Medida do Tempo de Execução de um Programa

Livro "Projeto de Algoritmos" – Nívio Ziviani

Capítulo 1 – Seção 1.3

http://www2.dcc.ufmg.br/livros/algoritmos/

Projeto de Algoritmos

- Projeto de algoritmos
  1. Análise do problema
  2. Decisões de projeto
  3. Algoritmo a ser utilizado de acordo com seu comportamento.
- Comportamento depende de
  - tempo de execução
  - espaço ocupado.

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 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

- Medidas são bastante 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.

- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - 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.
  - Nesse caso, tanto os custos reais das operações como os custos não aparentes, 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.
- 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 manipulações de índices, caso existam.

Função de Complexidade

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

- Nas aulas, f denota uma função de **complexidade de tempo**
  - Apesar do nome, ela não representa tempo diretamente
  - Representa **o número de vezes que determinada operação considerada relevante é executada**.

Exemplo: maior elemento

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

```
int Max(int A[n]) {
   int i, Temp;

   Temp = A[0];
   for (i = 1; i < n; i++)
     if (Temp < A[i])
       Temp = A[i];
    return Temp;
}
```

- 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.
- **Qual a função f(n)?**

Exemplo: maior elemento

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

```
int Max(int A[n]) {
   int i, Temp;

   Temp = A[0];
   for (i = 1; i < n; i++)
      if (Temp < A[i])
        Temp = A[i];
    return Temp;
}
```

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

Exemplo: maior elemento

- **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

- **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 testado, por meio de comparações, se é menor do que algum outro elemento.

Exemplo: maior elemento

- **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 testado, por meio de comparações, se é menor do que algum outro elemento.
  - Logo, **n-1 comparações são necessárias**

Exemplo: maior elemento

- **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 testado, por meio de comparações, se 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.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
  - Para um algoritmo de ordenação isso não ocorre

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
  - No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
  - 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

- **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** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

Análise de Melhor Caso, Pior Caso e Caso Médio

- 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.
- 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 recuperar registros do arquivo.
- 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

- 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:**
  - **pior caso:**
  - **caso médio:**

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
  - **pior caso:**
  - **caso médio:**

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:**
  - **caso médio:**

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;
  - **caso médio:**

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:**

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:

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

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:

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

- 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:**
    - $f(n) = 1$
  - **pior caso:**
    - $f(n) = n$
  - **caso médio:**
    - $f(n) = (n + 1)/2.$

Exemplo - Maior e Menor Elemento (1)

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

Exemplo - Maior e Menor Elemento (1)

- Problema: 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[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];
    }
}
```

Qual a função de complexidade para MaxMin1?

```
void MaxMin1(int A[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)

Qual a função de complexidade para MaxMin1?

```
void MaxMin1(int A[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];
    }
}
```

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

Exemplo - Maior e Menor Elemento (2)

- MaxMin1 pode ser facilmente melhorado: a comparação A[i] < Min só é necessária quando a comparação A[i] > Max dá falso.

```
void MaxMin2(int A[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];
    }
}
```

Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int A[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];
    }
}
```

Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int A[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];
   }
}
```

Melhor caso:

Pior caso:

Caso médio:

Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int A[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];
    }
}
```

Melhor caso:

- quando os elementos estão em ordem crescente;

Pior caso:

Caso médio:

Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int A[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];
    }
}
```

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[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];
   }
}
```

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[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];    (n-1)
   }
}
```

Melhor caso:

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

Pior caso:

- quando os elementos estão em ordem decrescente;
- ***f(n) = 2(n – 1)***

Caso médio:

Qual a função de complexidade para MaxMin2?

```
void MaxMin2(int A[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];
   }
}
```

Melhor caso:

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

Pior caso:

