Programação Competitiva

Matheus Pimenta

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sumário

Capítulo 1

Introdução

Este capítulo introduz os conceitos básicos fundamentais para o estudo de algoritmos e ferramentas e técnicas utilizadas em programação competitiva.

1.1 Notação assintótica

Popularizada por Donald Knuth, notação assintótica é uma ferramenta matemática utilizada para comparar o crescimento de funções.

Definimos as classes de funções $O(f(n))$, $\Theta(f(n))$ e $\Omega(f(n))$ da seguinte maneira.

- $O(f(n)) = \{g(n) \mid \text{existem constantes } n_0 \text{ e } c \text{ tais que } g(n) \leq cf(n), \forall n \geq n_0\}$
  Se $g(n) \in O(f(n))$, dizemos que $f(n)$ é um *limite superior assintótico* para $g(n)$, ou simplesmente *limite superior*.

- $\Theta(f(n)) = \{g(n) \mid \text{existem constantes } n_0, c_1 \text{ e } c_2 \text{ tais que } c_1 f(n) \leq g(n) \leq c_2 f(n), \forall n \geq n_0\}$
  Se $g(n) \in \Theta(f(n))$, dizemos que $f(n)$ é um *limite estreito assintótico* para $g(n)$, ou simplesmente *limite estreito*.

- $\Omega(f(n)) = \{g(n) \mid \text{existem constantes } n_0 \text{ e } c \text{ tais que } cf(n) \leq g(n), \forall n \geq n_0\}$
  Se $g(n) \in \Omega(f(n))$, dizemos que $f(n)$ é um *limite inferior assintótico* para $g(n)$, ou simplesmente *limite inferior*.

Apesar de ser abuso de notação, a escrita popular de relações assintóticas utiliza o sinal de igual ($=$) ao invés do sinal de pertence ($\in$). Adotaremos esta convenção daqui para frente.

Propriedades:

(1) $g(n) = \Theta(f(n))$ se, e somente se, $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$.

(2) Se $g(n) = c_1 g_1(n) + c_2 g_2(n) + ... + c_k g_k(n) + c_{k+1}$, então $g(n) = \Theta(g_i(n))$, para algum $i \in \{1, 2, ..., k\}$. Em outras palavras: a) constantes aditivas e multiplicativas não influenciam relações assintóticas; e b) em uma função $g(n)$ há sempre um termo dominante que determina o limite estreito de $g(n)$.

Os métodos gerais para verificar o relacionamento assintótico entre duas funções $g(n)$ e $f(n)$ são:

- $g(n) = O(f(n)) \iff \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

- $g(n) = \Omega(f(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

Tabela 1.1.1: Exemplos de relações assintóticas.

Exemplo	Relação assintótica	Constantes	Figura 1.1.1
1	$3n^3 + 5n = \Theta(n^3)$	$n_0 = 2, c_1 = 1, c_2 = 10$	(a)
2	$3n^5 = O(2^n)$	$n_0 = 30, c = 1$	(b)
3	$7 \log_{35} n = \Theta(\log_2 n)$	$n_0 = 2, c_1 = 1, c_2 = 2$	(c)
4	$0.7n^{0.4} = \Omega(\log_2 n)$	$n_0 = 1000, c = 1$	(d)

A Tabela 1.1.1 mostra exemplos que ilustram as propriedades descritas acima e as principais relações assintóticas entre funções. Note que, em cada linha, as constantes são apenas uma escolha de um conjunto infinito de escolhas que verificam a respectiva relação assintótica. A coluna da direita indica as figuras que ilustram os respectivos exemplos. Vejamos algumas observações acerca de cada exemplo.

- O Exemplo 1 ilustra as propriedades acima.
- Exponenciais crescem mais rápido que polinômios. Esta é a relação ilustrada pelo Exemplo 2.
- O Exemplo 3 ilustra o fato de que logaritmos em diferentes bases sempre crescem com a mesma rapidez. Isto decorre do fato de que $\log_{c_1} n / \log_{c_2} n = \log_{c_2} c_1$, ou seja, logaritmos em diferentes bases diferem apenas por uma constante. Por esta razão, utilizaremos uma única notação para o logaritmo de um número n: $\lg n$. Observe que o mesmo não ocorre para exponenciais! Exponenciais em diferentes bases *não* diferem apenas por uma constante!
- Como ilustrado no Exemplo 4, polinômios crescem mais rápido que logaritmos.

De modo geral, se $k > 1$, então

$$1 \ll \lg n \ll \sqrt[k]{n} \ll n \ll n \lg n \ll n^k \ll k^n \ll n! \ll n^n$$

A seguir estão listados mais alguns exemplos.

- $7n^2 + 8n = O(n^2)$
- $100n = O(n^2)$
- $42 = O(n^2)$
- $327 = O(1)$
- $327 = \Theta(1)$
- $327 = \Omega(1)$
- $500n^2 + 30 = \Omega(n^2)$
- $500n^2 + 30 = \Omega(n)$
- $500n^2 + 30 = \Omega(1)$

Há mais definições e propriedades a serem vistas acerca deste conteúdo que são dispensáveis para competições de programação.

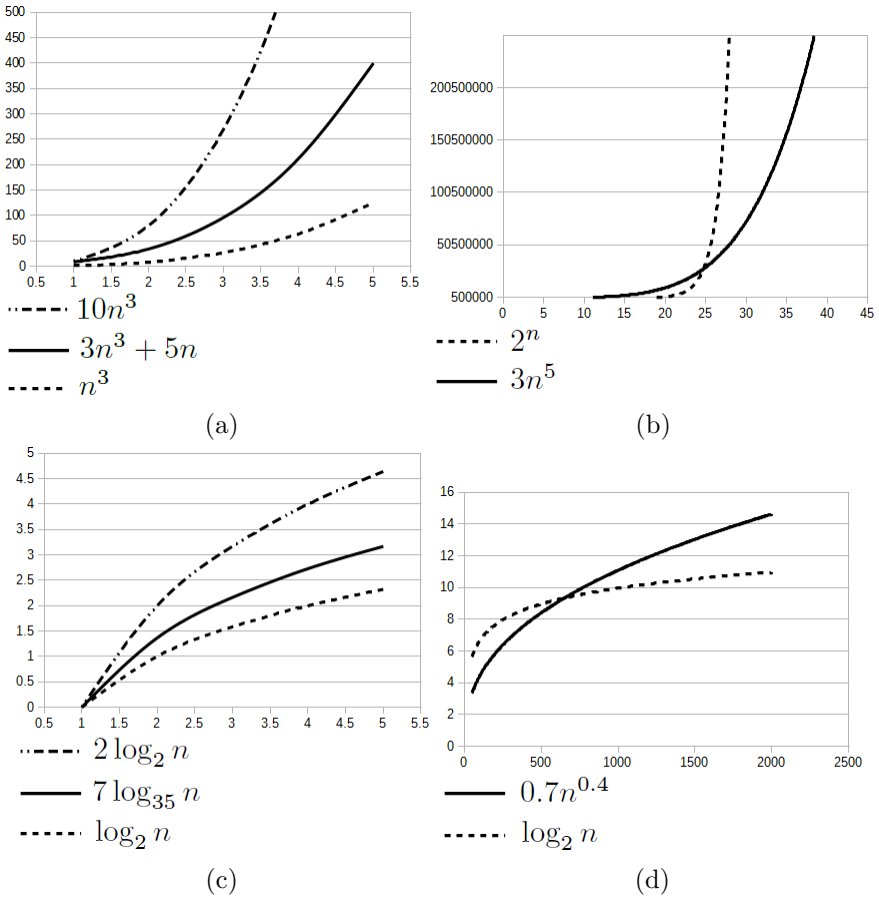

Figura 1.1.1: Gráficos dos exemplos de relações assintóticas da Tabela 1.1.1.

1.2 Análise de algoritmos

Antes de implementar um algoritmo para resolver um problema em um torneio, um competidor deve estar completamente convencido de que este algoritmo está correto e de que haverá tempo e memória suficientes para ele executar. Implementar um algoritmo sem ter estas certezas é um erro fatal. Desenvolvida por Donald Knuth, a análise de algoritmos é responsável por determinar os custos de um algoritmo em relação a um recurso, como tempo, ou espaço.

A quantidade de um recurso utilizada por um algoritmo varia em diferentes execuções. Tal quantidade depende do *tamanho da entrada*. Então, analisar a complexidade de um algoritmo signifca encontrar funções que modelam os custos de tempo/espaço do algoritmo em termos do tamanho da entrada. É importante ressaltar que o tamanho da entrada pode significar coisas distintas para problemas distintos. Em problemas de grafos, por exemplo, é comum haverem dois tamanhos de entrada: o número vértices e o número de arestas. Um exemplo menos trivial é um problema cuja entrada é simplesmente uma quantidade fixa números. Neste caso, os tamanhos da entrada são as respectivas quantidades de dígitos utilizados para representar cada número. Devemos definir como tamanho da entrada o que for mais conveniente para termos uma boa medição dos custos de um algoritmo.

Para simplificar esta discussão inicial sobre análise de algoritmos, vamos agora, sem perda de generalidade, considerar um problema qualquer com um único tamanho de entrada n. Podemos facilmente estender as definições que vêm a seguir para problemas com múltiplos tamanhos de entrada. Fixando o valor de n em um valor n_0 qualquer, há ainda outro fator que pode influenciar nos custos de um algoritmo. Este fator é a entrada em si. Ou seja, distintas entradas de tamanho n_0 ainda podem ter diferentes custos de tempo/espaço. Dizemos que ocorre o *melhor caso* quando a utilização de recursos de um algoritmo é mínima. Analogamente, dizemos que ocorre o *pior caso* quando a utilização de recursos de um algoritmo é máxima. Para o algoritmo de ordenação *bubble sort*, por exemplo, ocorre o melhor caso quando o vetor já está ordenado e o pior caso quando o vetor está invertido. Então, redefinindo, analisar a complexidade de um algoritmo signifca encontrar funções que modelam os custos de tempo/espaço do algoritmo em termos do tamanho da entrada no melhor, ou no pior caso.

Em um torneio de programação, geralmente basta conhecer o pior caso de um algoritmo para poder dizer se ele vai ter utilidade, ou não. As instâncias de teste são geralmente bem calibradas para testar o pior caso. No entanto, conhecer a complexidade de um algoritmo também no melhor caso pode ser útil para descartá-lo.

Para concluir, definimos a complexidade de um algoritmo da seguinte maneira. Seja $f_i(n)$ uma função do tamanho da entrada n que modela o custo de {tempo, espaço} de um algoritmo em seu melhor caso e seja $f_s(n)$ uma função do tamanho da entrada n que modela o custo de {tempo, espaço} do algoritmo em seu pior caso. Então, dizemos que a complexidade de {tempo, espaço} do algoritmo é $\Omega(f_i(n))$ e $O(f_s(n))$. Se, além disso, existe uma função $f(n)$ tal que $f_i(n) = \Theta(f(n))$ e $f_s(n) = \Theta(f(n))$, então o custo do algoritmo é o mesmo em qualquer caso e dizemos que a complexidade de {tempo, espaço} do algoritmo é $\Theta(f(n))$. De maneira mais sucinta, dizemos simplesmente que um algoritmo é $\Omega(f_i(n))$, $O(f_s(n))$, ou $\Theta(f(n))$.

Analisar um algoritmo é uma tarefa que não pode ser feita por outro algoritmo e exige um pouco de criatividade e experiência. A seguir são dados alguns exemplos.

1.2.1 Complexidade de espaço

Analisar a complexidade de espaço de um algoritmo é literalmente encontrar uma função que modele o uso de memória do algoritmo. Em geral, isto é mais fácil do que analisar complexidade de tempo.

Começamos com um exemplo simples, onde a complexidade de espaço é $\Theta(n)$. Observe que as únicas variáveis do Algoritmo 1.2.1 são v e i. O espaço ocupado por v é kn e o espaço ocupado por i é k, onde k é o espaço ocupado por um inteiro. Observe também que o custo de espaço não depende da entrada em si, mas apenas do seu tamanho. Logo, $S(n) = kn + k = \Theta(n)$.

Algoritmo 1.2.1

Entrada: $v[1..n]$: vetor de inteiros
1: i : inteiro
2: **para** $i \leftarrow 1$ **até** n **faça**
3: $\quad v[i] \leftarrow v[i] + 1$
4: **fim para**

Agora, vemos um exemplo um pouco mais complexo, onde a complexidade de espaço é $\Theta(n^2)$. Inicialmente pode parecer que a complexidade de espaço do Algoritmo 1.2.2 é a mesma do Algoritmo 1.2.1, pois ambos recebem um vetor de inteiros e declaram uma variável auxiliar. No entanto, o Algoritmo 1.2.2 chama n vezes uma função que recebe um inteiro, declara uma matriz $n \times n$ de inteiros e declara um inteiro. Observe que as decisões tomadas pelo algoritmo ocorrem após a chamada da função, de modo que a complexidade de espaço deve ser a mesma em qualquer caso. Logo, $S(n) = (kn + k) + (k + kn^2 + k) = \Theta(n^2)$.

Algoritmo 1.2.2

Entrada: $v[1..n]$: vetor de inteiros
1: i : inteiro
2: **para** $i \leftarrow 1$ **até** n **faça**
3: $\quad$ **se** FOO$(n) \neq v[i]$ **então**
4: $\quad\quad v[i] \leftarrow 0$
5: $\quad$ **fim se**
6: **fim para**
7: **função** FOO(n : inteiro)
8: $\quad m[1..n][1..n]$: matriz de inteiros
9: $\quad d$: inteiro
10: $\quad$ sorteie valores para todas as posições de m
11: $\quad d \leftarrow$ determinante de m
12: $\quad$ **retorne** d
13: **fim função**

O próximo exemplo aparenta ser simples, mas pode confundir um pouco. Desta vez a complexidade é $\Omega(1)$ e $O(2^n)$. O Algoritmo 1.2.3 apenas recebe o inteiro i, assim como a função que ele chama. Isto nos faz pensar inicialmente que a sua complexidade de espaço é $S(n) = k + k = \Theta(1)$. No entanto, esta é a complexidade apenas no melhor caso ($i \leq 0$), ou seja, o algoritmo usa espaço $\Omega(1)$. Observe que o espaço alocado por uma função só é liberado quando a função chega ao fim; e que a função chamada pelo algoritmo é *recursiva*. Então, para o pior caso, precisamos analisar o algoritmo para saber quantas chamadas recursivas podem acontecer para uma dada entrada. Observe que este valor é exatamente i. Logo, $S(n) = k + ki = \Theta(i)$. O que há de errado com esta análise? Este é um caso especial em que a complexidade do algoritmo pode ser expressa em termos do *valor* da entrada. Como queremos encontrar uma complexidade em função do *tamanho* da entrada, precisamos encontrar uma relação entre i e n. Note que, neste caso, o tamanho da entrada n é o número de *bits* utilizados para representar i, ou seja, $\lg i$. Logo, $S(n) = k + ki = k + k2^{\lg i} = k + k2^n = \Theta(2^n)$. Logo, o algoritmo usa espaço $O(2^n)$.

Analisamos o Algoritmo 1.2.4 e vemos que o melhor caso ocorre quando a soma dos elementos de v é diferente de zero. Neste caso, o algoritmo usa espaço $S_i(n) = kn + k + k = \Theta(n)$. Quando a soma é zero, o algoritmo chama uma função que consome espaço $k + kn^2 + k$, de modo que $S_s(n) = (kn + k + k) + (k + kn^2 + k) = \Theta(n^2)$. Logo, não podemos dizer que a complexidade de espaço do algoritmo é $\Theta(n)$, nem que é $\Theta(n^2)$, mas podemos dizer que é $\Omega(n)$ e que é $O(n^2)$.

Algoritmo 1.2.3

Entrada: i : inteiro

```
1: FOO(i)
2: função FOO(i : inteiro)
3:     se i > 0 então
4:         FOO(i − 1)
5:         imprima i
6:     fim se
7: fim função
```

Algoritmo 1.2.4

Entrada: $v[1..n]$: vetor de inteiros

```
1: sum : inteiro
2: i : inteiro
3: sum ← 0
4: para i ← 1 até n faça
5:     sum ← sum + v[i]
6: fim para
7: se sum = 0 então
8:     imprima FOO(n)
9: senão
10:    imprima sum
11: fim se
12: função FOO(n : inteiro)
13:    m[1..n][1..n] : matriz de inteiros
14:    d : inteiro
15:    sorteie valores para todas as posições de m
16:    d ← determinante de m
17:    retorne d
18: fim função
```

1.2.2 Complexidade de tempo

Antes de analisar a complexidade de tempo de um algoritmo, é preciso estabelecer uma convenção. Note que há uma grande quantidade de fatores envolvidos na execução de um programa em um computador que influenciam no seu tempo de execução. Tais fatores podem ser a CPU, o SO, a concorrência por recursos entre os processos do SO, etc. Análise de algoritmos despreza todos estes detalhes e adota uma convenção onde operações simples, como atribuições, operações aritméticas e algumas funções, têm custo 1. Este custo é adimensional. Ou seja, não medimos custo de tempo em termos de segundos, mas em termos de número de operações.

Estabelecida a convenção de custo temporal, podemos agora examinar um bom método para se medir a complexidade de tempo de um algoritmo. Tal método não é um algoritmo, mas é um bom ponto de partida. Como dito anteriormente, não existe um algoritmo para calcular a complexidade de outro algoritmo.

Primeiro, rotulamos cada linha do algoritmo com um custo. Para uma linha em que há um laço de repetição, o custo é o número de iterações deste laço em função de algum valor, como o tamanho da entrada n, por exemplo. Para uma linha em que há uma chamada de função, o custo é o custo da função, que deve ser previamente calculado. Para funções recursivas, obtemos uma relação de recorrência e calculamos seu valor assintótico. Para qualquer outra linha, o custo é 1. Em seguida, calculamos o custo total de cada laço de repetição e atualizamos os rótulos. Para isto funcionar, devemos começar dos laços mais internos. O custo total de um laço é o custo rotulado em sua linha multiplicado pelo somatório dos custos rotulados nas linhas internas ao laço. Finalmente, o custo total do algoritmo é o somatório dos custos de cada linha que não seja interna a um laço de repetição. Com o custo total calculado, podemos encontrar a complexidade assintótica do algoritmo.

Começamos analisando a complexidade de tempo do Algoritmo 1.2.1. Primeiro, rotulamos cada linha. As linhas 1, 3 e 4 são rotuladas com 1, enquanto a linha 2 é rotulada com n. O custo total do laço de repetição é n. Logo, o custo total do algoritmo é $n + 3 = \Theta(n)$. Portanto, o Algoritmo 1.2.1 executa em tempo linear.

Analisamos agora o Algoritmo 1.2.2. Primeiro, calculamos o custo da função FOO(n). Vamos assumir que o custo de sortear valores para uma matriz $n \times n$ seja n^2 e que o custo para calcular o determinante seja n^3. Assim, a linha 10 tem custo n^2 e a linha 11 tem custo n^3. As outras linhas da função têm custo 1. Logo, a função é $n^3 + n^2 + 3 = \Theta(n^3)$. Assim, a linha 2 tem custo n e a linha 3 tem custo n^3, enquanto as outras linhas têm custo 1. Note que agora é necessário avaliar os casos do algoritmo, enquanto que a análise do Algoritmo 1.2.1 dispensa esta preocupação. No melhor caso, a linha 4 nunca é executada. Então o custo total do bloco condicional é $n^3 + 1$ e o custo total do laço é $n(n^3 + 1) = \Theta(n^4)$. Logo, o algoritmo é $\Omega(n^4)$. No pior caso, a linha 4 sempre é executada. Então o custo total do bloco condicional é $n^3 + 2$ e o custo total do laço é $n(n^3 + 2) = \Theta(n^4)$. Logo, o algoritmo é $O(n^4)$. Como o algoritmo é $\Omega(n^4)$ e $O(n^4)$, então o algoritmo na verdade é $\Theta(n^4)$.

Para a análise do Algoritmo 1.2.3, observe que basta calcular o custo $T(i)$ da função FOO(i). No melhor caso, quando $i \leq 0$, as linhas 4 e 5 não executam e o custo total da função é $T(i) = 2 = \Theta(1)$. Logo, o algoritmo é $\Omega(1)$. No pior caso, as linhas 4 e 5 executam e o custo total da função é $T(i) = T(i-1)+3$. Abrindo a relação de recorrência, temos $T(i) = T(i-1)+3 = T(0)+\sum_{j=1}^{i} 3 = 2+3i = \Theta(i)$. Como vimos anteriormente, $i = 2^n$. Logo, o algoritmo é $O(2^n)$.

Por último, analisamos o Algoritmo 1.2.4. Note que a função FOO(n) é exatamente a mesma função do Algoritmo 1.2.2. Logo, a linha 8 tem custo n^3, enquanto a linha 10 tem custo 1. No melhor caso, a linha 10 é executada no lugar da linha 8. Somando os custos totais, temos $n + 7 = \Theta(n)$. Logo, o algoritmo é $\Omega(n)$. No pior caso, a linha 8 é que executa e o custo total do algoritmo é $n + 5 + n^3 = \Theta(n^3)$. Logo, o algoritmo é $O(n^3)$.

1.3 Juízes eletrônicos

Juízes eletrônicos são a ferramenta mais importante para o treino de um competidor. Um juíz

eletrônico (abreviamos com OJ, do inglês *online judge*) possui um grande banco de problemas, que são, dependendo do juíz, bem classificados em relação aos assuntos que envolvem suas possíveis soluções. Um usuário pode submeter uma solução para um problema e o juíz deve julgar esta solução. Para realizar esta tarefa, o juíz possui um conjunto de entradas e de saídas esperadas. O código submetido é executado e alimentado com estas entradas. A saída deve ser idêntica à saída esperada para que o juíz aceite a solução. Caso contrário, o juíz dá um veredito de erro. Os vereditos utilizados por juízes eletrônicos seguem o padrão das competições organizadas pela ACM e estão listados a seguir.

