UNIP

UNIVERSIDADE PAULISTA

Análise de Algoritmos

Autor: Prof. Roberto Eduardo Bruzetti Leminski
Colaboradoras: Profa. Vanessa Lessa
Profa. Larissa Rodrigues Damiani

Professor conteudista: Roberto Eduardo Bruzetti Leminski

Roberto E. B. Leminski é formado em Tecnologia em Materiais, Processos e Componentes Eletrônicos (MPCE) pela Faculdade de Tecnologia de São Paulo (Fatec) desde 1998, mestre em Engenharia Elétrica (ênfase em Microeletrônica) pela Escola Politécnica da Universidade de São Paulo desde 2001. É docente dos cursos de Ciência da Computação e Sistemas de Informação da Universidade Paulista desde 2007, com passagem por disciplinas dos cursos de Ciências Contábeis e Administração. Também é docente da UniFECAF desde 2023. É parceiro da empresa IBFC, na elaboração e revisão de questões para concurso público. Foi líder da disciplina Circuitos Digitais do curso de Ciência da Computação na UNIP.

Dados Internacionais de Catalogação na Publicação (CIP)

L554a Leminski, Roberto Eduardo Bruzetti.

Análise de Algoritmos / Roberto Eduardo Bruzetti Leminski. – São Paulo: Editora Sol, 2024.

144 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Algoritmos. 2. Função. 3. Multithread. I. Título.

CDU 519.67

U519.76 – 24

Sumário

Análise de Algoritmos

Unidade II

APRESENTAÇÃO

Esta disciplina fornece uma visão do conceito de complexidade computacional e algoritmos, e apresenta técnicas algorítmicas empregadas para resolver diferentes problemas práticos em diferentes áreas. Como esta disciplina aplica e analisa diferentes tipos de algoritmo, assume-se que o aluno conheça os conceitos de programação estruturada e lógica de programação, para compreender os problemas apresentados e as soluções empregadas em situações reais.

A disciplina também se utiliza de conhecimentos que envolvem as áreas de estrutura de dados e de inteligência artificial (IA), uma vez que versões de algoritmos vistos nessas disciplinas são discutidas e comparadas em diferentes situações.

INTRODUÇÃO

Algoritmo é geralmente definido como sequência finita e ordenada de passos para resolver um problema ou executar uma tarefa. Muitos autores reportam o primeiro algoritmo computacional a Ada Lovelace (1815-1852), que elaborou um conjunto de instruções para operar a máquina diferencial de Charles Babbage em meados do século 19.

Pode causar surpresa o fato de, quase dois séculos depois, ainda existirem problemas matemáticos ou de automação para os quais não há um algoritmo específico que os resolva. Na verdade existem problemas para os quais não se sabe nem se há solução algorítmica específica; nesses casos podemos usar um algoritmo genérico que, embora resolva o problema, não temos certeza se é a melhor solução.

Ao longo do livro-texto, vamos aprender a classificar e comparar algoritmos, além de estudar diferentes casos em que o uso das técnicas triviais de programação não seja suficiente. Também analisaremos soluções aplicadas e algoritmos específicos, tendo sempre como norte a pergunta "Esse é o melhor algoritmo possível?".

O material está dividido em oito tópicos, resumidos a seguir:

- **Tópico 1**: apresentaremos e discutiremos os conceitos de complexidade de tempo e espaço, que permitem comparar algoritmos diferentes de forma independente de um eventual computador em que estejam sendo executados. Também mostraremos a **análise assintótica**, que permite medir a complexidade de um algoritmo em função do número de entradas.

- **Tópico 2**: analisaremos os conceitos bem conhecidos de algoritmos iterativos e recursivos, comparando-os. Também introduziremos técnicas de programação dinâmica, que serão utilizadas nos tópicos seguintes.

- **Tópico 3**: estudaremos alguns algoritmos sobre árvores e, apresentando o primeiro problema prático específico, serão apresentados os heaps, uma alternativa para filas de prioridades.

- **Tópico 4**: abordaremos diferentes tipos de algoritmo sobre cadeia de dados, que são estruturas sequenciais de dados (vetores, listas etc.). Vamos estudar a complexidade das operações mais comuns envolvendo essas estruturas e analisar os principais algoritmos de ordenação. Também apresentaremos algoritmos para localizar subcadeias em cadeias maiores.

- **Tópico 5**: vamos analisar alguns algoritmos sobre grafos e algoritmos para estudar fluxo em redes e calcular o fluxo máximo em uma rede. Também apresentaremos a técnica de análise amortizada, usada quando a assintótica não fornece um bom resultado.

- **Tópico 6**: estudaremos algoritmos gulosos, amplamente empregados em IA como alternativa otimizada para resolver questões de força bruta. Também vamos conferir o código de Huffman, uma abordagem algorítmica que codifica e compacta dados em situações reais.

- **Tópico 7**: trataremos de algoritmos multithread, em que diferentes tarefas são executadas em paralelo e cuja otimização é requisito para a computação paralela.

- **Tópico 8**: aqui vamos aos limites do conhecimento referente às técnicas de desenvolvimento de algoritmos, mostrando problemas para os quais não sabemos se há um algoritmo específico e sugerindo recursos que possam resolvê-los.

Esse conjunto de tópicos abordará as situações mais comuns que envolvem análise de algoritmo, e diversos exemplos serão demonstrados na linguagem Python para ilustrar os assuntos tratados ao longo do livro-texto. A linguagem de programação foi particularmente escolhida por ser de alto nível (sendo bastante simples compreender seus códigos) e uma das linguagens mais utilizadas atualmente. Assim unificaremos e expandiremos o conhecimento adquirido em diferentes áreas da computação, mostrando que técnicas de elaboração de algoritmos são versáteis se bem aplicadas.

Unidade I

1 COMPLEXIDADE DE ALGORITMOS

Neste tópico vamos entender como é possível comparar diferentes algoritmos computacionais sob dois aspectos: tempo de execução e memória necessária para execução.

Quando vamos implementar um algoritmo na forma de um programa de computador, algumas perguntas surgem: Esse algoritmo é o melhor para resolver o problema? Existem outras opções de algoritmo melhor? Qual o tempo de execução desse algoritmo? Quanta memória será consumida por esse algoritmo em execução? O código escrito numa dada linguagem de programação é o melhor possível?

A última pergunta é a mais difícil de responder, pois determinadas linguagens de programação favorecem (ou dificultam) a implementação de diferentes algoritmos. Vamos tratar neste livro dos algoritmos em si, mostrando, quando necessário, versões do algoritmo em pseudocódigo ou Python. Assim, a questão da linguagem de programação não será tratada, e os algoritmos serão abordados com enfoque mais matemático.

Sobre o tempo de execução, é impossível dar uma resposta absoluta de quanto tempo um algoritmo consome: a execução do mesmo algoritmo, sobre a mesma base de dados, será diferente de computador para computador, pois será influenciada por inúmeros parâmetros além do controle e conhecimento de quem elaborou o algoritmo (Dasgupta; Papadimitriou; Vazirani, 2009, p. 5).

Uma abordagem que permite uma análise comparativa entre algoritmos é analisar quantas operações serão executadas em função do número de entradas que o algoritmo recebe. A seguir, um exemplo: consideremos os dois algoritmos apresentados na figura 1:

Versão 1	Versão 2
Se (condição 1) então: Instruções 1 Se (condição 2) então: Instruções 2 Se (condição 3) então: Instruções 3 Se (condição 4) então: Instruções 4	Se (condição 1) então: Instruções 1 Senão se (condição 2) então: Instruções 2 Senão se (condição 3) então: Instruções 3 Senão se (condição 4) então: Instruções 4

Figura 1 – Duas versões para um mesmo algoritmo com estruturas condicionais

Alguns pressupostos: primeiro, o tempo de execução de cada instrução é o mesmo; segundo, as quatro condições são mutuamente excludentes (se uma for verdadeira, as outras três automaticamente serão falsas) e equiprováveis (cada uma tem 25% de chance de ser verdadeira).

Na versão 1, todas as condições serão testadas em todas as execuções, independentemente de uma delas já ter sido verdadeira. Na versão 2, cada condição só será testada se todas as condições testadas anteriormente se mostrarem falsas. Assim, em mil execuções desse algoritmo, a versão 1 irá realizar exatamente 1000 testes de condições; já a versão 2 irá executar por volta de 250. A médio e longo prazo, a versão 2 do algoritmo precisará de menos tempo para realizar sucessivas execuções.

Dessa forma, podemos comparar dois algoritmos em como eles trabalham com uma mesma quantidade n de entradas. O algoritmo que cumprir menos etapas de processamento para esse conjunto de entradas será consequentemente mais rápido. O número de etapas terminadas em função do número de entradas que o algoritmo recebe é denominado **complexidade de tempo** do algoritmo; e o consumo de memória em função do número de entradas é denominado **complexidade de espaço**.

1.1 Redução assintótica

Representar a complexidade de um algoritmo com uma função do número de entradas pode resultar em uma função com diferentes parâmetros, sendo parte delas constante. Por exemplo, consideremos um algoritmo para calcular o fatorial de um número. Teremos as seguintes operações:

- entrada do número;
- n – 1 multiplicações (n é o número recebido);
- saída do resultado calculado.

Supondo que todas as operações consumam o mesmo tempo, teremos um tempo de execução $f(n) = 1 + (n - 1) + 1 = n + 1$.

Podemos simplificar: se o número de entradas for muito grande, os outros parâmetros tendem a se tornar insignificantes se comparados com ele. Dessa forma podemos reduzir nossa função ao elemento mais significativo, desprezando os demais parâmetros da função. Essa simplificação recebe o nome de **redução assintótica** (Dasgupta; Papadimitriou; Vazirani, 2009, p. 7). Com ela podemos reduzir a função para nosso algoritmo como $f(n) = n$.

Observação

Na matemática, o termo assintótico significa "para todos os valores suficientemente grandes" ou "tendendo ao infinito". Aqui vamos usá-lo para indicar um número muito grande de entradas em um algoritmo.

Assim, um algoritmo que execute $f(n) = 5x^4 + 3x^2 + 7$ operações para um número n de entradas pode ter sua função representada assintoticamente como $f(n) = x^4$.

1.2 Notação Grande-O

Utilizando a redução assintótica, a notação para indicar a complexidade de tempo ou de espaço de um algoritmo é a **notação O** (lê-se "grande O", ou "ozão", do inglês big O). Assim, o algoritmo do fatorial que usamos como exemplo terá complexidade descrita como O(n) (Grande O de n). A notação Grande-O permite comparar a ordem de grandeza das complexidades de um algoritmo com as de outro algoritmo. Porém, em alguns casos, obter uma função matemática objetiva para a complexidade de um algoritmo não é possível; para esses casos usa-se uma metodologia denominada **análise amortizada** (apresentada no tópico 5).

Essa notação será utilizada ao longo de todo o livro-texto para analisar os diferentes algoritmos apresentados. Consideremos agora dois algoritmos distintos para resolver o mesmo problema: o algoritmo 1 tem uma complexidade de tempo $f(n) = 2n^2 + 2$; já o algoritmo 2 tem uma complexidade de tempo $g(n) = 2n + 30$.

A tabela 1 apresenta as operações com cada algoritmo:

Tabela 1– Número de operações com cada algoritmo em função do número de entradas

Entradas (n)	Número de operações	
	Algoritmo 1	Algoritmo 2
1	4	32
2	10	34
3	20	36
4	34	38
5	52	40
6	74	42
10	202	50
50	5002	130
100	20002	230
200	80002	430

O algoritmo 1 (com função f(n)) é bem mais rápido (executa menos operações) para um número pequeno de entradas ($n < 5$), perdendo eficiência em relação ao algoritmo 2 (com função g(n)) quando n se torna muito grande.

A notação Grande-O pode relacionar dois algoritmos. Em suma, relacionar dois algoritmos com funções A1 e A2 na forma A1 = O(A2) significa dizer que **a função A1 não cresce mais rápido que a função A2, para n assintótico** (Cormen, 2013, p. 15). Dessa forma, para os dois algoritmos do exemplo, g = f(n), e isso significa que g(n) não cresce mais rápido que f(n). Por outro lado, f ≠ O(g), pois f(n) realmente cresce mais rápido que g(n), apesar dos resultados menores para valores muito pequenos de n.

Exemplo de aplicação

Consideremos um algoritmo para inserir um valor na sua posição correta em uma lista ordenada já existente, com n elementos. A primeira opção, mais fácil de implementar, consiste em comparar o valor a ser inserido com o primeiro elemento da lista, depois com o segundo, terceiro etc., até localizar a posição onde o novo valor deve ser inserido.

Para um número suficientemente grande de execuções, teremos uma quantidade média de comparações igual a f1(n) = n/2. Assim, podemos definir que a complexidade de tempo desse algoritmo será O(n); isso significa que será linear e diretamente proporcional ao tamanho da lista.

Uma segunda abordagem do problema seria fazer uma busca binária: dividimos a lista no meio, verificamos em qual metade o valor deve ser inserido, e então dividimos novamente essa metade em duas e repetimos o processo. A lista precisará ser dividida e realizar comparações uma quantidade de vezes a f2(n) = $\log_2$ n. Assim, podemos dizer (após a simplificação assintótica) que a complexidade da solução será O(log n) (para n suficientemente grande, podemos considerar o logaritmo 2 como da mesma ordem de grandeza que o logaritmo de 10).

Para comparar as diferenças, a tabela 2 ilustra o caso:

Tabela 2

Tamanho da lista (n)	Número médio de operações	
	f1(n) = n/2	f2(n) = $\log_2$ n
10	5	3,32
50	25	5,64
100	50	6,64
500	250	8,97
1000	500	9,97

Assim, f1(n) escala muito mais rápido que f2(n), ou seja, f2 = O(f1). Dessa forma, o algoritmo de busca binária tem complexidade de tempo menor, sendo mais eficiente que a comparação individual entre os valores.

Saiba mais

O exemplo do algoritmo para inserir um elemento numa lista está detalhado passo a passo no capítulo 3.3 deste livro:

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Rio de Janeiro: LTC, 2016. Disponível em: https://tinyurl.com/473hwnuj. Acesso em: 7 dez. 2023.

1.3 Notações Ômega e Theta

Como visto no tópico anterior, dizer que f = O(g) significa que a função f(n) tem um consumo de tempo ou de memória menor que a função g(n). Para representar a ideia oposta – ou seja, indicar que um algoritmo é mais complexo que outro –, usamos a notação Ômega. Em outras palavras, f = Ω(g) significa que a execução de f(n) realiza um número de operações (ou tem consumo de memória) maior que g(n).

No exemplo anterior, temos que f2 = O(f1) e f1 = Ω(f2). A notação Ômega, diferente da notação Grande-O, não indica a complexidade de um algoritmo individual, mas exclusivamente compara dois algoritmos (Cormen, 2013, p. 17). Podemos aprofundar a ideia dizendo que, quando f = Ω(g), a complexidade da função f será, no mínimo, a complexidade da função g.

De forma semelhante, podemos utilizar a notação Theta para indicar que dois algoritmos têm mesma ordem de complexidade: f1 = Θ(f2) significa que f1 = O(f2) e f1 = Ω(f2), simultaneamente (e vice-versa). A notação Theta também indica que a complexidade de um algoritmo será, no máximo, uma dada complexidade. Dessa forma, dizer que f = $\Theta(2^n)$, por exemplo, significa dizer que a função f terá no máximo complexidade exponencial.

Convém observar que dois algoritmos terem a mesma ordem de grandeza de complexidade não significa que eles sejam iguais. Por exemplo, um algoritmo com complexidade de espaço f1(n) = 3n consome o triplo de memória que um algoritmo com complexidade f2(n) = n, mas como ambos têm funções de mesma natureza (lineares), dizemos que f1 = Θ(f2) (Toscani; Veloso, 2012, p. 29).

1.4 Principais funções de complexidade

Algumas funções matemáticas são recorrentes se analisarmos a complexidade de algoritmos. A seguir, confira alguns casos de complexidade com exemplos de algoritmo (alguns serão detalhados em tópicos seguintes). Essa pequena lista não cobre todas as possíveis funções relacionadas à complexidade de algoritmos, apenas os casos mais comuns.

1.4.1 Função fatorial O(n!)

O fatorial de um número cresce muito rapidamente (para n = 10, temos n! maior que 3 milhões). Um algoritmo com complexidade O(n!), seja de tempo ou de espaço, é extremamente indesejado, pois escalona rápido demais. Sempre que um algoritmo envolver esse tipo de complexidade, tenta-se substituí-lo por outro ou adotar uma solução parcial.

Dentre os casos que caem nessa categoria, temos:

- algoritmos que envolvam gerar todas as permutações (ou combinações, arranjos etc.) sobre os elementos de um conjunto de dados – conforme o capítulo 4 de Gersting (2016);
- resolução por força bruta do problema do caixeiro-viajante (detalhado no tópico 8 deste livro-texto).

1.4.2 Função exponencial O(2^n)

Em complexidade de algoritmos, utilizamos uma função exponencial de base 2. Uma função exponencial cresce na forma de progressão geométrica: cada nova entrada dobra o número de operações (ou de consumo de memória) por parte do algoritmo. Também é um caso de complexidade muito ruim, não sendo viável para grandes valores de n.

Dentre seus exemplos, podemos citar:

- algoritmos de busca cega para encontrar trajetos em grafos;
- algoritmos de força bruta em geral.

Saiba mais

"A maior limitação da raça humana é a incapacidade de compreender o poder da função exponencial", disse dr. Albert Bartlett em sua famosa conferência de 1969, "Aritmética, população e energia". Pelo endereço a seguir, você pode assisti-la na íntegra (em inglês) ou ler sua transcrição (em inglês, francês e espanhol):

ARITHMETIC, population and energy – a talk by Al Bartlett. *Albartlett.org*, 11 jul. 2015. Disponível em: http://tinyurl.com/5aw63kjb. Acesso em: 13 dez. 2023.

