MCTA003-17 - Análise de Algoritmos

Aula 05 - Algoritmos Recursivos

Maycon Sambinelli
m.sambinelli@ufabc.edu.br
https://professor.ufabc.edu.br/~m.sambinelli/

2022.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Outline

Introdução

Disclaimer

UFABC

Esses Slides são baseados nos slides dos professores:

- Cid Carvalho de Souza
- Cândida Nunes da Silva
- Lehilton Pedrosa
- Orlando Lee

UFABC

Na aula de hoje

1. Projeto de Algoritmos por indução
2. Correção de Algoritmos recursivos
3. Análise de Algoritmos recursivos
4. Paradigma de Divisão e Conquista

UFABC

Resolvendo um problema com recursão

- **Instâncias pequenas:** resolvemos diretamente

UFABC

Resolvendo um problema com recursão

- Instâncias pequenas: resolvemos diretamente
- Instâncias grandes:

UFABC

Resolvendo um problema com recursão

- Instâncias pequenas: resolvemos diretamente
- Instâncias grandes:
  - Construímos uma instância menor do mesmo problema

UFABC

Resolvendo um problema com recursão

- Instâncias pequenas: resolvemos diretamente
- Instâncias grandes:
  - Construímos uma instância menor do mesmo problema
  - Resolvemos essa substância recursivamente

UFABC

Resolvendo um problema com recursão

- Instâncias pequenas: resolvemos diretamente
- Instâncias grandes:
  - Construímos uma instância menor do mesmo problema
  - Resolvemos essa substância recursivamente
  - Encontramos solução para a instância original

UFABC

- Encontrar e definir um subproblema adequado

Projeto de Algoritmos Recursivos

UFABC

- Encontrar e definir um subproblema adequado
- Supor que sabemos resolver instâncias menores

Projeto de Algoritmos Recursivos

UFABC

- Encontrar e definir um subproblema adequado
- Supor que sabemos resolver instâncias menores
- Construir o algoritmo recursivo para o problema

Projeto de Algoritmos Recursivos

UFABC

- Encontrar e definir um subproblema adequado
- Supor que sabemos resolver instâncias menores
- Construir o algoritmo recursivo para o problema

Projeto de Algoritmos Recursivos

UFABC

- Encontrar e definir um subproblema adequado
- Supor que sabemos resolver instâncias menores
- Construir o algoritmo recursivo para o problema

O Subproblema escolhido:

- Não precisa ser o problema original

Projeto de Algoritmos Recursivos

UFABC

- Encontrar e definir um subproblema adequado
- Supor que sabemos resolver instâncias menores
- Construir o algoritmo recursivo para o problema

O Subproblema escolhido:

- Não precisa ser o problema original
- Costuma ser uma variante ”mais forte”

Recursão e Indução

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas
  - caso geral (*passo*): instâncias grandes

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas
  - caso geral (*passo*): instâncias grandes
    - a chamada recursiva corresponde à hipótese de indução

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas
  - caso geral (*passo*): instâncias grandes
    - a chamada recursiva corresponde à hipótese de indução
    - a construção da solução corresponde ao passo da indução

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas
  - caso geral (*passo*): instâncias grandes
    - a chamada recursiva corresponde à hipótese de indução
    - a construção da solução corresponde ao passo da indução

Recursão e Indução

UFABC

- É fácil obter um algoritmo recursivo da prova indutiva
  - caso básico (*base*): instâncias pequenas
  - caso geral (*passo*): instâncias grandes
    - a chamada recursiva corresponde à hipótese de indução
    - a construção da solução corresponde ao passo da indução

Vantagem: a correção vem da prova indutiva

Problema: Valor do Polinômio

Valor de Polinômio

UFABC

Problema: Valor do Polinômio

Entrada: uma sequência de números reais $A = a_n, a_{n-1}, \ldots, a_1, a_0$

Saída: o valor de $P_A(x)$, onde

$$P_A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Projeto de Algoritmos por Indução

UFABC

1. Você sabe fazer um caso pequeno (*você consegue fazer a base*)?

Projeto de Algoritmos por Indução

UFABC

1. Você sabe fazer um caso pequeno (*você consegue fazer a base*)?
2. Assumindo que a hipótese de indução vale, você consegue resolver o problema?

