Complexidade de Algoritmos

Alessandro

1 Introdução

Algoritmos são o cerne da computação. Um programa codifica um algoritmo de modo a ser executado em um computador, resolvendo um problema. Na prática é fundamental que um programa produza a solução com dispêndio de tempo e de memória razoáveis. Daí a importância de análise de algoritmos.

Um algoritmo é um procedimento, consistindo de um conjunto de regras não ambíguas, as quais especificam para cada entrada, uma sequência finita de operações, terminando com uma saída correspondente.

Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes, para sua execução. O fato de um algoritmo resolver um problema não implica que seja aceitável na prática. Os recursos de espaço e tempo requeridos têm grande importância em casos práticos.

Às vezes, o algoritmo mais imediato está longe de ser razoável em termos de eficiência. Um exemplo, é o caso da solução de equações lineares. O método de Cramer, calculando o determinante através de sua definição, requer dezenas de milhões de anos para resolver um sistema $20\ x\ 20$. Um sistema como esse pode ser resolvido em tempo razoável pelo método de Gauss. A tabela 1 mostra o desempenho desses dois algoritmos que calculam o determinante de uma matriz $n\ x\ n$, considerando tempos de operações de um computador real.

n	Método de Cramer	Método de Gauss
2	22 μs	50μs
3	102 μs	159μs
4	456 μs	353μs
5	2,35 ms	666μs
10	1,19 min	4,95 ms
20	15. 225 séculos	38, 63 ms
40	5 * 10^{33} séculos	0, 315 s

Tabela 1: Tamanho do Problema x Tempo de Execução

O crescente avanço tecnológico, permitindo a criação de máquinas cada vez mais rápidas, pode, ingenuamente, parecer ofuscar a importância da complexidade de tempo de um algoritmo. Entretanto, acontece exatamente o inverso. As máquinas, tornando-se mais rápidas, passam a poder resolver problemas maiores, e é a complexidade do algoritmo que determina o novo tamanho máximo do problema que pode ser resolvido. Para um algoritmo rápido, qualquer melhoria na velocidade de execução das operações básicas é sentida, e o conjunto de problemas que podem ser resolvidos por ele aumenta sensivelmente. Esse impacto é menor nos algoritmos menos eficientes.

Na área de análise de algoritmos , existem dois tipos de problemas bem definidos:

- **Análise de um algoritmo particular**: Qual é o custo de usar um dado algoritmo para resolver um problema específico? Neste caso, características importantes do algoritmo em questão devem ser investigadas. Por exemplo, no problema das equações lineares, para entradas menores que três ($n \leq 3$), o tempo requerido utilizando o método de Cramer é mais rápido que o método de Gauss, mas para entrada maiores, o método de Gauss é muitas vezes mais rápido (Vide tabela 1).
- **Análise de uma classe de algoritmos**: Qual é o algoritmo de menor custo possível para resolver um problema particular? Neste caso, toda uma família de algoritmos para resolver um problema específico é investigada com o objetivo de identificar um que seja o melhor possível. Isso significa impor limites para a complexidade computacional dos algoritmos pertencentes à classe

Em muitas situações podem existir vários algoritmos para resolver o mesmo problema, sendo pois necessário escolher o melhor. Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado para resolver o problema em questão.

2 Complexidade e desempenho de algoritmos

A complexidade de um algoritmo reflete o esforço computacional requerido para executá-lo. Esse esforço computacional mede a quantidade de trabalho, em termos de tempo de execução ou da quantidade de memória requerida. As principais medidas de complexidade são **tempo** e **espaço**, relacionadas à velocidade e à quantidade de memória, respectivamente, para a execução de um algoritmo.

