

Computabilidade e Complexidade de Algoritmos



Cruzeiro do Sul Virtual
Educação a distância

Material Teórico



Análise de Desempenho de Algoritmos

Responsável pelo Conteúdo:

Prof. Dr. Luciano Rossi

Revisão Textual:

Prof.^a Esp. Kelciane da Rocha Campos

UNIDADE

Análise de Desempenho de Algoritmos



- **Algoritmos de Ordenação;**
- **Algoritmos de Busca;**
- **Algoritmos Sobre Grafos.**



OBJETIVO DE APRENDIZADO

- Apresentar ao aluno os algoritmos clássicos para ordenação, busca e percurso em grafos, bem como as respectivas análises de desempenho.



Orientações de estudo

Para que o conteúdo desta Disciplina seja bem aproveitado e haja maior aplicabilidade na sua formação acadêmica e atuação profissional, siga algumas recomendações básicas:



Assim:

- ✓ Organize seus estudos de maneira que passem a fazer parte da sua rotina. Por exemplo, você poderá determinar um dia e horário fixos como seu “momento do estudo”;
- ✓ Procure se alimentar e se hidratar quando for estudar; lembre-se de que uma alimentação saudável pode proporcionar melhor aproveitamento do estudo;
- ✓ No material de cada Unidade, há leituras indicadas e, entre elas, artigos científicos, livros, vídeos e sites para aprofundar os conhecimentos adquiridos ao longo da Unidade. Além disso, você também encontrará sugestões de conteúdo extra no item **Material Complementar**, que ampliarão sua interpretação e auxiliarão no pleno entendimento dos temas abordados;
- ✓ Após o contato com o conteúdo proposto, participe dos debates mediados em fóruns de discussão, pois irão auxiliar a verificar o quanto você absorveu de conhecimento, além de propiciar o contato com seus colegas e tutores, o que se apresenta como rico espaço de troca de ideias e de aprendizagem.

Algoritmos de Ordenação

Vimos, na unidade anterior, que os algoritmos podem ser classificados de acordo com seus respectivos desempenhos. Além disso, tivemos um primeiro contato com a notação assintótica, a qual utilizamos para classificar os algoritmos de acordo com o desempenho apresentado.

Nesta unidade, daremos continuidade ao estudo do desempenho dos algoritmos. Vamos considerar diferentes algoritmos para a realização de suas respectivas análises. Inicialmente, vamos continuar a analisar a classe dos algoritmos de ordenação. Anteriormente, vimos como é feita uma análise de desempenho, considerando o *Insertion Sort*, onde cada instrução que compõe o algoritmo foi avaliada.

Para relembrarmos o processo de análise, vamos considerar o algoritmo *Bubble Sort*. A complexidade desse algoritmo é $O(n^2)$, para o pior caso, que é a mesma complexidade vista para o algoritmo *Insertion Sort*. Lembre-se de que para cada instrução, consideraremos um tempo de execução associado e a quantidade de vezes que a instrução é realizada. Uma versão possível do *Bubble Sort* é descrita no Algoritmo 1; para simplificar a análise, considere que $n = |V|$, ou seja, n é o tamanho do vetor V .

Algoritmo 1 – Algoritmo de ordenação pelo método de bolhas (*Bubble Sort*)

BubbleSort(V)

- | | | |
|--|-------|--------------------------|
| 1. para $i \leftarrow 0$ até $n - 1$ | c_1 | $n + 1$ |
| 2. para $j \leftarrow n - 1$ até $i + 1$ | c_2 | $\sum_{t=0}^n t_j$ |
| 3. se $V[j] < V[j - 1]$ | c_3 | $\sum_{t=0}^n (t_j - 1)$ |
| 4. trocar $V[j]$ e $V[j - 1]$ | c_4 | $\sum_{t=0}^n (t_j - 1)$ |

Para cada linha de instrução temos o respectivo tempo de execução (c_i) e o número de vezes que é executada

Fonte: adaptado de CORMEN (2009)

A linha 1 do *Bubble Sort* descreve o laço externo, governado pela variável i , que é responsável por percorrer todo o vetor ($V[0, \dots, n-1]$). O laço apresenta um tempo de execução para cada iteração, correspondente à constante c_1 , e a quantidade de vezes que ele será executado é igual a $n + 1$. Veja que a unidade adicional ao valor de n corresponde ao último teste do critério de parada, que, para esse caso, corresponde a $i \leq n - 1$.

O laço interno (linha 2), governado pela variável j , segue a mesma lógica anterior, porém t_i corresponde à quantidade de vezes que a instrução será executada para cada iteração do laço externo (i). Assim, considerando somente o pior caso, temos que $t_i = i$ para $i = [0, 1, \dots, n - 1]$. Nesse sentido, para $i = 0$ a linha 2 será executada $n - 1$ vezes, para $i = 1$ a linha será executada $n - 2$ vezes, e assim por diante até que para $i = n - 1$ a linha 2 será executada 1 vez. O número de vezes que a linha 2 será executada é igual a $(n - 1) + (n - 2) + \dots + 1$.

Seguindo a mesma lógica, a condicional da linha 3 será executada uma vez menos que a linha anterior. Desse modo, temos que o total de vezes que a linha 3 será executada é igual a $(n - 2) + (n - 1) + \dots + 0$.

As somatórias que descrevem a quantidade de vezes que as instruções serão executadas, nas linhas 2, 3 e 4, podem ser resolvidas da seguinte forma:

$$\sum_{i=0}^n t_i = (n-1) + (n-2) + \dots + 1 = \frac{(n+1)n}{2},$$

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

Veja na unidade anterior como essas somatórias podem ser resolvidas.

