Física Além da Sala de Aula (FASA - 2025)

Brevíssima Introdução à Análise de Algoritmos

Versão: 13/06/2025.

Sumário

1 Introdução

As notações que usamos para descrever o tempo de execução assintótico de um algoritmo são definidas em termos de funções cujos domínios são o conjunto dos números naturais. Tais notações são convenientes para descrever a função $T(n)$ do tempo de execução do pior caso, que em geral é definida somente para tamanhos de entrada inteiros. Contudo, às vezes, consideramos que é conveniente abusar da notação assintótica de vários modos. Por exemplo, poderíamos estender a notação ao domínio dos números reais ou, como alternativa, restringi-la a um subconjunto dos números naturais. Porém, é importante entender o significado preciso da notação para que, quando abusarmos, ela não seja mal utilizada. Estas notas (em grande medida extraídas do famoso livro *Algoritmos Teoria e Prática*) define as notações assintóticas básicas e também apresenta alguns abusos comuns.

1.1 Notação assintótica, funções e tempos de execução

Usaremos a notação assintótica primariamente para descrever o tempo de execução de algoritmos, como fizemos em aula quando escrevemos que o tempo de execução do pior caso para a ordenação por inserção é $\Theta(n^2)$. Lembre-se de que caracterizamos o tempo de execução do pior caso da ordenação por inserção como $T(n) = an^2 + bn + c$, para algumas constantes a, b e c. Quando afirmamos que o tempo de execução da ordenação por inserção é $\Theta(n^2)$, abstraímos alguns detalhes dessa função. Como a notação assintótica aplica-se a funções, o que quisemos dizer é que $\Theta(n^2)$ era a função $T(n)$ que, aqui, por acaso caracteriza o tempo de execução do pior caso da ordenação por inserção. Nestas notas, as funções às quais aplicamos a notação assintótica, normalmente caracterizarão os tempos de execução de algoritmos. Porém, a notação assintótica pode se aplicar a funções que caracterizam algum outro aspecto dos algoritmos (a quantidade de espaço que eles usam, por exemplo) ou até mesmo a funções que absolutamente nada têm a ver com algoritmos.

Mesmo quando utilizamos a notação assintótica para o tempo de execução de um algoritmo, precisamos entender a qual tempo de execução estamos nos referindo. Às vezes, estamos interessados no tempo de execução do pior caso. Porém, frequentemente queremos caracterizar o tempo de execução, seja qual for a entrada. Em outras palavras, muitas vezes desejamos propor um enunciado abrangente que se aplique a todas as entradas, e não apenas ao pior caso. Veremos que as notações assintóticas prestam-se bem à caracterização de tempos de execução, não importando qual seja a entrada.

1.1.1 Notação Θ

Nas aulas vimos que o tempo de execução do pior caso da ordenação por inserção é $T(n) = \Theta(n^2)$. Vamos definir o que significa essa notação.

Definição 1.1. Seja $g(n)$ uma função positiva. Dizemos que uma função $f(n)$ pertence a $\Theta\Big(g(n)\Big)$ se existem constantes positivas c_1 e c_2, e um natural n_0, tais que

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{para todo } n \geq n_0.$$

A definição indica que uma função $f(n)$ (note que $f : \mathbb{N} \mapsto \mathbb{R}_+$ se estamos considerando tempos de execução de algoritmos) pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que ela possa ser "encaixada" entre $c_1 g(n)$ e $c_2 g(n)$, para um valor de n suficientemente grande. Como $\Theta(g(n))$ é um conjunto, poderíamos escrever $f(n) \in \Theta(g(n))$ para indicar que $f(n)$ é um membro de (ou pertence a) $\Theta(g(n))$. Em vez disso, em geral escreveremos $f(n) = \Theta(g(n))$ para expressar a mesma noção. Esse abuso da igualdade para denotar

a condição de membro de um conjunto (pertinência) pode parecer confuso, mas veremos mais adiante que tem algumas ventagens.

A Fig. XX apresenta um quadro intuitivo de funções $f(n)$ e $g(n)$, onde $f(n) = \Theta(g(n))$. Para todos os valores de n em n_0 ou à direita de n_0, o valor de $f(n)$ encontra-se em $c_1 g(n)$ ou acima dele e em $c_2 g(n)$ ou embaixo desse valor. Neste caso dizemos que $g(n)$ é um *limite assintoticamente restrito* para $f(n)$.

A definição 1.1 exige que todo membro $f(n) \in \Theta(g(n))$ seja *assintoticamente não negativo*, isto é, que $f(n)$ seja não negativa sempre que n for suficientemente grande. Por consequência, a própria função $g(n)$ deve ser assintoticamente não negativa, senão o conjunto $\Theta(g(n))$ é vazio. Por isso, consideraremos que toda função usada dentro da notação Θ é assintoticamente não negativa.

