Algoritmos Greedy

Pedro Ribeiro

DCC/FCUP

2016/2017

Algoritmos Greedy

- Vamos falar de algoritmos **greedy**.
  Em português são conhecidos como:
  - Algoritmos **ávidos**
  - Algoritmos **gananciosos**
  - Algoritmos **gulosos**

Algoritmo Greedy

- Em cada passo escolher a **"melhor"** próxima escolha
- Nunca olhar "para trás" ou mudar decisões tomadas
- Nunca olhar "para a frente" para verificar se a nossa decisão tem consequências negativas.

O problema do troco (problema do cashier)

Input: Um conjunto de valores de moedas S e uma quantia K a criar com as moedas

Output: O menor número de moedas que fazem a quantia K (podemos repetir moedas)

Exemplo de Input/Output

Input: $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$
(temos infinitas moedas de cada tipo)
$K = 42$

Output: 3 moedas $(20 + 20 + 2)$

O Problema do Troco

Um algoritmo greedy

Em cada passo escolher a maior moeda que não faz passar da quantia k

Exemplos (com $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$):

- $K = \mathbf{35}$
  - **20** (total: 20) + **10** (total: 30) + **5** (total: 35) [3 moedas]
- $K = \mathbf{38}$
  - 20 + 10 + 5 + 2 + 1 [5 moedas]
- $K = \mathbf{144}$
  - 100 + 20 + 20 + 2 + 2 [5 moedas]
- $K = \mathbf{211}$
  - 200 + 10 + 1 [3 moedas]

O Problema do Troco

- Este algoritmo resulta sempre no **mínimo** número de moedas?

- Para os sistemas de moedas comuns (ex: euro, dólar)... sim!
- Para um sistema de moedas qualquer... não!

Exemplos:

- $S = \{1, 2, 5, 10, 20, 25\}$, $K = 40$
  - Greedy dá 3 moedas $(25 + 10 + 5)$, mas é possível 2 moedas $(20 + 20)$
- $S = \{1, 5, 8, 10\}$, $K = 13$
  - Greedy dá 4 moedas $(10 + 1 + 1 + 1)$, mas é possível 2 moedas $(5 + 8)$

  (Será que basta que uma moeda seja $\geq$ que o dobro da anterior?)
- $S = \{1, 10, 25\}$, $K = 40$
  - Greedy dá 7 moedas $(25 + 10 + 1 + 1 + 1 + 1 + 1)$, mas é possível 4 moedas $(10 + 10 + 10 + 10)$

Algoritmos Greedy

- Ideia **"simples"**, mas nem sempre funciona
  - Dependendo do problema, pode dar resposta não ótima
- Normalmente a **complexidade temporal é baixa** (ex: linear ou linearítmica)
- Um contra-exemplo prova que um greedy está errado...
- ...o **difícil é provar** a otimalidade!
- Tipicamente é aplicado em **problemas de optimização**
  - Encontrar a "melhor" solução entre todas as soluções possíveis, segundo um determinado critério (função objectivo)
  - Geralmente descobrir um máximo ou ou mínimo
- Uma passo de pré-processamento muito comum é... **ordenar**!

Propriedades para um algoritmos greedy funcionar

Subestrutura Ótima

Quando a solução óptima de um problema contém nela própria soluções óptimas para subproblemas do mesmo tipo

Exemplo

Seja $min(k)$ o menor número de moedas para fazer a quantia k. Se essa solução usar uma moeda de valor v, então o resto das moedas a usar é precisamente $min(k - v)$.

- Se um problema apresenta esta característica, diz-se que respeita o **princípio da optimalidade**.

Propriedades para um algoritmos greedy funcionar

Propriedade da Escolha Greedy

Uma solução ótima é consistente com a escolha greedy que o algoritmo faz.

Exemplo

No caso das moedas de euro, não existe nenhuma solução ótima que não use a maior moeda menor ou igual à quantia a fazer.

- **Provar** esta propriedade é o mais complicado

Fractional Knapsack

Problema da Mochila Fracionada (fractional knapsack)

Input: Uma mochila com capacidade C
Um conjunto de n materiais, cada um com peso w_i e valor v_i

Output: A alocação de materiais para a mochila que maximize o valor transportado.

Os materiais podem ser "partidos" em pedaços mais pequenos, ou seja, podemos decidir levar apenas quantidade x_i do objecto i, com $0 \leq x_i \leq 1$.

O que queremos é portanto respeitar o seguinte:

- Os materiais cabem na mochila $(\sum_i x_i w_i \leq C)$
- O valor da mochila é o maior possível (maximizar $\sum_i x_i v_i$)

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material de maior valor:

i	1	2	3	4	5
x_i	0	0	1	0.5	1

Isto daria um peso total de 100 e um valor total de 146.

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material mais leve:

i	1	2	3	4	5
x_i	1	1	1	1	0

Isto daria um peso total de 100 e um valor total de 156.

Fractional Knapsack

Exemplo de Input

Input: 5 objectos e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Qual é a resposta ótima neste caso?