Exemplo de aplicação

Lei de Moore é uma observação feita em 1965 por Gordon E. Moore – engenheiro e empresário, cofundador da Intel – de que a capacidade de processamento dos computadores dobraria a cada ano (em 1975 ele revisou esse valor para dois anos).

Tendo em vista que a Lei de Moore propõe um crescimento exponencial, é possível afirmar que ela ainda seja válida?

1.4.3 Função polinomial $O(n^x)$

Para esse tipo de complexidade, n é a base de uma potência, sendo x um expoente inteiro constante. Um valor de x igual a 2 (função quadrática) pode ser aceitável se o número n de entradas for pequeno ou moderado; funções polinomiais com expoentes maiores novamente se tornam inviáveis do ponto de vista de aplicações práticas. Porém, em alguns casos, reduzir uma complexidade maior (exponencial ou fatorial) a uma complexidade polinomial pode ser considerado melhoria (assunto tratado no tópico 8.3, "Exemplos de problemas NP-completos").

Alguns exemplos desse tipo de complexidade:

- multiplicação de matrizes de tamanho $n \times n$ tem complexidade de tempo cúbica $O(n^3)$ (Dasgupta; Papadimitriou; Vazirani, 2009, p. 97);
- soma de matrizes de tamanho $n \times n$ tem complexidade de tempo quadrática $O(n^2)$;
- diversos algoritmos de ordenação apresentam complexidade de tempo quadrática $O(n^2)$, como o Bubble Sort e o Selection Sort (o tópico 4, "Algoritmos sobre cadeias", analisa alguns dos principais algoritmos de ordenação em termos de complexidade computacional);
- algoritmos com um conjunto de estruturas condicionais encadeadas têm complexidade de tempo polinomial com expoente igual aos níveis de encadeamento (duas estruturas encadeadas terão complexidade quadrática, três terão complexidade cúbica e assim por diante).

1.4.4 Função n logaritmo de n O(n log n)

Em complexidade de algoritmo, essa função é chamada às vezes de subquadrática ou superlinear, porque é melhor do que $O(n^2)$, mas pior que $O(n)$. Normalmente é o limite até o qual se consegue diminuir algoritmos com função quadrática de complexidade. Para valores de n muito pequenos ($n < 10$), essa complexidade tende a ser melhor que a complexidade linear; mas assintoticamente acaba sendo maior.

Alguns exemplos dessa complexidade:

- algoritmos de busca binária em estruturas de dados ordenadas, conforme o exemplo do tópico 1.3, "Notação Grande-O");
- alguns algoritmos de ordenação, como o Merge Sort e o Quick Sort (este para os melhores casos), têm complexidade de tempo O(n log n);
- o código de Huffman, utilizado para comprimir dados (esse algoritmo será detalhado no tópico 6.4).

Lembrete

Muitos algoritmos de ordenação são influenciados pelo estado original dos dados e pelo número de operações a realizar. Por exemplo, Quick Sort e Insertion Sort são muito sensíveis às condições originais dos dados.

1.4.5 Função linear O(n)

Aqui começamos a ter algoritmos com uma complexidade aceitável para implementação computacional de grandes valores de n. Um algoritmo com complexidade O(n) terá seu número de operações (ou consumo de memória) diretamente proporcional ao número de entradas n (complexidade desejada para um algoritmo). A seguir, alguns exemplos:

- busca de dados em uma estrutura não ordenada, como uma árvore, vetor ou matriz (mais detalhes serão abordados nos tópicos 3 e 4);
- algoritmos com várias estruturas condicionais sequenciais, não aninhadas (lembremos que uma função do tipo f(n) = xn, sendo x um número natural, é assintoticamente simplificada para O(n));
- algoritmo de Knuth-Morris-Pratt (KMP) para buscar uma sequência de caracteres num conjunto maior de caracteres (esse algoritmo será detalhado no tópico 4);
- alguns algoritmos de ordenação, como Insertion Sort, quando atuam num conjunto de dados favorável.

1.4.6 Função logarítmica O(log n)

Essa função tem uma taxa de crescimento menor que a taxa de crescimento das entradas. Assim, um aumento significativo na quantidade de entradas não reflete num aumento de mesma escala no número de operações que o algoritmo realiza. Por exemplo, se multiplicarmos por 10 o número n de entradas, o número de operações apenas dobra.

Alguns exemplos desse tipo de complexidade:

- execução do algoritmo Quick Sort para ordenar uma base de dados já quase ordenada;
- construção e manuseio de um heap, estrutura em árvore que gere uma lista de prioridades (heaps serão detalhados no tópico 3);
- inserção ou busca de elementos em uma estrutura de dados ordenada.

1.4.7 Função constante O(1)

Essa complexidade indica que é fixo o número de operações ou consumo de memória, não sendo influenciado pelo número de entradas. Seria o caso desejado para todo algoritmo, mas na prática raramente algum algoritmo todo detém essa complexidade, apenas algumas operações em algoritmo maior. A complexidade O(1) é mais encontrada no consumo de memória (complexidade de espaço) do que na complexidade de tempo.

Alguns exemplos de complexidade constante:

- inserção ou remoção de um elemento numa pilha ou fila (o tamanho da estrutura não influencia a operação);
- consumo de memória (complexidade de espaço) de diversos algoritmos de ordenação, como Bubble Sort, Selection Sort e Insertion Sort;
- determinar uma propriedade simples de um valor numérico – por exemplo, se é par ou ímpar, ou se é múltiplo de um dado valor.

Para ilustrar melhor a diferença entre as complexidades, as funções apresentadas estão representadas graficamente. A figura 2 mostra os piores casos de complexidade: fatorial, exponencial e polinomial, para $O(n^3)$ e $O(n^2)$. Para valores muito pequenos de n, perceba que já temos um número de operações da ordem de centenas.

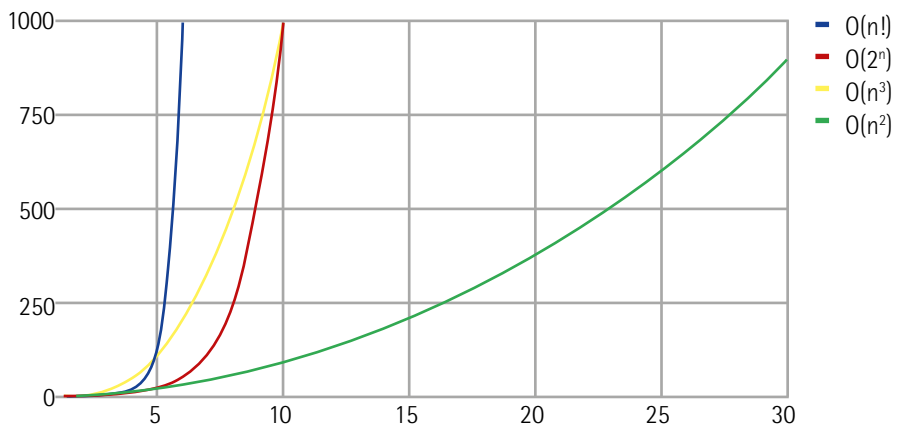


Figura 2 – Gráfico das funções $O(n!)$, $O(2^n)$, $O(n^3)$ e $O(n^2)$

Outros casos apresentam crescimento muito menor. A figura 3 mostra as funções (n log n) e linear. Assintoticamente, a complexidade O(n log n) se comporta de forma quase linear, embora seja maior que O(n) para n > 10.

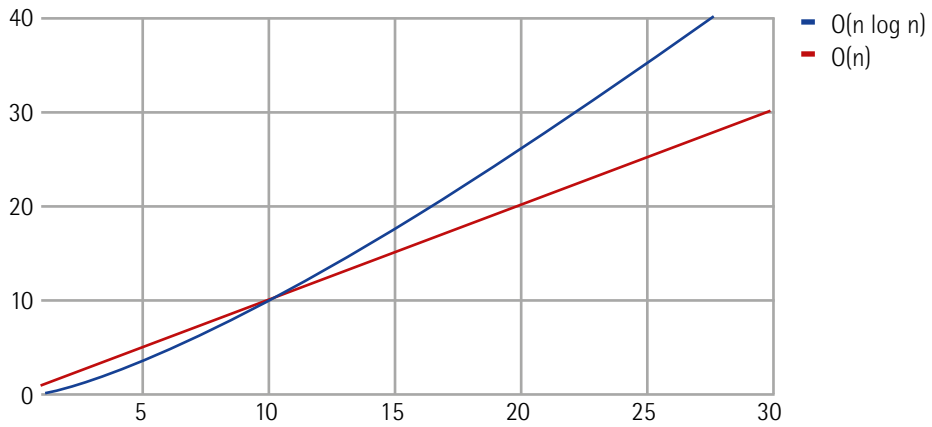


Figura 3 – Gráfico das funções O(n log n) e O(n)

Por fim, temos na figura 4 os dois melhores casos de algoritmo: a função logarítmica e a função constante. Na função logarítmica o crescimento é muito pequeno com a variação da quantidade de entradas; já na função constante o crescimento simplesmente não existe:

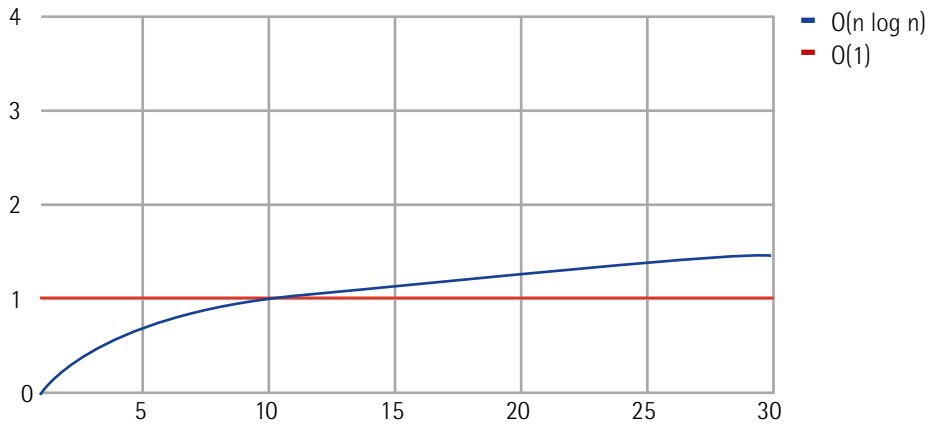


Figura 4 – Gráfico das funções O(log n) e O(1)

Esses conceitos serão aplicados ao longo de todo o livro-texto, e as funções apresentadas são as mais utilizadas para descrever as complexidades de tempo e espaço.

Observação

Na prática, algoritmos e programas completos são uma combinação de mais de uma função. Por exemplo: uma complexidade para execução e outra para entrada e saída de dados.

1.5 Exemplos

Neste livro-texto tentamos reduzir a complexidade dos algoritmos, apresentando novos algoritmos e abordagens como alternativas para resolver diferentes problemas, ainda que alguns deles possam, por sua própria natureza, ser mais complexos. Alguns casos:

Exemplos de aplicação

Exercício 1. Para uma função que construa todos os pares possíveis com os elementos de um conjunto, a função em Python a seguir constrói uma lista e armazena todos os pares possíveis com os elementos de uma lista L de tamanho n qualquer e devolve uma lista (denominada saida) com todos os pares compostos por dois elementos de L, incluindo repetições:

```
def pares (L):
    saida=[]
    for i in range(len(L)):
       for j in range (len(L)):
          saida.append([L[i],L[j]])
    return saida
```

Pela análise combinatória, temos um caso do **princípio fundamental da contagem**: temos n elementos possíveis para a primeira posição do par e o mesmo valor n para a segunda posição. Assim, serão produzidos $n \times n = n^2$ elementos. Dessa forma, tanto a complexidade de tempo para produzir esses pares quanto a complexidade de espaço para armazená-los serão polinomiais $O(n^2)$.

Mudando um pouco o problema, colocamos uma restrição: não pode haver repetição de elementos em um mesmo par, e pares com os mesmos elementos serão contados como apenas um (ou seja, o par A, B será igual ao par B, A).

Obviamente, o conjunto de pares produzidos por essa função será um subconjunto do produzido pela primeira. A forma mais eficiente de fazer isso é não produzir pares de elementos que não serão aceitos. Para isso, basta modificar o valor inicial do segundo comando for para ele se iniciar no elemento imediatamente seguinte ao primeiro (a modificação está destacada em vermelho):

```
def pares (L):
    saida=[]
    for i in range(len(L)):
       for j in range (i+1, len(L)):
          saida.append([L[i],L[j]])
    return saida
```

O número total de pares produzidos é a combinação de n elementos, p a p (no caso, p = 2). Embora a fórmula da combinação entre dois números envolva fatoriais, podemos diminuir a complexidade porque temos um valor fixo de elementos (um par) em cada componente da saída.

Assim, temos:

$$C_{n,2} = \frac{n!}{2! \times (n-2)!} = \frac{n \times (n-1) \times (n-2)!}{2! \times (n-2)!} = \frac{n \times (n-1)}{2} = \frac{n^2 - n}{2}$$

A complexidade desse algoritmo, tanto em termos de tempo quanto de espaço, será também polinomial $O(n^2/2)$.

Exercício 2. Construir um algoritmo que gere todos os possíveis subconjuntos com os elementos de um conjunto de n elementos distintos (incluindo o conjunto vazio). A função a seguir em Python executa esse algoritmo:

```
def subconjuntos(L):
    from itertools import combinations
    saida = []
    for i in range(len(L) + 1):
       for j in combinations(L, i):
          saida.append(list(j))
    return saida
```

Algumas explicações sobre a função: o conjunto original é apresentado na forma de uma lista L, que é o parâmetro de entrada. O módulo itertools do Python contém diversos blocos de construção para códigos iterativos; deste módulo usaremos a função combinations (S, N), que gera todos os conjuntos de tamanho N possíveis com os elementos de uma sequência (uma lista, no caso) S. Esses conjuntos serão concatenados na lista saida, que é o retorno da função.

A saída da função para a lista L = ["A", "B", "C", "D"] é dada a seguir:

```
[[], ['A'], ['B'], ['C'], ['D'], ['A', 'B'], ['A', 'C'], ['A', 'D'],
['B', 'C'], ['B', 'D'], ['C', 'D'], ['A', 'B', 'C'], ['A', 'B', 'D'],
['A', 'C', 'D'], ['B', 'C', 'D'], ['A', 'B', 'C', 'D']]
>>>
```

Figura 5

A forma mais simples de visualizar quantos subconjuntos são gerados (e calcular a complexidade do algoritmo) é pelos diagramas de Venn. A figura 6 mostra todos os subconjuntos para conjuntos de um a quatro elementos:

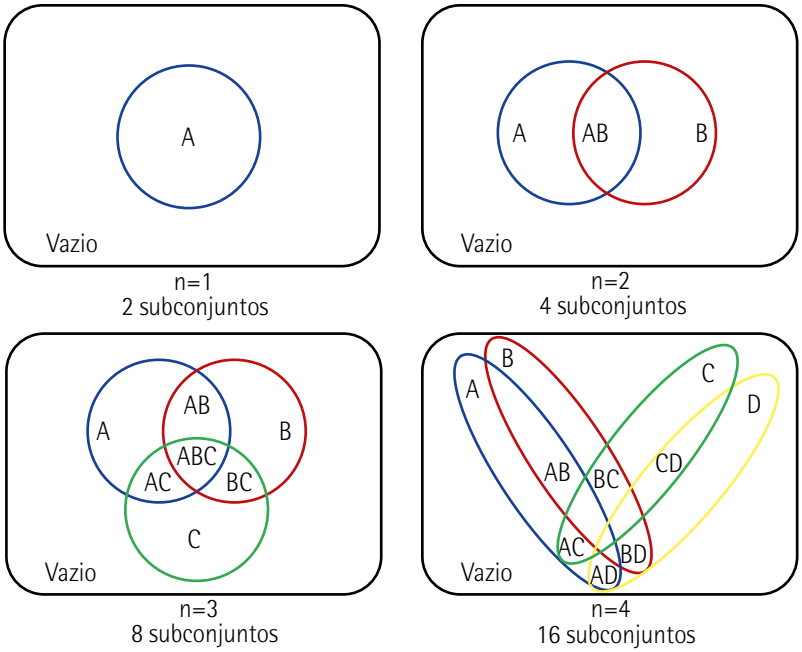


Figura 6 – Diagramas de Venn mostrando os subconjuntos para n = 1, 2, 3 e 4 elementos

É possível especular com facilidade que o número de subconjuntos é 2n, sendo n o número de elementos. Assim, o algoritmo tem complexidades de tempo e espaço exponenciais O(2n). Ou seja, a execução se torna custosa tanto em termos de tempo para produzir os pares quanto de consumo de memória: um conjunto com apenas 10 elementos produz 1024 subconjuntos, por exemplo.

Observação

Terá complexidade fatorial o algoritmo que combinar estes algoritmos do exemplo e gerar todos os demais possíveis arranjos entre elementos do conjunto.

Exemplos de aplicação

Exercício 1 (Cespe/Cebraspe 2022, DPE-RO, adaptado).

```
função algoritmo1()
0 : inicio
1 : se n = 0 então
2 :  retorne 0
3 : senão
4 :   se n = 1 então
5 :    retorne 1
6 :   senão
7 :    retorne algoritmo1 (n-1) + algoritmo1 (n-2)
8 :  fim se
9 : fim se
10 :fim
```

Figura 7

```
função algortimo2 (n)
0:  inicio
1:  se n = 0 então
2:   retorne 0
3:  senão
4:     se n = 1 então
5:     retorne 1
6:     senão
7:      penultimo = 0
8:      ultimo =1
9:      para i =2 até n faça
10:         atual = penultimo + ultimo
11:         penultimo = ultimo
12:         ultimo = atual
13:     fim para
14:     retorne atual
15:    fim se
16: fim se
17: fim
```

Figura 8

Quais são as complexidades de tempo de algoritmo1 e algoritmo2, respectivamente?

