Análise de Algoritmos

Maggie Johnson
Universidade de Stanford

Tradução: Flávio Velloso Laper
Universidade Fumec

Fevereiro de 2013

Sumário

Introdução

Este trabalho é uma tradução de alguns documentos sobre análise de algoritmos escritos por Maggie Johnson da Universidade de Stanford. Seu objetivo é servir como referência para a disciplina "Análise de Algoritmos" ministrada no curso de Ciência da Computação da Universidade Fumec. Por algum tempo, procuramos por um texto básico para essa disciplina que apresentasse algumas características fundamentais: ele deveria ser simples e didático (para ser facilmente utilizado por alunos de graduação), preciso e objetivo (a disciplina tem apenas quarenta horas semestrais), e em língua portuguesa. O trabalho original tem todas essas características exceto a última, o que motivou sua tradução.

Supomos que os alunos já tenham conhecimentos básicos em matemática discreta e computação. As estruturas de dados básicas (listas encadeadas, árvores, etc.) devem ser familiares, bem como os algoritmos recursivos e os de ordenação (seleção, inserção, bolha, mergesort, heapsort, quicksort, etc.). Habilidade de programação em alguma linguagem de alto nível (C, C++, Java, Pascal) é indispensável.

Procuramos ser sempre fiéis ao texto original, embora algumas pequenas alterações tenham sido introduzidas:

- O texto da seção sobre classes de problemas (seção 1.8) foi um pouco alterado para deixar mais clara a distinção entre problemas NP e NP-completo; além disso, alguns termos receberam uma definição mais formal.
- A definição sobre problemas indecidíveis (p. 15) foi acrescentada à lista de categorias de problemas original.
- A explicação da estratégia *branch and bound* (seção 3.3) do capítulo de exploração de algoritmos estava um tanto confusa e também sofreu algumas alterações.
- Algumas notas de rodapé foram inseridas para fornecer detalhes extras aos assuntos em questão. Todas essas notas estão identificadas com a abreviação N.T. (nota do tradutor).
- Alguns *links* da Web, mencionados no trabalho original, estão inativos e foram substituídos por outros equivalentes.

Para versões futuras, esperamos fazer algumas alterações mais profundas de forma a deixar o trabalho mais completo e mais adequado à disciplina ministrada na Universidade Fumec. As seguintes alterações estão planejadas:

- Acréscimo de exemplos extras de análise de algoritmos não recursivos: busca binária (para introduzir um algoritmo de complexidade $O(\log n)$) e soma máxima de subsequências de vetores de inteiros (para apresentar um caso com soluções cujas complexidades variam de $O(n)$ a $O\left(n^3\right)$).
- Inclusão de capítulo sobre análise de correção de algoritmos.
- Inclusão de capítulo sobre análise amortizada.
- Inclusão de capítulo sobre análise prática de algoritmos.

Os documentos originais[1]podem ser encontrados na Web. Todos foram licenciados sob os termos da Creative Commons Attribution 2.5 License. Essa licença, basicamente, permite copiar, distribuir, exibir e criar trabalhos derivados (inclusive traduções e adaptações) do trabalho original, desde que para fins não comerciais e mantendo os créditos para o autor original. Além disso, a licença original deve ser mantida. Esperamos ter obedecidos a todos esses critérios. Assim, o presente trabalho é licenciado sob os seguintes termos:

> *Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.*

[1] `http://code.google.com/p/gcu/downloads/list`; ver *label* "stanford-algorithms".

O tradutor agradece aos coordenadores do curso de Ciência da Computação da Universidade Fumec, os professores Air Rabelo, Becsom Salles de Carvalho e Emerson Eustáquio Costa, o apoio e o incentivo recebidos. Quaisquer erros encontrados, bem como críticas e sugestões, podem ser encaminhados ao *e-mail* flavio.laper@fumec.br.

1 O caso não-recursivo

Tópicos principais:

- Introdução
- Generalização do tempo de execução
- Execução de uma análise de tempo
- Notação O-grande
- Operações O-grande
- Análise de programas simples — sem chamadas de subprogramas
- Análise de pior caso e de caso médio
- Análise de programas com chamadas não-recursivas de subprogramas
- Classes de problemas

1.1 Por que analisar algoritmos?

Podem existir diversas maneiras diferentes de solucionar um problema particular. Por exemplo, há diversos métodos para ordenar números. Como você pode decidir qual é o melhor método em uma determinada situação? Como você definiria "melhor" — é o método mais rápido ou aquele que ocupa o menor espaço de memória?

Entender a eficiência relativa de algoritmos projetados para realizar a mesma tarefa é muito importante em todas as áreas da computação. É assim que os cientistas da computação decidem qual algoritmo usar para uma aplicação em particular. Nas décadas de 1950 e 1960, muitos matemáticos e cientistas da computação desenvolveram o campo da análise de algoritmos. Um pesquisador em particular, Donald Knuth, escreveu um texto de três volumes denominado *The Art of Computer Programming*[2] que fundamentou esse assunto. Curiosamente, o Prof. Knuth lecionava em Stanford até a sua aposentadoria a vários anos atrás. Ele agora está se dedicando a escrever mais volumes de sua obra seminal, e ocasionalmente dá palestras no *campus*[3].

Como mencionado acima, um algoritmo pode ser analisado em termos de sua eficiência na utilização do tempo ou espaço. Nós iremos nos preocupar agora apenas com a primeira. O tempo de execução de um algoritmo é influenciado por diversos fatores:

1. A velocidade da máquina que executa o programa.
2. A linguagem na qual o programa foi escrito. Por exemplo, programas escritos em *assembly* geralmente rodam mais rápido que aqueles escritos em C ou C++, que por sua vez tendem a rodar mais rápido que aqueles escritos em Java.

[2] A Arte da Programação de Computadores (N.T.).

[3] Como o texto original foi escrito em Stanford, ele procura destacar a importante participação do Prof. Knuth naquela instituição (N.T.).

3. A eficiência do compilador que criou o programa.

4. O tamanho da entrada: processar 1000 registros leva mais tempo que apenas 10.

5. A organização da entrada: se o item que estamos procurando está no topo da lista, será necessário menos tempo para encontrá-lo do que se ele estivesse no fundo.

Os três primeiros itens da lista são problemáticos. Nós não queremos usar uma medida exata do tempo de execução: dizer que um algoritmo em particular, escrito em Java, rodando em um determinado processador[4] leva um determinado número de milissegundos para executar não nos diz nada a respeito da eficiência de tempo geral do algoritmo, porque esta medida é específica para o ambiente em questão. Ela não terá utilidade alguma para alguém em um ambiente diferente. Precisamos de uma métrica genérica para a eficiência de tempo de um algoritmo; uma que seja independente das velocidades do processador e da linguagem, ou da eficiência do compilador.

O quarto item da lista não é específico de um ambiente, mas é uma consideração importante. Um algoritmo rodará mais devagar se precisa processar mais dados, mas este decréscimo na velocidade não se deve à construção do algoritmo. Isto acontece simplesmente porque há mais trabalho a realizar. Como resultado dessa consideração, nós normalmente expressamos o tempo de execução de um algoritmo como uma função do tamanho da entrada. Portanto, se o tamanho da entrada é n, nós expressamos o tempo de execução como $T(n)$. Dessa forma nós levamos em conta o tamanho da entrada, mas não o consideramos como um elemento definidor do algoritmo.

Finalmente, o último item da lista exige que consideremos outro aspecto da entrada, que normalmente não é parte intrínseca do algoritmo. Para levá-lo em conta, nós expressamos a análise de tempo em termos de "pior caso", "caso médio" ou "melhor caso" com base na organização dos dados, ou da probabilidade de encontrar um elemento rapidamente. Para nossos propósitos, nas próximas seções, nós assumiremos uma organização de "pior caso" (ou seja, não nos preocuparemos com a organização da entrada por enquanto).

1.2 Generalização do tempo de execução

Para tratar o problema de generalizar uma análise de tempo, é costume lançar mão de ordens de magnitude ou taxas de crescimento no lugar de utilizar diretamente números exatos. Em outras palavras, procura-se responder à pergunta:

Como o tempo de execução de um algoritmo varia à medida que o tamanho da entrada cresce?

Eles crescem juntos? Um cresce mais rapidamente que o outro — e quão mais rápido? A situação ideal é aquela na qual o tempo de execução cresce vagarosamente quando mais entrada é adicionada. Então, no lugar de lidar com valores exatos, mantemos a generalidade pela comparação do crescimento do tempo de execução, à medida que a entrada cresce, com o crescimento de funções conhecidas. As funções a seguir são tipicamente utilizadas:

Tamanho da entrada (n)	(1)	$\log n$	n	$n \log n$	n^2	n^3	2^n
5	1	3	5	15	25	125	32
10	1	4	10	33	100	10^3	10^3
100	1	7	100	644	10^4	10^6	10^{30}
1000	1	10	1000	10^4	10^6	10^9	10^{300}
10.000	1	13	10.000	10^5	10^8	10^{12}	10^{3000}

Com um computador rápido, quase não é possível notar diferenças nos tempo de execução para entradas pequenas. Para uma entrada de 100.000, entretanto, assumindo que cada instrução consome 1 μs, um algoritmo comparável à função $n \log n$ levará cerca de 1,7 s de CPU. um algoritmo n^2 cerca de 2,8 h de CPU (o que é inaceitável), e um algoritmo n^3 precisaria de 31,7 *anos* de CPU. Não há a menor possibilidade de utilizar esse algoritmo para lidar com entradas grandes.

[4] O texto original menciona o processador Pentium IV. Optou-se por utilizar um termo geral para evitar que o texto se torne desnecessariamente desatualizado (N.T.).

1.3 Análise do tempo de execução

Considera-se que $T(n)$, ou o tempo de execução de um algoritmo em particular para uma entrada de tamanho n, é o número de vezes que as instruções de um algoritmo são executadas[5]. Como uma ilustração, considere o algoritmo em pseudo-código abaixo que calcula a média de um conjunto de n números:

```
1 n ← lê entrada do usuário;
2 soma ← 0;
3 i ← 0;
4 enquanto i < n faça
5 |   número ← lê entrada do usuário;
6 |   soma ← soma + número;
7 |   i ← i + 1;
8 média ← soma/n;
```

Instruções 1, 2 e 3 são executadas apenas uma vez. Instruções 5, 6 e 7 são executadas n vezes. A instrução 4 (que controla o laço) é executada $n+1$ vezes (uma verificação adicional é necessária — por quê?), e a instrução 8 é executada uma vez. Isto está resumido abaixo:

Instrução	Número de execuções
1	1
2	1
3	1
4	$n+1$
5	n
6	n
7	n
8	1

Portanto, o cálculo do tempo para este algoritmo em termos do tamanho da entrada é: $T(n) = 4n + 5$. Nós podemos ver, intuitivamente, que à medida que n cresce, o valor desta expressão cresce linearmente. Dizemos que $T(n)$ tem uma "ordem de magnitude (ou taxa de crescimento) de n". Isto é geralmente indicado com a notação O-grande: $T(n) = O(n)$, e diz-se que o algoritmo tem uma *complexidade* de $O(n)$. Em alguns casos, pode-se dizer que o algoritmo tem uma "complexidade temporal" de $O(n)$ para distinguir sua taxa de crescimento do tempo da quantidade de memória, ou espaço, que ele usaria durante sua execução. Obviamente, intuição não é suficiente para nós, tipos céticos da ciência da computação — precisamos ser capazes de mostrar matematicamente qual das funções padronizadas listadas na tabela acima mostra a taxa de crescimento correta. O significado formal da notação O-grande é mostrado a seguir.

Notação O-grande

Definição 1: sejam $f(n)$ e $g(n)$ duas funções. Escreve-se

$$f(n) = O(g(n)) \text{ ou } f = O(g)$$

(leia-se "f de n é O-grande de g de n" ou "f é O-grande de g") se existe um inteiro positivo C tal que $f(n) \leq C \cdot g(n)$ para todo inteiro positivo n.

A idéia básica da notação O-grande é esta: suponha que f e g são ambas funções de valores reais de uma variável real x. Se, para valores grandes de x, o gráfico de f fica próximo do eixo horizontal do que o gráfico de algum múltiplo de g, então f é da ordem de g, ou seja, $f(x) = O(g(x))$. Então $g(x)$ representa um *limite superior* para $f(x)$.

Exemplo 1

Suponha que $f(n) = 5n$ e $g(n) = n$. Para mostrar que $f = O(g)$, precisamos mostrar a existência de uma constante C tal como descrito na Definição 1. Claramente 5 é uma constante tal que $f(n) = 5g(n)$.

[5]Nesta consideração, assume-se tacitamente que todas as instruções precisam do mesmo tempo para serem executadas. Trata-se, obviamente, de uma aproximação para simplificar os cálculos (N.T.).

Poderíamos escolher uma constante maior tal como 6, porque a definição exige que $f(n)$ seja menor ou igual a $C \cdot g(n)$, mas usualmente tentamos encontrar a menor. Assim, a constante C existe (nós só precisamos de uma) e $f = O(g)$. □

Exemplo 2

Na análise de tempo anterior, nós encontramos $T(n) = 4n+5$ e concluímos intuitivamente que $T(n) = O(n)$ porque o tempo de execução cresce linearmente à medida que n cresce. Agora, entretanto, podemos provar isto matematicamente.

Para mostrar que $f(n) = 4n + 5 = O(n)$, precisamos de uma constante C tal que:

$$f(n) \leq Cn \text{ para todo } n.$$

Se tentarmos $C = 4$ não funcionará, porque $4n + 5$ não é menor que $4n$. Precisamos que C seja pelo menos 9 para cobrir *todo* n. Se $n = 1$, C deve ser 9, mas C pode ser menor para valores maiores de n (se $n = 100$, C pode ser 5). Como o C escolhido deve funcionar para todo n, precisamos usar 9:

$$4n + 5 \leq 4n + 5n = 9n.$$

Como obtivemos uma constante C que funciona para todo n, podemos concluir:

$$4n + 5 = O(n). \square$$

Exemplo 3

Suponha-se que $f(n) = n^2$. Nós iremos provar que $f(n) \neq O(n)$. Para isto, precisamos mostrar que não existe uma constante C que satisfaça a definição O-grande. A prova será feita por contradição.

Suponha que exista uma constante C que funcione; então, pela definição, $n^2 \leq Cn$ para *todo* n. Suponha que n seja qualquer número positivo real maior que C. Então $n \cdot n > C \cdot n$, ou $n^2 > Cn$. Então existe um número real n tal que $n^2 > Cn$. Isto contradiz a suposição, portanto a suposição é falsa. Não existe C que funcione para todo n: $f(n) \neq O(n)$ quando $f(n) = n^2$. □

Exemplo 4

Suponha que $f(n) = n^2 + 3n - 1$. Queremos mostrar que $f(n) = O\left(n^2\right)$.

$$\begin{aligned} f(n) &= n^2 + 3n - 1 \\ &< n^2 + 3n \qquad \text{(subtração faz as coisas decrescerem, então descarte-a)} \\ &\leq n^2 + 3n^2 \qquad \text{(porque } n \leq n^2 \text{ para todo inteiro } n) \\ &= 4n^2. \end{aligned}$$

Portanto, se $C = 4$, mostramos que $f(n) = O\left(n^2\right)$. Note que tudo que fizemos foi encontrar uma função simples que é um limite superior para a função original. Por causa disto, também poderíamos dizer que $f(n) = O\left(n^3\right)$, uma vez que n^3 é um limite superior para n^2. Esta seria uma descrição mais fraca, mas ainda válida. □

Exemplo 5

Mostrar que $f(n) = 2n^7 - 6n^5 + 10n^2 - 5 = O\left(n^7\right)$:

$$\begin{aligned} f(n) &< 2n^7 + 6n^5 + 10n^2 \\ &\leq 2n^7 + 6n^7 + 10n^7 \\ &= 18n^7 \end{aligned}$$

e, portanto, com $C = 18$ mostramos que $f(n) = O\left(n^7\right)$. □

Você provavelmente está notando um padrão aqui. *Todo polinômio é O-grande do seu termo de maior grau*. Nós também estamos ignorando constantes. Uma prova disto é bastante direta. Qualquer polinômio (incluindo um generalizado) pode ser manipulado para satisfazer a definição O-grande da mesma forma que fizemos no último exemplo: toma-se o valor absoluto de cada coeficiente (isto só pode aumentar a função); então como

$$n^j \leq n^d \text{ se } j \leq d$$

podemos mudar os expoentes de todos os termos para o grau mais alto (a função original deve também ser menor que essa). Finalmente, adicionamos esses termos para obter a maior constante C que precisamos para encontrar uma função que seja um limite superior da original.

O-grande, portanto, é um método útil para caracterizar uma aproximação do limite superior do tempo de execução de um algoritmo. Ao ignorar as constantes e os termos de menor grau, terminamos com um limite superior aproximado. Em muitos casos, isto é suficiente. Algumas vezes, entretanto, isto não basta para comparar os tempos de execução de dois algoritmos. Por exemplo, se um algoritmo roda em tempo $O(n)$ e outro em tempo $O\left(n^2\right)$, não se pode afirmar qual é o mais rápido para n grande. Mais provavelmente o primeiro seja o mais rápido, mas talvez o segundo não tenha sido analisado com cuidado. É importante lembrar que a notação O-grande não dá uma informação precisa sobre quão bom um algoritmo é, mas apenas um *limite superior* para quão ruim ele pode ser.

Há uma outra notação que por vezes é útil na análise de algoritmos. Para especificar um *limite inferior* para a taxa de crescimento de $T(n)$, podemos usar a notação Ω-grande (ômega-grande). Se o menor tempo de execução de um algoritmo é $g(n)$, dizemos $T(n) = \Omega(g(n))$. Por exemplo, qualquer algoritmo com m entradas e n saídas que use todas as entradas para gerar a saída iria precisar de um trabalho pelo menos $\Omega(m+n)$. *Grosso modo*, a notação Ω-grande diz que um algoritmo precisa pelo menos de um certo tempo para executar, e portanto é um *limite inferior* para o tempo de execução ou, alternativamente, pode ser encarada como o *melhor caso* para este tempo.

Uma última variação deste tema é a notação Θ-grande (teta-grande). Θ-grande estabelece limites acima e abaixo de uma função, portanto duas constantes precisam ser fornecidas no lugar de uma única ao fazer as provas formais dos limites para uma dada função.

Discutiremos um pouco mais Ω-grande e Θ-grande em seguida. Antes disso, há um ajuste que precisa ser feito em nossa definição de O-grande. Você pode ter notado que, até agora, evitamos utilizar os logaritmos em nossa discussão. Nós não podemos evitá-los por mais tempo, porém, porque muitos algoritmos têm uma taxa de crescimento que condiz com funções logaritmas (alguns deles: busca binária, Mergesort e Quicksort). Como uma breve revisão, lebre-se que $\log_2 n$ é o número de vezes que precisamos dividir n por 2 para obter 1 ou, alternativamente, o número de 2's que precisamos multiplicar para obter n:

$$n = 2^k \Leftrightarrow \log_2 n = k.$$

Muitos algoritmos de "Divisão e Conquista" resolvem um problema ao dividi-lo em dois problemas menores, e depois em dois problemas ainda menores. A divisão continua até chegar ao ponto em que a solução do problema é trivial. Esta divisão constante por dois sugere um tempo de execução logarítmico.

Assim, como $\log(1) = 0$, não existe constante C tal que $1 \leq C \cdot \log n$ para todo n. Note, entretanto, que para todo $n \geq 2$, ocorre $1 \leq \log n$ e, portanto, a constante $C = 1$ funciona para n suficientemente grande (maior que 1). Isto sugere que precisamos de uma definição para a notação O-grande mais forte que a que foi dada anteriormente.

Definição 2: sejam $f(n)$ e $g(n)$ duas funções. Escreve-se:

$$f(n) = O(g(n)) \text{ ou } f = O(g)$$

se existem inteiros positivos C e N tais que $f(n) \leq C \cdot g(n)$ para todo inteiro $n \geq N$.

Usando esta definição mais geral para O-grande, pode-se dizer que se $f(n) = 1$, então $f(n) = O(\log(n))$ porque $C = 1$ e $N = 2$ funcionarão.

Com essa definição, a diferença entre os três tipo de notação pode ser vista com clareza. Nos três gráficos da Figura 1, n_0 é o menor valor possível para obter limites válidos, mas qualquer valor maior funcionará.

