Noções de complexidade de algoritmos

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Quando fazemos o projeto de uma função ou de um tipo de dado separamos a especificação (o que) da implementação (como).

Isso trás diversos benefícios, entre eles:

- Oculta a complexidade da implementação (abstração);
- Permite o desenvolvimento independente;
- Permite implementações alternativas.

Se podemos fazer a implementação de diversas maneirais, quais critérios podemos utilizar para escolher uma implementação?

- Simplicidade;
- Consumo de recurso (tempo, memória, energia, etc).

Formalmente, o consumo de recurso de um algoritmo é chamada de **complexidade do algoritmo**.

Para podermos determinar qual algoritmo é mais eficiente (tem menor complexidade), precisamos de:

- Determinar a complexidade;
- Expressar a complexidade;
- Comparar a complexidade.

O processo de determinar a complexidade de algoritmos é chamado de **análise de algoritmos**.

Para expressar e comparar complexidades de algoritmos vamos utilizar a **notação assintótica**.

Formas de análise

A análise de um algoritmo pode ser:

- Experimental;
- Teórica;

A análise experimental é mais específica pois dependente da linguagem, do compilador / interpretador, do hardware, etc.

A análise teórica (ou analítica) é mais geral e provê entendimento das propriedades e limitações inerentes ao algoritmo.

As duas formas de análise são complementares.

Nessa disciplina vamos focar na análise teórica.

Na **análise teórica** adotamos uma máquina teórica de computação e expressamos a complexidade de um algoritmo através de uma **função que relaciona o tamanho da entrada com o consumo de recurso** nessa máquina teórica.

A máquina teórica que vamos adotar tem operações lógicas e aritméticas, cópia de dados e controle de fluxo, e tem as seguintes características:

- As instruções são executadas uma por vez e em sequência;
- Cada operação é executa em uma unidade de tempo.

Em geral, não estamos procurando uma função precisa para a complexidade de um algoritmo, mas uma que descreve de forma razoável como o consumo do recurso cresce em relação ao crescimento do tamanho da entrada, o que chamamos de **ordem de crescimento**. Além disso, estamos interessados em entradas suficientemente grandes, para que o algoritmo demore algum tempo razoável para executar e não termine rapidamente.

Por esse motivo, em alguns casos podemos fazer simplificações na análise, como por exemplo, levar em consideração apenas as **operações que são mais executadas**.

Quando olhamos para entradas suficientemente grandes e consideramos relevante apenas a ordem de crescimento, estamos estudando a **eficiência assintótica** do algoritmo em relação ao uso de algum recurso.

Dessa forma, um algoritmo assintoticamente mais eficiente será a melhor escolha, exceto para entradas muito pequenas.

Exemplo máximo

```python
def maximo(lst: list[int]) -> int:
    '''
    Encontra o valor máximo
    de *lst*.

    Requer que *lst* seja não vazia.

    Exemplos
    >>> maximo([4, 1, 6, 2])
    6
    '''
    assert len(lst) != 0
    max = lst[0]
    for i in range(1, len(lst)):
        if max < lst[i]:
            max = lst[i]
    return max
```

Vamos fazer a análise da função máximo para determinar a sua complexidade de tempo, isto é, determinar como o tempo de execução ($T(n)$) está relacionado com o tamanho da entrada (n - quantidade de elementos de `lst`).

Quais são as operações mais executadas pela função? O incremente e comparação de `i` (que estão implícitos) e a comparação de `max`.

Quantas vezes a operação `<` é executada? $n - 1$.

Dessa forma, podemos dizer que a complexidade de tempo de `maximo` é $T(n) = n - 1$.

Entrada específica

O tempo de execução de um algoritmo pode depender não apenas do tamanho da entrada, mas do valor específico da entrada. Em outras palavras, para um *mesmo tamanho de entrada*, o tempo de execução pode mudar de acordo com os *valores da entrada*.