Projeto de Algoritmos por Indução

UFABC

1. Você sabe fazer um caso pequeno (*você consegue fazer a base*)?
2. Assumindo que a hipótese de indução vale, você consegue resolver o problema?
    - O valor dado pela hipótese de indução seria retornado pela chamada recursiva.

Projeto de Algoritmos por Indução

UFABC

1. Você sabe fazer um caso pequeno (*você consegue fazer a base*)?
2. Assumindo que a hipótese de indução vale, você consegue resolver o problema?
    - O valor dado pela hipótese de indução seria retornado pela chamada recursiva.

UFABC

1. Você sabe fazer um caso pequeno (*você consegue fazer a base*)?
2. Assumindo que a hipótese de indução vale, você consegue resolver o problema?
   - O valor dado pela hipótese de indução seria retornado pela chamada recursiva.

Hipótese de Indução

Dada uma sequência de números reais $A' = a_{n-1}, \ldots, a_1, a_0$ e um número real x, sabemos calcular o valor de

$$P_{A'}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0.$$

UFABC

Hipótese de Indução

Dada uma sequência de números reais $A' = a_{n-1}, \ldots, a_1, a_0$ e um número real x, sabemos calcular o valor de

$$P_{A'}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0.$$

Caso geral ($n > 0$)

- Calculamos $P_{A'}(x)$ recursivamente
- Somamos $a_n x^n$ ao resultado obtido

Algoritmo Cálculo-Polinômio(A, x)

UFABC

1: **Função** Cálculo-Polinômio(A, n)
2: **Se** $n = 0$ **então**
3: $y \leftarrow a_0$
4: **Senão**
5: $A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
6: $y' \leftarrow$ Cálculo-Polinômio(A', x)
7: $xn \leftarrow 1$
8: **Para** $i \leftarrow 1$ até n **faça**
9: $xn \leftarrow xn \cdot x$
10: $y \leftarrow y' + a_n \cdot xn$
11: **Devolve** y

UFABC

Correção de Algoritmos Recursivos: prova por indução

UFABC

Correção de Algoritmos Recursivos: prova por indução

Teorema

Cálculo-Polinômio resolve corretamente o problema do Valor do Polinômio

UFABC

Correção de Algoritmos Recursivos: prova por indução

Teorema

Cálculo-Polinômio resolve corretamente o problema do Valor do Polinômio

Demonstração

Seja $A = a_n, a_{n-1}, \ldots, a_1, a_0$ a sequência de números coeficientes do polinômio dada na entrada e seja x o número dado na entrada.

A prova segue por indução em n.

UFABC

Base ($n = 0$)

- Quando $n = 0$, então $P_A(x) = a_0$.

UFABC

Base ($n = 0$)

- Quando $n = 0$, então $P_A(x) = a_0$.
- Note que esse é exatamente o valor retornado por Cálculo-Polinômio, pois, quando $n = 0$, o teste da linha 1 dá verdadeiro, o que leva o algoritmo a retornar $y = a_0$ (linhas 2 e 10).

UFABC

Passo ($P(n-1) \Rightarrow P(n)$)

UFABC

Passo ($P(n-1) \Rightarrow P(n)$)

Agora suponha que $n > 0$.

UFABC

Passo ($P(n-1) \Rightarrow P(n)$)

Agora suponha que $n > 0$.

Então o teste da linha 1 falha e o algoritmo executa a linha 4 fazendo

$$A' = a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \tag{1}$$

UFABC

Passo ($P(n-1) \Rightarrow P(n)$)

Agora suponha que $n > 0$.

Então o teste da linha 1 falha e o algoritmo executa a linha 4 fazendo

$$A' = a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \tag{1}$$

Na linha 4 fazemos

$$y' = \text{Cálculo-Polinômio}(A', x) \tag{2}$$

UFABC

Passo ($P(n-1) \Rightarrow P(n)$)

Agora suponha que $n > 0$.