A primeira função é recursiva (mais detalhes no tópico 2), pois faz duas chamadas a si mesma na linha 7. Assim, cada execução dela resulta no seu dobro; por exemplo, cada aumento de apenas um no número de entradas n dobra o número de chamadas e execuções dela mesma. Temos aqui um exemplo de **complexidade de tempo exponencial $O(n^2)$**.

A segunda função executa uma instrução de repetição na linha 9, fazendo o conjunto de instruções se repetir n vezes, ou seja, diretamente proporcionais ao número de entradas. Fora dessa repetição, temos algumas instruções com tempo de execução fixo, que independem de n. Assim, pela redução assintótica temos uma função de **complexidade de tempo linear O(n)**.

Exercício 2 (FGV 2021, TJ-RO, adaptado). João precisa codificar uma função f(A), sendo A um array unidimensional de números inteiros que deve retornar o maior valor armazenado em A. Qual a complexidade de um algoritmo eficiente para a função f, para um array com n ($n \geq 1$)?

A questão não traz nenhuma observação sobre o estado dos números no vetor. Assim, se os números não estiverem em ordem (o pior caso possível), o algoritmo precisará percorrer toda a estrutura de dados em busca do maior valor. Nesse cenário, teremos que executar até n comparações (se o valor estiver na última posição do vetor). Assim, temos uma **complexidade de tempo linear O(n)**.

Exercício 3 (Sugep-UFRPE 2016, adaptado). Complexidade computacional é a área da ciência da computação que se ocupa, entre outros, de estudar e analisar o custo de tempo de execução e espaço ocupado pelos algoritmos.

Sobre complexidade computacional, indique se as afirmações a seguir são verdadeiras ou falsas:

I – A função de complexidade de tempo de algoritmo indica o tempo necessário para executar o programa que o implementa em função do tamanho da entrada.

Justificativa: complexidade de tempo é uma função que indica o tempo de execução do algoritmo de acordo com seu número de entradas, sendo independente de implementação ou da forma em que esteja (computacional ou não). Portanto, a afirmação é **falsa**.

II – Se f for uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior que f(n).

Justificativa: essa é a ideia por trás do conceito de complexidade de tempo; o tempo máximo, no pior caso, será indicado pela complexidade, sendo menor se as condições no problema favorecerem a execução do algoritmo. Portanto, a afirmação é **verdadeira**.

III – Na análise do caso médio, toma-se a média aritmética do pior caso com o melhor caso.

Justificativa: muitos algoritmos não têm complexidade de tempo linear. Assim, uma média aritmética simples vai dar um resultado incorreto de uma estimativa média nestes casos não lineares. Portanto, a afirmação é **falsa**.

2 ALGORITMOS RECURSIVOS E PROGRAMAÇÃO DINÂMICA

Neste tópico compararemos dois tipos de algoritmo: iterativos e recursivos. Também vamos analisar a metodologia denominada programação dinâmica, empregada para tentar obter a melhor solução algorítmica para um problema.

2.1 Iteração e recursão

Em muitas situações, mais de um algoritmo pode ser empregado para resolver determinado problema ou efetuar determinado cálculo, mas muitas vezes apresentam complexidades diferentes (de tempo ou de espaço) para chegar no resultado. Uma situação comum disso é o uso de algoritmos iterativos versus algoritmos recursivos. Algoritmo iterativo realiza um conjunto de iterações, ou seja, executa uma ou mais estruturas de repetições com o objetivo de gerar uma sequência de soluções que vão, a cada iteração, se aproximar da solução final desejada. Praticamente qualquer algoritmo que use uma estrutura de repetição (for, while etc.) é um algoritmo iterativo.

Já o conceito básico de algoritmo recursivo consiste em reduzir, sucessivamente, um problema em um problema menor ou mais simples, até que seja possível resolver alguma parte do problema reduzido de forma direta. Assim, retroativamente, esta solução vai sendo aplicada a cada uma das soluções

mais complexas até chegarmos à solução final. Sendo mais técnicos, podemos dizer que um algoritmo recursivo instancia a si mesmo diversas vezes para chegar até a solução final do problema (Cormen, 2013, p. 20).

Para ilustrar essa diferença, a figura 9 mostra duas funções em Python para calcular o fatorial de um número, uma iterativa e uma recursiva:

```
def fat_iterativo (N):
    fat = 1
    for i in range (1,N+1):
      fat*=i
    return fat
```

A)

```
def fat_recursivo (N):
if N <=1:
return 1
else:
return N * fat_iterativo (N-1)
```

B)

Figura 9 – Duas funções em Python para calcular o fatorial: A) função iterativa; B) função recursiva

A primeira e mais clara diferença é que a função iterativa tem estrutura de repetição (for), e a função recursiva tem apenas uma estrutura condicional (if); nesta a repetição se dá ao se instanciar sucessivamente. Ambas as funções conseguem a mesma quantidade de multiplicações, que será o valor com o qual se deseja calcular o fatorial menor 1; dessa forma, a complexidade de tempo para realizar as operações é a mesma para ambas, O(n).

No entanto, a função recursiva produz uma pilha de dados: o valor de n será armazenado enquanto se calcula o valor do fatorial de n–1; o valor de n–1 também precisará ser empilhado enquanto se calcula o fatorial de (n–1) –1, e assim sucessivamente, até chegar no valor do fatorial de 1. Dessa forma haverá um acréscimo de tempo de execução devido à inserção e remoção dos elementos na pilha de dados (as complexidades referentes a essa estrutura são detalhadas no tópico 4).

Outro ponto importante é que a função iterativa se utiliza, em termos de memória, de apenas uma variável, que atua como acumuladora onde todos os cálculos serão armazenados, sobrescrevendo o cálculo anterior. Já a função recursiva, como foi dito, cria uma estrutura de dados composta (pilha), aumentando o consumo de memória. Assim, a complexidade de espaço da função iterativa será O(1), enquanto a complexidade de espaço da função recursiva será O(n). Portanto, em termos de espaço, fat_iterativa = Ω(fat_recursiva).

De forma geral, podemos concluir que algoritmos recursivos, apesar de apresentarem um código menor (o que é vantajoso do ponto de vista de desenvolvimento), muitas vezes adquirem maiores complexidades, tanto de tempo quanto de espaço; todavia apresentam duas outras vantagens no contexto geral:

- Podem ser usados se o algoritmo for complexo demais na sua forma iterativa.
- Se o algoritmo que estiver sendo desenvolvido já exigir a construção de uma pilha, o aumento de complexidade é minimizado ou até mesmo inexistente. Por exemplo, o algoritmo de ordenação Quick Sort já envolve a construção de uma pilha; assim, uma função recursiva não afeta suas complexidades.

Lembrete

Todo algoritmo recursivo tem uma versão equivalente que é iterativa.

2.2 Métodos numéricos

Muitos métodos de cálculo numérico e ferramentas de simulação computacional usam algoritmos iterativos e recursivos para calcular suas respostas. Usualmente, parte-se de uma resposta inicial e, iterativamente, vai se recalculando para obter aproximações cada vez melhores da resposta final (Chapra; Canale, 2016, p. 65). Tais métodos numéricos são ditos convergentes, pois a cada iteração convergem em direção à resposta final (é claro, se o algoritmo estiver devidamente construído e se os parâmetros que o alimentam estiverem corretos).

Uma característica comum em algoritmos iterativos é o chamado **erro**. Diferente do uso comum da palavra, na matemática não significa algo incorreto, mas uma diferença entre o valor exato de algo e o valor aproximado utilizado num cálculo. Assim, o algoritmo irá iterar (repetir sua execução) até que o erro seja menor que um valor estipulado ou seja insignificante.

Métodos numéricos são um campo próprio da matemática e da ciência da computação, e foge ao propósito deste livro detalhar o assunto. Assim, segue uma pequena lista com alguns exemplos de métodos numéricos iterativos:

- algoritmos genéticos;
- métodos de busca heurística;
- método das diferenças finitas;
- método dos elementos finitos;
- método de Monte Carlo.

Saiba mais

Para se aprofundar em simulação computacional, leia este livro:

BRASIL, M. L. R. F.; BALTHAZAR, J. M.; GÓIS, W. *Métodos numéricos e computacionais na prática de engenharias e ciências*. São Paulo: Blucher, 2015. Disponível em: https://tinyurl.com/s6rubyny. Acesso em: 7 dez. 2023.

2.3 Divisão e conquista

Uma estratégia muita empregada para solucionar problemas matemáticos e de algoritmos é a chamada divisão e conquista, resumida em três etapas (Dasgupta; Papadimitriou; Vazirani, 2009, p. 56):

- divide-se o problema original em subproblemas;
- os subproblemas são resolvidos geralmente de forma recursiva;
- as respostas dos subproblemas são adequadamente combinadas para obter a resposta do problema original.

Observação

Podemos definir **instância** como qualquer elemento criado em algoritmo que herde alguma característica do problema original. Em orientação a objetos, o termo tem significado mais restrito.

Assim, um método de divisão e conquista executa três frentes de trabalho, coordenadas pela estrutura recursiva central do algoritmo: divisão do problema; resolução das partes; e concatenação das respostas parciais.

Do ponto de vista matemático, podemos estabelecer um padrão para a complexidade desses algoritmos: um problema de tamanho **n** é dividido em uma quantidade **a** de subproblemas de tamanho **n/b**, resolvidos recursivamente. Por fim, as respostas são combinadas em um tempo $O(n^d)$ para a, b, d > 0, e o tempo de execução será dado pela somatória das resoluções mais o tempo de recombinar as soluções parciais.

Assim:

$T(n) = aT(n/b) + O(n^d)$

Essa expressão pode ser simplificada assintoticamente para a programação dinâmica:

- $T(n) = O(n^d)$ se $d > \log_b a$
- $T(n) = O(n^d \log n)$ se $d = \log_b a$
- $T(n) = O(n^{\log_b a})$ se $d < \log_b a$

Essa formulação pode ser aplicada para determinar a complexidade de tempo da maioria dos algoritmos que usam essa metodologia.

Exemplos de aplicação

Exercício 1. O algoritmo mais famoso que se utiliza da metodologia de divisão e conquista é a busca binária em um arquivo ordenado com n registros. Primeiro dividimos o arquivo em duas partes de tamanho n/2, em seguida reaplicamos recursivamente o algoritmo na metade adequada.

Como a recursão ocorre apenas uma vez, temos a = 1 e b = 2, porque o arquivo é dividido em duas partes. A montagem das respostas não é necessária, portanto d = 0. Dessa forma, $d = \log_b a$ ($0 = \log_2 1$) e, consequentemente, a complexidade de tempo desse algoritmo é O(log n) (Dasgupta; Papadimitriou; Vazirani, 2009, p. 50).

Exercício 2. Construir uma função para determinar o máximo divisor comum (MDC) entre dois números naturais A e B (para facilitar a discussão, assumimos que $A \geq B$).

A solução iterativa consistiria em partir uma variável m se iniciando em B–1 e ir decrescendo em direção a 1, e testar cada valor de m até encontrar um que seja divisor de A e B simultaneamente. Teremos então um algoritmo iterativo, cujo número máximo de iterações será B–1.

A função MDC1, em Python, ilustra esse algoritmo:

```
def MDC1 (A,B):
    m = B-1
    while m >=1:
      if A%x == 0 and B%x == 0:
        return m
        break
      else:
        m-=1
```

A função tem complexidade linear igual a O(B).

Outra abordagem seria utilizar o algoritmo de Euclides para calcular o MDC baseando-se no princípio de que o MDC não se altera se o menor número for subtraído do maior. O funcionamento é o seguinte:

- Se A = 0, então MDC é B.
- Se B = 0, então MDC é A.
- Se A e B forem diferentes de 0, reaplicamos a função usando B e o resto da divisão de A por B como novas entradas.

Assim, a função recursiva MDC2 em Python mostra a implementação desse algoritmo:

```
def MDC (A,B):
    if A == 0:
      return B
    elif B == 0:
      return A
    else:
      return MDC (B, A%B)
```

Como a recursão ocorre apenas uma vez, temos a = 1 e b = A/B, porque o arquivo é dividido em duas partes. A montagem das respostas não é necessária, portanto d = 0. Como logaA/B (1) = 0, para qualquer valor de A/B temos que esse algoritmo também tem complexidade de tempo logarítmica O(B).

O algoritmo de Euclides apresentado no exemplo tem muitas aplicações práticas; uma delas é ser componente fundamental dos algoritmos de criptografia de chave pública RSA.

Exercício 3 (Enade 2017, adaptado). Na matemática, um produtório é definido como:

$$\prod_{i=m}^{n} X_i = X_m \times X_{m+1} \times X_{m+2} \times \cdots \times X_{n-1} \times X_n$$

Com base nessa equação, e considerando que Xi = i + 1/i, com i > 0, faça o que se pede:

a) Escreva uma função iterativa, em linguagem Python, que receba os parâmetros de limite inferior m e de limite superior n. Calcule e retorne o resultado do produtório.

Resposta: uma função iterativa irá sequencialmente realizar as multiplicações, armazenando-as em uma variável (no caso, a variável result, que é iniciada com o valor 1, elemento neutro da multiplicação). O retorno da função será essa variável na qual a sequência de multiplicações foi acumulada:

```
def iterativa (m,n):
    result = 1
    for i in range (m,n+1):
      print(i+(1/i))
      result *= i+(1/i)
    return result
```

b) Escreva uma função recursiva, em linguagem Python, que receba os parâmetros de limite inferior m e de limite superior n. Calcule e retorne o resultado do produtório.

Resposta: uma função recursiva chama a si mesma. O resultado será o produto do valor atual (começamos com o intervalo superior n) multiplicado pelo produto dos elementos anteriores. No caso, chamamos novamente, de forma recursiva, a função para executar o cálculo de n–1:

```
def recursiva (m,n):
    i = n
    if i == m:
      return (i+(1/i))
    else:
    return (i+(1/i))*recursiva(m,i-1)
```

Exercício 4 (Enade 2017, adaptado). Considere a função recursiva F a seguir, que em sua execução chama a função G:

```
1        void F (int n) {
2          if (n > 0) {
3            for (int i = 0; i < n; i++) {
4              G(i);
5            }
6          F(n/2);
7        }
8      }
```

Com base nos conceitos de teoria da complexidade, avalie as informações a seguir:

I – A equação de recorrência que define a complexidade da função F é a mesma do algoritmo clássico de ordenação Merge Sort.

Resposta: a função recursiva F será chamada novamente para N/2 (conforme a linha 6 do código). Assim, por exemplo, se N = 20, a função será executada para 20, 10, 5, 2 e 1. Ou seja, a função será chamada uma quantidade de vezes igual ao logaritmo de 2, mais um, de n (assintoticamente, isso equivale a F = O(log n)).

O algoritmo de ordenação Merge Sort (que será tratado no tópico 4.2) tem complexidade bem conhecida de O(n log n). Como a função F não tem a mesma complexidade, a afirmativa é **falsa**.

II – O número de chamadas recursivas da função F é Θ(log n).

Resposta: pela mesma explicação da afirmação I, esta também é **verdadeira**.

III – O número de vezes que a função G da linha 4 é chamada é O(n log n).

Resposta: a função G é chamada dentro de uma estrutura de repetição (na linha 3 do código), que se repete n vezes. Assim, a cada execução de F, G será executada n vezes. Dessa forma, G = n O(F) = O(n log n). Portanto a afirmação também é **verdadeira**.

2.4 Programação dinâmica

Recebe o nome **programação dinâmica** uma metodologia para construir algoritmos cujo foco é encontrar, sempre que possível, a melhor solução (com menos complexidade de tempo, normalmente) para situações que envolvem diferentes combinações possíveis de um conjunto. Simplificando, se uma série de operações precisa ser feita, qual seria a melhor ordem para realizá-las?

Existem duas abordagens na programação dinâmica: **top-down** e **bottom-up**. A primeira, como diz o nome, parte de cima para baixo, realizando um processo recursivo (a função recursiva para o cálculo do fatorial já apresentada é um exemplo dessa abordagem). Na segunda (de baixo para cima) o problema é dividido em subproblemas, que vão sendo calculados dos menores para o maior, aumentando-se a dificuldade a cada iteração. No caso, sabemos que cada iteração anterior já foi resolvida, logo não precisamos ficar armazenando resultados parciais toda vez, como na abordagem top-down (a função iterativa apresentada para calcular o fatorial é um exemplo dessa abordagem).

2.5 Aplicações

Um exemplo clássico de programação dinâmica pela abordagem bottom-up é o problema da multiplicação sequencial de cadeias de matrizes (ou *matrix chain ordering problem*, muitas vezes abreviado como MCOP). É um problema de otimização no qual se busca determinar a melhor forma de realizar o produto de uma dada sequência de matrizes. O problema aqui não é a multiplicação em si, mas determinar a melhor sequência em que a operação deve ser realizada.

Observação

Em teoria de algoritmos, podemos dividir os problemas em três grandes categorias: **otimização**, **busca** e **tomada de decisão**. Isso será discutido com mais profundidade no tópico 8.

Vamos detalhar o problema: a multiplicação de matrizes é associativa, pois não importa como o produto esteja entre parênteses, o resultado obtido permanecerá. Por exemplo, para quatro matrizes A, B, C, e D, teríamos cinco opções:

((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))

Lembrete

O produto de matrizes não é comutativo: A × B pode ser diferente de B × A, sendo possível inclusive que um dos produtos sequer seja realizado.

A ordem dos produtos (ou seja, como são colocados os parênteses) afeta o número de operações aritméticas simples necessárias para calculá-los.

Exemplo de aplicação

Consideremos as seguintes matrizes:

- $A_{10\times5}$
- $B_{5\times20}$
- $C_{20\times30}$
- $D_{30\times40}$

O produto de duas matrizes $X_{M\times N}$ e $Y_{N\times P}$ irá realizar $(M \times N \times P)$ operações de multiplicação, resultando em uma matriz de ordem $M \times P$.

