Complexidade de Algoritmos

Mariana Kolberg

Projeto de Algoritmos
Algoritmos Gulosos

Aula 2

Complexidade de Algoritmos

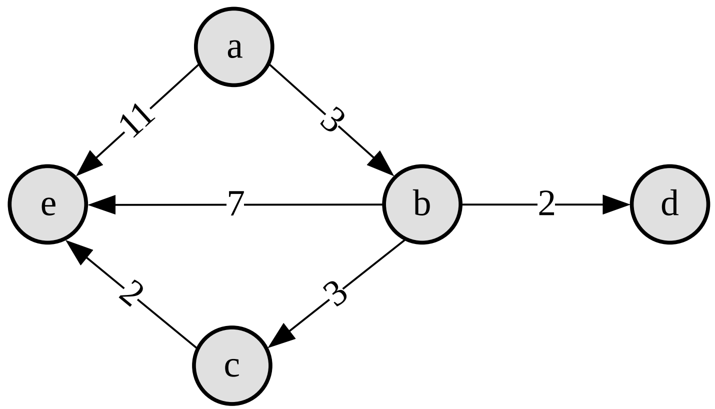
Caminhos de custo mínimo em grafo orientado

Este problema consiste em determinar um caminho de custo mínimo a partir de um vértice fonte a cada vértice do grafo.

Considere um grafo orientado G = < V, E > com 5 vértices: V = {a, b, c, d, e} e

6 arestas com a seguinte matriz de custos:

Custo	a	b	c	d	e
a	0	3	∞	∞	11
b	∞	0	3	2	7
c	∞	∞	0	∞	2
d	∞	∞	∞	0	∞
e	∞	∞	∞	∞	0



Complexidade de Algoritmos

Algoritmo: Custo mínimo de caminhos a partir de fonte em grafo orientado

Inicialização

0\. $V_0 \leftarrow V - \{v_0\}$; {vértices não fonte: $v_1, \ldots, v_n$}

1\. $p \leftarrow V_0$; {inicializa porção da entrada com $\{v_1, \ldots, v_n\}$}

2\. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u> dist[i] ← custo $[v_0, v_i]$ {inicializa resposta parcial}

Iteração

3\. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u> {itera: 10}

4\. $e \leftarrow$ vértice $v_j \in p$ com dist[j] mínimo; {seleciona novo vértice}

5\. $p \leftarrow p - \{e\}$; {remove vértice selecionado}

6\. <u>para</u> <u>cada</u> $v_i \in V_0$ <u>faça</u> {vértices $v_1, \ldots, v_n$: 9}

7\. $c \leftarrow$ dist[j] + custo$[v_j, v_i]$; {distância usando v_j}

8\. <u>se</u> $c \leq$ dist[i] <u>então</u> dist[i] ← c ; {atualiza distância}

9\. <u>fim-para</u> {6: cada $v_i \in V_0$}

10\. <u>fim-para</u> {3: repita}

Finalização

11\. <u>retorne-saída</u>(dist) ; {dá como saída resposta pronta}

12\. <u>fim-Função</u> {fim do algoritmo Dist_fnt: distância a partir da fonte}

Complexidade de Algoritmos

O algoritmo

recebe como **entrada**

Um grafo orientado valorado G com **fonte $\mathbf{v_0}$** e uma **matriz de custos**

fornece como **saída**

Um vetor dist (com os custos dos melhores caminhos a partir de v_0).

Consideremos um grafo orientado G com conjunto $\mathbf{V = \{ v_0, v_1, \ldots, v_n \}}$ de vértices.

As operações fundamentais do algoritmo são as manipulações com conjuntos (de vértices) e matrizes; e para o tamanho da entrada o número **n** de vértices não fonte.

Complexidade de Algoritmos

Encontre a complexidade pessimista do algoritmo abaixo, que tem contribuições dadas por suas componentes: **inicialização, iteração e finalização**.

