Programação Script

Busca e Ordenação

Aula 13

Prof. Felipe A. Louza

Roteiro

Roteiro

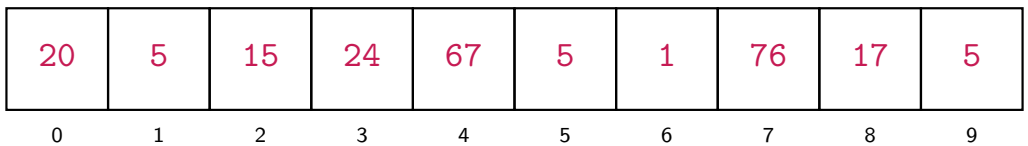
O Problema da Busca

O **problema da busca** é um dos problemas básicos em Computação e possui diversas aplicações.

- Basicamente, um **algoritmo de busca** deve verificar se uma dada informação ocorre ou não em uma coleção de elementos.

O Problema da Busca

Vamos assumir que os dados estão em uma `lista` de `inteiros`:



- Vamos considerar:
  1. `find(x):` x ocorre na lista?
  2. `count(x):` quantas vezes x ocorre na lista?
  3. `locate(x):` em quais posições x ocorre na lista?

O Problema da Busca

Os dados poderiam estar em `lista` de registros (`tuplas`).

- Os algoritmos seriam os mesmos

```
>>> lista = [(21, "João"), (5, "Maria"), (34, "Mario"), (44,"José"), \
             (5, "Luiza"), (52, "Antonio")]
```

Precisariamos definir uma chave de busca, ex: `tupla[0]`.

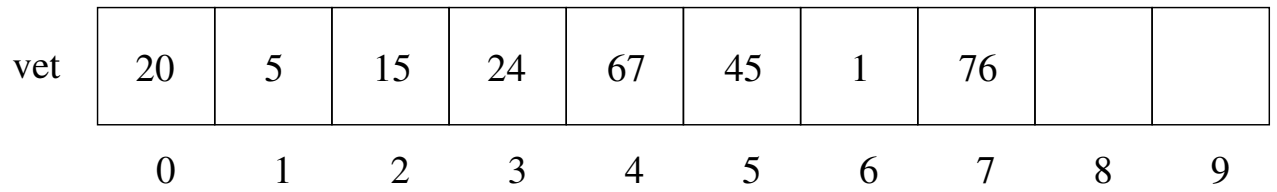
A busca sequencial

A busca sequencial é o algoritmo mais simples de busca:

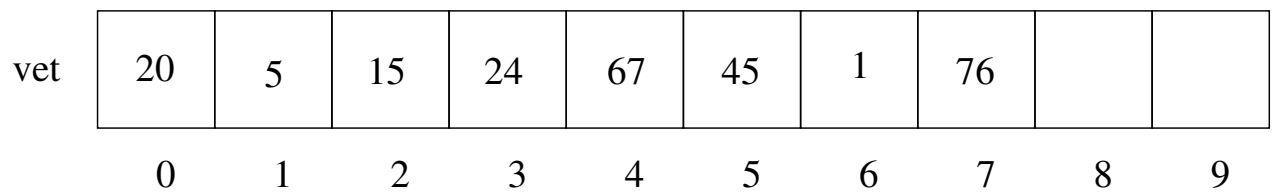
- Percorre toda lista comparando a chave com o valor de cada posição.
- Se for igual para alguma posição, a busca encontrou um elemento.

A busca sequencial

chave = 45 tam = 8



chave = 100 tam = 8



No primeiro exemplo, a busca encontra a chave em 5, no segundo exemplo a busca não encontra nada.

A busca sequencial

chave = 45 tam = 8

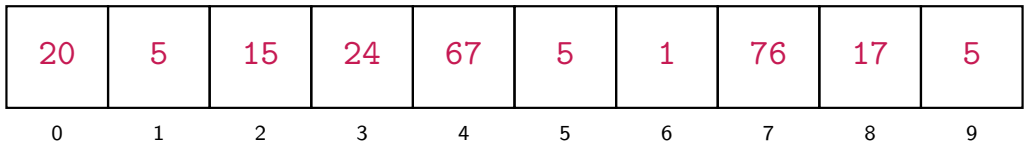
vet	20	5	15	24	67	45	1	76		
	0	1	2	3	4	5	6	7	8	9

chave = 100 tam = 8

vet	20	5	15	24	67	45	1	76		
	0	1	2	3	4	5	6	7	8	9

No primeiro exemplo, a busca encontra a chave em 5, no segundo exemplo a busca não encontra nada.

A busca sequencial

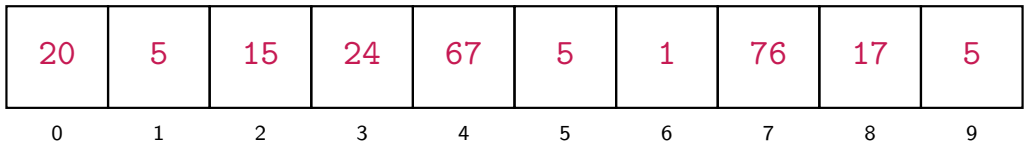


- Operação `find(x)`:

```
1 def find(lista, x):
2   for i in range(len(lista)):
3     if (lista[i] == x):
4       return True
5   return False
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> find(lista, 24)
3 True
4 >>> find(lista, 100)
5 False
```

A busca sequencial

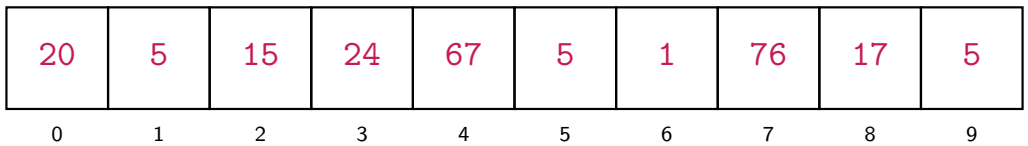


- Operação `find(x)`:

```
1 def find(lista, x):
2   for i in range(len(lista)):
3     if (lista[i] == x):
4       return True
5   return False
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> find(lista, 24)
3 True
4 >>> find(lista, 100)
5 False
```

A busca sequencial

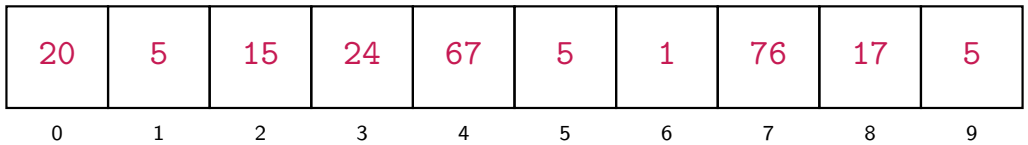


- Operação `count(x)`:

```
1 def count(lista, x):
2   total = 0
3   for i in range(len(lista)):
4     if (lista[i] == x):
5       total += 1
6   return total
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> count(lista, 5)
3 3
4 >>> count(lista, 100)
5 0
```

A busca sequencial



- Operação `count(x)`:

```
1 def count(lista, x):
2   total = 0
3   for i in range(len(lista)):
4     if (lista[i] == x):
5       total += 1
6   return total
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> count(lista, 5)
3 3
4 >>> count(lista, 100)
5 0
```

A busca sequencial

Mas o **Python** já possuí um método em `listas` que faz a busca e counta as ocorrências: `count()`.

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> lista.count(5)
3 3
```

- O problema é que esse método não funciona, por exemplo, para `listas de tuplas`:

```
1 >>> lista = [(21, "João"), (5, "Maria"), (34, "Mario"), (44,"José"),\
2             (5, "Luiza"), (52, "Antonio")]
3 >>>
4 >>> lista.count(5)
5 0
```

Além disso, o foco do curso é a lógica de programação (desenvolver algoritmos).

A busca sequencial

Mas o **Python** já possuí um método em `listas` que faz a busca e counta as ocorrências: `count()`.

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> lista.count(5)
3 3
```

- O problema é que esse método não funciona, por exemplo, para `listas de tuplas`:

```
1 >>> lista = [(21, "João"), (5, "Maria"), (34, "Mario"), (44,"José"),\
2              (5, "Luiza"), (52, "Antonio")]
3 >>>
4 >>> lista.count(5)
5 0
```

Além disso, o foco do curso é a lógica de programação (desenvolver algoritmos).

A busca sequencial

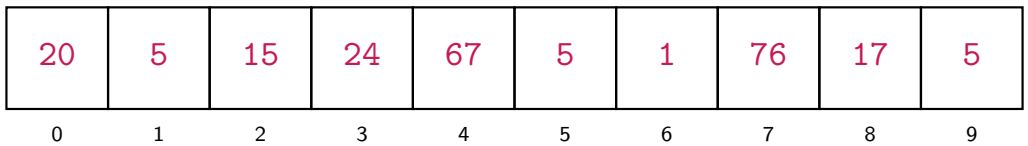
Na verdade existe uma possível solução utilizando o método `count()`:

```
>>> lista = [(21, "João"), (5, "Maria"), (34, "Mario"), (44,"José"),\
             (5, "Luiza"), (52, "Antonio")]
```

- Envolve criar uma outra lista:

```
>>> [x[0] for x in lista].count(5)
2
```

A busca sequencial

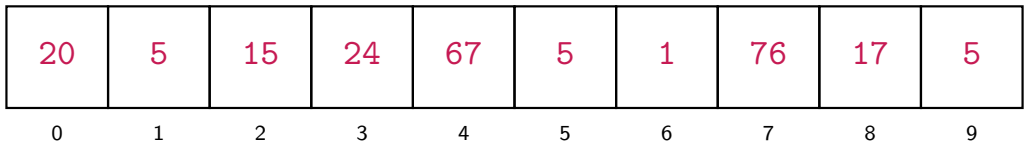


- Operação `locate(x)`:

```
1 def locate(lista, x):
2   res = []
3   for i in range(len(lista)):
4     if (lista[i] == x):
5       res.append(i)
6   return False if len(res)==0 else res
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> locate(lista, 5)
3 [1, 5, 9]
4 >>> locate(lista, 100)
5 False
```

A busca sequencial



- Operação `locate(x)`:

```
1 def locate(lista, x):
2   res = []
3   for i in range(len(lista)):
4     if (lista[i] == x):
5       res.append(i)
6   return False if len(res)==0 else res
```

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> locate(lista, 5)
3 [1, 5, 9]
4 >>> locate(lista, 100)
5 False
```

A busca sequencial

Da mesma forma, existe um *método similar* em **Python**: `index()`.

- Esse método encontra apenas o primeiro elemento, e retorna um erro quando não encontra nada.

