

MC102 — Algoritmos, Busca e Ordenação

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2026-06-22 10:27

MÍNIMO

MÍNIMO: Dada uma lista l de números, encontrar o índice do menor valor que aparece em l

Ideia do Algoritmo:

- Chutar que 0 é o índice do mínimo
- Percorrer l do início para o fim
 - Se eu encontrar um valor menor no índice atual
 - Eu atualizo qual é o meu chute

```
1 Minimo(l)
2   Seja n o tamanho de l
3   min = 0
4   Para i = 1 até n - 1
5       Se l[min] > l[i]
6           min = i
7   Devolva min
```

MÍNIMO

```
1 Minimo(l)
2     Seja n o tamanho de l
3     min = 0
4     Para i = 1 até n - 1
5         Se l[min] > l[i]
6             min = i
7     Devolva min
```

Suponha que, no começo de uma iteração, **min** é o índice do menor valor de $l[0], \dots, l[i - 1]$

- É verdade que, ao final da iteração **i**, **min** é o índice do menor valor de $l[0], \dots, l[i - 1], l[i]$?
 - Se $l[\text{min}] \leq l[i]$, **min** já é o índice do menor valor
 - Caso contrário, **min** passa a ser o índice do menor valor

Note que:

- No começo da iteração **1**, **min** é o índice do menor valor de $l[0], \dots, l[i - 1]$
- No final da última iteração, **min** é o índice do menor valor de $l[0], \dots, l[n - 1]$

ORDENAÇÃO

ORDENAÇÃO: Dada uma lista l de n elementos, rearranjar os elementos de l de forma que $l[0] \leq l[1] \leq \dots \leq l[n-1]$.

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Aquecimento — Ordenação de três elementos

Queremos ordenar uma lista de três elementos

- Há um algoritmo que verifica as $3! = 6$ possibilidades e ordena
 - Mas isso claramente não escala para n elementos...
- Ideia de um algoritmo menor (e útil):
 - Vamos colocar o menor elemento na primeira posição
 - Em seguida ordenamos $l[1]$ e $l[2]$

```
1 Se  $l[0] > l[1]$ 
2   Troque  $l[0]$  com  $l[1]$ 
3 Se  $l[0] > l[2]$ 
4   Troque  $l[0]$  com  $l[2]$ 
5 Se  $l[1] > l[2]$ 
6   Troque  $l[1]$  com  $l[2]$ 
```

O algoritmo resolve o problema pois:

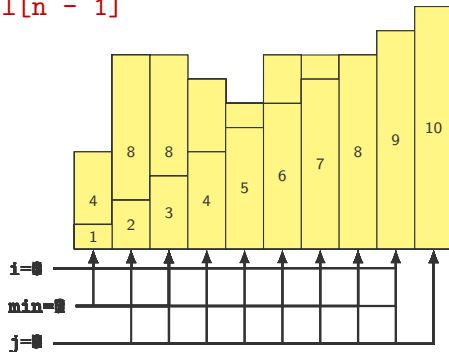
- Após a linha 2, $l[0]$ tem o valor mínimo entre $l[0]$ e $l[1]$
- Após a linha 4, $l[0]$ tem o valor mínimo entre $l[0]$, $l[1]$ e $l[2]$
- Após a linha 6, l está ordenada

Ordenação por Seleção

Ideia:

- Trocar $l[0]$ com o mínimo de $l[0], \dots, l[n - 1]$
- Trocar $l[1]$ com o mínimo de $l[1], \dots, l[n - 1]$
- ...
- Trocar $l[i]$ com o mínimo de $l[i], \dots, l[n - 1]$

```
1 SelectionSort(l)
2   Seja n o tamanho de l
3   Para i = 0 até n - 2
4       min = i
5       Para j = i + 1 até n - 1
6           Se l[min] > l[j]
7               min = j
8       Troque l[min] com l[i]
```



