Complexidade de algoritmos

emanoelim@utfpr.edu.br

Análise da complexidade de algoritmos

- Existem diferentes maneiras de resolver o mesmo problema, ou seja, diferentes algoritmos podem ser utilizados.
- Diante de várias possibilidades é importante analisar qual algoritmo é mais eficiente na resolução de um determinado problema.
- É aí que entra a **análise da complexidade de algoritmos**.

Análise da complexidade de algoritmos

- A análise da complexidade de algoritmos estuda o comportamento dos algoritmos para definir sua eficiência.
- Os dois aspectos mais analisados são o tempo de execução (complexidade de tempo) e a memória utilizada (complexidade de espaço).
- Primeiramente vamos analisar um algoritmo quanto ao tempo de execução.

Formas de analisar a complexidade de um algoritmo

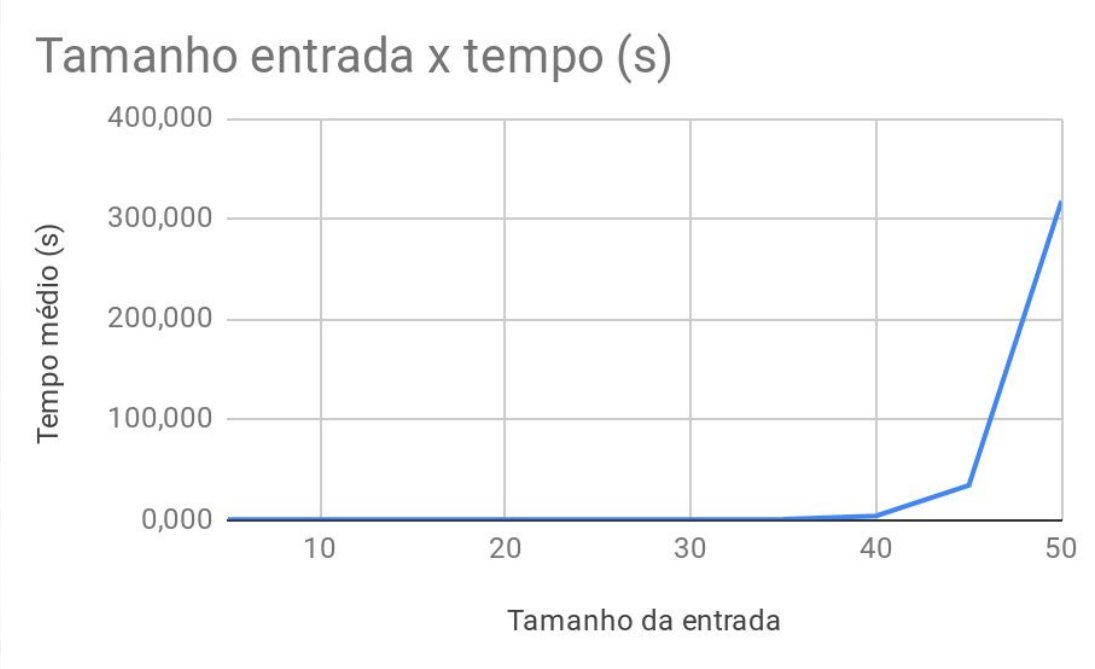
- **Experimental/empírica:**

  Executar o algoritmo mediante diversos tamanhos de entrada e tomar nota do tempo gasto para executar cada entrada, analisando os resultados posteriormente (montando gráficos, por exemplo).

Formas de analisar a complexidade de um algoritmo

Entrada	Tempo 1 (s)	Tempo 2 (s)	Tempo 3 (s)	Tempo médio (s)
5	0,031	0,031	0,031	0,031
10	0,031	0,031	0,047	0,036
15	0,031	0,031	0,031	0,031
20	0,063	0,031	0,047	0,047
25	0,031	0,033	0,047	0,037
30	0,063	0,078	0,069	0,070
35	0,423	0,536	0,357	0,439
40	3,258	3,836	4,124	3,739
45	27,408	41,387	33,913	34,236
50	293,622	347,575	311,999	317,732

Formas de analisar a complexidade de um algoritmo



Formas de analisar a complexidade de um algoritmo

Este tipo de análise vai variar conforme as configurações do computador utilizado, conforme a linguagem de programação utilizada, etc.

Formas de analisar a complexidade de um algoritmo

- **Método matemático**

  Esta forma é mais genérica. Conseguimos ter uma análise invariante às configurações do hardware ou linguagem utilizada. Depende somente da forma como a algoritmo foi construído.