O tempo de execução do algoritmo *Bubble Sort* é uma função $T(n)$ obtida a partir da somatória dos produtos do custo de cada instrução pelo número de vezes que ela é executada. Assim, temos:

$$T(n) = c_1(n+1) + c_2\left(\frac{(n+1)n}{2}\right) + c_3\left(\frac{(n+1)(n-2)}{2}\right) + c_4\left(\frac{(n+1)(n-2)}{2}\right),$$

$$T(n) = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right)n + (c_1 - c_3 - c_4).$$

Podemos substituir as constantes multiplicadoras das variáveis por novas constantes da seguinte forma:

$$a = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right),$$

$$b = \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right),$$

$$c = (c_1 - c_3 - c_4).$$

Desse modo, temos que a função $T(n) = an^2 + bn + c$, ou seja, um polinômio de grau 2.

Na unidade anterior, nós vimos que o conjunto $O(g(n))$ é descrito por:

$$O(g(n)) = \{f(n) | \exists c, n_0 > 0 \text{ sendo } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

Assim, temos que encontrar valores positivos não nulos para c e n_0 , de modo que $0 \leq f(n) \leq cg(n)$ e $n \geq n_0$. Para identificar os valores de c e n_0 vamos considerar, a título de exemplo, que os coeficientes do polinômio são $a = b = c = 1$. Veja que esses coeficientes referem-se aos tempos de execução das instruções e a atribuição de valor a esses coeficientes somente é necessária para a prova de que $n^2 + n + 1 \in O(n^2)$, sendo $f(n) = n^2 + n + 1$ e $g(n) = n^2$.

Nesse contexto, temos que mostrar que $n^2 + n + 1 \leq cn^2$, da seguinte forma:

$$n^2 + n + 1 \leq cn^2,$$

$$\frac{n^2 + n + 1}{n^2} \leq c.$$

$$1 + \frac{n+1}{n^2} \leq c.$$

Tomando $n = 1$, temos que $c \geq 3$, assim podemos dizer que $n^2 + n + 1$ é $O(n^2)$ pois existe $n_0 = 1$ e $c = 3$, tal que $n^2 + n + 1 \leq cn^2$.

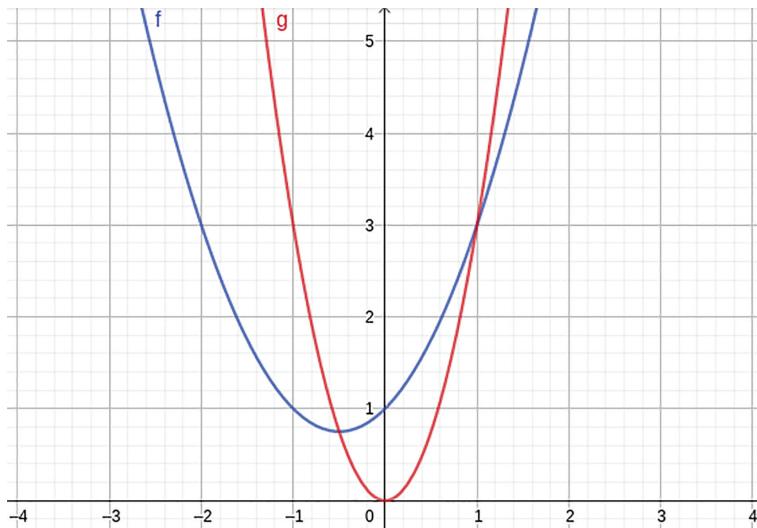


Figura 1 – Gráfico das funções $f(n) = n^2 + n + 1$ e $g(n) = 3n^2$,
 $g(n)$ é um limite superior para $f(n)$, para $n \geq 1$

Fonte: Acervo do conteudista

A Figura 1 apresenta o gráfico das funções $f(n)$ e $g(n)$, representadas pelas cores azul e vermelho, respectivamente. Veja que a partir de $n = 1$ (eixo horizontal no gráfico), a curva de $g(n)$ supera a curva de $f(n)$, o que se mantém infinitamente. Com essa representação, é possível visualizar que a afirmação de que $f(n) \in O(g(n))$ é verdadeira.

Vamos considerar um exemplo mais prático. Suponha que temos dois algoritmos que realizam a mesma tarefa. Os algoritmos realizam quantidades diferentes de operações, que são representadas por funções da seguinte forma:

- o algoritmo A realiza $f_1(n) = 2n^2 + 5n$ operações; e
- o algoritmo B realiza $f_2(n) = 500n + 4000$ operações.

Nesse contexto, qual algoritmo deveríamos escolher?

Para o caso de uma instância de entrada com tamanho $n = 10$, teríamos: $f_1(10) = (2 \times 10^2) + (5 \times 10) = 250$ operações e $f_2(10) = (500 \times 10) + 4000 = 9000$ operações. Aqui é simples ver que o algoritmo A realiza menos operações e deve ser escolhido. Porém, devemos lembrar que estamos interessados em valores de n grandes, ou seja, um comportamento assintótico. Vamos considerar agora uma instância de tamanho $n = 100$; assim, $f_1(100) = (2 \times 100^2) + (5 \times 100) = 20500$ operações e $f_2(100) = (500 \times 100) + 4000 = 9000$ operações.

Veja que para o segundo exemplo o algoritmo B realiza menos operações, em oposição ao primeiro exemplo. Na verdade, estamos tratando de valores grandes para n , ou seja, $n \rightarrow \infty$ (leia n tendendo ao infinito). Desse modo, podemos afirmar que $f_1(n) \in O(g(n^2))$ e $f_2(n) \in O(g(n))$.

Uma consideração importante a respeito do método de análise é que, como estamos interessados em valores grandes, os polinômios que descrevem o crescimento do tempo de execução de acordo com o tamanho da instância têm a maior parte de seus componentes descartados. Por exemplo, para $f(n) = 2n^2 + 30n + 12$ vimos que esta função é da ordem de $O(n^2)$; assim, veja que somente consideramos a variável de maior grau, descartando todo o restante.

