

# Complexidade de Algoritmos

Prof. Guilherme N. Ramos

Os objetivos de um algoritmo computacional são resolver corretamente a tarefa e utilizar os recursos computacionais de forma eficiente [1]. Embora a corretude seja prioritária (de que adianta um algoritmo eficiente que não resolve o problema?), a eficiência pode ser extremamente relevante (de que adianta detectar uma colisão após o acidente?), então é importante entender e otimizar algoritmos.

O custo de execução de um algoritmo depende, em primeiro lugar, da *métrica* definida. As medidas mais comuns são *tempo* (a duração da execução), e *espaço* (a quantidade de memória necessária para execução). Considerar o tempo necessário para terminar a execução de um algoritmo permite planejar se é viável sua execução (considerando as restrições existentes), como organizar sequências de algoritmos, entre outros. Considerar o espaço de memória, permite verificar a viabilidade de uma solução [correta e que leve tempo hábil], pois pode não haver recursos suficientes para executá-la. Neste contexto introdutório, supondo que as atividades não lidam com um volume excessivo de dados, considera-se apenas o *tempo* como métrica.

## 1 Complexidade de Algoritmos

O custo envolvido na execução de instruções sequenciais e bifurcações é proporcional a quantidade de instruções, já o custo de instruções com repetições é proporcional a quantidade de instruções e a quantidade de repetições destas. Mais especificamente, o custo também depende de *quais* instruções são consideradas. Por exemplo, poderia estar interessado apenas na quantidade de *atribuições* realizadas (ou *comparações*, ou *operações matemáticas*, ou todas juntas).

Considerando que cada instrução de atribuição (*atr*), comparação (*cmp*), adição (*ad*), e entrada/saída de dados (*es*) tenha seu custo, qual seria o custo total de execução deste algoritmo?

```
1 int i, n;
2
3 scanf("%d", &n);           /* 1 es */
4
5 if(n < 0)                   /* 1 cmp */
6     printf("Valor inválido!"); /* 1 es */
7 else {
8     printf("Valor válido!");   /* 1 es */
9     printf("Mas vou mudá-lo..."); /* 1 es */
10    printf("... para facilitar a demonstração."); /* 1 es */
11 }
12
13 n = 4;                      /* 1 atr */
14 for(i = 0; i < n; ++i)      /* n+1 atr, n+1 cmp, n ad */
15     printf("i = %d\n", i); /* n es */
```

Os comentários indicam as operações, todas devem ser bem evidentes. Nas linhas 5 a 11 há uma comparação e duas possibilidades: ou executa-se a linha 6 ou as linhas 8-10 (mas nunca todas as quatro). Portanto, dependendo do valor de  $n$ , o custo será de 1 ou 3, mas proporcional a quantidade de instruções sequenciais. Na linha 15, há uma operação de E/S, que por estar em um laço de repetição, será executada  $n$  vezes. Na linha 14 há 1 atribuição  $i = 0$ , a comparação  $i < n$  será realizada  $n + 1$  vezes, e a cada uma das  $n$  iterações, o contador  $i$  será atribuído e incrementado ( $++i$ ).

Como  $n$  é definido com o valor 4 (linha 13), pode se dizer que o custo deste algoritmo é, dependendo do caminho tomado na bifurcação:

$$(n)ad + (n + 1)atr + (n + 2)cmp + (n + 2)es \rightarrow 4ad + 5atr + 6cmp + 6es$$

ou

$$(n)ad + (n + 1)atr + (n + 2)cmp + (n + 4)es \rightarrow 4ad + 5atr + 6cmp + 8es$$

Esta análise é válida para qualquer situação, mas o custo específico depende, evidentemente, da velocidade do computador e das especificidades da linguagem em que o programa será executado. Supondo que, neste exemplo, a adição custa 2ms, a atribuição 1ms, a comparação 2 ms, e E/S 10ms; pode-se dizer que sua execução demora entre 85 e 105ms.