Nas aulas introduzimos uma noção informal da notação Θ que consistia em descartar os termos de ordem mais baixa e ignorar o coeficiente inicial do termo de ordem mais alta. Vamos ver alguns exemplos que podem justificar esta intuição

Exemplo 1.1. Vamos mostrar que se $T(n) = \frac{1}{2}n^2 - 3n$, então $T(n) = \Theta(n^2)$. É claro que basta definir constantes positivas c_1, c_2 e n_0 tais que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq n_0$. Dividindo por n^2 temos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

A desigualdade do lado direito pode ser considerada válida para qualquer valor de $n \geq 1$, se escolhermos qualquer constante $c_2 \geq 1/2$. Do mesmo modo, a desigualdade da esquerda pode ser considerada válida para qualquer valor de $n \geq 7$, se escolhemos qualquer constante $c_1 \leq 1/14$. Assim, escolhendo[1] $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, podemos verificar que $T(n) = \Theta(n^2)$.

Exemplo 1.2. Vamos mostrar que $6n^3 \neq \Theta(n^2)$. A título de contradição, suponha que existam c_2 e n_0 tais que $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Mas, então, divisão por n^2 dá $n \leq c_2/6$, o que não pode ser válido para um valor de n arbitrariamente grande, já que c_2 é constante.

Intuitivamente, os termos de ordem mais baixa de uma função assintoticamente positiva podem ser ignorados na determinação de limites assintoticamente restritos porque eles são insignificantes para grandes valores de n. Quando n é grande, até uma minúscula fração do termo de ordem mais alta é suficiente para dominar os termos de ordem mais baixa. Desse modo, definir c_1 como um valor ligeiramente menor que o coeficiente do termo de ordem mais alta e definir c_2 como um valor ligeiramente maior permite que as desigualdades na definição da notação Θ sejam satisfeitas. Da mesma maneira, o coeficiente do termo de ordem mais alta pode ser ignorado, já que ele só muda c_1 e c_2 por um fator constante igual ao coeficiente.

Exemplo 1.3. Vamos mostras que se $T(n) = an^2 + bn + c$, onde a, b e c são constantes e $a > 0$, então $T(n) = \Theta(n^2)$. Basta encontrar constantes c_1, c_2 e n_0 tais que

$$0 < c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$$

para todo $n \geq n_0$. Note que para qualquer $n \geq 1$, temos que $|b|n \leq |b|n^2$ e $|c| \leq |c|n^2$, então

$$T(n) = an^2 + bn + c \leq an^2 + |b|n + |c| \leq (a + |b| + |c|)n^2.$$

[1] Certamente, existem outras opções para as constantes, mas o importante é que existe *alguma* opção.

Logo podemos tomar $c_2 = a + |b| + |c| > 0$. Notemos agora que

$$T(n) = an^2 + bn + c \geq an^2 - (|b|n + |c|),$$

então para n lo suficientemente grande que assegure que $|b|n + |c| \geq an^2/2$, então temos que

$$T(n) \geq an^2 - \frac{1}{2}an^2 = \frac{1}{2}an^2.$$

Logo podemos tomar $c_1 = a/2$. Para evitar muitos cálculos vamos considerar que $|b|n \geq an^2/4$ e $|c| \geq an^2/4$, então debemos escolher

$$n_0 = \max\left(1, \left\lceil 4\frac{|b|}{a} \right\rceil, \left\lceil 2\sqrt{\frac{|c|}{a}} \right\rceil\right).$$

Tendo em vista que qualquer constante é um polinômio de grau 0, podemos expressar qualquer função constante como $\Theta(n_0)$ ou $\Theta(1)$. Porém, esta última notação é um pequeno abuso porque a expressão não indica qual variável está tendendo a infinito. Usaremos com frequência a notação $\Theta(1)$ para indicar uma constante ou uma função constante em relação a alguma variável.

1.1.2 Notação O

A notação Θ limita assintoticamente uma função acima e abaixo. Quando temos apenas um limite assintótico superior, usamos a notação O

Definição 1.2. Seja $g(n)$ uma função positiva. Dizemos que uma função $f(n)$ pertence a $O\Big(g(n)\Big)$ se existe uma constante positiva c e um natural n_0 tais que

$$0 \leq f(n) \leq c\, g(n) \quad \text{para todo } n \geq n_0.$$

Usamos a notação O para dar um limite superior a uma função, dentro de um fator constante. A Fig xx mostra a intuição por trás da notação O. Para todos os valores n em n_0 ou à direita de n_0, o valor da função $f(n)$ está abaixo de $cg(n)$.

