

Análise da Complexidade de Algoritmos

Estrutura de Dados — QXD0010



UNIVERSIDADE
FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2025

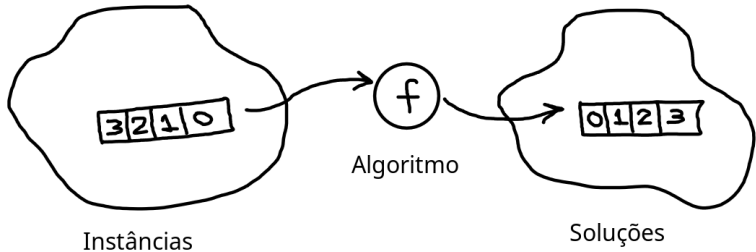


Objetivo

- Estimar o tempo de execução de um algoritmo de forma analítica.
- Estimar a pior entrada que pode ser dada a um algoritmo (aquela que demorará mais tempo para ser executada).
- Comparar a eficiência de diferentes algoritmos para um mesmo problema usando a análise assintótica.

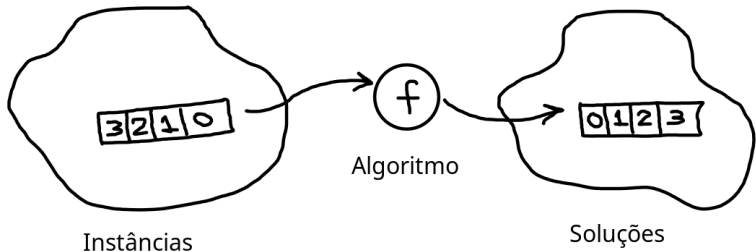
Problema Computacional

Um **problema computacional** é uma relação entre um conjunto de instâncias e um conjunto de soluções.



Problema Computacional

Um **problema computacional** é uma relação entre um conjunto de instâncias e um conjunto de soluções.



Um **algoritmo** é uma sequência finita de instruções que recebe uma **instância** (entrada) de um problema computacional e devolve uma **solução** (saída) correspondente à instância recebida. Cada instrução é um comando que indica ao computador uma ação básica a ser executada.

Algoritmo

Um **algoritmo** é um procedimento computacional composto de 3 partes:

1. **Entrada de dados:** são os dados do algoritmo informados pelo usuário.
2. **Processamento de dados:** são os procedimentos utilizados para chegar ao resultado. É responsável pela obtenção dos dados de saída com base nos dados de entrada.
3. **Saída de dados:** são os dados já processados, apresentados ao usuário.

Algoritmo

Um **algoritmo** é um procedimento computacional composto de 3 partes:

1. **Entrada de dados:** são os dados do algoritmo informados pelo usuário.
2. **Processamento de dados:** são os procedimentos utilizados para chegar ao resultado. É responsável pela obtenção dos dados de saída com base nos dados de entrada.
3. **Saída de dados:** são os dados já processados, apresentados ao usuário.

Muitas vezes, um mesmo problema pode ser resolvido por vários algoritmos diferentes. Logo, é importante saber comparar algoritmos para determinar qual o melhor para cada contexto.

Análise de Algoritmos

- A **análise de algoritmos** é a área que estuda como estimar **teoricamente** a quantidade de recursos que um algoritmo precisará a fim de resolver um problema computacional.

Análise de Algoritmos

- A **análise de algoritmos** é a área que estuda como estimar **teoricamente** a quantidade de recursos que um algoritmo precisará a fim de resolver um problema computacional.
- Ao se analisar um algoritmo, estamos geralmente preocupados com duas medidas:
 - **tempo de execução** (ou tempo de processamento)
 - **espaço de memória** utilizado pelo algoritmo



Análise de Algoritmos

- A **análise de algoritmos** é a área que estuda como estimar **teoricamente** a quantidade de recursos que um algoritmo precisará a fim de resolver um problema computacional.
- Ao se analisar um algoritmo, estamos geralmente preocupados com duas medidas:
 - **tempo de execução** (ou tempo de processamento)
 - **espaço de memória** utilizado pelo algoritmo



- Neste momento, nos interessa apenas estudar o tempo de execução, mas a análise feita aqui se estende também à análise do espaço de memória.

Análise Empírica × Análise Matemática

Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens:

- **análise empírica:** execução dos programas no computador e posterior comparação do uso dos recursos.
- **análise matemática:** estudo das propriedades do algoritmo.

Análise empírica

Essa análise avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado.

Essa análise avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado.

Vantagens:

- avaliar o desempenho de uma determinada configuração de computador/linguagem;
- considerar custos não aparentes. Por exemplo, o custo da alocação de memória;
- comparar computadores;
- comparar linguagens.

Análise empírica

Desvantagens:

- Em certos contextos, não é uma boa opção. **Por quê?**

Análise empírica

Desvantagens:

- Em certos contextos, não é uma boa opção. **Por quê?**
- **Depende do compilador**
 - Pode preferir algumas construções ou otimizar melhor.

Análise empírica

Desvantagens:

- Em certos contextos, não é uma boa opção. **Por quê?**
- **Depende do compilador**
 - Pode preferir algumas construções ou otimizar melhor.
- **Depende do hardware**
 - GPU vs. CPU, desktop vc. smartphone.

Análise empírica

Desvantagens:

- Em certos contextos, não é uma boa opção. **Por quê?**
- **Depende do compilador**
 - Pode preferir algumas construções ou otimizar melhor.
- **Depende do hardware**
 - GPU vs. CPU, desktop vc. smartphone.
- **Depende da linguagem de programação e habilidade do programador**

Complexidade do algoritmo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de n (a quantidade de dados)

Complexidade do algoritmo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de n (a quantidade de dados)

Para evitar análises dependente de tempo, considera-se que um algoritmo é subdividido, ao invés de milissegundos, em uma quantidade finita de **passos**.