Considere que se deseja buscar um elemento em um vetor, uma tarefa trivialmente implementada da seguinte forma:

```
apc_busca.h
1 /* Retorna o índice do elemento cujo conteúdo é "valor", se
2 existir no vetor de n elementos, -1 caso contrário. */
3 int busca_sequencial(int valor, int* vetor, int n) {
4     int i;
5
6     for(i = 0; i < n; ++i) {
7         if(vetor[i] == valor)
8             return i;
9     }
10
11     return -1;
12 }
```

Quantas comparações seriam necessárias para executar este algoritmo? No melhor caso, o elemento desejado é o primeiro e basta 1 comparação. No pior caso, o elemento não existe no vetor, e são necessárias  $n$  comparações para terminar a busca. Desta forma, sabe-se que o custo será de pelo menos 1 e nunca mais que  $n$ .

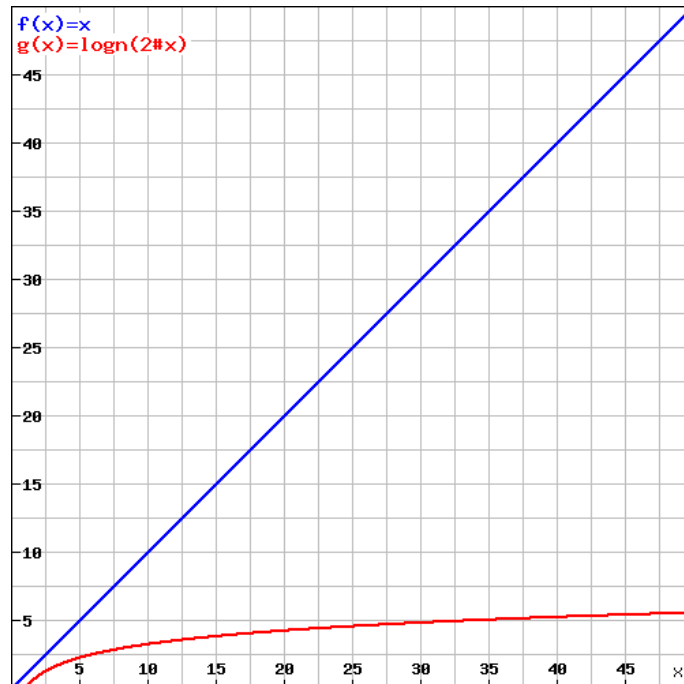
Considere que se deseja buscar uma página em um livro, o algoritmo acima certamente está correto para esta tarefa. Mas ninguém procura uma página desta forma, a forma “tradicional” poderia ser descrita como:

1. Abra o livro no meio.
2. Se está aberto na página desejada, termine.
3. Se a página desejada é menor que a página atual, desconsidere toda a metade posterior do livro, e repita o processo na metade anterior (imagine rasgar o livro ao meio e jogar a segunda metade fora).
4. Senão, desconsidere toda a metade anterior do livro, e repita o processo na metade posterior (imagine rasgar o livro ao meio e jogar a primeira metade fora).

Este processo é muito mais eficiente que o anterior, porque demanda menos passos (tente implementá-lo). A cada tentativa, descarta-se metade dos elementos, de modo que a quantidade decresce em progressão geométrica  $n$  elementos até que haja apenas um, possibilitando o término do processo, em  $T$  tentativas. Portanto, a quantidade de tentativas é:

$$2^T = n \Rightarrow T = \log_2(n)$$

Comparando esta abordagem (conhecida como busca binária) a busca sequencial vista, fica claro o quão mais eficiente ela é.



Considere o código abaixo, utilizado para ordenar os elementos de um vetor:

apc\_ordenacao.h

```
1 /* Ordena os elementos do vetor em ordem crescente. */
2 void bubble_sort1(int* vetor, int n) {
3     int i, j;
4
5     for(i = 0; i < n; ++i)
6         for(j = i + 1; j < n; ++j)
7             if(!crescente(vetor[i], vetor[j]))
8                 troca(vetor + i, vetor + j);
9 }
```

Supondo que se esteja interessado apenas na quantidade de *trocas* realizadas, pode-se estimar o custo de execução deste algoritmo simplesmente contando quantas vezes a função `troca` é chamada. Isto ocorre sempre que a linha 8 for executada, mas a linha 7, que é executada antes, força um teste de modo que a função só é executada se os elementos comparados não estiverem em ordem. Se isto nunca acontecer, então nunca há uma troca e, portanto, o custo de execução do algoritmo é zero (o melhor caso possível).