- quando os elementos estão em ordem decrescente;
- $f(n) = 2(n - 1)$

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[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];     (n-1)/2
   }
}
```

Melhor caso:

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

Pior caso:

- quando os elementos estão em ordem decrescente;
- $f(n) = 2(n - 1)$

Caso médio:

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

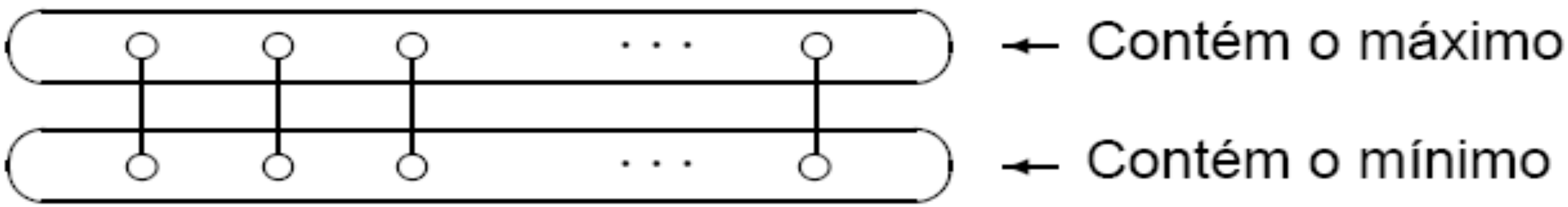
Exemplo - Maior e Menor Elemento (3)

- Podemos fazer melhor ainda para encontrar o mínimo e o máximo?

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

Exemplo - Maior e Menor Elemento (3)

- 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



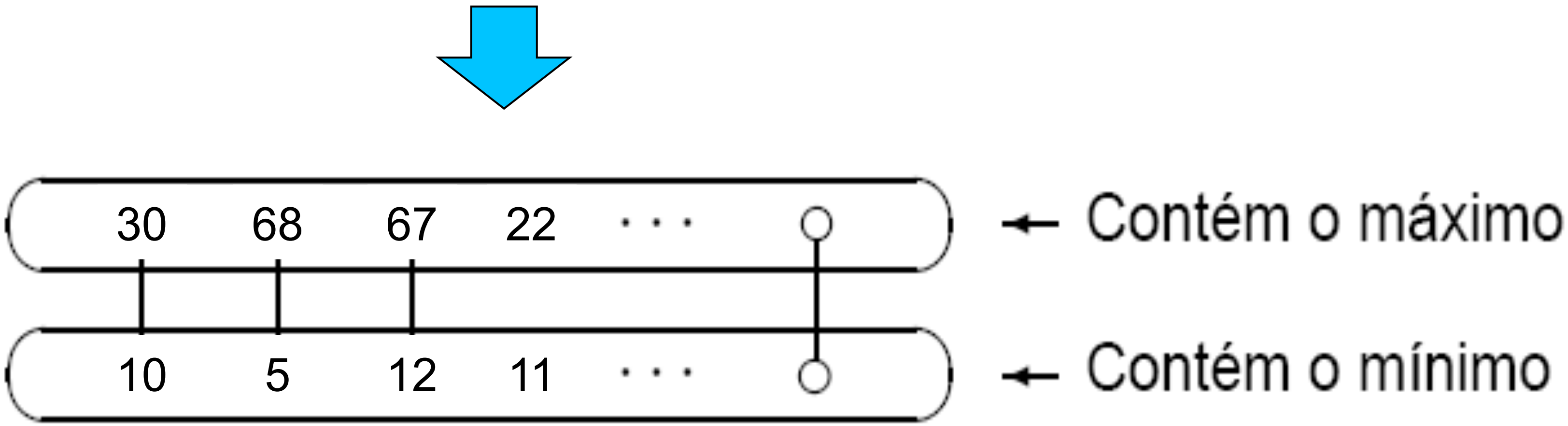
10	30	5	68	12	67	22	11	. . .

Exemplo - Maior e Menor Elemento (3)

10	30	5	68	12	67	22	11	. . .

Exemplo - Maior e Menor Elemento (3)

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



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 (3)

```
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;
   }
}
```

Exemplo - Maior e Menor Elemento (3)

```
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;
   }
}
```

10	30	5	68	12	67	...

Exemplo - Maior e Menor Elemento (3)

```
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;
   }
}
```

10	30	5	68	12	67	...

Comparação 1 → 1

Comparação 2 → (n/2) - 1

Comparação 3 → (n/2) - 1

Comparação 4 → (n/2) - 1

Comparação 3

Comparação 4

Qual a função de complexidade para MaxMin3?

- Quantas comparações são feitas em MaxMin3?

Qual a função de complexidade para MaxMin3?

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

Qual a função de complexidade para MaxMin3?

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

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

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

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

Comparação entre os Algoritmos

- A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- 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	$3n/2-2$	$3n/2-2$	$3n/2-2$

Exemplo

- Qual é a função de complexidade f(n) para o algoritmo abaixo?

