

- **Finalização:** Ao final da execução do laço **for**, temos $j = n$, e portanto a invariante corresponde a dizer que o vetor $A[0..n - 1]$ obtido ao final da execução do algoritmo está ordenado, e é uma permutação do vetor original $A[0..n - 1]$. Assim, concluímos a prova da correção do algoritmo InsertionSort.

Exercício 83. Prove que o algoritmo BubbleSort a seguir é correto.

```

1 for i = 0 to n - 2 do
2   for j = 0 to n - 2 - i do
3     if A[j + 1] < A[j] then
4       swap A[j] and A[j + 1];
5     end
6   end
7 end

```

Algoritmo 3: BubbleSort($A[0..n - 1]$)

Exercício 84. Prove que o algoritmo BubbleSort2 [8] a seguir é correto.

```

1 for i = 0 to n - 2 do
2   for j = n - 1 downto i + 1 do
3     if A[j] < A[j - 1] then
4       swap A[j] and A[j - 1];
5     end
6   end
7 end

```

Algoritmo 4: BubbleSort2($A[0..n - 1]$)

Exercício 85. Prove que o algoritmo SelectionSort a seguir é correto.

```

1 for i = 0 to n - 2 do
2   min ← i;
3   for j = i + 1 to n - 1 do
4     if A[j] < A[min] then
5       min ← j;
6     end
7   end
8   swap A[i] and A[min];
9 end

```

Algoritmo 5: SelectionSort($A[0..n - 1]$)

A complexidade do algoritmo de ordenação por inserção

AULA 06

Agora faremos uma análise da complexidade do algoritmo de ordenação por inserção semelhante análoga à feita para a busca sequencial. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada que neste caso é o número n de elementos a serem ordenados. Novamente assumiremos que cada linha do pseudocódigo é executada em tempo constante, mas este tempo pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde ao tempo de execução da i -ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O laço **for** da linha 1 é executado n vezes, enquanto que o corpo do laço é executado $n - 1$ vezes, uma vez para cada $j = 1, \dots, n - 1$. Denotaremos por t_j o número de vezes que o teste do laço **while** da linha 4 é executado, de forma que temos o seguinte custo por linha:

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1 \cdot n$
2	c_2	$n - 1$	$c_2 \cdot (n - 1)$
3	c_3	$n - 1$	$c_3 \cdot (n - 1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4 \cdot \sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1} (t_j - 1)$	$c_5 \cdot \sum_{j=1}^{n-1} (t_j - 1)$
6	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$	$c_6 \cdot \sum_{j=1}^{n-1} (t_j - 1)$
8	c_8	$n - 1$	$c_8 \cdot (n - 1)$

$$T(n) = c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot \sum_{j=2}^n t_j + c_5 \cdot \sum_{j=2}^n (t_j - 1) + c_6 \cdot \sum_{j=2}^n (t_j - 1) + c_8 \cdot (n - 1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, um vetor que tenha mais elementos a serem reposicionados terá um custo maior para ser ordenado. Portanto, a análise do **melhor caso** se dá quando o vetor já estiver ordenado pois $t_j = 1$, para todo $2 \leq j \leq n$:

$$\begin{aligned} T_b(n) &= c_1 \cdot n + c_2 \cdot (n - 1) + c_3 \cdot (n - 1) + c_4 \cdot (n - 1) + c_8 \cdot (n - 1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8) \cdot n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

ou seja, uma função linear de n . Por outro lado, a análise do pior caso se dá quando o vetor estiver ordenado decrescentemente pois $t_j = j$ (por que?), e portanto

$$T_w(n) = c_1 \cdot n + (c_2 + c_3 + c_8) \cdot (n - 1) + c_4 \cdot \left(\frac{(n-1) \cdot n}{2}\right) + (c_5 + c_6) \cdot \left(\frac{(n-2) \cdot (n-1)}{2}\right)$$

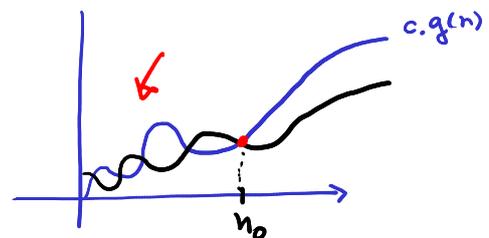
ou seja, uma **função quadrática de n** .

