Notas de aula: Análise de Algoritmos
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Invariante de laço e notação assintótica
(*Última atualização: 25 de março de 2019*)

Referências e materiais complementares desse tópico

Livro Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C.. Introduction to Algorithms. 2nd ed. MIT Press. 2002.
Capítulo 2.1, 2.2, 3.1.

Livro Dasgupta, S.; Papadimitriou, C.; Vazirani, U.. **Algorithms**. Boston: McGraw-Hill. 2008.
Capítulo 0.3.

Vídeos Vídeo aulas do prof. Tim Roughgarden, de Stanford, em inglês (com legendas)
Aulas **1.8**, **2.1**, **2.2**, **2.3**, **2.4** e **2.5**.

Sumário

1 Invariante de laço

- É uma afirmação, uma frase, sobre um laço (*for*, *while*) de um algoritmo tal que:
  1. ela é válida antes da primeira iteração do laço;
  2. se ela vale antes de uma iteração, então ela vale antes da próxima.
- É útil quando, após o término da execução do laço, fornece propriedades que ajudam a mostrar que o algoritmo está correto.

2 Exemplo

- **Vetores** são estruturas de dados simples que armazenam um conjunto de objetos do mesmo tipo de forma contínua na memória.

- Essa forma de armazenamento permite que o acesso a um elemento do vetor possa ser feito de forma direta, através do *índice* do elemento.
- Um vetor A com capacidade para n elementos será representado por $A[1..n]$ ou $A = (a_1, a_2, \ldots, a_n)$ e $A[i] = a_i$ é o elemento que está armazenado na posição i, para todo $1 \leq i \leq n$.
- Para quaisquer $1 \leq i < j \leq n$, denotamos por $A[i..j]$ o subvetor de A que contém os elementos $A[i], A[i+1], \ldots, A[j]$.

> PROBLEMA: ORDENAÇÃO
>
> ENTRADA: um vetor $A[1..n]$ com n números reais.
> SAÍDA: Uma permutação (reordenação) dos elementos de A em um vetor $A'[1..n]$ tal que $A'[1] \leq A'[2] \leq \cdots \leq A'[n]$.

- Um algoritmo dos mais simples para o problema de ordenação usa a ideia de ordenação por inserção.
  - Considerando um elemento por vez, do primeiro ao último, insira-o na posição correta *relativa aos elementos que já foram considerados*.
  - Em outras palavras: para um certo i, com $1 \leq i \leq n$, sabendo que o subvetor $A[1..i-1]$ já está ordenado, queremos deixar o subvetor $A[1..i]$ ordenado, movendo o elemento $A[i]$.
  - Considere o vetor $A = (6, 1, 8, 7, 3, 9, 5, 2, 4)$ e veja uma execução do algoritmo mencionado a seguir. Para cada linha, o elemento $A[i]$ considerado está em negrito. Os elementos sublinhados já estão ordenados ($A[1..i-1]$).

6	**1**	8	7	3	9	5	2	4
1	6	**8**	7	3	9	5	2	4
1	6	8	**7**	3	9	5	2	4
1	6	7	8	**3**	9	5	2	4
1	3	6	7	8	**9**	5	2	4
1	3	6	7	8	9	**5**	2	4
1	3	5	6	7	8	9	**2**	4
1	2	3	5	6	7	8	9	**4**
1	2	3	4	5	6	7	8	9

  - Veja uma **animação** desse algoritmo.
- Esse algoritmo é conhecido como INSERTIONSORT e ele é formalmente apresentado no Algoritmo 1.
- A seguir analisaremos as três perguntas que devemos fazer sempre que descrevemos um algoritmo para um problema.
- O INSERTIONSORT está correto? Isto é, ele ordena **qualquer** vetor A de números reais?
  - Considere a seguinte frase sobre o Algoritmo 1: "antes de cada iteração do laço **para** (indexado por i), o subvetor $A[1..i-1]$ contém os elementos contidos originalmente em $A[1..i-1]$ mas em ordem não-decrescente".