```
>>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
>>> lista.index(5)
1
>>> lista.index(100)
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    lista.index(100)
ValueError: 100 is not in list
```

  - Também não funciona, por exemplo, para `listas de tuplas`:

A busca sequencial

Da mesma forma, existe um *método similar* em **Python**: `index()`.

- Esse método encontra apenas o primeiro elemento, e retorna um erro quando não encontra nada.

```
>>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
>>> lista.index(5)
1
>>> lista.index(100)
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    lista.index(100)
ValueError: 100 is not in list
```

  - Também não funciona, por exemplo, para `listas de tuplas`:

Roteiro

Custo computacional

Podemos dizer que algoritmo de busca sequencial é eficiente?

- Como estimar o seu tempo de execução?

Podemos fazer um estudo análitico da **quantidade de recursos** que qualquer algoritmo utiliza (tempo, memória, banda de rede, ...)

- Vamos ver (de forma simplificada) como avaliar o tempo de execução de um algoritmo:
  1. quantas operações o algoritmo executa?
  2. quanto tempo cada operação demora?

Custo computacional

Podemos dizer que algoritmo de busca sequencial é eficiente?

- Como estimar o seu tempo de execução?

Podemos fazer um estudo análitico da **quantidade de recursos** que qualquer algoritmo utiliza (tempo, memória, banda de rede, ...)

- Vamos ver (de forma simplificada) como avaliar o tempo de execução de um algoritmo:
  1. quantas operações o algoritmo executa?
  2. quanto tempo cada operação demora?

Custo computacional

Podemos dizer que algoritmo de busca sequencial é eficiente?

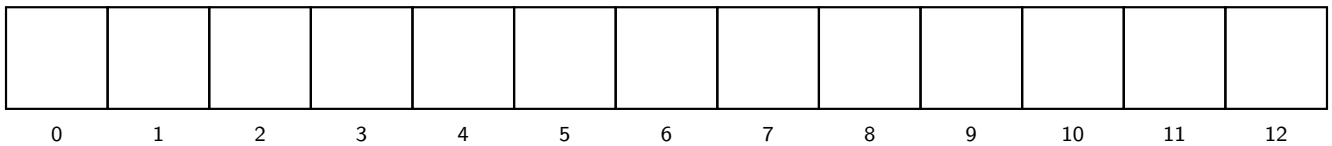
- Como estimar o seu tempo de execução?

Podemos fazer um estudo análitico da **quantidade de recursos** que qualquer algoritmo utiliza (tempo, memória, banda de rede, ...)

- Vamos ver (de forma simplificada) como avaliar o tempo de execução de um algoritmo:
  1. quantas operações o algoritmo executa?
  2. quanto tempo cada operação demora?

Eficiência dos Algoritmos

Na busca sequencial existem três possibilidades para a operação `find()`:



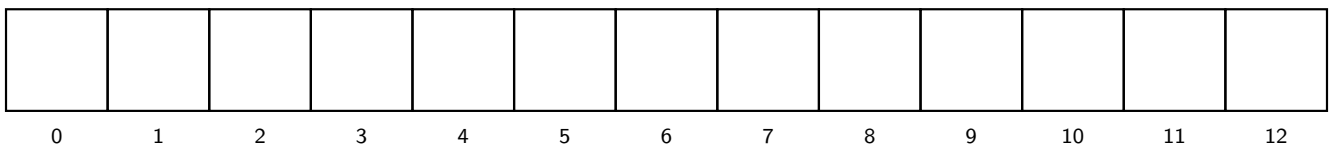
- No melhor caso, a busca estará na posição 0. Portanto teremos uma única comparação.
- No pior caso, a busca está no último elemento ou **não pertence** à lista, e portanto acessaremos todas as **n** posições.
- No caso médio, se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será

$$(n+1)/2$$

na média, onde **n** é o tamanho da lista.

Custo computacional

Vamos supor que temos uma lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.

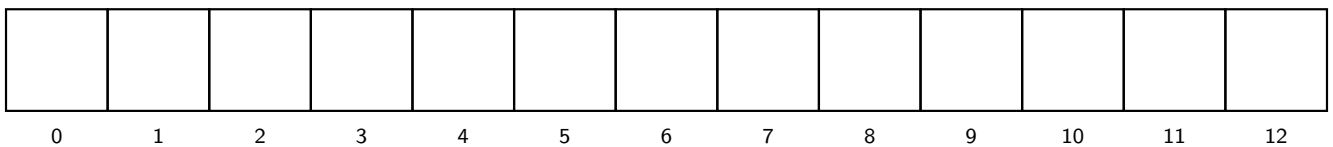


Vamos assumir que cada comparação é feita em 1 milisegundo (10^{-3}).

1. No pior caso: 2000 s $\approx$ 33 minutos
2. No caso médio: 1000 s $\approx$ 16 minutos $\leftarrow$ Muito ruim!!

Custo computacional

Vamos supor que temos uma lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.

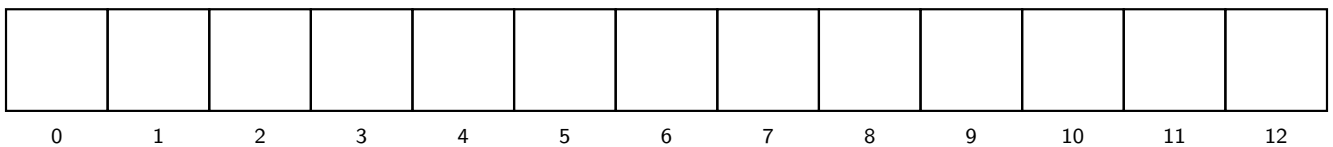


Vamos assumir que cada comparação é feita em 1 milisegundo (10^{-3}).

1. No pior caso: 2000 s $\approx$ 33 minutos
2. No caso médio: 1000 s $\approx$ 16 minutos $\leftarrow$ Muito ruim!!

Custo computacional

Vamos supor que temos uma lista telefonica com **2 milhões** de registros do tipo `(nome, telefone)`.



Vamos assumir que cada comparação é feita em 1 milisegundo (10^{-3}).

1. No pior caso: 2000 s $\approx$ 33 minutos
2. No caso médio: 1000 s $\approx$ 16 minutos $\leftarrow$ Muito ruim!!

Custo computacional

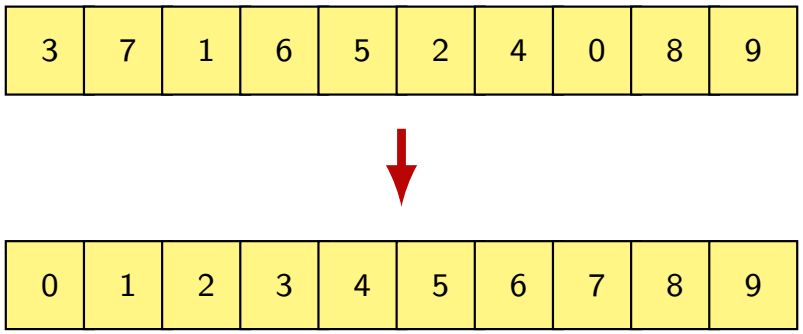
Na próxima aula vamos ver um algoritmo de busca muito mais eficiente.

- Para isso, precisamos antes falar de **Ordenação**.

Roteiro

Ordenação

Queremos ordenar uma `lista`:



Vamos assumir que os dados estão em uma `lista` de `inteiros`

- Os algoritmos seriam os mesmos caso tivessemos uma `lista` de registros.
  - O valor usado para a ordenação é a chave de ordenação
  - Podemos até desempatar por outros campos

Ordenação

Existem diferentes algoritmos para **a ordenação**.

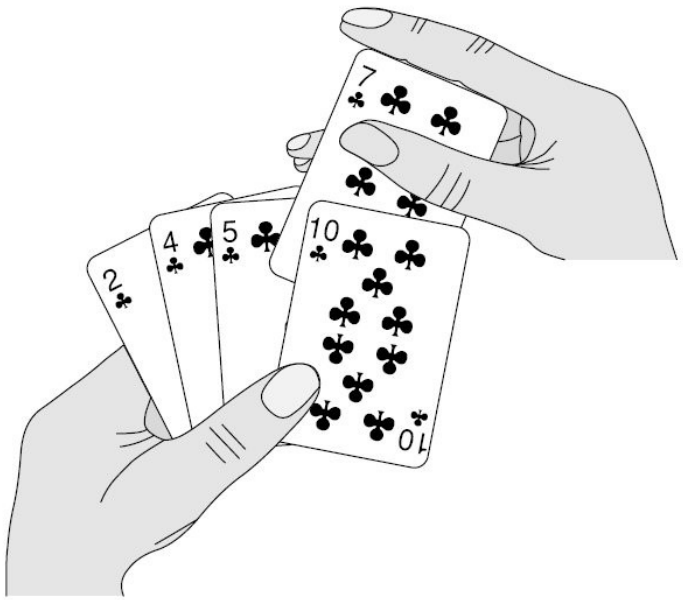
- Inclusive, o **Python** tem uma função pronta para ordenar

