

# 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/>

# 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

- Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
  - os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - 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.
  - 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

- 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

- Para medir o custo de execução de um algoritmo é comum definir uma 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  $n$ .
- Utilizaremos  $f$  para denotar uma função de complexidade de tempo daqui para a frente.
- 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

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

```
#define n 10
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  $A[n]$ ;  $n \geq 1$ .

```
#define n 10
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 investigado por meio de comparações, que 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 investigado por meio de comparações, que 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 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.
- 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 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 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 seqüencial**.

# 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:

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

# 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

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

# 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

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

- 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[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];  
    }  
}
```

# 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  $n > 0$ , para o melhor caso, pior caso e caso médio.

## Exemplo - Maior e Menor Elemento (2)

- MaxMin1 pode ser facilmente melhorado: a comparação  $A[i] < \text{Min}$  só é necessária quando a comparação  $A[i] > \text{Max}$  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];  
        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];
        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:

# 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];  
        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.
- $f(n) = 3n/2 - 3/2$



# 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:

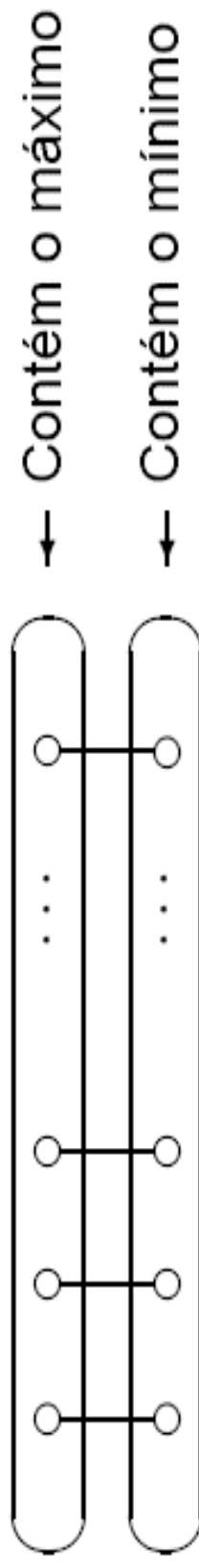
- No caso médio,  $A[i]$  é maior do que Max a metade das vezes.
- $f(n) = n - 1 + (n - 1)/2 = 3n/2 - 3/2$

## 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  $\lceil n/2 \rceil$  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 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  $\lceil n/2 \rceil$  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



# 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  $A[n]$  é duplicado na posição  $A[n + 1]$  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(Vetor A, int *Max, int *Min) {
    int i, FimDoAne1;

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

    if (A[0] > A[1]) {
        *Max = A[0]; *Min = A[1];
    }
    else {
        *Max = A[1]; *Min = A[0];
    }
    i = 3;
    while (i <= FimDoAne1) {
        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;
    }
}
```

# 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?
  - 1<sup>a</sup>. comparação feita 1 vez
  - 2<sup>a</sup>. comparação feita  $n/2 - 1$  vezes
  - 3<sup>a</sup>. e 4<sup>a</sup>. 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	$3n/2 - 2$	$3n/2 - 2$	$3n/2 - 2$

# 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.
- Técnica muito utilizada: 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  $3n/2 - 2$  comparações.
- **Prova:** A técnica utilizada define um oráculo que descreve o comportamento do algoritmo por meio de um conjunto de  $n$ -tuplas, mais 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

- Uma 4-tupla, representada por  $(a; b; c; d)$ , onde os elementos de:
  - a: nunca foram comparados;
  - b: foram vencedores e nunca perderam em comparações realizadas;
  - c: foram perdedores e nunca venceram comparações realizadas;
  - d: foram vencedores e perdedores em comparações realizadas.
- 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 tupla  $(a; b; c; d)$  consegue progredir apenas se ela assume um dentre os seis estados possíveis abaixo:
  - $(a - 2, b + 1, c + 1, d)$ 
    - se  $a \geq 2$  (dois 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  $a \geq 1$  (um elemento de  $a$  comparado com um de  $b$  ou um de  $c$ )
  - $(a, b - 1, c, d + 1)$ 
    - se  $b \geq 2$  (dois elementos de  $b$  são comparados)
  - $(a, b, c - 1, d + 1)$ 
    - se  $c \geq 2$  (dois elementos de  $c$  são comparados)

# Exemplo de Uso de um Oráculo

- O primeiro passo 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.

# Exemplo de Uso de um Oráculo

- Logo, para obter o estado  $(0, 1, 1, n - 2)$  a partir do estado  $(n, 0, 0, 0)$  são necessárias  $n/2 + n/2 - 1 + \lceil n/2 \rceil - 1 = \lceil 3n/2 \rceil - 2$  comparações.
- 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**.

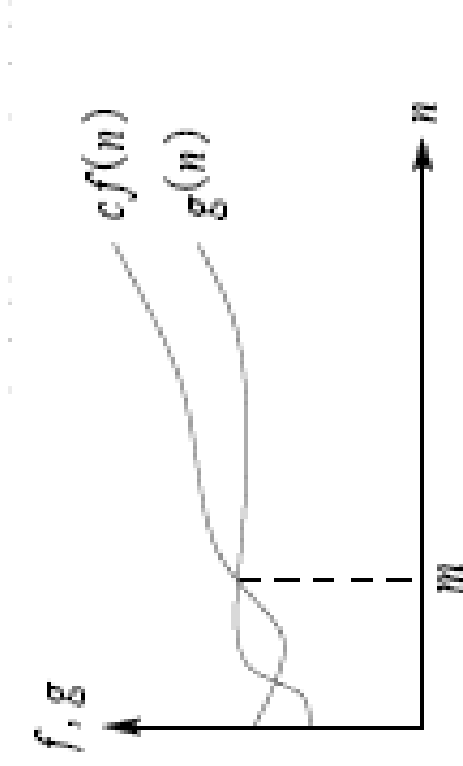
# Comportamento Assintótico de Funções

- O parâmetro  $n$  fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de  $n$ , qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de  $n$ .
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de  $n$ )
- O comportamento assintótico de  $f(n)$  representa o limite do comportamento do custo quando  $n$  cresce.



# Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **Definição:** Uma função  $f(n)$  **domina assintoticamente** outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $m$  tais que, para  $n \geq m$ , temos  $|g(n)| \leq c \times |f(n)|$ .



# Dominação assintótica

## Exemplo:

- Sejam  $g(n) = (n + 1)^2$  e  $f(n) = n^2$ .
- As funções  $g(n)$  e  $f(n)$  dominam assintoticamente uma a outra, desde que

$$|(n + 1)^2| \leq 4|n^2| \text{ para } n \geq 1 \text{ e}$$

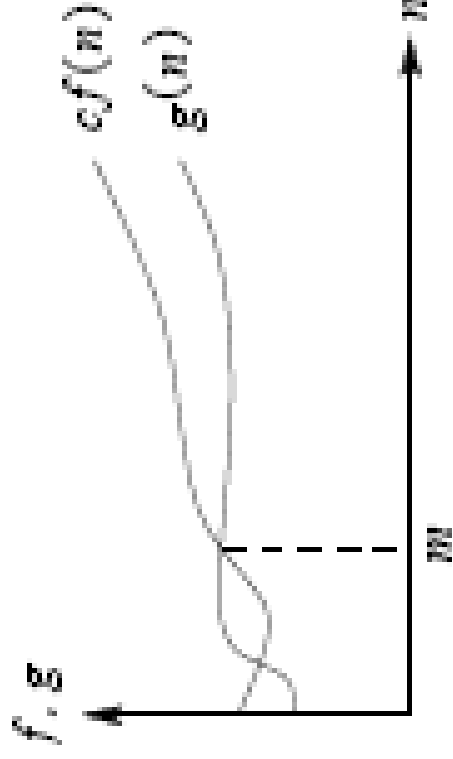
$$|n^2| \leq |(n + 1)^2| \text{ para } n \geq 0.$$

# Notação O

- **Definição:** Uma função  $g(n)$  é  $O(f(n))$  se existem duas constantes positivas  $c$  e  $m$  tais que  $g(n) \leq cf(n)$ , para todo  $n \geq m$ .
- O valor da constante  $m$  mostrado é o menor valor possível, mas qualquer valor maior também é válido.

# Notação O

- Escrevemos  $g(n) = O(f(n))$  para expressar que  $f(n)$  domina assintoticamente  $g(n)$ . Lê-se  $g(n)$  é da ordem no máximo  $f(n)$ .
- Exemplo: quando dizemos que o tempo de execução  $T(n)$  de um programa é  $O(n^2)$ , significa que existem constantes  $c$  e  $m$  tais que, para valores de  $n \geq m$ ,  $T(n) \leq cn^2$ .
- Exemplo gráfico de dominação assintótica que ilustra a notação  $O$ .



# Exemplos de Notação $O$

- **Exemplo:**  $g(n) = (n + 1)^2$ .
  - Logo  $g(n)$  é  $O(n^2)$ , quando  $m = 1$  e  $c = 4$ .
  - Isto porque  $(n + 1)^2 \leq 4n^2$  para  $n \geq 1$ .
- **Exemplo:**  $g(n) = n$  e  $f(n) = n^2$ .
  - Sabemos que  $g(n)$  é  $O(n^2)$ , pois para  $n \geq 0$ ,  $n \leq n^2$ .
  - Entretanto  $f(n)$  não é  $O(n)$ .
  - Suponha que existam constantes  $c$  e  $m$  tais que para todo  $n \geq m$ ,  $n^2 \leq cn$ .
  - Logo  $c \geq n$  para qualquer  $n \geq m$ , e não existe uma constante  $c$  que possa ser maior ou igual a  $n$  para todo  $n$ .

# Operações com a Notação $O$

$$f(n) = O(f(n))$$

$$c \times O(f(n)) = O(f(n)) \quad c = \text{constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

$$O(f(n))O(g(n)) = O(f(n)g(n))$$

$$f(n)O(g(n)) = O(f(n)g(n))$$

# Exemplos de Notação $O$

- **Exemplo:**  $g(n) = 3n^3 + 2n^2 + n$  é  $O(n^3)$ .
  - Basta mostrar que  $3n^3 + 2n^2 + n \leq 6n^3$ , para  $n \geq 0$ .
  - A função  $g(n) = 3n^3 + 2n^2 + n$  é também  $O(n^4)$ , entretanto esta afirmação é mais fraca do que dizer que  $g(n)$  é  $O(n^3)$ .
- **Exemplo:**  $g(n) = \log_5 n$  é  $O(\log n)$ .
  - O  $\log_b n$  difere do  $\log_c n$  por uma constante que no caso é  $\log_b c$ .
  - Como  $n = c^{\log_c n}$ , tomando o logaritmo base  $b$  em ambos os lados da igualdade, temos que
$$\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c.$$

# Operações com a Notação $O$

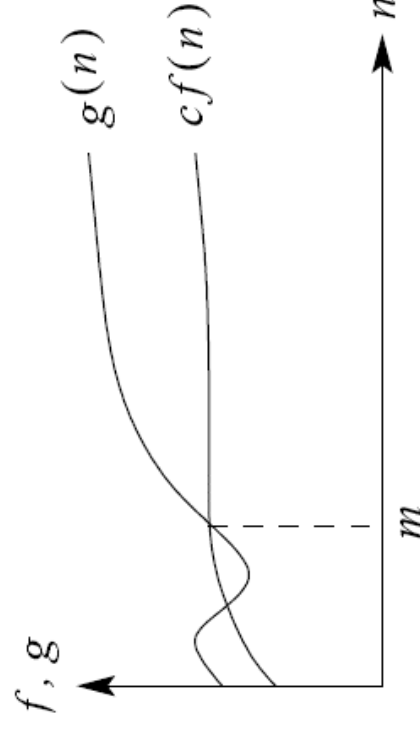
**Exemplo:** regra da soma  $O(f(n)) + O(g(n))$ .

