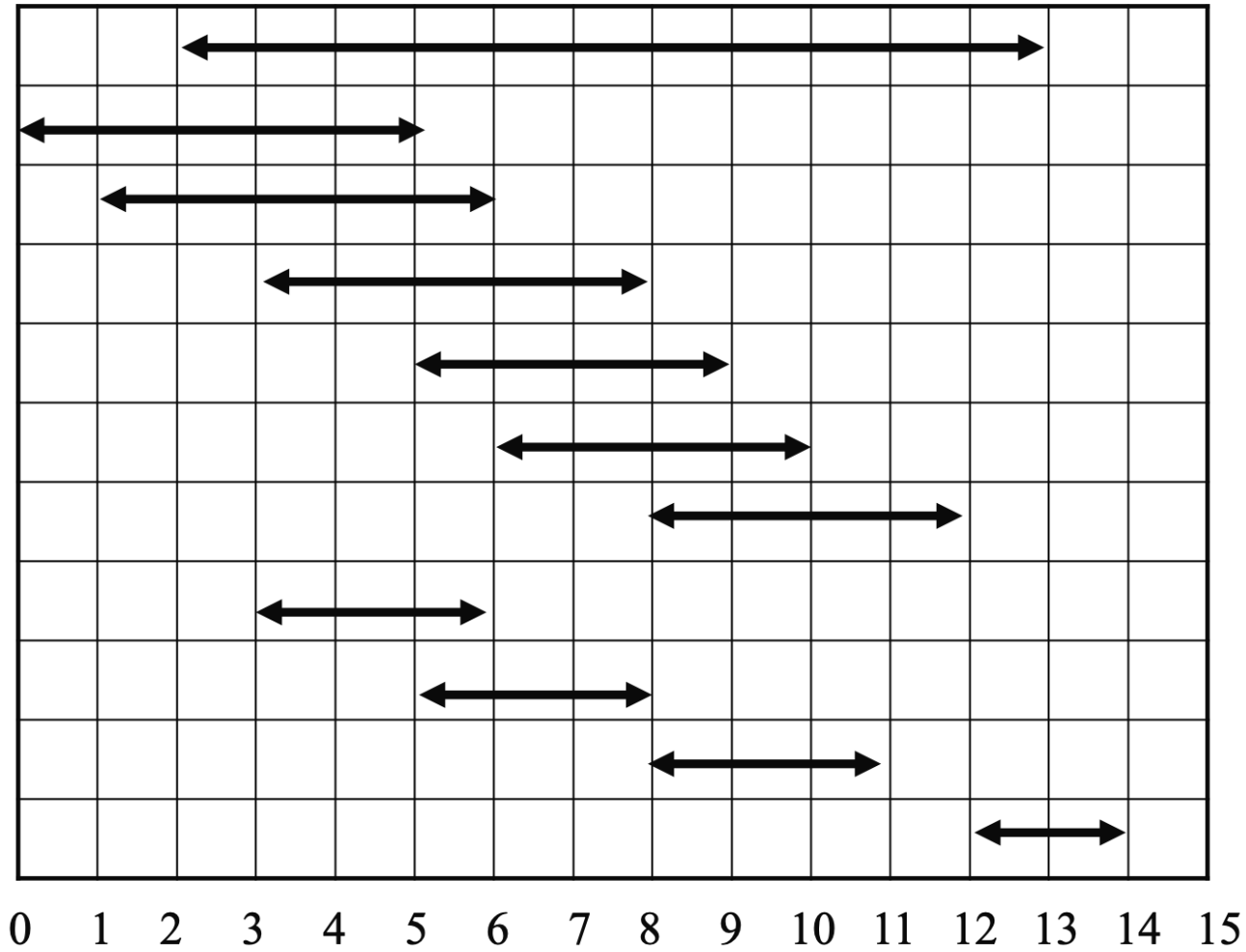
Pongamos una instancia...