Inicialização	0. $V_0 \leftarrow V - \{v_0\}$;	{vértices não fonte: $v_1, \ldots, v_n$}
	1. $p \leftarrow V_0$;	{inicializa porção da entrada com $\{v_1, \ldots, v_n\}$}
	2. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u> dist[i] $\leftarrow$ custo[0, i];	{inicializa resposta parcial}
Iteração	3. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u>	{itera: 10}
	4. $e \leftarrow$ vértice $v_j \in p$ com dist[j] mínimo;	{seleciona novo vértice}
	5. $p \leftarrow p - \{e\}$;	{remove vértice selecionado}
	6. <u>para</u> <u>cada</u> $v_i \in V_0$ <u>faça</u>	{vértices $v_1, \ldots, v_n$: 9}
	7. $c \leftarrow$ dist[j] + custo[v_j, v_i];	{distância usando v_j}
	8. <u>se</u> $c \leq$ dist[i] <u>então</u> dist[i] $\leftarrow c$;	{atualiza distância}
	9. <u>fim-para</u>	{6: cada $v_i \in V_0$}
	10. <u>fim-para</u>	{3: repita}
Finalização	11. <u>retorne-saída</u>(dist);	{dá como saída resposta pronta}
	12. <u>fim-Função</u>	{fim do algoritmo Dist_fnt: distância a partir da fonte}

Complexidade de Algoritmos

O desempenho do **Algoritmo** tem contribuições dadas por suas componentes:

inicialização, iteração e finalização.

Inicialização

```
0. V0 ← V-{v0 };
1. p ← V0 ;
2. para i de 1 até n faça dist[i] ← custo[0, i];
3. para i de 1 até n faça
4.   e ← vértice vj ∈ p com dist[j] mínimo ;
5.   p ← p-{e};
6.   para cada vi ∈ V0 faça
7.   c ← dist[j] + custo[vj, vi];
8.   se c ≤ dist[i] então dist[i] ← c ;
9.   fim-para
10. fim-para
11. retorne-saída(dist);
12. fim-Função
```

A inicialização fornece valores iniciais às variáveis.
Portanto, temos

comando	desempenho
0. $V_0 \leftarrow V - \{v_0\}$;	1
1. $p \leftarrow V_0$;	n
2. para i de 1 até n faça dist[i] ← custo[0, i];	n

Logo,

$$\text{desemp[Inicialização]} = 1 + n + n = 2n + 1$$

Complexidade de Algoritmos

A iteração **executa** n vezes a **seleção, remoção e inclusão de um elemento** na resposta parcial se viável.

```
V0 ← V - {v0 };
p ← V0;
para i de 1 até n faça dist[i] ← custo[0, i];
para i de 1 até n faça
  e ← vértice vj ∈ p com dist[j] mínimo;
  p ← p - {e};
  para cada vi ∈ V0 faça
  c ← dist[j] + custo[vj, vi];
  se c ≤ dist[i] então dist[i] ← c ;
  fim-para
fim-para
retorne-saída(dist);
fim-Função
```

Iteração

As variáveis p e dist variam da seguinte maneira

I	0	1	...	k	...	n
$\|p_i\|$	n	n-1	...	n-k	...	0
Dim($dist_i$)	n	n	...	n	...	n

Complexidade de Algoritmos

0. $V_0 \leftarrow V - \{v_0\}$;
1. $p \leftarrow V_0$;
2. para i de 1 até n faça $dist[i] \leftarrow custo[0, i]$;

Iteração

3. para i de 1 até n faça
4. $e \leftarrow$ vértice $v_j \in p$ com $dist[j]$ mínimo;
5. $p \leftarrow p - \{e\}$;
6. para cada $v_i \in V_0$ faça
7. $c \leftarrow dist[j] + custo[v_j, v_i]$;
8. se $c \leq dist[i]$ então $dist[i] \leftarrow c$;
9. fim-para
10. fim-para
11. retorne-saída$(dist)$;
12. fim-Funcão

No ínicio da i-ésima iteração, $| p_i | = n - i + 1$

O desempenho da iteração é dado pela soma das contribuições das linhas de 4 a 5 e das linhas de 6 a 9.