Nesse contexto, contar o número de vezes que as instruções são executadas, como fizemos no exemplo para o *Bubble Sort*, é trabalhoso e a precisão obtida não se reflete na classificação. Assim, vamos analisar os algoritmos, daqui em diante, considerando uma menor precisão, tendo sempre em foco a notação assintótica vista na unidade anterior.

Podemos ter a impressão de que os algoritmos de ordenação têm sempre a mesma eficiência, ou falta dela, pois todos os algoritmos vistos até aqui foram classificados como $O(n^2)$. Porém, existem outros algoritmos de ordenação que são mais eficientes que isso. O algoritmo que utiliza o método da intercalação, conhecido por *Merge Sort*, é um desses algoritmos mais eficientes que estudaremos a partir de agora.

Vamos começar com uma analogia, que já utilizamos anteriormente, sobre cartas de baralho. Suponha que temos duas pilhas de cartas ordenadas, onde as cartas menores estão no topo da pilha e as maiores na base, e queremos combiná-las em uma única pilha, também ordenada. Veja que temos uma pilha da qual retiraremos cartas com a mão esquerda e outra na qual utilizaremos a mão direita. Primeiro retiramos uma carta de cada pilha e as comparamos. Caso a menor das cartas seja da pilha esquerda, ela será inserida na pilha principal e retiraremos outra carta da pilha esquerda. O mesmo ocorre se a menor carta for da pilha direita, ela entra na pilha principal e retiramos outra carta da pilha direita.

O processo descrito é repetido até que todas as cartas nas pilhas esquerda e direita estejam na pilha principal. O resultado desse processo é a ordenação das cartas na pilha principal, com as menores cartas na base da pilha e as maiores no topo. Nesse sentido, o número de operações realizadas é igual à quantidade de cartas existentes.

O Algoritmo 1 descreve os passos para o procedimento *Merge()*. Note que esse procedimento é o mesmo apresentado para a analogia das cartas de baralho. O algoritmo recebe um vetor V , a posição inicial p , a posição final r e a posição q que divide o vetor em duas partes ordenadas. As linhas 1 e 2 determinam os tamanhos dos vetores auxiliares E e D , os quais representam as pilhas de cartas direita e esquerda. Veja que, na linha 3, os dois vetores auxiliares são criados com uma posição adicional.

Os laços das linhas 4 e 6 fazem a transferência dos valores do vetor principal V para os vetores auxiliares E e D . Assim, temos dois vetores ordenados que serão reposicionados no vetor principal em ordem crescente.

Algoritmo 1 – algoritmo de ordenação pelo método de intercalação (*Merge Sort*)

Merge(V, p, q, r)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. *sejam* $E[0\dots n_1]$ e $D[0\dots n_2]$ novos vetores
4. **para** $i \leftarrow 0$ **até** $n_1 - 1$
5. $E[i] \leftarrow V[p + i]$
6. **para** $j \leftarrow 0$ **até** $n_2 - 1$
7. $D[j] \leftarrow V[q + j + 1]$
8. $E[n_1] \leftarrow \infty$
9. $D[n_2] \leftarrow \infty$
10. $i \leftarrow 0$
11. $j \leftarrow 0$
12. **para** $k \leftarrow p$ **até** r
13. *se* $E[i] \leq D[j]$
14. $V[k] \leftarrow E[i]$
15. $i \leftarrow i + 1$
16. *senão*
17. $V[k] \leftarrow D[j]$
18. $j \leftarrow j + 1$

Fonte: Adaptado de CORMEN (2009)

Esse procedimento é parte do algoritmo completo. Nas linhas 8 e 9, são inseridas sentinelas nas últimas posições dos vetores auxiliares. Nesse caso, as sentinelas são o valor ∞ (infinito) que será útil para marcar o final de cada um dos vetores. As linhas 10 e 11 atribuem zero às variáveis i e j , indicando os inícios dos vetores esquerdo e direito, respectivamente.

O laço da linha 12 é responsável por mapear todas as posições, da primeira à última, no vetor original. Além disso, há uma comparação entre os valores das posições iniciais de E e D , o menor entre os valores armazenados nessas posições é selecionado para ocupar a posição indicada pelo laço. Desse modo, tanto o valor selecionado quanto a posição no vetor original têm seus indexadores incrementados em uma unidade. O processo se repete até que todos os valores estejam ordenados no vetor original.

O algoritmo *Merge()* gasta tempo proporcional a $\Theta(n)$, com $n = r - p + 1$. Veja que as instruções nas linhas 1 a 3 e 8 a 11 levam um tempo constante para a execução. Isso significa que, independentemente do tamanho do vetor, o tempo gasto por essas instruções é sempre o mesmo. Por outro lado, os laços descritos nas linhas 4 a 7 levam o tempo $\Theta(n_1 + n_2) = \Theta(n)$ e o laço da linha 12 faz n operações, cada uma em tempo constante. Portanto, o procedimento *Merge()* leva tempo $\Theta(n)$.

O algoritmo de ordenação por intercalação é definido por um procedimento *MergeSort()* que utiliza o procedimento anterior *Merge()* como sub-rotina. A estratégia considerada por esse algoritmo é a de dividir o vetor em dois subvetores, sucessivamente, até que haja somente um elemento em cada subvetor. Nesse ponto, os subvetores são rearranjados, por meio do procedimento *Merge()*, até que o vetor original seja obtido com seus elementos ordenados.

