COMPLEXIDADE DE ALGORITMOS

Prof. Muriel Mazzetto

Estrutura de Dados

Complexidade

- Projetar algoritmos eficientes.
  - Diferentes algoritmos podem resolver o mesmo problema sem ter a mesma eficiência.

Complexidade

- Projetar algoritmos eficientes.
- Medir complexidade:

Complexidade

- Projetar algoritmos eficientes.
- Medir complexidade:
  - Tempo de execução:

Complexidade

- Projetar algoritmos eficientes.
- Medir complexidade:
  - Tempo de execução:
    - Diferente para cada máquina;
    - Diferente para cada linguagem;
    - Diferente para cada SO;
    - Diferente para cada compilador.

Complexidade

- Projetar algoritmos eficientes.
- Medir complexidade:
  - Tempo de execução:
    - Diferente para cada máquina;
    - Diferente para cada linguagem;
    - Diferente para cada SO;
    - Diferente para cada compilador.
  - Memória utilizada.

Complexidade

- Projetar algoritmos eficientes.
- Medir complexidade:
  - Tempo de execução:
    - Diferente para cada máquina;
    - Diferente para cada linguagem;
    - Diferente para cada SO;
    - Diferente para cada compilador.
  - Memória utilizada.
  - **Quantidade de operações: independe de hardware e software.**

Complexidade

- Quantidade de operações.
  - Atribuições/Acessos;
  - Comparações/Expressões lógicas;
  - Operações aritméticas.

Complexidade

- Quantidade de operações.
  - Atribuições/Acessos;
  - Comparações/Expressões lógicas;
  - Operações aritméticas.

```c
int BuscaSequencial(int* vetor, int tam, int chave)
{
    int i;
    for(i = 0; i < tam; i++)
    {
        if(vetor[i] == chave)
        {
            return i; //retornar índice ou ponteiro
        }
    }
    return -1;//não encontrou indice;
}
```

Complexidade

- Quantidade de operações.
  - Atribuições/Acessos;
  - Comparações/Expressões lógicas;
  - Operações aritméticas.

```
                                          n
int BuscaSequencial(int* vetor, int tam, int chave)
{
                  1     n+1     n
    int i;
    for(i = 0; i < tam; i++)
    {
        if(vetor[i] == chave)  n
        {
                   1
            return i; //retornar índice ou ponteiro
        }
    }
                  1
    return -1;//não encontrou indice;
}
```

Complexidade

- Quantidade de operações.
  - Atribuições/Acessos;
  - Comparações/Expressões lógicas;
  - Operações aritméticas.

n

```
int BuscaSequencial(int* vetor, int tam, int chave)
{
    int i;         1      n+1     n
    for(i = 0; i < tam; i++)
    {
        if(vetor[i] == chave)  n
        {                    1
            return i; //retornar índice ou ponteiro
        }
    }                 1
    return -1; //não encontrou indice;
}
```

3n + 3

Complexidade

- A quantidade de operações está em função da entrada e ordenação dos dados.

- Pior caso: executar a maior quantidade de operações para a entrada.
- Caso médio: comportamento para casos comuns, deve conhecer o problema trabalhado.
- Melhor caso: executar a menor quantidade de operações para a entrada.

Funções de complexidade

- Uma função f(n) define a complexidade de um algoritmo.

- $f1(n) = 4n + 4$
- $f2(n) = 3n^2 + 3n$
- $f3(n) = n^3 + n$
- $f4(n) = nlogn + 29$

Complexidade

- Exercício: determine a função de custos para o trecho a seguir.

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        matriz[i][j] = j;
    }
}
```

Complexidade

- Exercício:
  - Pior caso.
    - Crescente.
  - Melhor caso.
    - Decrescente.

