

# Medida do Tempo de Execução de um Programa

David Menotti  
Algoritmos e Estruturas de Dados II  
DInf – UFPR

## Medida do Tempo de Execução de um Programa

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de **tempo** de execução e **espaço** ocupado.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

## Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo particular.**
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?

## Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo 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,
    - estudo da quantidade de memória necessária

## Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo 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,
    - estudo da quantidade de memória necessária.
- **Análise de uma classe de algoritmos.**
  - Qual é o algoritmo de menor custo possível para resolver um problema 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

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da 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 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.

## Medida do Custo pela Execução do Programa

- n Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
  - q os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - q os resultados dependem do *hardware*;
  - q quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.
- n Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - q Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza.
  - q Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

## Medida do Custo por meio de um Modelo Matemático

- n Usa um modelo matemático baseado em um computador idealizado.
- n Deve ser especificado o conjunto de operações e seus custos de execuções.
- n É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- n Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

## Função de Complexidade

- n Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade**  $f$ .
- n  $f(n)$  é a medida do tempo necessário para executar um algoritmo para um problema de tamanho  $n$ .
- n Função de **complexidade de tempo**:  $f(n)$  mede o tempo necessário para executar um algoritmo em um problema de tamanho  $n$ .
- n Função de **complexidade de espaço**:  $f(n)$  mede a memória necessária para executar um algoritmo em um problema de tamanho  $n$ .
- n Utilizaremos  $f$  para denotar uma função de complexidade de tempo daqui para a frente.
- n A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

## Exemplo: maior elemento

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

```

n int Max(int* A, int n) {
n     int i, Temp;
n
n     Temp = A[0];
n     for (i = 1; i < n; i++)
n         if (Temp < A[i])
n             Temp = A[i];
n     return Temp;
n }
  
```

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

- n **Qual a função  $f(n)$ ?**

## Exemplo: maior elemento

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

```

n int Max(int* A, int n) {
n     int i, Temp;
n
n     Temp = A[0];
n     for (i = 1; i < n; i++)
n         if (Temp < A[i])
n             Temp = A[i];
n     return Temp;
n }
  
```

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

- n **Logo  $f(n) = n - 1$**

## Exemplo: maior elemento

- n **Teorema:** Qualquer algoritmo para encontrar o maior elemento de um conjunto com  $n$  elementos,  $n \geq 1$ , faz pelo menos  $n - 1$  comparações.

## Exemplo: maior elemento

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

## Exemplo: maior elemento

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

## Exemplo: maior elemento

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

- n A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- n É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- n Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
- n No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho  $n$ .
- n Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

## Melhor Caso, Pior Caso e Caso Médio

- n **Melhor caso:** menor tempo de execução sobre todas as entradas de tamanho  $n$ .
- n **Pior caso:** maior tempo de execução sobre todas as entradas de tamanho  $n$ .
  - q Se  $f$  é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que  $f(n)$ .
- n **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho  $n$ .

## Análise de Melhor Caso, Pior Caso e Caso Médio

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

## Exemplo - Registros de um Arquivo

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

## Exemplo - Registros de um Arquivo

- n Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - q **melhor caso:**
    - q
  - q
  - q **pioor caso:**
    - q
  - q
  - q **caso médio:**

## Exemplo - Registros de um Arquivo

- n Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - q **melhor caso:**
    - q registro procurado é o primeiro consultado
  - q
  - q **pioor caso:**
    - q
  - q
  - q **caso médio:**

## Exemplo - Registros de um Arquivo

- n Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - q **melhor caso:**
    - q registro procurado é o primeiro consultado
    - q  $f(n) = 1$
  - q **pioor caso:**
    - q
  - q
  - q **caso médio:**

## Exemplo - Registros de um Arquivo

- n Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - q **melhor caso:**
    - q registro procurado é o primeiro consultado
    - q  $f(n) = 1$
  - q **pioor caso:**
    - q registro procurado é o último consultado ou não está presente no arquivo;
  - q
  - q **caso médio:**

## Exemplo - Registros de um Arquivo

- n Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - q **melhor caso:**
    - q registro procurado é o primeiro consultado
    - q  $f(n) = 1$
  - q **pioor caso:**
    - q registro procurado é o último consultado ou não está presente no arquivo;
    - q  $f(n) = n$
  - q **caso médio:**

## Exemplo - Registros de um Arquivo

- 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 recuperar o  $i$ -ésimo registro são necessárias  $i$  comparações, então:

$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \dots + n \times p_n$$

## Exemplo - Registros de um Arquivo

- 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

$$p_i = 1/n, 1 \leq i \leq n$$

## Exemplo - Registros de um Arquivo

- 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

$$p_i = 1/n, 1 \leq i \leq n$$

- Nesse caso:

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

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

## Exemplo - Registros de um Arquivo

- Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).