Algoritmo 2 – algoritmo de ordenação pelo método de intercalação (*Merge Sort*)

MergeSort(V, p, r)

1. *se* $p < r$
2. $q \leftarrow (p+r)/2$
3. *MergeSort(V, p, q)*
4. *MergeSort(V, q+1, r)*
5. *Merge(V, p, q, r)*

Fonte: Adaptado de CORMEN (2009)

Esse procedimento é parte do algoritmo completo. O procedimento *MergeSort()* é descrito no Algoritmo 2. Inicialmente, o procedimento verifica se há mais que um elemento em V , por meio da condicional $p < r$, e caso haja, o vetor é subdividido em dois subvetores que serão submetidos, recursivamente, ao procedimento *Merge()*. Essa é a etapa da divisão que consiste em calcular o índice q que divide o vetor $V[p, \dots, r]$ em dois subvetores $V[p, \dots, q]$ e $V[q+1, \dots, r]$, ambos com $[n/2]$ e $[n/2]$ elementos, respectivamente.



A expressão $[x]$ indica o menor inteiro maior ou igual a x e $\lceil x \rceil$ indica o maior inteiro menor ou igual a x . Supondo $x \in \mathbb{R}$, o piso de x ($[x]$) arredonda o valor de x para baixo e o teto de x ($\lceil x \rceil$) arredonda o valor de x para cima.

Veja que as duas chamadas recursivas ao procedimento *MergeSort()* dividem o vetor original em dois subvetores com metade de seu tamanho. Quando o tamanho dos subvetores for igual a um, o procedimento *Merge()* fará a intercalação dos subvetores, de modo que eles estejam ordenados. A Figura 2 apresenta a árvore de recursão que descreve esse processo.

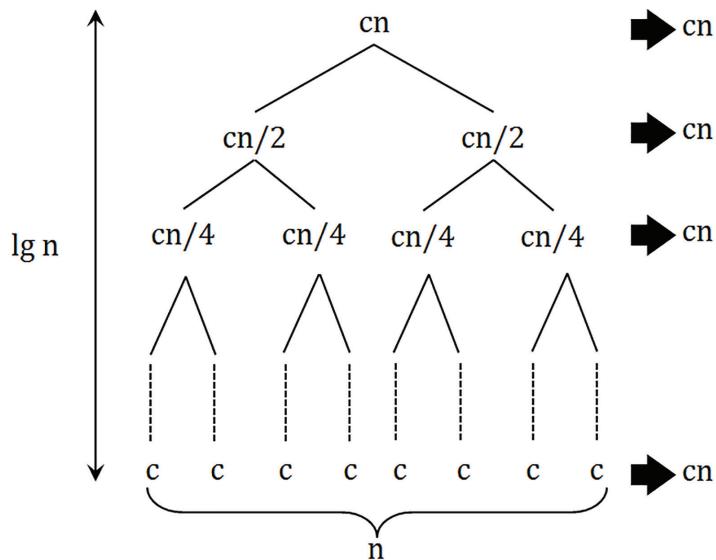


Figura 2 – Árvore de recursão ilustrativa da quantidade de operações realizadas pelo algoritmo *MergeSort()*

Fonte: Adaptada de CORMEN, 2009

O procedimento *Merge()* consome um tempo proporcional a $\Theta(n)$, conforme foi anteriormente descrito. Desse modo, o primeiro nível da árvore de recursão descreve a intercalação dos subvetores, no nível inferior, e vai consumir o equivalente a uma constante c para cada elemento na instância de entrada (n). Ou seja, o processo de intercalação de dois subvetores de tamanho $n/2$ consome um tempo cn .

Considerando o segundo nível da árvore de recursão, é possível observar que existem duas instâncias, de tamanho $n/2$, que serão intercaladas. Assim, cada uma dessas instâncias consumirá tempo equivalente a $cn/2$ e, considerando a soma desses tempos, temos que $cn/2 + cn/2 = cn$. O mesmo ocorre com o terceiro nível na árvore de recursão, onde temos quatro instâncias, cada uma com tamanho igual a $n/4$. Veja que $cn/4 + cn/4 + cn/4 + cn/4 = cn$.

O padrão observado se repete até que cada instância tenha um tamanho igual a um. Essa ocorrência corresponde ao último nível na árvore de recursão. Como cada instância tem tamanho um, o tempo consumido por cada uma delas será representado pela constante c ; desse modo, teremos n instâncias no último nível da árvore. Veja que nesse nível, também teremos cn como resultado da soma dos tempos para cada instância.

A conclusão de que cada nível na árvore de recursão consome tempo equivalente a cn nos leva a pensar sobre quantos níveis existem na árvore. Veja que o tempo de execução total para a ordenação por intercalação será a soma dos tempos em cada nível na árvore.

A altura de uma árvore de recursão é o número de desdobramentos que é executado na entrada. No contexto do procedimento *Merge()*, que divide a entrada em duas partes, para o caso de um vetor com dois elementos, ele seria dividido em outros dois vetores com um elemento cada e a árvore de recursão teria altura igual a um com dois níveis. Assim, a altura da árvore de recursão será $\lg n$ e o número de níveis na árvore será $\lg n + 1$. Veja que $\lg n$ é equivalente a $\log_2 n$ e que $\log_2 n = x \Rightarrow 2^x = n$, ou seja, queremos saber qual o valor do expoente para a base 2 que resultará em n . No nosso contexto, n é o tamanho da instância. Vamos supor que tenhamos um único elemento a ser ordenado;

assim, $n = 1$. Nesse caso, $\lg n + 1 = 1$, indicando que a árvore de recursão tem somente um nível. Supondo que o número de elementos a serem ordenados é igual a oito, temos $\lg 8 + 1 = 4$ níveis na árvore de recursão.

A partir da identificação da altura da árvore de recursão, e de seus respectivos níveis, podemos concluir que o tempo de execução do algoritmo de ordenação por intercalação é $cn(\lg n + 1)$, onde cn é referente ao consumo de tempo em cada nível e $\lg n + 1$ é referente ao número de níveis na árvore. Melhorando a descrição anterior, podemos afirmar que $cn(\lg n + 1) = cn \lg n + cn$; assim, ignorando o termo de menor ordem e a constante c , temos $\Theta(n \lg n)$. O algoritmo de ordenação por intercalação tem um desempenho, em termo de tempo de execução, melhor que os algoritmos vistos anteriormente.

