

Análise de Algoritmos

Vinicius A. Matias

May 3, 2021

1 Introdução

Muitos problemas reais podem ser aplicados computacionalmente por meio de algoritmos. A área de análise de algoritmos visa estudar e projetar algoritmos com base no tempo e espaço ocupado para uma solução ótima (de menor custo possível).

As maneiras mais comuns para medir o custo de algoritmos são: medição direta (medir o tempo de processamento com base no tempo real, logo, é influenciado pelo hardware); custo baseado em um computador ideal (valores tabulados por linguagem de programação para medir o custo de cada operação); e por meio das operações mais significativas (mais utilizada, focando em identificar as operações que aumentam o custo do algoritmo).

2 Função de complexidade

Para a análise de complexidade seguindo a operação de maior custo, podemos definir uma função de complexidade $f(n)$, onde n é o tamanho da entrada e $f(n)$ é o número de comparações necessárias para resolver um problema. Vamos exemplificar o problema utilizando o trecho de código em Python na Listing 1. Este código recebe um vetor ou lista A , que tem tamanho n (determinado por pelo comando `len()`).

Listing 1: Maior valor de um arranjo

```
1 def max_array(A):
2     max = A[0]
3     i = 1
4
5     while i < len(A):
6         if A[i] > max:
7             max = A[i]
8             i += 1
9
10    return max
```

A operação crítica para este algoritmo é determinada pelo `if` da linha 6, cuja troca pode ser feita, no pior caso, $n-1$ vezes. Note que antes de se iniciar o loop, `max` é definido como o primeiro valor do arranjo, consequentemente, é desnecessário utilizar este valor no `while` (logo, o loop começa do segundo valor e vai até o último, com $n-1$ comparações). Portanto, a função de complexidade para este algoritmo é $f(n) = n - 1, \forall n > 0$. Ainda não foram tratadas as técnicas para definir se um algoritmo é ótimo, mas no caso deste algoritmo, já foi provado que o mínimo de operações necessárias é $n-1$ (para um arranjo desordenado), logo, este é um algoritmo ótimo.

Projetemos agora um novo algoritmo que calcule o máximo e mínimo de um arranjo no mesmo laço (Listing2). Para desenvolvê-lo reaproveitamos o código do máximo valor em um arranjo e adicionamos uma segunda comparação, caso a primeira tenha falhado (isto é, se o valor na posição atual não for o maior, verificamos se é menor).

Listing 2: Maior e menor valor de um arranjo

```
1 def max_min_array(A):
2     max = min = A[0]
3     i = 1
4
5     while i < len(A):
6         if A[i] > max:
7             max = A[i]
8         elif A[i] < min:
9             min = A[i]
10        i += 1
11
12    return [max, min]
```

Pela análise do algoritmo, percebe-se que o melhor caso (menor número de comparações) ocorre quando realizamos apenas o primeiro `if`, isto é, apenas comparamos o valor atual do arranjo com o maior valor registrado até o momento. Este caso ocorre quando o arranjo é passado ordenado, logo, o valor mínimo nunca será

alterado e o valor máximo sempre será trocado, resultando em $n - 1$ operações.

O pior caso é quando realizamos a segunda operação em todas as iterações. Para isto acontecer basta que o primeiro valor do arranjo seja o valor máximo, portanto, o primeiro teste sempre irá falhar e o segundo sempre será executado. Importante notar que o pior caso inclui o arranjo em ordem decrescente, mas não somente. Dado que o laço corre $n - 1$ vezes e realizamos duas comparações nele, nosso algoritmo tem $f(n) = 2(n - 1)$. Novamente, tanto o melhor caso quanto o pior caso são dados para todo $n > 0$.

Tipicamente, estamos interessados em identificar o custo do algoritmo no pior caso, mas técnicas para determinar a complexidade de algoritmos no caso médio, melhor e pior caso serão discutidas nos próximos tópicos.

2.1 Exercícios

1. Determine a função de complexidade da busca sequencial de um vetor A de tamanho n para o melhor caso, pior caso e caso médio.