Vamos testar as cinco possibilidades e verificar em qual ordem é efetuado o menor número de multiplicações:

1) ((AB)C)D

$(10 \times 5 \times 20) + (10 \times 20 \times 30) + (10 \times 30 \times 40) = 19$ mil operações

2) ((A(BC))D)

$(5 \times 20 \times 30) + (10 \times 5 \times 30) + (10 \times 30 \times 40) = 16{,}5$ mil operações

3) (AB)(CD)

$(10 \times 5 \times 20) + (20 \times 30 \times 40) + (10 \times 20 \times 40) = 33$ mil operações

4) A((BC)D)

$(5 \times 20 \times 30) + (5 \times 30 \times 40) + (10 \times 5 \times 40) = 11$ mil operações

5) A(B(CD))

$(20 \times 30 \times 40) + (5 \times 20 \times 40) + (10 \times 5 \times 40) = 30$ mil operações

O exemplo aponta que uma multiplicação na sequência mais otimizada (sequência 4) irá realizar três vezes menos operações matemáticas do que na pior sequência (a sequência 3). Reforçamos que isso serve para as matrizes utilizadas no exemplo; para matrizes diferentes, a sequência otimizada de execução pode ser outra.

A quantidade de combinações cresce rapidamente conforme aumenta o número de matrizes; assim, tentar todos os arranjos possíveis de parênteses teria um tempo de execução $O(2^n)$ que começa a ficar impraticável para grandes valores de n. Uma solução mais rápida para o problema é dividi-lo num conjunto de problemas menores.

Se desejamos apenas descobrir o número mínimo de operações a executar, vamos partir do ponto de que só existe uma maneira de multiplicar um par de matrizes; assim o custo mínimo (número mínimo de operações aritméticas) para multiplicar esse par não pode ser outro. Um algoritmo recursivo para isso seria o seguinte:

- Dividir a sequência de matrizes e separá-la em duas subsequências.
- Redividir novamente essa subsequência até chegar num par de matrizes (aqui ocorre a recursão).
- Encontrar o custo mínimo de multiplicar para esse par.
- Adicionar esse custo ao de multiplicar as duas matrizes resultantes.
- Repetir isso a cada divisão possível – na qual a sequência de matrizes pode ser dividida – e registrar o mínimo de todas elas.

Podemos aproximar assintoticamente o número de subsequências possíveis para $n^2/2$ (sendo n o número de matrizes), o que já é melhor do que a complexidade exponencial original. Assim, a complexidade de tempo desse algoritmo é polinomial, entre $O(n^3)$ e $O(n^2)$, dependendo das matrizes.

O código em Python a seguir faz o cálculo:

```
#função que executa o cálculo
def MultCadeia(seq, i, j):

    import sys
    if j <= i + 1:
      return 0
    min = sys.maxsize

    for k in range(i + 1, j):
      cost = MultCadeia(seq, i, k)
      cost += MultCadeia(seq, k, j)
      cost += seq[i] * seq[k] * seq[j]
      if cost < min:
        min = cost
    return min

# inserção da sequência de matrizes
qtde = 0
seq = []
while qtde <= 1:
    qtde = int(input("Digite a quantidade de matrizes: "))
```

```
    if qtde <= 1:
      print("Mínimo de duas matrizes!")
    else:
      seq.append(int(input("Linhas da Matriz 1: ")))
      for i in range (2,qtde+1):
           seq.append(int(input("Colunas da Matriz {0}/Linhas da Matriz {1}:
".format(i-1,i))))
      seq.append(int(input("Colunas da Matriz {0}: ".format(qtde))))

#execução e saída
print('O custo mínimo é', MultCadeia(seq, 0, len(seq) - 1))
```

A função MultCadeia é recursiva e composta de duas partes:

- O primeiro trecho (vermelho) trava a execução caso a divisão da subsequência não seja possível (se a subsequência for menor que três valores, o que representa o produto de duas matrizes). Além disso, pelo método sys.maxsize(), o trecho atribui o maior valor inteiro possível para o número inicial de multiplicações, sendo esse valor substituído pelo número calculado.

- A segunda parte (azul) divide iterativamente a sequência e, recursivamente, chama a si mesma para dividir cada subsequência até chegar em um par que possa ser multiplicado.

O restante do código lê a sequência de matrizes e chama a função para calcular a sequência inserida.

A figura 10 mostra a execução desse código na sequência de quatro matrizes do exemplo anterior, confirmando os cálculos apresentados:

```
Digite a quantidade de
matrizes: 4
Linhas da matriz 1: 10
Colunas da matriz 1/ linhas
da matriz 2: 5
Colunas da matriz 2/ linhas
da matriz 3: 20
Colunas da matriz 3/ linhas
da matriz 4: 30
Colunas da matriz 4: 40
[10, 5, 20, 30, 40]
O custo mínimo é 11.000
>>>
```

Figura 10 – Código em Python para calcular o MCOP

Exemplo de aplicação

Embora o código apresentado mostre o valor do custo mínimo, ele não apresenta a sequência de multiplicações que gera esse custo. De que forma o código poderia ser modificado para também apresentar a sequência de custo mínimo?

Algoritmo recursivo pode ser uma solução mais rápida, como visto no exemplo do MCOP, no entanto algumas vezes um algoritmo iterativo pode representar uma solução mais eficiente. Por exemplo, vejamos uma função, adaptada de Feofiloff (2020), para calcular o n-ésimo elemento da função de Fibonacci.

Observação

Sequência de Fibonacci é uma sequência infinita cujo primeiro elemento é 1, segundo elemento 1, e cada elemento a partir do terceiro será a soma dos dois elementos imediatamente anteriores. Assim, os primeiros dez elementos são: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

A primeira função em Python é recursiva:

```
def fib_recursiva (n):
    if n <=1:
      return n
    else:
      return fib_recursiva(n-1) + fib_recursiva(n-2)
```

Para calcular cada elemento, ela recursivamente calcula os dois elementos anteriores, também recursivamente, em uma abordagem bottom-up. O problema com essa função é que não existe um aproveitamento dos cálculos executados anteriormente: para cada valor da sequência, os valores anteriores devem ser calculados novamente. Dessa forma, a função tem uma complexidade de tempo exponencial $O(2^n)$.

Agora vamos analisar outra função, dessa vez iterativa, para o mesmo cálculo. Essa função usa uma lista f, onde os valores da sequência são armazenados tão logo calculados:

```
def fib_iterativa (n):
    f=[0,1]
    for i in range (2, n+1):
      f.append(f[i-1]+f[i-2])
    return f[n]
```

Embora também tenhamos uma abordagem bottom-up – pois para calcular cada valor precisamos dos valores anteriores –, aqui não é preciso recalcular os elementos anteriores, já que estão armazenados. Assim, a função terá complexidade de tempo diretamente proporcional ao valor de n que se deseja calcular, ou seja, O(n).

3 ÁRVORES E HEAPS

Neste tópico vamos analisar alguns algoritmos que atuam sobre árvores, com destaque para a estrutura de dados denominada heap, que é uma alternativa para listas onde existe uma prioridade. Também será mostrada uma variação mais avançada denominada heap de Fibonacci.

3.1 Árvores de dados

Em estruturas de dados, árvores são uma estrutura não linear que armazena dados de forma hierárquica e sequencial (Goodrich; Tamassia, 2013, p. 284). O primeiro elemento, também chamado de nó, é denominado raiz, sendo o único elemento da estrutura que não possui nó anterior. O nó imediatamente anterior a outro nó é chamado de nó-pai, e quaisquer nós que sejam imediatamente subsequentes a ele são denominados de nós-filhos. Cada nó (exceto a raiz) terá apenas um nó-pai, mas pode ter qualquer quantidade de nós-filhos.

Apesar de denominarmos o primeiro elemento de raiz, ao representarmos graficamente essa estrutura, o colocamos na parte superior, com as relações pais-filhos representadas por uma linha (aresta). Como qualquer estrutura de dados, uma árvore é finita, ou seja, haverá elementos sem filhos na estrutura: esses nós terminais são denominados folhas (em oposição à raiz, na analogia).

A figura 11 ilustra uma árvore de dados: os círculos representam os nós, e as linhas, as arestas. A raiz é o nó com valor 100, enquanto as folhas são os nós com valores 28, 10, 27, 13 e 73.

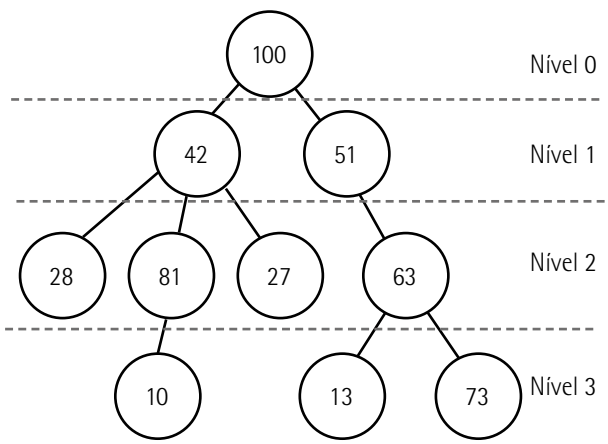


Figura 11 – Estrutura de dados em árvore

Uma árvore é dividida em níveis: o primeiro (nível zero) contém apenas a raiz; o segundo (nível 1) contém os filhos da raiz; o nível 2 contém os filhos dos filhos da raiz, e assim por diante. Perceba que pode haver folha em qualquer nível da árvore. Dentre essas estruturas, existe um subtipo denominado **árvore binária**, que apresenta como único diferencial o fato de cada nó poder ter, no máximo, dois nós-filhos (a árvore da figura 11, por exemplo, não é binária, pois o nó de valor 42 tem três nós-filhos).

Existem várias formas de representar uma árvore no Python; a mais fácil é pelo método Node, do módulo anytree, que define uma variável como um nó de árvore pela especificação de seu valor e de seu nó-pai (este parâmetro é opcional), pois a raiz da árvore não terá definido um pai. Assim, a árvore da figura 11 seria representada no Python conforme o código da figura 12 (com quebras de linha entre os níveis para facilitar a visualização). Construir árvores é uma operação de complexidade O(n) tanto do ponto de vista de tempo quanto de espaço.

```
from anytree import Node

raiz = Node(100)

nivel_1_filho_1 = Node(42, parent=raiz)
nivel_1_filho_2 = Node(51, parent=raiz)

nivel_2_filho_1 = Node(28, parent=nivel_1_filho_1)
nivel_2_filho_2 = Node(81, parent=nivel_1_filho_1)
nivel_2_filho_3 = Node(27, parent=nivel_1_filho_1)
nivel_2_filho_4 = Node(63, parent=nivel_1_filho_2)

nivel_3_filho_1 = Node(10, parent=nivel_2_filho_2)
nivel_3_filho_2 = Node(13, parent=nivel_2_filho_4)
nivel_3_filho_3 = Node(73, parent=nivel_2_filho_4)
```

Figura 12– Codificação em Python da árvore da figura 11

Observação

O módulo anytree não é nativo da maioria das IDEs de Python, portanto provavelmente será necessário instalá-lo pelo pip install.

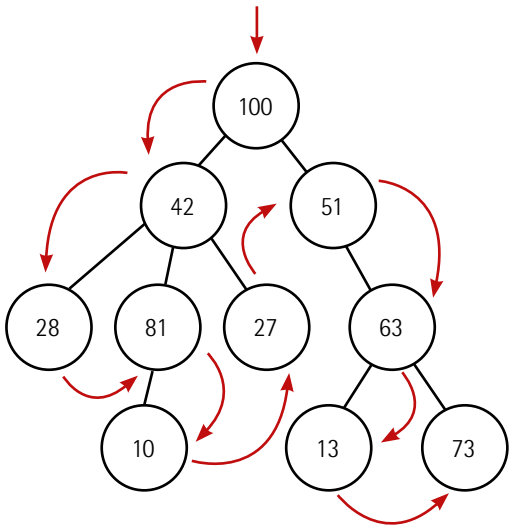
3.2 Algoritmos de busca em árvores

Existem dois algoritmos principais de busca não heurística em árvores: busca em profundidade e busca em largura. A primeira, através de uma busca não orientada, percorre a árvore da raiz e se aprofunda até chegar a uma folha, então o algoritmo retrocede até encontrar uma nova ramificação e continua a busca. A função recursiva em Python a seguir executa a busca em profundidade numa árvore construída pelo método Node descrito anteriormente:

```
from anytree import Node

def busca_profundidade (inicio,alvo):
    if inicio.name == alvo:
      print("Nó alvo encontrado:", inicio.name)
      return
    print(inicio.name)
    for no in inicio.children:
      busca_profundidade(no,alvo)
```

O método children retorna os filhos de determinado nó da estrutura; o método name verifica o valor contido em um nó da árvore (ambos os métodos estão contidos no módulo anytree); e a variável-alvo, que é parâmetro da função, recebe o número buscado. Na árvore da figura 11, o trajeto é o indicado a seguir, na figura 13 (no caso, se supõe que seguirá até o alvo ser localizado ou, caso não exista, até o término da árvore):



Trajeto: 100 42 28 81 10 27 51 63 13 73

Figura 13 – Busca em profundidade na árvore do exemplo anterior

Em termos de complexidade de tempo, como o algoritmo retrocede ao longo da estrutura, podemos (e precisamos) passar mais de uma vez por vários nós. Sendo n o número de nós e E o número de arestas, a complexidade de tempo pode ser definida como O(n + E). Assim como nessa estrutura, com um número suficiente de nós, podemos aproximar assintoticamente n = E, e a complexidade será linear O(2n), porém o uso desse algoritmo de busca implica criar uma lista que registrará os nós já visitados. Essa lista dependerá do número de ramificações presentes na árvore, mas seu tamanho máximo tenderá, no pior caso, ao número de nós presentes na estrutura. Dessa forma, a complexidade de espaço do algoritmo é O(n) (Dasgupta; Papadimitriou; Vazirani, 2009, p. 86).

A busca em largura também realiza uma busca não orientada que atravessa a árvore, porém ela é percorrida por níveis: primeiro o nível 0, depois todos os nós do nível 1, depois do nível 2, e assim por diante, até chegar aos nós do último nível. A função em Python a seguir executa a busca em largura numa árvore construída novamente com o método Node:

```
from anytree import Node, LevelOrderIter

def busca_largura (raiz, alvo):
    for no in LevelOrderIter(raiz):
      if no.name == alvo:
        print("Nó alvo encontrado:", no.name)
        return
        print(no.name)
```

O método LevelOrderIter executa iterações em uma árvore percorrendo-a por uma estratégia em níveis (os demais métodos presentes na função já foram explicados). Na figura 14, a busca em profundidade é mostrada para a árvore utilizada nos exemplos anteriores; o trajeto é o indicado a seguir (novamente, supondo que o algoritmo seguirá até o alvo ser localizado ou até o término da árvore):

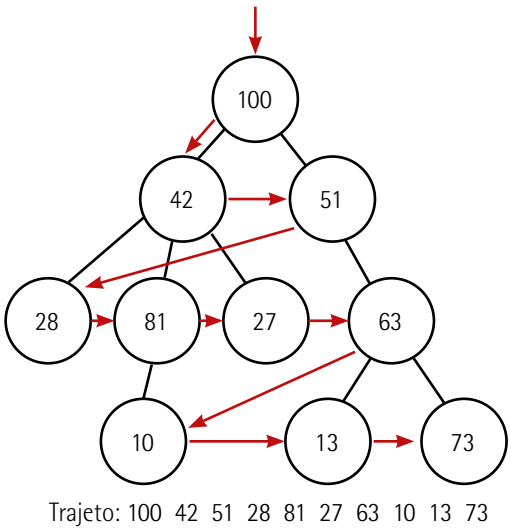


Figura 14 – Busca em largura na árvore do exemplo anterior

Em termos de complexidade de tempo, não temos aqui um retrocesso na estrutura como na busca em profundidade; assim, cada nó será percorrido apenas uma vez, e a complexidade de tempo será O(n). Porém o uso desse algoritmo de busca implica criar uma fila que irá definir a ordem em que os nós serão visitados; essa lista conterá todos os nós presentes na estrutura. Dessa forma, a complexidade de espaço do algoritmo é O(n) (Dasgupta; Papadimitriou; Vazirani, 2009, p. 106).

Esses algoritmos de busca são muitas vezes utilizados sobre grafos, que são tratados como uma árvore cuja raiz é um nó do qual se parte. Nesse tipo de aplicação, a complexidade dos algoritmos tende a aumentar, pois é possível o grafo se tornar uma árvore com infinitos níveis (isso será aprofundado no tópico 5, quando tratarmos de algoritmos sobre grafos).

3.3 Árvores binárias de busca

Árvore binária de busca é uma estrutura de dados na forma de uma árvore binária em que o posicionamento de seus elementos se baseia nas seguintes regras:

- O filho à esquerda de um nó sempre será menor que seu pai.
- O filho à direita de um nó sempre será maior que seu pai.

Uma árvore binária de busca não necessita ser completa nem balanceada. O objetivo da construção dessa estrutura é facilitar a busca binária em um conjunto de dados.

Busca binária é um algoritmo que encontra um elemento numa estrutura de dados ordenada dividindo repetidamente a estrutura pela metade (daí o nome binária) até chegar a uma única posição que contenha o elemento buscado (ou que deveria conter, no caso de o elemento não existir).

A figura 15 ilustra uma árvore binária de busca:

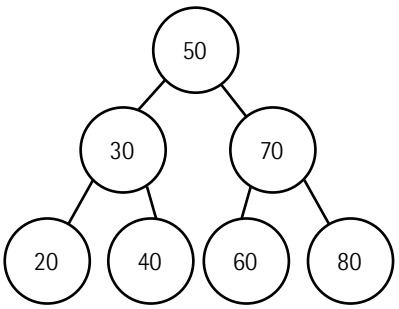


Figura 15 – Árvore binária de busca

Pela definição apresentada, uma árvore binária de busca pode ou não ser balanceada. Não é difícil visualizar outros arranjos dos nós apresentados na imagem anterior que respeitem as duas condições da árvore binária de busca.

Lembrete

Uma árvore é balanceada quando a diferença entre a maior e a menor distância das folhas até a raiz for no máximo 1.