Há vários aspectos a considerar na determinação da complexidade de um algoritmo. Uma questão é a medida empírica. Podemos pensar em medir experimentalmente a quantidade de trabalho (tempo ou memória) requerida por um algoritmo executado em um computador específico. Entretanto, uma medida empírica é fortemente dependente, tanto do programa, quanto da máquina usada para implementar o algoritmo. Assim, uma pequena mudança no programa pode não representar uma mudança significativa no algoritmo, mas apesar disso, afetar a velocidade de execução. Além disso, se dois programas são comparados, primeiro numa máquina, depois em outra, as comparações podem dar resultados diferentes.

Assim, apesar de a comparação de programas reais, executando em computadores reais, ser uma fonte de informação importante, os resultados são inevitavelmente afetados pela programação e pelas características das máquinas. Um alternativa útil vem de se fazer uma análise matemática do algoritmo. Essa análise, além de ser independente da implementação, permite, muitas vezes, antecipar o cálculo da complexidade para a fase de projeto do algoritmo. Além disso, esse gênero de análise é útil para avaliar as dificuldades intrínsecas da resolução computacional de um problema.

O esforço computacional de um algoritmo não pode ser descrito simplesmente por um número, porque a quantidade de trabalho requerido em geral, depende da entrada. Por exemplo, a quantidade de trabalho para ordenar um lista de números depende do números de elementos da lista; na multiplicação de matrizes, o número de operações (adição e multiplicação) varia com a dimensão da matriz. Esses exemplos mostram que o esforço computacional de um algoritmo depende do tamanho da entrada. Mesmo para entradas de mesmo tamanho, o esforço computacional requerido pelo algoritmo pode depender de uma entrada particular. Para ordenar uma lista quase classificada, pode não ser necessário o mesmo esforço para ordenar uma lista de mesmo tamanho, mas com os elementos fora de ordem.

A quantidade de trabalho para executa um algoritmo sobre uma determinada entrada pode ser referida como o desempenho do algoritmo (para uma dada entrada). Assim, o desempenho de um algoritmo dá o esforço computacional de execução para uma entrada dada. Por outro lado, a complexidade de um algoritmo reflete o esforço para executá-lo sobre um conjunto de entradas. A complexidade pessimista dá o pior valor: o máximo de esforço requerido. Já a complexidade média (ou esperada) dá o valor esperado: a média dos esforços, levando em conta a probabilidade de ocorrência de cada entrada.

Quando conseguimos determinar o menor custo possível para resolver problemas de determinada classe, temos a medida da dificuldade inerente para resolver tais problemas. Além disso, quando o custo de um algoritmo é igual ao menor custo possível, dissemos que o algoritmo é **ótimo** para a medida de custo considerada.

3 Medidas de desempenho e complexidade

Existem várias medidas de desempenho: tempo e espaço requeridos por um algoritmo, relacionados à velocidade e à quantidade de memória respectivamente. Uma das medidas de desempenho mais importante, é o tempo de execução. Nesse caso, trata-se da complexidade de tempo.

A complexidade de espaço usa como medida de desempenho a quantidade de memória necessária para a execução do algoritmo. A memória utilizada por um programa bem como o tempo requerido para executar um programa, depende da implementação particular. Entretanto, algumas conclusões sobre o espaço utilizado podem ser tiradas, examinando o algoritmo. Um programa requer uma área para guardar suas instruções, suas constantes, suas variáveis e os dados. Pode também utilizar uma área de trabalho para manipular os dados e guardar informações para levar adiante a computação. Os dados podem ser representados de várias formas, as quais requerem uma área maior ou menor. Se os dados têm uma representação natural, por exemplo, uma matriz, são considerados para análise de espaço somente o espaço extra, além do escopo utilizado para guardar as instruções do programa e os dados. Porém, se os dados podem ser representados de várias formas, como um grafo, conforme veremos ao final da disciplina, o espaço requerido para guardá-lo deve ser levado em consideração.

