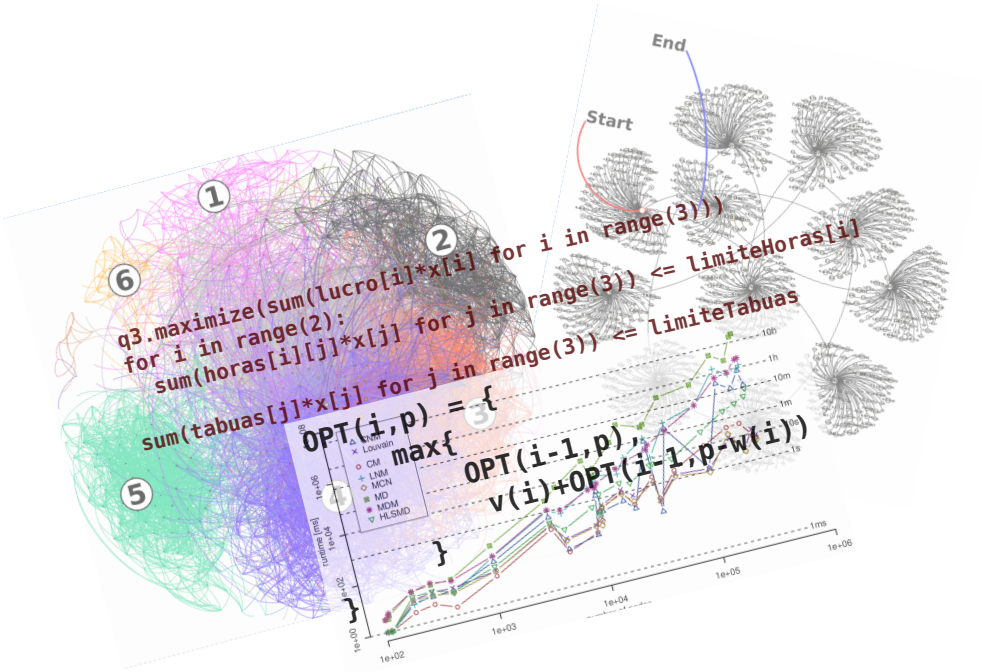
Rafael de Santiago
Álvaro Junio Pereira Franco
Gabriel Arthur Gerber Andrade
Lucas Pagotto Coutinho de Oliveira
Tális Breda

Anotações para a curso de extensão Boas práticas em algoritmos para programadores

Versão de 12 de dezembro de 2022



Universidade Federal de Santa Catarina

Sumário

CAPÍTULO 1

Problemas e Algoritmos

O que veremos nesse curso?

- Estimar a quantidade de recursos (tempo, memória) de um algoritmo = análise de complexidade;
- Técnicas e idéias gerais de projeto de algoritmos: divisão-e-conquista, programação dinâmica, algoritmos gulosos, programação linear (e inteira) ...
- Tema recorrente: natureza recursiva de vários problemas;
- Brevemente, algumas estruturas de dados.

1.1 Problemas

De acordo com Cormen et al. (2012), a especificação de um problema indica quais são as entradas e quais as saídas desejadas. Um algoritmo basicamente especifica o que deve ser feito com as entradas para que se possa gerar as saídas que atendem ao problema.

Para exemplificar, considere um problema de ordenação. Ele poderia ser definido formalmente por suas entradas e saídas (CORMEN et al., 2012):

- Entrada: uma sequência de números $\langle a_1, a_2, \ldots, a_n \rangle$.

- Saída: uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$, tal que $a'_i \leq a'_{i+1}$ para todo $i \in \{1, 2, \ldots n-1\}$.

. Um exemplo de entrada para esse problema seria $\langle 9, 5, 4, 10, 2 \rangle$ e um exemplo de saída seria $\langle 2, 4, 5, 9, 10 \rangle$. Um exemplo de entrada para um problema computacional é chamado de instância ou instância para o problema.

Existem diversos tipos de problemas caracterizados na literatura.

1.1.1 Tipos de Problemas Comuns

Imagine que tenhamos um conjunto de itens valiosos $I = \{1, 2, \ldots, n\}$ e que cada item i tem um peso $w_i \in \mathbb{Z}^+$ e seu valor monetário $v_i \in \mathbb{R}$. Temos também uma mochila para carregar alguns (ou todos) os itens. Essa mochila tem capacidade de peso dada por $W \in \mathbb{R}$, que deve ser utilizado como limite/restrição, porque a soma dos pesos dos itens colocados na mochila não pode ultrapassar W . Essa problema, pode ser visto de várias formas no contexto de problema computacional.

Como um **problema de decisão**, no contexto do problema da mochila, podemos procurar se é possível colocar um subconjunto de itens na mochila tal que o valor monetário total desses itens seja maior ou igual a um valor $V \in \mathbb{R}^+$.

Na versão de um **problema de busca** deseja-se saber qual o subconjunto de I é que traz uma resposta “sim” para o problema de decisão.

Quando se deseja descobrir qual o valor máximo que se pode obter de itens na mochila (respeitando sua capacidade), temos um **problema de valor ótimo**.

Em um **problema de otimização** deseja-se saber qual o subconjunto de I que corresponde ao valor ótimo.

A classificação acima não é uma convenção, mas é uma organização proposta em Kreher e Stinson (1999) para problemas combinatórios que captura um grande conjunto de problemas computacionais.

Para quê usar tanta formalidade?

A ideia de trazer elementos da matemática discreta para descrever problemas e algoritmos vêm da necessidade que temos de capturar um problema de forma genérica. Veja, no contexto do problema da mochila, estamos definindo a natureza do problema e seus limites, sem ter que "materializar"/informar de quantos itens vamos tratar, quais seus pesos e valores monetários, como é a mochila. A descrição genérica do problema, nos dá a possibilidade de capturar melhor o "espírito" de diversos realidades (quais métodos usar e quais as características do problema por exemplo).

Um pouco mais sobre Problemas de Otimização

Neste curso, lida-se com problemas os quais buscam por uma solução chamada de "solução ótima". Esses problemas são chamados de problemas de otimização. Uma solução ótima é uma resposta para o problema que obtém a melhor (mínima ou máxima) avaliação possível dada por uma função. Essa função é denominada de função objetivo. Um algoritmo para um problema de otimização sempre encontra a solução ótima.

Soluções que não são ótimas, podem ou não satisfazer restrições relacionadas ao problema. As soluções que satisfazem as restrições do problema, mas não possuem a melhor avaliação possível pela função objetivo, são chamadas de solução admissíveis não-ótimas. Toda solução ótima é uma solução admissível.

Desafio

Defina formalmente cinco problemas computacionais. Relacione um algoritmo para cada problema.

A importância dos "eficientes" algoritmos:

- projeto genoma de seres vivos;
- rede mundial de computadores;
- redes sociais;

- planejamento da produção;
- logística de distribuição;
- reconhecimento de padrões;
- entretenimento (games e filmes).

1.2 Algoritmos

Cormen et al. (2012) define um algoritmo como "qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída.". Um algoritmo resolve um problema computacional.

1.3 Dificuldade Computacional

O fato de conhecer algoritmos corretos não é suficiente. Precisa-se de algoritmos eficientes no contexto em que serão aplicados.

Para medir a complexidade computacional de um algoritmo, pode-se utilizar a medida empírica do tempo ou do espaço consumido. A medida empírica do tempo é questionável, pois considera situações diferentes (arquiteturas, linguagens de programaçao, sistemas operacionais e, até mesmo, condições do tempo). No contexto dessa disciplina, se deseja entender a eficiência de um algoritmo.

Para medir a dificuldade de complexidade de um algoritmo, serão utilizadas funções de complexidade. Uma função de complexidade é dada por um $f : n \in \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ e indica a quantidade de tempo requerido (contra-domínio) para uma entrada de tamanho n. De maneira geral, diz-se que um algoritmo é eficiente se o mesmo possui uma função de complexidade polinomial (polinomial × exponencial).

Para contextualizar a importância da eficiência dos algoritmos, considere dois tipos de algoritmos de ordenação: por inserção e por intercalação (Mergesort). Considere

também que um determinado algoritmo de inserção utiliza o tempo $c_1 n^2$ e outro algoritmo de inserção utiliza $c_2 n log_2 n$. Assuma que $c_1 < c_2$ e que esses valores são constantes sem qualquer relação com n. Apesar de $c_1 < c_2$, um algoritmo de intercalação é mais eficiente para um n relativamente grande.

Considere dois computadores: um computador rápido A e um computador lento B. Considere também que o computador A executa uma ordenação por inserção e o computador B executa uma ordenação por intercalação. Cada um deve ordenar um vetor de 10 milhões de elementos. O computador A executa 10 bilhões de instruções por segundo (!!!), enquanto o computador B executa 10 milhões de instruções por segundo. Considere que $c_1 = 2$ e $c_2 = 50$. Então, o demanda de tempo o computador A é

$$\frac{2 \cdot (10^7)^2 \text{ instruções}}{10^{10} \text{ instruções/segundo}} = 20.000 \text{ segundos } (5,5 \text{ horas})$$

. O computador B demanda

$$\frac{50 \cdot (10^7) \log_2 10^7 \text{instruções}}{10^7 \text{instruções/segundo}} \approx 1.163 \text{ segundos}$$

(CORMEN et al., 2012).

Existem casos onde a eficiência é ainda mais importante como os problemas NP-Difíceis. Não se conhece algoritmo com função de complexidade polinomial para esses problemas. São exemplos:

- roteamento de veículos para entregas (*vehicle routing*);
- calcular o número mínimo de containers para transportar um conjunto de caixas com produtos (*bin-packing* 3D);
- calcular a localização e o número mínimo de antenas para garantir a cobertura de uma certa região geográfica (*facility location*).

Para contextualizar essa dificuldade, imagine usar o computador A discutidos acima para a ordenação acima sobre um algoritmo exato para um problema mais simples

que o do "roteamento de entregas para vários veículos" conhecida como Problema do Caixeiro Viajante. Assuma o algoritmo de Held-Karp com complexidade $c_3 n^2 2^n$. Considere que $c_3 = 1$ e $n = 100$. Para o computador A (mais veloz) obtería-se

$$\frac{1 \cdot (100^2 \cdot 2^{100}) \text{ instruções}}{10^{10} \text{ instruções/segundo}} \approx 1,26 \times 10^{20} \text{ segundos}$$

$$\approx 3,52 \times 10^{16} \text{ horas} \approx 4,01 \times 10^{12} \text{ anos}$$

$$\approx 40169423537,53 \text{ séculos.}$$

1.4 Projeto e Análise de Algoritmos

Essa seção tem o objetivo de introduzir o projeto e análise de algoritmos através de alguns exemplos. Nesse contexto, é interessante analisar:

- Finitude: o algoritmo pára? (não cobrimos aqui)
- Corretude: o algoritmo faz o que promete? (não cobrimos aqui)
- Complexidade de tempo: quantas instruções são necessárias no pior caso[1]?

1.4.1 Busca Sequencial

Considere o seguinte problema de busca:

- Entrada:
  - uma sequência de números $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$;
  - uma valor chave k.
- Saída: a posição em A no qual k se encontra. Se não estiver em A, a saída deverá ser -1.

O algoritmo de busca sequencial resolve esse problema:

[1] Além do pior caso, podem ser considerados o melhor e o caso médio em pesquisas mais sensíveis.

```
1  def buscaSequencial(A, k):
2      for i in range(len(A)):
3          if k == A[i]:
4              return i
5      return -1
6
7  def main():
8      vetor = [9,8,7,6,5,4,0,1]
9      chave = 70
10     pos = buscaSequencial(vetor, chave)
11     print(pos)
12
13 if __name__ == "__main__":
14     main()
```

1.4.1.1 Complexidade de tempo

No melhor caso (menor tempo) encontraríamos a chave na primeira posição do vetor.

1	`def buscaSequencial(A, k):`	Custo	Num. Exec.
2	`    for i in range(len(A)):`	c_1	1
3	`        if k == A[i]:`	c_2	1
4	`            return i`	c_3	1
5	`    return -1`	c_4	0

. Logo, a complexidade de tempo computacional do melhor caso é pode ser dado por

$$T(n) = c_1 + c_2 + c_3.$$

Para o pior caso (maior tempo possível), a chave não seria encontrada. Teríamos o seguinte:

```
1 def buscaSequencial(A, k):      |  Custo      | Num. Exec.
2     for i in range(len(A)):     |  c1         | n+1
3         if k == A[i]:           |  c2         | n
4             return i            |  c3         | 0
5     return -1                   |  c4         | 1
```

. Logo, a complexidade de tempo computacional do pior caso é pode ser dado por

$$T(n) = c_1(n+1) + c_2 n + c_4 = c_1 n + c_2 n + c_1 + c_4 = (c_1 + c_2)n + c_1 + c_4.$$

Podemos dizer se $T(n)$ representa o tempo computacional da busca sequencial para todos os casos, podemos afirmar que:

$$c_1 + c_2 + c_3 \leq T(n) \leq (c_1 + c_2)n + c_1 + c_4.$$

E qual o espaço computacional (memória) requerido? Pense numa função $S(n)$ representando o espaço computacional para o algoritmo.

1.4.2 Ordenação por Inserção

Considere o seguinte problema de ordenação:

- Entrada: uma sequência de números $\langle a_0, a_1, \ldots, a_{n-1} \rangle$.
- Saída: uma permutação (reordenação) $\langle a'_0, a'_1, \ldots, a'_{n-1} \rangle$, tal que $a'_i \leq a'_{i+1}$ para todo $i \in \{0, 1, \ldots n-1\}$.

Um algoritmo para resolvê-lo pode ser visualizado no código abaixo:

```
1 def insertionSort(A):
2     for j in range(1, len(A)):
3         chave = A[j]
4         #inserir A[j] no subvetor <A[0], A[1], ..., A[j-1]>
5         i = j - 1
```

```
6         while i >= 0 and A[i]> chave:
7             A[i+1] = A[i]
8             i = i - 1
9         A[i+1] = chave
10
11
12 def main():
13     vetor = [9,8,7,6,5,4,0,1]
14     insertionSort(vetor)
15     print(vetor)
16
17 if __name__ == "__main__":
18     main()
```

1.4.2.1 Complexidade de Tempo

Agora, o objetivo é identificar a função de complexidade que corresponde ao tempo computacional demandado pelo algoritmo Insertion-sort. A complexidade de tempo deve ser realizada contanto o número de instruções básicas (operações elementares ou primitivas) em relação a entrada de tamanho n.

Primeiramente, identifica-se o custo computacional de cada instrução. Na tabela abaixo, considera-se que cada instrução possui um custo c_k, no qual k é a linha da instrução.

```
1 def insertionSort(A):                  |  Custo    | Num. Exec.
2     for j in range(1, len(A)):         |  c_1      |
3         chave = A[j]                   |  c_2      |
4         i = j - 1                      |  c_3      |
5         while i >= 0 and A[i]> chave:  |  c_4      |
6             A[i+1] = A[i]              |  c_5      |
```

```
7            i = i - 1                    |  c6       |
8        A[i+1] = chave                   |  c7       |
```

Considere que t_j é o número de vezes que o teste do laço *while* na linha 4 é executado para aquele valor de j. Considere também que o vetor possui n posições. Atualiza-se então a tabela com essas considerações.

```
1  def insertionSort(A):                  |  Custo    | Num. Exec.
2      for j in range(1, len(A)):         |  c1       | n
3          chave = A[j]                   |  c2       | n-1
4          i = j - 1                      |  c3       | n-1
5          while i >= 0 and A[i]> chave:  |  c4       | sum_{j=1}^{n-1} t_j
6              A[i+1] = A[i]              |  c5       | sum_{j=1}^{n-1} (t_j - 1)
7              i = i - 1                  |  c6       | sum_{j=1}^{n-1} (t_j - 1)
8          A[i+1] = chave                 |  c7       | n-1
```

O tempo de execução total é dado por

$$\begin{aligned} T(n) = & \\ & c_1 n + c_2(n-1) + c_3(n-1) + \\ & c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) \\ & + c_7(n-1). \end{aligned} \tag{1.1}$$

Precisa-se identificar agora o valor de t_j. No melhor caso, $t_j = 1$. Ele ocorre quando o vetor já está ordenado. Desse modo, têm-se

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned} \tag{1.2}$$

. Essa é uma função linear de n.

No pior caso, o vetor está na ordem inversa. Nesse caso, $t_j = j+1$ (j repetições mais um teste falso). Sabendo que

$$\sum_{j=1}^{n-1} j+1 = \left(\sum_{j=1}^{n-1} j\right) + \Big(n-1\Big) = \frac{(n-1)(n-1+1)}{2} + n - 1$$

$$= \frac{n^2-n}{2} + n - 1 = \frac{n^2-n}{2} = \frac{n^2+n-2}{2} = \frac{n^2+n}{2} - 1$$

e

$$\sum_{j=1}^{n-1} (j+1-1) = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$

, têm-se

$$\begin{aligned} T(n) = & \\ & c_1 n + c_2(n-1) + c_3(n-1) + c_7(n-1) \\ & + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) \\ = & \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7). \end{aligned} \tag{1.3}$$

Obtêm-se então uma função quadrática em relação ao tamanho da entrada no pior caso. Diz-se que a complexidade assintótica do pior caso é $\Theta(n^2)$

CAPÍTULO 2

Notação Assintóticas: o famoso Big O

Embora seja possível determinar o tempo computacional exato de um algoritmo, a precisão obtida não vale o esforço para instâncias suficientemente grandes. As constantes multiplicativas e os termos de ordem mais baixa ao determinar um tempo exato são dominados pelo termo de maior relevância (CORMEN et al., 2012).

"Sobre a entrada n, o algoritmo executa no máximo $1.62n^2+3.5n+8$ passos." (KLEINBERG; TARDOS, 2005). Este tipo de análise pode ser útil em alguns contextos, mas de modo geral não:

- Conseguir uma medida tão precisa pode ser exaustivo;
- O objetivo na complexidade de algoritmos é identificar classes de algoritmos que possuem comportamento similar;
- Precisa-se de uma medida menos detalhista.

2.1 Notação Assintótica

O objetivo dessa seção é formalizar essa e outras notações assintóticas. Nesse contexto, serão ignoradas as constantes multiplicativas e os termos de menor ordem. São discutidas aqui as ordens assintóticas O, Ω e Θ.

2.1.1 Notação Θ

Quando uma função $f(n)$ é encaixada entre os valores $c_2 g(n)$ e $c_1 g(n)$ para duas constantes c_1 e c_2 , diz-se que pertence ao grupo $\Theta\big(g(n)\big)$. Diz-se que $g(n)$ é um limite assintoticamente restrito para $f(n)$

$$\begin{aligned}\Theta\big(g(n)\big) = \big\{f(n) | \text{ existem constantes positivas } c_1 \text{ , } c_2 \text{ e } n_0 \\ \text{tais que } 0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \text{ para todo o } n \geq n_0\big\}\end{aligned} \tag{2.1}$$

A Figura 1 exibe a notação sendo utilizada para o caso $f(n) \in \Theta\big(g(n)\big)$.

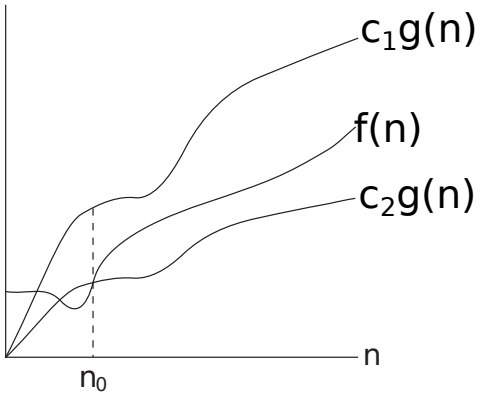


Figura 1 – Exemplo da aplicação da notação assintótica Θ. Nela, $f(n) \in \Theta\big(g(n)\big)$ (CORMEN et al., 2012).

Por força de representação, é comum considerar que $f(n) = \Theta\big(g(n)\big)$, embora o correto seja $f(n) \in \Theta\big(g(n)\big)$.

Exemplos nos quais $f(n) \in \Theta\big(g(n)\big)$:

- $f(n) = \frac{1}{2}n^2 - 3n$ e $g(n) = n^2$;
- $f(n) = 100n^3 - n^2 + 3.5n - 17$ e $g(n) = n^3$.

2.1.2 Notação O

Quando uma função $f(n)$ possui como limite assintótico superior $g(n)$ diz-se que $f(n)$ pertence ao grupo $O\big(g(n)\big)$. Diz-se que $g(n)$ é o limite assintótico superior para $f(n)$.

$$O\big(g(n)\big) = \big\{f(n) | \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \text{ para todo o } n \geq n_0\big\} \tag{2.2}$$

A Figura 2 exibe a notação sendo utilizada para o caso $f(n) \in O\big(g(n)\big)$.

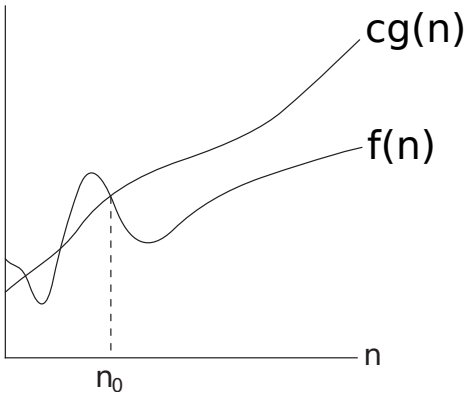


Figura 2 – Exemplo da aplicação da notação assintótica O. Nela, $f(n) \in O\big(g(n)\big)$ (CORMEN et al., 2012).

É importante ter em mente que se $f(n) \in \Theta\big(g(n)\big)$, então $f(n) \in O\big(g(n)\big)$. Isso se deve ao fato de que a notação Θ é uma noção mais forte de O.

A notação O é empregada para informar o pior caso da complexidade de um algoritmo. Desse modo, quando se diz que "a complexidade é $O(n^2)$" que o algoritmo demandará tempo de pior caso cn^2 para uma entrada de tamanho n.

Exemplos nos quais $f(n) \in O\big(g(n)\big)$:

- $f(n) = \frac{1}{4}n^2 - n$ e $g(n) = n^2 - 6n$;
- $f(n) = 100n^3 - n^2 + 3.5n - 17$ e $g(n) = 2^n + 3n^2$.

2.1.3 Notação Ω

Quando uma função $f(n)$ possui como limite assintótico inferior $g(n)$ diz-se que $f(n)$ pertence ao grupo $\Omega\big(g(n)\big)$.

$$\Omega\big(g(n)\big) = \big\{f(n) | \text{ existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq cg(n) \leq f(n) \text{ para todo o } n \geq n_0\big\} \tag{2.3}$$

A Figura 3 exibe a notação sendo utilizada para o caso $f(n) \in \Omega\big(g(n)\big)$.

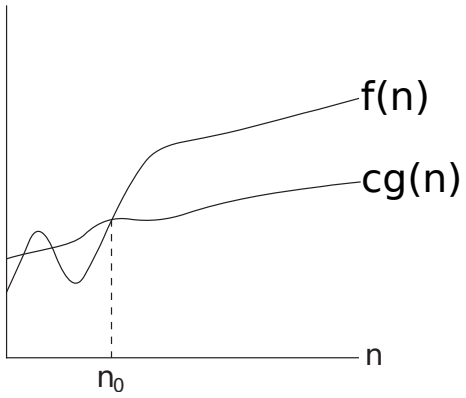


Figura 3 – Exemplo da aplicação da notação assintótica Ω. Nela, $f(n) \in \Omega\big(g(n)\big)$ (CORMEN et al., 2012).

Teorema 2.1.1. *Para quaisquer duas funções $f(n)$ e $g(n)$, $f(n) \in \Theta\big(g(n)\big)$ sse $f(n) \in O\big(g(n)\big)$ e $f(n) \in \Omega\big(g(n)\big)$.*

Exemplos nos quais $f(n) \in \Omega\big(g(n)\big)$:

- $f(n) = \frac{1}{2}n^2 - 3n$ e $g(n) = \frac{1}{2}n^2 - 2n$;
- $f(n) = 2^n + 3n^2$ e $g(n) = 100n^3 - n^2 + 3.5n - 17$.

2.1.4 Propriedades das Notações

Transitividade (CORMEN et al., 2012):

- $f(n) \in \Theta\big(g(n)\big)$ e $g(n) \in \Theta\big(h(n)\big)$ implicam $f(n) \in \Theta\big(h(n)\big)$;
- $f(n) \in O\big(g(n)\big)$ e $g(n) \in O\big(h(n)\big)$ implicam $f(n) \in O\big(h(n)\big)$;
- $f(n) \in \Omega\big(g(n)\big)$ e $g(n) \in \Omega\big(h(n)\big)$ implicam $f(n) \in \Omega\big(h(n)\big)$.

Reflexidade:

- $f(n) \in \Theta\big(f(n)\big)$;
- $f(n) \in O\big(f(n)\big)$;
- $f(n) \in \Omega\big(f(n)\big)$.

Simetria: $f(n) \in \Theta\big(g(n)\big)$ sse $g(n) \in \Theta\big(f(n)\big)$.
Simetria de Transposição:

- $f(n) \in O\big(g(n)\big)$ sse $g(n) \in \Omega\big(f(n)\big)$;

Pode-se utilizar uma analogia dessa relação entre funções com as comparações entre os números reais a e b:

- $f(n) \in O\big(g(n)\big)$ é como $a \leq b$;
- $f(n) \in \Omega\big(g(n)\big)$ é como $a \geq b$;
- $f(n) \in \Theta\big(g(n)\big)$ é como $a = b$.

Multiplicação por uma constante:

- $\Theta(cf(n)) = \Theta(f(n))$;
- $99n^2 \in \Theta(n^2)$.

Mais alto expoente de um polinômio:

- $3n^3 - 5n^2 + 100 \in \Theta(n^3)$;
- $6n^4 - 20n^2 \in \Theta(n^4)$;
- $0.8n + 224 \in \Theta(n)$.

Termo dominante:

- $2^n + 6n^3 \in \Theta(2^n)$;

- $n! - 3n^2 \in \Theta(n!)$;
- $nlogn + 3n^2 \in \Theta(n^2)$.

CAPÍTULO 3

Recursividade

Recursividade ocorre quando um método chama a si mesmo, direta ou indiretamente . As chamadas recursivas são empilhadas de modo que a mais recentemente invocada é a que está sendo executada no momento (ZIVIANI, 2007).

Um algoritmo recursivo deve considerar em seu projeto dois importantes critérios: o de continuidade e o de parada. Desse modo, um projeto de algoritmo recursivo deve estar sujeita a uma condição B que deve não ser satisfeita em algum momento da computação (parada). Um exemplo genérico de uma chamada recursiva P:

$$P \equiv \textbf{if } B \textbf{ then } \textit{ chama } P$$

ou

$$P \equiv \textbf{if } B \textbf{ then } \textit{ chama } P \textbf{ else } \textit{faz algo e pára}$$

.

Como exemplo clássico, vamos usar o da busca binária. O algoritmo da busca binária recebe uma sequência ordenada e retorna a posição de uma chave. Se a chave não estiver no vetor, retorna-se -1.

```
1 def buscaBinaria(A, k, inicio, fim):
```

```
2   if inicio > fim: #nÃ£o encontra
3     return -1
4   else:
5     meio = (fim-inicio+1)//2 + inicio #calcula indice do meio do
          vetor ordenado
6     if k == A[meio]: #se a chave estiver na posiÃ§Ã£o central
7       return meio
8     else:
9       if k > A[meio]: #se a chave for maior que o valor da
            posiÃ§Ã£o central
10        return buscaBinaria(A, k, meio + 1, fim)
11      else: #se a chave for menor que o valor da posiÃ§Ã£o
            central
12        return buscaBinaria(A, k, inicio, meio - 1)
13
14  def main():
15      vetor = [3, 4, 6, 7, 8]
16      chave = 7.5
17      pos = buscaBinaria(vetor, chave, 0, len(vetor)-1)
18      print(pos)
19
20
21  if __name__ == "__main__":
22      main()
```

Mas afinal, qual a complexidade de tempo da busca sequencial? Qual o melhor caso e o pior caso?

No melhor caso, encontramos a chave no meio do vetor, então demoraríamos uma quantidade constante instruções de alto nível. Para o pior caso, analisemos o algoritmo

(não encontrar a chave no vetor).

```
1  def buscaBinaria(A, k, inicio, fim):                | Custo
2    if inicio > fim:                                  |  c1
3      return -1                                       |  c2
4    else:
5      meio = (fim-inicio+1)//2 + inicio               |  c3
6      if k == A[meio]:                                |  c4
7        return meio                                   |  c5
8      else:
9        if k > A[meio]:                               |  c6
10         return buscaBinaria(A, k, meio + 1, fim)    |  T(n/2)
11       else:
12         return buscaBinaria(A, k, inicio, meio - 1)|  T(n/2)
```

. A complexidade para cada chamada da busca binária seria

$$T(n) \leq c_1 + c_3 + c_4 + c_6 + T\left(\frac{n}{2}\right),$$

independente se a condição da linha 9 for verdadeira ou falsa. Poderíamos dizer que

$$T(n) \leq T\left(\frac{n}{2}\right) + c',$$

assumindo que $c' \geq c_1 + c_3 + c_4 + c_6$.

Nesse caso temos uma função de recorrência, na qual a complexidade de tempo é uma recorrência (uma função que depende dela mesma). Precisamos entender o crescimento assintótico, então realizamos o tratamento de recorrências.

Por se tratar de uma recorrência, muitas vezes é útil identificar o caso base. O caso base se trata do caso de parada, ou seja, o problema vai ser dividido minimamente até que a chamada não represente nenhuma posição ($inicio > fim$), ou seja, o algoritmo não depende mais de uma chamada recursiva para obter a resposta correspondente à chamada (linhas 2 e 3). A complexidade do caso base é:

$$T(n) \leq c_1 + c_2.$$

Ou ainda

$$T(n) \leq c'',$$

se assumirmos que $c'' \geq c_1 + c_2$.

3.1 Tratamento de Recorrências

Relações de recorrências expressam a complexidade de algoritmos recursivos. É preciso resolver a recorrência para determinar a complexidade através da chamada fórmula fechada (que não depende da mesma ou de outra função).