- Escolher sempre o material com maior rácio *valor*/*peso*:

i	1	2	3	4	5
v_i/w_i	2	1.5	2.2	1.0	1.2
x_i	1	1	1	0	0.8

Isto daria um peso total de 100 e um valor total de 164.

Fractional Knapsack

Teorema

Escolher sempre a maior quantidade possível do material com maior rácio *valor*/*peso* é uma estratégia que dá valor ótimo

1) Subestrutura Ótima

Considere uma solução ótima e o seu material m com melhor rácio.

Se o retirmos da mochila, então o restante tem de conter a solução ótima para os outros materiais que não m e para uma mochila de capacidade $C - w_m$.

Caso assim não seja, então a solução inicial também não era ótima!

Fractional Knapsack

Teorema

Escolher sempre a maior quantidade possível do material com maior rácio *valor*/*peso* é uma estratégia que dá valor ótimo

2) Propriedade da Escolha Greedy

Queremos provar que a máxima quantidade possível do material m com maior rácio (v_i/w_i) deve ser incluida na mochila.

O valor da mochila: $valor = \sum_i x_i v_i$.

Seja $q_i = x_i w_i$ a quantidade de material i na mochila: $valor = \sum_i q_i v_i / w_i$

Se ainda temos material m disponível, então substituir um outro qualquer material i por m vai dar um melhor valor total:

$q_i v_m / w_m \geq q_i v_i / w_i$ (por definição de m)

Fractional Knapsack

Algoritmo greedy para Fractional Knapsack

- Ordenar materiais por ordem decrescente de rácio *valor*/*peso*
- Processar o próximo material na lista ordenada:
  - Se o elemento couber na totalidade na mochila, incluir todo e continuar para o próximo material
  - Se o elemento não couber na totalidade na mochila, incluir o máximo possível e terminar

Complexidade:

- Ordenar: $O(n \log n)$
- Processar: $O(n)$
- Total: $O(n \log n)$

Interval Scheduling

Problema do Planeamento de Intervalos (interval scheduling)

Input: Um conjunto de n actividades, cada uma com início no tempo s_i e final no tempo f_i.

Output: Descobrir o maior subconjunto de actividades que não tenham sobreposições

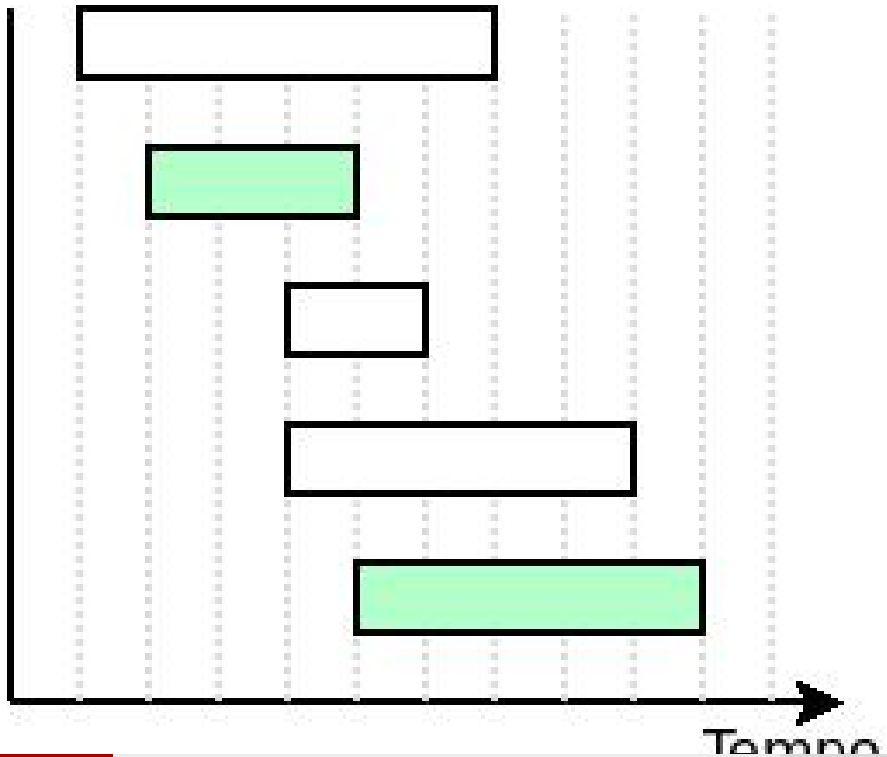
Dois intervalos i e j têm uma sobreposição se existe um tempo k no qual ambos estão activos.

