

# Estruturas de Dados

## Análise de Complexidade

---

Professores: Anisio Lacerda  
Wagner Meira Jr.

# Projeto de Algoritmos

- Projeto de algoritmos
  - Análise do problema
  - Decisões de projeto
    - Tipos Abstratos de Dados
    - Algoritmo a ser utilizado
- Principais Perguntas:
  - O Algoritmo funciona?
  - **O Algoritmo é eficiente?**

# Projeto de Algoritmos

- A eficiência de um algoritmo pode ser medida com várias métricas. Por exemplo:
  - **tempo** de execução
  - **espaço** ocupado
  - ...
- Esse tipo de estudo é chamado:  
**Análise de Algoritmos**

# Problemas na Análise de Algoritmos

- Análise de um **algoritmo particular**
- Análise de uma **classe de algoritmos**.

# Problemas na Análise de Algoritmos

- **Análise de um algoritmo particular**
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Estudamos as características de um algoritmo específico:
    - Análise do número de vezes que cada parte do algoritmo deve ser executada
    - Estudo da quantidade de memória necessária para suas estruturas de dados
    - ...

# Problemas na Análise de Algoritmos

## ■ 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.

# Medida de Custo

- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.
- Como medir o custo de um algoritmo?
  - Medida de custo pela **execução do algoritmo**
  - Medida de custo por um **modelo matemático**

# Medida do Custo pela Execução do Programa

- Medidas que dependam da execução em um computador real (em geral) são inadequadas:
  - ❑ os resultados são dependentes do compilador;
  - ❑ os resultados dependem do *hardware*;
  - ❑ quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.



# Medida do Custo pela Execução do Programa

- Apesar disso, há argumentos a favor de se obter medidas reais de tempo.
  - 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.
  - Quando queremos analisar o comportamento do algoritmo no ambiente onde ele vai ser utilizado
- Nesse caso, tanto os custos reais das operações quanto os custos não aparentes do sistema, tais como alocação de memória, indexação, carga, são considerados.

# Medida do Custo por meio de um Modelo Matemático

- Usa um modelo matemático baseado em um computador idealizado.
- Computador idealizado:
  - Modelo RAM – Random Access Machine*  
(Máquina de Acesso Aleatório)
  - Um processador que executa uma ação por vez
  - Memória que armazena os dados
  - Operações básicas de custo constante
    - Acesso a memória
    - Testes condicionais
    - Operações aritméticas
    - Etc...

# Medida do Custo por meio de um Modelo Matemático

- Deve ser especificado o conjunto de operações e seus custos de execuções.
  - É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- 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 manipulação de índices, caso existam.

# Custo de um Algoritmo

- **Determinando o menor custo possível** para resolver problemas de uma dada classe, temos a medida da **difículdade inerente para resolver o problema**.
- **Algoritmo Ótimo:** Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.

# Função de Complexidade

- O custo de execução de um algoritmo é dado por uma função de custo **ou função de complexidade  $f$** .
- $f(n)$  é a medida do custo necessário para executar um algoritmo para um **problema de tamanho  $n$** .

# Função de Complexidade

- Função de **complexidade de tempo**:
  - $f(n)$  mede o custo em **número de operações** 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

- Em geral, ao longo do curso,  $f(n)$  será uma função de **complexidade de tempo**
  - Ela não representa tempo diretamente
  - Ela representa o **número de vezes** que determinada operação considerada relevante é executada

# Custo para encontrar o maior elemento de um vetor

- Problema: Encontrar o maior elemento de um vetor de inteiros  $A[n]$ ;  $n \geq 1$ .

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



# Custo para encontrar o maior elemento de um vetor

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

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

O loop é executado de 1 até  $n-1$ , ou seja, **são realizadas  $n-1$  iterações** e em cada iteração é feita uma comparação

$$F(n) = n - 1$$

# Custo para encontrar o 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.

# Custo para encontrar o maior elemento de um vetor

- **Prova:** O que determina que um elemento é o maior de uma lista qualquer? É o fato de todos os outros elementos serem menores do que ele. Cada um dos  $n - 1$  elementos deve ser testado, por meio de comparações, se é menor do que algum outro elemento.
- Logo,  **$n-1$**  comparações são necessárias

# Custo para encontrar o maior elemento de um vetor

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

# Função de Complexidade

- A medida do custo de execução de um algoritmo é uma função do **tamanho da entrada dos dados**.
- Para alguns algoritmos, o custo de execução depende também da **organização dos dados**, não apenas do tamanho da entrada.
- Nessas situações vamos **ter diferentes funções de complexidade** para representar o **melhor caso, pior caso e caso médio**.