Exemplo contém

```python
def contem(lst: list[int], x: int) -> bool:
    '''
    Devolve True se *x* está em *lst*,
    False caso contrário.

    Exemplos
    >>> contem([4, 1, 2, 3], 1)
    True
    >>> contem([4, 1, 2, 3], 6)
    False
    '''
    contem = False
    i = 0
    while i < len(lst) and not contem:
        if lst[i] == x:
            contem = True
        i = i + 1
    return contem
```

Para uma entrada de tamanho n, quantas vezes a operação `==` é executada?

Depende dos valores entrada!

- Melhor caso: `x` é o primeiro de `lst`, 1 vez
- Pior caso: `x` não está em `lst`, n vezes
- Caso médio: considerando que `x` está em `lst` e tem a mesma probabilidade de estar em qualquer posição, $\frac{n+1}{2}$ vezes

Portanto, para o caso geral, a complexidade de tempo da função é $T(n) = n$.

Exemplo ordenação seleção

```
def ordena_selecao(lst: list[int]):
    '''Ordena os elementos de *lst*
    em ordem não decrescente.
    Exemplos
    >>> lst = [8, 1, 6, 3, 1]
    >>> ordena(lst)
    >>> lst
    [1, 1, 3, 6, 8]
    '''
    for i in range(0, len(lst) - 1):
        # Encontra o mínimo
        # e coloca na posição i
        min = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[min]:
                min = j
        t = lst[i]
        lst[i] = lst[min]
        lst[min] = t
```

Para uma entrada de tamanho n, quantas vezes a operação < é executada?

- Para $i = 0$, $n - 1$
- Para $i = 1$, $n - 2$
- Para $i = 2$, $n - 3$
- ...
- Para $i = n - 2$, 1

Portanto, o total de vezes que < é executado é

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

Portanto, a complexidade de tempo de `ordena_selecao` é $T(n) = \frac{n^2 - n}{2}$

Para podermos determinar qual algoritmo é mais eficiente (tem menor complexidade), precisamos de:

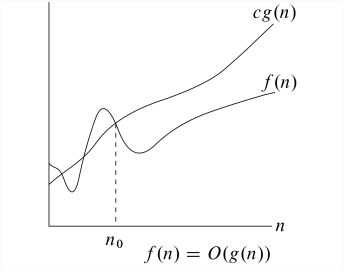
- Determinar a complexidade; Feito;
- Expressar a complexidade;
- Comparar a complexidade.

Agora vamos ver a **notação assintótica**, que permite expressar e comparar mais facilmente complexidades de tempos. Vamos ver três notações:

- Notação O
- Notação Ω
- Notação Θ

A notação O descreve um **limite assintótico superior** para uma função.

Para uma função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções $\{f(n)$: existem constantes positivas c e n_0 tal que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$.



$$f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, 0 \leq L < \infty.$$

Escrevemos $f(n) = O(g(n))$ para indicar que $f(n) \in O(g(n))$

Informalmente, dizemos que $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.

Exemplos

$n = O(n^3)$? Sim.

$10000n + 10000 = O(n)$? Sim.

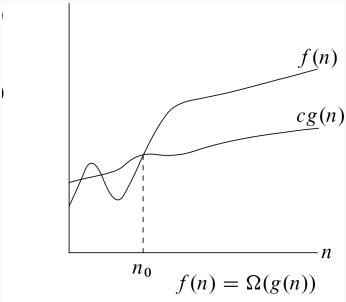
$n^3 + n^2 + n = O(n^3)$? Sim.

$n^3 = O(n^2)$? Não.

$n^3 = O(n^4)$? Sim.

A notação Ω descreve um **limite assintótico inferior** para uma função.

Para uma função $g(n)$, denotamos por $\Omega(g(n))$ o conjunto de funções $\{f(n)$: existem constantes positivas c e n_0 tal que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0\}$



$$f(n) \in \Omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, 0 < L \leq \infty.$$

Informalmente, dizemos que $f(n)$ cresce no mínimo tão rapidamente quanto $g(n)$.