Interval Scheduling

Exemplo de Input

Input: 5 actividades:

i	1	2	3	4	5
s_i	1	2	4	4	5
f_i	7	5	6	9	10



Interval Scheduling

"Padrão" greedy: estabelecer uma ordem segundo um determinado critério e depois ir escolher actividades que não sejam sobrepostas com as já escolhidas

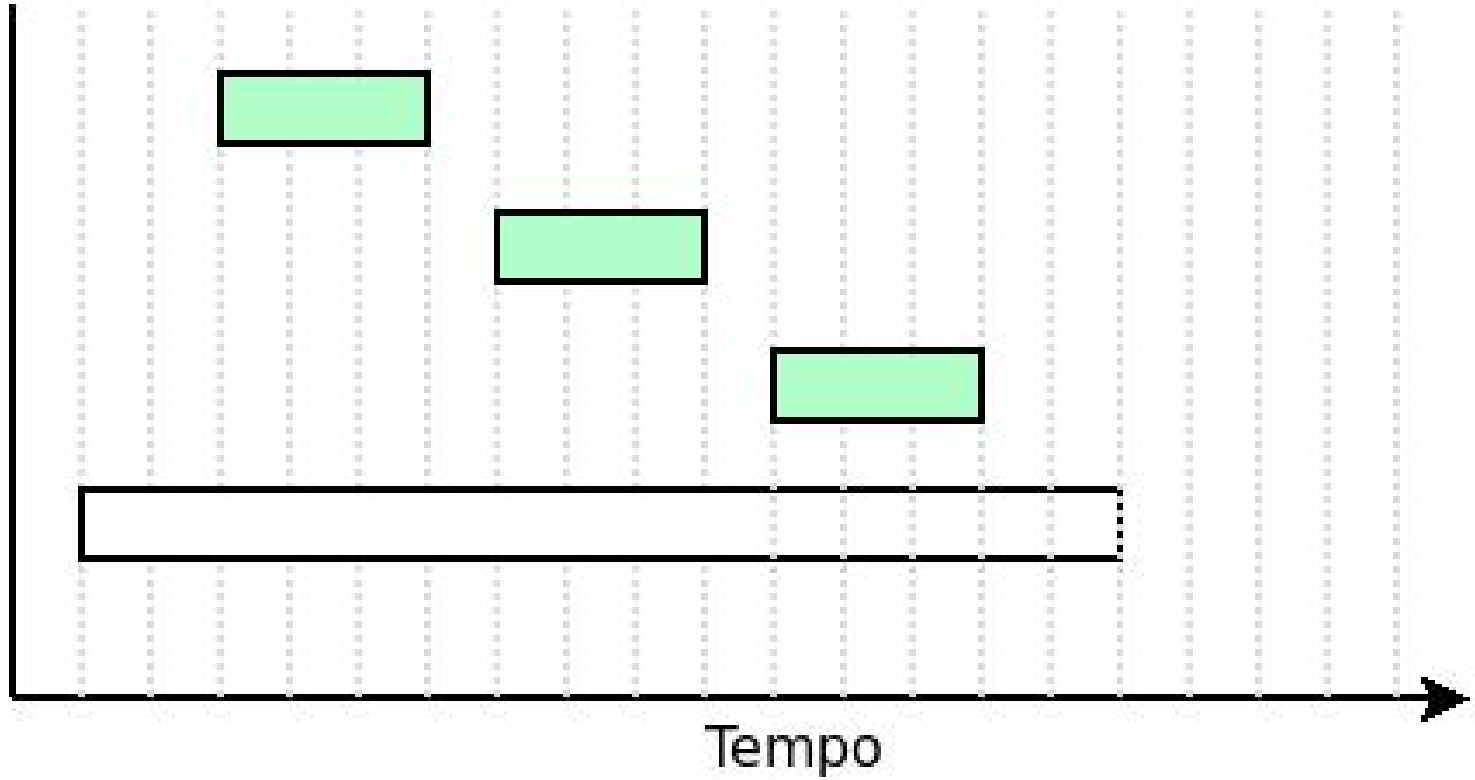
Algumas possíveis ideias:

- [Início mais cedo] Alocar por ordem ascendente de s_i
- [Final mais cedo] Alocar por ordem ascendente de f_i
- [Intervalo mais pequeno] Alocar por ordem ascendente de $f_i - s_i$
- [Menos conflitos] Alocar por ordem ascendente do número de outras actividades que estão sobrepostas

