Análise de algoritmos

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

Quais foram as nossas preocupações até agora no projeto de programas?

- Identificar o problema
- Resolver o problema com código bem escrito e testado

O que exatamente significa dizer que um programa resolve um problema?

- Que o programa produz a resposta esperada para todas as entradas.

Isso é suficiente? Não! O programa deve produzir as respostas esperadas consumindo uma quantidade aceitável de recursos (tempo, memória, etc).

Como determinar a quantidade de recursos que um algoritmo (programa, função) consome? Fazendo a análise do algoritmo.

A **análise de algoritmos** é processo de encontrar a complexidade computacional dos algoritmos, isto é, a quantidade de recursos (tempo, memória, etc) necessários para executá-los.

A análise pode ser feita de forma teórica ou experimental, e o resultado da análise é geralmente expresso por uma função que relaciona o tamanho da entrada do algoritmo com o número de passos (complexidade de tempo) ou com o número de células de memória (complexidade de espaço) necessários para executar o algoritmo.

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.

Exemplo - soma

```
1  def soma(lst: list[int]) -> int:
2      s = 0
3      i = 0
4      while i < len(lst):
5          s = s + lst[i]
6          i = i + 1
7      return s
```

Qual é a complexidade de tempo da função `soma`? Ou, quanto tempo a função `soma` consome? Ou, quantas instruções a função executa? Depende da quantidade n de elementos de `lst`.

Vamos "contar" as instruções

- linha 2: 1 instrução
- linha 3: 1 instrução
- linha 4: $(n+1)\times$
  - 1 instrução para comparação
  - umas 3 instrução para executar `len`
- linha 5: $n\times$ umas 3 instruções
- linha 6: $n\times$ 2 instruções
- linha 7: 1 instrução

Total: $9n + 7$

Portanto, a complexidade de tempo de `soma` é $T(n) = 9n + 7$.

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**.

Exemplo - soma

```python
def soma(lst: list[int]) -> int:
    s = 0
    i = 0
    while i < len(lst):
        s = s + lst[i]
        i = i + 1
    return s
```

```gleam
fn soma(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] ->
      primeiro + soma(resto)
  }
}
```

Qual é operação mais executada na função `soma`? A comparação `<`.
Quantas vezes ela é executada? $n + 1$.
Portanto, a complexidade de tempo de `soma` é $T(n) = n + 1$.

Qual é operação mais executada na função `soma`? A comparação de `lst` com `[]`. Ou a operação `+`. Ou a própria chamada da função.
Quantas vezes a função é chamada? $n + 1$.
Portanto, a complexidade de tempo de `soma` é $T(n) = n + 1$.

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.

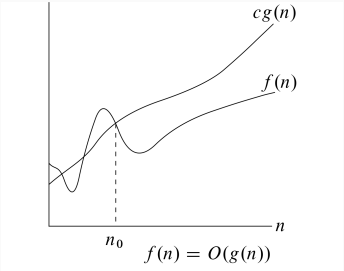
Para expressar e comparar a complexidade algoritmos, utilizamos a **notação assintótica**.

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 $g(n)$.

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

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

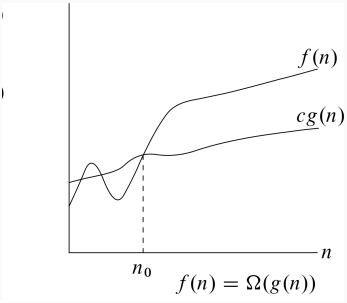
$n^3 + n^2 + n = O(n^3)$? Sim.

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

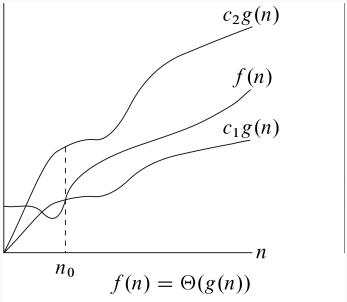
$n = \Omega(n^2)$? Não.