- CE: *Compile Error*, código não compila.
- RTE: *Runtime Error*, falha de segmentação, estouro de pilha ou limite de memória excedido.
- TLE: *Time Limit Exceeded*, limite de tempo excedido.
- WA: *Wrong Answer*, solução incorreta.
- PE: *Presentation Error*, solução correta, mas com erro de apresentação. Faltou/sobrou algum espaço, quebra de linha... É comum iniciantes obterem este erro.
- AC: *Accepted*, solução e saída corretas.

Geralmente, o maior adversário de um competidor é o veredito TLE. Para vencê-lo, é preciso projetar um algoritmo com complexidade de tempo adequada, tendo em mente que uma CPU típica de 2013 é capaz de processar 10^8 operações em poucos segundos [1].

1.3.1 O formato de um problema

A formatação de problemas em juízes eletrônicos também segue o padrão das competições organizadas pela ACM. A primeira seção descreve o problema em si. Em seguida, há uma seção que descreve como a entrada é dada e quais são seus limites de tamanho. A seção seguinte descreve o formato da saída. Por último, há um exemplo de entrada e saída esperada. Observe que não é necessário testar o formato da entrada, nem validar dados. Pode-se assumir que a entrada sempre segue o formato e as regras especificadas. Por exemplo, no problema da Figura 1.3.1 não é preciso testar se n de fato é inteiro, se $1 > n$, ou se $n > 10^6$.

1.4 Entrada e saída

É muito comum obter WA ou PE por não ler a entrada corretamente, ou não imprimir a saída corretamente, ainda que a solução para o problema esteja certa. Esta seção abrange os formatos de entrada e saída – E/S – mais comuns e as melhores maneiras para tratá-los.

1.4.1 A linguagem de programação C++

A linguagem de programação C++ é a mais utilizada por competidores, pelas seguintes razões:

- Alta facilidade para lidar com E/S (entrada e saída).
- Alta capacidade de escrita (por exemplo, C++ permite sobrecarga de operadores).
- As estruturas de dados da biblioteca padrão possuem nomes curtos e são fáceis de usar.
- C++ é uma extensão de C. Logo, um programador de C não precisa conhecer orientação a objetos profundamente para utilizar as vantagens de C++ em um torneio. Em outras palavras, é fácil *usar* C++ sem *conhecer* C++.

Somatório

Dado um número n, você deve calcular a soma $S(n) = 1 + 2 + ... + n$.

Entrada

A entrada contém vários casos de teste. Cada caso de teste é composto por uma única linha com um inteiro n, $1 \leq n \leq 10^6$.

Saída

A saída de um caso de teste deve ser uma única com valor de $S(n)$.

Entrada	Saída
3	6
5	15

Figura 1.3.1: Exemplo de problema.

```
int i1, i2, i3;
char c;
scanf("%d %c %d %d\n", &i1, &c, &i2, &i3);
```

Figura 1.4.1: Código C++ para ler uma linha com vários campos.

Pelas razões supracitadas, introduzimos técnicas de E/S com a linguagem C++.

As melhores páginas para consulta, tanto de características da linguagem (tutorial da linguagem, etc.), quanto de documentação da biblioteca padrão (do C e do C++), são:

- `http://www.cplusplus.com/`
- `http://en.cppreference.com/`

1.4.2 As funções scanf() e printf()

A função scanf() é capaz de ler dados da entrada padrão seguindo um formato complexo. O formato aceito pelo scanf() é quase tão poderoso quanto uma expressão regular (ferramenta utilizada por analisadores de texto para verificar a forma de uma cadeia de caracteres). Alguns exemplos de utilização do scanf() são dados a seguir.

- A primeira linha de um caso de teste é composto por um inteiro, uma letra e dois inteiros. Os campos são separados por espaço. A Figura 1.4.1 mostra o código para ler tal linha utilizando apenas um comando. Cada parte da cadeia de caracteres que descreve o formato irá casar com entrada. Isto significa que os espaços entre os campos e a quebra de linha serão retirados do *buffer* de entrada apropriadamente.
- Suponha que a entrada é uma linha com um CPF no formato XXX.XXX.XXX-XX e deve ser feita uma checagem de validade do CPF que utiliza cada um dos quatro campos numéricos. A Figura 1.4.2 mostra o código apropriado para realizar a leitura. Os caracteres ponto, hífen e quebra de linha casam com a entrada e os campos numéricos são lidos apropriadamente.

```
int a, b, c, d;
scanf("%d.%d.%d-%d\n", &a, &b, &c, &d);
```

Figura 1.4.2: Código C++ para ler um CPF no formato XXX.XXX.XXX-XX.

```
int T;
scanf("%d", &T);
for (int t = 1; t <= T; t++) {
  // aqui, ler um caso de teste conforme a especificacao
}
```

Figura 1.4.3: Código C++ para ler uma entrada onde o número de casos de teste é dado.

Uma característica extremamente útil do scanf() é o seu retorno. Esta função retorna o número de campos lidos com sucesso. Exemplos desta característica são dados na seção a seguir.

A função printf() é a contrapartida do scanf() para impressão na saída padrão. Um exemplo de sua utilização é dado na Seção 1.4.4.

As funções scanf() e printf() são originais da biblioteca padrão do C e para utilizá-las é necessário incluir a linha `#include <cstdio>` no início do código fonte.

1.4.3 Lendo casos de teste

A primeira preocupação ao escrever código para ler entrada é: quantos casos de teste há na entrada? A seguir estão listadas as quatro possibilidades.

- Um único caso de teste por entrada. Neste caso, basta ler a entrada como especificado na descrição do problema.
- A entrada é iniciada com uma linha com um único número inteiro que indica a quantidade de casos de teste a serem lidos. A Figura 1.4.3 mostra o código adequado para este caso.
- Há vários casos de teste e o último é iniciado por um campo cujo valor indica terminação. A Figura 1.4.4 mostra o código adequado para este caso.
- Há vários casos de teste e o programa deve lê-los até a entrada terminar. A Figura 1.4.5 mostra o código adequado para este caso.

Note que nos códigos das Figuras 1.4.4 e 1.4.5 compara-se o retorno do scanf() com 2. Isto é feito para verificar se o scanf() de fato conseguiu ler os seus dois campos. Se o número de campos lidos é diferente de 2, é porque a entrada terminou. Observe que este teste é só é necessário no caso em que casos de testes devem ser lidos até a entrada terminar. No caso em que a entrada, sim, indica fim de entrada, o teste só é feito para que a chamada ao scanf() possa ser colocada consistentemente na própria estrutura do laço `for`, pois a condição que interrompe o laço é a seguinte (a que testa o valor do campo lido).

```
// suponha que cada caso de teste inicie com uma linha com dois inteiros X e Y.
// o ultimo caso de teste eh iniciado com X = 0 e Y = 0 e nao deve ser processado.
for (int X, Y; scanf("%d %d", &X, &Y) == 2 && (X || Y);) {
  // aqui, ler o restante do caso de teste conforme a especificacao
}
```

Figura 1.4.4: Código C++ para ler uma entrada onde o último caso de teste é indicado por um valor especial.

```
// suponha que cada caso de teste inicie com uma linha com dois inteiros X e Y.
for (int X, Y; scanf("%d %d", &X, &Y) == 2;) {
  // aqui, ler o restante do caso de teste conforme a especificacao
}
```

Figura 1.4.5: Código C++ para ler casos de teste até a entrada terminar.

```
double X;           // sempre utilizar precisao dupla
scanf("%lf", &X);   // lendo ponto flutuante
printf("%.2lf", X); // imprimindo ponto flutuante
```

Figura 1.4.6: Código C++ para ler e imprimir ponto flutuante.

1.4.4 Ponto flutuante

A memória de um computador é finita. Por esta razão, qualquer representação numérica computacional só é capaz de representar uma quantidade finita de números. Mais que isso, representações computacionais só são capazes de representar alguns números racionais.

Para lidar com números não-inteiros, um computador converte uma representação em uma base qualquer para uma representação binária. A representação binária mais utilizada hoje é o *ponto flutuante* (o padrão permite precisão simples e precisão dupla, onde a precisão dupla utiliza o dobro de *bits* da precisão simples). Durante uma conversão como esta, e durante qualquer outra operação de ponto flutuante, pode ocorrer *perda de precisão*. Perda de precisão é o erro em que a representação resultante de alguma operação não representa o número exato que deveria. Por exemplo: $1/3 = 0.33333333...$ Mas um computador pode ser capaz de representar no máximo, digamos, 0.3333 (que é bem diferente da dízima 0.33333333...).

Um bom exemplo de como perda de precisão pode ser um grande problema é: imagine que você encontrou duas fórmulas distintas, mas equivalentes, para uma função qualquer. Por exemplo: $f(a, b, c) = a(b + c) = ab + ac$. Por conta da perda de precisão, o resultado de calcular $a(b + c)$ (somando primeiro) pode ser diferente de calcular $ab + ac$ (multiplicando primeiro). Então, como você sabe qual deve utilizar? Isto geralmente ocorre em problemas de teoria da probabilidade, onde é comum haverem diferentes abordagens para calcular uma mesma probabilidade.

Por esta razão, problemas envolvendo números não-inteiros geralmente são minoria em uma competição. No entanto, existe uma regra geral para lidar com tais números. *Sempre utilizar precisão dupla.* Seja para ler, calcular, ou imprimir. A Figura 1.4.6 mostra um exemplo de leitura e impressão de ponto flutuante. O número 2, no formato `"%.2lf"`, indica que a impressão deve ser feita com duas casas decimais.

Observe que perda de precisão também pode atrapalhar comparações. Portanto, é comum utilizar um pequeno valor positivo ϵ para determinar, por exemplo, se $x < y$. Ou seja, $x < y$ se, e somente se, $(x + \epsilon < y)$ e $(x < y - \epsilon)$. Além disso, observe que não existe uma regra geral para o valor de ϵ. Problemas envolvendo ponto flutuante geralmente determinam a precisão que deve ser utilizada na saída e/ou nos cálculos intermediários, mas é responsabilidade do competidor/time determinar os valores adequados para o programa.

Com todos os cuidados que tomamos, problemas de ponto flutuante ainda podem ser bastante injustos, pois nunca temos como saber qual o ϵ exato utilizado no programa gabarito, ou em quais comparações do gabarito o autor lembrou de comparar utilizando ϵ, ou até mesmo se o autor lembrou de utilizar precisão dupla (neste caso, é possível que uma resposta mais acurada resulte em WA!). Resumindo, a chance de obter o veredito AC na primeira tentativa é baixa.

```
// fora da main()
#include <iostream>  // necessario para o tipo std::string e objeto std::cin
using namespace std; // necessario para usar a biblioteca padrao do C++

// dentro da main()
string nome;
cin >> nome;
```

Figura 1.4.7: Código C++ para ler uma cadeia de caracteres.

```
// fora da main()
#include <iostream>  // necessario para o tipo std::string e objeto std::cin
using namespace std; // necessario para usar a biblioteca padrao do C++

// dentro da main()
string nome;
while (cin >> nome) {
  // aqui, realizar a leitura do restante do caso de teste
}
```

Figura 1.4.8: Código C++ para ler vários casos de teste que iniciam com uma cadeia de caracteres.

1.4.5 Lendo cadeias de caracteres

É possível utilizar a função scanf() para ler cadeias de caracteres. No entanto, C++ possibilita uma maneira mais fácil de realizar esta tarefa.

Através do tipo `string` e do objeto `cin` é possível ler cadeias de tamanhos arbitrários. A Figura 1.4.7 mostra como.

Uma propriedade útil da leitura de cadeias atráves do objeto `cin` é o seu retorno. Suponha que cada caso de teste é iniciado por uma linha com uma cadeia de caracteres que indica o nome de uma pessoa. Devem ser lidos casos de teste até a entrada terminar. Neste caso, é possível realizar a leitura dos casos de teste da maneira mostrada na Figura 1.4.8. O laço de repetição será interrompido se a entrada terminar e uma cadeia não puder ser lida.

Para usar as cadeias de caracteres de C++, procurar nas referências por `std::string`.

1.5 Estruturas de dados

Uma *estrutura de dados abstrata* é um conjunto de operações e regras que definem como um conjunto de dados é acessado e modificado. Os custos das operações podem ser parcialmente definidos.

Uma *estrutura de dados* – ED – é uma estrutura de dados abstrata em que o custo de espaço de armazenamento dos dados, os custos das operações de acesso e modificação e a maneira como os dados são organizados são completamente definidos.

Saber escolher a estrutura de dados ideal no projeto de um algoritmo é uma habilidade que deve ser dominada por um competidor. A seguir vemos algumas das estruturas de dados mais genéricas e de frequente uso no projeto de algoritmos. Tais estruturas são também implementadas nas bibliotecas padrão da maioria das linguagens de programação permitidas em competições.

1.5.1 Arranjos

Um *arranjo m-dimensional com dimensões* $d_1, d_2, ..., d_m$, em inglês *array*, é uma estrutura de dados que associa um dado a cada m-upla do conjunto $\{1, 2, ..., d_1\} \times \{1, 2, ..., d_2\} \times ... \times \{1, 2, ..., d_m\}$. Os dados são alocados de maneira contígua, de modo que o acesso e a modificação tenham custo de tempo $\Theta(1)$. A quantidade total de dados é $n = d_1 d_2 ... d_m$ e o custo total de espaço é $\Theta(n)$. Uma

Tabela 1.5.1: Operações com arranjos.

Operação	Custo de tempo
RECUPERAR$(a, i_1, i_2, ..., i_m) : x$	$\Theta(1)$
ATUALIZAR$(a, i_1, i_2, ..., i_m, x)$	$\Theta(1)$

vez que um arranjo é criado, não é possível inserir ou remover dados, modificando a quantidade total de dados armazenados. A Tabela 1.5.1 mostra as operações e custos para manipular um arranjo. A operação RECUPERAR() recebe um arranjo m-dimensional a, uma m-upla $(i_1, i_2, ..., i_m)$ e retorna o dado associado à m-upla de entrada. A operação ATUALIZAR() recebe um arranjo m-dimensional a, uma m-upla $(i_1, i_2, ..., i_m)$, um dado x e atualiza o valor do dado associado à m-upla de entrada com o valor x.

Geralmente, arranjos são implementados diretamente na linguagem de programção. Ou seja, não é necessário uma biblioteca para lidar com arranjos. Este é o caso, por exemplo, de C e C++. Por esta razão, utilizamos a notação $a[i_1][i_2]...[i_m]$ para indicar o dado associado a $(i_1, i_2, ..., i_m)$ no arranjo a em nossos algoritmos.

1.5.2 Vetores

Um *vetor*, em inglês *vector*, é um arranjo unidimensional. Se um vetor está ordenado, é possível realizar uma busca por um dado x em $O(\lg n)$ (busca binária, Seção 2.2.4).

Em algumas linguagens de programação, como em C++, a biblioteca padrão implementa um *vetor dinâmico* em que é possível inserir novos elementos em posições livres do final do vetor. Caso não haja posições livres, o vetor é realocado com o dobro do tamanho atual. Esta é uma excelente alternativa à listas encadeadas (a estrutura de dados descrita na seção a seguir), *caso inserção apenas no final seja necessária*. O que ocorre é que listas encadeadas alocam elementos individualmente, enquanto vetores devem ser completamente contíguos. Alocar/desalocar elementos individualmente implica fazer alocações/desalocações dinâmicas com grande frequência, o que por sua vez gera grande fragmentação da memória. Logo, listas encadeadas podem ser bem mais lentas do que vetores dinâmicos. Mais precisamente, inserir n elementos no fim de um vetor dinâmico pede no máximo $O(\lg n)$ realocações, enquanto inserir n elementos no fim de uma lista encadeada pede no máximo $O(n)$ alocações. Uma outra vantagem de vetores à listas é o percorrimento. Percorrimento em vetores tira vantagem de localidade de referência![1]

Para usar os vetores de C++, procurar nas referências por `std::vector`.

1.5.3 Listas encadeadas

Uma *lista encadeada*, ou simplesmente *lista*, em inglês *linked list*, ou *list*, é uma estrutura de dados sequencial que associa um dado a cada iterador de um conjunto de iteradores encadeados. Uma lista é descrita pelo seu iterador inicial. Cada iterador dá acesso ao próximo, de modo que uma busca precisa passar sequencialmente por uma parte dos dados até encontrar o dado desejado, ou seja, é feita em $O(n)$. A alocação é dinâmica, de modo que dados podem ser inseridos ou removidos após a criação de uma lista. Se há n dados armazenados em uma lista, seu custo de espaço é $\Theta(n)$. A Tabela 1.5.2 mostra as operações e custos para manipular uma lista encadeada. A operação RECUPERAR() recebe um iterador i e retorna o dado associado a i. A operação ATUALIZAR() recebe um iterador i, um dado x e atualiza o valor do dado associado a i para x. A operação INSERIR() recebe um iterador i, um dado x, insere o dado x no iterador i, aumentando o espaço utilizado pela lista em $\Theta(1)$, e ajusta os iteradores da lista para ficarem consistentes com esta inserção. A operação REMOVER() recebe um iterador i, remove o iterador i, diminuindo o espaço utilizado pela lista em $\Theta(1)$, e ajusta os

[1] Quanto mais próximos dois dados estão na memória, mais rápido (em média) o *hardware* consegue acessá-los, se um for acessado após o outro.

Tabela 1.5.2: Operações com listas.

Operação	Custo de tempo
RECUPERAR(i) : x	$\Theta(1)$
ATUALIZAR(i, x)	$\Theta(1)$
INSERIR(i, x)	$\Theta(1)$
REMOVER(i)	$\Theta(1)$
PRÓXIMO(i) : i	$\Theta(1)$

Tabela 1.5.3: Operações com pilhas.

Operação	Custo de tempo (vetor)	Custo de tempo (lista)
RECUPERAR-TOPO(p) : x	$\Theta(1)$	$\Theta(1)$
EMPILHAR(p, x)	$\Theta(1)$ no caso médio e $O(n)$	$\Theta(1)$
DESEMPILHAR(p)	$\Theta(1)$	$\Theta(1)$
VAZIA(p) : b	$\Theta(1)$	$\Theta(1)$

iteradores da lista para ficarem consistentes com esta remoção. A operação PRÓXIMO() recebe um iterador i e retorna o iterador seguinte.

Para usar as listas de C++, procurar nas referências por `std::list`.

1.5.4 Pilhas

Uma *pilha*, em inglês *stack*, é uma estrutura de dados abstrata com a política "o último que entra é o primeiro que sai". Geralmente, pilhas são implementadas com vetores com inserção ou listas. A Tabela 1.5.3 mostra as operações e custos para manipular uma pilha. A operação RECUPERAR-TOPO() recebe uma pilha p e retorna o último dado inserido em p. A operação EMPILHAR() recebe uma pilha p, um dado x e insere x em p. A operação DESEMPILHAR() recebe uma pilha p e remove o último dado inserido em p. A operação VAZIA() recebe uma pilha p e retorna um valor booleano b que indica se p está vazia.

Para usar as pilhas de C++, procurar nas referências por `std::stack`.

1.5.5 Filas

Uma *fila*, em inglês *queue*, é uma estrutura de dados abstrata com a política "o primeiro que entra é o primeiro que sai". Ou seja, é exatamente o oposto de uma pilha. A Tabela 1.5.4 mostra as operações e custos para manipular uma fila. A operação RECUPERAR-FRENTE() recebe uma fila f e retorna o primeiro dado inserido em f. A operação ENFILEIRAR() recebe uma fila f, um dado x e insere x em f. A operação DESENFILEIRAR() recebe uma fila f e remove o primeiro dado inserido em f. A operação VAZIA() recebe uma fila f e retorna um valor booleano b que indica se f está vazia.

Para usar as filas de C++, procurar nas referências por `std::queue`.

Tabela 1.5.4: Operações com filas.

Operação	Custo de tempo (vetor)	Custo de tempo (lista)
RECUPERAR-FRENTE(f) : x	$\Theta(1)$	$\Theta(1)$
ENFILEIRAR(f, x)	$\Theta(1)$ no caso médio e $O(n)$	$\Theta(1)$
DESENFILEIRAR(f)	$\Theta(1)$	$\Theta(1)$
VAZIA(f) : b	$\Theta(1)$	$\Theta(1)$

Tabela 1.5.5: Operações com filas de prioridade.

Operação	Custo de tempo (vetor)	Custo de tempo (árvore)
RECUPERAR-TOPO(f) : x	$\Theta(1)$	$\Theta(1)$
ENFILEIRAR(f, x)	$O(\lg n)$ no caso médio e $O(n)$	$O(\lg n)$
DESENFILEIRAR(f)	$O(\lg n)$	$O(\lg n)$
VAZIA(f) : b	$\Theta(1)$	$\Theta(1)$

Tabela 1.5.6: Operações com conjuntos.

Operação	Custo de tempo (*hash*)	Custo de tempo (árvore)
BUSCAR(c, x) : b	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$
INSERIR(c, x)	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$
REMOVER(c, x)	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$

1.5.6 Filas de prioridade

Uma *fila de prioridade*, em inglês *priority queue*, é uma estrutura de dados abstrata com a política "o de maior prioridade é o primeiro que sai". Geralmente, filas de prioridade são implementadas com vetores com inserção ou árvores de busca binária balanceadas. Em um fila de prioridade, os dados inseridos precisam possuir ordem ($x \neq y \implies x < y$ ou $x > y$). A Tabela 1.5.5 mostra as operações e custos para manipular uma fila de prioridade. A operação RECUPERAR-FRENTE() recebe uma fila de prioridade f e retorna o maior dado inserido em f. A operação ENFILEIRAR() recebe uma fila de prioridade f, um dado x e insere x em f. A operação DESENFILEIRAR() recebe uma fila de prioridade f e remove o maior dado inserido em f. A operação VAZIA() recebe uma fila de prioridade f e retorna um valor booleano b que indica se f está vazia.

Uma árvore binária com n dados utiliza espaço $\Theta(n)$.

Para usar as filas de prioridade de C++, procurar nas referências por `std::priority_queue`.

