Análise do Mergesort

Prof. Daniel Kikuti

Universidade Estadual de Maringá

Objetivos desta aula

- Revisar indução matemática.
- Como usar indução para mostrar a correção de algoritmos recursivos.
- Apresentar a técnica de projeto de algoritmos: **Divisão e Conquista**.
- Análise do MERGESORT.
  - Correção.
  - Complexidade (Equações de recorrência).
- Exercícios.

Fundamentação teórica

Princípio da Indução Matemática

Seja $p(n)$ uma sentença aberta sobre o conjunto $\mathbb{N}$ (naturais). Se

1. $p(n_0)$ é verdadeira, e
2. $p(k)$ é verdadeira $\Rightarrow p(k+1)$ é verdadeira , $\forall k \in \mathbb{N}$, $k \geq n_0$;

então $p(n)$ é verdadeira para todo $n \in \mathbb{N}$, com $n \geq n_0$.

Fundamentação teórica

Princípio da Indução Matemática

Seja $p(n)$ uma sentença aberta sobre o conjunto $\mathbb{N}$ (naturais). Se

1. $p(n_0)$ é verdadeira, e
2. $p(k)$ é verdadeira $\Rightarrow p(k+1)$ é verdadeira , $\forall k \in \mathbb{N}$, $k \geq n_0$;

então $p(n)$ é verdadeira para todo $n \in \mathbb{N}$, com $n \geq n_0$.

Como provar que $\forall (n \geq n_0) \in \mathbb{N}, p(n)$?

- **Base da indução:** provar que $p(n_0)$ é verdadeira;
- **Hipótese da indução:** supor que para algum $k \in \mathbb{N}$, $p(k)$ é verdadeira;
- **Passo da indução:** provar que $p(k+1)$ é verdadeira.

Exemplo: Mostre que

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \forall n \in \mathbb{N}$$

Fundamentação Teórica

Indução forte

Seja $n_0 \in \mathbb{N}$ e $p(n)$ uma sentença aberta sobre o conjunto $\mathbb{N}$. Se

- $p(n_0)$ é verdadeira;
- $p(n_0)$ e $p(n_0 + 1)$ e ... e $p(k)$ verdadeiras $\implies p(k+1)$ é verdadeira, $\forall k \geq n_0$;

então $p(n)$ é verdadeira para todo $n \in \mathbb{N}$, com $n \geq n_0$.

Casos em que é necessário provar **mais de uma base** e que o passo indutivo não depende apenas do k anterior.

Correção de algoritmos recursivos

O problema

Dado um conjunto de n elementos no vetor A, encontre o máximo.

$\textsc{Maximo}$(A[],n)

if $n = 1$ **then**
  return $A[1]$
return $max(\textsc{Maximo}(A[], n-1), A[n])$

Como provar que o algoritmo está correto?

Correção de algoritmos recursivos

Teorema

Para todo $n \geq 1$, $\textsc{Maximo}$(A[], n) devolve o maior elemento de $\{A[1], A[2], \ldots, A[n]\}$.

Correção de algoritmos recursivos

Teorema

Para todo $n \geq 1$, $\textsc{Maximo}$(A[], n) devolve o maior elemento de $\{A[1], A[2], \ldots, A[n]\}$.

- **Base:** Para $n = 1$, $\textsc{Maximo}$(A[], n) devolve $A[1]$ (máximo do vetor com um elemento).

Correção de algoritmos recursivos

Teorema

Para todo $n \geq 1$, $\textsc{Maximo}$(A[], n) devolve o maior elemento de $\{A[1], A[2], \ldots, A[n]\}$.

- **Base:** Para $n = 1$, $\textsc{Maximo}$(A[], n) devolve $A[1]$ (máximo do vetor com um elemento).
- **Hipótese:** Para $n \geq 1$, $\textsc{maximo}$(A[], n) devolve $max\{A[1], A[2], \ldots, A[n]\}$.

Correção de algoritmos recursivos

Teorema

Para todo $n \geq 1$, $\textsc{Maximo}$(A[], n) devolve o maior elemento de $\{A[1], A[2], \ldots, A[n]\}$.

- **Base:** Para $n = 1$, $\textsc{Maximo}$(A[], n) devolve $A[1]$ (máximo do vetor com um elemento).
- **Hipótese:** Para $n \geq 1$, $\textsc{maximo}$(A[], n) devolve $max\{A[1], A[2], \ldots, A[n]\}$.
- **Passo:** Queremos mostrar que $\textsc{maximo}$(A[], n+1) devolve $max\{A[1], A[2], \ldots, A[n], A[n+1]\}$.
  O algoritmo $\textsc{maximo}$(A[], n+1) devolve:
  $max(\textsc{maximo}(A[], n), A[n+1]) =$ (usando a hipótese)
  $max(max\{A[1], A[2], \ldots, A[n]\}, A[n+1]) =$
  $max\{A[1], A[2], \ldots, A[n+1]\}$

Sua vez

O problema

O n-ésimo número de Fibonacci F_n é definido como:

$$F_n = \begin{cases} 0, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ F_{n-1} + F_{n-2}, & \text{se } n > 1. \end{cases}$$

$\textsc{Fib}(\mathsf{n})$

if $n = 0$ **then return** 0
if $n = 1$ **then return** 1
return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

Prove que o algoritmo está correto.