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

Resumo

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

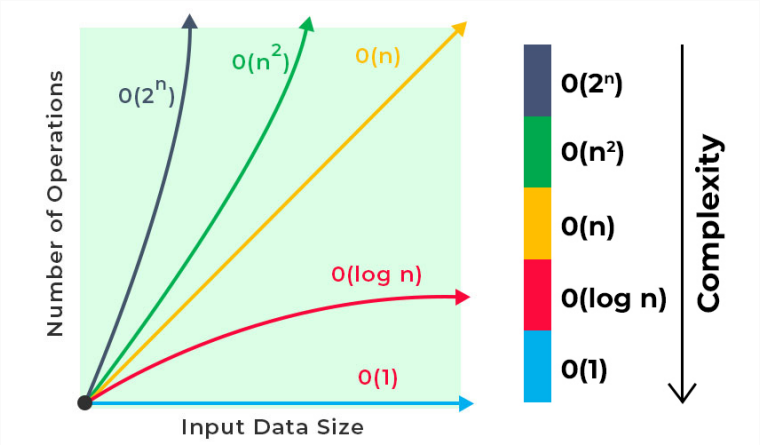
$$f(n) = O(g(n)) \text{ semelhante a } a \leq b$$

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

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

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/

Vamos fazer algumas análise e expressar a complexidade usando a notação assintótica.

Exemplo - contem

```
/// Devolve True se *v* está em *lst*,
/// False caso contrário.
fn contem(lst: List(Int), v: Int) -> Bool {
  case lst {
    [] -> False
    [primeiro, ..resto] ->
      case v == primeiro {
        True -> True
        False -> contem(resto, v)
      }
  }
}
```

Quantas vezes a função é executada?

Depende do valores de entrada!

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 - contem

```
/// Devolve True se *v* está em *lst*,
/// False caso contrário.
fn contem(lst: List(Int), v: Int) -> Bool {
  case lst {
    [] -> False
    [primeiro, ..resto] ->
      case v == primeiro {
        True -> True
        False -> contem(resto, v)
      }
  }
}
```

Quantas vezes a função é executada?

Melhor caso: `v` é primeiro elemento de `lst`. 1 vez.

Pior caso: `v` não está em `lst`. $n + 1$ vezes.

Caso médio: considerando que `v` está em `lst` e tem a mesma probabilidade de ser qualquer elemento de `lst`. $\frac{n+1}{2}$

Portanto, o tempo de execução de `contem` no pior caso é `T(n) = O(n)`.

Exemplo - maximo

```gleam
/// Inverte a ordem dos elementos de *lst*.
fn inverte(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] ->
      adiciona_fim(inverte(resto), primeiro)
  }
}

/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(lst, n) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] ->
      [primeiro,
       ..adiciona_fim(resto, n)]
  }
}
```

Como proceder com a análise nesse caso?

A função `inverte` é chamada $n + 1$ vezes. Na primeira vez, `lst` tem n elementos, na segunda vez $n - 1$, e assim por diante.

Quantas vezes a função `adiciona_fim` é chamada a partir de `inverte`?

Na primeira chamada feita a partir de `inverte`, quantas vezes `adiciona_fim` é executada? n vezes. E na segunda? $n - 1$ vezes. E na última vez? 1 vez.

Exemplo - maximo

```
/// Inverte a ordem dos elementos de *lst*.
fn inverte(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] ->
      adiciona_fim(inverte(resto), primeiro)
  }
}

/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(lst, n) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] ->
      [primeiro,
       ..adiciona_fim(resto, n)]
  }
}
```

Então, temos que `adiciona_fim` é chamada

$$n+(n-1)+(n-2)+\cdots 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

vezes.

Portanto, a complexidade de tempo de `inverte` é $T(n) = O(n^2)$.

Apesar de ser possível determinar a complexidade de tempo contanto “manualmente” o número de chamadas recursivas, como fizemos para `soma`, `contem` e `inverte`, este processo pode ser mais difícil para outras funções.

Por isso, em geral, usamos uma abordagem mais precisa, que é baseada em equações de recorrências e métodos de resolução de equações de recorrência.

Uma **equação de recorrência** descreve o tempo de execução de um algoritmo em termos do tempo de execução de outras chamadas do algoritmo.

Essa ideia parece familiar?

Uma **lista** é:

- Vazia;
- Ou não vazia, contendo o primeiro e o resto, que é uma **lista**.

Modelo de função

```
fn fn_para_lista(lst) {
  case lst {
    [] -> ...
    [primeiro, ..resto] ->
      primeiro
      ...
      fn_para_lista(resto)
  }
}
```

Tempo de execução

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n-1) + f(n) & \text{caso contrário} \end{cases}$$

Onde

- c é o custo do caso base;
- $T(n-1)$ é o custo da chamada recursiva para o resto;
- $f(n)$ é o custo de combinar a solução para o resto com o primeiro elemento.