```
1 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
2 >>> lista.sort()
3 >>> lista
4 [1, 5, 5, 5, 15, 17, 20, 24, 67, 76]
```

- Mais uma vez, estamos interessados em aprender algoritmos mais do que em utilizar a linguagem.
- Vamos ver os algoritmos: *Selection Sort* e *Bubble Sort*.

Ordenação por Seleção

Ordenação por Seleção:



Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9         menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9         menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

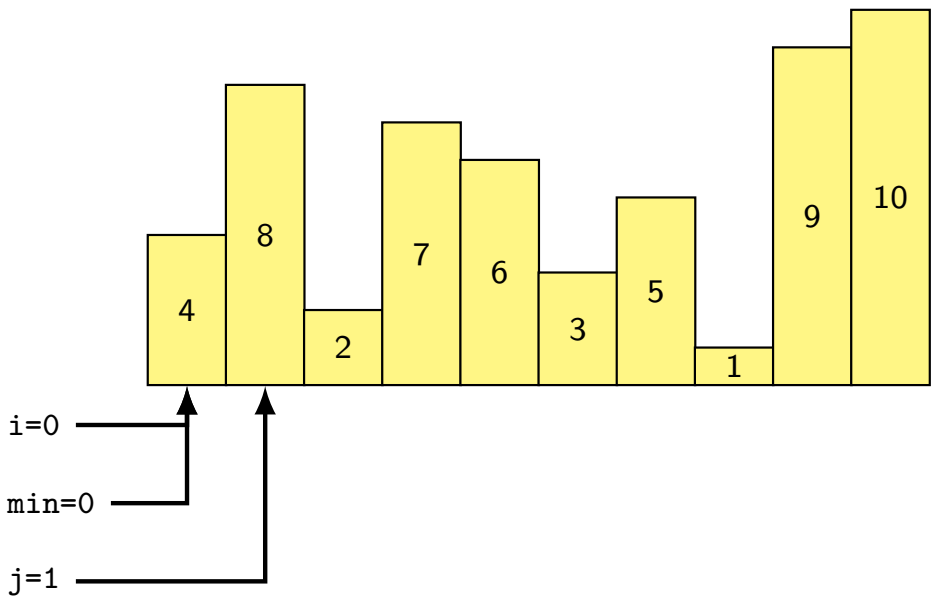
```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