Considere o caso oposto, os elementos estão sempre fora da ordem e, portanto, todo teste resulta na execução de `troca` (e pode-se, então, ignorar o teste da linha 7). Nesta situação, para saber o custo basta contar quantas vezes a função será chamada dentro do laço de repetição, ou seja, quantas são as iterações. No caso,  $j$  varia de  $i+1$  a  $n$ , então são  $n - i - 1$  iterações. Mas o laço da linha 7 também está dentro de um outro laço em que  $i$  varia de 0 a  $n - 1$  ( $n$  iterações).

Então, quando  $i = 0$ , serão  $n - 1$  chamadas de `troca`, quando  $i = 1$ , serão  $n - 2$ , e assim sucessivamente até  $i = n - 1$  e 0 chamadas:

$$\underbrace{(n-1)}_{a_1} + \underbrace{(n-2)}_{a_2} + \cdots + \underbrace{(2)}_{a_{n-2}} + \underbrace{(1)}_{a_{n-1}} + \underbrace{(0)}_{a_n}$$

Claramente uma progressão aritmética, cujo total pode ser calculado como:

$$\sum_{i=0}^n a_i = \frac{n}{2}(a_1 + a_n) = \frac{n(n-1)}{2}$$

Este algoritmo, considerando apenas `troca_i` como medida, pode ter um custo de execução que varia entre 0 (limite inferior/melhor caso) e  $\frac{n(n-1)}{2}$  (limite superior/pior caso). O custo exato depende dos valores dos  $n$  elementos do vetor, geralmente uma informação desconhecida, mas pode-se garantir que este custo nunca será maior que o limite superior nem menor que o inferior.

Se considerar apenas a execução da função `crescente` como medida, o mesmo raciocínio se aplica, mas como haverá uma comparação a cada iteração do laço, independentemente dos elementos no vetor, esta função será chamada  $\frac{n(n-1)}{2}$ .

*O custo de execução de um algoritmo está associado ao tamanho entrada.*

Tente analisar os algoritmos: *Insertion Sort* e *Selection Sort* em `apc_ordenacao.h`.

*“O sistema de análise matemática [...] constitui o maior avanço técnico do pensamento exato.”*

**John von Neumann**

Diversos fatores influenciam o tempo de execução de um algoritmo: a velocidade do processador, outros processos sendo executados, etc. Portanto, obter um custo exato, ou mesmo fazer estimativas mais precisas quanto a isso é muito complicada. A complexidade computacional tenta avaliar o quão difícil é a execução do algoritmo, possibilitando uma estimativa do custo (que pode levar a decisão de utilizar ou não a solução) e uma comparação entre [classes de] algoritmos de forma independente do hardware. Para facilitar esta análise, utiliza-se a notação assintótica, uma forma mais simples de avaliar o algoritmo. Ela indica *o quão rapidamente cresce o custo de execução em relação ao tamanho da entrada do algoritmo, supondo que este tamanho chegue a valores arbitrariamente [muito] grandes*.

Por exemplo, veja os seguintes algoritmos:

```
1 int f(int x) {
2     int i, total = 0;
3
4     for(i=0; i<50; ++i)
5         total += i;
6
7     return total;
8 }
```

```
1 int g(int x) {
2     int i, total = 0;
3
4     for(i=0; i<x; ++i)
5         total += i;
6
7     return total;
8 }
```

```
1 int h(int x) {
2     int i, j, total = 0;
3
4     for(i=0; i<x; ++i)
5         for(j=0; j<x; ++j)
6             total += i;
7
8     return total;
9 }
```