Então o teste da linha 1 falha e o algoritmo executa a linha 4 fazendo

$$A' = a_{n-1}, a_{n-2}, \ldots, a_1, a_0 \tag{1}$$

Na linha 4 fazemos

$$y' = \text{Cálculo-Polinômio}(A', x) \tag{2}$$

Note que A' tem $n-1$ elementos, então, por hipótese de indução, Cálculo-Polinômio(A', x) resolve corretamente o problema e, portanto, temos que

$$y' = P_{A'}(x) \tag{3}$$

Note que

$$P_A(x) = a_n x^n + P_{A'}(x) \tag{4}$$

Note que

$$P_A(x) = a_n x^n + P_{A'}(x) \tag{4}$$

É fácil perceber (exercício) que o laço da linha 7 para e que ao final da execução temos que

$$xn = x^n \tag{5}$$

Correção do Cálculo-Polinômio(A, x)

UFABC

Note que

$$P_A(x) = a_n x^n + P_{A'}(x) \tag{4}$$

É fácil perceber (exercício) que o laço da linha 7 para e que ao final da execução temos que

$$xn = x^n \tag{5}$$

Ao termino do laço da linha 7, o algoritmo faz

$$y = y' + a_n xn \tag{6}$$

Por (5), (3), (4), temos que

$$P_A(x) = a_n x^n + P_{A'}(x) = a_n xn + y' = y \tag{7}$$

Na sequência, linha 10, o algoritmo retorna y, que é o valor correto por (??).

Análise de Tempo de Execução

UFABC

```
1:  Função Cálculo-Polinômio(A, n)
2:      Se n = 0 então
3:          y ← a_0
4:      Senão
5:          A' ← a_{n-1}, ..., a_1, a_0
6:          y' ← Cálculo-Polinômio(A', x)
7:          xn ← 1
8:          Para i ← 1 até n faça
9:              xn ← xn · x
10:         y ← y' + a_n · xn
11:     Devolve y
```

UFABC

$$T(n) = \begin{cases} \Theta(1), \text{ se } n = 0 \\ T(n-1) + \Theta(n), \text{ se } n > 0 \end{cases}$$

UFABC

$$T(n) = \begin{cases} \Theta(1), \text{ se } n = 0 \\ T(n-1) + \Theta(n), \text{ se } n > 0 \end{cases}$$

Atualmente isso não diz muita coisa, nas próximas aulas vamos aprender a resolver tal recorrência e obtermos que

$$T(n) = \Theta(n^2)$$

Melhorando Cálculo-Polinômio

UFABC

- No algoritmo Cálculo-Polinômio, recalculamos potência de x
- Podemos reaproveitar o cálculo de x^{n-1}
- Para isso, reforçamos a hipótese de indução

Melhorando Cálculo-Polinômio

UFABC

- No algoritmo Cálculo-Polinômio, recalculamos potência de x
- Podemos reaproveitar o cálculo de x^{n-1}
- Para isso, reforçamos a hipótese de indução

Hipótese de Indução Reforçada

Sabemos calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ e também o valor de x^{n-1}

Melhorando Cálculo-Polinômio

UFABC

- No algoritmo Cálculo-Polinômio, recalculamos potência de x
- Podemos reaproveitar o cálculo de x^{n-1}
- Para isso, reforçamos a hipótese de indução

Hipótese de Indução Reforçada

Sabemos calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ e também o valor de x^{n-1}

Podemos fazer uma hipóteses mais fortes sobre a recursão

- mas agora também precisamos retornar o valor de x^n

Algoritmo Cálculo-Polinômio2(A, x)

UFABC

1: **Função** Cálculo-Polinômio2(A, n)
2: **Se** $n = 0$ **então**
3: $y \leftarrow a_0$
4: $xn \leftarrow 1$
5: **Senão**
6: $A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
7: $y', x' \leftarrow$ Cálculo-Polinômio(A', x)
8: $xn \leftarrow x \cdot x'$
9: $y \leftarrow y' + a_n \cdot xn$
10: **Devolve** y, xn

Algoritmo Cálculo-Polinômio2(A, x)

UFABC

Teorema

Cálculo-Polinômio2 recebe uma sequência de reais $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e um real x e devolve $P_A(x)$ e x^n.

Algoritmo Cálculo-Polinômio2(A, x)

UFABC

Teorema

Cálculo-Polinômio2 recebe uma sequência de reais $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e um real x e devolve $P_A(x)$ e x^n.