A Figura 1A mostra Θ-grande, que limita uma função entre dois fatores constantes (isto é, fornecendo um limite estrito *superior e inferior*). Pode-se escrever $f(n) = \Theta(g(n))$ se existem constantes positivas n_0, c_1

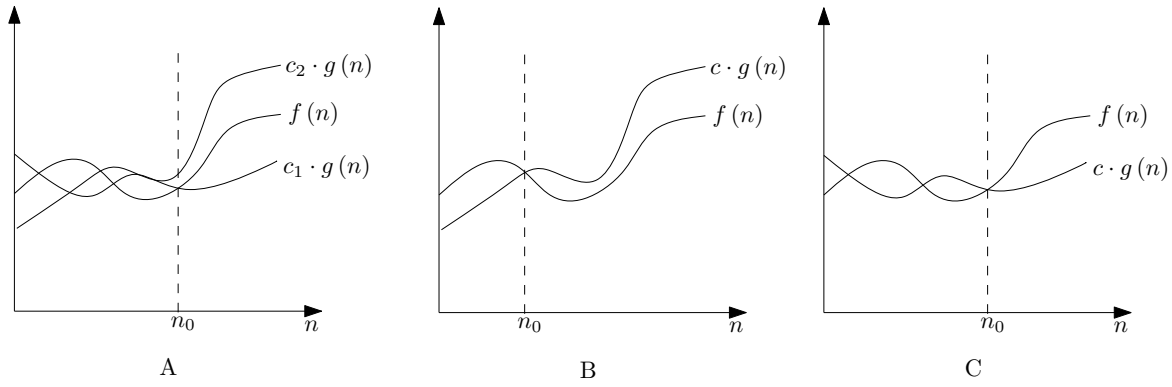


Figura 1: Gráficos das notações assintóticas

e c_2 tais que, à direita de n_0, o valor de $f(n)$ sempre esteja entre $c_1 g(n)$ e $c_2 g(n)$ (inclusive). Assim, para provar que uma função é $\Theta(g(n))$, nós devemos mostrar que esta função é tanto $O(g(n))$ quanto $\Omega(g(n))$.

A Figura 1B mostra O-grande, que fornece um *limite superior* para uma função a menos de um fator constante. Escreve-se $f(n) = O(g(n))$ se existem constantes positivas n_0 e c tais que, à direita de n_0, $f(n)$ sempre esteja sobre ou abaixo de $c \cdot g(n)$.

Finalmente, a Figura 1C mostra Ω-grande, que fornece um *limite inferior* para uma função a menos de um fator constante. Escreve-se $f(n) = \Omega(g(n))$ se existem constantes positivas n_0 e c tais que, à direita de n_0, $f(n)$ sempre esteja sobre ou acima de $c \cdot g(n)$.

Há um teorema muito prático que relaciona essas três notações:

Teorema: para duas funções quaisquer $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Exemplo 6

Mostrar que $f(n) = 3n^3 + 3n - 1 = \Theta\left(n^3\right)$. Como indicado pelo teorema acima, para mostrar este resultado, deve-se mostrar duas propriedades:

$$\begin{aligned} (1):\ f(n) &= O\left(n^3\right) \\ (2):\ f(n) &= \Omega\left(n^3\right) \end{aligned}$$

Primeiramente mostraremos (1), usando as mesmas técnicas que já vimos para O-grande. Consideraremos $N = 1$, e assim só precisaremos considerar $n \geq 1$ para mostrar o resultado para O-grande:

$$\begin{aligned} f(n) &= 3n^3 + 3n - 1 \\ &< 3n^3 + 3n + 1 \\ &\leq 3n^3 + 3n^3 + 1n^3 \\ &= 7n^3 \end{aligned}$$

e, portanto, com $C = 7$ e $N = 1$, mostramos que $f(n) = O\left(n^3\right)$.

Em seguida mostraremos (2). Agora precisamos fornecer um limite inferior para $f(n)$. Escolhe-se um valor para N, tal que o termo de maior ordem em $f(n)$ irá sempre dominar (ou seja, será maior que) os termos de menor ordem. Escolhe-se $N = 2$ porque, para $n \geq 2$, tem-se $n^3 \geq 8$. Isto faz com que n^3 seja maior que o restante do polinômio $(3n - 1)$ para todo $n \geq 2$. Então, subtraindo-se um termo n^3 extra, é possível formar um polinômio que será sempre menor que $f(n)$ para $n \geq 2$:

$$\begin{aligned} f(n) &= 3n^3 + 3n - 1 \\ &> 3n^3 - n^3 \ \text{(porque } n^3 > 3n - 1 \text{ para todo } n \geq 2) \\ &= 2n^3. \end{aligned}$$

Assim, com $C = 2$ e $N = 2$, mostrou-se que $f(n) = \Omega\left(n^3\right)$, porque $f(n)$ é sempre maior que $2n^3$. □

1.4 Operações O-grande

A maior parte do restante de nossa discussão sobre complexidade computacional será focado em limites O-grande, uma vez que este é o principal meio de caracterizar o tempo de execução de algoritmos (especialmente no pior caso). Em nosso uso de O-grande, nós estaremos interessados em um limite *estrito* (análogo ao Θ-grande, mas sem a necessidade de provar o limite inferior (Ω-grande) explicitamente). Assim, nós normalmente tentaremos encontrar a menor função válida que caracterize o tempo de execução do algoritmo. Portanto, embora *tecnicamente* uma função que seja $O(n)$ também seja $O\left(n^2\right)$, nós sempre caracterizaremos essa função como $O(n)$, considerando inapropriada a caracterização como $O\left(n^2\right)$, porque ela não é tão útil quanto o limite estrito.

Regra da soma
Suponha que $T_1 = O(f_1(n))$ e $T_2 = O(f_2(n))$. Além disso, suponha que f_2 não cresça mais rápido que f_1, isto é, $f_2(n) = O(f_1(n))$. Então podemos concluir que $T_1(n) + T_2(n) = O(f_1(n))$. De maneira geral, a regra da soma diz que $O(f_1(n) + f_2(n)) = O(\max(f_1(n), f_2(n)))$.

Prova: suponha-se que C e C' sejam constantes tais que $T_1(n) \leq C \cdot f_1(n)$ e $T_2(n) \leq C' \cdot f_2(n)$. Seja D igual ao maior entre C e C'. Então,

$$\begin{aligned} T_1(n) + T_2(n) &\leq C \cdot f_1(n) + C' \cdot f_2(n) \\ &\leq D \cdot f_1(n) + D \cdot f_2(n) \\ &\leq D \cdot (f_1(n) + f_2(n)) \\ &= O(f_1(n) + f_2(n)). \end{aligned}$$

Regra do produto
Suponha que $T_1(n) = O(f_1(n))$ e $T_2(n) = O(f_2(n))$. Então podemos concluir que $T_1(n) \cdot T_2(n) = O(f_1(n) \cdot f_2(n))$.

A regra do produto pode ser provada através de uma estratégia similar à da prova da regra da soma.

1.5 Análise de alguns programas simples (sem chamadas de subprogramas)

Regras gerais:

1. Todas as instruções básicas (atribuições, leituras, escritas, testes condicionais, chamadas de bibliotecas) rodam em tempo constante: $O(1)$.
2. O tempo de execução de um laço é a soma, sobre todas as iterações do laço, do tempo para executar as instruções no laço, mais o tempo para avaliar a condição de terminação. A avaliação de uma terminação básica é $O(1)$ em cada iteração do laço.
3. A complexidade de um algoritmo é determinada pela complexidade das instruções mais frequentemente executadas. Se um conjunto de instruções tem um tempo de execução $O\left(n^3\right)$ e o resto é $O(n)$, então a complexidade do algoritmo é $O\left(n^3\right)$. Este resultado é uma aplicação da regra da soma.

Exemplo 7

Calcular o O-grande do tempo de execução do seguinte fragmento de código C++:

```
1 for (i = 2; i < n; i++) {
2   sum += i;
3 }
```

O número de iterações de um laço **for** é igual ao valor superior do índice menos o valor inferior, mais uma instrução para levar em conta o teste da condição final. Nota: se a condição de terminação do laço **for** for i <= n no lugar de i < n, então o número de execuções do teste condicional será:

$$((\text{índice\_superior} + 1) - \text{índice\_inferior}) + 1.$$

Neste caso, temos $n-2+1=n-1$. A atribuição no laço é executada $n-2$ vezes. Então, temos $(n-1)+(n-2)=(2n-3)$ instruções executadas $=O(n)$. □

Problemas de complexidade podem pedir o "número de instruções executadas", o que significa que você deve fornecer uma equação em termos de n com o número preciso de instruções executadas. Ou apenas a complexidade pode ser solicitada; neste caso, você precisa fornecer apenas uma expressão O-grande (ou Θ-grande).

Exemplo 8

Considere o algoritmo de ordenação mostrado abaixo. Encontre o número de instruções executadas e a complexidade deste algoritmo.

```
1  for (i = 1; i < n; i++) {
2    SmallPos = i;
3    Smallest = Array[SmallPos];
4    for (j = i+1; j <= n; j++)
5      if (Array[j] < Smallest) {
6        SmallPos = j;
7        Smallest = Array[SmallPos];
8      }
9    Array[SmallPos] = Array[i];
10   Array[i] = Smallest;
11 }
```

A instrução 1 é executada n vezes $(n-1+1)$; instruções 2, 3, 9 e 10 (cada uma representando um tempo $O(1)$) são executadas $n-1$ vezes cada, uma para cada iteração do laço exterior. Na primeira passagem por esse laço com i $=1$, a instrução 4 é executada n vezes; instrução 5 $n-1$ vezes e, assumindo um pior caso onde os elementos do *array* estão em ordem decrescente, as instruções 6 e 7 (cada uma com tempo $O(1)$) são executadas $n-1$ vezes.

Na segunda passagem pelo laço exterior, com i $=2$, a instrução 4 é executada $n-1$ vezes e as instruções 5, 6 e 7 são executadas $n-2$ vezes, etc. Então, a instrução 4 é executada $n+(n-1)+\cdots+2$ vezes e as instruções 5, 6 e 7 são executadas $(n-1)+(n-2)+\cdots+2+1$ vezes. A primeira soma é igual a $n(n+1)/2-1$, e a segunda é igual a $n(n-1)/2$.

Portanto, o tempo de execução total é:

$$\begin{aligned} T(n) &= n+4(n-1)+n(n+1)/2-1+3[n(n-1)/2] \\ &= n+4n-4+\left(n^2+n\right)/2-1+\left(3n^2-3n\right)/2 \\ &= 5n-5+\left(4n^2-2n\right)/2 \\ &= 5n-5+2n^2-n \\ &= 2n^2+4n-5 \\ &= O\left(n^2\right). \square \end{aligned}$$

Exemplo 9

O fragmento de programa a seguir inicializa uma matriz A de duas dimensões (com n linhas e n colunas) de forma a torná-la uma matriz identidade $n \times n$ — ou seja, uma matriz com 1's na diagonal e 0's nas demais posições. Mais formalmente, se A é uma matriz identidade $n \times n$, então:

$$A \times M = M \times A = M$$

para toda matriz M $n \times n$. Qual é a complexidade deste código C++?

```
1 cin >> n;
2 for(i = 1; i <= n; i++)
3   for(j = 1; j <= n; j++)
4     A[i][j] = 0;
5 for(i = 1; i <= n; i++)
6   A[i][j] = 1;
```

Um programa como este pode ser analisado por partes, e então a regra da soma pode ser usada para encontrar o tempo de execução total. A linha 1 leva um tempo $O(1)$. As instruções nas linhas 5 e 6 são executadas $O(n)$ vezes. As instruções nas linhas 3 e 4 são executadas n vezes a cada vez que o laço é executado. O laço externo da linha 2 é executado n vezes, resultando em uma complexidade temporal de $O\left(n^2\right)$, devido às n execuções do laço interno $O(n)$. Assim, o tempo de execução do fragmento é $O(1) + O\left(n^2\right) + O(n)$. Aplicando a regra da soma, concluímos que o tempo de execução total é $O\left(n^2\right)$. □

1.6 Análise de pior caso e caso médio

Até agora, temos analisado programas considerando uma entrada de tamanho n, ou seja, o número máximo de itens de entrada. Isto é análise de pior caso; estamos procurando pelo maior número de passos de execução necessário e utilizando este número para calcular o tempo de execução. Entretanto, podem existir situações em que se deseja saber qual o tempo de execução do algoritmo no caso médio, não necessariamente no pior caso. A análise de caso médio é bem mais complicada que a de pior caso.

O que se precisa fazer é determinar a probabilidade de se precisar de um determinado número de passos (dada uma entrada particular) para resolver um dado problema. É também necessário determinar a distribuição dos diversos valores dos dados. Por exemplo, se você está escrevendo um algoritmo de busca, quão frequentemente o valor procurado será o primeiro, ou o décimo, ou o milésimo elemento do *array* pesquisado? A partir dessas probabilidades, você pode realizar uma análise de caso médio, mas ela será completamente dependente daquilo que foi assumido para estabelecer as probabilidades. Esta é sempre a parte mais difícil da análise de caso médio. Por causa dessas complicações, a análise de pior caso é bem mais comum (mas menos precisa).

Apenas para ilustrar como a análise de caso médio funciona, examinaremos a seguir um exemplo.

Exemplo 10

Abaixo encontra-se um algoritmo simples de busca linear que retorna o índice da posição de um valor em um *array*.

```
1 /* a é o array  de tamanho n sendo pesquisado */
2 i = 0;
3 while((i < n) && (x != a[i]))
4   i++;
5 if(i < n)
6   location = i;
7 else
8   location = -1;
```

Nós realizaremos a análise de caso médio pressupondo que o elemento x encontra-se no *array*. Há n tipos diferentes de entradas possíveis se x está no *array*. A seguir, consideraremos apenas a execução do laço **while**. Se x é o primeiro elemento do *array*, a condição do laço **while** será executada e falhará, então apenas uma linha do laço é executada. Se x é o segundo elemento, a condição do **while** será executada, bem como a instrução de incremento. Então temos que verificar novamente a condição do laço, para um total de 3 linhas. De maneira geral, se x é o i-ésimo elemento do *array*, $2i - 1$ linhas são executadas. Então o número médio de linhas executadas é:

$$1 + 3 + 5 + \cdots + (2n - 1)/n = (2(1 + 2 + 3 + \cdots + n) - n)/n.$$

Sabendo que $1 + 2 + 3 + \cdots + n = n(n + 1)/2$, o número de linhas executado é:

$$(2(n(n + 1)/2) - n)/n = n = O(n).$$

Note que assumimos que é igualmente provável encontrar x em qualquer posição do *array*. □

1.7 Análise de programas com chamadas de subprogramas não recursivos

O primeiro passo é analisar cada função individual para obter seu tempo de execução. Nós começamos pelas funções que não chamam nenhuma outra função. Então, avaliamos os tempos de execução das funções que chamam as previamente avaliadas. Suponha que o tempo de execução de uma função P tenha sido calculado como $O(f(n))$. Portanto, qualquer instrução que chame P, em qualquer parte do programa, também terá um tempo de execução $O(f(n))$. Nós simplesmente incluímos esse tempo em nossa análise em qualquer seção do programa em que P seja chamada.

Para uma atribuição ou instrução de escrita que inclua chamadas de funções, o tempo de execução é determinado com a utilização da regra da soma para adicionar os tempos de todas as funções chamadas. Se uma chamada de função com tempo de execução $O(f(n))$ aparece em uma condição ou em um teste de laço, seu tempo também é levado em conta tal como descrito nas regras a seguir.

1. Laço *enquanto* ou *repita*: adicionar $f(n)$ a cada iteração, e multiplicar o tempo resultante pelo número de iterações. Para um laço *enquanto*, deve-se adicionar um $f(n)$ extra para o teste final do laço.

2. Laço *para*: se a chamada de função está na inicialização do laço, adicionar $f(n)$ ao tempo total do laço. Se a chamada de função faz parte da condição de terminação, adicionar $f(n)$ para cada iteração.

3. Condicional *se*: adicionar $f(n)$ ao tempo de execução da instrução.

Para ilustrar estes pontos, considere-se o programa a seguir:

```
1  int a, n, x;
2
3  int bar(int x, int n) {
4    int i;
5
6    for(i = 1; i < n; i++)
7      x = x + i;
8    return x;
9  }
10
11 int foo(int x, int n) {
12   int i;
13
14   for(i = 1; i <= n; i++)
15     x = x + bar(i, n);
16   return x;
17 }
18
19 void main(void) {
20   n = GetInteger();
21   a = 0;
22   x = foo(a, n);
23   printf("%d", bar(a, n));
24 }
```

Nós começamos a análise por bar() já que esta função não chama nenhum outro subprograma. O laço **for** adiciona cada um dos inteiros de 1 até $n-1$ a x. O *valor* retornado por bar(x, n) é $x+n(n-1)/2$. O tempo de execução de bar() é analisado da seguinte forma: a linha 6 é executada n vezes, e a linha 7 é executada $n-1$ vezes. A linha 8 é executada 1 vez. Então, o tempo de execução para bar() é $O(n)+O(1)=O(n)$. Em seguida, analisamos foo(). A linha 15 normalmente levaria tempo $O(1)$, mas ela tem uma chamada para bar(), então seu tempo é $O(1)+O(n)=O(n)$. O laço **for** das linhas 14 e 15 é executado n vezes, então

multiplicamos $O(n)$ do corpo do laço por n iterações para obter $O(n^2)$, que é o tempo de execução de para uma chamada de foo().

Finalmente, analisamos main(). As linhas 20 e 21 levam $O(1)$ cada uma. A chamada para foo() na linha 22 leva $O(n^2)$. O printf da linha 23 leva $O(1)$ mais uma chamada para bar() o que nos dá $O(1)+O(n)$. Portanto, o tempo total para main() é $O(1) + O(1) + O(n^2) + O(n) = O(n^2)$.

O exemplo a seguir fica como um exercício. Abaixo está o corpo de uma função:

```
1 sum = 0;
2 for(i = 1; i <= f(n); i++)
3   sum += i;
```

onde f(n) é uma chamada de função. Encontre um limite superior O-grande para essa função se o tempo de execução de f(n) é $O(n)$ e o *valor* de f(n) é $n!$.

1.8 Classes de problemas

Imagine que você decidiu esquiar pelas Sierras nas férias de inverno. Como não há McDonalds entre Yosemite e Lee Vining, você precisará trazer toda a sua comida em sua bagagem. Nesta viagem, sabor é uma preocupação secundária. O que é importante é o peso da comida e seu conteúdo calórico. Depois de assaltar sua cozinha, você descobre que tem aproximadamente 200 tipos diferentes de alimentos, cada um com um peso e um certo número de calorias:

1	Salgadinhos	200 calorias	100 gramas
2	Coca-Cola *Diet*	1 caloria	200 gramas
...	...	...	...
200	Espaguete desidratado	500 calorias	450 gramas

Dada esta seleção de comida, sua tarefa é encontrar o melhor subconjunto que maximize o número de calorias, porém dentro de um limite de peso que você consiga carregar em sua mochila (por exemplo, 15 kg). Um algoritmo que certamente é capaz de encontrar uma solução é simplesmente tentar toda possível combinação. Infelizmente, há 2^{200} combinações, e os Sierras serão erodidos antes que seu algoritmo termine.

Existe algum algoritmo que rode em um tempo razoável (isto é, polinomial)? Se existe, ninguém ainda foi capaz de encontrá-lo. Isto pode fazê-lo suspeitar que realmente não existe solução em tempo polinomial, mas ninguém foi capaz de provar que isto é verdade tampouco. Este problema, conhecido como *Problema da Mochila* (em inglês, *Knapsack*), é um exemplo de uma classe geral de problemas conhecido como *problemas NP*. São problemas para os quais nenhuma solução de tempo polinomial existe, mas permanece sem prova se uma solução com tempo exponencial é realmente necessária.

Problemas, problemas, problemas.

Ao estudar algoritmos, vimos as funções e a notação O-grande utilizada para caracterizar seu tempo de execução. Existem alguns termos importantes que são usados para classificar problemas com base no tempo de execução dos algoritmos que solucionam esses problemas. Abaixo encontra-se uma revisão das categorias de funções para a notação O-grande:

- Aquelas que envolvem n como um expoente (c^n, n^n ou $n!$, onde c é uma constante[6]) são chamadas funções *exponenciais*.
- Funções cujo crescimento é menor ou igual a n^c, onde c é uma constante relativamente pequena, são chamadas *polinomiais*.
  - Funções *lineares* têm crescimento proporcional a n.
  - Funções *sublineares* têm crescimento proporcional a $\log n$.
  - Uma função de *tempo constante* tem crescimento independente de n (você conhece algum algoritmo com tempo de execução de ordem constante?).

[6]Tem-se $n! \leq n^n = 2^{n \log n}$ (N.T.).

Estas descrições são frequentemente utilizadas para caracterizar diferentes conjuntos de problemas. As categorias gerais são:

Intratáveis/Exponenciais: são problemas que precisam de tempo exponencial. Além disso, já foi provado que uma solução com tempo inferior ao exponencial é impossível. Por exemplo, seja o problema de listar todos os possíveis comitês de tamanho um ou mais que podem ser formados a partir de um grupo de n pessoas. Há $2^n - 1$ possíveis comitês, portanto qualquer algoritmo que resolva este problema necessariamente precisa de, pelo menos, $2^n - 1$ passos e, portanto, o tempo de execução é proporcional a 2^n. Não é possível encontrar uma solução de tempo polinomial para este problema.