1.5.7 Conjuntos

Um *conjunto*, em inglês *set*, é uma estrutura de dados abstrata que deve funcionar exatamente como um conjunto matemático. Ou seja, é uma coleção de dados distintos. Geralmente, conjuntos são implementados com tabelas de dispersão (tabelas *hash*) ou árvores de busca binária balanceadas. Quando árvores são usadas, é necessário que o tipo do dado possua ordem. A Tabela 1.5.6 mostra as operações e custos para manipular um conjunto. A operação BUSCAR() recebe um conjunto c, um dado x e retorna um valor booleano b que indica se x está em c. A operação INSERIR() recebe um conjunto c, um dado x e, se $x \notin c$, insere x em c. A operação REMOVER() recebe um conjunto c, um dado x e remove x de c.

Uma tabela *hash* de k posições com n dados utiliza espaço $O(k + n)$.

Para usar os conjuntos de C++, procurar nas referências por `std::set`.

1.5.8 Mapas

Um *mapa*, em inglês *map*, é uma estrutura de dados abstrata que deve funcionar exatamenta como uma função matemática. Um mapa associa um dado *valor* a um dado *chave*. Geralmente, mapas são implementados com tabelas de dispersão (tabelas *hash*) ou árvores de busca binária balanceadas. Quando árvores são usadas, é necessário que o tipo do dado chave possua ordem. A Tabela 1.5.7 mostra as operações e custos para manipular um mapa. A operação RECUPERAR() recebe um mapa m, uma dado chave k, realiza a busca do dado valor x associado a k e retorna x. A operação INSERIR() recebe um mapa m, um dado chave k, um dado valor x e insere (ou atualiza) o mapeamento $k \to x$

Tabela 1.5.7: Operações com mapas.

Operação	Custo de tempo (*hash*)	Custo de tempo (árvore)
RECUPERAR$(m, k) : x$	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$
INSERIR(m, k, x)	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$
REMOVER(m, k)	$\Theta(1)$ no caso médio e $O(n)$	$O(\lg n)$

Tabela 1.5.8: Operações com *bits*.

a	b	a OR b (ou $a \mid b$)	a AND b (ou a & b)	NOT a (ou $\sim a$)	a XOR b (ou $a \hat{} b$)
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	0	1
1	1	1	1	0	0

em m. A operação REMOVER() recebe um mapa m, um dado chave k, busca o mapeamento $k \to x$ e o remove.

Para usar os mapas de C++, procurar nas referências por `std::map`.

1.5.9 Máscaras de *bits*

Sabemos que as variáveis de um programa são, na prática, sequências de *bits*. As operações que podemos fazer com *bits* são ilustradas na Tabela 1.5.8.

Geralmente, uma variável tem 8, 16, 32, ou 64 *bits*. A Tabela 1.5.9 ilustra como representamos os *bits* de uma váriavel a de 8 *bits* que contém o valor decimal 171. O *bit* a_7 é o *mais significativo* e o *bit* a_0 é o *menos significativo*. Observe que $\sum_{i=0}^{7} a_i 2^i = 171$. Se a variável a possui *sinal*, seu valor decimal é $-a_7 2^7 + \sum_{i=0}^{6} a_i 2^i = -85$.

As operações da Tabela 1.5.8 são feitas *bit* a *bit* em variáveis. Em outras palavras, sejam duas variáveis de n *bits* $a = a_{n-1}...a_1a_0$ e $b = b_{n-1}...b_1b_0$. Temos que $a * b = (a_{n-1} * b_{n-1})...(a_1 * b_1)(a_0 * b_0)$, onde $*$ é uma das operações da Tabela 1.5.8. O custo de calcular $a * b$ é $\Theta(1)$.

Além de poder realizar as operações básicas da Tabela 1.5.8 em variáveis, também podemos realizar deslocamentos. Seja $a = a_{n-1}...a_1a_0$ uma variável de n *bits*. A seguir mostramos como são feitos os deslocamentos de k *bits* da variável a.

- Um *deslocamento à esquerda*, escrito $a \ll k$, é o valor $a_{n-k-1}...a_1a_000...0$, onde há k zeros à direita de a_0. Observe que esta operação é o mesmo que multiplicar o valor decimal de a por 2^k. Por exemplo, $00001111 \ll 2 = 00111100$. Temos que o valor decimal de 00001111 é 15 e o valor decimal de 00111100 é $60 = 15 \cdot 2^2$. Se fazemos $a \ll 1$, com o valor de a da Tabela 1.5.9, obtemos 01010110, cujo valor decimal é $86 \neq 171 \cdot 2^1$. Isto ocorre, porque o *bit* mais significativo de a é perdido no deslocamento à esquerda.

- Um *deslocamento à direita*, escrito $a \gg k$, é o valor $a_{n-1}a_{n-1}...a_{n-1}a_{n-2}...a_{k+1}a_k$, onde há $k + 1$ ocorrências de a_{n-1} à esquerda de a_{n-2}. Observe que esta operação é o mesmo que dividir o valor decimal de a por 2^k e obter a parte inteira, somente se $a_{n-1} = 0$. Por exemplo, $00000111 \gg 1 = 00000011$ e temos que o valor decimal de 00000111 é 7 e o valor decimal de 00000011 é $3 = \lfloor 7/2^1 \rfloor$. Em variáveis sem sinal, as k primeiras ocorrências de a_{n-1} são *sempre*

Tabela 1.5.9: Representação dos *bits* de uma variável a de 8 *bits*.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
1	0	1	0	1	0	1	1

substituídas por zeros, de modo que deslocamentos à direita são *sempre* divisões por potências de 2. Este é o caso de $a \gg 3 = 00010101$, se usamos o valor de a da Tabela 1.5.9 e consideramos que a não tem sinal. Observe que o valor decimal de 00010101 é $21 = \lfloor 171/2^3 \rfloor$.

Agora estamos prontos para mostrar como acessar e modificar os *bits* de uma variável diretamente. Seja $a = a_{n-1}...a_1a_0$ uma variável de n *bits*. O *bit* a_i vale 1 se, e somente se, o valor decimal de $(a \gg i)$ & 1 é diferente de 0, com $i \in \{0, 1, ..., n-1\}$. Observe, também, que o valor decimal de a é par se, e somente se, $a_0 = 0$, ou seja, se o valor decimal de a & 1 é 0. Para fazer $b = b_{n-1}...b_1b_0$, onde $b_i = 1$ e $b_j = a_j$, para todo $j \neq i$ e algum $i \in \{0, 1, ..., n-1\}$, fazemos $b = (a \mid (1 \ll i))$. E, por fim, para fazer $b = b_{n-1}...b_1b_0$, onde $b_i = 0$ e $b_j = a_j$, para todo $j \neq i$ e algum $i \in \{0, 1, ..., n-1\}$, fazemos $b = (a \,\&\, (\sim (1 \ll i)))$.

Máscaras de *bits*

Podemos estender as operações de acesso e modificação aos *bits* de uma variável para um criar uma estrutura de dados extremamente útil, simples e eficiente.

Muitos algoritmos podem se beneficiar de representar seus dados através dos *bits* de uma única variável a. O que fazemos é atribuir significados para cada parte da sequência de *bits* de a. Por exemplo, imagine um problema onde são dadas quantidades iniciais de um cartão vermelho e de um cartão azul. Se tais quantidades não excedem, por exemplo, $2^7 - 1$, a pode ser uma variável de 16 *bits*. Basta utilizar os *bits* de a_0 a a_6 para uma quantidade e os *bits* de a_7 a a_{13} para a outra quantidade. O Capítulo 2 contém exemplos de problemas que podem se beneficiar desta representação de dados, que chamamos de *máscara de bits* (*bitmask* em inglês).

Antes de detalhar esta ED, precisamos definir uma notação. Assim como é comum em diversas linguagens de programação, iniciaremos um número escrito na base hexadecimal com o prefixo $0x$.

Para exemplificar as operações com máscaras de *bits*, vamos considerar uma variável a de 8 *bits*, onde atribuímos um significado para os *bits* de a_0 a a_2 e um significado para os *bits* de a_3 a a_6. A seguir mostramos como acessar e modificar os valores dos campos $c_1 = a_2a_1a_0$ e $c_2 = a_6a_5a_4a_3$.

- Para obter o valor de c_1, usamos a expressão a & $0x7$. Observe que $0x7$ pode ser escrito em binário como 00000111. Neste caso, o que fazemos é "apagar" os *bits* de a que não fazem parte do campo c_1.

- Para obter o valor de c_2, usamos a expressão $(a \gg 3)$ & $0xF$. Observe que $0xF$ pode ser escrito em binário como 00001111. Neste caso, deslocamos a à direita de 3 posições e "apagamos" o restante dos *bits*, que não fazem parte do campo c_2.

- Para atualizar o valor do campo c_1, atualizamos o valor de a com $(a \,\&\, (\sim 0x7)) \mid c_1$. O lado esquerdo da operação OR "apaga" o conteúdo atual do campo que vamos atualizar, enquanto o lado direito "escreve" o novo valor no local adequado. Observe que $\sim 0x7$ é o mesmo que 11111000.

- Para atualizar o valor do campo c_2, atualizamos o valor de a com a expressão $(a \,\&\, (\sim (0xF \ll 3))) \mid (c_2 \ll 3)$. Novamente, o lado esquerdo da operação OR "apaga" o conteúdo atual do campo a ser atualizado. Desta vez, observe que precisamos deslocar o valor $0xF$ à esquerda, antes de negá-lo. O mesmo ocorre com o novo valor c_2 do campo. Observe que $\sim (0xF \ll 3)$ é o mesmo que 10000111.

O que os valores $0x7$, $0xF$, $\sim 0x7$ e $\sim (0xF \ll 3)$ têm em comum? Todos eles são exatamente o que chamamos de *máscaras de bits*. Aplicamos uma máscara através da operação AND para "apagar" os *bits* 0 e manter os valores dos *bits* onde a máscara vale 1. Para obter os valores dos campos c_1 e c_2, deslocamos o respectivo campo para a posição adequanda, no início da sequência de *bits*, e aplicamos uma máscara que apaga tudo, deixando só o valor do campo. Para atualizar os valores dos campos c_1 e c_2, aplicamos máscaras adequadas para apagar os valores atuais dos campos e escrevemos os novos valores nas respectivas posições, através da operação OR.

1.6 Trocando os valores de duas variáveis

O Algoritmo 1.6.1 utiliza a operação XOR, apresentada na Seção 1.5.9, para trocar os valores das variáveis a e b, sem utilizar uma variável auxiliar. Como exercício, tente provar que o algoritmo está correto!

Algoritmo 1.6.1 Algoritmo XOR para trocar os valores de duas variáveis sem utilizar variável auxiliar.

1: $a \leftarrow a \hat{} b$
2: $b \leftarrow a \hat{} b$
3: $a \leftarrow a \hat{} b$

Capítulo 2

Paradigmas de soluções de problemas

Este capítulo introduz técnicas de projeto de algoritmos extremamente abrangentes. Muitos algoritmos famosos se encaixam em uma ou mais das categorias aqui apresentadas. O algoritmo de Dijkstra para caminhos mínimos em grafos, por exemplo, apresentado na Seção 3.5.1, é um algoritmo guloso. Qualquer algoritmo para qualquer problema NP-completo é uma busca completa. O famoso algoritmo de ordenação *quick sort* é um algoritmo de divisão e conquista. Algoritmos de programação dinâmica são frequentemente utilizados em bioinformática. É fundamental para alguém que queira se sair bem em uma competição de programação conhecer bem todos estes paradigmas.

Começamos com o paradigma mais simples, popularmente conhecido como "força bruta".

2.1 Busca completa

Inúmeros problemas se resumem a examinar cada objeto de um dado conjunto. Chamamos este processo de *busca completa*, este conjunto de *espaço de busca* e cada objeto de *solução candidata*.

Seja S um espaço de busca. Problemas de busca completa geralmente se encaixam em alguma das seguintes situações:

- Dada uma propriedade $f : S \longrightarrow \{\mathrm{V}, \mathrm{F}\}$, listar/enumerar todas (ou simplesmente determinar se existe alguma) as soluções candidatas $x \in S$ tais que $f(x) = \mathrm{V}$. Tais soluções candidatas são chamadas simplesmente de *soluções*.

- Dada uma função $f : S \longrightarrow \mathbb{R}$, listar/enumerar todas (ou simplesmente encontrar o valor mínimo/máximo) as soluções candidatas $x \in S$ tais que o valor de $f(x)$ é mínimo/máximo. Tais soluções candidatas são chamadas de *soluções ótimas*. Problemas deste tipo são *problemas de otimização combinatória*.

Em qualquer situação, uma busca completa calcula uma função f para cada elemento de um espaço de busca S. Logo, a complexidade de tempo de uma busca completa é $\Theta(c_f|S|)$, onde c_f é o custo de calcular a função f e os custos c_f e $|S|$ podem depender do tamanho da entrada. Quando queremos apenas determinar se existe alguma solução, como na primeira situação, podemos interromper a busca assim que uma solução é encontrada. No melhor caso, a primeira solução candidata examinada é uma solução, de modo que uma busca completa é $\Omega(c_f)$. No pior caso, uma solução não existe e todo o conjunto S é examinado, de modo que uma busca completa é $O(c_f|S|)$.

Projetar uma busca completa se resume a identificar a forma do espaço de busca S, identificar a função f e bolar uma estratégia de atravessar S para calcular $f(x)$, para todo $x \in S$. Em grande

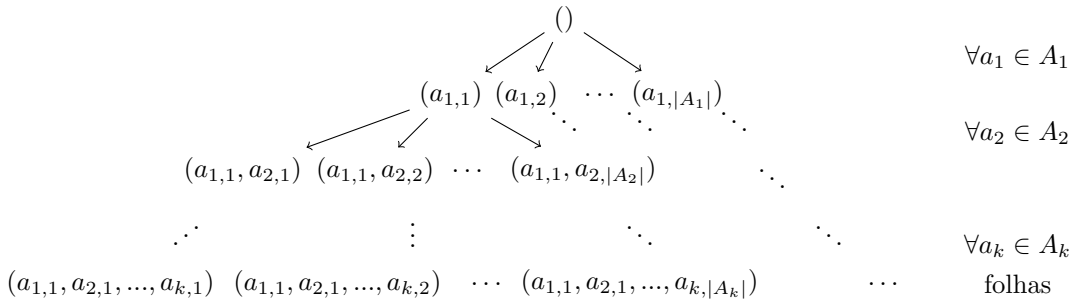

Figura 2.1.1: Árvore associada a uma busca recursiva. Denotamos o j-ésimo elemento do conjunto A_i por $a_{i,j}$.

parte dos problemas, um espaço de busca S é da forma

$$S = \{(a_1, a_2, ..., a_k) \mid a_i \in A_i\}$$

onde devemos determinar quais são os conjuntos A_i pela especificação do problema. No caso em que S assume esta forma, é comum construir soluções de maneira incremental, ou seja, *soluções parciais*, da forma $(a_1, a_2, ..., a_i)$, $i < k$, são incrementadas para se obter soluções candidatas. Neste caso, é fácil fazer uma busca recursiva, como discutimos a seguir.

2.1.1 Busca recursiva

Busca recursiva, conhecida em inglês como *backtracking*, é uma maneira de realizar uma busca completa utilizando recursividade.

Algoritmo 2.1.1 Forma geral de um *backtracking*.

```
1: BACKTRACK(0, ())                                          ▷ chamada inicial
2: função BACKTRACK(i : inteiro, x : solução parcial)
3:     se i = k então                                        ▷ neste caso, x é uma solução candidata
4:         calcule f(x) e faça alguma coisa com o resultado
5:     senão                                                 ▷ neste caso, x é uma solução parcial da forma (a_1, a_2, ..., a_i)
6:         para cada a ∈ A_{i+1} faça
7:             BACKTRACK(i + 1, (a_1, a_2, ..., a_i, a))
8:         fim para
9:     fim se
10: fim função
```

Em geral, um *backtracking* tem a forma do Algoritmo 2.1.1. Observe que a execução deste algoritmo pode ser imaginada como a árvore ilustrada na Figura 2.1.1. As chamadas à função BACKTRACK() são representadas pelos nós da árvore. Note que os nós do nível mais baixo representam soluções candidatas, enquanto todos os outros nós representam soluções parciais.

Para entender o Algoritmo 2.1.1, vamos fixar a como o primeiro elemento do conjunto A_1 que o algoritmo escolhe na linha 6. Observe que *todas* as soluções candidatas cujos primeiros elementos são a, são examinadas *antes* de se escolher outro elemento do conjunto A_1 para atribuir a a. Ou seja, na Figura 2.1.1, o algoritmo explora primeiro a subárvore com raiz $(a_{1,1})$, para então *retroceder* e explorar a subárvore com raiz $(a_{1,2})$. Este retrocesso (português para *backtracking*) ocorre sempre que o laço de repetição de alguma chamada à função BACKTRACK() termina e a respectiva chamada retorna. Esta busca também é conhecida como *busca em profundidade* (veja a Seção 3.3.1).

Agora, vamos analisar o Algoritmo 2.1.1 quando a chamada inicial é feita apropriadamente, ou seja, com $i = 0$. Denotando por $T(i)$ o custo de tempo do algoritmo para uma chamada com o valor i no parâmetro i da função BACKTRACK(), temos que $T(0) = |A_1|T(1) = |A_1||A_2|T(2) = ... = |A_1||A_2|...|A_k|T(k) = |S|c_f = O(c_f|S|)$.

2.1.2 Busca iterativa

Utilizamos busca recursiva porque é fácil de projetar e implementar. No entanto, algumas vezes nos deparamos com problemas de busca completa em que utilizar recursão é justamente a nossa ruína. Apesar do custo assintótico de uma *busca iterativa* ser o mesmo de um *backtracking*, busca iterativa não sofre do *overhead* causado por muitas chamadas a funções. Uma implementação iterativa pode chegar a ser 20 vezes mais rápida do que uma implementação recursiva, o que com certeza pode ser a diferença entre o AC e o TLE. Dependendo também do número de chamadas recursivas empilhadas de uma só vez, é possível receber o veredito RTE, por estouro de pilha. Também há casos em que a solução para um problema fica mais nítida se pensamos em uma iteração.

A desvantagem de busca iterativa é que não há um modelo de projeto simples para seguir, como o Algoritmo 2.1.1 para *backtracking*. Nesta seção, exemplificamos três tipos distintos de busca iterativa.

Implementação iterativa de busca recursiva

Algumas vezes, basta fazer uma implementação iterativa de uma busca recursiva para resolver um problema de busca completa. Estes são os casos em que o simples fator constante do *overhead* causado por muitas chamadas à funções é o grande problema da implementação.

Algoritmo 2.1.2 Forma geral de um *backtracking* iterativo.

```
1:  Seja a_i[1..|A_i|], onde a_i[j] ∈ A_i e a_i[j] ≠ a_i[l] se i ≠ l, ∀(i ∈ {1,2,...,k}, j,l ∈ {1,2,...,|A_i|}).
2:  Vamos construir x[1..k] como um vetor de índices para os vetores a_i, ou seja, a_i[x[i]] ∈ A_i.
3:  para i ← 1 até k faça
4:      x[i] ← 0
5:  fim para
6:  i ← 1
7:  enquanto 1 ≤ i ≤ k + 1 faça
8:      se i = k + 1 então                          ▷ neste momento, x é uma solução candidata
9:          calcule f(x) e faça alguma coisa com o resultado
10:     senão se x[i] < |A_i| então                 ▷ se ainda há possibilidades para x[i], testamos a próxima
11:         x[i] ← x[i] + 1
12:         i ← i + 1
13:     senão                                       ▷ se todas as possibilidades para x[i] já foram testadas, fazemos um retrocesso
14:         x[i] ← 0                                ▷ necessário para que outras tentativas em x[i] possam ser feitas
15:         i ← i − 1                               ▷ retrocesso
16:     fim se
17: fim enquanto
```

O Algoritmo 2.1.2 é um modelo para *backtrackings* iterativos. Observe que este algoritmo é genérico o bastante para lidar com conjuntos A_i quaisquer! Não é necessário, por exemplo, que A_i seja somente um conjunto de números. Esta flexibilidade está em criarmos vetores $a_i[1..|A_i|]$ para armazenar os elementos de A_i, $i \in \{1, 2, ..., k\}$, e construirmos uma solução candidata x como um vetor de índices para estes vetores. Ou seja, $x[i]$ é um índice para o vetor a_i, isto é, $a_i[x[i]] \in A_i$. No entanto, observe também que, na maioria dos problemas, os elementos de A_i são sim números inteiros. Nestes casos, podemos simplificar o Algoritmo 2.1.2 e fazer com que os elementos do vetor x sejam tirados diretamente dos conjuntos A_i. Fazemos isto removendo as linhas 1 e 2, transformando as linhas 4 e 15 em "$x[i] \leftarrow \min A_i - 1$" e a linha 10 em "**senão se** $x[i] < \max A_i$ **então**".

Note que o laço da linha 7 do Algoritmo 2.1.2 é uma simulação precisa do Algoritmo 2.1.1, logo seu custo de tempo é $O(c_f|S|)$. O custo do restante do algoritmo é $k + \sum_{i=1}^{k} |A_i|$, que é menor que $k + \prod_{i=1}^{k} |A_i| = O(|S|)$, se $|A_i| > 1$ para todo i. Logo o custo total é $O(c_f|S| + |S|) = O(c_f|S|)$.

Construção incremental de soluções candidatas

Certos problemas de busca completa em que as soluções candidatas podem ser construídas de maneira incremental ficam mais fáceis se pensamos de maneira iterativa. O Algoritmo 2.1.3 é um modelo para estes casos. Seja k o tamanho de uma solução candidata. Denotamos por $S = S_k$ o conjunto de todas soluções candidatas e por S_i o conjunto de todas as soluções parciais de tamanho $i < k$. Observe que o custo de tempo deste algoritmo é $O(c_f|S_k| + \sum_{i=1}^{k} |S_i|)$. Geralmente, ocorre que $|S_i| < |S_{i+1}|$, para todo $i < k$. Isto implica que $\sum_{i=1}^{k} |S_i| = O(k|S_k|) = O(|S_k|)$ e que o custo na verdade é $O(c_f|S_k|) = O(c_f|S|)$. Esta busca também é conhecida como *busca em largura* (veja a Seção 3.3.2).