Sua vez

Teorema

Para todo $n \geq 0$, $\textsc{Fib}(n)$ devolve F_n.

Sua vez

Teorema

Para todo $n \geq 0$, $\textsc{Fib}(n)$ devolve F_n.

- **Base:** Para $n = 0$ e $n = 1$, $\textsc{fib}(n)$ devolve os valores corretos de F_n.

Sua vez

Teorema

Para todo $n \geq 0$, $\textsc{Fib}(n)$ devolve F_n.

- **Base:** Para $n = 0$ e $n = 1$, $\textsc{fib}(n)$ devolve os valores corretos de F_n.
- **Hipótese:** Para $n \geq 2$ e para todo $0 \leq k \leq n$, $\textsc{Fib}(k)$ devolve corretamente F_k.

Sua vez

Teorema

Para todo $n \geq 0$, $\textsc{Fib}(n)$ devolve F_n.

- **Base:** Para $n = 0$ e $n = 1$, $\textsc{fib}(n)$ devolve os valores corretos de F_n.
- **Hipótese:** Para $n \geq 2$ e para todo $0 \leq k \leq n$, $\textsc{Fib}(k)$ devolve corretamente F_k.
- **Passo:** Queremos mostrar que $\textsc{Fib}(k+1)$ devolve F_{k+1}.
  O algoritmo $\textsc{fib}(k+1)$ devolve:
  $\textsc{fib}((k+1)-1) + \textsc{fib}((k+1)-2) = \textsc{fib}(k) + \textsc{fib}(k-1) =$ (usando a hipótese) $F_k + F_{k-1} =$ (pela definição) F_{k+1}.

Técnica de projeto de algoritmos

Divisão e conquista (visão geral)

- **Dividir** o problema em um número de subproblemas
- **Conquistar** os subproblemas resolvendo-os recursivamente:
  - Caso base: Se os subproblemas forem suficientemente pequenos, resolvê-los de maneira direta.
- **Combinar** as soluções dos subproblemas para obter a solução do problema original.

O problema de ordenação

Entrada

Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.

Saída

Uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

O problema de ordenação

Entrada

Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.

Saída

Uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Usando divisão e conquista

- Dividir o problema em dois subvetores: `A[início..meio]` e `A[meio+1..fim]`.
- Conquistar ordenando recursivamente os dois subvetores.
- Combinar pela intercalação os dois subvetores `A[início..meio]` e `A[meio+1..fim]` e produzir ordenado `A[início..fim]`.

Exemplo

Considere $A = \{5, 2, 4, 7, 1, 3, 2, 6\}$.

O algoritmo

Visão geral

- A divisão é feita no procedimento MERGESORT.
- Caso base: subvetor com um elemento (ordenado).
- A intercalação é feita pelo procedimento MERGE.
- A chamada inicial MERGESORT(A, 1, N).

MERGESORT($A[]$, *inicio*, *fim*)

if $inicio < fim$ **then**
  $meio \leftarrow \lfloor (inicio + fim)/2 \rfloor$
  MERGESORT(A, *inicio*, *meio*)
  MERGESORT(A, $meio + 1$, *fim*)
  MERGE(A, *inicio*, *meio*, *fim*)

O algoritmo Merge

```
Merge(A[], inicio, meio, fim)
tam ← fim − inicio + 1
p1 ← inicio
p2 ← meio + 1
for i ← 1 to tam do
    if p1 ≤ meio e p2 ≤ fim then
        if A[p1] < A[p2] then
            temp[i] ← A[p1] e p1 ← p1 + 1
        else
            temp[i] ← A[p2] e p2 ← p2 + 1
    else if p1 ≤ meio then
        temp[i] ← A[p1] e p1 ← p1 + 1
    else temp[i] ← A[p2] e p2 ← p2 + 1
Copie os valores de temp para A
```

Correção do procedimento merge

Invariante

- No início de cada iteração do laço **for** o subvetor `temp[1..i-1]` contém os $i-1$ menores elementos de $A[p1..meio]$ e $A[p2..fim]$, em sequência ordenada.
- Além disso, $A[p1]$ e $A[p2]$ são os menores elementos de seus vetores que não foram copiados.

