

Análise da Complexidade de Algoritmos

Estrutura de Dados

Prof. Atílio G. Luiz

Semestre 2025.2

Introdução

Um **algoritmo** é uma sequência finita de instruções expressa em forma lógica não ambígua para a resolução de um problema e cuja construção independe da linguagem de programação ou hardware utilizados. **Implementar** um algoritmo significa escrevê-lo numa linguagem de programação para posterior execução.

Em ciência da computação, a **análise de algoritmos** é a área de pesquisa cujo foco são os algoritmos. Em programação, podemos resolver um problema de várias maneiras diferentes, isto é, podemos utilizar algoritmos diferentes para um mesmo problema. Para comparar a eficiência dos algoritmos, foi criada uma medida chamada **complexidade computacional**.

Basicamente, a complexidade computacional indica o custo ao se aplicar um algoritmo, sendo $\text{custo} = \text{memória} + \text{tempo}$, em que a variável **memória** indica quanto de espaço o algoritmo vai consumir e **tempo** indica a duração da sua execução.

Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens:

- **análise empírica:** execução dos programas no computador e posterior comparação do uso dos recursos.
- **análise matemática:** estudo das propriedades do algoritmo.

Análise empírica

Basicamente, esse tipo de análise avalia o custo (ou complexidade) de um algoritmo a

partir da avaliação da execução do mesmo quando implementado. Na **análise empírica** um algoritmo é analisado pela execução de seu programa correspondente.

A análise empírica possui uma série de vantagens. Por meio dela, podemos:

- avaliar o desempenho e uma determinada configuração de computador/linguagem;
- considerar custos não aparentes. Por exemplo, o custo da alocação de memória;
- comparar computadores;
- comparar linguagens.

Apesar de suas vantagens, existem certas dificuldades na realização da análise empírica. Algumas dessas dificuldades são:

- necessidade de implementar o algoritmo. Isso depende da habilidade do programador;
- resultado pode ser mascarado pelo hardware (computador utilizado) ou software (eventos ocorridos no momento da execução);
- qual a natureza dos dados: dados reais, aleatórios ou perversos?

O uso de dados aleatórios permite avaliar o desempenho médio do algoritmo. Dados perversos permitem avaliar o desempenho no pior caso.

Análise matemática

Muitas vezes, é preferível que a medição do tempo gasto por um algoritmo seja feita de

maneira independente do hardware ou da linguagem usada na sua implementação.

A **análise matemática** permite o estudo formal de um algoritmo analisando apenas o seu pseudocódigo. Ela faz uso de um computador idealizado e de simplificações que buscam considerar somente os custos dominantes do algoritmo.

Nesse tipo de análise estamos avaliando a ideia por trás do algoritmo. Para tanto, detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo será executado, ou o conjunto de instruções da CPU, são ignorados. Esse tipo de análise permite entender como um algoritmo se comporta à medida que o conjunto de dados de entrada cresce. Assim, podemos expressar a relação entre o conjunto de dados de entrada e a quantidade de tempo necessária para processar esses dados.

Para evitar análises dependentes de tempo considera-se que um algoritmo é subdividido, ao invés de em milissegundos, em uma quantidade finita de instruções simples que chamaremos de **passos**, onde um passo pode ser interpretado como uma instrução indivisível e de tempo constante, ou seja, independente de condições de entrada ou processamento.

Um **passo** é uma instrução que pode ser executada diretamente pela CPU, ou algo muito perto disso.

Exemplos de instruções básicas que são vistas como passos são:

- atribuição de um valor a uma variável;
- acesso ao valor de determinado elemento de um vetor;
- comparação de dois valores;
- incremento de um valor;
- operações aritméticas básicas, como adição e multiplicação.

Por serem instruções básicas, consideramos que todas elas possuem o mesmo custo. Além disso, supomos que comandos de seleção (como o comando **if**) possuem custo zero, ou seja, não contam como instruções.

Contando passos de um algoritmo

Para entender como calcular o custo de um algoritmo, tome como exemplo o pequeno trecho de código mostrado no Algoritmo 1. Esse algoritmo procura o maior valor presente em um array A contendo n elementos e o armazena na variável M .