# Registros de um Arquivo

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

# Registros de um Arquivo

## ■ Pesquisa sequencial

```
// retorna a posição do registro ou  
// n se a chave não estiver presente
```

```
int Pesquisa(TipoRegistro *A, int n, int chave) {  
    int i;  
  
    i = 0;  
    while(i < n) {  
        if (A[i].chave == chave) {  
            break;  
        }  
        i++;  
    }  
    return i;  
}
```

Seja  $f$  uma função de complexidade tal que  $f(n)$  é o número de comparações de chaves. Determine  $f(n)$ .

# Melhor e pior caso

- **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  $n$ .
  - Se  $f$  é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que  $f(n)$ .



# Caso médio

- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho  $n$ .
  - 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.
  - A análise do caso médio é geralmente **muito mais difícil de obter** do que as análises do melhor e do pior caso.

# Registros de um Arquivo

## ■ Pesquisa sequencial

```
// retorna a posição do registro ou  
// n se a chave não estiver presente
```

```
int Pesquisa(TipoRegistro *A, int n, int chave) {  
    int i;  
  
    i = 0;  
    while(i < n) {  
        if (A[i].chave == chave) {  
            break;  
        }  
        i++;  
    }  
    return i;  
}
```

$$F(n) = 1$$

Qual seria o **melhor caso**?

A chave procurada foi encontrada  
no primeiro registro!

# Registros de um Arquivo

## ■ Pesquisa sequencial

```
// retorna a posição do registro ou  
// n se a chave não estiver presente
```

```
int Pesquisa(TipoRegistro *A, int n, int chave) {  
    int i;  
  
    i = 0;  
    while(i < n) {  
        if (A[i].chave == chave) {  
            break;  
        }  
        i++;  
    }  
    return i;  
}
```

$$F(n) = n$$

Qual seria o **pior caso**?

A chave foi encontrada no último registro  
ou a chave não foi encontrada

# Registros de um Arquivo

## ■ Pesquisa sequencial

```
// retorna a posição do registro ou  
// n se a chave não estiver presente
```

```
int Pesquisa(TipoRegistro *A, int n, int chave) {  
    int i;  
  
    i = 0;  
    while(i < n) {  
        if (A[i].chave == chave) {  
            break;  
        }  
        i++;  
    }  
    return i;  
}
```

Qual seria o **caso médio**?

É necessário fazer uma  
*análise probabilística...*

# Registros de um Arquivo

## ■ Caso médio:

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

$$f(n) = 1.p_1 + 2.p_2 + \cdots + n.p_n$$

# Registros de um Arquivo

## ■ Caso médio:

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

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

# Registros de um Arquivo

## ■ Caso médio:

- Considerando que:  $p_i = \frac{1}{n}, 1 \leq i \leq n$
- Temos:

$$f(n) = 1.p_1 + 2.p_2 + \cdots + n.p_n$$

$$f(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \cdots + n.\frac{1}{n}$$

$$f(n) = \frac{1}{n} (1 + 2 + \cdots + n)$$

$$f(n) = \frac{(n+1)}{2}$$

$$f(n) = \frac{1}{n} \sum_{i=1}^n i \quad f(n) = \frac{1}{n} \cdot \frac{n(n+1)}{2}$$

# Registros de um Arquivo

- Seja  $f$  uma função de complexidade tal que  $f(n)$  é o 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 ou não está presente
  - $f(n) = n$
- **Caso médio**
  - $f(n) = (n+1)/2$



# Maior e Menor Elemento

- **Problema:** encontrar o maior e o menor elemento de um vetor de inteiros  $A[n]$ ;  $n \geq 1$ .

```
void MaxMin1(int *A, int n, int *Max, int
*Min) {
    int i;

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

# Maior e Menor Elemento

- Seja  $f(n)$  o número de comparações entre os elementos de  $A$ , se  $A$  contiver  $n$  elementos.
  - Qual será o melhor, pior e caso médio?