3.1.1 Método da Mestre

O método mestre é utilizado para resolver recorrências na forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, onde $a, b > 1$ e $f(n)$ é uma função assintoticamente positiva. Considera-se que $\left\lfloor \frac{n}{b} \right\rfloor = \left\lceil \frac{n}{b} \right\rceil = \frac{n}{b}$. Então, $T(n)$ tem os seguintes limites assintóticos:

1. Se $f(n) \in O(n^{log_b a - \epsilon})$, para uma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{log_b a})$;
2. Se $f(n) \in \Theta(n^{log_b a})$, então $T(n) \in \Theta(n^{log_b a} \lg n)$;
3. Se $f(n) \in \Omega(n^{log_b a + \epsilon})$, para uma constante $\epsilon > 0$ e $af\left(\frac{n}{b}\right) \leq cf(n)$ para uma constante $c < 1$, então $T(n) \in \Theta\big(f(n)\big)$.

Em cada um dos três casos, compara-se a função $f(n)$ com a função $n^{\log_b a}$. A maior entre as duas funções estabelece a solução para a recorrência. Há ainda detalhes técnicos importantes a considerar (CORMEN et al., 2012):

- Para o primeiro caso, $f(n)$ não só tem que ser menor que $n^{\log_b a}$, mas tem que ser polinomialmente menor. Ou seja, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator n^{ϵ} para uma constante $\epsilon > 0$;
- Para o terceiro caso, $f(n)$ não só tem que ser maior que $n^{\log_b a}$, mas tem que ser polinomialmente maior e satisfazer a condição "$af\left(\frac{n}{b}\right) \leq cf(n)$". Ou seja, $f(n)$

deve ser assintoticamente maior que $n^{\log_b a}$ por um fator n^{ϵ} para uma constante $\epsilon > 0$.

Exemplo 1

Considere a seguinte recorrência $T(n) = 9T\left(\frac{n}{3}\right) + n$. Em seu formato, considera-se:

- $a = 9$;
- $b = 3$;
- $f(n) = n$.

Portanto, $n^{\log_b a} = n^{\log_3 9} \in \Theta(n^2)$. Desse modo, $f(n) = n^{\log_b a - \epsilon}$ com $\epsilon = 1$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^2)$.

Exemplo 2

Considere a seguinte recorrência $T(n) = T\left(\frac{2n}{3}\right) + 1$. Em seu formato, considera-se:

- $a = 1$;
- $b = \frac{3}{2}$;
- $f(n) = 1$.

Portanto, $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$. Desse modo, $f(n) = n^{\log_b a}$, então aplica-se o segundo caso, obtendo $T(n) \in \Theta(\lg n)$.

Exemplo 3

Considere a seguinte recorrência $T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$. Em seu formato, considera-se:

- $a = 3$;
- $b = 4$;
- $f(n) = n\lg n$.

Portanto, $n^{\log_b a} = n^{\log_4 3} = n^{0,793}$. Desse modo, $f(n) \in \Omega(n^{\log_4 3+\epsilon})$ com $\epsilon \approx 0,2$, então precisa-se verificar uma última condição para aplicar o terceiro caso: considerando $af\left(\frac{n}{b}\right) = 3\frac{n}{4}\lg\frac{n}{4}$ e $cf(n) = \frac{3}{4}n\lg n$, então para $af\left(\frac{n}{b}\right) \leq cf(n)$, $3\frac{n}{4}\lg\frac{n}{4} \leq \frac{3}{4}n\lg n$ para um $c = \frac{3}{4}$. Desse modo, aplica-se o caso 3, obtendo-se $T(n) \in \Theta(n\lg n)$.

Exemplo 4

Considere a seguinte recorrência $T(n) = 2T\left(\frac{n}{2}\right) + n\lg n$. Em seu formato, considera-se:

- $a = 2$;
- $b = 2$;
- $f(n) = n\lg n$.

Portanto, $n^{\log_b a} = n^{\log_2 2} = n$. Nesse caso, poderia-se enganar dizendo que $n\lg n$ é assintoticamente maior que $n^{\log_b a} = n$. O problema é que ela não é polinomialmente maior. A razão $\frac{f(n)}{n^{\log_b a}} = \frac{n\lg n}{n} = \lg n$ é assintoticamente menor que n^{ϵ}, o que faz a recorrência ficar entre os casos 2 e 3.

Exemplo 5

Considere a seguinte recorrência $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$. Em seu formato, considera-se:

- $a = 2$;
- $b = 2$;
- $f(n) = n$.

Portanto, $n^{\log_b a} = n^{\log_2 2} = n$. Desse modo, $f(n) = n^{\log_b a}$, então aplica-se o segundo caso, obtendo $T(n) \in \Theta(n\lg n)$.

Exemplo 6

Considere a seguinte recorrência $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$. Em seu formato, considera-se:

- $a = 8$;
- $b = 2$;
- $f(n) = \Theta(n^2)$.

Portanto, $n^{\log_b a} = n^{\log_2 8} = n^3$. Desse modo, $f(n) \in O(n^{\log_b a - \epsilon})$ com $\epsilon = 1$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^3)$.

Exemplo 7

Considere a seguinte recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$. Em seu formato, considera-se:

- $a = 7$;
- $b = 2$;
- $f(n) = n^2$.

Portanto, $n^{\log_b a} = n^{\lg 7} < n^{2,81}$. Desse modo, $f(n) \in O(n^{\log_b a - \epsilon})$ com $\epsilon \approx 0,8$, então aplica-se o primeiro caso, obtendo $T(n) \in \Theta(n^{\lg 7})$.

Exemplos onde não se aplica o método mestre

Estes são alguns exemplos onde não se aplica o método mestre:

- $T(n) = T(n-1) + n$;
- $T(n) = T(n-a) + T(a) + n$, com $a \in \mathbb{Z}^+$;
- $T(n) = T(\alpha n) + T\big((1-\alpha)n\big) + n$, com $0 < \alpha < 1$;
- $T(n) = T(n-1) + \log n$;
- $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$.

3.1.1.1 Método da Mestre e a Busca Binária

De posse do método mestre, e da recorrência da busca binária, vamos tentar descobrir a fórmula fechada.

Relembrado a recorrência da complexidade de tempo da busca binária

$$T(n) \leq T\left(\frac{n}{2}\right) + c',$$

pode-se considerar que $a = 1$, $b = 2$ e $f(n) = c'$, sabendo que c' é uma constante independente de n. Como $c' \in \Theta(n^{log_2 1})$ (pois $n^{log_2 1} = n^0 = 1$), então caímos no caso 2. Desse modo, a complexidade pessimista da busca binária é dada por

$$n^{log_2 1} log_2 n = n^0 log_2 n = log_2 n,$$

ou seja,

$$T(n) \in O(log_2 n).$$

3.1.1.2 Método Mestre Falha?

Considere o seguinte exemplo abaixo do algoritmo recursivo de inserir ao final de uma lista encadeada.

```
1  #classe nodo da lista encadeada
2  class Nodo:
3    def __init__(self, dado):
4      self.dado = dado
5      self.proximo = None
6
7  #insere no final da lista encadeada
8  def insereFinal(ref, dado):
9    if ref == None: #lista vazia
10     print("Erro: Lista deve conter no mÃnimo um 'Nodo'")
11     return False
```

```
12    else:
13      if ref.proximo == None: #final da lista
14        novo = Nodo(dado)
15        ref.proximo = novo
16      else: #caso ref nÃ£o seja o nodo do fim
17        insereFinal(ref.proximo, dado)
18      return True
19
20  #insere no final da lista encadeada
21  def imprimeLista(ref):
22    if ref != None:
23      print(ref.dado, end = ', ')
24      imprimeLista(ref.proximo)
25
26  def main():
27      inicio = Nodo("Rafael")
28      insereFinal(inicio, "Lucas")
29      insereFinal(inicio, "Talis")
30      imprimeLista(inicio)
31
32  if __name__ == "__main__":
33      main()
```

. A complexidade pessimista da inserção recursiva ao final em uma lista encadeada traria a seguinte recorrência (aqui considere que o tamanho do problema - n - seja o número de elementos previamente armazenados no vetor, antes da inserção):

$$T(n) \leq T(n-1) + c,$$

considerando c uma constante independente de n. Para calcular a fórmula fechada

através do método mestre, utilizaríamos $a = 1$ e $f(n) = c$, mas quem seria b? O método mestre não consegue resolver todas as recorrências. Existem outras formas de fazer o cálculo dessas recorrências. A seguir, apresente-se outra forma de tratar recorrências.

3.1.2 Método da Árvore de Recursão

O método da árvore de recursão converte a recorrência em uma árvore nos quais os nodos representam os custos de um único subproblema. Analisa-se então os custos em cada nível da árvore gerada. Usa-se algumas técnicas para limitar o tamanho da árvore.

De acordo com Cormen et al. (2012), uma árvore de recursão é bem mais usada para gerar um bom palpite. Se a árvore for desenhada detalhadamente e criteriosamente, a mesma pode ser utilizada como prova direta de uma solução de recorrência.

Como exemplo da aplicação de uma árvore de recursão, considera-se a complexidade da recorrência da inserção ao final de uma lista encadeada:

$$T(n) \leq T(n-1) + c.$$

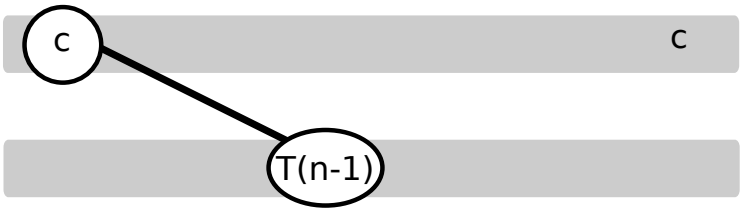
Considerando o subproblema mínimo igual a 1 (caso base $T(1) = c'$), então o número de níveis pode ser calculado imaginando o maior nível para n veja a Figura 4:
Considerando as profundidades $1, 2, \ldots, n-1$ mais um nivel para o caso base, têm-se n níveis. Observando a complexidade requerida em cada nível, têm-se uma nova definição para $T(n)$:

$$T(n) \leq c + c + c + \ldots + c' = \left(\sum_{i=1}^{n-1} c \right) + c' \tag{3.1}$$

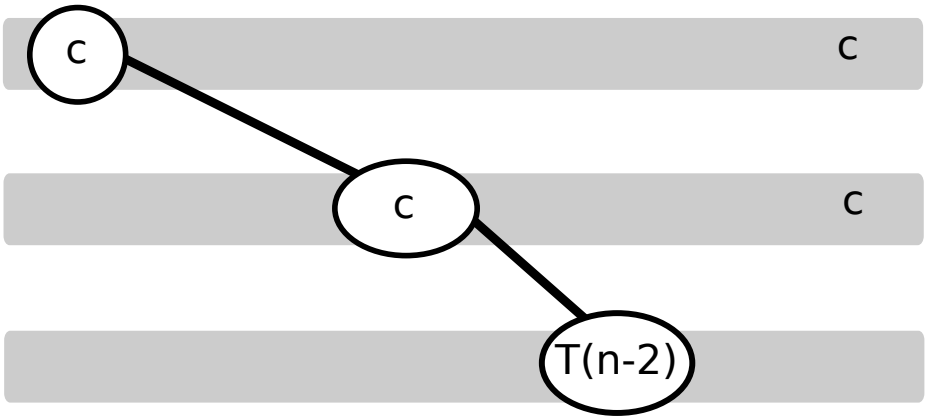
$$= (n-1) \cdot c + c' \in O(n).$$

Expandindo a recorrência para o primeiro passo:

Expandindo a recorrência para o segundo passo:



Expandindo a recorrência para terceiro passos:



Expandindo a recorrência para múltiplos passos:

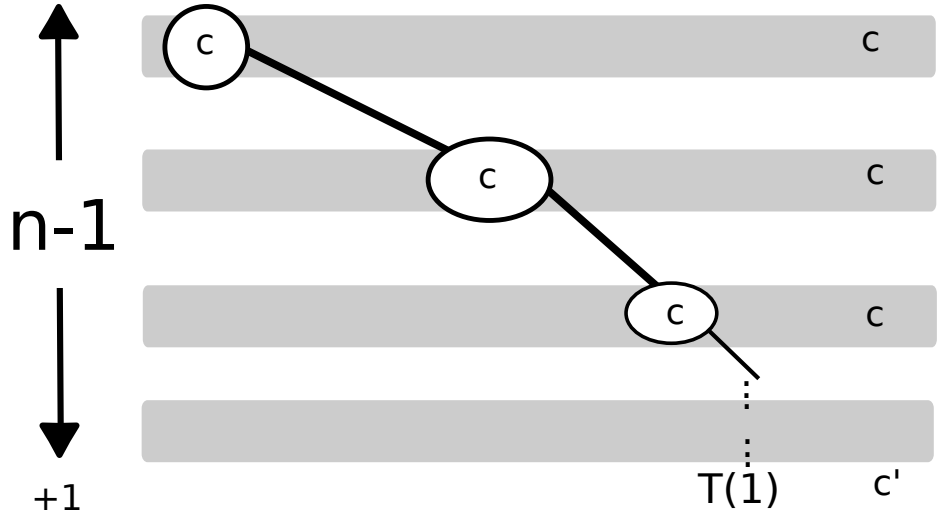


Figura 4 – Árvore de recursão sobre a recorrência $T(n) = T(n-1) + c$.

CAPÍTULO 4

Divisão e Conquista

A Divisão e Conquista é um paradigma em que se resolve um problema em três etapas a cada nível de recursão (CORMEN et al., 2012):

- **Divisão**: divide-se o problema em um número de subproblemas que são instâncias menores do problema original;
- **Conquista**: resolve-se os subproblemas recursivamente. Entretanto, se o tamanho dos subproblemas for suficientemente pequeno, passa-se a resolvê-los diretamente;
- **Combinação**: as soluções para os subproblemas são combinadas para resolver o problema que as originou.

4.1 Mergesort

Para apoiar a explicação de alguns desses métodos, será utilizado o algoritmo de Mergesort, mas que é revisitado aqui nas funções MERGE e MERGESORT.

```
import math


def merge(vetor, p, q, r):
```

```
    n1 = q-p+1
    n2 = r-q
    vetorL = [0] * (n1+1)
    vetorR = [0] * (n2+1)

    for i in range (n1):
        vetorL[i] = vetor[p+i]

    for i in range (n2):
        vetorR[i] = vetor[q+i+1]

    vetorL[n1] = math.inf
    vetorR[n2] = math.inf

    i = 0
    j = 0

    for k in range (p, r+1):
        if vetorL[i] <= vetorR[j]:
            vetor[k] = vetorL[i]
            i += 1
        else:
            vetor[k] = vetorR[j]
            j += 1

    return vetor

def mergesort(vetor, p, r):
```

```
    if p < r:
        q = (p+r)//2
        mergesort(vetor, p, q)
        mergesort(vetor, q+1, r)
        merge(vetor, p, q, r)

def main():
    A = [9, 5, 4,77, 3, 2, 1]
    mergesort(A, 0, len(A)-1)
    print(A)

if __name__ == "__main__":
    main()
```

Uma das características importantes do algoritmo de Mergesort é a possibilidade de ordenar dois vetores previamente ordenados em um terceiro vetor completamente ordenado (e com todos os elementos dos dois) em tempo linear (função MERGE).

Para o contexto do Mergesort, sabe-se que a recorrência que representa sua complexidade de tempo é:

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1. \end{cases} \tag{4.1}$$

4.2 Contar Número de Inversões

Dada uma sequência de números $P = \langle p_1, p_2, \ldots, p_n \rangle$, determinar quantas inversões existem em P. Uma inversão é um par de números a e b, os quais $a > b$ que aparecem em P, nas posições p_i e p_j respectivamente, e $i < j$. O número máximo de inversões possível é $\frac{n^2-n}{2}$.

O problema de contar número de inversões pode ser utilizado para estabelecer similaridades entre diferentes ranqueamentos. Por exemplo: quero comparar qual a similaridade de preferência de filmes entre os usuários x e y, posso utilzar o ranqueamento de cada um deles e contar as inversões para entender o quão distantes estão as duas preferências. Esse tipo de similaridade pode ser utilizada para realizar a recomendação de filmes (KLEINBERG; TARDOS, 2005).

O algoritmo abaixo apresenta uma forma não recursiva de resolver o problema.

```
def contaInversoesIterativo(P):
  c = 0
  for i in range(len(P)-1):
    for j in range (i+1, len(P)):
      if P[i] > P[j]:
        c = c + 1
  return c


def main():
    P = [10,9,8,7,6,5,4,3,2,1]
    print(contaInversoesIterativo(P))



if __name__ == "__main__":
    main()
```

Qual seria a complexidade de tempo do algoritmo apresentado? Será que é possível encontrar algoritmo melhor? A resposta é sim. Há um algoritmo de divisão e conquista para o problema com tempo computacional mais interessante. Pense sobre ele!

4.3 Multiplicação de Inteiros

Esta seção discute a multiplicação de dois números inteiros no qual o algoritmo de tempo quadrático tradicional é melhorado (ou acelerado) ao utilizar o paradigma de divisão e conquista (KLEINBERG; TARDOS, 2005).

No algoritmo tradicional, multiplica-se dois números x e y. Para cada dígito de y, multiplica-se os dígitos de x. Os resultados para cada dígito de y são somados, considerando um deslocamento equivalente a posição do dígito multiplicador de y. Ao contar o número de multiplicações para cada dígito de y, têm-se $O(n)$ operações, no qual n é o número de dígitos do número com maior quantidade de dígitos. Para somar cada um dos resultados obtidos, demanda-se mais $O(n)$ operações.

O algoritmo melhorado se baseia em uma maneira mais eficiente de dividir o produto em somas parciais. Para exemplificar, irá se considerar um número na base 2, mas isso não importa, pois o algoritmo é o mesmo para qualquer base, deve-se apenas se fazer os ajustes necessários. Assuma que $x = x_1 \cdot 2^{\frac{n}{2}} + x_0$, no qual x_1 corresponde aos $\frac{n}{2}$ bits de maior ordem (significância) e x_0 os $\frac{n}{2}$ bits de menor ordem. De maneira similar se considera o número $y = y_1 \cdot 2^{\frac{n}{2}} + y_0$. Desse modo, pode-se considerar a multiplicação de x e y da seguinte forma (KLEINBERG; TARDOS, 2005):

$$\begin{aligned} xy &= (x_1 \cdot 2^{\frac{n}{2}} + x_0)(y_1 \cdot 2^{\frac{n}{2}} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0. \end{aligned} \tag{4.2}$$

O tempo computacional para computar uma multiplicação como essa é de $T(n) \leq 4T\left(\frac{n}{2}\right) + n$.

```
import numpy as np
import math


B = 10 #base (binÃirio, octal, base 10)
```

```
def multiplicacao(x, y):
    #descobre nÃºmero de dÃgitos
    n = max(len(str(int(x))) , len(str(int(y))))

    if n == 1:
        return x*y
    else:
        m = n//2

        x1 = x//(B**m)
        x0 = x % (B**m)
        y1 = y//(B**m)
        y0 = y % (B**m)

        x1y1 = multiplicacao(x1, y1)
        x1y0 = multiplicacao(x1, y0)
        x0y1 = multiplicacao(x0, y1)
        x0y0 = multiplicacao(x0, y0)

        return x1y1 * B**(2*m) + (x1y0 + x0y1) * B**(m) + x0y0

def main():
    print (multiplicacao(3333333, 100000))


if __name__ == "__main__":
    main()
```

No entanto, é possível melhorar considerando o seguinte (truque de Karatsuba):

$$\begin{aligned} xy &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0 \\ &= x_1 y_1 \cdot 2^n + \left((x_1 + x_0) \cdot (y_1 + y_0) - x_0 y_0 - x_1 y_1\right) \cdot 2^{\frac{n}{2}} + x_0 y_0. \end{aligned} \tag{4.3}$$

A parte $(x_1 + x_0) \cdot (y_1 + y_0) - x_0 y_0 - x_1 y_1$ é equivalente a $x_1 y_0 + x_0 y_1$:

$$\begin{aligned} (x_1 + x_0)(y1 + y_0) - x_0 y_0 - x_1 y_1 &= x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0 - x_0 y_0 - x_1 y_1 \\ &= x_1 y_0 + x_0 y_1. \end{aligned} \tag{4.4}$$

Com essa forma de calcular, se reaproveita de cálculo que $x_0 y_0$ e $x_1 y_1$. O código-fonte abaixo exibe o algoritmo que utliza essa forma de multiplicação para a base 10.

```
import numpy as np
import math


B = int(10) #base (binÃirio, octal, base 10)


def multiplicacaoKaratsuba(x, y):
  #descobre nÃºmero de dÃgitos
  n = max(len(str(int(x))) , len(str(int(y))))


  if n == 1:
      return x*y
  else:
      m = n//2


      x1 = x//(B**m)
      x0 = x % (B**m)
      y1 = y//(B**m)
      y0 = y % (B**m)
```

```
        p = multiplicacaoKaratsuba(x1+x0, y1+y0)
        x1y1 = multiplicacaoKaratsuba(x1, y1)
        x0y0 = multiplicacaoKaratsuba(x0, y0)

        return x1y1 * B**(2*m) + (p - x1y1 - x0y0) * B**(m) +
            x0y0

def main():
    print (multiplicacaoKaratsuba(3333333, 100000))


if __name__ == "__main__":
    main()
```

4.3.1 Complexidade

O tempo computacional requerido é $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$.

Desafio

Dê a fórmula fechada para a recorrência $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$, identificando a respectiva complexidade.

4.4 Multiplicação de Matrizes

Para multiplicação de matrizes, há um algoritmo bem conhecido que demanda complexidade de tempo $\Theta(n^3)$. Uma versão do mesmo para multiplicação de duas matrizes quadradas é apresentado no código-fonte abaixo:

```
import numpy as np
```

```
def imprimeMatriz(A):
  print('\n'.join(['' .join(['{:8}'.format(item) for item in row])
      for row in A]))


def multiMatrizQuadrada(A,B):


    n = len(A)
    C = [[0]*n for i in range(n)]


    for i in range (0, n):
        for j in range (0, n):
            C[i][j] = 0
            for k in range (0, n):
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
    return C


def main():
  n = 4
  M1 = np.array([np.array([e*f for e in range(n)]) for f in
     range(n)])
  M2 = np.array([np.array([e*f/2 for e in range(n)]) for f in
     range(n)])
  imprimeMatriz(M1)
  print("x")
  imprimeMatriz(M2)
  print("=")
  imprimeMatriz(multiMatrizQuadrada(M1,M2))
```

```
if __name__ == "__main__":
    main()
```

Um algoritmo simples de divisão e conquista para multiplicação de matrizes pode ser visualizado no código-fonte abaixo. Na linha 6, o algoritmo divide as matrizes em quatro partes de mesmo formato para as matrizes.

```
 import numpy as np
def multiMatrizDC(A, B):
    n = len(A)

    if n == 1:
        return A[0][0]*B[0][0]
    else:
        # particionar A e B em quatro matrizes cada
        A00  =  A[0:len(A)//2, 0:len(A)//2 ]
        A01  =  A[0:len(A)//2, len(A)//2:len(A) ]
        A10  =  A[len(A)//2:len(A), 0:len(A)//2 ]
        A11  =  A[len(A)//2:len(A), len(A)//2:len(A) ]

        B00  =  B[0:len(B)//2, 0:len(B)//2 ]
        B01  =  B[0:len(B)//2, len(B)//2:len(B) ]
        B10  =  B[len(B)//2:len(B), 0:len(B)//2 ]
        B11  =  B[len(B)//2:len(B), len(B)//2:len(B) ]

        # gerando as quatro partes da matriz resultante
        C00 = multiMatrizDC(A00, B00) + multiMatrizDC(A01, B10)
        C01 = multiMatrizDC(A00, B01) + multiMatrizDC(A01, B11)
        C10 = multiMatrizDC(A10, B00) + multiMatrizDC(A11, B10)
        C11 = multiMatrizDC(A10, B01) + multiMatrizDC(A11, B11)
```

```
        #criando a matriz C a partir de suas quatro partes
        C = np.vstack((np.hstack((C00, C01)), np.hstack((C10,
            C11))))

        return C

def main():
  n = 4
  M1 = np.array([np.array([e*f for e in range(n)]) for f in
     range(n)])
  M2 = np.array([np.array([e*f/2 for e in range(n)]) for f in
     range(n)])
  imprimeMatriz(M1)
  print("x")
  imprimeMatriz(M2)
  print("=")
  imprimeMatriz(multiMatrizDC(M1,M2))

if __name__ == "__main__":
    main()
```

Assumindo que as entradas possuem um tamanho na potência de 2 exatamente, torna a análise da complexidade mais simples. A complexidade de tempo é igual $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$, sendo que $T(1) = \Theta(1)$. A parte $\Theta(n^2)$ é devido a soma das matrizes obtidas pelas chamadas recursivas.

Desafio

Dê a fórmula fechada para a recorrência $T(n) \leq 8T\left(\frac{n}{2}\right) + \Theta(n^2)$, identificando a respectiva complexidade.

Ainda há como melhorar a complexidade de tempo, utilizando como base a versão de divisão e conquista. Essa versão mais eficiente é conhecida como algoritmo de Strassen (código-fonte abaixo). A complexidade do algoritmo de Strassen é dada pela recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$, sendo $T(1) = \Theta(1)$.

Desafio

Dê a fórmula fechada para a recorrência $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$, identificando a respectiva complexidade.

```
import numpy as np
def Strassen(A, B):
    n = len(A)

    if n == 1:
        return A[0][0]*B[0][0]
    else:
        # particionar A e B em quatro matrizes cada
        A00  =  A[0:len(A)//2, 0:len(A)//2 ]
        A01  =  A[0:len(A)//2, len(A)//2:len(A) ]
        A10  =  A[len(A)//2:len(A), 0:len(A)//2 ]
        A11  =  A[len(A)//2:len(A), len(A)//2:len(A) ]

        B00  =  B[0:len(B)//2, 0:len(B)//2 ]
        B01  =  B[0:len(B)//2, len(B)//2:len(B) ]
        B10  =  B[len(B)//2:len(B), 0:len(B)//2 ]
```

```
        B11   =   B[len(B)//2:len(B), len(B)//2:len(B) ]


        #gerando as partes S1 atÃ© S10
        S1 = B01 - B11
        S2 = A00 + A01
        S3 = A10 + A11
        S4 = B10 - B00
        S5 = A00 + A11
        S6 = B00 + B11
        S7 = A01 - A11
        S8 = B10 + B11
        S9 = A00 - A10
        S10= B00 + B01


        #gerando as partes P1 atÃ© P7 (as multiplicacoes)
        P1 = Strassen(A00, S1)
        P2 = Strassen(S2, B11)
        P3 = Strassen(S3, B00)
        P4 = Strassen(A11, S4)
        P5 = Strassen(S5, S6)
        P6 = Strassen(S7, S8)
        P7 = Strassen(S9, S10)


        #gerando as quatro partes da matriz resultante
        C00 = P5 + P4 - P2 + P6
        C01 = P1 + P2
        C10 = P3 + P4
        C11 = P5 + P1 - P3 - P7
```

```
        #criando a matriz C a partir de suas quatro partes
        C = np.vstack((np.hstack((C00, C01)), np.hstack((C10,
            C11))))

        return C

def main():
  n = 4
  M1 = np.array([np.array([e*f for e in range(n)]) for f in
     range(n)])
  M2 = np.array([np.array([e*f/2 for e in range(n)]) for f in
     range(n)])
  imprimeMatriz(M1)
  print("x")
  imprimeMatriz(M2)
  print("=")
  imprimeMatriz(Strassen(M1,M2))

if __name__ == "__main__":
    main()
```

CAPÍTULO 5

Árvores

5.1 Árvores Binárias de Busca

Como o nome sugere, Árvores Binárias de Busca é uma estrura de dados alternativa para buscas binárias. Então, que tal a gente começar a seção com um exemplo de busca binária?

```
usernames = sorted([
    'Miguel', 'Davi', 'Arthur', 'Sophia', 'Alice', 'Julia'
])
def print_bsearch_step(container, query='Sophia'):
    n, middle = len(container), len(container)//2
    query, pivot = query, container[middle]
    print("Data:", container)
    print("Split:", container[:middle], container[middle:])
    print("Query:", query)
    print("Pivot:", pivot)
    print(query+(" < " if query < pivot else " >= ")+pivot)
print_bsearch_step(usernames)
```

```
Data: ['Alice', 'Arthur', 'Davi', 'Julia', 'Miguel', 'Sophia']
```

```
Split: ['Alice', 'Arthur', 'Davi'] ['Julia', 'Miguel', 'Sophia']
Query: Sophia
Pivot: Julia
Sophia >= Julia
```

Imagine que o Facebook possua um sistema que registra os nomes dos usuários cadastrados. Quando um usuário loga no Facebook, esse sistema precisa confirmar se o nome desse usuário existe. Uma busca binária confirma rapidamente a existência ou não do nome. Essa rapidez deve-se a sucessivas divisões simétricas do espaço do busca. No exemplo acima uma busca por 'Sophia' dividiu os nomes de usuários pela metade e, então, decidiu com uma única comparação qual das duas partes 'Sophia' estaria - se estiver cadastrada no sistema. Independente de qual parte foi escolhida, esse passo certamente diminui o espaço de busca na mesma quantidade - pela metade.

```
print_bsearch_step(usernames+['Bernado'], query='Bernado')
```

```
Data: ['Alice', 'Arthur', 'Davi', 'Julia', 'Miguel', 'Sophia',
   'Bernado']
Split: ['Alice', 'Arthur', 'Davi'] ['Julia', 'Miguel', 'Sophia',
   'Bernado']
Query: Bernado
Pivot: Julia
Bernado < Julia
```

Mas uma busca binária só funciona corretamente se o vetor de dados estiver ordenado. Veja acima como a busca binária escolheria a parte incorreta da divisão, já que a inserção de 'Bernado' na sua posição livre do vetor (no fim) deixou 'usernames' fora de ordem.

Aqui nos deparamos com um problema: toda vez que um novo usuário cria uma conta, o sistema necessita inserir o novo nome e, ainda, perservar a ordem alfabética. Inserir um novo elemento e reordenadar todo o vetor adicionaria ao usuário de uma

busca binária o indesejável tempo assintótico de execução na ordem de O(n) para operações de inserção (e deleção).

