Programação 2

# Analise de Algoritmos

Instituto Federal do Espírito Santo - Ifes
Campus Colatina
Professor: Giovany Frossard Teixeira

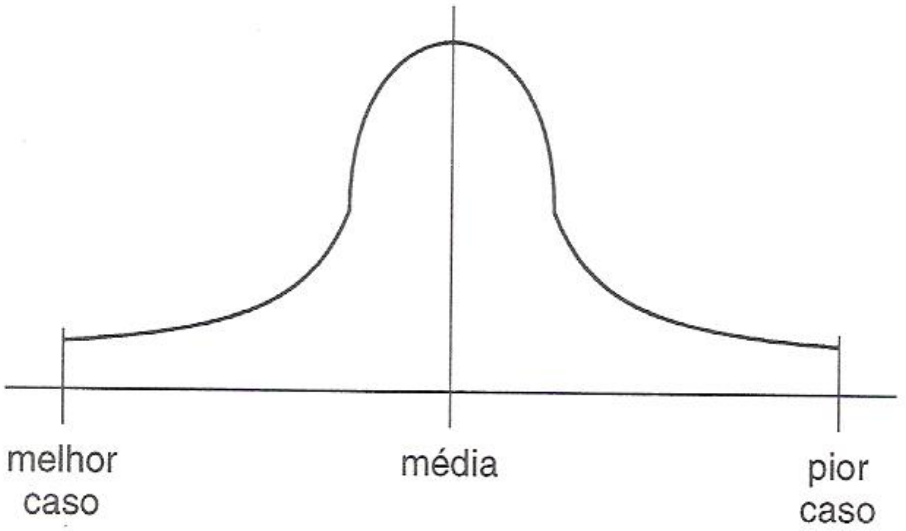
# Análise de Algoritmos

- Consiste em estudar o comportamento de um algoritmo frente a diferentes tamanhos e valores de entrada.

- Na realidade, o que se busca é estimar o tempo que um certo algoritmo consome para resolver um problema.

# Definições

- Melhor caso: entrada de dados que possibilita o melhor desempenho do algoritmo para resolver o problema.
- Pior caso: entrada de dados que implica no pior desempenho do algoritmo para resolver o problema.
- Caso médio: demais entradas de dados que normalmente equidistantes do melhor e do pior caso (estimativa).



# Notação O

- Dizemos que uma função f(n) é O(g(n)), se para valores suficientemente grandes de n, a função f(n) não será maior que g(n) a menos de um fator c.
- Exemplos:

$2n^2 - 5 = O(n^2)$ ==> pois $2n^2 - 5 <= 2n^2$ (para n qualquer)

$3n^2 + n = O(n^2)$ ==> pois $3n^2 + n <= 4n^2$ (para n>=1)

$2n^3 + n + 10 = O(n^3)$ ==> pois $2n^3 + n + 10 <= 3n^3$ (para n >= 3)

# Notação O

- Na análise de algoritmos são as seguintes as complexidades mais comuns (em ordem crescente):
  - O(1) ou constante;
  - O(log n) ou logaritímica;
  - O(n) ou linear;
  - O(n log n) ou n log de n;
  - $O(n^2)$ ou quadrática;
  - $O(n^3)$ ou cúbica;
  - $O(k^n)$ ou exponencial; (k uma constante qualquer)

# Notação O

- **Regras:**

I. Regra da complexidade *polinomial*: se P(n) é um polinômio de grau k, então P(n) = $O(n^k)$;

II. f(n) = O(f(n));

III. Regra da constante: O(c*f(n)) = c* O(f(n)) = O(f(n));

IV. Regra da *soma* de tempos: se T1(n) = O(f(n)) e T2(n) = O(g(n))

então T1(n) + T2(n) = O(max(f(n),g(n)));

Isto significa que a complexidade de um algoritmo com dois trechos em sequência com tempos de execução diferentes é dada como a complexidade do trecho de maior complexidade.

# Notação O

- **Regras:**

V. Regra do *produto* de tempos:

Se $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$

então $T_1(n) * T_2(n) = O(f(n) * g(n))$

Isto significa que a complexidade de um algoritmo com dois trechos aninhados, em que o segundo é repetidamente executado pelo primeiro, é dada como o produto da complexidade do trecho mais interno pela complexidade do trecho mais externo.

# Importância do estudo de complexidade

- Considere 5 algoritmos com as complexidades de tempo. Suponhamos que uma operação leve 1 ms.

| $n$ | $f_1(n) = n$ | $f_2(n) = n \log n$ | $f_3(n) = n^2$ | $f_4(n) = n^3$ | $f_5(n) = 2^n$ |
|---|---|---|---|---|---|
| 16 | 0.016s | 0.064s | 0.256s | 4s | 1m 4s |
| 32 | 0.032s | 0.16s | 1s | 33s | 46 dias |
| 512 | 0.512s | 9s | 4m 22s | 1 dia 13h | $10^{137}$ séculos |

- Se utilizássemos uma máquina mais rápida onde uma operação leve 1 ps (pico segundo) ao invés de 1 ms teríamos que, ao invés de $10^{137}$ séculos, seriam $10^{128}$ séculos.
- Podemos muitas vezes melhorar o tempo de execução de um programa otimizando o código (por exemplo: usar x + x ao invés de 2x, evitar re-cálculo de expressões já calculadas, etc.).
- Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com outra complexidade de tempo, por exemplo um algoritmo de O(n log n) ao invés de $O(n^2)$.

