

# BCC202 - Estrutura de Dados I

## Aula 04: Análise de Algoritmos (Parte 1)

**Reinaldo Fortes**

Universidade Federal de Ouro Preto, UFOP  
Departamento de Computação, DECOM

Website: [www.decom.ufop.br/reifortes](http://www.decom.ufop.br/reifortes)  
Email: [reifortes@iceb.ufop.br](mailto:reifortes@iceb.ufop.br)

Material elaborado com base nos slides do [Prof. Túlio Toffolo](#) (curso de 2013/01).

2014/01



# Conteúdo

## 1 Introdução

## 2 Análise de Algoritmos

- Análise de Algoritmos
- Custo de um algoritmo
- Função de complexidade
- Tamanho da entrada de dados
- Melhor Caso, Pior Caso e Caso Médio

## 3 Conclusão

## 4 Exercícios

# Conteúdo

## 1 Introdução

## 2 Análise de Algoritmos

- Análise de Algoritmos
- Custo de um algoritmo
- Função de complexidade
- Tamanho da entrada de dados
- Melhor Caso, Pior Caso e Caso Médio

## 3 Conclusão

## 4 Exercícios

# Análise de Algoritmos

- **Analisar** um algoritmo consiste em “**verificar**” o **custo** do algoritmo em relação ao:
  - **Tempo** gasto para executá-lo.
  - **Espaço** (**memória**) ocupado em sua execução.
- Esta **análise** é necessária para que se possa **escolher** o algoritmo mais **adequado** para resolver um dado problema.
- É essencialmente importante em áreas de pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

## Cálculo do custo **real** pela execução do algoritmo

- Medidas são **inadequadas** e o resultado não pode ser generalizado.
- Tais medidas são dependentes do compilador, que pode favorecer algumas construções em detrimento de outras.
- Resultados dependem do *hardware*.
- Quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.

## Cálculo do custo **real** pela execução do algoritmo

- Apesar disto, há argumentos a favor de se obter medidas reais da execução do algoritmo.
- **Um exemplo:** quando há vários algoritmos distintos para resolver um mesmo problema, todos possuindo um custo de execução dentro de uma mesma **ordem de grandeza**.

# Conteúdo

## 1 Introdução

## 2 Análise de Algoritmos

- Análise de Algoritmos
- Custo de um algoritmo
- Função de complexidade
- Tamanho da entrada de dados
- Melhor Caso, Pior Caso e Caso Médio

## 3 Conclusão

## 4 Exercícios

\_\_\_\_\_

- Um algoritmo em particular.
- Uma classe de algoritmos.



# Análise de um algoritmo em particular

- Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- Características que devem ser investigadas:
  - Análise do número de vezes que cada parte do algoritmo deve ser executada (**tempo**).
  - Estudo da quantidade de memória necessária (**espaço**).

# Análise de uma **classe de algoritmos**

- Qual é o algoritmo de menor custo possível para resolver um problema em particular?
- Toda uma **família** de algoritmos é investigada.
- Procura-se identificar um que seja o melhor possível.
- Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

## Custo de um algoritmo

- O menor custo possível para resolver problemas de uma classe nos dá a dificuldade inerente para resolver o problema.
- **Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.**
- Podem existir vários algoritmos ótimos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

## Custo de um algoritmo

- Utilizaremos um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas operações e considerar apenas as mais significativas.
- Ex.: Em **algoritmos de ordenação** consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as demais operações.

# Função de complexidade

- Para medir o custo de execução de um algoritmo definiremos uma função de complexidade ou função de custo  $f$ .
- Função de **complexidade de tempo**:  $f(n)$  mede o tempo necessário para executar um algoritmo em um problema de tamanho  $n$ .
- Função de **complexidade de espaço**:  $f(n)$  mede a memória necessária para executar um algoritmo em um problema de tamanho  $n$ .