Algoritmo 1: Busca do maior

Entrada: Um vetor $A[0..n - 1]$ e um elemento x

Saída: Maior valor contido no vetor.

```

1  $M = A[0];$ 
2 for  $i \leftarrow 1$  até  $n - 1$  do
3   if  $A[i] > M$  then
4      $M = A[i];$ 
5 return  $M;$ 

```

O custo da linha 1 é de 1 passo. Na linha 1, o valor da primeira posição do array é copiado para a variável M . Vamos considerar que essa tarefa requer apenas uma instrução básica para acessar o valor $A[0]$ e atribuí-lo à variável M .

O custo da inicialização do laço **for** (linha 3) é de 2 instruções. No Algoritmo 1, o comando de laço **for** é utilizado para percorrer o array A . Porém, antes mesmo de percorrer o array, ele precisa ser inicializado ao custo de uma instrução ($i = 1$). Além disso, mesmo que o array tenha tamanho zero (isto é, não possua nenhum elemento), ao menos uma comparação será executada ($i < n$), o que, ao todo, custa mais de uma instrução. Portanto, temos um total de duas instruções na inicialização do laço **for**. Perceba que essas duas instruções serão executadas antes da primeira iteração do laço **for**.

Ao fim de cada iteração do laço **for**, precisamos executar mais duas instruções: uma de incremento ($i++$) e uma comparação para verificar se vamos continuar no laço **for** ($i \leq n - 1$). No Algoritmo 1, o comando de laço **for** será executado n vezes (para i variando de 1 até n pois estou contando a última iteração da falha da condição). Assim, esses duas instruções também serão executadas n vezes,

ou seja, o seu custo total para executar o comando de laço **for** da linha 3 é de $2n$ instruções.

A linha 3 do Algoritmo 1 compara dois valores, vamos considerar que ela executa apenas 1 passo. Essa linha é executada sempre que entramos no laço **for**. Como entramos no laço **for** o total de $n - 1$ vezes, então a linha 3 executa o total de $n - 1$ passos.

A linha 4 só será executada se o teste condicional do comando **if** for avaliado como verdadeiro. Na pior das hipóteses, pode ser que o teste sempre seja verdadeiro e sempre executemos a linha 4 toda vez que entramos no bloco do comando **for**. Logo, no pior caso, a linha 4 executará também a mesma quantidade de vezes que a linha 3 executa, ou seja, $n - 1$ vezes. A linha 4 atribui um valor a uma variável, vamos considerar que isso custa apenas 1 passo. Logo, a linha 4 custa no total $n - 1$ passos. A linha 5, por sua vez, executa apenas 1 passo.

Portanto, a fim de descobrir a quantidade total de passos que o Algoritmo 1 executa, basta somar a quantidade total de passos de cada linha, que foi computado acima. Haja vista que o tempo total da busca é sensível ao número de elementos do array A então é conveniente expressar a complexidade deste algoritmo em função de n . A função que dá a complexidade $T(n)$ do Algoritmo 1 no pior caso é:

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

ou seja, $T(n) = 4n$. Mas, o que significa essa função de fato? Essa função nos dá a relação entre o tamanho do array de entrada e o custo do algoritmo (o seu número de passos). Tendo em mãos esta função, é possível plotar um gráfico da função, como o mostrado na Figura 1. No eixo horizontal do gráfico temos os valores do tamanho da entrada até $n = 25$ e, no eixo vertical, temos os valores do número de passos $T(n)$. Essa função nos revela que a complexidade do algoritmo aumenta de forma linear em relação ao tamanho da entrada.

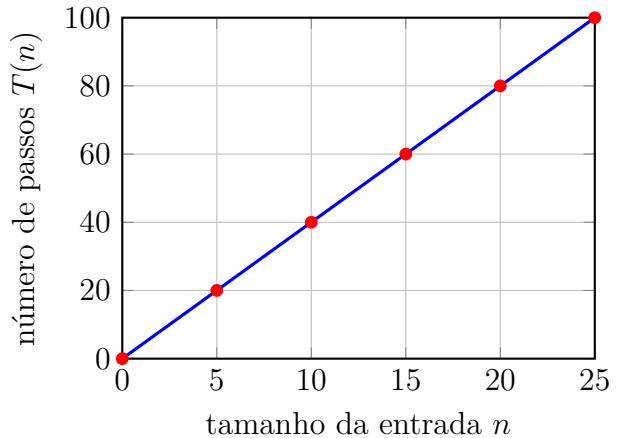


Figura 1: Gráfico da função $T(n) = 4n$.