```c
int main(void)
{
    int n = 5;
    int A[5] = {1, 2, 3, 4, 5};
    int i, M = 0;

    for(i = 0; i < n; i++)
    {
        if(A[i] > M)
        {
            M = A[i];
        }
    }

    return 0;
}
```

Funções de complexidade

- Quando se analisa um algoritmo, espera-se uma entrada muito grande.
- $n \rightarrow \infty$ (n tendendo a infinito)

- $f1(n) = 4n + 4 \approx \boldsymbol{4n}$
- $f2(n) = 3n^2 + 3n \approx \boldsymbol{3n^2}$
- $f3(n) = n^3 + n \approx \boldsymbol{n^3}$
- $f4(n) = nlogn + 29 \approx \boldsymbol{nlogn}$

Funções de complexidade

- Quando se analisa um algoritmo, espera-se uma entrada muito grande.
- $n \to \infty$ (n tendendo a infinito)

- $f1(n) = 4n + 4 \approx \mathbf{4n}$
- $f2(n) = 3n^2 + 3n \approx \mathbf{3n^2}$
- $f3(n) = n^3 + n \approx \mathbf{n^3}$
- $f4(n) = nlogn + 29 \approx \mathbf{nlogn}$

DESCARTAR COEFICIENTES DEVIDO ÀS DIFERENTES LINGUAGENS E COMPILADORES.

FOCAR APENAS NA IDEIA DO ALGORITMO E NO CUSTO GERAL.

Funções de complexidade

- Quando se analisa um algoritmo, espera-se uma entrada muito grande.
- $n \to \infty$ (n tendendo a infinito)

- $f1(n) = 4n + 4 \approx \boldsymbol{n}$
- $f2(n) = 3n^2 + 3n \approx \boldsymbol{n^2}$
- $f3(n) = n^3 + n \approx \boldsymbol{n^3}$
- $f4(n) = nlogn + 29 \approx \boldsymbol{nlogn}$

Comportamento assintótico

- $n \to \infty$ (n tendendo a infinito)
- Para entradas grandes onde apenas a **ordem** da função é importante para diferenciá-las.

- $f1(n) = 105 \approx \mathbf{1}$
- $f2(n) = 15n + 2 \approx \boldsymbol{n}$
- $f3(n) = n^2 + 5n + 2 \approx \boldsymbol{n^2}$
- $f4(n) = nlogn + 112 \approx \boldsymbol{nlogn}$

Notação assintótica

- Definir limites nas funções de complexidade:
  - $O(f(n))$ – Big O;
  - $\Omega(f(n))$ – Big Ômega;
  - $\Theta(f(n))$ – Big Theta;
  - $o(f(n))$ – Small O;
  - $\omega(f(n))$ – Small Ômega;
  - $\theta(f(n))$ – Small Theta.
- Estimar uma entrada mínima n_0, para restringir as funções.

Notação assintótica

- **O(f(n)) – Big O:**
  - Define o máximo que a função alcançará.
  - O pior caso.
- Considere:
  - Funções g(n) e f(n);
  - **Uma constante C;**

$$g(n) \in O(f(n)) \quad se \quad g(n) \leq Cf(n) \; \forall \; n \geq n_0$$

- g(n) está "abaixo **ou é igual**" de f(n).

Notação assintótica

- **o(f(n)) – Small O:**
  - Similar a Big O, mas não “chega” na função.
- Considere:
  - Funções g(n) e f(n);
  - Uma constante C;

$$g(n) \in o(f(n)) \quad se \quad g(n) < Cf(n) \; \forall \, n \geq n_0$$

- g(n) está “abaixo” de f(n).

Notação assintótica

- **Ω(f(n)) – Big Ômega:**
  - Define o mínimo que a função alcançará.
  - O melhor caso.
- Considere:
  - Funções g(n) e f(n);
  - Uma constante C;

$$g(n) \in \Omega(f(n)) \quad se \quad g(n) \geq Cf(n)\ \forall\ n \geq n_0$$

- g(n) está "acima **ou é igual**" de f(n).