Escrevemos $f(n) = O(g(n))$ para indicar que uma função $f(n)$ é um membro do conjunto $O(g(n))$. Observe que $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, já que a notação Θ é uma noção mais forte que a notação O. Em termos da teoria de conjuntos, escrevemos $\Theta(g(n)) \subset O(g(n))$. Assim, nossa prova de que qualquer função quadrática $an^2 + bn + c$, onde $a > 0$ está em $\Theta(n^2)$ também mostra que qualquer função quadrática desse tipo está em $O(n^2)$. O que pode ser mais surpreendente é que, quando $a > 0$, qualquer função linear $an + b$ está em $O(n^2)$, o que é facilmente verificado fazendo $c = a + |b|$ e $n_0 = \max(1, -b/a)$.

Vamos ver alguns exemplos de utilização de notação O para fixar os conceitos

Exemplo 1.4 . Vamos mostrar que $\log(n) = O(n^\alpha)$ para qualquer $\alpha > 0$. Neste caso é útil a desigualdade[2] $\log(x) \leq x - 1$ se $x \in \mathbb{R}_+$. Em particular podemos garantir que se $n > 1$ então $\log(n) \leq n - 1$. Basta, então, demonstrar que existem constantes c e n_0 tais que

$$n - 1 < cn^\alpha$$

para todo $n > n_0$ e $\alpha > 0$. Notemos que se $\alpha \geq 1$, então $n - 1 < n < n^\alpha$, ou seja podemos tomar $c = 1$ e $n_0 = 1$. No caso em que $0 < \alpha < 1$ a desigualdade $n < n^\alpha$ não funciona. Suponha

[2]Esta desigualdade pode ser demonstrada, por exemplo, olhando que a função $f(x) = x - 1 - \log(x)$ é sempre não negativa.

então $n \geq 2$, então existe um $m \geq 1$ tal que $2^m \leq n < 2^{m+1}$, ou seja que $\log(n) < \log(2^{m+1}) = (m+1)\log(2)$. Por outro lado $n^\alpha \geq (2^m)^\alpha = 2^{\alpha m}$. Juntando as duas desigualdades temos que

$$\frac{\log(n)}{n^\alpha} \leq \frac{(m+1)\log(2)}{2^{\alpha m}}.$$

Agora notemos que a sequência $2^{\alpha m}$ é geométrica com razão $2^\alpha > 1$. Uma sucessão geométrica com razão maior que 1 cresce muito mais rápido que qualquer função linear em m (ou seja $m+1$). Por isso existe um m_0 tal que, para todo $m \geq m_0$

$$2^{\alpha m} \geq (m+1)\log(2) \;\Rightarrow\; \frac{\log(n)}{n^\alpha} \leq 1,$$

lembrando que esta desigualdade é valida se $n \geq n_0 \geq 2^{m_0}$.

Se você já viu a notação O antes, poderá achar estranho que escrevamos, por exemplo, $n = O(n^2)$. Na literatura, verificamos que, às vezes, a notação O é utilizada informalmente para descrever limites assintoticamente justos, isto é, aquilo que definimos usando a notação Θ. Contudo, nestas notas, quando escrevermos $f(n) = O(g(n))$, estaremos simplesmente afirmando que algum múltiplo constante de $g(n)$ é um limite assintótico superior para $f(n)$, sem qualquer menção de precisão. A distinção entre limites assintóticos superiores e limites assintoticamente justos é padrão na literatura de algoritmos.

Usando a notação O, podemos descrever frequentemente o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo. Por exemplo, a estrutura de loop duplamente aninhado do algoritmo de ordenação por inserção analisado em aula produz imediatamente um limite superior $O(n^2)$ para o tempo de execução do pior caso: o custo de cada iteração do loop interno é limitado na parte superior por $O(1)$ (constante), os índices i e j são no máximo n, e o loop interno é executado no máximo uma vez para cada um dos n^2 pares de valores para i e j. Tendo em vista que a notação O descreve um limite superior, quando a empregamos para limitar o tempo de execução do pior caso de um algoritmo temos um limite para o tempo de execução do algoritmo em cada entrada — o enunciado abrangente do qual falamos anteriormente. Desse modo, o limite $O(n^2)$ para o tempo de execução do pior caso da ordenação por inserção também se aplica a seu tempo de execução para toda entrada. Porém, o limite $\Theta(n^2)$ para o tempo de execução do pior caso da ordenação por inserção não implica um limite $\Theta(n^2)$ para o tempo de execução da ordenação por inserção em toda entrada. Por exemplo, vimos também em aulas que, quando a entrada já está ordenada, a ordenação por inserção funciona no tempo $\Theta(n)$.

Tecnicamente, é um abuso dizer que o tempo de execução da ordenação por inserção é $O(n^2)$, visto que, para um dado n, o tempo de execução real varia, dependendo da entrada específica de tamanho n. Quando afirmamos que "o tempo de execução é $O(n^2)$", queremos dizer que existe uma função $f(n)$ que é $O(n^2)$ tal que, para qualquer valor de n, não importando qual entrada específica de tamanho n seja escolhida, o tempo de execução para essa entrada tem um limite superior determinado pelo valor $f(n)$. De modo equivalente, dizemos que o tempo de execução do pior caso é $O(n^2)$.