Prova de correção muito similar a do Algoritmo Cálculo-Polinômio (exercício)

Análise de Tempo do Algoritmo Cálculo-Polinômio2(A, x)

UFABC

1: **Função** Cálculo-Polinômio2(A, n)
2: **Se** $n = 0$ **então**
3: $y \leftarrow a_0$
4: $xn \leftarrow 1$
5: **Senão**
6: $A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
7: $y', x' \leftarrow$ Cálculo-Polinômio(A', x)
8: $xn \leftarrow x \cdot x'$
9: $y \leftarrow y' + a_n \cdot xn$
10: **Devolve** y, xn

Análise de Tempo do Algoritmo Cálculo-Polinômio2(A, x)

UFABC

$$T(n) = \begin{cases} \Theta(1), \text{ se } n = 0 \\ T(n-1) + \Theta(1), \text{ se } n > 0 \end{cases}$$

Problema: Árvores Binárias

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Vamos relembrar árvore binária de busca

- cada nó v tem uma chave
- a subárvore esquerda tem chaves menores
- a subárvore direita tem chave maiores

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Vamos relembrar árvore binária de busca

- cada nó v tem uma chave
- a subárvore esquerda tem chaves menores
- a subárvore direita tem chave maiores

a altura de um nó v é o maior número de arestas de v até uma nó folha na subárvore enraizada em v

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Vamos relembrar árvore binária de busca

- cada nó v tem uma chave
- a subárvore esquerda tem chaves menores
- a subárvore direita tem chave maiores

a altura de um nó v é o maior número de arestas de v até uma nó folha na subárvore enraizada em v

Definição

O fator de balanceamento (f.b.) de um nó v é a diferença entre as alturas da subárvore esquerda e direita. Uma árvore binária de busca é balanceada se todo nó v tem f.b. limitado

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Vamos relembrar árvore binária de busca

- cada nó v tem uma chave
- a subárvore esquerda tem chaves menores
- a subárvore direita tem chave maiores

a altura de um nó v é o maior número de arestas de v até uma nó folha na subárvore enraizada em v

Definição

O fator de balanceamento (f.b.) de um nó v é a diferença entre as alturas da subárvore esquerda e direita. Uma árvore binária de busca é balanceada se todo nó v tem f.b. limitado

- e.g., em uma árvore AVL, todo nó tem f.b. -1, 0, ou $+1$

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema: Fator de Balanceamento

Entrada: uma árvore binária T com n nós

Saída: o fator de balanceamento de cada nó T

UFABC

Problema: Fator de Balanceamento

Entrada: uma árvore binária T com n nós

Saída: o fator de balanceamento de cada nó T

Vamos projetar o algoritmo indutivamente

Hipótese de Indução

Sabemos como calcular fatores de balanceamento de árvores com menos de n nós.

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva
- o f.b. depende das alturas das subárvores

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva
- o f.b. depende das alturas das subárvores
  - mas sabemos apenas o f.b.!

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva
- o f.b. depende das alturas das subárvores
  - mas sabemos apenas o f.b.!
- conclusão: precisamos de um h.i. mais forte

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva
- o f.b. depende das alturas das subárvores
  - mas sabemos apenas o f.b.!
- conclusão: precisamos de um h.i. mais forte

Problema Fator de Balanceamento em Árvores Binárias

UFABC

Problema:

- A definição de f.b. não é recursiva
- o f.b. depende das alturas das subárvores
  - mas sabemos apenas o f.b.!
- conclusão: precisamos de um h.i. mais forte

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0
2. Se $n > 1$

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0
2. Se $n > 1$
   - recursivamente, obtemos a altura da árvore esquerda h_e

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0
2. Se $n > 1$
   - recursivamente, obtemos a altura da árvore esquerda h_e
   - do mesmo modo, obtemos a altura da árvore direita h_d

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0
2. Se $n > 1$
   - recursivamente, obtemos a altura da árvore esquerda h_e
   - do mesmo modo, obtemos a altura da árvore direita h_d
   - por definição, f.b. é $h_e - h_d$

UFABC

Nova Hipótese de Indução

Sabemos como calcular fatores de balanceamento e alturas de árvores com menos de n nós.