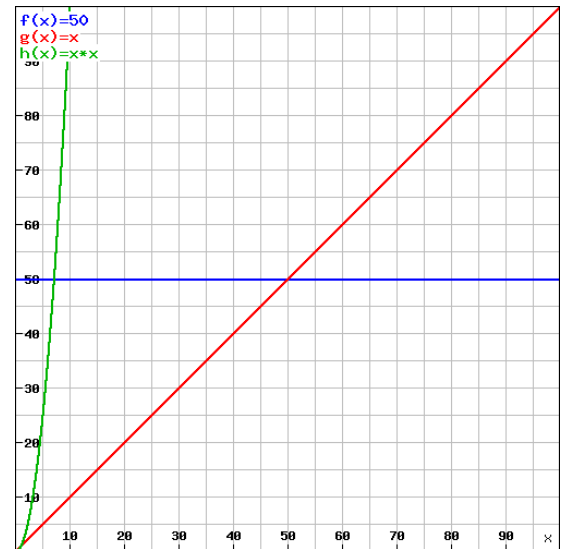
Para facilitar, considere que apenas as instruções envolvendo a atualização da variável `total`. No caso de  $f(x)$ , `total` é alterada 50 vezes no laço de repetição. Isso acontece para uma entrada de dados qualquer, ou seja, independentemente do valor de  $x$  (1, 100,  $10^9$ , ...), as mesmas 50 atualizações serão realizadas. No caso de  $g(x)$ , as alterações são realizadas em um laço de repetição cujo critério de parada depende de  $x$ , portanto a quantidade de instruções a serem executadas é diretamente proporcional a quantidade definida por  $x$ . Ou seja, se  $x = 1$ , será o custo de 1 instrução; se  $x = 100$ , será o custo de 100 instruções; se  $x = 10^9$ , será o custo de  $10^9$  instruções, e assim sucessivamente.

Por fim, no caso de  $h(x)$ , as alterações são realizadas em dois laços de repetição aninhados, cujos critérios de parada dependem de  $x$ . Assim, a quantidade de instruções a serem executadas é diretamente proporcional ao quadrado da quantidade definida por  $x$ . Ou seja, se  $x = 1$ , será o custo de 1 instrução; se  $x = 100$ , será o custo de 10000 instruções; se  $x = 10^9$ , será o custo de  $10^{18}$  instruções, e assim sucessivamente. Considere então a seguinte função, que chama  $f$ ,  $g$  e  $h$ :

```

1 int funcao(int n) {
2     int total;
3
4     total = f(n); /* 50 instruções */
5     total += g(n); /* n instruções */
6     total += h(n); /* n*n instruções */
7
8     return total;
9 }

```



Já foi visto que a influência de cada trecho do código no custo total da função muda conforme o tamanho da entrada. O gráfico ilustra claramente que para valores arbitrariamente maiores de entrada, o custo associado ao termo  $n^2$  cresce muito mais que os demais, efetivamente *dominando* os outros. Para um valor suficientemente grande de  $n$ , este crescimento é tão maior que os coeficientes menores podem ser ignorados, ou seja, o custo pode ser aproximado por uma função mais simples:  $n^2 + n + 50 \approx n^2$ .

**A notação assintótica** (*Grande-O*) é uma notação matemática usada para analisar o comportamento de funções e utilizada para descrever o uso de recursos computacionais, permitindo prever o comportamento do algoritmo e determinar qual algoritmo utilizar. De forma simplificada, para valores grandes o suficiente, o termo de maior coeficiente da função de custo domina os demais, tornando-se o único a ser considerado.