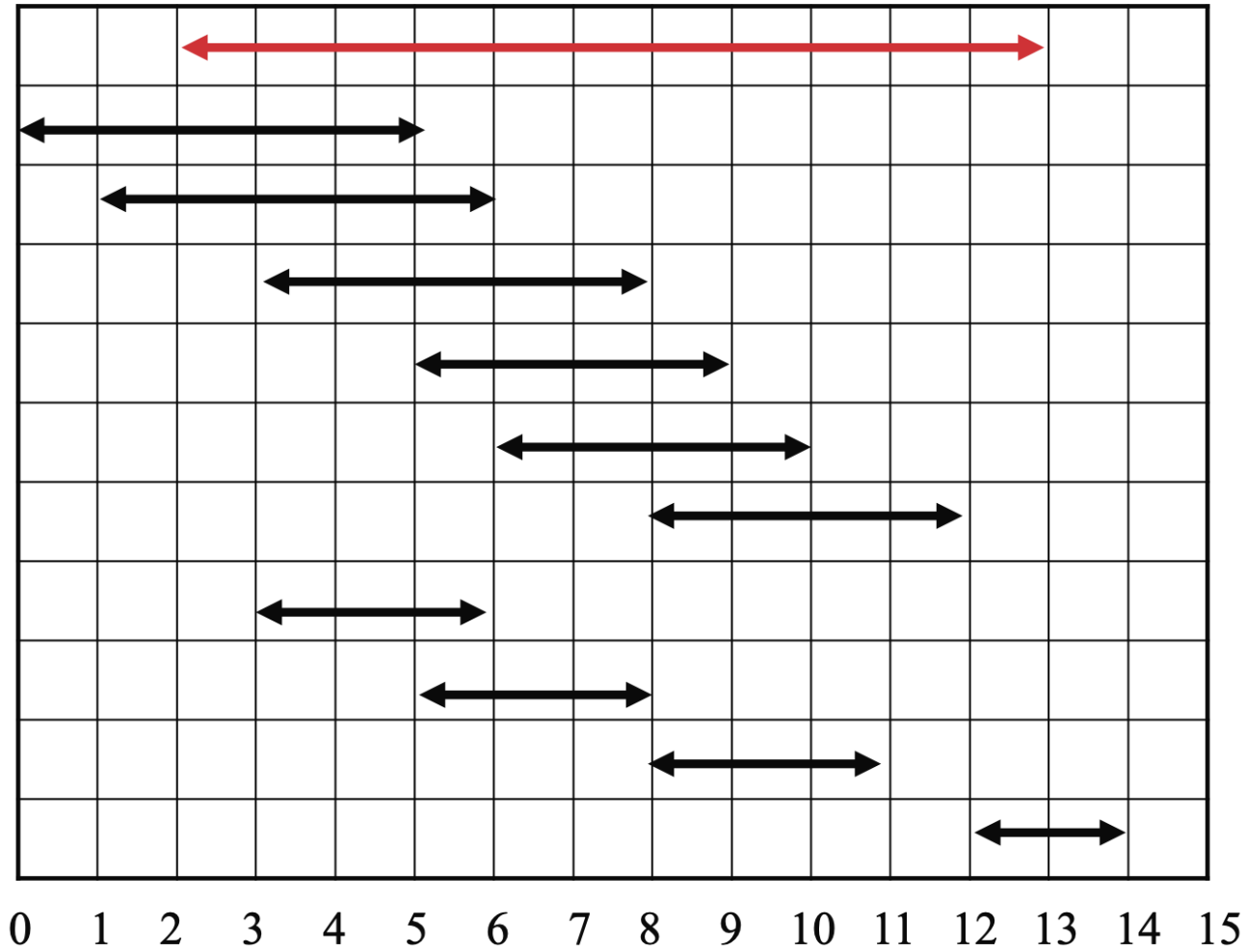
La solución óptima sería



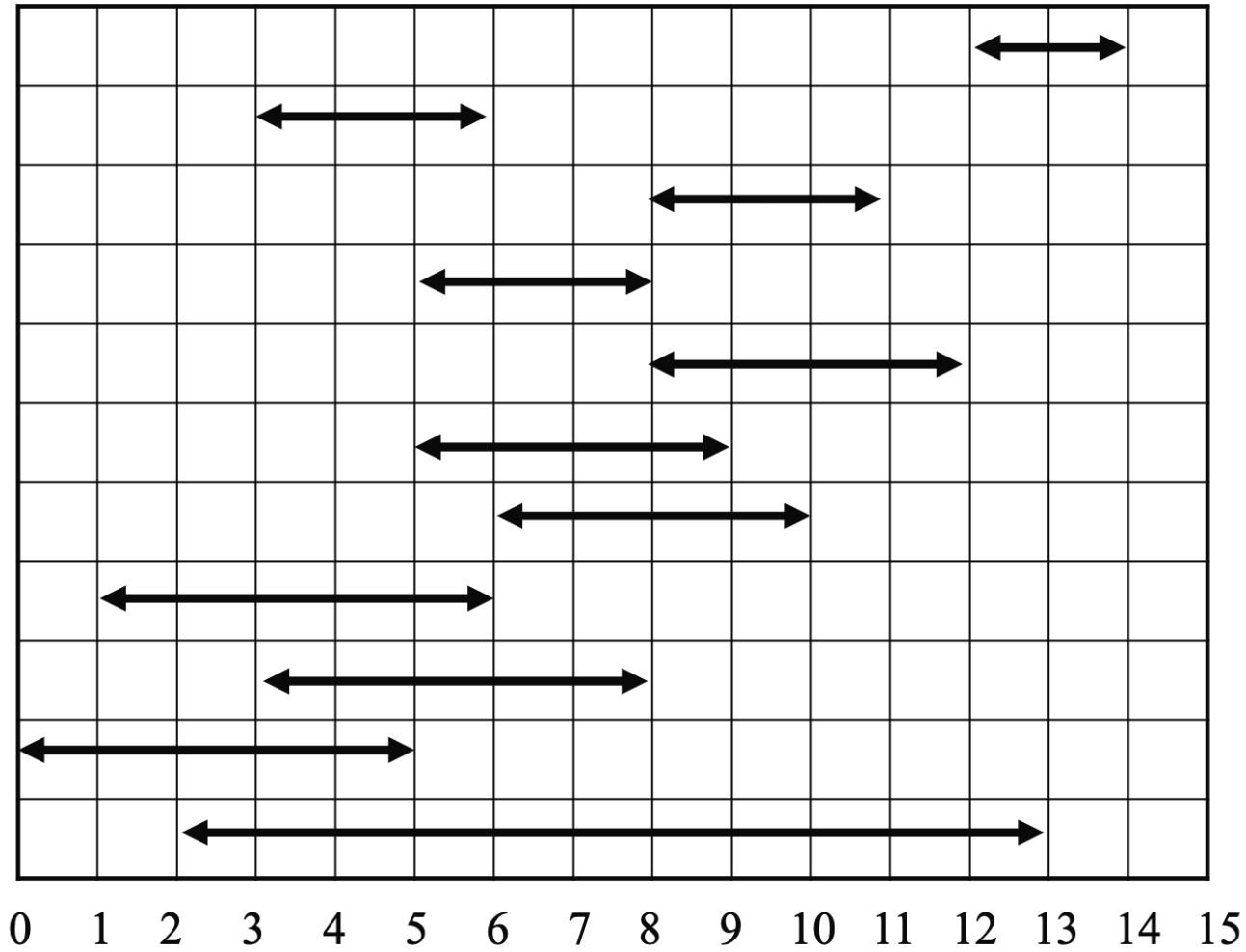
Estrategia: De mayor a menor duración → ordenamos