A Figura 3 mostra um gráfico comparativo entre as funções $f(n) = n^2$, $f(n) = n \lg n$ e $f(n) = n$, que representam os tempos de execução dos algoritmos *Bubble Sort*, *Merge Sort* e um algoritmo de tempo linear que veremos no decorrer do curso, respectivamente. A curva na cor vermelha representa o crescimento do tempo de execução, em função do tamanho da entrada, para algoritmos de tempo polinomial de grau dois, como é o caso do *Bubble Sort*. Veja que essa curva supera a curva verde, referente ao algoritmo *Merge Sort*. Se considerarmos, a título de exemplo, uma instância de entrada de tamanho oito, teremos para o *Bubble Sort* 64 instruções executadas, enquanto que para o *Merge Sort* teremos apenas 24 instruções. Se considerarmos um algoritmo de tempo linear, representado pela curva verde no gráfico, o número de instruções seria igual a oito.

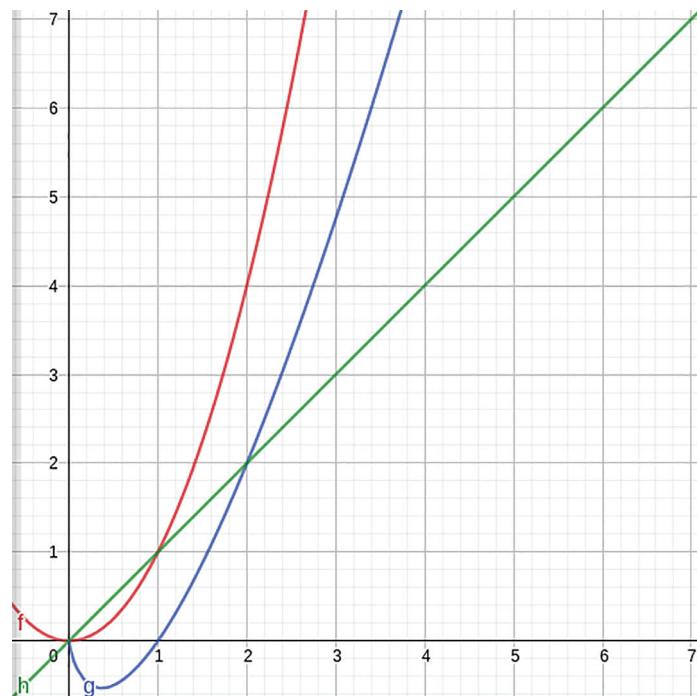


Figura 3 – Gráfico comparativo entre as funções $f(n) = n^2$, $f(n) = n \lg n$ e $f(n) = n$, representadas pelas curvas vermelha, azul e verde, respectivamente



Considere uma instância de entrada de tamanho igual a 100 e que todas as instruções do algoritmo consomem tempo igual a 0,001 segundo. Quanto tempo os algoritmos *Bubble Sort* e *Merge Sort* levariam para ordenar a sequência?

Algoritmos de Busca

Os algoritmos de busca ou pesquisa têm por objetivo verificar a existência de um determinado elemento em um conjunto de elementos. Existem diferentes algoritmos de busca que podem ser considerados, como, por exemplo, as buscas sequencial e binária, as árvores de busca binária e as tabelas *hash*. Essas estratégias de busca são discutidas nas disciplinas de estruturas de dados lineares e não lineares.

Nesta unidade, vamos nos concentrar na complexidade computacional dos algoritmos de busca. Mais especificamente, vamos estudar a estratégia e a complexidade da busca binária. Essa estratégia é muito útil quando temos um conjunto de elementos previamente ordenados.

Para compreendermos o funcionamento do algoritmo de busca binária, vamos utilizar, novamente, uma analogia. Atualmente, quando precisamos encontrar o número do telefone de alguma pessoa ou empresa podemos recorrer a uma pesquisa na *internet* pelo seu respectivo nome. Antigamente, quando não existia essa possibilidade, utilizávamos uma lista telefônica, a qual disponibilizava, em forma de um catálogo com muitas páginas, o nome, o endereço e o número do telefone das pessoas cadastradas. Essa lista era ordenada em ordem alfabética pelo nome das pessoas.

A pesquisa na lista telefônica consistia de, sabendo-se o nome da pessoa de interesse, abrir a lista mais ou menos ao meio e verificar se o nome buscado estava na parte anterior ou na posterior da lista. Por exemplo, se buscamos por Maria e abrimos a lista na letra *F*, sabíamos que o nome buscado estaria na parte posterior (páginas do lado direito) da lista, pois *M* vem depois de *F* no alfabeto. Assim, agora considerando somente as páginas da parte posterior selecionada, repetímos o processo até encontrar a página que continha o nome da pessoa de interesse.

A grande vantagem da busca binária está em que, a cada passo, reduzimos o tamanho do conjunto de elementos pela metade. Se a nossa lista tivesse mil páginas, no primeiro passo nós eliminariamos 500 páginas, ou seja a metade do total, e repetiríamos o processo. No segundo passo, restariam 250 páginas, e assim sucessivamente até identificarmos a página com o nome da pessoa de interesse. Ou seja, a cada passo o conjunto é dividido por dois, daí o nome busca binária.

Considere o Algoritmo 3, no qual há a descrição das instruções que compõem a função *BuscaBinaria()*. Essa função recebe um vetor de elementos ordenados *V*, os indexadores referentes à primeira (*p*) e última (*q*) posições do vetor e ao elemento de busca *x*. Inicialmente, verifica-se se a primeira posição é maior que a última; veja que isso indica um vetor vazio e, no nosso contexto, a função não encontrou o elemento de busca *e*, assim, retornará o valor *NULL*.