As cotas superiores para as linhas 4 e 5 são

Comando	cota superior
4. $e \leftarrow$ vértice $v_j \in p$ com $dist[j]$ mínimo;	$n-i+1$
5. $p \leftarrow p - \{e\}$;	$n-i+1$

Logo,

$$desemp[\text{linhas 4-5}](p_i, dist_i) = 2(n-i+1)$$

Complexidade de Algoritmos

0. $V_0 \leftarrow V\text{-}\{v_0\}$;
1. $p \leftarrow V_0$;
2. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u> dist[i] ← custo[0, i];
3. <u>para</u> i <u>de</u> 1 <u>até</u> n <u>faça</u>
4. e ← vértice $v_j \in$ p com dist[j] mínimo;
5. p ← p-{e};
6. <u>para</u> <u>cada</u> $v_i \in V_0$ <u>faça</u>
7. c ← dist[j] + custo[v_j, v_i];
8. <u>se</u> c ≤ dist[i] <u>então</u> dist[i] ← c ;
9. <u>fim-para</u>
10. <u>fim-para</u>
11. <u>retorne-saída</u>(dist);
12. <u>fim-Função</u>

Iteração (linhas 3–10)

As cotas superiores para o desempenho das linhas 7 e 8 são

	comando	cota superior
7.	c ← dist[j] + custo[v_j, v_i];	1
8.	<u>se</u> c ≤ dist[i] <u>então</u> dist[i] ← c	1
	linhas 7-8	1 + 1 = 2

Logo, temos

$$desemp[\text{linhas 6-9}](p_{i+1}, dist_i) = 2.n$$

O desempenho do corpo da iteração na i-ésima iteração é

$$desemp[\text{linhas 4-5}](p_i, dist_i) = 2(n-i+1)$$
$$+$$
$$desemp[\text{linhas 6-9}](p_{i+1}, dist_i) = 2.n$$

$$desemp[\text{Crp\_Dist\_fnt}](p_i, dist_i) = 2(n-i+1) + 2.n$$

Complexidade de Algoritmos

A iteração repete n vezes o corpo da iteração, logo o seu desempenho é

$$\text{desemp[Iteração]} = \sum_{i=1}^{n}[2(n-i+1)+2n]$$

$$= \sum_{i=1}^{n} 2(n-i+1) + \sum_{i=1}^{n} 2n$$

$$= 2\sum_{i=1}^{n}(n-i+1) + 2n^2$$

$$= 2\sum_{i=1}^{n}(i) + 2n^2$$

$$= 2\frac{n(n+1)}{2} + 2n^2$$

$$\text{desemp[Iteração]} = n^2 + n + 2n^2 = 3n^2 + n$$

Complexidade de Algoritmos

Projeto e Análise de Algoritmos

O desempenho do algoritmo é dado predominantemente pelo desempenho da inicialização e da iteração. Assim, temos

$$\begin{array}{l} \text{desemp[Inicialização]} = 2n + 1 \\ \qquad\qquad + \\ \text{desemp[Iteração]} = 3n^2 + n \\ \hline \text{desemp[Algoritmo]} = 3n^2 + 3n + 1 \end{array}$$

A complexidade pessimista do algoritmo é

$$c_p^{\leq}[\text{Algoritmo}](n) = O(n^2)$$

Complexidade de Algoritmos

- Algoritmos gulosos
  - Sempre faz a escolha que parece ser a melhor no momento
  - Escolha ótima para condições locais
    - Esperando que esta escolha leve a uma solução ótima global
    - Nem sempre produz uma solução ótima
- Subestrutura ótima
  - Uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas
  - Problemas com subestrutura ótima podem ser solucionados por:
    - Algoritmos gulosos (+ propriedade da escolha gulosa)
      - Normalmente existe uma solução usando programação dinâmica para um algoritmo guloso (mas é mais custoso em termos de espaço!).
    - Programação dinâmica (+ subproblemas superpostos)

Complexidade de algoritmos

- Etapas para projetar um algoritmo guloso:
  - Moldar o problema de otimização como um problema no qual fazemos uma escolha e ficamos com um único subproblema para resolver
  - Provar que existe uma solução ótima para o problema original que torna a escolha gulosa, de modo que a escolha gulosa é sempre segura
  - Demonstrar que, tendo feito a escolha gulosa, o que resta é um subproblema com a propriedade de que, se combinarmos uma solução ótima para o subproblema com a escolha gulosa que fizemos, chegamos a uma solução ótima global.