Dada essa característica, a complexidade de tempo de execução pode variar de O(n), no caso de uma árvore de um único ramo, até O(log n) para uma árvore perfeitamente balanceada. Assim, quando utilizarmos essa estrutura, é interessante que a inserção e a remoção dos elementos considerem o balanceamento da árvore.

A função recursiva em Python a seguir executa a busca binária em uma árvore de busca binária construída com os métodos do módulo anytree (como na figura 12):

```
def busca_binaria(raiz, chave):
    print(raiz)
    #Realiza a verificação do nó atual
    if raiz is None or raiz.name == chave:
      return raiz
    #Realiza a recursão se o valor ainda não foi encontrado
    if chave < raiz.name and raiz.children:
      return busca_binaria(raiz.children[0], chave)
    elif chave > raiz.name and raiz.children:
      return busca_binaria(raiz.children[-1], chave)
    else:
      return None
```

Os parâmetros de entrada da função são a raiz da árvore e o valor buscado. Os retornos são o valor buscado (se encontrado) ou None, caso ele não exista na estrutura.

Observação

A função assume como árvore uma árvore binária de busca; se não for, a função não funcionará adequadamente e poderá não encontrar o valor buscado.

3.4 Fila de prioridades

Além de filas e pilhas, temos outra situação no uso de listas ao trabalharmos com estruturas de dados: as chamadas **filas de prioridade**. Nessas estruturas cada elemento tem uma prioridade e, ao retirarmos um elemento da estrutura, deve-se sempre retirar primeiro o elemento de maior prioridade.

Se pensarmos na situação como uma lista comum, podemos lidar com ela de duas formas (Cormen, 2013, p. 82):

- A lista é previamente ordenada, e cada elemento será inserido na posição correta. No caso, temos uma complexidade linear de tempo de O(n) para inserir cada elemento na lista e uma complexidade constante de tempo O(1) para remover o elemento de maior prioridade (sempre em uma das extremidades da lista).

- A lista não é ordenada, e cada elemento é inserido em uma das extremidades, como em uma fila ou pilha. No caso, teremos uma complexidade de tempo constante O(1) para inserir um elemento na estrutura e uma complexidade de tempo linear O(n) para localizar e extrair o elemento de maior prioridade.

3.5 Heaps

Uma solução para isso é a estrutura chamada heap (do inglês "amontoar"), que reflete a natureza dessa estrutura de dados: os elementos são "amontoados" em uma árvore binária, com os elementos de maior prioridade (primeiros a retirar) mais próximos à raiz. Para efeito dos algoritmos apresentados, consideraremos que o valor do nó é a prioridade correspondente daquele elemento.

Essa estrutura possibilita inserir novos elementos e extrair o de maior prioridade com complexidade de tempo O(log n), sendo mais eficiente do que as estruturas em listas descritas anteriormente. Como o próximo elemento a sair estará sempre na raiz, pode-se verificar, com complexidade de tempo constante O(1), a maior prioridade em determinado momento.

Como foi dito, heap é uma árvore binária completa, mas também tem duas outras características necessárias:

- O valor de determinado nó da árvore sempre será maior ou igual aos valores de seus dois filhos.
- Heap é uma árvore binária completa ou quase completa da esquerda para a direita, o que significa que existirá uma ordem específica para a inserção dos elementos na estrutura: sempre serão inseridos filhos no nó mais à esquerda possível, só depois passando para o próximo nó imediatamente à direita.

Observação

Árvore binária completa ou quase completa significa que um novo nível só é criado quando não há mais espaço nos níveis já existentes; apenas o último nível pode estar incompleto.

Assim, as duas árvores binárias da figura 16 não são heaps. A imagem da esquerda não é porque existem nós com valores maiores que seu nó-pai (o nó de valor 85 é filho do nó de valor 75, por exemplo). A imagem da direita não é porque um filho foi inserido na posição incorreta, fora da sequência da esquerda para a direita (o nó de valor 10 deveria ser filho do nó 60).

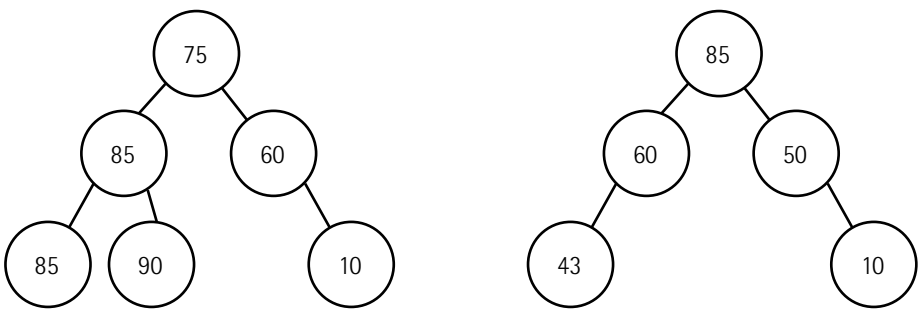


Figura 16 – Árvores binárias que não são heaps

Já as árvores da figura 17 são heaps, pois satisfazem as duas condições necessárias para uma árvore binária ser esse tipo de estrutura.

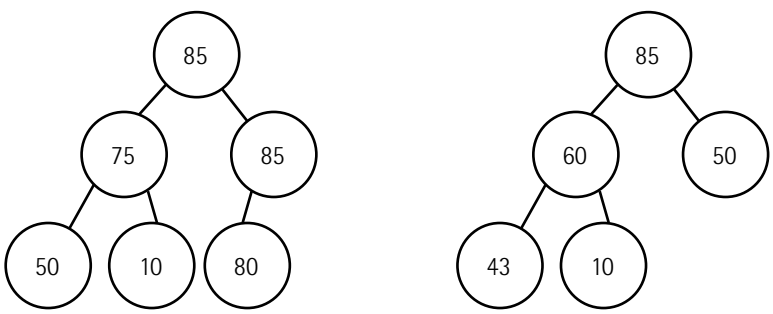


Figura 17 – Árvores binárias que podem ser consideradas heaps

Em árvores binárias, muitas operações têm complexidade de tempo O(h), sendo h a altura da árvore. Assim, para reduzir o tempo de execução, é vantajoso que a altura seja a menor possível, o que nos leva a utilizar uma árvore completa ou quase completa, que é o caso do heap. Como a altura mínima de uma árvore binária é $\log_2 n$ (sendo n o número de nós), uma heap construída assim terá a complexidade de tempo de suas operações como O(log n), o que explica a complexidade apresentada incialmente.

O fato de o heap ser uma árvore completa também permite usar uma estrutura de dados linear (vetor ou lista) para representá-lo, o que facilita muito o processo para os algoritmos, pois dispensa o uso de ponteiros (Feofiloff, 2009, p. 115). Cada elemento do heap ocupa uma posição no vetor como se fosse resultado de uma busca em largura.

A figura 18 mostra um heap e as posições que cada elemento ocuparia no vetor. Como os elementos não estão em ordem crescente (como quando o heap é representado na forma de árvore), a visualização se torna menos intuitiva.

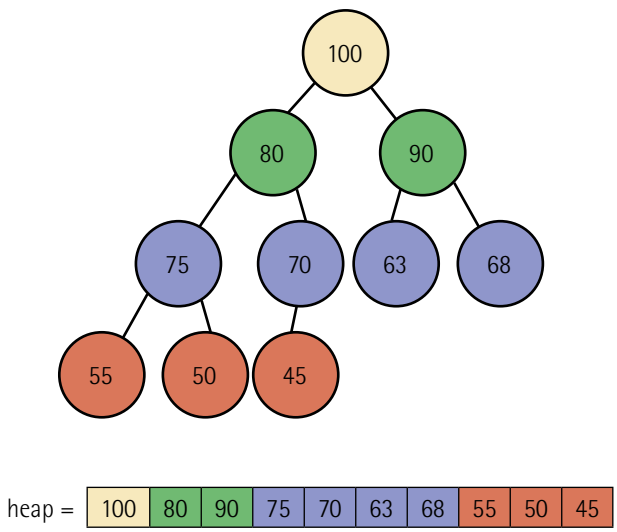


Figura 18 – Representação de heap com vetor

Os elementos foram dispostos em níveis, pois assim a inserção de elementos ao final do vetor garante o formato do heap como árvore binária. No entanto, inserir novos elementos ao final (e a eventual retirada do elemento na posição de saída, que é a raiz) não garante que a ordem sequencial permaneça. Assim, vamos tratar da remoção de elementos da estrutura, tanto na forma de árvore quanto de vetor.

Para navegar numa estrutura em árvore, a partir de cada nó é necessário acessar seu nó-pai e seus nós-filhos (se houver, é claro). Ao representarmos isso na forma de vetor, deve ser possível estabelecer uma relação entre o índice do nó atual e os índices dos nós que se conectam com ele. A relação é a seguinte:

- o filho à esquerda de um nó será dado pela fórmula 2 × índice +1;
- o filho à direita de um nó será dado pela fórmula 2 × (índice +1);
- o pai de um nó será dado pela fórmula (índice –1) / 2, arredondado para baixo (se fracionário).

A figura 19 revela que essa relação é constante e se aplica a todo e qualquer nó, independente do tamanho do vetor que contenha o heap. Qualquer elemento na estrutura, como foi dito, será sempre inserido na primeira posição livre da árvore, o que na prática significa a última posição do vetor. Isso garante que a árvore seja sempre completa ou quase completa; no entanto, a regra de que um nó é sempre menor ou igual ao seu pai deve ser respeitada para continuarmos com a estrutura do heap.

Assim, o elemento inserido deve ser comparado com o elemento imediatamente acima (seu nó-pai): se for maior, devem trocar de posição. Em caso de troca, o elemento que "subiu" na lista de prioridades deve ser novamente comparado e eventualmente trocado, até que esteja abaixo de um elemento maior ou igual. Confira o processo:

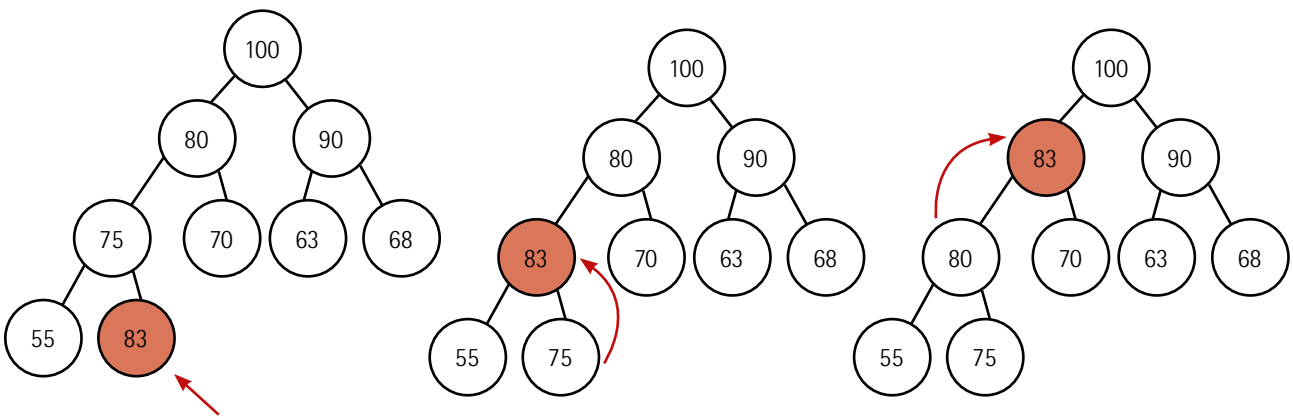


Figura 19 – Inserção de um elemento num heap

Já a remoção será sempre do elemento da raiz, que é o elemento de maior prioridade, no entanto sua simples remoção distorce totalmente a estrutura. A metodologia que resolve esse problema é a seguinte: o elemento da raiz será trocado de posição com o último elemento da estrutura; assim, apenas o elemento presente na raiz estará fora de posição.

O método de reposicionar o elemento da raiz é denominado heapify (heapificação): a raiz será comparada com seus dois filhos, e o maior dos três elementos assumirá a raiz, trocando de posição com o que estava lá (o terceiro elemento não tem sua posição modificada). O elemento que "desceu" é novamente comparado, via heapify, com seus dois filhos, e o processo se repete até que todos os elementos estejam devidamente posicionados e a condição básica do heap (filhos menores ou iguais aos pais) seja alcançada.

A figura 20 ilustra o processo de remoção:

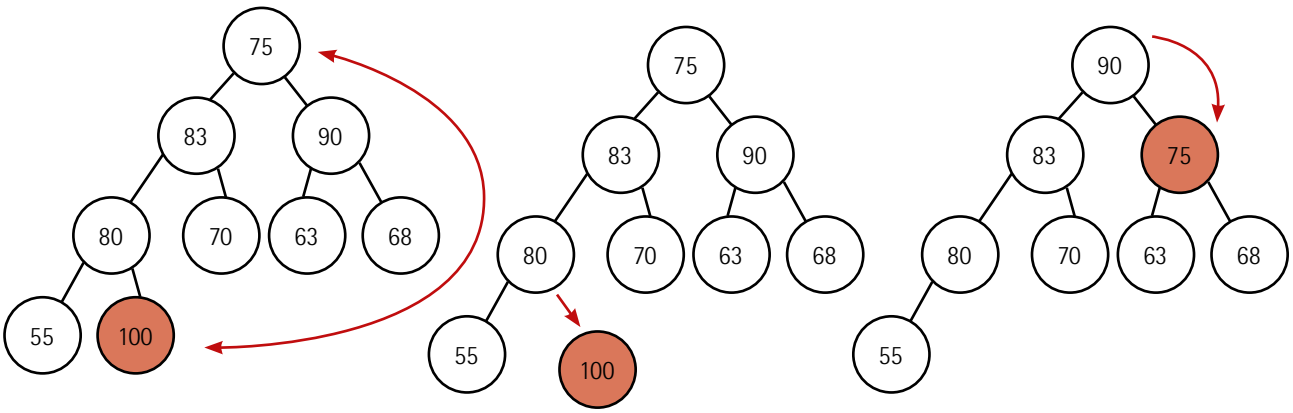


Figura 20 – Remoção de elemento num heap

No pior cenário imaginável, o elemento que foi para raiz descerá até as folhas da árvore (última camada do heap). Nesse caso, a quantidade de comparações e trocas será a altura da árvore, que é $\log_2 n$. Assim, a complexidade de tempo dessa operação é O(log n).

Exemplo de aplicação

Considerando uma lista incialmente vazia como um heap:

a) Insira, na ordem estipulada, os elementos 85, 60, 90, 70, 10, 25, 50, 75.

- Inserção do 85:

  [85]

- Inserção do 60:

[85, 60]

- Inserção e reposicionamento do 90:

[85, 60, 90]

[90, 60, 85]

- Inserção e reposicionamento do 70:

[90, 60, 85, 70]

[90, 70, 85, 60]

- Inserção do 10:

[90, 70, 85, 60, 10]

- Inserção do 25:

[90, 70, 85, 60, 10, 25]

- Inserção do 50:

[90, 70, 85, 60, 10, 25, 50]

- Inserção e reposicionamento do 75:

[90, 70, 85, 60, 10, 25, 50, 75]

[90, 70, 85, 75, 10, 25, 50, 60]

[90, 75, 85, 70, 10, 25, 50, 60]

b) Remova dois elementos da estrutura.

- Primeira remoção:

  – Troca de posição entre 90 e 60:

[60, 75, 85, 70, 10, 25, 50, 90]

  – Remoção do 90:

[60, 75, 85, 70, 10, 25, 50]

  – Reposicionamento do 60:

[85, 75, 60, 70, 10, 25, 50]

- Segunda remoção:

  – Troca de posição entre 85 e 50:

[50, 75, 60, 70, 10, 25, 85]

– Remoção do 85:

[50, 75, 60, 70, 10, 25]

– Reposicionamento do 50:

[75, 50, 60, 70, 10, 25]
[75, 70, 60, 50, 10, 25]

Observação

O reposicionamento dos elementos no heap é de difícil visualização na forma de vetor. O leitor pode representar o heap como um grafo para acompanhar as mudanças, caso sinta dificuldade.

Para transformar um vetor qualquer em um heap, aplicaríamos o método heapify sucessivamente em todos os nós, começando pelo nó-pai da última folha e seguindo em direção à raiz do heap (primeiro elemento do vetor).

O código em Python a seguir ilustra o funcionamento de um heap utilizando uma lista, inicialmente vazia, e permitindo inserir e remover elementos na estrutura, exibindo a lista a cada etapa. Diferente de mostrar apenas funções para usos específicos, vamos apresentar agora um programa completo:

```
#Executa a troca entre dois elementos
#quando solicitado durante a inserção ou heapify
def troca (x,y):
     aux = heap[x]
     heap[x] = heap[y]
     heap[y]=aux

#insere um elemento no heap e o posiciona
#na posição correta por meio de trocas sucessivas
def inserir (x):
     heap.append(x)
     index=len(heap)-1
     while heap[index]>heap[int((index-1)/2)]:
        if heap[index]>heap[int((index-1)/2)]:
           troca(index, int((index-1)/2))
        if index > 0:
           index = int((index-1)/2)

#remove um elemento e aplica o heapify
#percorre um caminho logarítmico para não realizar comparações desnecessárias
def remover():
     if len(heap)==0:
        print("Heap vazio")
     else:
```

```
        import math
        troca(0,-1)
        print("Saída: ", heap[-1])
        del(heap[-1])
        x = 0
        while x < int(math.log(len(heap)+1)/math.log(2)) :
          if 2*(x+1) < len(heap):
            if heap[2*x+1] > heap[2*(x+1)]:
                maior= 2*x+1
              else:
                maior=2*(x+1)
            if heap[x]<heap[maior]:
              troca(x,maior)
              x = maior
            else:
              x=int(math.log(len(heap)+1)/math.log(2))


#Corpo principal do programa
heap=[]
op="1"
while op!="4":
    op=input("1. Exibir\n2. Inserir\n3. Remover\n4. Sair\n")
    if op =="1":
      print(heap)
    elif op =="2":
      novo=int(input("Digite o que será inserido: "))
      inserir(novo)
      print(heap)
    elif op =="3":
      remover()
      print(heap)
    elif op !="4":
        print("Opção inválida")
```

A função **troca** executa a troca de posição entre dois elementos, necessária para inserir elementos ou para o heapify. As funções **inserir** e **remover** executam a inserção e remoção dos elementos, com seus respectivos posicionamentos (se necessário), como foi descrito anteriormente.