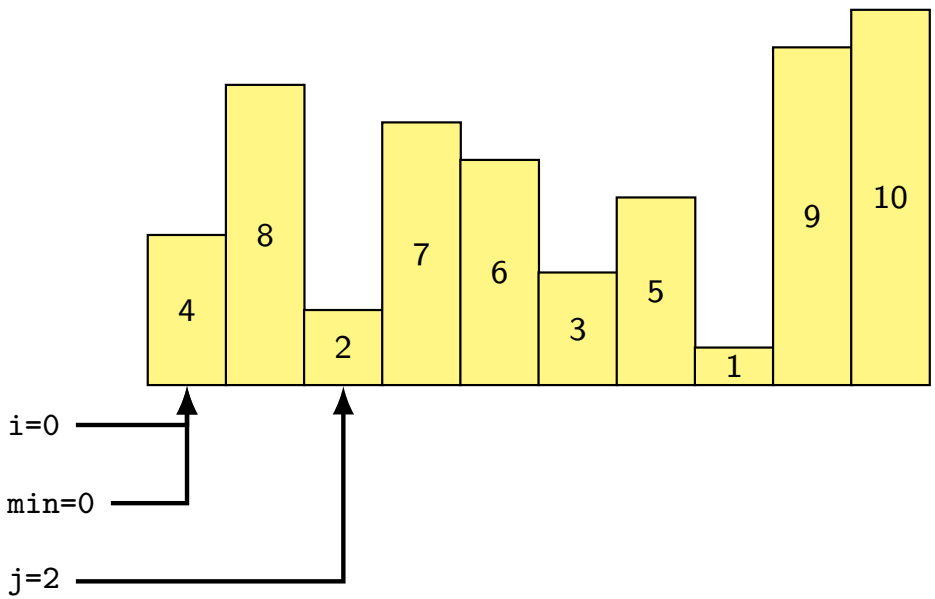


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

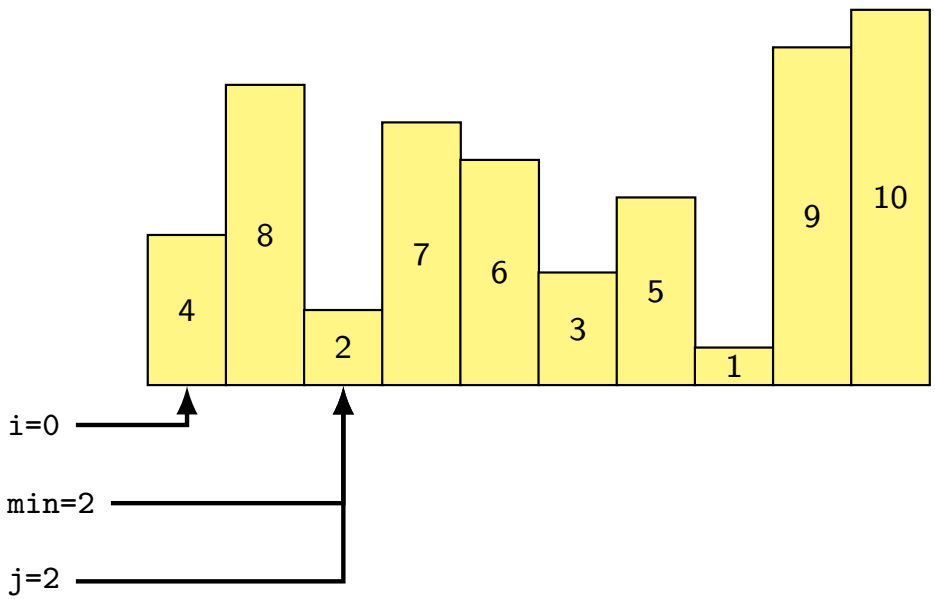


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

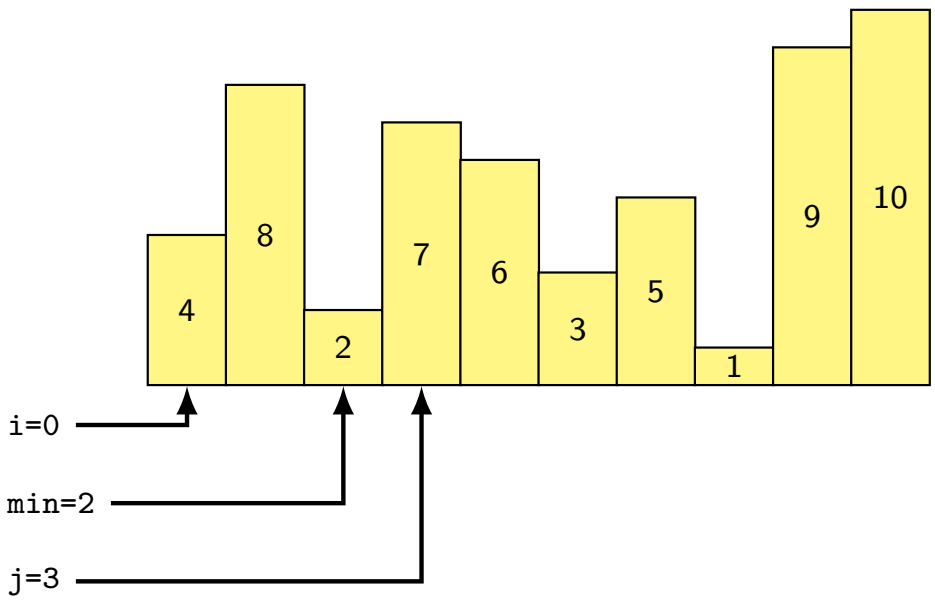


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

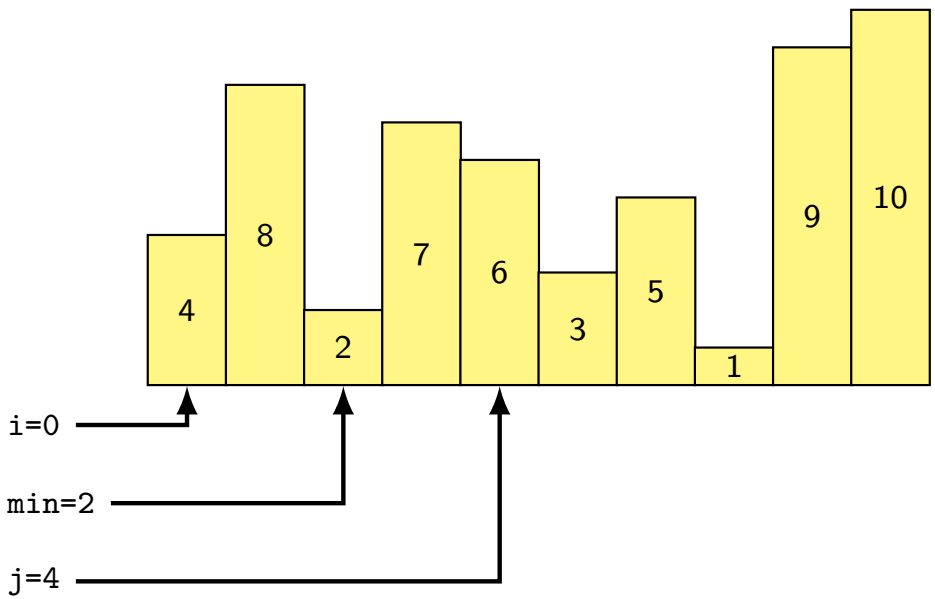


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

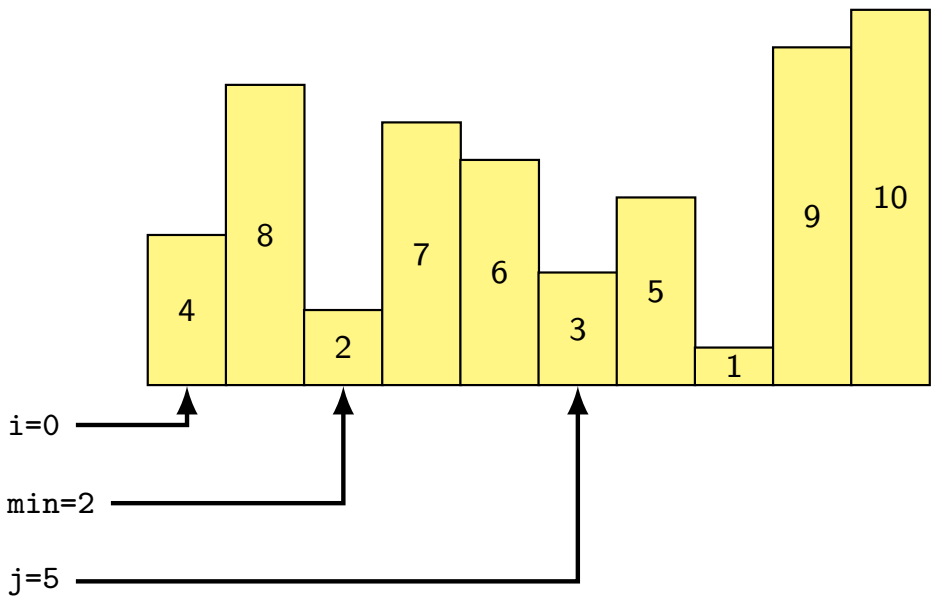


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

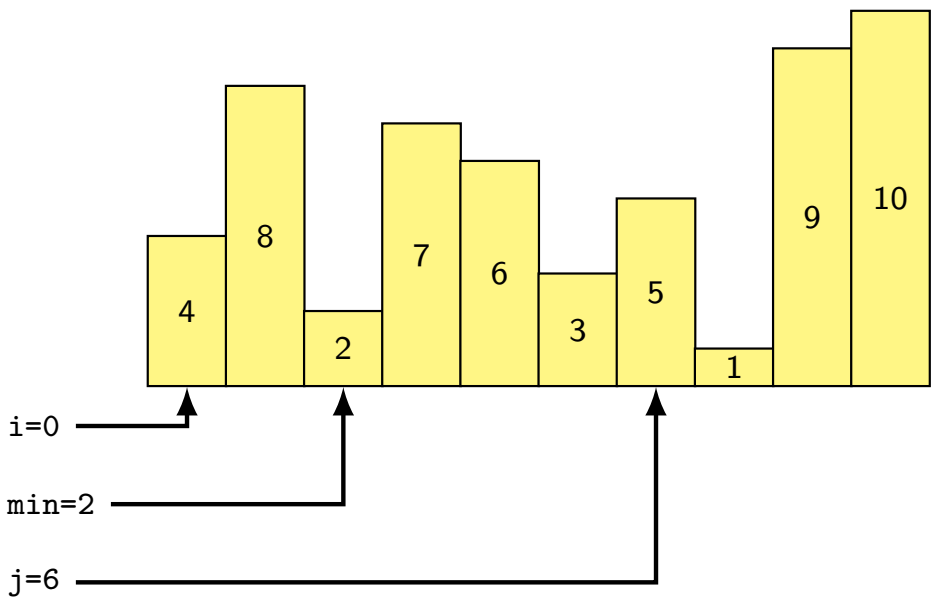


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

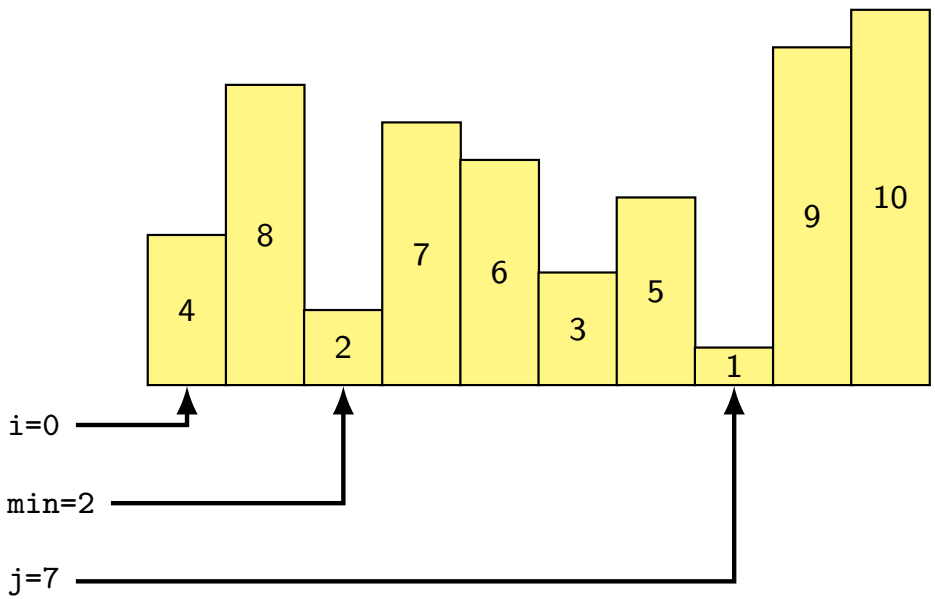


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

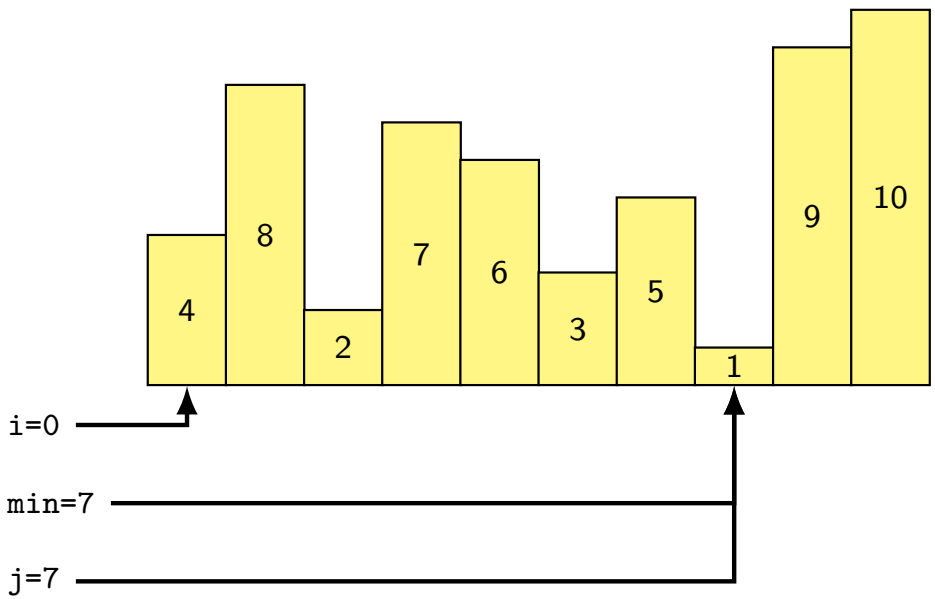


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