Complexidade do algoritmo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de n (a quantidade de dados)

Para evitar análises dependente de tempo, considera-se que um algoritmo é subdividido, ao invés de milissegundos, em uma quantidade finita de **passos**.

Um **passo** é uma instrução indivisível e de tempo constante, ou seja, independente de condições de entrada e processamento.

- **Exemplo:** soma, multiplicação, atribuição, comparação.

Complexidade do algoritmo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de n (a quantidade de dados)

Para evitar análises dependente de tempo, considera-se que um algoritmo é subdividido, ao invés de milissegundos, em uma quantidade finita de **passos**.

Um **passo** é uma instrução indivisível e de tempo constante, ou seja, independente de condições de entrada e processamento.

- **Exemplo:** soma, multiplicação, atribuição, comparação.
- A quantidade de passos necessários ao cumprimento de um algoritmo é denominada **complexidade de tempo do algoritmo**.

Medindo a complexidade de algoritmos



Busca sequencial

O que faz o algoritmo abaixo?

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)
 - No pior caso, essa linha é executada n vezes

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)
 - No pior caso, essa linha é executada n vezes
- Linha 5: tempo c_5 (acesso e return)

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)
 - No pior caso, essa linha é executada n vezes
- Linha 5: tempo c_5 (acesso e return)
- Linha 6: tempo c_6 (return)

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)
 - No pior caso, essa linha é executada n vezes
- Linha 5: tempo c_5 (acesso e return)
- Linha 6: tempo c_6 (return)

O tempo de execução é menor ou igual a

Busca sequencial e consumo de tempo

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
 - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos e comparação)
 - No pior caso, essa linha é executada n vezes
- Linha 5: tempo c_5 (acesso e return)
- Linha 6: tempo c_6 (return)

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \end{aligned}$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n \end{aligned}$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n , depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$

Isto é, o crescimento do tempo é linear em n

Busca sequencial e consumo de tempo

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

Busca sequencial e consumo de tempo

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

d não interessa tanto, depende apenas do computador...

Busca sequencial e consumo de tempo

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

d não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

Busca sequencial e consumo de tempo

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

d não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da ordem de n

Busca sequencial e consumo de tempo

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

d não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de n**

- A **ordem de crescimento** do tempo é igual a de $f(n) = n$

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

```
1 void soma(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = A[i][j] + B[i][j];  
5         }  
6     }  
7 }
```

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

```
1 void soma(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = A[i][j] + B[i][j];  
5         }  
6     }  
7 }
```

- Qual o tempo de execução desse algoritmo?

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

```
1 void soma(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = A[i][j] + B[i][j];  
5         }  
6     }  
7 }
```

- Qual o tempo de execução desse algoritmo?
- A operação que predomina no algoritmo é a soma (realizada na linha 4).

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

```
1 void soma(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = A[i][j] + B[i][j];  
5         }  
6     }  
7 }
```

- Qual o tempo de execução desse algoritmo?
- A operação que predomina no algoritmo é a soma (realizada na linha 4).
- Logo, expressamos a complexidade como sendo o total de vezes que a soma acontece.

Soma de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser somadas.

```
1 void soma(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = A[i][j] + B[i][j];  
5         }  
6     }  
7 }
```

- Qual o tempo de execução desse algoritmo?
- A operação que predomina no algoritmo é a soma (realizada na linha 4).
- Logo, expressamos a complexidade como sendo o total de vezes que a soma acontece.
- Associando-se os laços, contabilizam-se um total de $n \cdot n = n^2$ iterações. Logo, a complexidade é dada por $f(n) = n^2$.

Multiplicação de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser multiplicadas entre si.

Multiplicação de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser multiplicadas entre si.

```
1 void mult(int n, int A[n][n], int B[n][n], int C[n][n]) {
2     for(int i = 0; i < n; i++) {
3         for(int j = 0; j < n; j++) {
4             C[i][j] = 0;
5             for(int k = 0; k < n; k++) {
6                 C[i][j] += A[i][k] * B[k][j];
7             }
8         }
9     }
10 }
```

Multiplicação de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser multiplicadas entre si.

```
1 void mult(int n, int A[n][n], int B[n][n], int C[n][n]) {  
2     for(int i = 0; i < n; i++) {  
3         for(int j = 0; j < n; j++) {  
4             C[i][j] = 0;  
5             for(int k = 0; k < n; k++) {  
6                 C[i][j] += A[i][k] * B[k][j];  
7             }  
8         }  
9     }  
10 }
```

- Qual o tempo de execução desse algoritmo?

Multiplicação de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser multiplicadas entre si.

```
1 void mult(int n, int A[n][n], int B[n][n], int C[n][n]) {
2     for(int i = 0; i < n; i++) {
3         for(int j = 0; j < n; j++) {
4             C[i][j] = 0;
5             for(int k = 0; k < n; k++) {
6                 C[i][j] += A[i][k] * B[k][j];
7             }
8         }
9     }
10 }
```

- Qual o tempo de execução desse algoritmo?
- A operação de multiplicação aparece uma vez dentro de um aninhamento de três laços **invariantes**, isto é, todos de n iterações.