4 Critérios de complexidade

O tempo requerido por um algoritmo sobre uma dada entrada pode ser medido pelo número de execuções de algumas operações. Para medir a quantidade de trabalho realizado por um algoritmo, costuma-se escolher uma operação, chamada **operação fundamental**. A operação escolhida como fundamental deve ser tal que a contagem do número de vezes que ela é executada expresse a quantidade de trabalho do algoritmo, dispensando outras medidas.

Considere que para um algoritmo de ordenação, uma operação fundamental natural é a comparação entre os elementos quando à ordem. Para o caso da busca, uma operação fundamental pode ser o número de comparações, quanto à igualdade, de um elemento da estrutura com o valor buscado.

Escolhida uma noção de custo, consideremos a sequência de passos executados pelo algoritmo como uma determinada entrada. Nesse caso, é contado do número de execuções das operações fundamentais na execução do algoritmo. Assim, temos o custo: esforço computacional desse algoritmo sobre essa entrada.

$$d \rightarrow \underbrace{s_0 \cdots \rightarrow s_k \rightarrow \cdots s_n}$$

Definição A complexidade de um algoritmo, para uma certa entrada é uma função $f(n)$, que expressa a quantidade estimada de passos necessários para computar a saída do algoritmo com tamanho da entrada igual a n.

1. Se a medida é de tempo, então $f(n)$ é chamada de função de complexidade de tempo ou temporal do algoritmo.
2. Se a medida é a da memória necessária (espaço) para executar o algoritmo, então $f(n)$ é a função de complexidade espacial.

Para ilustrar alguns desses conceitos, considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $v[0..n-1]$, $n \geq 1$. Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de v, se v contiver n. Logo,

$$f(n) = n - 1, \; para \; n > 0$$

```
int maximo(int [] v, int n){
        int max = v[0];
        for (int i = 1; i < n; i++){
                if (max < v[i])
                        max = v[i];
        }
        return max;
}
```

Listing 1: Algoritmo para obter o maior elemento de um vetor.

Será que o algoritmo da listagem 1, para obter o maior elemento de um vetor, é ótimo? Ou seja, que nenhum outro algoritmo, que utilize como medida de custo o número de comparações consegue encontrar o maior elemento de um vetor v de tamanho n em tempo menor a $f(n) = n - 1$?

Teorema 4.1 *Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, para $n \geq 1$ faz pelo menos $n - 1$ comparações.*

Proof Cada um dos $n - 1$ elementos tem que ser verificado, por meio de comparações, que cada um dos $n - 1$ elementos é menor do que algum outro elemento. Logo, $n - 1$ comparações são necessárias.

Portanto, como o algoritmo apresentado, que obtém o maior valor de um vetor v de n elementos possui complexidade igual ao limite inferior de custo, então o **algoritmo é ótimo**.

Normalmente, a medida de custo de execução depende do tamanho da entrada, mas, este não é o único fato que influencia o custo. O custo de execução pode sofrer influência do tipo da entrada. Considere um outro algoritmo para obter o máximo e o mínimo de um vetor de tamanho igual a n. Um algoritmo simples para resolver esse problema pode ser derivado do algoritmo 1.

```
int[] maximo_minimo(int[]  v, int n){

  int max, min;
  max = min = v[0];

  for (int i = 1; i <  n; i++){
    if (max < v[i])
        max = v[i]
     if (min > v[i])
        min = v[i];
  }
  return {max, min};
}
```

Listing 2: Algoritmo para obter o maior e o menor elemento de um vetor.