Interval Scheduling

[Início mais cedo] Alocar por ordem ascendente de s_i

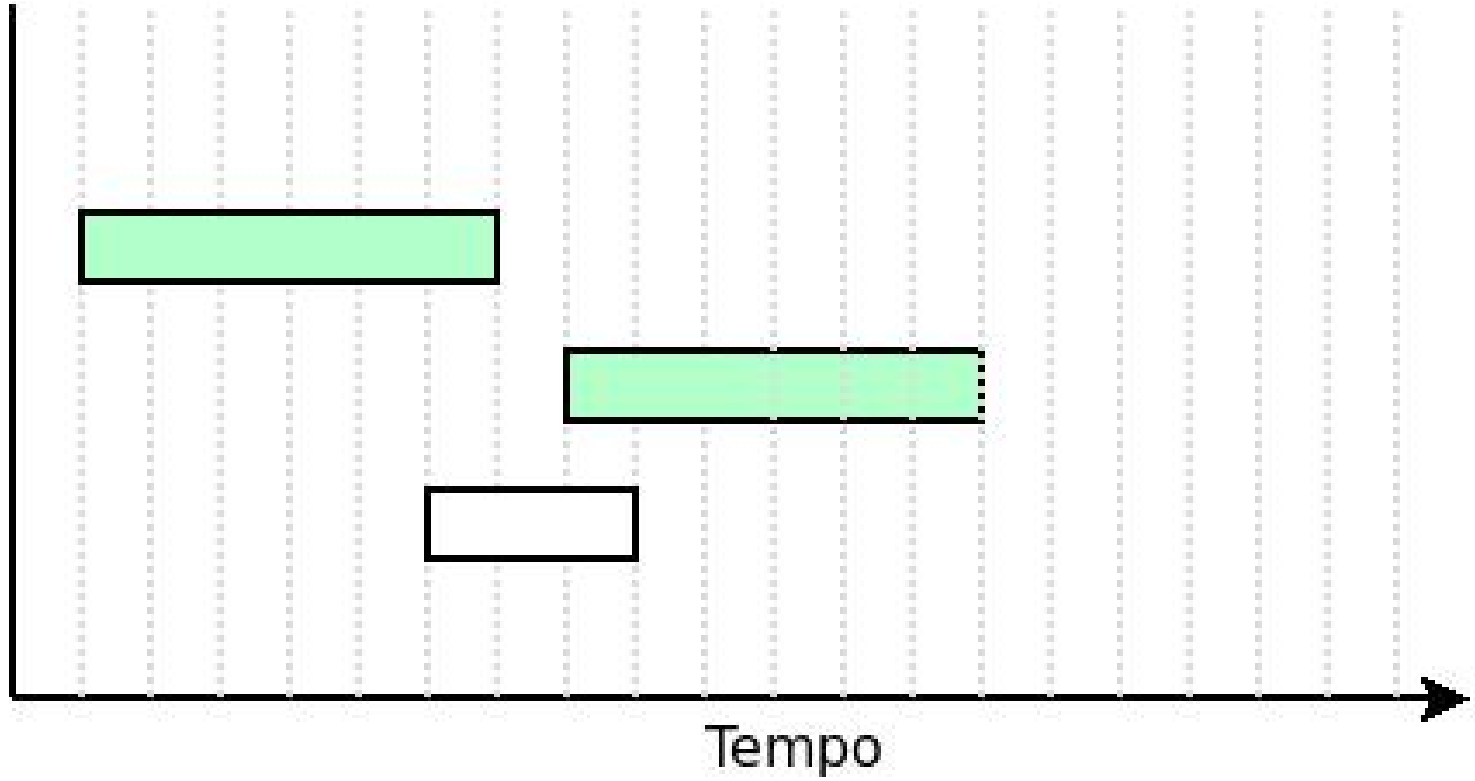
Contra-Exemplo:



Interval Scheduling

[Intervalo mais pequeno] Alocar por ordem ascendente de $f_i - s_i$

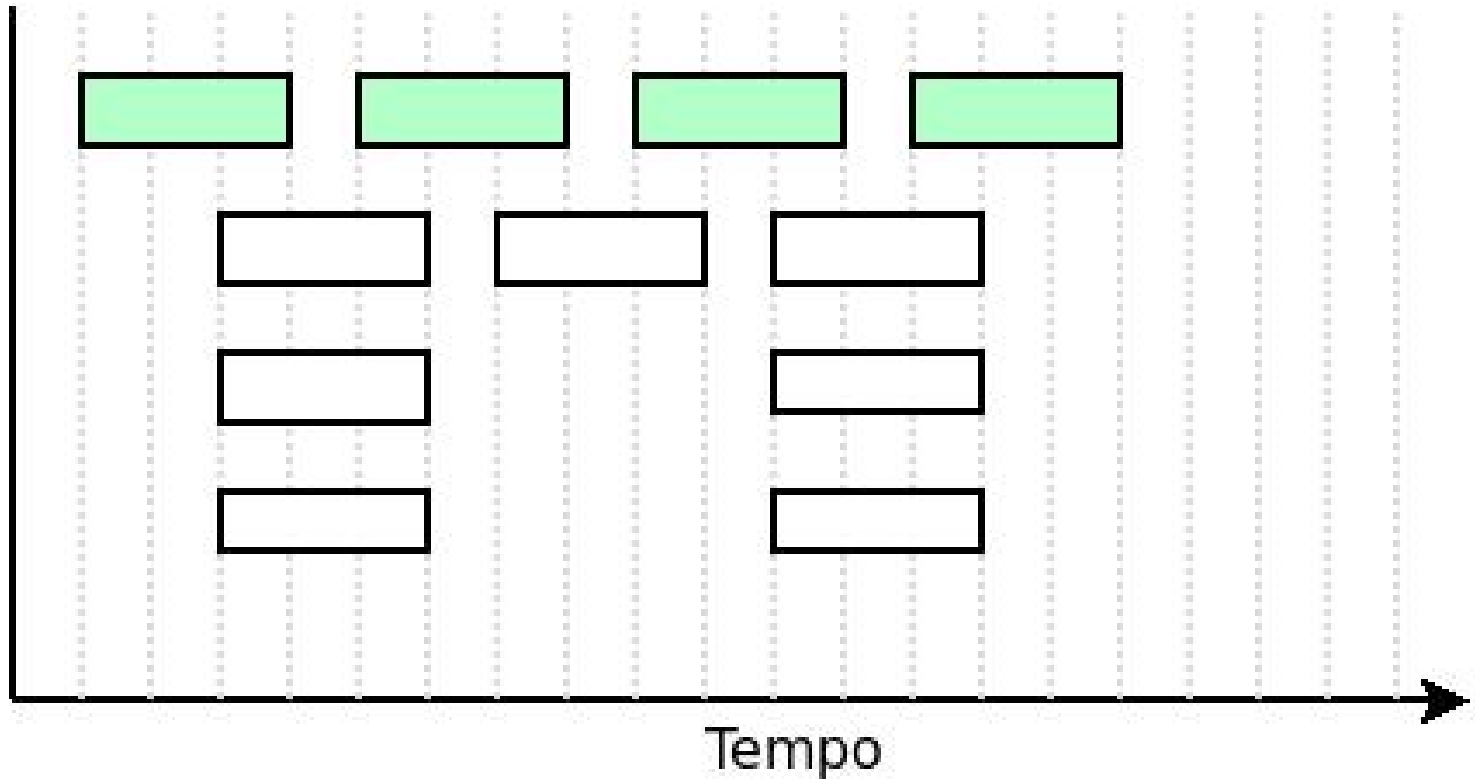
Contra-Exemplo:



