Análise do Algoritmo Insertion Sort

Prof. Daniel Kikuti

Universidade Estadual de Maringá

Solução do problema 1.1 – Cormen

Problema

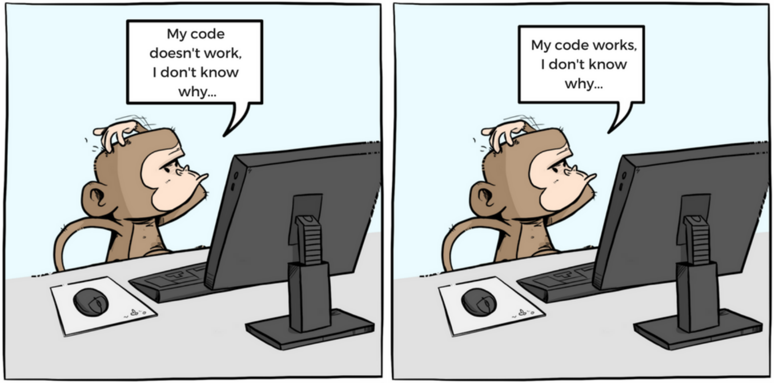
Se o algoritmo leva $f(n)$ microssegundos para resolver um problema, determine o valor n que pode ser resolvido no tempo t (ou seja, $f(n) = 10^6 t$).

	1 segundo	**1 minuto**	**1 hora**	**1 dia**
$\lg n$	2^{10^6}	$2^{6\times 10^7}$	$2^{3{,}6\times 10^9}$	$2^{8{,}64\times 10^{10}}$
$\sqrt{n}$	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

- Correção de algoritmos iterativos
  - Invariante de laço.
  - Correção do INSERTION SORT.
- Análise da complexidade.
  - Melhor caso.
  - Pior caso.
  - Caso médio.
- Exercícios.

Correção de algoritmos – Who cares?



Fonte: `https://neoteric.eu/what-do-programmers-hate`

Máximo de um vetor de n elementos

O problema

Dado um vetor com n elementos $(n \geq 1)$ como demonstrar que o algoritmo a seguir está correto?

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

```
max ← A[1]
for i ← 2 to n do
    if A[i] > max then
        max ← A[i]
return max
```

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.

Para fixar a idéia

⋮

ANTES: O invariante é verdadeiro (no caso do **for**, considere após a inicialização das variáveis de controle)

while *Condição do laço* **do**

  O invariante é verdadeiro

  CORPO DO LAÇO

  O invariante é verdadeiro para a próxima iteração

DEPOIS: Condição é falsa e o invariante é verdadeiro.

⋮

Invariante de laço para o algoritmo max

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

$max \leftarrow A[1]$
for $i \leftarrow 2$ *to* n **do**
  if $A[i] > max$ **then**
    $max \leftarrow A[i]$
return max

Definindo a invariante

max contém o valor do maior elemento do vetor $A[1 \ldots i-1]$.

Verificação

- inicialização;
- manutenção;
- término.

Demonstração

Definindo a invariante

max contém o valor do maior elemento do vetor $A[1 \ldots i-1]$.

Verificação

- **Inicialização:** *max* contém o valor $A[1]$, que é o maior elemento do vetor $A[1 \ldots 2-1] = A[1]$.
- **Manutenção:** assuma que o invariante é válido para uma iteração i qualquer. Executando o corpo do laço, existem duas possibilidades:
  - $A[i] > max$: o elemento $A[i]$ é maior que todos os elementos de $A[1 \ldots i-1]$, então atualiza-se o valor de *max* (linha 4).
  - $A[i] \leq max$: o algoritmo não atualiza o valor de *max*.

  Nos dois casos, no fim do corpo do laço, *max* contém o maior elemento do vetor $A[1 \ldots i]$, e o invariante será válido para a próxima iteração.
- **Término:** Quando $i = n+1$, a condição do laço se torna falsa e o invariante afirma que *max* contém o valor do maior elemento do vetor $A[1 \ldots n+1-1] = A[1 \ldots n]$, ou seja, o algoritmo devolve o maior elemento do vetor.

O problema de ordenação

Entrada

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

Saída

Uma permutação (reordenação) $\langle a'_1, a'_2, \dots, a'_n \rangle$ 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[])
for j ← 2 to A.length do
    key ← A[j]
    /* Insert A[j] into the sorted sequence A[1 . . . j−1] */
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```

Como demonstrar que este algoritmo está correto?

Correção do INSERTION SORT

Declarar a invariante

Para o laço **for** (linhas 1–8), $A[1 \dots j-1]$ contém os mesmos elementos contidos originalmente em $A[1 \dots j-1]$, 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
`for j = 2 to A.length`	c_1	
`  key = A[j]`	c_2	
`  /*insert A[j] ... */`	0	
`  i = j - 1`	c_4	
`  while i > 0 and A[i] > key`	c_5	
`    A[i + 1] = A[i]`	c_6	
`    i = i - 1`	c_7	
`  A[i + 1] = key`	c_8	

Análise de Complexidade

`Insertion Sort(A)`	custo	# de execuções
`for j = 2 to A.length`	c_1	n
`  key = A[j]`	c_2	$n-1$
`  /*insert A[j] ... */`	0	$n-1$
`  i = j - 1`	c_4	$n-1$
`  while i > 0 and A[i] > key`	c_5	$\sum_{j=2}^{n} t_j$
`    A[i + 1] = A[i]`	c_6	$\sum_{j=2}^{n}(t_j - 1)$
`    i = i - 1`	c_7	$\sum_{j=2}^{n}(t_j - 1)$
`  A[i + 1] = key`	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

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

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 $A[i] > key$ 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 $A[1 \ldots j-1]$.
- 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 $A[1 \ldots j-1]$?
- Em média, metade dos elementos em $A[1 \ldots j-1]$ 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 desprezados), 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

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