- Suponha três trechos cujos tempos de execução são  $O(n)$ ,  $O(n^2)$  e  $O(n \log n)$ .
- O tempo de execução dos dois primeiros trechos é  $O(\max(n, n^2))$ , que é  $O(n^2)$ .
- O tempo de execução de todos os três trechos é então  $O(\max(n^2, n \log n))$ , que é  $O(n^2)$ .



# Notação $\Omega$

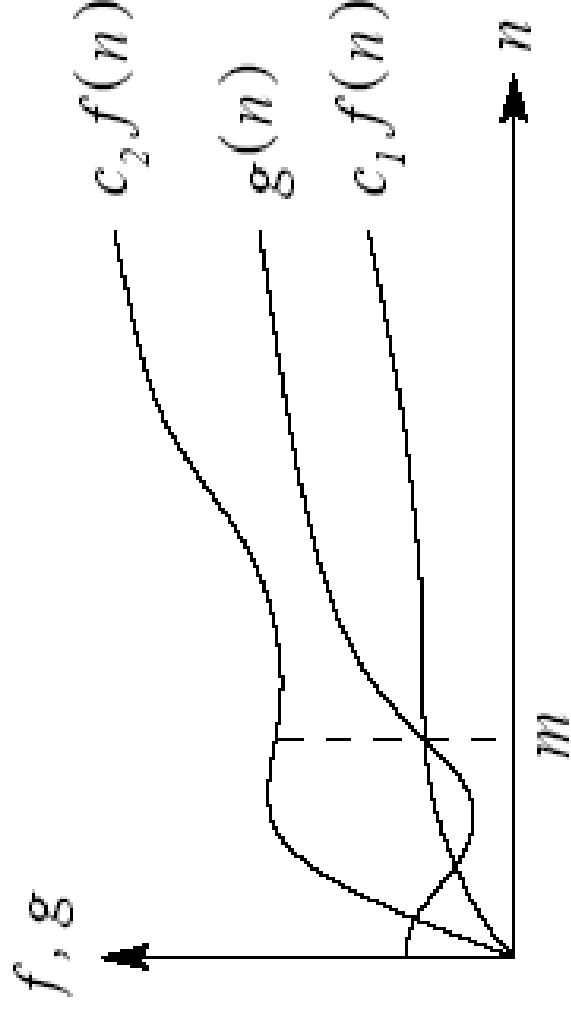
- Especifica um limite inferior para  $g(n)$ .
- **Definição:** Uma função  $g(n)$  é  $(f(n))$  se existirem duas constantes  $c$  e  $m$  tais que  $g(n) \geq cf(n)$ , para todo  $n \geq m$ .
- **Exemplo:** Para mostrar que  $g(n) = 3n^3 + 2n^2$  é  $\Omega(n^3)$  basta fazer  $c = 1$ , e então  $3n^3 + 2n^2 \geq n^3$  para  $n \geq 0$ .
- Exemplo gráfico para a notação:



- Para todos os valores à direita de  $m$ , o valor de  $g(n)$  está sobre ou acima do valor de  $cf(n)$ .

# Notação $\Theta$

- **Definição:** Uma função  $g(n)$  é  $\Theta(f(n))$  se existirem constantes positivas  $c_1$ ,  $c_2$  e  $m$  tais que  $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$ , para todo  $n \geq m$ .
- Exemplo gráfico para a notação:



# Notação $\Theta$

- Dizemos que  $g(n) = \Theta(f(n))$  se existirem constantes  $c_1$ ,  $c_2$  e  $m$  tais que, para todo  $n \geq m$ , o valor de  $g(n)$  está sobre ou acima de  $c_1 f(n)$  e sobre ou abaixo de  $c_2 f(n)$ .
- Isto é, para todo  $n \geq m$ , a função  $g(n)$  é igual a  $f(n)$  a menos de uma constante.
- Neste caso,  $f(n)$  é um **limite assintótico firme**.

# Notação $o$

- Usada para definir um limite superior que não é assintoticamente firme.
- **Definição:** Uma função  $g(n)$  é  $o(f(n))$  se, para qualquer constante  $c > 0$ , então  $0 \leq g(n) < cf(n)$  para todo  $n \geq m$ .
- **Exemplo:**  $2n = o(n^2)$ , mas  $2n^2 \neq o(n^2)$ .

# Notação $o$

- Em  $g(n) = O(f(n))$ , a expressão  $0 \leq g(n) \leq cf(n)$  é válida para alguma constante  $c > 0$ , mas em  $g(n) = o(f(n))$ , a expressão  $0 \leq g(n) < cf(n)$  é válida para todas as constantes  $c > 0$ .
- Na notação  $o$ , a função  $g(n)$  tem um crescimento muito menor que  $f(n)$  quando  $n$  tende para infinito.
- Alguns autores usam  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$  para a definição da notação  $o$ .

# Notação $\omega$

- Por analogia, a notação  $\omega$  está relacionada com a notação  $\Omega$  da mesma forma que a notação  $o$  está relacionada com a notação  $O$ .
- **Definição:** Uma função  $g(n)$  é  $\omega(f(n))$  se, para qualquer constante  $c > 0$ , então  $0 <= cf(n) < g(n)$  para todo  $n >= m$ .
- **Exemplo:**  $\frac{n^2}{2} = \omega(n)$ , mas  $\frac{n^2}{2} \neq \omega(n^2)$ .
- A relação  $g(n) = \omega(f(n))$  implica  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$ , se o limite existir.

# Classes de Comportamento Assintótico

- Se  $f$  é uma **função de complexidade** para um algoritmo  $F$ , então  $O(f)$  é considerada a **complexidade assintótica** ou o comportamento assintótico do algoritmo  $F$ .
- A relação de dominação assintótica permite comparar funções de complexidade.
- Entretanto, se as funções  $f$  e  $g$  dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
- Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.

# Classes de Comportamento Assintótico

- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é,  $f(n) = 3g(n)$ , sendo que  $O(f(n)) = O(g(n))$ .
- Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante.



# Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.
- Um programa com tempo de execução  $O(n)$  é melhor que outro com tempo  $O(n^2)$ .
- Porém, as constantes de proporcionalidade podem alterar esta consideração.

# Comparação de Programas

- Exemplo: um programa leva  $100n$  unidades de tempo para ser executado e outro leva  $2n^2$ .
- Qual dos dois programas é melhor?
  - depende do tamanho do problema.
  - Para  $n < 50$ , o programa com tempo  $2n^2$  é melhor do que o que possui tempo  $100n$ .
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é  $O(n^2)$ .
  - Entretanto, quando  $n$  cresce, o programa com tempo de execução  $O(n^2)$  leva muito mais tempo que o programa  $O(n)$ .

# Principais Classes de Problemas

- $f(n) = O(1)$ .
  - Algoritmos de complexidade  $O(1)$  são ditos de **complexidade constante**.
  - Uso do algoritmo independe de  $n$ .
  - As instruções do algoritmo são executadas um número fixo de vezes.

# Principais Classes de Problemas

- $f(n) = O(\log n)$ .
  - Um algoritmo de complexidade  $O(\log n)$  é dito ter **complexidade logarítmica**.
  - Típico em algoritmos que transformam um problema em outros menores.
  - Pode-se considerar o tempo de execução como menor que uma constante grande.
  - Quando  $n$  é mil,  $\log_2 n \approx 10$ , quando  $n$  é 1 milhão,  $\log_2 n \approx 20$ .
  - Para dobrar o valor de  $\log n$  temos de considerar o quadrado de  $n$ .
  - A base do logaritmo muda pouco estes valores: quando  $n$  é 1 milhão, o  $\log_2 n$  é 20 e o  $\log_{10} n$  é 6.

# Principais Classes de Problemas

- $f(n) = O(n)$ .
  - Um algoritmo de complexidade  $O(n)$  é dito ter **complexidade lineal**
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - É a melhor situação possível para um algoritmo que tem de processar/produzir  $n$  elementos de entrada/saída.
  - Cada vez que  $n$  dobra de tamanho, o tempo de execução dobra.

# Principais Classes de Problemas

- $f(n) = O(n \log n)$ .
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois.
  - Quando  $n$  é 1 milhão,  $n \log_2 n$  é cerca de 20 milhões.
  - Quando  $n$  é 2 milhões,  $n \log_2 n$  é cerca de 42 milhões, pouco mais do que o dobro.

# Principais Classes de Problemas

- $f(n) = O(n^2)$ .
  - Um algoritmo de complexidade  $O(n^2)$  é dito ter **complexidade quadrática**.
    - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
    - Quando  $n$  é mil, o número de operações é da ordem de 1 milhão.
    - Sempre que  $n$  dobra, o tempo de execução é multiplicado por 4.
    - Úteis para resolver problemas de tamanhos relativamente pequenos.
- $f(n) = O(n^3)$ .
  - Um algoritmo de complexidade  $O(n^3)$  é dito ter **complexidade cúbica**.
    - Úteis apenas para resolver pequenos problemas.
    - Quando  $n$  é 100, o número de operações é da ordem de 1 milhão.
    - Sempre que  $n$  dobra, o tempo de execução fica multiplicado por 8.

# Principais Classes de Problemas

- $f(n) = O(2^n)$ .
  - Um algoritmo de complexidade  $O(2^n)$  é dito ter **complexidade exponencial**.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
  - Quando  $n$  é 20, o tempo de execução é cerca de 1 milhão. Quando  $n$  dobra, o tempo fica elevado ao quadrado.
- $f(n) = O(n!)$ .
  - Um algoritmo de complexidade  $O(n!)$  é dito ter complexidade exponencial, apesar de  $O(n!)$  ter comportamento muito pior do que  $O(2^n)$ .
  - Geralmente ocorrem quando se usa **força bruta** para na solução do problema.
  - $n = 20 \Rightarrow 20! = 2432902008176640000$ , um número com 19 dígitos.
  - $n = 40 \Rightarrow$  um número com 48 dígitos.



# Comparação de Funções de Complexidade

Função de custo	Tamanho $n$					
	10	20	30	40	50	60
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0,35 s	0,0036 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.	$10^{13}$ séc.

# Algoritmos Polinomiais

- **Algoritmo exponencial** no tempo de execução tem função de complexidade  $O(c^n)$ ;  $c > 1$ .
- **Algoritmo polinomial** no tempo de execução tem função de complexidade  $O(p(n))$ , onde  $p(n)$  é um polinômio.
  - A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
  - Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais.
- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.

# Algoritmos Polinomiais

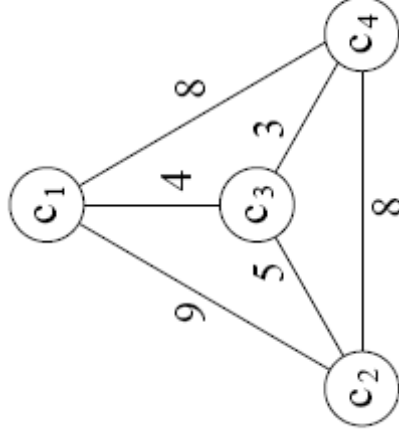
- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema.
- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.

# Algoritmos Polinomiais x Algoritmos Exponenciais

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- Exemplo: um algoritmo com função de complexidade  $f(n) = 2^n$  é mais rápido que um algoritmo  $g(n) = n^5$  para valores de  $n$  menores ou iguais a 20.
- Também existem algoritmos exponenciais que são muito úteis na prática.
- Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

# Exemplo de Algoritmo Exponencial

- Um **caixeiro viajante** deseja visitar  $n$  cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.
- A figura ilustra o exemplo para quatro cidades  $c_1$ ,  $c_2$ ,  $c_3$ ,  $c_4$ , em que os números nos arcos indicam a distância entre duas cidades.



- O percurso  $\langle c_1, c_3, c_4, c_2, c_1 \rangle$  é uma solução para o problema, cujo percurso total tem distância 24.