Interval Scheduling

[Menos conflitos] Alocar por ordem ascendente do número de outras actividades que estão sobrepostas

Contra-Exemplo:



Interval Scheduling

[Final mais cedo] Alocar por ordem ascendente de f_i

Contra-Exemplo: Não existe!
De facto esta estratégia greedy produz solução ótima!

Teorema

Escolher sempre a actividade não sobreposta com as já escolhidas que tenha o menor tempo de finalização produz uma solução ótima.

1) Subestrutura Ótima

Considere uma solução ótima e actividade m com menor f_m.

Se retirarmos essa actividade então o restante tem de conter a solução ótima para as outras actividades que começam depois de f_m.

Caso assim não seja, então a solução inicial também não era ótima!

Interval Scheduling

Teorema

Escolher sempre a actividade não sobreposta com as já escolhidas que tenha o menor tempo de finalização produz uma solução ótima.

2) Propriedade da Escolha Greedy

Vamos assumir que as actividades estão ordenadas por ordem crescente de tempo de finalização

Seja $G = \{g_1, g_2, \ldots, g_m\}$ a solução criada pelo algoritmo greedy.

Vamos mostrar por **indução** que dada qualquer outra solução ótima H, podemos modificar as primeiras k actividades de H para corresponderem às primeiras k actividades de G, sem introduzirmos nenhuma sobreposição.

Quando $k = n$, a solução H corresponde a G e logo $|G| = |H|$.

Interval Scheduling

Caso base: $k = 1$

- Seja outra solução ótima $H = \{h_1, h_2, \ldots, h_m\}$
- Temos de mostrar que g_1 podia substituir h_1
- Por definição, temos que $f_{g_1} \leq f_{h_1}$
- Sendo assim, g_1 podia ficar no lugar de h_1 sem criar nenhuma sobreposição
- Isto prova que g_1 pode ser o início de qualquer solução ótima!

Interval Scheduling

Passo Indutivo (assumindo que é verdade até k)

- Assumimos que outra solução ótima $H = \{g_1, \ldots, g_k, h_{k+1}, \ldots h_m\}$
- Temos de mostrar que g_{k+1} podia substituir h_{k+1}
- $s_{g_{k+1}} \geq f_{g_k}$ (não existe sobreposição)
- Logo, $f_{g_{k+1}} \leq f_{h_{k+1}}$ (o algoritmo greedy escolhe desse modo)
- Sendo assim, g_{k+1} podia ficar no lugar de h_{k+1} sem criar nenhuma sobreposição
- Isto prova que g_{k+1} pode ser ser escolhido para extender a solução greedy!

Interval Scheduling

Algoritmo greedy para Interval Scheduling

- Ordenar actividades por ordem crescente de tempo de finalização
- Começar por iniciar $G = \emptyset$
- Ir adicionando a G a próxima actividade da lista (com menor f_i, portanto) que não esteja sobreposta com nenhuma actividade de G

Complexidade:

- Ordenar: $O(n \log n)$
- Processar: $O(n)$
- Total: $O(n \log n)$

Cobertura Mínima

Problema da cobertura mínima

Input: Um conjunto de n segmentos de linha com coordenadas não negativas $[l_i, r_i]$, e um número M.