Custo de instruções

Instruções simples (executadas diretamente pelo CPU (ou muito próximo disso) terão custo 1 (uma unidade de tempo):

- Atribuição de valores;
- Acesso a um elemento de um vetor via índice;
- Comparação entre dois valores;
- Operações matemáticas básicas: soma, multiplicação…

Custo de instruções

IF: o comando “If” não tem custo, o que tem custo é a condição do if. O if abaixo, por exemplo, tem somente uma comparação, ou seja, seu custo é 1.

```
if(x > 0)
```

O próximo if é composto por duas comparações, tendo custo 2:

```
if(x > 0 && x < 10)
```

Custo de instruções

FOR: segue a mesma ideia do if.

```
...
for(i = 0; i < n; i++) {
    ...
```

Mesmo que n seja 0, duas instruções serão executadas, a instrução i = 0 e a comparação i < n, gerando um custo de 2 unidades de tempo.

Custo de instruções

Se n não for igual o zero:

- A instrução i = 0 será executada uma vez.
- A instrução de incremento i++ será executada n vezes.
- A instrução de comparação i < n será executada n + 1 vezes.

Custo de instruções

- Para exemplificar considere n = 4;

para i = 0: **i < n** -> sim -> **i++**

para i = 1: **i < n** -> sim -> **i++**

para i = 2: **i < n** -> sim -> **i++**

para i = 3: **i < n** -> sim -> **i++**

para i = 4: **i < n** -> não -> sai do laço

Custo de instruções

Então, este for (sem contar as instruções que possam estar dentro do for) terá um custo de:

$f(n) = 1 + n + (n + 1)$

$f(n) = 2 + 2n$

Custo de instruções

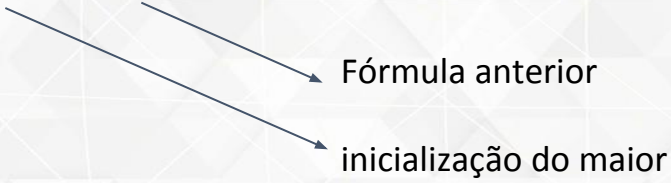
Considere agora um código para achar o maior número em um vetor:

```
...
    int maior = v[0];
    for(i = 0; i < n; i++) {
        if(v[i] > maior) {
            m = v[i];
        }
    }
...
```

Custo de instruções

Antes de considerarmos as instruções de dentro do for, temos:

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



Custo de instruções

Considerando as instruções dentro do for:

- Dentro do for existe um if, ou seja, a execução das instruções pode acontecer ou não.
- Como definir o custo nesse caso?
- Vamos considerar duas situações.

Custo de instruções

- 1ª - queremos achar o maior item no vetor [5, 4, 3, 2, 1] (melhor caso)
- 2ª - queremos achar o maior item no vetor [1, 2, 3, 4, 5] (pior caso)

Custo de instruções

Na primeira situação, o resultado do if será sempre falso, então a atribuição m = v[i] nunca será executada. Logo, cada iteração do for executa apenas uma instrução que é a própria comparação. Ou seja, a comparação é executada n vezes. Assim temos:

$f(n) = 3 + 2n + n$

$f(n) = 3 + 3n$

Custo de instruções

Na segunda situação, o resultado do if sempre será verdadeiro, então atribuição m = v[i] sempre será executada. Logo, cada iteração do for executa duas operações (a comparação e também a atribuição). Ou seja, essas duas instruções são executada n vezes. Assim, temos:

$f(n) = 3 + 2n + 2n$

$f(n) = 3 + 4n$

Custo de instruções

Quando escrevemos um algoritmo, nem sempre sabemos qual será nossa entrada, então, supomos que é possível que o pior caso aconteça. Assim, ao analisar a complexidade de um algoritmo, geralmente consideramos o pior caso.

Comportamento assintótico

Será que todos os termos de uma função são necessários para analisar o comportamento de um algoritmo conforme o tamanho da entrada cresce?

Comportamento assintótico

No caso anterior, por exemplo:

- inicialização do maior;
- a inicialização do for;
- a primeira comparação do for.

sempre serão feitas, independente do tamanho da entrada. Ou seja, elas terão um custo constante para o algoritmo, não importa o tamanho de n.

Comportamento assintótico

Assim, na função:

$f(n) = 3 + 4n$

podemos descartar o que tiver custo constante, sobrando:

$f(n) = 4n$

Comportamento assintótico

Todas as constantes que multiplicam o termo “n” também podem ser descartadas, uma vez que queremos uma análise genérica, independente da linguagem de programação.