Não seria interessante se inserir um novo nome modificasse a estrutura de dados diretamente na posição que mantém a ordenação de seus elementos? Essa é a idéia por trás da estrutura de dados que iremos estudar nesta seção: 'Árvore Binária de Busca'.

Uma árvore binária de busca tem a seguinte forma:

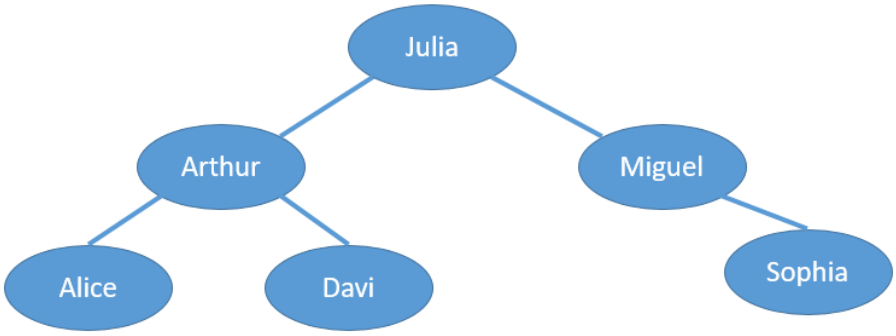


Figura 5 – Um exemplo de árvore binária de busca (balanceada).

Cada nó da árvore possui até dois filhos. O filho da esquerda contém um valor **menor** do que o pai, enquanto o filho da direita contém um valor **maior**. Essa propriedade nos possibilita realizar uma busca binária a partir do nó raiz da árvore. Por exemplo, digamos que queremos localizar 'Sophia' na árvore da figura-exemplo. Começamos a busca pela raiz: 'Julia'. Como 'S' vem após 'J', seguimos pelo filho da direita, e encontramos o nó do 'Miguel'. 'S' também vem após 'M', então pegamos a direita mais uma vez. E encontramos o nó da 'Sophia'!

A operação de busca, assim como as outras operações básicas que veremos, leva um tempo proporcional à altura da árvore (h). Neste exemplo a altura da árvore é $h = 3 \approx log_2\ 7$.

Podemos representar uma árvore binária através de uma estrutura de dados vinculada, na qual cada nó é um objeto.

```
class BST_Node:
    def __init__(self, key, parent=None):
```

```
        self.key = key
        self.parent = parent
        self.left = None
        self.right = None
```

Além da informação chave (’key’) - e, opcionalmente, de informações adicionais (como um ’value’) -, cada nó também contém atributos para o nó filho da esquerda (’left’), para o nó filho da direita (’right’) e para o seu nó pai (’parent’). Caso um filho ou pai não exista, o atributo apropriado contém o valor nulo (’None’, para Python). O nó raiz da árvore é o único cujo atributo ’parent’ é nulo.

Abaixo temos uma função recursiva que converte um vetor de busca binária para uma árvore binária de busca.

```
def sorted_array_to_bst(container, parent=None):
    if not container:
        return None
    length = len(container)
    if length == 1:
        node = BST_Node(container[0], parent=parent)
    else:
        middle = length // 2
        pivot = container[middle]
        node = BST_Node(pivot, parent=parent)
        node.left = sorted_array_to_bst(container[:middle], node)
        node.right = sorted_array_to_bst(container[middle+1:],
            node)
    return node


def array_to_bst(container):
    return sorted_array_to_bst(sorted(container))
```

A ideia chave de *sorted_array_to_bst* é criar nós que reproduzam um passo da busca binária - de dividir o espaço de busca pela metade. Assim cada chamada da função divide o vetor (espaço de busca) em três partes - elementos do vetor como sub-árvore da esquerda, um elemento como chave do nó raiz, elementos como sub-árvore da direita - de maneira que as sub-árvores da esquerda e da direita tenham um mesmo número de nó. Com essa invariante, *sorted_array_to_bst* garante a construção de uma árvore "balanceada". Já, já ficará claro por quê esse adjetivo é importânte. Por hora, vamos ver o que *sorted_array_to_bst* nos oferece.

```
bst_usernames = sorted_array_to_bst(usernames)
```

Árvore de busca pronta. Agora, um método de impressão ...

```
def bst_print(node, height=1, tab="\t"):
    if node:
        bst_print(node.left, height+1, tab)
        print(tab*(height-1)+str(node.key))
        bst_print(node.right, height+1, tab)
```

```
bst_print(bst_usernames)
```

```
                Alice
        Arthur
                Davi
Julia
                Miguel
        Sophia
```

Pronto. Com uma adaptação do algoritmo INORDER-TREE-WALK, conseguimos imprimir a nossa árvore. O nome mais a esquerda, 'Julia', é o nó raiz da árvore. Se estivermos procurando por um nome que suceda 'Julia' na ordem alfabética, basta procurar por um nó abaixo de 'Julia' - 'Miguel' e 'Sophia'. 'Sophia' é o filho a esquerda de Júlia - pois é nó mais a esquerda daqueles que estão abaixo de 'Júlia'. Se quisermos um

nome que anteceda 'Julia', a gente sobe - 'Arthur', 'Davi', 'Alice'. Assim como na figura exemplo, nossa árvore captura que 'Miguel' antecede 'Sophia' na ordem alfabética. Mas há uma diferença: houve uma rotação e, agora, o nó de 'Sophia' é pai de 'Miguel'.

Com a propriedade das árvores que estamos estudando, a operação de busca faz uma caminhada do nó raiz para um de seus filhos até encontrar o nó com o valor chave especificado. Caso a árvore não possua um nó com esse valor, a operação retorna nulo.

```
def bst_recursive_search(node, key):
    if node is None:
        return None # nenhum nodo na arvore possui a chave
    if node.key == key:
        return node
    if key < node.key:
        return bst_recursive_search(node.left, key)
    else:
        return bst_recursive_search(node.right, key)
```

Formalmente, a propriedade das árvores binárias de busca é:

- Digamos que x é um nó de uma árvore binária de busca. Se y é um nó da subárvore esquerda de x, então $y.key \leq x.key$. Se y é um nó da subárvore direita de x, então $y.key \geq x.key$.

A função $sorted\_array\_to\_bst$ garante essa propriedade quando os elementos do vetor que recebe como parâmetro seguem uma ordem crescente de valores chave. O vetor 'usernames' satisfaz essa condição e, portanto, $sorted\_array\_to\_bst$ constrói uma árvore binária de busca com a propriedade necessária.

Podemos re-escrever a função $sorted\_array\_to\_bst$ em uma versão iterativa, a qual costuma ser mais eficiente na maoria dos computadores.

```
def bst_search_iter(node, key):
    while node is not None and node.key != key:
```

```
        if key < node.key:
            node = node.left
        else:
            node = node.right
    return node
```

E agora podemos tratar do que nos motivou a implementar essa estrutura: inserções.

Operações de inserção causam uma mudança na árvore. Sua estrutura de dados precisa ser modificada para refletir a mudança. Mas também é necessário preservar a propriedade da ordem relativa entre esquerda-chave-direita. Inserir um novo elemento é, na verdade, bem simples.

```
def bst_insert(root, key):
    #parent recebe None se a raiz nÃ£o existir
    parent = root.parent if root else None
    # encontrar onde estÃi o candidato a recebe o nodo da nova
       chave
    candidate = root
    while candidate is not None:
        parent = candidate
        if key < candidate.key:
            candidate = candidate.left
        else:
            candidate = candidate.right

    # insere nodo na Ãirvore
    node = BST_Node(key)
    node.parent = parent
    if parent is None:
        root = node
```

```
    elif node.key < parent.key:
        parent.left = node
    else:
        parent.right = node
    return root
```

A função iterativa *bst_insert* recebe dois parâmetros: o nó raiz da árvore (’root’) e o nó a ser inserido nela (’node’). Primeiro, *bst_insert* caminha pela árvore, seguindo a lógica de busca binária, até encontrar um nó folha nulo. Em seguida *bst_insert* substitui o nó nulo pelo nó a ser inserido. Para isso precisamos de uma referência para o nó pai. A variável ’parent’ mantém o ponteiro para o pai do nó que estamos visitando. Quando o ’while’ terminar, ’parent’ conterá uma referência para o pai do nó folha encontrado. Com essa informação, *bst_insert* modifica dois atributos para inserir ’node’ na árvore. Altera o atributo ’node.parent’. E o atributo apropriado de ’parent’: ’parent.left’ ou ’parent.right’. *bst_insert* também cuida do caso em que a árvore está vazia e, portanto, o nó inserido passa a ser a raiz da árvore (’root’).

```
bst_print(bst_usernames)
print("\n# Inserting 'Bernardo'\n")
bst_usernames = bst_insert(bst_usernames, 'Bernardo')
bst_print(bst_usernames)
```

```
                Alice
        Arthur
                Davi
Julia
                Miguel
        Sophia


# Inserting 'Bernardo'
```

```
                Alice
        Arthur
                        Bernado
                Davi
Julia
                Miguel
        Sophia
```

bst_insert consegue atualizar a árvore em O(log h) e continuar realizando buscas em O(log h). Até certo ponto.

O resultado de inserir 'Bernado' na árvore não tão bom quanto esperado, né? A inserção aumentou a altura (h) da árvore e desbalanceou ela um pouquinho.

O que aconteceria se *bst_insert* fosse utilizado para construir a árvore do zero?

```
bst_by_sorted_keys = None
for key in usernames:
    bst_by_sorted_keys = bst_insert_key(bst_by_sorted_keys, key)
bst_print(bst_by_sorted_keys)
```

```
Alice
        Arthur
                Davi
                        Julia
                                Miguel
                                        Sophia
```

Vê como a árvore inclina para uma única direção? Ela está desbalanceada, sua altura ficou igual ao número de nós ($h = n$), e, assim, o tempo de busca passou a ser O(n).

Essa é uma desvantagem das Árvores Binárias de Busca: seu desempenho depende da árvore estar balanceada. Há árvores cujas operações de inserção (e deleção) preservam o balanceamento dos nós. Exemplos são red-black-trees e b-trees (onde b-trees

são um tipo especial de árvore binária para gerenciar o armazenamento de bancos de dados em disco rígido, por exemplo).

Por quê ocorreu o desbalanceamento?

Em vista de *bst_insert* nunca cogitar que a chave do nó raiz mostrou-se uma escolha ruim com os valores subsequêntes que recebeu.

Vamos deixar essas questões de como outras árvores mantém o balanceamento para outra hora. Agora a pergunta pertinente é: a árvore seria balanceada caso as inserções ocorressem aleatóriamente?

```
import numpy as np
bst_by_random_keys = None
for key in np.random.permutation(usernames):
    bst_by_random_keys = bst_insert_key(bst_by_random_keys, key)
bst_print(bst_by_random_keys)
```

```
        Alice
                Arthur
Davi
                        Julia
                Miguel
        Sophia
```

Quando *bst_insert* insere valores-chave aleatórios, a altura esperada para a árvore é O(log n). Temos uma árvore mais balanceada! (Caso seja do seu interesse, o capítulo 12.4 que Cormen escreveu em seu livro apresenta uma prova formal desta complexidade.)

Uma árvore de busca é como um conjunto dinâmico, e suporta várias operações - incluindo SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, PREDECESSOR, SUCESSOR. Essas operações nos permitem utilizá-la tanto como um dicionário quanto como

uma fila de prioridade.

A seguir veremos como podemos implementar as outras operações básicas. Mas deixaremos DELETE por último.

```
def bst_minimum(node):
    while node.left is not None:
        node = node.left
    return node
```

```
def bst_maximum(node):
    while node.right is not None:
        node = node.right
    return node
```

Podemos encontrar o nó de uma árvore binária de busca cuja chave é mínima se seguirmos pelos nós filhos da esquerda da raiz até encontrar um nó sem filho a esquerda (None). A propriedade destas árvores garante a corretude de $bst\_minimum$. (Se um nó x não possui uma subárvore à esquerda então, como todo valor chave presente na subárvore a direita é maior ou igual a x.key, o valor mínimo presente na árvore enraizada em x é x.key. Se um nó x possui uma subárvore a esquerda então, como nenhuma chave na árvore a direita é menor do que x.key e nenhuma chave na subárvore a esquerda é maior que x.key, a chave mínima da árvore enraizada no nó x reside na subárvore a esquerda.)

O código para encontrar a chave máxima é simétrico. Estas duas funções rodam em tempo proporcional à altura da árvore: O(log h) já que a sequência de nós visitados forma um caminho simples a partir da raiz da árvore.

Algumas vezes também precisamos encontrar o sucessor de um nó, determinado pela ordem crescente de chaves de uma caminhada inorder (a mesma que fizemos com $print\_bst$). Se todas as chaves são distintas, o sucessor de um nó x é o nó cuja chave é a menor dentre os nós cujas chaves são maiores do que 'x.key'. A estrutura de uma

árvore binária de busca nos permite determinar um sucessor sem precisar comparar chaves. A função abaixo retorna o sucessor de um nó (’node’), se um sucessor existir; caso node.key seja a maior chave da árvore, um sucessor não existe, e retornamos None.

```
def bst_successor(node):
    if node.right is not None:
        return bst_minimum(node.right)
    parent = node.parent
    while parent is not None and node == parent.right:
        node = parent
        parent = node.parent
    return parent
```

A função que determina o precedessor é simétrica.

```
def bst_predecessor(node):
    if node.left is not None:
        return bst_maximum(node.left)
    parent = node.parent
    while parent is not None and node == parent.left:
        node = parent
        parent = node.parent
    return parent
```

Quase lá. Agora o que falta é a operação de deleção.

Para essa operação temos que lidar com quatro casos. Cada caso refere-se a uma diferente situação em que nó a ser deletado se encontra:

1. O nó a ser deletado não possui filhos.

2. O nó a ser deletado só possui um filho a direita.

3. O nó a ser deletado só possui um filho a esquerda.

4. O nó a ser deletado possui um filho a esquerda e um filho a direita.

E cada caso atualiza o nó pai de uma maneira diferente:

1. O nó pai atualiza o referido nó para o valor nulo.
2. O nó pai atualiza o referido nó para o filho da direita.
3. O nó pai atualiza o referido nó para o filho da esquerda.
4. O nó pai atualiza o referido nó para o seu sucessor.

Esse último caso, em que o nó possui dois filhos, necessita das seguintes ações:

- Encontrar o sucessor do nó. Ou seja, o nó da sub-árvore a direita cuja chave é a de menor valor.
- Garantir que o sucessor é a raiz da sub-árvore a direita.
- Transplantar a sub-árvore a direita como filho do nó pai.
- Transplantar a sub-árvore a esquerda como filho esquerdo da sub-árvore a direita.

Figura 6 ajuda a entender cada caso, e sua respectivas ações.

Na figura, a letra q denota o nó pai de z. O nó z pode ser a raiz da árvore ($q = None$), filho a esquerda de q, ou filho a direita de q. (a) Nó z não possui filho a esquerda. Substituímos z por seu filho a direita r; r pode ser nulo. (b) Nó z possui filho a esquerda l mas não possui um filho a direita. Substituímos z por seu filho a esquerda l. (c) Nó z possui dois filhos e o filho a direita é seu sucessor y. Chamemos de x o filho a direita do nó sucessor y; x pode ser um nulo. Substituímos z por x, atualizando o filho a esquerda de x para l e mantendo x como seu filho a direita. (d) Nó z possui dois filhos e o seu sucessor encontra-se dentro da subárvore enraizada no seu filho a direita r (i.e. $y \neq x$). Substituímos y por seu próprio filho a direita x, e fazemos y ser pai de r. Por fim, fazemos y ser filho de q e pai de l.

Conclusão? Antes de tratar da operação de deleção, precisamos de uma operação que transplante um nó de uma posição para uma outra.

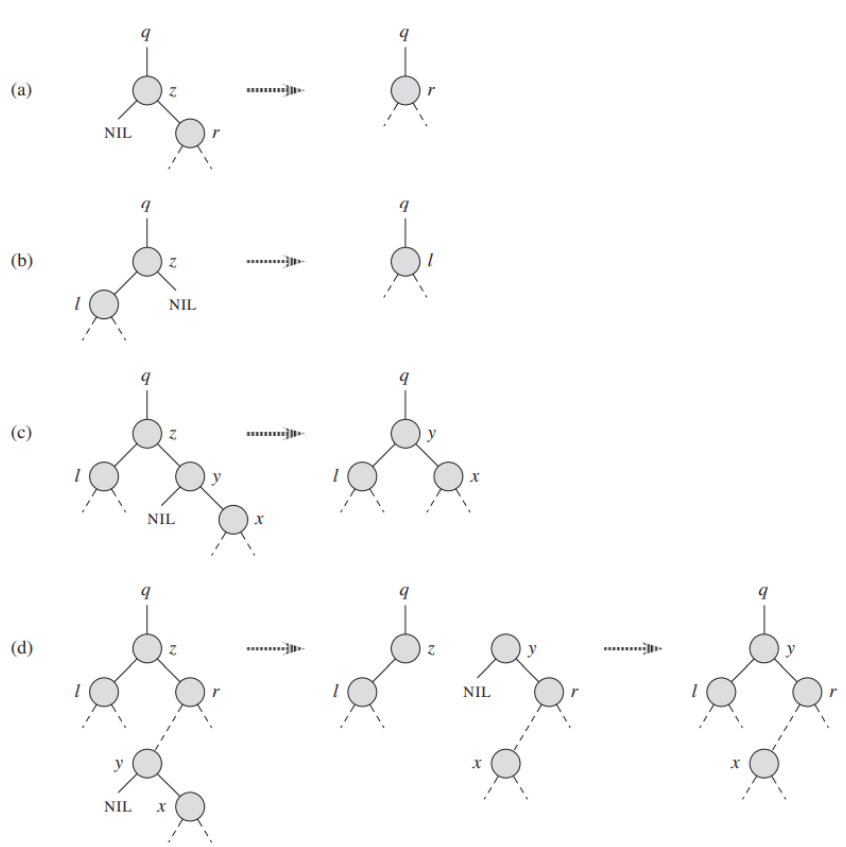


Figura 6 – Deleção de um nó z de uma árvore binária de busca.

```
def bst_transplant(root, pivot, target):
    if pivot.parent is None: # pivot is the root
        root = target # node becomes the root
    elif pivot == pivot.parent.left:
        pivot.parent.left = target
    else:
        pivot.parent.right = target
    if target is not None:
        target.parent = pivot.parent
    return root
```

Pronto. $bst\_transplant$ faz as devidas modificações do atributo 'parent' para substituir um nó 'pivot' por um nó 'target'. As primeiras linhas da função atualizam o nó pai

de 'pivot'. Caso 'pivot' não possua um pai, 'pivot' é o nó raiz e *bst_transplant* atualiza 'root'. Caso 'pivot' seja o filho a esquerda, *bst_transplant* atualiza o atributo 'left' de 'pivot.parent'. Caso 'pivot' seja o filho a direita, *bst_transplant* atualiza o atributo 'right' de 'pivot.parent'. Por fim, *bst_transplant* atualiza o atributo 'parent' de 'target', se o mesmo não for nulo.

```
print('### Original tree ###')
bst_usernames = sorted_array_to_bst(usernames)
bst_print(bst_usernames)
pivot = bst_search_iter(bst_usernames, 'Arthur')
target = bst_search_iter(bst_usernames, 'Sophia')
bst_usernames = bst_transplant(bst_usernames, pivot, target)
print('### Transplate Arthur by Sophia ###')
bst_print(bst_usernames)
print('### Subtree rooted at Arthur ###')
bst_print(pivot)
print('### Subtree rooted at Sophia ###')
bst_print(target)
print('### Undo ###')
bst_usernames = bst_transplant(bst_usernames, target, pivot)
bst_print(bst_usernames)
```

```
### Original tree ###
                Alice
        Arthur
                Davi
Julia
                Miguel
        Sophia
### Transplate Arthur by Sophia ###
```

```
                Miguel
        Sophia
Julia
                Miguel
        Sophia
### Subtree rooted at Arthur ###
        Alice
Arthur
        Davi
### Subtree rooted at Sophia ###
        Miguel
Sophia
### Undo ###
                Alice
        Arthur
                Davi
Julia
                Miguel
        Sophia
```

bst_transplant atualiza apenas o atributo 'parent', e encarrega a função chamadora da responsabilidade de atualizar os demais atributos, como 'target.parent.left' ou 'target.parent.right', e 'target.left', e 'target.right'. Mantenha isso em mente quando for implementar a operação de deleção.

```
def bst_delete(root, node):
    if node.left is None: #caso (a)
        root = bst_transplant(root, node, node.right)
        node.right = None
    elif node.right is None: #caso (b)
```

```
        root = bst_transplant(root, node, node.left)
        node.left = None
    else: #casos (c) e (d)
        successor = bst_successor(node)
        if successor != node.right: #caso (d)
            bst_transplant(None, successor, successor.right)
            successor.right = node.right
            node.right.parent = successor
            successor.parent = None
        #casos (c) e (d)
        root = bst_transplant(root, node, successor)
        successor.left = node.left
        node.left.parent = successor
        node.right = None
        node.left = None
    return root
```

bst_delete trata dos quatro casos de deleção da seguinte forma. Primeiro trata dos dois casos em que 'node' não possui um filho a esquerda - o que inclui o caso em que 'node' não possui filho algum -, e transplanta o filho da direita no lugar de 'node'. Se 'node' possui um filho a esquerda, *bst_delete* checa se possui um filho a direita. Caso não possua, *bst_delete* transplanta o filho da esquerda no lugar de 'node'. Caso contrário, 'node' possui dois filhos e *bst_delete* procura pelo sucessor de 'node'. Com o sucessor em mãos, *bst_delete* faz uma comparação para saber se é necessário ajustar a sub-árvore a direita de 'node'. Se 'successor' não é o filho a direita de 'node', *bst_delete* segue o diagrama (d) e move o sucessor para se tornar a raiz da sub-árvore a direita de 'node'. Após garantir que temos uma subárvore enraizada em 'successor', *bst_delete* transplanta 'node' por 'successor'.

Cada linha de *bst_delete* leva um tempo constante de execução, inclusive as cha-

madas de *bst_transplant*. A única excessão é a chamada de *bst_successor*, a qual leva O(h). Portanto, *bst_delete* executa em O(h) em uma árvore de altura *h*.

```
print("### Tree before deletion ###")
bst_print(bst_usernames)
print("### Tree after deletion of Arthur ###")
bst_usernames = bst_delete(bst_usernames,
   bst_search_iter(bst_usernames, 'Arthur'))
bst_print(bst_usernames)
print("### Tree after deletion of Sophia ###")
bst_usernames = bst_delete(bst_usernames,
   bst_search_iter(bst_usernames, 'Sophia'))
bst_print(bst_usernames)
print("### Tree after deletion of Julia ###")
bst_usernames = bst_delete(bst_usernames,
   bst_search_iter(bst_usernames, 'Julia'))
bst_print(bst_usernames)
```

```
### Tree before deletion ###
                Alice
        Arthur
                Davi
Julia
                Miguel
        Sophia
### Tree after deletion of Arthur ###
                Alice
        Davi
Julia
                Miguel
```

```
        Sophia
### Tree after deletion of Sophia ###
                Alice
        Davi
Julia
        Miguel
### Tree after deletion of Julia ###
                Alice
        Davi
Miguel
```

Podemos observar que *bst_delete* parece estar funcionando como queríamos. Porém, assim como *bst_delete*, nossa função pode desbalancear a árvore e, assim, degradar o tempo de suas operações para O(n).

Antes de finalizar essa seção, que tal a gente comparar a execução da busca binária com relação às duas estruturas de dados que vimos até o momento: vetor e árvore?

	array	binary search tree
SEARCH	O(log n)	O(h)
INSERT	O(n)	O(h)
DELETE	O(n)	O(h)
SUCCESSOR	O(1)	O(h)
PREDECESSOR	O(1)	O(h)
MINIMUM	O(1)	O(h)
MAXIMUM	O(1)	O(h)

Como a busca por um elemento leva O(log n) na média e O(n) no pior caso, a vantagem de uma árvore binária de busca está em reduzir o tempo de execução das operações de inserção e deleção. Mas isso vem com um custo para as outras operações básicas de conjuntos dinâmicos, como MINIMUM.

Heaps são conjuntos dinâmicos mais apropriados para realizar operações agregadoras de mínimo e máximo com atualizações de inserções e deleções. Essa vantagem, porém, vem com o custo de não suportar operações de busca.

Há ainda outras estruturas de dados que o leitor possa estar interessado. Red-Black Trees e B-Trees são opções de árvores binárias que mantér o balanceamento após inserções e deleções. Radix-Trees, Fibonacci Heaps e Splay-Trees são exemplos de ainda outras estruturas de árvores.

Caso o leitor esteja interessado estruturas de dados mais avançadas, sugerimos:

- B-trees.
- Red-Black trees.
- Heaps.
- Fibonacci Heaps.
- Radix-Trees.
- Splay trees. Geek for Geekspost, or Code Forces post.

Fonte:

- Gokking Algorithms, p. 205-206
- Cormen et al. (2012).

CAPÍTULO 6

Grafos

6.1 Definições Iniciais

Antes de visitar a representação de grafos, é importante que saibamos o que são vértices e arestas. Vértices geralmente são representados como unidades, elementos ou entidades, enquanto as arestas representam as ligações/conexões entre pares de vértices. Geralmente, chamaremos o conjunto de vértices de V e o conjunto de arestas de E. Define-se que $E \subseteq V \times V$. Também usaremos n e m para denotarem o número de vértices e arestas respectivamente, então $n = |V|$ e $m = |E|$. O número de arestas possível em um grafo é $\frac{n^2-n}{2}$.

Um grafo pode ser representado de duas formas (CORMEN et al., 2012). A primeira forma é chamada de lista de adjacências e tem mais popularidade em artigos científicos. Nela, o grafo é representado como uma dupla para especificar vértices e arestas. Por exemplo, para um grafo G, pode-se dizer que o mesmo é uma dupla $G = (V, E)$, especificando assim que o grafo G possui um conjunto V de vértices e E de arestas. A segunda forma seria uma através de uma matriz binária, chamada de matriz de adjacência. Normalmente representada pela letra $A(G)$, a matriz é definida por $A(G) = \{0,1\}^{|V|\times|V|}$, a qual seus elementos $a_{u,v} = 1$ se existir uma aresta entre os vértices u e v. Um exemplo das das formas para um mesmo grafo pode ser visualizado no Exemplo 6.1.1.

Example 6.1.1. A Figura 7 exibe um grafo de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}). \tag{6.1}$$

A representação por uma matriz de adjacência ficaria assim

$$A(G) = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}.$$

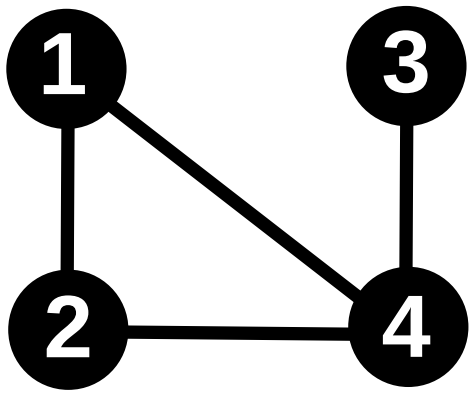


Figura 7 – Exemplo de grafo com 4 vértices e 4 arestas.

6.1.1 Grafos Valorados ou Ponderados

Um grafo é valorado quando um peso ou valor é associado a suas arestas. Na literatura, a definição do grafo passa a ser uma tripla $G = (V, E, w)$, na qual V é o conjunto de vértices, E é o conjunto de arestas e $w : e \in E \to \mathbb{R}$ é a função que especifica o valor.

Quando não se possui valornas arestas, parte-se de uma relação binária entre existir ou não uma aresta entre dois vértices. Neste caso, se u e v possui uma aresta, geralmente se simboliza essa ligação com o valor 1, e se não existir 0.

Em uma matriz de adjacências para grafos valorados, o valor das arestas aparecem nas células da matriz. Em um par de vértices que não possui valor estabelecido (não há aresta), representa-se com uma lacuna ou com um valor simbólico para o problema que o grafo representa. Por exemplo, se os valores representam as distâncias, geralmente se associa o valor infinito aos pares de vértices que não possuem arestas.

Um exemplo de grafo valorado e suas representações pode ser visualizado no Exemplo 6.1.2.

Example 6.1.2. A Figura 8 exibe um grafo valorado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{\{1,2\},\{1,4\},\{2,4\},\{3,4\}\}, w). \tag{6.2}$$

A função w teria os seguintes valores: $w(\{1,2\}) = 8$, $w(\{1,4\}) = 9$, $w(\{2,4\}) = 5$ e $w(\{3,4\}) = 7$.

A representação por uma matriz de adjacência ficaria assim

$A(G) =$

	1	2	3	4
1	0	8	0	9
2	8	0	0	5
3	0	0	0	7
4	9	5	7	0

ou desta outra forma para o caso de uma aplicação a problemas que envolvam distâncias

$A(G) =$

	1	2	3	4
1	∞	8	∞	9
2	8	∞	∞	5
3	∞	∞	∞	7
4	9	5	7	∞

.

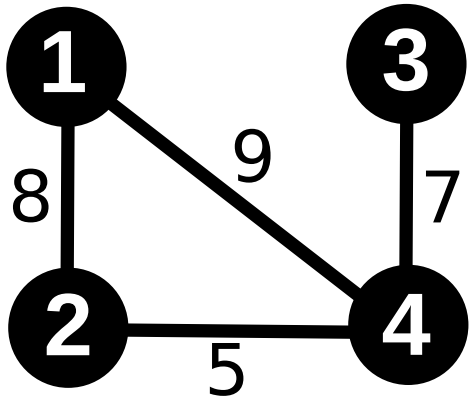


Figura 8 – Exemplo de grafo valorado com 4 vértices e 4 arestas.

Grafos com Sinais

Para representar alguns problemas, utiliza-se valores negativos associados às arestas. Um exemplo disso, seriam grafos que representem relações de amizade e de inimizade. Para amizade, utiliza-se o valor 1 e para inimizade o valor -1. Nesse caso, não dizemos que o grafo é valorado ou ponderado, mas sim um grafo com sinais. Quando os valores negativos e positivos podem ser diferentes de 1 e -1, diz-se que os grafos são valorados e com sinais.