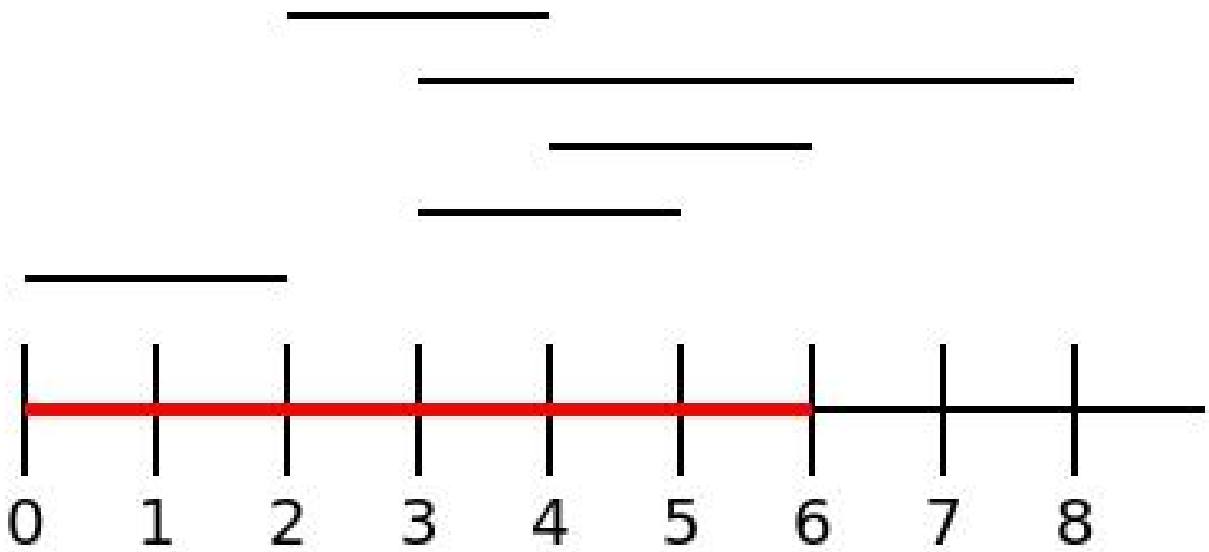
Output: Descobrir a menor quantidade possível de segmentos que cobrem o segmento $[0, M]$.

Cobertura Mínima

Exemplo de Input

Input: 5 segmentos, M=6 :

i	1	2	3	4	5
l_i	0	3	4	3	2
r_i	2	5	6	8	4



Cobertura Mínima

"Padrão" greedy: estabelecer uma ordem segundo um determinado critério e depois ir escolher segmentos cubram zona ainda não coberta

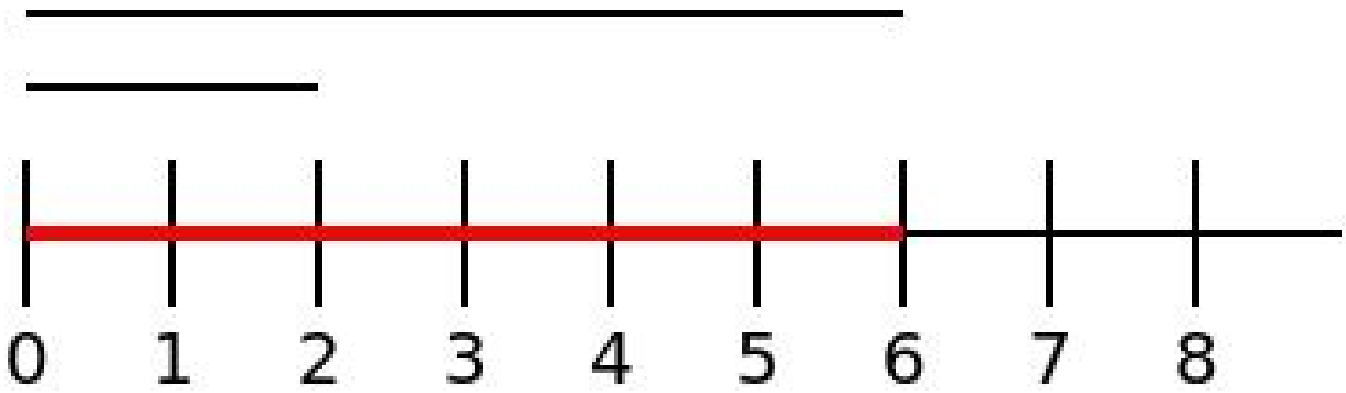
Algumas possíveis ideias:

- [Início mais cedo] Alocar por ordem ascendente de l_i
- [Final mais cedo] Alocar por ordem ascendente de r_i
- [Tamanho maior] Alocar por ordem descendente de $r_i - l_i$