Por exemplo, considere que a execução de cada linha de código abaixo tenha custo igual, independentemente de qual instrução. O custo total de cada função pode ser aproximado como o custo da função que cresce mais rapidamente.

```

1 void O_1(int n) {
2     printf("Gol da Alemanha!"); /* 1 */
3     printf("Gol da Alemanha!"); /* 1 */
4     printf("Gol da Alemanha!"); /* 1 */
5     printf("Gol da Alemanha!"); /* 1 */
6     printf("Gol da Alemanha!"); /* 1 */
7     printf("Gol da Alemanha!"); /* 1 */
8     printf("Gol da Alemanha!"); /* 1 */
9     printf("Gol do Brasil!"); /* 1 */
10 } /* custo: 8 -> O(1) */
11
12 void O_n(int n) {
13     O_1(n); /* 8 */
14     while(n--) /* n */
15         printf("Gol da Alemanha?"); /* n */
16 } /* custo: 2n + 8 -> O(n) */
17
18 void O_n2(int n) {
19     int i, soma = 0; /* 1 */
20
21     O_1(n); /* 8 */
22     O_n(n); /* 2n + 8 */
23     for(i = 0; i < n; ++i) /* 3n */
24         for(j = n; j > 0; --j) { /* 3n^2 */
25             printf("(%d, %d)", i, j); /* n^2 */
26             soma += i + j; /* 3n^2 */
27         }

```

Os algoritmos podem, então, ser descritos em função de suas classes:

$O(1)$  custo constante (independente de  $n$ ).

$O(\log(n))$  custo logarítmico.

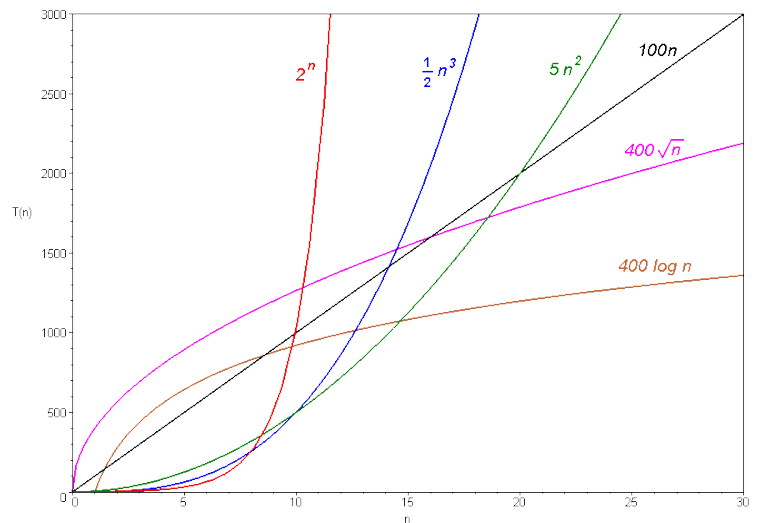
$O(n)$  custo linear.

$O(n \log(n))$  custo log-linear.

$O(n^c)$  custo polinomial.

$O(c^n)$  custo exponencial.

$O(n!)$  custo fatorial.



A notação assintótica serve para comparar algoritmos de classes diferentes, no caso de algoritmos de mesma classe, uma análise mais detalhada da função de custo é necessária. Por exemplo, qual o custo de comparar cada elemento de um vetor aos outros elementos?

#### Algoritmo

```
1 for (i = 0; i < n; ++i)
2   for (j = 0; j < n; ++j)
3     if (i != j)
4       compare(i, j)
```

#### Algoritmo Otimizado

```
1 for (i = 0; i < n; ++i)
2   for (j = i+1; j < n; ++j)
3     compare(i, j)
```

Em ambos os casos, a complexidade é  $O(n^2)$ , mas o algoritmo otimizado tem custo menor.

Uma implementação correta que seja eficiente, em termos de custo computacional, é o objetivo final do desenvolvimento de software. Entretanto, na maioria dos casos é melhor ter código claro (e simples) que instruções obscuras e otimizadas.

*“Otimização prematura é a raiz de todos os males.”*

Donald Knuth

## 2 Ordenação

O ordenação é o processo de organizar os elementos de um conjunto, e pode ser extremamente útil (por exemplo, na busca binária). A análise de algoritmos de ordenação possibilita não só lidar com formas diferentes e interessantes de resolver uma mesma tarefa, como uma boa oportunidade para analisar a complexidade de algoritmos.