Resolução:

- Pior caso: a busca passará pelos n elementos, logo, $f(n) = n$
- Melhor caso: o primeiro elemento é o valor buscado, logo, $f(n) = 1$
- Caso médio: Para este problema, podemos dizer que devemos passar por 50% dos elementos para encontrar o valor, portanto $f(n) = n/2$

Listing 3: Busca sequencial

```

1 def linear_search(A, target):
2     n = len(A)
3
4     for i in range(n):
5         if A[i] == target:
6             return i
7
8     return -1

```

3 Crescimento Assintótico

Como já deve ter ficado claro, as funções de complexidade dependem de n, o que deve ser o responsável por aumentar o tempo de execução

do algoritmo. A tabela 1 mostra o crescimento na quantidade de operações para três n's em três diferentes funções de complexidade. O algoritmo para retornar os valores da tabela foi construído em R.

Table 1: Comportamento Assintótico

	100	1000	10^6
$\log n$	2	3	6
n	100	1000	1e+06
$n \log n$	200	3000	6e+06
n^2	1e+04	1e+06	1e+12
$100n^2 + 15n$	1e+06	1e+08	1e+14
2^n	1.3e+30	1e+301	Inf

Note que a entrada n sempre aumentará a quantidade de operações, mas a função de complexidade interfere muito mais no aumento do custo. Na análise da complexidade dos algoritmos nos interessará encontrar, por exemplo, a partir de qual valor de n uma função se torna maior que outra (ou, em palavras bonitas, quando uma função domina assintoticamente outra função). Veja na figura 1 um exemplo. Caso necessário, dê zoom na imagem, mas o eixo x representa os valores de n (0 à 100) e y a quantidade de operações. A partir de $n = 5$ a função vermelha passa a crescer mais que a função azul mediante o aumento do n. À título de curiosidade, a função vermelha cresce em x^3 e a função azul cresce em $5x$.

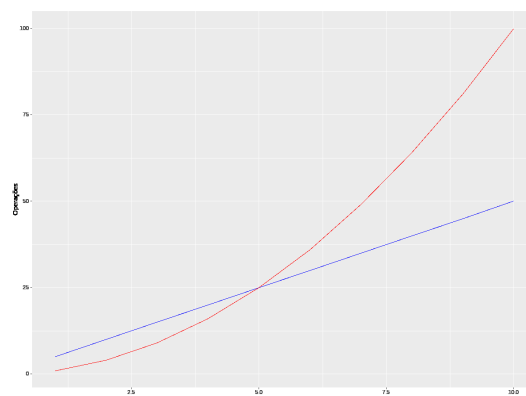


Figure 1: Crescimento no número de operações para diferentes valores de n em duas funções de complexidade

4 Notação \mathcal{O}

A notação \mathcal{O} (leia-se O grande ou Big-O) diz que $f(n)$ cresce no máximo ou tanto quanto $g(n)$ pela notação $f(n) = \mathcal{O}(g(n))$.

Definition 4.1. $\mathcal{O}(g(n)) = \{f(n): \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n), \text{ para todo } n \geq n_0\}$

4.1 Exemplo

Demonstrar que $f(n) = \frac{3}{2}n^2 - 2n \in \mathcal{O}(n^2)$.

Isso é o mesmo que dizer que existem constantes positivas c e n_0 onde: $0 \leq f(n) \leq cg(n)$

Resolvendo $0 \leq f(n)$:

Step 1: $0 \leq \frac{3}{2}n^2 - 2n$

Step 2: $0 \leq n(\frac{3}{2}n - 2)$

Como $n \geq 0$, a multiplicação $n(\frac{3}{2}n - 2)$ vai ser maior ou igual à zero se $\frac{3}{2}n - 2$ for maior ou igual à zero. Resolvendo essa multiplicação temos:

Step 3: $0 \leq \frac{3}{2}n - 2n$

Step 4: $2 \leq \frac{3}{2}n$

Step 5: $\frac{2}{3/2} \leq n$

Step 6: $\frac{4}{3} \leq n$

Logo, n deve ser maior ou igual à $4/3$

Resolvendo $\frac{3}{2}n^2 - 2n \leq cn^2$ (note que $g(n) = n^2$ para $\mathcal{O}(n^2)$):

Step 1: $\frac{3}{2}n^2 - 2n \leq cn^2$

Step 2: $n(\frac{3}{2}n - 2) \leq cn^2$

Step 3: $\frac{3}{2}n - 2 \leq cn$

Step 4: $-2 \leq cn - \frac{3}{2}n$

Step 5: $cn - \frac{3}{2}n \geq -2$

Step 6: $n(c - \frac{3}{2}) \geq -2$

Como $n \geq 0$, a multiplicação $n(c - \frac{3}{2})$ é maior ou igual que -2 quando $(c - \frac{3}{2}) \geq 0$

Step 7: $c - \frac{3}{2} \geq 0$

Step 8: $c \geq \frac{3}{2}$

Logo, $c \geq 3/2$ e $n \geq 0$

Como $n \geq 4/3, n \geq 0$ e $c \geq 3/2$, podemos escolher constantes n_0 e c que satisfaçam essas condições. Como exemplo:

$n_0 = 2, c = 3/2$

5 Notação Ω

A notação Ω (Omega) significa que $f(n)$ cresce mais ou tão rápido quanto $g(n)$, ou que $f(n)$ domina $g(n)$

Definition 5.1. $\Omega(g(n)) = \{f(n): \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n), \text{ para todo } n \geq n_0\}$

5.1 Exemplo

Demonstrar que $f(n) = \frac{3}{2}n^2 - 2n \in \Omega(n^2)$.

Isso é o mesmo que dizer que existem constantes positivas c e n_0 onde: $0 \leq cg(n) \leq f(n)$

Resolvendo $0 \leq cg(n)$:

Step 1: $0 \leq cn^2$

Se $n \geq 0$, a condição também é satisfeita para $c \geq 0$

Resolvendo $cg(n) \leq f(n)$

Step 1: $cn^2 \leq \frac{3}{2}n^2 - 2n$

Step 2: $cn^2 \leq n(\frac{3}{2}n - 2)$

Step 3: $cn \leq \frac{3}{2}n - 2$

Step 4: $2 \leq \frac{3}{2}n - cn$

Step 5: $2 \leq n(\frac{3}{2} - c)$

Como n e c são maiores que 0, a inequação será verdadeira para $(\frac{3}{2} - c) > 0$ e um consequente valor mínimo de n .

Step 6: $(\frac{3}{2} - c) > 0 \rightarrow c < \frac{3}{2}$

Como precisamos de um $c < 3/2$, consideraremos $c = 1$ aplicado ao passo 5

Step 7: $2 \leq n(\frac{3}{2} - 1)$

Step 8: $2 \leq n(\frac{1}{2})$

Step 9: $4 \leq n \rightarrow n \geq 4$

Logo, $\frac{3}{2}n^2 - 2n \in \Omega(n^2)$ para $n_0 = 4$ e $c = 1$

6 Notação Θ

A notação Θ (Theta) diz que uma função de complexidade $f(n)$ cresce no mínimo tão lentamente quanto $c_1g(n)$ e no máximo tão rapidamente quanto $c_2g(n)$, logo, $f(n)$ deverá crescer junto à $g(n)$, ou seja, tão rapidamente quanto $g(n)$.

Definition 6.1. $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ para todo } n \geq n_0\}$

Note que $0 \leq c_1 g(n) \leq f(n)$ é a definição de \mathcal{O} para um $c = c_1$ ($f(n)$ domina $g(n)$), e que $0 \leq f(n) \leq c_2 g(n)$ é a definição de Ω para $c = c_2$ ($g(n)$ domina $f(n)$). A partir disso pode-se concluir duas coisas.

A primeira conclusão é algo que foi mencionado no parágrafo anterior à definição: $\Theta(g(n))$ significa que uma função cresce tão rapidamente quanto $g(n)$. Se $f(n)$ domina e é dominado por $g(n)$, $f(n)$ deve crescer tão rapidamente quanto $g(n)$.

