CAPÍTULO 15

Complexidade de algoritmos

Neste capítulo, discutiremos a famosa **notação big-O** (O grande) e a teoria de **NP-completo** (NP-completeness); veremos também como podemos nos divertir com alguns algoritmos e aperfeiçoar o nosso conhecimento a fim de melhorar nossas habilidades de programação e de resolução de problemas.

Notação big-O

No Capítulo 13, *Algoritmos de ordenação e de busca*, introduzimos o conceito de notação big-O. O que isso significa, exatamente? Essa notação é usada para descrever o desempenho ou a complexidade de um algoritmo. A notação big-O é usada para classificar algoritmos de acordo com o tempo que eles demorarão para executar, conforme os requisitos de espaço/memória à medida que o tamanho da entrada aumentar.

Ao analisar algoritmos, as classes de funções a seguir são as mais comumente encontradas.

Notação	Nome
$O(1)$	Constante
$O(log(n))$	Logarítmica
$O((log(n))c)$	Polilogarítmica
$O(n)$	Linear
$O(n^2)$	Quadrática
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial

Compreendendo a notação big-O

Como podemos medir a eficiência de um algoritmo? Em geral, usamos recursos como uso (tempo) de CPU, utilização de memória, de disco e de rede. Quando falamos da notação big-O, em geral, consideramos o uso (tempo) de CPU.

Vamos tentar entender como a notação big-O funciona usando alguns exemplos.

O(1)

Considere a função a seguir.

```
function increment(num){
  return ++num;
}
```

Se tentarmos executar a função `increment(1)`, teremos um tempo de execução igual a `x`. Se tentarmos executar essa mesma função novamente com um parâmetro diferente (por exemplo, `num` igual a `2`), o tempo de execução também será `x`. O parâmetro não importa; o desempenho da função de incremento será o mesmo. Por esse motivo, podemos dizer que a função anterior tem complexidade igual a *O(1)* (é constante).

O(n)

Vamos agora usar o algoritmo de busca sequencial que implementamos no Capítulo 13, *Algoritmos de ordenação e de busca*, como exemplo:

```
function sequentialSearch(array, value, equalsFn = defaultEquals) {
  for (let i = 0; i < array.length; i++) {
    if (equalsFn(value, array[i])) { // {1}
      return i;
    }
  }
  return -1;
}
```

Se passarmos um array com 10 elementos ([1, ..., 10]) para essa função e procurarmos o elemento 1, na primeira tentativa, encontraremos o elemento que estávamos procurando. Vamos supor que o custo seja 1 para cada vez que a linha `{1}` é executada.

Vamos tentar outro exemplo. Suponha que estamos procurando o elemento 11. A linha {1} será executada 10 vezes (uma iteração será feita por todos os valores do array e o valor que estamos procurando não será encontrado; desse modo, `-1` será devolvido). Se a linha {1} tiver um custo igual a 1, executá-la 10 vezes terá um custo igual a 10, isto é, 10 vezes mais em comparação com o primeiro exemplo.

Suponha agora que o array tenha 1.000 elementos ([1, ..., 1.000]). Procurar o elemento 1.001 resultará em a linha {1} ser executada 1.000 vezes (e então `-1` será devolvido).

Observe que o custo total da execução da função `sequentialSearch` depende do número de elementos do array (tamanho) e do valor que procuramos. Se o item que estivermos procurando estiver presente no array, quantas vezes a linha {1} será executada? Se o item procurado não existir, a linha {1} será executada o número de vezes correspondente ao tamanho do array, que chamaremos de cenário de pior caso.

Considerando o cenário de pior caso da função `sequentialSearch`, se tivermos um array de tamanho 10, o custo será igual a 10. Se tivermos um array de tamanho 1.000, o custo será igual a 1.000. Podemos concluir que a função `sequentialSearch` tem uma complexidade de $O(n)$ – em que n é o tamanho do array (entrada).

Para ver a explicação anterior na prática, vamos modificar o algoritmo a fim de calcular o custo, isto é, `cost` (cenário de pior caso), assim:

```
function sequentialSearch(array, value, equalsFn = defaultEquals) {
  let cost = 0;
  for (let i = 0; i < array.length; i++) {
    cost++;
    if (equalsFn(value, array[i])) {
      return i;
    }
  }
  console.log(`cost for sequentialSearch with input size ${array.length} is ${cost}`);
  return -1;
}
```

Experimente executar o algoritmo anterior usando tamanhos distintos de entrada para que você veja os diferentes resultados.