6.1.2 Grafos Orientados

Um grafo orientado é aquele no qual suas arestas possuem direção. Nesse caso, não chamamos mais de arestas e sim de arcos. Um grafo orientado é definido como uma dupla $G = (V, A)$, a qual V é o conjunto de vértices e A é o conjunto de arcos. O conjunto de arcos é composto por pares ordenados (u, v), os quais $u, v \in V$ e representam um arco saindo de u e incidindo em v. Duas funções importantes devem ser consideradas nesse contexto: a função de arcos saintes $\delta^+(v) = \{(v, u) : (v, u) \in A\}$ e arcos entrantes $\delta^-(v) = \{(u, v) : (u, v) \in A\}$.

O Exemplo 6.1.3 exibe a representação de um grafo orientado.

Example 6.1.3. A Figura 9 exibe um grafo orientado de 4 vértices e 4 arestas. Na representação por listas de adjacências, o grafo pode ser representado da seguinte forma

$$G = (\{1,2,3,4\}, \{(1,4),(2,1),(4,2),(4,3)\}). \tag{6.3}$$

A representação por uma matriz de adjacência ficaria assim

$$A(G) = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 \end{array}.$$

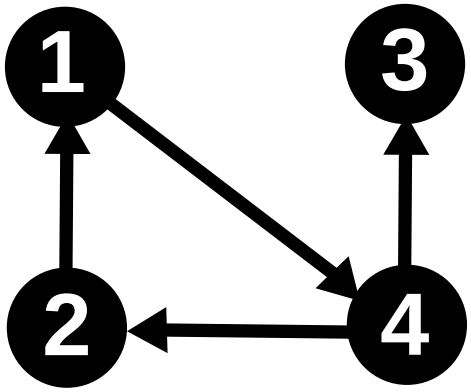


Figura 9 – Exemplo de grafo orientado com 4 vértices e 4 arcos.

6.1.3 Hipergrafo

Um hipergrafo $H = (V, E)$ é um grafo no qual as arestas podem conectar qualquer número de vértices. Cada aresta é chamada de hiperaresta $E \subseteq 2^V \setminus \{\}$.

6.1.4 Multigrafo

Um multigrafo $G = (V, E)$ é um grafo que permite múltiplas arestas para o mesmo par de vértices. Logo, não se tem mais um conjunto de arestas, mas sim uma tupla de arestas.

Para o exemplo da Figura 10, têm-se $E = (\{1,2\},\{1,2\},\{1,4\},\{2,4\},\{3,4\},\{3,4\})$.

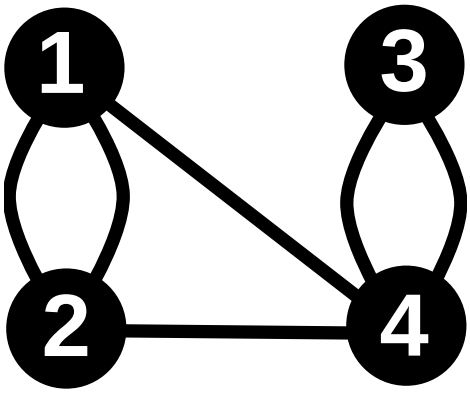


Figura 10 – Exemplo de um multigrafo com 4 vértices e 6 arestas.

6.1.5 Grau de um Vértice

O grau de um vértice é a quantidade de arestas que se conectam a determinado vértice. É denotada por uma função d_v, onde $v \in V$. Em um grafo orientado, o número de arcos saintes para um vértice v é denotado por d_v^+, e o número de arcos entrantes é denotado por d_v^-.

6.1.6 Igualdade e Isomorfismo

Diz-se que dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são iguais se $V_1 = V_2$ e $E_1 = E_2$. Os dois grafos são considerados isomorfos se existir uma função bijetora (uma-por-uma) para todo $v \in V_1$ e para todo $u \in V_2$ preserve as relações de adjacência (NETTO, 2006).

6.1.7 Partição de Grafos

Uma partição de um grafo é uma divisão disjunta de seu conjunto de vértices. Um grafo $G = (V, E)$ é dito k-partido se existir uma partição $P = \{p_i | i = 1, \ldots, k \wedge \forall\, j \in \{1, \ldots, k\}, j \neq i (p_i \cap p_j \neq \{\})\}$. Quando $k = 2$, dize que o grafo é bipartido (NETTO, 2006).

6.1.8 Vizinhança

A vizinhança de vértices é diferente para grafos não-orientados e orientados. Para um grafo grafo não-orientado $G = (V, E)$, uma função de vizinhança é definida por $N : v \in V \rightarrow \{u \in V : \{v, u\} \in E\}$ e indica o conjunto de todos os vizinhos de um vértice específico. Para o grafo do Exemplo 6.1.1, $N(1) = \{2, 4\}$.

Para um grafo orientado $G = (V, A)$, diz-se que um vértice $u \in V$ é sucessor de $v \in V$ quando $(v, u) \in A$; e $u \in V$ é antecessor de $v \in V$ quando $(u, v) \in A$. As funções de vizinhança para um grafo orientado G são $N^+ : v \in V \rightarrow \{u \in V : (v, u) \in A\}$, $N^- : v \in V \rightarrow \{u \in V : (u, v) \in A\}$, e $N(v) = N^+(v) \cup N^-(v)$.

Diz-se que a vizinhança de v é fechada quando esse mesmo vértice se inclui no conjunto de vizinhos. A função que representa vizinhança fechada v é simbolizada neste texto como $N_*(v) = N(v) \cup \{v\}$.

As funções de vizinhança também podem ser utilizadas para identificar um conjunto de vértices vizinhos de um grupo de vértices em um grafo $G = (V, E)$ (orientado ou não). Nesse contexto, $N(S) = \bigcup_{v \in S} N(v)$, $N^+(S) = \bigcup_{v \in S} N^+(v)$, e $N^-(S) = \bigcup_{v \in S} N^-(v)$.

As noções de sucessor e antecessor podem ser aplicadas iterativamente. As Equações (6.4), (6.5), (6.6) e (6.7) exibem exemplos de fechos transitivos diretos.

$$N^0(v) = \{v\} \tag{6.4}$$

$$N^{+1}(v) = N^+(v) \tag{6.5}$$

$$N^{+2}(v) = N^+\left(N^{+1}(v)\right) \tag{6.6}$$

$$N^{+n}(v) = N^+\left(N^{+(n-1)}(v)\right) \tag{6.7}$$

Chama-se de fecho transitivo direto aqueles que correspondem aos vizinhos sucessivos e os inversos os que correspondem aos vizinhos antecessores. Um fecho transitivo

direto de um vértice v de um grafo $G = (V, E)$ são todos os vértices atingíveis a partir v no grafo G; ele é representado pela função $R^+(v) = \bigcup_{k=0}^{|V|} N^{+k}(v)$. Um fecho transitivo inverso de v é o conjunto de vértices que atingem v; ele é representado pela função $R^-(v) = \bigcup_{k=0}^{|V|} N^{-k}(v)$. Diz-se que w é descendente de v se $w \in R^+(v)$. Diz-se que w é ascendente de v se $w \in R^-(v)$.

6.1.9 Grafo Regular

Um grafo não-orientado $G = (V, E)$ que tenha $d(v) = k \forall v \in V$ é chamado de grafo k-regular ou de grau k. Um grafo orientado $G_o = (V, A)$ que possui a propriedade $d^+(v) = k \forall v \in V$ é chamado de grafo exteriormente regular de semigrau k. Se G_o tiver $d^-(v) = k \forall v \in V$ é chamado de grafo interiormente regular de semigrau k.

6.1.10 Grafo Simétrico

Um grafo orientado $G = (V, A)$ é simétrico se $(u, v) \in A \iff (v, u) \in A \forall u, v \in V$.

6.1.11 Grafo Anti-simétrico

Um grafo orientado $G = (V, A)$ é anti-simétrico se $(u, v) \in A \iff (v, u) \notin A \forall u, v \in V$.

6.1.12 Grafo Completo

Um grafo completo $G = (V, E)$ é completo se $E = V \times V$.

Grafos bipartidos completos $G_B = ((X, Y), E)$ possuem $E = X \times Y$.

6.1.13 Grafo Complementar

Para um grafo $G = (V, E)$, um grafo complementar é definido por $G^c = \overline{G} = (V, (V \times V) \setminus E)$.

6.1.14 Percursos em Grafos

"Um percurso, itinerário ou cadeia é uma família de ligações sucessivamente adjacentes, cada uma tendo uma extremidade adjacente a anterior e a outra à subsequente (à execeção da primeira e da última)" (NETTO, 2006). Diz-se que um percurso é aberto quando a última ligação é adjacente a primeira. Têm-se desse modo um ciclo.

Um percurso é considerado simples se não repetir ligações (NETTO, 2006).

6.2 Representações Computacionais

Duas formas de representação computacional de grafos são amplamente utilizadas. São elas "listas de adjacências" e "por matriz de adjacências" (CORMEN et al., 2012). Elas possuem vantagens e desvantagens principamente relacionadas à complexidade computacional (consumo de recursos em tempo e espaço). Detalhes sobre vantagens e desvantagens não aparecerão nesse documento. Um de nossos objetivos do momento será implementar e avaliar as duas formas de representação.

6.2.1 Lista de Adjacências

A representação de um grafo $G = (V, E)$ por listas de adjacências consiste em um arranjo, chamado aqui de Adj. Esse arranjo é composto por $|V|$ listas, e cada posição do arranjo representa as adjacências de um vértice específico (CORMEN et al., 2012). Para cada $\{u, v\} \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$ e $Adj[v] = (\ldots, u, \ldots)$ quando G for não-dirigido. Quando G for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = (\ldots, v, \ldots)$.

Para grafos ponderados, Cormen et al. (2012) sugere o uso da própria estrutura de adjacências para armazenar o peso. Dado um grafo ponderado não-dirigido $G = (V, E, w)$, para cada $\{u, v\} \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w(\{u, v\})\big), \ldots\big)$ e $Adj[v] = \big(\ldots, \big(u, w(\{u, v\})\big), \ldots\big)$. Quando o grafo for dirigido, para cada $(u, v) \in E$, têm-se $Adj[u] = \big(\ldots, \big(v, w((u, v))\big), \ldots\big)$.

O Algoritmo abaixo representa a carga de um grafo dirigido e ponderado $G = (V, A, w)$ em uma lista de adjacências Adj.

```
1  def createAdj(Grafo):
2  Adj = []
3  for v in Grafo.vertices:
4    Adj.append([])
5  for (u,v) in Grafo.arestas:
6    Adj[u-1].append((v,Grafo.pesos[(u,v)]))
7  return Adj
```

6.2.2 Matriz de Adjacências

Uma matriz de adjacência é uma representação de um grafo através de uma matriz A. Para um grafo não-dirigido $G = (V, E)$, $A = \mathbb{B}^{|V|\times|V|}$, na qual cada elemento $a_{u,v} = 1$ e $a_{v,u} = 1$ se $\{u, v\} \in E$; $a_{u,v} = 0$ e $a_{v,u} = 0$ caso $\{u, v\} \notin E$. Para todo grafo não-dirigido G, $a_{u,v} = a_{v,u}$.

Para um grafo dirigido $G = (V, X)$, $A = \mathbb{B}^{|V|\times|V|}$, na qual cada elemento $a_{u,v} = 1$ se $(u, v) \in A$; $a_{u,v} = 0$ e $a_{u,v} = 0$ caso $(u, v) \notin X$.

Para um grafo não-dirigido e ponderado $G = (V, E, w)$, a matriz será formada por células que comportem o tipo de dado representado pelos pesos. Assumindo que os pesos serão números reais, então a matriz de adjacências será $A = \mathbb{R}^{|V|\times|V|}$. Cada elemento $a_{u,v} = w(\{u, v\})$ e $a_{v,u} = w(\{u, v\})$ se $\{u, v\} \in E$; $a_{u,v} = \epsilon$ e $a_{v,u} = \epsilon$ caso $\{u, v\} \notin E$. ϵ é um valor que representa a não conexão, geralmente 0, $+\infty$ ou $-\infty$ dependendo do contexto de aplicação.

6.2.3 Exercícios

Implemente as duas bibliotecas para grafos. Preencha a seguinte tabela a partir da análise computacional, de acordo com as operações abaixo determinadas.

	Lista de Adjacências	Matriz de Adjacências
Teste se $\{u, v\} \in E$		
Percorrer vizinhos		
Grau de um vértice		

6.2.4 Biblioteca

Abaixo segue um exemplo de uma classe para abrir grafos não-dirigidos no formato Pajek (.net).

```
1  class Grafo: # por "lista" de adjacencias
2    def __init__(self, arquivo):
3
4      with open(f'{arquivo}', 'r') as manipulador:
5        vertices = False
6        edges = False
7        for linha in manipulador:
8          campos = linha.split()
9          if campos[0][:4] == "*ver":
10           vertices = True
11           self.vertices = {}
12           continue
13         if campos[0][:4] == "*edg":
14           vertices = False
15           edges = True
16           self.adjs = {v:{} for v in self.vertices}
17           continue
18         if vertices:
19           self.vertices[int(campos[0])] =
                linha[len(campos[0]):-1]
```

```
20        if edges:
21          u = int(campos[0])
22          v = int(campos[1])
23          w = 1.0 if len(campos)<=2 else float(campos[2])
24          self.adjs[u][v] = w
25          self.adjs[v][u] = w
26
27    def N(self, v):
28      return self.adjs[v]
29
30    def d(self, v):
31      return len(self.adjs[v])
32
33    def pesoAresta(self, u, v):
34      if u in self.adjs:
35        if v in self.adjs[u]:
36          return self.adjs[u][v]
37      return None
```

Abaixo segue um exemplo de uma classe para abrir grafos dirigidos no formato Pajek (.net).

```
1  class GrafoDirigido: # por "lista" de adjacencias
2    def __init__(self, arquivo = None):
3      #cria grafo vazio
4      if arquivo is None:
5        self.vertices = {}
6        self.adjsSaintes   = {}
7        self.adjsEntrantes = {}
8        return None
```

```
9
10    #cria grafo a partir de arquivo
11    with open(f'{arquivo}', 'r') as manipulador:
12      vertices = False
13      edges = False
14      for linha in manipulador:
15        campos = linha.split()
16        if campos[0][:4] == "*ver":
17          vertices = True
18          self.vertices = {}
19          continue
20        if campos[0][:4] == "*arc":
21          vertices = False
22          edges = True
23          self.adjsSaintes   = {v:{} for v in self.vertices}
24          self.adjsEntrantes = {v:{} for v in self.vertices}
25          continue
26        if vertices:
27          self.vertices[int(campos[0])] =
                linha[len(campos[0]):-1]
28        if edges:
29          u = int(campos[0])
30          v = int(campos[1])
31          w = 1.0 if len(campos)<=2 else float(campos[2])
32
33          self.adjsSaintes[u][v] = w
34          self.adjsEntrantes[v][u] = w
35
```

```
36   def NPlus(self, v):
37     return self.adjsSaintes[v]
38   def NMinus(self, v):
39     return self.adjsEntrates[v]
40   def dPlus(self, v):
41     return len(self.adjsSaintes[v])
42   def dMinus(self, v):
43     return len(self.adjsEntrantes[v])
44
45   def pesoAresta(self, u, v):
46     if u in self.adjsSaintes:
47       if v in self.adjsSaintes[u]:
48         return self.adjsSaintes[u][v]
49     return None
```

6.3 Buscas em Grafos

6.3.1 Busca em Largura

Dado um grafo $G = (V, E)$ e uma origem s, a **busca em largura** (Breadth-First Search - BFS) explora as arestas/arcos de G a partir de s para cada vértice que pode ser atingido a partir de s. É uma exploração por nível. O procedimento descobre as distâncias (número de arestas/arcos) entre s e os demais vértices atingíveis de G. Pode ser aplicado para grafos orientados e não-orientados (CORMEN et al., 2012).

O algoritmo pode produzir uma árvore de busca em largura com raiz s. Nesta árvore, o caminho de s até qualquer outro vértice é um caminho mínimo em número de arestas/arcos (CORMEN et al., 2012).

O código-fonte abaixo descreve as operações realizadas em uma busca em largura.

Nele, criam-se três estruturas de dados que serão utilizadas para armazenar os resultados da busca. O arranjo $visitados[v]$ é utilizado para determinar se um vértice $v \in V$ foi visitado ou não; $distancia[v]$ determina a distância percorrida até encontrar o vértice $v \in V$; e $antecessor[v]$ determina o vértice antecessor ao $v \in V$ em uma busca em largura a partir de s (CORMEN et al., 2012).

```
1  import math
2  import networkx as nx
3
4  def BuscaEmLargura(g, verticeOrigem):
5    #preparando estruturas de dados
6    visitados    = {id: False    for id in g.vertices}
7    distancia    = {id: math.inf for id in g.vertices}
8    antecessor = {id: None     for id in g.vertices}
9
10   #configurando vÃ©rtice de origem
11   visitados[verticeOrigem] = True
12   distancia[verticeOrigem] = 0
13
14   #preparando fila de visitas
15   fila = []
16   fila.append(verticeOrigem)
17
18   #propagaÃ§Ã£o de visitas
19   while len(fila) > 0: #enquanto houver elementos na fila
20     u = fila.pop(0) #retira elementos por ordem de entrada na
         estrutura
21     for v in g.adjs[u]:
22       if visitados[v] == False:
```

```
23          visitados[v] = True
24          distancia[v] = distancia[u] + 1
25          antecessor[v] = u
26          fila.append(v)
27    return distancia, antecessor
28
29 def main():
30    arquivo = "/content/gdrive/My Drive/instancias/karate.net"
31
32    g1 = Grafo(arquivo)
33    s = 1
34    (D, A) = BuscaEmLargura(g1, s)
35    print(D)
36
37    #exibindo Ãirvore
38    A.pop(s)
39    g = nx.Graph({u:{A[u]:A[u]} for u in A})
40    nx.draw_networkx(g,  with_labels=True)
41
42 if __name__ == "__main__":
43    main()
```

6.3.1.1 Complexidade da Busca em Largura

O número de operações de enfileiramento e desenfileiramento é limitado a $|V|$ vezes, pois visita-se no máximo $|V|$ vértices. Como as operações de enfileirar e desenfileirar podem ser realizadas em tempo $\Theta(1)$, então para realizar estas operações demanda-se tempo de $O(|V|)$. Deve-se considerar ainda, que muitas arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma BFS a varredura de todas as

adjacências, que demandaria $\Theta(|E|)$. Diz-se então, que a complexidade computacional da BFS é $O(|V|+|E|)$.

6.3.2 Busca em Profundidade

A busca em profundidade (Depth-First Search - DFS) realiza a visita a vértices cada vez mais profundos/distantes de um vértice de origem s até que todos os vértices sejam visitados. Parte-se a busca do vértice mais recentemente descoberto do qual ainda saem arestas inexploradas. Depois que todas as arestas foram visitadas no mesmo caminho, a busca retorna pelo mesmo caminho para passar por arestas inexploradas. Quando não houver mais arestas inexploradas a busca em profundidade pára (CORMEN et al., 2012).

O código-fonte abaixo apresenta um pseudo-código para a busca em profundidade. Note que no lugar de usar uma fila, como na busca em largura, utiliza-se uma pilha. Os arranjos $visitados[v]$, $tempoV[v]$, e $antecessor[v]$ $\forall v \in V$ são respectivamente o arranjo de marcação de visitados, do tempo de visita e do vértice antecessor à visita.

```
1  import math
2 import networkx as nx
3
4 def BuscaEmProfundidade(g, verticeOrigem):
5   #preparando estruturas de dados
6   visitados    = {id: False    for id in g.vertices}
7   tempoV    = {id: math.inf for id in g.vertices}
8   antecessor = {id: None     for id in g.vertices}
9
10  #tempo de visita inicial
11  tempo = 0
12
13  #configurando vÃ©rtice de origem
```

```
14    visitados[verticeOrigem] = True
15
16    #preparando pilha de visitas
17    pilha = []
18    pilha.append(verticeOrigem)
19
20    #propagaÃ§Ã£o de visitas
21    while len(pilha) > 0: #enquanto houver elementos na pilha
22      u = pilha.pop(len(pilha)-1) #retira elementos por ordem de
            entrada na estrutura
23      tempo += 1
24      tempoV[u] = tempo
25      for v in g.adjs[u]:
26        if visitados[v] == False:
27          visitados[v] = True
28          antecessor[v] = u
29          pilha.append(v)
30    return tempoV, antecessor
31
32  def main():
33    arquivo = "/content/gdrive/My Drive/instancias/karate.net"
34
35    g1 = Grafo(arquivo)
36    s = 1
37    (T, A) = BuscaEmProfundidade(g1, s)
38    print(T)
39
40    #exibindo Ãrvore
```

```
41    A.pop(s)
42    g = nx.Graph({u:{A[u]:A[u]} for u in A})
43    nx.draw_networkx(g,  with_labels=True)
44
45  if __name__ == "__main__":
46    main()
```

Uma versão recursiva pode ser vista abaixo.

```
1   import math
2   import networkx as nx
3
4   def visitaBP(g, v, visitados, tempoV, antecessor, tempo):
5     visitados[v] = True
6     tempo[0]+=1
7     tempoV[v] = tempo[0]
8     for u in g.adjs[v]:
9       if visitados[u] == False:
10        antecessor[u] = v
11        visitaBP(g, u, visitados, tempoV, antecessor, tempo)
12
13
14  def BuscaEmProfundidadeRec(g, verticeOrigem):
15    #preparando estruturas de dados
16    visitados    = {id: False    for id in g.vertices}
17    tempoV    = {id: math.inf for id in g.vertices}
18    antecessor = {id: None     for id in g.vertices}
19
20    #tempo de visita inicial
21    tempo = [0]
```

```
22
23    #chama a busca
24    visitaBP(g, verticeOrigem, visitados, tempoV, antecessor,
          tempo)
25
26    return tempoV, antecessor
27
28  def main():
29    arquivo = "/content/gdrive/My Drive/instancias/karate.net"
30
31    g1 = Grafo(arquivo)
32    s = 1
33    (T, A) = BuscaEmProfundidadeRec(g1, s)
34    print(A)
35
36    #exibindo Ãirvore
37    A.pop(s)
38    g = nx.Graph({u:{A[u]:A[u]} for u in A})
39    nx.draw_networkx(g,  with_labels=True)
40
41  if __name__ == "__main__":
42    main()
```

6.3.2.1 Complexidade da Busca em Profundidade

Da mesma maneira que a complexidade da busca em largura, a busca em profundidade possui complexidade $O(|V|+|E|)$. As operações da pilha resultariam tempo $O(|V|)$. Muitos arestas/arcos incidem em vértices já visitados, então inclui-se na complexidade de uma DFS a varredura de todas as adjacências, que demandaria $\Theta(|E|)$.

CAPÍTULO 7

Algoritmos Gulosos

Um algoritmo é dito guloso ou cobiçoso se constrói uma solução em pequenos passos, tomando uma decisão "míope" a cada passo para otimizar algum critério relacionado ao problema (geralmente não óbvio). Quando um algoritmo guloso resolve algum problema não trivial, geralmente implica na descoberta de algum padrão útil na estrutura do problema (KLEINBERG; TARDOS, 2005).

7.1 Exemplo de Algoritmos Gulosos

7.1.1 Agendamento de Intervalos

Considere um conjunto de intervalos $R = \{r_1, r_2, \ldots, r_n\}$ no qual cada intervalo r_i inicia em $s(r_i)$ e termina em $f(r_i)$. Diz-se que dois intervalos são compatíveis se eles não possuem sobreposição de tempo. Deseja-se encontrar um subconjunto de R com o maior número de intervalos compatíveis entre si. Pense sobre quais regras naturais poderiam ser empregadas nessa busca (como critério simples) e imagine como elas trabalham. Alguns exemplos simples (KLEINBERG; TARDOS, 2005):

- Inserir pela ordem de $s(r_i)$;

- Inserir pela ordem de menor intervalo, pois quer se selecionar o maior número possível deles;

- Contagem de recursos não compatíveis e utilizar aquele com o menor número de incompatibilidades.

Nenhum desses critérios levariam a solução ótima para todas as instâncias do problema. A Figura 11 ajuda a enteder o porquê. Utilizando a ordem $s(r_i)$, o exemplo presente na Figura 11(a) a solução seria apenas um intervalo. Quanto a ordem pelo menor intervalo, a Figura 11(b) demonstra que apenas um intervalo seria marcado para uma instância cuja solução ótima conta com dois intervalos. Ao utilizar o critério a quantidade mínima de intervalos não compatíveis, a Figura 11(c) demonstra uma instância cuja solução ótima seria selecionar os 4 intervalos da primeira linha. No entanto, com esse último critério, iniciaria-se selecionando o segundo intervalo da segunda linha e depois qualquer outro intevalo não conflitante na esquerda e na direita, ou seja, no máximo três intervalos.

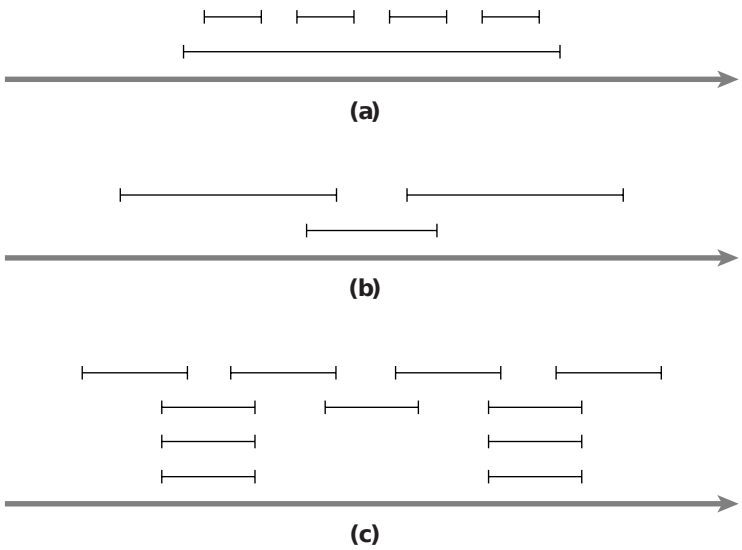


Figura 11 – Exemplos de Kleinberg e Tardos (2005) para refutar as três regras naturais levantadas anteriormente para o problema de agendamento do maior número de intervalos.

Para resolver o problema com uma estratégia gulosa, Kleinberg e Tardos (2005)

utilizam o menor valor de $f(i)$. Então, inicialmente se escolhe o intervalo de menor tempo que termina mais cedo na instância. Depois, seleciona-se o próximo intervalo compatível de menor tempo de fim, até que não haja mais intervalos compatíveis.

O pseudo-código abaixo exibe o algoritmo guloso sugerido por (KLEINBERG; TARDOS, 2005).

```
1  import math
2
3  #ordenaÃ§Ã£o pelo critÃ©rio
4  def merge_Intervalo(vetor, p, q, r,crit):
5      n1 = q-p+1
6      n2 = r-q
7      vetorL = [0] * (n1+1)
8      vetorR = [0] * (n2+1)
9
10     for i in range (n1):
11         vetorL[i] = vetor[p+i]
12
13     for i in range (n2):
14         vetorR[i] = vetor[q+i+1]
15
16     vetorL[n1] = math.inf
17     vetorR[n2] = math.inf
18
19     i = 0
20     j = 0
21
22     for k in range (p, r+1):
23         if crit[vetorL[i]] <= crit[vetorR[j]]:
```

```
24              vetor[k] = vetorL[i]
25              i += 1
26          else:
27              vetor[k] = vetorR[j]
28              j += 1
29
30      return vetor
31
32  def mergesort_Intervalo(vetor, p, r, crit):
33      if p < r:
34          q = (p+r)//2
35          mergesort_Intervalo(vetor, p, q, crit)
36          mergesort_Intervalo(vetor, q+1, r, crit)
37          merge_Intervalo(vetor, p, q, r, crit)
38
39  #algoritmo guloso de intervalos
40  def maximoIntervalos(R, s, f):
41    L = R.copy()
42
43    #mergesort pela ordem de fim de intervalo
44    f[math.inf] = math.inf #necessario para a funÃ§Ã£o merge do
          mergesort
45    mergesort_Intervalo(L, 0, len(L)-1, f)
46    #No lugar de mergesort, poderia ser usada a funÃ§Ã£o sorted do
          Python..
47    #... com o lambda especificando o critÃ©rio, conforme abaixo:
48    #L = sorted(L, key = lambda x: f[x])
49
```

```
50    A = []
51    i = 0
52    while i < len(L):
53      A.append(L[i])
54      j = i + 1
55      while j < len(L) and s[L[j]] <= f[L[i]]:
56        j+=1
57      i=j
58
59    return A
60
61  def main():
62      R = ["a", "b", "c", "d", "e"]
63      s = {"a": 1,  "b": 2, "c": 4, "d": 6, "e": 8}
64      f = {"a": 10, "b": 3, "c": 5, "d": 7, "e": 9}
65      A = maximoIntervalos(R, s, f)
66      print("MÃ¡ximo de "+str(len(A))+": "+str(A))
67
68  if __name__ == "__main__":
69      main()
```

7.1.1.1 Complexidade do Problema de Agendamento de Intervalos

Analisando a complexidade de tempo computacional para o algoritmo, destacam-se duas partes mais significativas. A primeira é relativa ao algoritmo de ordenação utilizado na linha 1. Sabe-se que o algoritmo mais eficiente para ordenação demanda $\Theta(n \log_2 n)$. Além dessa parte do algoritmo, há o laço de repetição nas linhas 4 à 9. Pode-se observar que as instruções internas ao laço de repetição da linha 4 serão executadas um número de vezes igual a $|R|$. Então, o algoritmo demanda tempo computacional de

$\Theta(|R|\log_2|R|)$.

7.1.2 Problema da Mochila Fracionário

Considere o seguinte conjunto de itens $I = \{1,2,\ldots,n\}$, que para cada item i há um estoque de peso total $w_i \in \mathbb{R}^+$ e o valor monetário total de todo estoque do item i é dado por $v_i \in \mathbb{R}$. Temos também uma mochila para carregar alguns (ou todos) os itens. Essa mochila tem capacidade de peso dada por $W \in \mathbb{R}$, que deve ser utilizado como limite/restrição, porque a soma dos pesos dos itens colocados na mochila não pode ultrapassar W .