Por exemplo, supondo a quantidade de operações como medida de custo, quais as complexidades dos seguintes algoritmos de ordenação?

apc\_ordenacao.h

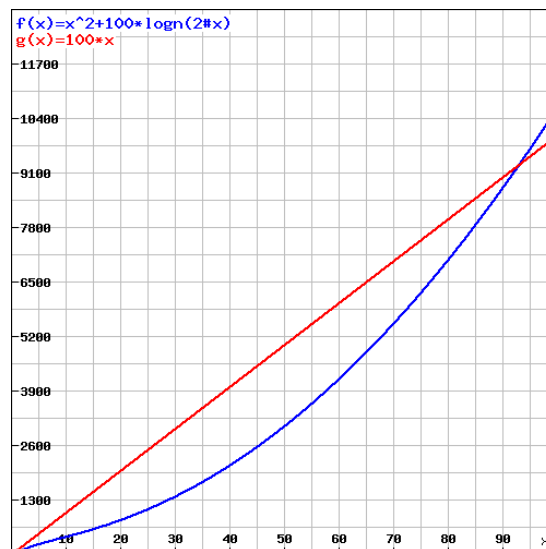
```
1 void bogosort(int* vetor, int n) {
2     while(!ordenado(vetor, n))
3         embaralha(vetor, n);
4 }
```

apc\_ordenacao.h

```
1 void bubble_sort2(int* vetor, int n) {
2     int i, houve_troca;
3
4     do {
5         houve_troca = 0;
6         for(i = 0; i < n-1; ++i)
7             if(!crescente(vetor[i], vetor[i+1])) {
8                 troca(vetor + i, vetor + i + 1);
9                 houve_troca = 1;
10            }
11    } while(houve_troca);
12 }
```

Suponha que o problema seja encontrar um elemento no vetor, e tem-se dois algoritmos de classes diferente para fazer isso: busca sequencial ( $O(n)$ ) ou binária ( $O(\log_2(n))$ ). Entretanto, a busca binária têm como requisito a ordenação do vetor, portanto o custo de ordenar o vetor e então realizar a busca é  $O(n^2) + O(\log_2(n))$ .

Claramente, o custo de ordenação e busca é superior ao de uma simples busca sequencial. Mas e se forem  $x$  buscas? Para determinados valores de  $x$ , é menos custoso ordenar uma vez e realizar  $x$  buscas (a custo logarítmico) que realizar  $x$  buscas sequenciais.



Esta amortização do custo computacional deve ser considerada na escolha o algoritmo a ser utilizado.

Certos problemas têm características que podem ser exploradas para obter uma solução mais eficiente. Por exemplo, a busca binária aproveita o fato do vetor estar ordenado para dividi-lo em duas partes, uma das quais é desconsiderada, efetivamente diminuindo o esforço necessário para encontrar o elemento.

Esta estratégia de dividir para conquistar também pode ser utilizada para para ordenação. A ideia é simples, dividir o problema em versões menores, resolver estas versões recursivamente, e combinar os resultados de forma a obter a solução completa [1]. É evidente que ordenar  $n/2$  elementos exige menos esforço que ordenar  $n$ , assim como combinar dois conjuntos já ordenados em um é mais fácil que ordenar um conjunto completo.

apc\_ordenacao.h

```
1 void merge_sort(int* vetor, int n) {
2     if(n < 2) return;
3
4     int meio = n/2;
5     merge_sort(vetor, meio);
```

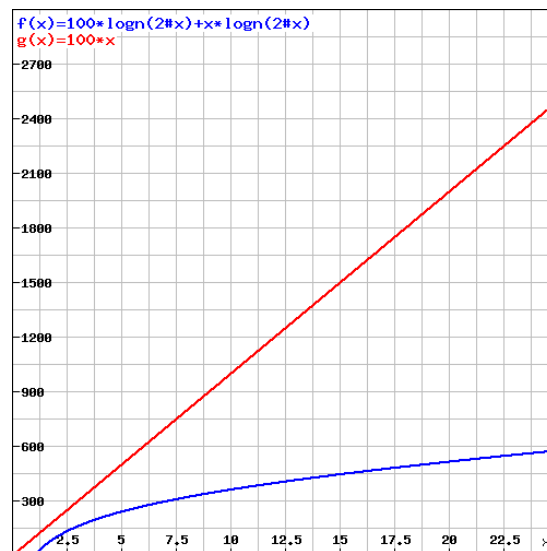
```

6  merge_sort(vetor + meio, n - meio);
7  merge(vetor, n);
8 }