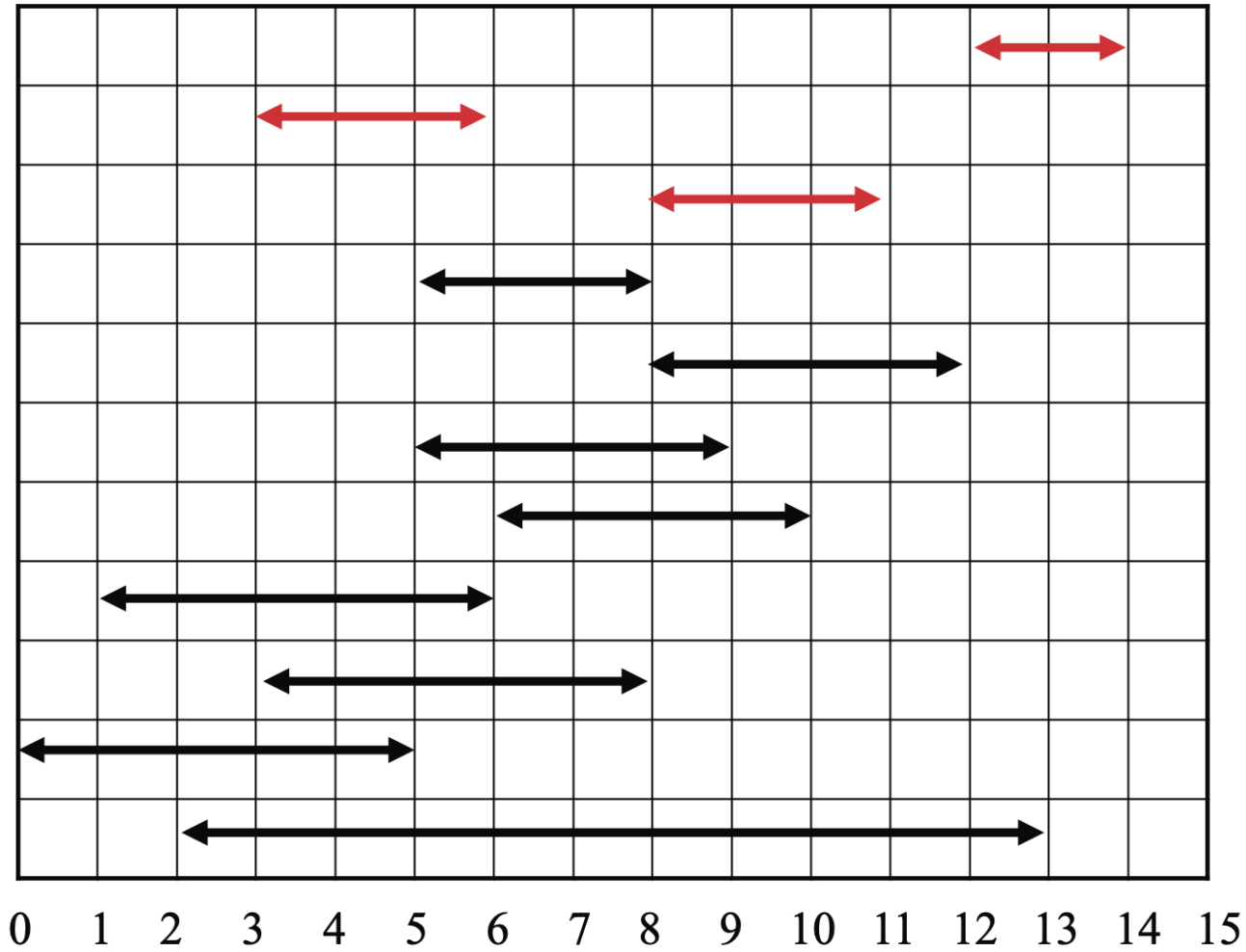
Solución: De mayor a menor duración