Algoritmo 2.1.3 Forma geral de uma busca completa incremental iterativa.

```
1: construir o conjunto S_1
2: para i ← 2 até k faça
3:     utilizar o conjunto S_{i−1} para construir o conjunto S_i
4: fim para
5: para cada x ∈ S_k faça
6:     calcule f(x) e faça alguma coisa com o resultado
7: fim para
```

Representação de soluções candidatas com máscaras de *bits*

Sempre é possível representar uma solução candidata através de uma sequência de *bits*. Mas quando esta sequência é pequena o suficiente, o algoritmo de busca completa se torna extremamente simples.

Para problemas em que uma solução candidata pode ser representada por um *bitmask* de n *bits*, onde n é no máximo, aproximadamente, 20, basta fazer um laço como o do Algoritmo 2.1.4. O custo de tempo neste caso é claramente $O(c_f 2^n)$. Note que $2^n = |S|$. Logo, o custo de tempo do Algoritmo 2.1.4 é $O(c_f|S|)$.

Algoritmo 2.1.4 Forma geral de uma busca completa com *bitmask*.

```
1: para i ← 0 até 2^n − 1 faça
2:     com os n bits menos significativos de i, construir uma solução candidata x ∈ S e calcular f(x)
3: fim para
```

2.1.3 Redução do espaço de busca

Muitas vezes podemos evitar construir certas soluções candidatas a partir de soluções parciais que já sabemos que não são ótimas, ou que não satisfazem a propriedada dada. Esta estratégia é denominada *redução do espaço de busca*. Em inglês usa-se o termo *pruning*, cujo português é *poda*. A origem deste termo vem do fato de que subárvores (da árvore implícita de um *backtracking*) não são exploradas. Este tipo de estratégia não altera o custo assintótico de uma busca completa, mas surte bastante efeito ao ser colocado em prática.

Pruning é, além de um tópico frequentemente explorado em competições, a melhor estratégia conhecida para lidar com qualquer problema NP-completo. Nesta seção, discutimos *pruning* para o

primeiro problema provado ser NP-completo, o *problema da satisfatibilidade booleana* – SAT, enquanto discutimos *o que são* problemas NP-completos na Seção 4.4.

Antes de discutir estratégias de *pruning* para o SAT, precisamos de algumas definições.

1. Um *alfabeto* é qualquer conjunto finito e não-vazio, cujos elementos chamamos de *variáveis*.
2. Uma *fórmula proposicional* (FP) sobre um alfabeto $\mathcal{P}$ é definida indutivamente da seguinte maneira. Se $p \in \mathcal{P}$, então p é uma FP. Se ϕ e ψ são FPs, então $\neg\phi$, $(\phi \vee \psi)$ e $(\phi \wedge \psi)$ são FPs.
3. Seja $\mathcal{P}$ um alfabeto e $p \in \mathcal{P}$. Dizemos que p e $\neg p$ são *literais*.
4. Uma *cláusula* é um conjunto de literais conectados por $\vee$, como em $(p \vee q \vee \neg r \vee s)$.
5. Uma FP está na *forma normal conjuntiva* (sigla em inglês: CNF) se ela for um conjunto de cláusulas conectadas por $\wedge$, como em $\phi \wedge \psi \wedge \eta$.
6. Uma *valoração* para um alfabeto $\mathcal{P}$ é uma função $\mathbb{V}_0 : \mathcal{P} \longrightarrow \{\mathrm{V}, \mathrm{F}\}$.
7. A valoração $\mathbb{V}(\phi)$ de uma FP ϕ estende uma valoração $\mathbb{V}_0 : \mathcal{P} \longrightarrow \{\mathrm{V}, \mathrm{F}\}$ da seguinte forma.
   (a) $\phi \in \mathcal{P} \implies \mathbb{V}(\phi) = \mathbb{V}_0(\phi)$
   (b) $\mathbb{V}(\phi) = \mathrm{V} \iff \mathbb{V}(\neg\phi) = \mathrm{F}$
   (c) $\mathbb{V}(\phi \vee \psi) = \mathrm{V} \iff \mathbb{V}(\phi) = \mathrm{V}$ ou $\mathbb{V}(\psi) = \mathrm{V}$
   (d) $\mathbb{V}(\phi \wedge \psi) = \mathrm{V} \iff \mathbb{V}(\phi) = \mathrm{V}$ e $\mathbb{V}(\psi) = \mathrm{V}$
8. SAT: Dada uma FP ϕ na CNF, determinar se existe $\mathbb{V}_0 : \mathcal{P} \longrightarrow \{\mathrm{V}, \mathrm{F}\}$ tal que $\mathbb{V}(\phi) = \mathrm{V}$.

O SAT que definimos é, na verdade, o CNF-SAT. Não entramos em detalhes sobre a versão original, já que esta é equivalente e muito mais simples.

Começamos identificando o espaço de busca S do problema. Note que S é o conjunto de todas as possíveis valorações para as variáveis de ϕ. Ou seja,

$$S = \{\mathbb{V}_0 \mid \mathbb{V}_0 : \mathcal{P} \longrightarrow \{\mathrm{V}, \mathrm{F}\}\}$$

Uma maneira mais conveniente de definir S é:

$$S = \{(v_1, v_2, ..., v_n) \mid v_i \in \{\mathrm{V}, \mathrm{F}\}\}$$

onde $n = |\mathcal{P}|$. Neste caso, $|S| = 2^n$.

O próximo passo é identificar a função f. Note que f é tal que $f(x) = \mathrm{V}$ se, e somente se, x é uma valoração para $\mathcal{P}$ tal que $\mathbb{V}(\phi) = \mathrm{V}$. Logo, para calcular $f(x)$, basta utilizar os valores de x para calcular $\mathbb{V}(\phi)$. Tal cálculo pode ser feito em tempo $O(m)$, onde m é o somatório do número de literais em cada cláusula de ϕ, ou simplesmente o *tamanho* de ϕ. Note também que o problema pede apenas para determinar se existe algum $x \in S$ tal que $f(x) = \mathrm{V}$, ou seja, a busca pode ser interrompida ao encontrar uma solução.

Com isto, chegamos a um *backtracking* $O(m2^n)$.

A primeira otimização que fazemos é modificar o *backtracking* para sempre atribuir valores à primeira variável que ocorre em ϕ e simplificar ϕ conforme esta atribuição. Esta simplificação nada mais é do que a antecipação do cálculo de $f(x)$, que agora é feito em parcelas[1], ou seja, o custo assintótico continua o mesmo. Esta é uma poda implícita que evita atribuir valores a variáveis que já desaparecem da fórmula com apenas uma solução parcial.

As otimizações descritas a seguir são conhecidas como o algoritmo DPLL.

[1] Observar que um custo é "pago em parcelas" é fazer *análise amortizada*.

1. Verificar se ϕ é um *conjunto consistente de literais*, ou seja, se tem a forma $p \wedge \neg q \wedge ... \wedge r$ e nenhum literal p aparece simultaneamente com sua forma negada $\neg p$. Se for, existe uma solução e o algoritmo pode ser interrompido.

2. Verificar se ϕ contém uma cláusula vazia. Neste caso, força-se um retrocesso (poda explícita).

3. Enquanto houver uma cláusula unitária em ϕ, atribuir V ao seu literal e simplificar ϕ. Novamente, simplificação é uma poda implícita.

4. Enquanto houver um *literal puro* nas cláusulas de ϕ, atribuir V a este literal e simplificar ϕ. Um literal puro é uma variável que só ocorre em uma fórmula com uma única polarização. Por exemplo: se ocorre $\neg p$, mas não ocorre p, $\neg p$ é um literal puro.

Note que estas otimizações, separadas ou combinadas, não são suficientes para encontrarmos, analiticamente, um limite superior menor que $O(m2^n)$ para o algoritmo. No entanto, todas fazem grande diferença prática.

Com esta discussão, encerramos o conteúdo acerca de busca completa. A seção a seguir discute o próximo paradigma, também famoso por inúmeros algoritmos.

2.2 Divisão e conquista

Alguns problemas podem ser resolvidos eficientemente da seguinte maneira:

1. Divide-se uma instância de tamanho n em a instâncias de tamanho n/b;

2. Calcula-se as soluções para as a instâncias menores;

3. Combina-se as soluções das instâncias menores para se obter uma solução.

O nome desta estratégia é *divisão e conquista*. O custo de tempo de algoritmos que utilizam esta estratégia é modelado pela relação de recorrência

$$T(n) = aT(n/b) + f(n)$$

onde $f(n)$ é o custo para realizar as divisões e combinações. A próxima seção apresenta um resultado geral para encontrar um limite estreito para $T(n)$, relacionando os valores a, b e $f(n)$.

2.2.1 Teorema mestre

Seja $T(n) = aT(n/b) + f(n)$, onde $a \geq 1$ e $b > 1$.

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$.

2. Se $f(n) = \Theta(n^{\log_b a} \lg^k n)$ para algum $k \geq 0$, então $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para algum $c < 1$ e todo n maior ou igual a algum n_0, então $T(n) = \Theta(f(n))$.

2.2.2 Exponenciação rápida

É comum cursos introdutórios de programação sugerirem o exercício de se implementar um programa para calcular uma potência inteira y de um número x. Geralmente, o algoritmo que implementamos é aquele que simplesmente multiplica o resultado por x y vezes, ou seja, o algoritmo $\Theta(y)$.

O Algoritmo 2.2.1 calcula x^y em tempo $\Theta(\lg y)$. Note que o custo de tempo da versão recursiva pode ser modelado por $T(y) = T(y/2) + 1$. Ou seja, $T(y) = aT(y/b) + f(y)$, onde $a = 1$, $b = 2$ e $f(y) = 1$. Neste caso, com $k = 0$, $y^{\log_b a} \lg^k y = 1 \implies f(y) = \Theta(y^{\log_b a} \lg^k y)$. Então, pelo caso 2

do teorema mestre, $T(y) = \Theta(y^{\log_b a} \lg^{k+1} y) = \Theta(\lg y)$. Para a versão iterativa, basta observar que cada iteração do laço divide y por 2, até que $y = 0$. Em outras palavras, o número m de iterações é tal que $\frac{y}{2^m} = 0$ e $t = \frac{y}{2^{m-1}} > 0$. Se $t = 2$, então $t/2 = 1 > 0$. Logo, $t = 1$. Neste caso, $\frac{y}{2^{m-1}} = 1 \implies y = 2^{m-1} \implies \lg y = m - 1 \implies m = \lg y + 1 = \Theta(\lg y)$.

Algoritmo 2.2.1 Algoritmo de exponenciação rápida nas versões recursiva e iterativa.

```
1: função POWER-RECURSIVO(x : inteiro, y : inteiro)
2:     se y = 0 então
3:         retorne 1
4:     senão se y = 1 então
5:         retorne x
6:     senão se y é par então
7:         retorne POWER-RECURSIVO(x^2, y/2)
8:     senão
9:         retorne xPOWER-RECURSIVO(x^2, (y − 1)/2)
10:    fim se
11: fim função
12: função POWER-ITERATIVO(x : inteiro, y : inteiro)
13:    r ← 1
14:    enquanto y > 0 faça
15:        se y é ímpar então
16:            r ← rx
17:        fim se
18:        x ← x^2
19:        y ← y/2
20:    fim enquanto
21:    retorne r
22: fim função
```

Observe que se medimos os custos de tempo dos algoritmos de exponenciação em termos do tamanho da entrada n, este parâmetro deve ser o número de *bits* usados para representar y, ou seja, $n = \lg y$. Neste caso, o algoritmo linear em y na verdade é exponencial no tamanho da entrada ($\Theta(2^n)$) e o algoritmo de exponenciação rápida na verdade é linear no tamanho da entrada ($\Theta(n)$).

O fato por trás deste algoritmo não-trivial, mas extremamente simples e rápido, é: qualquer valor do tipo $a * b$ que possa ser eficientemente calculado a partir de $a * (b/2)$ pode ser calculado com custo de tempo $O(\lg b)$. Exemplos: produto, potência, etc.

Podemos adaptar este algoritmo facilmente para essencialmente duas grandes aplicações. Uma é calcular potências modulares (o resto de x^y na divisão por algum número)[2]. A outra, que é absurdamente abrangente, é calcular potências matriciais (que também podem ser modulares!). Inúmeros problemas podem ser resolvidos através de exponenciação rápida de matrizes quadradas. Alguns exemplos são mostrados a seguir.

Cadeias de Markov

Cadeia de Markov é um modelo probabilístico para sistemas que mudam de estado ao longo do tempo. Este modelo se aplica a sistemas em que o conjunto de possíveis estados é finito e há uma probabilidade fixa do sistema ir de um estado j qualquer para um estado i qualquer. Note que podemos criar uma matriz quadrada P para representar este modelo, onde a entrada $p_{i,j}$ representa a probabilidade do sistema ir do estado j para o estado i. Note também que toda coluna de P deve ter soma 1.

[2]Potência modular aparece em problemas de teoria dos números e criptografia.

Uma aplicação simples de uma cadeia de Markov é determinar a probabilidade do sistema estar no estado i, dado que iniciou no estado j, após n ciclos de tempo. Basta calcular P^n e obter a entrada i, j. Um ciclo de tempo pode ser um turno/uma rodada de um jogo, ou uma travessia de uma aresta em um grafo, etc.

Algumas cadeias de Markov têm a propriedade de que, a partir de uma certa potência, todas as potências de P são iguais. Isto quer dizer que podemos calcular a probabilidade do sistema iniciar no estado j e terminar no estado i, passada uma quantidade infinita de tempo. Utilizando a ideia de exponenciação rápida, basta elevar P ao quadrado até que $P^2 = P$. Tais cadeias de Markov são aquelas em que o sistema tende a estabilizar. Isto ocorre, por exemplo, se alguma pontuação faz o jogo terminar[3], ou se o transeunte pára de andar quando chega a algum dado destino, etc.

Grafos

Alguns grafos podem ser representados pelo que chamamos de *matrizes de adjacência*. É possível obter números interessantes através da n-ésima potência de tais matrizes, como comprimentos de caminhos mínimos em grafos com n vértices, ou então quantidades de caminhos de comprimento n. Tais problemas são discutidos no Capítulo 3.

Relações de recorrência

Algumas relações de recorrência podem ser calculadas eficientemente através de potências de matrizes. Um exemplo clássico é o número de Fibonacci, que frequentemente aparece em torneios de programação. Este é o exemplo que discutimos em detalhes nesta seção.

O *n-ésimo número de Fibonacci* F_n é definido pela relação de recorrência a seguir.

$$F_n = \begin{cases} n & \text{se } n = 0, \text{ ou } n = 1 \\ F_{n-1} + F_{n-2} & \text{caso contrário} \end{cases}$$

Discutimos o cálculo dos números de Fibonacci pelo curioso fato de que existem duas maneiras *exponencialmente* mais rápidas que a maneira trivial de calcular estes números. A maneira trivial é o cálculo direto da relação de recorrência, através de uma simples função recursiva, cujo custo de tempo é $\Theta(\phi^n)$, onde $\phi = (1 + \sqrt{5})/2$ é a razão áurea. Uma das maneiras eficientes custa tempo $\Theta(n)$ e é discutida na Seção 2.4. A outra maneira é mostrada a seguir.

Seja $\mathbf{F}_1$ o vetor coluna a seguir.

$$\mathbf{F}_1 = \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Queremos uma matriz P tal que $P^n \mathbf{F}_1 = \mathbf{F}_{n+1}$. Neste caso,

$$P = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \implies P\mathbf{F}_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_1 + F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = \mathbf{F}_2$$

$$\implies P^n \mathbf{F}_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_n + F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \mathbf{F}_{n+1}$$

Logo, para calcular F_n basta calcular P^n e obter a entrada $2, 1$. Como vimos, isto pode ser feito em tempo $\Theta(\lg n)$.

Observe que n, no cálculo de F_n, é a entrada *em si*. Logo, se medimos os custos em termos do *tamanho* da entrada m, m deve ser o número de *bits* em n, ou seja, $m = \lg n \implies n = 2^m$. Logo, as complexidades são respectivamente $\Theta(\phi^{2^m})$ (dupla exponencial), $\Theta(2^m)$ (exponencial) e $\Theta(m)$ (linear).

[3] O famoso problema de teoria da probabilidade *ruína do apostador* pode ser resolvido com uma cadeia de Markov.

2.2.3 Ordenação

O Algoritmo 2.2.2 é o algoritmo de ordenação ótimo mais simples que existe: *merge sort*. Divide-se o vetor em duas metades (custo $O(1)$), ordena-se recursivamente cada metade (custo $2T(n/2)$) e combina-se os dois subvetores ordenados (custo $O(n)$). Pelo caso 2 do teorema mestre, a complexidade final é $\Theta(n \lg n)$. É possível provar que qualquer algoritmo de ordenação baseado em comparação, como o *merge sort*, custa tempo $\Omega(n \lg n)$. Logo, o *merge sort* é de fato ótimo.

Algoritmo 2.2.2 Algoritmo de ordenação *merge sort*.

```
1: função MERGE(A : vetor de inteiros, b, e : inteiros)
2:     m ← (b + e)/2
3:     ls ← m − b + 1
4:     rs ← e − m
5:     Sejam L[1..ls + 1] e R[1..rs + 1] vetores de inteiros.
6:     L[1..ls] ← A[b..m]
7:     R[1..rs] ← A[m + 1..e]
8:     L[ls + 1] ← ∞
9:     R[rs + 1] ← ∞
10:    li ← 1
11:    ri ← 1
12:    para i ← b até e faça
13:        se L[li] ≤ R[ri] então
14:            A[i] ← L[li]
15:            li ← li + 1
16:        senão
17:            A[i] ← R[ri]
18:            ri ← ri + 1
19:        fim se
20:    fim para
21: fim função
22: função MERGE-SORT(A : vetor de inteiros, b, e : inteiros)
23:    se b < e então                ▷ apenas vetores de tamanho maior que 1 precisam ser ordenados
24:        MERGE-SORT(A, b, (b + e)/2)
25:        MERGE-SORT(A, (b + e)/2 + 1, e)
26:        MERGE(A, b, e)
27:    fim se
28: fim função
```

Outro algoritmo de ordenação famoso, também baseado em comparação, é o *quick sort*. Neste algoritmo, é o passo da divisão que custa tempo $O(n)$, ao invés do passo de combinação. Uma desvantagem deste algoritmo é que pode ocorrer que a divisão deixe como subproblemas um vetor de tamanho 1 e outro de tamanho $n - 2$. Neste caso, é possível mostrar que o custo de tempo é $O(n^2)$. Por esta razão e pelo fato do *quick sort* ser mais complicado, ficaremos apenas com o *merge sort*.

Um dos poucos problemas em que é necessário adaptar um algoritmo de ordenação é o problema de se contar inversões. Uma inversão em uma sequência $(a_1, a_2, ..., a_n)$ é um par ordenado (i, j) tal que $i < j$ e $a_i > a_j$. Para fazer o *merge sort* retornar o número de inversões, adapte o algoritmo recursivo para retornar a soma de suas três chamadas. Na função de combinação, basta acumular $ri - 1$ sempre que $L[li] \leq R[ri]$ (dentro do laço) e retornar o total acumulado.

2.2.4 Busca binária

Esta seção apresenta *busca binária*, uma técnica de busca extremamente rápida e simples. Cursos

introdutórios de programação costumam ensinar esta técnica, mas mostram apenas sua aplicação mais trivial: busca binária em um vetor ordenado. Note, no entanto, que busca binária pode ser feita em qualquer *função* ordenada.

O Algoritmo 2.2.3 mostra uma busca binária para uma função $f : \mathbb{N} \longrightarrow \mathbb{R}$ não-decrescente. Dado y, a função LOWER-BOUND() encontra o menor n tal que $f(n) = y$. O valor -1 é retornado se $f(n) \neq y, \forall n$. Observe que, no pior caso, o algoritmo chama a si mesmo para um subproblema duas vezes menor. Ou seja, $T(e-b) = T((e-b)/2) + 1 = \Theta(\lg(e-b))$, assumindo que avaliar f em um ponto tem custo constante (o que ocorre na maioria das vezes, ainda que seja um custo alto).

Embora o Algoritmo 2.2.3 funcione apenas para funções não-decrescentes e com domínio $\mathbb{N}$, uma busca binária para funções não-crescentes e/ou com domínio $\mathbb{R}$ seria bastante análoga. Além disso, também é fácil adaptar a função para retornar o maior n tal que $f(n) = y$, ao invés do menor, ou seja, criar uma função UPPER-BOUND(). Basta modificar o intervalo de busca da última chamada recursiva, quando $b < e$ e $f(n) = y$. Faça todas estas adaptações como exercício!

Algoritmo 2.2.3 Busca binária em uma função $f : \mathbb{N} \longrightarrow \mathbb{R}$.

```
1: Assume-se que f : ℕ ⟶ ℝ seja uma função não-decrescente.
2: Seja N o ponto máximo do domínio que precisamos considerar.
3: LOWER-BOUND(50, 1, N)                          ▷ retorna o menor n tal que f(n) = 50
4: função LOWER-BOUND(y : real, b, e : inteiros)
5:     se b > e então
6:         retorne −1
7:     senão se b = e então
8:         se f(b) = y então
9:             retorne b
10:        fim se
11:        retorne −1
12:    fim se
13:    n ← (b + e)/2
14:    y' ← f(n)
15:    se y' < y então
16:        retorne LOWER-BOUND(y, n + 1, e)
17:    senão se y < y' então
18:        retorne LOWER-BOUND(y, b, n − 1)
19:    fim se
20:    retorne LOWER-BOUND(y, b, n)                ▷ neste caso, f(n) = y
21: fim função
```

2.2.5 Árvore de segmentos

Considere o seguinte problema. Dada uma sequência de números, indexados de 1 a n, execute uma série de S operações do tipo: 1) "U i v" – o número de índice i passa a ser v; e 2) "Q i j" – imprima o menor número com índice em $\{i, i+1, i+2, ..., j\}$ $(i \leq j)$. Este problema é conhecido como "consulta do mínimo do intervalo". Em inglês, *range minimum query* (sigla RMQ).