O algoritmo 2 podem ser facilmente melhorado. Basta observar que a comparação $v[i] < min$ somente é necessária quando o resultado da comparação $v[i] > max$ é falso. Uma nova versão do algoritmo pode ser vista no algoritmo 3. Para essa implementação, os casos a considerar são:

- melhor caso: $f(n) = n - 1$
- pior caso: $f(n) = 2(n - 1)$
- caso médio: $f(n) = \frac{3n-3}{2}$

O melhor caso ocorre quando os elementos de v estão em ordem crescente. O pior caso ocorre quando os elementos de v estão em ordem decrescente. No caso médio, $v[i]$ é maior do que *max* a metade duas vezes. Logo, $f(n) = n - 1 + \frac{n-1}{2} = \frac{3n-3}{2}$, $para\ n > 0$.

```
int[] maximo_minimo_melhorado(int[]  v, int n){

  int max, min;
  max = min = v[0];

  for (int i = 1; i <  n; i++){
    if (max < v[i])
        max = v[i]
     else if (min > v[i])
        min = v[i];
  }
  return {max, min};
}
```

Listing 3: Algoritmo melhorado para obter o maior e o menor elemento de um vetor.

5 Comportamento Assintótico de Funções

Como mencionado anteriormente, o custo para obter uma solução para um dado problema aumenta com o tamanho da entrada (n) do problema. O número de comparações para encontrar o maior elemento de um conjunto n de inteiros, ou para ordenar os elementos de um conjunto com n elementos, aumenta com n. O parâmetro n fornece uma medida da dificuldade para se resolver o problema, no sentido de que o tempo necessário para resolver o problema cresce quando n cresce.

Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os algoritmos ineficientes. Isto é, a escolha do algoritmo não é um problema crítico para problemas de tamanho pequeno. Logo, a análise de algoritmos é realizada para valores grandes de n. Para tal, considera-se o comportamento assintótico das funções de custo.

O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.

Definição Uma função $f(n)$ domina assintoticamente outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos $|g(n)| \leq c * |f(n)|$.

A figura 1, expressa graficamente o significado da definição anterior.

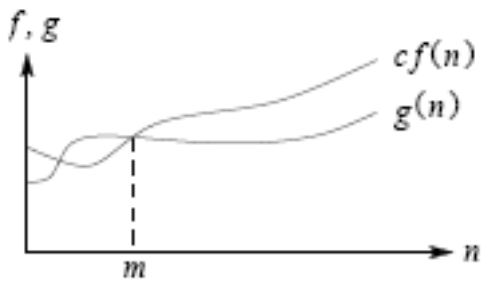

Figura 1: Dominação assintótica de $f(n)$ sobre $g(n)$.

Exemplo Sejam as equações (1) e (2) abaixo:

1. $g(n) = (n+1)^2$
2. $f(n) = n^2$.

As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, desde que:

- $|(n+1)^2 \leq 4|n^2| para n \geq 1$
- $|n^2| \leq |(n+1)^2|,\ para\ \geq 0$

5.1 Notação O (Big Oh)

Definição Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq c * f(n)$, para todo $n \geq m$.

Exemplo Seja $g(n) = (n+1)^2$. Logo, $g(n)$ é $O(n^2)$, quando m = 1 e c = 4. Isso porque $(n+1)^2 \ \leq 4n^2$, para $n \geq 1$. A figura 2 apresenta gráficamente a dominação assintótica que ilustra a notação O.

Exemplo $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$. Isso porque $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.

Exemplo $g(n) = \log_5 n$ é $O(\log n)$. Isso porque $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$.

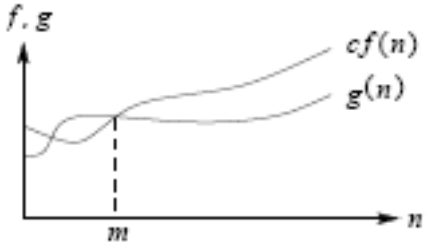

Figura 2: A notação O define um limite superior para a função.

5.2 Operações com a Notação O

A tabela 5.2 apresenta as operações que podem ser realizadas com a notação O.

$f(n) =$	$O(f(n))$
$c * O(f(n)) =$	$O(f(n))$ c = constante
$O(f(n)) + O(f(n)) =$	$O(f(n))$
$O(O(f(n))) =$	$O(f(n))$
$O(f(n)) + O(g(n)) =$	$O(max(f(n), g(n)))$
$O(f(n)) * O(g(n)) =$	$O(f(n) * g(n))$
$f(n) * O(g(n)) =$	$O(f(n) * g(n))$

Tabela 2: Operações com a notação O.