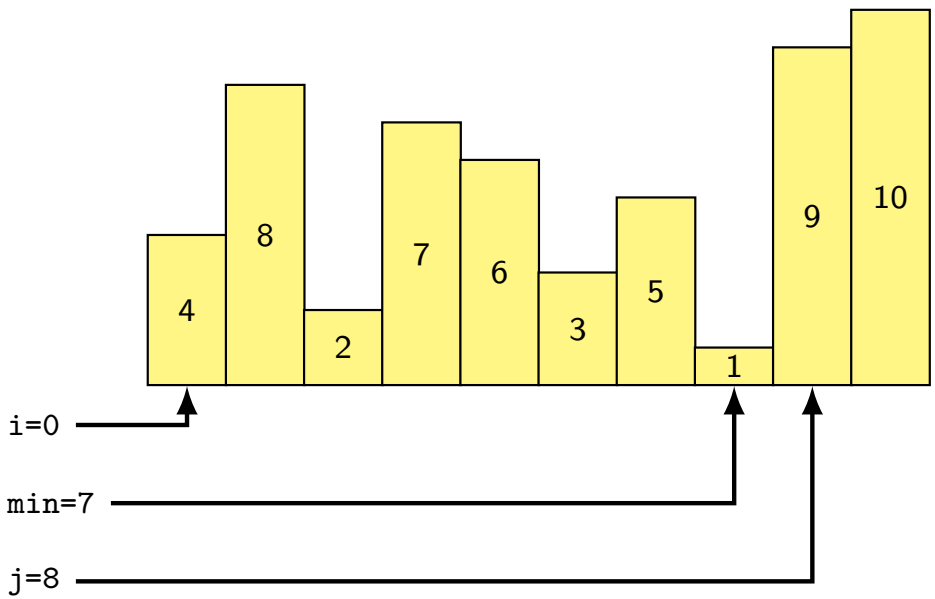


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

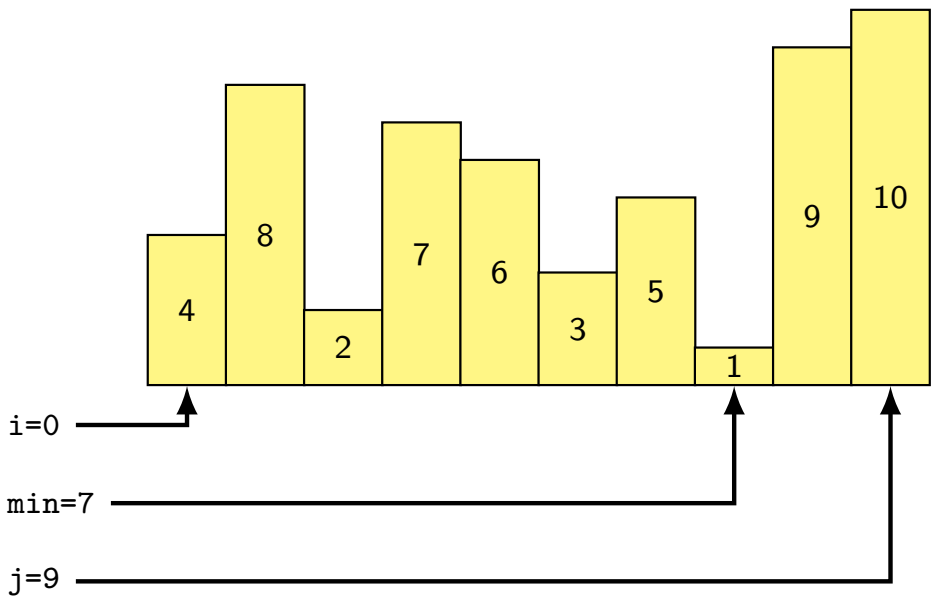


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

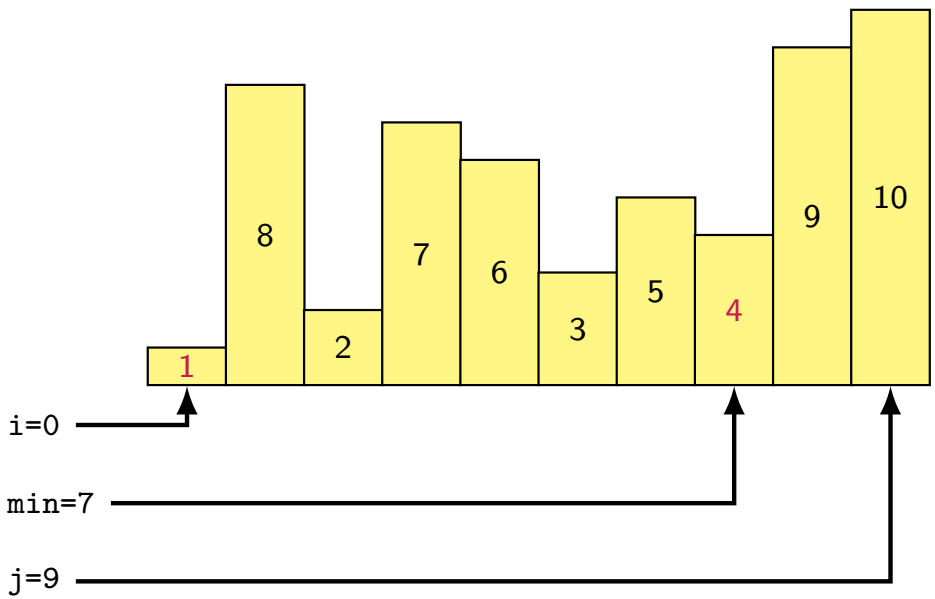


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

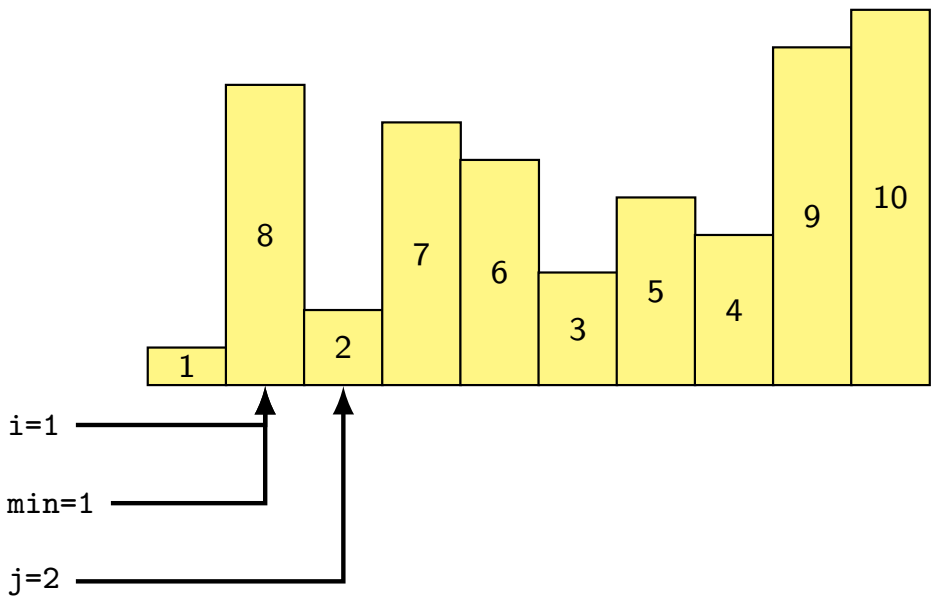


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

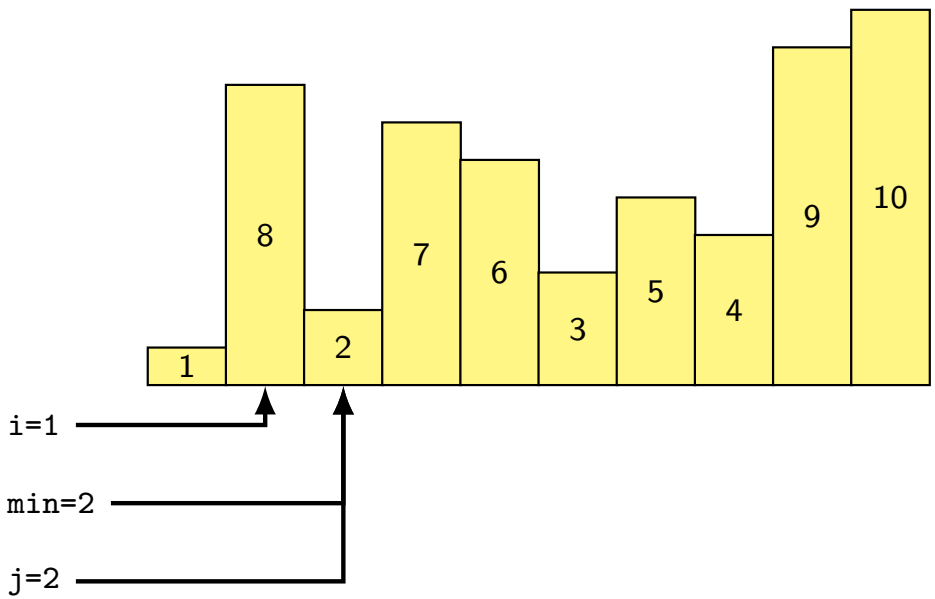


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

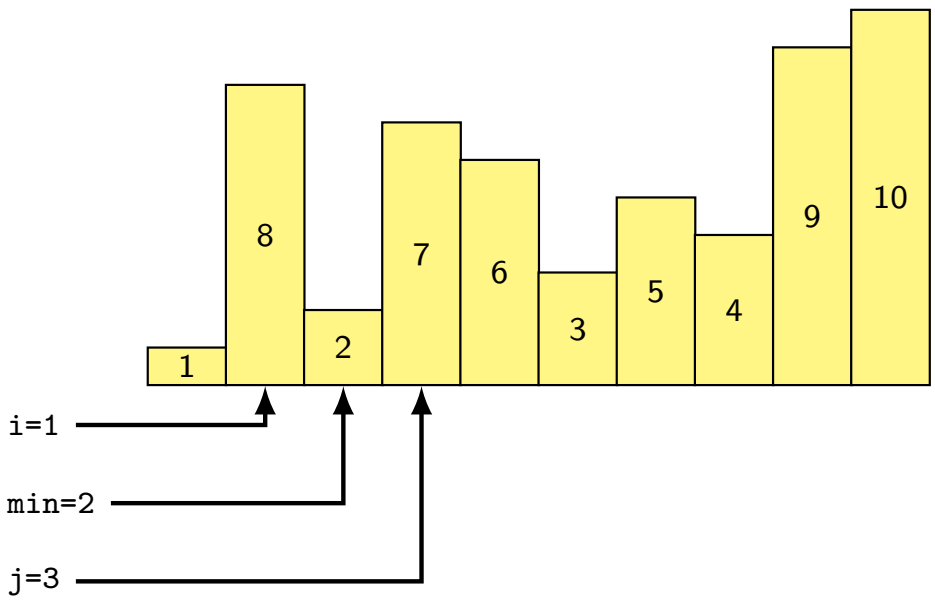


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

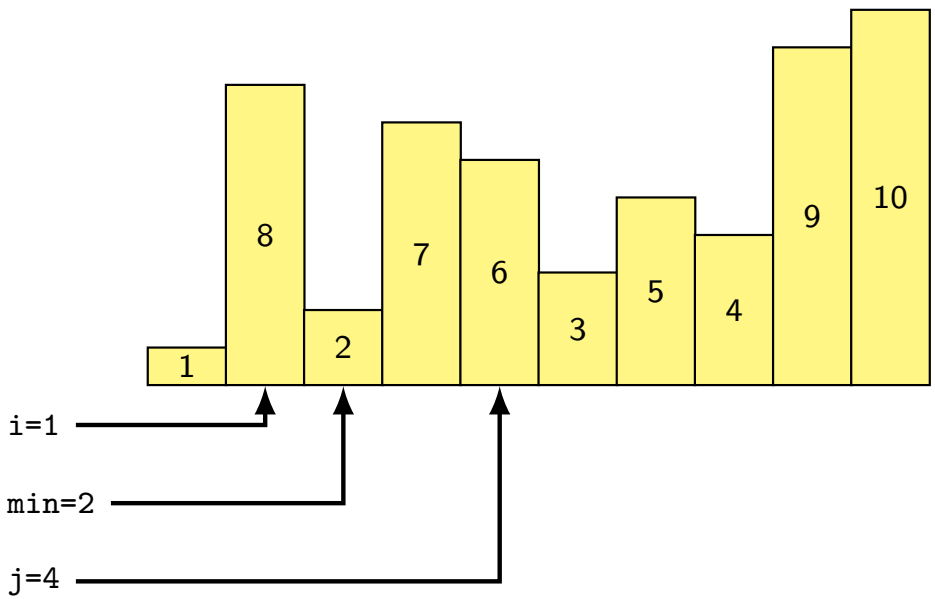


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

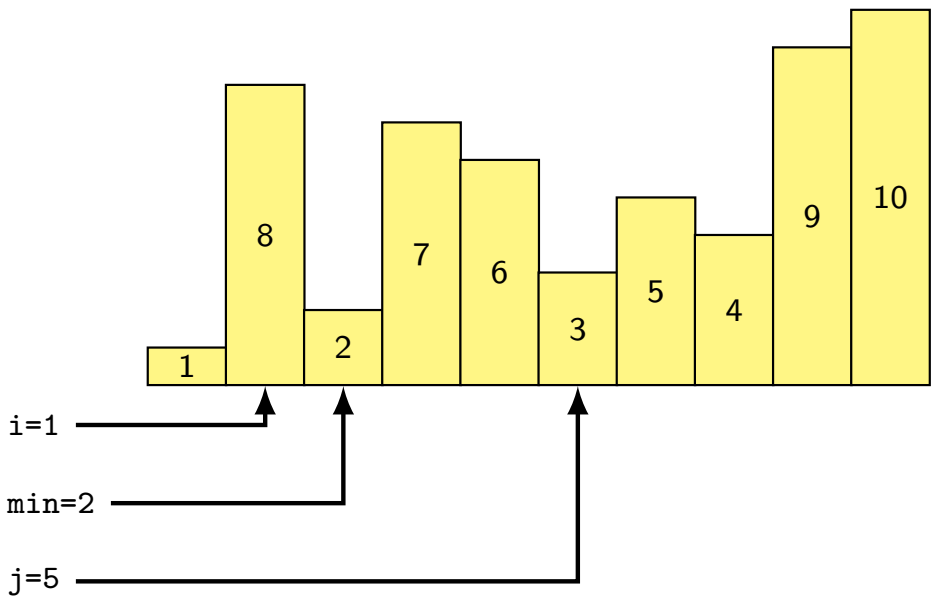