2 Algoritmos de Ordenamento

A ordenação é um dos problemas fundamentais em programação e ciência da computação. A razão é simples: muitas tarefas dependem de ter os dados organizados de forma eficiente. O problema de ordenamento consiste em:

Problema 2.1. Problema fundamental do ordenamento:

Dado: Um conjunto de n elementos $S = \{a_1, a_2, \ldots, a_n\}$ pertencentes a um conjunto parcialmente ou totalmente ordenável, com uma relação de ordem $\leq$ definida.

Objetivo: Encontrar uma permutação $(b_1, b_2, \ldots, b_n)$ de S tal que:

$$b_1 \leq b_2 \leq \cdots \leq b_n.$$

Considerações:

- A sequência de entrada pode conter elementos repetidos ou distintos.
- O algoritmo deve, idealmente, minimizar o número de comparações e movimentos.
- Ordenação estável: manter a ordem relativa de elementos equivalentes.
- Ordenação in-place: usar memória adicional mínima.

Ordenar não é um fim em si mesmo, mas um passo crítico para otimizar outras operações. Alguns exemplos do cotidiano e da ciência incluem:

Exemplo 2.1 (Busca eficiente em grandes bancos de dados). Imagine um catálogo com milhões de partículas simuladas em uma dinâmica molecular. Procurar uma partícula específica sem ordenação exige percorrer todos os registros. Com os dados ordenados por posição ou energia, podemos usar buscas binárias, reduzindo drasticamente o tempo de busca.

Exemplo 2.2 (Seleção de dados relevantes). Em simulações de física estatística, muitas vezes precisamos encontrar as k partículas com maior energia cinética ou maior momento. Se os dados estiverem desordenados, é necessário varrer o vetor inteiro várias vezes. Se os ordenarmos, selecionar os k elementos é imediato, basta pegar os primeiros k da sequência ordenada.

Exemplo 2.3 (Processamento de sinais e gráficos). Em experimentos físicos, como análise de espectros ou séries temporais, é comum precisar ordenar amplitudes ou frequências. Uma ordenação eficiente permite identificar rapidamente picos relevantes, construir histogramas ou gerar gráficos cumulativos sem gastar tempo computacional excessivo.

Exemplo 2.4 (Jogos, rankings e visualizações). Em simulações ou aplicações científicas interativas, como visualizações de clusters em sistemas físicos ou rankings de partículas por propriedades específicas, ordenar os elementos permite mostrar os mais significativos rapidamente, facilitando análises ou interações em tempo real.

Podemos concluir que estudar algoritmos de ordenação não é apenas aprender a mover números em um vetor. É compreender como organizar informações de forma eficiente, acelerar buscas, tomar decisões inteligentes sobre qual algoritmo usar e construir a base para resolver problemas mais complexos em ciência da computação, física computacional e engenharia.

2.1 Ordenamento simples

Vamos iniciar nosso estudo com algoritmos simples de ordenação. Mas o que significa "simples" aqui? Consideramos um algoritmo de ordenação simples quando a determinação do tempo de execução não envolve a resolução de recorrências complexas. Nesses casos, geralmente é suficiente utilizar álgebra elementar para estimar o tempo de execução do algoritmo.

Seja c_j o tempo necessário para executar a j-ésima linha do pseudocódigo. Então, o tempo total gasto por essa linha ao longo da execução do programa é

$$T_j(n) = c_j t_j,$$

onde t_j representa o número de vezes que a linha foi executada. Se o programa possui m linhas, o tempo total de execução do algoritmo será

$$T(n) = \sum_{j=1}^{m} T_j(n) = \sum_{j=1}^{m} c_j t_j. \tag{1}$$

Além disso, é comum analisar algoritmos considerando diferentes cenários de entrada. O *melhor caso* corresponde à entrada que minimiza o tempo de execução, e seu tempo é denotado por $T_{\min}(n)$. O *pior caso* corresponde à entrada que maximiza o tempo de execução, com tempo $T_{\max}(n)$. Dessa forma, temos que, para qualquer entrada de tamanho n,

$$T_{\min}(n) \leq T(n) \leq T_{\max}(n),$$

ou seja, $T_{\min}(n)$ e $T_{\max}(n)$ fornecem limites inferior e superior para o tempo de execução do algoritmo.

Essa abordagem permite analisar de forma direta o comportamento dos algoritmos simples sem recorrer a métodos mais sofisticados de análise assintótica ou recorrências.

2.1.1 Insertion-Sort

O Insertion-Sort é um dos algoritmos de ordenação mais simples e intuitivos. Ele funciona de maneira semelhante a como uma pessoa organiza cartas na mão: você pega uma carta de cada vez e a insere na posição correta em relação às cartas já ordenadas.

