Noções de Complexidade
Contagem de Operações

ALGORITMOS E ESTRUTURA DE DADOS I

Profa. Andréa Aparecida Konzen
Escola Politécnica - PUCRS

Baseado no material cedido pela professora Isabel Manssour e nas referências bibliográficas

O que vamos ver?

- Complexidade e análise de algoritmos
- Contando o tempo
- Contagem de operações
- Funções

Complexidade e Análise de Algoritmos

- No desenvolvimento de uma aplicação tem-se como objetivo projetar “boas” estruturas de dados e “bons” algoritmos
  - Otimizados
  - Simples

Como saber se um algoritmo é eficiente?

Contxtualização

- Qual o melhor código?
- O que é um bom código?

Legível

Escalável (*entrada x tempo*)

Cenário com tempo de execução (*o computador do desenvolvedor é diferente do dispositivo que executará efetivamente o seu código*) ...

Complexidade e Análise de Algoritmos

Computadores modernos são rápidos e cada vez estão mais rápidos...

Por que se preocupar com códigos eficientes?

*A questão é que os **data sets** também estão cada vez maiores*
(ex. 2014 o Google indexava 30.000.000.000.000 páginas - 100.000.000 Gb)
Quanto tempo levaria para fazer uma busca utilizando força bruta?

Fonte: *MIT 6.0001 course*

Soluções simples podem não escalar de forma aceitável...

Como podemos decidir qual opção de programa é mais eficiente?

Complexidade e Análise de Algoritmos

Análise de Algoritmos:

Estudo das características de desempenho de um determinado algoritmo.

- O <u>espaço ocupado</u> é uma característica de desempenho.
- <u>O tempo gasto</u> na execução é outra característica de desempenho.

Complexidade e Análise de Algoritmos

A **complexidade de um algoritmo** é a **medida do consumo de <u>recursos</u>** de que o algoritmo necessita durante a sua execução

- Tempo de processamento;
- Memória ocupada;
- Largura de banda de comunicação;
- Hardware necessário;
- etc.

Contando o tempo

Como podemos medir a eficiência de programas?

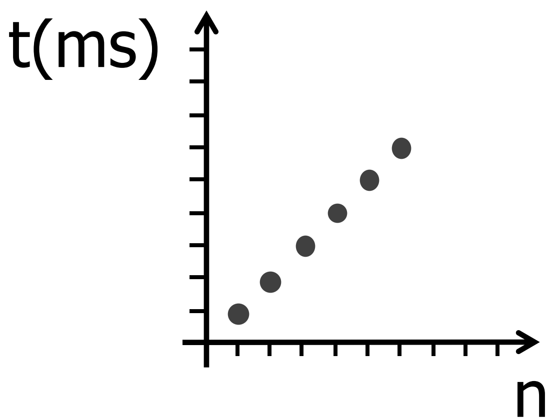
- Medir com um **timer**
- **Contar** o número de operações
- Noção abstrata de **ordem de crescimento**

vamos identificar porque esta é a melhor forma de medirmos o impacto das escolhas de algoritmos utilizadas para resolver um problema; e a forma de inferir a dificuldade inerente de um problema

Contando o tempo

Tempo de processamento

- Depende de uma série de fatores: hardware, software, tamanho e tipo da entrada de dados
- Algoritmo: tempo de execução X entrada de dados



Contando o tempo

- Para contar o tempo em Java
  - Método *System.currentTimeMillis()* retorna o tempo em ms de um determinado instante.

```java
long antes = System.currentTimeMillis();
// executa algoritmo...
long depois = System.currentTimeMillis();
```

- Subtraindo *antes* de *depois*, tem-se uma estimativa do tempo que a execução levou.

```java
long total = depois - antes;
```

- Exemplo: tempo de execução para algoritmo *bubble sort*

Contando o tempo

Exemplo 2: busca sequencial em um arranjo

```java
public static int localiza(int[] dados, int valor) {
    for(int pos=0; pos<dados.length; pos++)
          if(valor == dados[pos])
                return pos;
     return -1;
}
```

- O método recebe um arranjo e um valor a ser localizado, e retorna a posição do valor no arranjo (ou -1 se não achar).

Contando o tempo

Para "enxergar" a complexidade, deve-se executar o método várias vezes, com um arranjo cada vez maior:

```
public static void main(String[] args) {
    int[] lista;
    for(int total=1_000_000; total<8_000_000; total+=10000){
        lista = new int[total];
        for(int pos=0; pos<total; pos++)//preenche o arranjo
            lista[pos] = pos;

        long antes = System.currentTimeMillis();
        int loc = localiza(lista, total-1);//pior caso:último elem
        long depois = System.currentTimeMillis();

        long tempo = depois - antes;
        System.out.println(total + " " + tempo); // saída
                                    // total de elementos x tempo
    }
}
```

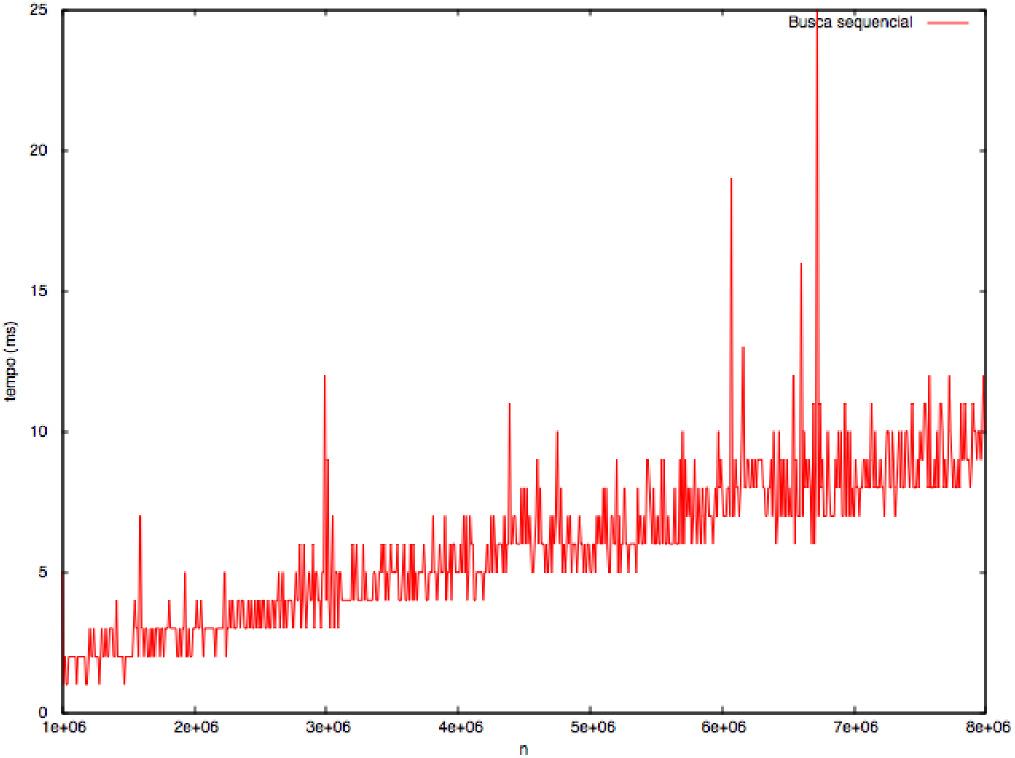
Contando o tempo

- A partir da saída do programa:

```
1000000 5
1010000 1
1020000 2
...
```

Qual a origem da variação no tempo de execução?

Gráfico da quantidade de dados de entrada x tempo

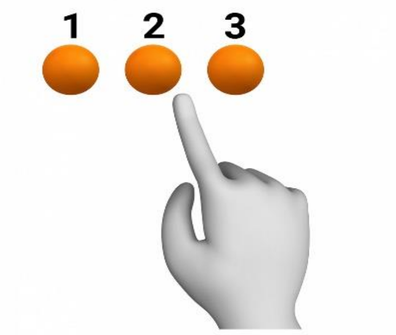


Contando operações

- Análise da eficiência de um algoritmo
  - Medir o tempo depende de hardware e software (sistema operacional, por exemplo)
  - Alternativa?

    Contar o número de operações

    Exemplo: atribuição, operação aritmética, comparação, etc.



Contagem de operações

- Análise de algoritmos
  - Não pode considerar o tempo de execução
  - Deve ser feita diretamente sobre o pseudocódigo de alto nível
  - *Consiste em contar quantas* ***operações primitivas*** *são executadas*
    - Operação primitiva: instrução de baixo nível com um tempo de execução constante
  - Assume-se que os tempos de execução de operações primitivas diferentes são similares

Contagem de operações

- Operações primitivas
  - Atribuição de valores a variáveis
  - Chamadas de métodos
  - Operações aritméticas (por exemplo, adição de dois números)
  - Comparação de dois números
  - Acesso a um arranjo
  - Retorno de um método

Contagem de operações

- Exemplo 1:
  - Contar o número de operações para atribuir para cada posição v[i] de um arranjo unidimensional o resultado de i*2

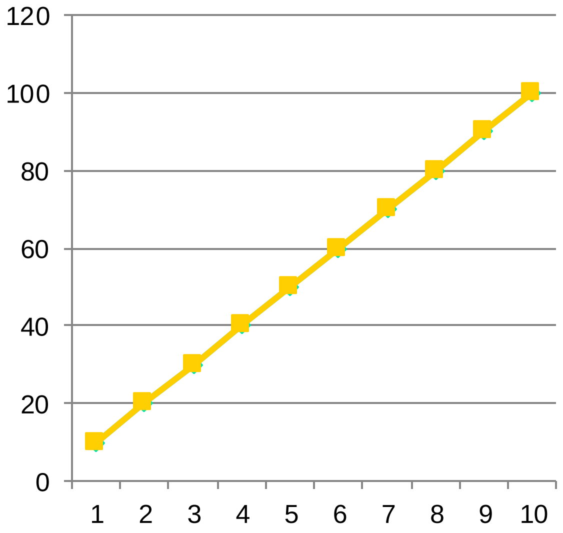
```
v[0..10] : inteiro
for (i = 0; i < v.comprimento; i++)
        v[i] = i * 2
```

Operação (multiplicação, atribuição e acesso as posições do arranjo): n vezes (n=10)

Contagem de operações

- Exemplo 1:
  - Contar o número de operações para atribuir para cada posição v[i] de um arranjo unidimensional o resultado de i*2

v[0..10] : inteiro
for (i = 0; i < v.comprimento; i++)
v[i] = i * 2



Contagem de operações

- Exemplo 2:
  - Contar o número de operações para atribuir para cada posição m[i,j] de um arranjo bidimensional o resultado de i*j

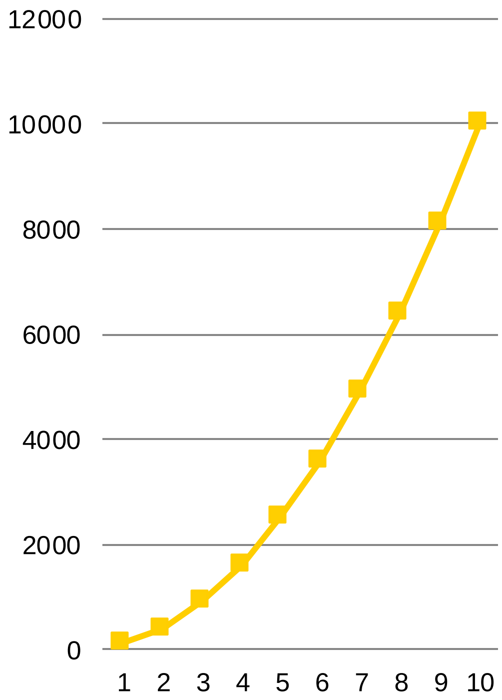
```
m[0..10][0..10] : inteiro
for (i=0; i<m.comprimento; i++)
        for (j=0; j<m[i].comprimento; j++)
                m[i][j] = i * j
```

Operação (multiplicação, atribuição e acesso as posições do arranjo): n*n vezes (n=10)

Contagem de operações

- Exemplo 2:
  - Contar o número de operações para atribuir para cada posição m[i,j] de um arranjo bidimensional o resultado de i*j

```
m[0..10][0..10] : inteiro
for (i=0; i<m.comprimento; i++)
        for (j=0; j<m[i].comprimento; j++)
                m[i][j] = i * j
```



Contagem de operações

Exercícios: implementar e contar o número de operações das funções listadas a seguir.

```
int f1(n)
    r=0
    for (i=1; i<n; i++)
        r = r + 1
    return r
```

```
int f2(n)
    r=0
    for (i=1; i<n; i++)
        for (j=i+1; j<n; j++)
            r = r + 2
    return r
```

Contagem de operações

```
int f3(n)
    cont=0
    for (i=1; i<n; i++)
            for (j=1; j<n; j++)
                    print i*j
                    cont++
    return  cont
```

```
int f4(n)
    r=0
    for (i=1; i<n; i++)
            for (j=i; j<2*i; j++)
                    for (k=i; k<j; k++)
                            r = r + 1
    return r
```

Contagem de operações

```
int f5(n)
    r=0
    for (i=1; i<n; i++)
            for (j=i; j<i+3; j++)
                    for (k=i; k<j; k++)
                            r = r + 1
    return r

                                int f6(n)
                                    if (n==0)
                                            return 1
                                    else
                                            return  f6(n-1) + f6(n-1)
```

Funções

- Uma **classe de complexidade** é uma forma de agrupar algoritmos que apresentam complexidade similar. Por exemplo:

  - Complexidade **constante**: o algoritmo sempre ocupa a mesma quantidade de recursos.

  - Complexidade **linear**: o algoritmo consome recursos de forma diretamente proporcional ao tamanho do problema.

Funções

- Sete funções mais comuns usadas em análise de algoritmos:
  - Constante: 1
  - Logaritmo: $\log n$
  - Linear: n
  - n-log-n: $n \log n$
  - Quadrática: n^2
  - Cúbica: n^3
  - Exponencial: a^n

Funções

- **Constante: 1**
  - Função mais simples
  - f(n) = c
  - Não importa o valor de n, sempre será igual ao valor da constante c

  - **Exemplo:** função que recebe um arranjo de inteiros e retorna o valor do primeiro elemento multiplicado por 2

Funções

- **Logaritmo: log *n***
  - log base 2
  - O número de operações realizadas para solução do problema não cresce da mesma forma que n
    - Se dobra o valor de n, o incremento do consumo é bem menor
  - **Exemplo:** conversão de número decimal para binário e busca binária (*binary search*)

Funções

- **Linear: n**
  - Se dobra o valor de n, dobra o consumo de recursos
  - f(n) = n

  - **Exemplo:** localizar um elemento em uma lista.

Funções

- **n-log-n: *n* log *n***
  - f(n) = n log n
  - Atribui para uma entrada n o valor de n multiplicado pelo logaritmo de base 2 de n
  - Cresce mais rápido que a função linear e mais devagar que a função quadrática

  - **Exemplo:** Algoritmos de ordenação *mergesort* e *heapsort*

http://www.sorting-algorithms.com/

Funções

- **Quadrática:** ***n*²**
  - Função polinomial com expoente 2
  - $f(n) = n^2$
  - Não cresce de forma abrupta,mas dificultam o uso em problemas grandes.

  - **Exemplo:** Ordenação com o algoritmo *bubblesort*

http://www.sorting-algorithms.com/

Funções

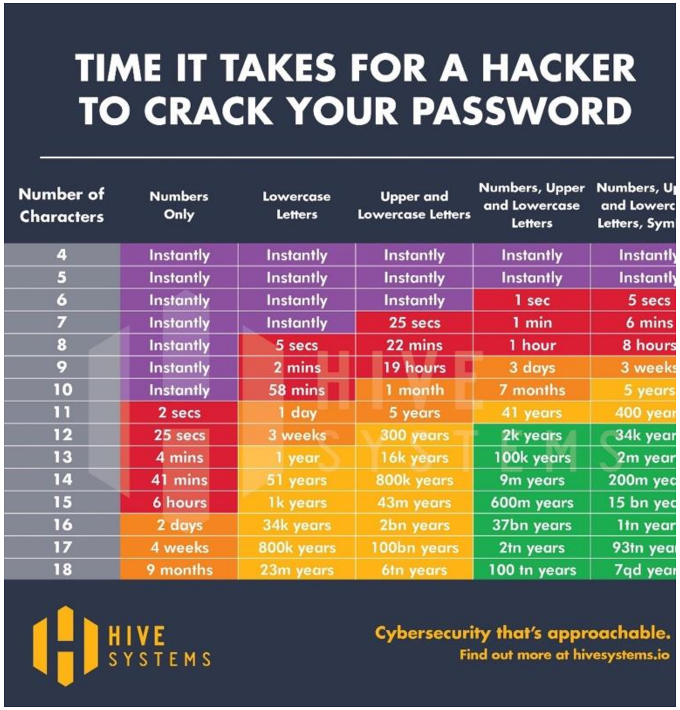
- **Cúbica:** $\boldsymbol{n^3}$
  - Função polinomial com expoente 3
  - $f(n) = n^3$
  - Aparece com menos frequência na análise de algoritmos do que as funções constante, linear ou quadrática

  - **Exemplo:** multiplicar duas matrizes

Funções

- **Exponencial:** a^n
  - $f(n) = a^n$
  - Expoente é variável de acordo com a entrada de dados
  - São considerados algoritmos “ruins”, pois crescem abruptamente
  - Aplicáveis apenas em problemas pequenos.

  - **Exemplo:** quebrar senhas com força bruta

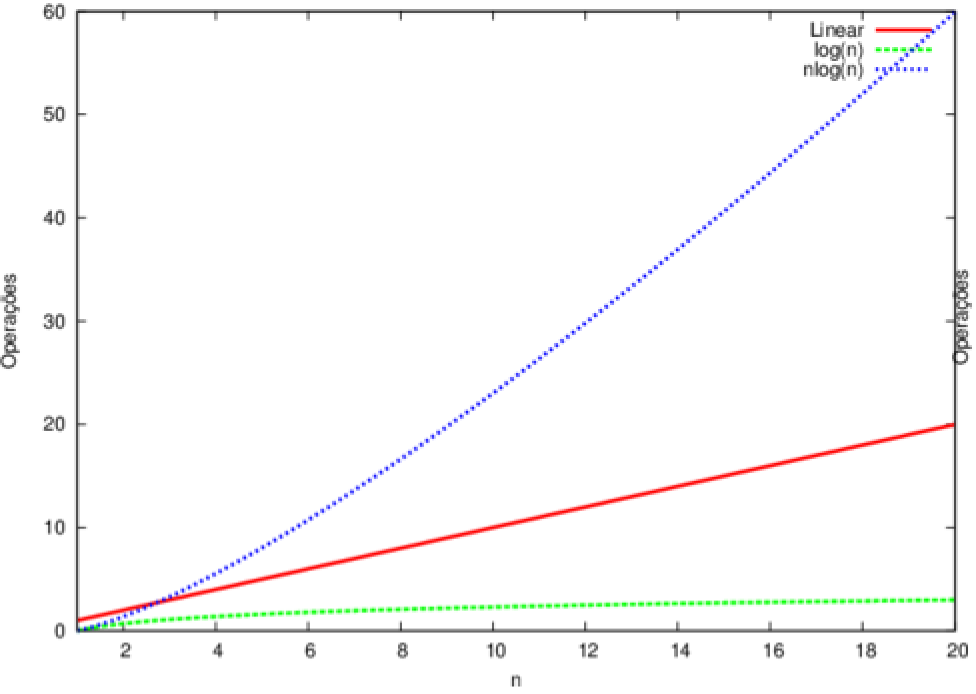


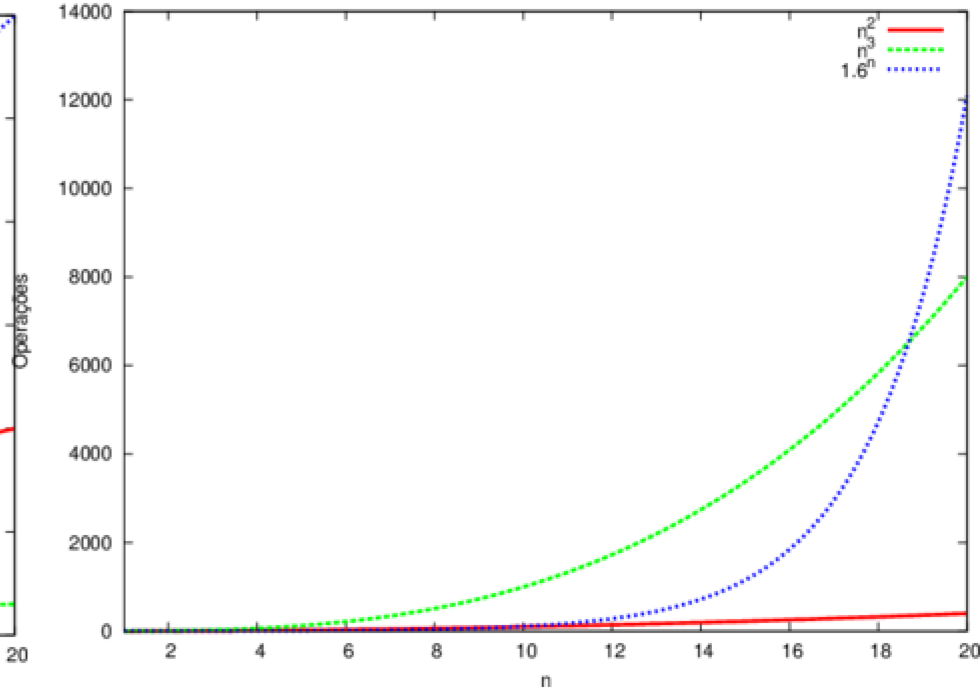
TIME IT TAKES FOR A HACKER TO CRACK YOUR PASSWORD

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Up and Lowerc Letters, Sym
4	Instantly	Instantly	Instantly	Instantly	Instantl
5	Instantly	Instantly	Instantly	Instantly	Instantl
6	Instantly	Instantly	Instantly	1 sec	5 secs
7	Instantly	Instantly	25 secs	1 min	6 mins
8	Instantly	5 secs	22 mins	1 hour	8 hours
9	Instantly	2 mins	19 hours	3 days	3 weeks
10	Instantly	58 mins	1 month	7 months	5 years
11	2 secs	1 day	5 years	41 years	400 year
12	25 secs	3 weeks	300 years	2k years	34k year
13	4 mins	1 year	16k years	100k years	2m year
14	41 mins	51 years	800k years	9m years	200m yea
15	6 hours	1k years	43m years	600m years	15 bn yea
16	2 days	34k years	2bn years	37bn years	1tn year
17	4 weeks	800k years	100bn years	2tn years	93tn year
18	9 months	23m years	6tn years	100 tn years	7qd year

Funções

- Taxas de crescimento para as funções usadas em análise de algoritmos





Funções

- Um problema é dividido em funções/métodos
  - Cada função/método tem um “custo” diferente
  - Estes custos são somados para determinar o custo total para solução do problema

Comparação

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Funções

- **Exercício**
  - Dois algoritmos para resolver o mesmo problema

    $f_1(n) = 2n^2 + 5n$ operações

    $f_2(n) = 500n + 4000$ operações

  - Considere o número de operações de cada um para diferentes valores de n (por exemplo, n=10 e n=1000)

  - **Qual é a melhor solução?**

Referências

- Livro do Goodrich, capítulo 4
- Livro do Cormen, capítulo 3

Importante a Leitura!!