Exemplo de aplicação

(Enade 2011, adaptado). As filas de prioridade (heaps) são estruturas de dados importantes no projeto de algoritmos. Em especial, heaps podem recuperar informação em grandes bases de dados constituídos por textos. Basicamente, para exibir o resultado de uma consulta, os documentos recuperados são ordenados de acordo com a relevância presumida para o usuário. Uma consulta pode recuperar milhões de documentos que certamente não serão examinados em sua totalidade; na verdade, o usuário examina os primeiros **m** documentos dos **n** recuperados, sendo **m** da ordem de algumas dezenas.

Considerando as características dos heaps e sua aplicação no problema descrito, avalie as seguintes afirmações:

I – Uma vez que o heap é implementado como árvore binária de busca essencialmente completa, o custo computacional para sua construção é O(n log n).

Resposta: um heap não é organizado como árvore binária de busca. Embora ambos sejam árvores binárias, a de busca não é necessariamente completa; além disso, uma árvore terá cada nó com um filho de maior e um filho de menor valor, enquanto em um heap ambos os filhos terão sempre valores menores ou iguais aos do nó-pai. Assim, a afirmativa é **incorreta**.

II – A implementação de heaps utilizando-se vetores é eficiente em tempo de execução e em espaço de armazenamento, pois o pai de um elemento na posição i está armazenado na posição 2i+1.

Resposta: a afirmação está **incorreta**. A posição do pai de um nó será dada pela fórmula (i – 1) / 2, arredondado para baixo (se fracionário).

III – O custo computacional para recuperar de forma ordenada os **m** documentos mais relevantes armazenados em um heap de tamanho n é O(m log n).

Resposta: remover um elemento de um heap sempre implica executar uma operação de heapify na sequência, para reposicionar os elementos que continuam na estrutura e manter sua ordem necessária. Como o tempo de execução de um heapify é O(log n), a extração de m elementos e consequentes heapifys terá, sim, um custo de tempo de O(m log n). Assim, a afirmativa está **correta**.

IV – Determinar o documento com maior valor de relevância armazenado em um heap implica custo computacional O(1).

Resposta: a principal característica de um heap é ter sempre seu elemento de maior prioridade na raiz da árvore, ou na primeira posição do vetor. Assim, inspecionar o elemento de maior relevância terá realmente custo computacional O(1), pois sua posição é sempre conhecida. Assim, a afirmativa está correta.

3.6 Heap de Fibonacci

É a forma mais avançada de um heap, sendo não uma, mas uma coleção de árvores que, individualmente, têm propriedades de heap; porém não precisam necessariamente ser binárias e não precisam estar, como conjunto, ordenadas pela prioridade de suas raízes. Assim, a estrutura é menos rígida que um heap comum. Tal flexibilidade implica uma representação com uma estrutura de dados mais complexa, muitas vezes sendo necessária uma lista circular duplamente ligada para sua representação.

Uma vantagem dessa estrutura é que a complexidade média de tempo de inserção de um elemento se torna constante O(1) ao longo de diversas execuções sucessivas do algoritmo, mas a complexidade de sua implementação faz com que a estrutura seja muitas vezes preterida em função dos heaps comuns, que precisam apenas da estrutura de dados mais simples. Neste livro-texto não vamos detalhar o funcionamento dessa estrutura, mantendo-nos na sua descrição básica.

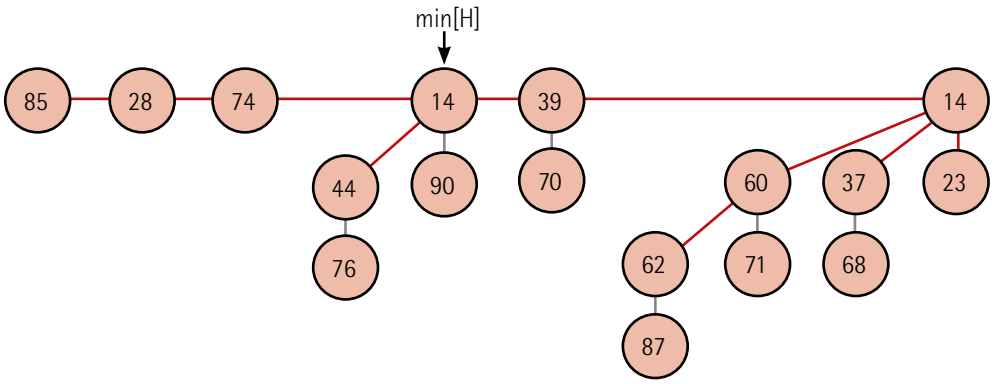


Figura 21 – Exemplo de um heap de Fibonacci

Disponível em: https://tinyurl.com/2z8ujc87. Acesso em: 6 dez. 2023.

3.7 Exemplo: algoritmo de Dijkstra

Embora seja um algoritmo sobre grafos (tratados em outro tópico deste livro-texto), o algoritmo de Dijkstra é um exemplo clássico da aplicação de um heap como fila de prioridades. Caso o leitor não esteja familiarizado com a teoria dos grafos, recomendamos que leia o início do tópico 5 antes de seguir.

Esse algoritmo foi desenvolvido em 1959 pelo cientista da computação holandês que lhe dá o nome, Edsger W. Dijkstra, e tem como objetivo encontrar o menor caminho entre dado vértice de um grafo ponderado e os demais vértices da estrutura, transformando o grafo em uma árvore cuja raiz é o vértice de origem e se ramifica infinitamente (embora não seja necessário construir a totalidade da árvore). Em seguida, uma busca em largura é realizada na árvore (Dasgupta; Papadimitriou; Vazirani, 2009, p. 109).

A seguir, o funcionamento do algoritmo:

1) Cria-se uma lista de menores caminhos a partir do vértice inicial para todos os vértices do grafo. Ao caminho do vértice para ele mesmo é atribuído o valor **zero**, e para todos os demais valores é atribuído o valor **infinito** (não existem caminhos antes de iniciar o algoritmo). Também se cria uma lista dos nós ainda não visitados durante a execução do algoritmo (no início, todos os nós estarão nessa lista).

2) Exploram-se todos os vizinhos do vértice inicial, registrando e atualizando-se as menores distâncias.

3) O vértice com menor caminho atual é selecionado, e exploram-se os caminhos que passam por ele.

4) A etapa 3 é repetida até todos os vértices serem visitados.

Durante a execução, diferentes caminhos para o mesmo nó serão encontrados e armazenados. Para um grafo pequeno, uma lista ordenada simples é uma opção para registrar esses caminhos;

porém, caso o grafo seja muito grande, um heap passa a ser a melhor opção para armazenar cada um desses conjuntos.

A função em Python a seguir executa esse algoritmo, recebendo como parâmetros um grafo representado na forma de uma matriz de adjacências (ver tópico 5) e o vértice para o qual se pretende encontrar os caminhos (na forma de um índice da lista que representa a matriz):

```
def algoritmo_dijkstra(grafo, origem):
    #Inicia as listas com as distâncias
    import sys
    distancia = [sys.maxsize] * len(grafo)
    distancia[origem] = 0
    visitados = [False] * len(grafo)

    for _ in range(len(grafo)):
      menor_distancia = sys.maxsize
      menor_distancia_no = None
      for i in range(len(grafo)):
        if not visitados[i] and distancia[i] < menor_distancia:
          menor_distancia = distancia[i]
          menor_distancia_no = i

        # Marca como visitado o nó no qual o algoritmo já foi aplicado
        visitados[menor_distancia_no] = True

        # Atualiza as distâncias dos nós vizinhos e armazena a menor
        for vizinho in range(len(grafo)):
          if not visitados[vizinho] and grafo[menor_distancia_no][vizinho] > 0:
            distancia_vizinho = distancia[menor_distancia_no] + grafo[menor_
distancia_no][vizinho]
            if distancia_vizinho < distancia[vizinho]:
              distancia[vizinho] = distancia_vizinho
    return distancia
```

A lista **menor_distancia_no** contém prioridades e armazena, a cada iteração, as distâncias entre o vértice de origem e um dado vértice. A lista **distancia**, que é o retorno da função, recebe as prioridades da lista anterior a cada iteração do algoritmo.

Lembrete

O método sys.maxsize, do módulo sys do Python, retorna o maior valor possível comportável por uma variável inteira; e o usamos nesse caso como uma aproximação com o infinito.

Não é difícil perceber que o algoritmo de Dijkstra é na verdade a otimização de uma busca em largura exaustiva, onde se aplicam conceitos de programação dinâmica (localizar um caminho de cada vez). A complexidade de tempo do algoritmo de Djikstra, quando utilizamos o grafo na forma de uma matriz de adjacências, é O(E log V), sendo E o número de arestas do grafo e V o número de vértices.

A complexidade de espaço será O(2 V), dada a construção de duas estruturas de dados para executar o algoritmo.

Exemplo de aplicação

Embora o algoritmo de Dijkstra seja a solução de mais fácil implementação, não é o mais rápido, além de apresentar problemas em grafos direcionados ou com arestas de valores negativos. Assim, pesquise os algoritmos de Floyd-Warshall e de Bellman-Ford, que são outras opções para resolver o problema, e analise as diferenças entre eles.

A seguir, confira mais um exemplo de aplicação, para fixar as características do algoritmo de Dijkstra.

Exemplo de aplicação

(Enade 2021, adaptado). O algoritmo de Dijkstra para o problema do caminho mínimo em dígrafos com pesos utiliza uma fila de prioridades de vértices, na qual as prioridades são uma estimativa do custo final. A cada iteração, um vértice é retirado da fila, e os arcos que começam nesse vértice são analisados.

Considere o seguinte grafo, no qual se deseja conhecer o custo de um caminho mínimo para cada vértice, a partir do vértice D. Saiba que –1 representa um custo "infinito", ou seja, nenhum caminho até o vértice foi descoberto até o momento.

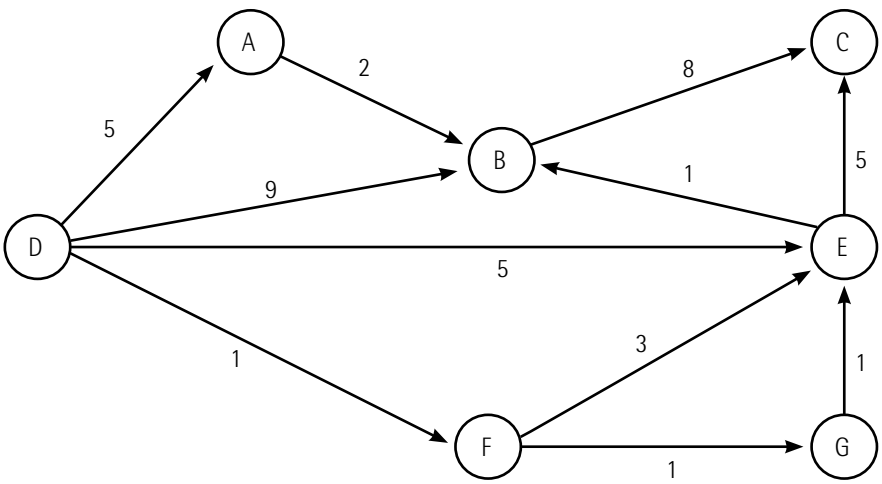


Figura 22

Com base nas informações e no grafo apresentado, calcule a estimativa de custo após duas iterações do algoritmo.

Resposta: o algoritmo se inicia com todas as distâncias como –1 (que representa infinito, conforme o enunciado), exceto com a distância de D até ele mesmo, que será igual a zero.

Após a primeira iteração, temos os seguintes caminhos a partir do vértice D:

- Distância para A: 5
- Distância para B: 9
- Distância para C: –1
- Distância para D: 0
- Distância para E: 5
- Distância para F: 1
- Distância para G: –1

A menor das distâncias é aquela para o vértice F. Então a segunda iteração partirá de F, verificando estes caminhos:

- Distância de F a E: 3
- Distância de F a G: 1

Somamos então esses valores com a menor distância de D a F (que é 1):

- Distância de D a E: 4 (lista de prioridade: [4, 5])
- Distância de D a G: 2 (lista de prioridade: [2, –1])

Como essas distâncias são menores que aquelas obtidas a partir da primeira iteração, esses valores as substituem, pois têm maior prioridade. O vértice F será marcado como já visitado.

Assim, a resposta para a questão será:

- Distância para A: 5
- Distância para B: 9
- Distância para C: –1
- Distância para D: 0
- Distância para E: 4

- Distância para F: 1 (já visitado)
- Distância para G: 2

Embora a questão já tenha sido respondida, vamos dar continuidade até o término do algoritmo. A terceira iteração começa no vértice G, pois atualmente é a menor distância, e o processo se repetiria:

- Distância de E a B: 1
- Distância de E a C: 5

Somamos então esses valores com a menor distância de D a G (que é 2):

- Distância de D a E: 3 (lista de prioridade: [3, 4, 5])
- Distância para A: 5
- Distância para B: 9
- Distância para C: –1
- Distância para D: 0
- Distância para E: 3
- Distância para F: 1 (já visitado)
- Distância para G: 2 (já visitado)

A quarta iteração começa no vértice E:

- Distância de E a B: 1
- Distância de E a C: 5

Somamos então esses valores com a menor distância de D a E (que é 3):

- Distância de B a E: 4 (lista de prioridade: [1, 9])
- Distância de B a C: 8 (lista de prioridade: [8, –1])

Após a quarta iteração já teremos um caminho para cada vértice a partir do vértice D:

- Distância para A: 5
- Distância para B: 4
- Distância para C: 8
- Distância para D: 0
- Distância para E: 3 (já visitado)
- Distância para F: 1 (já visitado)
- Distância para G: 2 (já visitado)

Embora o algoritmo ainda vá executar mais três iterações, os menores caminhos já foram encontrados ao término desta execução, ainda que o algoritmo não tenha como verificar isso. Deixamos para o leitor executar as outras iterações e verificar o resultado.

4 ALGORITMOS SOBRE CADEIAS

Por cadeias de dados entende-se qualquer estrutura de dados sequencial e linear, como vetores, listas, pilhas e filas. Vamos tratar aqui das principais operações nessas estruturas e algoritmos que atuam sobre cadeias, como os principais algoritmos de ordenação e busca.

4.1 Operações básicas sobre cadeias

Independentemente do tipo de cadeia de dados, algumas operações são recorrentes em todas elas. Algumas foram abordadas rapidamente em outros tópicos, então vamos organizar aqui as principais operações realizadas em cadeias. As complexidades de tempo para essas operações serão sempre, no máximo, O(n), sendo n o número total de elementos na cadeia.

- Preenchimento de uma cadeia inicialmente vazia: O(n).
- Inserção/remoção de elemento na extremidade da cadeia: O(1).
- Inserção/remoção de elemento em estrutura ordenada: O(log n).
- Substituir um elemento de posição conhecida: O(1).
- Busca de elemento em estrutura não ordenada: O(n).

- Busca de elemento em estrutura ordenada: O(log n).
- Copiar uma cadeia: O(n).
- Concatenar duas cadeias: O(n), sendo n a dimensão da cadeia que está sendo inserida.

Outras operações podem ser executadas em uma cadeia, como: trocar a posição de dois elementos da cadeia, inserir ou remover um trecho de uma cadeia etc., mas, para manter a lista mais objetiva, reduzimos as operações básicas, que envolviam inserir, remover, copiar, substituir e localizar. Como uma cadeia tem tamanho constante (seja ela estática, como um vetor, ou dinâmica, como no caso de uma lista), a complexidade de espaço envolvida nessas operações será sempre O(n).

Pilhas e filas, apesar de seu funcionamento diferente entre si (como já analisamos neste livro-texto), têm as complexidades das suas operações equivalentes às já descritas. Da mesma forma, as **filas duplamente terminadas** (deque, do inglês *double-ended queue*) têm a mesma complexidade, e a complexidade de tempo de inserir (usualmente referida como push) ou remover (pop) elementos nessas estruturas será sempre uma constante O(1). Não se costuma fazer outras operações envolvendo essas estruturas.

Listas – sejam elas simples, duplamente ligadas ou circulares – terão complexidade de tempo para ser completamente percorridas, quando necessário, de O(n). As complexidades de inserção e remoção de elementos dependerão das regras de funcionamento para cada lista em particular, de acordo com a lista apresentada anteriormente.

Exemplo de aplicação

(Enade 2021, adaptado). A biblioteca de coleções da linguagem Java disponibiliza implementações de propósito geral para estruturas de dados elementares, como listas, filas e pilhas. Considere as seguintes definições de classe que representam implementações de estruturas de dados disponíveis na biblioteca da linguagem:

- **Classe A**: os objetos são organizados em uma ordem linear e podem ser inseridos somente no início ou no final dessa sequência.
- **Classe B**: os objetos são organizados em uma ordem linear determinada por uma referência ao próximo objeto.
- **Classe C**: os objetos são removidos na ordem oposta em que foram inseridos.
- **Classe D**: os objetos são inseridos e removidos respeitando a seguinte regra: o elemento a remover é sempre o que foi inserido primeiro.

Nesse contexto, indique as estruturas que correspondem, respectivamente, às estruturas de dados implementadas pelas classes A, B, C e D.

- A chave para identificar a estrutura implementada pela classe A está na frase "podem ser inseridos somente no início ou no final". Uma fila ou pilha comuns podem ter a inserção de elementos apenas no seu final; já as listas permitem a inserção em qualquer ponto da estrutura. Assim, a única estrutura de dados que satisfaz essa condição é o **deque**.

- Uma estrutura de dados na qual cada elemento apenas registra qual a posição do elemento seguinte da estrutura é uma **lista simplesmente ligada**.