# Função de complexidade

- Utilizaremos  $f$  para denotar uma função de complexidade de **tempo** daqui para frente.
- A complexidade de tempo na realidade **não representa tempo diretamente**:
  - Representa o número de vezes que determinadas operações, ditas **relevantes**, são executadas.

## Exemplo: Maior elemento de um vetor

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros  $A[n]$ ;  $n \geq 1$ .

```
1 | int Max(int* A, int n) {  
2 |     int i, Temp;  
3 |     Temp = A[0];  
4 |     for(i = 1; i < n; i++)  
5 |         if(Temp < A[i]) // Comparação envolvendo os elementos  
6 |             Temp = A[i];  
7 |     return Temp;  
8 | }
```

- Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de comparações envolvendo os elementos de  $A$ , se  $A$  contiver  $n$  elementos.
- Qual é a função  $f(n)$ ?

## Exemplo: Maior elemento de um vetor

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros  $A[n]$ ;  $n \geq 1$ .

```
1 | int Max(int* A, int n) {  
2 |     int i, Temp;  
3 |     Temp = A[0];  
4 |     for(i = 1; i < n; i++)  
5 |         if(Temp < A[i]) // Comparação envolvendo os elementos  
6 |             Temp = A[i];  
7 |     return Temp;  
8 | }
```

- Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de comparações envolvendo os elementos de  $A$ , se  $A$  contiver  $n$  elementos.
- Qual é a função  $f(n)$ ?  $f(n) = n - 1$ .



## Exemplo: Maior elemento de um vetor

- **Teorema:** Qualquer algoritmo para encontrar o maior elemento de um conjunto com  $n$  elementos,  $n \geq 1$ , faz pelo menos  $n - 1$  comparações.
- **Prova:** Cada um dos  $n - 1$  elementos tem de ser investigado por meio de comparações, que é menor do que algum outro elemento.
  - Logo,  $n - 1$  comparações são necessárias.

O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função **Max** do programa anterior é ótima.

## Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente** do **tamanho da entrada** dos dados.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - Ou seja, o custo pode ser diferente para entradas distintas, mas de mesmo tamanho.

## Tamanho da entrada de dados

- No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho  $n$ .
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então pode ser que o algoritmo trabalhe menos.

## Definição

- **Melhor caso:** menor tempo de execução sobre todas as entradas de tamanho  $n$ .
- **Pior caso:** maior tempo de execução sobre todas as entradas de tamanho  $n$ .
- **Caso médio** (ou *caso esperado*): média dos tempos de execução de todas as entradas de tamanho  $n$ .

**Melhor caso  $\leq$  Caso médio  $\leq$  Pior caso**

## Análise do caso médio

- A análise do **caso médio** é geralmente muito mais difícil de obter do que as análises do melhor e do pior casos.
- Supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho  $n$ .
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são **igualmente prováveis**.
  - Na prática isso nem sempre é verdade.

## Exemplo: Registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperá-lo.
- **O problema:** dada uma chave qualquer, localize o registro que contenha esta chave.
- **O algoritmo de pesquisa** mais simples é o que faz a **pesquisa sequencial**.

## Exemplo: Registros de um arquivo (**Análise**)

- Seja  $f(n)$  uma função de complexidade, onde  $n$  corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$ .
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$ .
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**

## Exemplo: Registros de um arquivo (**Análise**)

- Seja  $f(n)$  uma função de complexidade, onde  $n$  corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$ .
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$ .
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**



## Exemplo: Registros de um arquivo (**Análise**)

- Seja  $f(n)$  uma função de complexidade, onde  $n$  corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$ .
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$ .
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**

## Exemplo: Registros de um arquivo (**Análise**)

- Seja  $f(n)$  uma função de complexidade, onde  $n$  corresponde ao número de registros.
- A complexidade será definida pelo número de registros consultados (número de comparações de chaves):
  - **Melhor caso:** O registro procurado é o **primeiro** consultado!!!
    - $f(n) = 1$ .
  - **Pior caso:** O registro procurado é o **último** consultado!!!
    - $f(n) = n$ .
  - **Caso médio:**
    - Nem sempre é tão simples de se calcular.
    - **Como fazer para este problema?**