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

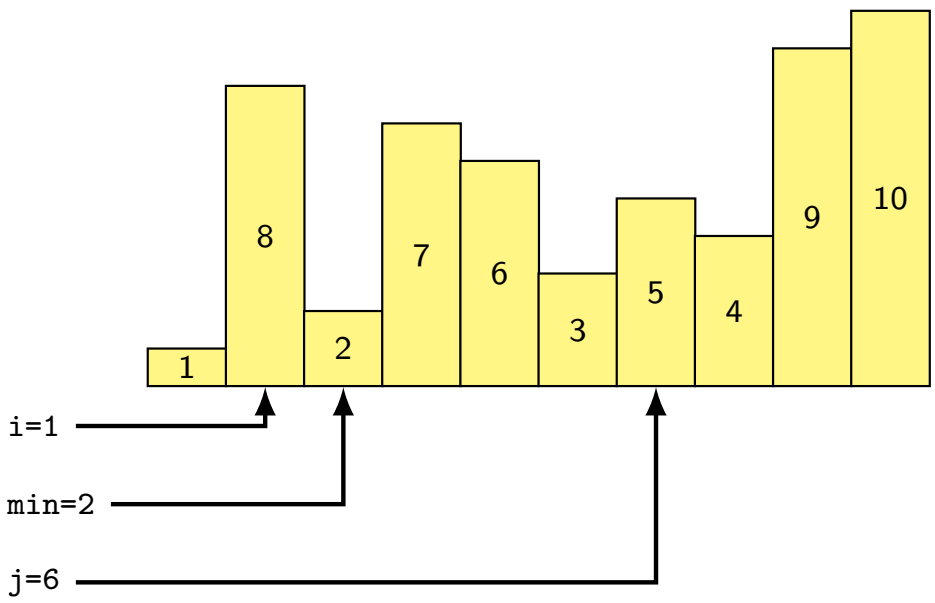


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

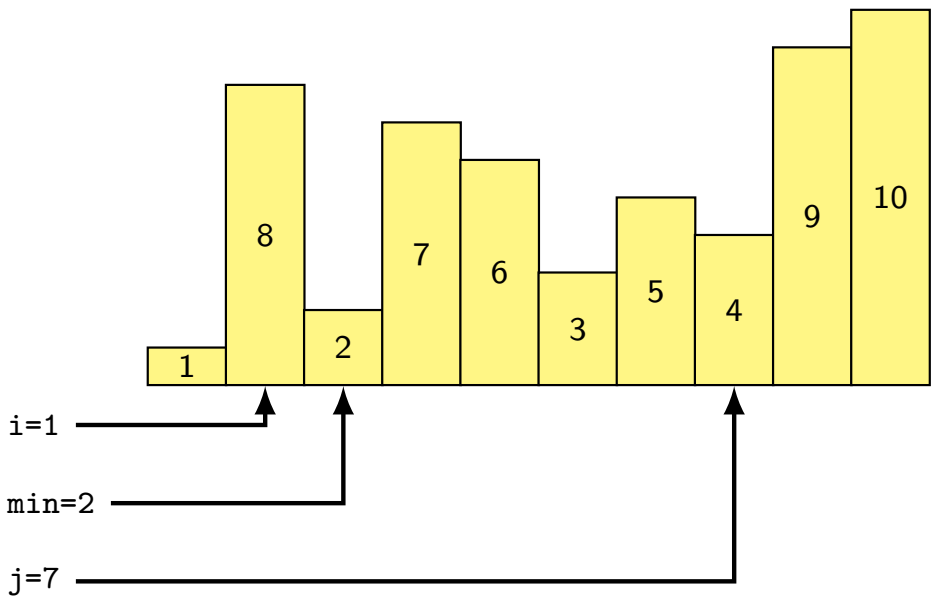


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

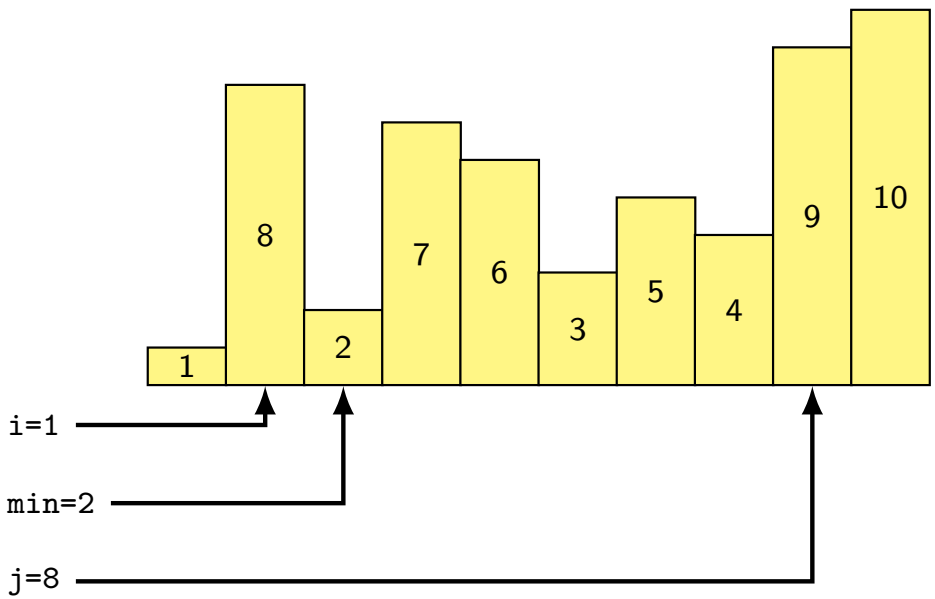


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

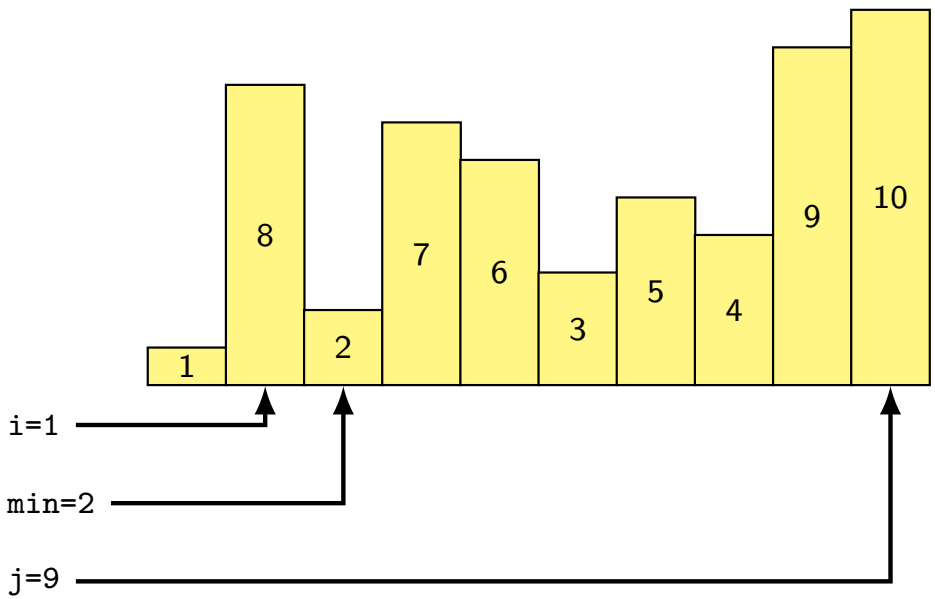


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

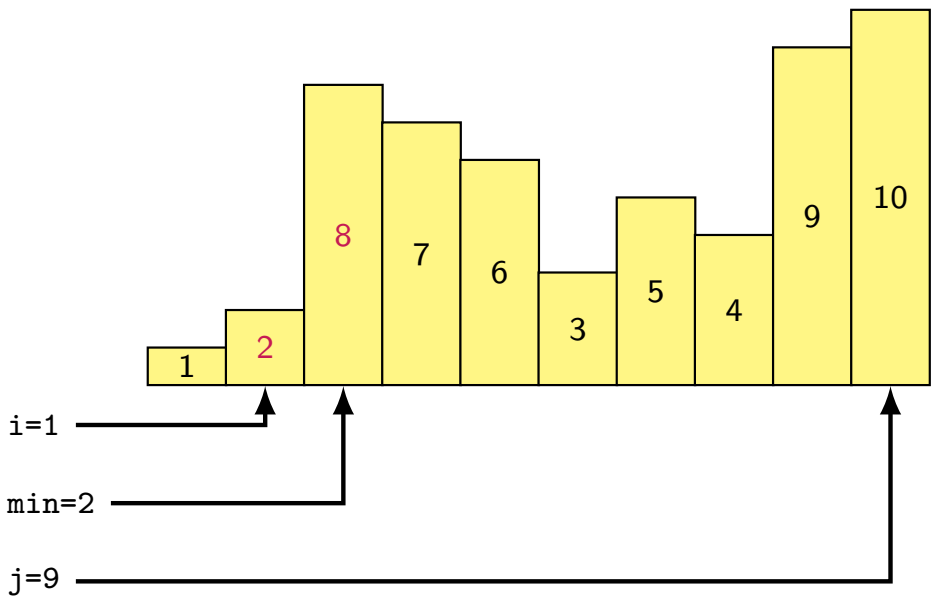


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

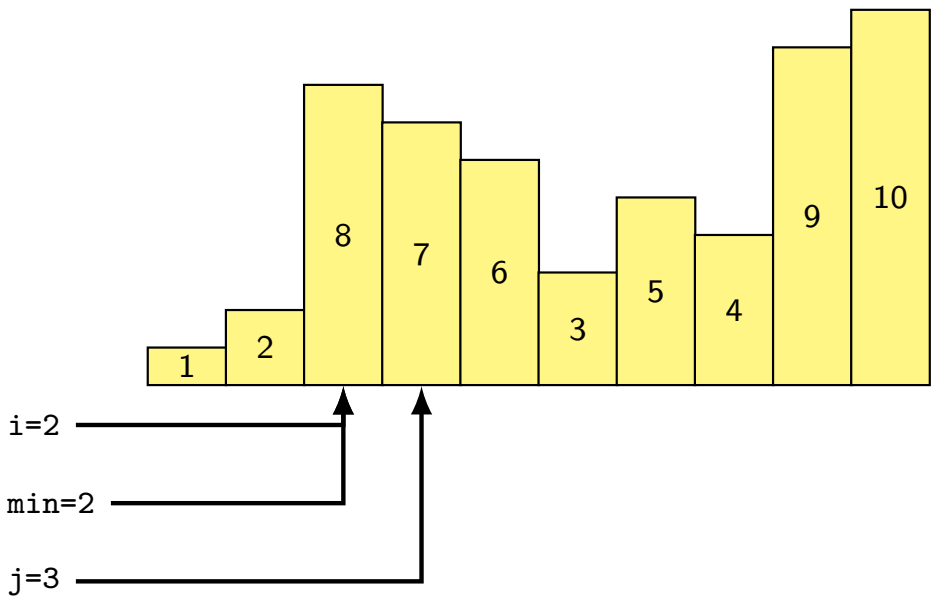


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

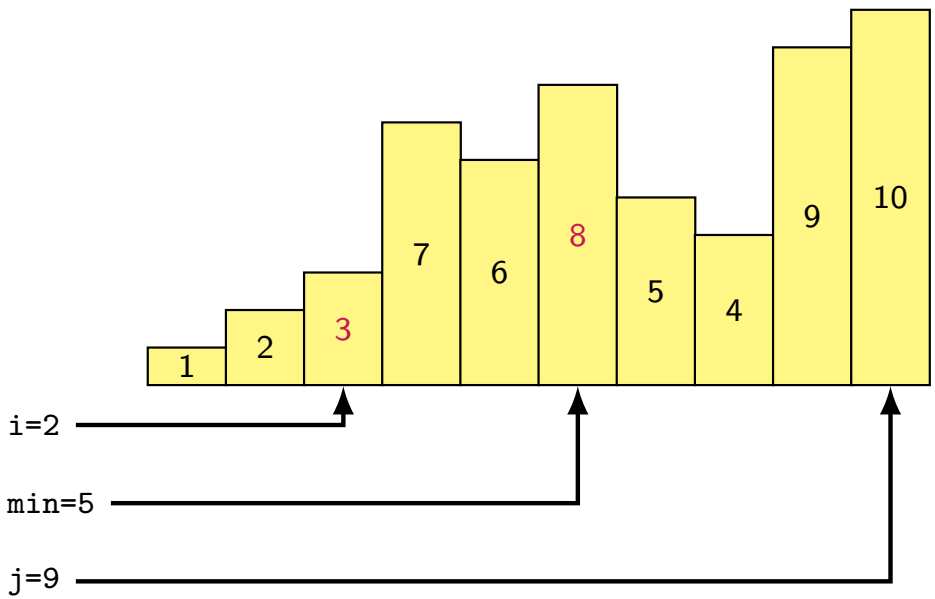


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

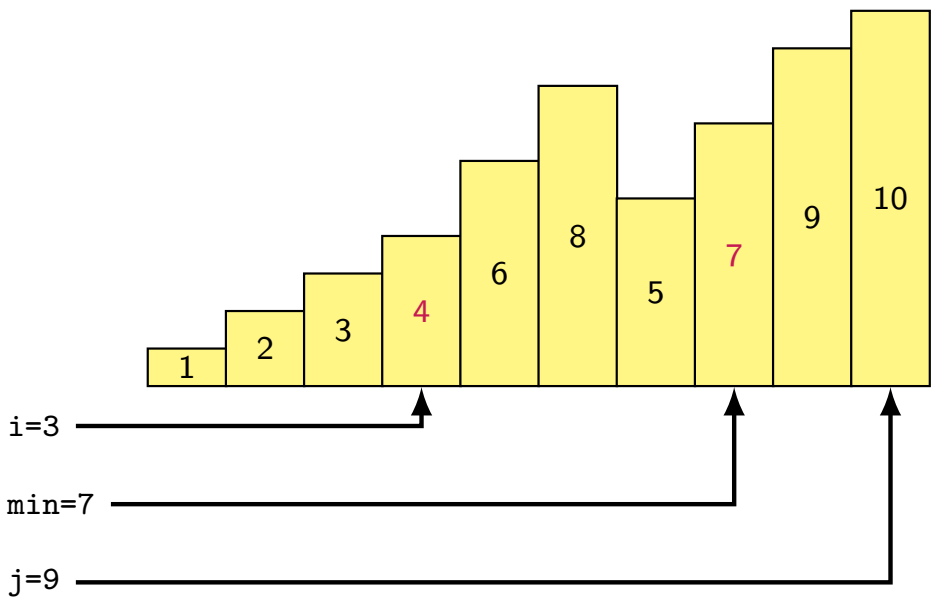


