

# Técnicas para Programação Competitiva Paradigmas de Resolução de Problemas

Prof. Andrei Braga



# Conteúdo

- Divisão e conquista
- Algoritmos gulosos
- Referências

# Motivação

- Vamos estudar paradigmas de resolução de problemas comumente utilizados para tratar problemas de competições de programação
- Para ter sucesso em competições de programação, precisamos ter um bom domínio sobre estes paradigmas, sabendo usar a opção apropriada para o problema em questão

# Divisão e conquista

- **Divisão e conquista** é um método de resolução de problemas que executa os três seguintes passos:
  1. Divida o problema em um ou mais subproblemas menores – os subproblemas são instâncias menores do mesmo problema (**divisão**)
  2. Resolva os subproblemas (**conquista**)
  3. Combine (se necessário) as soluções dos subproblemas para formar uma solução para o problema original
- Um exemplo típico de um algoritmo que segue a estratégia de divisão e conquista é o mergesort

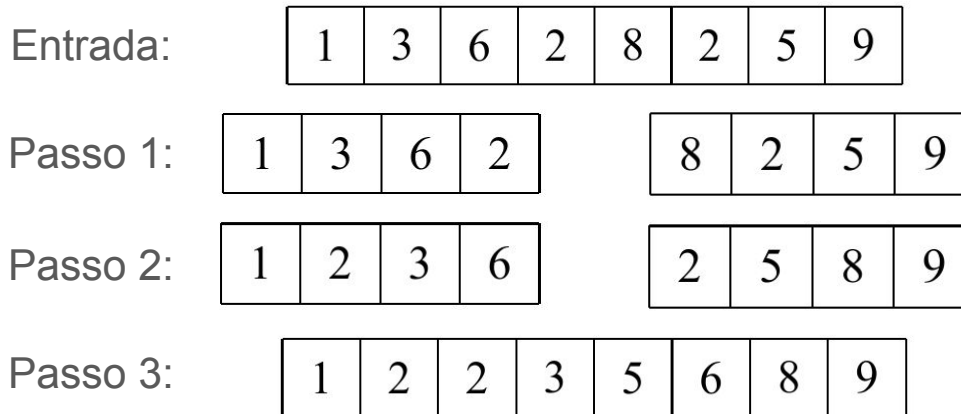
# Mergesort – divisão e conquista

- O **mergesort** é um algoritmo que
  - recebe como entrada um vetor  $\text{vet}$  e dois índices  $i$  e  $f$  tal que  $i \leq f$  e
  - ordena os elementos da parte  $\text{vet}[i..f]$  deste vetor, ou seja, a parte que consiste no elemento de índice  $i$  até o elemento de índice  $f$  do vetor
- Este algoritmo executa os três seguintes passos:
  1. Calcule  $m = \lfloor (i + f) / 2 \rfloor$  e divida a parte  $\text{vet}[i..f]$  a ser ordenada em duas subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  a serem ordenadas (**divisão**)
  2. Ordene recursivamente as subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  (**conquista**)
  3. Combine as subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  já ordenadas para obter a parte  $\text{vet}[i..f]$  ordenada

# Mergesort – divisão e conquista


- Este algoritmo executa os três seguintes passos:
  1. Calcule  $m = \lfloor (i + f) / 2 \rfloor$  e divida a parte  $\text{vet}[i..f]$  a ser ordenada em duas subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  a serem ordenadas (**divisão**)
  2. Ordene recursivamente as subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  (**conquista**)
  3. Combine as subpartes  $\text{vet}[i..m]$  e  $\text{vet}[m+1..f]$  já ordenadas para obter a parte  $\text{vet}[i..f]$  ordenada

- Exemplo:



# Mergesort – divisão e conquista

- O **mergesort** é um algoritmo que
  - recebe como entrada um vetor  $vet$  e dois índices  $i$  e  $f$  tal que  $i \leq f$  e
  - ordena os elementos da parte  $vet[i..f]$  deste vetor, ou seja, a parte que consiste no elemento de índice  $i$  até o elemento de índice  $f$  do vetor
- Este algoritmo executa os ~~três~~ quatro seguintes passos:
  1. Se  $i == f$ , então não faça nada, pois a parte  $vet[i..f]$  contém um elemento só e, portanto, já está ordenada
  2. Calcule  $m = \lfloor (i + f) / 2 \rfloor$  e divida a parte  $vet[i..f]$  a ser ordenada em duas subpartes  $vet[i..m]$  e  $vet[m+1..f]$  a serem ordenadas (**divisão**)
  3. Ordene recursivamente as subpartes  $vet[i..m]$  e  $vet[m+1..f]$  (**conquista**)
  4. Combine as subpartes  $vet[i..m]$  e  $vet[m+1..f]$  já ordenadas para obter a parte  $vet[i..f]$  ordenada