Algoritmo 3 – Algoritmo de busca binária

BuscaBinaria(V, p, q, x)

1. **se** $p > q$
2. **retorna** *NULL*
3. **senão**
4. $m \leftarrow (p + q) / 2$

5. *se* $x = V[m]$
6. *retorna* m
7. *senão*
8. *se* $x < V[m]$
9. *retorna* $\text{BuscaBinaria}(V, p, m-1, x)$
10. *senão*
11. *retorna* $\text{BuscaBinaria}(V, m+1, q, x)$

O próximo passo consiste em dividir o vetor ao meio (linha 4) e, caso o elemento nessa posição seja o elemento objetivo (linha 5), a função retornará à sua posição correspondente (linha 7). Caso o elemento não seja aquele buscado, verifica-se se o elemento objetivo é menor que o elemento na posição central (linha 8) e, caso seja, a função é chamada recursivamente, tendo como parâmetro o subvetor esquerdo (linha 9); caso contrário, considera-se o envio do subvetor direito como parâmetro (linha 11). As chamadas recursivas à função *BuscaBinaria()* continuam até que todo o vetor seja considerado.

Veja que todas as instruções na função *BuscaBinaria()* são executadas em tempo constante uma única vez. Isso significa que o tamanho da instância de entrada não impacta na execução da função individualmente. O impacto do tamanho da instância se dará no número de vezes que a função chama a si mesma recursivamente. Nesse sentido, a cada chamada recursiva, o tamanho da instância será dividido por dois. Isso nos leva a um contexto similar ao apresentado na Figura 2, na Seção 1, onde a árvore de recursão descreve o tempo de execução em cada um de seus níveis.

A altura da árvore de recursão para a Busca Binária será igual a $\lg n$; com a adição de uma unidade referente ao último nível na árvore, temos que o número de execuções da função é igual a $\lg n + 1$, o que nos leva a concluir que o algoritmo Busca Binária é da ordem de $O(\lg n)$, visto que o valor constante 1 é desconsiderado nessa representação.

A comparação entre a Busca Sequencial e a Busca Binária revela a diferença de desempenho entre ambas as estratégias. A Busca Sequencial compara o elemento objetivo com todos os elementos no vetor. Assim, no pior caso, o número de comparações será igual ao número de elementos no vetor; portanto, o algoritmo Busca Sequencial é da ordem de $O(n)$.

Algoritmos Sobre Grafos

Um grafo G é uma estrutura que contempla duas partições $G(V, E)$, onde $G(V)$ é o conjunto de elementos denominados vértices e $G(E)$ é o conjunto das relações, denominadas arestas, entre os elementos em $G(V)$. Por exemplo, sejam $a = \text{Maria}$ e $b = \text{Pedro}$, usuários de uma rede social; assim, $G(V)$ é o conjunto de usuários da rede social e $a, b \in G(V)$. Se a e b são amigos na rede social, então $(a, b) \in G(E)$; veja que (a, b) é uma aresta no grafo.

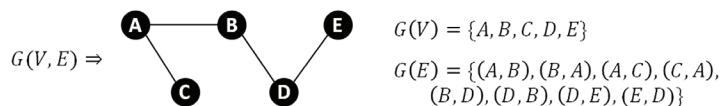


Figura 4 – Grafo genérico ilustrativo, as partições $G(V)$ e $G(E)$ representam os vértices e as arestas do grafo, respectivamente

A Figura 4 apresenta um grafo genérico como exemplo. Comumente, utiliza-se uma representação na qual os vértices são representados por circunferências e as arestas por segmentos de reta que conectam as circunferências. Na linguagem dos conjuntos, as partições $G(V)$ e $G(E)$ representam os vértices e as arestas, respectivamente.

Existem diversas aplicações no mundo real para a modelagem por meio de grafos. Os grafos são utilizados para a representação de circuitos elétricos, estrutura de moléculas, conectividade na *internet*, mapas geográficos, derivação sintática de linguagens, enfim, são muitas as aplicações possíveis para esse tipo de representação.

No contexto dos algoritmos, vamos estudar o algoritmo de busca em largura ou *BFS* (*Breadth First Search*). Esse algoritmo é muito útil para explorar as arestas em um grafo, de modo que se possa descobrir todos os vértices que podem ser alcançados a partir de um vértice origem. Além disso, o algoritmo armazena as distâncias entre o vértice origem e todos os outros vértices alcançáveis.

O nome busca em largura se deve ao fato de que o algoritmo descobre primeiro os vértices com uma aresta de distância, depois com duas arestas e assim sucessivamente. Considerando-se o grafo na Figura 4 como exemplo, se o vértice origem (inicial) é o B , a ordem de descobrimento dos outros vértices seria A, D, C e E .

O Algoritmo 4 descreve as instruções que compõem o algoritmo de busca em largura (BFS). Consideraremos como exemplo o grafo da Figura 4, no qual o vértice B será a origem da busca. Cada passo do algoritmo está ilustrado na Figura 5.