## Exemplo: Registros de um arquivo (Análise do Caso médio)

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se  $p_i$  for a probabilidade de que o  $i$ -ésimo registro seja procurado, e considerando que para recupera-lo são necessárias  $i$  comparações, então:

$$f(n) = 1 * p_1 + 2 * p_2 + 3 * p_3 + ... + n * p_n$$

## Exemplo: Registros de um arquivo (Análise do Caso médio)

- Para calcular  $f(n)$  basta conhecer a distribuição de probabilidades  $p_i$ .
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então:

$$p_i = \frac{1}{n}, \text{ para } 1 \leq i \leq n$$

- Logo:

$$f(n) = \frac{1}{n} (1 + 2 + 3 + \dots + n) = \frac{1}{n} \left( \frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

## Exemplo: Registros de um arquivo (**Análise**)

- **Melhor caso:**

- O registro procurado é o **primeiro** consultado!!!
- $f(n) = 1$ .

- **Pior caso:**

- O registro procurado é o **último** consultado!!!
- $f(n) = n$ .

- **Caso médio:**

- $f(n) = (n + 1)/2$ .
- *Considerando a mesma probabilidade de acesso para todos os registros.*

## Exemplo: Maior e menor elementos de um vetor

- Considere o problema de encontrar o maior e o menor elementos de um vetor de inteiros  $A[n]$ , onde  $n \geq 1$ .
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o **maior** elemento:

```
1 int Max(int* A, int n) {  
2     int i, Temp;  
3     Temp = A[0];  
4     for(i = 1; i < n; i++)  
5         if(Temp < A[i]) // Comparação envolvendo os elem.  
6             Temp = A[i];  
7     return Temp;  
8 }
```

- Recordando que, para Max:  $f(n) = n - 1$ .

## Exemplo: Maior e menor elementos de um vetor (MaxMin1)

```
1 void MaxMin1(int* A, int n, int* pMax, int* pMin) {  
2     int i;  
3     *pMax = A[0];  
4     *pMin = A[0];  
5     for(i = 1; i < n; i++)  
6         if(*pMax < A[i]) // Comparação envolvendo os elementos  
7             *pMax = A[i];  
8         if(*pMin > A[i]) // Comparação envolvendo os elementos  
9             *pMin = A[i];  
10 }
```

- Seja  $f(n)$  o número de comparações entre os elementos de  $A$ , se  $A$  contiver  $n$  elementos.
- Então,  $f(n) = 2(n - 1)$  para  $n > 0$ , para o melhor caso, pior caso e caso médio.

## Exemplo: Maior e menor elementos de um vetor (**MaxMin1**)

```
1 void MaxMin1(int* A, int n, int* pMax, int* pMin) {  
2     int i;  
3     *pMax = A[0];  
4     *pMin = A[0];  
5     for(i = 1; i < n; i++)  
6         if(*pMax < A[i]) // Comparação envolvendo os elementos  
7             *pMax = A[i];  
8         if(*pMin > A[i]) // Comparação envolvendo os elementos  
9             *pMin = A[i];  
10 }
```

- Seja  $f(n)$  o número de comparações entre os elementos de  $\mathbf{A}$ , se  $\mathbf{A}$  contiver  $n$  elementos.
- Então,  $f(n) = 2(n - 1)$  para  $n > 0$ , para o melhor caso, pior caso e caso médio.