Exercício

Implemente o SelectionSort em Python

SelectionSort funciona?

```
1 SelectionSort(l)
2   Seja n o tamanho de l
3   Para i = 0 até n - 2
4       min = i
5       Para j = i + 1 até n - 1
6           Se l[min] > l[j]
7               min = j
8       Troque l[min] com l[i]
```

Os passos de SelectionSort são simples e ele sempre termina

Suponha que no início da iteração (linha 3) $l[0], \dots, l[i - 1]$ estão ordenados e são os menores elementos de l

- Isso é verdade no início da primeira iteração

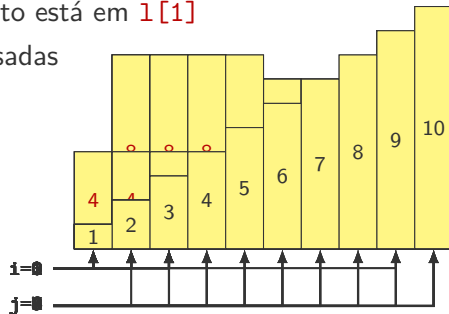
No final da iteração, $l[0], \dots, l[i]$ estão ordenados e são os menores elementos da lista?

- Sim, pois pegamos o menor valor de $l[i], l[i + 1], \dots, l[n - 1]$ e trocamos com $l[i]$

BubbleSort

- Se l não está ordenada, existe j com $l[j - 1] > l[j]$
- Então, do fim para o começo, trocamos pares invertidos
- Porém, apenas uma passada pode ser insuficiente...
- Após a primeira passada, o menor elemento está em $l[0]$
- Após a segunda, o segundo menor elemento está em $l[1]$
- E assim por diante...realizando $n - 1$ passadas

```
1 BubbleSort(l)
2   Seja n o tamanho de l
3   Para i = 0 até n - 2
4     Para j = n - 1 até i + 1
5       Se  $l[j - 1] > l[j]$ 
6         Troque  $l[j - 1]$  com  $l[j]$ 
```



Otimização

- Se fizermos uma passada e não houver trocas, a lista já está ordenada
 - Não havia posição j com $l[j - 1] > l[j]$

```
1 BubbleSort(l)
2   Seja n o tamanho de l
3   Para i = 0 até n - 2
4       trocou = Falso
5       Para j = n - 1 até i + 1
6           Se l[j - 1] > l[j]
7               Troque l[j - 1] com l[j]
8               trocou = Verdadeiro
9   Se trocou é Falso
10       Pare
```

Exercício

Implemente o BubbleSort em Python

BubbleSort funciona?

```
1 BubbleSort(l)
2   Seja n o tamanho de l
3   Para i = 0 até n - 2
4     Para j = n - 1 até i + 1
5       Se l[j - 1] > l[j]
6         Troque l[j - 1] com l[j]
```

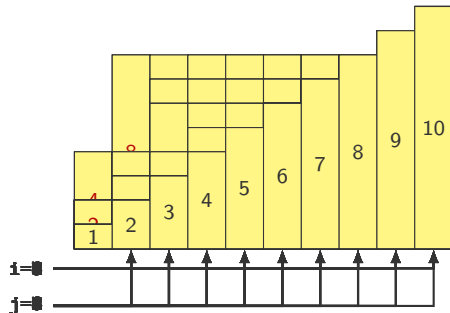
Claramente os passos de BubbleSort são simples e ele termina

- Como o menor valor de $l[i]$, $\dots$, $l[n - 1]$ será deslocado até $l[i]$ através de trocas
- O argumento que ele está correto é similar ao de SelectionSort

Ordenação por Inserção

- Se já temos $l[0], l[1], \dots, l[i-1]$ ordenados
- Inserimos $l[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $l[0], l[1], \dots, l[i]$ ordenados

```
1 InsertionSort(l)
2   Seja n o tamanho de l
3   Para i = 1 até n - 1
4     Para j = i até 1
5       Se  $l[j - 1] > l[j]$ 
6         Troque  $l[j - 1]$  com  $l[j]$ 
```



Otimização

- Não é necessário trocar sempre até o começo da lista
 - podemos parar quando o elemento está na posição correta
- Não é necessário fazer trocas
 - podemos ir deslocando os elementos para a direita
 - abrindo o espaço para o elemento a ser inserido

```
1 InsertionSort(l)
2   Seja n o tamanho de l
3   Para i = 1 até n - 1
4       aux = l[i]
5       j = i
6       Enquanto j > 0 e l[j - 1] > aux
7           l[j] = l[j - 1]
8           j = j - 1
9       l[j] = aux
```

Exercício

Implemente o InsertionSort em Python

InsertionSort funciona?

```
1 InsertionSort(l)
2   Seja n o tamanho de l
3   Para i = 1 até n - 1
4       aux = l[i]
5       j = i
6       Enquanto j > 0 e l[j - 1] > aux
7           l[j] = l[j - 1]
8           j = j - 1
9       l[j] = aux
```

Claramente os passos são simples e ele termina

Suponha que no início da iteração da linha 3 $l[0], \dots, l[i - 1]$ estão ordenados

- Isso é verdade no início da primeira iteração
 - $l[0]$ sozinho está ordenado

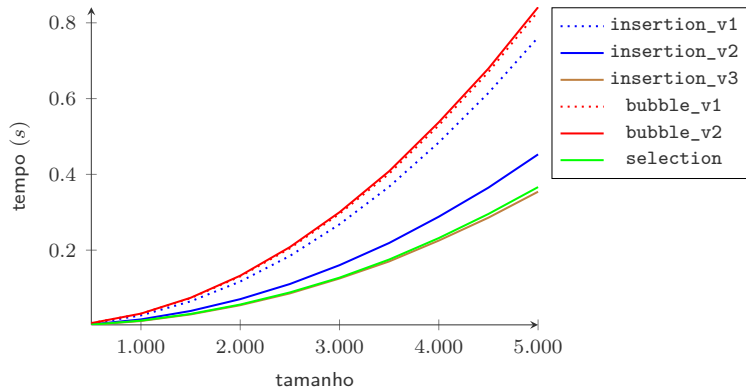
No final da iteração i , $l[0], \dots, l[i]$ estão ordenados?

- Sim, pois $l[i]$ é inserido de forma a manter a ordenação

Ou seja, no final da última iteração, a lista está ordenada

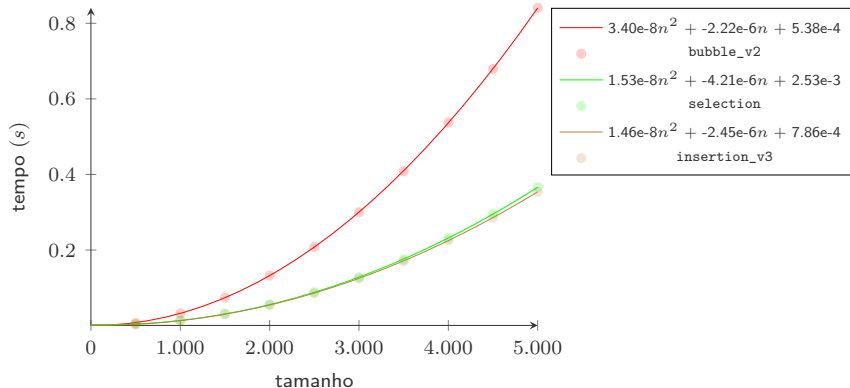
Experimento

- Listas de tamanho 500, 1000, ..., 5000
- Cada elemento escolhido aleatoriamente entre 0 e 1
- Tiramos a média do tempo de 10 execuções
- `insertion_v3` usa `for` com `break` ao invés de `while`



Experimento

- Listas de tamanho 500, 1000, ..., 5000
- Cada elemento escolhido aleatoriamente entre 0 e 1
- Tiramos a média do tempo de 10 execuções
- `insertion_v3` usa `for` com `break` ao invés de `while`



BUSCA

BUSCA: Dada uma lista **l** de números e um número **x**, encontrar, se existir, um índice **i** tal que **l[i] = x**.

Ideia do algoritmo:

- Percorrer a lista do início para o fim, procurando **x**
- Se encontrarmos, devolvemos o índice onde **x** está
- Se não encontrarmos, então **x** não está na lista

```
1 BuscaSequencial(l, x)
2     Seja n o tamanho de l
3     Para i = 0 até n - 1
4         Se l[i] = x
5             Devolva i
6     Devolva -1
```

BUSCA

BUSCA: Dada uma lista **l** de números e um número **x**, encontrar, se existir, um índice **i** tal que **l[i] = x**.

```
1 BuscaSequencial(l, x)
2     Seja n o tamanho de l
3     Para i = 0 até n - 1
4         Se l[i] = x
5             Devolva i
6     Devolva -1
```

BuscaSequencial é um algoritmo para **BUSCA**?

- Seus passos são simples o suficiente
- Ele claramente sempre termina
- Ele dá a resposta correta?
 - Se a 1ª ocorrência de **x** em **l** é a posição **k**, ele devolve **k**
 - Se **x** não está em **l**, a linha 4 sempre falha e devolvemos **-1**

Busca em uma lista ordenada

BUSCAORDENADA: Dada uma lista l ordenada de números e um número x , encontrar, se existir, um índice i tal que $l[i] = x$.

Já temos **BuscaSequencial**, mas ele não se aproveita do fato da lista estar ordenada...

Ideia: Digamos que, ao invés de olhar a primeira posição da lista, eu olhe uma posição m . O que pode acontecer?

- Eu posso descobrir que $l[m] = x$... Ótimo!
- Eu posso descobrir que $l[m] < x$ ou $l[m] > x$
 - Se $l[m] < x$, então se x estiver na lista, está da posição $m + 1$ para a frente...
 - Se $l[m] > x$, então se x estiver na lista, está da posição $m - 1$ para trás...

Qual m escolher?

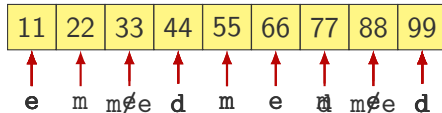
- $n / 2$ parece bom porque “descarto” metade da lista

Posso repetir a mesma ideia para a parte não descartada

Busca Binária

```
1 BuscaBinaria(l, x)
2   Seja n o tamanho de l
3   e = 0
4   d = n - 1
5   Enquanto e <= d
6     m = (e + d) / 2      # divisão inteira
7     Se l[m] == x
8       Devolva m
9     Se l[m] < x          # está para a direita
10      e = m + 1
11     Se l[m] > x          # está para a esquerda
12      d = m - 1
13   Devolva -1
```

Buscando por 33: Buscando por 80:



Exercício

Implemente a Busca Binária em Python

Busca Binária sempre termina?

```
1 BuscaBinaria(l, x)
2   Seja n o tamanho de l
3   e = 0
4   d = n - 1
5   Enquanto e <= d
6       m = (e + d) / 2      # divisão inteira
7       Se l[m] == x
8           Devolva m
9       Se l[m] < x          # está para a direita
10          e = m + 1
11       Se l[m] > x         # está para a esquerda
12          d = m - 1
13   Devolva -1
```

No começo $e = 0$ e $d = n - 1$ e a cada passo:

- m é um número maior ou igual a e
- m é um número menor ou igual a d
- Portanto, exatamente uma das opções ocorre:
 - ou e aumenta
 - ou d diminui

Busca Binária funciona?

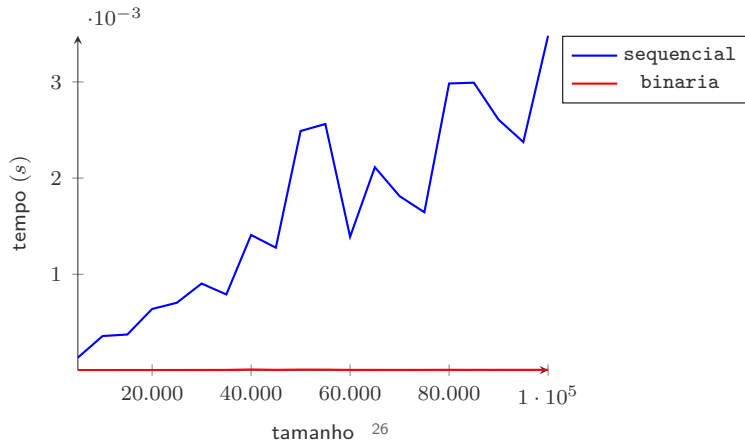
```
1 BuscaBinaria(l, x)
2   Seja n o tamanho de l
3   e = 0
4   d = n - 1
5   Enquanto e <= d
6       m = (e + d) / 2      # divisão inteira
7       Se l[m] == x
8           Devolva m
9       Se l[m] < x          # está para a direita
10          e = m + 1
11       Se l[m] > x          # está para a esquerda
12          d = m - 1
13   Devolva -1
```

Suponha que x esteja em $l[e]$, ..., $l[d]$ no começo da iteração, então:

- Se $l[m] == x$, o algoritmo dá a resposta correta
- Se $l[m] > x$, então x está em $l[e]$, ..., $l[m - 1]$
- Se $l[m] < x$, então x está em $l[m + 1]$, ..., $l[d]$

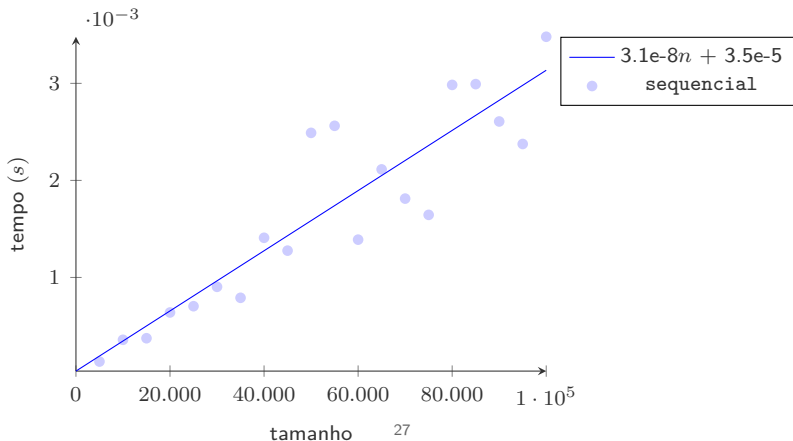
Experimento 1

- Listas de tamanho 5000, 10000, ..., 100000
- $l[i] = i$
- Tiramos a média do tempo de 20 execuções
- Escolhemos um i aleatório e procuramos por $l[i]$ em l



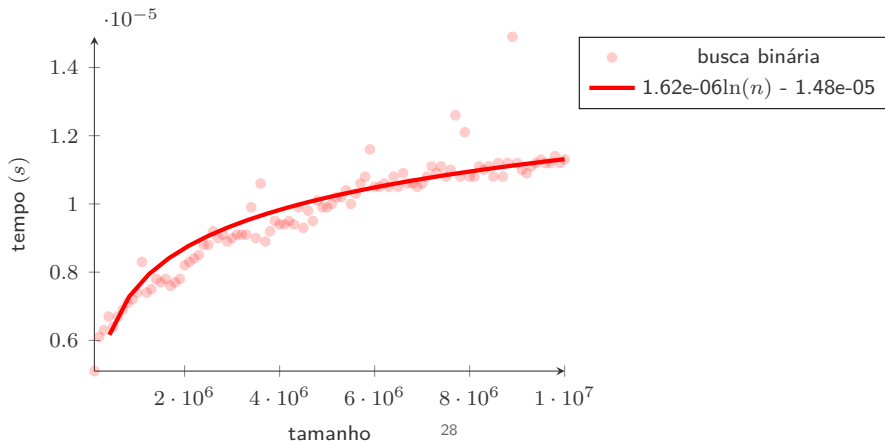
Experimento 1

- Listas de tamanho 5000, 10000, ..., 100000
- $l[i] = i$
- Tiramos a média do tempo de 20 execuções
- Escolhemos um i aleatório e procuramos por $l[i]$ em l



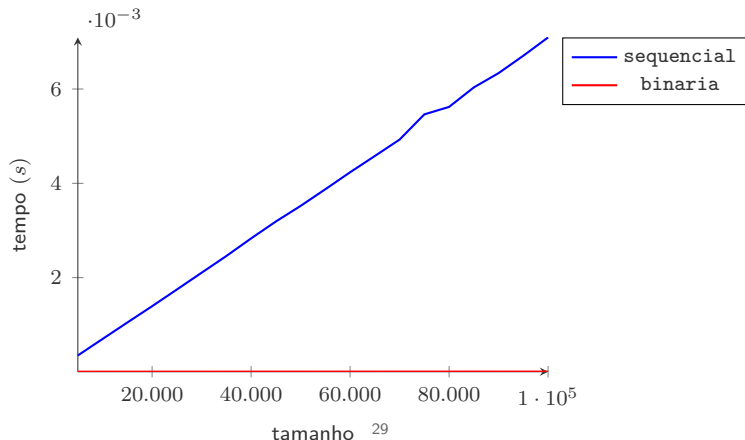
Experimento 1b — apenas busca binária

- Listas de tamanho 100000, 200000, ..., 10000000
- $l[i] = i$
- Tiramos a média do tempo de 100 execuções
- Escolhemos um i aleatório e procuramos por $l[i]$ em l



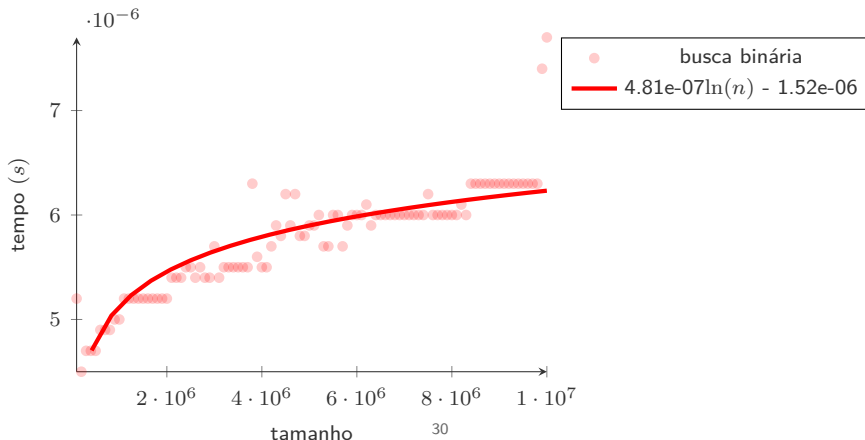
Experimento 2

- Listas de tamanho 5000, 10000, ..., 100000
- `l[i] = i`
- Tiramos a média do tempo de 20 execuções
- Buscamos por `len(l)` (que não está na lista)



Experimento 2b — apenas busca binária

- Listas de tamanho 100000, 200000, ..., 10000000
- $l[i] = i$
- Tiramos a média do tempo de 100 execuções
- Buscamos por $len(l)$ (que não está na lista)



Exercícios

1. Faça um algoritmo que dado uma lista ordenada e um elemento indica onde inserir esse elemento na lista para mantê-la ordenada...
 - Se o elemento já estiver na lista, você deve inseri-lo de forma a ser o primeiro do bloco repetido.
2. Use o algoritmo anterior para implementar uma busca em uma lista ordenada
3. Repita o primeiro exercício mas de forma que o elemento é o último do bloco repetido.
4. Faça um algoritmo que dado uma lista ordenada e um elemento, encontra o intervalo que contém todos os elementos iguais a ele na lista.