```
void MaxMin1(int *A, int n, int *Max, int
*Min) {
    int i;

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

$2*(n-1)$

# Maior e Menor Elemento

- Seja  $f(n)$  o número de comparações entre os elementos de  $A$ , se  $A$  contiver  $n$  elementos.
  - Qual será o melhor, pior e caso médio?

$$F(n) = 2(n - 1)$$

- A função de complexidade não varia com a organização da entrada

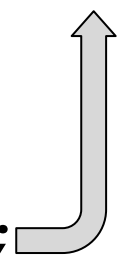
# Maior e Menor Elemento

- Como podemos diminuir a complexidade do algoritmo?

```
void MaxMin1(int *A, int n, int *Max, int
*Min) {
    int i;

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

A comparação  $A[i] < \text{Min}$   
só é necessária quando  
a comparação  $A[i] > \text{Max}$   
dá falso.



# Maior e Menor Elemento

- Quais são as funções de complexidade para o melhor, pior e caso médio?

```
void MaxMin2(int *A, int n, int *Max, int
*Min) {
    int i;

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

# Maior e Menor Elemento

## ■ Melhor caso:

- Quando os elementos estão em ordem crescente

$$F(n) = n - 1$$

```
void MaxMin2(int *A, int n, int *Max, int
*Min) {
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i]; (n-1)
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

# Maior e Menor Elemento

## ■ Pior caso:

- Quando o maior elemento é o primeiro elemento do vetor

$$F(n) = 2(n - 1)$$

```
void MaxMin2(int *A, int n, int *Max, int
              *Min) {
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

(n-1)  
(n-1)

# Maior e Menor Elemento

## ■ Caso médio:

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

$$F(n) = \frac{3}{2}(n - 1)$$

```
void MaxMin2(int *A, int n, int *Max, int
*Min) {
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

(n-1)

(n-1)/2

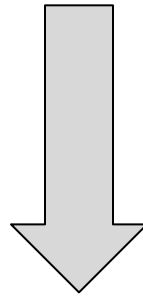


# Maior e Menor Elemento

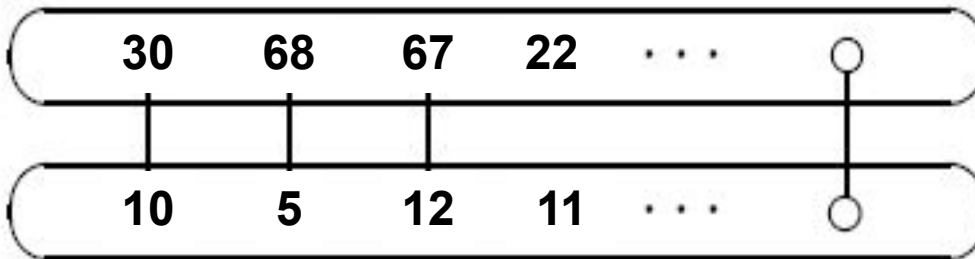
- É possível diminuir ainda mais a complexidade?

# Maior e Menor Elemento

10	30	5	68	12	67	22	11	...
----	----	---	----	----	----	----	----	-----

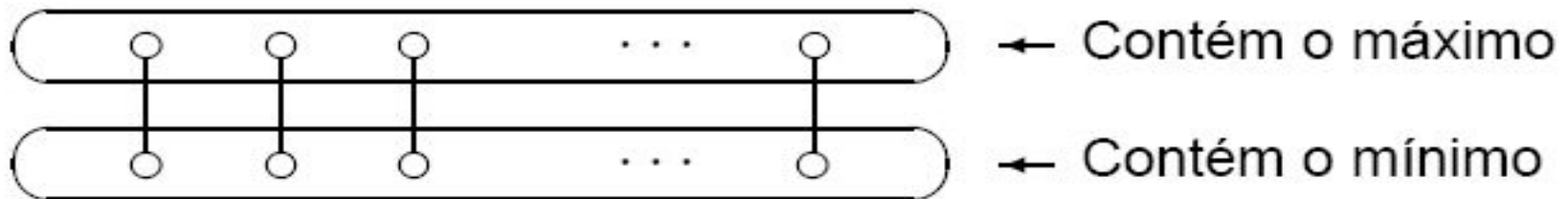


Compara-se os  
elementos  
aos pares



# Maior e Menor Elemento

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



# Qual a função de complexidade para este novo algoritmo?

- Os elementos de  $A$  são comparados dois a dois. Os elementos maiores são comparados com  $Max$  e os elementos menores são comparados com  $Min$ .
- 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:

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

no pior caso, melhor caso e caso médio

# Exemplo: Maior e Menor Elemento