Polinomiais (P): são problemas para os quais existe uma solução sublinear, linear ou polinomial. Ordenação por inserção, *quicksort*, *mergesort* e outros algoritmos de ordenação são bons exemplos, assim como a busca pelo menor ou maior valor em uma lista. Qualquer algoritmo cujo tempo de execução cresça proporcionalmente a uma função polinomial pode resolver problemas de tamanho moderado, desde que a constante no expoente seja pequena.

NP: nenhuma solução polinomial foi encontrada, mas existe uma solução exponencial. Entretanto, não foi possível demonstrar que não existe solução polinomial. Alguns exemplos famosos:

1. Insanidade geral instantânea: dados n cubos, onde cada face de um cubo é pintada em uma dentre n cores, é possível empilhar os cubos de maneira que cada uma das n cores apareça exatamente uma vez em cada lado da pilha?
2. Caixeiro viajante (CV): dado um mapa de cidades e um custo de viagem entre cada par de cidades, é possível visitar cada cidade exatamente uma vez e retornar para casa por menos de k reais?
3. Organização em n prateleiras: você tem k livros para colocar em n prateleiras. Eles irão caber? (Os livros são todos de tamanhos diferentes.) Esta é apenas uma formulação diferente do problema da mochila.

Este grupo de problemas é muito interessante por diversas razões. Em primeiro lugar, "NP" significa "não-determinístico polinomial". Não-determinístico não significa que nós não somos capazes de dizer se o problema é polinomial. Significa que o algoritmo que poderíamos definir para resolver um desses problemas é não-determinístico, isto é, existe um elemento de sorte no algoritmo. Por exemplo, poderíamos escrever a seguinte solução para o problema CV:

```
1 Escolha um dos possíveis caminhos;
2 Calcule o custo total do caminho escolhido;
3 se o custo calculado não é maior que o custo permitido então
4 |  retorna sucesso;
5 senão
6 |  retorna nada;
```

Alguém precisa escolher um dos possíveis caminhos (esta é a parte não-determinística). Se a escolha for feita corretamente, o problema é rapidamente resolvido; caso contrário, não temos nenhuma nova informação. Nós definimos o tempo de um algoritmo NP como o tempo necessário para executar o algoritmo se fazemos a escolha "correta", indicando a escolha que levaria a uma solução em uma quantidade de tempo ótima. Assim, estamos tentando medir quão bom é o algoritmo, e não quão boas são nossas escolhas.

De acordo com esta definição, se os processos de escolha, cálculo e verificação são polinomiais, o problema CV tem uma solução polinomial. Se fizermos a escolha correta, temos apenas que calcular o custo para n cidades, que é proporcional a n.

Obviamente, a classe dos problemas NP inclui os problemas realmente polinomiais. Se somos capazes de projetar uma solução determinística (sem escolhas) polinomial para o problema, certamente podemos projetar uma não-determinística. Além disso, o problema CV sugere que podem existir problemas NP que não possuam soluções polinomiais. Em outras palavras, parecem existir problemas que podem ser resolvidos em tempo polinomial por algoritmos não-determinísticos, mas que precisam de tempo

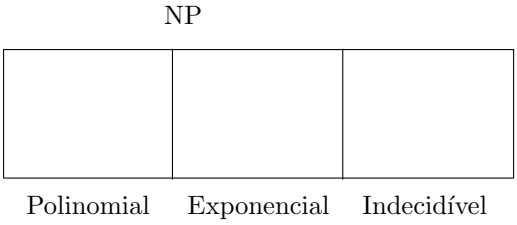


Figura 2: Classificação de problemas

exponencial para um determinístico. Entretanto, ninguém foi capaz de provar isto. Em particular, ninguém foi capaz de provar que não existe uma solução polinomial determinística para o problema CV. Esta é uma das questões em aberto em teoria da Ciência da computação (mas a maioria dos cientistas da computação pensam que tal solução não existe para o problema CV).

Indecidíveis: são problemas para os quais é possível obter uma prova de que não são solúveis em todas as situações, não importando quanto tempo ou espaço estejam disponíveis[7].

Então, temos a classificação de problemas mostrada na Figura 2.Vamos estudar alguns problemas NP com mais detalhes para entendê-los melhor.

Três problemas famosos

1. *Satisfabilidade* (SAT): consiste em determinar se qualquer fórmula lógica dada é satisfazível, ou seja, se existe uma forma de atribuir valores lógicos TRUE ou FALSE às variáveis lógicas de modo que o resultado seja TRUE.

   Seja uma fórmula lógica com as seguintes características:

   - Ela é composta de variáveis lógicas a, b, c, ..., e de seus complementos $\neg a$, $\neg b$, $\neg c$, ...
   - Ela representada por uma série de termos, onde cada termo é um OU lógico ($\vee$) das variáveis lógicas e seus complementos.
   - Ela é expressa através de ANDs lógicos ($\wedge$) desses termos.

   Existe alguma maneira de atribuir valores às variáveis de forma que o valor lógico da fórmula seja TRUE? Se tal atribuição existe, a fórmula é dita *satisfazível*. Exemplos (soluções deixadas como exercícios):

   - $a \wedge (b \vee c) \wedge (\neg c \vee \neg a)$ é satisfazível?
   - $a \wedge (b \vee c) \wedge (\neg c \vee \neg a) \wedge (\neg b)$ é satisfazível?

2. *Problema da mochila*: o nome do problema se refere ao empacotamento de itens em uma mochila. Existe uma maneira de selecionar os itens a empacotar de forma que sua "soma" (a quantidade de espaço ocupada) seja exatamente igual à capacidade da mochila? É possível expressar este problema como um caso de adição de inteiros: dado um conjunto de inteiros não negativos e um valor alvo, existe um subconjunto desses inteiros cuja soma seja igual ao valor alvo?

   Formalmente, dado um conjunto $\{a_1, a_2, \ldots, a_n\}$ e uma soma alvo T, onde $a_i \geq 0$, existe um vetor de seleção $V = [v_1, v_2, \ldots, v_n]$ onde v_i é 0 ou 1, tal que:

   $$\sum_{i=1}^{n} (a_i \cdot v_i) = T.$$

   Por exemplo, o conjunto pode ser $\{4, 7, 1, 12, 10\}$. Uma solução para a soma alvo $T = 17$ existe quando $V = [1, 0, 1, 1, 0]$ uma vez que $4 + 1 + 12 = 17$. Nenhuma solução existe se $T = 25$.

[7]Mais precisamente, um problemas indecidível é um problema de decisão (problema cuja solução se reduz a uma resposta "sim" ou "não") para o qual não é possível construir um algoritmo que forneça uma resposta "sim" ou "não" correta em todas as situações (N.T.).

3. *Problema dos cliques em teoria dos grafos*: um clique de um grafo é um subconjunto de seus vértices tais que todos os vértices desse subconjunto estão conectados (dois a dois) por uma aresta (ou seja, todo par de vértices do clique é adjacente). O tamanho de um clique é a quantidade de seus vértices. O problema dos cliques consiste em determinar se um grafo contém um clique maior que um dado tamanho.

Características dos problemas NP

Os problemas acima são representantes razoáveis da classe de problemas NP. Eles têm as seguintes características:

- Cada problema é solúvel, e existe uma abordagem relativamente simples para encontrar a solução (embora essa abordagem possa consumir muito tempo). Para cada um deles, nós podemos simplesmente enumerar todas as possibilidades, todas as maneiras de atribuir valores lógicos para n variáveis, ou todos os subconjuntos do conjunto de inteiros. Se existe uma solução, ela irá aparecer na enumeração; se não, isto ficará patente da mesma forma.
- Há 2^n casos a considerar na abordagem da enumeração. Cada possibilidade pode ser testada em um tempo relativamente pequeno, e o tempo para testar todas as possibilidades e responder "sim" ou "não" é proporcional a 2^n.
- Os problemas são aparentemente não relacionados e vêm da lógica, teoria dos números, teoria dos grafos, etc.
- Se fosse possível adivinhar com perfeição (ou seja, encontrar uma solução por um processo não-determinístico), resolveríamos cada problema em um pequeno intervalo de tempo. O processo de verificação pode ser feito em tempo polinomial, dada uma boa adivinhação.

Assim, nós definimos NP como o conjunto de todos os problemas que podem ser resolvidos por algoritmos não-determinísticos em tempo polinomial. Outra definição, equivalente, diz que NP é o conjunto de todos os problemas cuja solução pode ser verificada em tempo polinomial.

Problemas NP-completo

Existe uma subclasse dos problemas NP, denominada NP-completo, que é muito importante, porque representa os problemas NP mais "difíceis", no sentido de que se um problema NP-completo tiver uma solução polinomial determinística, todos os demais problemas NP também o terão.

Para entender a classe NP-completo, são necessários os conceitos de *transformação polinomial* (ver adiante) e *problema de decisão*. Problemas de decisão são aqueles que fornecem resultados "sim/não". Por exemplo, SAT é um problema de decisão: dada uma fórmula lógica, um algoritmo que resolva o problema deve retornar "sim" se a fórmula for satisfazível ou "não" se não for. Todos os problemas em NP-completo são problemas de decisão.

Como dizer se um problema é NP-completo?

Suponha que você está trabalhando em um problema Π, e não seja capaz de encontrar uma solução que rode em tempo polinomial. Além disso, você não consegue provar que o problema precisa de tempo exponencial. Isto significa que Π é NP-completo? Infelizmente, as coisas não são tão simples assim. Pode acontecer que exista uma solução simples que você ainda não tenha encontrado, ou talvez o problema seja intratável e você não tenha encontrado a prova.

Para provar que um problema é NP-completo, precisamos introduzir o conceito de *transformação polinomial* de um problema de decisão. Intuitivamente, uma transformação polinomial de um problema Π_1 para um problema Π_2 indica que Π_1 pode ser rescrito como um problema Π_2 diferente, e a resposta para Π_1 será *sim* se, e somente se, a resposta para Π_2 for *sim*. Se isto for verdade, dizemos que Π_1 se *reduz* a Π_2, escrevemos $\Pi_1 \propto \Pi_2$. Esta definição é bastante confusa, então, algumas vezes, é mais fácil pensar em transformações em diferentes termos. Informalmente, se $\Pi_1 \propto \Pi_2$, podemos pensar em uma solução para Π_1 sendo obtida a partir de uma solução para Π_2 em tempo polinomial. O código abaixo dá uma idéia do que está envolvido se $\Pi_1 \propto \Pi_2$:

```
Convert_To_P2 p1 = ... /* Toma uma instância de P1 e a converte
                          para uma instância de P2 em tempo
                          polinomial */

Solve_P2 p2 = ... /* Resolve o problema P2 */

Solve_P1 p1 = Solve_P2(Convert_To_P2 p1);
```

A transformação polinomial é transitiva: se $\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_3$, então $\Pi_1 \propto \Pi_3$.

Dada a definição de transformação acima, os teoremas a seguir não deveriam ser muito surpreendentes (onde P é a classe dos problemas polinomiais):

Se $\Pi_1 \propto \Pi_2$ então $\Pi_2 \in \mathrm{P} \rightarrow \Pi_1 \in \mathrm{P}$.
Se $\Pi_1 \propto \Pi_2$ então $\Pi_2 \notin \mathrm{P} \rightarrow \Pi_1 \notin \mathrm{P}$.

Isto leva à seguinte definição: um problema de decisão Π é NP-completo se, e somente se, $\Pi \in$ NP e, para todo $\Pi' \in$ NP, $\Pi' \propto \Pi$.

Os teoremas acima e a transitividade da transformação polinomial sugerem uma técnica para provar que um dado problema Π é NP-completo. Para provar que $\Pi \in$ NP:

1. Encontre um problema NP-completo conhecido Π_{NP}.
2. Encontre uma transformação tal que $\Pi_{\mathrm{NP}} \propto \Pi$.
3. Prove que a transformação é polinomial.

Observando a técnica de prova esboçada acima, é óbvio que deve existir um "primeiro" problema NP-completo para iniciar o processo. Tal problema existe: é o problema da Satisfabilidade.

O significado da NP-completude

Um importante pesquisador de nome Cook (1971) mostrou que o problema da satisfabilidade (SAT) é NP-completo, significando que ele pode representar toda uma classe de problemas NP. Sua importante conclusão foi que se existe um algoritmo polinomial e determinístico para SAT, então há um algoritmo polinomial e determinístico para cada problema no conjunto NP. Como P $\subseteq$ NP (se existe um algoritmo polinomial determinístico para um problema, então também existe um não-determinístico), se SAT possuir solução polinomial determinística, então P = NP. Portanto, se SAT puder ser resolvido em tempo polinomial, então tudo mais em NP pode ser resolvido em tempo polinomial, e se algum problema em NP for intratável, SAT também deve ser. Então SAT representa o "núcleo" da classe de problemas NP-completo. Em termos mais formais, para todo problema $\Pi \in$ NP, $\Pi \propto$ SAT.

A questão P = NP (ou P $\neq$ NP) é um dos problemas em aberto mais importantes em Teoria da Computação. A maioria dos pesquisadores acredita que P $\neq$ NP, mas ninguém conseguiu ainda obter uma prova.

Outro pesquisador de nome Karp (1972) estendeu o trabalho de Cook ao identificar diversos outros problemas, todos com a mesma propriedade de que se qualquer um deles puder ser resolvido através de um algoritmo polinomial determinístico, então todos o serão. O problema da mochila e o problema dos cliques foram dois dos problemas identificados por Karp.

Os resultados de Cook e Karp incluem o inverso: se para qualquer desses problemas (ou qualquer problema NP-completo) puder ser mostrado que não existe um algoritmo polinomial determinístico que o solucione, então também não existe para nenhum outro.

É importante distinguir um problema de uma instância de um problema. Uma instância é um caso específico: uma fórmula lógica, um conjunto de inteiros particular, um grafo. Certos grafos simples, ou certas fórmulas simples, podem ter soluções que são rápidas e fáceis de identificar. Um problema, entretanto, é mais geral. É a descrição de todas as instâncias de um determinado tipo. Por exemplo, a descrição formal de SAT dada acima é a descrição de um problema, porque especifica como cada instância particular do problema deve ser. Resolver um problema significa encontrar um algoritmo que resolva todas as suas instâncias.

1.9 Notas históricas

A origem da notação O-grande é atribuída a Paul Bachmann (1837–1920) em seu texto de teoria dos números escrito em 1892. O símbolo O é, por vezes, chamado de símbolo de Landau como referência a Edmund Landau (1877–1938), que usou essa notação em seu trabalho. Hartmanis e Stearns foram os pioneiros do estudo do tempo de execução dos programas e da complexidade dos problemas [Hartmanis e Sterns 1995]). A série de livros de Donald Knuth (mencionados nas referências) estabeleceram o estudo do tempo de execução dos algoritmos como um ingrediente essencial da Ciência da Computação. A classe de problemas polinomiais foi estabelecida em 1964 por Alan Cobham ("The Instrinsic Computational Dificulty of Functions", em *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*) e, independentemente, por Jack Edmonds em 1965 ("Paths, Trees and Flowers" em *Canadian Journal of Mathematics*, 17).

2 O caso recursivo

Tópicos principais:

- Relações de recorrência
- Solução de relações de recorrência
- As torres de Hanoi
- Análise de subprogramas recursivos

Até agora, temos analisado algoritmos não recursivos, observando como a notação O-grande pode ser usada para caracterizar a taxa de crescimento do tempo de execução para vários algoritmos. Esta análise torna-se um pouco mais complicada quando voltamos nossa atenção para os algoritmos recursivos. Assim, precisamos aumentar nosso repertório de ferramentas para nos auxiliar em tais análises. Uma dessas ferramentas é um entendimento sobre as relações de recorrência, que são discutidas em seguida.

2.1 Relações de recorrência

Uma definição recursiva ou indutiva de uma sequência tem as seguintes partes:

1. *Caso base*: a condição base ou inicial define o primeiro elemento (ou poucos primeiros) da sequência.
2. *Caso indutivo* (ou *recursivo*): um passo indutivo mostra como termos da sequência são definidos a partir dos termos anteriores.

O passo indutivo na definição recursiva pode ser expresso como uma *relação de recorrência*, mostrando como termos anteriores estão relacionados com os posteriores. A caixa abaixo traz uma definição mais formal.

Uma *relação de recorrência* para uma sequência $a_1, a_2, a_3, \ldots$ é uma fórmula que relaciona cada termo a_k com alguns termos anteriores $a_{k-1}, a_{k-2}, \ldots, a_{k-i}$, onde i é um inteiro fixo e k é qualquer inteiro maior ou igual a i. As condições iniciais para essa relação de recorrência especificam os valores de $a_1, a_2, \ldots, a_{i-1}$.

Uma relação de recorrência é uma forma de definir uma sequência em matemática (outras maneiras incluem a enumeração ou a apresentação de uma fórmula para expressar a sequência). Suponha que você tem uma sequência enumerada que satisfaça uma definição recursiva (ou relação de recorrência) dada. Com frequência, é muito útil dispor de uma fórmula para os elementos da sequência (além da relação de recorrência), especialmente se for necessário determinar um grande número de membros. Uma fórmula explícita desse tipo é chamada de *solução* da relação de recorrência. Se um membro da sequência pode ser calculado por um número fixo de operações elementares, dizemos que temos uma *fórmula fechada*. A solução das relações de recorrência é a chave para a análise de subprogramas recursivos.

Exemplo 1

1. Faça uma lista de todas as cadeias de bits de comprimento 0, 1, 2 e 3 que não contêm o padrão de bits 11. Quantas dessas cadeias existem?

2. Encontre o número de cadeias de tamanho dez que não contêm o padrão 11.

Solução:

1. Uma maneira de resolver este problema é através de enumeração:

Tamanho	Enumeração	Quantidade
0	Cadeia vazia	1
1	0,1	2
2	00,01,10	3
3	000,001,010,100,101	5

2. Enumerar todas as cadeias de tamanho dez não seria prático: há 1024 cadeias possíveis. Deve haver uma forma melhor.

   Suponha que o número de cadeias de bits de comprimento menor que um inteiro k que não contenham o padrão 11 seja conhecido (isto é verdade, por exemplo, para os comprimentos 0, 1, 2 e 3). Para encontrar este número para cadeias de tamanho k, podemos tentar expressar cadeias com essa propriedade em termos de cadeias menores com a mesma propriedade. Isto é chamado *raciocínio recursivo*.

   Considere o conjunto de todas as cadeias de tamanho k que não contêm 11. Toda cadeia no conjunto começa com 0 ou 1. Se a cadeia começa com 0, então os $k-1$ caracteres restantes podem formar qualquer sequência de 0's e 1's em que o padrão 11 não apareça. Se a cadeia começa com 1, o segundo caracter deve ser 0; os $k-2$ caracteres restantes podem formar qualquer sequência de 0's e 1's que não contenha o padrão 11. Então, o conjunto de todas as cadeias de comprimento k que não contêm 11 pode ser particionado da seguinte forma:

   conjunto de todas as cadeias de bits da forma $0\underbrace{--\cdots--}_{k-1 \text{ bits sem } 11}$ conjunto de todas as cadeias de bits da forma $10\underbrace{--\cdots--}_{k-2 \text{ bits sem } 11}$

   O número de elementos no conjunto completo é igual à soma dos elementos nas duas partições:

   $$\begin{array}{lcll} \text{Número de cadeias} & & \text{Número de cadeias} & \text{Número de cadeias} \\ \text{de tamanho } k \text{ que} & = & \text{de tamanho } k-1 & + \text{ de tamanho } k-2 \\ \text{não contêm 11} & & \text{que não contêm 11} & \text{que não contêm 11} \end{array}$$

   Ou, na forma de uma relação de recorrência:

   - $s_0 = 1$, $s_1 = 2$.
   - $s_k = s_{k-1} + s_{k-2}$.

   Se fizermos o cálculo para s_{10}, obteremos 144. □

2.2 Na vida, como na morte... traga a relação de recorrência

Muitos problemas em biologia, administração e ciências sociais levam a sequências que satisfazem relações de recorrência (lembram-se dos coelhos de Fibonacci?). A seguir estão as famosas *Equações de Combate de Lancaster*.

Dois exércitos estão combatendo. Cada exército conta o número de soldados em condição de combater ao final de cada dia. Sejam a_0 e b_0 os números de soldados no primeiro e no segundo exércitos, respectivamente, antes do combate começar, e sejam a_n e b_n, respectivamente, os números de soldados no primeiro e no segundo exércitos ao final do n-ésimo dia. Então $a_{n-1} - a_n$ representa o número de soldados perdido pelo primeiro

exército no n-ésimo dia; da mesma forma, $b_{n-1} - b_n$ é o número de soldados perdido pelo segundo exército no n-ésimo dia.