- Se os elementos só puderem ser removidos na ordem oposta em que foram inseridos, temos um caso de "primeiro a entrar, último a sair" (*first in*, *last out* – Filo). Assim, a estrutura de dados implementada pela classe C é uma **pilha**.

Na classe D a estrutura implementada é, sob certos aspectos, oposta à da classe C: "primeiro a entrar, primeiro a sair (*first in, first out* – Fifo). Assim, a estrutura de dados implementada pela classe C é uma fila.

4.2 Algoritmos de ordenação

Vamos analisar, sob o viés da complexidade computacional, alguns algoritmos de ordenação. Um algoritmo de ordenação organiza os dados de uma cadeia de forma crescente ou decrescente (a ordem que ele insere na cadeia não faz diferença para a análise feita). Todos os algoritmos de ordenação se baseiam em dois elementos: comparações e movimentações (trocas de posição) entre elementos das cadeias. Assim, teremos duas complexidades envolvidas nesses algoritmos, sendo n o número de registros na cadeia que será ordenada:

- número de comparações C(n) entre chaves;

- número de movimentações M(n) dos elementos.

Outro aspecto importante é sua estabilidade. Um algoritmo de ordenação é estável se os valores iguais mantiverem suas posições originais relativas após o processo de ordenação. Caso essa ordem seja modificada, o algoritmo é chamado de instável.

A figura 23 mostra a estabilidade de um algoritmo de ordenação:

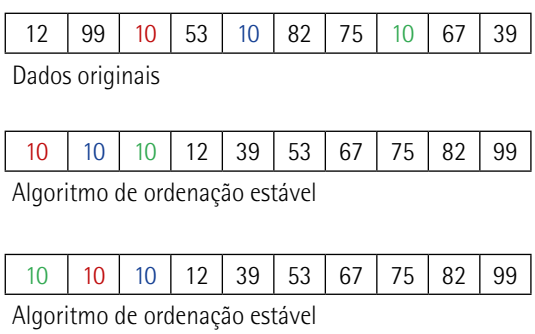


Figura 23 – Algoritmos de ordenação estáveis e instáveis

Esses algoritmos foram escolhidos com base em sua popularidade, aplicação e características de interesse, não sendo uma lista completa de todos os algoritmos existentes. A seguir, uma função em Python utilizando algoritmos para ordenar uma lista homogênea recebida como parâmetro da função; fica o convite ao leitor que quiser se aprofundar na execução, análise e modificação das funções apresentadas.

Observação

Considerando métodos de ordenação e variantes, existem mais de vinte algoritmos conhecidos, com diferentes recomendações de aplicação, para diferentes estruturas de dados.

4.2.1 Bubble Sort

É um algoritmo iterativo, de muito fácil implementação, sendo muitas vezes o primeiro do tipo com que os estudantes entram em contato. O preço da simplicidade é ser um algoritmo de execução muito lenta, além de ser um dos menos influenciados pelo estado original dos dados que serão ordenados. Tem complexidades $C(n) = O(n^2)$ e $M(n) = O(n^2)$, e é um algoritmo estável. Como o algoritmo utiliza a própria cadeia para fazer a ordenação, a complexidade de espaço é constante $O(1)$.

Exemplo de Bubble Sort:

```
def BubbleSort (lista):
    #Esta repetição percorre a lista várias vezes
    for i in range(len (lista)):
        troca=True
        while troca:
          troca=False
          #Esta repetição executa as comparações e trocas a cada iteração
          for j in range(0, len (lista) - i - 1):
            if lista[j] > lista[j + 1]:
              lista[j], lista[j+1] = lista[j+1], lista[j]
                        troca=True
```

4.2.2 Insertion Sort

Percorre a cadeia de uma extremidade a outra (da esquerda para a direita, por exemplo) e, conforme avança, vai ordenando os elementos na extremidade inicial pela inserção de cada elemento na posição correspondente. Sua complexidade $C(n)$ varia entre $O(n)$ e $O(n^2)$, dependendo do estado original dos dados. A complexidade de movimentações $M(n)$ fica na mesma ordem, de novo em função do estado original dos dados, e é um algoritmo de ordenação estável. Como também utiliza a própria cadeia para fazer a ordenação, a complexidade de espaço é constante $O(1)$ (Goodrich; Tamassia, 2013, p. 536).

Exemplo de Insertion Sort:

```
def InsertionSort(lista):
    for i in range(1, len(lista)):
      #A variável elemento é o elemento da lista que está
      #sendo posicionado nesta iteração.
      elemento = lista[i]
      j = i - 1
      while j >= 0 and elemento < lista[j]:
        lista[j + 1] = lista[j]
        j -= 1
        lista[j + 1] = elemento
```

4.2.3 Selection Sort

Seleciona o menor item da cadeia e o coloca na primeira posição, depois seleciona o segundo menor e o coloca na segunda posição, e assim sucessivamente até o final da cadeia que está sendo ordenada. Tem complexidades $C(n) = O(n^2)$ e $M(n) = O(n)$, e é um algoritmo de ordenação instável. Também utiliza a própria cadeia para realizar a ordenação, assim, a complexidade de espaço é constante $O(1)$.

Exemplo de Selection Sort:

```
def SelectionSort(lista):
    for i in range(len(lista)):
      #A variável mínimo indica o índice selecionado para a
      #movimentação do elemento
      minimo = i
      for j in range(i + 1, len(lista)):
        if lista[j] < lista[minimo]:
            minimo = j
      if minimo != i:
            lista[i], lista[minimo] = lista[minimo], lista[i]
```

4.2.4 Quick Sort

Como diz o nome, é o algoritmo de ordenação mais rápido (quick) em muitos cenários. Seu objetivo é dividir o problema de ordenar um conjunto com n itens em dois problemas menores, utilizando-se de programação dinâmica (metodologia muito parecida com a utilizada para resolver o MCOP no tópico 2). As subcadeias resultantes da divisão da cadeia original são ordenadas, e os resultados são combinados para produzir a solução final, que é a cadeia reconstruída e ordenada. Tem complexidade $C(n) = O(n^2)$ no pior caso e $C(n) = O(n \log n)$ em casos melhores, sendo um dos algoritmos mais influenciados pelo estado original dos dados.

A complexidade M(n) não se aplica ao Quick Sort, uma vez que não movimenta elementos ao longo da cadeia. Esse algoritmo de ordenação é instável e "desmonta" e "remonta" a cadeia que está sendo ordenada, portanto a complexidade de espaço é linear O(n) (Goodrich; Tamassia, 2013, p. 536).

Exemplo de Quick Sort:

```
def QuickSort(lista):
    if len(lista) <= 1:
      return lista
    #A variável pivo marca o ponto de divisão da lista
    pivo = lista[0]
    #Três listas são criadas a partir da lista original
    menores, iguais, maiores = [], [], []
    for elemento in lista:
      if elemento < pivo:
        menores.append(elemento)
      elif elemento == pivo:
          iguais.append(elemento)
      else:
        maiores.append(elemento)
    #A função é recursiva, retornando as três listas
    return QuickSort(menores) + iguais + QuickSort(maiores)
```

4.2.5 Merge Sort

Assim como o Quick Sort, é um método dinâmico de ordenação que se utiliza da estratégia "dividir para conquistar" para resolver problemas (como vimos no tópico 2). Tem complexidade C(n) = O(n log n) para todos os casos e, como no Quick Sort, a complexidade M(n) não se aplica, pois também não realiza substituições.

Esse algoritmo de ordenação é estável e, em termos de complexidade de espaço, é análogo ao Quick Sort, tendo complexidade linear O(n) (Goodrich; Tamassia, 2013, p. 536).

Exemplo de Merge Sort:

```
def MergeSort(lista):
    if len(lista) > 1:
      #Divide a lista ao meio
      meio = len(lista) // 2
      esquerda = lista[:meio]
      direita = lista[meio:]
      #Se aplica recursivamente às duas metades
      MergeSort(esquerda)
      MergeSort(direita)
      i = j = k = 0
      #Processo de "fusão" das listas
      while i < len(esquerda) and j < len(direita):
        if esquerda[i] < direita[j]:
            lista[k] = esquerda[i]
            i += 1
        else:
          lista[k] = direita[j]
          j += 1
          k += 1
      while i < len(esquerda):
            lista[k] = esquerda[i]
```

```
            i += 1
            k += 1
        while j < len(direita):
           lista[k] = direita[j]
           j += 1
              k += 1
```

4.2.6 Shell Sort

Criado por Donald Shell em 1959, é uma variante do algoritmo de ordenação por inserção (Insert Sort) que possibilita trocar registros distantes um do outro, diferente do algoritmo no qual se baseia, que apenas troca elementos adjacentes. Até hoje a função de complexidade de tempo do algoritmo é desconhecida: ainda não existe uma fórmula fechada para sua função de complexidade, embora se estime que seja algo entre a linear e a quadrática. É de ordenação instável e tem complexidade de espaço linear O(n).

Exemplo de Shell Sort:

```
def ShellSort(lista):
    n = len(lista)
    #A variável gap define o intervalo entre as trocas
    gap = n // 2
    while gap > 0:
       for i in range(gap, n):
          elemento = lista[i]
          j = i
          #Compara elementos distantes na lista
          while j >= gap and lista[j - gap] > elemento:
             lista[j] = lista[j - gap]
             j -= gap
          lista[j] = elemento
          gap //= 2
```

Saiba mais

Para mais detalhes sobre os algoritmos de ordenação apresentados, leia os capítulos 8 e 11 deste livro:

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. Porto Alegre: Bookman, 2013. Disponível em: https://tinyurl.com/2eba5xvv. Acesso em: 7 dez. 2023.

A tabela 3 resume as informações apresentadas sobre esses algoritmos:

Tabela 3 – Comparação dos algoritmos de ordenação selecionados

Algoritmo	Comparações		Movimentações		Estabilidade	Espaço
	Melhor caso	Pior caso	Melhor caso	Pior caso		
Bubble Sort	$O(n^2)$		$O(n^2)$		Estável	O(1)
Insertion Sort	O(n)	$O(n^2)$	O(n)	$O(n^2)$	Estável	O(1)
Selection Sort	$O(n^2)$		O(n)		Instável	O(1)
Quick Sort	O(n log n)	$O(n^2)$	NA		Instável	O(n)
Merge Sort	O(n log n)		NA		Estável	O(n)
Shell Sort	$O(n^{1,25})$ ou $O(n (ln\ n)^2)$*		NA		Instável	O(n)

NA: Não aplicável
*: estimado

Adaptada de: Cormen (2013, p. 48).

Observação

Os métodos de ordenação nativos do Python, como o sort() e o sorted(), se utilizam de uma variação do algoritmo Selection Sort e são estáveis (Built-in..., 2008).

Exemplo de aplicação

(Enade 2021, adaptado) Existe um grande número de implementações para algoritmos de ordenação. Um dos fatores a considerar, por exemplo, é o número máximo e médio de comparações necessárias para ordenar um vetor com n elementos. Diz-se também que um algoritmo de ordenação é estável se ele preserva a ordem de elementos iguais; isto é, se tais elementos aparecem na sequência ordenada na mesma ordem em que estão na sequência inicial.

Analise o algoritmo a seguir, onde A é um vetor, e i, j, lo e hi são índices do vetor:

```
algoritmo ordena(A, lo, hi)
    se lo < hi então
       p := particao(A, lo, hi)
       ordena(A, lo, p - 1)
       ordena(A, p + 1, hi)
algoritmo particao(A, lo, hi)
    pivot := A[hi]
    i := lo
    repita para j := lo até hi
       se A[j] < pivot entao
       troca A[i] com A[j]
       i := i + 1
    troca A[i] com A[hi]
    return i
```

Com relação ao algoritmo apresentado, avalie as afirmações:

I – O algoritmo precisa de um espaço adicional O(n) para a pilha de recursão.

Resposta: o algoritmo **partição** divide o vetor em duas partes. Todo algoritmo de ordenação que envolve uma divisão da estrutura de dados original deverá construir uma segunda estrutura de dados de complexidade de espaço O(n) para "remontar" a estrutura durante a ordenação. Portanto a afirmação é **verdadeira**.

II – O algoritmo apresentado é de ordenação, recursivo e estável.

Resposta: analisando o trecho A[j] < pivot, verificamos que, caso tenhamos dois valores iguais na mesma partição, a ordenação será estável. Porém, se dois valores iguais estiverem em partições diferentes, a estabilidade não é garantida. Em relação à recursividade, a chamada da função **troca()**, dentro da função em que ela é naturalmente chamada, torna o algoritmo recursivo. Assim, a afirmação é **falsa**.

III – O algoritmo precisa, em média, de O(n log n) comparações para ordenar n itens.

Resposta: a partição em duas partes e a comparação da variável selecionada como pivot tornam o número de comparações logarítmico. Portanto a afirmação é **verdadeira**.

IV – O primeiro elemento do vetor como "pivot" é mais eficiente que o último.

Resposta: a partição funciona com mais eficiência quanto mais próxima do meio do vetor. Uma partição com o primeiro ou com o último elemento será igualmente ineficiente. Assim, a afirmação é **falsa**.

4.3 Autômatos e máquina de Turing

No estudo de linguagens formais, é comum identificar se determinada cadeia de símbolos pertence ou não a uma dada linguagem. Para entender melhor esse tema, vamos rever rapidamente alguns conceitos fundamentais de linguagens formais (Menezes, 2011, p. 52):

- Alfabeto é um conjunto finito de símbolos, que são a menor unidade constituinte de uma linguagem.
- Cadeia é um conjunto finito de símbolos do alfabeto.
- Gramática é um conjunto de regras que constrói cadeias de símbolos do alfabeto.
- Linguagem é o conjunto de todas as cadeias construídas por uma gramática (que tenta definir a própria linguagem).

Assim, reconhecedor será um algoritmo que identifica se dada cadeia foi construída ou não de acordo com a gramática de dada linguagem. A construção e o formato desse algoritmo serão definidos pela natureza da linguagem sobre a qual irá atuar.

Saiba mais

Encontre mais detalhes sobre hierarquia das linguagens no capítulo 9 deste livro:

MENEZES, P. B. *Linguagens formais e autômatos*. Porto Alegre: Grupo A, 2011. v. 3. Disponível em: https://tinyurl.com/yt7hd9em. Acesso em: 7 dez. 2023.

Algoritmos para identificar cadeias são usualmente **sistemas de estados finitos**, ou seja, modelos matemáticos em que um componente (ou estado; computacionalmente, podemos pensar nele como uma variável) pode variar entre um conjunto finito e predefinido de estados; essa mudança de estado se vincula à última operação realizada pelo algoritmo. No caso dos reconhecedores, a mudança pode depender ou não do último símbolo lido na cadeia e do estado do sistema naquele momento.

O modelo mais simples de reconhecedor é o **autômato de estados finitos**, que identifica se uma cadeia pertence ou não a uma **gramática regular**, que por sua vez permite apenas três operações:

- **Concatenação**: um símbolo se segue a outro. Muitas vezes não utiliza nenhuma simbologia para representar essa operação, mas alguns autores utilizam o sinal de multiplicação (×).
- **Alternância**: um ou outro símbolo de um par serão inseridos em dado ponto da cadeia. Geralmente é indicada pelo sinal de soma (+).
- **Fechamento transitivo**: também chamado de fecho de Kleene, indica a repetição, numa quantidade finita (que pode ser zero), de dado símbolo. Costuma ser indicado por asterisco (*) após o símbolo em questão.

Um autômato de estados finitos terá **estado inicial** – aquele em que todo e qualquer processo de reconhecimento se inicia – e um ou mais **estados de aceitação**. Se, ao término da execução do algoritmo, o estado for um destes, a cadeia será aceita como construída de acordo com a gramática; caso contrário, será rejeitada. Cada autômato finito usado como reconhecedor é dedicado, ou seja, é construído para uma gramática específica, não servindo para reconhecer outras gramáticas.

A seguir, um exemplo de gramática de linguagem regular e seu respectivo autômato:

- O alfabeto da linguagem será composto por três símbolos: {a, b, c}.
- O primeiro símbolo será um a ou b.

- Após o primeiro símbolo, há uma sequência de exatamente dois símbolos b.
- Na sequência, haverá um ou mais símbolos c.

Observação

No estudo de linguagens formais, gramática regular é usualmente representada na forma de uma expressão regular. Para simplificar, utilizaremos aqui essa descrição sequencial da gramática.

Um autômato finito para identificar essa linguagem seria este:

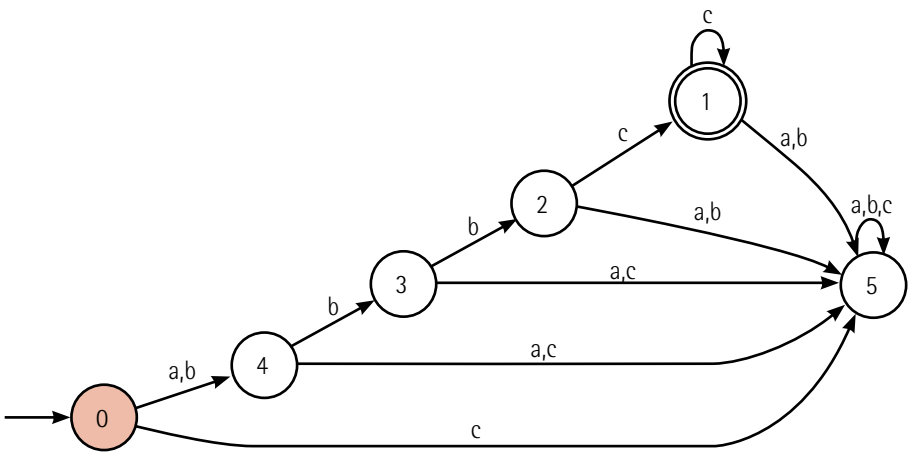


Figura 24 – Autômato de estados finitos para a gramática apresentada

O autômato tem seis estados (indicados pelos nós de 0 a 5), e cada seta indica uma transição condicional; o estado inicial é 0, e o final é 1. Podemos pensar nele como um conjunto de estruturas condicionais encadeadas.