A forma de análise feita para InsertionSort acima, assim como para SequentialSearch na seção anterior, apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador, da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades, e fazer uma análise que seja independente destes aspectos, consiste na utilização de uma notação adequada, a *notação assintótica*, que considera o comportamento de funções no limite, isto é, para valores suficientemente grandes do parâmetro n . A ideia é que possamos pegar uma função como $T_w(n) = c_1 + c_2 \cdot n + c_3 \cdot (n - 1) + c_6 + c_8$ que expressa o custo no pior caso do algoritmo de busca sequencial, e dizer que ela cresce como n , sem a necessidade de considerar as constantes. Faremos isto considerando o conjunto das funções que são limitadas superiormente por um múltiplo constante de n . Observe que podemos facilmente construir uma cota superior para a função $T_w(n)$ da seguinte forma $T_w(n) = c_1 + c_2 \cdot n + c_3 \cdot (n - 1) + c_6 + c_8 \leq c_1 \cdot n + c_2 \cdot n + c_3 \cdot n - c_3 + c_6 \cdot n + c_8 \cdot n \leq (c_1 + c_2 + c_3 + c_6 + c_8) \cdot n \leq c \cdot n$ para qualquer constante $c \geq c_1 + c_2 + c_3 + c_6 + c_8$ e $n \geq 1$. Neste caso, dizemos que a função $T_w(n)$ é $O(n)$, ou seja, que $T_w(n)$ é de ordem n . Formalmente, temos a seguinte definição para o conjunto $O(g(n))$ que contém todas as funções que são da ordem de $g(n)$:

Definição 25. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.\}$*

Observe que uma função $f(n)$ pode estar em $O(g(n))$ mesmo que $f(n) > g(n), \forall n$. O ponto importante é que $f(n)$ tem que ser limitada por um múltiplo constante de $g(n)$. A relação entre $f(n)$ e $g(n)$ para valores pequenos de n também é desconsiderada. Intuitivamente, os termos de menor ordem

$$\underline{f(n)} = \underline{O(g(n))}$$



de uma função assintoticamente positiva podem ser ignorados na determinação da cota superior porque são insignificantes para valores grandes do parâmetro n . Assim, quando n é grande qualquer porção ou fração do termo de maior ordem é suficiente para dominar os termos de menor ordem.

Normalmente, escrevemos $T(n) \in O(n^2)$ para dizer que $T(n)$ é $O(n^2)$ já que $O(n^2)$ é um conjunto. No entanto, é comum encontrarmos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$. A conveniência do uso da igualdade será vista posteriormente, mas o importante aqui é entender que esta igualdade é unidirecional, e portanto não pode ser confundida com a igualdade tradicional. Por exemplo, escrevemos $T(n) = O(n^2)$, mas $O(n^2) = T(n)$ não é correto. O número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece: por exemplo, na expressão $\sum_{i=1}^n O(i)$ contém apenas uma função anônima (a função que tem parâmetro i), e portanto esta expressão não é o mesmo que $O(1) + O(2) + \dots + O(n)$ (que não possui uma interpretação clara). A notação assintótica também pode aparecer do lado esquerdo de uma equação: $2n^2 + O(n) = O(n^2)$. Neste caso, independentemente da forma como as funções anônimas são escolhidas do lado esquerdo da equação, existe uma forma de escolher funções anônimas do lado direito da equação de forma que a equação se verifique. No caso do exemplo acima, temos que para qualquer $f(n) = O(n)$, existe uma função $g(n) = O(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$.