Algoritmo 1 Algoritmo de inserção para o problema de ordenação.

```
1: função InsertionSort(A, n)
2:     para i ← 2 até n faça
3:         atual ← A[i]
4:         j ← i − 1
5:         enquanto j ≥ 1 e A[j] > atual faça
6:             A[j + 1] ← A[j]
7:             j ← j − 1
8:         A[j + 1] ← atual
9:     devolve A
```

  - Ela é uma invariante de laço, pois
    - ela é verdadeira antes da primeira iteração, onde $i = 2$: o subvetor $A[1..1]$ contém apenas o elemento $A[1]$ e, portanto, trivialmente está ordenado;
    - suponha que ela vale antes de uma certa iteração (fixe um valor de i qualquer entre 2 e n), isto é, que o subvetor $A[1..i-1]$ contém os elementos originais em ordem. Note que durante essa iteração, o elemento $A[i]$ é "movido" para a esquerda pelo laço **enquanto**, para alguma posição onde todos os elementos à sua direita (até a posição i) são maiores do que ele e os elementos à sua esquerda são menores. Com isso, o subvetor $A[1..i]$ fica ordenado e contém os elementos originalmente naquelas posições, ou seja, a invariante se mantém verdadeira para antes da próxima iteração.
  - E note que, ao fim do laço **para**, que termina deixando $i = n + 1$, temos que a invariante é verdadeira (pois a última iteração a manteve verdadeira para a próxima – que, no caso, é inexistente). Assim, vale que o subvetor $A[1..n]$, ou seja, o vetor todo, está ordenado e contém todos os elementos do vetor inicial.

- A análise anterior formalmente demonstra que o algoritmo InsertionSort está correto.
- Exercício: Escreva um algoritmo para o seguinte problema: dados dois números a e b de n dígitos cada, produza $a + b$. Prove que seu algoritmo está correto por meio de uma invariante de laço.
- Quanto tempo esse algoritmo leva?
  - Seja c_k uma constante que indica o tempo de execução (lembre-se, número de passos básicos) da linha k do Algoritmo 1.
  - Seja t_i o número de vezes que o laço **enquanto** executa para um dado valor de i do laço **para**.
  - Seja $T(n)$ o tempo total de execução do InsertionSort sobre um vetor de tamanho n. Note que $T(n)$ é simplesmente a soma dos tempos de cada linha (que por sua vez é o custo da linha multiplicado pelo número de vezes que ela executa):

$$T(n) = c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \quad . \tag{1}$$

  - Agora perceba que a linha 2 sempre executará n vezes e as linhas 3, 4 e 8 $n - 1$ vezes, independente da entrada recebida.

– Se o vetor de entrada já estiver ordenado, no entanto, o laço **enquanto** nunca será executado (o teste da linha 5 será realizado uma vez, de modo que $t_i = 1$). Portanto, nesse caso temos que

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_2 + c_3 + c_4 + c_5 + c_8)n - (c_3 + c_4 + c_5 + c_8) = an + b \ . \end{aligned}$$

– Por outro lado, se o vetor de entrada estiver ordenado de forma decrescente, então o laço **enquanto** irá executar o maior número de vezes possível, de forma que $t_i = i$. Nesse caso, temos que

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} i + c_6 \sum_{i=2}^{n} (i-1) + c_7 \sum_{i=2}^{n} (i-1) + c_8(n-1) \\ &= c_2 n + c_3(n-1) + c_4(n-1) + c_5 \frac{n^2+n-2}{2} + c_6 \frac{n^2-n}{2} + c_7 \frac{n^2-n}{2} + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_2 + c_3 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_3 + c_4 + c_5 + c_8) \\ &= a'n^2 + b'n + c' \ . \end{aligned}$$

– Agora, cada uma das $n!$ permutações dos n elementos tem a mesma chance de ser o vetor de entrada (os vetores ordenados de forma crescente ou decrescente são apenas dois casos muito especiais), isto é, cada número tem a mesma probabilidade de estar em qualquer uma das n posições do vetor. Isso significa que, em média, metade dos elementos do subvetor $A[1..i-1]$ são menores do que $A[i]$, o que faz com que $t_i = i/2$, em média. Analisando a expressão $T(n)$ da Equação 1 novamente, ainda teremos algo da forma $an^2 + bn + c$.

3 Tempo de execução de um algoritmo

- É definido como o número de passos básicos em função do **tamanho da entrada**.

  – Depende do problema que está sendo estudado. No caso de muitos problemas, como ordenação, a medida mais natural é o número de elementos na entrada (por exemplo, o tamanho n do vetor). Em outros problemas, como da multiplicação de dois inteiros, a melhor medida é o número total de bits necessários para representar a entrada em notação binária.

- Mesmo para entradas de um mesmo tamanho, pode depender de *qual* entrada.

  – **Pior caso**: é o maior tempo de execução entre todas as entradas de um mesmo tamanho.

    * Permite fazer **garantias** sobre o tempo de execução do algoritmo: ele nunca vai levar mais tempo do que o tempo do pior caso.
    * No InsertionSort, acontece quando o vetor está ordenado de forma decrescente.

