AULA 02 – ANÁLISE DO ALGORITMO INSERTION SORT

Prof. Daniel Kikuti

Universidade Estadual de Maringá

23 de julho de 2014

Resumo da aula anterior

- **Primeira parte:** blá blá blá.
  - Sobre a disciplina (avaliação/trabalhos).
  - Datas.
- **Segunda parte:** mais blá blá blá.
  - Definição do objeto de estudo.
  - O que analisar.
- **Na última parte da aula:** esboço do que faremos durante boa parte da disciplina.
  - Três algoritmos para multiplicação de inteiros longos com análise de complexidade de informal.
  - Exercício para verificar crescimento de funções.

Solução do exercício

	1 segundo	**1 minuto**	**1 hora**	**1 dia**
$\lg n$	2^{10^6}	$2^{6\times10^7}$	$2^{3{,}6\times10^9}$	$2^{8{,}64\times10^{10}}$
$\sqrt{n}$	10^{12}	$3,6 \times 10^{15}$	$1,3 \times 10^{19}$	$7,46 \times 10^{21}$
n	10^6	6×10^7	$3,6 \times 10^9$	$8,64 \times 10^{10}$
$n \lg n$	62746	2801417	$1,33 \times 10^8$	$2,76 \times 10^9$
n^2	1000	7745	60000	293938
n^3	100	391	1532	4420
2^n	19	25	31	36
$n!$	9	11	12	13

Objetivos desta aula

- Técnica de Projeto de Algoritmos.
  - Abordagem incremental.
- Correção do *Insertion sort*.
  - Invariante de laço.
- Análise da complexidade.
  - Melhor caso.
  - Pior caso.
  - Caso médio.
- Exercícios.

Técnica de projeto de algoritmos

Cenário

- Você possui um grande volume de dados.
- Ao longo do tempo este volume de dados está sujeito a diversas (relativamente) pequenas mudanças.
- Você está processando estes dados e deseja manter o resultado (parcial) atualizado com o volume de dados alterado.

Técnica de projeto de algoritmos

Cenário

- Você possui um grande volume de dados.
- Ao longo do tempo este volume de dados está sujeito a diversas (relativamente) pequenas mudanças.
- Você está processando estes dados e deseja manter o resultado (parcial) atualizado com o volume de dados alterado.

Algoritmo incremental

É um algoritmo que parte de uma solução inicial e vai atualizando esta solução à medida que os dados da entrada são processados.

Exemplo de Geometria Computacional

Fecho convexo (envoltória convexa) de um conjunto de pontos

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

Definições

Um polígono P é **convexo** se para quaisquer pontos $x, y \in P$, todos os pontos do segmento de reta $\overline{xy}$ estão em P.
O **fecho convexo** é o menor polígono convexo que contém todos os pontos.

Um algoritmo incremental para fecho convexo

Insertion Hull

Dado um conjunto $I = \{p_1, \ldots, p_n\}$ de n pontos e seja P o polígono que irá representar o fecho convexo.

1. Comece com o triângulo $P = (p_1, p_2, p_3)$ (fecho convexo trivial);
2. Para $i = 4$ até n faça
   - Se i não está no interior de P então encontre as duas retas que tangenciam o ponto i e P e atualize P;
3. Devolva P.

Algoritmo incremental de modo geral

- Seja $I = \{i_1, i_2, \ldots, i_n\}$ o conjunto de n elementos da entrada e o objetivo seja computar uma função $f(I)$.
- Começamos com um conjunto vazio S (ou uma solução trivial) e mantemos $f(S)$ a cada iteração do algoritmo.
- Então o algoritmo fará um total de n iterações (aproximadamente) e ao final devolverá solução S para o problema.

Algoritmo incremental de modo geral

- Seja $I = \{i_1, i_2, \ldots, i_n\}$ o conjunto de n elementos da entrada e o objetivo seja computar uma função $f(I)$.
- Começamos com um conjunto vazio S (ou uma solução trivial) e mantemos $f(S)$ a cada iteração do algoritmo.
- Então o algoritmo fará um total de n iterações (aproximadamente) e ao final devolverá solução S para o problema.

Análise de algoritmos incrementais

- A complexidade de tempo de um algoritmo incremental depende do tempo gasto em cada iteração.
- A correção do algoritmo em geral se dá usando invariante de laço.

Veremos detalhadamente a análise do algoritmo incremental *Insertion sort*.

