Projeto e Análise de Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

21 de abril de 2017

Conceito de Algoritmo

Definição de Algoritmo:

Um procedimento computacional bem definido que toma um conjunto de valores como entrada e produz um conjunto de valores como saída.

É uma ferramenta tecnológica para resolver um problema computacional bem especificado.

Exemplos de Problemas computacionais:

Problema: Determinar se um número é quadrado perfeito.

Entrada: 8.769
Saída: Não.

Entrada: 18.769
Saída: Sim.

Consegue pensar em um algoritmo que resolva este problema?

Exemplos de Problemas computacionais:

Problema: Determinar se um número é quadrado perfeito.

Entrada: 8.769
Saída: Não.

Entrada: 18.769
Saída: Sim.

Consegue pensar em um algoritmo que resolva este problema?

Exemplos:

Problema: Determinar o terceiro maior número de uma seqüência.

Entrada:

1											n
25	18	33	14	73	65	33	57	18	49	52	9

Saída: 57.

E se fosse pedido o 7º maior número, se a ordem fosse uma entrada? O algoritmo seria o mesmo?

Exemplos:

Problema: Determinar o terceiro maior número de uma seqüência.

Entrada:

1											n
25	18	33	14	73	65	33	57	18	49	52	9

Saída: 57.

E se fosse pedido o 7º maior número, se a ordem fosse uma entrada? O algoritmo seria o mesmo?

Um problema computacional define uma relação entre a entrada e a saída, sem especificar como deve ser atingida a saída.

Um algoritmo define de forma específica como deve-se obter a saída a partir da entrada. O resultado deve ser correto para toda e qualquer instância de entrada.

No entanto, um algoritmo é específico para um determinado problema, se mudarmos o problema, o algoritmo não irá garantir a saída correta para qualquer instância.

Um problema computacional define uma relação entre a entrada e a saída, sem especificar como deve ser atingida a saída.

Um algoritmo define de forma específica como deve-se obter a saída a partir da entrada. O resultado deve ser correto para toda e qualquer instância de entrada.

No entanto, um algoritmo é específico para um determinado problema, se mudarmos o problema, o algoritmo não irá garantir a saída correta para qualquer instância.

O algoritmo também é uma tecnologia computacional. Por exemplo, podemos melhorar a tecnologia de hardware aumentando o *clock* do processador para aumentarmos a eficiência na execução dos programas. Da mesma forma, podemos ganhar eficiência melhorando o algoritmo.

Vamos supor dois computadores, o primeiro 10 vezes mais rápido (1G operações/s) que o segundo (100M operações/s). No primeiro usamos um algoritmo que consegue encontrar o *k-ésimo* elemento em um vetor, em tempo $c_1 \cdot n \lg n$, enquanto que no segundo o algoritmo executa a operação em tempo $c_2 \cdot n$.

Qual o mais eficiente em um vetor de 1 milhão de elementos?

Cálculo para o primeiro computador: (supor $c_1 = 2$)

$$T = \frac{2 \cdot 10^6 \cdot \lg 10^6 \text{instruções}}{10^9 \text{instruções/segundo}} = 39.86ms$$

Cálculo para o segundo computador: (supor $c_2 = 3$)

$$T = \frac{3 \cdot 10^6 \text{instruções}}{10^8 \text{instruções/segundo}} = 30ms$$

Apesar do primeiro computador ser mais rápido, e ter uma constante multiplicativa no número de operações menor, o segundo computador resolve mais rápido. A tecnologia envolvida no segundo algoritmo o torna mais eficiente.

Cálculo para o primeiro computador: (supor $c_1 = 2$)

$$T = \frac{2 \cdot 10^6 \cdot \lg 10^6 \text{instruções}}{10^9 \text{instruções/segundo}} = 39.86ms$$

Cálculo para o segundo computador: (supor $c_2 = 3$)

$$T = \frac{3 \cdot 10^6 \text{instruções}}{10^8 \text{instruções/segundo}} = 30ms$$

Apesar do primeiro computador ser mais rápido, e ter uma constante multiplicativa no número de operações menor, o segundo computador resolve mais rápido. A tecnologia envolvida no segundo algoritmo o torna mais eficiente.

