AULA 03 – ANÁLISE DO ALGORITMO MERGESORT

Prof. Daniel Kikuti

Universidade Estadual de Maringá

29 de julho de 2014

Resumo da aula anterior

- Técnica de desenvolvimento estudada: **abordagem incremental**.
  - Constrói a solução a medida em que processa a entrada elemento a elemento (viés dinâmico).
  - Demonstração de correção do algoritmo por **invariante de laço**.
  - Tempo de execução depende do tempo gasto em cada iteração.
- Demonstração por invariante de laço:
  - Declaração do invariante (propriedade que será usada para demonstrar a correção).
  - Demonstração em três passos (inicialização, manutenção e término).
- Análise de complexidade:
  - Função especificando a ordem de crescimento em relação ao tamanho da entrada do algoritmo.
  - Contagem do número de execuções de cada linha.

Solução do exercício da aula anterior

O algoritmo *Selection Sort*

```
selection-sort(A)
for i = 1 to A.length - 1
    menor = i
    for j = i + 1 to A.length
        if A[j] < A[menor]
            menor = j
    Troque A[i] e A[menor]
```

Solução do exercício da aula anterior

O algoritmo *Selection Sort*

```
selection-sort(A)
for i = 1 to A.length - 1
    menor = i
    for j = i + 1 to A.length
        if A[j] < A[menor]
            menor = j
    Troque A[i] e A[menor]
```

Correção do Selection sort

- Invariante do laço interno: `A[menor]` contém o menor elemento de `A[i..j-1]`.
- Invariante do laço externo: O subvetor `A[1..i-1]` consiste dos `i-1` menores elementos do vetor `A[1..n]` e, este subvetor está ordenado.

Solução do exercício da aula anterior

Análise de complexidade (no pior caso)[1]

`selection-sort(A)`	custo	# de execuções
`1 for i = 1 to A.length - 1`	c_1	n
`2    menor = i`	c_2	$n-1$
`3    for j = i + 1 to A.length`	c_3	$\sum_{j=2}^{n} j$
`4       if A[j] < A[menor]`	c_4	$\sum_{j=2}^{n}(j-1)$
`5          menor = j`	c_5	$\sum_{j=2}^{n}(j-1)$
`6    Troque A[i] e A[menor]`	c_6	$n-1$

Custo total

Multiplicando pelas constantes e desenvolvendo temos:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3 \sum_{j=2}^{n} j + c_4 \sum_{j=2}^{n}(j-1) + \\
&\quad c_5 \sum_{j=2}^{n}(j-1) + c_6(n-1) \\
&= \left(\frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2}\right)n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6\right)n - \\
&\quad (c_2 + c_3 + c_6).
\end{aligned}
$$

Função quadrática.

[1] No melhor caso a linha 5 não executa nenhuma vez.

Objetivos desta aula

- Técnica de Projeto de Algoritmos.
  - Divisão e conquista.
- Correção do *Merge-Sort*.
- Análise da complexidade.
  - Equações de recorrência.
- Exercícios.

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.

Exemplo de Geometria Computacional

Fecho convexo (envoltória convexa) de um conjunto de pontos

- Entrada: n pontos no plano.
- Saída: Fecho convexo.

Exemplo de Geometria Computacional

Fecho convexo (envoltória convexa) de um conjunto de pontos

- Entrada: n pontos no plano.
- Saída: Fecho convexo.

Usando a abordagem de divisão e conquista

- Divida os n pontos em duas metades.
- Encontre o fecho convexo de cada subconjunto.
- Combine os dois fechos convexos em um único fecho convexo.

Exemplo de Geometria Computacional

Fecho convexo (envoltória convexa) de um conjunto de pontos

- Entrada: n pontos no plano.
- Saída: Fecho convexo.

Usando a abordagem de divisão e conquista

- Divida os n pontos em duas metades.
- Encontre o fecho convexo de cada subconjunto.
- Combine os dois fechos convexos em um único fecho convexo.

Parte complicada

Em geral, demonstrar que a etapa de combinação está correta é o grande desafio.

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]`.

O algoritmo

Visão geral

- A divisão é feita no procedimento `merge-sort`.
- Caso base: subvetor com um elemento (ordenado).
- A intercalação é feita pelo procedimento `merge`.
- A chamada inicial `merge-sort(A, 1, n)`.

```
merge-sort(A, inicio, fim)
if inicio < fim then
    meio = ⌊(inicio + fim)/2⌋;
    merge-sort (A, inicio, meio);
    merge-sort (A, meio+1, fim);
    merge (A, inicio, meio, fim);
```

O algoritmo

```
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. E, 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*

```
merge-sort(A, inicio, fim)
if inicio < fim then
    meio = ⌊(inicio + fim)/2⌋;
    merge-sort (A, inicio, meio);
    merge-sort (A, meio+1, fim);
    merge (A, inicio, meio, fim);
```

Análise de complexidade do *MergeSort*

```
merge-sort(A, inicio, fim)
if inicio < fim then
    meio = ⌊(inicio + fim)/2⌋;
    merge-sort (A, inicio, meio);
    merge-sort (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 (*merge*) custa $C(n) = cn$.
- $D(n) + C(n) = cn$, portanto a recorrência para `merge-sort` 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).