  – **Caso médio**: é a média do tempo de execução entre todas as entradas de um mesmo tamanho.

    * Permite fazer **previsões**, mas requer análise mais complicada que envolva probabilidade e suposições sobre a distribuição das entradas.

  - **Melhor caso**: é o menor tempo de execução entre todas as entradas de um mesmo tamanho.
    * Um algoritmo nunca vai levar menos tempo do que o tempo do melhor caso.

4 Como comparar dois algoritmos?

- Pela **ordem de crescimento** das funções que descrevem seus **tempos de execução**.
  - No problema do número de Fibonacci, um dos algoritmos tinha tempo maior do que $2^{0.684n}n$ e o outro tinha tempo cn^2, onde c é uma constante que não depende de n.

5 Como saber se um algoritmo é bom?

- Considere por um momento que um algoritmo é *eficiente* se ele executa rapidamente sobre instâncias reais. Mas:
  - onde e o quão bem implementado foi esse algoritmo? Mesmo algoritmos ruins podem executar rapidamente se executados sobre casos de teste pequenos ou processadores muito rápidos.
  - o que é uma instância real? Não sabemos todas as instâncias que serão encontradas na prática, e algumas podem ser bem mais difíceis do que outras.
  - quão bem esse algoritmo escala? Ele pode ser rápido para instâncias de tamanho 100 e muito ruim para instâncias de tamanho 1000 ou 10000.
- Então a definição de eficiência precisa considerar independência de plataforma, de instância e ser previsível conforme o tamanho da instância aumenta.
- Vamos então considerar que um algoritmo é eficiente se seu tempo de execução no pior caso pode ser descrito por uma função que cresce bem devagar com o tamanho da entrada.
  - Se c e d são constantes e n é o tamanho da entrada, cn cresce mais devagar do que dn^3, conforme n fica muito grande.
  - Perceba que as constantes que vimos nos exemplos anteriores dependem da arquitetura/compilador/programador/linguagem.

6 Notação assintótica

- Ferramenta para analisar algoritmos e descrever a ordem de crescimento dos tempos de execução.
- Provê formalismo e vocabulário matemático que nos permitem argumentar sobre a qualidade e eficiência de algoritmos.
- O importante é o comportamento dos algoritmos **conforme o tamanho da entrada cresce** indefinidamente (caso contrário força bruta resolve).

  - Por exemplo, no pior caso o InsertionSort leva tempo an^2+bn+c para constantes a, b e c, sendo que o termo realmente importante quando n cresce é o n^2.
  - Dizemos então que o tempo de execução no pior caso é "da *o*rdem de n^2".
  - Exercício: plote as funções $an^2 + bn + c$ e n^2 para vários valores de a, b e c e compare-as.

- A notação assintótica nos permite ignorar os detalhes de implementação (as constantes), sobre os quais não temos controle.
- Assim, a ideia principal da notação assintótica é **ignorar constantes e termos de mais baixa ordem**.
  - Mas ela é precisa o bastante para nos permitir decidir entre diferentes algoritmos para resolver um problema.
- Exercício: volte ao Algoritmo 1 e faça uma análise mais rápida, agora levando em conta que constantes são desprezíveis e que a ordem de crescimento é mais importante.
- Exercício: O problema da busca é descrito a seguir: dados um vetor A com n elementos e um inteiro x, devolva i caso exista algum i tal que $A[i] = x$ e devolva -1 caso contrário. Um algoritmo *sequencial* para resolvê-lo percorre o vetor da posição 1 até a posição n comparando cada elemento com x. Escreva um pseudocódigo para a busca sequencial e descreva seu tempo de execução. Prove que seu algoritmo está correto por meio de uma invariante de laço.

6.1 Exercícios

Descreva a ordem do tempo de execução dos algoritmos a seguir, justificando sua resposta.

```
1: para i ← 1 até n faça
2:     se A[i] = t então
3:         devolve verdadeiro
4: devolve falso
```

```
1: para i ← 1 até n faça
2:     se A[i] = t então
3:         devolve verdadeiro
4: para i ← 1 até n faça
5:     se B[i] = t então
6:         devolve verdadeiro
7: devolve falso
```

```
1: para i ← 1 até n faça
2:     para j ← 1 até n faça
3:         se A[i] = B[j] então
4:             devolve verdadeiro
5: devolve falso
```

```
1: para i ← 1 até n faça
2:     para j ← i + 1 até n faça
3:         se A[i] = A[j] então
4:             devolve verdadeiro
5: devolve falso
```