# Exemplo de Algoritmo Exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há  $(n - 1)!$  rotas possíveis e a distância total percorrida em cada rota envolve  $n$  adições, logo o número total de adições é  $n!$ .
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria  $50! \approx 10^{64}$ .
- Em um computador que executa  $10^9$  adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que  $10^{45}$  séculos só para executar as adições.
- O problema do caixeiro viajante aparece com frequência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.

# Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo;
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas,
  - produtos,
  - permutações,
  - fatoriais,
  - coeficientes binomiais,
  - solução de **equações de recorrência**.

# Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita:  $O(1)$ .
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é  $O(1)$ .
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente  $O(1)$ ), multiplicado pelo número de iterações.



# Análise do Tempo de Execução

- **Procedimentos não recursivos:** cada um deve ser computado separadamente um a um, iniciando com os que não chamam outro procedimentos. Avalia-se então os que chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos:** associada uma função de complexidade  $f(n)$  desconhecida, onde  $n$  mede o tamanho dos argumentos.

# Procedimento não Recursivo

Algoritmo para ordenar os  $n$  elementos de um conjunto  $A$  em ordem ascendente.

```
void Ordena(Vetor A)
{ /*ordena o vetor A em ordem ascendente*/
  int i, j, min,x;
  for (i = 1; i < n; i++)
    { min = i;
      for (j = i + 1; j <= n; j++)
        if ( A[j - 1] < A[min - 1] )
          min = j;
        /*troca A[min] e A[i]*/
        x = A[min - 1];
        A[min - 1] = A[i - 1];
        A[i - 1] = x;
      }
    }
}
```

# Procedimento não Recursivo

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento  $A[1]$ .
- Repita as duas operações acima com os  $n - 1$  elementos restantes, depois com os  $n - 2$ , até que reste apenas um.

# Análise do Procedimento não Recursivo

## Anel Interno

- Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
- Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que será sempre executado.
- O tempo para incrementar o índice do anel e avaliar sua condição de terminação é  $O(1)$ .
- O tempo combinado para executar uma vez o anel é  $O(\max(1, 1, 1)) = O(1)$ , conforme regra da soma para a notação  $O$
- □ Como o número de iterações é  $n - i$ , o tempo gasto no anel é  $O((n - i) \times 1) = O(n - i)$ , conforme regra do produto para a notação  $O$ .

# Análise do Procedimento não Recursivo

## Anel Externo

- Contém, além do anel interno, quatro comandos de atribuição.  $O(\max(1, (n - i), 1, 1, 1)) = O(n - i)$ .
- A linha (1) é executada  $n - 1$  vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de  $(n - i)$ :

$$\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

- Se **considerarmos o número de comparações** como a medida de custo relevante, o programa faz  $(n^2)/2 - n/2$  comparações para ordenar  $n$  elementos.
- Se considerarmos o número de trocas, o programa realiza exatamente  $n - 1$  trocas.

# Algoritmos Recursivos

# Conceito de Recursividade

- Fundamental em Matemática e Ciência da Computação
  - Um programa recursivo é um programa que chama a si mesmo
  - Uma função recursiva é definida em termos dela mesma
- Exemplos
  - Números naturais, Função fatorial
- Conceito poderoso
  - Define conjuntos infinitos com *comandos* finitos

# Definições

- Condição necessária e suficiente para codificar programas recursivos
  - Procedimentos ou sub-rotinas
- Procedimento diretamente recursivo
  - Chama a si mesmo
- Procedimento indiretamente recursivo
  - **A** chama **B** que chama **A**



# Condição de terminação

- Nenhum programa nem função pode ser exclusivamente definido por si
  - Um programa seria um loop infinito
  - Uma função teria definição circular
- Condição de terminação
  - Permite que o procedimento pare de executar
  - $F(x) > 0$  onde  $x$  é decrescente

# Implementação de Recursividade

- Usa-se uma **pilha para armazenar os dados** usados em cada chamada de um procedimento que ainda não terminou.
- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação.
- Quando uma ativação anterior prossegue, os dados da pilha são recuperados.

# Sumário

- Exemplos simples de recorrência matemática
  - Utilização não prática
- Uso prático
  - Dividir Para Conquistar!
- Remoção da recursividade
  - Utilização de pilha explícita

# Função fatorial

- Função fatorial

$$\square N! = N \times (N - 1)! \quad N > 0, 0! = 1$$

```
int fatorial(int N)
{
    if (N == 0) return 1;
    else return N * fatorial(N - 1);
}
```

# Análise da função fatorial

- Complexidade de tempo
  - $O(n)$
- Complexidade de espaço
  - $O(n)!!!$

```
int fatorial(int N)
{
    int i, fat;
    fat = 1;
    for (i = 2; i <= N; i++) fat = fat * i;
    return fat;
}
```

# Sequência de Fibonacci

## ■ Definição

- $F_n = F_{n-1} + F_{n-2}$        $n > 2, F_0 = F_1 = 1$
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

```
int fibonacci(int N)
{
    if (N < 2) return 1;
    else return fibonacci(N - 1) + fibonacci(N - 2);
}
```

# Análise da função Fibonacci

- Exemplo ainda menos *convincente*
  - Não use recursividade cegamente
- Ineficiência em Fibonacci
  - Termos  $F_{n-1}$  e  $F_{n-2}$  são computados independentemente
  - Custo para cálculo de  $F_n$ 
    - $O(\phi^n)$  onde  $\phi = (1 + \sqrt{5})/2 = 1,61803\dots$
    - *Golden ratio*
    - Exponencial!!!