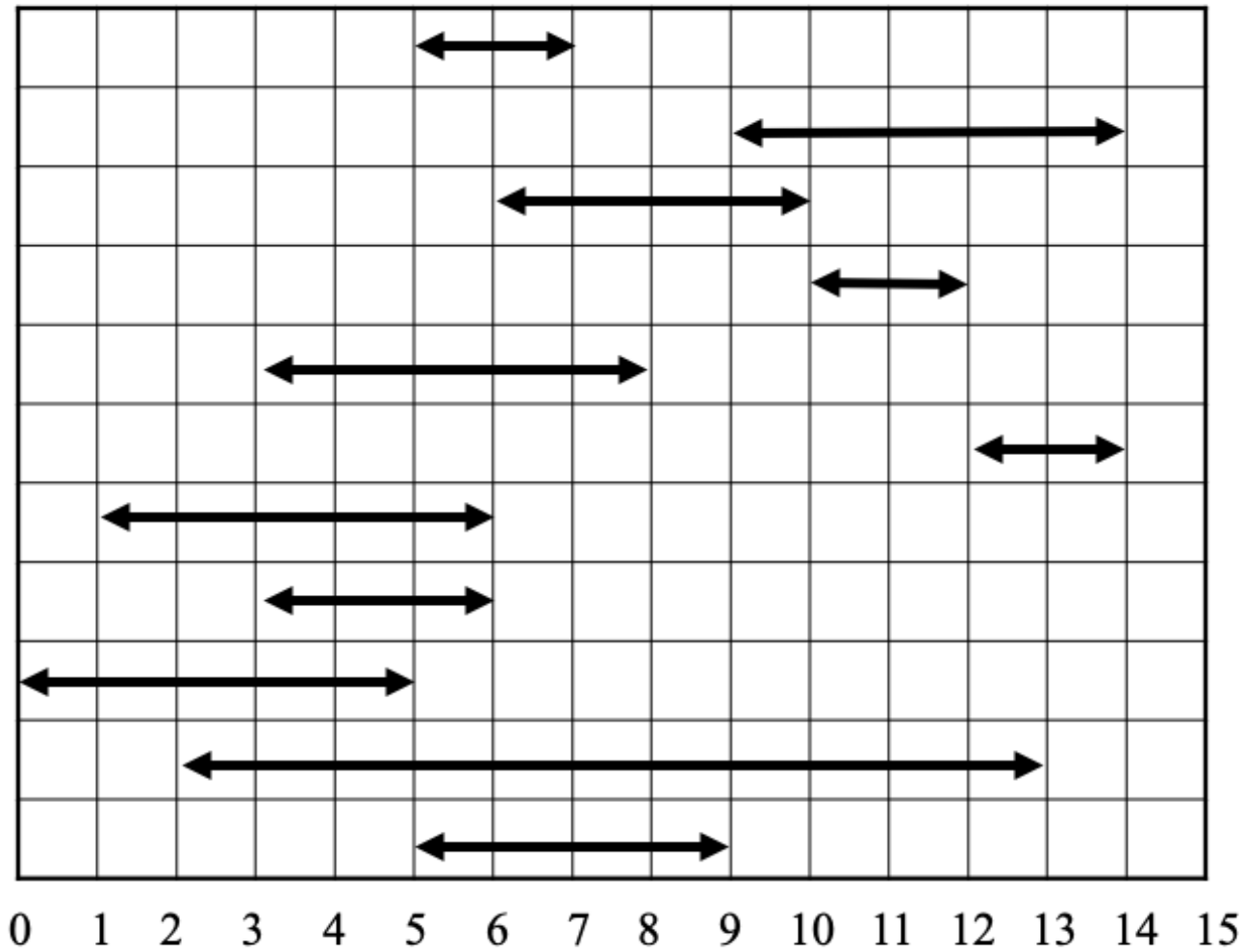
Estrategia: De menor a mayor duración → ordenamos