```

Uma análise pouco detalhada do código mostra que a cada passo há uma divisão do conjunto e duas partes (linhas 5 e 6), a um custo logarítmico, e a função merge junta dois vetores de forma linear, portanto a complexidade do algoritmo é da ordem  $O(n \log_2(n))$ .

O merge sort é mais eficiente que o bubble sort [2], e poderia ser considerado na tarefa de realizar  $x$  buscas (com o algoritmo de busca binária).



Outros algoritmos de ordenação interessantes são o Heapsort e o quick sort. O funcionamento deles pode ser visto em:

- [www.sorting-algorithms.com](http://www.sorting-algorithms.com)
- [sorting.at](http://sorting.at)
- <https://www.youtube.com/user/AlgoRythmics/videos>
- <https://www.youtube.com/watch?v=kPRA0W1kECg>
- <https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

O quick sort [3] merece destaque pois pode ser um algoritmo muito eficiente se bem implementado [4] (em termos de custo real), mas sua complexidade é da ordem  $O(n^2)$ . Você consegue descobrir o motivo disso<sup>1</sup>? A ideia também é dividir para conquistar, escolhe-se um elemento (pivô), e separa-se os elementos restantes em um grupo com os elementos menores que o pivô, outro com os elementos maiores. Desta forma, o pivô está devidamente posicionado, e os problemas restantes são mais fáceis de resolver. O processo se repete (recursivamente) para cada grupo.

```

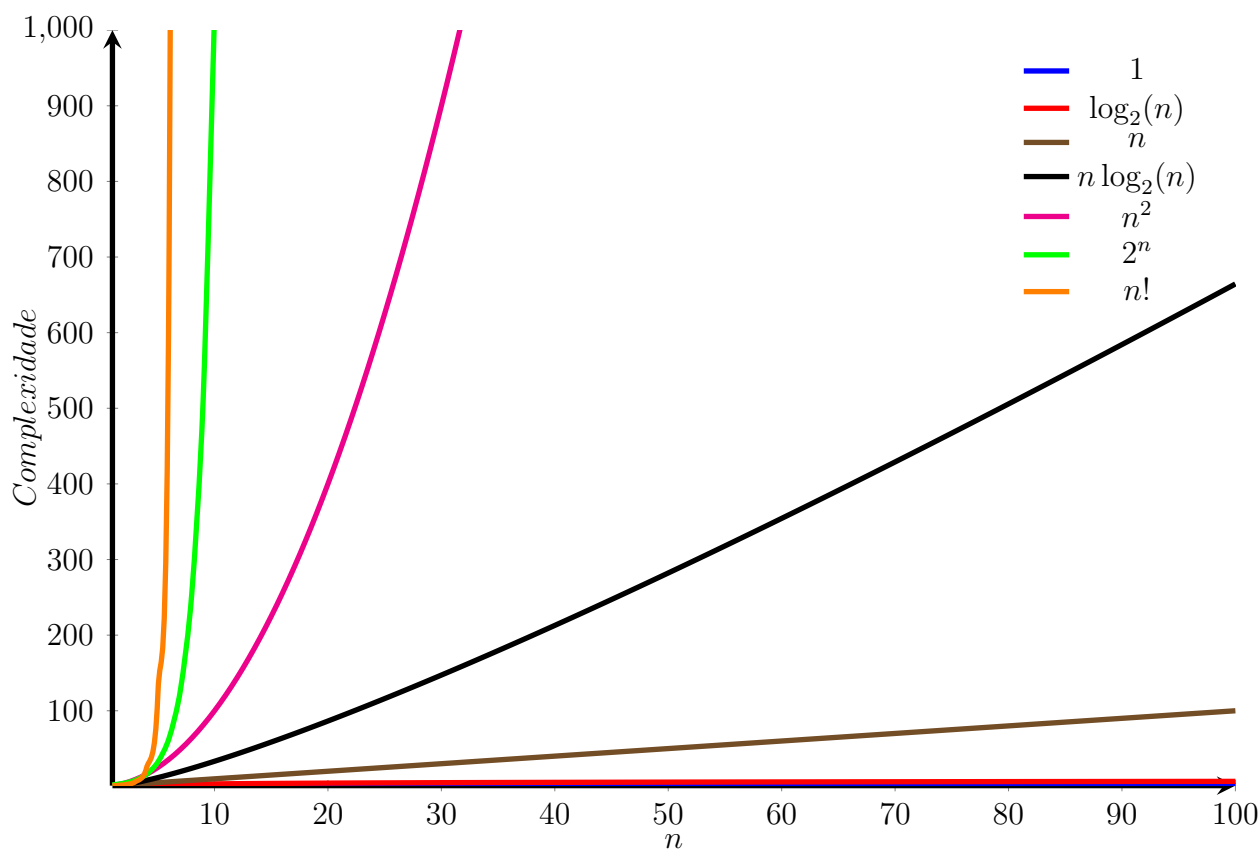
1 void quick_sort(int *vetor, int n) {
2     if (n < 2) return;
3
4     int i, pivo;
5
6     pivo = escolhe_pivo(vetor, n);
7     for (i = 1; i < n; ++i)
8         /* organiza elementos em função do pivô */
9
10    quick_sort(/* a esquerda do pivô */);
11    quick_sort(/* a direita do pivô */);
12 }