Multiplicação de matrizes

Sejam A e B duas matrizes quadradas de ordem n que devem ser multiplicadas entre si.

```
1 void mult(int n, int A[n][n], int B[n][n], int C[n][n]) {
2     for(int i = 0; i < n; i++) {
3         for(int j = 0; j < n; j++) {
4             C[i][j] = 0;
5             for(int k = 0; k < n; k++) {
6                 C[i][j] += A[i][k] * B[k][j];
7             }
8         }
9     }
10 }
```

- Qual o tempo de execução desse algoritmo?
- A operação de multiplicação aparece uma vez dentro de um aninhamento de três laços **invariantes**, isto é, todos de n iterações.
- Assim, o total de multiplicações realizadas, que corresponde a complexidade do algoritmo, vale $f(n) = n \cdot n \cdot n = n^3$.

Complexidade de melhor caso e pior caso

- Diferentemente do algoritmo para soma de matrizes, na maior parte dos algoritmos conhecidos a função de complexidade $f(n)$ não é geral, ou seja, **pode mudar conforme características da entrada**.

Complexidade de melhor caso e pior caso

- Diferentemente do algoritmo para soma de matrizes, na maior parte dos algoritmos conhecidos a função de complexidade $f(n)$ não é geral, ou seja, **pode mudar conforme características da entrada**.
- Exemplo:** busca sequencial.

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Complexidade de melhor caso e pior caso