## Exemplo: Maior e menor elementos de um vetor (**MaxMin1**)

```
1 void MaxMin1(int* A, int n, int* pMax, int* pMin) {  
2     int i;  
3     *pMax = A[0];  
4     *pMin = A[0];  
5     for(i = 1; i < n; i++)  
6         if(*pMax < A[i]) // Comparação envolvendo os elementos  
7             *pMax = A[i];  
8         if(*pMin > A[i]) // Comparação envolvendo os elementos  
9             *pMin = A[i];  
10 }
```

- Seja  $f(n)$  o número de comparações entre os elementos de  $\mathbf{A}$ , se  $\mathbf{A}$  contiver  $n$  elementos.
- Então,  $f(n) = 2(n - 1)$  para  $n > 0$ , para o melhor caso, pior caso e caso médio.

## Exemplo: Maior e menor elementos de um vetor (**MaxMin2**)

- MaxMin1 pode ser facilmente melhorado:
  - A comparação  $*pMin > A[i]$  só é necessária quando a comparação  $*pMax < A[i]$  é falsa.

```
1 void MaxMin2(int* A, int n, int* pMax, int* pMin) {  
2     int i;  
3     *pMax = A[0];  
4     *pMin = A[0];  
5     for(i = 1; i < n; i++)  
6         if(*pMax < A[i])  
7             *pMax = A[i];  
8         else if(*pMin > A[i]) // A diferença está aqui!  
9             *pMin = A[i];  
10 }
```

- E agora, qual é a função de complexidade?

## Exemplo: Maior e menor elementos de um vetor (**MaxMin2**)

- MaxMin1 pode ser facilmente melhorado:
  - A comparação  $*pMin > A[i]$  só é necessária quando a comparação  $*pMax < A[i]$  é falsa.

```
1 void MaxMin2(int* A, int n, int* pMax, int* pMin) {  
2     int i;  
3     *pMax = A[0];  
4     *pMin = A[0];  
5     for(i = 1; i < n; i++)  
6         if(*pMax < A[i])  
7             *pMax = A[i];  
8         else if(*pMin > A[i]) // A diferença está aqui!  
9             *pMin = A[i];  
10 }
```

- E agora, qual é a função de complexidade?

## Exemplo: Maior e menor elementos de um vetor (**MaxMin2**)

- **Melhor caso:**

- Quando os elementos estão em ordem crescente.
- $f(n) = n - 1$ .

- **Pior caso:**

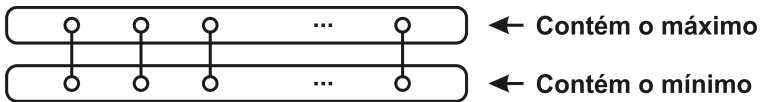
- Quando o maior elemento é o primeiro no vetor.
- $f(n) = 2(n - 1)$ .

- **Caso médio:**

- Quando  $*pMax < A[i]$  em metade das vezes.
- $f(n) = 3n/2 - 3/2$ .

## Exemplo: Maior e menor elementos de um vetor (MaxMin3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  - Compare os elementos de A aos pares, separando-os em **dois subconjuntos** (**maiores** em um e **menores** em outro), a um custo de  $\lceil n/2 \rceil$  comparações.
  - O **máximo** é obtido do subconjunto que contém os maiores elementos, a um custo de  $\lceil n/2 \rceil - 1$  comparações.
  - O **mínimo** é obtido do subconjunto que contém os menores elementos, a um custo de  $\lceil n/2 \rceil - 1$  comparações.



## Exemplo: Maior e menor elementos de um vetor (**MaxMin3**)

- Os elementos de  $A$  são comparados dois a dois.
  - Os elementos maiores são comparados com **\*pMax**.
  - Os elementos menores são comparados com **\*pMin**.
- Quando  $n$  é ímpar, o elemento que está na posição  $A[n - 1]$  é duplicado na posição  $A[n]$  para evitar um tratamento de exceção.
- Para esta implementação:
  - Melhor caso = Pior caso = Caso médio:**