1. Se $n = 1$
   - f.b. é igual a 0
   - a altura é 0
2. Se $n > 1$
   - recursivamente, obtemos a altura da árvore esquerda h_e
   - do mesmo modo, obtemos a altura da árvore direita h_d
   - por definição, f.b. é $h_e - h_d$
   - a altura é $max(h_e, h_d) + 1$

Algoritmo Fator de Balanceamento em Árvores Binárias

UFABC

1: **Função** FATOR-ALTURA(A, n)
2: **Se** $n = 0$ **então**
3: $f \leftarrow 0$
4: $h \leftarrow 0$
5: **Senão**
6: $f_e, h_e \leftarrow$ Fator-Altura(A_e, n_e)
7: $f_d, h_d \leftarrow$ Fator-Altura(A_d, n_d)
8: $f \leftarrow f_e - f_d$
9: $h \leftarrow max(h_e, h_d) + 1$
10: **Devolve** y, xn

Divisão e Conquista

Divisão e Conquista

UFABC

- É uma técnica de projeto de algoritmos que envolve recursão
- A ideia é dividir a instância do problema em duas o mais instâncias menores (divisão), resolvê-los de forma recursiva e combinar as soluções em uma solução da instância original (conquista)

Problema de Ordenação

UFABC

Problema: Ordenação

Entrada: $< A, n >$, onde $A = (a_1, a_2, \ldots, a_n)$ é um vetor com n números

Saída: permutação $(a'_1, a'_2, \ldots, a'_n)$ dos elementos de A tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

Problema de Ordenação

UFABC

Problema: Ordenação

Entrada: $< A, n >$, onde $A = (a_1, a_2, \ldots, a_n)$ é um vetor com n números

Saída: permutação $(a'_1, a'_2, \ldots, a'_n)$ dos elementos de A tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- Vimos que o InsertionSort resolve esse problema em $O(n^2)$

Problema de Ordenação

UFABC

Problema: Ordenação

Entrada: $< A, n >$, onde $A = (a_1, a_2, \ldots, a_n)$ é um vetor com n números

Saída: permutação $(a'_1, a'_2, \ldots, a'_n)$ dos elementos de A tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- Vimos que o InsertionSort resolve esse problema em $O(n^2)$
- Vamos tentar fazer um algoritmo recursivo para esse problema

Problema de Ordenação

UFABC

Problema: Ordenação

Entrada: $< A, n >$, onde $A = (a_1, a_2, \ldots, a_n)$ é um vetor com n números

Saída: permutação $(a'_1, a'_2, \ldots, a'_n)$ dos elementos de A tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- Vimos que o InsertionSort resolve esse problema em $O(n^2)$
- Vamos tentar fazer um algoritmo recursivo para esse problema

Problema de Ordenação

UFABC

Problema: Ordenação

Entrada: $< A, n >$, onde $A = (a_1, a_2, \ldots, a_n)$ é um vetor com n números

Saída: permutação $(a'_1, a'_2, \ldots, a'_n)$ dos elementos de A tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- Vimos que o InsertionSort resolve esse problema em $O(n^2)$
- Vamos tentar fazer um algoritmo recursivo para esse problema

Hipóte de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

UFABC

Hipótese de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

UFABC

Hipótese de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

Caso geral/passo ($n > 1$)

- Recursivamente, ordenamos $A[1..n-1]$

UFABC

Hipótese de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

Caso geral/passo ($n > 1$)

- Recursivamente, ordenamos $A[1..n-1]$
- Reorganizamos A para colocar $A[n]$ no lugar correto

Problema de Ordenação

UFABC

```
1:  Função ORDENA(A, n)
2:      Se n = 0 então
3:          Devolve
4:      Ordena(A, n − 1)
5:      atual ← A[n]
6:      j ← n − 1
7:      Enquanto j > 0 e A[j] > atual faça
8:          A[j + 1] ← A[j]
9:          j ← j − 1
10:     A[j + 1] ← atual
```

Problema de Ordenação

UFABC

```
1:  Função ORDENA(A, n)
2:      Se n = 0 então
3:          Devolve
4:      Ordena(A, n − 1)
5:      atual ← A[n]
6:      j ← n − 1
7:      Enquanto j > 0 e A[j] > atual faça
8:          A[j + 1] ← A[j]
9:          j ← j − 1
10:     A[j + 1] ← atual
```