Complexidade de algoritmos

- Propriedade da escolha gulosa
  - Solução global ótima pode ser alcançada fazendo uma escolha local ótima (gulosa)
  - Escolha local:
    - Considera o que é melhor no problema atual
    - Pode depender de escolhas até o momento, mas não de escolhas futuras ou soluções de subproblemas (diferença da prog. dinâmica)
    - É top-down - reduz de modo iterativo cada instância do problema dado a um problema menor
    - Frequentemente, usando uma estrutura de dados apropriada (muitas vezes filas de prioridades), podemos fazer escolhas gulosas rapidamente.

Complexidade de algoritmos

- Problema de seleção de atividade
  - Programação do uso de uma sala entre diversas atividades concorrentes
  - Objetivo: selecionar um conjunto de tamanho máximo de atividades mutualmente compatíveis.
  - Conjunto $S=\{a_1,a_2,\ldots,a_n\}$ de n atividades
  - Atividade ai tem um tempo de inicio si e tempo de término f_i
    - $0 \leq f_i < \infty$
  - Duas atividades a_i e a_j com intervalos $[s_i,f_i)$ e $[s_j,f_j)$
    - São compativeis se os intervalos não se superpõem ($s_i \geq f_j$ ou $s_j \geq f_i$)

Complexidade de algoritmos

- Considere o conjunto S de atividades
  - Ordenado de forma crescente pelo tempo de término:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

  - Qual o subconjunto máximo de atividades mutualmente compatíveis?
  - Existe só um?

Complexidade de algoritmos

- Substrutura ótima:
  - Definição do espaço de subproblemas
    - $S_{ij}=\{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$
    - Conjunto de atividades em S que podem começar após a atividade a_i terminar, e que terminam antes da atividade a_j começar
    - Suponha a_k pertencendo a $S_{ij} = S_{ik} \cup S_{kj} + a_k$
      - □ $f_i \leq s_k < f_k \leq s_j$
      - □ S_{ik} – atividades que começam depois de a_i e terminam antes de a_k
      - □ S_{kj} – atividades que começam depois de a_k e terminam antes de a_j
  - Suponha uma solução ótima para S_{ij} é A_{ij} e inclui a_k. Implica que
    - A_{ik} e A_{kj} devem ser ótimos
    - Se existisse A'_{ik} com mais atividades do que A_{ik}, entao poderiamos ter outra solução para S_{ij}, e A_{ij} não seria ótima → contradição

Complexidade de algoritmos

- Propriedade da escolha gulosa
  - De fato precisamos apenas considerar uma escolha, e ao optarmos por esta escolha, um dos subproblemas tem garantia de ser vazio, restando apenas 1 subproblema para resolver.
  - Considere qualquer subproblema não vazio S_{ij} e seja a_m a atividade em S_{ij} com tempo de término mais antigo.
  - Se escolhermos sempre o subproblema com tempo de término mais antigo (p. ex. a_m), dividimos nosso problema em 2 subproblemas possíveis:
    - S_{im} este conjunto é vazio
    - S_{mj} único que pode ser escolhido por ser não vazio
  - Escolhendo sempre a atividade com tempo de termino mais antigo, maximizamos a quantidade de tempo restante para outras atividades.

Complexidade de algoritmos

- Algoritmo:

```
Greedy-AS( S )
  // Assumes S sorted in order of increasing f_k
  A = {a_1} ;   i = 1
  for m = 2 to n
    if s_m ≥ f_i              // a_m is compatible with a_i
      A = A ∪ {a_m}  ;   i = m
  return A
```

- Qual a complexidade deste algoritmo?

Complexidade de algoritmos

- Problema da mochila
  - Dados:
    - Uma mochila que admite um certo peso;
    - Um conjunto de objetos, cada um com um valor e um peso;
  - Objetivo:
    - Selecionar o conjunto de objetos que caibam dentro da mochila de forma a maximizar o valor total dentro da mochila.
  - Mochila binária
    - Um ladrão rouba uma loja que contém n itens: o item i tem peso wi e vale vi, ∈ N. Ele quer levar o maior valor possível em uma mochila de carga máxima W. Quais itens escolher?
    - Pense em ouro em barras
  - Mochila fracionaria
    - Mesmo formulação anterior, mas agora ele pode carregar frações dos itens, ao invés da escolha binária (0-1).
    - Pense em ouro em pó

Complexidade de algoritmos

- Problema da mochila
  - Subestrutura ótima:
    - Ambos tem sub-estrutura ótima: Se removermos j de uma solução ótima, a carga restante deve ser a de maior valor para carga máxima W − wj tomando n − 1 itens (caso 0-1).
  - Algum destes problemas tem uma escolha gulosa que pode ser aplicada?