Deseja-se obter o maior valor monetário possível, sabendo que os itens podem ser colocados de forma fracionada (pode-se colocar metade de um item, por exemplo).

Pense em um projeto de algoritmo guloso para o problema em questão.

7.1.3 Dijkstra

O algoritmo de Dijkstra resolve o problema de encontrar um caminho mínimos de fonte única em um grafo $G = (V, E, w)$ ponderados dirigidos ou não. Para esse algoritmo, as arestas/arcos não devem ter pesos negativos. Então, a função de pesos é redefinida como $w : E \to \mathbb{R}^+_*$ (CORMEN et al., 2012).

O algoritmo repetidamente seleciona o vértice de menor custo estimado até então. Quando esse vértice é selecionado, ele não é mais atualizado e sua distância é propagada para suas adjacências. A estrutura de dados C é utilizada no pseudo-código abaixo para definir se um vértice foi visitado (contém **true**) ou não (contém **false**).

```
1 import math
2
3 def dijkstra_inocente(g, s):
4   #inicializa estruturas de dados
5   D = {v:math.inf for v in g.vertices}
```

```
6    A = {v:None for v in g.vertices}
7    C = {v:False for v in g.vertices}
8
9    #configura custo do vÃ©rtice de origem
10   D[s] = 0
11
12   #busca caminhos para cada vÃ©rtice iterativamente
13   while len([v for v in g.vertices if C[v] == False])>0:
        #enquanto existir vÃ©rtices nÃ£o visitados
14     u = min([v for v in g.vertices if C[v] == False ],
          key=lambda x:D[x]) #seleciona o vÃ©rtice com a menor
          distancia nÃ£o visitado
15     C[u] = True
16     for v in g.N(u):
17       if C[v] == False and D[v] > D[u] + g.pesoAresta(u,v):
18         D[v] = D[u] + g.pesoAresta(u,v)
19         A[v] = u
20   return D,A
21
22 def main():
23   arquivo = "/content/gdrive/My Drive/instancias/fln_pequena.net"
24
25   g1 = Grafo(arquivo)
26   s = 1
27   (D, A) = dijkstra_inocente(g1, s)
28   print(D)
29
30   #exibindo Ãirvore
```

```
31    A.pop(s)
32    g = nx.Graph({u:{A[u]:A[u]} for u in A})
33    nx.draw_networkx(g,  with_labels=True, labels=g1.vertices)
34
35  if __name__ == "__main__":
36    main()
```

7.1.3.1 Complexidade de Dijkstra

Se o algoritmo de Dijkstra manter uma fila de prioridades mínimas para mapear a distância estimada no lugar de D, o algoritmo torna-se mais eficiente. Seria utilizada uma operação do tipo "EXTRACT-MIN" no lugar da descoberta do vértice u, para encontrar o vértice com a menor distância. Ao extraí-lo da estrutura de prioridade, não mais seria necessário. Poderia-se gravar sua distância mínima em uma estrutura auxiliar e não mais utilizar a estrutura de visitas C. Ao atualizar as distâncias, poderia-se utilizar a operação de "DECREASE-KEY" da fila de prioridades no lugar da atualização da chave $K[v]$.

Para essa fila de prioridades, poderia se utilizar um Heap, como o Heap Binário, no qual a implementação das operações supracitadas tem complexidade de tempo computacional $O(\log_2 n)$ para o "DECREASE-KEY" e $O(\log_2 n)$ para o "EXTRACT-MIN". Para essa aplicação, $n = |V|$. Utilizando essa estrutura de dados, sabe-se que no máximo executa-se $|E|$ operações de "DECREASE-KEY" e $|V|$ operações de "EXTRACT-MIN". Então a complexidade computacional seria $O\big((|V|+|E|)\log_2|V|\big)$. Com Heap Fibonacci, é possível obter resultados ainda melhores em tempo computacional.

7.1.4 Árvores Geradoras Mínimas

Uma árvore geradora é um subconjunto acíclico (uma árvore) de todos os vértices de um grafo. Dado um grafo ponderado $G = (V, E, w)$, o problema de encontrar uma arvore geradora mínima é aquele no qual busca-se encontrar uma árvore que conecte

todos os vértices do grafo G, tal que a soma dos pesos das arestas seja o menor possível. Seja T uma árvore geradora, diz-se que o custo da árvore geradora de T é dado por $w(T) = \sum_{\{u,v\} \in T} w\big(\{u,v\}\big)$.

Este capítulo apresenta dois métodos para encontrar árvores geradoras mínimas: Kruskal e Prim. Esses dois algoritmos gulosos se baseiam em um método genérico apresentado por Cormen et al. (2012). Dado um grafo não-dirigido e ponderado $G = (V, E, w)$, esse método genérico encontra uma árvore geradora mínima.

O Algoritmo 1 apresenta o método genérico para uma árvore geradora mínima para G. O método se baseia na seguinte invariante de laço: "Antes de cada iteração, A é um subconjunto de alguma árvore geradora mínima.". A cada iteração, determina-se uma aresta que pode ser adicionada a A sem violar a invariante de laço. A aresta segura é uma aresta que se adicionada a A mantém a invariante.

Algoritmo 1: Método Genérico de Cormen et al. (2012).

Input : um grafo $G = (V, E, w)$

```
1 A ← {}
2 while A não formar uma árvore geradora do
3     encontrar uma aresta {u, v} que seja segura para A
4     A ← A ∪ {{u, v}}
5 return A
```

7.1.4.1 Algoritmo de Kruskal

O algoritmo de Kruskal inicia com $|V|$ árvores (conjuntos de um vértice cada). Ele encontra uma aresta segura e a adiciona a uma floresta, que está sendo construída, conectando duas árvores. Essa aresta $\{u, v\}$ é de peso mínimo. Suponha que C_1 e C_2 sejam duas árvores conectadas por uma aresta $\{u, v\}$, sabe-se que essa deve ser uma aresta leve e que $\{u, v\}$ é uma aresta segura para C_1 (pois C_1 e C_2 são árvores formadas por conjuntos disjuntos de vértices). Kruskal é um algoritmo guloso, pois ele adiciona à floresta uma aresta de menor peso possível a cada iteração.

O código-fonte do algoritmo de Kruskal pode ser visualizado abaixo. O algoritmo se

inicia definindo as $|V|$ árvores desconectadas; cada uma contendo um vértice. Então, cria-se uma lista das arestas L ordenadas por peso. Depois, iterativamente, se tenta inserir uma aresta leve em duas árvores que contém nodos não foram conectadas ainda. Quando a inserção ocorre, a estrutura de dados $S[v]$ que mapeia a árvore de cada vértice v é é atualizada. O procedimento se repete até que todas as arestas tenham sido avaliadas no laço.

```
1  def kruskal(g):
2    #define uma Ãirvore para cada vÃ©rtice (S[v] contÃ©m os nodos
         da Ãirvore)
3    S = [{v} for v in range(len(g.vertices)+1)]
4    #critÃ©rio: pelo menor peso
5    L = sorted(g.arestas, key=lambda x: g.pesoAresta(x[0], x[1]))
6    #define a estrutura da soluÃ§Ã£o
7    A = []
8    for [u, v] in L:
9      #se as Ãirvores forem diferentes, {u, v} Ã© uma aresta segura
10     if S[u] != S[v]:
11       #adiciona aresta Ã  soluÃ§Ã£o
12       A.append([u, v])
13       #atualiza Ãirvores
14       x = S[u].union(S[v])
15       for z in  x:
16         S[z] = x
17   #retorna a soluÃ§Ã£o (conjunto de arestas)
18   return A
19
20
21 def main():
```

```
22    arquivo = "/content/gdrive/My Drive/instancias/agm_tiny.net"
23
24    g1 = Grafo(arquivo)
25    A = kruskal(g1)
26    print(A)
27    print('Soma: ', sum([g1.pesoAresta(u, v) for [u, v] in A]))
28
29  if __name__ == "__main__":
30    main()
```

Complexidade do Algoritmo de Kruskal

A complexidade de tempo do algoritmo de Kruskal depende da estrutura de dados utilizada para implementar o mapeamento das árvores de cada vértice, ou seja, da estrutura S algoritmo de Kruskal. Para a implementação mais eficiente, verifique as operações de conjuntos disjuntos no Capítulo 21 de Cormen et al. (2012). Sabe-se que para ordenar o conjunto de arestas na linha 5, deve-se executar um algoritmo de ordenação. O mais eficiente conhecido demanda tempo $\Theta(|E|\log_2|E|)$.

Desafio

Qual estrutura de dado implementa a versão mais eficiente do Algoritmo de Kruskal?

7.1.4.2 Algoritmo de Prim

O algoritmo de Prim é (também) um caso especial do método genérico apresentado no Algoritmo 1. O algoritmo de Prim é semelhante ao algoritmo de Dijkstra, como pode ser observado no código-fonte abaixo. As arestas no vetor A formam uma árvore única. Essa árvore tem raízes em um vértice arbitrário r. Essa arbitrariedade não gera problemas de corretudo no algoritmo, pois uma árvore geradora mínima deve conter todos os

vértices do grafo. A cada iteração, seleciona-se o vértice que é atingido por uma aresta de custo mínimo, sendo que o vértice selecionado adiciona suas adjacências e pesos na estrutura de prioridade Q, que, futuramente, selecionará a chave de menor custo, ou seja, o vértice conectado a árvore que possui o menor custo. Esse comportamento adiciona-se apenas arestas seguras em A.

Antes de cada iteração do laço de repetição mais externo, a árvore obtida é dada por $\{\{v, A[v]\} : v \in V \backslash \{r\} \backslash Q\}$. Os vértices que já participam da solução final são $V \backslash Q$. Além disso, para todo $v \in Q$, se $A_v \neq$**null**, então $K[v] < \infty$ e $K[v]$ é o peso de uma resta leve $\{v, A[v]\}$ que conecta o vértice v a algum vértice que já está na árvore que está sendo construída. Depois, identificam um vértice u, o qual $Q[u] = True$, incidente em alguma aresta leve que cruza o corte $(V \backslash Q, Q)$, exceto na primeira iteração. Ao remover u de Q, o mesmo é acrescido ao conjunto $V \backslash Q$ na árvore, adicionando a aresta $\{u, A[u]\}$ na solução.

Se G é conexo, para montar a árvore geradora mínima a partir do vetor resultante A, deve-se criar o conjunto $\{\{v, A[v]\} : v \in V \backslash \{r\}\}$.

```
1  import math
2
3  #r Ã© um vÃ©rtice arbitrÃirio, raiz da AGM
4  def prim(g, r=1):
5    #inicializa estruturas de dados
6    A = {v:None for v in g.vertices}     #antecessores
7    K = {v:math.inf for v in g.vertices} #chave de prioridade
8
9    #raiz inicia com a chave mÃnima
10   K[r] = 0
11
12   #Quem estÃi fora da AGM?
13   Q = {v:True for v in g.vertices}
```

```
14
15    for i in range(len(Q)):
16      u = min([v for v in Q if Q[v] == True], key=lambda x:K[x])
17      Q[u] = False
18      #passar por cada vizinho
19      for v in g.N(u):
20        if Q[v] == True and g.pesoAresta(u, v) < K[v]:
21          A[v] = u
22          K[v] = g.pesoAresta(u, v)
23
24    #retorna a soluÃ§Ã£o (Ãirvore de antecessores)
25    return A
26
27
28  def main():
29    arquivo = "/content/gdrive/My Drive/instancias/agm_tiny.net"
30
31    g1 = Grafo(arquivo)
32    A = prim(g1)
33
34    print(A)
35    print('Soma: ', sum([g1.pesoAresta(u, A[u]) for u in A if A[u]
        != None]))
36
37  if __name__ == "__main__":
38    main()
```

Complexidade do Algoritmo de Prim

Para implementar o algoritmo de Prim de maneira mais eficiente possível, deve-se encontrar uma estrutura de prioridade que realize as operações de extração do valor mínimo e da alteração das chave de maneira rápida.

Desafio

Qual a complexidade em tempo computacional da versão mais eficiente do Algoritmo de Prim?

CAPÍTULO 8

Programação Dinâmica

Em momentos anteriores nessas notas, problemas foram tratados por estratégias gulosas que operavam eficientemente. Infelizmente, para a maioria dos problemas, a dificuldade não está em determinar qual critério guloso utilizar, pois não há estratégia gulosa que conduza à solução ótima. Para problemas como esses, é importante usar outras estratégias. A Divisão e Conquista pode servir como uma alternativa, mas as versões conhecidas podem não ser suficientemente fortes para evitar uma busca de força-bruta (KLEINBERG; TARDOS, 2005).

Neste capítulo lida-se com a Programação Dinâmica. É fácil dizer o que é programação dinâmica depois de trabalhar com alguns exemplos. A ideia básica vem de uma intuição a partir da divisão e conquista e é essencialmente o oposto de uma estratégia gulosa. A Programação Dinâmica implica explorar o espaço de todas as soluções possíveis, por cuidadosamente, decompor o problema em subproblemas, e então construir soluções corretas para problemas maiores e maiores. Pode-se considerar que a Programação Dinâmica opera perigosamente próxima a algoritmos de força-bruta. Apesar disso, essa estratégia opera sobre um conjunto exponencialmente grande de subproblemas sem a necessidade de examinar todos eles explicitamente (KLEINBERG; TARDOS, 2005). Na estratégia de Programação Dinâmica, soluções ótimas de tamanho menor auxiliam na busca por uma solução ótima maior.

8.1 Organização de Intervalos com Pesos

Anteriormente, fora conhecido um problema de identificar o conjunto maximal de intervalos quando visitou-se o conceito de algoritmos gulosos, agora trabalha-se com uma variação desse problema, chamado de problema de Intervalos com Pesos. Considere um conjunto de intervalos $I = \{I_1, I_2, \ldots, I_n\}$ onde cada intervalo I_i inicia no tempo $s(i) \in \mathbb{Z}^+$, termina no tempo $f(i) \in \mathbb{Z}^+$, e possui o valor $v(i) \in \mathbb{Z}^+$, precisa-se encontrar o conjunto de intervalos que maximiza o valor de $\sum_{i \in S} v(i)$, onde S é um subconjunto de intervalos sem sobreposição de tempo (KLEINBERG; TARDOS, 2005).

Supõe-se que o os intervalos estejam organizados em uma ordem crescente de tempos de fim: $F = (f(I_1) \leq f(I_2) \leq \ldots \leq f(I_n))$. Agora supõe-se a existência de uma função $p : I \rightarrow I$. $p(I_i)$ determina o intervalo I_j que é o primeiro intervalo compatível antes de I_i na lista F. Considere a solução ótima O. Se $I_n \in O$, não há intervalos compatíveis entre $p(I_n)$ e I_n. Ainda considerando que I_n pertence a solução ótima, considera-se como potenciais candidatos a solução ótima $\{I_1, I_2, \ldots, p(I_n)\}$. Caso $I_n \notin O$, deve-se considerar como potenciais candidatos a solução ótima $\{I_1, I_2, \ldots, I_{n-1}\}$.

Pensando na recorrência para a programação dinâmica, considera-se $OPT(I_i)$ como o valor da solução ótima para os intervalos $\{I_1, I_2, \ldots, I_i\}$. Por questões de convenção, assume-se que $OPT(I_0) = 0$. A recorrência representando a solução ótima considerando o problema é dado por

$$OPT(I_i) = \begin{cases} 0 & \text{para } i = 0, \\ \textbf{max}\Big\{v(I_i) + OPT\big(p(I_i)\big), OPT(I_{i-1})\Big\} & \text{para } i > 0. \end{cases} \tag{8.1}$$

Então, se está assumindo que I_i pertence a solução ótima sse $v(I_i) + OPT\big(p(I_i)\big) > OPT(I_{i-1})$.

Um algoritmo recursivo OPTAIP para resolver a recorrência pode ser visualizado abaixo.

```
1 def optAIP(I, i, v, p):
```

```
2   if i == -1:
3     return 0
4   else:
5     return max(v[I[i]]+optAIP(I, p[i], v, p), optAIP(I, i-1, v,
        p))
6
7 def main():
8   I = ["a"   , "b"  ,  "c"   , "d",     "e"] #identificaÃ§Ã£o
      dos intervalos
9   s = {"a": 1, "b": 6, "c":12, "d":  4, "e": 9} #tempo de
      inÃcio de cada intervalo
10  f = {"a": 5, "b":11, "c":16, "d":  7, "e":13} #tempo de fim de
      cada intervalo
11  v = {"a":25, "b":10, "c":45, "d": 50, "e":20} #valor do
      intervalo
12
13  #ordenando I na ordem de fim
14  Io = sorted(I, key=lambda x:f[x])
15
16  #criando o p
17  p = {0:-1 } #como indexamos de 0 atÃ© n-1, o caso base Ã© -1
18  for i in range(len(Io)-1, -1, -1):
19    j = i-1
20    while j>=0 and s[Io[i]]<f[Io[j]]:
21      j-=1
22    p[i] = j
23
24  print(optAIP(Io, len(Io)-1, v, p))
```

```
25
26 if __name__ == "__main__":
27   main()
```

Infelizmente, ao executar o código acima, percebe-se que a árvore de subproblemas cresce muito rapidamente, o que impacta no tempo de execução (não-polinomial). A árvore cresce rapidamente porque há recálculo de subproblemas. A próxima tentativa irá utilizar uma memória para evitar tais recálculos (KLEINBERG; TARDOS, 2005). Tal método é conhecido como Memoização (CORMEN et al., 2012).

No novo algoritmo, a memoização é implementada através de um vetor M indexado de 0 a n. Assuma que $M_i = -1$ para todo $i \in \{1,2,\ldots,n\}$. O código-fonte abaixo define a ideia.

```
1  def optAIP_M(I, i, v, p, M):
2    if i == -1:
3      return 0
4    else:
5      if M[i] is None:
6        M[i] = max(v[I[i]]+optAIP_M(I, p[i], v, p, M), optAIP_M(I,
             i-1, v, p, M))
7      return M[i]
8
9  def main():
10   I = ["a"   , "b"  ,  "c"   , "d",     "e"] #identificaÃ§Ã£o
       dos intervalos
11   s = {"a": 1, "b": 6, "c":12, "d":  4, "e": 9} #tempo de
       inÃcio de cada intervalo
12   f = {"a": 5, "b":11, "c":16, "d":  7, "e":13} #tempo de fim de
       cada intervalo
13   v = {"a":25, "b":10, "c":45, "d": 50, "e":20} #valor do
```

```
    intervalo
14
15  #ordenando I na ordem de fim
16  Io = sorted(I, key=lambda x:f[x])
17
18  #criando o p
19  p = {0:-1 } #como indexamos de 0 atÃ© n-1, o caso base Ã© -1
20  for i in range(len(Io)-1, -1, -1):
21    j = i-1
22    while j>=0 and s[Io[i]]<f[Io[j]]:
23      j-=1
24    p[i] = j
25
26  M = [None]*len(Io)
27  print(optAIP_M(Io, len(Io)-1, v, p, M))
28
29 if __name__ == "__main__":
30  main()
```

A versão iterativa pode ser visualizada abaixo:

```
1  def optAIP_it(I, v, p, M):
2    for i in range(len(I)):
3      M[i+1] = max(v[I[i]]+M[p[i]+1], M[i])
4    return M[len(I)]
5
6  def main():
7    I = ["a"   , "b"  ,  "c"   , "d",     "e"] #identificaÃ§Ã£o
       dos intervalos
8    s = {"a": 1, "b": 6, "c":12, "d":  4, "e": 9} #tempo de
```

```
    inÃcio de cada intervalo
9   f = {"a": 5, "b":11, "c":16, "d":  7, "e":13} #tempo de fim de
    cada intervalo
10  v = {"a":25, "b":10, "c":45, "d": 50, "e":20} #valor do
    intervalo
11
12  #ordenando I na ordem de fim
13  Io = sorted(I, key=lambda x:f[x])
14
15  #criando o p
16  p = {0:-1 } #como indexamos de 0 atÃ© n-1, o caso base Ã© -1
17  for i in range(len(Io)-1, -1, -1):
18    j = i-1
19    while j>=0 and s[Io[i]]<f[Io[j]]:
20      j-=1
21    p[i] = j
22
23  M = [None]*(len(Io)+1) #ajustado para caso base ser gravado na
    posiÃ§Ã£o 0
24  M[0] = 0
25  print(optAIP_it(Io, v, p, M))
26
27 if __name__ == "__main__":
28  main()
```

8.1.1 Complexidade

A complexidade de tempo da versão recursiva (com memoização) e iterativa é $O(n)$.

8.2 Subsequência Comum Mais Longa

Um filamento de DNA consiste em uma cadeia de moléculas denominadas bases. Dentre essas bases têm-se Adenina (A), Guanina (G), Citosina (C) e Timina (T). Dois indivíduos podem ser considerados semelhantes se suas cadeias de DNA também o são. Nesse contexto, emerge a necessidade de comparar duas cadeias que compartilhem de uma terceira cadeia formada em uma mesma ordem, mas não necessariamente na mesma sequência. Chama-se esse problema de Subsequência Comum Mais Longa (SCML, ou *Longest Common Subsequence*) (CORMEN et al., 2012).

Considere que as cadeias $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, procura-se encontrar a subsequência mais longa $Z = \langle z_1, z_2, \ldots, z_k \rangle$. Para compreender as propriedades o teorema a seguir, assuma que dada uma cadeia W, $W_i = \langle w_1, w_2, \ldots, w_i \rangle$ é o prefixo de W até o i-ésimo símbolo.

A seguir, há um importante teorema quanto a esse problema que define a subestrutura ótima

Teorema 8.2.1. *Sejam $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as sequências, e $Z = \langle z_1, z_2, \ldots, z_k \rangle$ uma subsequência mais longa de X e Y, têm-se:*

1. *Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} uma subsequência mais longa de X_{m-1} e Y_{n-1};*
2. *Se $x_m \neq y_n$, então $z_k = x_m$ implica que Z é uma subcadeia mais longa de X_{m-1} e Y;*
3. *Se $x_m \neq y_n$, então $z_k = y_n$ implica que Z é uma subcadeia mais longa de X e Y_{n-1}.*

Prova: Para o item (1), se $z_k \neq x_m$, então podemos anexar a $x_m = y_n$ a Z para obter a subsequência comum mais longa de comprimento $k+1$, contradizendo de que Z é uma subsequência comum mais longa. Desse modo, deve-se ter $z_k = x_m = y_n$. O prefixo Z_{k-1} é uma subsequência comum de comprimento $k-1$ de X_{m-1} e Y_{n-1}. Deseja-se mostrar que ela é uma subsequência comum mais longa. Supõe-se por contradição, que há uma sequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que $k-1$. Então, anexar

$x_m = y_n$ a W produz uma subsequência máxima de X e Y cujo o comprimento é maior que k, o que é uma contradição.

Para o item (2), se $z_k \neq x_m$, então Z é a subsequência comum de X_{m-1} e Y. Se existisse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W seria também uma subsequência comum de X_m e Y, contradizendo a suposição de que Z é uma subsequência comum mais longa.

Para o item (3), a prova é simétrica à explicação ao item (2). ■

Segue a recorrência da subsequência mais longa, considerando as cadeias $X = \langle x_1, x_2, \ldots \rangle$ e $Y = \langle y_1, y_2, \ldots \rangle$:

$$OPT(m,n) = \begin{cases} 0 & \text{para } m = 0 \text{ ou } n = 0, \\ OPT(m-1, n-1) + 1, & \text{quando } x_m = y_n, \\ \textbf{max}\{OPT(m-1,n), OPT(m,n-1)\}, & \text{quando } x_m \neq y_n. \end{cases} \tag{8.2}$$

Segue o código-fonte, implementando a recorrência subsequência comum mais longa.

```
1  def SCML_opt(X, Y, m, n):
2    if m == -1 or n == -1:
3      return 0
4    elif X[m] == Y[n]:
5      return SCML_opt(X, Y, m-1, n-1) + 1
6    else:
7      return max(SCML_opt(X, Y, m-1, n),SCML_opt(X, Y, m, n-1))
8
9  def main():
10   X = ["CafÃ© RU", "Praia Mole", "Restaurante RU", "UFSC",
       "PraÃ§a Trindade", "Pizzaria RU"]
```

```
11   Y = ["Lanchonete RU", "UFSC", "Restaurante RU", "UFSC", "Praia
        Mole", "PraÃ§a Trindade", "Lancheria RU"]
12
13   print(SCML_opt(X, Y, len(X)-1, len(Y)-1))
14
15 if __name__ == "__main__":
16   main()
```

Abaixo, há o uso a memoização para economizar tempo (evitar recálculo de subproblemas).

```
1  def SCML_M(X, Y, m, n, M):
2    if m == -1 or n == -1:
3      return 0
4    if M[m][n] != None:
5      return M[m][n]
6    if X[m] == Y[n]:
7      M[m][n] = SCML_M(X, Y, m-1, n-1, M) + 1
8    else:
9      M[m][n] = max(SCML_M(X, Y, m-1, n, M), SCML_M(X, Y, m, n-1,
           M))
10   return M[m][n]
11
12 def main():
13   X = ["CafÃ© RU", "Praia Mole", "Restaurante RU", "UFSC",
        "PraÃ§a Trindade", "Pizzaria RU"]
14   Y = ["Lanchonete RU", "UFSC", "Restaurante RU", "UFSC", "Praia
        Mole", "PraÃ§a Trindade", "Lancheria RU"]
15
16   #preparando matriz de memoria
```

```
17    M = [None]*len(X)
18    for i in range(len(M)):
19      M[i] = [None]*len(Y)
20
21    #executando
22    print(SCML_M(X, Y, len(X)-1, len(Y)-1, M))
23
24  if __name__ == "__main__":
25    main()
```

Finalmente, uma versão do algoritmo memoizado iterativo.

8.3 O Problema da Mochila

Antes de começar a analisar o problema da mochila, apresenta-se um problema mais simples, que chamaremos aqui de "Subconjunto de Somas".

O Subconjunto de Somas é um problema no qual, dado um conjunto dos items $I = \{1, 2, \ldots, n\}$, uma função de peso $w : I \to \mathbb{Z}^+$ e um limite de peso $W \in \mathbb{Z}^+$, deseja-se encontrar um subconjunto de $S \subseteq I$ com a maior soma $\sum_{s \in S} w(s)$ tal que $\sum_{s \in S} w(s) \leq W$ (KLEINBERG; TARDOS, 2005).

Pode-se tentar resolver por programação dinâmica utilizando o $OPT(i)$ para denotar o valor da solção ótima para a subsolução $\{1, 2, \ldots, n\}$. Desse modo, considerando O a solução ótima, se $i \notin O$, então $OPT(i) = OPT(i-1)$, senão $OPT(i) = w(i) + OPT(i-1)$. O problema nessa tentativa está em que ao aceitar i não implica a rejeição de qualquer outro item. Isso indica a necessidade de trabalhar com mais subproblemas.

A recorrência que funciona para o problema de Subconjunto de Somas utiliza a ideia de considerar o limite de peso W, no qual, se um item é adicionado a subsolução, ele diminui seu peso ao valor de W. Logo, utiliza-se a seguinte recorrência

$$OPT(i,p) = \begin{cases} 0 & \text{para } i = 0 \text{ ou } p = 0, \\ OPT(i-1,p) & \text{para } p < w(i), \\ \textbf{max}\{OPT(i-1,p), w(i) + OPT(i-1,p-w(i))\} & \text{para } p \geq w(i). \end{cases} \tag{8.3}$$

, na qual w é o valor corrente de peso restante a ser considerado para os subproblemas.

O Problema da Mochila (ou *Knapsack Problem*) é muito semelhante ao de Subconjunto de Somas. Nele, além do conjunto de itens I, da função de peso w para cada item e do limite de peso W, considera-se que cada item tem um valor associado, dado pela função $v : I \to \mathbb{R}$. Deve-se buscar o subconjunto $S \subseteq I$ no qual $\sum_{s \in S} v(s)$ tal que $\sum_{s \in S} w(s) \leq W$. Podemos então utilizar uma função de recorrência muito semelhante à proposta para o problema do Subconjunto de Somas. Segue a recorrência adaptada

$$OPT(i,p) = \begin{cases} 0 & \text{para } i = 0 \text{ ou } p = 0, \\ OPT(i-1,p), & \text{para } p < w(i) \\ \textbf{max}\{OPT(i-1,p), v(i) + OPT(i-1,p-w(i))\}, & \text{para } p \geq w(i). \end{cases} \tag{8.4}$$

O código-fonte apresenta um exemplo de implementação sem a memoização.

```
1 def mochila_opt(I, v, w, i, p):
2   if i == -1 or p == 0:
3     return 0
4   else:
5     if p < w[I[i]]:
6       return mochila_opt(I, v, w, i-1, p)
7     else:
8       return max(mochila_opt(I, v, w, i-1, p),
           v[I[i]]+mochila_opt(I, v, w, i-1, p-w[I[i]]))
```

```
9
10 def main():
11   I = ["a"    ,"b"   ,"c"     ,"d"   ,"e"]
12   w = {"a": 10,"b":  5,"c":  8,"d":  4,"e":  9}
13   v = {"a": 90,"b": 65,"c": 70,"d": 30,"e":100}
14   W = 20
15
16   print(mochila_opt(I, v, w, len(I)-1, W))
17
18 if __name__ == "__main__":
19   main()
```

Finalmente, abaixo pode-se visualizar o código-fonte com implementação.

```
1  def mochila_M(I, v, w, i, p, M):
2      if M[i+1][p] is not None:
3        return M[i+1][p]
4      else:
5        if p < w[I[i]]:
6          M[i+1][p] = mochila_M(I, v, w, i-1, p, M)
7        else:
8          M[i+1][p] = max(mochila_M(I, v, w, i-1, p, M),
               v[I[i]]+mochila_M(I, v, w, i-1, p-w[I[i]], M))
9        return M[i+1][p]
10
11
12 def main():
13   I = ["a"    ,"b"   ,"c"     ,"d"   ,"e"]
14   w = {"a": 10,"b":  5,"c":  8,"d":  4,"e":  9}
15   v = {"a": 90,"b": 65,"c": 70,"d": 30,"e":100}
```

```
16    W = 20
17
18    #preparando memoria
19    M = [None]*(len(I)+1)
20    for i in range(len(M)):
21      M[i] = [None]*(W+1)
22
23    #preparando subproblemas base
24    for i in range(len(I)+1):
25      M[i][0] = 0 #nenhum peso
26    for j in range(W+1):
27      M[0][j] = 0 #nenhum item
28
29    #executando
30    print(mochila_M(I, v, w, len(I)-1, W, M))
31
32  if __name__ == "__main__":
33    main()
```

Pense em como implementar as seguintes versões:

- descobrir os itens que estão na mochila;
- realizar a descoberta de itens a partir da memória M;
- forma iterativa de resolver os subproblemas.