$$f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$$

## Exemplo: Maior e menor elementos de um vetor (MaxMin3)

```

1 void MaxMin3(int* A, int n, int* pMax, int* pMin) {
2     int i, FimDoAnel;
3     if((n % 2) > 0) { A[n] = A[n-1]; FimDoAnel = n; }
4     else             { FimDoAnel = n-1; }
5     if(A[0] > A[1]) { *pMax = A[0]; *pMin = A[1]; } // Comp. 1
6     else           { *pMax = A[1]; *pMin = A[0]; }
7
8     for(i=2; i<FimDoAnel; i+=2) {
9         if(A[i] > A[i+1]) { // Comp. 2
10             if(A[i] > *pMax) *pMax = A[i]; // Comp. 3
11             if(A[i+1] < *pMin) *pMin = A[i+1]; // Comp. 4
12         } else{
13             if(A[i] < *pMin) *pMin = A[i]; // Comp. 3
14             if(A[i+1] > *pMax) *pMax = A[i+1]; // Comp. 4
15         }
16     }
17 }

```

## Exemplo: Maior e menor elementos de um vetor (MaxMin3)

- Qual a função de complexidade?

- Número de comparações:

- Comp. 1: 1 vez.
- Comp. 2:  $n/2 - 1$  vezes.
- Comp. 3:  $n/2 - 1$  vezes.
- Comp. 4:  $n/2 - 1$  vezes.

- Então:

$$f(n) = 1 + (n/2 - 1) + (n/2 - 1) + (n/2 - 1)$$

$$f(n) = 3(n/2 - 1) + 1$$

$$f(n) = 3n/2 - 2$$



## Exemplo: Maior e menor elementos de um vetor (comparação)

Algoritmo	Melhor caso	Pior caso	Caso médio
MaxMin1	$2 (n-1)$	$2 (n-1)$	$2 (n-1)$
MaxMin2	$n - 1$	$2 (n-1)$	$3 n/2 - 3/2$
MaxMin3	$3 n/2 - 2$	$3 n/2 - 2$	$3 n/2 - 2$

- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e muito próximo quanto ao caso médio.

# Conteúdo

## 1 Introdução

## 2 Análise de Algoritmos

- Análise de Algoritmos
- Custo de um algoritmo
- Função de complexidade
- Tamanho da entrada de dados
- Melhor Caso, Pior Caso e Caso Médio

## 3 Conclusão

## 4 Exercícios

## Conclusão

- Nesta aula tivemos o primeiro contato com a **Análise de Complexidade de Algoritmos**.
- Este tópico é muito importante no estudo de algoritmos, pois possibilita avaliar o **custo** dos mesmos, servindo de referência para a escolha daquele que será mais adequado para uso em determinadas situações.
- Foco principal para a definição da **função de complexidade** e a análise de **melhor caso**, **pior caso** e **caso médio**.
- *Próxima aula*: Análise de Algoritmos (Parte 2) – Comportamento Assintótico e Dominação Assintótica.
- **Dúvidas?**

# Conteúdo

## 1 Introdução

## 2 Análise de Algoritmos

- Análise de Algoritmos
- Custo de um algoritmo
- Função de complexidade
- Tamanho da entrada de dados
- Melhor Caso, Pior Caso e Caso Médio

## 3 Conclusão

## 4 Exercícios

# Exercício 01

- Avalie os dois códigos fonte abaixo e responda:
  - O resultado será o mesmo? Justifique sua resposta.
  - Qual a função de complexidade de cada um dos procedimentos? Defina as operações relevantes.
  - Caso o resultado seja o mesmo, qual dos dois você escolheria?

```
1 void Procedimento1() {  
2     int i = 0, a = 0;  
3     while(i < n) {  
4         a += i;  
5         i += 2;  
6     }  
7 }
```

```
1 void Procedimento2() {  
2     int i, j, a = 0;  
3     for(i = 0; i < n; i++)  
4         for(j = 0; j < i; j++)  
5             a += i + j;  
6 }
```

Dica: Avalie o código e faça testes de mesa, só depois de responder às perguntas, implemente o código e execute os procedimentos para conferir os resultados.