```

<sup>1</sup>Por exemplo, analisando a linha 6.



Algoritmos de complexidade logarítmica são melhores que polinomiais, mas a escolha do algoritmo correto depende do valor de  $n$ . Para pequenos valores, a maioria dos algoritmos não representa variação significativa de desempenho, e estuda-se a complexidade somente para grandes valores de  $n$ , pois o comportamento assintótico representa o limite do comportamento do custo quando  $n$  cresce.



A análise pode ser considerada por diferentes perspectivas:

$\Omega(n)$  **Melhor:** definida pelo menor número de passos executados para qualquer instância de tamanho  $n$ .

**Médio:** definida pela média do número de passos executados para qualquer instância de tamanho  $n$ .

$O(n)$  **Pior:** definida pelo maior número de passos executados para qualquer instância de tamanho  $n$ .

Existem algumas formas rápidas de estimar a complexidade de algoritmos:

- considerar memória infinita
- não considerar o sistema operacional nem o compilador
- analisar o algoritmo e não o programa
- levar em conta o tamanho das entradas
- ter cuidado ao escolher a função de custo (atribuição, adição, multiplicação, comparação, etc.)

Para dois algoritmos que executem a mesma tarefa, a diferença entre complexidades pode ser muito mais significativa que a entre hardware/software. Por exemplo, considere:

**Bubble Sort** com custo  $c_1 \cdot n^2$ , rodando no computador  $A$

**Merge Sort** com custo  $c_2 \cdot n \log_2(n)$ , rodando no computador  $B$

Supondo que  $A$  executa  $10^{10}$  instruções por segundo, sendo 1000 vezes mais rápido que  $B$ ; e as implementações dos algoritmos se caracterizam por  $c_1 = 0,5$  devido ao superprogramador e  $c_2 = 50$  devido a inexperiência do estagiário. Se uma entrada de tamanho  $n = 10^7$  é fornecida, qual é a melhor opção?

## Referências

- [1] Thomas H. Cormen. *Algorithms Unlocked*. The MIT Press, 2013.
- [2] Donald Ervin Knuth. *The art of computer programming*. Addison-Wesley, 3rd ed edition.
- [3] C. A. R. Hoare. Algorithm 64: Quicksort. 4(7):321, 07 1961.
- [4] Steven S. Skiena. *The algorithm design manual*. Springer, 2nd ed edition.