# Importância do estudo de complexidade

| FUNÇÃO DE COMPLEXIDADE | n (tamanho do problema) | | |
|---|---|---|---|
| | 20 | 40 | 60 |
| n | 0.0002 s | 0.0004 s | 0.0006 s |
| $n \log_2 n$ | 0.0009 s | 0.0021 s | 0.0035 s |
| $n^2$ | 0.0040 s | 0.0160 s | 0.0360 s |
| $n^3$ | 0.0800 s | 0.6400 s | 2.1600 s |
| $2^n$ | 10.0000 s | 27 dias | 3660 séculos |
| $3^n$ | 580 minutos | 38550 séculos | $1.3*10^{14}$ séculos |

É digna de nota a taxa de crescimento dos algoritmos de ordem exponencial. Em geral, seu desempenho torna-se de custo proibitivo, devendo ser usados apenas quando não se conheça solução de menor complexidade.

# Importância do estudo de complexidade

| FUNÇÃO DE COMPLEXIDADE | Tamanho da maior instância solucionável em 1 hora | | |
|---|---|---|---|
| | máquina original | máquina 100 vezes mais rápida | máquina 1.000 vezes mais rápidas |
| $n$ | N | 100 N | 1.000 N |
| $n \log_2 n$ | N1 | 22.5 N1 | 140.2 N1 |
| $n^2$ | N2 | 10 N2 | 31.6 N2 |
| $n^3$ | N3 | 4.6 N3 | 10 N3 |
| $2^n$ | N4 | N4 + 6 | N4 +10 |
| $3^n$ | N5 | N5 + 4 | N5 + 6 |

# Complexidade de algumas estruturas de controle

- Regras rígidas sobre o cálculo da complexidade de qualquer algoritmo não existem, cada caso deve ser estudado em suas condições.
- No entanto, as estruturas de controle clássicas da programação estruturada permitem uma estimativa típica de cada uma.
- A partir disso, algoritmos construídos com combinações delas podem ter sua complexidade mais facilmente estabelecida.

# Complexidade de algumas estruturas de controle

**A) Comando Simples:** tem um tempo de execução constante, **O**(c) = **O**(1).

**B) Seqüência:** tem um tempo igual à soma dos tempos de cada comando da seqüência; se cada comando é **O**(1), assim, também será a seqüência; senão, pela regra da soma, a seqüência terá a complexidade do comando de maior complexidade.

**C) Alternativa:** qualquer um dos ramos pode ter complexidade arbitrária; a complexidade resultante é a maior delas; isto vale para alternativa dupla (*if-else*) ou múltipla (*switch*).

# Complexidade de algumas estruturas de controle

D) **Repetição contada**: é aquela em que cada iteração (ou "volta") atualiza o controle mediante uma *adição* (geralmente, quando se usa uma estrutura do tipo *for*, que especifica incremento/decremento automático de uma variável inteira). Se o número de iterações é independente do tamanho do problema, a complexidade de toda a repetição é a complexidade do corpo da mesma, pela regra da constante (ou pela regra da soma de tempos).

```
for (i=0; i<k ; i++)

   trecho com O(g(n))
```

**// se *k* não é *f(n)* então**
**// o trecho é O(g(n))**

```
for (i=0; i<10 ; i++)
{
   x = x+v;
   printf ("%d", x);
}
```

**// isto é O(1), logo toda**
**// a repetição é O(1)**

# Complexidade de algumas estruturas de controle

- Se o número de iterações é função de n, pela regra do produto teremos a complexidade da repetição como a complexidade do corpo multiplicada pela função que descreve o número de iterações. Isto é:

```
for (i=0; i<n ; i++)          // como o número de iterações é f(n)=n
                              // então o trecho é O(n*g(n))
  trecho com O(g(n))
```

Exemplo:

```
for (i=0; i<k*n ; i++)        // o trecho é O(f(n)*g(n)), no caso
                              // O(k*n*log n), ou seja: O(n log n)
  trecho com O(log n)
```

# Complexidade de algumas estruturas de controle

- Uma aplicação comum da regra do produto é a determinação da complexidade de repetições aninhadas.