Suponha que o decréscimo no número de soldados em cada exército seja proporcional ao número de soldados no outro exército no início de cada dia. Então, existem constantes A e B tais que $a_{n-1} - a_n = A \cdot b_{n-1}$ e $b_{n-1} - b_n = B \cdot a_{n-1}$. Essas constantes medem a efetividade dos armamentos dos diferentes exércitos, e podem ser calculadas usando relações de recorrência para a e b.

2.3 Solução de relações de recorrência com substituições repetidas

Um método para resolver relações de recorrência é chamado de *substituições repetidas*. Ele funciona como mágica. Mais seriamente, ele funciona desta forma: você toma a relação de recorrência e a usa para enumerar os elementos da sequência de trás para frente a partir de k. Para encontrar o elemento anterior a k, nós substituímos k por $k - 1$ na relação de recorrência.

Exemplo 2

Seja a seguinte relação de recorrência:

$$\begin{aligned} a_0 &= 1 \\ a_k &= a_{k-1} + 2 \end{aligned}$$

Problema: qual é a fórmula desta relação?

Nós começamos substituindo $k - 1$ na definição recursiva acima para obter:

$$a_{k-1} = a_{k-2} + 2$$

Então, substituímos esse valor de a_{k-1} na equação recorrente original. Fazemos isto porque queremos encontrar uma equação para a_k, então, se enumeramos a partir de k, porém substituindo essas equações na fórmula para a_k, talvez encontremos um padrão. O que estamos procurando é uma série de representações diferentes da fórmula de a_k baseadas nos elementos prévios da sequência. Então:

$$a_k = a_{k-1} + 2 = (a_{k-2} + 2) + 2 = a_{k-2} + 4$$

Em seguida, substituímos $k - 2$ na equação original para obter $a_{k-2} = a_{k-3} + 2$, e levamos esse resultado para a_{k-2} na expressão prévia, e assim sucessivamente:

$$\begin{aligned} a_k &= a_{k-2} + 4 = (a_{k-3} + 2) + 4 = a_{k-3} + 6 \\ a_k &= a_{k-3} + 6 = (a_{k-4} + 2) + 6 = a_{k-4} + 8 \\ a_k &= a_{k-4} + 8 = \cdots \end{aligned}$$

O padrão que emerge é óbvio: $a_k = a_{k-i} + 2i$. Entretanto, precisamos nos livrar de algumas dessas variáveis. Especificamente, precisamos nos livrar de a_{k-i} no lado direito da equação para obter uma forma fechada para a fórmula. Em outras palavras, não será necessário avaliar elementos anteriores da sequência para chegar ao que queremos. Então, fazemos $i = k$, obtendo:

$$\begin{aligned} a_k &= a_0 + 2k \\ a_k &= 1 + 2k \end{aligned}$$

Para confirmar esta substituição ($i = k$), precisaremos de uma prova indutiva para mostrar que a relação de recorrência representa a mesma sequência que fórmula fechada. Esta prova é feita a seguir.

Prova: Seja $P(n)$ a seguinte proposição: se $a_0, a_1, a_2, \ldots$ é a sequência definida por:

$$\begin{aligned} a_0 &= 1 \\ a_k &= a_{k-1} + 2 \text{ para } k \geq 1 \end{aligned}$$

então $a_n = 1 + 2n$ para todo $n \geq 1$.

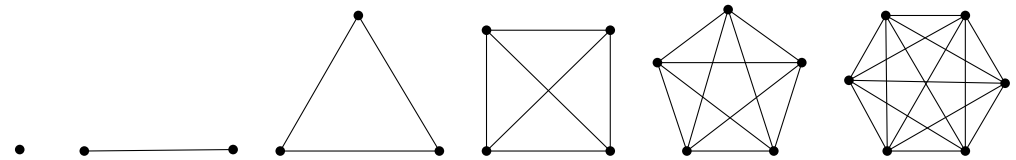


Figura 3: Grafos completos

Caso base: prova-se que $P(1)$ é verdadeiro. Usando a definição da relação de recorrência temos $a_1 = a_0 + 2 = 1 + 2 = 3$. Agora, mostramos que o mesmo resultado é obtido usando a fórmula $a_n = 1 + 2n$ para $n = 1$: $a_1 = 1 + 2 \cdot 1 = 1 + 2 = 3$. Então, o caso base está provado.

Passo indutivo: a hipótese indutiva é assumir $P(k)$: $a_k = 1 + 2k$. Precisamos então mostrar $P(k+1)$: $a_{k+1} = 1 + 2(k+1)$. Tem-se:

$$\begin{array}{lcll} a_{k+1} & = & a_{((k+1)-1)} + 2 & \text{da relação de recorrência (isto é "dado")} \\ a_{k+1} & = & a_k + 2 & \text{álgebra} \\ a_{k+1} & = & 1 + 2k + 2 & \text{substiuição da hipótese indutiva } (a_k = 1 + 2k) \\ a_{k+1} & = & 1 + 2(k+1) & \text{álgebra} \end{array}$$

Assim, chegamos ao resultado desejado. Como $P(k+1)$ é verdadeiro quando $P(k)$ é verdadeiro, e $P(1)$ é verdadeiro, mostramos que $P(n)$ é verdadeiro para todo $n \geq 1$. □

Exemplo 3

Seja K_n um grafo completo de n vértices (isto é, entre dois vértices quaisquer existe uma aresta). A Figura 3 mostra alguns exemplos. Defina uma relação para o número de arestas em K_n e então resolva-a por enumeração e adivinhando uma fórmula.

Solução: por inspeção, podemos ver que $K_1 = 0$, porque esse grafo não tem arestas. Agora, conjecturamos que a adição de um novo vértice ao grafo irá requerer a adição de uma nova aresta conectando cada vértice já existente ao novo vértice. Então, se o grafo tem n vértices, quando acrescentamos o $(n+1)$-ésimo vértice, precisamos adicionar n novas arestas ao grafo para mantê-lo completo. Isto leva à seguinte relação de recorrência:

$$K_{n+1} = K_n + n$$

ou, equivalentemente,

$$K_n = K_{n-1} + (n-1).$$

Os primeiros valores dessa sequência são:

$$\begin{array}{lcl} K_2 & = & K_1 + 1 = 0 + 1 = 1 \\ K_3 & = & K_2 + 2 = 1 + 2 = 3 \\ K_4 & = & K_3 + 3 = 3 + 3 = 6 \\ K_5 & = & K_4 + 4 = 6 + 4 = 10 \end{array}$$

Estes números conferem com os grafos mostrados na Figura 3. Até aqui, tudo bem. Agora, só precisamos generalizar a fórmula para essa sequência. Da relação de recorrência, podemos ver que isto pode ser feito pela fórmula (para $n \geq 1$):

$$K_n = 0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

Nós não apresentaremos uma prova indutiva completa de que esta fórmula está, de fato, correta, mas o leitor mais curioso pode rapidamente fazer uma por si mesmo para tornar-se mais confortável com o uso de indução. □

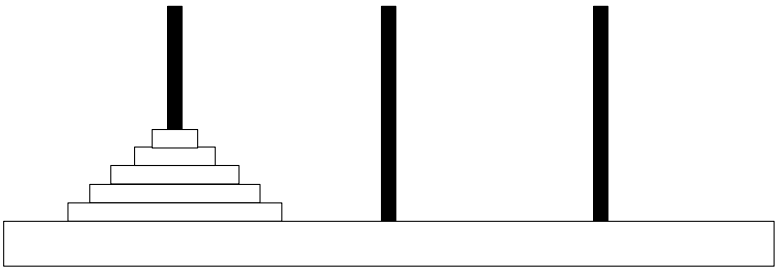


Figura 4: Torre de Hanoi (configuração inicial)

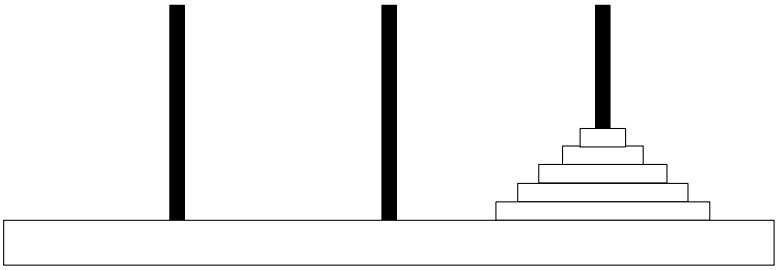


Figura 5: Torre de Hanoi (configuração final)

Exemplo 4

Considere o jogo Torre de Hanoi, que consiste em três pinos e uma série de discos de vários tamanhos que podem ser colocados nos pinos. Os discos começam sendo colocados em ordem de tamanho no primeiro pino. Esta configuração pode ser vista na Figura 4.

O objetivo do jogo é mover todos os discos do primeiro para o terceiro pino, obedecendo às seguintes regras:

1. Apenas um disco pode ser movido por vez (mais especificamente, o disco do topo em um pino).
2. Um disco maior nunca pode ser colocado sobre um menor.

A configuração final do jogo pode ser vista na Figura 5. O pseudocódigo para um algoritmo recursivo que resolve esse problema pode ser visto a seguir, onde k é o número de discos a mover, Beg é o pino a partir do qual os discos devem ser movidos, Aux é um pino que pode ser utilizado como área de "armazenamento auxiliar", e End é o pino para o qual os discos devem ser movidos.

```
1 Tower(k,Beg,Aux,End) {
2   if(k==1)
3     move disk from Beg to End;
4   else {
5     Tower(k-1, Beg,End,Aux);
6     move disk from Beg to End;
7     Tower(k-1, Aux,Beg,End);
8   }
9 }
```

Seja H_n o número de movimentos necessários para resolver a Torre de Hanoi com n discos. Mover $n-1$ discos do pino inicial para o auxiliar requer H_{n-1} movimentos. Então, usa-se mais um movimento para transferir o maior disco do pino inicial para o final. Por fim, os $n-1$ discos no pino auxiliar são movidos para o pino final, precisando de mais H_{n-1} movimentos. Isto leva à seguinte relação de recorrência:

$$H_n = 2H_{n-1} + 1.$$

O caso base para essa relação é $H_0 = 0$, porque se não há discos, então nenhum movimento é necessário para mover "todos" os discos para o pino final. Para determinar uma fórmula fechada onde não existe H_n do lado direito da equação, iremos tentar substituições repetidas:

$$\begin{aligned} H_n &= 2H_{n-1} + 1 \\ &= 2\left(2H_{n-2} + 1\right) + 1 = 2^2 H_{n-2} + 2 + 1 \\ &= 2^2\left(2H_{n-3} + 1\right) + 2 + 1 = 2^3 H_{n-3} + 2^2 + 2 + 1 \end{aligned}$$

O padrão para essa relação de recorrência é:

$$H_n = \left(2^i \cdot H_{n-i}\right) + 2^{i-1} + 2^{i-2} + \cdots + 2^2 + 2^1 + 2^0$$

Pode-se fazer $n = i$ (isto precisa de uma prova indutiva):

$$H_n = \left(2^n \cdot H_0\right) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2^1 + 2^0$$

Substituindo $H_0 = 0$ tem-se:

$$H_n = 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2^1 + 2^0$$

Este é um somatório bem conhecido; por indução, prova-se facilmente que é igual a $2^n - 1$. Portanto, o número de movimentos necessários para n discos é $2^n - 1$.

Existe uma lenda antiga com este jogo que diz que há uma torre em Hanoi onde monges estão transferindo 64 discos de ouro entre os pinos de acordo com as regras discutidas. Eles levam um segundo para mover um disco. A lenda diz que o mundo terminará quando eles finalizarem o jogo. Um cálculo rápido revela quanto tempo isto levará:

$$2^{64} - 1 = 18.446.744.073.709.551.615 \text{ segundos.}$$

Portanto, os monges estarão ocupados por cerca de 500 bilhões de anos solucionando o jogo. Sorte nossa, mas azar dos monges que provavelmente não estão se divertindo muito movendo discos de pino para pino. □

Exemplo 5

Encontre uma solução fechada para as relações de recorrência a seguir. Por brevidade, as provas indutivas são omitidas.

1. Relação de recorrência:

$$\begin{aligned} a_0 &= 2 \\ a_n &= 3a_{n-1} \end{aligned}$$

   Solução:

$$\begin{aligned} a_n &= 3a_{n-1} \\ a_n &= 3\left(3a_{n-2}\right) = 3^2 a_{n-2} \\ a_n &= 3^2\left(3a_{n-3}\right) = 3^3 a_{n-3} \\ &\cdots \\ a_n &= 3^{i-1}\left(3a_{n-i}\right) = 3^i a_{n-i} \end{aligned}$$

   Fazendo $i = n$ tem-se:

$$\begin{aligned} a_n &= 3^n a_0 \\ a_n &= 2 \cdot 3^n \end{aligned}$$

2. Relação de recorrência:

$$\begin{aligned} a_0 &= 1 \\ a_n &= 2a_{n-1} - 1 \end{aligned}$$

Solução:

$$\begin{aligned} a_n &= 2\left(2a_{n-2}-1\right)-1 = 2^2 a_{n-2} - 2 - 1 \\ a_n &= 2^2\left(2a_{n-3}-1\right)-2-1 = 2^3 a_{n-3} - 2^2 - 2 - 1 \\ &\cdots \\ a_n &= 2^i a_{n-i} - 2^{i-1} - 2^{i-2} - \cdots - 2^0 \end{aligned}$$

Fazendo $i = n$ tem-se:

$$\begin{aligned} a_n &= 2^n a_0 - 2^{n-1} - 2^{n-2} - \cdots - 2^0 = 2^n - 2^{n-1} - 2^{n-2} - \cdots - 2^0 \\ a_n &= 2^n - (2^n - 1) \quad \text{(Notar que } 2^{n-1} + 2^{n-2} + \cdots + 1 = 2^n - 1\text{)} \\ a_n &= 1 \end{aligned}$$

Note que, embora a relação de recorrência aparentemente seja muito semelhante à do problema da Torre de Hanoi, os resultados são completamente diferentes.

3. Relação de recorrência:

$$\begin{aligned} a_0 &= 5 \\ a_n &= n a_{n-1} \end{aligned}$$

Solução:

$$\begin{aligned} a_n &= n\left(n-1\right)a_{n-2} \\ a_n &= n\left(n-1\right)\left(n-2\right)a_{n-3} \\ &\cdots \\ a_n &= n\left(n-1\right)\left(n-2\right)\cdots\left(n-i+1\right)a_{n-i} \end{aligned}$$

Fazendo $i = n$ tem-se:

$$\begin{aligned} a_n &= n\left(n-1\right)\left(n-2\right)\cdots\left(1\right)a_0 = a_0 n! \\ a_n &= 5n!\,\square \end{aligned}$$

2.4 Análise de programas recursivos

A solução de relações de recorrência é a chave para a análise de programas recursivos. Nós precisamos de uma fórmula que represente o número de chamadas recursivas e o trabalho feito em cada chamada, enquanto o programa converge para o caso base. Essa fórmula é então usada para determinar um limite superior para o desempenho de pior caso do programa. Em termos de análise de algoritmos, uma relação de recorrência pode ser vista da seguinte forma:

Uma *relação de recorrência* expressa o tempo de execução de um algoritmo recursivo. Isto inclui o número de chamadas recursivas geradas em cada nível de recursão, o quanto do problema é resolvido por cada chamada, e quanto trabalho é feito em cada nível.

Suponha que tenhamos um tempo de execução desconhecido para a função F, definido em termos de um parâmetro n de F (uma vez que esse parâmetro controla o número de chamadas recursivas). Nós chamamos esse tempo de $T_F\left(n\right)$. O valor de $T_F\left(n\right)$ é estabelecido por uma definição indutiva com base no tamanho do argumento n. Existem dois casos, examinados a seguir:

1. n é suficientemente pequeno para que nenhuma chamada recursiva seja feita. Este caso corresponde ao *caso base* na definição indutiva de $T_F\left(n\right)$.

2. n é suficientemente grande para que seja necessária uma chamada recursiva. Entretanto, nós assumimos que qualquer chamada recursiva feita por F utilizará parâmetros com valores menores. Este caso corresponde ao passo indutivo na definição de $T_F\left(n\right)$.

A relação de recorrência para $T_F\left(n\right)$ é derivada a partir do exame do código de F e da definição dos tempos de execução para os dois casos acima. A relação de recorrência é então resolvida para obter uma expressão O-grande para F.

Exemplo 6

Considere a função a seguir para o cálculo de fatoriais:

```
1 int factorial(int n) {
2   if(n <= 1)
3     return 1;
4   else
5     return (n * factorial(n-1));
6 }
```

Nós representaremos o tempo de execução desconhecido desta função por $T(n)$, onde n é o tamanho do parâmetro. Para o caso base da definição indutiva de $T(n)$, tomaremos n igual a 1, uma vez que nenhuma chamada recursiva é feita nesse caso. Quando $n = 1$, apenas as linhas 2 e 3 são executadas, e cada uma tem um tempo de execução constante (isto é, $O(1)$), que será designado por um tempo a.

Quando $n > 1$, a condição da linha 2 é falsa, então as linhas 2 e 5 são executadas. A linha 2 leva um tempo constante b e a linha 5 leva $T(n-1)$, uma vez que está sendo feita uma chamada recursiva. Se o tempo de execução de factorial () com parâmetro de valor n é $T(n)$, então esse tempo para uma chamada de factorial () com parâmetro $n-1$ é $T(n-1)$. Isto leva à seguinte relação de recorrência:

$$\begin{aligned} T(1) &= a \text{ (caso base)} \\ T(n) &= b + T(n-1) \text{ para } n > 1 \text{ (passo indutivo).} \end{aligned}$$

Esta relação deve ser resolvida para obter uma fórmula fechada. Podemos enumerar alguns casos para tentar obter um padrão:

$$\begin{aligned} T(1) &= a \\ T(2) &= b + T(1) = b + a \\ T(3) &= b + T(2) = b + (b + a) = a + 2b \\ T(4) &= b + T(3) = b + (a + 2b) = a + 3b \\ &\cdots \\ T(n) &= a + (n-1)\, b \text{ para todo } n \geq 1. \end{aligned}$$

Para completar, precisamos de uma prova por indução para mostrar que, de fato, a fórmula para o tempo de execução $T(n) = a + (n-1)\, b$ é verdadeira para todo $n \geq 1$.

Prova: Seja $P(n)$ a seguinte proposição: o tempo de execução de factorial () é $T(n) = a + (n-1)\, b$. Recordando, a relação de de recorrência é T(1) = a e $T(n) = b + T(n-1)$ para $n > 1$.

Caso base: provar que $P(1)$ é verdadeiro. Tem-se $T(1) = a + (1-1)\, b = a + 0 \cdot b = a$. Esta equação se verifica porque o caso base de nossa definição indutiva afirma que $T(1) = a$.

Caso indutivo: a hipótese indutiva assume $P(k)$: $T(k) = a + (k-1)\, b$, e deve-se provar $P(k+1)$: $T(k+1) = a + kb$. Sabemos, da definição da relação de recorrência, que $T(k+1) = b + T(k)$. Usamos a hipótese indutiva para substituir $T(k)$:

$$\begin{aligned} T(k+1) &= b + T(k) \\ &= b + a + (k-1)\, b \\ &= b + a + kb - b \\ &= a + kb. \end{aligned}$$

Então $P(k+1)$ se verifica quando se assume $P(k)$, e $P(1)$ se verifica, portanto $P(n)$ é verdadeiro para todo $n \geq 1$.

Agora que conhecemos uma fórmula fechada para a relação de recorrência, usamos essa fórmula para determinar a complexidade. $T(n) = a + (n-1)\, b = a + bn - b$ tem uma complexidade $O(n)$. Isto faz sentido: para calcular $n!$, fazemos n chamadas de factorial (), e cada uma requer tempo $O(1)$. □

Exemplo 7

Considere o seguinte algoritmo recursivo de ordenação por seleção:

```
1  void SelectionSort(int A[], int i, int n) {
2    int j, small, temp;
3
4    if(i < n) {
5      small = i;
6      for(j = i+1; j <= n; j++) {
7        if(A[j] < A[small])
8          small = j;
9      }
10     temp = A[small];
11     A[small] = A[i];
12     A[i] = temp;
13     SelectionSort(A, i+1, n);
14   }
15 }
```

O parâmetro n indica o tamanho do vetor, e o parâmetro i diz quanto do vetor falta ordenar, mais especificamente A[i..n]. Então, uma chamada SelectionSort(A,1,n) irá ordenar todo o vetor com chamadas recursivas.

Nós agora desenvolvemos uma relação de recorrência. Note que o tamanho do vetor a ser ordenado em cada chamada é $n-i+1$. Iremos representar este valor como m, o tamanho do vetor durante uma chamada recursiva específica. Existe um caso base ($m=1$). Neste caso, apenas a linha 4 é executada, levando uma quantidade de tempo constante representada por a. Note que $m=0$ não é um caso base porque a recursão converge para uma lista com um único elemento: quando $i=n$, $m=1$.