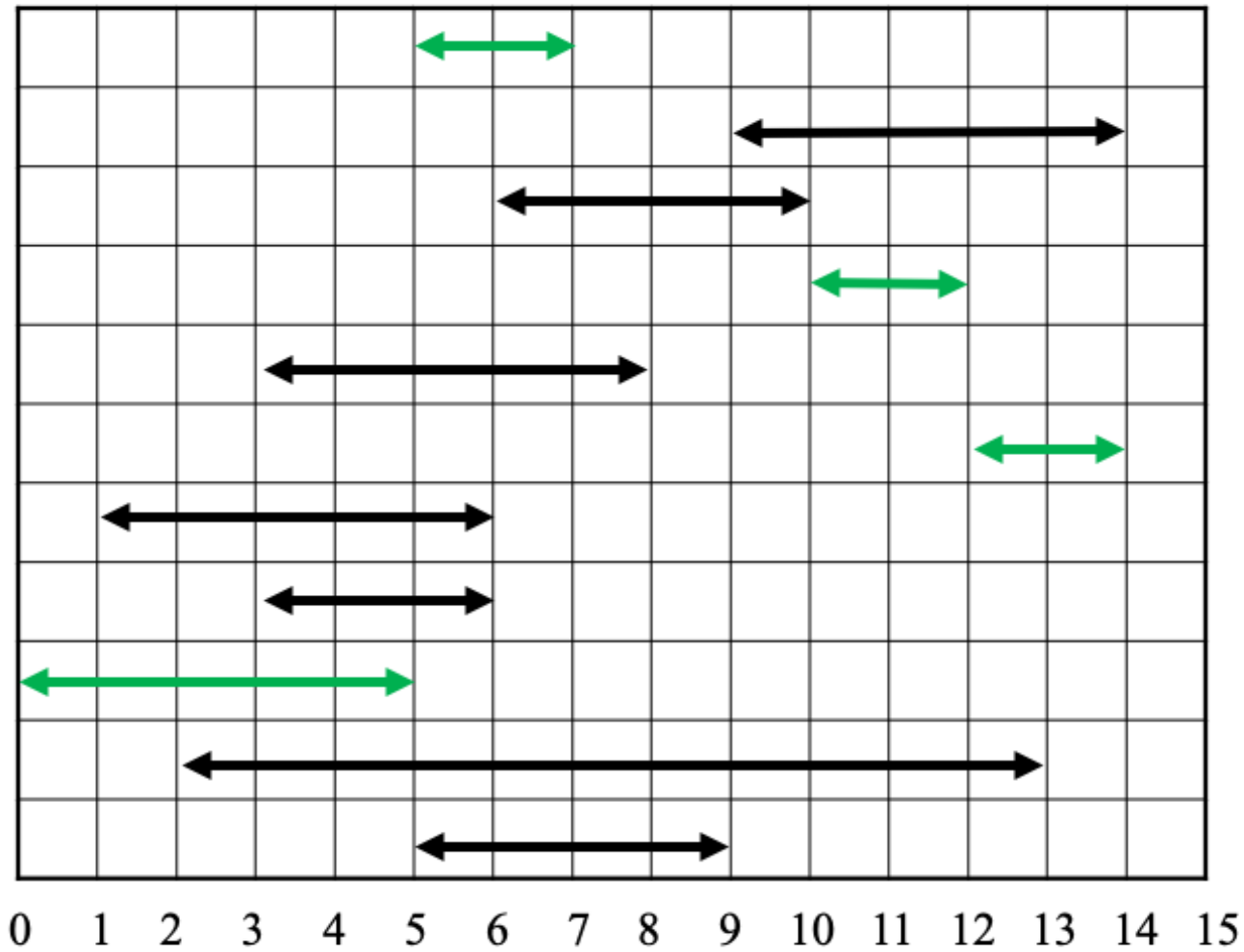
Solución: De menor a mayor duración



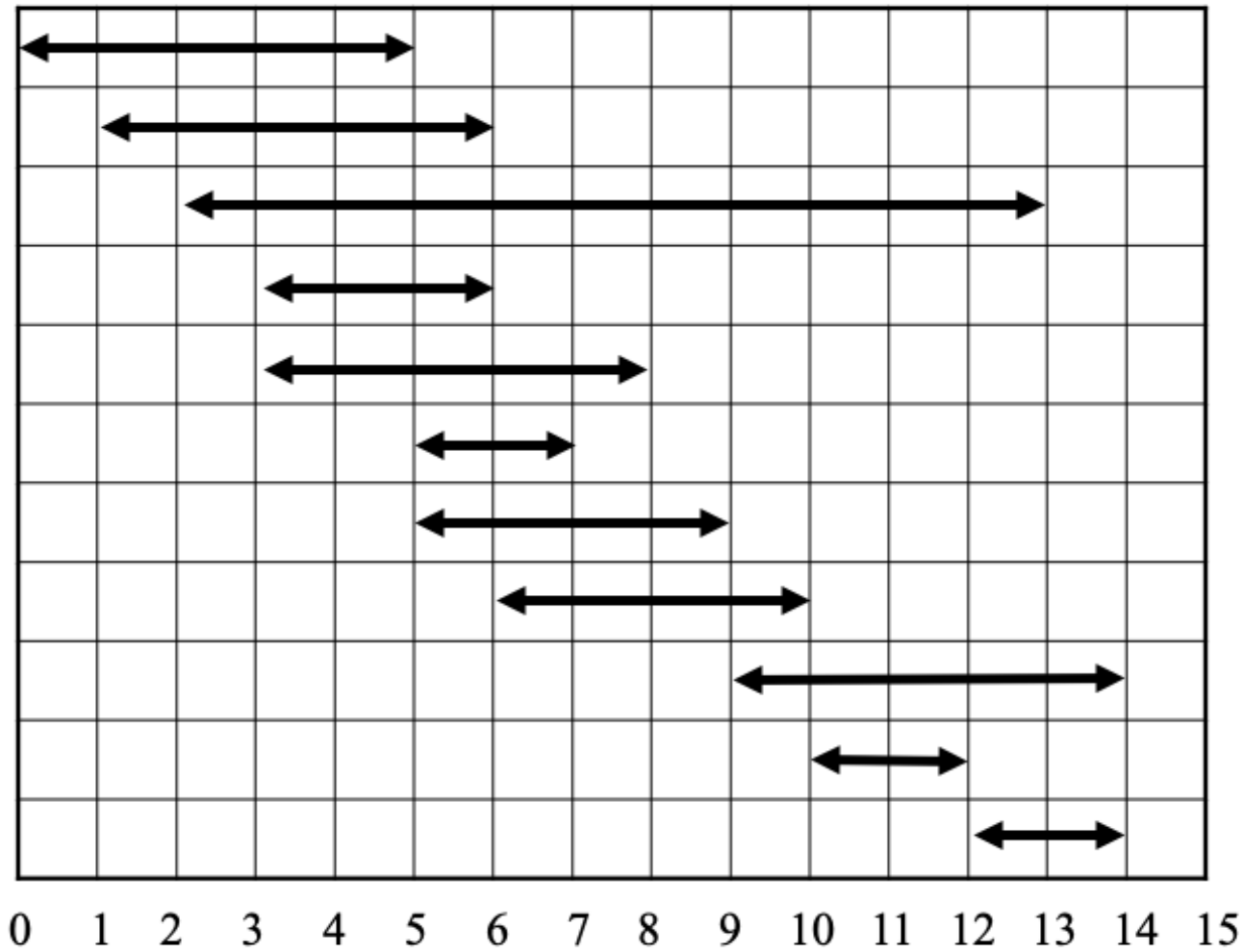
Pongamos otra instancia...