Inicialmente, o problema RMQ parece ser trivial. Basta manter um vetor de tamanho n. A cada operação U, atualizamos uma posição deste vetor, com custo $O(1)$. A cada operação Q, encontramos o menor número do intervalo com um laço, com custo $O(n)$. Esta solução parece funcionar muito bem. No entanto, se o número S de operações e o valor de n forem da ordem de 10^4, esta solução, de custo de tempo $O(Sn)$, é certamente TLE.

Esta seção apresenta a *árvore de segmentos*, uma ED abstrata capaz de resolver o problema *range query*, a versão genérica do RMQ para qualquer tipo de consulta, com custo de tempo $O(n + S \lg n)$.

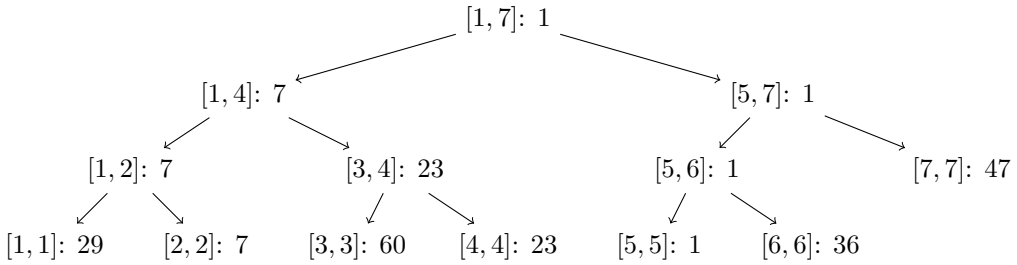

Figura 2.2.1: Árvore de segmentos do RMQ para a sequência $(29, 7, 60, 23, 1, 36, 47)$.

Uma árvore de segmentos é uma árvore binária, onde um nó é responsável por armazenar a resposta da consulta para um intervalo. O filho esquerdo armazena a resposta para a metade esquerda do intervalo do pai, enquanto o filho direito armazena a resposta para a metade direita. Observe, então, que a raiz armazena a resposta para a sequência inteira; e que cada folha armazena um número da sequência. A Figura 2.2.1 ilustra a árvore de segmentos associada a uma sequência de tamanho 7, para o RMQ.

A propriedade fundamental de uma árvore de segmentos é o comprimento máximo de um caminho da raiz até uma folha. Observe que este comprimento máximo é $O(\lg n)$. A justificativa é que, a cada nó que atravessamos, o tamanho do intervalo de um nó é metade do anterior. Basta lembrar que só podemos dividir n por 2 no máximo $O(\lg n)$ vezes, antes que o resultado seja 1.

As próximas seções discutem cada operação da árvore de segmentos. A implementação é feita com um vetor de índices $1, 2, ..., 2n-1$. A raiz da árvore está no índice 1. O filho esquerdo do nó no índice p está no índice $2p$, enquanto o filho direito está no índice $2p+1$. Note que $2p \neq 2q$, $2p \neq 2q+1$, $2p+1 \neq 2q$ e que $2p+1 \neq 2q+1$, para todo $p \neq q$, com $p, q \geq 1$.

Inicialização

O Algoritmo 2.2.4 inicializa a árvore de segmentos assumindo que $a = (a_1, a_2, ..., a_n)$ é a sequência de entrada. Basta fazer uma travessia em profundidade (filho esquerdo, filho direito, pai), calculando o valor de um nó na volta da recursão. O custo é $O(n)$.

Algoritmo 2.2.4 Inicialização de uma árvore de segmentos.

```
1: INIT(1, 1, n)                                          ▷ chamada inicial
2: função INIT(p, L, R : inteiros)
3:     se L = R então                                     ▷ visitando nó folha
4:         st[p] ← a_L
5:     senão
6:         INIT(2p, L, (L + R)/2)
7:         INIT(2p + 1, (L + R)/2 + 1, R)
8:         st[p] ← min{st[2p], st[2p + 1]}                ▷ executa na volta da recursão
9:     fim se
10: fim função
```

Consulta

O Algoritmo 2.2.5 retorna o menor número no intervalo $[i, j]$. A ideia é descer por no máximo dois ramos da árvore, resultando na complexidade $O(2 \lg n) = O(\lg n)$. Ao descer no ramo esquerdo, interrompemos a travessia para nós "esquerdos", que já estão fora do intervalo da consulta, retornando o valor neutro da operação da árvore (∞, para a operação "min"). Interrompemos a travessia também

para um nó "direito", que já conhece a resposta para um grande "subintervalo" de $[i, j]$, retornando o valor deste nó. Para um nó cujo intervalo se sobrepõe com o intervalo da consulta de uma maneira não-trivial, descemos em ambos os filhos e combinamos as respostas. A travessia no ramo direito é análoga. Interrompemos para nós "direitos", que já estão fora do intervalo da consulta, e interrompemos para um nó "esquerdo", que já conhece a resposta para um grande subintervalo de $[i, j]$.

Algoritmo 2.2.5 Operação de consulta em uma árvore de segmentos.

```
1: Assume-se que st[1..2n − 1] é o vetor que representa a árvore de segmentos.
2: QUERY(1, 1, n, i, j)                                              ▷ chamada inicial
3: função QUERY(p, L, R, i, j : inteiros)
4:     se j < L ou R < i então                  ▷ os intervalos [L, R] e [i, j] não se sobrepõem
5:         retorne ∞
6:     senão se i ≤ L ≤ R ≤ j então                    ▷ [L, R] está totalmente contido em [i, j]
7:         retorne st[p]
8:     senão                          ▷ os intervalos se sobrepõem de alguma maneira não-trivial
9:         retorne min{QUERY(2p, L, (L + R)/2, i, j), QUERY(2p + 1, (L + R)/2 + 1, R, i, j)}
10:    fim se
11: fim função
```

Atualização

O Algoritmo 2.2.6 atualiza a posição i da sequência associada à árvore de segmentos. Este algoritmo é mais simples que o algoritmo de consulta. Basta descer no ramo do elemento a ser modificado e, na volta da recursão, atualizar os nós atravessados com o valor mínimo de seus filhos. O custo de tempo é $O(\lg n)$.

Algoritmo 2.2.6 Operação de atualização em uma árvore de segmentos.

```
1: Assume-se que st[1..2n − 1] é o vetor que representa a árvore de segmentos.
2: UPDATE(1, 1, n, i, v)                                             ▷ chamada inicial
3: função UPDATE(p, L, R, i, v : inteiros)
4:     se i = L = R então          ▷ chegamos à folha do intervalo [i, i], armazenada em st[p]
5:         st[p] ← v
6:     senão
7:         se i ≤ (L + R)/2 então
8:             UPDATE(2p, L, (L + R)/2, i, v)
9:         senão
10:            UPDATE(2p + 1, (L + R)/2 + 1, R, i, v)
11:        fim se
12:        st[p] ← min{st[2p], st[2p + 1]}                  ▷ executa na volta da recursão
13:    fim se
14: fim função
```

Atualização de intervalo

Alguns problemas incluem a operação "atualizar intervalo", que deve atualizar um intervalo da sequência com um único valor. Isto pode ser feito através de uma técnica chamada *propagação preguiçosa*. Utilizamos um vetor auxiliar $staux[1..2n - 1]$, para armazenar valores que devem ser "propagados". O algoritmo é semelhante ao da consulta. Quando o intervalo de um nó está inteiramente contido no intervalo da atualização, este nó p atualiza $st[p]$, $staux[2p]$, $staux[2p + 1]$ e

retorna. Isto sinaliza para os filhos de p que, assim que alguma operação visitá-los, eles devem se atualizar e propagar esta atualização para seus próprios filhos. Observe que a implementação desta operação requer alteração em todas as outras! Todas as operações devem iniciar com uma chamada a PROPAGATE(). Projetar o algoritmo de atualização em intervalo e adaptar as outras operações ficam de exercício para o leitor. Observe que os custos de todas as operações devem permancer $O(\lg n)$.

Inserção e remoção

Para completar o CRUD (*create, retrieve, update, delete* = inserir, recuperar, atualizar, remover), estão faltando as operações de inserção e remoção de elementos. Isto muda tudo! Operações que alteram o tamanho da sequência subjacente não são fáceis de implementar. Isto requer uma ED com alocação dinâmica de elementos. Em outras palavras, o vetor $st[1..2n-1]$ é inútil!

A solução é utilizar o que chamamos de *árvore de segmentos implícita*, que é também uma *árvore de busca binária*, ou BST – *binary search tree*. O nome "implícita" vem de uma característica em particular. Em uma árvore de segmentos tradicional, cada nó é sempre responsável pelo mesmo intervalo. Isto quer dizer que poderíamos armazenar os limites do intervalo dentro do nó, ao invés de calculá-los em cada chamada (é computacionalmente mais vantajoso calcular do que armazenar, pois o cálculo custa tempo $O(1)$, enquanto armazenar custaria dois vetores de tamanho $2n-1$ a mais). Em uma árvore de segmentos implícita, o intervalo associado a um nó varia à medida que elementos são inseridos na sequência subjacente (ou removidos). Isto quer dizer que *não* poderíamos armazenar os limites do intervalo dentro do nó, pois mantê-los atualizados *em toda a árvore* custaria tempo $O(n)$ por operação. É preciso calculá-los sempre! Portanto, o intervalo associado a um nó não é explícito, mas *implícito*. Esta mesma característica se aplica à *chave* da busca binária. Normalmente, cada nó de uma BST armazena uma chave e os dados associados a esta chave. No caso de uma árvore de segmentos implícita, a chave de um nó é o índice (na sequência subjacente) do elemento armazenado pelo nó. Ou seja, também precisa ser calculada em cada visita (é implícita!).

Uma árvore de segmentos implícita:

- associada a uma sequência de tamanho n tem exatamente n nós, um para cada elemento da sequência. Diferentemente de como é em uma árvore de segmentos tradicional, os $n-1$ nós "intermediários" não são necessários.
- é cartesiana. Significa que uma travessia simétrica (filho esquerdo, pai, filho direito) gera a sequência subjacente. Esta característica vem do fato de que a chave (implícita) de um nó é o índice (na sequência subjacente) do elemento armazenado por ele.
- deve ser balanceada, para que os custos das operações sejam $O(\lg n)$. Neste caso, podemos utilizar qualquer BST balanceada para a implementação (como árvores AVL, árvores rubro-negras, ou *treaps*).

O cálculo da chave de um nó é feito com o intervalo do nó (passado como parâmetro para a operação que está sendo executada) e com o tamanho da subárvore esquerda deste nó. Este tamanho pode ser armazenado pelo nó e atualizado eficientemente conforme for necessário. Por outros detalhes de implementação, um nó também precisa armazenar o tamanho de sua subárvore direita.

A implementação da árvore de segmentos implícita é deixada como exercício. Para realizar esta tarefa, é preciso ter em mente que a informação sobre o intervalo de um nó precisa ser atualizada em qualquer operação que não seja de consulta, ou seja, em qualquer operação que efetivamente modifique a árvore.

2.3 Algoritmos gulosos

Alguns problemas de otimização combinatória (veja a Seção 2.1) podem ser resolvidos pelo que chamamos de *algoritmos gulosos*. Sempre que isto é possível, a solução gulosa é preferível a busca completa, pois é muito mais eficiente.

O Algoritmo 2.3.1 mostra a estrutura geral de um algoritmo guloso. A ideia é fazer uma série de escolhas. Cada escolha reduz o problema restante A e melhora a solução final x. Sempre fazemos a escolha que parece ser a melhor naquele momento e, por isso, a chamamos de *escolha gulosa*.

Algoritmo 2.3.1 Estrutura geral de um algoritmo guloso.

```
1: Iniciamos com um problema A.
2: x ← ∅
3: enquanto A ≠ ∅ faça
4:     c ← ESCOLHA-GULOSA(A) ▷ Determina a escolha gulosa c a partir do problema restante A
5:     A ← SUBPROBLEMA(A, c)      ▷ Reduz o problema restante A a partir da escolha gulosa c
6:     x ← MELHORA-SOLUÇÃO(x, c)      ▷ Melhora a solução ótima x com a escolha gulosa c
7: fim enquanto
```

Para que um problema possa ser solucionado corretamente por um algoritmo guloso, ele precisa satisfazer as seguintes propriedades:

- *Subestrutura ótima*: uma solução ótima para um problema A contém soluções ótimas para subproblemas de A.
- *Propriedade da escolha gulosa*: a melhor escolha que pode ser feita a partir do subproblema restante pode depender das escolhas feitas até agora, mas não de escolhas futuras, ou de soluções para subproblemas. Além disso, a escolha gulosa deve deixar um único subproblema. Sem esta propriedade, fazer a escolha que parece ser a melhor no momento irá gerar resultados *incorretos*.

Na maioria das vezes, mostrar que um problema satisfaz as propriedades acima não é trivial. Por esta razão, quando um problema guloso aparece em uma competição, ou ele é um dos problemas mais fáceis, ou é um dos mais difíceis. Problemas *ad hoc*[4] também são os mais fáceis, ou os mais difíceis, mas aparecem com muito mais frequência que problemas gulosos.

Ao tentar atacar um problema com a abordagem gulosa, tentamos responder as seguintes perguntas:

- O que é a solução x? Qual é a forma do problema A? Ele exibe subestrutura ótima?
- O que poderia ser uma escolha c? Qual seria a gulosa?
- Como a escolha gulosa reduziria o problema? Restaria um único subproblema?
- A escolha gulosa poderia depender de escolhas futuras, ou de soluções para subproblemas?
- Como utilizaríamos a escolha gulosa para melhorar a solução ótima?

Aqui, damos dois exemplos de problemas que podem ser resolvidos pela abordagem gulosa. Em cada um, respondemos as perguntas acima. Em seguida, mostramos um problema famoso, conhecido como o *problema do troco*.

[4]Um problema é *ad hoc* se a ideia geral de sua solução dificilmente pode ser aproveitada para resolver outros problemas. Por isto, problemas *ad hoc* geralmente não são famosos.

Como Treinar Seu Dragão

Enunciado resumido do problema[5]: dada uma sequência de no máximo 10^5 dragões, encontrar o melhor escalonamento para treinar estes dragões. Apenas um dragão pode ser treinado de cada vez e um treinamento não pode ser interrompido. O i-ésimo dragão que aparece na entrada chega ao centro de treinamento no i-ésimo dia, demora T_i dias para ser treinado e fica dormindo enquanto não começa seu treinamento. É paga uma multa de valor F_i para cada dia que o i-ésimo dragão passa dormindo. O centro de treinamento não é avisado previamente de que um dragão será enviado para ser treinado. A resposta final pedida é o prejuízo mínimo. Restrições: $1 \leq T_i, F_i \leq 10^3$ e $T_i/F_i \neq T_j/F_j$, se $i \neq j$.

Antes de discutirmos a abordagem gulosa, vamos considerar o custo de uma busca completa. Uma busca completa para este problema precisaria considerar cada sequência de dragões possível. Se temos n dragões, temos $n!$ sequências possíveis. Para testar uma sequência o custo é $O(n)$. Neste caso, uma busca completa custaria $O(n \cdot n!)$. Para $n \leq 10^5$, não há computador no mundo que seja capaz de executar uma busca completa em menos tempo do que leva uma vida inteira, ainda que implementássemos todas as técnicas de *prunning* possíveis.

A principal característica deste problema que permite que ele seja solucionado por um algoritmo guloso é: $T_i/F_i \neq T_j/F_j$, se $i \neq j$. Neste caso, ao escolher um dragão para ser treinado entre um par qualquer de dragões distintos, sempre é possível determinar o dragão que traria mais prejuízo se continuasse dormindo. Em palavras, devemos escolher i, se $F_i \cdot Tj > F_j \cdot T_i$. Neste caso, sempre que o centro for selecionar um dragão, ele pode simplesmente escolher aquele que traria mais prejuízo se continuasse dormindo, em comparação com todos os outros. Com isto, podemos responder as cinco perguntas:

- A solução x é um número, o prejuízo do escalonamento ótimo. O problema A é um conjunto de dragões. Para verificar que o problema exibe subestrutura ótima, tome um conjunto A qualquer de dragões. Seja $S = (a_1, a_2, ..., a_{|A|})$ o escalonamento ótimo para A, ou seja, cada $a_i \in A$. Imagine que vamos incluir um novo dragão $d \notin A$ na lista de espera. Neste caso, basta procurar i tal que $a_i < d < a_{i+1}$, onde $x < y$ indica que o dragão x deve ser treinado antes do dragão y. Ou seja, o escalonamento $S' = (a_1, ..., a_i, d, a_{i+1}, ..., a_{|A|})$ é ótimo para o problema $A \cup \{d\}$. Note, então, que S é uma subsequência de S', ou seja, S' contém S.

- Uma escolha é selecionar um dragão do problema A para ser treinado. A escolha gulosa é selecionar aquele dragão que trará mais prejuízo se continuar dormindo.

- Escolher um dragão para ser treinado reduz o conjunto A em um elemento. Claramente, resta um único subproblema.

- Para treinar um dragão, não interessa saber quais dragões vão ser treinados depois, pois um dragão é treinado até que seu treinamento esteja completo.

- "Melhoramos" a solução x somando a ela o prejuízo que o dragão selecionado deu até agora.

Lembre-se que, no pior caso, são 10^5 dragões. Se há n dragões, precisamos fazer n seleções. Se cada seleção é feita com um laço linear, a complexidade resultante é $O(n^2)$, que é obviamente inadequada para o pior caso. Por isso, usamos uma fila de prioridade (veja a Seção 1.5.6), que é capaz de selecionar e remover um dragão em tempo $O(\lg n)$, resultando na complexidade $O(n \lg n)$, que é razoável para o pior caso.

Por fim, observe que a entrada não pode ser processada de uma só vez, considerando todos os dragões em uma única ordenação. Novos dragões devem ser incluídos na fila apropriadamente, à medida que os dias passarem. Ou seja, o problema A aumenta e diminui de tamanho à medida que a simulação é feita. A solução final consiste de um laço inicial, para fazer escolhas gulosas enquanto há dragões chegando, e um laço que finaliza a simulação, treinando os dragões restantes.

Faça o algoritmo para este problema como exercício.

[5]Problema 1851 do URI OJ.

Cuarenta e Dois

Enunciado resumido do problema[6]: é dado um conjunto A de N números. Entre todos os subconjuntos de A, com no mínimo K elementos, selecione um cuja operação binária AND de todos os seus elementos seja máxima. Restrições: $1 \leq N \leq 35$, $1 \leq K \leq 7$, cada número de A é no máximo $2^{30} - 1$.

Uma busca completa para este problema precisaria testar cada um dos 2^N subconjuntos de A, onde cada teste custaria $O(N)$. O custo total seria $O(N2^N)$, impraticável para $N \leq 35$.

Agora, seja a o maior número de A e i o índice do *bit* 1 mais significativo de a. O i-ésimo *bit* do resultado final será 1 se, e somente se, existirem no mínimo $K - 1$ números no conjunto A, além de a, cujos i-ésimos *bits* também valem 1. Esta observação justifica o Algoritmo guloso 2.3.2.

Algoritmo 2.3.2 Algoritmo guloso para o problema "Cuarenta e Dois".

```
1: Assume-se que A contém os N números da entrada.
2: x ← 0
3: para i ← lg(max(A)) até 0 faça
4:     C ← ∅
5:     para cada a ∈ A faça
6:         se (a & (1 ≪ i)) ≠ 0 então
7:             C ← C ∪ {a}
8:         fim se
9:     fim para
10:    se |C| ≥ K então
11:        x ← x | (1 ≪ i)
12:        A ← C
13:    fim se
14: fim para
```

Para esclarecer a ideia do algoritmo, vamos responder as cinco perguntas.

- A solução x é um número, o AND do subconjunto de A que maximiza este valor. A forma do problema é o par (A, i). Para verificar sua subestrutura ótima, suponha que $S \subseteq A$ seja o subconjunto ótimo de A. Agora, considere o problema $A' = A \cup \{d\}$, onde $d \notin A$, e seu subconjunto ótimo $S' \subseteq A'$. Suponha que $\text{AND}(S') < \text{AND}(S)$. Isto é claramente um absurdo, pois iríamos preferir S como solução de A'. Logo, $\text{AND}(S') \geq \text{AND}(S)$. Ou seja, a solução $\text{AND}(S')$ inclui a solução $\text{AND}(S)$; e isto vale para todo i.

- Escolhemos entre permanecer com A, ou trocar A pelo seu subconjunto C. A escolha gulosa é ficar com C somente se $|C| \geq K$.

- Seja X o conjunto escolhido entre A e C. Então $(X, i-1)$ é o único subproblema que resta, que claramente é menor que (A, i).

- Remover elementos de um conjunto *agora* claramente não depende de remover elementos que sobrarem neste mesmo conjunto *mais tarde*.

- Se escolhemos C, atribuímos 1 ao i-ésimo *bit* de x.

[6] Problema 1590 do URI OJ.

2.3.1 Problema do troco

Dada uma sequência ordenada de n valores de moedas $d_1, d_2, ..., d_n$ e um valor monetário v, pede-se a menor quantidade de moedas necessárias para formar o valor v. Podemos assumir que $d_1 = 1$ (para que seja possível formar qualquer valor).

Considere o Algoritmo 2.3.3. Vamos executá-lo para a sequência de moedas $1, 5, 10, 50$ e o valor 132.

1. $x \leftarrow 0 + 2$ e $v \leftarrow 32$
2. $x \leftarrow 2 + 3$ e $v \leftarrow 2$
3. $x \leftarrow 5 + 0$ e $v \leftarrow 2$
4. $x \leftarrow 5 + 2$ e $v \leftarrow 0$

A quantidade $x = 7$ está correta. Agora, vamos executar para as moedas $1, 5, 7$ e o valor 10.

1. $x \leftarrow 0 + 1$ e $v \leftarrow 3$
2. $x \leftarrow 1 + 0$ e $v \leftarrow 3$
3. $x \leftarrow 1 + 3$ e $v \leftarrow 0$

A quantidade $x = 4$ está incorreta, pois é possível pegar somente 2 moedas de 5 para formar 10.