O(n2)

Para o exemplo de $O(n^2)$, usaremos o algoritmo de *bubble sort*:

```
function bubbleSort(array, compareFn = defaultCompare) {
  const { length } = array;
  for (let i = 0; i < length; i++) { // {1}
    for (let j = 0; j < length - 1; j++) { // {2}
      if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) {
        swap(array, j, j + 1);
      }
    }
  }
  return array;
}
```

Considere que as linhas {1} e {2} tenham, cada uma, um custo igual a 1. Vamos modificar o algoritmo a fim de calcular o custo (cost) da seguinte maneira:

```
function bubbleSort(array, compareFn = defaultCompare) {
  const { length } = array;
  let cost = 0;
  for (let i = 0; i < length; i++) { // {1}
    cost++;
    for (let j = 0; j < length - 1; j++) { // {2}
      cost++;
      if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) {
        swap(array, j, j + 1);
      }
    }
  }
  console.log(`cost for bubbleSort with input size ${length} is ${cost}`);
  return array;
}
```

Se executarmos bubbleSort para um array de tamanho 10, cost será igual a 100 (10^2). Se executarmos bubbleSort para um array de tamanho 100, cost será 10.000 (100^2). Observe que a execução demorará mais tempo ainda sempre que aumentarmos o tamanho da entrada.

Note que o código cuja complexidade é $O(n)$ tem apenas um laço for, enquanto, para $O(n^2)$, há dois laços for aninhados. Se o algoritmo tiver três laços for iterando pelo array, provavelmente ele terá uma complexidade de $O(n^3)$.

Comparando as complexidades

Podemos criar uma tabela com alguns valores para exemplificar o custo do algoritmo, dado o tamanho de sua entrada, da seguinte maneira:

Tamanho da entrada (n)	O(1)	O(log (n))	O(n)	O(n log(n))	$O(n^2)$	$O(2^n)$
10	1	1	10	10	100	1.024
20	1	1,30	20	26,02	400	1.048.576
50	1	1,69	50	84,94	2.500	Número bem grande
100	1	2	100	200	10.000	Número bem grande
500	1	2,69	500	1.349,48	250.000	Número bem grande
1.000	1	3	1.000	3.000	1.000.000	Número bem grande
10.000	1	4	10.000	40.000	100.000.000	Número bem grande

Podemos desenhar um gráfico baseado nas informações apresentadas na tabela anterior para exibir o custo de diferentes complexidades em notação big-O, assim:

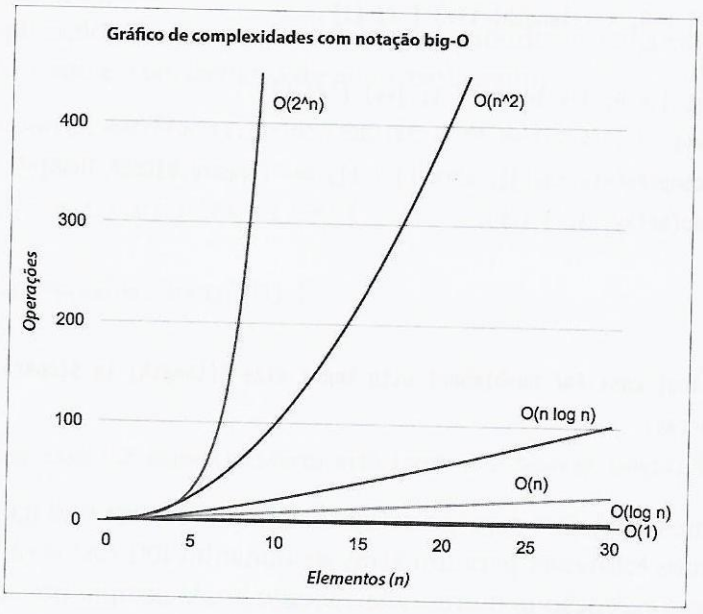

Figura 15.1

O gráfico anterior também foi gerado com JavaScript. Veja o código-fonte dele no diretório `examples/chapter15` no pacote de códigos-fontes.

Na próxima seção, você encontrará uma "colinha" que mostra as complexidades dos algoritmos implementados neste livro.

Se quiser uma versão impressa da folha de cola com a notação big-O, o link a seguir contém uma boa versão: `http://www.bigocheatsheet.com` (observe que, para algumas estruturas de dados como pilhas e filas, implementamos neste livro uma versão melhorada da estrutura de dados e, desse modo, temos uma complexidade big-O menor do que aquela exibida no link citado.

Estruturas de dados

A tabela a seguir mostra as complexidades das estruturas de dados.

Estrutura de dados	Casos médios			Piores casos		
	Inserção	Remoção	Busca	Inserção	Remoção	Busca
Array/Pilha/Fila	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Lista ligada	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Lista duplamente ligada	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Tabela hash	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Árvore binária de busca	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(n)$	$O(n)$	$O(n)$
Árvore AVL	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$
Árvore rubro-negra	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$
Heap binário	$O(log(n))$	$O(log(n))$	$O(1)$: encontrar máx./mín.	$O(log(n))$	$O(log(n))$	$O(1)$

Grafos

A tabela a seguir mostra as complexidades para os grafos.