Complexidades de melhor caso e de pior caso

Na maior parte dos algoritmos a função de complexidade $T(n)$ não é geral, ou seja, pode mudar conforme características da entrada. Nestes casos a análise de complexidade consiste em avaliar o algoritmo em situações extremas, ou seja, determinar que funções de complexidade descrevem os casos de melhor e de pior entrada.

A **complexidade de pior caso** é aquela que revela o máximo de passos que o algoritmo pode efetuar para uma dada entrada denominada **entrada do pior caso**. Analogamente, a complexidade de melhor caso equivale à quantidade mínima de passos que o algoritmo efetua para uma dada entrada denominada **entrada de melhor caso**.

Matematicamente, seja $E = \{e_1, e_2, \dots, e_m\}$ o conjunto de todas as entradas de um algoritmo A e t_i a complexidade de A quando recebe a entrada e_i , com $i \in \{1, 2, \dots, n\}$. Então, se $t_m = \min_{1 \leq i \leq n} t_i$ então t_m é a complexidade de melhor caso e e_m a entrada de melhor caso. Analogamente, se $t_p = \max_{1 \leq i \leq n} t_i$ então t_p é a complexidade de pior caso e e_p a entrada de pior caso.

Para ilustrar as complexidades de melhor e pior caso, tomemos o Algoritmo 2. Para todos os valores que i e j assumem durante o laço **for** tem-se que $L[i] \leq L[j]$. Ademais, quando i e j mudam então os novos valores de $L[i]$ e $L[j]$ são respectivamente menor e maior que os

anteriores. Consequentemente, cada vez que i muda então j não deve mudar e vice-versa. O **else** provoca este comportamento. Nesta situação podemos falar em pior e melhor caso. No melhor caso, todos os testes da linha 4 obtêm êxito, impedindo a execução dos testes da linha 6 e fazendo assim um total de $n - 1$ testes. No pior caso, todos os testes da linha 4 devem falhar exigindo assim que todos os testes da linha 6 ocorram, contabilizando $2n - 2$ testes. Assim, a complexidade de melhor caso é $T(n) = n - 1$ e de pior caso é $T(n) = 2n - 2$. A Figura 2 mostra o gráfico dessas duas funções.

Algoritmo 2: Mínimo e máximo

Entrada: Um vetor $L[0..n - 1]$ e o seu tamanho n

Saída: Mínimo e máximo do vetor

```

1  $i \leftarrow 0;$ 
2  $j \leftarrow 0;$ 
3 for  $k \leftarrow 1$  até  $n - 1$  do
4   if  $L[k] < L[i]$  then
5      $i \leftarrow k;$ 
6   else if  $L[k] > L[j]$  then
7      $j \leftarrow k;$ 
8 return  $(L[i], L[j]);$ 
```

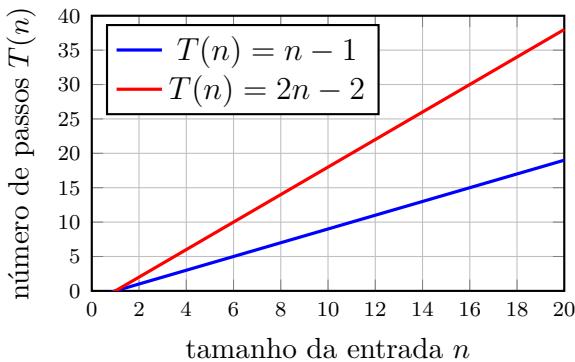


Figura 2: Comparação das funções $T(n) = n - 1$ e $T(n) = 2n - 2$.

Comportamento assintótico

Vimos que o custo de pior caso para o Algoritmo 2 é dado pela função $T(n) = 2n - 2$.