O algoritmo percorre o vetor do segundo elemento até o último. Para cada elemento, ele compara com os anteriores e o "desloca" para a esquerda até encontrar sua posição correta. Este processo garante que, a cada passo, a sub-lista à esquerda do elemento atual esteja ordenada. O Algoritmo 1 mostra o pseudocódigo do Insertion-Sort.

Algorithm 1: Insertion-Sort

Input: Arreglo $A[0..n-1]$
Output: A ordenado em ordem não decrescente

```
1 for j ← 1 to n − 1 do
2     key ← A[j];
3     i ← j − 1;
4     while i ≥ 0 and A[i] > key do
5         A[i + 1] ← A[i];
6         i ← i − 1;
7     A[i + 1] ← key;
```

Olhando para o Algoritmo 1 teremos a seguinte tabela de custos e vezes para cada linha da ordenação por inserção:

linha	*operação*	*custo*	*vezes*
1	Laço `for`	c_1	n
2	Atribuição `key = A[j]`	c_2	$n-1$
3	Atribuição `i = j-1`	c_3	$n-1$
4	Laço `while i >= 0 and A[i] > key`	c_4	$\sum_{j=1}^{n-1} \tau_j$
5	Deslocamento `A[i+1] = A[i]`	c_5	$\sum_{j=1}^{n-1}(\tau_j - 1)$
6	Decremento `i = i-1`	c_6	$\sum_{j=1}^{n-1}(\tau_j - 1)$
7	Inserção `A[i+1] = key`	c_7	$n-1$

Tabela 1: Análise de custos do Insertion-Sort. Neste analises τ_j é número de vezes que o teste do laço `while` na linha 4 é executado para um j fixo. Quando um laço `for` ou `while` termina da maneira usual (isto é, devido ao teste no cabeçalho do laço), o teste é executado uma vez mais do que o corpo do laço.

O tempo de execução do algoritmo é

$$\begin{aligned} T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} \tau_j + c_5 \sum_{j=1}^{n-1} (\tau_j - 1) \\ + c_6 \sum_{j=1}^{n-1} (\tau_j - 1) + c_7(n-1). \end{aligned} \tag{2}$$

Note que mesmo para entradas de um tamanho dado, o tempo de execução pode depender de qual entrada desse tamanho é dada. Em Insertion-Sort, o melhor casso ocorre se o arranjo já está ordenado. Então para cada $j = 1, 2, \ldots, n-1$, descobrimos que `A[i] <= key` na linha 4 quando i tem seu valor inicial $j-1$. Portanto, $\tau_j = 1$ para $j = 1, 2, \ldots, n-1$, e o tempo de execução do melhor caso é

$$\begin{aligned} T_{\min}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} 1 + c_7(n-1) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7). \end{aligned} \tag{3}$$

Podemos expressar esse tempo de execução como $T_{\min}(n) = an + b = O(n)$ para constantes a e b que dependem dos custos de instrução c_j.

Se o arranjo estiver em ordem inversa – ou seja, em ordem decrescente –, resulta o pior caso. Devemos comparar cada elemento `A[j]` com cada elemento do sub-arranjo ordenado inteiro, `A[0..j-1]`, e então $\tau_j = j$ para $j = 1, 2, \ldots, n-1$. Observando que

$$\sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} \quad \text{e} \quad \sum_{j=1}^{n-1} (j-1) = \frac{(n-1)(n-2)}{2} \tag{4}$$

conseguimos

$$\begin{aligned} T_{\max}(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \frac{(n-1)n}{2} \\ &\quad + (c_5 + c_6)\frac{(n-1)(n-2)}{2} + c_7(n-1) \\ &= (c_4 + c_5 + c_6)\frac{n^2}{2} + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7\right) n \\ &\quad - \left(c_2 + c_3 - c_5 - c_6 + c_7\right). \end{aligned} \tag{5}$$

Podemos expressar esse tempo de execução do pior caso como $T_{\max}(n) = an^2 + bn + c = O(n^2)$ para constantes, a, b e c que, mais uma vez, dependem dos custos de instrução c_j.

Vamos implementar o Algoritmo 1 utilizando C (que é nossa linguajem modelo), veja Code 1.

```c
void insertionSort(int arr[], int n) {
    int j; for (j = 1; j < n; j++) {
        int key = arr[j];
        int i = j - 1;
        while (i >= 0 && arr[i] > key) {
            arr[i + 1] = arr[i];
            i--;
        }
        arr[i + 1] = key;
    }
}
```

Code 1: Insertion-Sort em C

2.1.2 Selection-Sort

O Selection-Sort é outro algoritmo de ordenação simples, mas funciona de maneira diferente do Insertion-Sort. Em vez de inserir cada elemento em uma sub-lista já ordenada, o Selection-Sort encontra o menor elemento da parte não ordenada do vetor e o coloca na posição correta, trocando-o com o elemento que deveria estar ali.