# Alternativa para fibonacci

```
int fibonacci(int N)
{
    int t1, t2, cnt, aux;
    if (N < 2) return 1;
    else {
        t1 = 1; t2 = 1; cnt = 2;
        while (cnt < N) {
            aux = t1 + t2;
            t2 = t1; t1 = aux; cnt = cnt + 1;
        }
        return t1;
    }
}
```



# Versão iterativa do Cálculo de Fibonacci

- O programa tem complexidade de tempo  $O(n)$  e complexidade de espaço  $O(1)$ .
- Devemos evitar uso de recursividade quando existe solução óbvia por iteração.
- Comparação versões recursiva e iterativa:

$n$	20	30	50	100
<i>Recursiva</i>	1 seg	2 min	21 dias	$10^9$ anos
<i>Iterativa</i>	1/3 mseg	1/2 mseg	3/4 mseg	1,5 mseg

# Quando Não Usar Recursividade

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo.
- Estes podem ser caracterizados pelo esquema

$P \equiv \text{if } B \text{ then } (S; P)$

- Tais programas são facilmente transformáveis em uma versão não recursiva

$P \equiv (x := x0; \text{while } B \text{ do } S)$

# Problemas com recursividade

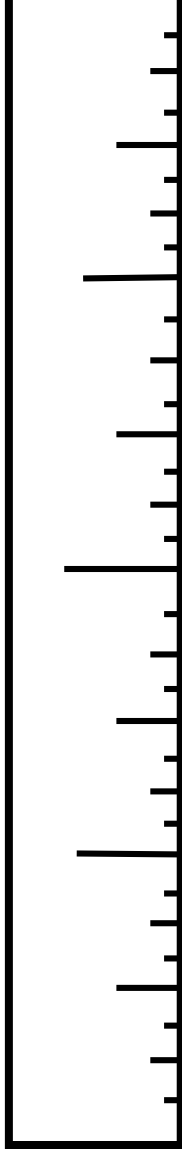
- Programa recursivo x Função definida recursivamente
  - Relação mais filosófica do que prática
- Problemas de implementação
  - Não com o conceito de recursividade

# Dividir para Conquistar

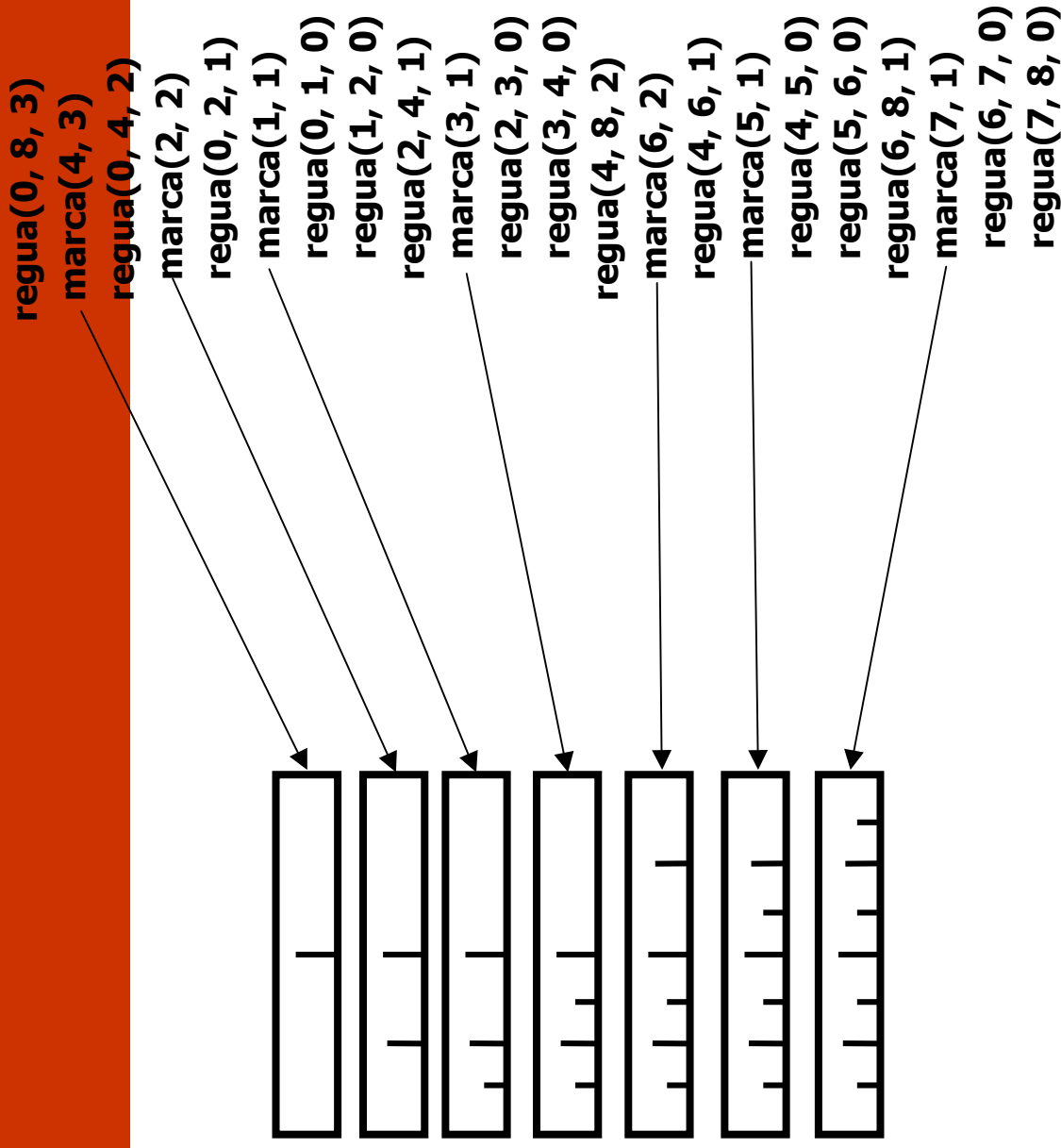
- Duas chamadas recursivas
  - Cada uma resolvendo a metade do problema
- Muito usado na prática
  - Solução eficiente de problemas
  - Decomposição
- Não se reduz trivialmente como fatorial
  - Duas chamadas recursivas
- Não produz recomputação excessiva como fibonacci
  - Porções diferentes do problema