Isso é um ”InsertionSort” no qual o laço externo foi trocado por recursão

Problema de Ordenação: Nova tentativa

UFABC

Hipóte de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

Problema de Ordenação: Nova tentativa

UFABC

Hipóte de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

Caso geral/passo ($n > 1$)

- Recursivamente, ordenamos $A[1..\lfloor n/2 \rfloor]$ e $A[\lfloor n/2 \rfloor + 1..n]$
- Combinamos os dois subvetores ordenados

Problema de Ordenação: Nova tentativa

UFABC

Hipóte de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos

Caso base ($n = 1$)

- $A[1..1]$ está trivialmente ordenado

Caso geral/passo ($n > 1$)

- Recursivamente, ordenamos $A[1..\lfloor n/2 \rfloor]$ e $A[\lfloor n/2 \rfloor + 1..n]$
- Combinamos os dois subvetores ordenados

Esse algoritmo é conhecido com **MergeSort**

”MergeSort”

UFABC

1: **Função** ”MergeSort”(A, n)
2: **Se** $n = 1$ **então**
3: **Devolve**
4: MergeSort$(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)$
5: MergeSort$(A[\lfloor n/2 \rfloor + 1], \lceil n/2 \rceil)$
6: Combina$(A, \lfloor n/2 \rfloor)$

MergeSort

UFABC

1: **Função** MERGESORT(A, ini, fim)
2: **Se** $ini < fim$ **então**
3: $meio \leftarrow \lfloor (ini + fim)/2 \rfloor$
4: MergeSort($A, ini, meio$)
5: MergeSort($A, meio + 1, fim$)
6: Combina($A, ini, fim, meio$)

Combina

```
1:  Função COMBINA(A, ini, fim, meio)
2:      n_1 ← meio − ini + 1
3:      n_2 ← fim − meio
4:      Sejam E[1..n_1 + 1] e D[1..n_2 + 1] dois vetores de números
5:      Para i = 1 até n_1 faça
6:          E[i] = A[ini + i − 1]
7:      Para i = 1 até n_2 faça
8:          D[i] = A[meio + i]
9:      E[n_1 + 1] ← ∞
10:     D[n_2 + 1] ← ∞
11:     i ← 1
12:     j ← 1
13:     Para k = ini até fim faça
14:         Se E[i] ≤ D[j] então
15:             A[k] = E[i]
16:             i ← i + 1
17:         Senão
18:             A[k] = D[j]
19:             j ← j + 1
```

Análise do Tempo de Execução do Combina

UFABC

- Para este algoritmo temos que o tamanho da entrada n é $n = fim - ini + 1$

Análise do Tempo de Execução do Combina

UFABC

- Para este algoritmo temos que o tamanho da entrada n é $n = fim - ini + 1$
- O bloco A leva um tempo constante para executar e é executado apenas uma vez, logo o bloco A leva $\Theta(1)$ para executar.

Análise do Tempo de Execução do Combina

UFABC

- Para este algoritmo temos que o tamanho da entrada n é $n = fim - ini + 1$
- O bloco A leva um tempo constante para executar e é executado apenas uma vez, logo o bloco A leva $\Theta(1)$ para executar.
- Note que no bloco B cada entrada de $A[ini..fim]$ é acessada para ser copiada para um dos vetores E ou D. Portanto, o trecho B leva tempo $\Theta(fim - ini + 1) = \Theta(n)$.

Análise do Tempo de Execução do Combina

UFABC

- Para este algoritmo temos que o tamanho da entrada n é $n = fim - ini + 1$
- O bloco A leva um tempo constante para executar e é executado apenas uma vez, logo o bloco A leva $\Theta(1)$ para executar.
- Note que no bloco B cada entrada de $A[ini..fim]$ é acessada para ser copiada para um dos vetores E ou D. Portanto, o trecho B leva tempo $\Theta(fim - ini + 1) = \Theta(n)$.
- O trecho C possui apenas instruções simples e pode ser executado em $\Theta(1)$.