- Diferentemente do algoritmo para soma de matrizes, na maior parte dos algoritmos conhecidos a função de complexidade $f(n)$ não é geral, ou seja, **pode mudar conforme características da entrada**.
- Exemplo:** busca sequencial.

```
1 int busca(int v[], int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

- Nestes casos, a análise de complexidade consiste em avaliar o algoritmo em situações extremas.

Complexidade de melhor caso e pior caso

Na análise da complexidade, definimos três casos de entrada para um algoritmo, como forma de mensurar o custo do algoritmo de resolver determinado problema diante de diferentes entradas.

- **Melhor caso:** o menor tempo de execução (analítico) considerando qualquer entrada de tamanho n .

Complexidade de melhor caso e pior caso

Na análise da complexidade, definimos três casos de entrada para um algoritmo, como forma de mensurar o custo do algoritmo de resolver determinado problema diante de diferentes entradas.

- **Melhor caso:** o menor tempo de execução (analítico) considerando qualquer entrada de tamanho n .
- **Pior caso:** o maior tempo de execução (analítico) considerando qualquer entrada de tamanho n .

Complexidade de melhor caso e pior caso

Na análise da complexidade, definimos três casos de entrada para um algoritmo, como forma de mensurar o custo do algoritmo de resolver determinado problema diante de diferentes entradas.

- **Melhor caso:** o menor tempo de execução (analítico) considerando qualquer entrada de tamanho n .
- **Pior caso:** o maior tempo de execução (analítico) considerando qualquer entrada de tamanho n .
- **Caso médio:** é a média dos tempos de execução (analítico) considerando qualquer entrada de tamanho n .

Pior caso, caso médio e melhor caso

Em geral, queremos analisar o **pior** caso do algoritmo.

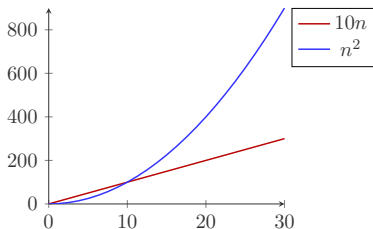
- A análise do **melhor** caso pode ser de interesse, mas é rara.
- A análise do caso **médio** é mais difícil
 - É uma análise probabilística
 - Precisamos fazer suposições sobre os dados de entrada

Crescimento assintótico de funções



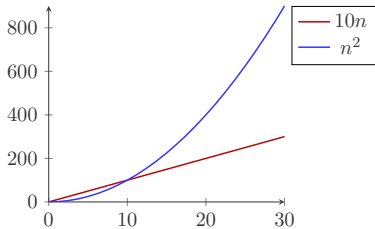
Comportamento assintótico

- **Motivação:** Para valores pequenos de n , praticamente qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
 - **Logo:** a escolha do algoritmo tem pouquíssima influência em problemas de tamanho pequeno.



Comportamento assintótico

- **Motivação:** Para valores pequenos de n , praticamente qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
 - **Logo:** a escolha do algoritmo tem pouquíssima influência em problemas de tamanho pequeno.



- A análise de algoritmos é realizada para valores grandes de n .
 - Estudamos o **comportamento assintótico** das funções de complexidade: comportamento da função para valores grandes de n .

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

Nem todos os termos da função são necessários para termos uma noção do custo do algoritmo.

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

Nem todos os termos da função são necessários para termos uma noção do custo do algoritmo.

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

Nem todos os termos da função são necessários para termos uma noção do custo do algoritmo.

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Comparando funções

Queremos comparar duas funções f e g

- Queremos entender a velocidade de crescimento de f
- Queremos dizer que f cresce mais lentamente ou igual a g

Comparando funções