O problema do troco é famoso por ter duas soluções: uma gulosa e outra com programação dinâmica (o assunto da próxima seção). A gulosa custa tempo $O(n)$, mas só funciona para sistemas de moedas *canônicos*, que são os sistemas usados na prática. A definição de sistema de moedas canônico é bastante irônica (e inútil): um sistema de moedas é canônico se, e somente se, o algoritmo guloso dá o resultado ótimo para qualquer valor de entrada. Por outro lado, a solução com programação dinâmica funciona para qualquer sistema de moedas, mas custa tempo $O(nv) = O(n2^m)$, onde $m = \lg v$ é o número de *bits* necessários para codificar v.

Algoritmo 2.3.3 Algoritmo guloso para o problema do troco.

```
1: x ← 0
2: para i ← n até 1 faça
3:     x ← x + v/d_i
4:     v ← v  mod d_i
5: fim para
```

Uma condição incompleta para um sistema ser canônico, que pode ajudar em alguns casos[7], é: se d_i divide d_{i+1}, para $i = 1, 2, ..., n - 1$, então o sistema $d_1, d_2, ..., d_n$ é canônico. Agora, considere o conjunto de todas as sequências estritamente crescentes de números naturais (começando em 1) para as quais o algoritmo guloso é capaz de dar a resposta ótima, para qualquer valor de entrada v. Será que todas estas sequências são da forma descrita acima? Se sim, como provar? Se não, como encontrar um contraexemplo?

[7] Problema 1936 do URI OJ.

2.4 Programação dinâmica

Aplicamos *programação dinâmica* para resolver um problema, quando este apresenta *subestrutura ótima* e *sobreposição de subproblemas*. Na Seção 2.3, vimos que um problema apresenta subestrutura ótima se a sua solução contém soluções para subproblemas. Damos o primeiro exemplo de sobreposição de subproblemas e programação dinâmica com o cálculo de F_n, o n-ésimo número de Fibonacci. Para lembrar a definição vista na Seção 2.2.2:

$$F_n = \begin{cases} n & \text{se } n = 0, \text{ ou } n = 1 \\ F_{n-1} + F_{n-2} & \text{caso contrário} \end{cases}$$

Algoritmo 2.4.1 Algoritmo recursivo ingênuo para o cálculo do n-ésimo número de Fibonacci.

```
1: função F(n : inteiro)
2:     se n ≤ 1 então
3:         retorne n
4:     fim se
5:     retorne F(n − 1) + F(n − 2)
6: fim função
```

Considere o Algoritmo 2.4.1, um algoritmo recursivo ingênuo, concebido diretamente da definição de F_n. É possível mostrar que seu custo de tempo é $\Theta(\phi^n)$, onde $\phi = (1 + \sqrt{5})/2$ é a razão áurea. A Figura X ilustra a árvore associada às chamadas recursivas do cálculo de F_5. Observe que F_2 aparece nas subárvores esquerda e direita do nó F_5, cujas raízes são respectivamente F_4 e F_3. Neste caso, dizemos que os subproblemas F_4 e F_3 do problema original F_5 se sobrepõem, ou seja, a interseção dos conjuntos de subproblemas de cada um deles é não-vazia; e dizemos que o subproblema F_2 pertence a esta interseção. Perceba que o algoritmo calcula F_2 recursivamente sempre que seu valor é necessário; e que é aí que está o defeito de um algoritmo recursivo ingênuo. Não é necessário calcular F_2 mais de uma vez. Podemos simplesmente calcular e armazenar o valor de F_2 em um vetor, para somente acessá-lo e utilizá-lo em ocasiões futuras. Armazenar valores para reutilização futura chama-se *memoização*. Quando fazemos memoização em um problema que apresenta sobreposição de subproblemas, estamos fazendo *programação dinâmica*.

Agora, tente imaginar como seria a árvore da Figura X, se memoização fosse aplicada. Qual seria a quantidade de nós? Seriam exatamente $n - 2$ nós para calcular F_n, excluindo os dois casos base da definição. Em outras palavras, um algoritmo recursivo memoizado tem complexidade de tempo $\Theta(n)$ (muito melhor que o algoritmo ingênuo, que é $\Theta(\phi^n)$).

ao invés de satisfazer a propriedade da escolha gulosa (veja a Seção 2.3). Esta é também a característica que diferencia programação dinâmica de divisão e conquista (veja a Seção 2.2). As três técnicas são aplicadas a problemas que podem ser quebrados em problemas menores para simplificar a solução. Observe que, no entanto, em um problema de divisão e conquista, os subproblemas que a divisão deixa não se sobrepõem. Podemos exemplificar esta característica com a ordenação de um vetor. O algoritmo *merge sort* quebra um vetor em dois vetores com metade do tamanho original e os ordena recursivamente. Note que os dois vetores menores são completamente disjuntos!

melhor exemplo de trade-off tempo x espaço

2.4.1 Implementação descendente

top-down

2.4.2 Implementação ascendente

bottom-up

2.4.3 Representação de estados com máscaras de *bits*

vei

Capítulo 3

Grafos

Um *grafo simples* é um par ordenado $G = (V, E)$, onde

1. V é um conjunto finito de *vértices*.
2. $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ é um conjunto de *arestas*.
   Dizemos que $\{u, v\}$ é *incidente* a u e a v.

Tal simples estrutura matemática possui inúmeras aplicações e pode ser usada como uma estrutura de dados extremamente versátil. Por estas razões, grafos são frequentemente cobrados em competições de programação.

Observe que é muito comum representar grafos através de diagramas. A Figura 3.0.1, por exemplo, representa o grafo com arestas múltiplas do famoso problema das sete pontes de Königsberg. A solução deste problema, por Leonhard Euler, deu origem à teoria dos grafos.

Antes de discutirmos os tipos de grafos mais comuns, precisamos de algumas definições.

O *grau* $d(u)$ do vértice u é definido como o número de arestas incidentes a u.

Um *caminho* é uma sequência de vértices p, de no mínimo dois elementos, tal que, para todo par de elementos adjacentes (u, v) que ocorre em p, $\{u, v\} \in E$. Se um caminho p é da forma $(u, x_1, x_2, ..., x_{n-1}, v)$, dizemos que p é um caminho *entre* u e v, ou que p é um caminho que *conecta* u e v, de comprimento n. Um caminho da forma $(u, ..., u)$ é dito um *ciclo*. Um grafo sem ciclos é dito *acíclico*.

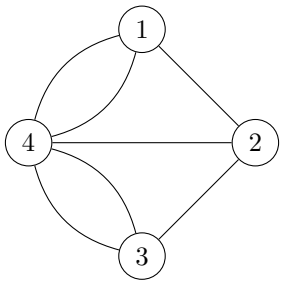

Figura 3.0.1: Grafo com arestas múltiplas das sete pontes de Königsberg.

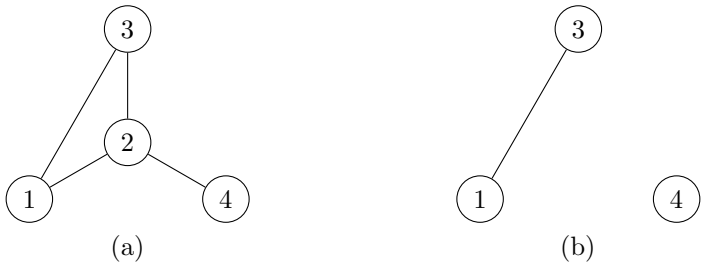

Figura 3.1.1: (a) Grafo conexo. Observe que o único ponto de articulação é o vértice 2 e a única ponte é a aresta $\{2, 4\}$. (b) Grafo desconexo. Observe que este é o grafo da figura (a) sem o ponto de articulação 2. Nenhum vértice é ponto de articulação e a única ponte é a aresta $\{1, 3\}$. As componentes são $\{1, 3\}$ e $\{4\}$.

3.1 Classificação

3.1.1 Conectividade

Um grafo simples é *conexo*, ou *conectado*, se para todo par de vértices distintos u e v existe um caminho que os conecta.

Um grafo simples é *desconexo*, ou *desconectado*, se existe um par de vértices distintos u e v tal que nenhum caminho os conecta.

Uma *componente*, em um grafo simples, é um subconjunto conexo C de vértices tal que adicionar a C qualquer vértice externo resultaria em um subconjunto desconexo. Neste caso, um grafo conexo possui uma *única* componente e um grafo desconexo possui *no mínimo* duas componentes.

Um *ponto de articulação* é um vértice tal que removê-lo aumenta o número de componentes do grafo.

Uma *ponte* é uma aresta tal que removê-la aumenta o número de componentes do grafo. Note que, ironicamente, nenhuma das sete pontes de Königsberg é de fato uma ponte!

3.1.2 Direcionamento

Um *grafo dirigido*, ou *digrafo*, é tal que uma aresta é um par ordenado (u, v), ao invés de um par não-ordenado $\{u, v\}$. Dizemos que a aresta (u, v) *sai* de u e *entra* em v, o que ela vai na *direção* de u para v. Para este tipo de grafo, é necessário adaptar as definições discutidas até agora.

O *grau de entrada* $d^-(u)$ do vértice u é o número de arestas que entram em u.

O *grau de saída* $d^+(u)$ do vértice u é o número de arestas que saem de u.

Um *caminho* é uma sequência de vértices p, de no mínimo dois elementos, tal que, para todo par de elementos adjacentes (u, v) que ocorre em p, $(u, v) \in E$. Se um caminho p é da forma $(u, ..., v)$, dizemos que p é um caminho de u *até* v, ou que u *atinge* v e v é *atingido* por u. As definições relativas a ciclos não precisam de alteração.

Seja $G = (V, E)$ um grafo dirigido qualquer e $G' = (V, E')$ um grafo simples tal que $\{u, v\} \in E'$ se, e somente se, $(u, v) \in E$ e/ou $(v, u) \in E$. Dizemos que G' é o grafo simples *subjacente* de G. Definimos a conectividade de G como a conectividade de G' e acrescentamos a definição a seguir.

Uma *componente fortemente conexa* (sigla inglesa SCC), em um grafo dirigido, é um subconjunto C de vértices tal que:

1. Para todo par de vértices distintos $u, v \in C$, u atinge v e v atinge u.
2. Se $u \notin C$, então existe $v \in C$ tal que u não atinge v e/ou v não atinge u.

Um fato interessante é que se contrairmos as componentes formente conexas de um grafo dirigido G, isto é, se criarmos um novo grafo dirigido G' tal que cada vértice de G' representa uma componente

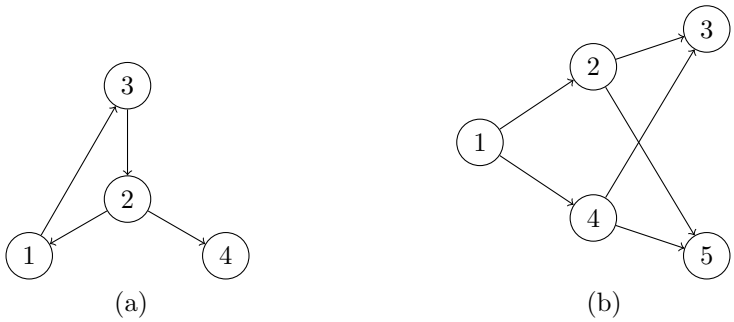

Figura 3.1.2: (a) Grafo dirigido com duas componentes formente conexas: $\{1, 2, 3\}$ e $\{4\}$. (b) Grafo dirigido com base $\{1\}$ e anti-base $\{3, 5\}$. Como a base é unitária, o vértice 1 é a raiz do grafo. Como a anti-base não é unitária, o grafo não possui anti-raiz.

formente conexa de G, este grafo G' precisa ser acíclico! Caso contrário, os vértices de cada ciclo estariam na mesma componente fortemente conexa de G e seriam representados por um único vértice em G'. Mencionamos este fato para introduzir a sigla inglesa DAG, para grafo dirigido acíclico. G' é um DAG!

Para finalizar, acrescentamos a definição de base e anti-base, útil, por exemplo, no problema do fluxo máximo (veja a Seção 3.6).

A *base* de um grafo dirigido é um subconjunto B de vértices tal que:

1. Se $u, v \in B$, então u não atinge v e v não atinge u.
2. Se $v \notin B$, então existe $u \in B$ tal que u atinge v.

Se a base de um grafo dirigido é unitária, então o vértice que a compõe é a *raiz* do grafo.

A *anti-base* de um grafo dirigido é um subconjunto A de vértices tal que:

1. Se $u, v \in A$, então u não atinge v e v não atinge u.
2. Se $u \notin A$, então existe $v \in A$ tal que u atinge v.

Se a anti-base de um grafo dirigido é unitária, então o vértice que a compõe é a *anti-raiz* do grafo.

3.1.3 Densidade

Densidade é uma definição parametrizada. Dado um parâmetro l, o *limiar de densidade*, dizemos que um grafo G é *denso* se $|E| > l$, ou que G é *esparso* em caso contrário. Escolhemos l da maneira que for mais conveniente para lidar com um problema em particular.

Para grafos simples e grafos dirigidos, é comum fazer $l = k|V|$, para algum k muito menor que $|V|$, pois o número máximo de elementos em E para um grafo simples é $|V|(|V| - 1)/2$, e $|V|(|V| - 1)$ para grafos dirigidos.

3.1.4 Árvores

Árvores possuem propriedades particularmente interessantes. As seguintes afirmações são todas equivalentes:

- T é uma árvore.
- T é um grafo simples conexo e acíclico.
- T é um grafo simples conexo com $|V| - 1$ arestas.

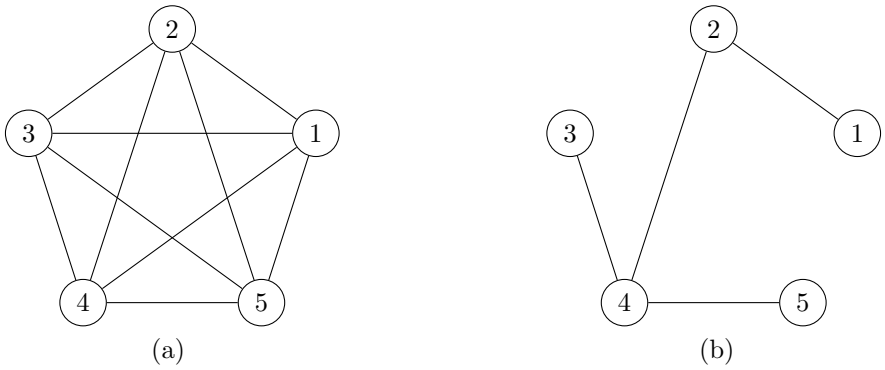

Figura 3.1.3: (a) Grafo denso. Este grafo também é conhecido como K_5 – grafo completo de 5 vértices. (b) Grafo esparso.

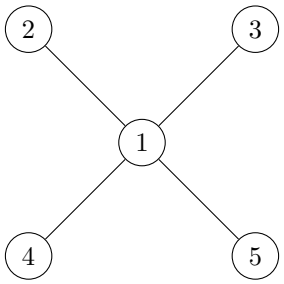

Figura 3.1.4: Árvore.

- T é um grafo simples conexo tal que toda aresta é uma ponte.
- T é um grafo simples tal que existe um único caminho da forma $(u, ..., v)$, para todo par de vértices distintos u e v.

Uma *folha* em uma árvore é qualquer vértice de grau 1.
Uma *floresta* é um grafo simples desconexo tal que cada componente é uma árvore.
Observe que toda árvore é esparsa.

3.1.5 Árvores dirigidas

Uma *árvore descendente* T é um grafo dirigido tal que:

1. T possui raiz.
2. Para todo vértice v que não é a raiz, existe uma única aresta (u, v). Dizemos que u é *pai* de v e que v é *filho* de u.

Em uma árvore descendente, os vértices da anti-base são também chamados de *folhas*.
Uma *árvore ascendente* T é um grafo dirigido tal que:

1. T possui anti-raiz.
2. Para todo vértice u que não é a anti-raiz, existe uma única aresta (u, v). Dizemos que v é *pai* de u e que u é *filho* de v.

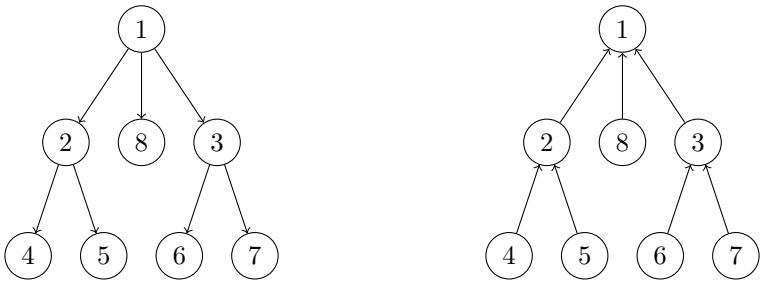

Figura 3.1.5: Árvores dirigidas duais. A árvore da esquerda é descendente, enquanto a da direita é a sua dual, ascendente.

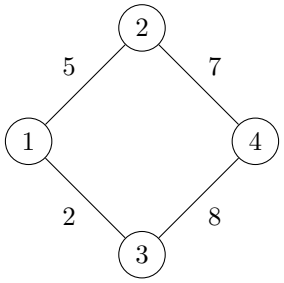

Figura 3.1.6: Grafo com peso nas arestas.

Em uma árvore ascendente, os vértices da base são também chamados de *folhas*.

Note que uma árvore descendente/ascendente também possui exatamente $|V|-1$ arestas (esparsa), é acíclica e seu grafo simples subjacente é uma árvore. Além disso, qualquer árvore descendente/ascendente possui uma única árvore ascendente/descendente associada, obtida pela inversão da direção de cada aresta.

3.1.6 Peso

Um *grafo com peso nas arestas*, ou *grafo ponderado*, é tal que usamos uma função $w : E \longrightarrow \mathbb{R}$ para definir o peso de cada aresta.

3.1.7 Laços

Um *grafo com laços* é tal que a restrição $u \neq v$ para uma aresta $\{u, v\}$, ou (u, v), é desconsiderada.

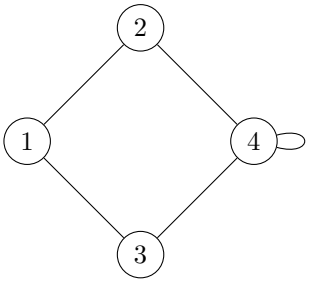

Figura 3.1.7: Grafo com o laço $\{4\}$.

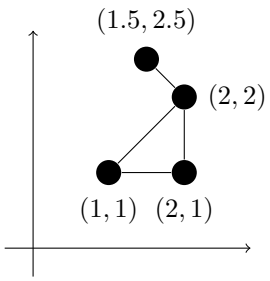

Figura 3.1.8: Grafo embutido.

3.1.8 Arestas múltiplas

Um *grafo com arestas múltiplas* $G = (V, E)$, ou *multigrafo*, é tal que E é um multiconjunto de arestas, como o grafo das sete pontes de Königsberg, apresentado no início do capítulo.

3.1.9 Grafo embutido

Um grafo é *embutido* se seus vértices são pontos geométricos. Um grafo que não é embutido é *topológico*.

3.1.10 Grafo implícito

Um *grafo implícito* é tal que as arestas incidentes a um determinado vértice são definidas a partir de uma fórmula. Em outras palavras, um grafo implícito dispensa uma estrutura de dados para armazenar suas arestas.

Uma busca completa, por exemplo, é uma travessia em um grafo implícito. A computação de uma programação dinâmica também é uma travessia em um grafo implícito, mas neste caso o grafo é também um DAG.

Outro exemplo é o grafo associado a uma matriz/grade que representa um mapa, ou o tabuleiro de um jogo. Neste caso, dado o vértice $u = (i, j)$, as arestas incidentes a u são as que conectam u com posições adjacentes, ou seja, podem ser calculadas a partir de i e j; e, geralmente, são no máximo oito arestas (o grafo é esparso).

3.2 Estruturas de dados para grafos

Neste capítulo, você verá diversos laços como o do Algoritmo 3.2.1. Neste trecho de código, estamos passando por cada aresta incidente a um vértice u. O custo deste laço depende da estrutura de dados que usamos para representar o grafo. Esta seção apresenta as três estruturas de dados usadas mais comumente para este propósito. Geralmente, é melhor representar grafos esparsos com *lista de adjacência* e grafos densos com *matriz de adjacência*, embora alguns algoritmos sejam mais facilmente implementados se uma matriz de adjacência for utilizada, independentemente da densidade do grafo, como o algoritmo Floyd-Warshall para caminhos mínimos (veja a Seção 3.5.5) e o algoritmo Edmonds-Karp para fluxo máximo (veja a Seção 3.6.1). Para árvores ascendentes, a melhor estrutura de dados é o *vetor de vértice pai*.

Algoritmo 3.2.1 Explorando os vértices vizinhos de u.

```
1: Um vértice u ∈ V é fixado.
2: para cada {u, v} ∈ E faça
3:     Algo é feito com o vértice v.
4: fim para
```

3.2.1 Lista de adjacência

Ao representar um grafo com lista de adjacência, para cada vértice $u \in V$, temos uma lista com as arestas incidentes a (ou que saem de) u. Estas listas podem ser de fato listas encadeadas (veja a Seção 1.5.3), ou podem ser vetores dinâmicos (veja a Seção 1.5.2). Podemos utilizar uma lista de números inteiros, ou uma lista de pares ordenados caso o grafo seja ponderado.

Utilizando esta representação, o número de iterações do laço do Algoritmo 3.2.1 é $O(d(u))$.

3.2.2 Matriz de adjacência

É comum utilizar uma matriz (arranjo bidimensional) $G = (g_{ij})$ para representar grafos densos. Uma entrada g_{ij} pode ser o peso da aresta (i, j). Logo, $g_{ii} = 0$ e $g_{ij} = \infty$, se $(i, j) \notin E$. Alguns algoritmos em multigrafos podem pedir que uma entrada g_{ij} seja o número de arestas (i, j).

Utilizando esta representação, o número de iterações do laço do Algoritmo 3.2.1 é $O(|V|)$.

3.2.3 Vetor de vértice pai

Esta representação só pode ser utilizada para árvores. Se o algoritmo faz travessias que sobem dos vértices até a raiz, esta é a melhor estrutura de dados a ser usada. Para cada vértice $u \in V$, temos um valor $pai(u)$, que determina que a aresta $\{u, pai(u)\}$ pertence a E.