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 a abordagem incremental

- Manteremos uma parte dos dados ordenada;
- Esta parte aumentará a cada iteração;
- No final, teremos todos os elementos ordenados.

O algoritmo

O algoritmo a seguir é uma versão que faz **ordenação local**, isto é, dentro do próprio vetor A, usando no máximo um número constante de espaço de memória para auxiliar no processo.

```
insertion-sort(A)
1 for j = 2 to A.length
2     chave = A[j]
3     /*insere A[j] na sequência ordenada A[1..j-1]*/
4     i = j - 1
5     while i > 0 and A[i] > chave
6         A[i + 1] = A[i]
7         i = i - 1
8     A[i + 1] = chave
```

Como demonstrar que este algoritmo está correto?

Invariante de laço

Declarar a invariante

Propriedade que se mantém verdadeira antes do início do laço, durante a manutenção do laço e na saída do laço. Deve ser uma propriedade que auxilie na demonstração da correção do algoritmo.

Inicialização

A propriedade deve ser verdadeira antes da primeira iteração

Manutenção

Verificar que a propriedade se mantém a cada iteração. A propriedade deve permanecer válida para as próximas iterações.

Término

Verificar a propriedade quando a última iteração do laço for executada.

Uma figura para fixar a idéia

```
⋮
ANTES
/* O invariante é verdadeiro */
enquanto Condição faça
    /* O invariante é verdadeiro */
    CORPO DO LAÇO
    /* O invariante é verdadeiro */
fim enquanto
/* Condição é falsa e o invariante é verdadeiro */
DEPOIS
⋮
```

Correção do *Insertion sort*

Declarar a invariante

Para o laço **for** (linhas 1–8), $A[1 \ldots j-1]$ contém os mesmos elementos contidos originalmente em $A[1 \ldots j-1]$, mas em sequência ordenada.

Correção do *Insertion sort*

Declarar a invariante

Para o laço **for** (linhas 1–8), $A[1 \ldots j-1]$ contém os mesmos elementos contidos originalmente em $A[1 \ldots j-1]$, mas em sequência ordenada.

Inicialização

Quando j=2, o subvetor $A[1 \ldots j-1]$ consiste apenas do elemento $A[1]$ que é o mesmo elemento original em $A[1]$ e está trivialmente ordenado.

Correção do *Insertion sort*

Declarar a invariante

Para o laço **for** (linhas 1–8), $A[1 \ldots j-1]$ contém os mesmos elementos contidos originalmente em $A[1 \ldots j-1]$, mas em sequência ordenada.

Inicialização

Quando j=2, o subvetor $A[1 \ldots j-1]$ consiste apenas do elemento $A[1]$ que é o mesmo elemento original em $A[1]$ e está trivialmente ordenado.

Manutenção

O corpo do laço **for** consiste em deslocar os elementos $A[j-1], A[j-2], \ldots$ uma posição a direita, até encontrar a posição adequada do elemento $A[j]$ (linhas 4–7). Neste ponto, o valor de $A[j]$ é inserido. O subvetor $A[1 \ldots j]$ consiste então dos elementos originalmente em $A[1 \ldots j]$, mas de forma ordenada.

Continuação da Correção do *Insertion sort*

Término

O laço externo termina quando j excede n, isto é, quando $j = n + 1$. Substituindo $j = n + 1$ no enunciado do invariante de laço, temos que o subvetor $A[1 \ldots n]$ consiste nos elementos originalmente contidos no vetor $A[1 \ldots n]$, mas em sequência ordenada. Contudo, o subvetor $A[1 \ldots n]$ é o vetor inteiro. Portanto, o algoritmo está correto.

Análise de complexidade do *Insertion sort*

Considerações

- Usaremos o modelo RAM.
- O recurso que queremos prever para o insertion-sort é o tempo.
- O tempo de execução de um algoritmo em uma determinada entrada é o número de operações primitivas executadas.
- O tempo de execução depende da quantidade de itens na entrada (tamanho da entrada).
- Vamos assumir que cada linha executada consome um período constante de tempo.