Por que a preocupação com algoritmos?

Atualmente temos diversos problemas em várias áreas que podem ser resolvidos de forma computacional, e qualidade do algoritmo traduz na melhor solução tecnológica. Exemplos:

- Projeto Genoma Humano: *determinar 3 bilhões de pares de bases químicas que definem o DNA humano.*
- Rotas em GPS: *qual a menor rota, ou a rota mais rápida entre dois pontos?*
- Logística em Portos: *qual a ordem de embarque e desembarque dos containers para melhor rapidez na operação evitando que o navio aderne?*

Problemas difíceis

Infelizmente nem todos problemas possuem algoritmos eficientes que possam ser empregados. A vantagem é que para muitos destes problemas, até o momento, não foi possível provar que esta solução não existe.

Um grande conjunto de problemas possui a característica de que se for encontrada uma solução eficiente para um problema do conjunto, então todos problemas possuem soluções eficientes. Estes são problemas considerados *NP*-Completos

Apresentação do algoritmo

Um algoritmo pode ser apresentado de várias formas:

- Usando linguagem de programação de alto nível: *C*, *Pascal*, *Java*, ...
- Em português
- Em linguagem de pseudo-código, como no CLRS

Trabalharemos principalmente com as duas últimas opções nestes slides.

Apresentação do algoritmo

Um algoritmo pode ser apresentado de várias formas:

- Usando linguagem de programação de alto nível: *C*, *Pascal*, *Java*, ...
- Em português
- Em linguagem de pseudo-código, como no CLRS

Trabalharemos principalmente com as duas últimas opções nestes slides.

Uma das características de algoritmos é a sua corretude. Códigos que não resultam a saída esperada para cada instância de entrada fornecida não podem ser considerados algoritmos que resolvam o problema computacional.

Um algoritmo deve possuir um momento de inicialização, quando recebe a entrada; um momento de execução, quando processa a entrada; e um momento de término, quando entrega a saída. Todo algoritmo correto termina sua execução.

É necessário provar que durante estas fases o algoritmo se mantém correto, e provar que ele atinge a terceira fase, assim, o algoritmo irá terminar e irá entregar a saída correta.

Loops Invariantes

Os loops invariantes auxiliam a provar que um algoritmo é correto. Para tanto é necessário provar três fases:

1. Inicialização: *O invariante é verdadeiro antes da primeira iteração do loop.*
2. Manutenção: *Se o invariante é verdadeiro antes de uma iteração do loop, ele continua verdadeiro após a iteração.*
3. Término: *Quando concluído, o invariante fornece uma propriedade útil que nos ajuda a mostrar que o algoritmo é correto.*

Loops Invariantes

Os loops invariantes auxiliam a provar que um algoritmo é correto. Para tanto é necessário provar três fases:

1. Inicialização: *O invariante é verdadeiro antes da primeira iteração do loop.*
2. Manutenção: *Se o invariante é verdadeiro antes de uma iteração do loop, ele continua verdadeiro após a iteração.*
3. Término: *Quando concluído, o invariante fornece uma propriedade útil que nos ajuda a mostrar que o algoritmo é correto.*

Loops Invariantes

Os loops invariantes auxiliam a provar que um algoritmo é correto. Para tanto é necessário provar três fases:

1. Inicialização: *O invariante é verdadeiro antes da primeira iteração do loop.*
2. Manutenção: *Se o invariante é verdadeiro antes de uma iteração do loop, ele continua verdadeiro após a iteração.*
3. Término: *Quando concluído, o invariante fornece uma propriedade útil que nos ajuda a mostrar que o algoritmo é correto.*