Existem variações do problema da mochila, como por exemplo:

- com cópias ilimitadas de cada item;
- com cópias limitadas de cada items;
- com múltiplas mochilas.

8.3.1 Complexidade

A complexidade do algoritmo de programação dinâmica proposto é de $\Theta(|I|W)$ ou seja, não pode ser considerado resolvível em tempo polinomial.

CAPÍTULO 9

Fluxo Máximo

9.1 Redes de Fluxo

Uma rede de fluxo, no contexto da Teoria dos Grafos, é um grafo dirigido ponderado $G = (V, A, c)$, na qual V representa o conjunto de vértices, A o conjunto de arcos e $c : V \times V \to \mathbb{R}^+$ é a função de capacidade de cada arco. Impõe-se ainda que se há um arco (u, v) não há um arco (v, u). Se $(u, v) \notin A$, então $c\big((u, v)\big) = 0$. Há dois vértices distintos: o vértice de origem ou fonte, geralmente chamado de s, e o vértice de destino ou sorvedouro, chamado de t. or conveniência, assume-se que todo o vértices $v \in V \setminus \{s, t\}$, $s \rightsquigarrow v \rightsquigarrow t$ (CORMEN et al., 2012).

Um fluxo em G é uma função $f : V \times V \to \mathbb{R}$ que satisfaz as seguintes propriedades:

- Restrição de capacidade: para todo $u, v \in V$, $0 \leq f\big((u, v)\big) \leq c\big((u, v)\big)$;

- Conservação de fluxo: para todo $u \in V \setminus \{s, t\}$,

$$\sum_{v \in V} f\big((v, u)\big) = \sum_{v \in V} f\big((u, v)\big).$$

Quando $(u, v) \notin A$, então $f\big((v, u)\big) = 0$.

A quantidade não negativa $f\big((v, u)\big)$ é denominada o fluxo do vértice u a v. O valor

de fluxo é dado por $F = \sum_{v \in V} f\big((s,v)\big) - \sum_{v \in V} f\big((v,s)\big)$, ou seja, o total de fluxo que sai de s menos o total de fluxo que entra em s.

Quando $(u,v),(v,u) \in A$, deve-se remover (u,v) de A, adicionar um novo vértice v' a V, adicionar os arcos (u,v') e (v',v) a A definido as capacidades $c\big((u,v')\big) \rightarrow c\big((u,v)\big)$ e $c\big((v',v)\big) \rightarrow c\big((u,v)\big)$ e $c\big((u,v)\big) \rightarrow 0$.

Para múltiplas origens $s_1, s_2, \ldots, s_k$ pode-se adicionar um vértice inicial s' a G e os arcos (s', s_i) com capacidade $c\big((s',s_i)\big) \rightarrow \infty$ para todo $i \in \{1,2,\ldots,k\}$.

Para múltiplos destinos $t_1, t_2, \ldots, t_l$ pode-se adicionar um vértice final t' a G e os arcos (t_j, t') com capacidade $c\big((t_j,t')\big) \rightarrow \infty$ para todo $j \in \{1,2,\ldots,l\}$.

O problema de fluxo máximo busca encontrar o maior fluxo em G de s a t tal que as capacidades de cada arco sejam respeitadas.

9.2 Rede Residual

Uma rede residual consiste em um grafo $G_f = (V, A_f, c_f)$ composto por arcos com capacidades $c_f\big((u,v)\big) = c\big((u,v)\big) - f\big((u,v)\big)$ para todo arco A. Para cada arco (u,v) em A, têm-se um arco invertido em A_f com capacidade $c_f\big((v,u)\big) = f\big((u,v)\big)$. Se um arco $(u,v) \notin A$, então $c_f\big((v,u)\big) = 0$. Desse modo, $|A_f| = 2|A|$.

9.3 Caminhos Aumentantes

Em uma rede de fluxo G, um caminho aumentante p é um caminho simple de s a t na rede residal G_f. Ele aumenta o fluxo na rede sem ferir as restrições de capacidades impostas em c. A capacidade residual de p é $c_f(p) = \mathbf{min}\{c_f\big((u,v)\big) : (u,v) \in p\}$.

9.4 Ford-Fulkerson

O algoritmo de Ford-Fulkerson se baseia em três ideias principais: redes residuais, caminhos aumentantes e cortes. Começa-se com $f\big((u,v)\big) = 0$ para todo $u, v \in V$. O

método de Ford-Fulkerson aumenta iterativamente o valor de fluxo, identificando os arcos que podem alterar seu fluxo, consultando suas capacidades. O fluxo aumenta até que não haja mais caminhos aumentantes.

O algoritmo é demonstrado no código-fonte abaixo.

```
1  def FordFulkerson(g, s, t):
2    #criar grafo residual
3    gl = criaGrafoResidual(g)
4
5    #fluxo maximo
6    F = 0
7
8    #enquanto houver caminho aumentante
9    (p, cap) = buscaCaminhoAumentante(gl, s, t)
10   while cap > 0:
11     #atualiza fluxo
12     F += cap
13     #print (cap, p) #se quiser imprimir os caminhos aumentades e
           a capacidade, descomente
14     for i in range(len(p)-1):
15       u, v = p[i], p[i+1]
16       gl.adjsSaintes[u][v]   -= cap #reduz a capacidade no arco
17       gl.adjsEntrantes[v][u] += cap #aumenta a capacidade do
             arco de retorno
18
19     #proximo caminho aumentante (se houver)
20     (p, cap) = buscaCaminhoAumentante(gl, s, t)
21   return (F, gl)
```

9.4.1 Complexidade

A complexidade de tempo computacional de $Ford - Fulkerson$ depende do fluxo máximo. A Figura 9.4.1 demonstra um exemplo de um pior caso. O tempo para encontrar um caminho de s para t é de $O(|V| + |A|) = O(|A|)$. Se as capacidades dos arcos da rede são dadas por números inteiros e como um caminho aumenta o fluxo sempre em relação ao anterior, a quantidade de caminhos aumentantes é de f^*, ou seja, o valor de fluxo máximo. Então a complexidade de tempo do algoritmo é $O(|A|f^*)$.

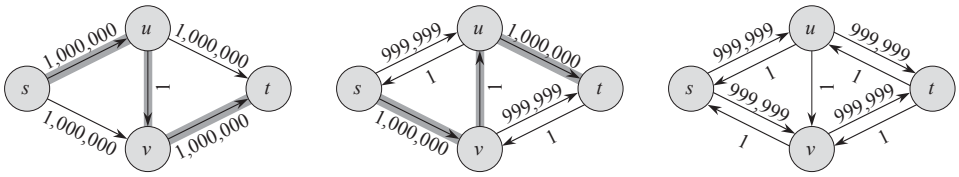


Figura 12 – Exemplo citado por Cormen et al. (2012) para demonstrar o pior caso quanto a complexidade de tempo.

9.5 Edmonds-Karp

Dado a complexidade do algoritmo de Ford-Fulkerson, é impraticável executá-lo para instâncias muito grandes que caem no caso de caminhos aumentantes pequenos. O algoritmo Edmonds-Karp propõe uma pequena adaptação para superar esse problema. No lugar de encontrar um caminho aumentante arbitrário, Edmonds-Karp usa uma busca em largura descrita no código-fonte abaixo.

```
1 #busca por caminho aumentante: busca em largura
2 def buscaCaminhoAumentante(g, s, t):
3   #preparando estruturas de dados
4   visitados    = {id: False    for id in g.vertices}
5   distancia    = {id: math.inf for id in g.vertices}
6   antecessor   = {id: None     for id in g.vertices}
7
```

```
8   #configurando vÃ©rtice de origem
9   visitados[s] = True
10  distancia[s] = 0
11
12  #preparando fila de visitas
13  fila = []
14  fila.append(s)
15
16  #propagaÃ§Ã£o de visitas
17  while len(fila) > 0: #enquanto houver elementos na fila
18    u = fila.pop(0) #retira elementos por ordem de entrada na
          estrutura
19    for v in g.adjsSaintes[u]:
20
21      #se o vertice ainda nao foi visitado e se ha capacidade
            residual
22      if visitados[v] == False and g.adjsSaintes[u][v] > 0:
23        visitados[v] = True
24        distancia[v] = distancia[u] + 1
25        antecessor[v] = u
26        fila.append(v)
27        #se chegou ao destino
28        if v == t:
29          #monta caminho
30          p = [t]
31          z = t
32          cap = math.inf
33          while z != s:
```

```
34              cap = min(cap, g.pesoAresta(antecessor[z], z))
                   #capacidade do caminho
35              z = antecessor[z]
36              p.insert(0, z)
37            return p, cap
38    return [], 0
```

Uma versão completa do algoritmo de Edmonds-Karp com todas as etapas pode ser visualizada no código-fonte abaixo.

```
1  import copy, math
2
3  #adiciona arcos de retorno
4  def criaGrafoResidual(g):
5    #copia grafo
6    gl = GrafoDirigido()
7    gl.vertices = g.vertices.copy()
8    gl.adjsSaintes = copy.deepcopy(g.adjsSaintes)
9    gl.adjsEntrantes = copy.deepcopy(g.adjsEntrantes)
10
11   #checando se ha arestas paralelas
12   remover = []
13   for i in range(len(g.arcos)):
14     [u,v] = g.arcos[i]
15     if gl.pesoAresta(v, u) is not None:
16       #criar arestas antiparalelas
17       w = gl.pesoAresta(u, v)
18       del gl.adjsSaintes[u][v]
19       del gl.adjsEntrantes[v][u]
20       x = len(gl.vertices)+1
```

```
21        gl.vertices[x] = x-1
22        gl.adjsSaintes[u][x] = w
23        gl.adjsSaintes[x] = {}
24        gl.adjsSaintes[x][v] = w
25        gl.adjsEntrantes[x]={}
26        gl.adjsEntrantes[x][u] = w
27        gl.adjsEntrantes[v][x] = w
28
29    #define arcos de retorno
30    for [u,v] in g.arcos:
31      gl.arcos.append([v,u])
32      gl.adjsSaintes[v][u] = 0
33      gl.adjsEntrantes[u][v] = 0
34    return gl
35
36  #busca por caminho aumentante: busca em largura
37  def buscaCaminhoAumentante(g, s, t):
38    #preparando estruturas de dados
39    visitados    = {id: False    for id in g.vertices}
40    distancia    = {id: math.inf for id in g.vertices}
41    antecessor   = {id: None     for id in g.vertices}
42
43    #configurando vÃ©rtice de origem
44    visitados[s] = True
45    distancia[s] = 0
46
47    #preparando fila de visitas
48    fila = []
```

```
49  fila.append(s)
50
51  #propagaÃ§Ã£o de visitas
52  while len(fila) > 0: #enquanto houver elementos na fila
53    u = fila.pop(0) #retira elementos por ordem de entrada na
        estrutura
54    for v in g.adjsSaintes[u]:
55
56      #se o vertice ainda nao foi visitado e se ha capacidade
          residual
57      if visitados[v] == False and g.adjsSaintes[u][v] > 0:
58        visitados[v] = True
59        distancia[v] = distancia[u] + 1
60        antecessor[v] = u
61        fila.append(v)
62        #se chegou ao destino
63        if v == t:
64          #monta caminho
65          p = [t]
66          z = t
67          cap = math.inf
68          while z != s:
69            cap = min(cap, g.pesoAresta(antecessor[z], z))
                #capacidade do caminho
70            z = antecessor[z]
71            p.insert(0, z)
72          return p, cap
73  return [], 0
```

```
74
75 #Edmonds-Karp: Ford-Fulkerson com caminho de Edmonds-Karp
76 def EdmondsKarp(g, s, t):
77   #criar grafo residual
78   gl = criaGrafoResidual(g)
79
80   #fluxo maximo
81   F = 0
82
83   #enquanto houver caminho aumentante
84   (p, cap) = buscaCaminhoAumentante(gl, s, t)
85   while cap > 0:
86     #atualiza fluxo
87     F += cap
88     #print (cap, p) #se quiser imprimir os caminhos aumentades e
       a capacidade, descomente
89     for i in range(len(p)-1):
90       u, v = p[i], p[i+1]
91       gl.adjsSaintes[u][v]   -= cap #reduz a capacidade no arco
92       gl.adjsEntrantes[v][u] += cap #aumenta a capacidade do
           arco de retorno
93
94     #proximo caminho aumentante (se houver)
95     (p, cap) = buscaCaminhoAumentante(gl, s, t)
96
97   return (F, gl)
98
99
```

```
100 def main():
101   arquivo = "/content/gdrive/My Drive/instancias/wiki.net"
102
103   g1 = GrafoDirigido(arquivo)
104   s = 1
105   t = len(g1.vertices)
106   (F, gl) = EdmondsKarp(g1, s, t)
107   print(F)
108
109   #exibindo grafo
110   g = nx.DiGraph([ (u, v, {"weight": g1.pesoAresta(u,v)} ) for
          u, v in g1.arcos])
111   pos=nx.spring_layout(g, k = 1)
112   labels = nx.get_edge_attributes(g,'weight')
113   nx.draw_networkx_edge_labels(g,pos,edge_labels=labels)
114   nx.draw_networkx(g,  pos, with_labels=True)
115
116 if __name__ == "__main__":
117   main()
```

9.5.1 Complexidade

Como cada caminho aumentante pode ter um arco crítico[1], o número total de arcos críticos é $O(|V| \cdot |A|)$. Como cada iteração do Ford-Fulkerson pode demandar passar por no máximo $O(|A|)$ arcos, a complexidade do algoritmo Edmonds-Karp é de $O(|V||A|^2)$.

[1] Considera-se arco crítico todo arco que define um fluxo para um caminho aumentante (que define o valor de $c_f(p)$ em um caminho p).

CAPÍTULO 10

Caminhos Disjuntos em Grafos

CAPÍTULO 11

Programação Linear

11.1 Modelagem

Modelos/Problemas de programação linear inteira mista

- Variáveis (contínuas, discretas - binárias);
- Restrições (=, ≤, ≥);
- Função Objetivo (maximização, minimização).

Um modelo de Programação Linear Inteira Mista é composto de:

- Variáveis de decisão;
- Parâmetros.

Com isso, pode ser escrito da seguinte forma.

Função objetivo:

$$\min \sum_{i=1}^{n} c_i x_i + \sum_{j=1}^{p} h_j y_j$$

sujeito às restrições:

- $\sum_{i=1}^{n} a_{\ell i} x_i + \sum_{j=1}^{p} g_{\ell j} y_j \leq b_i \quad \ell \in \{1, \ldots, m\}$,
- $x_i \geq 0 \quad i \in \{1, \ldots, n\}$,

- $y_j \in \mathbb{Z}^+ \quad j \in \{1,\ldots,p\}$.

Em que x_i são as variáveis contínuas, y_j são variáveis inteiras, c_i, h_j, $a_{\ell i}$ e $g_{\ell j}$ são parâmetros racionais conhecidos a priori.

Uma solução ótima para uma instância do modelo é uma solução viável (que obedece todas as restrições) e, ao mesmo tempo, minimiza o valor da função objetivo.

Nos casos em que $h_j = 0, \forall\, j \in \{1,\ldots,n\}$ e $g_{\ell j} = 0, \forall\, \ell \in \{1,\ldots,m\}, \forall\, j \in \{1,\ldots,p\}$, tem-se um modelo de Programação Linear. Caso contrário, tem-se um modelo de Programação Linear Inteira Mista.

Para a solução instâncias de modelos de Programação Linear (sem variáveis inteiras), pode-se utilizar o Algoritmo Simplex ou Algoritmos de Pontos Interiores. Contudo, caso haja variáveis inteiras, são utilizados algoritmos de branch-and-cut. Independente da presença de variáveis inteiras, há pacotes computacionais para a solução de tais problemas.

11.2 Ferramentas

Resolvedores (*solvers*):

- SCIP(<https://scipopt.org>);
- COIN-OR CBC (<https://www.coin-or.org>);
- Gurobi (<www.gurobi.com>);
- IBM ILOG CPLEX (<https://www.ibm.com/analytics/cplex-optimizer>);
- entre outros (<https://en.wikipedia.org/wiki/List_of_optimization_software>).

Tipicamente, um resolvedor fornece ao menos uma API para uma linguagem de programação com funcionalidades que permitem a definição e resolução de instâncias dos modelos de otimização.

Alternativamente, podem ser utilizadas ferramentas para modelagem.

Ferramentas para modelagem

- AMPL (<https://ampl.com>)
- PuLP (<https://coin-or.github.io/pulp/>)
- JuMP (<https://jump.dev>)
- Python-MIP (<https://www.python-mip.com>)

Essas ferramentas também permitem a definição de modelos de otimização, mas a resolução é delegada a um resolvedor. Diversas linguagens de modelagem (como Python-MIP) utilizam algum resolver padrão (como o CBC) mas permitem a utilização de outros (Gurobi, CPLEX, SCIP etc). Isto é, a mesma interface de programação fornece acesso a diversos resolvedores. O que, em geral, não é possível quando se utiliza diretamente a API fornecida pelo resolver.

11.3 Exemplos

11.3.1 Problema do político

Um político gostaria de se eleger em sua cidade. Para isso, ele precisará investir em propaganda. É claro que ele quer investir o mínimo possível!

A equipe de marketing do político propôs classificar os eleitores da cidade de acordo com a localização geográfica da residência de cada um. Então, a classificação dos eleitores ficou assim, moradores da região:

- centro e urbana (RC);
- não-centro e urbana (RNC);
- rural (RR).

A região centro e urbana possui 150.000 eleitores. A região não-centro e urbana possui 350.000. E a região rural possui 70.000 eleitores.

A mesma equipe de marketing sabe que aumentam as chances do político se eleger, se ele investir em propaganda nos seguintes assuntos:

- asfaltar ruas ($P0$);
- aumentar a segurança ($P1$);
- aumentar os subsídios agrícolas ($P2$);
- aumentar o salário mínimo ($P3$);
- diminuir impostos sobre a gasolina ($P4$).

A equipe de marketing diz que a cada R$ 1.000,00 gastos em:

- asfaltar ruas: o político perde 2.000 eleitores que moram na RC, mas ganha 5.000 eleitores que moram na RNC e ganha 3.000 eleitores que moram na RR;
- aumentar a segurança: o político ganha 5.000 eleitores que moram na RC, ganha 6.000 eleitores que moram na RNC, e 1.000 eleitores que moram na RR;
- aumentar os subsídios agrícolas: o político não perde e nem ganha eleitores que moram na RC e RNC, mas ganha 3.000 eleitores que moram na RR;
- aumentar o salário mínimo: o político ganha 2.000 eleitores que moram na RC, ganha 4.500 eleitores que moram na RNC e ganha 500 eleitores que moram na RR;
- diminuir impostos sobre a gasolina: o político ganha 7.500 eleitores que moram na RC, ganha 2.500 eleitores que moram na RNC, e ganha 5.000 eleitores que moram na RR.

Qual o valor mínimo em propaganda que o político deve gastar de tal forma que ele ganhe pelo menos a metade dos eleitores moradores de cada região?

Quais são as variáveis do problema? Quantas variáveis temos?

Note que o político quer investir em propaganda (asfaltar ruas (P0), aumentar a segurança (P1), ...), mas não sabe quanto! Então, vamos chamar de x_{Pi} a quantidade em mil reais a serem investidos na propaganda Pi, sendo i igual a 0, 1, 2, 3 e 4. Dessa forma, temos cinco variáveis.

Qual é o objetivo?

O político (claro!) quer investir o mínimo possível! Então, $\min \sum_{i=0}^{4} x_{Pi}$.

Quais são as restrições?

- Atender metade dos eleitores que moram na RC:

  $-2.000x_{P0} + 5.000x_{P1} + 0x_{P2} + 2.000x_{P3} + 7.500x_{P4} \geq 75.000$

- Atender metade dos eleitores que moram na RNC:

  $5.000x_{P0} + 6.000x_{P1} + 0x_{P2} + 4.500x_{P3} + 2.500x_{P4} \geq 175.000$

- Atender metade dos eleitores que moram na RR:

  $3.000x_{P0} + 1.000x_{P1} + 3000x_{P2} + 500x_{P3} + 5.000x_{P4} \geq 35.000$

Vamos implementar!

```
1 from mip import Model, xsum, minimize, CONTINUOUS
2
3 m = Model("PolÃtico")
4
5 x = [m.add_var(var_type = CONTINUOUS) for i in range(5)]
6
7 m.objective = minimize(x[0] + x[1] + x[2] + x[3] + x[4])
8
9 m += -2000 * x[0] + 5000 * x[1] + 2000 * x[3] + 7500 * x[4] >=
    75000
```

```
10 m +=  5000 * x[0] + 6000 * x[1] + 4500 * x[3] + 2500 * x[4] >=
      175000
11 m +=  3000 * x[0] + 1000 * x[1] + 3000 * x[2] + 500 * x[3] +
      5000 * x[4] >= 35000
12
13 m.optimize()
14
15 for i in range(5):
16   print("P%d => R$ %.2f" %(i, 1000 * x[i].x))
17
18 print()
19 print("Valor que o polÃtico deverÃi investir em propagandas
      para se eleger (segundo equipe de marketing): R$ %.2f"
      %(m.objective_value * 1000))
```

CAPÍTULO 12

Tabelas de Espalhamento

12.1 Introdução

Imagine que você trabalha como caixa em uma feira. Quando o cliente compra um produto, você tem que olhar o preço no catálogo. Se o catálogo não está em ordem alfabética, você vai levar um bom tempo para olhar cada uma das linhas do catálogo. Você estaria fazendo uma busca simples. Lembras qual o tempo assintótico desse algoritmo? $O(n)$. Por outro lado, se o catálogo está em ordem alfabética, uma busca binária é possível. E isso levaria apenas $O(log\ n)$.

Há uma grande diferença entre $O(n)$ e $O(log\ n)$, mas nenhum atendente de caixa merece procurar a todo momento pelo preço no catálogo de cada item que um cliente deseja comprar, mesmo que o catálogo esteja ordenado. O que você realmente precisa é de um colega que já tenha memorizado todos os produtos e seus respectivos preços. Assim você não precisa consultar o catálogo: você pergunta, e o colega responde instantaneamente.

O seu colega pode dizer o preço em $O(1)$, para qualquer produto e independente do quão grande o catálogo é. E esse é o tema deste capítulo: mostrar um algorimo ainda mais rápido do que busca binária.

E que tal começarmos com uma atividade mental? Como você implementaria um

católogo de items?

Que estrutura de dados você utilizaria para realizar consultas como "Qual o preço do abacaxi?", e atualizações como "O preço do caju agora está R$ 12,00 o kilo"?

```
# Let's create an 'interface' with Python's built-in hashtable
class HashTable:
  def __init__(self, default_value=None):
    self.slots = dict() # same as "self.slots = {}"
    self.default_value = default_value
  def __getitem__(self, key):
    return self.slots[key] if key in self.slots else
       self.default_value
  def __setitem__(self, key, value):
    self.slots[key] = value
  def __delitem__(self, key):
    del self.slots[key]
  def __contains__(self, key):
    return key in self.slots
  def __str__(self):
    return str(self.slots)
```

```
def prompt(*args, **kwargs):
  if debug:
    print(*args, **kwargs)
debug = True
def test_search(htable, key):
  value = htable[key]
  prompt("SEARCH {}: {}".format(key, value))
  return value
def test_insert(htable, key, value):
```

```
  test_sign = "INSERT '{}', {}".format(key, value)
  fail_msg = "FAIL "+test_sign+": {} should {}"
  prompt(test_sign)
  prompt("|",end=" ")
  before = test_search(htable, key)
  assert before != value, fail_msg.format(before, "be different")
  htable[key] = value
  prompt("|",end=" ")
  after = test_search(htable, key)
  assert after == value, fail_msg.format(after, "be equal")
  prompt("-"*len(test_sign))
def test_delete(htable, key):
  test_sign = "DELETE '{}'".format(key)
  fail_msg = "FAIL "+test_sign+": {} should {}"
  prompt(test_sign)
  prompt("|",end=" ")
  before = test_search(htable, key)
  assert before is not None, fail_msg.format(before, "be not
     None")
  del htable[key]
  prompt("|",end=" ")
  after = test_search(htable, key)
  assert after is None, fail_msg.format(after, "be None")
  prompt("-"*len(test_sign))
```

```
def run_fruits_test(init_table):
  products = init_table()
  test_insert(products, "Abacaxi", 4)
  test_insert(products, "Abacate", 6)
```

```
  test_insert(products, "Banana Maca", 60)
  test_insert(products, "Caju", 120)
  test_insert(products, "Limao", 20)
  prompt(products,"\n"+"-"*18)
  test_search(products, "Limao")
  test_insert(products, "Mamao", 3)
  test_delete(products, "Abacaxi")
  prompt(products,"\n"+"-"*18)
run_fruits_test(lambda: HashTable())
```

```
INSERT 'Abacaxi', 4
| SEARCH Abacaxi: None
| SEARCH Abacaxi: 4
--------------------
INSERT 'Abacate', 6
| SEARCH Abacate: None
| SEARCH Abacate: 6
--------------------
INSERT 'Banana Maca', 60
| SEARCH Banana Maca: None
| SEARCH Banana Maca: 60
------------------------
INSERT 'Caju', 120
| SEARCH Caju: None
| SEARCH Caju: 120
------------------
INSERT 'Limao', 20
| SEARCH Limao: None
| SEARCH Limao: 20
```

```
------------------
{'Abacaxi': 4, 'Abacate': 6, 'Banana Maca': 60, 'Caju': 120,
   'Limao': 20}
------------------
SEARCH LimÃ£o: 20
INSERT 'MamÃ£o', 3
| SEARCH MamÃ£o: None
| SEARCH MamÃ£o: 3
-----------------
DELETE 'Abacaxi'
| SEARCH Abacaxi: 4
| SEARCH Abacaxi: None
----------------
{'Abacate': 6, 'Banana Maca': 60, 'Caju': 120, 'Limao': 20,
   'Mamao': 3}
------------------
```

Esse catálogo é uma aplicação de conjuntos dinâmicos, os quais suportam operações de dicionário: **consulta**, **inserção**, e **deleção**. Muitas aplicações necessitam dessas operações. Por essa razão, Python já vem imbutido com uma implementação de dicionário: *dict*.

Nosso objetivo, por outro lado, é de mostrar como implementar um dicionário; uma estrutura cujas operações básicas levam $O(1)$.

Como? Vamos responder essa pergunta em etapas.

12.2 Tabela de endereçamento direto

Primeiro, vamos simplificar o nosso exemplo. Considere que cada fruta possui uma chave numérica associada: "Abacaxi", "Abacate", ..., "Mamão"mapeiam para número

inteiros 0, 1, ..., 6.

Então, com esta simplificação, você já tem uma idéia de como implementar um dicionário?

```
class DirectAddressTable:
  NULLVALUE = None
  def __init__(self, capacity, default_value=None):
    self.slots = [None for key in range(capacity)]
    self.capacity = capacity
    self.default_value = default_value
  def __getitem__(self, key): # search
    return self.slots[key] if self._is_valid(key) else
       self.default_value
  def __setitem__(self, key, value): # insert
    if self._is_valid(key):
      self.slots[key] = value
    # else: nothing. silent recovery.
  def __delitem__(self, key): # delete
    if self._is_valid(key):
      self.slots[key] = DirectAddressTable.NULLVALUE
    # else: nothing. silent recovery.
  def __contains__(self, key):
    return all([
      self._is_valid(key),
      self.slots[key] is not DirectAddressTable.NULLVALUE,
    ])
  def __str__(self):
    return str(self.slots)
  def _is_valid(self, key):
```

```
    return 0 <= key < self.capacity
```

```
from collections import namedtuple
def run_fruits_test_with_adhoc_keys(init_table):
  Entry = namedtuple("Entry", "name key")
  PINEAPPLE = Entry("Abacaxi", 0)
  AVOCADO = Entry("Abacate", 1)
  ACEROLA = Entry("Acerola", 2)
  BANANA = Entry("Banana Maca", 3)
  CASHEW = Entry("Caju", 4)
  LEMON = Entry("Limao", 5)
  PAPAYA = Entry("Mamao", 6)
  dm_products = init_table()
  test_insert(dm_products, PINEAPPLE.key, 4)
  test_insert(dm_products, AVOCADO.key, 6)
  test_insert(dm_products, ACEROLA.key, 120)
  test_insert(dm_products, BANANA.key, 120)
  test_insert(dm_products, CASHEW.key, 120)
  test_insert(dm_products, LEMON.key, 20)
  print(dm_products)
  test_search(dm_products, LEMON.key)
  test_insert(dm_products, PAPAYA.key, 3)
  test_delete(dm_products, PINEAPPLE.key)
  print(dm_products)
run_fruits_test_with_adhoc_keys(lambda: DirectAddressTable(7))
```

```
INSERT '0', 4
| SEARCH 0: None
| SEARCH 0: 4
```

```
-------------
INSERT '1', 6
| SEARCH 1: None
| SEARCH 1: 6
-------------
INSERT '2', 120
| SEARCH 2: None
| SEARCH 2: 120
---------------
INSERT '3', 120
| SEARCH 3: None
| SEARCH 3: 120
---------------
INSERT '4', 120
| SEARCH 4: None
| SEARCH 4: 120
---------------
INSERT '5', 20
| SEARCH 5: None
| SEARCH 5: 20
--------------
[4, 6, 120, 120, 120, 20, None]
SEARCH 5: 20
INSERT '6', 3
| SEARCH 6: None
| SEARCH 6: 3
-------------
DELETE '0'
```

```
| SEARCH 0: 4
| SEARCH 0: None
----------
[None, 6, 120, 120, 120, 20, 3]
```

Duas estruturas básicas que suportam operações de consulta são **vetores** e **listas**. Consultas em vetores levam $O(1)$. Consultas em listas levam $O(n)$. Podemos utilizar vetores para implementar nosso catálogo, basta atribuir a cada produto uma posição no vetor e armanezar nela o seu respectivo preço. Assim, desde que a posições dos produtos sejam fixas e únicas, podemos consultar o preço de qualquer produto a $O(1)$.