6.2 Notação O

- Sejam $f(n)$ e $g(n)$ duas funções cujo domínio são os números naturais.
- Dizemos que $f(n) = O(g(n))$, ou "$f(n)$ é $O(g(n))$", se existirem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.
- Isto é, para n suficientemente grande (maior do que n_0), $f(n)$ é **limitada superiormente** por $cg(n)$.
- Por exemplo, se $f(n) = 10n$ e $g(n) = n^2$, então vale que $f(n) = O(g(n))$, pois tomando $c = 5$ e $n_0 = 2$, temos que $10n \leq 5n^2$ para todo $n \geq 2$.
- Como essa notação dá um limite superior, se usarmos ela para descrever o tempo de execução de pior caso de um algoritmo, então teremos um limitante superior no tempo de execução de qualquer entrada.

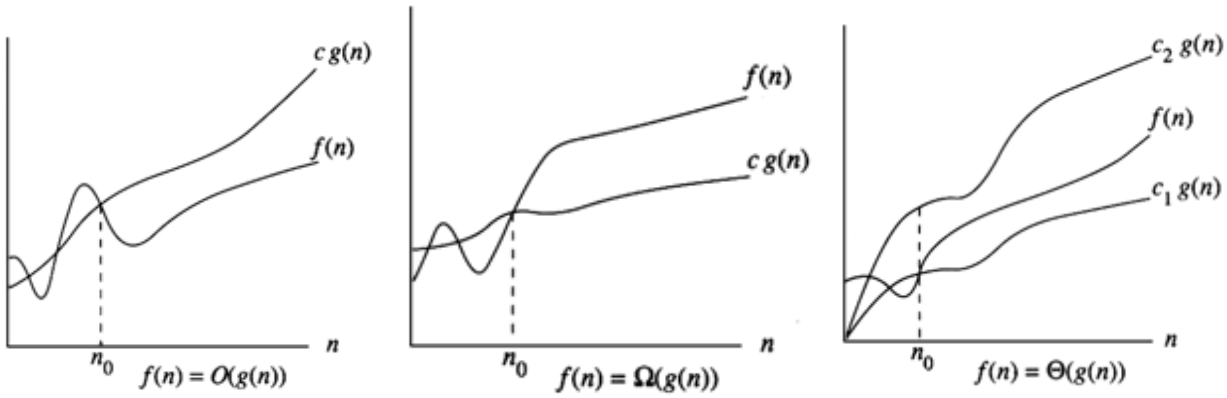
6.3 Notação Ω

- Sejam $f(n)$ e $g(n)$ duas funções cujo domínio são os números naturais.
- Dizemos que $f(n) = \Omega(g(n))$, ou "$f(n)$ é $\Omega(g(n))$", se existirem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$.
- Isto é, para n suficientemente grande (maior do que n_0), $f(n)$ é **limitada inferiormente** por $cg(n)$.
- Por exemplo, se $f(n) = 10n^2$ e $g(n) = n^2$, então vale que $f(n) = \Omega(g(n))$, pois tomando $c = 10$ e $n_0 = 1$, temos que $10n^2 \leq 10n^2$ para todo $n \geq 1$.
- Como essa notação dá um limite inferior, se usarmos ela para descrever o tempo de execução de melhor caso de um algoritmo, então teremos um limitante inferior no tempo de execução de qualquer entrada.

6.4 Notação Θ

- Sejam $f(n)$ e $g(n)$ duas funções cujo domínio são os números naturais.
- Dizemos que $f(n) = \Theta(g(n))$, ou "$f(n)$ é $\Theta(g(n))$", se existirem constantes positivas c_1, c_2, e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$.
- Não é difícil perceber que $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

- Chamamos $g(n)$ de limitante assintótico apertado para $f(n)$.



6.5 Exemplos

- Podemos usar as definições formais para verificar que $6n^3 \neq O(n^2)$, por exemplo.
  - Vamos supor, para fins de contradição, que existem constantes c e n_0 tais que $6n^3 \leq cn^2$ para todo $n \geq n_0$.
  - Então deve ser o caso que $c \geq 6n$ para todo $n \geq n_0$.
  - Quer dizer, a constante c, que é um valor *independente de* n, tem um valor que é maior ou igual a $6n$, onde n é algo que *cresce indefinidamente*.
  - Isso é claramente um absurdo. Logo, $6n^3 \neq O(n^2)$.
- Com frequência usamos $\Theta(1)$, $\Omega(1)$ e $O(1)$ para denotar tempos de execução *constantes*.
- $10n + \log n + 15$ é $\Omega(n + \log^2 n)$. Como provar isso?
  - Uma forma possível é tentar desenvolver a expressão $10n + \log n + 15$ por meio de "$\geq$", tendo em mente que o objetivo é chegar em algo do tipo $c(n + \log^2 n)$ onde c é uma constante:

$$\begin{aligned} 10n + \log n + 15 &= 5n + 5n + \log n + 15 \\ &\geq 5n + 5\log^2 n + \log n + 15 \\ &\geq 5n + 5\log^2 n \\ &= 5(n + \log^2 n) \ . \end{aligned}$$