Exemplo: Ordenação por inserção

Algoritmo $\text{INSERTION-SORT}(A)$
  para $j \leftarrow 2$ **até** $comprimento[A]$ **faça**
    $chave \leftarrow A[j]$
      $\triangleright$ Inserir $A[j]$ na seqüência ordenada $A[1 \cdot \cdot j - 1]$.
    $i \leftarrow j - 1$
    enquanto $i > 0$ e $A[i] > chave$ **faça**
      $A[i + 1] \leftarrow A[i]$
      $i \leftarrow i - 1$
    $A[i + 1] \leftarrow chave$

Vamos provar que o loop *para* do algoritmo acima é um loop invariante

Exemplo: Ordenação por inserção

Algoritmo INSERTION-SORT(A)
  para $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**
    chave $\leftarrow A[j]$
      $\triangleright$ Inserir $A[j]$ na seqüência ordenada $A[1 \cdot \cdot j - 1]$.
    $i \leftarrow j - 1$
    enquanto $i > 0$ e $A[i] >$ *chave* **faça**
      $A[i + 1] \leftarrow A[i]$
      $i \leftarrow i - 1$
    $A[i + 1] \leftarrow$ *chave*

Vamos provar que o loop *para* do algoritmo acima é um loop invariante

Invariante do loop:

No começo de cada iteração do loop *para* das linhas 2 a 8, o subarranjo $A[1..j-1]$ consiste nos elementos contidos originalmente em $A[1..j-1]$, mas em seqüência ordenada.

1. **Inicialização:** O loop é válido antes da primeira iteração: $j \leftarrow 2$. $A[1 \cdot \cdot j-1]$ consiste de um único elemento: $A[1]$, é o elemento original e está trivialmente ordenado.
2. Manutenção: Se no final da iteração $[j-1]$ o invariante é válido, na iteração j, o loop *enquanto* faz com que o elemento $A[j]$ troque com cada elemento anterior até a posição em que ele não é menor que o anterior. Com isto a seqüência $A[1 \cdot \cdot j]$ contém os elementos originais de $A[1 \cdot \cdot j]$ e está ordenada. O invariante é válido.
3. Término: Para $j = n+1$ o algoritmo para. Os elementos de $A[1 \cdot \cdot n]$ estão ordenados.

Invariante do loop:

No começo de cada iteração do loop *para* das linhas 2 a 8, o subarranjo $A[1..j-1]$ consiste nos elementos contidos originalmente em $A[1..j-1]$, mas em seqüência ordenada.

1. Inicialização: O loop é válido antes da primeira iteração: $j \leftarrow 2$. $A[1 \cdot \cdot j-1]$ consiste de um único elemento: $A[1]$, é o elemento original e está trivialmente ordenado.
2. **Manutenção:** Se no final da iteração $[j-1]$ o invariante é válido, na iteração j, o loop *enquanto* faz com que o elemento $A[j]$ troque com cada elemento anterior até a posição em que ele não é menor que o anterior. Com isto a seqüência $A[1 \cdot \cdot j]$ contém os elementos originais de $A[1 \cdot \cdot j]$ e está ordenada. O invariante é válido.
3. Término: Para $j = n+1$ o algoritmo para. Os elementos de $A[1 \cdot \cdot n]$ estão ordenados.

Invariante do loop:

No começo de cada iteração do loop *para* das linhas 2 a 8, o subarranjo $A[1..j-1]$ consiste nos elementos contidos originalmente em $A[1..j-1]$, mas em seqüência ordenada.

1. Inicialização: O loop é válido antes da primeira iteração: $j \leftarrow 2$. $A[1 \cdot \cdot j-1]$ consiste de um único elemento: $A[1]$, é o elemento original e está trivialmente ordenado.
2. Manutenção: Se no final da iteração $[j-1]$ o invariante é válido, na iteração j, o loop *enquanto* faz com que o elemento $A[j]$ troque com cada elemento anterior até a posição em que ele não é menor que o anterior. Com isto a seqüência $A[1 \cdot \cdot j]$ contém os elementos originais de $A[1 \cdot \cdot j]$ e está ordenada. O invariante é válido.
3. **Término:** Para $j = n + 1$ o algoritmo para. Os elementos de $A[1 \cdot \cdot n]$ estão ordenados.

