Análise de Algoritmos
Conceitos Básicos

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação

26 de novembro de 2024

Algoritmos

O que é um algoritmo?

É possível descrever um algoritmo de várias maneiras:

- Usando linguagem comum.
- Usando uma linguagem de programação de alto nível: C, Python, Java, entre outras.
- Implementando-o em linguagem de máquina diretamente executável em hardware.
- Por meio de um pseudocódigo.

Usaremos essencialmente pseudocódigo nesta disciplina!

Exemplo de pseudocódigo

Algoritmo ORDENA-POR-INSERÇÃO: Ordena de forma crescente um vetor $A[1...n]$.

```
ORDENA-POR-INSERÇÃO(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1 . . . j−1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

Análise de algoritmos

O que é importante analisar?

- **Finitude**: O algoritmo para?
- **Corretude ou Exatidão**: O algoritmo faz o que promete?
- **Eficiência ou complexidade no tempo**: Calcula-se o tempo de execução do algoritmo (i.e. análise empírica), ou quantas instruções são executadas em função do tamanho da entrada (i.e. análise de complexidade).

Finitude

```
ORDENA-POR-INSERÇÃO (A, n)
1. para j = 2 até n faça
   ...
4.    i = j - 1
5.    enquanto i >= 1 e A[i] > chave faça
6.       ...
7.       i = i - 1
   ...
```

- No laço da linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução para em algum momento por causa do teste condicional $i \geq 1$.
- O laço na linha 1 evidentemente para porque o contador j atingirá o valor $n + 1$. Portanto, o algoritmo para.

Corretude

- Um algoritmo é correto se, para toda instância do problema, ele para e devolve uma resposta correta.

- Usamos o método de **loops invariantes** para nos ajudar a entender por que um algoritmo é correto. Como?
  - Inicialização: O invariante é verdadeiro antes da primeira iteração do laço (trivial, em geral).
  - Manutenção: Se o invariante for verdadeiro antes de uma iteração, ele permanecerá verdadeiro antes da próxima.
  - Término: Se o algoritmo para e o invariante vale no início da última iteração, então o algoritmo é correto.

- ORDENA-POR-INSERÇÃO: No começo de cada iteração do laço "para" (linha 1), o vetor $A[1...j-1]$ está ordenado.

Invariante de laço

- **Definição**: Um invariante é uma propriedade que relaciona as variáveis do algoritmo a cada execução completa do laço.
- O invariante deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- Em geral, é mais difícil descobrir um invariante apropriado do que mostrar sua validade se ele for dado de "bandeja".

Eficiência

- Em geral, não basta saber que um dado algoritmo para e é correto. Se ele for muito lento, terá pouca utilidade.
- Queremos projetar algoritmos eficientes (ou "rápidos").
- Mas o que seria uma boa medida de eficiência?
- Existe um critério uniforme para comparar algoritmos?

Antes de responder essas perguntas, é preciso definir um modelo computacional de máquina.

Modelo computacional

- Simula máquinas convencionais de verdade.
- Estabelece quais são os recursos disponíveis, as instruções básicas e quanto elas custam (= **tempo**).
- Com o modelo, é possível estimar, através de uma análise matemática, o tempo que um algoritmo gasta em função do tamanho da entrada (= **análise de complexidade**).
- A análise de complexidade depende sempre do modelo computacional adotado.

Modelo computacional

- Salvo mencionado o contrário, usaremos sempre o Modelo Abstrato RAM (Random Access Machine):
  - Possui um único processador;
  - Executa instruções sequencialmente;
  - Tipos básicos são números inteiros e reais;
  - Executa operações aritméticas, comparações, movimentação de dados de tipo básico e fluxo de controle (chamada e retorno de rotina, teste if/else, etc.) **em tempo constante**;
  - ...
- Veja maiores detalhes do modelo RAM no Livro Texto.

Complexidade de algoritmos

- Vamos analisar a **eficiência** (ou complexidade no tempo) do algoritmo que encontra o maior elemento de um vetor A com n elementos.
- Basicamente, vamos encontrar o tempo de execução de cada passo do algoritmo em função do tamanho da entrada.

```
MAXIMO (A, n)                    Custo    #execuções
1. max = A[1]                     c1
2. para i = 2 até n faça          c2
3.    se max < A[i]               c3
4.       então max = A[i]         c4
5. retorne (max)                  c5
```

Complexidade de algoritmos

- Vamos analisar a **eficiência** (ou complexidade no tempo) do algoritmo que encontra o maior elemento de um vetor A com n elementos.
- Basicamente, vamos encontrar o tempo de execução de cada passo do algoritmo em função do tamanho da entrada.

```
MAXIMO (A, n)                    Custo      #execuções
1. max = A[1]                      c1             1
2. para i = 2 até n faça           c2             n
3.   se max < A[i]                 c3           n - 1
4.     então max = A[i]            c4    0 <= t <= n - 1
5. retorne (max)                   c5             1
```

Complexidade de algoritmos

- O tempo total de execução $T(n)$ do algoritmo é dado por

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5.$$

- Esse tempo de execução é da forma $an + b$ para constantes a e b que dependem apenas dos custos.

- Em outras palavras, o algoritmo tem como complexidade no tempo uma **função linear** no tamanho da entrada.

Complexidade de algoritmos

- Mas nem sempre entradas de tamanho igual (i.e. mesmo valor de n) apresentam o mesmo tempo de execução.
- Veja o caso da busca linear, cujo algoritmo é mostrado abaixo. A ideia é verificar a existência do elemento x num vetor A.

```
LINEAR (A, n, x)                          Custo  #execuções
1. i = 1                                   c1
2. enquanto i <= n e x != A[i] faça        c2
3.   i = i + 1                             c3
4. se i <= n                               c4
5.   então escreva ("encontrado")          c5
6.   senão escreva ("ausente")             c6
```

Complexidade de algoritmos

- Mas nem sempre entradas de tamanho igual (i.e. mesmo valor de n) apresentam o mesmo tempo de execução.
- Veja o caso da busca linear, cujo algoritmo é mostrado abaixo. A ideia é verificar a existência do elemento x num vetor A.

```
LINEAR (A, n, x)                              Custo  #execuções
1. i = 1                                       c1          1
2. enquanto i <= n e x != A[i] faça            c2   1 <= t <= n + 1
3.   i = i + 1                                 c3   0 <= t <= n
4. se i <= n                                   c4          1
5.   então escreva ("encontrado")              c5       0 ou 1
6.   senão escreva ("ausente")                 c6       1 ou 0
```

Complexidade de algoritmos

- No **melhor caso**, quando o elemento procurado encontra-se na primeira posição, o tempo total de execução $T(n)$ é

$$T(n) = c_1 + c_2 + c_4 + c_5 \text{ ou } c_6.$$

- Nesse caso, o tempo de execução é uma **constante**, ou seja, não depende do tamanho da entrada.

- Considerando o **pior caso**, quando o elemento procurado não encontra-se no vetor, o tempo total de execução $T(n)$ é

$$T(n) = c_1 + c_2(n+1) + c_3 n + c_4 + c_5 \text{ ou } c_6.$$

- Já aqui o tempo de execução é da forma $an + b$, ou seja, uma **função linear** no tamanho da entrada.

Exemplo: Algoritmo BubbleSort

- O algoritmo para?
- O algoritmo é correto?
- Qual é a complexidade no tempo do algoritmo?
- Existe pior e melhor caso? Por exemplo, a ordenação inicial do vetor de entrada influencia na eficiência do algoritmo?

```
BUBBLESORT (A, n)
1. para i = 1 até n - 1 faça
2.   para j = 1 até n - i faça
3.     se A[j] > A[j + 1] então
4.         temp = A[j]
5.         A[j] = A[j + 1]
6.         A[j + 1] = temp
```

Exemplo: Algoritmo BubbleSort

- A seguir, um exemplo de aplicação do BubbleSort, com a ordenação desejada obtida após a 4a iteração.

Iteração	Vetor	Trocas
vetor inicial	40 37 95 42 39 51 60	
após 1a iteração	37 40 42 39 51 60 95	5
após 2a iteração	37 40 39 42 51 60 95	1
após 3a iteração	37 39 40 42 51 60 95	1
após 4a iteração	37 39 40 42 51 60 95	0

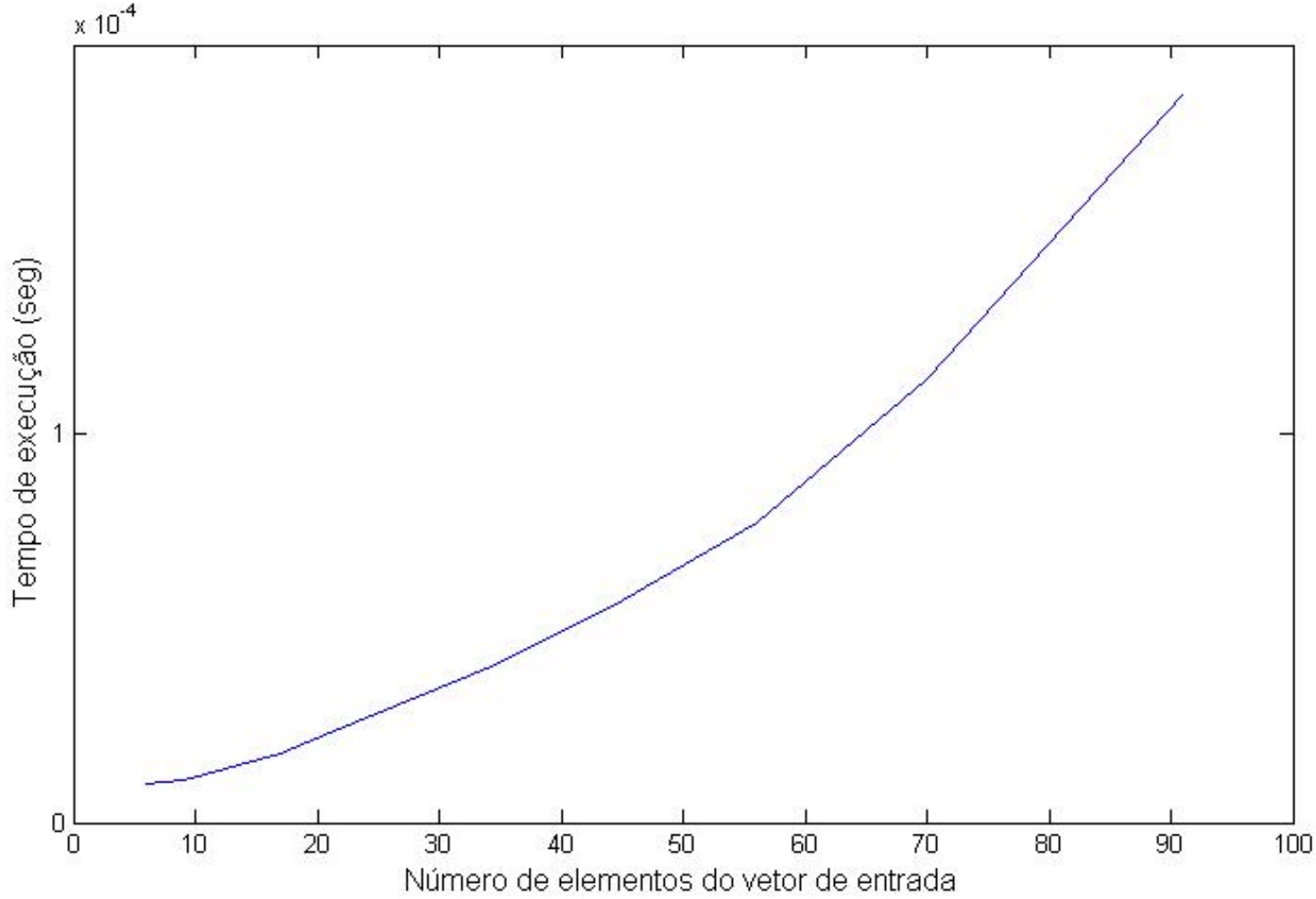
Exemplo: Algoritmo BubbleSort

- Durante cada passo, o BubbleSort compara sucessivamente os elementos adjacentes, trocando-os quando necessário.
- Quando o i-ésimo passo começa, os $i-1$ maiores elementos estão em suas posições corretas. Durante esta etapa, $n-i$ comparações são usadas.
- Assim, o número total de comparações (linha 3) usadas pelo BubbleSort para ordenar um vetor de n elementos é

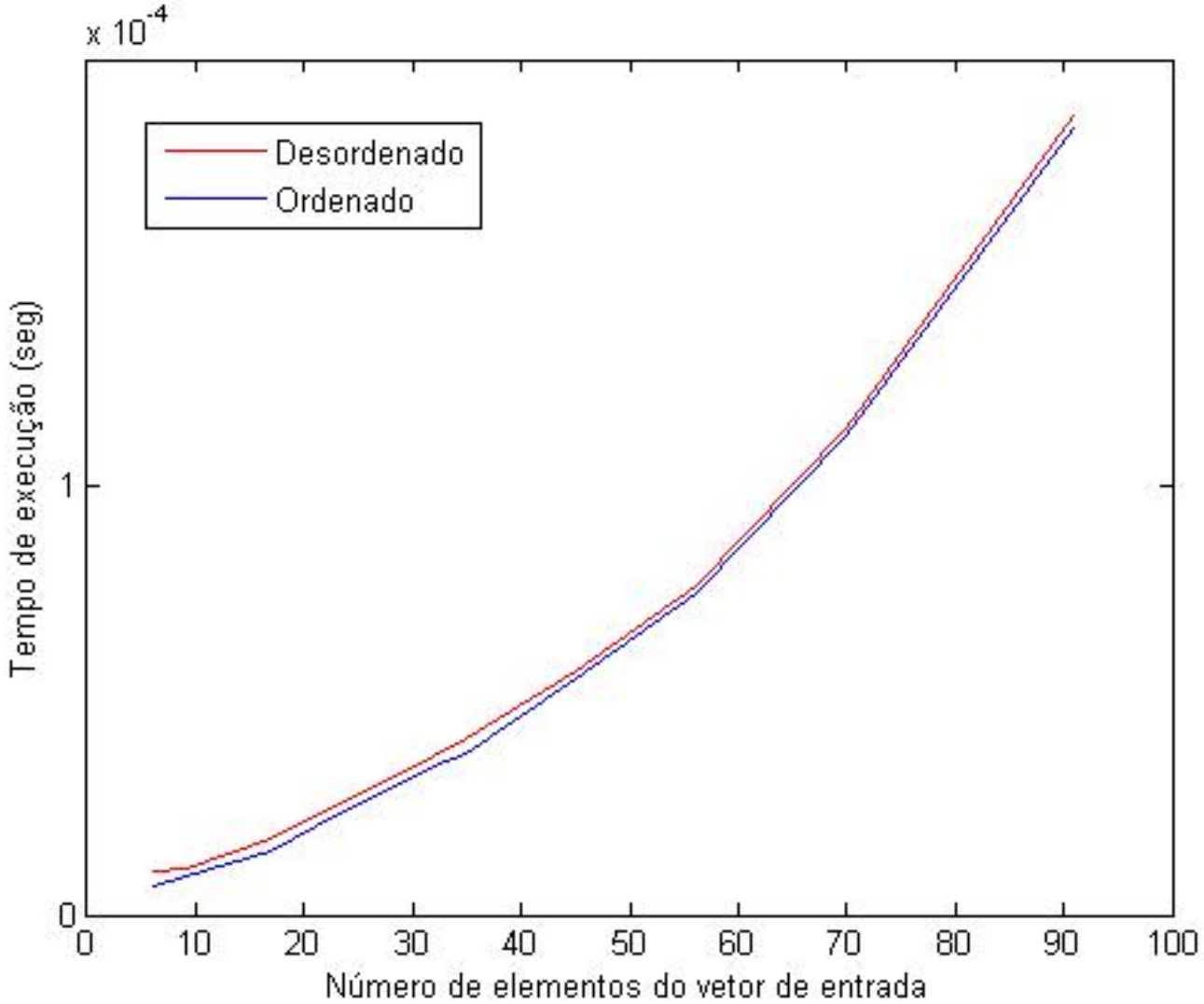
$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}$$

- O tempo de execução é uma **função quadrática** no tamanho da entrada e **não** depende da ordenação inicial do vetor.

Exemplo: Algoritmo BubbleSort



Exemplo: Algoritmo BubbleSort



Caso médio

- O **caso médio** (ou caso esperado) corresponde à média dos tempos de execução de todas as entradas de tamanho n.
- Na análise do caso médio, uma distribuição de probabilidades sobre o conjunto de entrada de tamanho n é suposta, e o custo médio é obtido com base nessa distribuição.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Porém, na prática, isso nem sempre é verdade.
- Por essa razão, a análise do caso médio é geralmente mais difícil de obter do que as análises de melhor e pior caso.

Exemplo: Caso médio

- Uma pesquisa linear com sucesso examina aproximadamente metade dos registros no caso médio.

- Considere que toda pesquisa recupera um registro:

  Caso p_i seja a probabilidade de que o i-ésimo registro seja procurado e que para recuperar o i-ésimo registro são necessárias i comparações, então

  $$f(n) = 1p_1 + 2p_2 + 3p_3 + ... + np_n.$$

  Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = \frac{1}{n}$, $0 \leq i \leq n$. Nesse caso,

  $$f(n) = \frac{1}{n}(1 + 2 + 3 + ... + n) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}.$$

Comportamento assintótico

- Salvo contrário, considera-se o **pior caso** e o **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- O estudo assintótico permite "jogar para debaixo do tapete" os valores das constantes e os termos não dominantes.

n	$4n + 52$	$4n$
64	308	256
512	2100	2048
2048	8244	8192
4096	16436	16384
8192	32820	32768
16384	65588	65536
32768	131124	131072

Comportamento assintótico

- De um modo geral, podemos nos concentrar nos **termos dominantes** e esquecer os demais.

- Considere a função quadrática $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

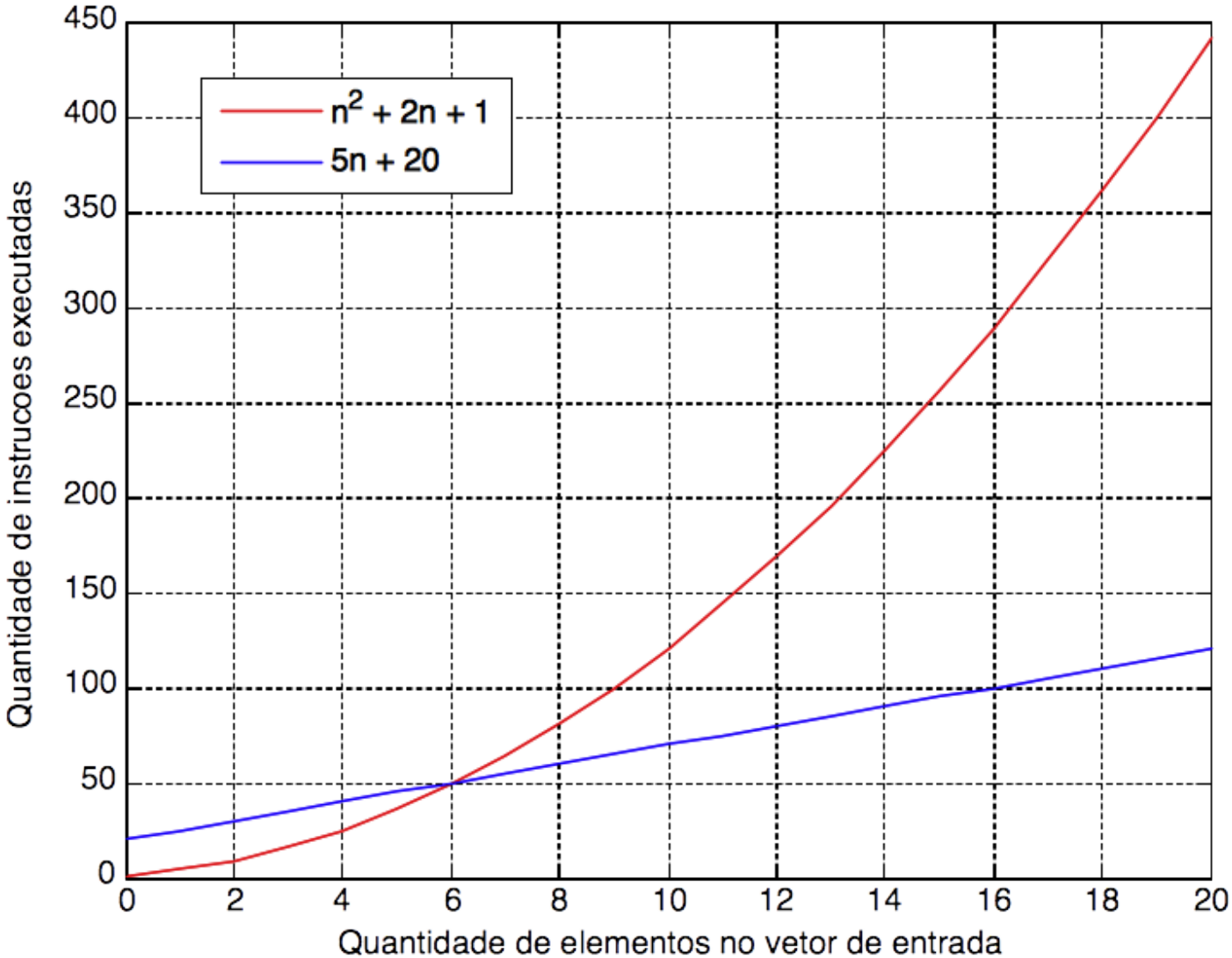
Comparando algoritmos

- Suponha que o Fulano desenvolveu um algoritmo para resolver o problema da ordenação por inserção, chamado *order_Ful*.
- Agora, suponha que o Beltrano fez outro algoritmo para resolver o mesmo problema, chamado *order_Bel*.
- Como saber qual dos dois algoritmos é mais eficiente?
- Podemos cronometrar o tempo?
  - Sim! Mas a análise empírica é uma ideia trabalhosa. Por quê?
  - Porque teríamos que medir o tempo de *order_Ful* e *order_Bel* para 1 elemento, 2 elementos, ..., 10 elementos, ..., 100 elementos, ..., 1000 elementos, ...

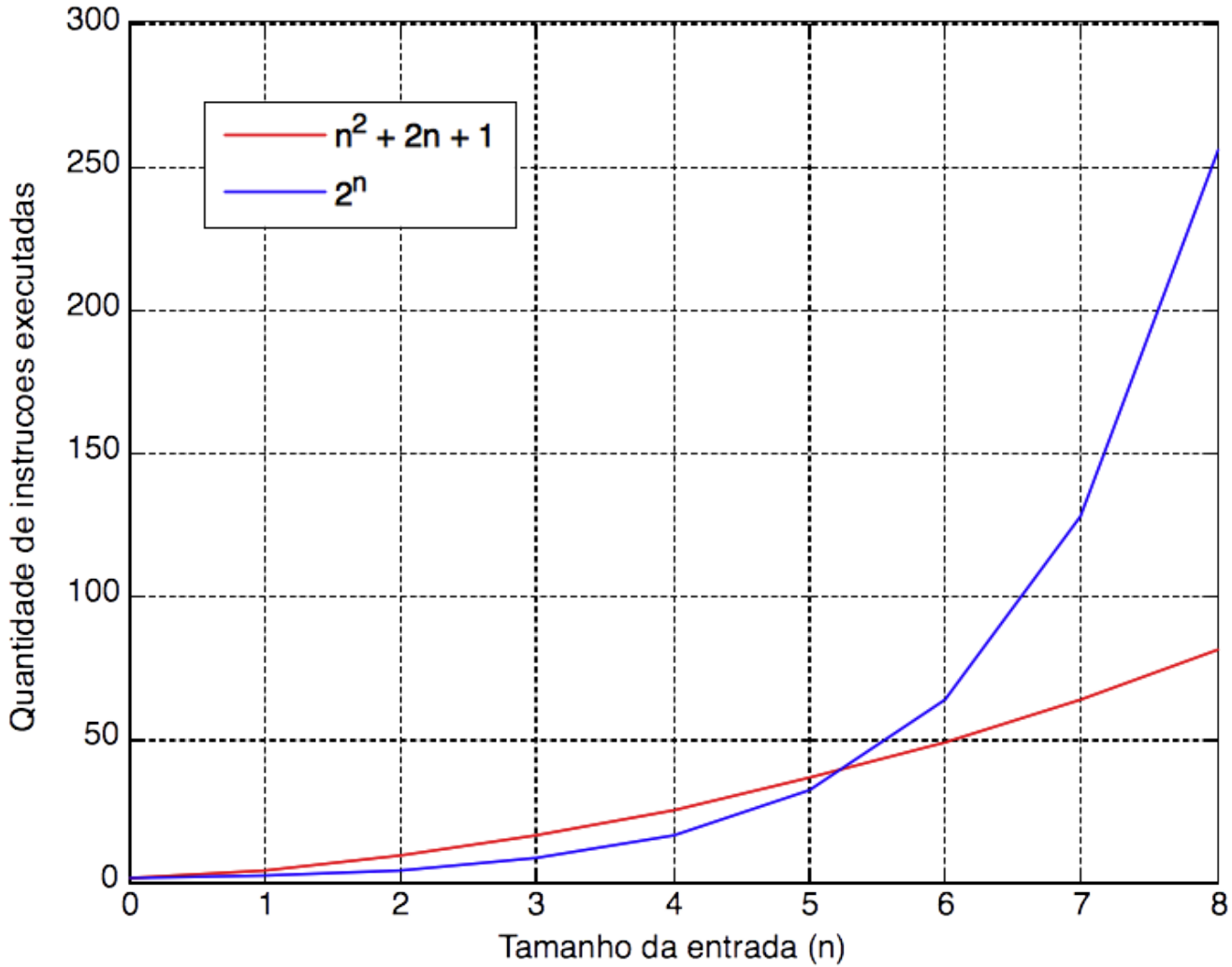
Comparando algoritmos

- Em geral, faz-se uma análise de complexidade no tempo.
- Calcula-se, para cada algoritmo, quantas instruções são executadas dada uma entrada de tamanho n.
- Suponha que o algoritmo *order_Ful* executa uma quantidade de instruções de acordo com a função: $f(n) = n^2 + 2n + 1$.
- Já o outro algoritmo *order_Bel* tem como complexidade no tempo a função: $f(n) = 5n + 20$.
- Por exemplo, se passarmos 50 elementos para *order_Ful*, ele fará $f(50) = 50^2 + 2 \,.\, 50 + 1 = 2.601$ instruções. Já *order_Bel* é mais eficiente, pois executará apenas 270 instruções.

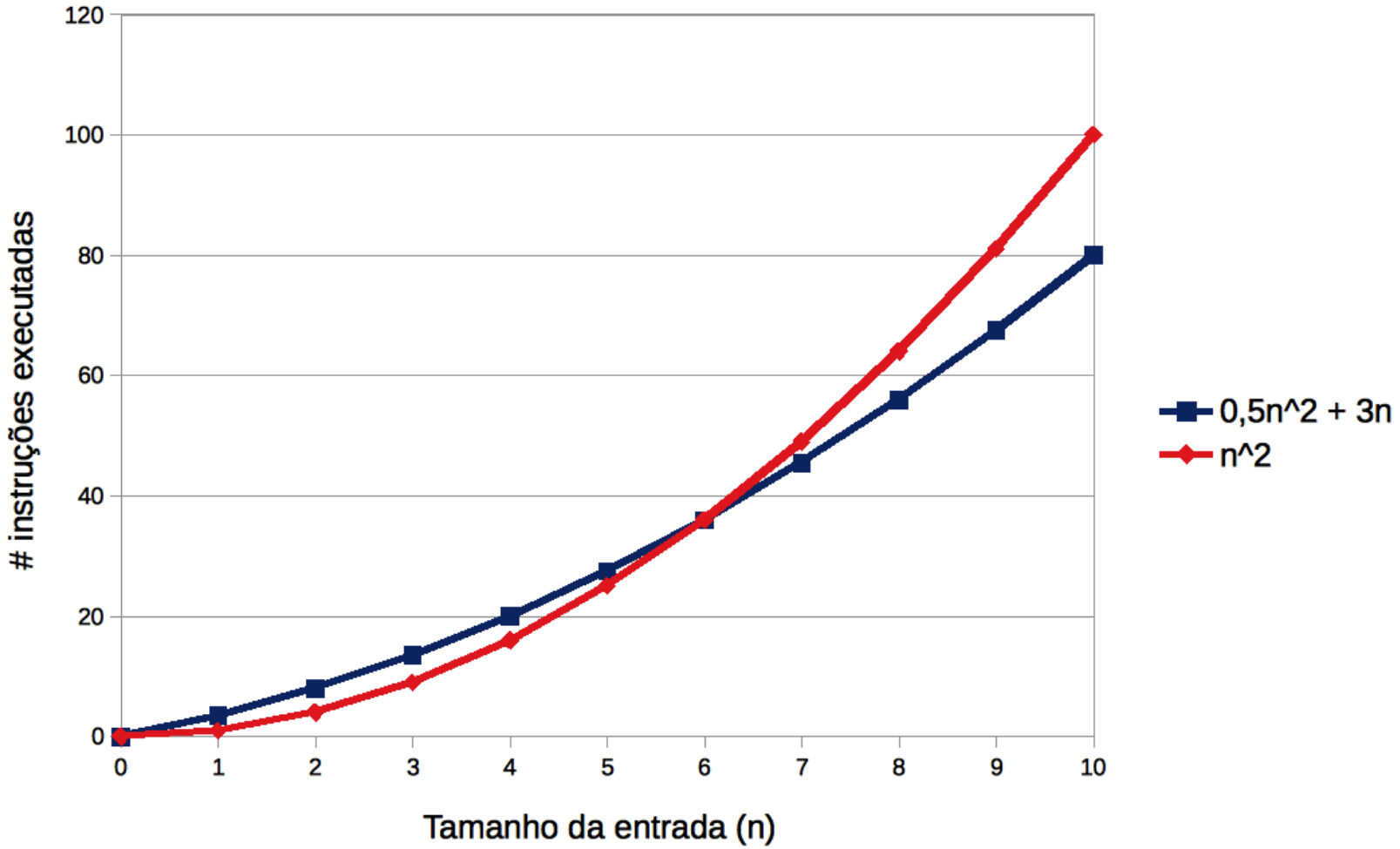
Comparando algoritmos



Comparando algoritmos



Comparando algoritmos



Conclusões

- A complexidade no tempo (= eficiência) de um algoritmo pode ser calculada pelo **número de instruções básicas que ele executa em função do tamanho da entrada**.
- Adota-se uma "atitude pessimista" (**análise do pior caso**).
- A análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE (**análise assintótica**).
- Um algoritmo é chamado eficiente se a função que mede sua complexidade no tempo é limitada por um **polinômio**.

  Ex: $n, 7n - 3, 14n^2 + 4n - 8, n^5$.

Conclusões

- Existe uma solução eficiente para qualquer tipo de problema?
- A resposta para essa pergunta é não, ou seja, existem certos problemas para os quais **não** se conhece algoritmos eficientes capazes de resolvê-los.
- Problema do caixeiro viajante: Tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem.
- Mas por que é importante identificar quando estamos lidando com um problema "intratável"?

Exercícios

1. Qual é o menor valor de entrada n (considere $n > 0$) tal que um algoritmo cujo tempo de execução é $10n^2$ é mais rápido que um algoritmo cujo tempo de execução é 2^n? Qual desses algoritmos você considera mais eficiente?

Exercícios

2. Cada um dos algoritmos abaixo recebe um inteiro positivo e devolve outro inteiro positivo. Os dois algoritmos devolvem o mesmo número se receberem o mesmo valor de entrada n? Qual dos dois algoritmos é mais eficiente?

```
Soma-Quadrados-A (n)
1.  x = 0
2.  para j = 1 até n faça
3.      x = x + j * j
4.  retorne (x)
```

```
Soma-Quadrados-B (n)
1.  x = n * (n + 1) * (2n + 1)
2.  x = x/6
3.  retorne (x)
```

Exercícios

3. Implemente em uma linguagem de programação a sua escolha o algoritmo abaixo que verifica se o valor de entrada é, ou não, um número primo. Existe pior e melhor caso?

```
PRIMO (n)
1. j = 2
2. enquanto j < n e mod(n,j) != 0 faça
3.     j = j + 1
4. se j = n
5.     então escreva ("é primo")
6.     senão escreva ("não é primo")
```

Exercícios

4. Implemente em uma linguagem de programação a sua escolha o algoritmo abaixo que lista os números primos menores ou igual ao valor de entrada. Existe pior e melhor caso?

```
LISTA_PRIMO (n)
1. x = 1
2. para i = 2 até n faça
3.    j = 2
4.    enquanto j < i e mod(i,j) != 0 faça
5.       j = j + 1
6.    se j = i
7.       A[x] = i
8.       x = x + 1
9. retorne (A[x])
```

Exercícios

5. Compare assintoticamente o tempo de execução (apresente dados, tabelas e gráficos) dos algoritmos de ordenação:
   - Inserção;
   - BubbleSort; e
   - Seleção.

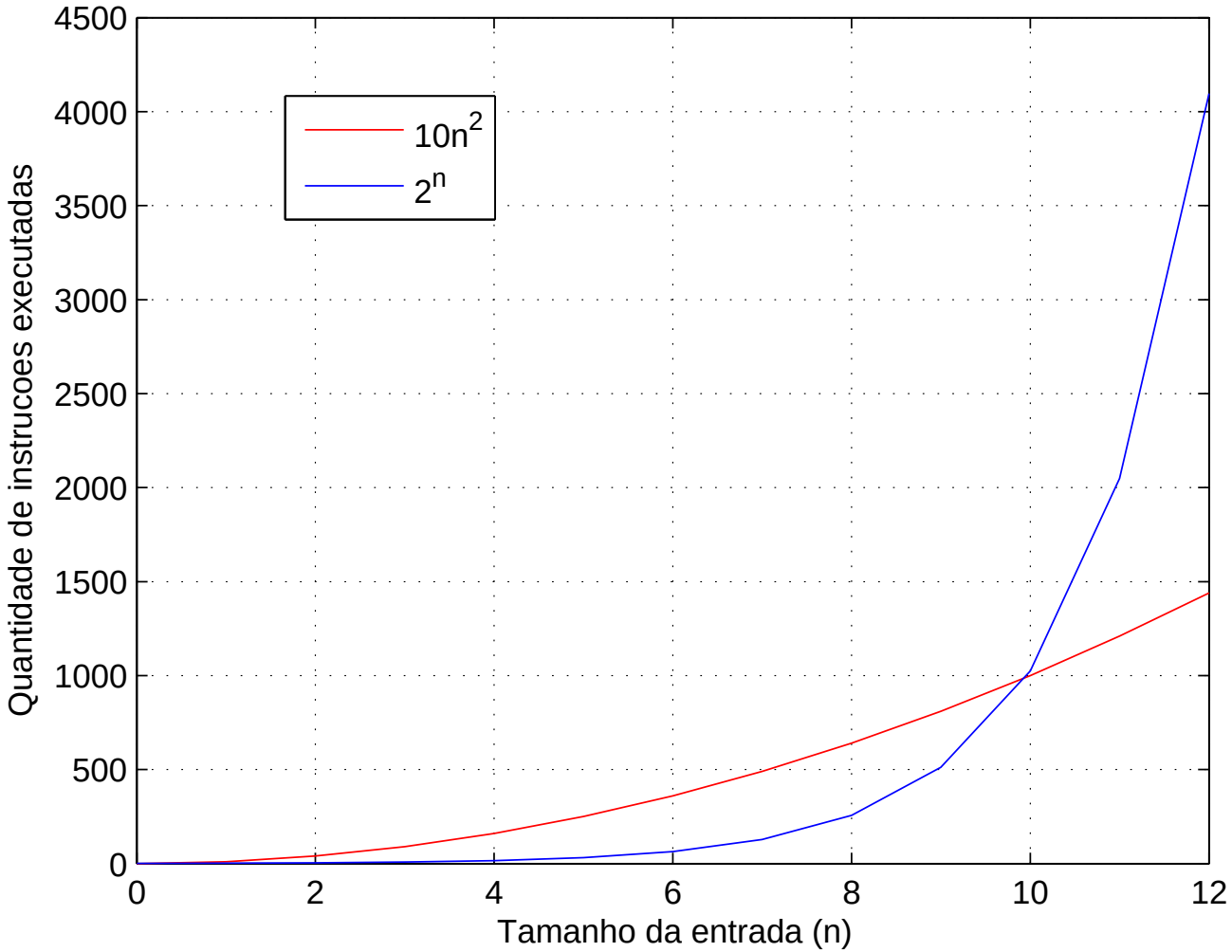
   Em seguida, responda os seguintes questionamentos.

   a. Qual desses algoritmos é o mais eficiente?

   b. A ordem inicial do vetor de entrada influencia no desempenho dos algoritmos?

   c. Qual desses algoritmos você usaria na sua aplicação?

Exercício 1


4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
12
Quantidade de instrucoes executadas
Tamanho da entrada (n)
10n^2
2^n

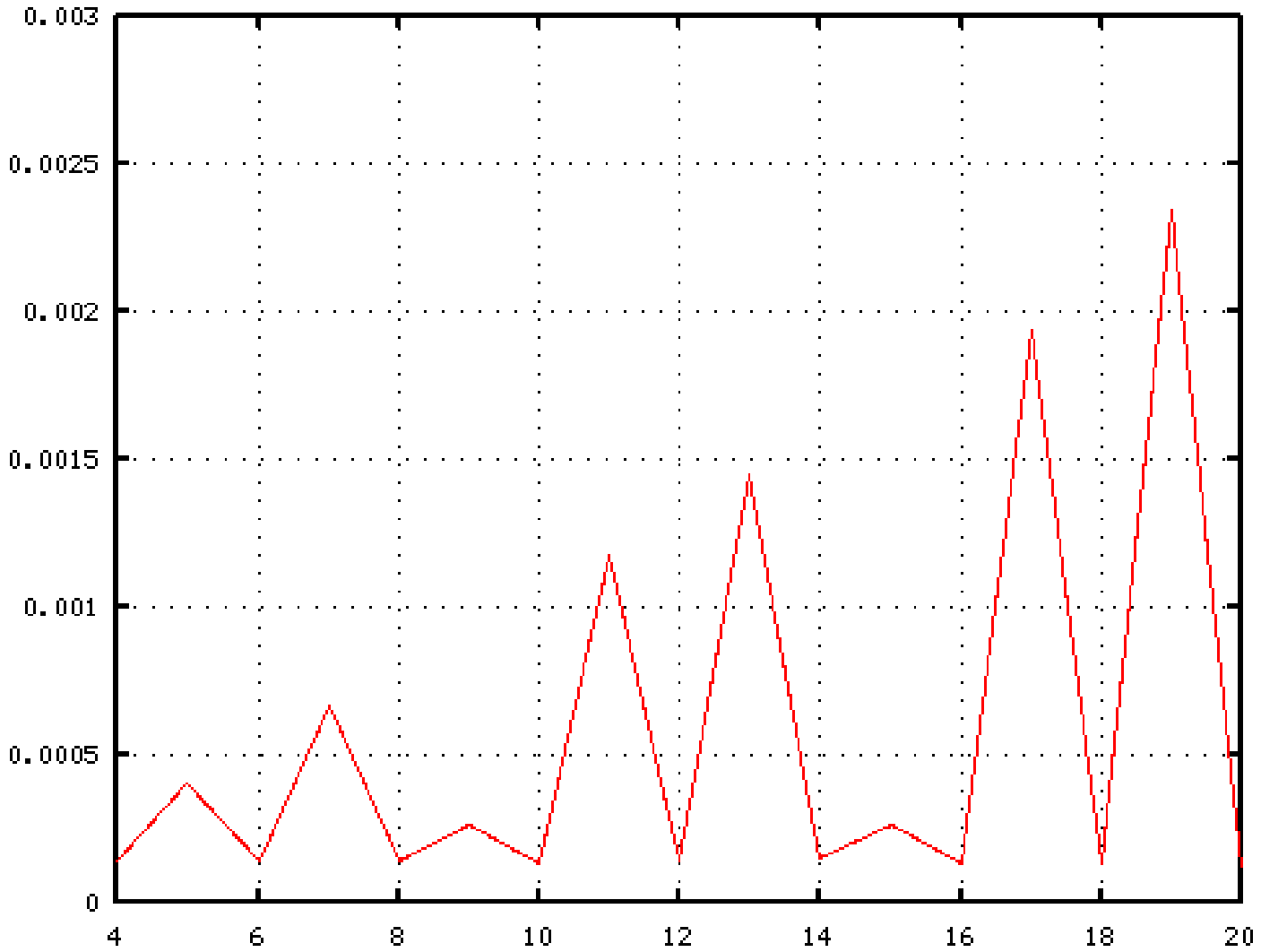
Exercício 2

- Os dois algoritmos devolvem o mesmo número se receberem o mesmo valor de entrada n.
- Em uma análise assintótica, o algoritmo A é mais lento (ou menos eficiente) que o algoritmo B.
- Isso porque a quantidade de vezes que a linha 3 do algoritmo A será executada depende do valor de entrada n, o que leva a uma **complexidade linear**.
- Já o algoritmo B não possui laço de repetição, mantendo **constante** o número de instruções executadas em função do tamanho da entrada.

Exercício 3

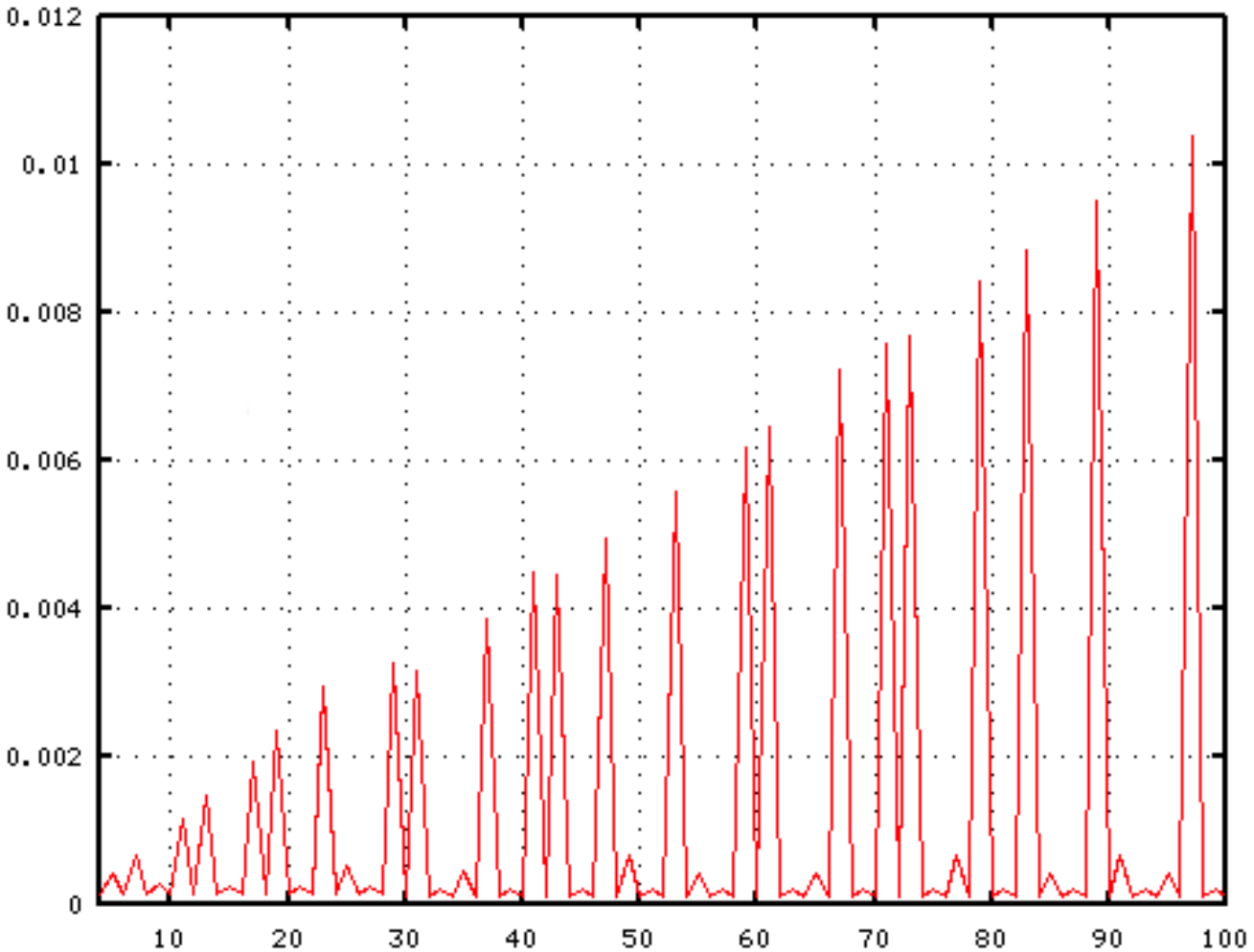
```
PRIMO (n)
1. j = 2
2. enquanto j < n e mod(n,j) != 0 faça
3.     j = j + 1
4. se j = n
5.     então escreva ("é primo")
6.     senão escreva ("não é primo")
```

Exercício 3: valor de entrada x tempo de execução



3,32451, 0,00327150

Exercício 3: valor de entrada x tempo de execução

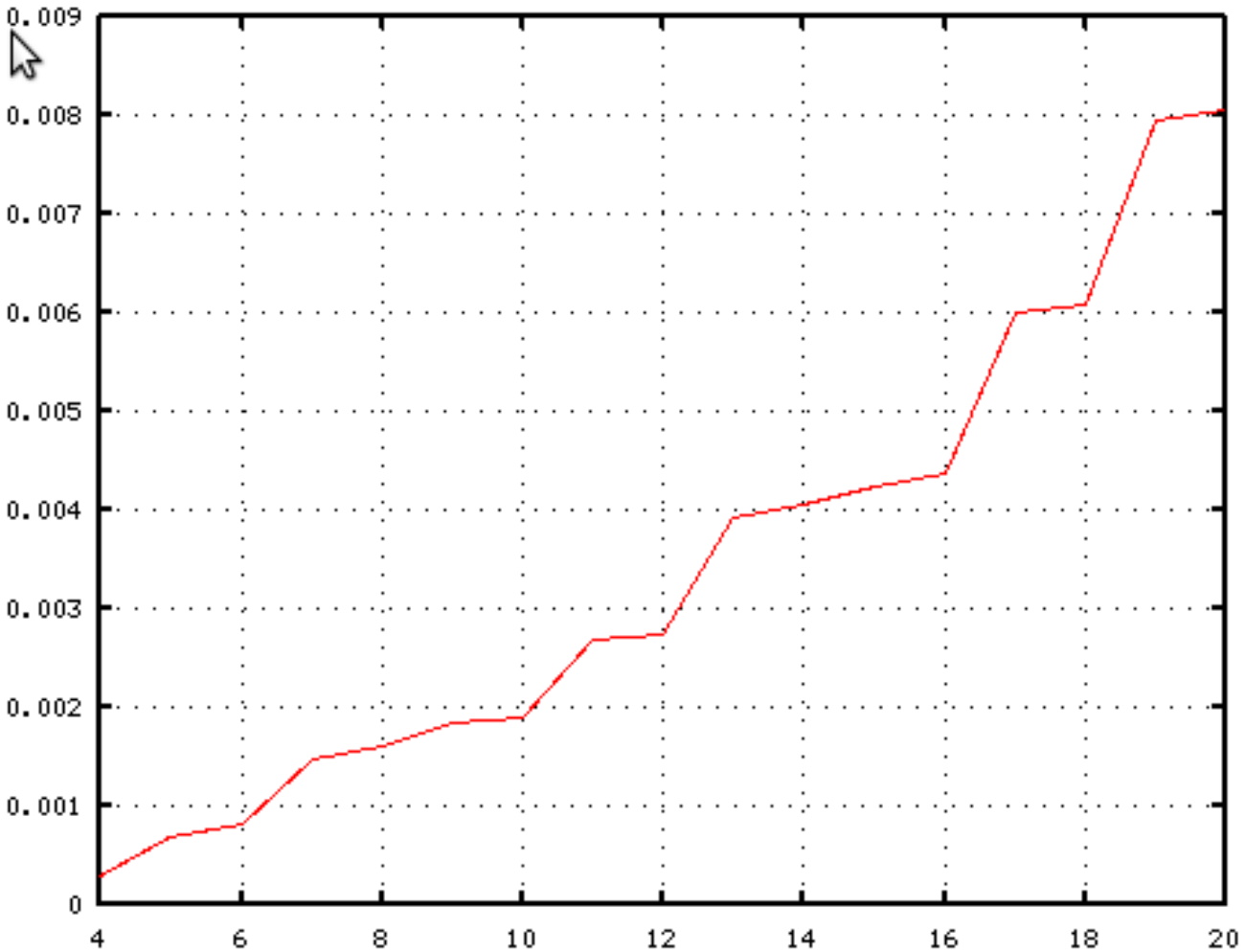


13,6786, 0,00705184

Exercício 4

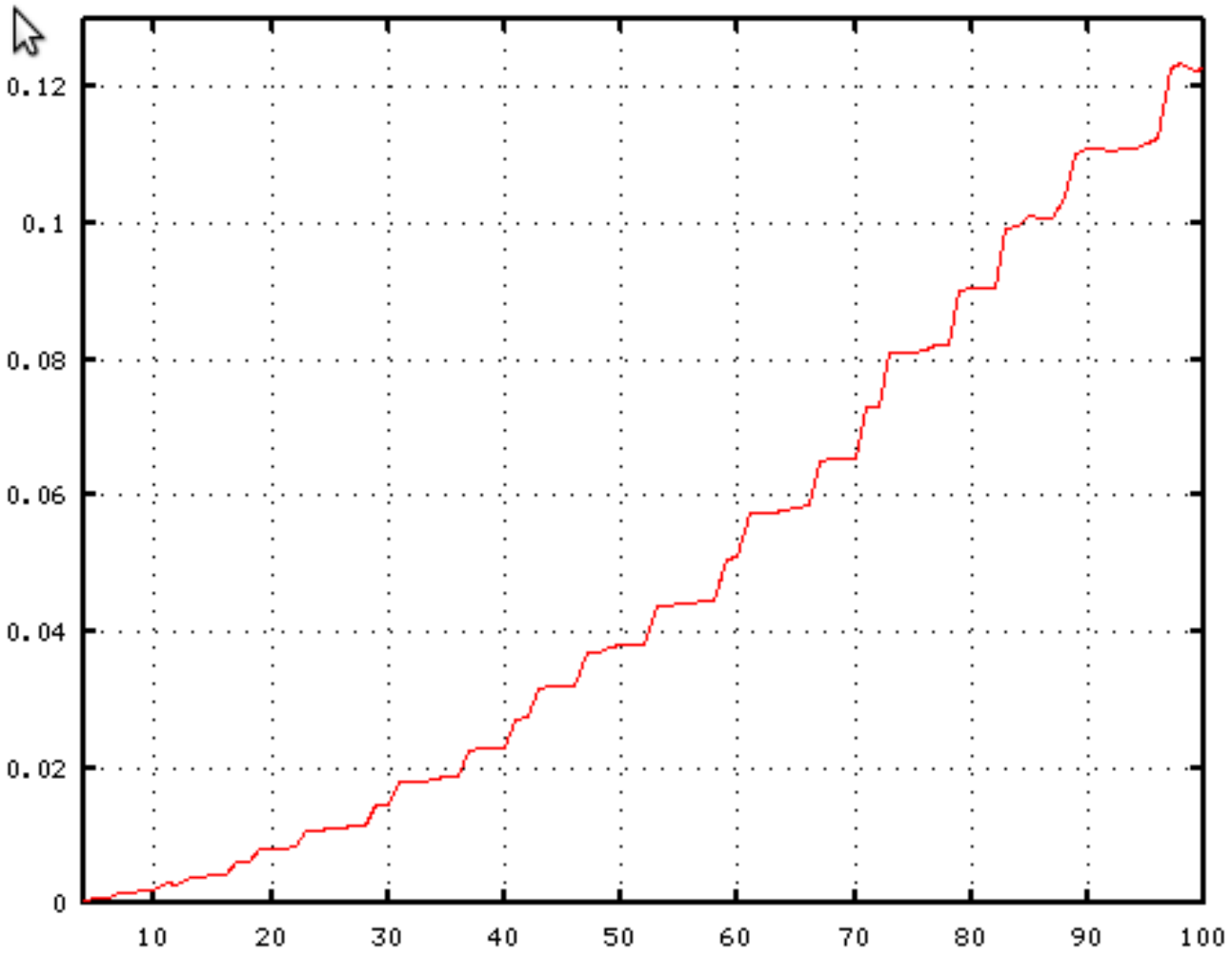
```
LISTA_PRIMO (n)
1. x = 1
2. para i = 2 até n faça
3.    j = 2
4.    enquanto j < i e mod(i,j) != 0 faça
5.       j = j + 1
6.    se j = i
7.       A[x] = i
8.       x = x + 1
9. retorne (A[x])
```

Exercício 4: valor de entrada x tempo de execução



2.73472, 0.00881390

Exercício 4: valor de entrada x tempo de execução



-2.04915, 0.131130

Exercício 5

- Algoritmo de ordenação por seleção:

```
Selecao (A, n)
1. para i = 1 até n - 1 faça
2.    minimo = i
3.    para j = i + 1 até n faça
4.       se A[j] < A[minimo]
5.          então minimo = j
6.    temp = A[i]
7.    A[i] = A[minimo]
8.    A[minimo] = temp
```

- Sua complexidade no tempo é **quadrática** e não depende da ordenação inicial do vetor de entrada.

Exercício 5

- Exemplo de aplicação do algoritmo de ordenação por seleção:

Iteração	Vetor
vetor inicial	40 37 95 42 39 51 60
após 1a iteração	37 40 95 42 39 51 60
após 2a iteração	37 39 95 42 40 51 60
após 3a iteração	37 39 40 42 95 51 60
após 4a iteração	37 39 40 42 95 51 60
após 5a iteração	37 39 40 42 51 95 60
após 6a iteração	37 39 40 42 51 60 95

Exercício 5

ORDENA-POR-INSERÇÃO(A, n)	Custo	# execuções
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $\quad chave \leftarrow A[j]$	c_2	?
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	?
4 $\quad i \leftarrow j-1$	c_4	?
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	?
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	?
7 $\quad\quad i \leftarrow i-1$	c_7	?
8 $\quad A[i+1] \leftarrow chave$	c_8	?

- No melhor caso, i.e. vetor em ordem crescente, tem-se:

$$f(n) = c_1(n) + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1).$$

- Nessa situação, o tempo de execução do algoritmo é uma **função linear** no tamanho da entrada.

Exercício 5

- No pior caso, i.e. vetor em ordem decrescente, a linha 5 será executada j vezes a cada interação do laço "para":

  $2 + 3 + 4 + 5 + ... + n = \frac{n(n+1)}{2} - 1.$

- Também é possível analisar o algoritmo pelas linhas 6 e 7, que serão executadas $j - 1$ vezes a cada interação do laço "para":

  $1 + 2 + 3 + 4 + ... + n - 1 = \frac{n(n+1)}{2} - n.$

- Nessa situação, o tempo de execução do algoritmo é uma **função quadrática** no tamanho da entrada.

Exemplo 5

- Exemplo de aplicação do algoritmo de ordenação por inserção:

Iteração	Vetor	Trocas
vetor inicial	40 37 95 42 23 51 27	
após $j = 2$	37 40 95 42 23 51 27	1
após $j = 3$	37 40 95 42 23 51 27	0
após $j = 4$	37 40 42 95 23 51 27	1
após $j = 5$	23 37 40 42 95 51 27	4
após $j = 6$	23 37 40 42 51 95 27	1
após $j = 7$	23 27 37 40 42 51 95	5

Exercício 5

- **Inserção**: Bom método a ser usado quando a sequência está quase ordenada, ou quando se deseja adicionar poucos itens a uma sequência já ordenada.

- **Seleção**: Entre os algoritmo de ordenação, apresenta umas das menores quantidades de movimentos entre os elementos, assim pode haver algum ganho quando se necessita ordenar estruturas complexas.

- **Bubblesort**: Tem um número muito grande de movimentação de elementos, assim não deve ser usado se a estrutura a ser ordenada for complexa.

Exercício 5: comparação com entrada aleatória