- melhor caso:

- registro procurado é o primeiro consultado

- $f(n) = 1$

- pior caso:

- registro procurado é o último consultado ou não está presente no arquivo;

- $f(n) = n$

- caso médio:

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

## Exemplo - Maior e Menor Elemento (1)

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

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

## Qual a função de complexidade para MaxMin1?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

## Qual a função de complexidade para MaxMin1?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

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

## Exemplo - Maior e Menor Elemento (2)

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

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

Melhor caso:

Pior caso:

Caso médio:

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

Melhor caso:

- quando os elementos estão em ordem crescente;

Pior caso:

Caso médio:

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

Melhor caso:

- quando os elementos estão em ordem crescente;
- $f(n) = n - 1$

Pior caso:

Caso médio:

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

### Melhor caso:

- quando os elementos estão em ordem crescente;
- $f(n) = n - 1$

### Pior caso:

- quando os elementos estão em ordem decrescente;

### Caso médio:

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

### 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:

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

### 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:

- No caso médio, A[i] é maior do que Max a metade das vezes.

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

### 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:

- No caso médio, A[i] é maior do que Max a metade das vezes.
- $f(n) = n - 1 + (n - 1)/2 = 3n/2 - 3/2$

## Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
    int i;

    Max = A[0];
    Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        else if (A[i] < *pMin) *pMin = A[i];
    }
}
```

### 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:

- No caso médio, A[i] é maior do que Max a metade das vezes.
- $f(n) = 3n/2 - 3/2$

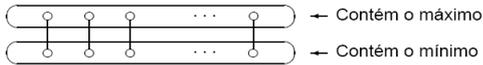
## Exemplo - Maior e Menor Elemento (3)

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  $\lfloor n/2 \rfloor$  comparações.
- O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de  $\lfloor n/2 \rfloor - 1$  comparações
- O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de  $\lfloor n/2 \rfloor - 1$  comparações

## Exemplo - Maior e Menor Elemento (3)

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



## Qual a função de complexidade para este novo algoritmo?

- n Os elementos de A são comparados dois a dois. Os elementos maiores são comparados com \*pMax e os elementos menores são comparados com \*pMin.
- n 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.
- n Para esta implementação:

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

no pior caso, melhor caso e caso médio

## Exemplo - Maior e Menor Elemento (3)

```
void MaxMin3(int* A, int n, int* pMax, int* pMin) {
    int i, FimDoAne1;

    if ((n % 2) > 0)
    { A[n] = A[n - 1]; FimDoAne1 = n; }
    else FimDoAne1 = n - 1;

    if (A[0] > A[1]) { *pMax = A[0]; *pMin = A[1]; } // Comparação 1
    else { *pMax = A[1]; *pMin = A[0]; }

    for (i=3; i<=FimDoAne1; i+=2)
    {
        if (A[i - 1] > A[i]) // Comparação 2
        {
            if (A[i - 1] > *pMax) *pMax = A[i - 1]; // Comparação 3
            if (A[i] < *pMin) *pMin = A[i]; // Comparação 4
        }
        else
        {
            if (A[i - 1] < *pMin) *pMin = A[i - 1]; // Comparação 3
            if (A[i] > *pMax) *pMax = A[i]; // Comparação 4
        }
    }
}
```

## Qual a função de complexidade para MaxMin3?

- n Quantas comparações são feitas em MaxMin3?

## Qual a função de complexidade para MaxMin3?

- n Quantas comparações são feitas em MaxMin3?
  - q 1ª. comparação feita 1 vez
  - q 2ª. comparação feita  $n/2 - 1$  vezes
  - q 3ª. e 4ª. comparações feitas  $n/2 - 1$  vezes

n

## Qual a função de complexidade para MaxMin3?

- n Quantas comparações são feitas em MaxMin3?
  - q 1ª. comparação feita 1 vez
  - q 2ª. comparação feita  $n/2 - 1$  vezes
  - q 3ª. e 4ª. comparações feitas  $n/2 - 1$  vezes

n

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

$$f(n) = (3n - 6)/2 + 1$$

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

## Comparação entre os Algoritmos

- ⁿ A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- ⁿ Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- ⁿ O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n - 1)$	$2(n - 1)$	$2(n - 1)$
MaxMin2	$n - 1$	$2(n - 1)$	$3n/2 - 3/2$
MaxMin3	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

## Exercício - Função de Complexidade

```
void exercicio1 (int
n)
{
  int i, a;
  a=0;i=0;
  while (i<n)
  {
    a+=i;
    i+=2;
  }
}
```

```
void exercicio2 (int n)
{
  int i,j,a;
  a=0;
  for (i=0; i<n; i++)
    for (j=0; j<i; j++)
      a+=i+j;
}
```