Gerenciamento de nós/arestas	Tamanho do repositório	Adição de vértice	Adição de aresta	Remoção de vértice	Remoção de aresta	Consulta
Lista de adjacências	$O(\|V\| + \|E\|)$	$O(1)$	$O(1)$	$O(\|V\| + \|E\|)$	$O(\|E\|)$	$O(\|V\|)$
Matriz de adjacências	$O(\|V\|2)$	$O(\|V\|^2)$	$O(1)$	$O(\|V\|^2)$	$O(1)$	$O(1)$

Algoritmos de ordenação

A tabela a seguir mostra as complexidades dos algoritmos de ordenação.

Algoritmo (aplicado a um array)	Complexidade do tempo		
	Melhores casos	Casos médios	Piores casos
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n\ log(n))$	$O\ (n\ log^2(n))$	$O\ (n\ log^2(n))$
Merge sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(n\ log(n))$
Quick sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(n^2)$
Heap sort	$O(n\ log(n))$	$O(n\ log(n))$	$O(n\ log(n))$
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$
Radix sort	$O(nk)$	$O(nk)$	$O(nk)$

Algoritmos de busca

A tabela a seguir mostra as complexidades dos algoritmos de busca.

Algoritmo	Estrutura de dados	Piores casos
Busca sequencial	Array	$O(n)$
Busca binária	Array ordenado	$O(log(n))$
Busca por interpolação	Array ordenado	$O(n)$
Busca em profundidade (DFS)	Grafo com $\|V\|$ vértices e $\|E\|$ arestas	$O(\|V\| + \|E\|)$
Busca em largura (BFS)	Grafo com $\|V\|$ vértices e $\|E\|$ arestas	$O(\|V\| + \|E\|)$

Introdução à teoria de NP-completo

Em geral, dizemos que um algoritmo é eficiente se tiver complexidade $O(n^k)$ para alguma constante k, e ele é chamado de algoritmo polinomial.

Dado um problema em que há um algoritmo polinomial mesmo para o pior caso, o algoritmo é representado por P (polinomial).

Há outro conjunto de algoritmos chamado **NP** (**Nondeterministic Polynomial**, ou Polinomial Não Determinístico). Um problema NP é um problema para o qual a solução pode ser verificada em um tempo polinomial.

Se um problema P tiver um algoritmo que execute em tempo polinomial, podemos também verificar a sua solução em tempo polinomial. Então, é possível concluir que *P* é um subconjunto de *NP* ou é igual a ele. No entanto, não sabemos se $P = NP$.

Problemas NP-completos são os mais difíceis de um conjunto NP. Um problema de decisão *L* será NP-completo se:

1. *L* está em NP (isto é, qualquer dada solução para problemas NP-completos pode ser verificada rapidamente, mas não há nenhuma solução eficiente conhecida).
2. Todo problema em NP pode ser reduzido a *L* em tempo polinomial.

Para compreender o que é a redução de um problema, considere *L* e *M* como dois problemas de decisão. Suponha que o algoritmo *A* resolva *L*. Isso quer dizer que, se *y* for uma entrada para *M*, o algoritmo *B* responderá *Sim* ou *Não* conforme *y* pertencer a *M* ou não. A ideia é encontrar uma transformação de *L* para *M* de modo que o algoritmo *B* faça parte de um algoritmo *A* para resolver *A*.

Também temos outro conjunto de problemas chamado **NP-difícil** (NP-hard). Um problema será NP-difícil se tiver a propriedade 2 (de NP-completo) e não precisar ter a propriedade 1. Assim, o conjunto NP-completo também é um subconjunto do conjunto NP-difícil.

Saber se $P = NP$ ou não é uma das principais perguntas em ciência da computação. Se alguém descobrir a resposta para essa pergunta, haverá um grande impacto em criptografia, pesquisa de algoritmos, inteligência artificial e muitas outras áreas.

A seguir, vemos o diagrama de Euler para os problemas **P**, **NP**, **NP-completo** e **NP-difícil**, considerando que $P <> NP$.

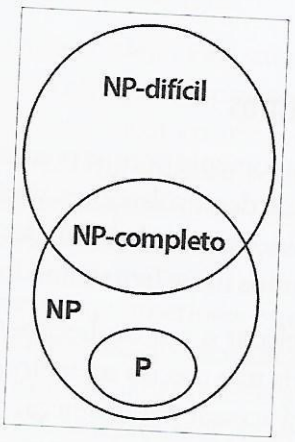

Figura 15.2

Como exemplos de problemas NP-difícil que não sejam problemas NP-completos, podemos mencionar o **problema da parada** (halting problem) e o **SAT** (**Boolean Satisfiability Problem**, ou Problema de Satisfação Booleana).