```
Void funcao(int A[n], int B[n]) {
int i, j;

   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         if (A[i] > B[j])
            A[i] = A[i] + B[j];
         else
            B[j] = B[j] - A[i];
      }
   }
}
```

Limite Inferior - Uso de um Oráculo

- 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.
- Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação).
- Para derivar o limite inferior, 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 $\lceil 3n/2 \rceil$ - 2 comparações.**

- **Prova: A técnica utilizada define um oráculo** que descreve o comportamento do algoritmo utilizando:
  - um conjunto de n–tuplas,
  - 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

- Para o problema do maior e menor elemento, utilizamos 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 venceramem comparações realizadas (mínimo);
  - d: foram vencedores e perdedores em comparações realizadas (elementos intermediários).

- O algoritmo inicia no estado (n, 0, 0, 0) e termina com (0, 1, 1, n - 2).

Exemplo de Uso de um Oráculo

- Após cada comparação, (a; b; c; d) assume um dentre os 6 estados possíveis abaixo:
  - (a - 2, b + 1, c + 1, d) se a ≥ 2 (2 elementos de a são comparados)
  - (a - 1, b + 1, c, d) ou

    (a - 1, b, c + 1, d) ou

    (a - 1, b, c, d + 1)

    se a ≥ 1 (1 elemento de a comparado com 1 de b ou 1 de c)
  - (a, b - 1, c, d + 1) se b ≥ 2 (2 elementos de b são comparados)
  - (a, b, c - 1, d + 1) se c ≥ 2 (2 elementos de c são comparados)

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)

$\downarrow$ 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)

$\lceil n/2 \rceil$ comparação de 2 a 2 elementos de a (caminho mais rápido para zerar a).

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

n/2-1 comparação de elementos em b para encontrar o máximo

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

n/2-1 comparação de elementos em c para encontrar o mínimo

(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 $\lceil n/2 \rceil$ 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).
- A seguir, para reduzir o componente **b** até um são necessárias n/2 - 1 e mudanças de estado (mínimo de comparações necessárias para obter o maior elemento de b).
- Idem para c, com n/2 - 1 mudanças de estado.
- Logo, para obter o estado (0, 1, 1, n - 2) a partir do estado (n, 0, 0, 0) são necessárias

  $\lceil n/2 \rceil + n/2 - 1 + n/2 - 1 = \lceil 3n/2 \rceil - 2$ comparações.

Exemplo de Uso de um Oráculo

- O teorema nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo, então o algoritmo MaxMin3 é **ótimo**.

Solução

(a, b, c, d)

φ > < ><

Estado inicial

(n, 0, 0, 0)

Estado Final

(0, 1, 1, n-2)

$(a,b,c,d) \xrightarrow{2es\ (a)} (a-2, b+1, c+1, d)$

$(a,b,c,d) \xrightarrow{1a\ c/\ 1b} (a-1, b, c+1, d)$

$(a,b,c,d) \xrightarrow{1a\ c/1c} $

$(a-1, b, c, d+1)$

$(a-1, b+1, c, d)$

$(a-1, b, c, d+1)$

=

$(a,b,c,d) \xrightarrow{2b} (a, b-1, c, d+1)$

$(a,b,c,d) \xrightarrow{2c} (a, b, c-1, d+1)$

Solução (cont)

$$(n,0,0,0) \to \left(0,\frac{n}{2},\frac{n}{2},0\right) \to (0,1,1,n-2)$$

$n/2$

$\frac{n}{2}-1$

$\frac{n}{2}-1$

$$n/2 + n/2 - 1 + n/2 - 1 = \frac{3n}{2} - 2$$