Tabelas de endereçamento direto. Mapear cada chave para uma posição fixa de um vetor é uma técnica simples, e oferece operações de dicionário a $O(1)$. Essa técnica funciona bem quando o universo de chaves, U, é razoávelmente pequeno. Quando uma aplicação precisa de um conjunto dinâmico para o qual cada elemento já possui uma chave (e a mesma é única, ou seja, nenhum par de elementos possui a mesma chave), podemos utilizar um vetor para representar o conjunto dinâmico. Esse vetor também é chamado de **tabela de endereçamento direto** (ou *direct address table*).

E se o universo de chaves for grande? Nesse caso, armazenar uma tabela com um elemento para cada valor de chave do universo ($|U|$ elementos) é imprático, ou impossível, dada a memória disponível em um computador típico. E uma outra técnica é mais apropriada: tabelas de espalhamento (*hash tables*).

E se houver algum par de elementos com a mesma chave? Nesse caso, esses dois elementos mapeiam para a mesma posição na tabela. Chamamos isso de "colisão", e falaremos sobre o assunto daqui a pouco.

Há outro caso em que as tabelas de endereçamento direto são inapropriadas.

A premissa destas tabelas é declarar um vetor grande o suficiente para suportar cada chave possível que a aplicação pode vir a utilizar. Imagine que nossa aplicação seja oferecida para usuários com seleções bem diferentes de seus produtos: um supermercado que importa produtos de várias regiões do Brasil, e uma feira de frutas organizada

por pequenos produtores da região. Nestes dois casos, a aplicação alocaria a mesma quantidade de memória mas teria uma diferença na porção da memória alocada que é desperdiçada; porção essa cujos elementos da tabela permanecem com o valor de inicialização (None) durante o ciclo de vida da aplicação.

Quando armazenamos um conjunto de chaves que é relativamente pequeno com relação ao tamanho do universo ($|U|$), maior o desperdício do espaço alocado pela tabela.

Tabelas de espalhamento, por outro lado, são projetadas justamente para essas situações em que o conjunto de chaves armazenadas é menor do que o conjunto universo de chaves. Seja por quê o universo é muito grande, ou por quê o acesso feito pelo usuário é variado.

Exercício. O método de inicalização da classe 'DirectAddressTable' roda em $O(|U|)$. Como você implementaria uma versão dela que inicializa em $O(1)$? Essa complexidade é importante para viabilizar essa abordagem quando o universo de chaves é muito grande.

12.3 Tabela de espalhamento

Os requisitos para armazenar uma tabela podem ser reduzidos se utilizarmos uma **função *hash*** h que compute o índice de uma chave k. A função h mapeia o universo de chaves, U, para os índices disponíveis na tabela de espalhamento, T:

$$h: \quad U \rightarrow 0, 1, \ldots, m-1$$

, onde m é o tamanho da tabela. A função *hash* reduz o intervalo de índices da tabela e assim o tamanho do seu vetor de dados. Em vez do tamanho do universo de chaves, $|U|$, o vetor pode ter um tamanho m escolhido pelo programador.

Dizemos que um elemento com chave k **mapeia** (hashes) para a posição $h(k)$. Também dizemos que $h(k)$ é **valor hash** da chave k. Duas chaves distintas podem

mapear para o mesmo índice. Chamamos isso de **colisão**. Há técnicas eficazes para resolver o conflito criado por colisões.

A solução ideal é evitar colisões por completo. Porém, como $m < |U|$, ao menos duas chaves devem necessariamente ter o mesmo valor *hash*; portanto, evitar colisões por completo é impossível. Assim,enquanto uma função *hash* bem projetada consegue reduzir o número de colisões, ainda precisaremos de um método para tratar das colisões que irão acontecer.

Resolução de colisões por encadeamento. Uma opção é colocar todos os elementos que mapeiam para um mesmo índice dentro de uma lista encadeada. O índice j da tabela de espalhamento contém, então, um ponteiro para a cabeça de uma lista com todos os elementos armazenados que mapeiam para j; caso não hajam tais elementos, o índice j contém NIL (None, em Python).

Operações de dicionário em uma tabela de espalhamento compreendem uma função *hash* e uma resolução de colisões.

```
from collections import namedtuple
Element = namedtuple("Element", "key value")
# another option is 'from collections import deque'
class LinkedList:
  ################################################
  class Node:
    def __init__(self, left, data, right):
      self.left = left
      self.data = data
      self.right = right
  ################################################
  def __init__(self):
    self.tail = None
  def insert(self, key, value):
```

```
    data = Element(key, value)
    node = LinkedList.Node(None, data, None)
    if self.tail is None:
      self.tail = node
    else:
      self.tail.right = node
      node.left = self.tail
      self.tail = node
  def search(self, key):
    ll_ptr = self.tail
    while ll_ptr and ll_ptr.data.key != key:
      ll_ptr = ll_ptr.left
    return ll_ptr
  def remove(self, key):
    ll_ptr = self.search(key)
    self.unlink(ll_ptr)
  def unlink(self, ll_ptr):
    if self.tail == ll_ptr:
      self.tail = None
    LinkedList.delete(ll_ptr)
  @staticmethod
  def delete(ll_ptr):
    if ll_ptr is None:
      return
    if ll_ptr.left:
      ll_ptr.left.right = ll_ptr.right
    if ll_ptr.right:
      ll_ptr.right.left = ll_ptr.left
```

```
    del ll_ptr
  def __iter__(self):
    self._curr = self.tail
    return self
  def __next__(self):
    if self._curr is None:
      raise StopIteration
    else:
      element = self._curr.data
      self._curr = self._curr.left
      return element
  def __str__(self, sep="; "):
    buffer = ""
    ll_ptr = self.tail
    while ll_ptr:
      buffer = str(ll_ptr.data) + sep + buffer
      ll_ptr = ll_ptr.left
    buffer = buffer[:-len(sep)]
    return "{"+buffer+"}"
```

```
# NOTE: adapt lines marked with '#!!" if using deque
class ChainingHashTable:
  def __init__(self, capacity, hash_function):
    self.slots = [LinkedList() for _ in range(capacity)] #!!
    self.m = capacity
    self.h = hash_function
  def __getitem__(self, key):
    assert 0 <= self.h(key) < self.m
    for element in self.slots[self.h(key)]:
```

```
      if element.key == key:
        return element.value
    return None
  def __setitem__(self, key, value):
    self.slots[self.h(key)].insert(key, value) #!!
  def __delitem__(self, key):
    assert 0 <= self.h(key) < self.m
    self.slots[self.h(key)].remove(key) #!!
  def __contains__(self, key):
    assert 0 <= self.h(key) < self.m
    return key in self.slots[self.h(key)]
  def __str__(self):
    return "\n".join(str(_) for _ in (self.slots))
  # for faster delete ...
  def search(self, key):
    assert 0 <= self.h(key) < self.m
    return self.slots[self.h(key)].search(key) # ... return
       ll_ptr
  def delete(self, ll_ptr):
    assert ll_ptr and ll_ptr.data
    key = ll_ptr.data.key
    assert 0 <= self.h(key) < self.m
    self.slots[self.h(key)].delete(ll_ptr)
```

```
dummy = lambda key: key
run_fruits_test_with_adhoc_keys(lambda: ChainingHashTable(7,
   dummy))
```

```
INSERT '0', 4
```

```
| SEARCH 0: None
| SEARCH 0: 4
-------------
INSERT '1', 6
| SEARCH 1: None
| SEARCH 1: 6
-------------
INSERT '2', 120
| SEARCH 2: None
| SEARCH 2: 120
---------------
INSERT '3', 120
| SEARCH 3: None
| SEARCH 3: 120
---------------
INSERT '4', 120
| SEARCH 4: None
| SEARCH 4: 120
---------------
INSERT '5', 20
| SEARCH 5: None
| SEARCH 5: 20
--------------
{Element(key=0, value=4)}
{Element(key=1, value=6)}
{Element(key=2, value=120)}
{Element(key=3, value=120)}
{Element(key=4, value=120)}
```

```
{Element(key=5, value=20)}
{}
SEARCH 5: 20
INSERT '6', 3
| SEARCH 6: None
| SEARCH 6: 3
-------------
DELETE '0'
| SEARCH 0: 4
| SEARCH 0: None
----------
{}
{Element(key=1, value=6)}
{Element(key=2, value=120)}
{Element(key=3, value=120)}
{Element(key=4, value=120)}
{Element(key=5, value=20)}
{Element(key=6, value=3)}
```

```
h = lambda key: key % 3
run_fruits_test_with_adhoc_keys(lambda: ChainingHashTable(3, h)
```

```
INSERT '0', 4
| SEARCH 0: None
| SEARCH 0: 4
-------------
INSERT '1', 6
| SEARCH 1: None
| SEARCH 1: 6
```

```
-------------
INSERT '2', 120
| SEARCH 2: None
| SEARCH 2: 120
---------------
INSERT '3', 120
| SEARCH 3: None
| SEARCH 3: 120
---------------
INSERT '4', 120
| SEARCH 4: None
| SEARCH 4: 120
---------------
INSERT '5', 20
| SEARCH 5: None
| SEARCH 5: 20
--------------
{Element(key=0, value=4); Element(key=3, value=120)}
{Element(key=1, value=6); Element(key=4, value=120)}
{Element(key=2, value=120); Element(key=5, value=20)}
SEARCH 5: 20
INSERT '6', 3
| SEARCH 6: None
| SEARCH 6: 3
-------------
DELETE '0'
| SEARCH 0: 4
| SEARCH 0: None
```

```
----------
{Element(key=3, value=120); Element(key=6, value=3)}
{Element(key=1, value=6); Element(key=4, value=120)}
{Element(key=2, value=120); Element(key=5, value=20)}
```

Quão bom é o desempenho de *hash* com encadeamento? Em particular, quanto tempo leva para buscar um elemento com uma dada chave? Dada uma tabela *hash* T com m índices, os quais armazenam n elementos, defini-se **fator de carga**, α, como a média de elementos armazenados em uma cadeia, ou seja, $\alpha = m/n$. Em uma tabela *hash* cujas colisões são resolvidas com encadeamento, uma busca malsucedida leva um tempo médio de $O(1+\alpha)$ quando a função é simples e uniforme (já, já falaremos dessas propriedades). O caso de uma busca bem-sucedida é ligeiramente diferente, pois a probabilidade de uma lista encadeada receber uma busca é proporcional ao número de elementos que contém. De toda maneira, a análise da complexidade do tempo de busca permanece $O(1+\alpha)$.

O que esta análise significa? Se o número de índices da tabela de espalhamento é, ao menos, proporcional ao número de elementos armazenados, $n = O(m)$, temos que a operação de busca leva um tempo médio constante: $\alpha = n/m = O(m/m) = O(1)$. E, como operações de inserção e de deleção levam $O(1)$ no pior caso quando as listas são duplamente encadeadas, podemos suportar operações de dicionário em $O(1)$.

12.4 Funções *hash*

Uma função *hash* mapea uma chave para sua respectiva posição no vetor de dados. Você coloca uma palavra, "Abacaxi", e recebe a linha que tens que consultar no catálogo.

Uma função *hash* tem que ser consistente. Se você coloca "Abacaxi"e recebe "4", então toda outra vez que colocares "Abacaxi"você continua recebendo "4". Sem isso, a tabela de espalhamento não funcionaria.

Uma função *hash* sabe o quão grande é o vetor de dados, e só retorna índices válidos.

Se a tabela possui 5 índices, não utilizaremos uma função que pode retornar 6; um valor inválido para indexar o vetor.

Uma função hash também mapeia diferentes chaves para diferentes índices. "Abacaxi"mapeia para 0. "Abacate"mapeia para 1.

Isso é, exceto quando ocorre colisões.

O que faz uma função *hash* ser boa? Uma boa função *hash* satisfaz (aproximadamente) a suposição de **hashing simples e uniforme**: toda chave possui uma mesma probabilidade de mapear para cada um dos m índices da tabela, independente de para quais índices as outras chaves mapeiam.

Por exemplo, dado que as chaves são números reais obtidos através de escolhas independentes e aleatório do intervalo $0 \leq k < 1$ e que essas escolhas produzem uma distribuição uniforme dos mesmos, então sabe-se que a função hash $h(k) = \lfloor km \rfloor$ é simples e uniforme.

Porém normalmente desconhecemos a distribuição aleatória por trás da seleção das chaves. E tipicamente não temos como validar essas propriedades de "simples e uniforme".

Um outro fator é a possibilidade de aplicações realizarem acessos com inter-dependência entre as chaves. Como, por exemplo, um católogo de preços no qual dois produtos que costumam ser comprados juntos apesar de serem vendidos separados.

Assim, na prática, técnicas heurísticas são utilizadas para criar uma função *hash* com um bom desempenho. E, por conseguinte, avaliamos que o desempenho de uma função *hash* é bom se a mesma deriva o valor *hash* de uma maneira que é independente de qualquer padrão que possa existir nos dados.

Duas maneiras para criar boas funções *hash*, **hashing por divisão** e **hashing por multiplicação**, são heurísticas por natureza. Uma terceira maneira, ***hashing* universal**, utiliza de aleatoriedade para oferecer um desempenho que é provavelmente bom.

12.4.1 Chaves como números naturais

A maoria das funções *hash* assumem que o universo de chaves é conjunto de números naturais. Assim, se as chaves não são números naturais, é necessário definir uma maneira de interpretá-los como tal.

Um exemplo é interpretar uma cadeia de characteres como um número inteiro expresso notação de base. Dado que o conjunto ASCII contém 128 caracteres, podemos interpretar "Ab"como um par de números inteiros, $(65,98)$, e expressar "Ab"como um inteiro na base 128: $(65 * 128) + 98 = 8418$.

```
def radix_notation(string, base=128):
  integer, unit = 0, 1
  for c in string[::-1]:
    integer+= ord(c) * unit
    unit*= base
  return integer
print(radix_notation("Ab"))
print(radix_notation("\0Ab"))
print(radix_notation("Ab\0"))
print(radix_notation("Ab\0\0\0\0\0"))
print(radix_notation("Abacaxi"))
```

```
8418
8418
1077504
289240277581824
289266525043817
```

Por vezes, o suporte de cadeias de caracteres como chaves também compreende adaptar as técnicas de *hashing* para lidar com números naturais gigantes. Esse não é um assunto para esse capítulo.

12.4.2 Método da divisão

O método da divisão define funções *hash* através do resto da divisão de uma chave k pelo número de índices m da tabela: $h(k) = k\%m$. Como requer apenas uma simples divisão, esse método é bem rápido.

Ao utilizar o método da divisão, costumamos evitar certos valores de m. Por exemplo, quando m é potência de 2, $m = 2^p$, $h(k)$ retorna os p bits menos significativos da chave k. Logo m só deve ser uma potência 2 se sabemos que todo possível combinação dos bits menos significativos das chaves possue a mesma probabilidade de ser acessado pelo usuário - o que, assim, satisfazeria a condição de uniformidade. Caso o acesso do usuário siga algum padrão binário, é melhor que designemos uma função *hash* que utilize todos os bits das chaves, e não apenas os menos significativos. Escolher uma potência de dois, $m = 2^{p-1}$, quando a chave k é uma interpretação em base 128 de uma cadeia de characteres é uma decisão ruim pois certas permutações dos characteres da chave k não alteram o seu valor *hash*.

```
def hash_by_division(m):
  return lambda k: k % m
```

```
def test_hash_function(h, k):
  print("Hashing",k,"to",h(k))


def run_abacaxi_test(hashing_method, m):
  h = hashing_method(m)
  hash_string = lambda string: h(radix_notation(string))
  test_hash_function(hash_string, "Abacaxi")
  test_hash_function(hash_string, "    axi")
  test_hash_function(hash_string, "cabAaxi")
```

```
m = 2 ** 10
run_abacaxi_test(hash_by_division, m)
```

```
Hashing Abacaxi to 105
Hashing     axi to 105
Hashing cabAaxi to 105
```

Um número primo que não está próximo de uma potência de dois costuma ser uma boa escolha para m. Suponha que queremos alocar uma tabela *hash*, cujas colisões são resolvidas por encadeamento, para armazenar aproximadamente $n = 2000$ cadeias de caracteres, onde cada caractere possui 8 bits. Estaria tudo bem se uma busca malsucedida examinasse uma média de 3 elementos. Logo, decidimos por alocar uma tabela cujo tamanho é $m = 701$. A escolha de $m = 701$ é boa pois 701 é um número primo próximo da quantidade de índices almejados, $2000/3 = 666$, e que também não está próximo de uma potência de dois. Após tratar cada chave k como um número inteiro, nossa função *hash* seria $h(k) = k \% 701$.

```
m = 701
run_abacaxi_test(hash_by_division, m)
```

```
Hashing Abacaxi to 205
Hashing     axi to 568
Hashing cabAaxi to 134
```

12.4.3 Método da multiplicação

O método da multiplicação cria funções *hash* que operam em dois passos. Primeiro, o método multiplica a chave k por uma constante A do intervalo $0 < A < 1$ e extrai a parte fracionária do número resultante, $kA \bmod 1$. Em seguida, o método multiplica esse valor por m e arredonda o resultado para o número inteiro mais próximo. Ou seja, a função *hash* é $h(k) = \lfloor (kA \bmod 1)m \rfloor$.

Uma vantagem do método é que o valor de m não é crítico. E normalmente escolhemos uma potência de dois como valor de m, uma vez que essa é a base com que computores típicos naturalmente utilizam.

O método também funciona bem com qualquer valor da constante A, mas há certos valores com que funciona melhor. A escolha ótima depende das características dos dados que estão sendo mapeados para a tabela. Uma sugestão, de Donald Ervin Knuth, que costuma funcionar razoavelmente bem é $A = (\sqrt{5}-1)/2 = 0.6180339887$. Esse conjugado da expressão áurea é chamado de ***hashing* de Fibonacci**.

```
import math
def hash_by_multiplication(m, A):
  return lambda k: math.floor(((k * A) % 1) * m)
```

```
A = (math.sqrt(5) - 1)/2
m = 701
h = hash_by_multiplication(radix_notation(key), m, A)
print("A: ", A)
test_hash_function(h, "Abacaxi")
test_hash_function(h, "    axi")
test_hash_function(h, "cabAaxi")
```

```
A:  0.6180339887498949
Hashing Abacaxi to 416
Hashing     axi to 427
Hashing cabAaxi to 306
```

Vejamos um outro exemplo, com chaves numéricas. Digamos que nossa tabela possui 2^{14} índices e queremos mapear a chave $k = 123456$.

```
A = (math.sqrt(5) - 1)/2
m = 2 ** 14
h = hash_by_multiplication(m, A)
k = 123456
print("A: ", A)
test_hash_function(h, k)
```

```
A:  0.6180339887498949
Hashing 123456 to 67
```

Neste exemplo, a função nos retorna o índice 67 da tabela.

Podemos ir além na adapção do nosso código e utilizar de manipulações algébricas para realizar operações binárias.

```
def binary_hash_by_multiplication(s, p, w):
  def _function(key):
    step1_multiplication = key * s
    step2_w_lowest_bits = step1_multiplication & ((2**w)-1)
    step3_p_highest_bits = step2_w_lowest_bits >> (w - p)
    return step3_p_highest_bits
  return _function
```

```
s = 2654435769  # A = (math.sqrt(5) - 1)/2
p = 14          # m = 2 ** p
w = 32
h = binary_hash_by_multiplication(s, p, w)
k = 123456
print("s: ", s)
test_hash_function(h, k)
```

```
s:  2654435769
Hashing 123456 to 67
```

Com a configuração de $m = 2^{14}$, $p = 14$, e $w = 32$, o código acima adapta a constante A sugerida por Knuth para a forma fracionária $s/2^{32}$ que é mais próxima do seu valor

original.

$$\begin{aligned} A &= 2654435769/2^{32} \\ s &= 2654435769 \\ A &= s/2^{32} \\ (k * A) \bmod 1 &= (k * s/2^{32}) \bmod 1 \end{aligned} \tag{12.1}$$

E aplica o método da multiplicação com operações binárias que preservam o valor *hash* $h(k) = 67$.

$$\begin{aligned} k * s &= 327706022297664 \\ &= 76300 * 2^{32} + 17612864 \\ (k * s/2^{32}) \bmod 1 &= 17612864/2^{32} \end{aligned} \tag{12.2}$$

Para $k = 123456$, a multiplicação $k * A$ produz um número cuja parte inteira é 76300 e a parte fracinária é 17612864×2^{-32}, portanto, o resultado de $(k * A \bmod 1) * 2^{32}$ é 17612864.

Podemos calcular esse número inteiro, 17612864, através de um operador de conjunção binária, &,que equivala a operação modular 2^{32}.

$$\begin{aligned} ((k * A) \bmod 1) * 2^{32} &= (k * s) \bmod 2^{32} \\ &= (k * s)\&(2^{32} - 1) \end{aligned} \tag{12.3}$$

Há mais uma operação binária que podemos fazer no método mutiplicação. Lembre

que $m = 2^p$ e $x/2^e = x \gg e$. Então:

$$\begin{aligned}
\lfloor((k * A) \bmod 1)m\rfloor &= \lfloor(((k * s) \bmod 2^{32})/2^{32})m\rfloor \\
&= 17612864 \gg (32-14) = 67, p = 14 \\
&= \lfloor(((k * s) \bmod 2^{32})/2^{32})2^p\rfloor \\
&= \lfloor(((k * s) \bmod 2^{32})2^{p-32})\rfloor \\
&= \lfloor(((k * s) \bmod 2^{32})/2^{32-p})\rfloor, p < 32 \\
&= \lfloor((k * s) \bmod 2^{32})/2^{32-p}\rfloor \\
&= \lfloor((k * s)\&(2^{32}-1))/2^{32-p}\rfloor \\
&= \lfloor((k * s)\&(2^{32}-1)) \gg (32-p)\rfloor \\
(k * A) \bmod 1 &= 17612864/2^{32}, k = 123456 \\
(2^{32}) * (k * A) \bmod 1 &= 17612864 \\
\lfloor((k * A) \bmod 1)m\rfloor &= ((2^{32}) * (k * A) \bmod 1)/2^{32-p} \\
&= 17612864/2^{32-p} \\
&= 17612864 \gg (32-p)
\end{aligned} \tag{12.4}$$

12.4.4 *Hashing* universal

Enquanto a função *hash* for fixa, a tabela está vulnerável a um comportamento $O(n)$ de pior caso.

Imagine que um adversário conheça a função *hash* da sua aplicação e maliosamente escolhe chaves que possuem o mesmo valor *hash*. Como as chaves seriam mapeadas para o mesmo índice da tabela, o tempo médio para recuperar os respctivos elementos destas chaves na lista encadeada seria $O(n)$.

A única maneira efetiva de melhorar esse cenário de pior caso é se a escolha da função *hash* é **aleatória** e **independente** das chaves que serão armazenadas. Essa abordagem é chamada de **hashing universal**, e possui um bom desempenho médio independente das chaves que um adversário possa escolher.

Hashing universal. A inicialização de uma tabela com *hashing* universal consiste em selecionar uma função *hash* ao aleatório. Para isso um coleção de funções *hash* é cuidadosamente projetada. A aleatoridade oferece garantias de que nenhum grupo de operações do usuário (como inserções de chaves) irá produzir um comportamento de pior caso. Como a função *hash* é escolhida ao acaso, a tabela exibe diferentes comportamentos em cada uma de suas execuções, mesmo aquelas em que recebe o mesmo grupo de operações. O comportamento de baixo desempenho ainda é possível, e ocorre quando a tabela seleciona uma função cujo mapeamento de um conjunto de entrada é ineficiente. Portanto, a coleção de funções *hash* é projetada para garantir que a probabilidade de que essa situação aconteça seja baixa.

Um conjunto de funções *hash*, `H`, que mapeiam um certo conjunto de chaves, U, para um dado intervalo de índices, $\{0, 1, \ldots, m-1\}$, é **universal** se, para toda função *hash* da coleção, a chance de ocorrer uma colisão entre duas chaves distintas não é maior do que a probabilidade natural $1/m$ de escolher dois valores *hash* aleatórios e os mesmos serem iguais: $Pr\{h(k) = h(l)\} \leq 1/m$.

Projetando uma coleção universal de funções *hash* universal. O projeto de uma coleção universal baseai-se na Teoria dos Números. O método original, proposto por Carter e Wegman, define uma coleção de funções $h_{a,b}(k)$ que realizam uma transformação linear, $ak+b$, seguida por duas reduções modulares; primeiro uma redução em um número primo p, depois uma redução em m. O número primo p há de ser grande o suficiente para incluir todo valor chave k possível: $0 < k \leq p-1$. E os parâmetros da transformação linear, a e b, são números inteiros aleatórios pertecentes ao intervalo $[0, p-1]$, e $a \neq 0$.

```
def universal_hash(a, b, p, m):
  assert type(a) == int and 1 <= a < p
  assert type(b) == int and 0 <= b < p
  return lambda k: ((a*k+b) % p) % m
```

```
import random
def select_hash_function(m, p, seed=None):
  random.seed(seed)
  a = random.randint(1, p-1)
  b = random.randint(0, p-1)
  return universal_hash(a, b, p, m)
```

Como a pode assumir p valores e b pode assumir $p-1$ valores, acabamos de definir uma coleção com $p(p-1)$ funções *hash*. Agora, para provar que a coleção é universal, precisamos avaliar a probabilidade $Pr\{h(k) = h(l)\}$ de duas chaves distintas mapearem para o mesmo índice.

Sabemos que toda função $h_{a,b}(k)$ da coleção realiza uma transformação linear módulo um número primo p:

$$r = (ak+b) \pmod{p} s = (al+b) \pmod{p}$$

Com a teoria de números, podemos manipular essas expressões para:

$$r - s \equiv a(k-l) \pmod{p}$$

E provar que $r \neq s$ quando tanto a quanto $k-l$ são diferentes de 0 na redução modular com o número primo p:

$$a \not\equiv 0 \pmod{p} k - l \not\equiv 0 \pmod{p}$$

Como $r \neq s$, a probabilidade $Pr\{h(k) = h(l)\}$ de duas chaves distintas colidirem é igual à probabilidade de $r \equiv s \pmod{m}$ quando r e s são valores distintos módulo p e escolhidos ao aleatório. Para um valor de r qualquer, de todos os outros $p-1$ valores possíveis para s, o número de valores de s que $s \neq r$ e $s \equiv r \pmod{p}$ é no máximo $\lceil p/m \rceil - 1$, que se reduz para $(p-1)/m$. Assim $Pr\{h(k) = h(l)\} \leq (1/(p-1))((p-1)/m) = 1/m$, e temos que $h_{ab}(k)$ define uma coleção universal de funções *hash*.

Também podemos definir uma coleção universal com base em operadores binárias.

A chave para esta adaptação é de preservar a primalidade relativa entre p e a, ou seja, a precisa ser um número inteiro ímpar: $a \equiv 1 \pmod 2$. Além disso p precisa ser uma potência de dois que inclui o universo de chaves: $p = 2^w$, onde w é o número de bits necessários para representar as chaves. E assim podemos definir uma coleção com base em multiplicações e deslocamentos binários:

$$h_{ab}(k) = ((ak + b) \,\%\, 2^w)/2^{w-M},$$

onde $M = \lceil log_2\ m \rceil$ e a divisão por 2^{w-M} obtém os M bits mais significativos de $((ak + b) \,\%\, 2^w)$.

```
def universal_hash_binary(a, b, M):
  assert type(a) == int and 1 <= a and a % 2 == 1
  assert type(b) == int and 0 <= b
  return lambda k: ((a*k+b)&0xffff) >> (32-M)


import random
def select_hash_function_binary(m, p, seed=None):
  M = math.ceil(math.log(m, 2))
  random.seed(seed)
  p = 1 << 32
  i = random.randint(0, 1 << 31)
  a = (i << 1) + 1
  b = random.randint(0, p)
  return universal_hash_binary(a, b, M)
```

12.5 Endereçamento Aberto

Até o momento vimos uma única abordagem para resolver colisões: encadeamento. Outra abordagem é **endereçamento aberto**.

Endereçamento aberto Na resolução com enderaçamento aberto, cada elemento pode ocupar qualquer um dos índices da tabela. Cada índice contém ou um elemento do conjunto dinâmico ou um valor nulo (*None*). Assim, uma busca examina sistematicamente cada um dos índices da tabela até encontrar o elemento desejado, ou até determinar que o mesmo não se encontra na tabela. Nenhum elemento é armazenado fora da tabela; diferente da resolução por encadeamento. Com endereçamento aberto, a tabela pode ficar cheia e, portanto, não permitir operações de inserção subsequêntes; em outras palavras, o **fator de carga** nunca excede 1.

Pode-se, na verdade, utilizar uma lista para enceadear elementos e inserí-los em índices inutilizados pela tabela. Mas a vantagem de endereçamento aberto é justamente de evitar ponteiros. Em vezes de seguir ponteiros, a tabela segue uma **sequência** de índices a serem examinados. A memória extra liberada por não armazenarmos ponteiros permite o uso de **tabelas maiores**, que possuem um maior número de índices para uma mesma quantidade de memória alocada; o que potencialmente produz menos colisões e aumenta a rapidez em recuperar dados.