Provando o loop invariante como correto, provamos que o algoritmo fornece um resultado correto.

Observe que existe outro loop no algoritmo e seria necessário também, em uma prova mais completa, provar que existe um invariante correto, este invariante representa a frase que usamos: *"o loop enquanto faz com que o elemento* $A[j]$ *troque com cada elemento anterior até a posição em que ele não é menor que o anterior"*. A prova do invariante para o loop *enquanto* deve ser parte da prova que realizamos do invariante do loop *para*.

Invariantes para o loop *enquanto*, linhas $5 \cdot \cdot 7$:

1. $A[1 \cdot \cdot i]$ e $A[i+2 \cdot \cdot j]$ contém os elementos de $A[1 \cdot \cdot j-1]$ antes de entrar no laço que começa na linha 5.
2. $A[1 \cdot \cdot i]$ e $A[i+2 \cdot \cdot j]$ são crescentes.
3. $A[1 \cdot \cdot i] \leqslant A[i+2 \cdot \cdot j]$.
4. $A[i+2 \cdot \cdot j] > chave$.

Provando os invariantes acima, da mesma forma que foi provado o invariante do loop *para*, conseguimos completar a prova da corretude do algoritmo.

Quanto tempo demora para executar o algoritmo?

Não adianta um algoritmo ser correto se ele for *muuuuuiiiiiito leeeeeeeeento*.

Precisamos medir o tempo de execução do algoritmo. Mas o tempo de execução depende de muitos fatores: tempo de clock da máquina, número de instruções que o compilador gera, número de clocks que cada instrução gasta, paralelismo, ...

Não queremos comparar tempos de máquinas, e sim tempos de algoritmos. Precisamos adotar um modelo uniforme para medida da eficiência do algoritmo.

Para medida de eficiência será utilizado o modelo abstrato RAM - (Random Access Machine)

- simula máquinas convencionais (de verdade), possui um único processador que executa instruções seqüencialmente,
- tipos básicos são números inteiros e reais, há um limite no tamanho de cada palavra de memória: se a entrada tem "tamanho" n, então cada inteiro/real é representado por $c \lg n$ bits onde $c \geqslant 1$ é uma contante.
- executa operações aritméticas, comparações, movimentação de dados de tipo básico e fluxo de controle (teste if/else, chamada e retorno de rotinas) em tempo constante,

Veja detalhes no CLRS

Vamos medir a eficiência do algoritmo de Ordenação por inserção.

Vamos utilizar como parâmetro para medida da eficiência o tamanho do vetor, e não os valores de cada elemento do vetor.

Como a máquina RAM define um tempo para cada instrução básica, precisamos contar quantas instruções básicas são executadas no algoritmo.

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	?
3	*chave* $\leftarrow A[j]$	c_2	?
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	?
4	$i \leftarrow j-1$	c_4	?
5	**enquanto** $i > 0$ e $A[i] >$ *chave* **faça**	c_5	?
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow$ *chave*	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	$chave \leftarrow A[j]$	c_2	?
	▷ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	?
4	$i \leftarrow j-1$	c_4	?
5	**enquanto** $i > 0$ e $A[i] > chave$ **faça**	c_5	?
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow chave$	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	*chave* $\leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	?
4	$i \leftarrow j-1$	c_4	?
5	**enquanto** $i > 0$ e $A[i] >$ *chave* **faça**	c_5	?
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow$ *chave*	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT(A)*	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	*chave* $\leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	?
5	**enquanto** $i > 0$ e $A[i] >$ *chave* **faça**	c_5	?
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow$ *chave*	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	*chave* $\leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**enquanto** $i > 0$ e $A[i] >$ *chave* **faça**	c_5	?
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow$ *chave*	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** $comprimento[A]$ **faça**	c_1	n
3	$chave \leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**enquanto** $i > 0$ e $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	?
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow chave$	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	$chave \leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**enquanto** $i > 0$ e $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i \leftarrow i-1$	c_7	?
8	$A[i+1] \leftarrow chave$	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** $comprimento[A]$ **faça**	c_1	n
3	$chave \leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**enquanto** $i > 0$ e $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow chave$	c_8	?