Exemplo:

```
for (i=0; i<n ; i++)
  for (j=0; j<n ; j++)
    trecho com O(1)
```

**// o trecho é O(f(n)*g(n)), no caso**
**// g(n)=n*1 (laço interno); logo,**
**// O(n*n), ou seja: O(n²)**

Exemplo:

```
for (i=1; i<=n ; i++)
  for (j=1; j<=i ; j++)
    trecho com O(1)
```

**// o laço interno é executado 1+2+3**
**// +...n-1 +n=n*(n+1)/2 vezes, logo,**
**// O(n*(n+1)/2), ou seja:**
**// O(0.5(n²+n)) ou seja O(n²)**

# Complexidade de algumas estruturas de controle

Exemplo:

```
for (i=1; i<=n ; i++)
  for (j=n; i<=j ; j--)
    trecho com O(1)
```

**// o laço interno é executado n+n-1**
**// +n-2+...+2+1=n*(n+1)/2 vezes, ou**
**// seja: O(n²) como no caso anterior**

- Os dois últimos exemplos podem ser generalizados para quaisquer aninhamentos de repetições contadas em *k* níveis, desde que todos os índices dependam do tamanho do problema. Nesse caso, a complexidade da estrutura aninhada será da ordem de $n^k$.

# Complexidade de algumas estruturas de controle

*E)* ***R*epetição Multiplicativa:** é aquela em que cada iteração atualiza o controle mediante uma multiplicação ou divisão.

Exemplo:

```
limite=1;
while (limite<=n)
{
  trecho com O(1)
  limite = limite*2;
}
```

**// o número de iterações depende**
**// de n; limite vai dobrando a cada**
**// iteração; depois de k iterações, limite = $2^k$ e**
**// k = $\log_2$ limite; como o valor**
**// máximo de limite é n, então**
**// o trecho é O($\log_2$n) = O(log n)**

**OBS**: Na verdade **O**(log n) independe da base do logaritmo:
Pois $\log_a n = \log_a b * \log_b n = c * \log_b n$. (notar que a e b são constantes, logo $\log_a b = c$)

# Complexidade de algumas estruturas de controle

Exemplo:

```
int limite;
for (limite=n; limite!=0; limite /=2)
  trecho com O(1)
```

**/* o número de iterações depende de n; limite vai-se subdividindo a cada iteração; depois de $k=\log_2 n$ iterações, encerra; então o trecho é O(log n)*/**

- Os dois exemplos anteriores também podem ser generalizados, adotando-se um fator genérico de multiplicação *fator*. Nesse caso, o número de iterações será dado por $k = log_{fator} limite$ = **O**(log f(n)), se o limite é função de n.

# Complexidade de algumas estruturas de controle

Exemplo:

```
int limite=n;
while (limite!=0)
{
  for (i=1; i<=n; i++)
    trecho com O(1)
  limite = limite/2;
}
```

/* o número de iterações depende de n; limite vai-se subdividindo a cada iteração; o laço interno é O(n), o externo O (log n); logo, o trecho é O (n log n)*/

# Complexidade de algumas estruturas de controle

## F) Chamada de Procedimento

Pode ser resolvida considerando-se que o procedimento também tem um algoritmo com sua própria complexidade. Esta é usada como base para cálculo da complexidade do algoritmo invocador. Por exemplo: se a invocação estiver num ramo de uma alternativa, sua complexidade será usada na determinação da máxima complexidade entre os dois ramos; se estiver no interior de um laço, será considerada no cálculo da complexidade da seqüência repetida, etc.

# Complexidade de algumas estruturas de controle

- A questão se complica ao se tratar de uma chamada recursiva.
- Embora não haja um método único para esta avaliação, em geral a complexidade de um algoritmo recursivo será função de componentes como: a *complexidade da base* e do *núcleo* da solução e a *profundidade da recursão.* Por este termo entende-se o *número de vezes que o procedimento é invocado recursivamente.* Este numero, usualmente, depende do *tamanho do problema* e da *taxa de redução do tamanho do problema* a cada invocação. E é na sua determinação que reside a dificuldade da análise de algoritmos recursivos.

# Complexidade de algumas estruturas de controle

- Como exemplo, considere o algoritmo do cálculo fatorial.

```
int fatorial (int n)
{
  if (n==0)
    return 1; // Base
  else
    return n*fatorial(n- 1); //Núcleo
}
```

A redução do problema se faz de uma em uma unidade, a cada reinvocação do procedimento, a partir de *n,* até alcançar *n = 0.* Logo, a profundidade da recursão é igual a *n*. O núcleo da solução (que é repetido a cada reinvocação) tem complexidade **O**(1), pois se resume a uma multiplicação. A base tem complexidade **O**(1), pois envolve apenas uma atribuição simples. Nesse caso, conclui-se que o algoritmo tem um tempo *T(n) = n*1+1* = **O**(n).

# Bibliografia


- SANTOS, Henrique José. **Curso de Linguagem C da UFMG**, apostila.
- FORBELLONE, André Luiz. **Lógica de Programação – A Construção de Algoritmos e Estruturas de Dados.** São Paulo: MAKRON, 1993.