Neste caso, o laço do Algoritmo 3.2.1 não pode ser utilizado.

3.3 Travessia

Muitos problemas em grafos dependem de encontrar caminhos entre pares de vértices, ou entre um vértice e todos os outros. Chamamos esta classe de problemas de *problemas de travessia em grafos*.

Há duas maneiras simples de se atravessar um grafo: *busca em profundidade* (*depth-first search*, ou simplesmente DFS) e *busca em largura* (*breadth-first search*, ou simplesmente BFS). Ambas recebem como entrada um grafo $G = (V, E)$ e um *vértice fonte* $s \in V$ e retornam como saída uma árvore de extensão para a componente de s — uma DFS/BFS *spanning tree*. O conjunto de arestas é $\{\{u, pai(u)\} | u, pai(u) \in V\}$.

As seções a seguir apresentam detalhes sobre DFS e BFS e discutem aplicações e variações destes algoritmos.

3.3.1 Busca em profundidade

O Algoritmo 3.3.1 mostra a forma geral de uma busca em profundidade. Dizemos que este algoritmo é uma busca em profundidade, porque, para um vértice fonte u, ele visita uma subárvore de u de cada vez, recursivamente. Isto tende a formar caminhos mais longos, ou mais "profundos" (em número de arestas). Após ir até o fim de um ramo da DFS *spanning tree*, a busca faz um retrocesso (sim, uma DFS é um *backtracking*. Veja a Seção 2.1.1) e tenta visitar outro ramo. Se $cor(u) =$ BRANCO, o vértice u ainda não foi visitado. Se $cor(u) =$ CINZA, estamos visitando uma das subárvores de u neste momento. Se $cor(u) =$ PRETO, todas as subárvores de u já foram visitadas anteriormente. Neste caso, observe que cada vértice é visitado no máximo uma vez e que visitar um vértice u é o mesmo que executar DFS(u). Logo, o custo de tempo de uma busca em profundidade é $O(\sum_{u \in V} d(u)) = O(|E|)$.

Algoritmo 3.3.1 Busca em profundidade.

```
1:  função DFS(u : vértice)
2:      cor(u) ← CINZA
3:      para cada {u, v} ∈ E faça
4:          se cor(v) = BRANCO então
5:              pai(v) ← u
6:              DFS(v)
7:          fim se
8:      fim para
9:      cor(u) ← PRETO
10: fim função
11: para cada u ∈ V faça
12:     cor(u) ← BRANCO
13: fim para
14: pai(s) ← nulo
15: DFS(s)
```

3.3.2 Busca em largura

O Algoritmo 3.3.2 mostra a forma geral de uma busca em largura. Dizemos que este algoritmo é uma busca em largura, porque vértices mais próximos da fonte s (em número de arestas) são visitados primeiro, então é como se a "largura" da BFS *spanning tree* aumentasse gradativamente. Esta é uma propriedade extremamente interessante, que permite que vários problemas sejam resolvidos com facilidade. Por exemplo: em programação dinâmica é necessário resolver problemas menores primeiro, para então resolver problemas maiores. Logo, BFS é uma ótima escolha para problemas

de programação dinâmica[1]. Além disso, note que uma BFS *spanning tree* é a união dos caminhos mínimos (em número de arestas) que começam com a fonte s. Logo, para saber o menor caminho (em número de arestas) entre um par de vértices, basta realizar uma BFS usando um dos dois vértices como fonte e interromper a busca assim que o outro for atingido. Note que se $cor(u) =$ BRANCO, o vértice u ainda não foi visitado. Se $cor(u) =$ CINZA, o vértice u já foi descoberto, mas ainda não foi visitado. Se $cor(u) =$ PRETO, o vértice u já foi visitado. Note ainda que um vértice u ser visitado é o mesmo que u ser retirado da fila Q em uma iteração do laço "enquanto". Como um vértice é marcado como CINZA assim que é descoberto, ele só pode ser enfileirado uma única vez. Logo, cada vértice é visitado no máximo uma vez. Então, o custo de tempo de uma BFS é $O(\sum_{u \in V} d(u)) = O(|E|)$.

Algoritmo 3.3.2 Busca em largura.

```
1: para cada u ∈ V faça
2:     cor(u) ← BRANCO
3: fim para
4: cor(s) ← CINZA
5: pai(s) ← nulo
6: Q ← ∅                                          ▷ uma fila é necessária
7: ENFILEIRAR(Q, s)
8: enquanto Q ≠ ∅ faça
9:     u ← DESENFILEIRAR(Q)
10:    cor(u) ← PRETO
11:    para cada {u, v} ∈ E faça
12:        se cor(v) = BRANCO então
13:            cor(v) ← CINZA
14:            pai(v) ← u
15:            ENFILEIRAR(Q, v)
16:        fim se
17:    fim para
18: fim enquanto
```

3.3.3 Encontrando componentes

Para encontrar as componentes de um grafo, podemos estender qualquer um dos dois algoritmos de travessia. Se a saída é uma *spanning tree* da componente do vértice fonte s, basta executar travessias para os vértices que ainda não foram visitados. O Algoritmo 3.3.3 ilustra essa ideia. Note que o custo de tempo é $O(|E|)$.

Algoritmo 3.3.3 Encontrando as componentes de um grafo.

```
1: para cada u ∈ V faça
2:     cor(u) ← BRANCO
3: fim para
4: C ← 0
5: para cada u ∈ V faça
6:     se cor(u) = BRANCO então
7:         C ← C + 1
8:         Fazer uma travessia a partir de u. Para cada v visitado, faça componente(v) ← C.
9:     fim se
10: fim para
```

[1] Problema 1742 do URI OJ.

3.3.4 Verificação de grafo bipartido

Primeiro, um *grafo bipartido* é tal que seu conjunto de vértices é da forma $V = L \cup R$, onde $L \cap R = \emptyset$ e para qualquer $\{u, v\} \in E$, ou $u \in L$ e $v \in R$, ou $v \in L$ e $u \in R$. Em linguagem natural, o conjunto de vértices de um grafo bipartido pode ser particionado em dois conjuntos L (esquerdo) e R (direito) de forma que qualquer aresta é incidente a um vértice de L e a um vértice de R.

Para verificar se um grafo é bipartido, qualquer algoritmo de travessia é suficiente. Assim que descobrimos um novo vértice v, na vizinhança de um vértice u, determinamos que o seu conjunto é o oposto do de u. Se encontrarmos um vértice v, na vizinhança de um vértice u, que já está em algum conjunto, verificamos se este conjunto é o oposto do de u. Se não for (se u e v estão no mesmo conjunto), o grafo não é bipartido e interrompemos o algoritmo. Se, ao final da travessia, nenhuma contradição desse tipo foi encontrada, o grafo é bipartido. O Algoritmo 3.3.4 é uma adaptação de uma DFS que implementa esta ideia. Usamos CINZA para o conjunto esquerdo e PRETO para o conjunto direito. Note que o custo de tempo permanece $O(|E|)$.

Algoritmo 3.3.4 Verificando se um grafo é bipartido.

```
1: função DFS(u : vértice)
2:     se cor(u) = CINZA então                ▷ determinando c como o conjunto oposto do de u
3:         c ← PRETO
4:     senão
5:         c ← CINZA
6:     fim se
7:     para cada {u, v} ∈ E faça
8:         se cor(v) = BRANCO então
9:             cor(v) ← c
10:            se DFS(v) = falso então
11:                retorne falso
12:            fim se
13:        senão se cor(v) = cor(u) então
14:            retorne falso
15:        fim se
16:    fim para
17:    retorne verdadeiro
18: fim função
19: para cada u ∈ V faça
20:    cor(u) ← BRANCO
21: fim para
22: cor(s) ← CINZA
23: se DFS(s) = verdadeiro então
24:    O grafo é bipartido.
25: senão
26:    O grafo não é bipartido.
27: fim se
```

3.3.5 Encontrando ciclos

Sempre é possível detectar se um grafo possui algum ciclo, ou não. Basta fazer uma busca em profundidade. Ao encontrar um vértice marcado como CINZA, encontramos um ciclo[2]. Por outro lado, encontrar *todos* os ciclos de um grafo qualquer não é tão simples, ainda que seja imposta a

[2]Parte da solução para o problema 1792 do URI OJ.

restrição, por exemplo, de que um ciclo só é de fato um ciclo se ele não contém subciclos menores dentro (neste caso, (u, v, u, v, u) não seria um ciclo). Para ilustrar esta dificuldade, vamos contar quantos ciclos sem subciclos existem no grafo completo K_n. Observe que qualquer ciclo sem subciclos em K_n deve ser da forma $(u, v_1, v_2, ..., v_k, u)$, onde $1 \leq k < n$ e, para qualquer par de números i e j distintos, $v_i \neq v_j$ e $u \neq v_i$. Logo, o número que procuramos é $\Omega(\sum_{k=1}^{n-1} k!) = \Omega(n!)$. Agora, reflita: qualquer algoritmo que encontra todos os ciclos sem subciclos de K_n deve ter complexidade $\Omega(n!)$.

Sob algumas restrições, é fácil encontrar todos os ciclos de um grafo. Se é garantido, por exemplo, que: (1) qualquer vértice está em no máximo um ciclo[3]; ou que (2) sai no máximo uma aresta de cada vértice em um grafo dirigido[4]. Note que (2) $\implies$ (1). Quando (1) ocorre, basta fazer o procedimento para detecção de ciclos. Ao encontrar um ciclo, armazene-o! O Algoritmo 3.3.5 mostra como isso pode ser feito, sem alterar o custo de tempo $O(|E|)$ de uma DFS.

Algoritmo 3.3.5 Detecção de ciclos em um grafo onde cada vértice está em no máximo um ciclo.

```
1: função DFS(u : vértice)
2:     cor(u) ← CINZA
3:     para cada {u, v} ∈ E faça
4:         se cor(v) = CINZA então
5:             C ← C + 1
6:             t ← u
7:             enquanto t ≠ v faça
8:                 ciclo(t) ← C
9:                 t ← pai(t)
10:            fim enquanto
11:            ciclo(v) ← C
12:        senão se cor(v) = BRANCO então
13:            pai(v) ← u
14:            DFS(v)
15:        fim se
16:    fim para
17:    cor(u) ← PRETO
18: fim função
19: para cada u ∈ V faça
20:    cor(u) ← BRANCO
21:    ciclo(u) ← 0
22: fim para
23: pai(s) ← nulo
24: C ← 0                          ▷ variável global para identificar o próximo ciclo
25: DFS(s)
```

3.3.6 Ordenação topológica

Uma *ordenação topológica* s de um grafo dirigido $G = (V, E)$ é uma sequência com todos os vértices de G que satisfaz a seguinte propriedade: se $(u, v) \in E$, então u aparece antes de v em s. Note que um grafo dirigido possui uma ordenação topológica se, e somente se, ele for também acíclico, ou seja, somente DAGs possuem ordenação topológica.

Para obter uma ordenação topológica, basta incluir uma pequena modificação na DFS. Após marcar um vértice como PRETO, empilhe este vértice. Após a DFS, a extração dos vértices empilhados resulta em uma ordenação topológica. Logo, o custo de tempo permanece $O(|E|)$.

[3]Ocorre no problema 1711 do URI OJ.
[4]Ocorre no problema 1902 do URI OJ.

3.3.7 Encontrando componentes fortemente conexas

Antes de seguir com o algoritmo que encontra as componentes fortemente conexas de um grafo dirigido (veja a Seção 3.1.2), precisamos da seguinte definição. Seja $G = (V, E)$ um grafo dirigido. Dizemos que $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$ é o *grafo transposto* de G, onde $E^{\mathrm{T}} = \{(u, v) | (v, u) \in E\}$, ou seja, G^{T} é o grafo obtido invertendo a direção de cada aresta de G.

Algoritmo 3.3.6 Algoritmo de Kosaraju.

```
1: S ← ∅                                              ▷ pilha global de vértices
2: Faça uma busca em profundidade em G. Após marcar u como PRETO, empilhe u em S.
3: Calcule G^T.
4: SCC ← 0                                            ▷ identificador global da SCC atual
5: para cada u ∈ V faça
6:     scc(v) ← 0
7: fim para
8: enquanto S ≠ ∅ faça                                ▷ uma segunda travessia é necessária
9:     u ← DESEMPILHAR(S)
10:    se u ainda não foi visitado na segunda travessia então
11:        SCC ← SCC + 1
12:        Faça uma travessia em G^T a partir de u. Para cada v visitado, scc(v) ← SCC.
13:    fim se
14: fim enquanto
```

O Algoritmo 3.3.6, conhecido como algoritmo de Kosaraju, encontra a SCC de cada vértice de um grafo dirigido G de entrada. Se quisermos também construir o grafo contraído de G, basta armazenar a aresta $(scc(v), SCC)$, sempre que encontramos, na segunda travessia, um vértice v tal que $scc(v) > 0$ e $scc(v) \neq SCC$, ou seja, um vértice que já foi visitado na segunda travessia anteriormente e pertence a outra SCC. A complexidade de tempo do algoritmo é claramente $\Theta(|V| + |E|)$.

Para provar a corretude do algoritmo de Kosaraju, precisamos provar que cada árvore de extensão encontrada pela travessia em G^{T} é uma SCC de G. Esta prova segue por indução no número de árvores de extensão encontradas. A base é $k = 0$ (por vacuidade, as zero primeiras árvores de extensão são SCCs). A hipótese indutiva é: as k primeiras árvores de extensão são SCCs. O passo indutivo deve usar esta hipótese para provar que a $(k+1)$-ésima árvore de extensão, T_{k+1}, encontrada pela travessia em G^{T}, é também uma SCC de G. Para isto, considere a SCC C da raiz de T_{k+1}. Se $u \in C$, então é claro que u aparece em T_{k+1}. Resta mostrar que se u aparece em T_{k+1}, então $u \in C$. Para obter uma contradição, suponha que $u \notin C$. Neste caso, u pertence a outra SCC C'. É claro que C' não é nenhuma das k primeiras árvores, pois u aparece em T_{k+1}. Chegamos ao detalhe crucial da prova. A busca em profundidade, ao empilhar um vértice somente após explorar todas as suas subárvores, cria uma ordem para os vértices de G. Agora, note que se considerarmos o grafo contraído de G, esta ordem é uma ordenação topológica (veja a Seção 3.3.6)! Logo, qualquer aresta de G^{T} que sai de C para outra SCC deve ser sim para uma das SCCs que já foram encontradas! Um absurdo!

3.3.8 Empacotamento de grafos

Considere o grafo simples da Figura 3.3.1.a. Chame-o de G. Imagine que impomos a seguinte restrição: em um caminho, as arestas incidentes ao vértice 2 só podem ser usadas se o vértice 4 for visitado antes. Neste caso, qual é o menor caminho (em número de arestas) de 1 a 3? A resposta é: $1 - 4 - 1 - 2 - 3$. Mas qual algoritmo poderia encontrar esta resposta? Uma simples BFS em G encontraria esta resposta? Não. Uma simples BFS em G daria uma resposta incorreta: $1 - 2 - 3$. Para resolver este problema, precisamos utilizar uma abstração bastante interessante. Podemos pensar da seguinte forma. Em qualquer ponto de um caminho qualquer, ou já passamos pelo vértice 4, ou não. Vamos pensar em cada uma destas duas possibilidades como um possível *estado* s. Vamos dar símbolos

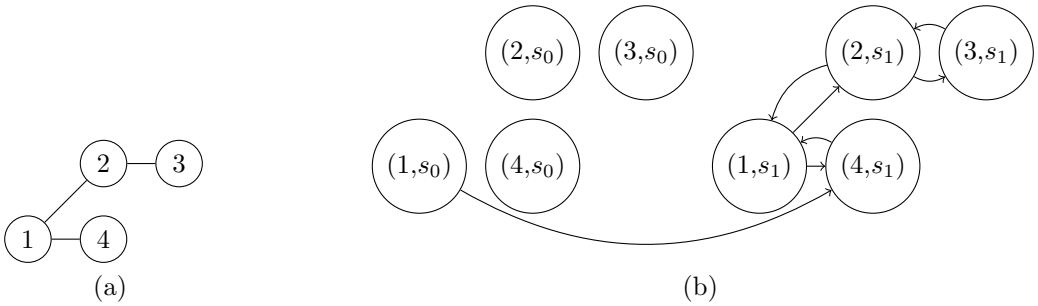

Figura 3.3.1: Exemplo de empacotamento. No grafo da Figura (a), só é permitido entrar no vértice 2, se o vértice 4 aparecer antes no caminho. Logo, o caminho de menos arestas de 1 a 3 é $1-4-1-2-3$. A Figura (b) ilustra o grafo "empacotado", resultado de escolher o conjunto de vértices como o produto cartesiano $V \times S$, onde $S = \{s_0, s_1\}$ é o conjunto de possíveis estados.

para estes dois estados. Se $s = s_0$, é porque ainda não passamos pelo vértice 4. Se $s = s_1$, é porque já passamos pelo vértice 4. Agora, seja $S = \{s_0, s_1\}$ o conjunto dos possíveis estados e considere o produto cartesiano $V' = V \times S = \{(1, s_0), (2, s_0), (3, s_0), (4, s_0), (1, s_1), (2, s_1), (3, s_1), (4, s_1)\}$. Se construímos o conjunto de arestas E' apropriado, uma simples BFS no grafo dirigido $G' = (V', E')$ resolve o problema, ou seja, encontra o caminho mínimo (em número de arestas) $(1, s_0) - (4, s_1) - (1, s_1) - (2, s_1) - (3, s_1)$. A Figura 3.3.1.b ilustra o grafo G', um "empacotamento" de G com a restrição que impusemos. Observe que o grafo G' é um *grafo parcialmente implícito*, ou seja, não é necessário construir de fato o grafo empacotado na memória do computador, não é necessário armazenar todas as suas arestas em alguma estrutura de dados real. Basta realizar transições de forma apropriada, atravessando a estrutura de dados que armazena as aresta de G e verificando quais delas de fato podem ser usadas, de acordo com o estado s do vértice $(u, s) \in V'$ que está sendo explorado.

Em geral, podemos descrever empacotamento de grafos da seguinte forma. Seja $G = (V, E)$ um grafo e de $R(G)$ um conjunto de *restrições de travessia* sobre G. Após derivar o conjunto $S(G)$ dos possíveis *estados de travessia* de G, a partir de G e $R(G)$, podemos construir o *grafo empacotado* $G' = (V', E')$, onde:

1. $V' = V \times S(G)$
2. $E' = \{((u, s_i), (v, s_j)) | \{u, v\} \in E, s_i, s_j \in S(G)$ e (u, v, s_i) satisfaz $R(G)$ transitando para $s_j\}$.

Em resumo, o trabalho de se empacotar um grafo com suas restrições de travessia é apenas identificar quais são os possíveis estados de travessia. A limitação desta técnica é que o número total de estados não pode ser muito grande. Geralmente, os estados são descritos por máscaras de *bits*. Se a máscara possui n *bits*, a complexidade de espaço de qualquer algoritmo (não necessariamente um algoritmo de travessia) que utiliza empacotamento é $\Omega(|V|2^n)$. Portanto, cuidado!

3.4 Árvore de extensão mínima

Um problema famoso e bastante recorrente em competições é encontrar uma *árvore de extensão mínima* para um grafo, ou uma MST (sigla inglesa). Uma MST é uma árvore de extensão (veja a Seção 3.3) tal que a soma dos pesos de suas arestas é mínima. Note que para alguns grafos é possível encontrar mais de uma MST. Por exemplo: qualquer grafo conexo, cíclico e sem peso nas arestas (caso em que atribuímos um mesmo peso qualquer para todas as arestas). Nesta seção, discutimos três algoritmos gulosos que solucionam este problema e aproveitamos para discutir algumas estruturas de dados bastante interessantes, que são utilizadas nestas soluções.

3.4.1 Algoritmo de Prim

O Algoritmo 3.4.1, conhecido como algoritmo de Prim, cresce uma árvore de extensão mínima adicionando uma nova aresta a cada escolha. A escolha gulosa é escolher exatamente a aresta segura mais leve. Uma aresta segura é uma aresta que não cria um ciclo na MST que o algoritmo está crescendo. A entrada é um grafo $G = (V, E)$ e um vértice $s \in V$. A saída é uma árvore com conjunto de arestas $\{\{u, pai(u)\} | u, pai(u) \in V\}$.

Algoritmo 3.4.1 Algoritmo de Prim.

```
1:  Q ← ∅ ▷ Uma fila de prioridade é necessária. Os elementos são arestas direcionadas. Uma aresta
    e tem mais prioridade que uma aresta e', se w(e) < w(e').
2:  ENFILEIRAR(Q, (nulo, s))
3:  para cada u ∈ V faça
4:      visitado(u) ← falso
5:  fim para
6:  enquanto Q ≠ ∅ faça
7:      (p, u) ← DESENFILEIRAR(Q)
8:      se visitado(u) = falso então
9:          visitado(u) ← verdadeiro
10:         pai(u) ← p
11:         para cada {u, v} ∈ E faça
12:             se visitado(v) = falso então
13:                 ENFILEIRAR(Q, (u, v))
14:             fim se
15:         fim para
16:     fim se
17: fim enquanto
```

Analisar o custo de tempo do algoritmo de Prim não é simples. Primeiro, precisamos notar que as linhas de 9 a 15 são executadas exatamente $|V|$ vezes, pois cada vértice é visitado exatamente uma vez. Ou seja, por mais que o laço da linha 6 execute, digamos, A vezes, as linhas de 9 a 15 serão executadas apenas $|V| \leq A$ vezes. Então, para facilitar a análise, devemos considerar os custos das linhas de 9 a 15 separadamente, imaginando que elas estão fora do laço da linha 6. Note que, para cada vértice $u \in V$, passamos por cada aresta incidente a u. Logo, estamos simplesmente passando por todas as arestas do grafo. Agora, note que: ou enfileiramos uma aresta, ou não. Então, o custo das linhas de 9 a 15, no pior caso, é $\Theta(|E| \lg |E|)$. Voltando ao laço da linha 6, desconsideramos os custos das linhas de 9 a 15. O que resta é desenfileirar cada elemento de Q. Portanto, o custo do laço principal também é $O(|E| \lg |E|)$. Portanto, o custo total é $O(2(|E| \lg |E|)) = O(|E| \lg |E|)$. Por fim, vamos lembrar que $|E| = O(|V|^2) \implies \lg |E| = O(2 \lg |V|) = O(\lg |V|)$. Logo, o custo de tempo do algoritmo de Prim é $O(|E| \lg |V|)$.