O caso indutivo é para $m>1$: isto corresponde à execução das chamadas recursivas. Linhas 5, 10, 11 e 12 precisam de tempo constante. O laço **for** das linhas 6, 7 e 8 será executado m vezes (onde $m=n-i+1$). Então a chamada recursiva para essa função será dominada pelo tempo necessário para executar o laço m vezes, que representaremos por $O(m)$. O tempo para a chamada recursiva da linha 13 é $T(m-1)$. Então, a definição indutiva doSelectionSort() recursivo é:

$$\begin{aligned} T(1) &= a \\ T(m) &= T(m-1)+O(m) \end{aligned}$$

Para resolver esta relação de recorrência, nós primeiramente nos livramos da expressão O-grande fazendo uma substituição baseada em sua definição: $f(n)=O(g(n))$ se $f(n)\leq C\cdot g(n)$. Então, substituímos $O(m)$ por Cm:

$$\begin{aligned} T(1) &= a \\ T(m) &= T(m-1)+Cm \end{aligned}$$

Agora podemos tentar substituições repetidas ou simplesmente enumerar alguns casos para encontrar um padrão. Vamos usar substituições repetidas:

$$\begin{aligned} T(m) &= T(m-1)+Cm \\ &= T(m-2)+2Cm-C && \text{porque } T(m-1)=T(m-2)+C(m-1) \\ &= T(m-3)+3Cm-3C && \text{porque } T(m-2)=T(m-3)+C(m-2) \\ &= T(m-4)+4Cm-6C && \text{porque } T(m-3)=T(m-4)+C(m-3) \\ &= T(m-5)+5Cm-10C && \text{porque } T(m-4)=T(m-5)+C(m-4) \\ &\cdots \\ &= T(m-j)+jCm-(j(j-1)/2)C \end{aligned}$$

Para obter uma fórmula fechada, fazemos $j=m-1$. Fazemos isto porque nosso caso base é $T(1)$. Se continuássemos com as substituições repetidas até a última substituição possível, pararíamos em $T(1)$ porque

$T(0)$ não é o caso base para o qual a recursão converge. Note que, mais tarde, será preciso fazer uma prova indutiva para mostrar que esta substituição é possível. Mas, no momento:

$$\begin{aligned} T(m) &= T(1) + (m-1)Cm - ((m-1)(m-2)/2)C \\ &= a + m^2C - Cm - \left(m^2C - 3Cm + 2C\right)/2 \\ &= a + \left(2m^2C - 2Cm - m^2C + 3Cm - 2C\right)/2 \\ &= a + \left(m^2C + Cm - 2C\right)/2. \end{aligned}$$

Então, finalmente, temos uma fórmula fechada $T(m) = a + \left(m^2C + Cm - 2C\right)/2$. A complexidade dessa fórmula é $O\left(m^2\right)$, que é a mesma complexidade da ordenação por seleção iterativa, então não houve nenhuma economia de tempo com a implementação recursiva. A relação de recorrência acima em geral ocorre com qualquer subprograma que tenha um laço simples de alguma forma, e depois faça uma chamada recursiva (em geral, uma recursão de cauda está envolvida). □

Exemplo 8 MergeSort é um algoritmo de ordenação eficiente que usa uma estratégia "dividir e conquistar". Ou seja, o problema de ordenar uma lista é reduzido ao problema de ordenar duas listas menores, e então intercalar essas listas já classificadas. Suponha que tenhamos um vetor ordenado A com r elementos e outro vetor ordenado B com s elementos. Intercalação (*merging*) é uma operação que combina os elementos de A com os elementos de B em uma vetor maior C com $(r + s)$ elementos. MergeSort é baseado neste algoritmo de intercalação. Nós usamos uma função recursiva que divide uma lista em duas metades, e então classificamos essas duas metades e as intercalamos. A questão é: como as duas metades são ordenadas? Nós as ordenamos usando o MergeSort recursivamente. O algoritmo em pseudocódigo é mostrado abaixo:

```
1 Algoritmo: MergeSort (L)
2 se comprimento de L > 1 então
3 |   Divida a lista L em primeira e segunda metades
4 |   MergeSort (primeira metade)
5 |   MergeSort (segunda metade)
6 |   Intercale a primeira e a segunda metades na lista ordenada
```

Nota: nós assumimos que os procedimentos de divisão e intercalação da lista executam em $O(n)$.[8] As chamadas recursivas continuam dividindo a lista em duas metades até que cada metade tenha apenas um único elemento; obviamente, uma lista com um elemento está ordenada. O algoritmo então intercala essas metades menores em metades maiores classificadas até que tenhamos uma única grande lista ordenada.

Por exemplo, suponha que a lista contem os seguintes inteiros:

7	9	2	4	3	6	1	8

Em primeiro lugar, dividimos a lista em duas metades (iremos nos referir a esta chamada como MergeSort #0 = chamada original):

7	9	4	2

3	6	1	8

Nós então chamamos MergeSort recursivamente na primeira metade, o que novamente divide a lista em duas metades (MergeSort #1):

7	9

4	2

Chamamos MergeSort recursivamente mais uma vez para a primeira metade, o que divide a lista em duas partes novamente (MergeSort #2):

7

9

[8] Se a lista for implementada por um vetor, o procedimento de divisão se resume ao cálculo do índice central e, portanto, é $O(1)$ (N.T.).

Quando chamamos MergeSort a próxima vez, a condição do *se* da linha 2 será falsa, e retornamos um nível de recursão para MergeSort #2. Agora executamos a próxima linha em MergeSort #2 (linha 5), que executa na lista contendo apenas 9. Outra vez, a condição é falsa, e retornamos para MergeSort #2. Executamos então a próxima linha (linha 6), que intercala as listas contendo 7 e 9, produzindo:

7	9

A chamada MergeSort #2 está completa. Agora retornamos um nível de recursão para MergeSort #1 e encontramos a lista contendo 4 e 2. Executamos a próxima linha de MergeSort #1 (linha 5), chamando MergeSort para essa lista, e obtemos listas separadas com 4 e 2:

4

2

Prosseguimos intercalando essas duas listas ordenadas em uma contendo 2 e 4, da mesma forma que fizemos com 7 e 9, produzindo:

2	4

Uma vez que está trabalho está terminado, retornamos a MergeSort #1 e executamos a intercalação das duas listas, cada uma com dois elementos. Isto produz a lista ordenada:

2	4	7	9

Agora retornamos para a chamada MergeSort #0 (a chamada original), e executamos a linha seguinte (linha 5). Isto provoca um novo ciclo de recursões semelhante ao primeiro. Quando esse ciclo estiver terminado, teremos uma segunda lista ordenada:

1	3	6	8

A última linha do MergeSort (linha 6) intercala essas duas listas classificadas de quatro elementos em uma única lista ordenada:

1	2	3	4	6	7	8	9

Para analisar a complexidade do MergeSort, precisamos definir uma relação de recorrência. O caso base corresponde à lista de 1 elemento, e apenas a linha 2 da função é executada. Portanto, o caso base é de tempo constante $O(1)$.

Se o teste da linha 2 falha, precisamos executar as linhas 3–6. O tempo gasto nessa função quando o comprimento da lista é maior que 1 é a soma dos seguintes:

1. $O(1)$ para o teste da linha 2.
2. $O(n)$ para a chamada da função de divisão da linha 3.
3. $T(n/2)$ para a chamada recursiva da linha 4.
4. $T(n/2)$ para a chamada recursiva da linha 5.
5. $O(n)$ para a chamada da função de intercalação da linha 6.

Aplicando a regra da soma, tem-se $2T(n/2) + O(n)$. Substituindo constantes no lugar da notação O-grande, chega-se a:

$$\begin{aligned} T(1) &= a \\ T(n) &= 2T(n/2) + bn \end{aligned}$$

Para resolver esta relação de recorrência, podemos enumerar alguns poucos valores. Consideraremos apenas n's que são potências de 2, para que as divisões resultem sempre em valores iguais:

$$\begin{array}{lllllll} T(2) & = & 2T(1) + 2b & & & = & 2a + 2b \\ T(4) & = & 2T(2) + 4b & = & 2(2a + 2b) + 4b & = & 4a + 8b \\ T(8) & = & 2T(4) + 8b & = & 2(4a + 8b) + 8b & = & 8a + 24b \\ T(16) & = & 2T(8) + 16b & = & 2(8a + 24b) + 16b & = & 16a + 64b \end{array}$$

Há um padrão óbvio emergindo, mas ele não é tão fácil de representar quanto os anteriores. Note as seguintes relações:

Valor de n:	2	4	8	16
Coeficiente de b:	2	8	24	64
Razão:	1	2	3	4

Então, aparentemente, o coeficiente de b é n vezes outro fator que cresce 1 sempre que n dobra. A razão é $\log_2 n$ porque $\log_2 2 = 1$, $\log_2 4 = 2$, $\log_2 8 = 3$, etc. Nosso "palpite" para a solução dessa relação de recorrência é $T(n) = an + bn \log_2 n$.

Nós poderíamos ter utilizado substituições repetidas e teríamos encontrado a seguinte fórmula:

$$T(n) = 2^i T(n/2) + ibn.$$

Se fizermos $i = \log_2 n$ chegamos a:

$$n \cdot T(1) + bn \log_2 n = an + bn \log_2 n \quad (\text{porque } 2^{\log_2 n} = n).$$

Obviamente, precisamos fazer uma prova indutiva para mostrar que essa fórmula é, de fato, válida para todo $n \geq 1$, mas isto será deixado como exercício para o leitor. Finalmente, note que a complexidade de $an + bn \log_2 n$ é $O(n \log n)$, que é consideravelmente mais rápido que SelectionSort. Você irá ver esta relação de recorrência com frequência em ordenações com "divisão e conquista". □

2.5 Um teorema em casa ou no escritório é uma boa coisa...

Felizmente, existe um teorema bem prático que é frequentemente útil na análise da complexidade de algoritmos de "divisão e conquista" gerais[9].

Teorema Mestre: seja f uma função crescente que satisfaz a seguinte relação de recorrência:

$$f(n) = af(n/b) + cn^d$$

sempre que $n = b^k$, onde k é um inteiro positivo, $a \geq 1$, b é um inteiro maior que 1, c é um real positivo e d é um real não negativo. Então:

$$f(n) = \begin{cases} O\left(n^d\right) & \text{se } a < b^d \\ O\left(n^d \log n\right) & \text{se } a = b^d \\ O\left(n^{\log_b a}\right) & \text{se } a > b^d \end{cases}$$

Exemplo 9

Problema: use o Teorema Mestre para encontrar o tempo de execução O-grande do MergeSort (exemplo anterior).

Solução: No exemplo anterior, expressamos a relação de recorrência do MergeSort como:

$$T(n) = 2T(n/2) + xn$$

onde substituímos a constante positiva b na relação original por x, para evitar confusões com os nomes das variáveis a seguir.

Para aplicar o Teorema Mestre, precisamos mostrar que essa relação de recorrência atende a forma funcional $f(n)$ no teorema. Escolhemos: $a = 2$, $b = 2$, $c = x$ e $d = 1$. Com estas constantes, a função $f(n)$ no teorema torna-se:

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

Note que $f(n)$ corresponde à função $T(n)$ acima, tornando o Teorema Mestre aplicável à relação de recorrência. Pela nossa escolha de constantes, sabemos que $a = b^d$, uma vez que $2 = 2^1$. Assim, o Teorema Mestre afirma que a complexidade de nossa relação de recorrência é $O\left(n^d \log n\right)$. Como escolhemos $d = 1$, obtemos o resultado $O(n \log n)$, que é o mesmo valor obtido via substituições repetidas. □

É interessante considerar os três casos do Teorema Mestre, que são baseados na relação entre a e b^d. Abaixo estão alguns exemplos a considerar:

[9] Uma demonstração desse teorema pode ser encontrada em [Cormen, Leiserson e Rivest 1991] (N.T.)

- $a < b^d$: $f(n) = 2f(n/2) + xn^2$. Neste exemplo, $2 < 2^2$, e portanto $f(n) = O\left(n^2\right)$. O termo xn^2 cresce mais rapidamente que $2f(n/2)$, e portanto é dominante.
- $a = b^d$: $f(n) = 2f(n/2) + xn^1$. Neste exemplo, $2 = 2$, e portanto $f(n) = O(n \log n)$. Os dois termos crescem juntos.
- $a > b^d$: $f(n) = 8f(n/2) + xn^1$. Neste exemplo, $8 > 2^2$, e portanto $f(n) = O\left(n^{\log 8}\right) = O\left(n^3\right)$. O termo $8f(n/2)$ cresce mais rapidamente que xn^2 e domina.

Exemplo 10

Problema: use substituições repetidas para encontrar a complexidade temporal da função recurse() abaixo. Verifique seu resultado usando indução.

```
1  /* Assuma que apenas valores não negativos e pares de n são passados */
2  void recurse(int n) {
3    int i, total = 0;
4
5    if(n == 0) return 1;
6    for(i = 0; i < 4; i++) {
7      total += recurse(n-2);
8    }
9    return total;
10 }
```

Solução:

- Quando $n = 0$ (o caso base), esta função só executa a linha 5, fazendo um trabalho $O(1)$. Chamaremos esta quantidade constante de trabalho de a.
- Quando $n \geq 2$ (o caso recursivo), esta função realiza trabalho $O(1)$ ao executar as linhas 5, 6 e 9, que escreveremos como b, e também faz quatro chamadas recursivas com parâmetro de valor $n-2$ no corpo do laço (linha 7).

Nós usamos os pontos acima para executar a análise da relação de recorrência abaixo:

$$\begin{array}{rcll} T_0 & = & a & \text{(a partir do caso base)} \\ T_n & = & 4T_{n-2} + b & \text{(a partir do caso recursivo).} \end{array}$$

Empregando substituições repetidas:

$$\begin{array}{rcl} T_n & = & 4T_{n-2} + b \\ T_n & = & 4\left(4T_{n-4} + b\right) + b = 4^2 T_{n-4} + 4b + b \\ T_n & = & 4^2\left(4T_{n-6} + b\right) + 4b + b = 4^3 T_{n-6} + 4^2 b + 4b + b \\ & \cdots & \\ T_n & = & 4^i T_{n-2i} + 4^{i-1} b + 4^{i-2} b + \cdots + 4^0 b \end{array}$$

Notando que $4^k = 2^{2k}$ vem:

$$T_n = 2^{2i} T_{n-2i} + 2^{2(i-1)} b + 2^{2(i-2)} b + \cdots + 2^0 b$$

Fazendo $i = n/2$:

$$T_n = 2^n T_0 + 2^{n-2} b + 2^{n-4} b + \cdots + 2^0 b = 2^n a + \left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right) b.$$

Precisamos calcular a forma fechada para $\left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right) = (2^n - 1)/3$. Este resultado foi computado da seguinte forma: seja $x = \left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right)$. Então:

$$2^2 x = 4x = 2^2\left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right) = \left(2^n + 2^{n-2} + \cdots + 2^2\right)$$

e, lembrando que $2^2x - x = 3x$, tem-se:

$$\begin{aligned} 3x &= \left(2^n + 2^{n-2} + \cdots + 2^2\right) - \left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right) \\ 3x &= 2^n - 2^0 = 2^n - 1 \\ x &= \left(2^n - 1\right)/3 \end{aligned}$$

Substituindo esta forma fechada para o somatório de volta na equação para T_n vem:

$$\begin{aligned} T_n &= 2^n a + \left(2^{n-2} + 2^{n-4} + \cdots + 2^0\right) b \\ T_n &= 2^n a + \left(\left(2^n - 1\right)/3\right) b. \end{aligned}$$

Agora, verificamos este resultado usando indução. Seja $P(n)$ a proposição $T_n = 2^n a + \left(\left(2^n - 1\right)/3\right) b$.

- Caso base: Mostra-se $P(0)$. Tem-se:

  $$T_0 = 2^0 a + \left(\left(2^0 - 1\right)/3\right) b = a + \left(\left(1 - 1\right)/3\right) b = a.$$

  Esta é a condição inicial.

- Caso indutivo: assume-se a hipótese indutiva $P(n)$: $T_n = 2^n a + \left(\left(2^n - 1\right)/3\right) b$. Deve-se mostrar $P(n+2)$: $T_{n+2} = 2^{n+2} a + \left(\left(2^{n+2} - 1\right)/3\right) b$. Note-se que seguimos considerando apenas valores pares de n. Tem-se:

T_{n+2}	=	$4T_n + b$	da definição da recorrência
T_{n+2}	=	$4\left(2^n a + \left(\left(2^n - 1\right)/3\right) b\right) + b$	substituindo T_n pela hipótese indutiva
T_{n+2}	=	$2^2\left(2^n a + \left(\left(2^n - 1\right)/3\right) b\right) + b$	álgebra: $2^2 = 4$
T_{n+2}	=	$2^{n+2} a + \left(\left(2^{n+2} - 2^2\right)/3\right) b + b$	álgebra: multiplicando 2^2
T_{n+2}	=	$2^{n+2} a + \left(\left(2^{n+2} - 3 - 1\right)/3\right) b + b$	álgebra: $-2^2 = -3 - 1$
T_{n+2}	=	$2^{n+2} a + \left(\left(2^{n+2} - 1\right)/3\right) b - b + b$	álgebra: $(-3/3)\, b = -b$
T_{n+2}	=	$2^{n+2} a + \left(\left(2^{n+2} - 1\right)/3\right) b$	este é o resultado desejado.

  Finalmente, calculamos a complexidade O-grande para a função da seguinte maneira: da relação de recorrência $T_n = 2^n a + \left(\left(2^n - 1\right)/3\right) b$, vemos que o algoritmo tem complexidade temporal $O\left(2^n a\right) + O\left(2^n b\right) = O\left(2^n\right) + O\left(2^n\right) = O\left(2^n\right)$. □

3 Exploração de algoritmos

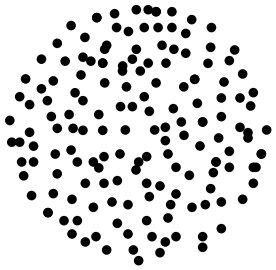
3.1 Caixeiro viajante I: força bruta, abordagem gulosa e heurística

Introdução

Imagine que você tem a tarefa de viajar por dez diferentes cidades em sua área, a cada mês, para entregar alguma coisa importante. É desnecessário dizer que você gostaria de reduzir ao máximo a distância percorrida. Cada cidade está a uma distância diferente da sua, e de cada uma das outras. Como você escolheria uma rota para minimizar a distância percorrida? Você pode reconhecer este problema como o famoso problema do *Caixeiro viajante* (PCV)[10].

PCV é um problema NP-completo muito importante. Vários problemas do mundo real se reduzem a ele, de forma que ele é um dos problemas Np-completo mais importantes. As pessoas precisam resolvê-lo, a despeito do fato de não haver solução polinomial conhecida para todas as instâncias. Nós iremos explorar o PCV neste módulo como uma forma de ilustrar uma coleção de diferentes algoritmos e abordagens para a solução de problemas. Tudo o que existe já foi tentado para PCV!.

[10] O problema do caixeiro viajante exige também que cada cidade seja visitada uma única vez (N.T.)



Se pudéssemos calcular um bilhão de rotas por segundo,
o cálculo total do PCV para 30 cidades iria precisar de mais que
8×10^{15} anos, ou 8 quatrilhões de anos!

Figura 6: Explosão combinatorial

Como resolvê-lo?

Uma forma de resolver este problema (e qualquer outro problema NP-completo) é enumerar todas as possíveis rotas, que são 10! (3.628.800) para 10 cidades, e então escolher a mais curta. Este é um algoritmo de *força bruta*. Ele precisaria de uns poucos segundos em um computador típico para calcular todas as possíveis rotas e distâncias para 10 cidades. E você só precisaria fazer isto uma vez.

Há alguns problemas com algoritmos de força bruta, entretanto. Um é a *explosão combinatorial*: se adicionarmos mais uma cidade, o acréscimo no número de rotas que precisaremos calcular será de de 1000%. Se chegarmos a 20 cidades, não conseguiremos enumerar todas as rotas em nosso tempo de vida (ver Figura 6).

Frequentemente é possível aplicar técnicas mais inteligentes para reduzir a quantidade de enumerações produzidas por um algoritmo de força bruta. Por que não minimizar o trabalho envolvido, se conseguirmos? Algoritmos *gulosos* algumas vezes permitem reduzir o trabalho necessário para um algoritmo de força bruta.

Um algoritmo guloso é aquele em que fazemos a melhor escolha em cada estágio do algoritmo, dada a informação imediatamente disponível. Essas escolhas não levam em conta todos os dados disponíveis em todos os estágios. Algumas vezes a informação imediata é suficiente e uma solução ótima é encontrada; algumas vezes isso não é suficiente, e chegamos a soluções não ótimas.

Há um algoritmo guloso simples para resolver PCV: comece em qualquer cidade. A partir dali, vá para a cidade não visitada mais próxima. Continue até que você tenha visitado todas as cidades, e neste ponto retorne à cidade inicial. Na prática, esta estratégia funciona muito bem, mas ela pode ou não encontrar uma solução ótima.