Complexidade de algoritmos

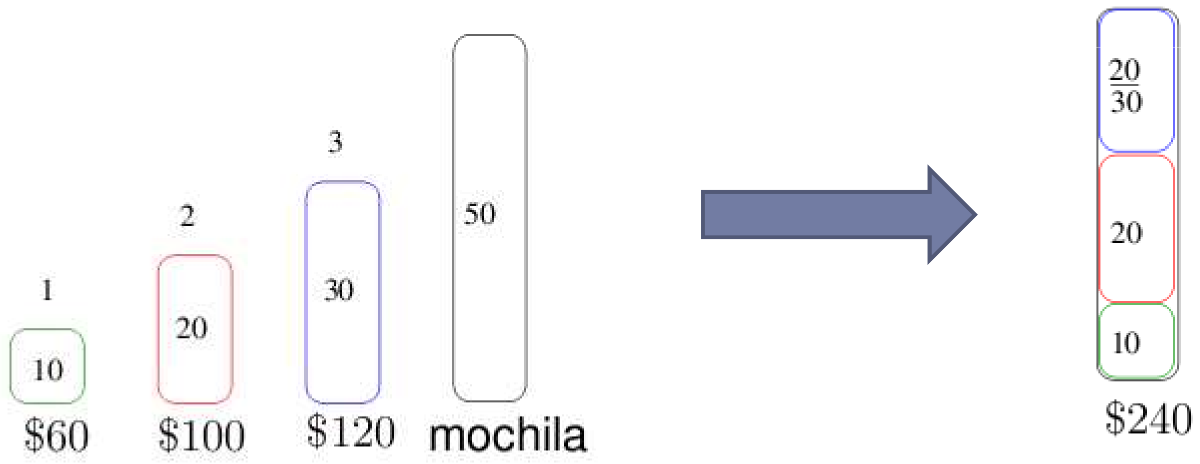
- Problema da mochila
  - Subestrutura ótima:
    - Ambos tem sub-estrutura ótima: Se removermos j de uma solução ótima, a carga restante deve ser a de maior valor para carga máxima W − wj tomando n − 1 itens.
  - Algum destes problemas tem uma escolha gulosa que pode ser aplicada?

Complexidade de algoritmos

- Problema da mochila
  - Algum destes problemas tem uma escolha gulosa que pode ser aplicada?
    - Problema da mochila fracionaria!
  - Qual seria a melhor ordenação da entrada?
    - ordenar pelo valor/peso
  - Algoritmo:
    - Para resolver o Problema da Mochila Fracionada ordene os itens por valor/peso decrescentemente
    - Começando em i = 1 coloque na mochila o máximo do item i que estiver disponível e for possível, e se puder levar mais passe para o próximo item.

Complexidade de algoritmos

- Valor/peso:
  - 1 – 6,00 $
  - 2 – 5,00 $
  - 3 – 4,00 $



Complexidade de algoritmos

Algoritmo MOCHILA_FRACIONADA
Entrada: Conj. ordenado S de itens de valor v_i e peso w_i cada e capacidade máxima W
Saída: x_i de cada item i que maximiza o valor sem exceder W