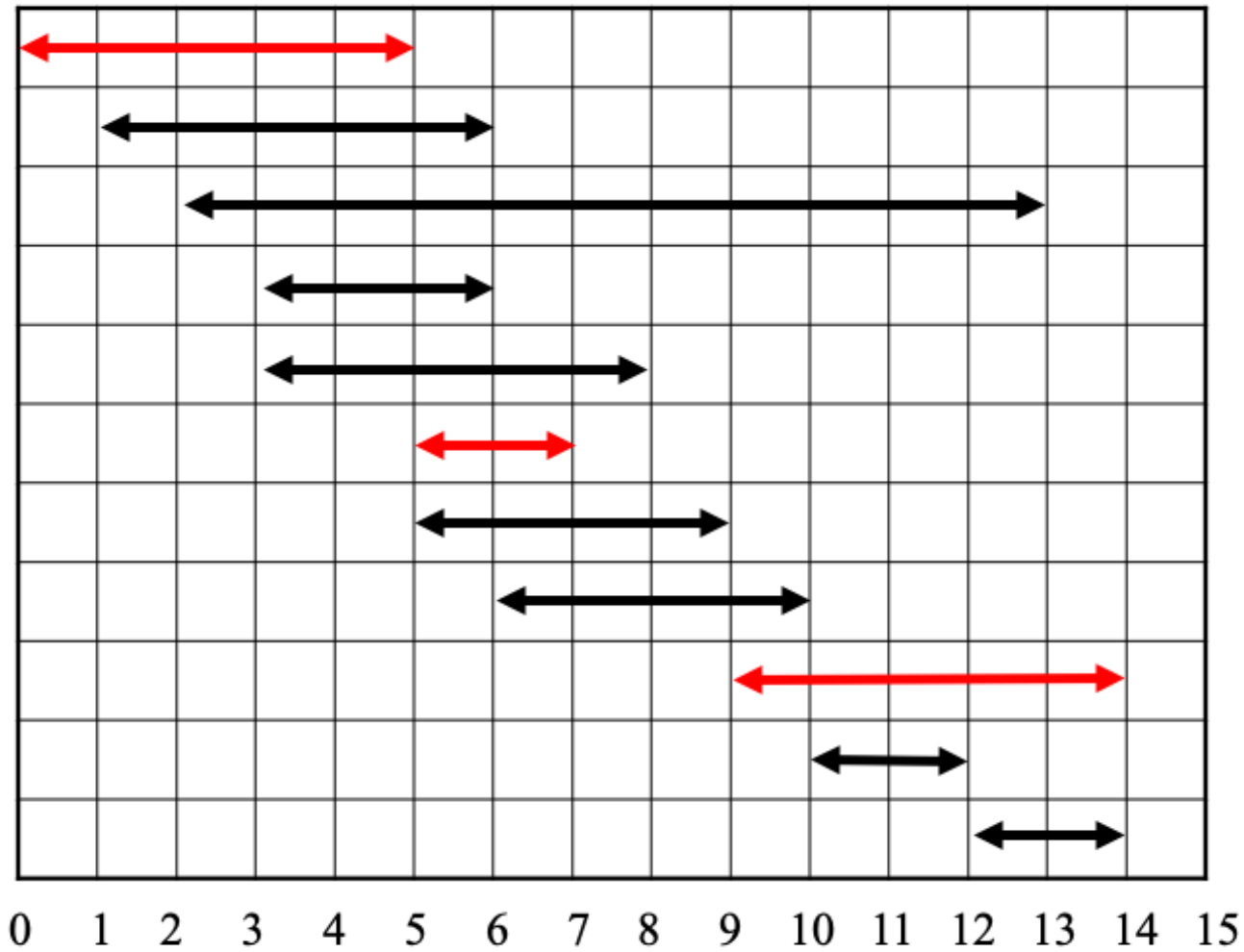
La solución óptima sería



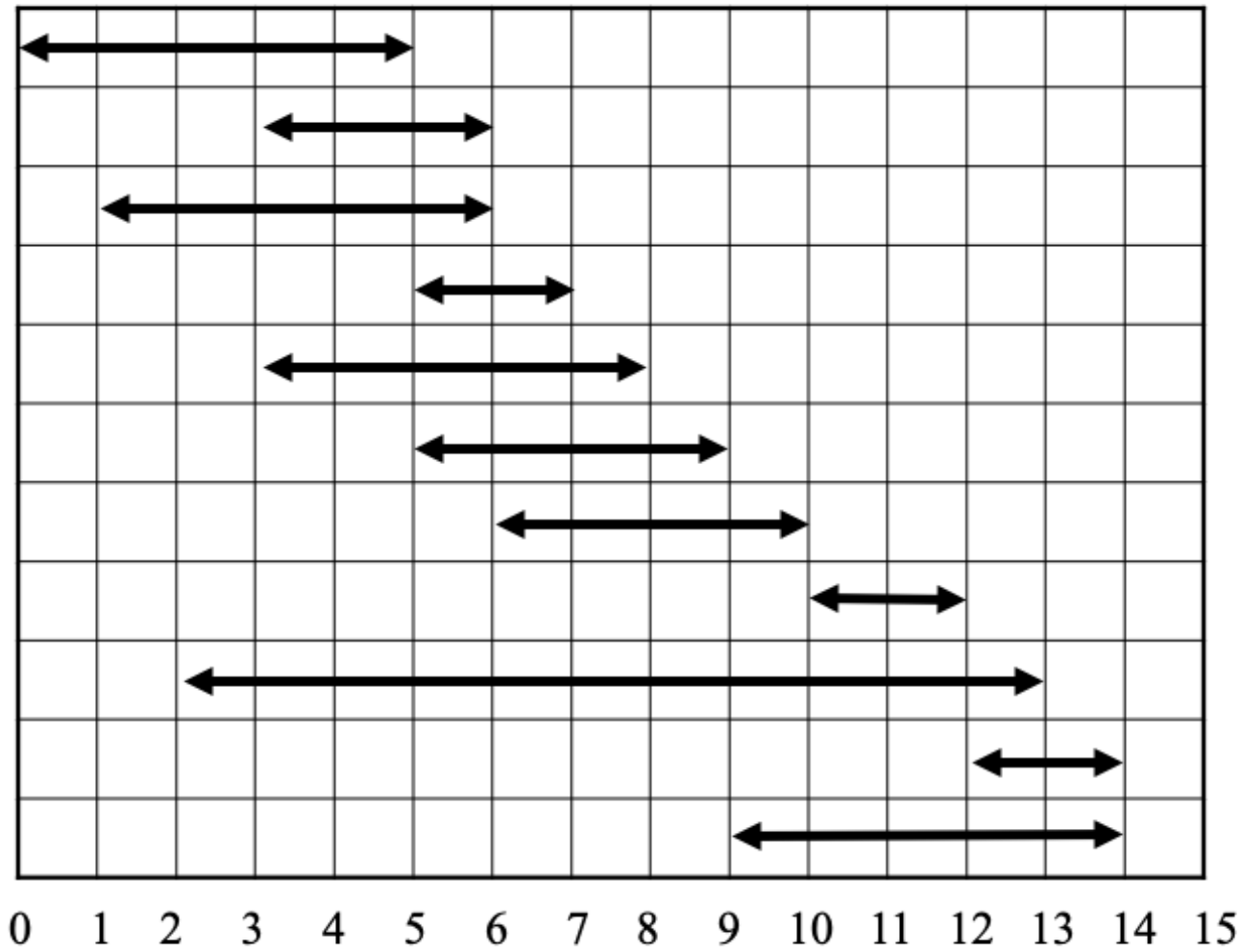
Estrategia: De menor a mayor instante de inicio → ordenamos