Cobertura mínima

[Final mais cedo] Alocar por ordem ascendente de r_i

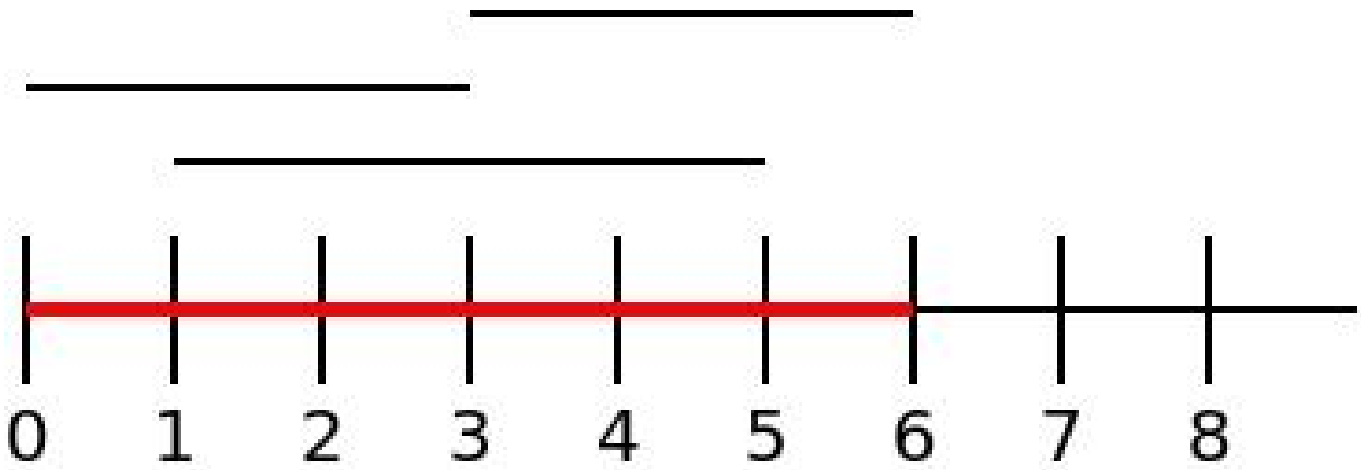
Neste problema não faz sentido!
Contra-Exemplo:



Cobertura mínima

[Tamanho maior] Alocar por ordem descendente de $r_i - l_i$

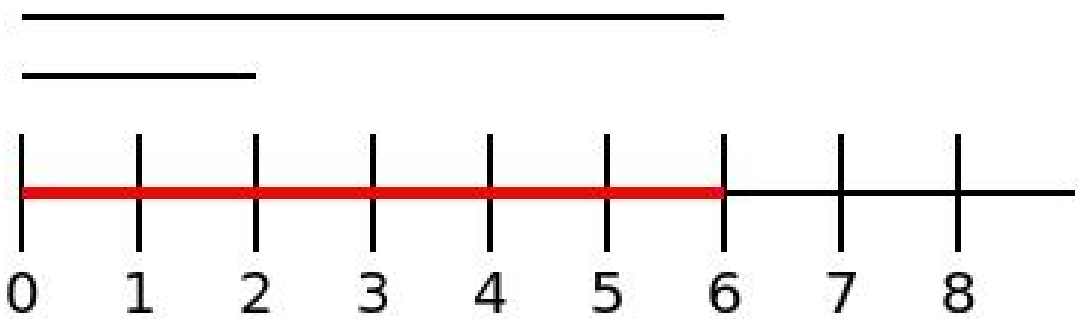
Contra-Exemplo:



Cobertura mínima

[Início mais cedo] Alocar por ordem ascendente de l_i

Parece ser uma boa ideia, porque precisamos de alocar o espaço desde início....
Mas o que acontece se existirem empates?

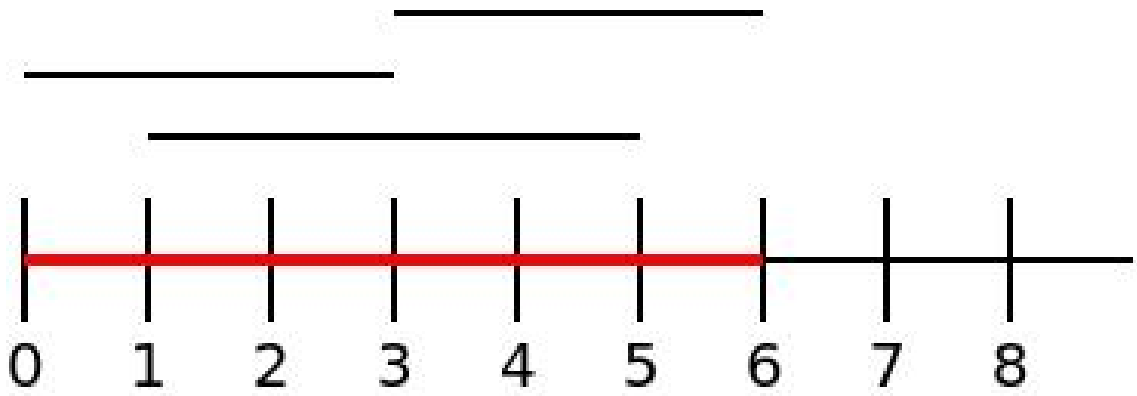


Em caso de empate escolhemos o maior! (o que termina depois)
E será isto suficiente?