Comportamento assintótico

A seguinte instrução em C:

```
maior = v[i];
```

É escrita da seguinte forma na linguagem Pascal:

```
maior := v[i];
```

Que em C seria o mesmo que fazer:

```
if(i >= 0 && i <= n)
    maior = v[i]
```

Comportamento assintótico

Ou seja: em C teríamos 1 instrução para fazer a atribuição. Em Pascal teríamos 3 instruções. Assim, desconsiderando os aspectos de linguagem, e considerando apenas a ideia do algoritmo, a função do exemplo anterior pode ser reduzida a:

$f(n) = n$

Comportamento assintótico

Se a função possuir diversos termos envolvendo “n”, por exemplo:

$f(n) = n^2 + n$

Podemos considerar apenas o termo com o maior expoente, pois sempre será ele o responsável pelo maior crescimento da função. Assim, no exemplo acima, teríamos:

$f(n) = n^2$

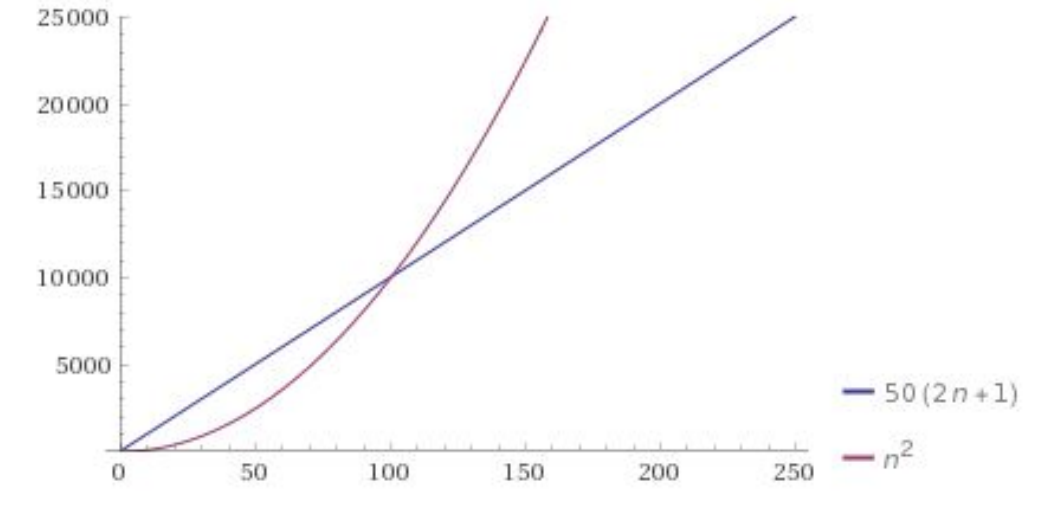
Comportamento assintótico

Considere as duas funções:

- $f(n) = 100n + 50$
- $g(n) = n^2$

Apesar de inicialmente f(n) gerar valores maiores, existe um valor de n, a partir do qual g(n) passa a gerar resultados sempre maiores:

Comportamento assintótico



Assim fica claro ver, que a função ou termo com o maior expoente, a partir de um certo ponto, sempre irá ser o responsável pelo maior crescimento.

Comportamento assintótico

Qual a comportamento assintótico das funções abaixo?

a) $f(n) = 5$
b) $f(n) = 3n + n + 2$
c) $n^2 + n$
d) $5n^3 + n^2 + n + 20$
e) $200 + n$

Comportamento assintótico

De modo geral, para saber a complexidade de um programa simples (sem recursividade) devemos analisar os laços aninhados:

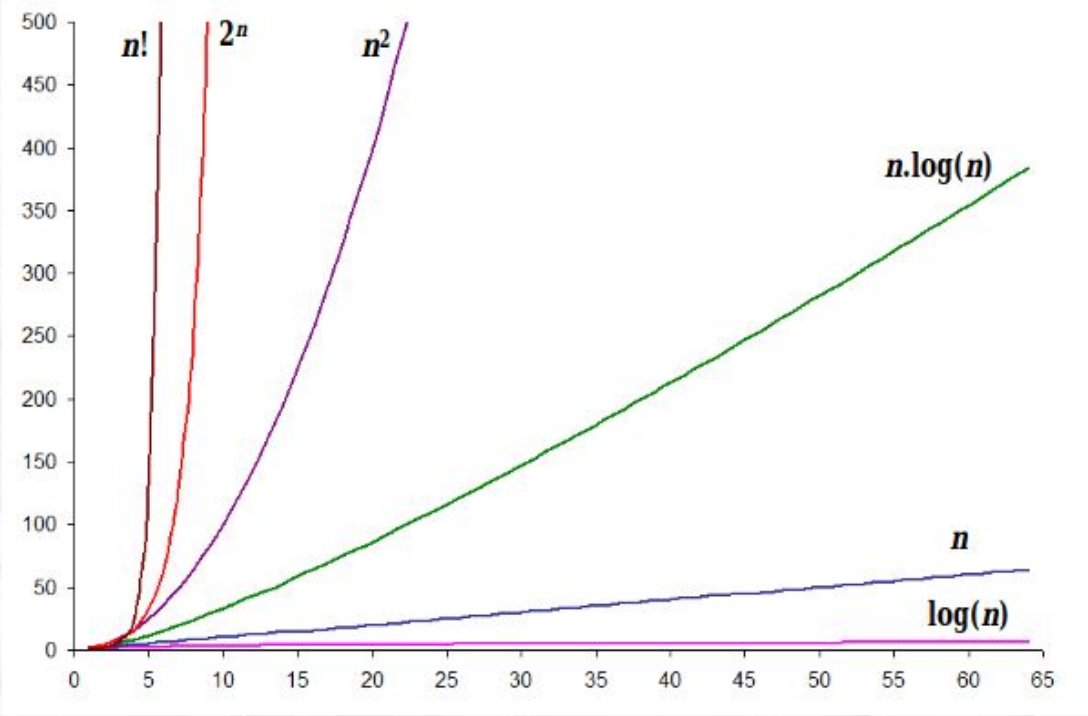
- Programa sem laço: f(n) = constante = 1
- Programa com 1 laço: f(n) = n
- Programa com 2 laços aninhados: f(n) = n^2, e assim por diante.