Equações também podem ser encadeadas como em $2n^2 + 3n + 1 = 2n^2 + O(n) = O(n^2)$, e podem ser interpretadas separadamente de acordo com as regras anteriores. Assim, a primeira equação nos diz que existe alguma função $f(n) = O(n)$ para a qual a equação se verifica para todo n . A segunda equação nos diz que para toda função $g(n) = O(n)$, existe uma função $h(n) = O(n^2)$ tal que a equação se verifica para todo n . Este encadeamento é transitivo, ou seja, podemos concluir que $2n^2 + 3n + 1 = O(n^2)$.

O lema a seguir nos permite utilizar limites para utilizar a notação assintótica:

Lema 26. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, incluindo o caso em que $c = 0$, então $f(n) = O(g(n))$.

Demonstração. Exercício. □

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lfloor \lg n \rfloor}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lfloor \lg n \rfloor} \leq 2^{\lfloor \lg n \rfloor + 1} = 2 \cdot 2^{\lfloor \lg n \rfloor} = 2 \cdot g(n), \forall n$. No entanto, o limite $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 27. 1. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = O(g(n))$ e $g(n) = O(f(n))$;

2. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $g(n) \neq O(f(n))$;

3. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$ então $f(n) \neq O(g(n))$, mas $g(n) = O(f(n))$.

Demonstração. Exercício. □

No caso de InsertionSort, a análise do pior caso nos dá a função $T_w(n) = c_1 \cdot n + (c_2 + c_3 + c_8) \cdot (n-1) + c_4 \cdot \frac{(n-1) \cdot n}{2} + (c_5 + c_6) \cdot \frac{(n-2) \cdot (n-1)}{2}$ que é $O(n^2)$.

Como exercício, mostre em detalhes que a complexidade do pior caso de InsertionSort é $O(n^2)$.

Assim, considerando as expressões (ou polinômios) construídas(os) até agora, observamos que a classe de complexidade é obtida considerando-se o monômio de maior grau sem levar em conta o coeficiente. Portanto, a construção do polinômio a partir do custo de cada linha do algoritmo não é uma estratégia eficiente porque no final consideraremos apenas a parcela mais significativa, ou seja, o monômio de maior grau. Vamos então buscar diretamente a parte do algoritmo que nos dá este monômio de maior grau. Observando a Tabela 9 concluímos que o termo quadrático vem da linha 4, mais precisamente da comparação $A[i] > key$ que é executada em cada iteração do laço **for**. Então podemos fazer uma análise bem mais direta do que a feita anteriormente para chegarmos à mesma conclusão. Como durante a i -ésima iteração do laço **for**, a linha 4 é executada i vezes, temos:

$$T_w(n) = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} = O(n^2).$$

$\exists c, n_0 > 0 : \frac{(n-1) \cdot n}{2} \leq c \cdot n^2, \forall n \geq n_0$
 $\frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2} = \frac{1}{2} \cdot n^2, \forall n \geq 1$

A análise do melhor caso também pode ser feita da mesma forma considerando que a cada iteração do laço **for**, a linha 4 é executada uma única vez:

$$\rightarrow T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

Da mesma forma, na busca sequencial o custo linear do pior caso pode ser obtido calculando diretamente o número de comparações feitas na linha 2. A notação O nos dá uma cota superior para o custo de execução de algoritmos, mas ela também pode ser utilizada para estabelecer uma cota para a complexidade de espaço utilizado durante a execução de um algoritmo. Tanto a busca sequencial quanto o algoritmo de ordenação por inserção não necessitam de espaço adicional de armazenamento, e portanto, em ambos os casos a complexidade é constante, ou seja, é igual a $O(1)$. Dizemos que algoritmos de ordenação que não demandam espaço adicional fazem a ordenação *in place*. Posteriormente estudaremos algoritmos que necessitam de espaço adicional. Na tabela abaixo, resumimos as análises feitas até agora:

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$

Uma ferramenta bastante útil na análise assintótica é conhecida como *regra do máximo*:

$$O(f(n) + g(n)) = O(\max(f(n), g(n))) \quad (3.1)$$

Depois de alguns exercícios, e de apresentarmos mais alguns detalhes sobre a notação assintótica, estudaremos um pouco da chamada análise do caso médio. A análise do melhor caso nos dá uma ideia de situações específicas em que o algoritmo tem a melhor performance possível, mas a análise do melhor caso não costuma ser muito informativa e normalmente não é relevante. A análise do pior caso, por outro lado, tem bastante relevância e será explorada exaustivamente nas próximas seções. Ela é importante porque nos fornece o pior cenário possível para o algoritmo. Com isto sabemos que o algoritmo não pode ter um comportamento menos eficiente do que o apresentado pela análise do pior caso. No entanto, esta análise pode ser excessivamente pessimista considerando uma situação mais realista. Por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A análise do caso médio pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis, o que não corresponde à média entre as análises do melhor e pior casos.

Por fim, é importante ter em mente que a notação assintótica nos permite analisar a taxa de crescimento, ou ordem de crescimento do tempo de execução de um algoritmo, e portanto as simplificações feitas na obtenção da cota superior não devem ser esquecidas em situações práticas. Por exemplo, considere dois algoritmos A e B com complexidades, respectivamente, iguais a $O(n^2)$ e $O(n^3)$. Qual dos dois algoritmos é mais eficiente? Para valores grandes de n certamente o algoritmo A é mais eficiente,

mas devemos levar em consideração que no cálculo destas classes de complexidade diversas constantes foram ignoradas. Se soubéssemos, por exemplo, que o algoritmo A realiza $100 \cdot n^2$ operações, enquanto que o algoritmo B realiza $5 \cdot n^3$ operações para resolver o mesmo problema, então agora sabemos que para $n < 20$ o algoritmo B é mais eficiente.

Exercício 86. Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios 7 e 9.

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort			
Selection sort			

Exercício 87. Mostre que $n = O(n^2)$. $\exists c, n_0 : n \leq c \cdot n^2, \forall n \geq n_0$

Tomar $c = 1, n_0 = 1$.

Exercício 88. Mostre que $100n + 5 = O(n^2)$.

Exercício 89. Mostre que $\frac{n(n-1)}{2} = O(n^2)$.

Suponha que $\exists c, n_0 :$
 $n^2 \leq c \cdot n^2, \forall n \geq n_0. \quad n_0 \gg c.$
 \Downarrow
 $c \geq n, \forall n \geq n_0. \quad \nexists.$

Exercício 90. Mostre que $n^3 \neq O(n^2)$.

Exercício 91. Sejam $f(n), g(n)$ e $h(n)$ funções dos inteiros não-negativos nos reais positivos. Mostre que se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$.

Assim, como $O(g(n))$ estabelece uma cota superior para funções, o conjunto $\Omega(g(n))$ estabelece uma cota inferior para as funções:

Definição 28. Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $\Omega(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c \cdot g(n) \leq f(n), \forall n \geq n_0$. Alternativamente, $\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$

Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n , o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor caso. Por exemplo, no melhor caso, o algoritmo InsertionSort é $\Omega(n)$, e portanto, o tempo de execução do algoritmo InsertionSort está entre $\Omega(n)$ e $O(n^2)$. A definição alternativa para o conjunto $\Omega(g(n))$ em termos de limites é dada pelo lema a seguir:

Lema 29. Uma função $f(n) = \Omega(g)$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$, incluindo o caso em que o limite é igual a ∞ .

Demonstração. Exercício. □

A forma mais precisa de expressar o comportamento assintótico de um algoritmo é fornecendo cotas superiores e inferiores ao mesmo tempo. No parágrafo anterior, apresentamos uma cota superior e uma cota inferior para o algoritmo InsertionSort. No entanto, estas cotas são de classes diferentes, o conjunto $\Theta(g(n))$, definido a seguir, é utilizado quando ambas as cotas são da mesma classe.