# Exemplo simples: régua

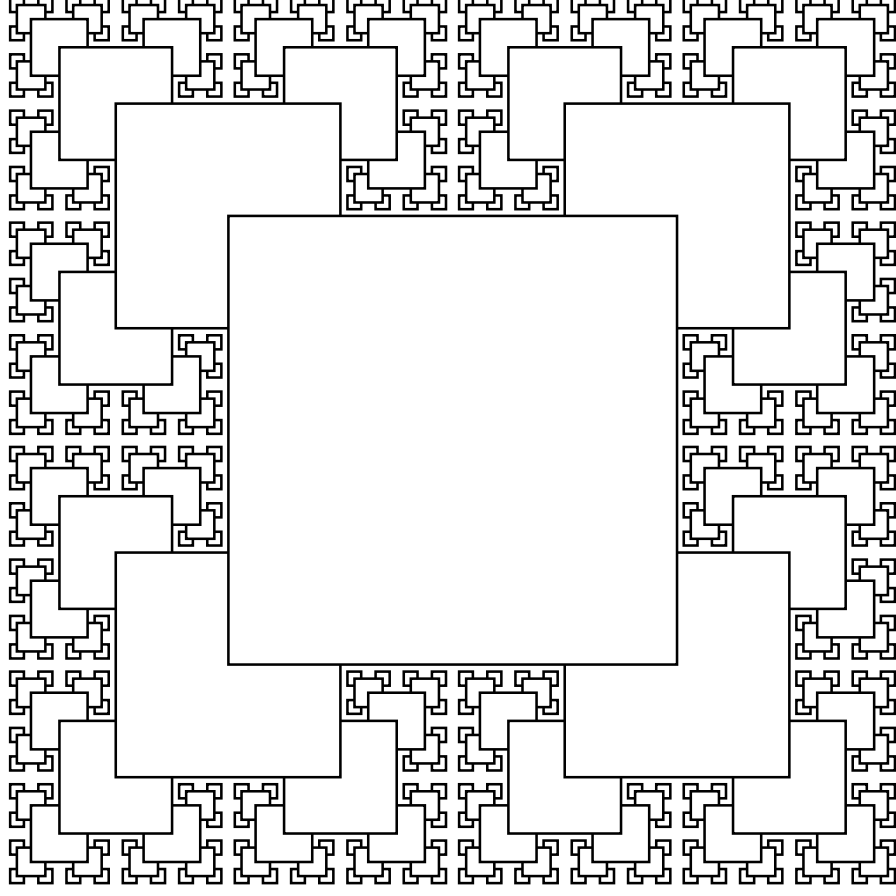
```
void regua(int l, int r, int h)
{
    int m;
    if (h > 0) {
        m = (int) ((l + r) / 2);
        marca(m, h);
        regua(l, m, h - 1);
        regua(m, r, h - 1);
    }
}
```



# Execução: régua



# Outros exemplos de recursividade



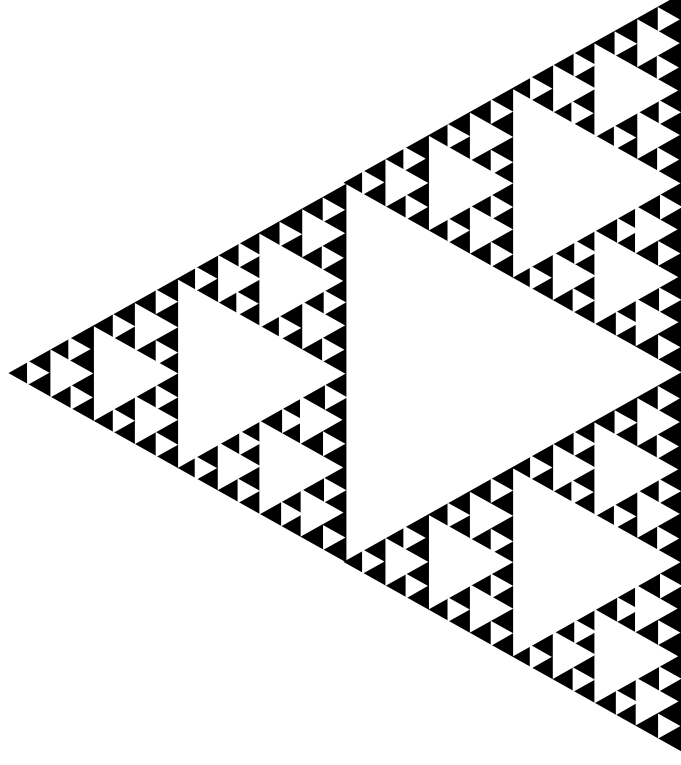
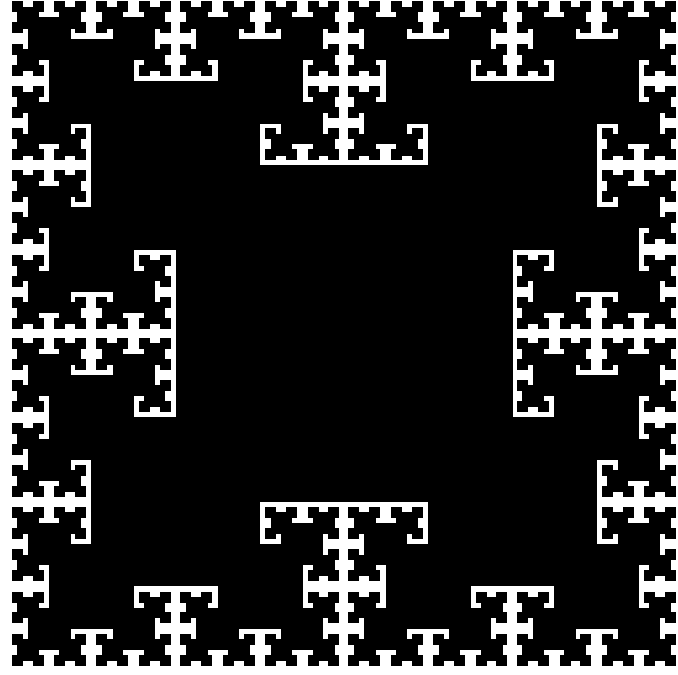
```
void estrela(int x, y, r)
{
    if (r > 0) {
        estrela(x-r, y+r, r / 2);
        estrela(x+r, y+r, r / 2);
        estrela(x-r, y-r, r / 2);
        estrela(x+r, y-r, r / 2);
        box(x, y, r);
    }
}
```