Algoritmo 4 – Algoritmo de busca em largura (BFS)

$BFS(G, s)$

1. **para** cada vértice $u \in G(V) \setminus \{s\}$
2. $u.cor \leftarrow BRANCO$
3. $u.distância \leftarrow \infty$
4. $u.pai \leftarrow NULL$
5. $s.cor \leftarrow CINZA$
6. $s.distância \leftarrow 0$
7. $s.pai \leftarrow NULL$
8. $Q \leftarrow \emptyset$
9. **Enqueue**(Q, s)
10. **enquanto** $Q \neq \emptyset$
11. $u \leftarrow Dequeue(Q)$
12. **para** cada $v \in \{w | (u, w) \text{ ou } (w, u) \in G(E)\}$
13. **se** $v.cor = BRANCO$
14. $v.cor \leftarrow CINZA$
15. $v.distância \leftarrow u.distância + 1$
16. $v.pai \leftarrow u$
17. **Enqueue**(Q, v)
18. $u.cor \leftarrow PRETO$

O algoritmo *BFS()* recebe, como parâmetros de entrada, o grafo G e um vértice origem s que, para a nossa simulação, é $s = B$. Há duas etapas principais na execução do algoritmo, uma etapa de inicialização (linhas 1-9) e o percurso no grafo (linhas 10-18). Na etapa de inicialização, um laço atribui valores para os atributos de cada vértice, exceto o vértice origem. Os atributos considerados, para cada vértice, são a cor, que indica se o vértice não foi visitado (branco), foi visitado (cinza) ou foi encerrado (preto), a distância da origem e o predecessor ou pai.

Nesse primeira etapa, os vértices são inicializados com a cor branca, indicando que ainda não foram visitados, a distância é igual a infinito, por conta de ainda não ter sido calculada, e o pai recebe o valor *NULL*. Exceção feita ao vértice origem B , que recebe a cor cinza, indicando que ele foi visitado, a distância igual a 0, visto que esse valor é a distância referente a ele mesmo, e o pai é *NULL*; como ele é o primeiro vértice considerado, não há um predecessor a ele.

Antes do início do percurso, o vértice origem B é inserido em uma fila. O procedimento *Enqueue()*, na linha 9, é responsável pela inserção. A definição da estrutura de dados denominada fila é feita na disciplina de Estrutura de Dados Lineares e há uma vasta descrição a respeito, também, na *internet*.



Veja uma videoaula a respeito do tema fila estática. Disponível em: <https://youtu.be/rIzGHX6ai70>

Na Figura 5, o passo (a) ilustra o nosso grafo com seus vértices inicializados de acordo com as instruções do algoritmo *BFS()*. Veja que o vértice B está inserido na fila. A etapa que realiza o percurso no grafo será realizada enquanto houver elementos na fila. Inicialmente, é retirado um vértice da fila Q (linha 11) e as instruções posteriores serão aplicadas a todos os vértices adjacentes (vizinhos) ao vértice retirado da fila (linha 12).

Para cada vértice v , adjacente ao vértice $u = B$ retirado da fila, verifica-se se a sua cor é branca e, caso seja, altera-se para a cor cinza, sua distância será igual à distância de $u + 1$ e o pai de v será u . Ao final das atualizações dos atributos do vértice, ele é inserido no final da fila. Veja que o passo (b) ilustra a atualização dos vértices A e D , os quais são adjacentes ao vértice B . Após todos os vértices adjacentes serem atualizados, muda-se a cor do vértice B para preto.

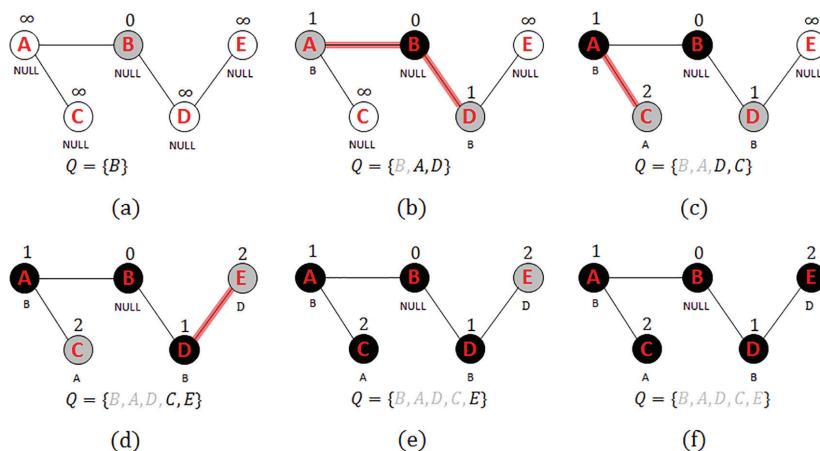


Figura 5 – Passo a passo ilustrativo da execução do algoritmo de busca em largura (BFS)

Fonte: Adaptado de CORMEN, 2009

O próximo vértice a ser retirado da fila é o A , todo o processo de atualização dos atributos será repetido com os vértices adjacentes ao A . Note que o vértice B é adjacente ao A , porém B não é branco, o que indica que ele já foi visitado; no caso, B é preto, o que indica que ele foi finalizado. Assim, somente o vértice C será atualizado, conforme ilustra o passo (c) da Figura 5.

O processo prossegue com a atualização dos vértices adjacentes aos vértices D , C e E . Nesse ponto, a fila estará vazia e o laço é encerrado. Veja os passos (d), (e) e (f), que ilustram a atualização dos vértices.

O resultado da execução do algoritmo $BFS()$ é a identificação de todos os vértices alcançados a partir do vértice origem B ; além disso, veja que todos os vértices possuem um valor que define a distância em relação ao vértice origem. A distância entre dois vértices é o número de arestas existentes entre eles. Por exemplo, a distância entre o vértice E e o vértice origem B é igual a dois, ou seja, existem duas arestas conectando esses vértices.

Quando todos os vértices são rotulados com uma distância em relação ao vértice origem, isso significa que o grafo é uma única componente conexa, ou seja, todos seus vértices estão conectados. Se um vértice não é alcançado a partir do vértice origem, ele manterá o atributo distância com o valor infinito (∞), indicando que o acesso a esse vértice não é possível a partir da origem.

O algoritmo $BFS()$ é um dos mais simples dos algoritmos de busca em grafos e serve como base para diversos outros algoritmos importantes que executam sobre grafos.