Caso  
base  
da  
recur-  
são

# Problemas de Otimização

- Um **problema de otimização** é um problema tal que
  - cada **solução** para o problema tem um **valor** associado e
  - o objetivo é encontrar uma **solução de valor ótimo** – valor mínimo ou valor máximo – **ou apenas** este **valor ótimo**
- Também chamamos uma solução de valor ótimo de **solução ótima**
- Um problema de otimização pode ter mais de uma solução ótima – por isso, usamos o termo **uma** solução ótima, em vez de *a* solução ótima, para o problema



# Problemas de Otimização

- Um **problema de otimização** é um problema tal que
  - cada **solução** para o problema tem um **valor** associado e
  - o objetivo é encontrar uma **solução de valor ótimo** – valor mínimo ou valor máximo – **ou apenas** este **valor ótimo**
- Exemplo:  
**Problema da Mochila:** Dado um conjunto de itens com seus valores e pesos, qual é o maior valor total de itens que um ladrão consegue carregar em sua mochila (sem ultrapassar a capacidade de peso da mochila)?

# Problemas de Otimização

- Um algoritmo para resolver um problema de otimização geralmente realiza uma sequência de passos, a cada passo fazendo uma ou mais escolhas
- Algoritmos deste tipo podem ser classificados em categorias bastante conhecidas
- Duas destas categorias são **algoritmos gulosos** e **algoritmos de programação dinâmica**

# Algoritmos gulosos

- Um algoritmo para resolver um problema de otimização geralmente realiza uma sequência de passos, a cada passo fazendo uma ou mais escolhas
- Um **algoritmo guloso** faz, a cada passo, uma **escolha** que parece a **melhor possível no momento**
- Em outras palavras, um algoritmo guloso sempre faz uma **escolha localmente ótima** com a esperança de que esta escolha leve a uma **solução globalmente ótima**
- Em muitos casos, **não** é possível resolver o problema através de um algoritmo guloso!

# Algoritmos gulosos

- **Problema:** Dados um valor  $n$  em centavos e moedas de 50, 10, 5 e 1 centavo, determine o menor número possível de moedas tal que a soma das moedas seja  $n$ . Não há restrições quanto a usar várias moedas de um mesmo valor.
- Algoritmo guloso:  
MinNumMoedas( $n$ )
  3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
  4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- **Problema:** Dados um valor  $n$  em centavos e moedas de 50, 10, 5 e 1 centavo, determine o menor número possível de moedas tal que a soma das moedas seja  $n$ . Não há restrições quanto a usar várias moedas de um mesmo valor.
- Algoritmo guloso:  
MinNumMoedas( $n$ )
  1. Se  $n == 0$ :
  2.     Retorne 0
  3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
  4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- **Problema:** Dados um valor  $n$  em centavos e moedas de 50, 10, 5 e 1 centavo, determine o menor número possível de moedas tal que a soma das moedas seja  $n$ . Não há restrições quanto a usar várias moedas de um mesmo valor.
- Algoritmo guloso (iterativo):  
MinNumMoedas( $n$ )
  1. num\_moedas = 0
  2. Enquanto  $n \neq 0$ :
  3.     Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
  4.     num\_moedas = num\_moedas + 1
  5.      $n = n - c$
  6. Retorne num\_moedas

# Algoritmos gulosos

- Para que seja possível de resolver através de um algoritmo guloso, um problema deve ter duas propriedades:
  - Propriedade da **subestrutura ótima**:
    - Uma solução ótima para o problema contém soluções ótimas para subproblemas
  - Propriedade da **escolha gulosa**:
    - Podemos fazer uma escolha que pareça a melhor possível no momento sem ter que considerar resultados de subproblemas

Em outras palavras, podemos obter uma solução globalmente ótima fazendo escolhas localmente ótimas

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2.     Retorne 0
3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- Passos para construir um algoritmo guloso para um problema:
  1. Reconfigure o problema como um problema onde, após feita uma escolha gulosa, exista um subproblema a ser resolvido
  2. Prove que sempre existe uma solução ótima para o problema que satisfaça à escolha gulosa
  3. Prove que, após feita a escolha gulosa, é possível combinar
    - uma solução ótima para o subproblema a ser resolvido com
    - a escolha gulosae obter uma solução ótima para o problema
- Em uma competição, **não vale a pena** fazer as provas dos itens 2 e 3 acima!
- Mas devemos nos convencer de que as condições destes itens são válidas

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2. Retorne 0
3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$