O algoritmo percorre o vetor do primeiro até o penúltimo elemento. Para cada posição i, ele busca o menor elemento do sub-vetor `A[i..n-1]` e o troca com `A[i]`. Desta forma, a cada passo, a sub-lista à esquerda de i já está ordenada.

O Algoritmo 2 mostra o pseudocódigo do Selection-Sort.

Algorithm 2: Selection-Sort

Input: Arreglo $A[0..n-1]$
Output: A ordenado em ordem não decrescente

```
1 for i ← 0 to n − 2 do
2     min_index ← i;
3     for j ← i + 1 to n − 1 do
4         if A[j] < A[min_index] then
5             min_index ← j;
6     if min_index ≠ i then
7         temp ← A[i];
8         A[i] ← A[min_index];
9         A[min_index] ← temp;
```

Olhando para o Algoritmo 2 teremos a seguinte tabela de custos e vezes para cada linha da ordenação por seleção:

linha	*operação*	*custo*	*vezes*
1	Laço externo `for i = 0..n-2`	c_1	n
2	Inicialização `min_index = i`	c_2	$n-1$
3	Laço interno `for j = i+1..n-1`	c_3	$\sum_{i=0}^{n-2}(n-i)$
4	Comparação `if A[j] < A[min_index]`	c_4	$\sum_{i=0}^{n-2}(n-1-i)$
5	Atribuição `min_index = j`	c_5	$\sum_{i=0}^{n-2}\tau_i$
6	Teste `if min_index` $\neq$ `i`	c_6	$n-1$
7	Swap `A[i]` $\leftrightarrow$ `A[min_index]`	c_7	$\sum_{i=0}^{n-2}\sigma_i$

Tabela 2: Análise de custos do Selection-Sort. Aqui, τ_i é o número de vezes que a atribuição `min_index = j` é executada no passo i, e σ_i indica se o swap é realizado no passo i (0 ou 1). O número de comparações internas é dado por $\sum_{i=0}^{n-2}(n-1-i)$.

Note que c_7 representa o custo do `swap` (bloque de código composto pelas linhas 7,8 e 9). O tempo de execução do algoritmo é

$$\begin{aligned} T(n) = c_1 n + c_2(n-1) + c_3\sum_{i=0}^{n-2}(n-i) + c_4\sum_{i=0}^{n-2}(n-i-1) \\ + c_5\sum_{i=0}^{n-2}\tau_i + c_6(n-1) + c_7\sum_{i=0}^{n-2}\sigma_i. \end{aligned} \tag{6}$$

Observando que

$$\sum_{i=0}^{n-2}(n-i) = \frac{(n+2)(n-1)}{2} \quad \text{e} \quad \sum_{i=0}^{n-2}(n-i-1) = \frac{(n-1)n}{2}, \tag{7}$$

temos que

$$\begin{aligned} T(n) = c_1 n + c_2(n-1) + \frac{c_3}{2}(n+2)(n-1) + \frac{c_4}{2}(n-1)n \\ + c_5\sum_{i=0}^{n-2}\tau_i + c_6(n-1) + c_7\sum_{i=0}^{n-2}\sigma_i. \end{aligned} \tag{8}$$

O melhor caso acontece se o arranjo já está ordenado. Então teremos que a comparação `A[j]<A[[min_index]` é sempre falsa e, pelo tanto, a atribuição `min_index = j` (linha 5) nunca é executada. O anterior implica que $\tau_i = 0$ para $i = 0, 1, \ldots n-2$. Neste caso na linha 6 temos que `min_index == i`, então o `swap` nunca é executado, ou seja teremos $\sigma_i = 0$ para $i = 0, 1, \ldots, n-2$. Então temos

$$\begin{aligned} T_{\min}(n) &= c_1 n + c_2(n-1) + \frac{c_3}{2}(n+2)(n-1) + \frac{c_4}{2}(n-1)n + c_6(n-1) \\ &= (c_3 + c_4)\frac{n^2}{2} + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} + c_6\right) n - (c_2 + c_3 + c_6)\,. \end{aligned} \tag{9}$$

O pior caso para aplicar o Algoritmo 2 es quando o arranjo está ordenado em ordem inversa. Neste caso a comparação `A[j]<A[[min_index]` é sempre verdadeira e, pelo tanto, a atribuição `min_index = j` sempre é executada, $\tau_i = n - i - 1$ para $i = 0, 1, \ldots, n-2$. Além disso, a operação `swap` sempre sera executada, deixando $\sigma_i = 1$ para todo $i = 0, 1, \ldots, n-2$. Neste casso

$$\begin{aligned} T_{\max}(n) = c_1 n + c_2(n-1) + \frac{c_3}{2}(n+2)(n-1) + \frac{c_4}{2}(n-1)n \\ + c_5 \sum_{i=0}^{n-2}(n-i-1) + c_6(n-1) + c_7 \sum_{i=0}^{n-2} 1. \end{aligned} \tag{10}$$