Queremos comparar duas funções f e g

- Queremos entender a velocidade de crescimento de f
- Queremos dizer que f cresce mais lentamente ou igual a g

f pode ser o tempo de execução do algoritmo e g uma função mais simples

- $f(n) = 3n^2 + 10 \lg n$ e $g(n) = n^2$

Comparando funções

Queremos comparar duas funções f e g

- Queremos entender a velocidade de crescimento de f
- Queremos dizer que f cresce mais lentamente ou igual a g

f pode ser o tempo de execução do algoritmo e g uma função mais simples

- $f(n) = 3n^2 + 10 \lg n$ e $g(n) = n^2$

f e g podem ser os tempos de execução de dois algoritmos

- $f(n) = dn$ e $g(n) = c + c \lg n$

Primeira Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para todo $n \geq 0$

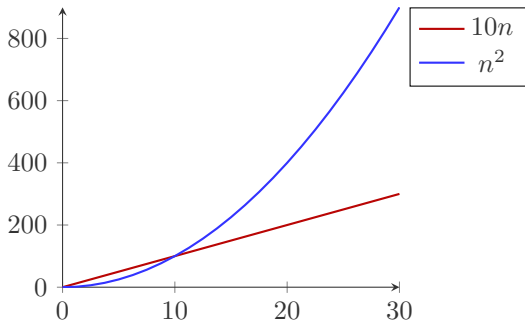
Exemplo: Gostaríamos de provar que $10n < n^2$ para todo n

Primeira Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para todo $n \geq 0$

Exemplo: Gostaríamos de provar que $10n < n^2$ para todo n

Problema: $10n > n^2$ para $n < 10$

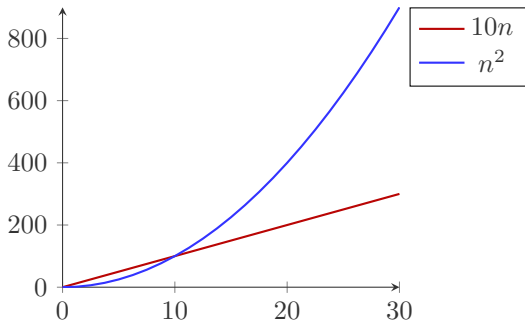


Primeira Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para todo $n \geq 0$

Exemplo: Gostaríamos de provar que $10n < n^2$ para todo n

Problema: $10n > n^2$ para $n < 10$



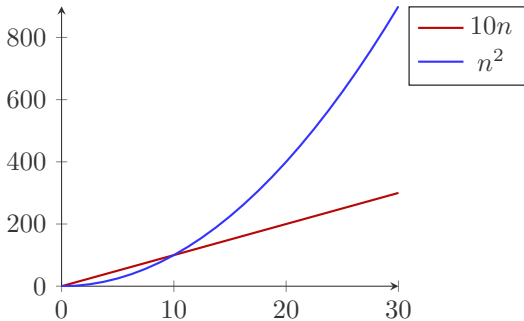
Solução: Ao invés de comparar todo n , comparar apenas n suficientemente grande

Primeira Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para todo $n \geq 0$

Exemplo: Gostaríamos de provar que $10n < n^2$ para todo n

Problema: $10n > n^2$ para $n < 10$



Solução: Ao invés de comparar todo n , comparar apenas n suficientemente grande

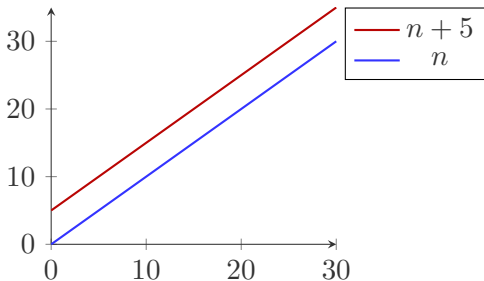
- Para todo $n \geq n_0$ para algum n_0

Segunda Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para $n \geq n_0$

Problema: $n + 5 > n$ para todo n

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos

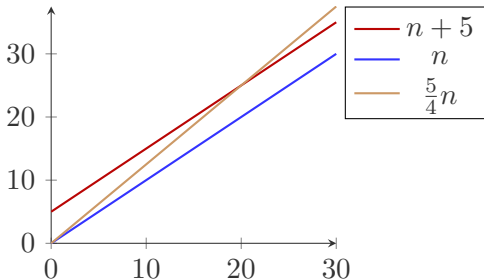


Segunda Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para $n \geq n_0$

Problema: $n + 5 > n$ para todo n

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos

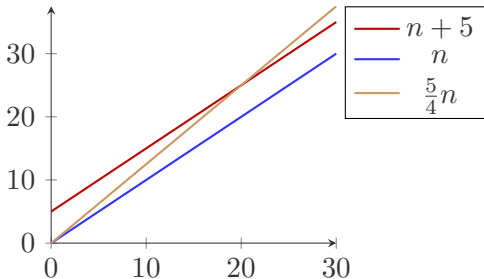


Segunda Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para $n \geq n_0$

Problema: $n + 5 > n$ para todo n

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos



Solução: Ao invés de comparar f com g , comparar com $c \cdot g$, onde c é uma constante

Notação O



Notação Assintótica — Notação O

Dadas funções $f(n)$ e $g(n)$, dizemos que $f(n) = O(g(n))$ se:

- existe constante positiva c e
- existe constante positiva n_0 , tais que

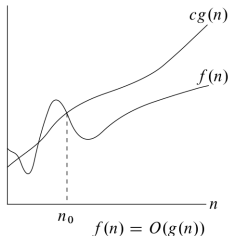
$$f(n) \leq c \cdot g(n), \quad \text{para todo } n \geq n_0.$$

Notação Assintótica — Notação O

Dadas funções $f(n)$ e $g(n)$, dizemos que $f(n) = O(g(n))$ se:

- existe constante positiva c e
- existe constante positiva n_0 , tais que

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

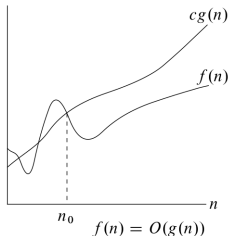


Notação Assintótica — Notação O

Dadas funções $f(n)$ e $g(n)$, dizemos que $f(n) = O(g(n))$ se:

- existe constante positiva c e
- existe constante positiva n_0 , tais que

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

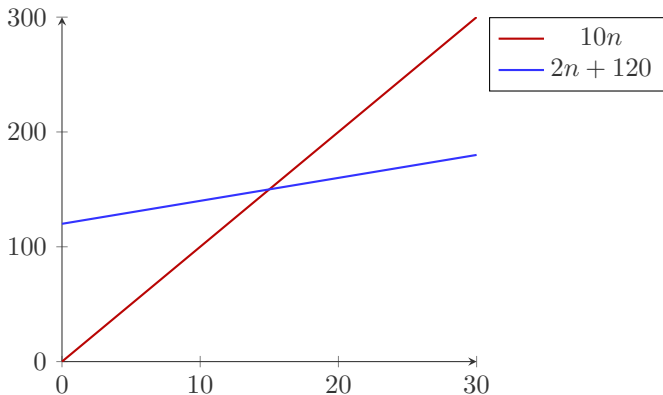


Quando $f(n) = O(g(n))$ dizemos também que a função $g(n)$ domina assintoticamente a função $f(n)$.

Exemplo: $2n + 120 = O(n)$

Exemplo: $2n + 120 = O(n)$

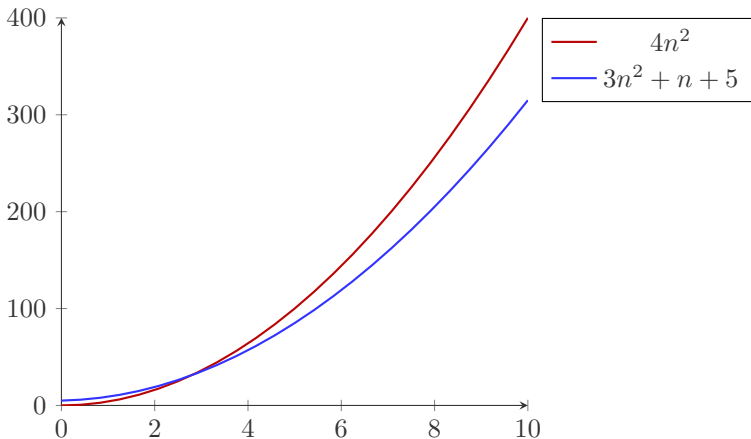
Basta escolher, por exemplo, $c = 10$ e $n_0 = 15$



Exemplo: $3n^2 + n + 5 = O(n^2)$