A segunda conclusão é que para definir as constantes de $\Theta(g(n))$, podemos utilizar as constantes c e n_0 encontradas em $\mathcal{O}(g(n))$ e $\Omega(g(n))$, tomando o cuidado de se escolher um n_0 que satisfaça as condições de $\mathcal{O}(g(n))$ e $\Omega(g(n))$.

6.1 Exemplo

Demonstrar que $f(n) = \frac{3}{2}n^2 - 2n \in \Theta(n^2)$.

Isso é o mesmo que dizer que existem constantes positivas c_1, c_2 e n_0 onde:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Como $cf(n) \in \mathcal{O}(g(n))$ para $c = \frac{3}{2}$ e $n_0 = 2$;

e

$$cf(n) \in \Omega(g(n)) \text{ para } c = 1 \text{ e } n_0 = 4$$

Podemos concluir que $f(n) = (3/2)n^2 - 2n \in \Theta(n^2)$ para $c_1 = 1, c_2 = 3/2$ e $n_0 = 4$.

7 Notação o

A notação $o(g(n))$ ("o" minúsculo ou "o" pequeno) diz que uma função $f(n)$ cresce mais lentamente que $g(n)$.

Definition 7.1. $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid 0 \leq f(n) < cg(n), \forall n \geq n_0\}$

7.1 Exemplo

Demonstrar que $1000n^2 \in o(n^3)$ (uma função quadrática cresce mais lentamente que uma função cúbica)

Demonstrar que $f(n) \in o(g(n))$ é o mesmo que mostrar que para toda constante positiva c , existe um valor de $n_0 > 0$ que satisfaça a inequação $0 \leq f(n) < cg(n)$

Logo, deve-se encontrar uma função de n_0 em relação à c para $0 \leq 1000n^2 < cn^3$

Resolvendo $0 \leq 1000n^2$ identificamos que o resultado é válido para qualquer valor real de n .

Resolvendo $1000n^2 < cn^3$:

$$\text{Step 1: } 1000n^2 < cn^3$$

$$\text{Step 2: } 1000 < cn$$

$$\text{Step 3: } 1000/c < n$$

$$\text{Step 4: } n > 1000/c$$

Dada esta inequação, precisamos formular uma equação para n_0 que siga as diretrizes de n estritamente maior que $1000/c$. Uma equação possível é $n_0 = \frac{1000}{c} + 1$.

Assim, encontramos uma função de n_0 em função de c que satisfaz $0 \leq 1000n^2 < cn^3$, sendo essa a função $n_0 = \frac{1000}{c} + 1$.

8 Notação ω

A notação $\omega(g(n))$ (ômega pequeno) diz que uma função $f(n)$ cresce mais rapidamente que $g(n)$.

Definition 8.1. $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \mid 0 \leq cg(n) < f(n), \forall n \geq n_0\}$

8.1 Exemplo

Demonstrar que $\frac{n^2}{1000} \in \omega(n)$ (uma função quadrática cresce mais rapidamente que uma função linear)

Demonstrar que $f(n) \in \omega(g(n))$ é o mesmo que mostrar que para toda constante positiva c , existe um valor de $n_0 > 0$ que satisfaça a inequação $0 \leq cg(n) < f(n)$

Logo, deve-se encontrar uma função de n_0 em relação à c para $0 \leq cn < \frac{n^2}{1000}$

Resolvendo $0 \leq cn$ identificamos que o resultado é válido para qualquer valor positivo de n .

Resolvendo $cn < \frac{n^2}{1000}$

$$\text{Step 1: } cn < \frac{n^2}{1000}$$

$$\text{Step 2: } c < \frac{n}{1000}$$

$$\text{Step 3: } 1000c < n \rightarrow n > 1000c$$

Dada esta inequação, precisamos formular uma equação para n_0 que siga as diretrizes de n estritamente maior que $1000c$. Uma equação possível é $n_0 = 1000c + 1$.

Assim, encontramos uma função de n_0 em função de c que satisfaz $0 \leq cn < \frac{n^2}{1000}$, sendo essa a função $n_0 = 1000c + 1$.