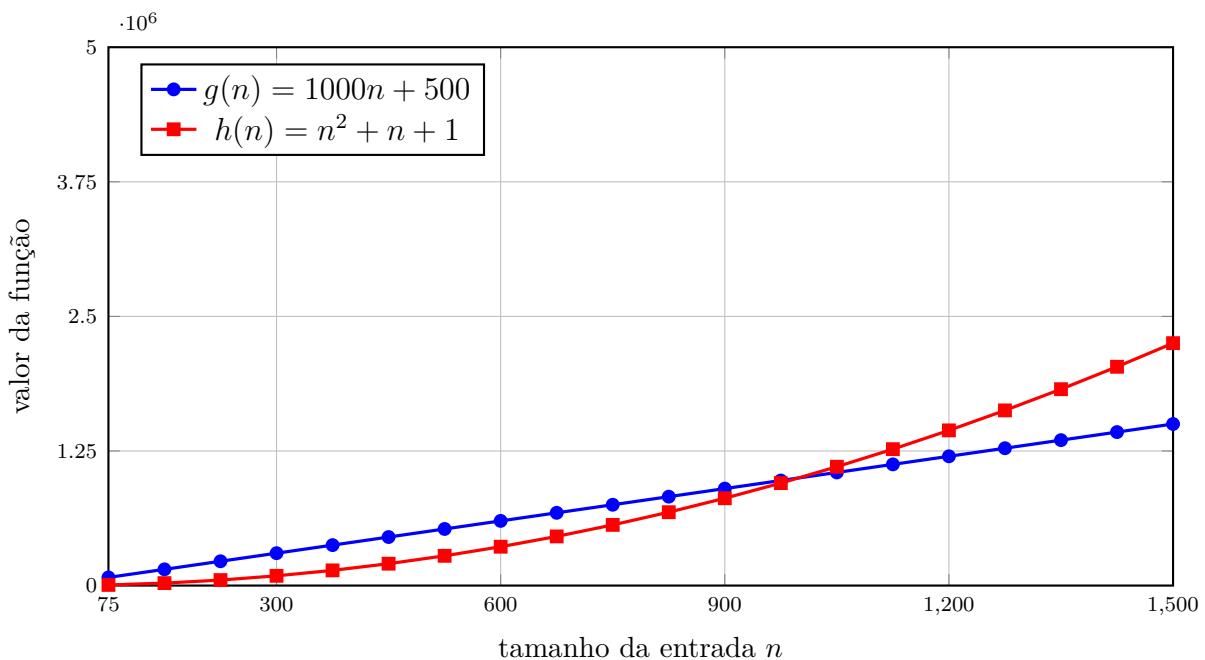
Essa é a função de complexidade de tempo, que nos dá uma ideia do custo de execução do algoritmo para um problema de tamanho n . É importante salientar que é possível criar o mesmo tipo de função para a análise do espaço gasto.

Será que todos os termos da função f são necessários para termos uma noção do custo do algoritmo? De fato, nem todos os termos são necessários. Ou seja, podemos descartar certos termos na função e manter apenas os que nos dizem o que acontece com a função quando o tamanho dos dados de entrada (n) cresce muito.

Diante desse fato, podemos descartar todos os termos que crescem lentamente e manter apenas os que crescem mais rápido à medida que o valor de n se torna maior. Nossa função $T(n) = 2n - 2$, possui dois termos $2n$ e -2 . O termo -2 pode ser descartado já que se mantém constante para cada entrada n . Como o termo -2 não se altera à medida que n aumenta, nossa função pode ser reduzida para $T(n) = 4n$. Constantes que multiplicam o termo n da função também podem ser descartadas. Ao descartarmos essas constantes, nossa função se torna muito simples, $T(n) = n$.

Ao descartarmos todos os termos constantes e manter apenas o de maior crescimento, obtemos uma função que nos dá o **comportamento assintótico** do algoritmo. Chamamos de comportamento assintótico o comportamento de uma função $T(n)$ quando n tende ao infinito.