Notação assintótica

- **ω(f(n)) – Small Ômega:**
  - Similar a Big Ω, mas não "chega" na função.
- Considere:
  - Funções g(n) e f(n);
  - Uma constante C;

$$g(n) \in \omega(f(n)) \quad se \quad g(n) > Cf(n)\ \forall\ n \geq n_0$$

- g(n) está "acima" de f(n).

Notação assintótica

- **Θ(f(n)) – Big Theta:**
  - Define uma faixa de restrição.
  - Caso médio.
- Considere:
  - Funções g(n) e f(n);
  - Uma constante C;

$$g(n) \in \Theta\big(f(n)\big) \;\; se \;\; C_1 f(n) \leq g(n) \leq C_2 f(n) \; \forall \; n \geq n_0$$

- g(n) está “entre **ou é igual**” a C_1f(n) e C_2f(n).

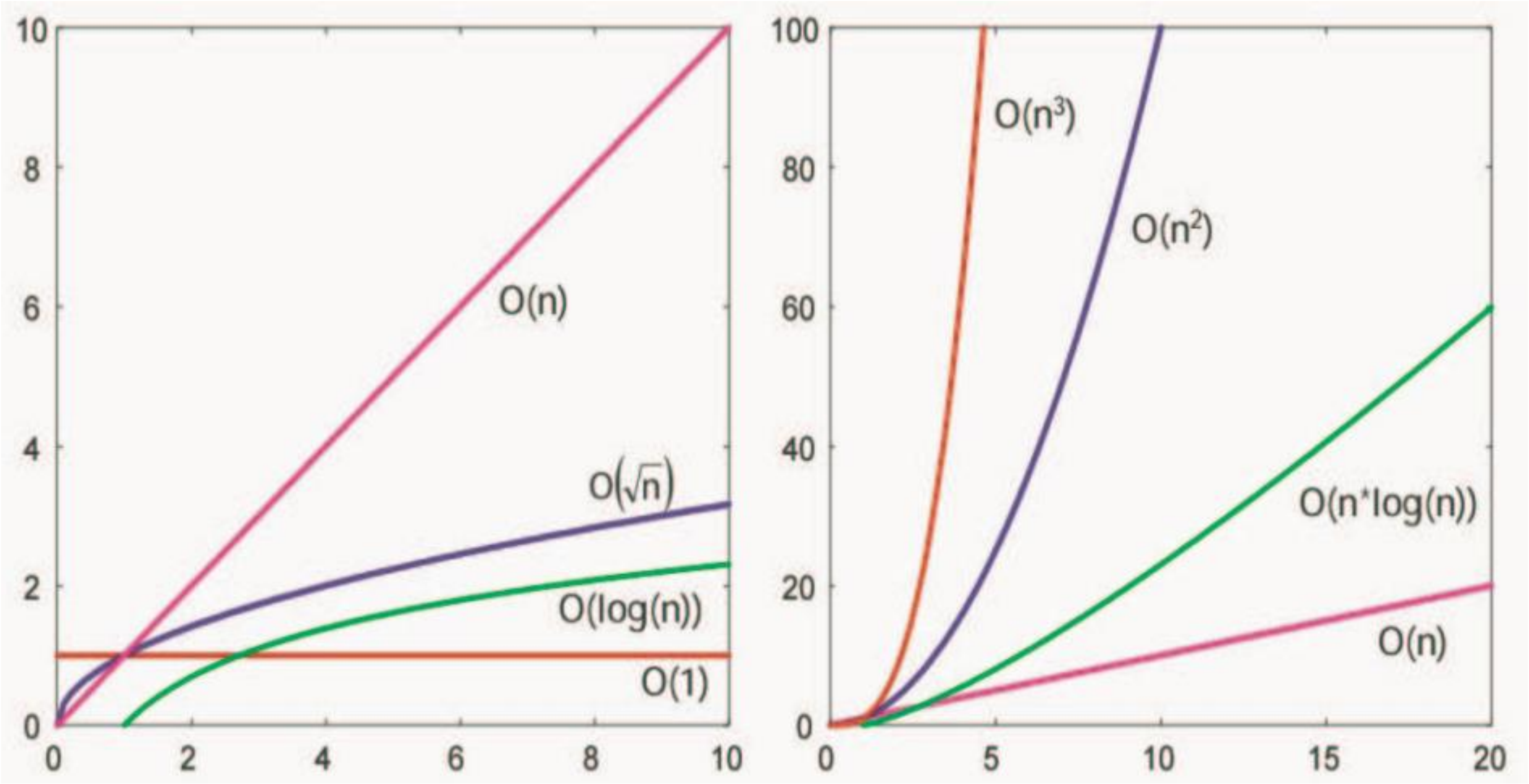
Notação assintótica

- **θ(f(n)) – Small Theta:**
  - Similar a Big Θ, mas não "encosta" nas funções.
- Considere:
  - Funções g(n) e f(n);
  - Uma constante C;

$$g(n) \in \Theta\big(f(n)\big) \quad se \quad C_1 f(n) < g(n) < C_2 f(n)\ \forall\, n \geq n_0$$

- g(n) está "entre" C_1f(n) e C_2f(n).

Notação assintótica



Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $2n^2 + 3n + 4 \leq C * n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $2n^2 + 3n + 4 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \leq C * n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $2n^2 + 3n + 4 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq (2 + 3 + 4) * n^2 \leq C * n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $2n^2 + 3n + 4 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq (2 + 3 + 4) * n^2 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 9 * n^2 \leq C * n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $2n^2 + 3n + 4 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq (2 + 3 + 4) * n^2 \leq C * n^2$
  - $2n^2 + 3n + 4 \leq 9 * n^2 \leq C * n^2$
  - Portanto C = 9

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $Para\ C = 9, determinar\ o\ n_0$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $Para\ C = 9, determinar\ o\ n_0$
  - $2n^2 + 3n + 4 = 9n^2$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $Para\ C = 9, determinar\ o\ n_0$
  - $2n^2 + 3n + 4 = 9n^2$
  - $2n^2 - 9n^2 + 3n + 4 = 0$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $Para\ C = 9, determinar\ o\ n_0$
  - $2n^2 + 3n + 4 = 9n^2$
  - $2n^2 - 9n^2 + 3n + 4 = 0$
  - $-7n^2 + 3n + 4 = 0$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - $Para\ C = 9, determinar\ o\ n_0$
  - $2n^2 + 3n + 4 = 9n^2$
  - $2n^2 - 9n^2 + 3n + 4 = 0$
  - $-7n^2 + 3n + 4 = 0$
  - Por Bhaskara: $\boldsymbol{n_1} = \mathbf{1}\ e\ n_2 = {}^{-8}/_{14}$

Notação Assintótica

- Mostre que f(n) = O(g(n)), sendo:
  - $f(n) = 2n^2 + 3n + 4$
  - $g(n) = n^2$

  - **f(n) = O(g(n))** $\boldsymbol{Para\ C = 9\ e\ n_0 = 1}$

Complexidade de funções recursivas

- Funções recursivas possuem a si mesma na definição.
- Métodos:
  - Iteração/Expansão;
  - Substituição;
  - Teorema Mestre.

Complexidade de funções recursivas

- **Iteração/Expansão:** calcular a complexidade para cada chamada, até generalizar para um k = 0.
- $C(n) = C(n-1) + 4$
- $C(n-1) = C(n-1-1) + 4 + 4$
- $C(n-1-1) = C(n-1-1-1) + 4 + 4 + 4$
- $C(n-k) = C(n-k-1) + 4*k$
- Para $k = n$
- $C(n-n) = C(0) + 4n$

Complexidade de funções recursivas

- **Iteração/Expansão:** calcular a complexidade para cada chamada, até generalizar para um k = 0.
- $C(n) = C(n-1) + 4$
- $C(n-1) = C(n-1-1) + 4 + 4$
- $C(n-1-1) = C(n-1-1-1) + 4 + 4 + 4$
- $C(n-k) = C(n-k-1) + 4*k$
- Para $k = n$
- $C(n-n) = C(0) + 4n$ → **Adicionar uma constante X para o critério de parada**

Complexidade de funções recursivas

- **Iteração/Expansão:** calcular a complexidade para cada chamada, até generalizar para um k = 0.
- $C(n) = C(n-1) + 4$
- $C(n-1) = C(n-1-1) + 4 + 4$
- $C(n-1-1) = C(n-1-1-1) + 4 + 4 + 4$
- $C(n-k) = C(n-k-1) + 4*k$
- Para $k = n$
- $C(n-n) = C(0) + 4n$ **portanto $f(n) = 4n + x$**

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = C(n-1) + 4$

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = C(n-1) + 4$
- Pressupor $C(n) = 4n + 2$

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = C(n-1) + 4$
- Pressupor $C(n) = 4n + 2$
- Aplicar $C(n-1) = 4(n-1) + 2$

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = C(n-1) + 4$
- Pressupor $C(n) = 4n + 2$
- Aplicar $C(n-1) = 4(n-1) + 2 = \mathbf{4n - 2}$

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = \mathbf{C(n-1)} + 4$
- Pressupor $C(n) = 4n + 2$
- Aplicar $\mathbf{C(n-1)} = 4(n-1) + 2 = \mathbf{4n - 2}$
- Testar em $C(n) = (\mathbf{4n-2}) + 4$

Complexidade de funções recursivas

- **Substituição:** pressupor uma solução para a iteração n. Aplicar para n-1. Testar em n.
- $C(n) = C(n-1) + 4$
- Pressupor $C(n) = \mathbf{4n + 2}$
- Aplicar $C(n-1) = 4(n-1) + 2 = 4n - 2$
- Testar em $C(n) = (4n-2) + 4$
- $C(n) = \mathbf{4n + 2}$

Complexidade de funções recursivas

- **Teorema Mestre:** Para funções recursivas que executam em uma taxa **n/b.**
- Funções expressas como:
  - $C(n) = aC(n/b) + f(n)$
  - a = constante de execução;
  - n/b = taxa de chamada recursiva;
  - f(n) = custo interno da função;
- Define regras para encontrar a faixa restrita por Big Theta.

Complexidade de funções recursivas

- **Teorema Mestre:** C(n) = aC(n/b) + f(n)

1. $Se\ f(n) < n^{\frac{\log(a)}{\log(b)}}$ então $\mathrm{C(n)} \in \Theta(n^{\log_b a})$
2. $Se\ f(n) = n^{\frac{\log(a)}{\log(b)}}$ então $\mathrm{C(n)} \in \Theta(n^{\log_b a} \log n)$
3. $Se\ f(n) > n^{\frac{\log(a)}{\log(b)}}$ então $\mathrm{C(n)} \in \Theta(f(n))$

Exercício

- Mostre se f(n) = O(g(n)), sendo:
  - $f(n) = 3n + 4$
  - $g(n) = n$

Exercício 1

- Determine a função de complexidade do código abaixo:

```
#include <stdio.h>
int main(void)
{
    int a=2, b=3, c=5, d=9;
    float x;
    if(!(d>5))
        x=(a+b)*d;
    else
        x=(a-b)/c;
    printf("%f", x);
    return 0;
}
```

Exercício 2

- Determine a função de complexidade do melhor e do pior caso do código abaixo:

```c
#include <stdio.h>
int main(void)
{
    int A, B, C, D;
    scanf("%d %d %d %d", &A, &B, &C, &D);

    float X=0;
    if(A>0)
    {
        if(A<10)
            X = 10/A;
    }
    else if(B>=10 && C<20)
        X = B+C/5;
    else if(D!=5)
        X = 4;
    else
        X = C+!D;
    printf("%f", X);
    return 0;
}
```

Exercício 3

- Determine a função de complexidade do melhor e do pior caso do código abaixo:

```
int BuscaSequencial(int* vetor, int tam, int chave)
{
    int i;
    for(i = 0; i < tam; i++)
    {
        if(vetor[i] == chave)
        {
            return i; //retornar índice ou ponteiro
        }
    }
    return -1;//não encontrou indice;
}
```

Exercício 4

- Determine a função de complexidade do melhor e do pior caso do código abaixo:

```
int func(int n)
{
    int i, j, resultado = 0;

    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            resultado += j;
        }
    }
    return resultado;
}
```

Exercício 5

- Determine a função de complexidade do código abaixo:

```
int func(int n)
{
    int i, j, resultado = 0;

    for(i = 0; i < n; i+=3)
    {
        for(j = 0; j < n; j+=2)
        {
            resultado += j;
        }
    }
    return resultado;
}
```

Exercício 6

- Determine a função de complexidade do código abaixo:

```
int func(int n)
{
    int i, resultado = 0;

    for(i = 1; i <= n; i*=2)
    {
        resultado += i;
    }
    return resultado;
}
```