# Algoritmos gulosos

- Passos para construir um algoritmo guloso para um problema:
  2. Prove que sempre existe uma solução ótima para o problema que satisfaça à escolha gulosa
- A seguir, vamos fazer a prova do item 2 acima

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2.     Retorne 0
3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- Seja  $c_1 = 1$ ,  $c_2 = 5$ ,  $c_3 = 10$ ,  $c_4 = 50$
- Considere que uma solução ótima para o problema é uma coleção com o menor número possível de moedas tal que a soma das moedas seja  $n$ 
  - Exemplo: Para  $n = 67$ , a coleção (50, 10, 5, 1, 1) é uma solução ótima
- **Observação 1:** Uma solução ótima contém menos que  $c_{i+1} / c_i$  moedas de valor  $c_i$  para qualquer  $i = 1, 2, 3$
- **Prova:**
  - Note que  $(c_{i+1} / c_i) \times c_i = c_{i+1}$
  - Se uma solução ótima contivesse  $c_{i+1} / c_i$  moedas de valor  $c_i$ , poderíamos trocar estas moedas por uma única moeda de valor  $c_{i+1}$  e obter uma solução com menor número de moedas; isto seria uma contradição  $\square$

# Algoritmos gulosos

- **Observação 2:** Se uma solução ótima não contém nenhuma moeda de valor  $c_j$  ou maior, então o seu valor total é no máximo  $c_j - 1$
- **Prova:**
  - O máximo valor total de uma solução ótima deste tipo é atingido quando a solução contém o maior número possível de moedas de valor  $c_i$  para todo  $i < j$
  - Pela Observação 1, este número é  $(c_{i+1} / c_i - 1)$  para todo  $i < j$
  - Neste caso, o valor total da solução é
  - $(c_2 / c_1 - 1) \times c_1 + (c_3 / c_2 - 1) \times c_2 + \dots + (c_j / c_{j-1} - 1) \times c_{j-1}$
  - $= (c_2 - c_1) + (c_3 - c_2) + \dots + (c_j - c_{j-1})$
  - $= c_j - 1 \quad \square$

# Algoritmos gulosos

- Passos para construir um algoritmo guloso para um problema:

2. Prove que sempre existe uma solução ótima para o problema que satisfaça à escolha gulosa

- **Prova:**

- Seja  $c_j$  o maior valor de moeda tal que  $c_j \leq n$
- Considere uma solução ótima para o problema
- Note que esta solução não pode conter uma moeda de valor maior que  $c_j$  (caso contrário, o valor total da solução seria maior que  $n$ )
- Pela Observação 2, se esta solução não contivesse uma moeda de valor  $c_j$ , então o seu valor total seria no máximo  $c_j - 1 < n$ ; isto seria uma contradição
- Portanto, toda solução ótima para o problema contém  $c_j$  □

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2.     Retorne 0
3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- **Problema (versão diferente):** Dados um valor  $n$  em centavos e moedas de 4, 3 e 1 centavo (suponha que estas moedas existem 😊), determine o menor número possível de moedas tal que a soma das moedas seja  $n$ . Não há restrições quanto a usar várias moedas de um mesmo valor.
- O algoritmo guloso anterior resolve o problema?

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2.     Retorne 0
3. Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4. Retorne  $(1 + \text{MinNumMoedas}(n - c))$

# Algoritmos gulosos

- **Problema (versão diferente):** Dados um valor  $n$  em centavos e moedas de 4, 3 e 1 centavo (suponha que estas moedas existem 😊), determine o menor número possível de moedas tal que a soma das moedas seja  $n$ . Não há restrições quanto a usar várias moedas de um mesmo valor.
- O algoritmo guloso anterior resolve o problema?  
**Não!**  
Se  $n = 6$ , a solução ( 3, 3 ) com 2 moedas é ótima e o algoritmo retorna 3
- A diferença é que, para os valores de moeda 4, 3 e 1, **não** é verdade que cada valor é divisível pelo valor imediatamente menor

MinNumMoedas( $n$ )

1. Se  $n == 0$ :
2.     Retorne 0
3.     Escolha uma moeda de maior valor  $c$  tal que  $c \leq n$
4.     Retorne (1 + MinNumMoedas( $n - c$ ))

Ainda assim, para outros valores de moeda, um alg. guloso pode funcionar

# Referências

- Esta apresentação é baseada nos seguintes materiais:
  1. Capítulo 15 do livro  
Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 4th. ed. MIT Press, 2022.
  2. Capítulo 3 do livro  
HALIM, S.; HALIM, F.; EFFENDY, S. Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s, book 1, chs. 1-4. Lulu, 2018.
  3. Capítulo 6 do livro  
LAAKSONEN, A. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests, 2. ed. Springer, 2020.