5.3 Classes de Comportamento Assintótico

Os algoritmos podem ser avaliados por meio da comparação de suas funções de complexidade, neglicenciando as constantes de proporcionalidade. Um algoritmo de tempo de execução $O(n)$ é melhor que um algoritmo de tempo de execução $O(n^2)$. Entretanto, as constantes de proporcionalidade em cada caso podem alterar essa consideração. Por exemplo, é possível que um programa leve $100n$ unidades de tempo para ser executado, enquanto outro leve $2n^2$ unidades de tempo. Qual dos dois programas é melhor?

A resposta a essa pergunta depende do tamanho do problema a ser executado. Para problemas de tamanho $n < 50$, o algoritmo com tempo de execução $2n^2$ é melhor do que o algoritmo com tempo de execução $100n$. Para problemas com entrada de dados pequena é preferível usado o algoritmo cujo tempo de execução é $O(n^2)$. Entretanto, quando n cresce, o algoritmo com tempo $O(n^2)$ leva muito mais tempo que o algoritmo $O(n)$.

A maioria dos algoritmos possui um parâmetro que afeta o tempo de execução de forma mais significativa, usualmente o número de itens a ser processado. Esse parâmetro pode ser o número de registros de um arquivo a ser ordenado ou o número de nós de um grafo. As principais classes de problemas possuem as seguintes funções de complexidade:

1. $f(n) = O(1)$. Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**. O uso do algoritmo independe do tamanho de n. Nesse caso, as instruções do algoritmo são executadas um número fixo de vezes.

2. $f(n) = O(\log n)$. Um algoritmo de complexidade $O(\log n)$ é dito **complexidade logarítmica**. Esse tempo de execução ocorre tipicamente em algoritmos que resolvem um problema, transformando-o em problemas menores.

3. $f(n) = O(n)$. Um algoritmo de complexidade $O(n)$ é dito de **complexidade linear**. Em geral, um pequeno trabalho é realizado sobre cada elemento da entrada. Essa é a melhor situação possível para um algoritmo que tem de processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho, o tempo de execução também dobra.

4. $f(n) = O(n \log n)$. Esse tempo de execução ocorre tipicamente em algoritmos que resolvem um problema

quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois juntando as soluções.

5. $f(n) = O(n^2)$. Um algoritmo de complexidade $O(n^2)$ é dito de **complexidade quadrática**. Algoritmos dessa ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro de outro. Quando n é mil, o número de operações é da ordem de 1 milhão. Sempre que n dobra, o tempo de execução é multiplicado por 4. Algoritmos desse tipo são úteis para resolver problemas de tamanho relativamente pequenos.

6. $f(n) = O(n^3)$. Um algoritmo de complexidade $O(n^3)$ é dito de **complexidade cúbica**. Algoritmos dessa ordem de complexidade são úteis apenas para resolver pequenos problemas.

7. $f(n) = O(2^n)$. Um algoritmo de complexidade $O(2^n)$ é dito de **complexidade exponencial**. Algoritmos dessa ordem de complexidade não são úteis do ponto de vista prático. Eles ocorrem na solução de problemas que usam força bruta para resolvê-los Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo de execução fica elevado ao quadrado.

8. $f(n) = O(n!)$ Um algoritmo de complexidade $O(n!)$ é também dito de **complexidade exponencial**, apesar de a **complexidade fatorial** $O(n)$ ter comportamento muito pior do que a complexidade $O(2^n)$. Algoritmos dessa ordem de complexidade geralmente ocorrem na solução de problemas quando se usa força bruta, para resolvê-los. Quando n é 20, 20! = 2432902008176640000, um número com 19 dígitos. Quando n é 40, 40! é um número com 48 dígitos.