Análise do Tempo de Execução do Combina

UFABC

- Para este algoritmo temos que o tamanho da entrada n é $n = fim - ini + 1$
- O bloco A leva um tempo constante para executar e é executado apenas uma vez, logo o bloco A leva $\Theta(1)$ para executar.
- Note que no bloco B cada entrada de $A[ini..fim]$ é acessada para ser copiada para um dos vetores E ou D. Portanto, o trecho B leva tempo $\Theta(fim - ini + 1) = \Theta(n)$.
- O trecho C possui apenas instruções simples e pode ser executado em $\Theta(1)$.
- Cada iteração do bloco D acessa uma entrada diferente de $A[ini..fim]$. Ademais, cada iteração leva um tempo constante, logo o tempo de D é $\Theta(fim - ini + 1) = \Theta(n)$.

Análise do Tempo de Execução do Combina

UFABC

- Assim, o tempo de execução de Combina é

$$T(n) = \Theta(1) + \Theta(n) + \Theta(1) + \Theta(n) = \Theta(n)$$

Análise do Tempo de Execução do MergeSort

UFABC

1: **Função** MERGESORT(A, ini, fim)
2: **Se** $ini < fim$ **então**
3: $meio \leftarrow \lfloor (ini + fim)/2 \rfloor$
4: MergeSort($A, ini, meio$)
5: MergeSort($A, meio + 1, fim$)
6: Combina($A, ini, fim, meio$)

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.
- A chamada da linha 3 gasta $\Theta(meio - ini + 1) = \Theta(\lceil n/2 \rceil)$

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.
- A chamada da linha 3 gasta $\Theta(meio - ini + 1) = \Theta(\lceil n/2 \rceil)$
- Já a da linha 3 gasta $\Theta(fim - meio) = \Theta(\lfloor n/2 \rfloor)$

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.
- A chamada da linha 3 gasta $\Theta(meio - ini + 1) = \Theta(\lceil n/2 \rceil)$
- Já a da linha 3 gasta $\Theta(fim - meio) = \Theta(\lfloor n/2 \rfloor)$
- Portanto, o tempo de MergeSort é

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.
- A chamada da linha 3 gasta $\Theta(meio - ini + 1) = \Theta(\lceil n/2 \rceil)$
- Já a da linha 3 gasta $\Theta(fim - meio) = \Theta(\lfloor n/2 \rfloor)$
- Portanto, o tempo de MergeSort é

Análise do Tempo de Execução do MergeSort

UFABC

- Quando $ini \geq fim$, o custo de MergeSort é $\Theta(1)$
- Quando $ini < fim$, então as linhas 1 e 2 pagam $\Theta(1)$
- A linha 5 gasta $\Theta(n)$.
- A chamada da linha 3 gasta $\Theta(meio - ini + 1) = \Theta(\lceil n/2 \rceil)$
- Já a da linha 3 gasta $\Theta(fim - meio) = \Theta(\lfloor n/2 \rfloor)$
- Portanto, o tempo de MergeSort é

$$T(n) = \begin{cases} \Theta(1), \text{ se n = 1} e \text{n = 0} \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n), \text{ se } n > 1 \end{cases}$$

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Precisamos primeiro demonstrar a correção de Combina

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Precisamos primeiro demonstrar a correção de Combina

Lemma 1 Se $A[ini..meio]$ e $A[meio + 1..fim]$ estão ordenados, então, após a execução de Combina($A, ini, fim, meio$), temos que $A[ini..fim]$ está ordenado.

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Precisamos primeiro demonstrar a correção de Combina

Lemma 1 Se $A[ini..meio]$ e $A[meio + 1..fim]$ estão ordenados, então, após a execução de Combina($A, ini, fim, meio$), temos que $A[ini..fim]$ está ordenado.

- Combinada é um algoritmo iterativo, portanto sua demonstração é análoga às vistas anteriormente (**exercício**)

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Demonstração

A prova segue por indução em $n = fim - ini + 1$

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Demonstração

A prova segue por indução em $n = fim - ini + 1$

Base ($n = 0$ ou $n = 1$)

- Se $n = 0$, então $fim = ini - 1$.

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Demonstração

A prova segue por indução em $n = fim - ini + 1$

Base ($n = 0$ ou $n = 1$)

- Se $n = 0$, então $fim = ini - 1$.
- Assim $A[ini..fim] = A[ini..ini - 1] = \emptyset$, e o resultado segue por vacuidade

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Demonstração

A prova segue por indução em $n = fim - ini + 1$

Base ($n = 0$ ou $n = 1$)

- Se $n = 0$, então $fim = ini - 1$.
- Assim $A[ini..fim] = A[ini..ini - 1] = \emptyset$, e o resultado segue por vacuidade
- Se $n = 1$, então $fim = ini$.