Para realizar um inserção em uma tabela com enderaçamento aberto, sucessivas acessos, **probes**, são realizados até encontrarmos o índice no qual a chave foi inserida. Acessos sequênciais, na ordem "0, 1, 2, ...", são ineficientes e produziriam buscas a $O(n)$. Em vez disso, a sequência de índices acessados **depende da chave que está sendo inserida**. Uma função *hash* continua a determinar em qual índice armazenaremos uma dada chave. Então, para determinar uma **sequência** de índices, as funções *hash* são extendidas para incluir um segundo parâmetro: o número do acesso (começando de zero) a ser realizado. Na abordagem de endereçamento aberto, toda chave k está associada a uma **sequência de acesso**; uma permutação do acesso linear, $< 0, 1, \ldots, m - 1 >$, em que todo índice da tabela *hash* é eventualmente considerado como uma opção para armazenar uma nova chave da tabela, até que a mesma fique cheia.

```
class OpenAddressingHashTable:
  def __init__(self, capacity, probe_sequence):
```

```
    self.slots = [None for key in range(capacity)]
    self.m = capacity
    self.h = probe_sequence
  def _probe(self, key, continue_if):
    idx = self.h(key, 0)
    i = 1
    while continue_if(self.slots[idx]) and i < self.m:
      idx = self.h(key, i)
      i = i + 1
    return idx if i < self.m else None
  def __getitem__(self, key):
    idx = self._probe(key, continue_if=lambda element: element
       is None)
    return self.slots[idx] if idx else None
  def __setitem__(self, key, value):
    idx = self._probe(key, continue_if=lambda element: element
       is None)
    if idx:
      self.slots[idx] = value
    else:
      raise "err: Hash Table is full"
  def __delitem__(self, key):
    raise "err: operation not implemented"
  def __contains__(self, key):
      return self.__getitem__(key) is not None
  def __str__(self):
    return str(self.slots)
```

Operações de inserção, busca e deleção seguem a sequência de índices determinada

pela função *hash* para a chave almejada. Cada índice visitado irá conter ou a chave almejada ou o valor nulo. Operações de inserção procuram pelo primeiro índice cujo valor é nulo, e nele armazenam a chave passada como argumento. Caso nenhum índice contém valor nulo, a tabela está cheia, e a operação de inserção sinaliza um erro.

Operações de busca seguem a sequência de índices determinada pela chave almejada, e retornam o índice no qual ela foi armezenada. A operação de busca termina (sem sucesso) assim que se depara com um valor nulo, uma vez que a chave haveria de ser armazenada neste índice em vez de ser armezanada em um outro índice subsequênte da sequência.

As operações de deleção são um desafio a parte. Quando uma chave k é deletada de um índice i, a tabela não pode simplesmente armazenar o valor nulo para marcar o índice como vazio. Se o fizesse, as operações de busca não conseguiriam mais recuperar nenhuma das chaves cujas inserções acessaram o índice i e constataram que estava ocupado. Uma maneira de resolver esse problema é marcar o índice i com um valor especial, *DELETED*, em vez de nulo. Desta maneira é possível sinalizar para as operações de inserção quais índices devem ter o seu valor tratado como nulo (e, portanto, estão livres para serem sobrescritos), e sinalizar para as operações de busca quais índices possuem uma chave diferente da buscada (e, portanto, a busca deve continuar). No entanto, o uso do valor *DELETED* degrada o tempo de busca, o qual passa a não depender mais do fator de carga α. Por essa razão, **a resolução por encadeamento costuma ser a opção para quando as chaves precisam ser deletadas**.

A seguir, veremos as três técnicas que costumam ser utilizadas para definir a sequência de acesso: **sequência linear**, **sequência quadrática**, e **mapeamento duplo**. Cada uma delas garante uma permutação do acesso linear para cada chave. Nenhuma, por outro lado, cumpre com a hipótese de *hashing* universal, afinal nenhuma das técnicas é capaz de gerar mais do que m^2 sequências de acesso diferentes (o que é menor do que as $m!$ sequências que *hashing* universal necessita). A técnica de mapeamento duplo produz o maior número de sequências e costuma, assim, oferecer os melhores

resultados.

12.5.1 Sequência linear de acessos

A técnica de acesso linear utiliza uma **função hash auxiliar** $h' : U \mapsto \{0, 1, \ldots, m-1\}$ junto de um deslocamento modular:

$$h(k, i) = (h'(k) + i) \bmod m,$$

onde i denota uma posição da sequência de acessos

Dada uma chave k, o primeiro acesso ocorre no valor *hash* dado pela função *hash* auxiliar: $h'(k)$. O acesso subsequente ocorre no índice posterior $h'(k)+'$, e assim segue-se sucessivamente até o índice $m-1$ for acessado. Em seguida o acesso ocorre no índice inicial da tabela, e segue-se o aumento incremental, "0, 1, 2, ...", até acessar o índice anterior ao primeiro acesso: $h'(k)-1$. O índice inicial da sequência, $h'(k)$, determina que todos os m índices da tabela foram acessados, e a sequência chega ao fim.

Essa técnica linear é fácil de implementar, mas sofre de um problema conhecido como **clustering primário**. Os índices da tabela são ocupados em uma longa sequência ininterrupta, que aumenta o tempo médio das operações de busca. **Clusters** se formam em vista da probabilidade de um índice estar ocupado aumenta conforme a sua posição no vetor de dados por trás da tabela *hash*: a probabilidade do índice i estar ocupado é $(i+1)/m$. Uma longa sequência contínua de índices que estão ocupados tende a ficar ainda mais longa, assim como o tempo médio de busca tende a aumentar.

```
def linear_probe_sequence(h, m):
  return lambda k,i: (h(k) + i) % m
```

12.5.2 Sequência quadrática de acessos

A técnica de acesso quadrática utiliza uma função hash auxiliar

$$h' : U \mapsto 0, 1, \ldots, m-1$$

junto da soma modular de uma função quadrática na posição i da sequência:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

onde c_1 e c_2 são constantes positivas, e $i = 0, 1, \ldots, m-1$. O acesso inicial será no índice $h'(k)$; os acessos subsequentes ocorrem com um deslocamento quadrático em i. Assim quanto maior a quantidade de acessos necessários para encontrar um índice livre da tabela, mais distante é o índice que procuramos acessar.

Esta técnica funciona melhor que o acesso linear. Mas é necessário fazer uma boa escolha dos valores de c_1, c_2, e m. Por exemplo, para $m = 2^n$, as constantes $c_1 = 1/2$ e $c_2 = 1/2$ são uma boa escolha; os valores de $h(k,i)$ são todos distintos, reproduzem, através da sequência de números triangulares, um deslocamento que aumenta linearmente, 1, 2, 3, 4, etc.

Caso duas chaves possuam o mesmo valor hash, $h'(k) = h'(l)$, temos uma colisão que produz a mesma sequência de acesso: $h'(k,0) = h'(l,0)$ e, consequêntemente, $h'(k,i) = h'(l,i)$. Esta é uma outra, mas mais suave, forma de agregação. Chamamos ela de **clustering secundário**.

```
def quadratic_probe_sequence(h, c1, c2, m):
  return lambda k,i: (h(k) + i*(c1 + c2*i)) % m
```

12.5.3 *Hashing* duplo

Hashing duplo oferece uma das melhores técnicas para endereçamento aberto. Isso por quê produz sequências com muitas das características de permutações aleatórias.

A técnica combina o valor *hash* de duas funções auxiliares em uma soma modular:

$$h(k,i) = (h'(k) + h''(k) \times i) \bmod m$$

O acesso inicial ocorre no valor *hash* dado por h'; os acessos sucessivos adicionam um deslocamento dado pela segunda função, h''. Assim, diferente dos acessos lineares

e quadráticos, essa técnica produz uma sequência de acesso que possui duas dependências com a chave k. Tanto o índice do acesso inicial quanto o deslocamento podem variar com k.

Para que a sequência compreenda todos os índices da tabela *hash*, é necessário que o deslocamento, $h''(k)$, seja um número relativamente primo com m. Uma maneira simples de garantir essa propriedade é adotar uma potência de dois como m e escolher uma função h'' que produz apenas números ímpares. Uma outra maneira é adotar um m primo e uma função h'' que retorna apenas números menores do que m.

Como cada par $(h'(k), h''(k))$ possível de valores *hash* produz uma sequência diferente de acessos, *hashing* duplo produz $O(m^2)$ sequências. Essa é uma melhora com relação aos acessos lineares e quadráticos, os quais produzem $O(m)$ sequências.

```
def double_hashing_sequence(h1, h2, m):
  return lambda k,i: (h1(k) + h2(k)*i) % m
```

12.6 *Hashing* Perfeito

Quando o conjunto de chaves é estático, e nunca muda após ser armazenado, podemos implementar uma tabela de espalhamento cujo desempenho permanece ótimo mesmo no pior caso.

A denominação de ***hashing* perfeito** refere-se a tabelas de espalhamento que necessitam de apenas um número constante, $O(1)$, de acessos à memória mesmo no pior cenário de uma operação de busca.

Para criar um esquema de *hashing* perfeito, dois níveis de *hashing* universal são empregados.

O primeiro nível é essencialmente o que vimos até o momento com a resolução de colisões com encadeamento: define-se o mapeamento de n chaves em m índices por meio de uma função *hash* cuidadosamente selecionada de uma coleção de funções universais.

O segundo nível é diferente. Em vez de utilizar uma lista encadeada para armazenar as chaves mapeadas para o índice $j = h(k)$, podemos utilizar uma tabela de espalhamento secundária, S_j, com sua respectiva função h_j. Com uma escolha cuidadosa da função *hash* secundária h_j, pode-se garantir que não haverá colisões no segundo nível.

Para garantir que não exista colisões no segundo nível, porém, precisa-se definir um tamanho m_j das tabelas secundárias S_j que seja grande o suficiente. Um tamanho quadrático com o número de chaves mapeadas para o índice j, $m_j = |S_j|^2$, é suficiente.

Intuitivamente, essa dependência quadrática de m_j com n_j parece ocasionar um requisitivo excessivo de armazenamento. Mas uma boa escolha de função *hash* para o primeiro nível consegue limitar a quantidade esperada em $O(n)$ de armazenamento.

Uma opção é escolher funções de uma coleção universal. Por exemplo, a função *hash* primária pode ser escolhida do conjunto $H_{p,m}$ onde p é um número primo maior do que qualquer chave. Neste caso toda chave é mapeada para um índice j e pode ser re-mapeada para uma tabela secundária com m_j índices através de uma função do conjunto H_{p,m_j}. (Quando $n_j = m_j = 1$, uma função *hash* $h_j(k)$ é desnecessária; um simples $a = b = 0$ é suficiente para $h_j(k) = h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$.)

Quando $m = n^2$, uma função *hash* h escolhida ao aleatório de um conjunto universal H, é mais provável de não produzir colisões do que o contrário. Logo, só algumas poucas tentativas são necessárias para encontrar uma função livre de colisões para um dado conjunto k de chaves (o qual, lembre, é fixo). Neste caso apenas um nível é necessário.

Quando n é grande, no entanto, uma tabela com $m = n^2$ índices é excessivo. Este é o caso em que adota-se a abordagem com dois níveis de função *hash*, e a escolha de uma função *hash* aleatória ocorre apenas para mapear as chaves na tabela secundária. Uma função *hash* primária mapeia n chaves para $m = n$ índices. Em seguida, dado que n_j chaves mapeiam para o índice j, utilizamos uma tabela secundária S_j com o tamanho para suportar $m_j = n_j^2$ índices e, assim, oferecer uma consulta livre de colisões e com tempo constante.

Se o tamanho da tabela primária é $m = n$, então a quantidade alocada de memória

é $O(n)$ para a tabela primária, para o armazenamento dos tamanhos m_j de cada tabela secundária, e para o armazenamento dos parâmetros a_j e b_j que definem cada função *hash* secundária que foi escolhida do conjunto H_{p,m_j} (exceto quando $n_j = 1$, e não é necessário armazenar que $a_j = b_j = 0$). O tamanho total esperado ao combinar o tamanho individual de cada tabela secundária é $2n$.

Agora o que falta é garantir que este esquema utiliza uma quantidade total de memória linear com o número de chaves: $O(n)$.

Uma vez que o tamanho m_j da j-ésima tabela secundária aumenta quadraticamente com o número n_j de chaves armazenadas, há o risco de que a quantidade de armazenamento ser excessivo.

A solução? Avaliar se a função primária aloca uma quantidade excessiva de armazenamento no segundo nível e, se sim, selecionar outra função.

Pode-se provar que a probabilidade de que o tamanho total das tabelas secundárias seja indesejado (maior ou igual a 4n) é apenas de 1/2. Ou seja, é rápido encontrar uma função primária que aloque uma quantidade razoável de armazenamento.

12.7 Outros casos de uso

12.7.1 Tradução dos Nomes de Domínio (DNS)

O seu celular possui uma conveniente aplicativo de contatos. Nele, cada nome está associado a um número de telefone.

A funcionalidade do aplicativo compreende:

- Adicionar o nome de um contato e o seu respectivo número de telefone,
- Entrar com o nome do contato e consultar o telefone associado.

Tabelas de espalhamento são boas em:

- Mapear uma coisa para outra,

- Consultar a informação associada a uma chave.

Assim um aplicativo de contatos é um perfeito caso de uso das tabelas.

Outro caso de uso, em uma outra escala muito maior, é o sistema de nomes de domínio (DNS). Esse é o sistema responsável por traduzir o nome de um domínio (digamos, “https://adit.io/“) para o seu respectivo endereço IP (“https://173.255.248.55/“).

Qualquer site que visitas tem o seu nome de domínio traduzido para um endereço IP. Tabelas de espalhamento são uma maneira de implementar a funcionalidade de DNS.

12.7.2 Prevenir entradas duplicadas

Imagine que és responsável por uma cabine de votação. Naturalmente, toda pessoa pode votar uma única vez apenas. Como você pode garantir que uma pessoa já não votou?

Quando alguém entra na cabine, você inquere o nome completo do eleitor. Em seguida, você verifica o nome na lista de pessoas que já votaram. Se o nome está na lista, a pessoa já votou - e você os chuta para fora! Caso contrário, você adiciona o nome deles na lista e os permite votar.

12.7.3 Memorização (Cache)

Imagine que sua prima adora fazer peguntas sobre planetas. Qual a distância entre a Terra e Marte? Quão distante é a Lua? E Júpite?! Toda vez que ela pergunta você têm que consultar o Google. Você leva alguns segundos. Agora, imagine que essa prima têm umas perguntas favoritas. Ela sempre, sempre, pergunta a distância entre a Lua e a Terra. E não demora muito que você decora a resposta, 384.400 km, e começa a respondê-la no mesmo instânte!

Essa é a maneira de memórias cache trabalham. Seja no seu computador, ou no navegador. Sites memorizam dados para diminuir o tempo de resposta.

Um servidor recebe muitas consultas para a sua página inicial. Logo, logo, o seu navegador não precisa mais pensar consultar o servidor para saber como renderizar a página que você quer ver, ele a memorizou. *Caching* diminui tanto o tempo de resposta do navegador quanto o processamento no servidor. Todo site grande utiliza *caching*. E esses dados estão salvos em uma tabela de espalhamento.

De uma maneira similar ao DNS, o seu navegador consulta a *cache* quando você requisita uma página *web*. Só se a página não está salva, o navegador consulta o servidor.

Referências

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012. Citado 23 vezes nas páginas 5, 8, 9, 17, 18, 19, 20, 26, 32, 35, 68, 69, 77, 82, 83, 85, 94, 97, 99, 106, 109, 117 e 120.

GROSS, J. T.; YELLEN, J. *Graph Theory and Its Applications*. FL: CRC Press, 2006. Citado na página 177.

KLEINBERG, J.; TARDOS, E. *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358. Citado 10 vezes nas páginas 17, 38, 39, 89, 90, 91, 103, 104, 106 e 112.

KREHER, D. L.; STINSON, D. R. Combinatorial algorithms: Generation, enumeration, and search. *SIGACT News*, Association for Computing Machinery, New York, NY, USA, v. 30, n. 1, p. 33–35, mar 1999. ISSN 0163-5700. Disponível em: <https://doi.org/10.1145/309739.309744>. Citado na página 6.

NETTO, P. O. B. *Grafos: teoria, modelos, algoritmos*. São Paulo: Edgard Blucher, 2006. Citado 2 vezes nas páginas 74 e 77.

ZIVIANI, N. *Projeto de Algoritmos*. 1. ed. São Paulo: Cengage Learning, 2007. ISBN 85-221-0525-1. Citado na página 23.

APÊNDICE A

Caminhos e Ciclos

Este capítulo tem o objetivo de introduzir o conceito de caminhos e ciclos, e seus principais problemas. Dois problemas clássicos serão definidos e algoritmos para os mesmos, apresentados.

Antes de iniciar a abordar os conteúdos deste capítulo, é importante entender o que é um caminho e um ciclo, para estabelecer suas diferenças no contexto de grafos. Um caminho[1] é uma sequência de vértices $\langle v_1, v2, \ldots v_n \rangle$ conectados por uma aresta ou arco. Gross e Yellen (2006) definem um caminho como um grafo com dois vértices com grau 1 e os demais vértices com grau 2, formando uma estrutura linear. Um ciclo (ou circuito)[2] é uma cadeia fechada de vértices $\langle v_1, v_2, \ldots, v_n, v_1 \rangle$ onde cada par consecutivo é conectado por uma aresta ou arco. É como um caminho com o fim e o início conectados.

A.1 Caminhos e Ciclos Eulerianos

Dado um grafo orientado ou não orientado $G = (V, E)$, um caminho Euleriano é uma "trilha" ou seja, uma sequência de arestas/arcos onde cada aresta/arco é visitada(o) uma única vez. O ciclo Euleriano é semelhante ao caminho, com exceção de que começa e termina na mesma aresta/arco. Um grafo é dito Euleriano se possui um ciclo Euleriano.

Os problemas de caminho e ciclo Euleriano surgiram com o conhecido problema das Sete Pontes de Königsberg por Euler em 1736. O problema consistia em atravessar as todas as sete pontes da cidade de Königsberg da Prussia (hoje Kaliningrado na Rússia) sem repetí-las.

---

1 Em inglês, chamado de *path*.

2 Em inglês, chamado de *cycle* ou *circuit*.

Desafio

Observando um mapa antigo das sete pontes, você consegue determinar o caminho Euleriano?

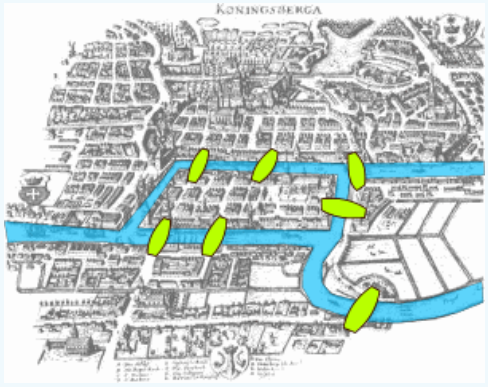


Figura 13 – Mapa das sete pontes de Königsberg na época de in Euler.

A.1.1 Algoritmo de Hierholzer

O algoritmo de Hierholzer (Algoritmo 2) foi desenvolvido em 1873. Ele identifica o ciclo Euleriano em tempo $O(|E|)$.

Algoritmo 2: Algoritmo de Hierholzer.

```
   Input  : um grafo G = (V, E)
1  foreach e ∈ E do
2      C_e ← false
3  v ← selecionar um v ∈ V arbitrariamente, que esteja conectado a uma aresta
   // "buscarSubcicloEuleriano" invoca o Algoritmo 3
4  (r, Ciclo) ← buscarSubcicloEuleriano(G, v, C)
5  if r = false then
6      return (false,null)
7  else
8      if ∃e ∈ E : C_e = false then
9          return (false,null)
10     else
11         return (true, Ciclo)
```

Algoritmo 3: Algoritmo de Auxiliar "$buscarSubcicloEuleriano$".

Input :um grafo $G = (V, E)$,um vértice $v \in V$, o vetor de arestas visitadas C

$Ciclo \leftarrow (v)$
$t \leftarrow v$
repeat
  // Só prossegue se existir uma aresta não-visitada conectada a $Ciclo$.
  if $\nexists u \in N(v) : C_{u,v} =$ ***false*** **then**
    return (**false**,**null**)
  else
    $\{v, u\} \leftarrow$ selecionar uma aresta $e \in E$ tal que $C_e =$**false**
    $C_{\{v,u\}} \leftarrow$ **true**
    $v \leftarrow u$
    // Adiciona o vértice v ao final do ciclo.
    $Ciclo \leftarrow Ciclo \cdot (v)$
until $v = t$
/* Para todo vértice x no $Ciclo$ que tenha uma aresta adjacente não visitada. */
foreach $x \in \{u \in Ciclo : \exists \{u, w\} \in \{e \in E : C_e =$ ***false***$\}\}$ **do**
  $(r, Ciclo') \leftarrow buscarSubcicloEuleriano(G, x, C)$
  if $r =$ ***false*** **then**
    return (**false**,**null**)
  Assumindo que $Ciclo = \langle v_1, v_2, \ldots, x, \ldots, v_1 \rangle$ e $Ciclo' = \langle x, u_1, u_2, \ldots, u_k, x \rangle$, alterar $Ciclo$ para $Ciclo = \langle v_1, v_2, \ldots, \underline{x}, u_1, u_2, \ldots, u_k, x, \ldots, v_1 \rangle$, ou seja, inserir o $Ciclo'$ no lugar da posição de x em $Ciclo$.
return (**true**, $Ciclo$)

Desafio

Explique porque a complexidade de Algoritmo de Hierholzer é de $O(|E|)$.

Teorema A.1.1. *Um grafo não-orientado $G = (V, E)$ é (ou possui um ciclo) Euleriano se e somente se G é conectado e cada vértice tem um grau par.*

Prova: Para o grafo conectado G, para todo $m \geq 0$, considere $S(m)$ ser a hipótese de que se G têm m arestas e todos os graus dos vértices forem pares, então G é Euleriano.

Vamos a prova por indução.

A base da indução é o $S(0)$. Nessa hipótese, G não tem arestas, então para todo $v \in V$, $d_v = 0$. Como zero é par G é trivialmente Euleriano.

O passo da indução implica que as hipoteses $S(0) \wedge S(1) \wedge \ldots \wedge S(k-1) \implies S(k)$. Suponha um $k \geq 1$ e assuma que $S(1) \wedge \ldots \wedge S(k+1)$ é verdade. Precisa-se provar que $S(k)$ é verdade. Suponha que G tenha k arestas, é conectado e possui somente vértices com valor de grau par.

- Desde que G é um grafo conectado e possui vértices com grau par, o menor grau é 2. Então esse grafo G precisa ter um ciclo C.
- Suponha um novo grafo H gerado a partir de G sem as arestas que estão no ciclo C. Note que H pode estar desconectado. Pode-se dizer que H é a união dos componentes conectados $H_1, H_2, \ldots H_t$. O grau dos vértices cada H_i precisa ser par.
- Aplicando a hipótese de indução a cada H_i, que é $S(|E(H_1)|), \ldots, S(|E(H_t)|)$, cada H_i terá um ciclo Euleriano C_i.

- Pode-se criar um circuito Euleriano para G por dividir o ciclo C em ciclos C_i. Primeiro, comece em qualquer vértice em C_i e percorra até atingir outro H_i. Então, percorra C_i e volte ao C até atingir o próximo H_i.

Finalmente, G precisa ser Euleriano. Isso completa o passo da indução como $S(0) \wedge S(1) \wedge \ldots \wedge S(k-1) \implies S(k)$. Por esse princípio, para $m \geq 0$, $S(m)$ é verdadeiro. ■

A.2 Caminhos e Ciclos Hamiltonianos

Ciclos ou caminhos Hamiltonianos são aqueles que percorrem todos os vértices de um grafo apenas uma vez. Mais especificamente para um ciclo Hamiltoniano, o início e o fim terminam no mesmo vértice. O nome Hamiltoniano vem de William Rowan Hamilton, o inventor de um jogo que desafia a buscar um ciclo pelas arestas de dodecaedro (figura tridimensional de 12 faces).

Um grafo é dito Hamiltoniano se possui um ciclo Hamiltoniano.

Há $|V|!$ diferentes sequências de vértices que podem ser caminhos Hamiltonianos, então, um algoritmo de força-bruta demanda muito tempo computacional. O problema de decisão para encontrar um caminho ou ciclo Hamiltoniano é considerado NP-Completo.

A.2.1 Caixeiro Viajante

Dado um grafo completo[3] $G = (V, E, w)$ no qual V é o conjunto de vértices, E é o conjunto de arestas e $w : E \to \mathbb{R}^+$ é a função dos pesos (ou custo ou distâncias), busca-se pelo ciclo Hamiltoniano de menor soma total de peso (menor custo ou distância).

Um dos algoritmos mais eficientes para resolvê-lo é o de programação dinâmica Held-Karp (ou Bellman–Held–Karp, no Algoritmo 4). No entanto, o mesmo demanda tempo computacional de $O(2^{|V|}|V|^2)$.

Algoritmo 4: Algoritmo de Bellman-Held-Karp.

Input : um grafo $G = (V, E = V \times V, w)$

```
for k ← 2 to |V| do
    C({k}, k) ← w({1, k})
for s ← 2 to |V| − 1 do
    foreach S ∈ {x ⊆ {2, 3, ..., |V|} : |x| = s} do
        foreach v ∈ S do
            C(S, v) ← min_{u≠v, u∈S} {C(S\{v}, u) + w({u, v})}
return min_{v∈V\{1}} {C({2, 3, ..., |V|}, v) + w({v, 1})}
```

Desafio

Execute o Algoritmo 4 sobre o grafo $G = (V = \{1,2,3,4\}, E = V \times V, w)$, no qual $w(\{1,2\}) \to 10$, $w(\{1,3\}) \to 15$, $w(\{1,4\}) \to 20$, $w(\{2,3\}) \to 35$, $w(\{2,4\}) \to 25$ e $w(\{3,4\}) \to 30$.

A resposta deve ser 80.

[3] Em um grafo completo, o conjunto de arestas é definido por $E = V \times V$.

APÊNDICE B

Revisão de Matemática Discreta

Conjuntos é uma coleção de elementos sem repetição em que a sequência não importa. No Brasil, utilizamos a seguinte notação para enumerar todos os elementos de um conjunto. Na Equação (B.1), é possível visualizar a representação de um conjunto denominado A, formado pelos elementos e_1, e_2, ..., e_n. Devido ao uso da vírgula como separador de decimais, usa-se formalmente o ponto-e-vírgula. Para essa disciplina, podemos utilizar a vírgula como o separador de elementos em um conjunto, desde que utilizados o ponto como separador de decimais[1]. Para dar nome a um conjunto, geralmente utiliza-se uma letra maiúscula ou uma palavra com a inicial em maiúscula.

$$A = \{e_1; e_2; \ldots; e_n\} \tag{B.1}$$

Há duas formas de definir conjuntos. A forma por enumeração por elementos, utiliza notação semelhante a da Equação (B.1). São exemplos de definição de conjuntos por enumeração:

- $N = \{\diamondsuit, \spadesuit, \heartsuit, \clubsuit\}$;
- $V = \{a, e, i, o, u\}$;
- $G = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$;
- $R = \{-100.9, 12.432, 15.0\}$;
- $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

A forma por descrição de propriedades utiliza-se de uma notação que evidencia a natureza de cada elemento pela descrição de um em um formato genérico. Por exemplo o conjunto D, descrito na Equação (B.2), denota um conjunto com os mesmos elementos em $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

$$D = \{x \in \mathbb{Z} | x > 1 \wedge x \leq 10\} \tag{B.2}$$

. Então para que complicar utilizando uma notação não enumerativa? Por dois motivos: por questões de simplicidade, dado a quantidade de conjuntos; ou para representar conjuntos infinitos, como no exemplo dos inteiros pares $Pares = \{x \in \mathbb{Z} | x \equiv 0 (\mod 2)\}$.

[1] Nas anotações presentes nesse documento, utiliza-se a "notação americana". Para a Equação (B.1) , teria-se $A = \{e_1, e_2, \ldots, e_n\}$.

Para o conjunto dos pares, ainda podemos utilizar uma descrição mais informal, mas que é dependente da conhecimento sobre a linguagem Portuguesa: *Pares* = $\{x \in \mathbb{Z} |$ *x é inteiro e par* $\}$.

Para denotar a cardinalidade (quantidade de elementos) de um conjunto, utilizamos o símbolo "|". Para os conjuntos apresentados acima, é correto afirmar que:

- $|N| = 4$;
- $|V| = 5$;
- $|R| = 3$;
- $|D| = 10$;
- $|Pares| = \infty$.

A cardinalidade pode ser utilizada para identificar quantos símbolos são necessários para representar um elemento. Por exemplo, $|12,66| = 5$

Para denotar conjuntos vazios, adota-se duas formas de representação: $\{\}$ ou $\emptyset$. Utilizando o operador de cardinalidade, têm-se $|\{\}| = |\emptyset| = 0$.

Como principais operações entre conjuntos, pode-se destacar:

- União ($\cup$): união de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{1,2,3,4,5,6,8\}$;
- Intersecção ($\cap$): intersecção de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{2,4\}$;
- Diferença ($-$ ou $\setminus$): diferença de dois conjuntos. Exemplo $\{1,2,3,4,5\} \setminus \{2,4,6,8\} = \{1,3,5\}$;
- Produto cartesiano ($\times$): Exemplo $\{1,2,3\} \times \{A,B\} = \{(1,A),(2,A),(3,A),(1,B),(2,B),(3,B)\}$;
- Conjunto de partes (ou *power set*): o conjunto de todos os subconjuntos dos elementos de um conjunto. Para o conjunto $A = \{1,2,3\}$ o conjunto das partes seria $2^A = P(A) = \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$.

Funções são representadas de forma diferente na matemática discreta. Busca-se estabelecer a relação entre um conjunto de domínio (entrada da função) e um contradomínio (resposta da função). A Equação (B.3) exibe a forma como é utilizada para formalizar uma função. Nesse formato, passa-se a natureza da entrada e da saída de um problema. Por exemplo, a função que gera a correspondência entre o domínio dos inteiros positivos em base decimal para base binária seria $f : x \in Z^+ \rightarrow \{0,1\}^{\log_2(|x|+1)}$.

$$\text{nome da funcao} : \text{dominio} \rightarrow \text{contradominio} \tag{B.3}$$

Para representar uma coleção de itens onde a sequência importa e a repetição pode ocorrer, utiliza-se as tuplas. Uma tupla é representada da forma demonstrada na Equação (B.4).

$$A = (e_1, e_2, \ldots, e_n) \tag{B.4}$$

. Um exemplo de uma tupla, pode ser lista de chamada de uma turma ordenada lexicograficamente.