Vamos então contar, a entrada possui tamanho n:

#	*INSERTION-SORT*(A)	custo	vezes
2	**para** $j \leftarrow 2$ **até** *comprimento*$[A]$ **faça**	c_1	n
3	*chave* $\leftarrow A[j]$	c_2	$n-1$
	$\triangleright$ Ins. $A[j]$ na seq. ord. $A[1 \cdot \cdot j-1]$	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**enquanto** $i > 0$ e $A[i] >$*chave* **faça**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$*chave*	c_8	$n-1$

Precisamos agora somar tudo isto!

Calculando o tempo de execução

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

Podemos perceber que o resultado irá depender da soma, ou seja, quantas vezes o loop **enquanto** será executado para cada elemento da entrada.

Não há como saber com exatidão, podemos apenas considerar o melhor e o pior caso.

Calculando o tempo de execução

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

Podemos perceber que o resultado irá depender da soma, ou seja, quantas vezes o loop **enquanto** será executado para cada elemento da entrada.

Não há como saber com exatidão, podemos apenas considerar o melhor e o pior caso.

No melhor caso a seqüência já está ordenada, logo $t_j = 1$ para $j = 2, 3, 4, ..n$

Calculando o tempo de execução

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

Podemos perceber que o resultado irá depender da soma, ou seja, quantas vezes o loop **enquanto** será executado para cada elemento da entrada.

Não há como saber com exatidão, podemos apenas considerar o melhor e o pior caso.

No pior caso a seqüência está na ordem inversa, cada elemento da posição j deve ser comparado com cada um dos $A[1 \cdot \cdot j - 1]$, ou seja, $j - 1$ comparações. Então $t_j = j$ para $j = 2, 3, 4, ..n$

Calculando o tempo de execução **no melhor caso**

$$
\begin{aligned}
T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot (n-1) + c_6 \cdot 0 \\
&\quad + c_7 \cdot 0 + c_8 \cdot (n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

Podemos resumir a: $T(n) = a \cdot n + b$, para o algoritmo executando no melhor caso.

Calculando o tempo de execução **no pior caso**

Primeiro vamos considerar as somas:
$\sum_{j=2}^{n} j = \frac{n(n-1)}{2} - 1$ e $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

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

Podemos resumir a: $T(n) = a \cdot n^2 + b \cdot n + c$, para o algoritmo executando no pior caso.

Qual a eficiência? Melhor ou Pior caso?

Normalmente é considerado o pior caso, pelos seguintes motivos:

- O tempo de execução no pior caso é um limite superior de tempo para qualquer entrada (não será pior que isto).
- Em muitos problemas o pior caso ocorre com bastante freqüência.
- Em muitos problemas o "caso médio" é quase tão ruim quanto o pior caso. (Por exempo a ordenação por inserção).

A análise do caso médio nem sempre é simples de realizar.

Representação de eficiência através de uma notação assintótica

Como vimos, o tempo de execução para o algoritmo de Ordenação por Inserção é representado por: $T(n) = an^2 + bn + c$. Podemos considerar apenas o termo mais dominante para estudo da ordem de crescimento do tempo de execução com relação ao número de entradas.

Podemos dizer que $T(n) = \Theta(n^2)$.

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

O tempo de execução do algoritmo de Ordenação por Inserção pertence a uma classe de funções que crescem com o quadrado do número de entradas.

Atividades baseadas no CLRS.

1. Ler Capítulos 1 (completo), 2.1 e 2.2
2. Resolver exercícios: 1.1-4, 1.2-3, Problema 1-1.
3. Resolver exercícios: 2.1-1, 2.1-2, 2.1-3, 2.2-1, 2.2-2, 2.2-3

Resolver a 1ª Lista de Exercícios.