Correção de MergeSort

UFABC

Teo MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$

Demonstração

A prova segue por indução em $n = fim - ini + 1$

Base ($n = 0$ ou $n = 1$)

- Se $n = 0$, então $fim = ini - 1$.
- Assim $A[ini..fim] = A[ini..ini - 1] = \emptyset$, e o resultado segue por vacuidade
- Se $n = 1$, então $fim = ini$.
- Assim, $A[ini..fim] = A[ini..ini] = A[ini]$ e, portanto, está trivialmente ordenado.

UFABC

Passo ($P(k) \Rightarrow P(n), \forall k < n$)

- Agora suponha que $n > 1$ e suponha que MergeSort ordena vetores quando o tamanho k ($k = fim' - ini' + 1$) é menor que n.

UFABC

Passo ($P(k) \Rightarrow P(n), \forall k < n$)

- Agora suponha que $n > 1$ e suponha que MergeSort ordena vetores quando o tamanho k ($k = fim' - ini' + 1$) é menor que n.
- Como $n > 1$, temos que $fim \geq ini$.

UFABC

Passo ($P(k) \Rightarrow P(n), \forall k < n$)

- Agora suponha que $n > 1$ e suponha que MergeSort ordena vetores quando o tamanho k ($k = fim' - ini' + 1$) é menor que n.
- Como $n > 1$, temos que $fim \geq ini$.
- Neste caso o teste da linha 1 da verdadeiro

UFABC

Passo ($P(k) \Rightarrow P(n), \forall k < n$)

- Agora suponha que $n > 1$ e suponha que MergeSort ordena vetores quando o tamanho k ($k = fim' - ini' + 1$) é menor que n.
- Como $n > 1$, temos que $fim \geq ini$.
- Neste caso o teste da linha 1 da verdadeiro
- Na linha 2, o algoritmo faz $meio = \lfloor (fim + ini)/2 \rfloor$

UFABC

Passo ($P(k) \Rightarrow P(n), \forall k < n$)

- Agora suponha que $n > 1$ e suponha que MergeSort ordena vetores quando o tamanho k ($k = fim' - ini' + 1$) é menor que n.
- Como $n > 1$, temos que $fim \geq ini$.
- Neste caso o teste da linha 1 da verdadeiro
- Na linha 2, o algoritmo faz $meio = \lfloor (fim + ini)/2 \rfloor$
- Na sequência o algoritmo faz duas chamadas recursivas. A primeira é sobre $A[ini..meio]$.

Correção de MergeSort

UFABC

- Note que $meio - ini + 1 = \lfloor (fim + ini)/2 - ini + 1 < n$, quando $n > 1$. Então essa chamada, por hipótese de indução, funciona.

Correção de MergeSort

UFABC

- Note que $meio - ini + 1 = \lfloor (fim + ini)/2 - ini + 1 < n$, quando $n > 1$. Então essa chamada, por hipótese de indução, funciona.
- Como $fim - (meio + 1) + 1 < n$, a segunda chamada também funciona.

Correção de MergeSort

UFABC

- Note que $meio - ini + 1 = \lfloor (fim + ini)/2 - ini + 1 < n$, quando $n > 1$. Então essa chamada, por hipótese de indução, funciona.
- Como $fim - (meio + 1) + 1 < n$, a segunda chamada também funciona.
- Assim $A[ini..meio]$ e $A[meio + 1..fim]$ estão ordenados

Correção de MergeSort

UFABC

- Note que $meio - ini + 1 = \lfloor (fim + ini)/2 - ini + 1 < n$, quando $n > 1$. Então essa chamada, por hipótese de indução, funciona.
- Como $fim - (meio + 1) + 1 < n$, a segunda chamada também funciona.
- Assim $A[ini..meio]$ e $A[meio + 1..fim]$ estão ordenados
- Pelo Lemma 1, após a execução de Combina($A, ini, fim, meio$), temos que $A[ini..fim]$ está ordenado $\square$