Usando a segunda soma em (7) obtemos que

$$\begin{aligned} T_{\max}(n) = c_1 n + c_2(n-1) + \frac{c_3}{2}(n+2)(n-1) + \frac{c_4}{2}(n-1)n \\ + \frac{c_5}{2}(n-1)n + c_6(n-1) + c_7(n-1). \end{aligned} \tag{11}$$

Em geral, ambos os casos o pior e o melhor podem ser escretos como $T_{\max}(n) = an^2 + bn + c = O(n^2)$. No Code 2 podemos ver uma implementação da busca por seleção.

```
1  void selectionSort(int arr[], int n) {
2      int i, j;
3      for (i = 0; i < n - 1; i++) {
4          int min_index = i;
5          for (j = i + 1; j < n; j++) {
6              if (arr[j] < arr[min_index]) {
7                  min_index = j;
8              }
9          }
10         // troca em uma linha
11         if (min_index != i) {
12             int tmp = arr[i];
13             arr[i] = arr[min_index];
14             arr[min_index] = tmp;
15         }
16     }
17 }
```

Code 2: Selection Sort em C

2.1.3 Bubble-Sort

O Bubble-Sort é um algoritmo de ordenação simples que funciona repetidamente percorrendo a lista, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. O processo se repete até que nenhum par de elementos precise ser trocado, garantindo que o vetor esteja ordenado.

O algoritmo percorre o vetor várias vezes. Em cada passagem, o maior elemento "borbulha" até sua posição correta no final da lista. Após cada passagem completa, o próximo maior elemento estará na posição correta, e assim sucessivamente. Este processo continua até que toda a lista esteja ordenada.

O Algoritmo 3 mostra o pseudocódigo do Bubble-Sort.

Algorithm 3: Bubble-Sort

Input: Arreglo $A[0..n-1]$
Output: A ordenado em ordem não decrescente

```
1 for i ← 0 to n − 2 do
2     for j ← 0 to n − i − 2 do
3         if A[j] > A[j + 1] then
4             temp ← A[j];
5             A[j] ← A[j + 1];
6             A[j + 1] ← temp;
```

Olhando para este algoritmo teremos a seguinte tabela de custos e vezes para cada linha

linha	*operação*	*custo*	*vezes*
1	Laço externo `for i = 0..n-2`	c_1	n
2	Laço interno `for j = 0..n-2-i`	c_2	$\sum_{i=0}^{n-2}(n-i)$
3	Comparação `if A[j] > A[j+1]`	c_3	$\sum_{i=0}^{n-2}(n-1-i)$
4	Swap `A[j]` $\leftrightarrow$ `A[j+1]`	c_4	$\sum_{i=0}^{n-2}\sigma_i$

Tabela 3: Análise de custos do Bubble-Sort. Aqui, σ_i indica o número de operações swap realizadas na iteração i do laço `for` externo (note que $0 \leq \sigma_i < n-i-1$). O número de comparações internas é dado por $\sum_{i=0}^{n-2}(n-1-i)$.

O tempo de execução do Bubble-Sort é dado por

$$T(n) = c_1 n + c_2 \sum_{i=0}^{n-2}(n-i) + c_3 \sum_{i=0}^{n-1}(n-i-1) + c_4 \sum_{i=0}^{n-2}\sigma_i, \tag{12}$$

que pode ser escrita, utilizando as somas 7, como

$$T(n) = c_1 n + \frac{c_2}{2}(n+2)(n-1) + \frac{c_3}{2}(n-1)n + c_4 \sum_{i=0}^{n-2}\sigma_i. \tag{13}$$

O melhor caso acontece, como é logico, quando o arranjo esta ordenado. Neste caso sempre é falso que `A[j]> A[j+1]` (linha 3), então $\sigma_i = 0$ para todo $i = 0, 1, \ldots, n-2$. Neste caso o tempo de execução do algoritmo é

$$T_{\min}(n) = (c_2 + c_3)\frac{n^2}{2} + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2}\right) n - c_2. \tag{14}$$

Na pior situação de ordenamento teremos o arranjo ordenado na ordem inversa. Neste caso sempre é verdadeiro que `A[j]> A[j+1]`, então a operação `swap` sempre será executada e $\sigma_i = n - i - 1$ para os possíveis valores de i. O pior tempo de execução será

$$\begin{aligned} T_{\max}(n) &= c_1 n + \frac{c_2}{2}(n+2)(n-1) + \frac{c_3}{2}(n-1)n + c_4 \sum_{i=0}^{n-2}(n-i-1) \\ &= c_1 n + \frac{c_2}{2}(n+2)(n-1) + \frac{c_3}{2}(n-1)n + \frac{c_4}{2}(n-1)n, \end{aligned} \tag{15}$$

que pode ser escreto como

$$T_{\max}(n) = (c_2 + c_3 + c_4)\frac{n^2}{2} + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right) n - c_2. \tag{16}$$

No Code 3 podemos ver uma implementação da busca por borbulha.

```c
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // troca em uma linha
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}
```

Code 3: Bubble Sort em C

2.1.4 Comparação

Nesta seção, analisamos o crescimento temporal dos algoritmos simples em termos de complexidade. A Tabela 4 resume o comportamento nos melhores e piores casos.

Algoritmo	*melhor caso*	*pior caso*
Insertion-Sort	$O(n)$	$O(n^2)$
Bubble-Sort	$O(n^2)$	$O(n^2)$
Selection-Sort	$O(n^2)$	$O(n^2)$

Tabela 4: Comparação da complexidade temporal nos melhores e piores casos dos algoritmos de ordenação.

Observações importantes:

- *Insertion-Sort* apresenta ótimo desempenho ($O(n)$) quando a lista já está ordenada, sendo eficiente para listas pequenas ou quase ordenadas.

- *Bubble-Sort* possui complexidade $O(n^2)$ na maioria dos casos; seu melhor caso é $O(n)$ se implementado com uma bandeira que detecta que não houve trocas.
- *Selection-Sort* sempre realiza $O(n^2)$ comparações, independentemente da ordem inicial, tornando-o menos eficiente que *Insertion-Sort* para listas quase ordenadas.

Para visualizar o crescimento, os gráficos de $T(n)$ vs n mostram claramente:

- A linha de *Insertion-Sort* no melhor caso cresce aproximadamente de forma linear.
- *Bubble-Sort* e *Selection-Sort* apresentam crescimento quadrático.

Interpretação: Para entradas pequenas ou quase ordenadas, *Insertion-Sort* é preferível. Para entradas maiores, algoritmos quadráticos tornam-se rapidamente ineficientes, o que motiva o uso de métodos mais avançados (*Merge-Sort*, *Quick-Sort*, etc.).

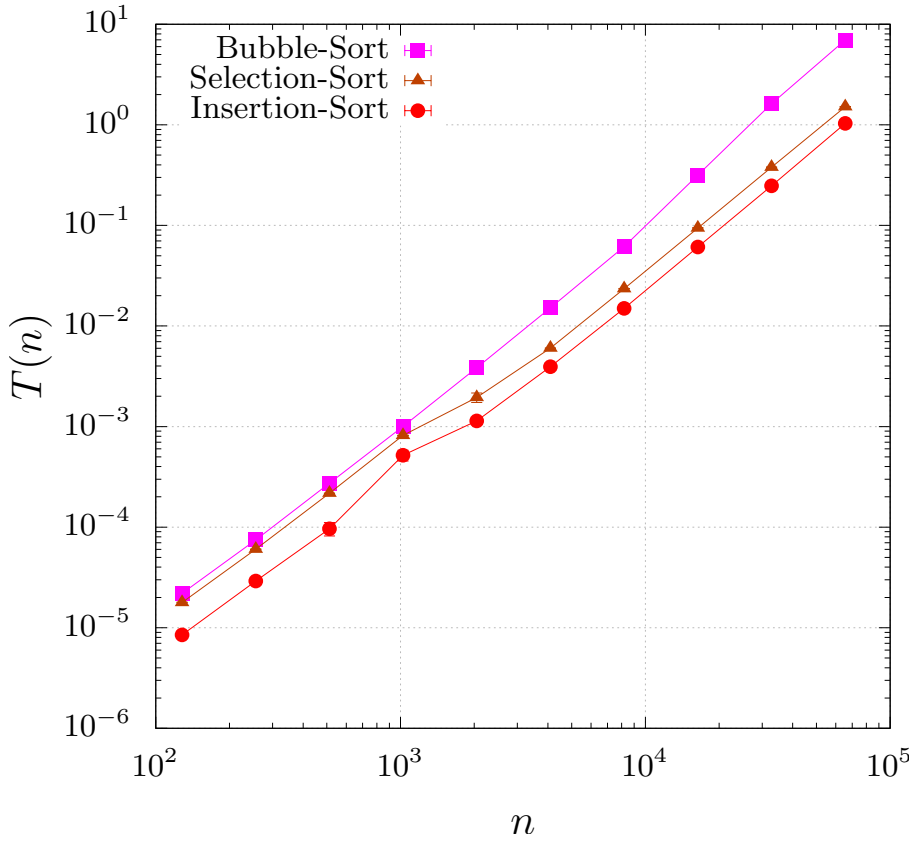

Fig. 1: Crescimento temporal $T(n)$ dos algoritmos.