Veja no artigo “[Algoritmos para Grafos via Sedgewick](https://bit.ly/370fQEg)” uma descrição detalhada sobre o tema, preparado pelo professor Paulo Feofiloff (IME – USP). Disponível em: <https://bit.ly/370fQEg>

A análise do tempo de execução do algoritmo busca em largura é feita considerando-se um grafo $G(V, E)$ como parâmetro de entrada. Veja que a etapa de inicialização do grafo, na qual os valores dos atributos são determinados, é feita para cada vértice $v \in G(V)$. Assim, considerando $n = |G(V)|$ (número de vértice no grafo), temos que, nessa etapa, a inicialização demora $O(n)$.

As inserções e remoções na fila são feitas uma única vez para cada vértice; assim, as operações de enfileirar e desenfileirar também demoram $O(n)$. De forma similar, os vértices adjacentes a cada vértice são varridos quando o vértice é desenfileirado, ou seja, cada aresta é considerada uma única vez por vértice. Sendo $m = |G(E)|$ (número de arestas no grafo), temos que o tempo gasto para a atualização dos adjacentes é $O(m)$.

Nesse contexto, o tempo de execução total do procedimento $BFS()$ é $O(n+m)$. Em outras palavras, o algoritmo gasta tempo linear ao tamanho da entrada para executar a busca de todos os vértices possíveis de serem alcançados a partir de um vértice origem.

Neste ponto já é possível perceber que a análise de tempo de execução de um algoritmo é uma tarefa mais interpretativa do que sobre contagem de instruções. A contagem de instruções, conforme foi vista para o algoritmo *Bubble Sort*, é, antes de tudo, uma

forma de se demonstrar, detalhadamente, como identificar o número de instruções que serão realizadas. Nesse sentido, é importante que tenhamos em mente alguns pontos de atenção que são importantes e muito úteis para analisar o tempo de execução dos algoritmos, em função do tamanho de suas respectivas entradas.

Instruções que são realizadas uma única vez, ou seja, aquelas que não estão inseridas em laços, não impactam o tempo de execução do algoritmo. Veja que quando identificamos a função que representa esse tempo de execução, consideramos somente a variável de maior grau, desconsiderando, assim, as constantes. Desse modo, uma instrução fora de um laço contabilizará um valor constante que, na classificação em termos de notação assintótica, será desconsiderado por não contribuir efetivamente na velocidade de crescimento da função.

Por outro lado, a existência de laços nos algoritmos é uma observação que deve chamar nossa atenção. A primeira característica importante a ser observada em um laço é a quantidade de vezes que ele irá executar as instruções que o compõem. Comumente, a execução dos laços está ligada ao tamanho da instância de entrada. Como vimos nos algoritmos de ordenação, a quantidade de elementos que serão ordenados é determinante para a execução dos laços. Quando consideramos grafos, por exemplo, a quantidade de vértices e arestas determinará, também, o número de vezes que os laços serão executados.

Na existência de dois laços que não estejam aninhados – por exemplo, dois laços que executam n vezes, teremos $2 \times n$ instruções executadas, ou $2n$. Veja que, novamente considerando a análise assintótica e tudo o que está implícito nessa consideração, o tempo de execução do algoritmo ainda será da ordem de n . Veja o exemplo do algoritmo de busca em largura; a análise nos mostrou que o tempo de execução é da ordem de $O(n + m)$, sendo n o número de vértices e m o número de arestas. Essa classificação indica um algoritmo cujo tempo de execução é linear no tamanho da entrada.

Se os laços são determinantes do tempo de execução dos algoritmos, o aninhamento de laços pode impactar, de forma importante, o número de execuções dos laços. Quando temos dois laços aninhados, supondo os laços interno e externo governados pela variável n , veja que o laço interno será executado n vezes. Ou seja, teremos $n \times n$ instruções executadas no corpo do laço interno, ou n^2 instruções. Um algoritmo com essas características executa suas instruções em tempo proporcional a um polinômio de grau 2, o que é muito mais ineficiente que um algoritmo que executa em tempo linear.

Existem outras características que devem ser observadas na análise do tempo de execução dos algoritmos. Ao longo do curso, nas próximas unidades, teremos a oportunidade de verificar essas características e aprimorar nossa visão sobre a eficiência dos algoritmos. Mais importante que saber analisar a complexidade de um algoritmo é, a partir do conhecimento das características que impactam no tempo de execução, poder projetar algoritmos que sejam eficientes. Essa capacidade, certamente, é um diferencial importante para aqueles que, de alguma forma, atuam na área.

Nas próximas unidades nós vamos considerar, principalmente, a análise do pior caso (O), porém é importante nos lembarmos de que existem outras classes para a análise de complexidade. A justificativa para priorizar a análise do pior caso é que ele nos fornece a informação sobre o limite máximo de tempo que será necessário para a finalização dos algoritmos.

Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

▶ Vídeos

Bubble Sort – Funcionamento e Cálculo do Custo

<https://youtu.be/E8yj1nRd32s>

Merge Sort, demonstração do algoritmo de ordenação de dados em JavaScript | Ordenação de Dados

<https://youtu.be/f7tlfh5tSwI>

Busca Binária

<https://youtu.be/l6pxuyV3mKQ>

Projeto e Análise de Algoritmos – Aula 12 – Algoritmos de busca em largura e profundidade em grafos

<https://youtu.be/dQ42rha2qFA>

Referências

CORMEN, T. H. *et al.* **Introduction to algorithms**. MIT press, 2009.

TOSCANI, L. V. **Complexidade de algoritmos**, v. 13: UFRGS. 3^a edição. Porto Alegre: Bookman 2012. (*e-book*)

ZIVIANI, N. **Projeto de algoritmos**: com implementações em JAVA e C. São Paulo: Cengage Learning, 2012. (*e-book*)



Cruzeiro do Sul
Educacional