```
void MaxMin3(int n, Vetor A, int *Max, int *Min) {
    int i, FimDoAnel;
    if ((n % 2) > 0) {
        A[n] = A[n - 1];
        FimDoAnel = n;
    }
    else FimDoAnel = n - 1;

    if (A[0] > A[1]) {
        *Max = A[0]; *Min = A[1];
    }
    else {
        *Max = A[1]; *Min = A[0];
    }
    i = 3;
    while (i <= FimDoAnel) {
        if (A[i - 1] > A[i]) {
            if (A[i - 1] > *Max) *Max = A[i - 1];
            if (A[i] < *Min) *Min = A[i];
        }
        else {
            if (A[i - 1] < *Min) *Min = A[i - 1];
            if (A[i] > *Max) *Max = A[i];
        }
        i += 2;
    }
}
```

# Comparação entre os Algoritmos

- Comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.

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	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

# Comparação entre os Algoritmos

- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.
- Existe possibilidade de obter um algoritmo MaxMin mais eficiente?
  - Qual é o **limite inferior** para essa classe de algoritmos?

# Limite Inferior

- Existe possibilidade de obter um algoritmo MaxMin mais eficiente?
- Para responder temos de conhecer o **limite inferior** para essa classe de algoritmos.
  - Como? Uso de um oráculo.



# Limite Inferior

- Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível.
- O oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível necessário para determinar a resposta final.

# Exemplo de Uso de um Oráculo

- **Teorema:** Qualquer algoritmo para encontrar o maior e o menor elemento de um conjunto com  $n$  elementos não ordenados,  $n > 1$ , faz pelo menos  $3n/2 - 2$  comparações.

# Exemplo de Uso de um Oráculo

- **Prova:** A técnica utilizada define um oráculo que descreve o comportamento do algoritmo utilizando:
  - um conjunto de  $n$ -tuplas (**estados**)
  - um conjunto de **regras** associadas que mostram as tuplas possíveis (**estados**) que um algoritmo pode assumir a partir de uma dada tupla e uma única comparação.

# Exemplo de Uso de um Oráculo

- Vamos utilizar uma 4–tupla, representada por **(a; b; c; d)**, onde os elementos de:
  - **a**: número de elementos **nunca comparados**;
  - **b**: **foram vencedores e nunca perderam** em comparações realizadas (máximo);
  - **c**: **foram perdedores e nunca venceram** em comparações realizadas (mínimo);
  - **d**: **foram vencedores e perdedores** em comparações realizadas (elementos intermediários).

# Exemplo de Uso de um Oráculo

- O algoritmo inicia no estado
  - $(n, 0, 0, 0)$ : nenhum dos  $n$  itens foram comparados
- E termina em
  - $(0, 1, 1, n - 2)$ : um máximo, um mínimo, e todos os outros  $n-2$  itens foram comparados

# Exemplo de Uso de um Oráculo

- Após cada comparação, (a; b; c; d) assume um dentre os 6 estados possíveis abaixo:
  - **2 elementos de a** são comparados
    - (a - 2, b + 1, c + 1, d)
  - **1 elemento de a** comparado com **1 de b** ou **1 de c** (se  $a \geq 1$ )
    - (a - 1, b + 1, c, d) – comparou **1 de a** com **1 de c** e **a** ganhou