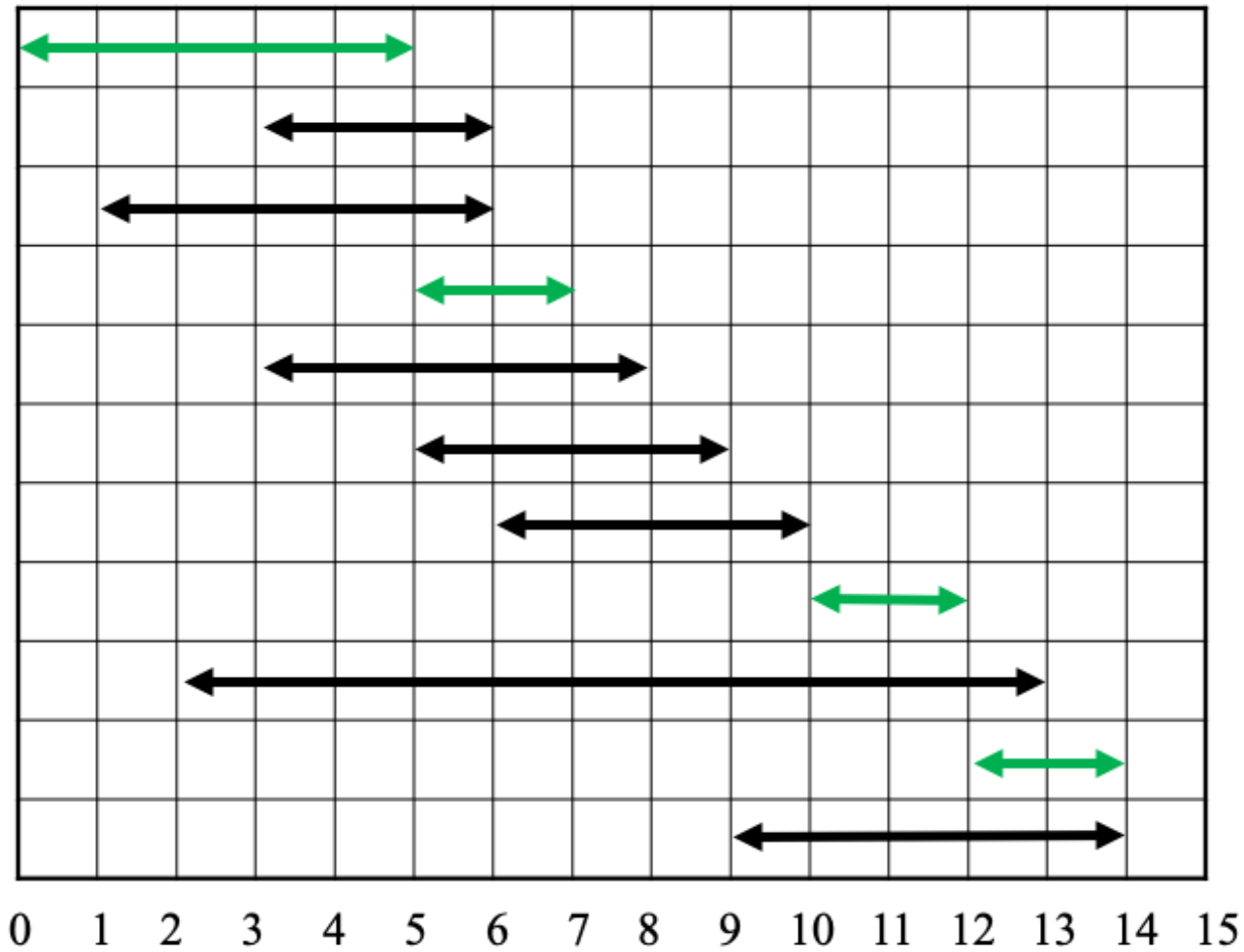
Solución: De menor a mayor instante de inicio