A notação Ω é o oposto da notação O, isto é $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

$n^3 \in \Omega(n^2)$? Sim.

$\sqrt{n} = \Omega(\lg n)$? Sim.

$n^2 + 10n = \Omega(n^2)$? Sim.

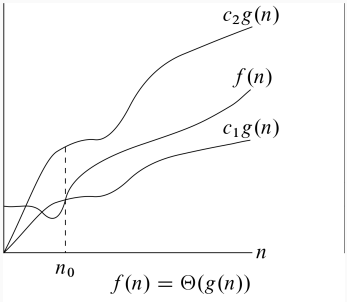
$n = \Omega(n^2)$? Não.

$n^2 = \Omega(n)$? Sim.

Notação Θ

A notação Θ descreve um **limite assintótico restrito** (justo) para uma função.

Para uma função $g(n)$, denotamos por $\Theta(g(n))$ o conjunto de funções $\{f(n)$: existem constantes positivas c_1, c_2 e n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$



$$f(n) \in \Theta(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, 0 < L < \infty.$$

Notação Θ

Para duas funções quaisquer $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Exercícios

$100n^2 = \Theta(n^2)$? Sim.

$\frac{1}{2}n^2 - 3n = \Theta(n^2)$? Sim.

$3n^2 + 20 = \Theta(n)$? Não.

$6n = \Theta(n^2)$? Não.

$720 = \Theta(1)$? Sim.

Sejam $f(n)$ e $g(n)$ funções, então:

$$f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, \quad 0 \leq L < \infty.$$

$$f(n) \in \Omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, \quad 0 < L \leq \infty.$$

$$f(n) \in \Theta(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = L, \quad 0 < L < \infty.$$

Analogia com números reais

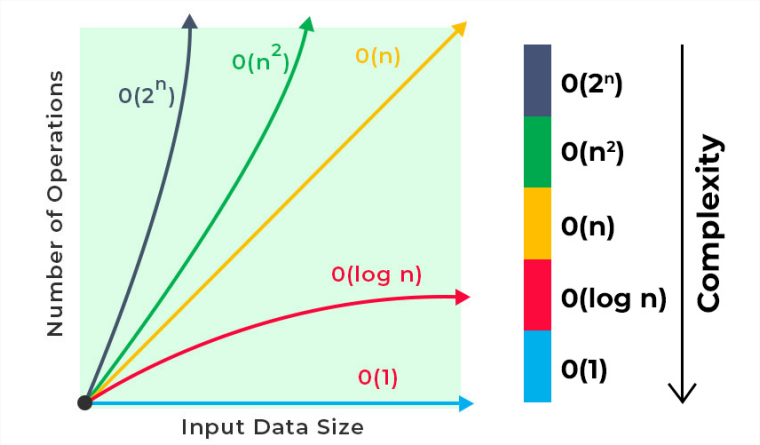
$$f(n) = O(g(n)) \text{ semelhante a } a \leq b$$

$$f(n) = \Omega(g(n)) \text{ semelhante a } a \geq b$$

$$f(n) = \Theta(g(n)) \text{ semelhante a } a = b$$

Tempos de execução comuns

Classe	Descrição
$O(1)$	Constante
$O(\lg n)$	Logarítmico
$O(n)$	Linear
$O(n \lg n)$	Log Linear
$O(n^2)$	Quadrático
$O(n^3)$	Cúbico
$O(2^n)$	Exponencial
$O(n!)$	Fatorial



Fonte: https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/

Referências

Capítulo 3 - Pesquisa, ordenação e análise de complexidade - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca da UEM)

Seção 3.1 - Notação assintótica - Algoritmos: Teoria e Prática, 3a. edição, Cormen, T. at all.