Exemplo - soma

```gleam
fn soma(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] ->
      primeiro + soma(resto)
  }
}
```

Qual é o custo do caso base? 1.

Qual é o custo da combinação? 1.

Portanto, a equação de recorrência que descreve o tempo de execução de `soma` é:

$$T(n) = \begin{cases} 1 & \text{se } n = 0 \\ T(n-1) + 1 & \text{caso contrário} \end{cases}$$

Ou de forma simplificada, omitindo o caso base

$$T(n) = T(n-1) + 1$$

Expansão

Como podemos obter o tempo de execução a partir de uma equação de recorrência?

Precisamos resolver a equação de recorrência, isto é, encontrar uma forma fechada, que não seja recursiva.

E como resolver uma equação de recorrência?

Uma maneira é expandir iterativamente as chamadas recursivas substituindo-as pela própria definição até atingir um ponto em que a solução fique clara.

Expansão

$$T(n) = T(n-1) + 1$$

$$T(n) = T(n-2) + 1 + 1$$

$$T(n) = T(n-3) + 1 + 1 + 1$$

$$\cdots$$

$$T(n) = T(n-n) + \underbrace{1 + \cdots + 1}_{n}$$

$$T(n) = T(0) + \underbrace{1 + \cdots + 1}_{n}$$

$$T(n) = 1 + \underbrace{1 + \cdots + 1}_{n} = n + 1 = O(n)$$

Exemplo - inverte

```gleam
/// Inverte a ordem dos elementos de *lst*.
fn inverte(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] ->
      adiciona_fim(inverte(resto), primeiro)
  }
}

/// Adiciona *n* ao final de *lst*.
fn adiciona_fim(lst, n) {
  case lst {
    [] -> [n]
    [primeiro, ..resto] ->
      [primeiro,
       ..adiciona_fim(resto, n)]
  }
}
```

Qual é o custo do caso base? 1.

Qual é o custo da combinação? n (custo da chamada de `adiciona_fim`).

Portanto, a equação de recorrência que descreve o tempo de execução de `inverte` é:

$$T(n) = T(n-1) + n$$

Exemplo - inverte

$$T(n) = T(n-1) + n$$

$$T(n) = T(n-2) + (n-1) + n$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

$$T(n) = T(n-n) + (n-(n-1)) + \cdots + (n-2) + (n-1) + n$$

$$T(n) = T(0) + \sum_{i=1}^{n} n = O(n^2)$$

Exemplo - maior repetição

```
fn maior_repeticao(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] ->
      case frequencia(lst, primeiro) >
             maior_repeticao(resto) {
        True -> frequencia(lst, primeiro)
        False -> maior_repeticao(resto)
      }
  }
}
fn frequencia(lst: List(Int), n: Int) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] if primeiro == n ->
      1 + frequencia(resto, n)
    [primeiro, ..resto] ->
      frequencia(resto, n)
  }
}
```

Considerando o caso em que todos os elementos de `lst` são diferentes.

Qual é o custo do caso base? 1.

Qual é o custo da combinação? n (custo de chamar `frequencia`).

Como a função `maior_repeticao` é chamada duas vezes para resto a equação de recorrência que descreve o tempo de execução de `maior_repeticao` é:

$$T(n) = 2T(n-1) + n$$

$$T(n) = 2T(n-1) + n$$

$$T(n) = 2(2T(n-2) + (n-1)) + n = 4T(n-2) + 2(n-1) + n$$

$$T(n) = 4(2T(n-3) + (n-2)) + 2(n-1) + 2n = 8T(n-3) + 4(n-2) + 2(n-1) + n$$

$$\dots$$

$$T(n) = 2^n T(n-n) + \sum_{i=0}^{n-1} 2^i (n-i) \leq n2^n = O(n2^n)$$

Exemplo - maior repetição

```
fn maior_repeticao(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] ->
      int.max(
        frequencia(lst, primeiro),
        maior_repeticao(resto),
      )
  }
}
fn frequencia(lst: List(Int), n: Int) -> Int {
  case lst {
    [] -> 0
    [primeiro, ..resto] if primeiro == n ->
      1 + frequencia(resto, n)
    [primeiro, ..resto] ->
      frequencia(resto, n)
  }
}
```

Qual é o custo do caso base? 1.

Qual é o custo da combinação? n (custo de chamar `frequencia` uma vez).

Portanto, a equação de recorrência que descreve o tempo de execução de `maior_repeticao` é:

$$T(n) = T(n-1) + n = O(n^2)$$

Referências

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