Como exemplos de problemas NP-completos, podemos mencionar também o problema da soma de subconjuntos (subset sum problem), o problema do caixeiro-viajante (traveling salesman problem) e o problema da cobertura de vértices (vertex cover problem).

Para mais informações sobre esses problemas, acesse `https://en.wikipedia.org/wiki/NP-completeness`.

Problemas impossíveis e algoritmos heurísticos

Alguns dos problemas mencionados são impossíveis de resolver. No entanto, algumas técnicas podem ser usadas para obter uma solução aproximada em um intervalo de tempo satisfatório. Uma técnica seria usar algoritmos heurísticos. Uma solução gerada por métodos heurísticos talvez não seja a melhor das soluções, mas será boa o suficiente para resolver o problema na ocasião.

Alguns exemplos de métodos heurísticos são: busca local, algoritmos genéticos, roteamento heurístico e aprendizado de máquina (machine learning). Para obter mais informações, acesse `https://en.wikipedia.org/wiki/Heuristic_(computer_science)`.

Métodos heurísticos são uma maneira ótima e divertida de tentar resolver um problema. Você pode tentar escolher um problema e desenvolver um método heurístico para o seu trabalho de conclusão de curso ou como tese de mestrado.

Divertindo-se com algoritmos

Não estudamos os algoritmos somente porque precisamos entendê-los na faculdade ou porque queremos nos tornar desenvolvedores. Você pode vir a ser um profissional mais bem-sucedido se aperfeiçoar suas habilidades para resolução de problemas usando os algoritmos que vimos neste livro como forma de solucionar problemas.

A melhor maneira de aperfeiçoar o seu conhecimento sobre resolução de problemas é praticando, e essa tarefa não precisa ser tediosa. Nesta seção, apresentaremos alguns sites que você poderá acessar para começar a se divertir com os algoritmos (e até mesmo ganhar um pouco de dinheiro enquanto faz isso!).

Eis uma lista de alguns sites úteis (alguns deles não aceitam uma solução escrita em JavaScript, mas podemos aplicar a lógica discutida neste livro também em outras linguagens de programação).

- **UVa Online Judge** (`http://uva.onlinejudge.org/`): esse site contém um conjunto de problemas usados em vários concursos de programação pelo mundo, incluindo o **International Collegiate Programming Contest** (**ICPC**) da ACM, patrocinado pela IBM. (Se você ainda está na faculdade, experimente participar desse concurso; se sua equipe vencer, você poderá viajar pelo mundo com todas as despesas pagas!) O site contém centenas de problemas para os quais podemos usar os algoritmos que aprendemos neste livro.
- **Sphere Online Judge** (`http://www.spoj.com/`): esse site é semelhante ao UVa Online Judge, mas aceita mais linguagens (incluindo submissões em JavaScript).
- **Coderbyte** (`http://coderbyte.com/`): esse site contém problemas (níveis fácil, médio e difícil) que também podem ser resolvidos com JavaScript.
- **Project Euler** (`https://projecteuler.net/`): esse site contém uma série de problemas de matemática/programação. Tudo que você tem a fazer é fornecer a resposta para o problema, mas podemos usar algoritmos para descobrir a resposta para nós.
- **HackerRank** (`https://www.hackerrank.com`): esse site contém desafios divididos em 16 categorias (você pode usar os algoritmos que vimos neste livro e muitos outros). Também aceita JavaScript, entre outras linguagens.
- **CodeChef** (`http://www.codechef.com/`): esse site também contém diversos problemas e organiza competições online.
- **Top Coder** (`http://www.topcoder.com/`): esse site organiza torneios de programação, geralmente patrocinados por empresas como NASA, Google, Yahoo, Amazon e Facebook. Alguns concursos oferecem oportunidades para trabalhar na empresa patrocinadora, enquanto outros podem oferecer prêmios em dinheiro. O site também disponibiliza ótimos tutoriais para resolução de problemas e algoritmos.

Outro aspecto interessante sobre os sites anteriores é que, em geral, eles apresentam um problema do mundo real, e é necessário identificar o algoritmo que pode ser usado para resolvê-lo. É uma forma de saber que os algoritmos que conhecemos neste livro não são apenas didáticos, mas também podem ser aplicados para resolver problemas do mundo real.

Estruturas de dados e algoritmos em JavaScript

Escreva um código JavaScript complexo e eficaz usando a mais recente ECMAScript

2ª Edição

Loiane Groner

Packt>

Novatec