Isso acontece porque o termo que possui o maior expoente domina o comportamento da função $T(n)$ quando n tende ao infinito. Para entender melhor isso, considere duas funções de custo $g(n) = 1000n + 500$ e $h(n) = n^2 + n + 1$. A Figura 3 mostra as curvas obtidas para essas duas funções à medida que n aumenta. Perceba que apesar da função $g(n)$ possuir constantes maiores multiplicando seus termos, existe um valor de n a partir do qual o resultado de $h(n)$ é sempre maior do que $g(n)$, tornando os demais termos e constantes pouco importantes. Ou seja, podemos suprimir os termos menos importantes da função

Figura 3: Gráfico das funções $g(n)$ e $h(n)$.

de modo a considerar apenas o termo de maior grau. Assim, podemos descrever a complexidade usando somente o seu custo dominante: n para a função $g(n)$ e n^2 para a função $h(n)$.

Notação assintótica

Denomina-se **notação assintótica** a forma matemática de representação simplificada de uma função $f(n)$ levando em conta as componentes de f que crescem mais rapidamente quando n cresce. Apresentaremos sucintamente duas destas importantes notações. A notação O (O-grande ou Big-O) e notação Ω (Ômega).

Notação O

Definição — Notação O

Diz-se que $g(n)$ é **limite superior** de $f(n)$, representado por $f(n) = O(g(n))$ (diz-se f é O de g), quando existem constantes reais positivas n_0 e c tais que, para todo $n \geq n_0$, tem-se que

$$0 \leq f(n) \leq c \cdot g(n).$$

Em outras palavras, um limite superior de-

nota um teto para $f(n)$, ou seja, toda imagem de $f(n)$ ficará abaixo de $c \cdot g(n)$ para valores de n a partir de n_0 . O cruzamento entre $f(n)$ e $c \cdot g(n)$ na Figura 4 é único, acusando que esta segunda função, a partir de n_0 , estará sempre acima da primeira.

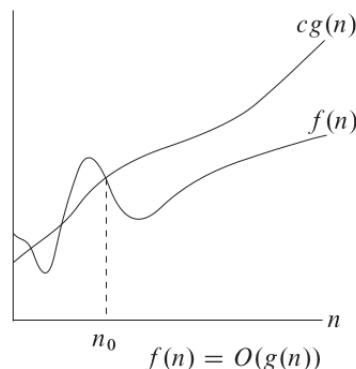


Figura 4: Ilustração das funções envolvidas na definição da Notação O.

Exemplo 1

Tarefa: Propor um limite superior para a função $f(n) = 3n^2 + 18$ juntamente com constantes reais n_0 e c positivas, de modo que, para todo $n \geq n_0$, tenhamos $0 \leq f(n) \leq c \cdot g(n)$.

Solução: Propomos $g(n) = n^2$ e como constantes reais positivas, propomos $n_0 = 3$ e

$c = 5$. Primeiramente, note que a função $3n^2 + 18$ é sempre não negativa para todo valor de $n \geq 0$. Logo, $0 \leq f(n)$ para todo n não negativo. Verifiquemos agora a segunda desigualdade,

$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ 3n^2 + 18 &\leq 5n^2 \\ 18 &\leq 2n^2 \\ 9 &\leq n^2 \end{aligned}$$

O conjunto solução dessa última desigualdade são todos os valores de n tais que $n \leq -3$ ou $n \geq 3$. Como $n_0 = 3$ e a desigualdade vale para todo valor $n \geq 3$, então $3n^2 + 18 = O(n^2)$.

Comentário: Note que, na solução da tarefa, não foi dada nenhuma dica de como obtivemos os palpites para $g(n)$ e para as constantes c e n_0 . Tanto a função quanto as constantes foram tiradas da cartola. No entanto, obter alguns desses valores não é uma tarefa impossível, você só precisa entender o que estão te pedindo para provar. Neste caso, a tarefa está pedindo para você propor um limite superior para a função $f(n) = 3n^2 + 18$. Como essa função é uma função quadrática, temos a intuição de que qualquer polinômio de grau maior do que 2 pode ser um limite superior para ela. Porém, será que a função quadrática $g(n) = n^2$ também pode vir a ser um limite superior? De acordo com a definição de notação O , para verificar se esse palpite é correto, precisamos encontrar constantes reais positivas c e n_0 tais que, para todo $n \geq n_0$, $f(n) \leq cg(n)$. Como 3 é a constante que multiplica o termo de maior grau de $f(n)$, parece que escolher para c qualquer valor maior que 3 pode ser bom. Na nossa solução, escolhemos $c = 5$. Escolhido $c = 5$ e $g(n) = n^2$, plugamos esses valores na desigualdade dada pela definição de notação O e fomos em busca de encontrar os valores de n que satisfazem a desigualdade. No fim, verificamos que ela era satisfeita para todos os valores de n maiores ou iguais a 3. Logo, todos os requisitos impostos pela definição de notação O foram satisfeitos e pudemos, por fim, concluir que $3n^2 + 18 = O(n^2)$.