O código em Python a seguir representa esse autômato:

```
alfa=("a","b","c")
estado = 0
cadeia = input("Cadeia a ser testada: ")
for i in cadeia:

    #Teste dos símbolos e dos estados
    #Se a condição for satisfeita, há transição de estado
    if i not in alfa:
      estado = 5
    elif estado == 0 and i in ("a","b"):
      estado = 4
    elif estado == 0 and i == "c":
      estado = 5
    elif estado == 1 and i in ("a","b"):
```

```
        estado = 5
    elif estado == 2 and i in ("a","b"):
        estado = 5
    elif estado == 2 and i == "c":
        estado = 1
    elif estado == 3 and i in ("a","c"):
        estado = 5
    elif estado == 3 and i == "b":
        estado = 2
    elif estado == 4 and i in ("a","c"):
        estado = 5
    elif estado == 4 and i == "b":
        estado = 3

    #Se o estado for 5, não é necessário continuar testando
    if estado == 5:
        break

    #Verificação do estado final
if estado == 1:
    print("Cadeia Aceita")
else:
    print("Cadeia Rejeitada")
```

Para um autômato finito, executaremos um número máximo de n repetições, onde n é o número de símbolos na cadeia. Sendo t o número de transições de estado (na nossa simplificação, o número de condições a testar), teremos a complexidade de tempo O(nt). Do ponto de vista de complexidade de espaço, a execução não demanda um acréscimo do consumo de memória para cadeias maiores (exceto, é claro, para armazenar a cadeia em si); assim, a complexidade de espaço é constante O(1).

Uma versão mais robusta são os autômatos de pilha, nos quais, simplificadamente, a transição de estados se dá inserindo e removendo elementos em uma pilha (é o estado final da pilha que definirá a aceitação ou rejeição da cadeia). A complexidade de tempo se torna análoga à dos autômatos finitos, mas aqui t representa o empilhar/desempilhar na estrutura. Por outro lado, se supormos que no pior cenário todos os elementos da cadeia precisem ser empilhados, teremos uma complexidade de espaço $\Theta(n)$.

Podemos considerar todos os autômatos aqui apresentados como simplificações de uma **máquina de Turing** – modelo teórico concebido pelo matemático britânico Alan Turing (1912-1954) na década de 1930, antes dos primeiros computadores –, que se restringe aos aspectos lógicos básicos de funcionamento de um computador: memória e máquina de estados (e suas transições).

Teoricamente, numa máquina de Turing qualquer algoritmo que envolva processamento linear pode ser realizado (e não processamento paralelo; para mais detalhes sobre o assunto, leia o tópico 7). Dessa forma, qualquer tentativa de definir uma complexidade geral para esse modelo esbarra no fato de que o algoritmo a ser implementado nela definirá a função de complexidade.

4.4 Algoritmo de Boyer-Moore

Um problema comum ao trabalhar com cadeias (principalmente cadeias de texto) é localizar a ocorrência (ou não) de uma cadeia menor dentro de uma maior. O caso mais comum é localizar uma palavra ou frase em um texto maior. Um dos principais algoritmos de busca de expressões é o de Boyer-Moore – nome dado por causa de seus criadores, Robert S. Boyer e J Strother Moore, que o desenvolveram em 1977.

Para entender esse algoritmo, vamos pensar no algoritmo de busca mais plausível, que consistiria em percorrer uma cadeia T (string no qual procuramos o texto) em busca de uma cadeia chamada padrão P (sendo P menor que T, obviamente). Na versão mais simples da busca, percorremos S, caractere por caractere, até achar uma correspondência com o primeiro caractere de W. Então o processo de percorrer T é interrompido, e checa-se a correspondência dos caracteres subsequentes de P com os subsequentes de T. Caso a correspondência seja total, encontrou-se uma ocorrência de W; caso contrário, reinicia-se o processo de percorrer T do ponto onde foi interrompida. Esse algoritmo terá complexidade de tempo O(pt), onde p é o número de caracteres de P, e t é o número de caracteres de T. Essa metodologia é chamada por alguns autores de **busca ingênua**.

O algoritmo de Boyer-Moore faz uma busca mais eficiente, dando saltos ao longo de T baseados no pré-tratamento de P, e o algoritmo começa a alinhar os caracteres iniciais de P e T. Além disso, "se houver uma coincidência entre o último caractere de P e um caractere de T, verifica-se a correspondência entre os caracteres anteriores de P e T, de trás para frente" (Rodrigues *et al.*, 2022, p. 237).

A verificação nesse algoritmo se baseia em dois princípios:

- **Regra do sufixo correto**: supondo uma correspondência do último caractere de P com um de T, percorreremos P até confirmar que P está contido em T, ou até que ocorra um caractere diferente. Os caracteres que corresponderem serão denominados sufixo correto.

- **Regra do caractere ruim**: se no caso anterior houver uma correspondência parcial do final de P, mas ao avançar em direção ao começo ocorrer um caractere discrepante, o caractere de D que não correspondeu é chamado caractere ruim.

No caso de caractere ruim, verifica-se se existe sufixo correto não precedido pelo caractere ruim em algum ponto anterior de P. Se houver, é dado um salto de P para que haja a correspondência desse novo sufixo; senão o salto se dá de forma que o primeiro caractere de P se alinhe com o caractere imediatamente posterior ao que o último estava alinhado.

Para ilustrar o conceito, vejamos um alfabeto de apenas quatro caracteres (A, B, C e D):

Exemplo de aplicação

Vamos aplicar o algoritmo de Boyer-Moore para localizar a cadeia P na cadeia T:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P	D	A	B	C	A	B									

Encontrou-se uma correspondência no último caractere de P, retornando-se ao longo de P até encontrar o primeiro caractere diferente. Assim, o sufixo correto é B, e o caractere ruim é A:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P	D	A	B	C	A	B									

Agora localiza-se em P a próxima ocorrência do sufixo bom precedido pelo caractere ruim; isso será o terceiro caractere de P. A comparação se desloca para que esse sufixo ocupe a posição do anterior:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P				D	A	B	C	A	B						

Repete-se o processo até encontrar o novo sufixo correto e o novo caractere ruim:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P				D	A	B	C	A	B						

Mais uma vez desloca-se para a ocorrência do sufixo bom precedido pelo caractere ruim:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P							D	A	B	C	A	B			

O processo continua conforme descrito:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P							D	A	B	C	A	B			

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P										D	A	B	C	A	B

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P										D	A	B	C	A	B

Por fim, encontra-se uma correspondência exata de P em T:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P										D	A	B	C	A	B

Para facilitar o processo de localizar uma ocorrência do sufixo bom não precedido pelo caractere ruim em D, muitas implementações desse algoritmo usam uma tabela indexada dos caracteres do alfabeto de D, o que é muito útil se D for longo; se for um padrão curto, uma busca ingênua simples pode ser aplicada sem o comprometimento da eficiência do algoritmo como um todo.

Se o padrão ocorrer no texto, a complexidade de tempo do algoritmo é O(pt) no pior caso, que seria a ocorrência de sufixo longo; ou seja, os saltos serão menores. Já no caso de ocorrência de P em T, a complexidade cai para O(p + t). Uma curiosidade desse algoritmo: quanto maior for P, maiores serão os saltos e mais rápida será a execução do algoritmo.

O programa em Python a seguir executa esse algoritmo usando uma tabela para agilizar a busca do sufixo bom precedido pelo caractere ruim dentro de P:

```
#Construção da tabela de indexação do padrão

def tabela_caractere_ruim(padrao):
    tabela = {}
    for i in range(len(padrao) - 1):
        tabela[padrao[i]] = len(padrao) - 1 - i
    return tabela
#Execução do algoritmo
def boyer_moore(padrao, texto):
    indices = []
    m, n = len(padrao), len(texto)
    tabela_ruim = tabela_caractere_ruim(padrao)
    salto = 0
    while salto <= n - m:
      i = m - 1
    while i >= 0 and padrao[i] == texto[salto + i]:
        i -= 1
      if i < 0:
        indices.append(salto)
        if salto + m < n:
          salto += m - 1 - i
        else:
          salto += 1
        else:
          salto += max(1, i - tabela_ruim.get(texto[salto + i], -1))
    return indices
#Programa principal
padrao = input("Digite o padrão (P): ")
texto = input("Digite o texto (T): ")
indices_encontrados = boyer_moore(padrao, texto)
if indices_encontrados:
    print(f"\nO padrão foi encontrado nos seguintes índices: {indices_encontrados}")
else:
    print("\nO padrão não foi encontrado no texto.")
```

4.5 Algoritmo KMP

Assim como o algoritmo de Boyer-Moore, o de Knuth-Morris-Pratt (KMP) procura a ocorrência de uma cadeia W dentro de uma cadeia S. Esse algoritmo utiliza a seguinte técnica: quando se encontra parte de W em S, mas verifica-se que não há correspondência completa (por exemplo, estamos procurando a palavra **algoritmo** num texto maior e encontramos apenas **algo**), a palavra procurada contém em si a informação necessária para determinar onde começar a próxima comparação.

Observação

O algoritmo KMP foi idealizado em 1977 por James Morris e descoberto independemente por Donald Knuth. Os dois se juntaram a Vaughan Pratt e publicaram um relatório que o definia; daí o algoritmo levar a primeira letra do sobrenome de cada um dos envolvidos.

Como na busca ingênua, esse algoritmo compara sequencialmente os caracteres da esquerda para a direita. Sua principal característica está no pré-processamento da cadeia, que fornecerá a informação necessária para definir em que ponto a busca recomeça em caso de incompatibilidade. Assim o algoritmo não reexamina os caracteres já vistos, reduzindo o número de comparações. Na prática podemos considerar a busca e identificação do caractere discrepante como um autômato finito.

Exemplo de aplicação

Vamos aplicar esse exemplo nas mesmas cadeias usadas para o algoritmo de Boyer-Moore, mas agora utilizando o algoritmo KMP:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P	D	A	B	C	A	B									

Após o primeiro movimento, há uma correspondência:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P		D	A	B	C	A	B								

Logo no caractere seguinte há uma discrepância; porém, como o caractere discrepante é exatamente o caractere do início de P, ele avança:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P			D	A	B	C	A	B							

Houve então uma correspondência de três caracteres até surgir um caractere discrepante. No intervalo analisado, como não foi encontrado nenhum caractere correspondente ao início de P, não há sentido em continuar a busca nesse intervalo. Assim, ele salta para imediatamente após o intervalo testado e continua procurando pelo início do padrão:

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P							D	A	B	C	A	B			

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P								D	A	B	C	A	B		

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P									D	A	B	C	A	B	

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P										D	A	B	C	A	B

T	A	D	D	A	B	B	A	A	B	D	A	B	C	A	B
P										D	A	B	C	A	B

Esse algoritmo tem uma complexidade de tempo O(n), porém a complexidade envolvida no pré-processamento e a memória consumida o tornam, na prática, mais eficiente para buscas em alfabetos menores. Sua principal função é buscar sequências específicas em base de DNA, já que estamos lidando com alfabeto de apenas quatro letras (A, C, G e T) (Cormen, 2013, p. 114).

O código em Python a seguir executa o algoritmo KMP. A função **calcular_prefixo** vai registrando os caracteres observados durante a verificação para definir o salto subsequente.

```
#Função para verificar o prefixo
def calcular_prefixo(padrao):
    m = len(padrao)
    prefixo = [0] * m
    j = 0
    for i in range(1, m):
       while j > 0 and padrao[i] != padrao[j]:
          j = prefixo[j - 1]
       if padrao[i] == padrao[j]:
          j += 1
       prefixo[i] = j
    return prefixo

#Busca e verificação
def kmp(padrao, texto):
    indices = []
```

```
    prefixo = calcular_prefixo(padrao)
    m, n = len(padrao), len(texto)
    j = 0
    for i in range(n):
      while j > 0 and texto[i] != padrao[j]:
        j = prefixo[j - 1]
      if texto[i] == padrao[j]:
        j += 1
      if j == m:
        indices.append(i - m + 1)
        j = prefixo[j - 1]
    return indices

#Programa principal
padrao = input("Digite o padrão (P): ")
texto = input("Digite o texto (T): ")
indices_encontrados = kmp(padrao, texto)
if indices_encontrados:
    print(f"\nO padrão foi encontrado nos seguintes índices: {indices_encontrados}")
else:
    print("\nO padrão não foi encontrado no texto.")
```

Resumo

Dada a diversidade de computadores hoje e suas diferentes características, não se pode mais comparar a eficiência entre diferentes algoritmos para resolver um mesmo problema diretamente a partir do tempo real de execução, por causa da influência do ambiente sobre o algoritmo executado. Assim, para comparar algoritmos, utilizamos a complexidade de tempo, uma função assintótica que define a quantidade de operações que um algoritmo irá executar em função do número de entradas que recebe. Existem as notações Grande-O, Ômega e Theta, que permitem indicar a complexidade de um algoritmo individual ou comparar algoritmos entre si. De forma semelhante, usamos a complexidade de espaço para indicar quanta memória um algoritmo consome durante sua execução (esta sofre menos influência do ambiente).

Muitos algoritmos são recursivos, ou seja, instanciam a si mesmos durante a execução, sendo, portanto, mais fáceis de implementar, mas usualmente apresentam maior complexidade de espaço devido à sua natureza. Uma metodologia que usa recursão para reduzir a complexidade de tempo de um algoritmo é a **programação dinâmica**, que divide um problema em subproblemas mais simples, resolvidos recursivamente.

Dentre as diferentes estruturas empregadas em ciência de dados, temos as **árvores**, que são ramificadas; diversos algoritmos de busca são facilitados por dados organizados assim. Dentre as principais aplicações de árvores, temos os heaps, uma forma de representar uma fila de prioridades. São **árvores binárias** (cada nó tem no máximo duas ramificações) balanceadas, cujos elementos são inseridos pelas folhas e removidos pela raiz (seguidas, em ambos os casos, por um ajuste da posição dos elementos para manter a característica da estrutura). Numa árvore, a complexidade de tempo e espaço se vincula ao número de elementos na estrutura e à forma como a árvore é construída, do ponto de vista de suas ramificações.

Outro modelo, ainda mais empregado que as árvores, são as cadeias, que são estruturas de dados lineares e sequenciais – textos, filas, pilhas e listas são exemplos. Temos as operações básicas de remoção, inserção, substituição e movimentação dos elementos dentro da estrutura, cujas complexidades se associam ao tamanho da cadeia. Também temos algoritmos para organizar elementos dentro da cadeia, outros para verificar se uma cadeia é válida (autômatos) e outros ainda para buscar subcadeias em uma cadeia maior. As complexidades de tempo e espaço desses algoritmos dependem do algoritmo em questão, mas também estão sempre vinculadas ao tamanho da cadeia.

Exercícios

Questão 1. (UFMG 2019, adaptada) Considere o trecho de código a seguir para multiplicar matrizes quadradas n × n.

```
for(i=0; i<n; i++){
     for(j=0; j<n; j++){
        for(k=0; k<n; k++){
           m[i][j] += m1[i][k]*m2[k][j]
        }
     }
}
```

Qual é a complexidade de pior caso desse algoritmo?

A) $O(n^2)$

B) $O(n^3)$

C) $O(n)$

D) $O(\log n)$

E) $O(1)$

Resposta correta: alternativa B.

Análise da questão

O trecho de código do enunciado tem três laços **for** aninhados, com o intuito de multiplicar matrizes quadradas n × n, em que n representa a ordem das matrizes m1 e m2 (que atuam como operandos), bem como a ordem da matriz m (resultante).

Conforme estudamos, algoritmos para multiplicar matrizes de tamanho n × n têm complexidade de tempo polinomial cúbica. Portanto, a complexidade de pior caso desse algoritmo é $O(n^3)$.

- O primeiro laço (externo) percorre as linhas da matriz resultante, o que é feito n vezes.
- O segundo laço (intermediário) percorre as colunas da matriz resultante, o que é feito n vezes.
- O terceiro laço (interno) percorre os elementos das colunas da matriz m1 e das linhas da matriz m2, também n vezes.

Em cada iteração do laço interno há uma multiplicação e uma adição, portanto o número total de operações é $n \times n \times n$, resultando em complexidade de $O(n^3)$.

Questão 2. (Consulpam 2023, adaptada) Algoritmo de ordenação é aquele que organiza os dados de uma cadeia com base em dois elementos: comparações e movimentações.

Sobre os algoritmos de ordenação, avalie as afirmativas.

I – Selection Sort é o método mais primitivo de ordenação de um vetor. A ideia desse método é percorrer um vetor de n posições n vezes, a cada vez comparando dois elementos e trocando-os caso o primeiro seja maior que o segundo.

II – Bubble Sort é uma forma intuitiva de ordenar um vetor, escolhendo o menor elemento do vetor e trocando-o com o primeiro elemento. Em seguida, escolhe-se o menor entre os restantes para trocá-lo com o segundo elemento e assim por diante, até o último elemento do vetor.

III – A complexidade do método Insertion Sort varia entre $O(n)$ e $O(n^2)$, dependendo do estado original dos dados.

É correto o que se afirma em:

A) I, apenas.

B) III, apenas.

C) I e II, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa B.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: o Selection Sort escolhe o menor elemento do vetor e coloca-o na primeira posição, depois seleciona o segundo menor elemento e coloca-o na segunda posição, e assim sucessivamente, até o final da cadeia que está sendo ordenada.

II – Afirmativa incorreta.

Justificativa: Bubble Sort é um algoritmo iterativo de fácil implementação que compara e troca elementos adjacentes até que a lista esteja ordenada. O texto apresentado descreve a atuação do Selection Sort.

III – Afirmativa correta.

Justificativa: o Insertion Sort percorre o vetor de uma extremidade a outra e, conforme avança, vai ordenando os elementos na extremidade inicial, inserindo cada elemento na posição correspondente. A complexidade desse método varia entre $O(n)$ e $O(n^2)$, dependendo do estado original dos dados. No melhor caso o algoritmo terá complexidade $O(n)$; no pior, terá complexidade $O(n^2)$.