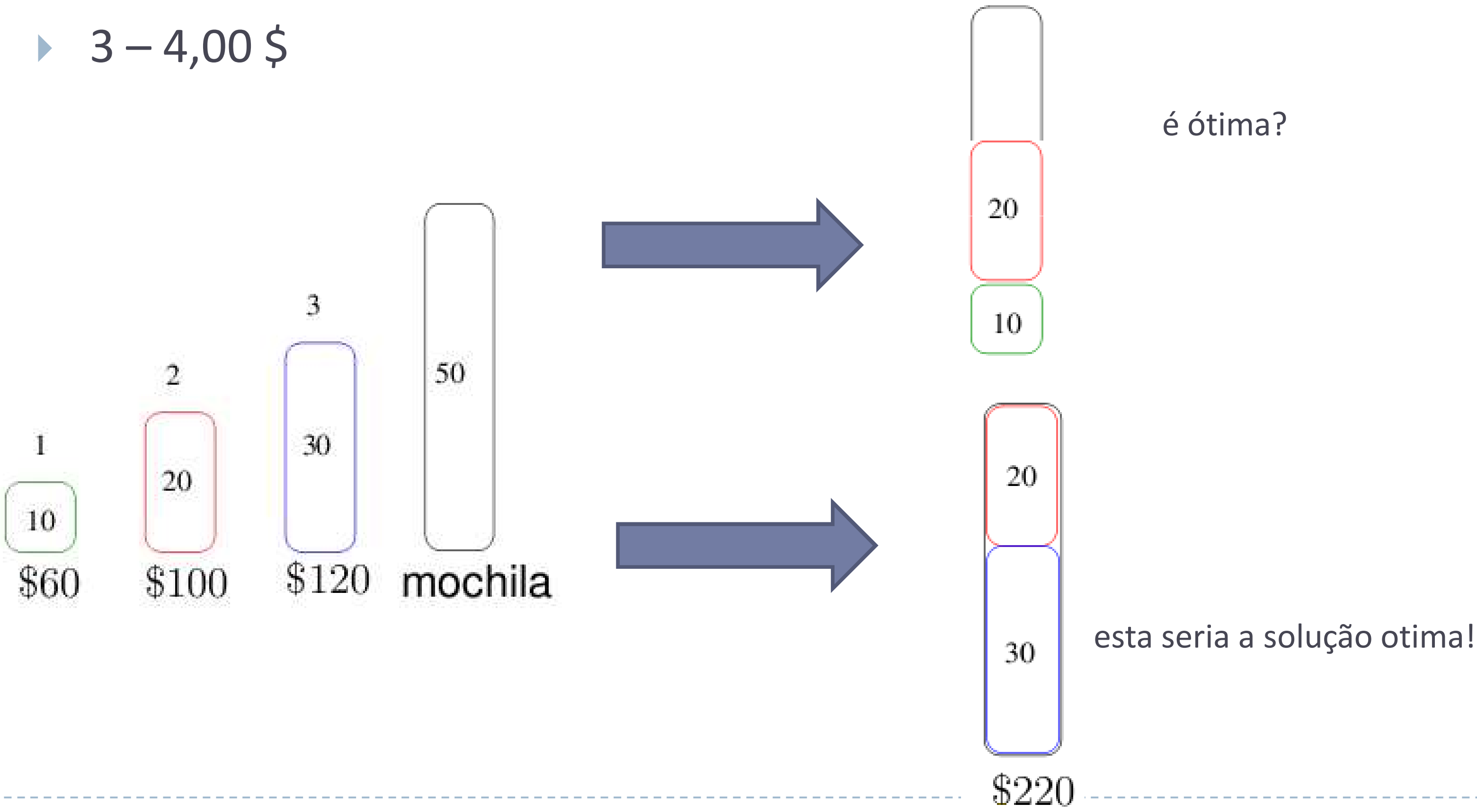
$load = 0$
$i = 1$
while $load < W$ e $i \leq n$ **do**
  if $w_i \leq W - load$ **then**
    Pegue todo o item i
  else
    Pegue $(W - load)/w_i$ do item i
  Adicione a $load$ o peso que foi pego
  $i = i + 1$

Complexidade de algoritmos

- Problema da mochila binária
  - Qual seria a melhor ordenação da entrada?
    - ordenar pelo valor/peso

Complexidade de algoritmos

- Valor/peso:
  - 1 – 6,00 $
  - 2 – 5,00 $
  - 3 – 4,00 $



Complexidade de algoritmos

- Problema da mochila
  - Complexidade do MOCHILA_FRACIONADA O(n)
  - O Problema da mochila 0-1 pode ser resolvido usando programação dinâmica em tempo O(nW). Algoritmo Pseudo-polinomial.