# Fractais

- Recursividade simples pode levar a computações aparentemente muito complexas
- Padrões geométricos definidos recursivamente
  - *Fractais*
- Padrões surpreendentemente diversificados podem ser obtidos
  - Primitivas de desenho mais sofisticadas
  - Funções recursivamente definidas com reais e no plano complexo



# Outros exemplos: fractais



# Finalizando

- Recursividade é um tópico fundamental
- Algoritmos recursivos aparecem na prática muito comumente
- Dividir e conquistar é uma técnica naturalmente recursiva para solução de problemas
- Mais recursividade no nosso futuro... ;-)

# Análise de Complexidade de Procedimento Recursivo

```
Pesquisa(n) {  
  (1)  if (n < 1) {  
  (2)    ‘inspecione elemento’ e termine;  
        }  
  else {  
  (3)    para cada um dos n elementos ‘inspecione  
          elemento’ ;  
        }  
  (4)  Pesquisa(n/3) ;  
}
```

# Procedimento Recursivo

- Para cada procedimento recursivo é associada uma função de complexidade  $f(n)$  desconhecida, onde  $n$  mede o tamanho dos argumentos para o procedimento.
- Obtemos uma equação de recorrência para  $f(n)$ .
- **Equação de recorrência:** maneira de definir uma função por uma expressão envolvendo a mesma função.

# Análise do Procedimento Recursivo

- Seja  $T(n)$  uma função de complexidade que represente o número de inspeções nos  $n$  elementos do conjunto.
- O custo de execução das linhas (1) e (2) é  $O(1)$  e o da linha (3) é exatamente  $n$ .
- Usa-se uma **equação de recorrência** para determinar o nº de chamadas recursivas.
- O termo  $T(n)$  é especificado em função dos termos anteriores  $T(1)$ ,  $T(2)$ , ...,  $T(n - 1)$ .

# Análise do Procedimento Recursivo

- $T(n) = n + T(n/3)$ ;
- $T(1) = 1$  (para  $n = 1$  fazemos uma inspeção)
- Por exemplo:
  - $T(3) = T(3/3) + 3 = 4$ ,
  - $T(9) = T(9/3) + 9 = 13$ ,
  - e assim por diante.
- Para calcular o valor da função seguindo a definição são necessários  $k - 1$  passos para computar o valor de  $T(3^k)$ .

# Exemplo de Resolução de Equação de Recorrência

- Substitui-se os termos  $T(k)$ ,  $k < n$ , até que todos os termos  $T(k)$ ,  $k > 1$ , tenham sido substituídos por fórmulas contendo apenas  $T(1)$ .

$$T(n) = n + T(n/3)$$

$$T(n/3) = n/3 + T(n/3/3)$$

$$T(n/3/3) = n/3/3 + T(n/3/3/3)$$

⋮  
⋮

$$T(n/3/3 \cdots /3) = n/3/3 \cdots /3 + T(n/3 \cdots /3)$$

# Exemplo de Resolução de Equação de Recorrência

- **Adicionando lado a lado, temos**

$$T(n) = n + n(1/3) + n(1/3^2) + n(1/3^3) + \dots + T(n/3/3\dots/3)$$

**que representa a soma de uma série geométrica de razão 1/3, multiplicada por n, e adicionada de  $T(n/3/3\dots/3)$ , que é menor ou igual a 1.**



# Exemplo de Resolução de Equação de Recorrência

$$T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \dots + (n/3 \dots /3)$$

- Se desprezarmos o termo  $T(n/3 \dots /3)$ , quando  $n$  tende para infinito, então

$$T(n) = n \sum_{i=0}^{\infty} (1/3)^i = n \left( \frac{1}{1 - \frac{1}{3}} \right) = \frac{3n}{2}.$$

- Se considerarmos o termo  $T(n/3/3 \dots /3)$  e denominarmos  $x$  o número de subdivisões por 3 do tamanho do problema, então  $n/3^x = 1$ , e  $n = 3^x$ . Logo  $x = \log_3 n$

# Exemplo de Resolução de Equação de Recorrência

- Lembrando que  $T(1) = 1$  temos

$$T(n) = \sum_{i=0}^{x-1} \frac{n}{3^i} + T\left(\frac{n}{3^x}\right) = n \sum_{i=0}^{x-1} (1/3)^i + 1 = \frac{n(1-(\frac{1}{3})^x)}{(1-\frac{1}{3})} + 1 = \frac{3n}{2} - \frac{1}{2}.$$

- Logo, o programa do exemplo é  $O(n)$ .

# Exemplo 1

```
void Pesquisa(Vetor A, int esq, int dir, Chave ch)
{
    int m;
    if (esq < dir) {
        m = (esq + dir)/2;
        if (A[m].chave == ch) printf("Achou elemento na pos %d!\n",m);
        else {
            if (A[m].chave > ch) Pesquisa(A,esq,m-1,ch);
            else Pesquisa (A,m+1,dir,ch);
        }
    }
    else if (A[esq].chave == ch)
        printf("Achou elemento na pos %d!\n",esq);
}
```

## Exemplo 2

```
void sort(Vetor A, int i,j)
{
    int k;
    if (i < j) {
        k = ( (j - i) + 1) / 3;
        sort(A, i, i+k-1);
        sort(A, i+k, i+2k-1);
        sort(A, i+2k, j);
        merge(A, i, i+k, i+2k, j);
        // Merge intercala subvetores a custo  $5n/3 - 2$ 
    }
}
```

## Exemplo 3

```
void Misterio (int n; Vetor A)
{
    int i;
    i= 1;
    while i <= n) {
        visita(i, A);
        i = i+1;
    }
}
```

```
void visita (int i, Vetor A)
{
    int j;
    if (i > 0) {
        for (j=1; j<= i; j++)
            A[j]:=TRUE;
        visita(i-1,A);
    }
}
```