A determinação do limite superior de uma função $f(n)$ pode ser feita pela seguinte análise,

- Eliminar constantes aditivas e multiplicativas ou que dividirão $f(n)$ em componentes para análise.
- Identificar a componente de $f(n)$ que cresce mais rapidamente que as demais (um gráfico poderá auxiliar a tarefa). Esta componente será o limite superior procurado.

A regra da soma

A notação O possui algumas propriedades, sendo a mais importante a regra da soma. A **regra da soma** é muito importante na análise da complexidade de diferentes algoritmos em sequência. Basicamente, se dois algoritmos são executados em sequência, a complexidade da execução dos dois algoritmos será dada pela complexidade do maior deles, ou seja,

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

Por exemplo, se temos dois algoritmos cujos tempos de execução são $O(n^2)$ e $O(n)$, então a execução desses algoritmos em sequência terá como complexidade $O(\max(n^2, n)) = O(n^2)$. Isso implica que o algoritmo de maior complexidade domina o tempo de execução como um todo.

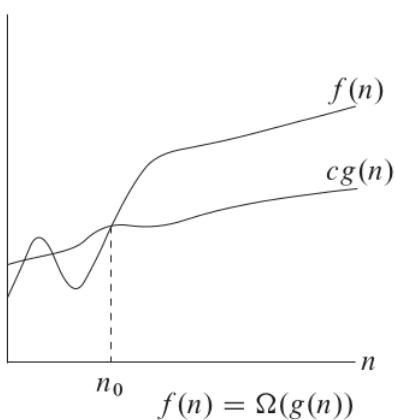
Notação Ω

Definição

Diz-se que $g(n)$ é **limite inferior** de $f(n)$, representado por $f(n) = \Omega(g(n))$ (diz-se f é ômega de g), quando existem constantes reais positivas n_0 e c tais que, para todo $n \geq n_0$, tem-se que

$$0 \leq c \cdot g(n) \leq f(n).$$

De forma similar ao limite superior, um limite inferior denota um piso para $f(n)$, ou



- [2] J. L. Szwarcfiter, L. Markenzon, *Estruturas de dados e seus algoritmos*, 3. ed. Rio de Janeiro, LTC, 2010.

Figura 5: Ilustração do comportamento das funções envolvidas na definição da Notação Ω .

seja, toda imagem de $f(n)$ ficará acima de $cg(n)$ para todos os valores de n a partir de n_0 . O cruzamento entre $f(n)$ e $cg(n)$ na Figura 5 é único, indicando que esta segunda função, a partir de n_0 , estará sempre abaixo da primeira.

Exemplo 2

Atividade: Propor um limite inferior para $f(n) = n^2 - 3n$ juntamente com constantes n_0 e c válidas.

Solução: Propomos $g(n) = n$ e como constantes válidas citamos $n_0 = 4$ e $c = 1$. Primeiramente, note que a imagem da função $cg(n) = cn$ é sempre não negativa para todo $n \geq 0$. Verifiquemos agora a segunda desigualdade da definição de notação Ω ,

$$\begin{aligned} c \cdot g(n) &\leq f(n) \\ 1 \cdot n &\leq n^2 - 3n \\ n^2 - 4n &\geq 0 \end{aligned}$$

O conjunto solução desta última desigualdade são todos os valores de n tais que $n \leq -2$ ou $n \geq 2$. Como $n_0 = 4$ e a desigualdade vale para todo $n \geq 2$, então $n^2 - 3n = \Omega(n)$.

Referências

- [1] A. R. Backes, *Algoritmos e estruturas de dados em linguagem C*, LTC, 2023.