Inicialização

- Para $i = 1$ o vetor `temp[1..i-1]` está vazio.
- Considerando que os subvetores $A[p1..meio]$ e $A[p2..fim]$ estão ordenados, `p1 = inicio` e `p2 = meio+1`, então podemos afirmar que $A[p1]$ e $A[p2]$ são os menores valores que não foram copiados para `temp[1..i-1]`.
- Portanto, o invariante é válido.

Correção do procedimento merge (continuação)

Manutenção

- O invariante é válido no início de uma iteração qualquer.
- Queremos mostrar que para a **próxima iteração**, `temp[1..i]` contém os i menores elementos de $A[p1..meio]$ e $A[p2..fim]$, em sequência ordenada. Além disso, ou $A[p1+1]$ e $A[p2]$, ou $A[p1]$ e $A[p2+1]$ são os menores elementos ainda não copiados para `temp`.
- Considerando o **if** da linha 5 como verdadeiro (senão todos os elementos de um dos vetores já foram copiados) e supondo que $A[p1] < A[p2]$, então $A[p1]$ é o menor elemento ainda não copiado para `temp`. A linha 7 copia-o para `temp` e atualiza o valor de `p1`. Assim, `temp[1..i]` contém os i menores elementos de $A[p1..meio]$ e $A[p2..fim]$ em ordem e; $A[p1+1]$ e $A[p2]$ são os menores elementos ainda não copiados (o caso em que $A[p1] \geq A[p2]$ é tratado analogamente na linha 8).
- Portanto, o invariante se mantém para a proxima iteração.

Correção do procedimento MERGE (continuação)

Término

- O algoritmo termina com i = n+1 (tam+1).
- Assim, `temp[1..n+1-1]` = `temp[1..n]` contém os n menores elementos dos vetores `A[p1..meio]` e `A[p2..fim]`, em sequência ordenada.

Análise de complexidade do Merge

- As linhas 1–3 executam uma única vez e consomem tempo constante.
- A linha 4 executará n vezes.
- As linhas 5–11 consomem $n - 1$ vezes uma constante no total (verifique).
- A linha 12 esconde um laço que consome tempo linear (n).
- O procedimento Merge tem **complexidade linear**.

Análise de complexidade de algoritmos de divisão e conquista

- Uso de **equação de recorrência**.
- $T(n)$ representa o tempo de execução de um problema de tamanho n.
- a: quantidade de subproblemas.
- $1/b$: tamanho do subproblema.
- $D(n)$: tempo para dividir o problema.
- $C(n)$: tempo para combinar as soluções.

$$T(n) = \begin{cases} c & \text{caso base} \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

Análise de complexidade do MergeSort

$\textsc{Mergesort}(A[], \mathit{inicio}, \mathit{fim})$

```
if inicio < fim then
    meio ← ⌊(inicio + fim)/2⌋
    MERGESORT(A, inicio, meio)
    MERGESORT(A, meio + 1, fim)
    MERGE(A, inicio, meio, fim)
```

Análise de complexidade do $\textsc{MergeSort}$

$\textsc{Mergesort}(A[], \textit{inicio}, \textit{fim})$

```
if inicio < fim then
    meio ← ⌊(inicio + fim)/2⌋
    MERGESORT(A, inicio, meio)
    MERGESORT(A, meio + 1, fim)
    MERGE(A, inicio, meio, fim)
```

- Caso base ocorre quando $n = 1$.
- Quando $n \geq 2$
  - **Dividir:** (achar o meio) tempo constante.
  - **Conquistar:** resolver 2 subproblemas recursivamente, cada um de tamanho $n/2$, isto é, $2T(n/2)$.
  - **Combinar:** intercalar ($\textsc{Merge}$) custa $C(n) = cn$.
- $D(n) + C(n) = cn$, portanto a recorrência para $\textsc{Mergesort}$ será:

$$T(n) = \begin{cases} c & \text{se } n = 1, \\ 2T(n/2) + cn & \text{se } n > 1. \end{cases}$$

Análise de complexidade do MERGE

- Árvore de recursão, que mostra as sucessivas expansões da recorrência.
- Para o problema original, há o custo cn, mais o custo dos dois subproblemas, cada um custando $T(n/2)$.
- Para cada subproblema de tamanho $n/2$, há o custo $cn/2$, mais dois subproblemas custando $T(n/4)$ cada.
- Continuar expandindo, até que o tamanho do (sub)problema diminua até 1.
- Cada nível possui custo cn.
- Existem $lgn + 1$ níveis.
- O custo total é a soma dos custo de cada nível.
- Custo total: $cnlgn + cn$
- $T(n) = \Theta(n \lg n)$.

Tarefa

Leitura

Leia o Capítulo 3 do Cormen e o Apêndice A (próxima aula abordaremos Análise Assintótica).

Exercício 1

Apresente um algoritmo recursivo que efetua busca binária. Analise a complexidade do algoritmo usando árvore de recorrência.

Exercício 2

Faça o exercício 2.4 do Cormen (inversões).