    Note que o primeiro "$\geq$" é verdadeiro (isto é, $n \geq \log^2 n$) apenas quando $n \geq 16$ e o segundo é verdadeiro (isto é, $\log n + 15 \geq 0$) para qualquer valor de $n \geq 0$. Sendo assim, tomamos $n_0 = 16$ e $c = 5$.
  - Outra forma possível é assumir que $10n + \log n + 15 \geq c(n + \log^2 n)$ e tentar encontrar os valores de c e n_0 (é preciso sempre encontrar esses valores):

$$10n + \log n + 15 \geq c(n + \log^2 n) \quad \Rightarrow \quad c \leq \frac{10n + \log n + 15}{n + \log^2 n} \ .$$

    Nesse caso não há mais simplificações a serem feitas na expressão. Basta agora encontrar um valor c que é menor do que $\frac{10n+\log n+15}{n+\log^2 n}$, qualquer que seja o valor de n, contanto que $n \geq n_0$.

Considere que log está na base 2. Uma análise na função $f(n) = \frac{10n+\log n+15}{n+\log^2 n}$ nos mostra que quando $n = 16$ temos $f(n) = 5.5937$ e para qualquer $n > 16$ temos $f(n) > 5.5937$. Assim, se tomarmos $c = 4$ (ou 5, ou 3, ou 2) e $n_0 = 16$, garantimos que $c \leq \frac{10n+\log n+15}{n+\log^2 n}$ vale sempre que $n \geq n_0$.

- Observe atentamente as seguintes afirmações:
  - O pior caso do InsertionSort, $T(n) = an^2 + bn + c$, é $O(n^2)$, pois $an^2 + bn + c \leq an^2 + bn^2 + cn^2 = (a + b + c)n^2$ para todo $n \geq 1$. Se o tempo no pior caso é $O(n^2)$, então o tempo do InsertionSort é $O(n^2)$.
  - O melhor caso do InsertionSort, $T(n) = an + b$, é $\Omega(n)$, pois $an + b \geq an$ para todo $n \geq 1$. Se o tempo no melhor caso é $\Omega(n)$, então o tempo do InsertionSort é $\Omega(n)$.
  - Também é verdade que $T(n) = an^2 + bn + c$ é $\Omega(n^2)$, pois $an^2 + bn + c \geq an^2$ para todo $n \geq 1$. Ou seja, o tempo no pior caso do InsertionSort é, de fato, $\Theta(n^2)$.
  - De forma equivalente, $T(n) = an + b$ é $O(n)$, pois $an + b \leq an + bn = (a + b)n$ para todo $n \geq 1$. Assim, o tempo no melhor caso do InsertionSort é $\Theta(n)$.
  - **Não é verdade**, no entanto, que o tempo do InsertionSort é $\Theta(n^2)$, já que vimos que existe uma entrada (vetor já ordenado) para a qual o algoritmo executa em tempo $\Theta(n)$.
- É verdade que $4n^2 = O(n^2)$?
- É verdade que $2^{0.694n} = O(2^n)$?
- É verdade que $10n^2 + 5n + 3 = \Theta(n^2)$?
- É verdade que $7n^2 = O(n)$?
- É verdade que $\frac{1}{2}n^2 + 3n = \Omega(n)$?
- É verdade que $\frac{1}{2}n^2 + 3n = O(n^3)$?
- É verdade que $n^k = O(n^{k-1})$?
- É verdade que $a_k n^k + \cdots + a_1 n + a_0 = O(n^k)$?
- É verdade que $100n + \log n = \Theta(n + \log^2 n)$?
- É verdade que $\log 2n = \Theta(\log 3n)$?
- É verdade que $\log_a n = \Theta(\log_b n)$ com a e b inteiros positivos?
- É verdade que $(n + a)^b = \Theta(n^b)$ com a e b inteiros positivos?
- É verdade que $4 \log n + 3\sqrt{n} + 5n^2 = \Theta(n^2)$?
- É verdade que $2^{5+n} = O(2^n)$?

7 Voltando ao InsertionSort

- Dá para fazer melhor?
  - Sim!