Análise de Complexidade

insertion-sort(A)	custo	# de execuções
1 for j = 2 to A.length	c_1	
2 chave = A[j]	c_2	
3 /*insere A[j] ... */	0	
4 i = j - 1	c_4	
5 while i > 0 and A[i] > chave	c_5	
6 A[i + 1] = A[i]	c_6	
7 i = i - 1	c_7	
8 A[i + 1] = chave	c_8	

Análise de Complexidade

`insertion-sort(A)`	custo	# de execuções
`1 for j = 2 to A.length`	c_1	n
`2  chave = A[j]`	c_2	$n-1$
`3  /*insere A[j] ...  */`	0	$n-1$
`4  i = j - 1`	c_4	$n-1$
`5  while i > 0 and A[i] > chave`	c_5	$\sum_{j=2}^{n} t_j$
`6  A[i + 1] = A[i]`	c_6	$\sum_{j=2}^{n}(t_j-1)$
`7  i = i - 1`	c_7	$\sum_{j=2}^{n}(t_j-1)$
`8  A[i + 1] = chave`	c_8	$n-1$

- t_j é o número de vezes que o teste do while é executado para o valor j.
- Para calcularmos o tempo de execução $T(n)$, somamos os produtos das colunas custo e # de execuções.

Análise de Complexidade

Custo total

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + \\ & \quad c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \end{aligned}$$

Analisando o t_j

O tempo de execução pode ser diferente (mesmo para entradas do mesmo tamanho).

- Como deve ser a entrada para executar o menor número de iterações?
- Como deve ser a entrada para executar o maior número de iterações?

Análise de Complexidade: Melhor Caso

- Ocorre quando o vetor já está ordenado.
- O teste `A[i] > chave` falha na primeira comparação quando $i = j - 1$
- Portanto, $t_j = 1$ para $j = 2, \ldots, n$, e o tempo de execução no melhor caso será:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- Esse tempo pode ser expresso como $T(n) = an + b$, para constantes a e b.
- É uma **função linear** de n.

Análise de Complexidade: Pior Caso

- Ocorre quando o vetor está em ordem inversa.
- Cada elemento de `A[j]` deve ser comparado com todos os elementos do subvetor ordenado `A[1...j-1]`.
- Portanto, $t_j = j$ para $j = 2, \ldots, n$. Como

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

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

o tempo de execução no pior caso é:

Análise de Complexidade: Pior Caso

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + \\
&\quad c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} + c_8\right)n - \\
&\quad (c_1 + c_4 + c_5 + c_8).
\end{aligned}
$$

- Esse tempo pode ser expresso como $T(n) = an^2 + bn + c$, para constantes a, b e c.
- É uma **função quadrática** de n.

Análise de Complexidade: Caso Médio

- Suponha a entrada com n números escolhidos aleatoriamente.
- Quantas comparações são necessárias para se descobrir o lugar adequado de `A[j]` dentro do subvetor ordenado `A[1 ...j-1]`?
- Em média, metade dos elementos em `A[1 ...j-1]` são menores que `A[j]` e metade maiores.
- Portanto, em média verificaremos metade do subvetor ordenado, ou seja $t_j = j/2$ para $j = 2, \ldots, n$, e o tempo de execução no melhor caso será quadrático.

Algumas considerações sobre ordem de crescimento

- Começamos ignorando o custo real de cada instrução (adotamos um custo abstrato c_i por linha).
- Fizemos a análise do algoritmo e chegamos no pior caso em uma função $T(n) = an^2 + bn + c$ onde a, b e c são constantes que dependem dos custos c_i (ignoramos também os custos abstratos).
- Abstraimos mais uma vez, considerando apenas o termo que expressa a ordem de crescimento (termo de mais alta ordem da função).
- Termos de mais baixa ordem são insignificantes para grandes valores de n.

Algumas considerações sobre ordem de crescimento

- Em geral, também ignoramos o coeficiente constante do termo de mais alta ordem (são menos significativos que a taxa de crescimento para determinar a eficiência computacional para grandes entradas).
- Um algoritmo é mais eficiente que outro se o seu tempo no pior caso tem uma ordem de crescimento menor.
- Um algoritmo com maior ordem de crescimento pode demorar menos para entradas pequenas que um algoritmo com uma menor ordem de crescimento (devido às constantes e termos de mais baixa ordem despresados), porém, para entradas suficientemente grandes, um algoritmo quadrático será mais rápido no pior caso que um algoritmo cúbico por exemplo.

Tarefa

Leitura

Leia o Capítulo 2 do Cormen (20 páginas). A primeira parte detalha a análise do *insertion sort*, a segunda parte apresenta a técnica de divisão e conquista e análise do *mergesort*

Exercício 1

Qual seria o invariante de laço para o problema do fecho convexo?

Exercício 2

Faça o exercício 2.2-2 do Cormen (demonstre a correção do algoritmo *selection sort* e faça a análise de complexidade).