Estrategia: De menor a mayor instante de finalización → ordenamos



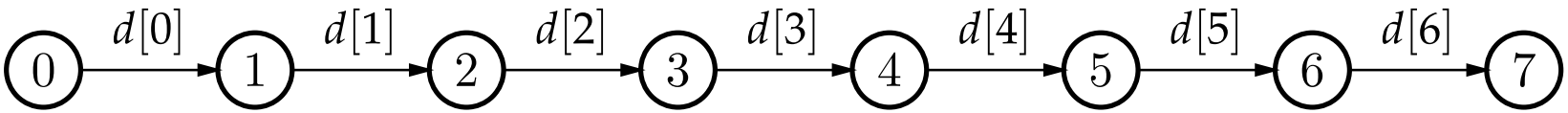
Solución: De menor a mayor instante de finalización ¡¡¡¡ÓPTIMO!!!



El repostaje

El repostaje

- Un camión puede hacer n kms con el depósito lleno.
- Queremos hacer un viaje y conocemos la distancia entre cada gasolinera y la siguiente (todas a menos de n kms entre sí).



Representación del grafo asociado a una ruta con 6 estaciones de servicio.

- El objetivo es parar el menor número de veces a repostar.

Estrategia voraz y algoritmo

Estrategia voraz: recorrer el mayor número posible de kms sin repostar.

```
def gas_stations(M, d, n):
    stop = [0]
    km = n
    for i in xrange(M+1):
        if d[i] >= km:
           stop.append(i)
           km = n
        km -= d[i]
    stop.append(M+1)
    return stop

from gas import gas_stations

print gas_stations (6, [65, 23, 45, 62, 12, 56, 26], 150)

[0, 3, 6, 7]
```

Como es habitual con la estrategia voraz, hemos diseñado un algoritmo muy rápido: lineal con el número de gasolineras, es decir, $\Theta(M)$.

Corrección: Reducción a la diferencia

- Supongamos que la ruta óptima es $(x_1, x_2, \ldots, x_m)$ y que tenemos una ruta alternativa, $(x'_1, x'_2, \ldots, x'_{m'})$ tal que coincide con la óptima hasta x_k. ¿Puede ser mejor que la óptima?
- x'_{k+1} ha de ser menor que x_{k+1}, pues x_{k+1} es la estación más lejana a la que se puede ir con el depósito lleno.
- ¿Y qué pasará con x_{k+2} y el resto de paradas? Que $x_{k+2} > x'_{k+2}$ Y $x_{k+3} > x'_{k+3}$ ...
- Al llegar al destino, la otra ruta consume al menos una parada más que la voraz.

Otros problemas

Otros problemas

- El *algoritmo de Kruskal* es de tipo voraz.
- Existen otros problemas que, aunque no admiten una solución voraz óptima, sí la admiten como aproximación o como heurístico:
  - *Cambio de monedas*: dado un sistema monetario (los tipos de monedas válidas) y dada una cantidad a devolver, el objetivo es representar dicha cantidad usando el menor número de monedas. Hay un algoritmo voraz pero a veces no encuentra la solución o encuentra una que no es óptima. ← **Algoritmo heurístico o una heurística**
  - *Bin packing:* Dado un conjunto de objetos y unas cajas que permiten guardar una cantidad determinada (todas con la misma capacidad), el objetivo consiste en guardar todos los objetos utilizando el menor número de cajas. No hay solución voraz óptima, pero sí una solución que aproxima a la óptima y es posible acotar la diferencia. ← **Algoritmo de aproximación**

Ejercicios

Todos los años se procede a leer el Quijote en el Día del Libro en el Círculo de Bellas Artes de Madrid. Se han presentado N candidaturas de personas que desean leer una parte, pero son muy exigentes: cada candidato i está dispuesto a leer únicamente si le asignan los párrafos desde p_i hasta f_i. Debes realizar una selección intentando contentar al mayor número de candidatos, sabiendo que las partes del libro no seleccionadas por estos podrán ser leídas sin problema por otros voluntarios.
Se pide realizar una función Python que reciba la lista de las N candidaturas: [C={(p_1,f_1), (p_2,f_2), . . . , (p_N,f_N)}] y que devuelva una lista de las candidaturas seleccionadas. Sigue una estrategia voraz e indica el coste del algoritmo desarrollado y si resuelve o no este problema de manera óptima.

```
C = [(23,40), (12,50), (4,8), (10,12), (20,25)]
print("Los candidatos seleccionados leerán los siguientes pasajes:")
print(seleccionar(C))
```

Ejercicios

Solución: Claramente este problema se corresponde al problema visto en voraces llamado "selección de actividades". Para este problema se conoce una estrategia voraz óptima consistente en elegir las actividades que no solapen ordenadas *por instante de finalización* (elegir primero las que terminen antes).

```
def actselection(C):
  x  = set()
  if len(C)>0:
    t2 = min(s for (s,t) in C)
    for (s,t) in sorted(C,key=lambda (s,t): t):
      if t2 <= s:
        x.add( (s,t) )
        t2 = t
  return x
```

Bibliografía

Bibliografía

- *Data Structures and Algorithms in Python*, de Goodrich M, Tamassia R. y Goldwasser M. Sección 13.4.
- *Algorithms*, de Jeff Erikson. Capítulo 4. Disponible en estos enlaces:
  - Todo el libro
  - Capítulo 4
- Capítulo *"Algoritmos voraces"* del libro *Curso de algoritmia* de Andrés Marzal, María José Castro y Pablo Aibar.