Cobertura mínima

[Início mais cedo] Alocar por ordem ascendente de l_i e em caso de empate escolher o maior

O que acontece neste caso?



Se já temos coberto até ao ponto *end*, temos de escolher o segmento que começa em ponto inferior ou igual a *end* e termina o mais para a frente possível!

Intuição: temos sempre de cobrir a partir de *end*. Logo, o melhor que podemos fazer é com um único segmento cobrir até o mais longe possível!

Cobertura mínima

Algoritmo greedy para cobertura mínima

- Ordenar actividades por ordem crescente do seu início (l_i).
- Começar por iniciar $end = 0$ (sendo que vamos sempre tendo coberto o segmento $[0, end]$)
- Processar na lista todos os segmentos que têm início pelo menos em end ($l_i \leq end$), e escolher destes o que termina depois (maior r_i).
- Actualizar end para o sítio onde termina o segmento escolhido e repetir o passo anterior até que $end \geq M$

Complexidade:

- Ordenar: $O(n \log n)$
- Processar: $O(n)$
- Total: $O(n \log n)$

Soma de Custo Mínimo

Imagine que somar a e b "custa" $a + b$. Por exemplo, somar 1 com 10 custaria 11.
Se agora quisermos somar os números $\{1, 2, 3\}$, existem várias maneiras (ordens) de o fazer, dando origem a custos totais diferentes:

- Hipótese **A**
  1. $1 + 2 = 3$ (custo 3)
  2. $3 + 3 = 6$ (custo 6)

  Custo Total: 9
- Hipótese **B**
  1. $1 + 3 = 4$ (custo 3)
  2. $4 + 2 = 6$ (custo 6)

  Custo Total: 10
- Hipótese **C**
  1. $2 + 3 = 5$ (custo 5)
  2. $5 + 1 = 6$ (custo 6)

  Custo Total: 11

Soma de Custo Mínimo

Problema da Soma de Custo Mínimo

Input: Um conjunto de n números inteiros.

Output: Descobrir o menor custo possível para os somar todos, sabendo que somar a com b tem um custo de $a + b$

Exemplo de Input/Output

Input: 3 números: $\{1, 2, 3\}$

Output: 9 (custo de fazer primeiro $1 + 2$ seguido de $3 + 3$).

Soma de Custo Mínimo

- Que estratégia **greedy** usar aqui? E já estou a ajudar muito ao dizer que greedy funciona...
- Escolher em cada momento os dois números mais pequenos!

  Intuição:

  - quanto menores os números, menor o custo
  - a soma final é inevitável (seja qual for a ordem terá custo igual à soma de todos)
  - consideremos os números a, b e c. Se a solução greedy optar por $a+b$ é porque $c \geq a$ e $c \geq b$. Sendo assim, o custo de $a+c$ ou de $b+c$ seria superior ou igual $a+b$ e depois viria uma soma com custo $a+b+c$.
  - nenhuma outra solução consegue ser melhor que greedy!

Soma de Custo Mínimo

Algoritmo greedy para soma de custo mínimo

- Repetir os seguintes passos até a lista de números ficar vazia
  - Remover os dois números mais pequenos a e b
  - Adicionar $a + b$ ao custo total
  - Inserir $a + b$ na lista de números

Complexidade:

- Número de Passos: $O(n)$
- Cada passo:
  - Remover dois mais pequenos
  - Adicionar dois números
  - Inserir dois números
- Total: Depende das operações de retirar dois mínimos e de inserir!

Soma de Custo Mínimo

- Adicionar dois números: $O(1)$!
- Inserir e Remover (os dois mais pequenos):
  - Array desordenado: inserir em $O(1)$ e remover em $O(n)$
  - Array ordenado: inserir em $O(n)$ e remover em $O(1)$
  - Estrutura de dados especializada (ex: heap): inserir em $O(\log n)$ e remover em $O(\log n)$

Complexidade:

- Número de Passos: $O(n)$
- Cada passo:
  - $O(n)$ para um algoritmo mais "básico"
  - $O(\log n)$ se tivermos uma estrutura de dados especializada
- Total: $O(n^2)$ ou $O(n \log n)$

Algoritmos Greedy

- Uma ideia muito poderosa e flexível
- O difícil é provar que dá origem a resultado ótimo
  - Optimalidade não é garantida porque não explora de forma completa todo o espaço de procura
  - Geralmente é mais fácil provar a incorrecção (via contra-exemplo)
  - Uma maneira de analisar é pensar num caso onde existem empates na condição greedy: o que escolhe o algoritmo nesse caso?
- Quando funcionam, costumam ter complexidade baixa
- Não existe "receita mágica" para todos os *greedy*: experiência é necessária!