Obs.: se o programa ou função analisada chamar outras funções, essas funções também devem ser analisadas e devem entrar na conta!

Comparação assintótica de funções

Se uma função f(x) tem um crescimento maior do que uma função f(x), dizemos que a função f(x) domina assintoticamente a função g(x). O gráfico a seguir mostra a curva de crescimento das funções mais comuns:

Comparação assintótica de funções



Comparação assintótica de funções

Hierarquia de funções do ponto de vista assintótico:

$$c < \log(n) < n < n \log(n) < n^2 < 2^n < n!$$

c = constante

Notações de complexidade

A notação mais usada para medir a complexidade de algoritmos é a Big-O. Ela define qual é o limite superior de uma função, ou seja, qual será seu custo no pior caso. Por exemplo:

$f(n) = n^2 + n + 1$, **$O(n) = n^2$**

quer dizer que o pior caso, o algoritmo terá um custo de n^2, nunca maior que isso.

Notação O: valores comuns

f(n) = O(1):

- Complexidade constante;
- Instruções executadas um número fixo de vezes, independente do tamanho da entrada.

Notação O: valores comuns

f(n) = O(log n):

- Complexidade logarítmica;
- Tipicamente são algoritmos que resolvem um problema dividindo-o em problemas menores.

Notação O: valores comuns

f(n) = O(n):

- Complexidade linear;
- Em geral, uma quantidade definida de operações é realizada sobre cada item da entrada;
- No caso do uso de vetores, por exemplo (achar o maior, buscar um item, etc).

Notação O: valores comuns

f(n) = O(n log n):

- Complexidade logarítmica;
- Tipicamente são algoritmos que resolvem um problema dividindo-o em problemas menores, resolvendo cada um independentemente e depois juntando as soluções.

Notação O: valores comuns

$f(n) = O(n^2)$

- Complexidade quadrática;
- Caracterizam-se pelo processamento dos dados em pares, geralmente contém um aninhamento com dois comandos de repetição;

Notação O: valores comuns

$f(n) = O(n^3)$

- Complexidade cúbica;
- Caracterizam-se pela presença de 3 comandos de repetição aninhados;
- Já começa a se tornar pouco eficiente para problemas grandes.

Notação O: valores comuns

f(n) = O(2^n)

- Complexidade exponencial;
- Geralmente soluções que usam força bruta;
  - Ex.: quebrar uma senha de 8 caracteres tentando todas as combinações possíveis;
- Não são úteis do ponto de vista prático;

Notação O: valores comuns

f(n) = O(n!)

- Complexidade exponencial;
- Geralmente soluções que usam força bruta;
  - n * (n - 1) * (n - 2) * ...
- Não são úteis do ponto de vista prático.

Notação O: valores comuns

- Link com as complexidades dos algoritmos mais comuns: http://bigocheatsheet.com/

Exercícios

1. Quais seriam o melhor caso e o pior caso para uma função com objetivo de remover um item de uma lista implementada por meio de vetores? Qual a complexidade no melhor caso? Qual a complexidade no pior caso?
2. Qual a complexidade de uma função que realiza a remoção do primeiro item de uma lista encadeada?
3. Qual a complexidade de uma função que realiza a remoção do último item de uma lista encadeada (sem usar um ponteiro para o último e usando um ponteiro para último)?

Exercícios

4. Escreva uma função que busca uma determinada chave dentro de uma lista e remove o item que contém esta chave (considere a implementação por vetores). Qual é a complexidade da função?
5. Dois algoritmos (A e B) possuem complexidade n^5 e 2^n, respectivamente. Você utilizaria o algoritmo B ao invés do A em algum caso? Cite exemplos.