    - (a - 1, b, c + 1, d) – comparou **1 de a** com **1 de b** e **a** perdeu
    - (a - 1, b, c, d + 1) – comparou **1 de a** com **1 de c** e **a** perdeu  
ou – comparou **1 de a** com **1 de b** e **a** ganhou
  - **2 elementos de b** são comparados (se  $b \geq 2$ )
    - (a, b - 1, c, d + 1)
  - **2 elementos de c** são comparados (se  $c \geq 2$ )
    - (a, b, c - 1, d + 1)

# Exemplo de Uso de um Oráculo

(a, b, c, d)

(n, 0, 0, 0)

(0, 1, 1, n-2)

# Exemplo de Uso de um Oráculo

(**a**, **b**, **c**, **d**)

$(n, 0, 0, 0)$



comparação de 2 a 2 elementos de **a**  
(caminho mais rápido para zerar **a**).

$(0, n/2, n/2, 0)$

$(0, 1, 1, n-2)$



# Exemplo de Uso de um Oráculo

(**a**, **b**, **c**, **d**)

(n, 0, 0, 0)



comparação de 2 a 2 elementos de a  
(caminho mais rápido para zerar **a**).

(0, n/2, n/2, 0)



**comparação de elementos em **b** para  
encontrar o máximo**

(0, 1, n/2, n/2-1)

(0, 1, 1, n-2)

# Exemplo de Uso de um Oráculo

(**a**, **b**, **c**, **d**)

(n, 0, 0, 0)



comparação de 2 a 2 elementos de **a**  
(caminho mais rápido para zerar **a**).

(0, n/2, n/2, 0)



comparação de elementos em **b** para  
encontrar o máximo

(0, 1, n/2, n/2-1)



comparação de elementos em **c** para  
encontrar o mínimo

(0, 1, 1, n-2)

# Exemplo de Uso de um Oráculo

(**a**, **b**, **c**, **d**)

$(n, 0, 0, 0)$

comparação de 2 a 2 elementos de **a**  
(caminho mais rápido para zerar **a**).

$n/2$

$(0, n/2, n/2, 0)$

comparação de elementos em **b** para  
encontrar o máximo

$n/2 - 1$

$(0, 1, n/2, n/2-1)$

comparação de elementos em **c** para  
encontrar o mínimo

$n/2 - 1$

$(0, 1, 1, n-2)$

# Exemplo de Uso de um Oráculo

- O passo 1 requer necessariamente a manipulação do componente **a**.
- O caminho mais rápido para levar **a** até zero requer  $n/2$  mudanças de estado e termina com a tupla  $(0, n/2, n/2, 0)$  (por meio de comparação dos elementos de **a** dois a dois).

# Exemplo de Uso de um Oráculo

- A seguir, para reduzir o componente **b** até um são necessárias  $n/2 - 1$  mudanças de estado (mínimo de comparações necessárias para obter o maior elemento de **b**).
- Idem para **c**, com  $n/2 - 1$  mudanças de estado.

# Exemplo de Uso de um Oráculo

- Para obter o estado  $(0, 1, 1, n - 2)$  a partir do estado  $(n, 0, 0, 0)$  são necessárias:
  - $n/2 + n/2 - 1 + n/2 - 1$  comparações.

**Logo,  $f(n) = 3n/2 - 2$ .**

# Exemplo de Uso de um Oráculo

- Voltando aos algoritmos anteriores (note que a função de complexidade  $f(n) = 3n/2 - 2$  não foi obtido analisando um **algoritmo**).
  - Qual é a informação que o teorema nos traz?
    - O algoritmo MaxMin3 é **ótimo**.

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	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

# Referências

- ❑ Ziviani, N., **Projeto de Algoritmos com Implementações em Pascal e C**, 3ª Edição, Cengage Learning, 2011.
  - Capítulo 1 (até seção 1.3.1)
  
- ❑ Cormen , T., Leiserson, C, Rivest R., Stein, C. **Algoritmos – Teoria e Prática**, 3a. Edição, Elsevier, 2012.
  - Seção 2.2