Antes de prosseguir, procure na Web[11] outros exemplos interessantes. Então volte à nossa discussão sobre algoritmos gulosos.

Como mostrado nos exemplos da máquina de troco e da mochila do *link* acima, algoritmos gulosos são alternativas interessantes aos de força bruta. Como mencionado anteriormente, a dificuldade com algoritmos gulosos é garantir que uma solução ótima será encontrada, enquanto que com a estratégia de força bruta isso sempre acontecerá. Mesmo assim, algoritmos gulosos são úteis, porque são normalmente fáceis de definir e fornecem boas aproximações de soluções ótimas. E, o mais importante, reduzem significativamente o trabalho envolvido.

Qual deles deve ser usado em uma determinada situação? Aqui está uma regra geral: se o tempo de computação não é muito grande, e o cálculo só precisa ser feito uma vez, força bruta funciona bem. Se o tempo de computação é muito grande para força bruta e/ou a solução deve ser calculada com frequência, faz sentido tentar reduzir o número de soluções a processar. Isto pode ser feito com um algoritmo guloso.

Mais alternativas de solução

Outra opção para reduzir o trabalho é empregar *heurísticas*. Uma heurística é um método prático e informal, como uma regra ou princípio geral, para ajudar a resolver um problema. Da mesma forma que com algoritmos gulosos, usamos heurísticas para reduzir o trabalho, em comparação com os algoritmos de força bruta. Também da mesma forma, não há nenhuma garantia de que uma solução ótima será encontrada mas, muito provavelmente, chegaremos a uma solução útil.

[11] `http://www.brpreiss.com/books/opus4/html/page441.html`.

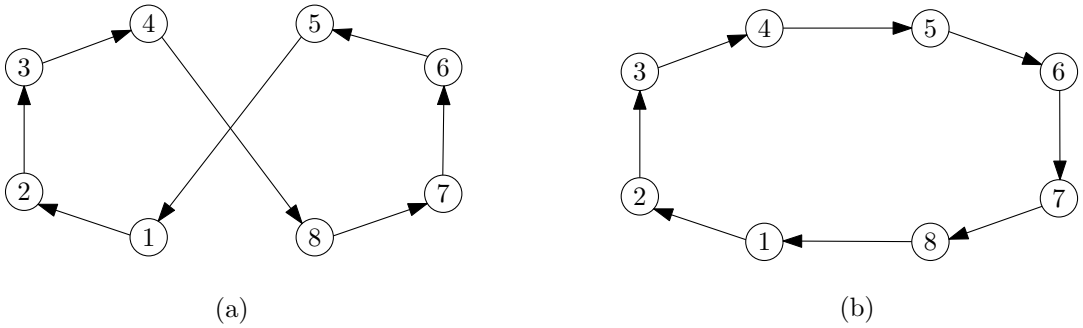


Figura 7: Heurística para PCV

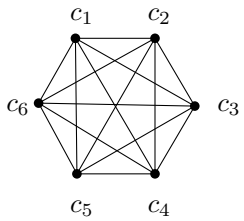


Figura 8: PCV — exemplo para algoritmo guloso

Considere a seguinte abordagem para resolver o PCV. Comece obtendo cerca de 1000 enumerações aleatórias. Então, dentre essas 1000, pegue a melhor — aquela que fornece a menor distância. É muito improvável que você tenha obtido *a* melhor solução. Mas ela será bastante boa. Além disso, testar essas enumerações é fácil. Não é necessário muito tempo para checar 1000 delas.

Agora vem a heurística: a técnica *2-opt*. Nós escolhemos aleatoriamente duas ligações entre cidades em nossa melhor solução. Então removemos essas ligações e as substituímos por outras de modo a manter a rota conectada.

Por exemplo, seja a situação mostrada na Figura 7. Nós removemos as ligações 5–1 e 4–8 na Figura 7(a), e as substituímos pelas ligações 4–5 e 8–1 na Figura 7(b). Note que algumas outras ligações precisaram ser invertidas para manter a rota conectada.

Em seguida, checamos se esta mudança melhora a distância total percorrida. Em caso afirmativo, usamos esta nova ordem e tentamos mudar mais algumas ligações aleatórias. Prosseguimos com esse processo até que um grande número de trocas tenha sido tentado sem melhora, e então paramos. A solução obtida pode não ser ótima, mas será muito boa.

Esta heurística pode ser generalizada para *k-opt*, isto é, você pode rearranjar três, quatro ou mais ligações, da mesma forma que foi feito para duas. Quantos mais ligações forem removidas, entretanto, mais complicado será reconstruir a rota de forma que ela permaneça conectada.

Outro exemplo guloso

O algoritmo guloso discutido acima para o PCV é chamado de *algoritmo do vizinho mais próximo*. Nós começamos por qualquer cidade e, a partir dali, mais para a cidade não visitada mais próxima, e continuamos até que todas as cidades tenham sido visitadas, quando então retornamos ara a cidade inicial. Existe uma versão mais sofisticada deste algoritmo guloso que considera todas as cidades mais próximas em cada estágio.

Seja o exemplo da Figura 8, com a matriz de distâncias abaixo:

	c_1	c_2	c_3	c_4	c_5	c_6
c_1	–	3	3	2	7	3
c_2	3	–	3	4	5	5
c_3	3	3	–	1	4	4
c_4	2	4	1	–	5	5
c_5	7	5	4	5	–	4
c_6	3	5	4	5	4	–

Nós começaremos pela cidade c_1. A cidade c_4 é a mais próxima de c_1 de acordo com a matriz, então criamos uma rota consistindo dessas duas cidades: c_1–c_4–c_1. Agora consideramos as duas cidades em nossa rota e encontramos a mais próxima a qualquer uma delas. A cidade c_3 é a mais próxima a c_4, então colocamos c_3 antes de c_4 em nossa nova rota c_1–c_3–c_4–c_1.

Agora há duas cidades, cada uma a 3 unidades de distância de cidades dessa rota: c_2 e c_6. Suponha que peguemos c_2 e a coloquemos antes de c_3 para obter a rota c_1–c_2–c_3–c_4–c_1. A cidade c_6 ainda está a 3 unidades de c_1, então inserimos c_6 antes de c_1: c_1–c_2–c_3–c_4–c_6–c_1. Finalmente, c_5 está a 4 unidades de c_3 e c_6, e então escolhemos inseri-la antes de c_6. Isto nos dá uma solução próxima do mínimo c_1–c_2–c_3–c_4–c_5–c_6–c_1 com um custo total de 19 unidades.

Nós podemos misturar heurísticas com um algoritmo guloso e tentar melhorar a solução obtida. Uma observação é que a distância total é, em geral, dependente da cidade inicial. Então, pela aplicação do algoritmo considerando cada uma das cidades como cidade inicial e tomando a menor das soluções obtidas, nós poderíamos conseguir uma melhora.

3.2 Caixeiro viajante II: divisão e conquista

Na seção anterior, vimos como as abordagens gulosa e de força bruta podem nos ajudar a resolver o PCV. Outra forma de resolver esse problema usa uma técnica com a qual você já conhece: divisão e conquista. Esta técnica trabalha quebrando recursivamente um problema em dois ou mais subproblemas de mesmo tipo (ou de tipo relacionado). Esta "divisão" continua até que os subproblemas se tornem simples o bastante para serem resolvidos diretamente. Então nós "conquistamos", ou seja, as soluções dos subproblemas são combinadas para obter uma solução para o problema original.

Esta abordagem é empregada na pesquisa binária:

```
1  /* Pesquisa binaria recursiva: encontre x em v[low]..v[high];
2     retorne o indice da localizacao. */
3  int binsearch(int x, int v[], int low, int high) {
4    int mid, loc;
5
6    mid = (low + high) / 2;
7    if(x == v[mid]
8      return mid;
9    else if((x < v[mid]) && (low < mid))
10     return binsearch(x, v, low, mid-1);
11   else if((x > v[mid]) && (high > mid))
12     return binsearch(x, v, mid+1, high);
13   else
14     return -1;
15 }
```

Emprega-se também em algoritmos de ordenação tais como mergesort ou quicksort:

```
1 Algoritmo: MergeSort (L)
2 se comprimento de L > 1 então
3 |  Divida a lista L em primeira e segunda metades
4 |  MergeSort (primeira metade)
5 |  MergeSort (segunda metade)
6 |  Intercale a primeira e a segunda metades na lista ordenada
```

Algoritmos de divisão e conquista tendem a ser inerentemente simples e eficientes, se os subproblemas forem corretamente divididos e combinados. O passo de combinação é, normalmente, o mais desafiador. Essa técnica presta-se, também, a implementações paralelas, onde os subproblemas são executados em processadores separados.

Há uma heurística de divisão e conquista para o PCV. É uma abordagem comum para frotas de transporte marítimo ou terrestre, que precisam resolver o PCV o tempo todo! Basicamente, o mapa com as paradas das rotas é dividido em pequenos grupos, e então rotas são construídas para esses grupos menores. Note que essa aplicação do PCV elimina o requisito de uma única visita em um determinado ponto.

Ruas também têm uma divisão natural em regiões, principalmente se uma estrada, rio ou outra barreira natural atravessa a região. É também comum atribuir todas as paradas de uma região a um único motorista, dando-lhe a oportunidade de se familiarizar com a área, o que aumenta a velocidade e a eficiência.

Esta é uma abordagem interessante para PCV, mas que não o "resolve"realmente. Uma solução exige que cheguemos a um algoritmo polinomial que funcione para todas as possíveis instâncias do problema. O que a divisão e conquista fornece é uma heurística que permite que nos aproximemos de uma solução pela divisão do mapa de rotas em mapas menores, que são mais fáceis de resolver.

3.3 Caixeiro viajante III: *branch and bound*

E se precisarmos de uma solução ótima?

Nas seções anteriores, apresentamos uma série de maneiras de resolver PCV, mas nenhuma (exceto a força bruta) garante que uma solução ótima será encontrada. Mas força bruta só é utilizável para um número pequeno de cidades. Existe alguma abordagem, que não de força bruta, capaz de garantir uma solução ótima? Felizmente, sim.

Para apresentar essa abordagem, começaremos falando de *árvores de busca* e *algoritmos de retrocesso* (*backtracking*). Algoritmos de retrocesso tentam cada possível solução até encontrar a correta. Isto é uma *busca em profundidade* (*depth-first search*) do conjunto de possíveis soluções. Durante a busca, se ficar claro que alternativa atual não é adequada, a busca retrocede ao último ponto em que alternativas diferentes foram apresentadas, e tenta a próxima alternativa. Isto prossegue até que todas as alternativas sejam tentadas. Normalmente, algoritmos de retrocesso têm implementações recursivas.

Uma árvore de busca é uma forma de representar a execução de um algoritmo de retrocesso. Nós começamos pela raiz e adicionamos novos nós à árvore para representar nossa exploração à medida que nos movemos em direção à solução. Se se tornar claro que chegamos a um beco sem saída, ou a uma folha da árvore, retrocedemos para explorar novos caminhos. O exemplo clássico é a exploração de um labirinto, onde os nós gerados representam tentativas de movimentação para norte, sul, leste e oeste (Figura 9).

Busca *branch and bound*

Busca *branch and bound*[12] é uma variação da técnica de retrocesso para problemas para os quais estamos procurando por uma solução ótima. O truque é calcular, para cada novo nó da árvore de busca, um limite (*bound*) para quais soluções esse nó será capaz de produzir.

Em outras palavras, quando adicionamos um novo nó à árvore, nós de alguma forma calculamos o valor da melhor solução possível que esse nó pode produzir. Esses limites servem a dois propósitos. Em primeiro lugar, se já temos uma solução que é melhor que o limite calculado para um nó, não precisamos mais explorar esse caminho (a árvore é "podada" nesse ponto). Em segundo lugar, usamos os limites como heurísticas para determinar quais nós devem ser expandidos primeiro.

Vamos aplicar essas ideias ao PCV. Primeiramente, precisamos encontrar uma maneira sistemática de explorar cada possível circuito. Considere o mapa da Figura 10. Nós podemos gerar uma árvore com todas as possíveis combinações de ligações, o que corresponde a uma abordagem de força bruta. Entretanto, o que queremos é, de alguma forma, podar essa árvore, ou seja, não queremos explorar todos os possíveis caminhos. Uma forma de conseguir isso é fornecer a cada nó algumas restrições definindo que ligações estão incluídas ou excluídas de nossa viagem.

[12] *Branch and bound* poderia ser traduzido por "enumeração e limitação". Entretanto, como não há uma tradução consensual em português, e o termo originaljá está consagrado na literatura, preferimos evitar a tradução (N.T.).

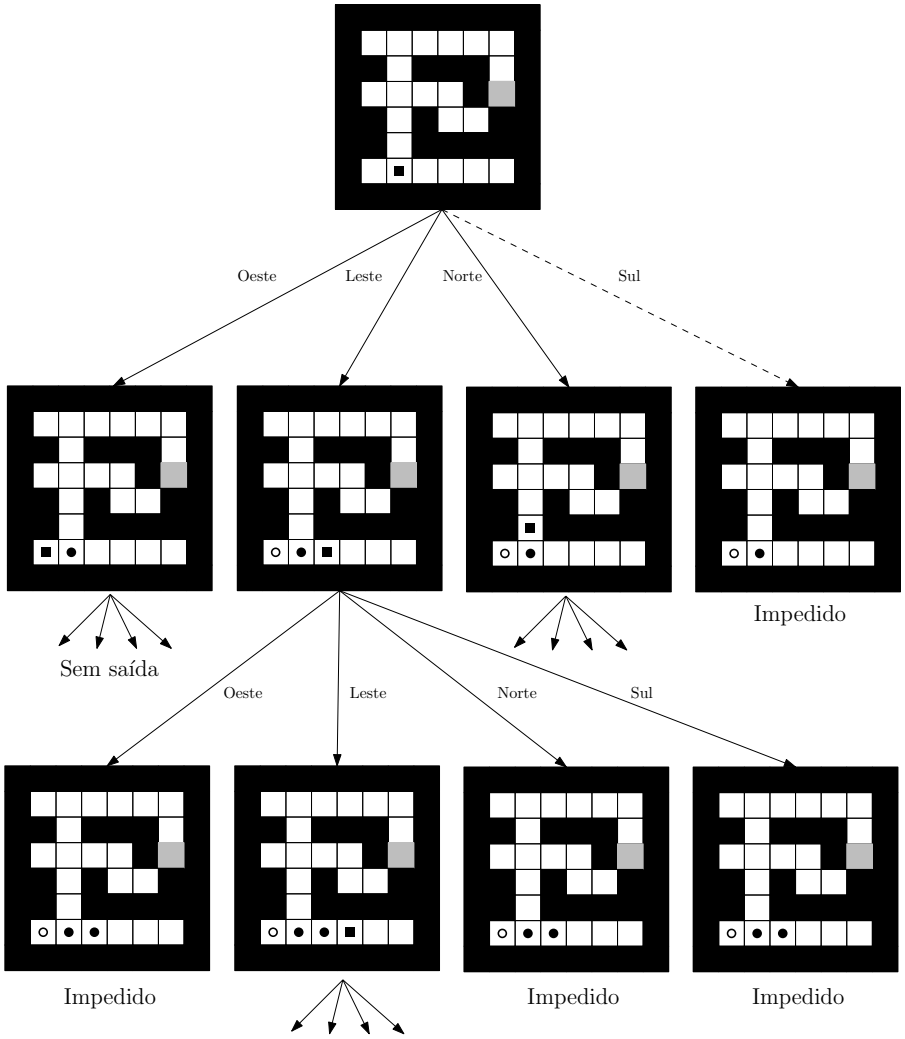


Figura 9: Exploração de labirinto

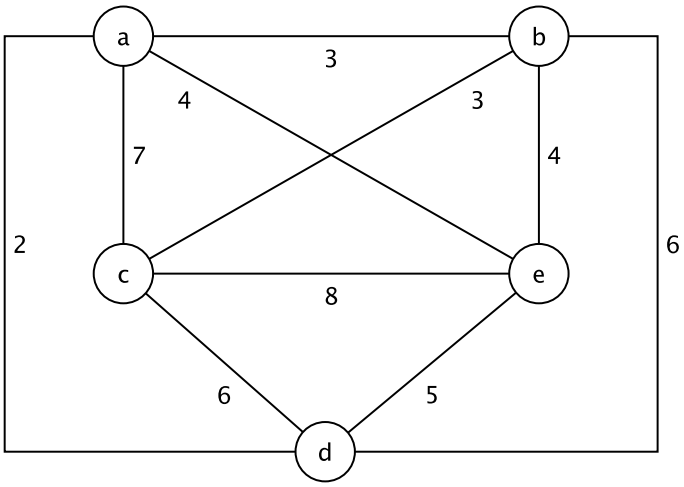


Figura 10: Exemplo *branch and bound*

Digamos que a raiz não tenha restrições. A partir dela, nós geramos duas possibilidades: uma incluindo e outra excluindo a primeira ligação (ao dizer "primeira", assumimos que as ligações estão ordenadas de alguma forma. Qualquer ordem serve; por exemplo, pode ser uma ordenação alfabética: *a*–*b*, *a*–*c*, *a*–*d*, ...). Nós prosseguimos de forma recursiva até que todas as possibilidades tenham sido geradas. Uma delas será a solução ótima.

A regra básica é: *no circuito final, cada cidade deve ter duas ligações incidentes* (uma vez que estamos construindo um *circuito*, ou seja, uma viagem em que as cidades não são visitadas mais de uma vez). Veremos logo em seguida como isto será utilizado.

Uma função limitante

Agora nós precisamos de uma *função limitante*. É possível provar que nenhum circuito pode custar menos que metade dos totais das duas ligações de menor custo que incidem sobre cada cidade[13]. Por exemplo, no grafo da Figura 10, as duas ligações mais baratas que incidem em *a* são *a*–*d* (2) e *a*–*b* (3), para um total de 5. Para a cidade *b* temos *a*–*b* (3) e *b*–*c* (3), para um total de 6. Prosseguindo desta forma para as demais cidades, temos que o limite inferior para um circuito completo desse grafo é $(5 + 6 + 8 + 7 + 9)/2 = 17,5$.

Agora, considere um circuito sujeito a algumas restrições. Suponha que a busca nos restrinja a incluir *a*–*e* e excluir *b*–*c*. As duas ligações de menor custo incidentes sobre *a* agora são *a*–*e* (à qual estamos restritos) e *a*–*d*, para um total de 6. Para *b* temos *b*–*e* e *b*–*d*, para um total de 10 (reparar que *b*–*c* foi explicitamente excluído pela restrição e *a*–*b* implicitamente, para que a cidade *a* não tenha três ligações incidentes). Para *c* selecionamos *c*–*d* e *c*–*e*, total 14, e assim por diante. O limite inferior para um circuito com essas restrições é 23.

Reparar que, no exemplo acima, algumas restrições foram inferidas. Sempre que um novo nó é adicionado a nosso espaço de busca, nós procuramos inferir tudo o que pudermos a respeito de quais ligações podem ser incluídas. Por exemplo, se precisarmos incluir *a*–*b* e *a*–*c*, então sabemos que não podemos incluir *a*–*d* e *a*–*e*, caso contrário *a* teria mais que duas ligações incidentes.

Dados as restrições, nós podemos calcular um limite inferior para o custo de um circuito. Se esse limite for pior que qualquer solução já encontrada, nós ignoramos o nó e retrocedemos. Caso contrário, geramos dois filhos para o nó, e usamos os limites inferiores de seus custos como heurística para determinar qual deles expandir primeiro. A Figura 11 mostra a árvore de busca construída para o grafo da Figura 10. Para simplificar, uma ligação tal como *a*–*b* foi representada simplesmente como *ab*. Se a ligação aparece entre parênteses, a restrição correspondente foi inferida tal como descrito acima.

A construção da árvore foi feita da seguinte forma:

1. Nós começamos pela raiz (nó (1)) sem restrição alguma. O limite inferior para a solução é 17,5.

2. A primeira ligação a considerar é *ab*. O nó (2) inclui essa ligação, e o nó (3) a exclui. Os limites inferiores para esses nós são calculados. Expandimos primeiramente o nó (2) porque ele tem um limite melhor que o (3).

3. A próxima ligação a considerar é *ac*. O nó (4) inclui essa ligação, e o nó (5) a exclui. Para o nó (4), podemos inferir que as ligações *ad* e *ae* devem ser excluídas. Novamente, escolhemos expandir primeiramente o nó mais promissor. Neste caso, expandiremos (5).

4. O nó (5) produz (6) e (7). (6) inclui *ad* e (7) a exclui. Para (6), podemos inferir que *ae* deve ser excluída; para (7), devemos incluir *ae* (caso contrário, *a* teria apenas uma ligação incidente). Escolhemos expandir (6) primeiro.

5. A próxima ligação a considerar é *bc*. Nó (8) inclui essa ligação, e nó (9) a exclui. Em ambos os casos, já temos informação suficiente para inferir um circuito completo. O circuito (8) custa 23, e o (9) 21. Este é o melhor circuito encontrado até o momento.