Exemplo: $3n^2 + n + 5 = O(n^2)$

Basta escolher, por exemplo, $c = 4$ e $n_0 = 4$



Outros exemplos

$$1 = O(1)$$

Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

$$\log_2 n = O(\log_{10} n)$$

Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

$$\log_2 n = O(\log_{10} n)$$

$$\log_{10} n = O(\log_2 n)$$

Nomenclatura e consumo de tempo

- $O(1)$: tempo constante
 - não depende de n
 - Ex: atribuição e leitura de uma variável
 - Ex: operações aritméticas: +, -, *, /
 - Ex: comparações (<, <=, ==, >=, >, !=)
 - Ex: operadores booleanos (&&, &, ||, |, !)
 - Ex: acesso a uma posição de um vetor

Nomenclatura e consumo de tempo

- $O(1)$: tempo constante
 - não depende de n
 - Ex: atribuição e leitura de uma variável
 - Ex: operações aritméticas: $+$, $-$, $*$, $/$
 - Ex: comparações ($<$, \leq , $=$, \geq , $>$, \neq)
 - Ex: operadores booleanos ($\&\&$, $\&$, $||$, $|$, $!$)
 - Ex: acesso a uma posição de um vetor
- $O(\lg n)$: logarítmico
 - \lg indica \log_2
 - quando n dobra, o tempo aumenta em uma constante
 - Ex: Busca binária
 - Outros exemplos durante o curso

Nomenclatura e consumo de tempo

- $O(n)$: linear
 - quando n dobra, o tempo dobra
 - Ex: Busca linear
 - Ex: Encontrar o máximo/mínimo de um vetor
 - Ex: Produto interno de dois vetores

Nomenclatura e consumo de tempo

- $O(n)$: linear
 - quando n dobra, o tempo dobra
 - Ex: Busca linear
 - Ex: Encontrar o máximo/mínimo de um vetor
 - Ex: Produto interno de dois vetores
- $O(n \lg n)$:
 - quando n dobra, o tempo um pouco mais que dobra
 - Ex: algoritmos de ordenação que veremos

Nomenclatura e consumo de tempo

- $O(n)$: linear
 - quando n dobra, o tempo dobra
 - Ex: Busca linear
 - Ex: Encontrar o máximo/mínimo de um vetor
 - Ex: Produto interno de dois vetores
- $O(n \lg n)$:
 - quando n dobra, o tempo um pouco mais que dobra
 - Ex: algoritmos de ordenação que veremos
- $O(n^2)$: quadrático
 - quando n dobra, o tempo quadriplica
 - Ex: BubbleSort, SelectionSort e InsertionSort

Nomenclatura e consumo de tempo

- $O(n)$: linear
 - quando n dobra, o tempo dobra
 - Ex: Busca linear
 - Ex: Encontrar o máximo/mínimo de um vetor
 - Ex: Produto interno de dois vetores
- $O(n \lg n)$:
 - quando n dobra, o tempo um pouco mais que dobra
 - Ex: algoritmos de ordenação que veremos
- $O(n^2)$: quadrático
 - quando n dobra, o tempo quadriplica
 - Ex: BubbleSort, SelectionSort e InsertionSort
- $O(n^3)$: cúbico
 - quando n dobra, o tempo octuplica
 - Ex: multiplicação de matrizes $n \times n$

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(2^n)$ é um milhão.

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(2^n)$ é um milhão.
- $f(n) = O(n!)$: complexidade exponencial

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(2^n)$ é um milhão.
- $f(n) = O(n!)$: complexidade exponencial
 - Pior que $O(c^n)$

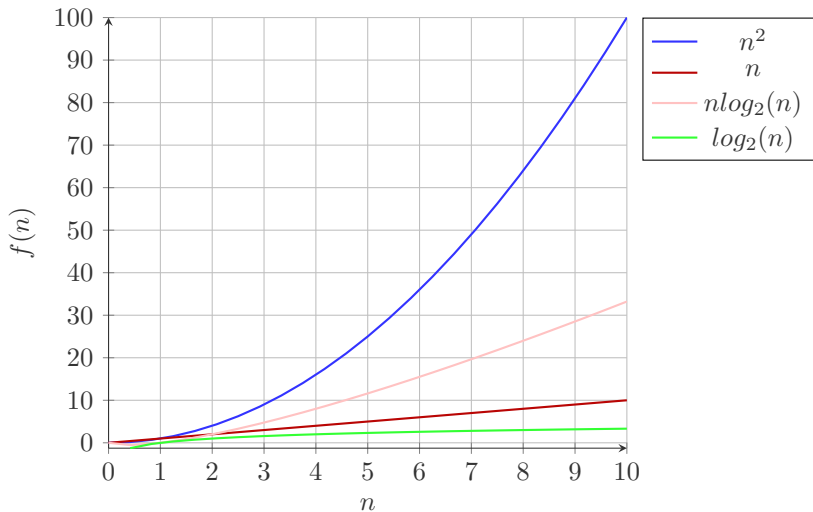
Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(2^n)$ é um milhão.
- $f(n) = O(n!)$: complexidade exponencial
 - Pior que $O(c^n)$
 - Não são úteis do pronto de vista prático.

Nomenclatura e consumo de tempo

- $f(n) = O(c^n)$: complexidade exponencial
 - Típico de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema.
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(2^n)$ é um milhão.
- $f(n) = O(n!)$: complexidade exponencial
 - Pior que $O(c^n)$
 - Não são úteis do ponto de vista prático.
 - Quando n é 20, $O(n!)$ é maior que 2 quintilhões.

Comparando quatro funções



Comparação de funções de complexidade