Definição 30. Seja g uma função dos inteiros não-negativos nos reais positivos. Então $\Theta(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem constantes reais positivas c_1 e c_2 , e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$. Alternativamente, $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.\}$

Como qualquer constante pode ser vista como um polinômio de grau 0, podemos representar funções constantes como $\Theta(n^0)$, ou simplesmente, $\Theta(1)$. O lema a seguir apresenta uma caracterização do conjunto $\Theta(g(n))$ em termos de limite:

Lema 31. Uma função $f(n) = \Theta(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$, para alguma constante $0 < c < \infty$.

Demonstração. Exercício. □

Teorema 32. 1. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;

2. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $f(n) \neq \Theta(g(n))$;

3. Se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$ então $f(n) = \Omega(g(n))$, mas $f(n) \neq O(g(n))$.

Demonstração. Exercício. □

Exercício 92. Prove que $\sum_{i=1}^n i^k = \Theta(n^{k+1})$ para qualquer inteiro $k \geq 0$ fixado.

Teorema 33. Dadas funções $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Demonstração. Exercício. □

Lema 34. 1. $f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;

2. Se $f(n) = \Theta(g(n))$ então $g(n) = \Theta(f(n))$;

3. Θ define uma relação de equivalência sobre as funções. Cada conjunto $\Theta(f(n))$ é uma classe de equivalência que chamamos de classe de complexidade;

4. $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$;

5. $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$;

Definição 35. Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $o(g(n)) = \{f(n) : \text{para qualquer constante positiva } c, \text{ existe uma constante positiva } n_0 \text{ tal que } 0 \leq f(n) < c \cdot g(n), \forall n \geq n_0.\}$

Lema 36. Uma função $f(n) = o(g)$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Definição 37. Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $\omega(g(n)) = \{f(n) : \text{para qualquer constante positiva } c, \text{ existe uma constante positiva } n_0 \text{ tal que } 0 \leq c \cdot g(n) < f(n), \forall n \geq n_0.\}$

Lema 38. Uma função $f(n) = \omega(g(n))$ se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, se este limite existir.

Lema 39. Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$ então $f(n) = O(h(n))$, ou seja, a notação O é transitiva. Também são transitivos Ω, Θ, o e ω .

Teorema 40. $\lg n = o(n^\alpha), \forall \alpha > 0$. Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de n (incluindo potências fracionárias)

Teorema 41. $n^k = o(2^n), \forall k > 0$. Ou seja, potências de n crescem mais lentamente que a função exponencial 2^n . Mais ainda, potências de n crescem mais lentamente do que qualquer função exponencial $c^n, c > 1$.

Exercício 93. Mostre que $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 94. Mostre que $6n^3 \neq \Theta(n^2)$.

Exercício 95. Sejam $f(n), g(n)$ e $h(n)$ funções não-negativas tais que $f(n) = O(h(n))$ e $g(n) = O(h(n))$. Prove que $f(n) + g(n) = O(h(n))$.

3.1.3 Análise do caso médio

Nas sessões anteriores fizemos a análise do melhor caso e do pior caso para a busca sequencial, e para o algoritmo de ordenação por inserção. Nesta seção definiremos cada uma destas noções, assim como a análise do caso médio. A complexidade do pior caso consiste em considerar, dentre todas as entradas possíveis de um dado tamanho, aquela ou aquelas entradas para as quais o algoritmo em consideração executa o maior número de passos possível, ou seja, o maior número de operações básicas:

Definição 42. Sejam D_n o conjunto das entradas de tamanho n para o algoritmo em questão, e $I \in D_n$. Seja $t(I)$ o número de operações básicas executadas pelo algoritmo na entrada I . Definimos a função W , que denota a complexidade do pior caso, por:

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

Analogamente, a complexidade do melhor caso é dada por:

Definição 43. Sejam D_n o conjunto das entradas de tamanho n para o algoritmo em questão, e $I \in D_n$. Seja $t(I)$ o número de operações básicas executadas pelo algoritmo na entrada I . Definimos a função B , que denota a complexidade do melhor caso, por: