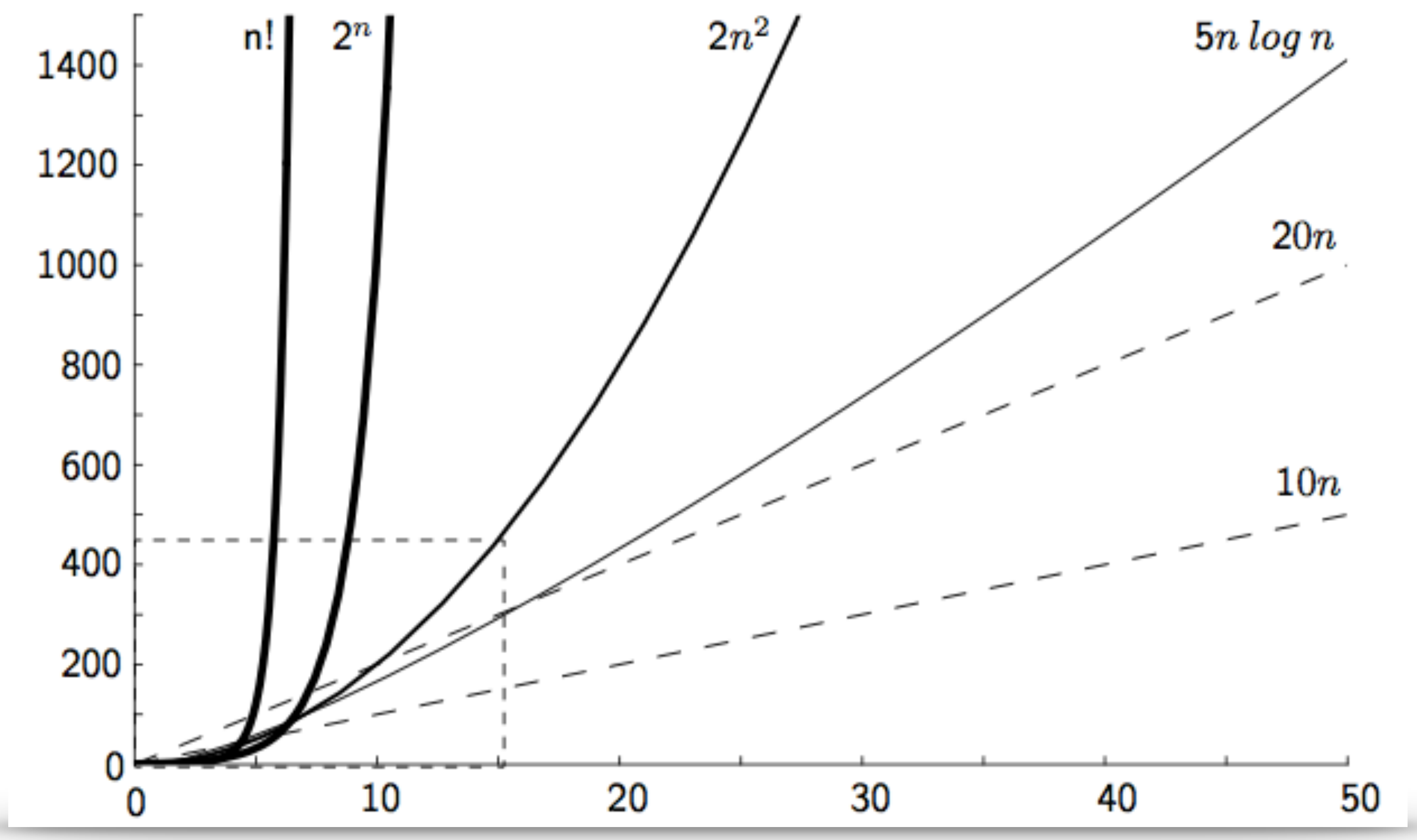
Profa. Dra. Raquel C. de Melo-Minardi
Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais



MÓDULO 3
COMPLEXIDADE DE ALGORITMOS
Função de complexidade – Parte VI

PESQUISA BINÁRIA

- Abrir a lista buscando o meio
- Caso o nome procurado esteja na primeira metade, cortar essa metade novamente buscando o meio e repetir o mesmo processo até encontrar a inicial do nome procurado e assim até encontrar o nome exato
- Esse mesmo processo pode ser utilizado para implementar um algoritmo de busca

PESQUISA BINÁRIA

- Informalmente, esse algoritmo seria como se segue:
  - Considere que os registros da lista sejam mantidos em **ordem crescente**
  - Compare a chave procurada com a chave da **posição do meio da lista**
  - Se a chave procurada for **menor** que a chave do meio, o registro estará então **na primeira metade da lista**
  - Se a chave procurada for **maior** que a chave do meio, o registro estará **na segunda metade da lista**
  - Repita o processo até que a chave seja **encontrada** ou que reste apenas um registro cuja chave não é a chave procurada, ou seja, a mesma **não se encontra na lista**

Desafio

1. Implemente esse algoritmo de pesquisa binária em uma lista
2. Você conseguiria nos dizer qual a função de complexidade da pesquisa binária em uma lista ordenada?

SOLUÇÃO

```
def pesquisaBinaria(reg, lista):
(I)     if len(lista) == 0: # Se a lista está vazia
                return -1
        esq = 0
        dir = len(lista)-1

        i = int((esq+dir)/2)
(II)    while esq <= dir and reg != lista[i]:
(III)           if reg > lista[i]:
                        esq = i + 1
                else:
                        dir = i - 1
                i = int((esq+dir)/2)
        if reg == lista[i]:
                return i
        else:
                return -1
```

SOLUÇÃO

- O código **I** é uma validação do conteúdo do arranjo que simplesmente encerra a função caso o arranjo seja vazio
- O laço **II** executa enquanto o registro procurado não for encontrado e a sub-lista sendo varrida ainda tiver tamanho maior ou igual a 1 `(esq <= dir and reg != lista[i])`
- Dentro desse laço, o condicional **III** subdivide virtualmente a lista de acordo com o local onde o registro poderá estar
    - Se o registro for maior que a chave atual, `esq` vai valer a posição atual mais 1 e se for menor, `dir` vai valer a posição atual menos 1
    - Note que dizemos que a divisão da lista é virtual pois, de fato, ela nunca é dividida ou copiado mas os índices `esq` e `dir` delimitam onde essa lista será varrida e a cada iteração é como se ela fosse dividida em duas

SOLUÇÃO

```
def pesquisaBinaria(reg, lista):
(I)     if len(lista) == 0: # Se a lista está vazia
            return -1
        esq = 0
        dir = len(lista)-1

        i = int((esq+dir)/2)
(II)    while esq <= dir and reg != lista[i]:
(III)       if reg > lista[i]:
                esq = i + 1
            else:
                dir = i - 1
            i = int((esq+dir)/2)
        if reg == lista[i]:
            return i
        else:
            return -1
```

SOLUÇÃO

- Por fim, o trecho **IV** apenas é uma verificação de se o registro foi encontrado ou não
  - Se não for encontrado, retorna o valor “-1” como preestabelecido

ANÁLISE DE COMPLEXIDADE

- Qual será então a função de complexidade desse algoritmo?
  - Note que o algoritmo sempre divide a entrada ao meio. Esse tipo de divisão nos leva a uma complexidade

$$f(n) = log_2(n)$$

  - A função *log* atenua os valores de entrada
  - $f(n) = log_2(n)$ significa que o tempo não cresce linearmente com o tamanho da entrada como ocorre na pesquisa sequencial, mas sim cresce proporcionalmente seu **logaritmo** na base *2* já que a **entrada é sempre dividida ao meio**

EXEMPLO

- Apenas para exemplificar o funcionamento da pesquisa binária, veja a lista abaixo na qual temos *13* elementos indexados de *0* a *12*

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	78	86	90	91	92	97	98

EXEMPLO

- Pesquisando pelo número *16*, a lista será dividida ao meio e *16* será comparado ao elemento do meio que é `lista[6] = 78`
    - Como *16* < *78* (em negrito), a segunda interação avaliará a metade da esquerda

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	33	35	42	61							

EXEMPLO

- Na segunda iteração a lista será dividida ao meio novamente e *16* será comparado ao elemento do meio que é `lista[2] = 33`
  - Como *16* < *33* (em negrito), a segunda interação avaliará a metade da esquerda

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	**33**	35	42	61							
0	31											

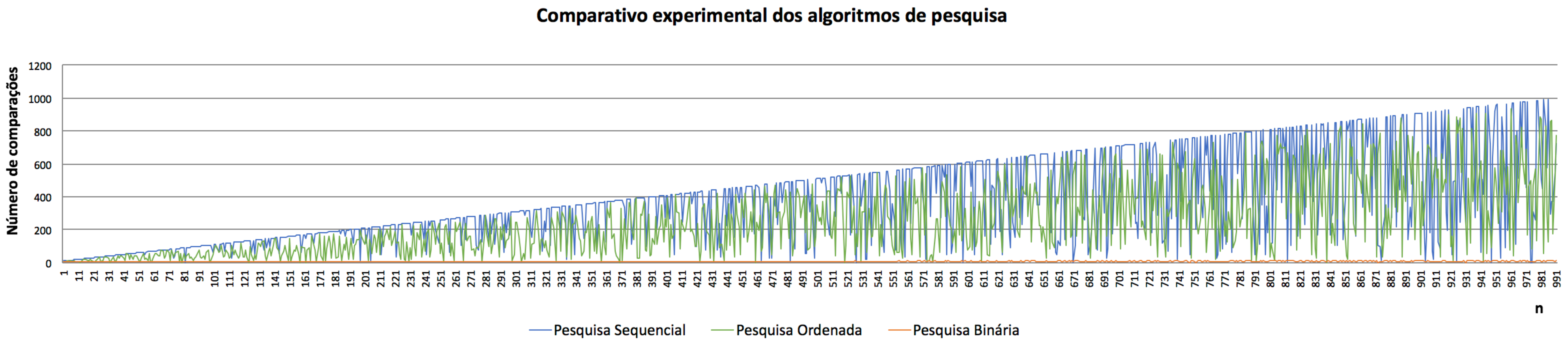
EXEMPLO

- Na terceira iteração a lista será dividida ao meio novamente e *16* será comparado ao elemento do meio que é `lista[0] = 0`
    - Como *16 != 0* (em negrito) e não há mais como dividir o arranjo, o processo termina e o registro *16* não foi encontrado

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	**33**	35	42	61							
0	31											

- Perceba que ao final do processo apenas ***3*** comparações (em negrito) foram realizadas das ***13*** que poderiam ter sido feitas em uma pesquisa sequencial

ANALISE EXPERIMENTAL



- Veja um comparativo experimental ilustrando o **número de comparações no eixo *y*** pelo **tamanho da entrada *n* no eixo *x***
  - Números aleatórios entre *0* e *1.000*
  - Tamanhos entre *10* e *1.000* (eixo *x*) e salvamos o número de comparações (eixo *y*)
  - A busca foi feita por um **número fixo** em todos as lista

ANALISE EXPERIMENTAL

- Muitas vezes, o número buscado pode não ser encontrado, gerando o pior caso dos algoritmos
- O algoritmo de **Pesquisa Sequencial** tem um comportamento em pior caso **linear**
- O algoritmo de **Pesquisa Ordenada** também apresenta um comportamento **linear** porém o seu pior caso (pesquisa na lista por inteiro quando o registro buscado não existe) raramente acontece
    - Essa é a vantagem do algoritmo de pesquisa utilizando a lista ordenada tendo em vista que sua classe de complexidade é a mesma da Pesquisa Sequencial $O(n)$
- Já a **Pesquisa Binária** é da classe de complexidade **logarítmica** apresentando um comportamento muito mais eficiente que os outros

CLASSES DE COMPLEXIDADE

- Esse exemplo nos introduz a um conceito muito importante em análise de algoritmos que são as **classes de complexidade**
  - Há diversas classes de complexidade, das quais a classe *log(n)* é de grande interesse por contemplar problemas que podem ser resolvidos de forma bastante eficiente