Tamanho n	Função de custo					
	$\lg_2 n$	n	$n \lg_2 n$	n^2	n^3	2^n
10	3	10	30	100	1000	1000
100	6	100	664	10^4	10^6	10^{30}
1000	9	1000	9965	10^6	10^9	10^{300}
10^4	13	10^4	10^5	10^8	10^{12}	10^{3000}
10^5	16	10^5	10^6	10^{10}	10^{15}	10^{30000}
10^6	19	10^6	10^7	10^{12}	10^{18}	10^{300000}

2 semanas $\approx 1,21 \cdot 10^6$ segundos

1 ano $\approx 3 \cdot 10^7$ segundos

1 século $\approx 3 \cdot 10^9$ segundos

1 milênio $\approx 3 \cdot 10^{10}$ segundos

Um cuidado

O que significa dizer que o tempo de um algoritmo é $O(n^3)$?

- Para instâncias grandes ($n \geq n_0$)
- O tempo é **menor ou igual** a um múltiplo de n^3

Um cuidado

O que significa dizer que o tempo de um algoritmo é $O(n^3)$?

- Para instâncias grandes ($n \geq n_0$)
- O tempo é **menor ou igual** a um múltiplo de n^3

Pode ser que o tempo do algoritmo seja $2n^2$...

- $2n^2 = O(n^3)$, mas...
- $2n^2 = O(n^2)$

Um cuidado

O que significa dizer que o tempo de um algoritmo é $O(n^3)$?

- Para instâncias grandes ($n \geq n_0$)
- O tempo é **menor ou igual** a um múltiplo de n^3

Pode ser que o tempo do algoritmo seja $2n^2$...

- $2n^2 = O(n^3)$, mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

- achamos que o algoritmo é muito pior do que é realmente

Notação Ω



Notação Assintótica – Notação Ω

Dada uma função $f(n)$, dizemos $f(n) = \Omega(g(n))$ se

- existem constantes positivas c e n_0 , tais que:

Notação Assintótica – Notação Ω

Dada uma função $f(n)$, dizemos $f(n) = \Omega(g(n))$ se

- existem constantes positivas c e n_0 , tais que:

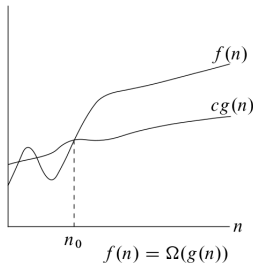
$$0 \leq c \cdot g(n) \leq f(n), \quad \text{para todo } n \geq n_0.$$

Notação Assintótica – Notação Ω

Dada uma função $f(n)$, dizemos $f(n) = \Omega(g(n))$ se

- existem constantes positivas c e n_0 , tais que:

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

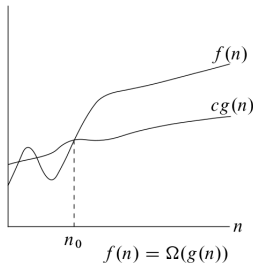


Notação Assintótica – Notação Ω

Dada uma função $f(n)$, dizemos $f(n) = \Omega(g(n))$ se

- existem constantes positivas c e n_0 , tais que:

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

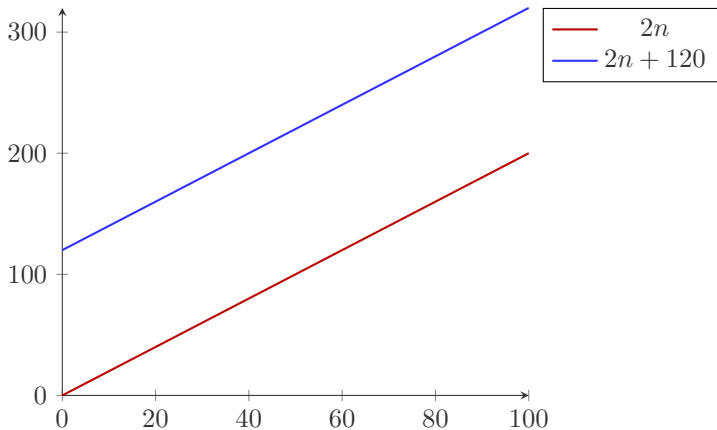


$f(n) = \Omega(g(n))$ se, para todo n suficientemente grande, $f(n)$ é maior ou igual a um múltiplo de $g(n)$

Exemplo: $2n + 120 = \Omega(n)$

Exemplo: $2n + 120 = \Omega(n)$

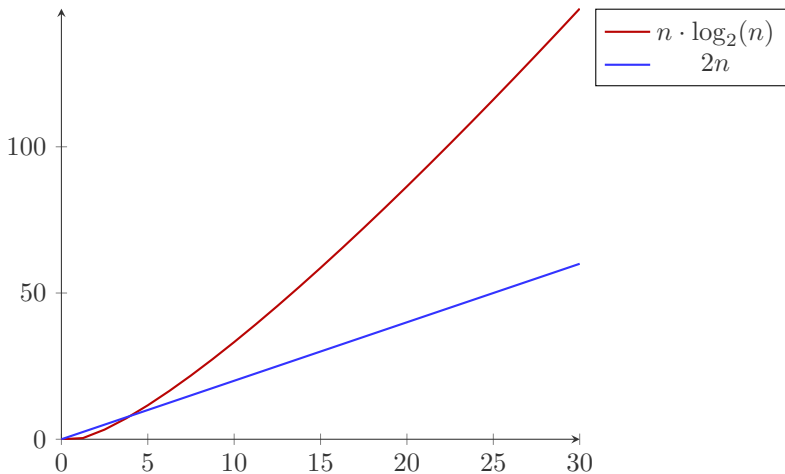
Basta escolher, por exemplo, $c = 2$ e $n_0 = 0$



Exemplo: $n \log_2(n) = \Omega(n)$

Exemplo: $n \log_2(n) = \Omega(n)$

Basta escolher, por exemplo, $c = 2$ e $n_0 = 4$



Conclusão

- A análise de algoritmos é útil para definir o algoritmo mais eficiente em determinados problemas.
- O objetivo final não é apenas fazer códigos que funcionem, mas que sejam também eficientes.

“Um bom algoritmo, mesmo rodando em uma máquina lenta, sempre acaba derrotando (para instâncias grandes do problema) um algoritmo pior rodando em uma máquina rápida. Sempre.”

— S. S. Skiena, The Algorithm Design Manual

Exercícios



Exercício

Para cada uma das afirmações abaixo, justifique formalmente (usando definições, manipulações algébricas e implicações) se for verdade ou dê um contraexemplo se for falso.

(a) $3n = O(n)$

(b) $2n^2 - n = O(n^2)$

(c) $\log 8n = O(\log 2n)$

(d) $2^{n+1} = O(2^n)$

(e) $2^n = O(2^{n/2})$

(f) $n^2 - 200n - 300 = O(n)$

(g) Se $f(n) = 17$, então $f(n) = O(1)$

(h) Se $f(n) = 3n^2 - n + 4$, então $f(n) = O(n^2)$

Exercício

Determine a complexidade de pior caso da função a seguir:

Algoritmo 3 Função F