A corretude do algoritmo de Prim segue da seguinte *invariante de laço*[5]. Antes da linha 7 executar, temos uma MST T com os vértices de G que já foram visitados. Durante uma iteração, a aresta direcionada mais leve (p, u) da fila Q é extraída. Se u já foi visitado, nada é feito e continuamos com a MST T. Se u ainda não foi visitado, marcamos u como visitado e adicionamos $\{p, u\}$ a T, formando uma nova árvore de extensão T'. Argumentamos que T' é uma MST com o laço que vem a seguir. Para cada vizinho v de u que ainda não foi visitado, enfileiramos a aresta direcionada (u, v). Logo, se todos os vértices de T já haviam sido visitados, todas as arestas incidentes a u e a algum vértice de T já haviam sido descobertas (enfileiradas). Então, é claro que T' é uma MST. Ao final do laço, é claro que todos os vértices da componente de s foram visitados, então é claro que a saída é uma MST com todos os vértices da componente de s.

3.4.2 Conjuntos disjuntos união-busca e o algoritmo de Kruskal

Antes de discutir o algoritmo de Kruskal, precisamos voltar nossa atenção rapidamente para uma pequena, simples e rápida estrutura de dados: *conjuntos disjuntos união-busca* (em inglês: *union-find disjoint sets*), ou simplesmente *union-find*. Como o nome sugere, esta ED gerencia um conjunto de conjuntos disjuntos, ou seja, um conjunto $\mathbb{U}$ de conjuntos tal que, para quaisquer $A, B \in \mathbb{U}$ distintos, $A \cap B = \emptyset$. Novamente como o nome sugere, as duas operações permitidas são FIND-SET(a), que retorna o elemento representante x tal que $a, x \in A$, e UNION(a, b), que une os dois conjuntos $A, B \in \mathbb{U}$ tais que $a \in A$ e $b \in B$. O Algoritmo 3.4.2 mostra a implementação desta ED. Representamos cada conjunto $A \in \mathbb{U}$ como uma árvore ascendente e elegemos como representante principal aquele elemento que for a anti-raiz. A operação UNION() faz duas chamadas à operação FIND-SET(), logo ambas as operações tem o mesmo custo assintótico de tempo. É possível mostrar que este custo é $O(\alpha(n))$ amortizado[6], onde $n = O(\sum_{A \in \mathbb{U}} |A|)$ e $\alpha(n)$ é a função inversa da função de Ackermann[7]: $A(n, n)$. Este custo de tempo é alcançado pela técnica *compressão de caminho*. Sempre que precisamos subir de um nó até a anti-raiz, fazemos o caminho de volta (descemos) e atualizamos o nó pai de todos os nós como a anti-raiz.

Algoritmo 3.4.2 Estrutura de dados *union-find*.

```
1:  função FIND-SET(a : inteiro)
2:      se pai(a) = a então
3:          retorne a
4:      fim se
5:      pai(a) ← FIND-SET(pai(a))
6:      retorne pai(a)
7:  fim função
8:  função UNION(a, b : inteiros)
9:      pai(FIND-SET(a)) ← FIND-SET(b)
10: fim função
```

Utilizando uma ED *union-find*, podemos encontrar uma floresta de extensão mínima com um algoritmo guloso extremamente simples e elegante, conhecido como algoritmo de Kruskal (Algoritmo 3.4.3). Basta tentar incluir cada aresta do grafo na floresta, percorrendo por ordem não-decrescente de peso. Fazemos isto verificando se os vértices de uma aresta estão no mesmo conjunto. Se não estiverem, incluímos a aresta e atualizamos o *union-find*. O custo total é $O(|V| + |E| \lg |E| + |E|\ \alpha(|V|)) = O(|E| \lg |V|)$. A saída é uma floresta com o conjunto de arestas E'.

[5] Propriedade que vale antes e depois de uma iteração de um laço.

[6] Quando o custo de uma operação em uma estrutura de dados é $O(M)$ amortizado, uma única execução desta operação pode custar mais, ou menos, que M. Mas N execuções não custam mais que $O(NM)$.

[7] A função de Ackermann é uma função que cresce absurdamente rápido. Sua inversa cresce muito mais lentamente que um logaritmo.

Algoritmo 3.4.3 Algoritmo de Kruskal.

```
1: para cada u ∈ V faça
2:     pai(u) ← u
3: fim para
4: Ordene E por ordem não-decrescente de peso.
5: E' ← ∅
6: pesototal ← 0
7: para cada {u, v} ∈ E faça
8:     se FIND-SET(u) ≠ FIND-SET(v) então
9:         UNION(u, v)
10:        E' ← E' ∪ {{u, v}}
11:        pesototal ← pesototal + w({u, v})
12:    fim se
13: fim para
```

Tabela 3.4.1: Operações com link/cut trees.

Operação	Custo de tempo
LINK(u, v)	$O(\lg n)$ amortizado
CUT(u, v)	$O(\lg n)$ amortizado
ROOT(u) : r	$O(\lg n)$ amortizado
MAKE-ROOT(u)	$O(\lg n)$ amortizado
PATH(u, v) : x	$O(\lg n)$ amortizado

3.4.3 Melhorando o algoritmo de Kruskal com link/cut tree

Nesta seção, apresentamos uma estrutura de dados para árvores dinâmicas, conhecida como *link/cut tree*, inventada por Daniel Dominic Sleator e Robert Endre Tarjan. Apresentamos apenas as operações da estrutura de dados e deixamos a implementação para outro texto.

Uma link/cut tree mantém informação sobre uma floresta, ou seja, um conjunto de árvores. Assumindo que criamos uma link/cut tree com n vértices, a Tabela 3.4.1 mostra as operações que podem ser feitas. A operação LINK(u, v) cria uma aresta entre os vértices u e v, assumindo que não existe aresta entre eles. A operação CUT(u, v) remove a aresta que liga os vértices u e v, assumindo que esta aresta de fato existe. A operação ROOT(u) retorna o vértice que é raiz da árvore a qual u pertence. A operação MAKE-ROOT(u) faz o vértice u se tornar a raiz da árvore a qual ele pertence. A operação PATH(u, v) retorna o resultado de alguma função do caminho que conecta u e v, assumindo que este caminho de fato existe. Esta função pode ser, por exemplo, um valor mínimo/máximo, ou uma soma, dos vértices pelo caminho, ou das arestas. Uma maneira fácil de modificar uma link/cut tree para manter informações sobre arestas é criar um vértice "virtual" para cada aresta. Note que esta estrutura de dados resolve o problema de conectividade dinâmica em árvores, com consultas de informações sobre *caminhos*. Uma estrutura de dados bastante semelhante é a *euler tour tree*, que possibilita consultas de informações sobre *subárvores*.

O Algoritmo 3.4.4 é uma adaptação do algoritmo de Kruskal. Se os vértices de uma nova aresta $\{u, v\}$ estão em árvores separadas, incluímos esta aresta. Se os vértices estão na mesma árvore, consultamos a aresta $\{x, y\}$ mais pesada do caminho que os conecta. Se ela for mais pesada que a aresta que estamos considerando incluir, cortamos $\{x, y\}$ e ligamos $\{u, v\}$. Caso contrário, descartamos a aresta $\{u, v\}$. Note que o custo de tempo é o mesmo do algoritmo da seção anterior, $O(|E| \lg |V|)$, mas não é mais necessário ordenar as arestas. Isto é bom para problemas que pedem o peso de uma árvore de extensão mínima a cada nova aresta que é dada na entrada.

Algoritmo 3.4.4 Algoritmo de Kruskal com link/cut tree.

1: Inicializamos uma link/cut tree T com os vértices de G. Observe que, neste ponto, a floresta T possui $|V|$ árvores, ou seja, cada vértice de G é raiz de sua própria árvore em T.
2: **para cada** $\{u, v\} \in E$ **faça**
3: **se** ROOT$(u) \neq$ ROOT(v) **então**
4: LINK(u, v)
5: **senão**
6: $\{x, y\} \leftarrow$ PATH(u, v)
7: **se** $w(\{x, y\}) > w(\{u, v\})$ **então**
8: CUT(x, y)
9: LINK(u, v)
10: **fim se**
11: **fim se**
12: **fim para**

3.5 Problema do caminho mínimo

vei

3.5.1 Algoritmo de Dijkstra

vei

3.5.2 Algoritmo Bellman-Ford

vei

3.5.3 Exponenciação da matriz de adjacência

vei

3.5.4 Contagem de caminhos

vei

3.5.5 Algoritmo Floyd-Warshall

vei

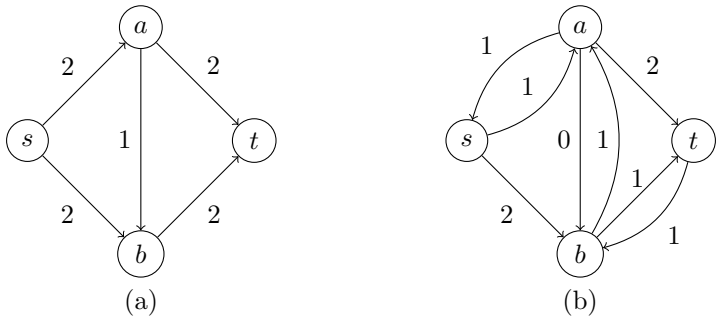

Figura 3.6.1: (a) Rede em que o procedimento incompleto de enviar fluxo pode gerar uma resposta incorreta. Se o caminho $s \to a \to b \to t$ for encontrado antes de qualquer outro, enviar um fluxo apenas subtraindo 1 dos pesos das arestas (s, a), (a, b) e (b, t) irá impedir que o fluxo máximo, que é 4, seja atingido. (b) Grafo da Figura (a) após a primeira iteração do método Ford-Fulkerson, onde o caminho $s \to a \to b \to t$ foi o primeiro a ser encontrado: subtrai-se 1 das arestas (s, a), (a, b) e (b, t) e soma-se 1 às arestas (t, b), (b, a) e (a, s).

3.6 Problema do fluxo máximo

Considere um grafo dirigido e ponderado, com dois vértices especiais: a fonte s (em inglês, *source*) e o sorvedouro t (em inglês, *sink*). Imagine este grafo como uma rede de distribuição de água, onde as arestas são os canos por onde a água passa, os vértices são aonde os canos se dividem e os pesos são a capacidade de cada cano (um diâmetro, por exemplo). O *problema do fluxo máximo* é determinar o volume máximo por unidade de tempo que pode passar por esta rede. Esta seção apresenta alguns algoritmos que resolvem este problema e alguns casos especiais de particular interesse, como *casamento máximo* e *cobertura mínima de vértices*. Aqui consideramos apenas pesos inteiros e positivos.

3.6.1 Método Ford-Fulkerson e algoritmo Edmonds-Karp

O método Ford-Fulkerson é basicamente o seguinte: enquanto for possível encontrar um caminho de aumento P, envie um fluxo por este caminho P. Um *caminho de aumento* começa no vértice s, termina no vértice t e, para toda aresta e que o compõe, $w(e) > 0$. Chamamos este procedimento de método, porque ele não especifica como o caminho de aumento deve ser encontrado (podemos provar sua corretude, mas não podemos analisar sua complexidade).

Para discutir o método Ford-Fulkerson, vamos observar primeiro que qualquer caminho de aumento possui pelo menos um *gargalo*, uma aresta de peso mínimo. O peso desta aresta é o fluxo máximo que pode passar por este caminho. Então, intuitivamente, enviar um fluxo por um caminho de aumento P seria: encontrar o peso mínimo $w(e)$ de uma aresta e que ocorre em P, subtrair $w(e)$ do peso de todas as arestas de P e somar $w(e)$ ao fluxo máximo total $|f^*|$. Acontece que isto não é suficiente. O grafo da Figura 3.6.1.a é um exemplo em que este procedimento incompleto pode falhar. Neste exemplo, utilizar a aresta (a, b) para enviar um fluxo 1 força enviar um fluxo 1 também pela aresta (s, b), o que não pode resultar no fluxo máximo 4 (onde utilizamos apenas os caminhos de aumento $s \to a \to t$ e $s \to b \to t$). O procedimento completo é subtrair $w(e)$ das arestas do caminho e adicionar $w(e)$ às arestas que vão no sentido contrário, de t para s. É possível provar que fazer isto sempre gera fluxos máximos. O grafo da Figura 3.6.1.b é o grafo da Figura 3.6.1.a após a primeira iteração do método Ford-Fulkerson, onde o caminho de aumento $s \to a \to b \to t$ foi o primeiro a ser encontrado. Note que se novamente um caminho de aumento que utiliza uma aresta entre a e b for encontrado, novamente o sentido desta aresta irá inverter e novamente acrescentaremos 1 ao fluxo máximo total. Se um caminho de aumento que passa por a e por b sempre for escolhido, faremos 4 iterações, sempre acrescentando 1 ao fluxo total. No final, a aresta (a, b) terá peso 1, pois nenhum fluxo máximo para este grafo utiliza esta aresta. Para finalizar esta discussão, vamos considerar um algoritmo que encontra um caminho

de aumento e vamos analisar a complexidade do método Ford-Fulkerson utilizando este algoritmo. Qualquer algoritmo de travessia, DFS ou BFS (custo $O(|E|)$), pode ser utilizado para encontrar um caminho de aumento. Como vimos no exemplo da Figura 3.6.1, o número de iterações (encontrar um caminho de aumento e enviar um fluxo por ele) pode coincidir com o valor do fluxo máximo, $|f^*|$. Logo, o custo do método Ford-Fulkerson com um algoritmo de travessia é $O(|E||f^*|)$. Note que é possível criar um grafo bem pequeno com um fluxo máximo bem grande, exatamente como o da Figura 3.6.1.a. Basta trocar os pesos de 2 para... 10^{15}, por exemplo.

Jack Edmonds e Richard Karp provaram que se buscarmos caminhos de aumento com BFS, ou seja, se escolhermos os caminhos de aumento com o menor número de arestas possível, são necessárias no máximo $O(|V||E|)$ iterações do método Ford-Fulkerson, resultando em custo $O(|V||E||E|) = O(|V||E|^2)$. No entanto, perceba que o algoritmo remove e insere arestas frequentemente, quando subtrai e adiciona pesos. Estruturas de dados eficientes para isto são bastante complicadas. Por esta razão, uma das implementações mais comuns do algoritmo Edmonds-Karp custa $O(|V||E||V|^2) = O(|V|^3|E|)$, por representar o grafo com matriz de adjacência. O que ocorre é que o custo de uma BFS passa de $O(|E|)$ para $O(|V|^2)$, por ser necessário passar por todas as entradas (possíveis arestas) de cada linha (cada vértice) da matriz. Felizmente, o custo de passar pelas arestas de um caminho de aumento P (em ambos os sentidos, para subtrair e somar pesos) ainda permance $O(|E|)$, pois basta seguir pelo caminho dado pelo vetor de pais encontrado pela BFS, $(u, pai(u))$, para cada $u \in P$.

3.6.2 Casamento máximo e cobertura mínima de vértices

Dar o exemplo "Torres que Atacam - URI".

Teorema de König

Capítulo 4

Teoria da complexidade

Este na verdade é um tópico a ser discutido por um texto de teoria da computação. No entanto, ter uma visão deste assunto sob o ponto de vista de um texto sobre algoritmos ajuda a compreender a sua importância fundamental. Damos uma breve introdução a classes de problemas e ao principal problema aberto da ciência da computação – P versus NP.

4.1 Classes de problemas

Ao longo deste texto, vimos a complexidade de diversos algoritmos. A teoria da complexidade, por outro lado, é construída sobre *complexidade de problemas*. Imagine, por exemplo, o problema da ordenação. Vimos que tal problema possui um algoritmo de tempo $\Theta(n \lg n)$ que o resolve (*merge sort*). Imagine, agora, todos os problemas que possuem algoritmos de tempo $\Theta(n \lg n)$. Tal conjunto de problemas é dito a classe dos problemas com solução de tempo $\Theta(n \lg n)$. A teoria da complexidade busca definir, levando em conta complexidade computacional, grandes classes de problemas e estabelecer relações entre elas.

A seguir são apresentadas algumas das classes mais notáveis em teoria da complexidade.

- P = $\{A \mid$ existe algoritmo de tempo polinomial para $A\}$
- NP = $\{A \mid$ existe algoritmo verificador de tempo polinimial para $A\}$
- PSPACE = $\{A \mid$ existe algoritmo de espaço polinomial para $A\}$
- EXPTIME = $\{A \mid$ existe algoritmo de tempo exponencial para $A\}$

Para entender melhor a classe de problemas NP, discutimos o seguinte exemplo.

No problema da satisfatibilidade booleana (SAT, apresentado na Seção 2.1.3), dada uma fórmula proposicional ϕ, queremos saber se existe uma valoração $\mathbb{V}_0$ para as n variáveis de ϕ que fazem $\mathbb{V}(\phi) = \text{V}$. Vimos que qualquer algoritmo para o SAT é $O(m2^n)$, onde m é o tamanho de ϕ. No entanto, vimos também que, dados ϕ e $\mathbb{V}_0$, verificar se $\mathbb{V}_0$ é tal que $\mathbb{V}(\phi) = \text{V}$ leva tempo $O(m)$, ou seja, leva tempo polinomial. Em outras palavras, para alguns problemas pode ser fácil determinar se uma solução candidata dada é uma solução, ainda que seja difícil encontrar uma solução sem nenhuma informação adicional de partida. Tais problemas formam a classe NP.

Observe o nome NP. A letra N vem do conceito de *não-determinismo*. Como dito anteriormente, as definições de classes de complexidade dadas aqui estão em termos de algoritmos. As definições exatas na verdade são feitas em termos de *máquinas de Turing*, um modelo matemático que captura precisamente o conceito de algoritmo, criado por Alan Turing em 1936. Deixamos tais conceitos para um texto de teoria da computação. Todavia, nossas definições são suficientes para entendermos o problema que vem a seguir.

4.2 P versus NP I

Hoje, sabemos que P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME. No entanto, não sabemos se nenhuma destas contenções é na verdade uma igualdade. O principal problema aberto da ciência da computação hojé é determinar se P = NP, ou se P $\subset$ NP. Por definição, já sabemos que $A \in \text{P} \implies A \in \text{NP}$. Mas ainda não conseguimos provar que $A \in \text{NP} \implies A \in \text{P}$, ou que $A \in \text{NP} \not\Longrightarrow A \in \text{P}$. Este problema é de grande interesse para o mundo, pois as consequências de P ser igual a NP são titânicas. Nos daríamos conta de que diversos problemas difíceis da biologia, engenharia, etc., na verdade são fáceis! Por outro lado, muitos sistemas de segurança que utilizam criptografia seriam facilmente burlados. É difícil de imaginar um mundo em que P = NP. Diante destes fatos, a solução conjecturada é P $\subset$ NP.

Desde que o problema se tornou famoso, grandes avanços já foram feitos. Para entender estes avanços, precisamos do conceito de redução, introduzido a seguir.

4.3 Reduções

Algo extremamente comum no projeto de algoritmos é utilizar reduções. Uma redução nada mais é do que utilizar a solução de um problema B para resolver um problema A, quando isto é possível. Por exemplo, o problema do casamento bipartido máximo se reduz ao problema do fluxo máximo. Pelo teorema de König, o problema da cobertura de vértices mínima se reduz ao problema do casamento bipartido máximo. Observe então que o problema da cobertura de vértices mínima também se reduz ao problema do fluxo máximo.

Na prática, uma redução R é um algoritmo que converte a entrada x de um problema A na entrada x' de um problema B, utiliza o algoritmo de B para obter a solução $f'(x')$ e finalmente converte $f'(x')$ em $f(x)$, uma solução de A.

Aqui, estamos interessados em *reduções de tempo polinomial*, contabilizando apenas o tempo gasto na conversão da entrada de A para uma entrada de B e na conversão da saída de B para uma saída de A. O tempo gasto pelo algoritmo de B é desconsiderado. Tais reduções são úteis nas definições que vêm a seguir.

4.4 Problemas NP-completos

Dizemos que um problema A é *NP-difícil*, se para cada problema de NP existe uma redução de tempo polinomial a A. Se, além disso, $A \in$ NP, dizemos que A é *NP-completo*.

Esta definição, no entanto, não é de grande impacto no problema P versus NP, até que de fato encontremos um problema NP-completo. E isto foi o que Stephen Cook fez em 1971. O teorema de Cook afirma que SAT é NP-completo. Utilizando o teorema de Cook, Richard Karp, do algoritmo Edmonds-Karp, encontrou reduções de tempo polinomial do SAT a determinados problemas, provando que tais problemas são também NP-completos. Em 1972, Karp publicou uma lista com 21 problemas NP-completos. Cook e Karp receberam o Prêmio Turing por suas descobertas.

Entre os 21 problemas NP-completos de Karp, estão:

- Determinar se um grafo possui um clique de k vértices.
- Determinar se um grafo possui uma cobertura de k vértices.
- Determinar se um grafo possui um caminho hamiltoniano entre um dado par de vértices.

Para concluir, discutimos o impacto de problemas NP-completos no problema P versus NP.

4.5 P versus NP II

Com a lista dos 21 problemas NP-completos de Karp, ou com qualquer problema NP-completo A, temos as seguintes soluções para P versus NP.

- Se encontramos um algoritmo de tempo polinomial para A, então P = NP.
- Se provamos que não existe algoritmo de tempo polinomial para A, então P $\subset$ NP.

Note que é infinitamente mais fácil atacar problemas específicos, problemas que conhecemos, do que atacar uma classe *infinita* de problemas. Daí a grandiosidade das descobertas de Cook e Karp.

Referências Bibliográficas

[1] Steven Halim and Felix Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests: Handbook for ACM ICPC and IOI Contestants*. Lulu. com, 2013.