[13] No melhor caso, as duas ligações de menor custo para cada uma das cidades serão utilizadas. A divisão por 2 é necessária porque a soma simples conta cada ligação duas vezes; por exemplo, o custo da ligação *a*–*b* é somado tanto para a cidade *a* quanto para *b* (N.T.).

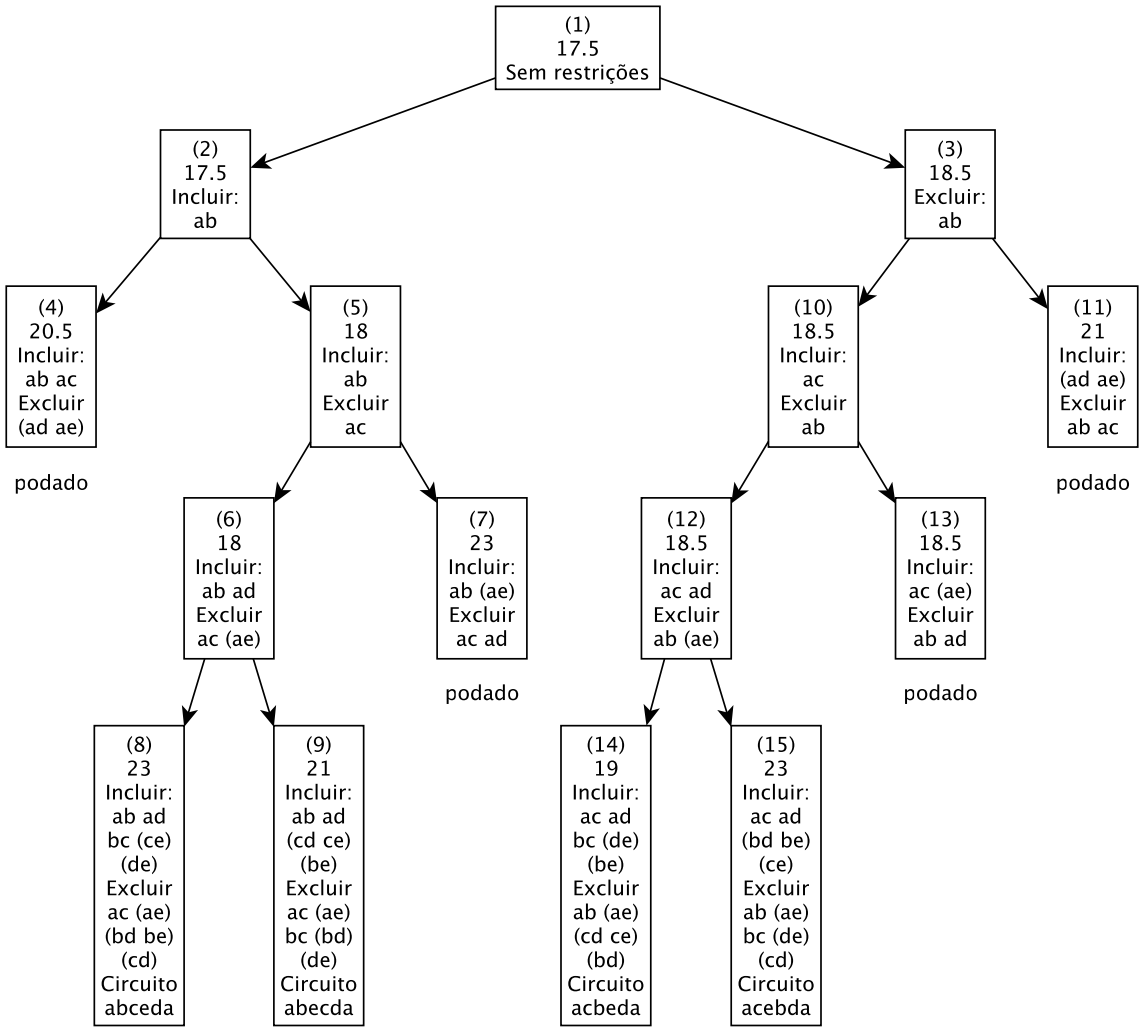


Figura 11: Árvore de busca

6. Neste ponto, (6) foi completamente expandido, e então retrocedemos para (5). Mas o outro filho de (5), (7), produz um circuito de custo 23 no melhor caso. Como isso é pior que o circuito já encontrado em (9), nós podemos podar esse nó.

7. Agora retrocedemos para (2), e descobrimos que também podemos podar (4) (20,5 é arredondado para 21, que não é melhor que nosso melhor circuito).

8. Agora retrocedemos para (1), e consideramos seu segundo filho (3).

9. Neste nível, estamos considerando a ligação *ac*. O nó (10) inclui essa ligação, e o nó (11) a exclui. Nós podemos podar (11) imediatamente, porque seu limite inferior é 21.

10. (10) produz (12) e (13). (13) também é imediatamente podado.

11. (12) produz (14) e (15). Ambos são circuitos completos. Vê-se que (14) produz um circuito de custo apenas 19. Este é agora registrado como nosso melhor circuito.

Quando todas as chamadas retornam, descobrimos que toda a árvore foi pesquisada, embora algumas partes tenham sido podadas. Então, o custo do melhor circuito é 19, dado o ponto de partida *a*.

Note que este algoritmo produz a melhor solução para o PCV. Como se sabe que PCV é NP-completo, nenhum algoritmo conhecido roda em tempo polinomial. Embora este algoritmo não seja polinomial, ele normalmente é mais rápido que a exploração de cada circuito possível. No pior caso, entretanto, ele poderia ser exponencial e reproduzir uma abordagem de força bruta.

3.4 Técnicas de programação dinâmica

Uma das técnicas mais poderosas para resolver problemas é quebrá-los em partes menores e mais fáceis de resolver. Problemas menores não são tão complexos, e permitem que nos concentremos em detalhes que podem se perder quando estamos considerando o problema global. Além disso, quebrar um problema em instâncias menores do mesmo tipo frequentemente sugere um algoritmo recursivo.

Há algumas categorias de algoritmos que se baseiam nessa estratégia de quebrar problemas em problemas menores. Anteriormente, vimos que *divisão e conquista* (seção 3.2) divide o problemas em algumas partes. Esses algoritmos tipicamente dividem o problema em duas metades, resolvem cada metade e então juntam as soluções para obter uma solução geral (por exemplo, quicksort). *Programação dinâmica* é uma técnica que quebra o problema em subproblemas, resolve esses subproblemas menores e guarda as soluções para que possam ser novamente usadas mais tarde.

Uma sequência de Fibonacci[14] é frequentemente usada para ilustrar o poder da programação dinâmica. A sequência é definida pela relação de recorrência a seguir:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \ (n \geq 2). \end{aligned}$$

Isto se traduz facilmente em uma função recursiva:

```
1 int Fibonacci(int n) {
2   if((n == 0) || (n == 1))
3     return n;
4   else
5     return (Fibonacci(n-1) + Fibonacci(n-2));
6 }
```

Isto parece bastante simples. Mas qual é o tempo de execução de Fibonacci()? Veja a chamada Fibonacci(4) na Figura 12.

Você deve se lembrar, de seu estudo sobre árvores binárias, que uma árvore praticamente completa, como a da Figura 12, tem aproximadamente 2^n nós, onde n é a altura da árvore (o número de níveis menos 1). Isto

[14] `http://en.wikipedia.org/wiki/Fibonacci_number`.

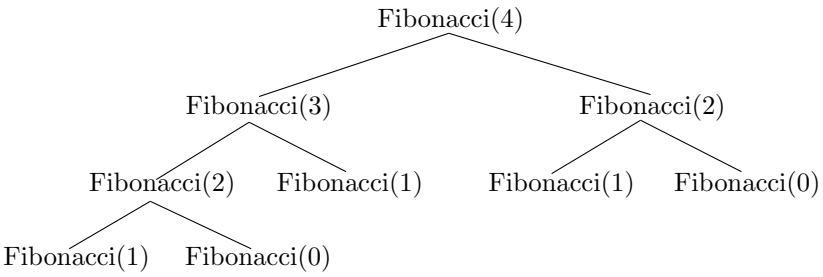


Figura 12: Execução de Fibonacci(4)

significa que o número de chamadas de função é aproximadamente 2^n, o que leva a um custo exponencial para a função, porque cada chamada é $O(1)$. Isto é muito caro para uma função tão simples!

A programação dinâmica fornece uma maneira muito elegante de resolver Fibonacci() e outros problemas caros. Note, na árvore da Figura 12, que há muito trabalho redundante sendo executado. Nós temos múltiplas chamadas para Fibonacci(0) e Fibonacci(1). Se armazenássemos esses valores e apenas os acessássemos quando fosse necessário, no lugar de fazer uma nova chamada recursiva, economizaríamos uma grande quantidade de tempo. É exatamente assim que a programação dinâmica trabalha. Aqui está o algoritmo para cálculo da sequência de Fibonacci rescrito com essa técnica:

```
1  int fib(int n) {
2    int f[n+1];
3
4    f[0] = 0;
5    f[1] = 1;
6
7    for(int i = 2; i <= n; i++)
8      f[i] = f[i-1] + f[i-2];
9
10   return f[n];
11 }
```

A primeira coisa a notar sobre essa função é que o tempo de execução é $O(n)$. Ela tem apenas um laço **for**() que faz aproximadamente n iterações. Isto é muito mais eficiente que a versão recursiva. Além disso, nós armazenamos valores sucessivos de série em f[i], que é uma característica chave da programação dinâmica. Estamos trocando espaço por tempo.

Problemas que podem ser resolvidos por programação dinâmica têm duas importantes propriedades:

- *Subestrutura ótima*: um problema tem essa propriedade se uma solução ótima pode ser eficientemente construída a partir de soluções ótimas para subproblemas.
- *Subproblemas sobrepostos*: um problema pode ser quebrado em subproblemas que são reutilizados diversas vezes. É isto que permite que salvemos resultados e os reutilizemos.

Nós podemos aplicar programação dinâmica de uma maneira *top-down*: o problema é dividido em subproblemas e esses subproblemas são resolvidos e têm as soluções guardados para o caso de serem novamente necessárias. Esta é uma abordagem que é implementada recursivamente.

Também podemos aplicar programação dinâmica de uma maneira *bottom-up*: os subproblemas que podem ser necessários são resolvidos previamente e então usados para construir soluções para problemas maiores. Nem sempre é claro quais subproblemas serão necessários, o que deixa esta abordagem mais desafiadora. Por outro lado, *bottom-up* é normalmente mais eficiente que *top-down* na utilização de espaço.

O problema do tabuleiro de damas[15] é outro excelente exemplo de aplicação de programação dinâmica. Algumas melhorias para o PCV[16] também foram desenvolvidas usando essa técnica.

[15] `http://en.wikipedia.org/wiki/Dynamic_programming#Checkerboard`.

[16] `http://citeseerx.ist.psu.edu/viewdoc/summary;jsessionid=A6B7BD7D9D879A059EC6BB7445AE4993?doi=10.1.1.22.7007`.

3.5 O quê usar, e quando?

Nas seções anteriores, construímos diversas soluções para o PCV (e vários outros problemas) baseadas em uma variaedade de abordagens diferentes. Nesta seção, vamos rever essas abordagens e fornecer diretrizes para a sua seleção em situações específicas. Na próxima seção, aplicaremos essas diretrizes em alguns outros problemas interessantes.

Como um desenvolvedor, é importante ter um conhecimento profundo dessas abordagens e de como usá-las. Além disso, desenvolvedores e cientistas da computação precisam de um conhecimento prático dos algoritmos mais comuns em várias categorias de problemas. Existem na web alguns bons catálogos[17] que podem ser consultados. Você deve gastar algum tempo examinando essas categorias para se certificar de que consegue utilizar esses algoritmos em seu trabalho.

Aqui está um resumo das abordagens discutidas até agora:

- Força bruta: esta técnica é, em geral, a primeira a ser tentada. Esses algoritmos são óbvios, fáceis de implementar, e fornecem soluções ótimas. Como desvantagem, eles são normalmente lentos, e algumas vezes tão lentos que inviabilizam sua utilização. Se você só precisa de calcular a solução uma única vez e o tempo de execução é razoável, força bruta pode ser uma abordagem válida.

  Exemplo: para o PCV, calcule todos os possíveis circuitos e escolha o de menor distância total. Isto é possível para $n \leq 10$.

- Algoritmos gulosos: são aqueles em que a melhor escolha é feita em cada estágio do algoritmo a partir da informação imediatamente disponível. Essas escolhas não levam em conta os dados de todos os estágios. Algumas vezes, a informação imediata é suficiente e uma solução ótima é encontrada; algumas vezes isto não acontece, e chega-se a soluções não ótimas. Como vimos, algoritmos gulosos não garante que uma solução ótima será encontrada.

  Para que um algoritmo guloso encontre uma solução ótima, o problema deve ter duas propriedades:

  - *Subestrutura ótima*: um problema tem esta propriedade se uma solução ótima puder ser eficientemente construída a partir de soluções ótimas para seus subproblemas.
  - *Escolha gulosa*: este termo se refere a como um algoritmo guloso faz a melhor escolha possível em cada estágio a partir da informação correntemente disponível. Ele faz uma escolha gulosa após a outra, reduzindo cada problema dado para um menor. Se uma solução ótima for encontrada usando um algoritmo guloso, então a informação imediatamente disponível em cada estágio foi suficiente para encontrar a solução ótima.

  Um algoritmo para encontrar a árvore geradora mínima[18] é um exemplo de algoritmo guloso que garante que uma solução ótima será encontrada porque o problema tem as propriedades de subestrutura ótima e escolha gulosa. Esse algoritmo foi desenvolvido por Robert Prim[19]. Existe também na Web uma página[20] com mais detalhes sobre árvores geradoras mínimas, e uma interessante animação sobre o algoritmo de Kruskal, outro algoritmo guloso que encontra uma solução ótima.

  PCV é um problema que não tem a propriedade de subestrutura ótima. A construção do circuito tem os requisitos de visita única a cada cidade e retorno à cidade inicial, e isto impossibilita a quebra de um mapa em "submapas" no caso geral. Para que uma solução gulosa funcione, os subproblemas devem ser *independentes*, ou seja, a solução de um subproblema não pode afetar a solução de outro. Este não é o caso dos submapas do PCV. Note que, no problema de encontrar uma árvore geradora mínima, os subproblemas são independentes.

  PCV também não possui a propriedade de escolha gulosa. Lembre-se do algoritmo para realizar uma busca do vizinho mais próximo[21] que propusemos (p. 33). Este é um exemplo de algoritmo guloso, mas que não fornece uma solução ótima. O algoritmo do vizinho mais próximo para PCV consiste em começar em qualquer cidade; a patir daí, vá para a cidade não visitada mais próxima, e continue

[17] Por exemplo, `http://www.cs.sunysb.edu/~algorith/`.
[18] `http://en.wikipedia.org/wiki/Minimum_spanning_tree`.
[19] `http://en.wikipedia.org/wiki/Prim%27s_algorithm#Description`.
[20] `http://www.cs.auckland.ac.nz/software/AlgAnim/mst.html`.
[21] `http://www.cs.sunysb.edu/%7Ealgorith/files/nearest-neighbor.shtml`.

até que todas tenham sido visitadas, quando então você retorna para a primeira cidade. As distâncias para os vizinhos mais próximos de uma cidade em particular não fornecem informação suficiente para encontrar a menor distância total.

Como dizer se um algoritmo tem essas propriedades? Esta informação seria muito útil porque o algoritmo guloso correto encontraria uma solução ótima. Infelizmente, não há maneira de determinar se uma estratégia gulosa resultará em uma solução ótima. Uma abordagem típica é tentar alguns algoritmos gulosos e ver como se comportam. Frequentemente, nós podemos intuir se um problema tem as duas propriedades tentando resolvê-lo com uma abordagem gulosa.

Entretanto, se um algoritmo guloso for capaz de produzir uma solução ótima, é frequentemente possível provar que essa solução é realmente ótima (por exemplo, no caso do algoritmo de Prim[22]).

- Heurísticas: são as diretrizes ou "princípios gerais" que podem ser usados para definir ou melhrar um algoritmo. É sempre uma boa ideia dar o passo seguinte de procurar por melhorias simples. Existe sempre uma questão de custo/benefício entre o grau de melhoria e o custo em termos do tempo de execução.

  Exemplo: se utilizamos um algoritmo guloso de busca pelo vizinho mas próximo para encontrar uma solução para PCV, é normal aplicar uma heurística para verificar se uma melhoria foi obtida. Começamos em qualquer cidade; a partir dalí, vamos para a cidade não visitada mais próxima, e continuamos até visitar todas as cidades, quando então retornamos à cidade inicial. A heurística consiste em executar esse algoritmo n vezes (para n cidades), em cada uma delas começando em uma cidade diferente. Isto frequentemente fornecerá uma distância total menor.

- Divisão e conquista: essa técnica trabalha recursivamente quebrando um problema em dois ou mais subproblemas do mesmo tipo, até que esses subproblemas se tornem suficientemente pequenos para serem resolvidos diretamente. As soluções dos subproblemas são então combinadas para fornecer uma solução para o problema original

  Esta é uma técnica comum para buscas e ordenações. Ela pode ser extremamente eficiente para certos tipos de problemas, particularmente quando os subproblemas não se sobrepõem. Se isso acontecer, como pr exemplo na relação de recorrência da série de Fibonacci, a divisão e conquista em geral se torna exponencial, e a programação dinâmica pode ser uma alternativa.

  Algoritmos de divisão e conquista podem fornecer soluções elegantes e eficientes para problemas que podem ser naturalmente quebrados em subproblemas. As Torres de Hanoi[23] são um excelente exemplo.

  Outros exemplos: mergesort, quicksort, busca binária.

- Programação dinâmica: técnica de divisão e conquista em que salvamos os resultados de subproblemas que se repretem. Problemas que podem ser resolvidos por programação dinâmica têm duas importantes propriedades:

  - *Subestrutura ótima*: um problema tem essa propriedade se pode ser eficientemente construído a partir de soluções ótimas para seus subproblemas.
  - *Subproblemas sobrepostos*: o problema pode ser dividido em subproblemas que são reutilizados diversas vezes. Isto é o que permite que os resultados sejam armazenados e reutilizados.

  Nós podemos aplicar programação dinâmica de uma maneira *top-down* ou *bottom-up* (ver p. 40). De qualquer maneira, ela é usada quando tentamos divisão e conquista e notamos que os subproblemas se repretem. Ela é também uma boa alternativa quando a abordagem gulosa não fornecer uma solução suficientemente ótima.

  Exemplo: a implementação iterativa da sequência de Fibonacci.

- *Branch and bound*: representa uma busca em profudidade no espaço de soluções de um problema. Nós organizamos a busca por uma solução na forma de uma árvore de busca, e tentamos podar essa árvore

[22] `http://en.wikipedia.org/wiki/Prim%27s_algorithm#Proof_of_correctness`.
[23] `http://en.wikipedia.org/wiki/Tower_of_Hanoi`.

com uma função limitante relacionada ao domínio do problema. Esta é uma forma eficiente de procurar uma solução ótima para um problema NP-completo, ou outros problemas sofisticados. No pior caso, dependendo de como a busca é organizada, cada possível solução será examinada, o que levará a um tempo de execução igual ao de força bruta. Entretanto, este não é o caso típico, embora também não seja uma solução polinomial.

Exemplos: a solução *branch and bound* para o PCV. A Web tem alguns outros exemplos[24] de aplicações *branch and bound*.

3.6 Alguns problemas

Os problemas a seguir são resolvidos com a aplicação das diretrizes fornecidas na seção 3.5. O objetivo é ajudá-lo a desenvolver suas habilidades e instintos em solução de problemas, e a determinar as técnicas mais adequadas em diferentes situações. Antes de ler os problemas abaixo, pode ser interessante ler na Web algum texto[25] que forneça um bom ponto de partida para a solução de problemas.

Problema #1

Vamos começar com um problema que nos permitirá usar algoritmos comuns. Ele consiste em encontrar o k-ésimo menor elemento em uma lista de inteiros.

Aqui estão alguns possíveis algoritmos que solucionam este problema:

- Ordenação: apenas classificamos a lista e extraímos o k-ésimo elemento a partir do começo da lista. O tempo de execução do algoritmo de ordenação mais eficiente é $O(n \log n)$ (quicksort, mergesort, heapsort).
- Nós podemos apenas percorrer a lista registrando o menor elemento encontrado até então. O tempo de execução é $O(n)$.
- Podemos fazer uma ordenação por seleção incompleta. Isto é, encontrar o menor valor e movê-lo par o início da lista. Encontrar o segundo menor e movê-lo para a segunda posição. Repetir até que o k-ésimo menor elemento seja movido. O tempo de execução é $O(kn)$.
- Utilizar o *algoritmo de seleção de Hoare*[26]. Ele é baseado no quicksort:

```
1  Algoritmo: seleção (lista,k,esquerda,direita)
2  Escolha um valor pivô lista[índicePivô]
3  novoPivôÍndice ← partição(lista,esquerda,direita,índicePivô)
4  se k = novoPivôÍndice então
5  |  retorna lista[k]
6  senão
7  |  se k < novoPivôÍndice então
8  |  |  retorna seleção (lista,k,esquerda,novoPivôÍndice-1)
9  |  senão
10 |  |  retorna seleção (lista,k-novoPivôÍndice,novoPivôÍndice+1,direita)
```

  O tempo de execução é como o do quicksort, que normalmente é $O(n \log n)$, mas pode executar em $O(n^2)$ se valores de particionamento ruins forem escolhidos de maneira consistente.

- Nós podemos usar uma estrutura de dados, tal como uma árvore binária, para minimizar o tempo de busca pelo k-ésimo elemento. O tempo de execução torna-se $O(\log n)$, mas haverá a sobrecarga de inserção e deleção de tal maneira que a árvore permaneça balanceada e em forma de árvore de busca.

Então, qual deles escolhemos? Há muitos fatores a considerar na escolha do algoritmo apropriado:

[24] `http://en.wikipedia.org/wiki/Branch_and_bound#Applications`.
[25] Por exemplo: `http://www.cs.sunysb.edu/~algorith/files/median.shtml`.
[26] `http://en.wikipedia.org/wiki/Selection_algorithm`.

- Quão frequentemente o algoritmo será executado? Se for apenas uma vez, então uma solução de tempo linear será adequada. Se precisarmos fazer essa seleção repetidamente, pode ser interessante investir na execução da ordenação de modo que a seleção possa ser feita em tempo $O(1)$, assumindo que a entrada não muda.

- Quão grande é a entrada? Ela muda? Em caso afirmativo, quão grande será o crescimento ou o encolhimento ao longo do tempo? A entrada complica as coisas. Se for muito grande, a ordenação se torna cara. Talvez a ordenação por seleção incompleta possa ser um bom compromisso. Se a entrada mudar frequentemente, a ordenação precisará ser refeita, mas talvez não todas as vezes se o elemento a ser inserido vem após o k-ésimo. Qual é o custo para se determinar isso?

- Quão importante é a rapidez? Se não for realmente tão importante (talvez o processo possa rodar em *background*) então uma solução linear pode ser adequada.

Problema #2

Frequentemente precisamos calcular *coeficientes binomiais* em problemas combinatórios. Em matemática, um coeficiente binomial é um coeficiente de qualquer um dos termos da expansão do bonômio $(x+y)^n$.

Aqui está um exemplo de como esses coeficientes são usados em combinatória. Digamos que existam n coberturas de sorvetes para serem escolhidas. Se você deseja criar um *sundae* com exatamente k coberturas, então os coeficientes binomiais expressam quantos tipos diferentes de *sundaes* com k coberturas existem.

Se já faz algum tempo que você trabalhou com coeficientes binomiais, a Web tem uma rápida revisão[27].

Nós estamos interessados em projetar um algoritmo para calcular um coeficiente binomial. Desta vez, as soluções estão apresentadas em ordem crescente de eficiência:

- Apenas aplique a fórmula:
$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!\,(n-k)!}.$$
Para evitar cálculos extras, podemos usar um truque[28] descrito na Web.

- Se precisamos apenas calcular um único coeficiente binomial, o cálculo de força bruta funciona bem. Outra ideia, para quando o cálculo não é tão frequente, é usar a definição recursiva[29]:
$$\begin{aligned} \left(\begin{array}{c} n \\ 0 \end{array} \right) &= 1 \\ \left(\begin{array}{c} n \\ k \end{array} \right) &= \frac{n \cdot \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right)}{k} \text{ para } k > 0 \end{aligned}$$
Como você pode imaginar, isto é extremamente lento, principalmente se usarmos uma versão recursiva do fatorial.

- Se precisarmos fazer o cálculo com frequência, podemos calcular o triângulo de Pascal uma única vez e depois realizar buscas. Esta é uma abordagem de programação dinâmica (ver detalhes[30] na Web).

- Finalmente, a solução mais elegante de todas é aquela definida por Lilavati[31] a mais de 850 anos. Ela roda em tempo $O(n)$.

[27] `http://mathworld.wolfram.com/BinomialCoefficient.html`.
[28] `http://www.zweigmedia.com/RealWorld/tutstats/bincoeffs.html`.
[29] Ver `http://penguin.ewu.edu/~trolfe/BinomialCoeff/index.html`.
[30] `http://www.cs.auckland.ac.nz/software/AlgAnim/binom.html`.
[31] `http://blog.plover.com/math/choose.html`.

4 Análise de algoritmos — problemas

Este conjunto de problemas permitirá que você adquira prática com a análise O-grande a partir dos pontos de vista matemático e algorítmico. Algumas vezes, você deverá fornecer uma contagem exata de instruções e, em outras, apenas uma aproximação O-grande. Ambas são habilidades importantes para cientistas da computação. Você também irá trabalhar com algumas relações de recorrência, e com análise de algoritmos recursivos. É provável que você ache esses últimos problemas mais desafiadores, então reserve mais tempo para eles.

Nos problemas a seguir, se for solicitada uma prova, seja preciso e sempre forneça referências se utilizar provas de teoremas de outras fontes; os passos mais óbvios, entretanto, podem ser deixados subtendidos (entretanto, não exagere). Se for solicitada uma prova indutiva, todos os seis passos devem ser incluídos (ver p. 46). Em todas as questões, mostre o seu desenvolvimento, de modo que possa ganhar créditos parciais.

O-grande

1. Dizemos que n_0 e c são *testemunhas* do fato de que $f(n)$ é $O(g(n))$ se, para todo $n \geq n_0$, é verdade que $f(n) \leq c \cdot g(n)$.

   (a) Se $n_0 = 1$, qual é o menor valor de c tal que n_0 e c sejam testemunhas do fato de que $(n+2)^2$ é $O(n^2)$?

   (b) Se $c = 5$, qual é o menor inteiro não negativo n_0 tal que n_0 e c sejam testemunhas do fato de que $(n+2)^2$ é $O(n^2)$?

   (c) Se $n_0 = 0$, para que valores de c é verdade que n_0 e c são testemunhas do fato de que $(n+2)^2$ é $O(n^2)$?

2. Indique se a afirmativa a seguir é verdadeira ou falsa e forneça uma prova de sua resposta: a^n é $O(b^n)$ se $1 < a \leq b$ (considere n inteiro).

3. Indique se a afirmativa a seguir é verdadeira ou falsa e forneça uma prova de sua resposta: se $f_1(n)$ e $f_2(n)$ são ambos limites estritos de uma função $T(n)$, então $f_1(n)$ e $f_2(n)$ são ambos O-grande um do outro. Aqui está a definição de "limite estrito": $f(n)$ é um limite O-grande estrito de $T(n)$ se:

   (a) $T(n)$ é $O(f(n))$;

   (b) Se $T(n)$ é $O(g(n))$ então também ocorre que $f(n)$ é $O(g(n))$.

   Informalmente, isto significa que não é possível encontrar uma função $g(n)$ que cresça pelo menos tão rápido quanto $T(n)$ mas cresça mais devagar que $f(n)$. Por exemplo, $T(n) = 2n^2 + 3$ e $f(n) = n^3$. Isto não representa um limite estrito porque podemos escolher $g(n) = n^2$. $T(n)$ é $O(g(n))$, mas $f(n)$ não é $O(g(n))$.

Análise de algoritmos

4. Escreva o algoritmo mais eficiente que você consiga imaginar (em C, Java ou pseudo-código) para o seguinte problema: dado um vetor de inteiros, encontre o menor deles. Qual é o tempo de execução em termos de O-grande, Θ-grande e Ω-grande? Explique sua resposta.

5. Escreva o algoritmo mais eficiente que você consiga imaginar para o seguinte problema: dado um conjunto de p pontos, encontre os pares de pontos mais próximos um dos outros. Qual é o tempo de execução em termos de O-grande, Θ-grande e Ω-grande? Explique sua resposta.

6. Escreva o algoritmo mais eficiente que você consiga imaginar para o seguinte problema: encontre o k-ésimo maior item em uma lista duplamente encadeada de n nós. Qual é o tempo de execução em termos de Θ-grande? Explique sua resposta.

Relações de recorrência

7. Resolva a seguinte relação de recorrência usando substituições repetidas. Faça uma prova indutiva mostrando que sua fórmula é correta:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= \frac{1}{2}T(n-1)+1 \text{ para inteiros } n>1. \end{aligned}$$

8. Uma cadeia que contem apenas 0's, 1's e 2's é chamada de cadeia ternária. Defina a relação de recorrência para o número de cadeias ternárias que contém dois símbolos consecutivos idênticos. Por exemplo, para $n=3$, as cadeias são 001, 000, 100, 002, 200, 110, 011, 111, 112, 211, 220, 022, 122, 221, 222.

9. Resolva a seguinte relação de recorrência usando a técnica que você preferir, e prove que sua resposta está correta usando indução:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 0 \\ a_n &= a_{n-2}/4 \text{ para } n \geq 2. \end{aligned}$$

10. Encontre o tempo de execução O-grande para as seguintes recorrências. Assuma que $T(1)=1$. Use o Teorema Mestre:

    (a) $T(n) = 2T(n/2) + n^3$.
    (b) $T(n) = 9T(n/3) + n$.
    (c) $T(n) = 25T(n/5) + n^2$.

Formato das provas indutivas

1. Construa $P(n)$.
2. Mostre que P (caso base) é verdadeiro.
3. Construa a hipótese indutiva (substitua n por k).
4. Mostre o que precisa ser provado (substitua n por $k+1$).
5. Afirme que você está começando a sua prova, e prossiga com a manipulação da hipótese indutiva (que é assumida como verdadeira) para encontrar uma ligação entre a hipótese indutiva e a afirmação a ser provada. Deixe sempre explícito o ponto em que você está usando a hipótese indutiva. Uma prova que não usa a hipótese indutiva não é uma prova por indução!
6. Sempre termine sua prova com uma afirmação do tipo "$P(k+1)$ é verdadeiro quando $P(k)$ é verdadeiro, portanto $P(n)$ é verdadeiro para todos os números naturais" (por vezes, apenas um "QED" pode ser suficiente). Nós apenas queremos que fique claro o ponto em que você sente que terminou.

Nota importante: em exames e exercícios para entregar, é importante apresentar sua prova na forma correta. Você deve *sempre* apresentar os passos 1–6 acima. O formato é importante em provas indutivas.

Um exemplo:

1. $P(n)$ é a afirmação $1+3+5+\cdots+(2n-1)=n^2$, ou seja, a soma dos primeiros n inteiros ímpares é n^2.
2. Caso base: prove que $P(1)$ é verdadeiro. Tem-se $1=1^2=1$; a soma do primeiro número inteiro ímpar é igual a 1 (inclua ambas as partes).
3. A hipótese indutiva é assumir $P(k)$: $1+3+5+\cdots+(2k-1)=k^2$.

4. A afirmação a ser provada é $P(k+1)$: $1+3+5+\cdots+(2k-1)+(2k+1)=(k+1)^2$.

5. Prova: simule a d\adição de outro termo a ambos os lados da hipótese indutiva:

$$\begin{aligned} 1+3+5+\cdots+(2k-1)+(2k+1) &= k^2+(2k+1) \\ &= k^2+2k+1 \\ &= (k+1)^2. \end{aligned}$$

6. Conclusão: $P(k+1)$ é verdadeiro quando $P(k)$ é verdadeiro, portanto $P(n)$ é verdadeiro para todo $n \in \mathbb{N}$.

5 Exploração de algoritmos — problemas

Como um desenvolvedor ou engenheiro de software, é importante que você adquira um entendimento profundo das abordagens algorítmicas apresentadas na seção 3, e como e quando utilizá-las. Além disso, desenvolvedores e cientistas da computação precisam de um conhecimento prático dos algoritmos clássicos. Neste conjunto de problemas, você terá a oportunidade de analisar diversos problemas e empregar as abordagens estudadas.

1. Escreva o algoritmo mais eficiente que você consiga imaginar para o seguinte problema: encontre o k-ésimo item em uma lista duplamente encadeada de n nós. Qual é o tempo de execução em termos de Θ-grande?

2. Escreva o algoritmo mais eficiente que você consiga imaginar para o seguinte problema: dado um conjunto de p pontos, encontre os pares de pontos mais próximos uns dos outros. Certifique-se de tentar diversas abordagens, principalmente divisão e conquista. Qual é o tempo de execução em termos de Θ-grande?

3. Projete e implemente um algoritmo que encontre todos os elementos duplicados em uma sequência aleatória de inteiros.

4. Reconsidere seus algoritmos para os três problemas acima à luz da seguinte informação: a entrada agora é 1.000.000.000 de vezes maior.

5. Você tem dois vetores de inteiros. Dentro de cada um, não há valores duplicados, mas pode haver valores em comum com o outro vetor. Assuma que os vetores representam dois conjuntos diferentes. Defina um algoritmos $O(n \log n)$ que forneça um terceiro vetor representando a união dos dois conjuntos. O valor n é a soma dos tamanhos dos dois vetores de entrada.

6. Sejam dadas duas sequências de números, uma crescente $u_1, u_2, \ldots, u_m$ e outra decrescente $d_1, d_2, \ldots, d_n$. Seja um número C. Determine se C pode ser escrito como a soma de um u_i e um d_j. há um método de força bruta óbvio (apenas compare todas as soma a C) , mas existe uma solução muito mais eficiente. Defina um algoritmo que trabalhe em tempo linear.

7. Projete e implemente um algoritmo de programação dinâmica para solucionar o problema do troco[32]. Seu algoritmo deve sempre encontrar uma solução ótima — mesmo quando um algoritmo guloso falhar.

8. José tem uma fábrica com várias máquinas interessantes. Seu negócio é criar qualquer acessório que alguém precise. Ele apenas alinha as máquinas em diferentes sequências para pintar, cortar metal, ou furar buracos.

   José está com muitas encomendas atualmente. Hoje ele tem n trabalhos para concluir: $(j_1, j_2, \cdots, j_n)$. Cada trabalho j_i leva um tempo t_i, tem uma valor v_i um horário de entrega d_i. José não tem máquinas suficientes para realizar mais de um trabalho por vez, então cada trabalho deve ser executado sem interrupção por um tempo t_i. Se o trabalho j_i for completado dentro de seu horário d_i, então José tem um lucro v_i. Se ele não termina um trabalho a tempo, então ele não recebe.

[32] `http://www.brpreiss.com/books/opus4/html/page442.html`.

Tudo isto é muito complicado para José, portanto ele contratou você para projetar um algoritmo para determinar a "melhor" escala de trabalhos, ou seja, aquela em que ele obtém mais lucro. Para simplificar, assuma que todos os tempos são inteiros entre 1 e n. Determine o tempo de execução de seu algoritmo.

9. O *problema da mochila* aparece quando há alguma alocação de recursos sujeita a várias restrições. Por exemplo, dado um orçamento fixo, como selecionar o que comprar? Tudo tem um custo e um valor, então queremos o maior valor para um dado custo. O termo "problema da mochila" invoca a imagem do trilheiro que está restrito a uma mochila de tamanho fixo e precisa enchê-la apenas com os itens mais úteis.

   Uma forma típica é o *problema da mochila 0/1*, em que cada item deve ser inteiramente colocado na mochila ou deixado de fora. Objetos não podem ser quebrados arbitrariamente em pedaços menores. Esta propriedade 0/1 deixa o problema difícil. É fácil encontrar um algoritmo guloso para uma seleção ótima quando podemos quebrar os objetos. Para cada item, poderíamos calcular seu "preço por quilo", e pegar o máximo do item mais valioso até esgotá-lo ou encher a mochila. Então faríamos o mesmo com o próximo item mais valioso, etc., até que a mochila fique cheia. Infelizmente, a restrição 0/1 normalmente é um requisito.

   Digamos então que você seja um ladrão com uma mochila de certa capacidade. Você examina o lugar que acabou de invadir e vê itens de diferentes valores e pesos. Você gostaria que os itens escolhidos tivessem o maior valor possível, e ainda assim coubessem na mochila. Este é um "problema da mochila 0/1" porque você, o ladrão, pode levar (1) ou deixar (0) cada item.

   Dado um conjunto de itens, cada um com um custo e um valor, determine o número de cada um deles a incluir na coleção de forma que o custo total não exceda um determinado limite e o valor total seja o maior possível. O problema da mochila 0/1 restringe o número de cada item a zero ou um.

   Nós devemos maximizar:

   $$\sum_i q_i \cdot v_i$$

   onde q_i é a quantidade do item i ($q_i = 0$ ou 1), v_i é o valor do item i e o somatório é sobre todos os itens, sujeito à seguinte restrição:

   $$\sum_i q_i \cdot p_i \leq C$$

   onde p_i é o peso do item i, C é um limite de peso e, novamente, o somatório é sobre todos os itens.

   Há diversas abordagens a tentar:

   - Gula por lucro: a cada passo, selecione um dos itens restantes que seja o de maior valor v_i. Esta abordagem tenta maximizar o lucro escolhendo os itens mais lucrativos primeiro.
   - Gula por peso: a cada passo, selecione um dos itens restantes que seja o de menor peso p_i. Esta abordagem tenta maximizar o lucro colocando na mochila o maior número possível de itens.
   - Gula por densidade de lucro: a cada passo, selecione um dos itens restantes que seja o de maior densidade de lucro v_i/p_i. Esta abordagem tenta maximizar o lucro escolhendo os itens de maior valor por unidade de peso.

   Projete um algoritmo que resolva este problema para um pequeno conjunto de dados. Assuma cinco itens de pesos 20, 50, 40, 10 e 30 com valores 15, 20, 5, 25 e 10, respectivamente. Teste várias abordagens gulosas e heurísticas e veja se consegue encontrar uma solução ótima para esta instância. Então, após definir seu algoritmo, teste-o com conjuntos de dados cada vez maiores e veja como ele se comporta.

10. Determine a melhor forma de resolver o PCV dados os seguintes requisitos:

    - Há 20 paradas a fazer e cada uma deve ser visitada exatamente uma vez.
    - A cada vez que precisamos viajar, as paradas (e as distâncias) mudam.
    - Soluções próximas do ótimo são aceitáveis — não precisamos de soluções ótimas, mas elas devem ser boas por causa dos custos envolvidos.

Comece implementando um algoritmo do vizinho mais próximo (p. 32) para um conjunto específico de dados (crie esse conjunto). Para os objetivos deste problema, assuma que cada parada tem ligações com todas as outras, então não precisamos nos preocupar em encontrar um circuito — apenas precisamos encontrar o circuito com a menor distância total. Uma forma comum de armazenar a informação de distância é utilizar uma matriz $n \times n$ (onde n é a quantidade de paradas).

Em seguida, incremente sua solução empregando a abordagem de todos os vizinhos mais próximos (p. 33). Finalmente, aplique a heurística de deixar que cada cidade seja a cidade inicial (p. 42). Compare os tempos de execução para cada uma das soluções com seu conjunto de dados.

Agora tente todas as três soluções em um conjunto de dados completamente diferente e compare os resultados. Tente mais dez conjuntos diferentes e compare. Qual dos algoritmos se comporta melhor?

Referências

[Aho e Hopcroft 1974]AHO, A.; HOPCROFT, J. *The Design and Analysis of Computer Algorithms*. Reading: Addison-Wesley, 1974.

[Aho e Ullman 1992]AHO, A.; ULLMAN, J. D. *Foundations of Computer Science*. New York: W. H. Freeman, 1992.

[Brassard e Bratley 1988]BRASSARD, G.; BRATLEY, P. *Algorithmics: Theory and Practice*. Englewood Cliffs: Prentice Hall, 1988.

[Cormen, Leiserson e Rivest 1991]CORMEN, T.; LEISERSON, C.; RIVEST, R. *Introduction to Algorithms*. New York: McGraw-Hill, 1991.

[Hartmanis e Sterns 1995]HARTMANIS, J.; STERNS, R. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, v. 117, 1995.

[Knuth 1973]KNUTH, D. *Fundamental Algorithms*. 2. ed. Menlo Park: Addison-Wesley, 1973. (The Art of Programming, v. 1).

[Knuth 1973]KNUTH, D. *Seminumerical Algorithms*. 2. ed. Menlo Park: Addison-Wesley, 1973. (The Art of Programming, v. 2).

[Knuth 1973]KNUTH, D. *Sorting and Searching*. 2. ed. Menlo Park: Addison-Wesley, 1973. (The Art of Programming, v. 3).

[Manber 1989]MANBER, U. *Introduction to Algorithms: A Creative Approach*. Reading: Addison-Wesley, 1989.

[Sedgewick 1989]SEDGEWICK, R. *Algorithms*. 2. ed. Reading: Addison-Wesley, 1989.

Versão 1.0.6