```
1: Função F(int L[ ], int n)
2:    $s \leftarrow 0$ 
3:   para  $i \leftarrow 0$  até  $n - 2$  faça
4:     para  $j \leftarrow i + 1$  até  $n - 1$  faça
5:       if  $L[i] > L[j]$  then
6:          $s \leftarrow s + 1$ 
7:       fim if
8:     fim para
9:   fim para
10:  retorne  $s$ 
11: fim Função
```

Exercícios Resolvidos



Exercício

Exercício: Proponha um **limite superior** para a função $f(n) = 3n^2 + 18$ juntamente com constantes c e n_0 válidas.

Exercício

Exercício: Proponha um **limite superior** para a função $f(n) = 3n^2 + 18$ juntamente com constantes c e n_0 válidas.

Solução: Como limite superior, propomos a função $g(n) = n^2$ e como constantes válidas citamos $c = 4$ e $n_0 = 5$.

Exercício

Exercício: Proponha um **limite superior** para a função $f(n) = 3n^2 + 18$ juntamente com constantes c e n_0 válidas.

Solução: Como limite superior, propomos a função $g(n) = n^2$ e como constantes válidas citamos $c = 4$ e $n_0 = 5$.

Vamos verificar essas constantes:

Exercício

Exercício: Proponha um **limite superior** para a função $f(n) = 3n^2 + 18$ juntamente com constantes c e n_0 válidas.

Solução: Como limite superior, propomos a função $g(n) = n^2$ e como constantes válidas citamos $c = 4$ e $n_0 = 5$.

Vamos verificar essas constantes:

$$c \cdot g(n) \geq f(n)$$

$$4n^2 \geq 3n^2 + 18$$

$$n^2 \geq 18 \Rightarrow \{n \leq -\sqrt{18} \cup n \geq \sqrt{18}\}$$

Exercício

Exercício: Proponha um **limite superior** para a função $f(n) = 3n^2 + 18$ juntamente com constantes c e n_0 válidas.

Solução: Como limite superior, propomos a função $g(n) = n^2$ e como constantes válidas citamos $c = 4$ e $n_0 = 5$.

Vamos verificar essas constantes:

$$c \cdot g(n) \geq f(n)$$

$$4n^2 \geq 3n^2 + 18$$

$$n^2 \geq 18 \Rightarrow \{n \leq -\sqrt{18} \cup n \geq \sqrt{18}\}$$

Como $c = 4$ e $n = 5 > 4.25 \approx \sqrt{18}$, então $3n^2 + 18 = O(n^2)$.

Exercício

Exercício: Suponha $f(n) = 2n^2 + 30n + 400$ e $g(n) = n^2$. Mostre que $f = O(g)$.

Exercício

Exercício: Suponha $f(n) = 2n^2 + 30n + 400$ e $g(n) = n^2$. Mostre que $f = O(g)$.

Solução: Para todo n positivo, temos:

$$\begin{aligned} f(n) &= 2n^2 + 30n + 400 \\ &\leq 2n^2 + 30n^2 + 400n^2 \\ &= 432n^2 \\ &= 432g(n). \end{aligned}$$

Resumindo, $f(n) \leq 432g(n)$ para todo $n \leq 1$. Além disso, note que $f(n)$ e $g(n)$ são assintoticamente não-negativas. Portanto, $f(n) = O(g(n))$.

Exercício

Exercício: Suponha $f(n) = \lceil n/2 \rceil + 10$ e $g(n) = n$. Mostre que $f(n) = O(g(n))$.

Exercício

Exercício: Suponha $f(n) = \lceil n/2 \rceil + 10$ e $g(n) = n$. Mostre que $f(n) = O(g(n))$.

Solução: De fato, temos que:

$$\begin{aligned} f(n) &= \lceil n/2 \rceil + 10 \\ &\leq n/2 + 1 + 10 \\ &= n/2 + 11 \\ &\leq 20n \text{ para todo } n \geq 1. \end{aligned}$$

Portanto, $f(n) = O(g(n))$.

Exercício

Exercício: Suponha $f(n) = 5n \lg n + 8 \lg^2 n - 11$ e $g(n) = n \lg n$.
Mostre que $f(n) = O(g(n))$.

Exercício

Exercício: Suponha $f(n) = 5n \lg n + 8 \lg^2 n - 11$ e $g(n) = n \lg n$.
Mostre que $f(n) = O(g(n))$.

Solução: Desta vez, vamos usar limites:

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{5n \lg n + 8 \lg^2 n - 11}{n \lg n} \\ &= \lim_{n \rightarrow \infty} 5 + 8 \frac{\lg n}{n} - 11 \frac{1}{n \lg n} \\ &= 5 + 8(0) - 11(0) \\ &= 5. \end{aligned}$$

Logo, como o limite existe, então $f(n) = O(g(n))$.

Exercício

Exercício: Suponha $f(n) = 5n \lg n + 8 \lg^2 n - 11$ e $g(n) = n \lg n$.
Mostre que $f(n) = O(g(n))$, sem usar limites.

Exercício

Exercício: Suponha $f(n) = 5n \lg n + 8 \lg^2 n - 11$ e $g(n) = n \lg n$.
Mostre que $f(n) = O(g(n))$, sem usar limites.

Solução:

$$\begin{aligned} 5n \lg n + 8 \lg^2 n - 11 &\leq 5n \lg n + 8 \lg^2 n \\ &\leq 5n \lg n + 8n \lg n \text{ pois } \lg n < n \quad \forall n \geq 1 \\ &= 13n \lg n \end{aligned}$$

Logo, concluímos que $5n \lg n + 8 \lg^2 n - 11 \leq 13n \lg n$ para todo $n \geq 1$. Portanto, fazendo $n_0 = 1$ e $c = 13$, temos que $0 \leq f(n) \leq 13g(n)$ para todo $n \geq n_0$. Assim, $f(n) = O(g(n))$.

Exercícios

1. É verdade que $\log_2 n = O(\log_3 n)$? É verdade que $\log_3 n = O(\log_2 n)$?

Exercícios

1. É verdade que $\log_2 n = O(\log_3 n)$? É verdade que $\log_3 n = O(\log_2 n)$?
2. Mostre que $15n = O(n \lg n)$ mas que $n \lg n \neq O(n)$

Exercícios

1. É verdade que $\log_2 n = O(\log_3 n)$? É verdade que $\log_3 n = O(\log_2 n)$?
2. Mostre que $15n = O(n \lg n)$ mas que $n \lg n \neq O(n)$
 - Essa análise é folgada, já que $15n = O(n)$

Exercícios

1. É verdade que $\log_2 n = O(\log_3 n)$? É verdade que $\log_3 n = O(\log_2 n)$?
2. Mostre que $15n = O(n \lg n)$ mas que $n \lg n \neq O(n)$
 - Essa análise é folgada, já que $15n = O(n)$
3. Mostre que $42n = O(n^2)$ mas que $n^2 \neq O(42n)$

Exercícios

1. É verdade que $\log_2 n = O(\log_3 n)$? É verdade que $\log_3 n = O(\log_2 n)$?
2. Mostre que $15n = O(n \lg n)$ mas que $n \lg n \neq O(n)$
 - Essa análise é folgada, já que $15n = O(n)$
3. Mostre que $42n = O(n^2)$ mas que $n^2 \neq O(42n)$
 - Essa análise é folgada, já que $42n = O(n)$

FIM