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

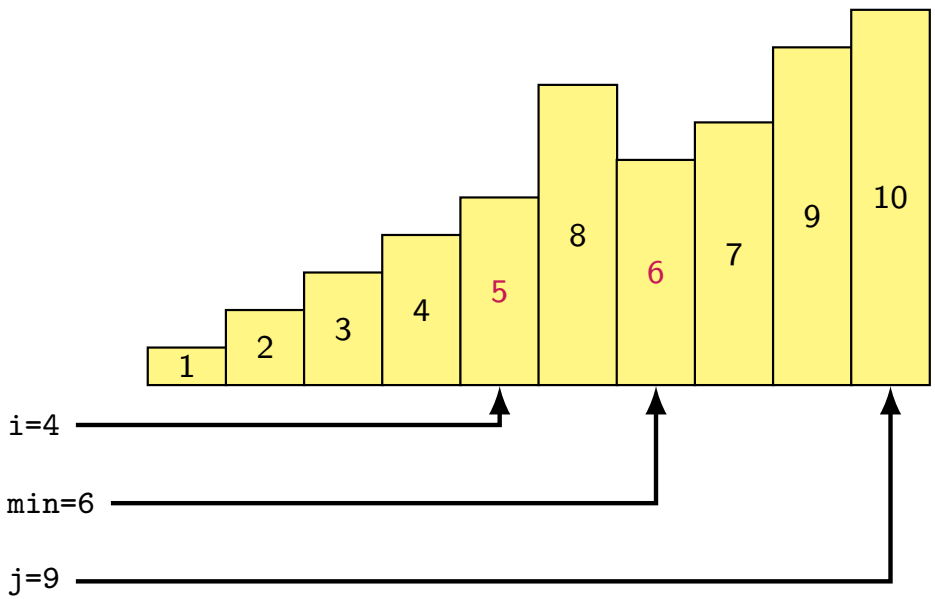


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

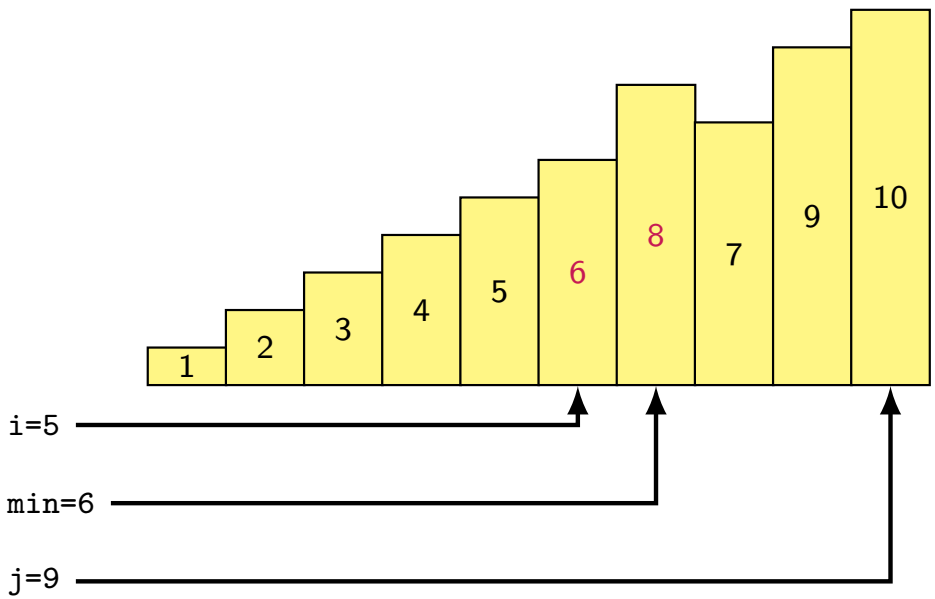


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

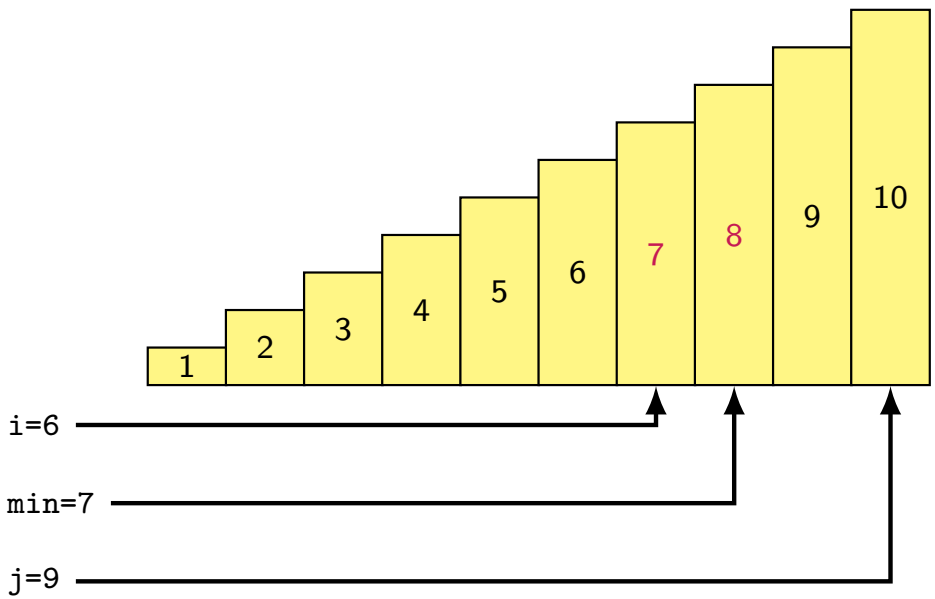


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

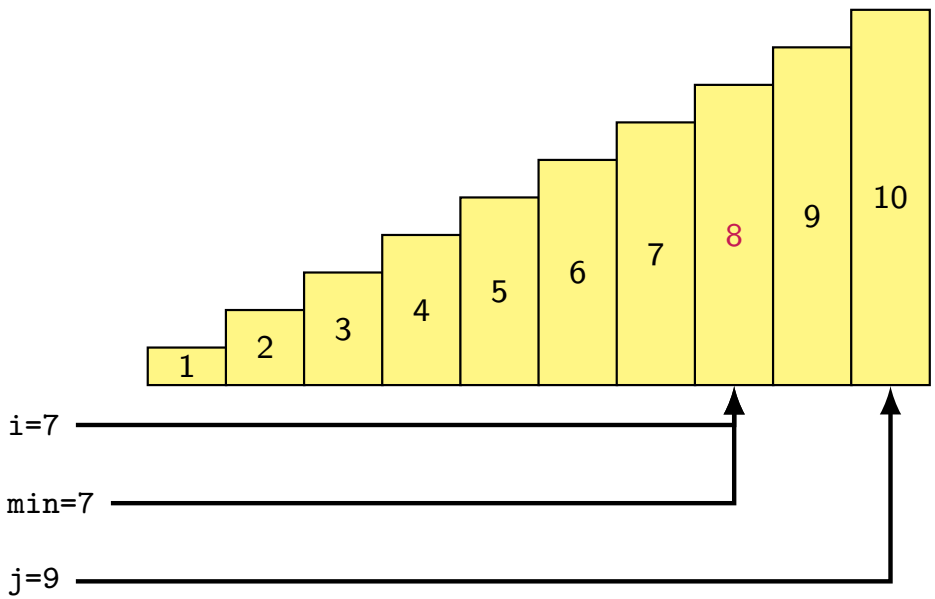


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

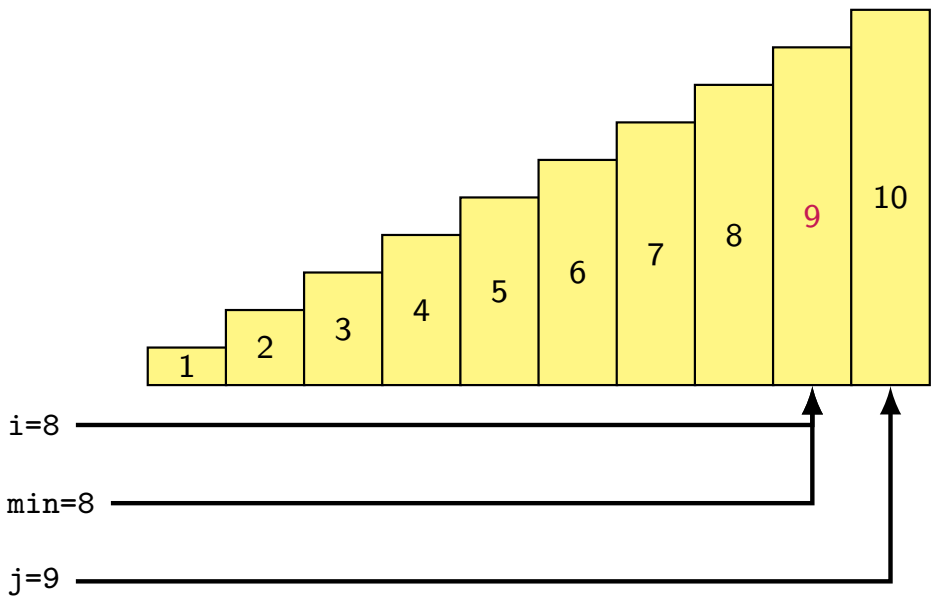


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```

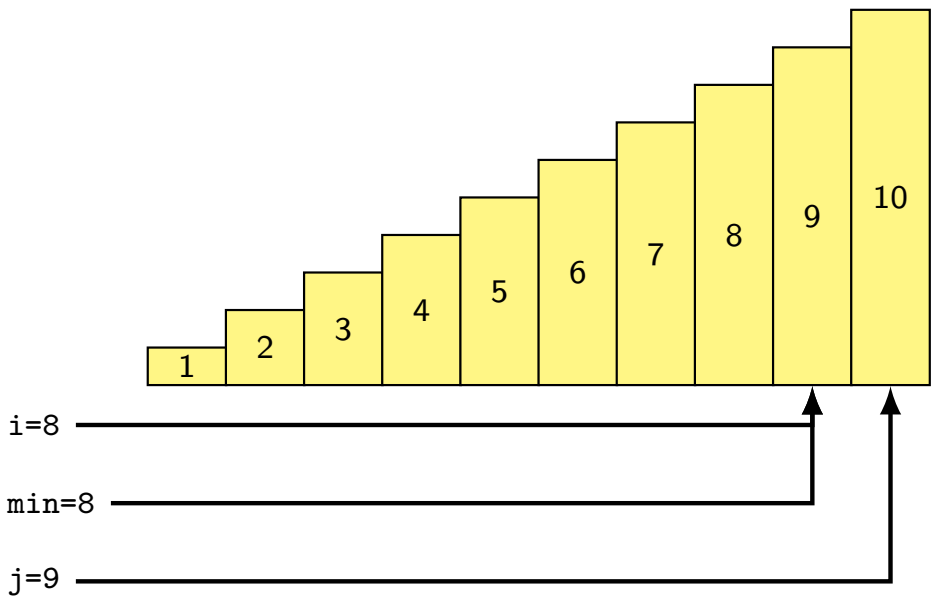


Ordenação por Seleção

Ideia do algoritmo:

- No passo `i` procuramos o i-ésimo menor elemento e trocamos com `lista[i]`.

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def selection_sort(lista):
5   for i in range(len(lista)-1):
6     menor_pos = i
7     for j in range(i+1, len(lista)):
8       if(lista[j]<lista[menor_pos]):
9           menor_pos = j
10    swap(lista, i, menor_pos)
11  return lista
```



Ordenação por Seleção

```
1  def swap(l, i, j)
2    l[i], l[j] = l[j], l[i]
3
4  def selection_sort(lista):
5    for i in range(len(lista)-1):
6      menor_pos = i
7      for j in range(i+1, len(lista)):
8        if(lista[j]<lista[menor_pos]):
9            menor_pos = j
10     #swap
11     swap(lista, i, menor_pos)
12   return lista
```

- Número de comparações:

$$(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = \frac{n^2-n}{2}$$

- Número de trocas: $n-1$

Ordenação por Seleção

```
1  def swap(l, i, j)
2    l[i], l[j] = l[j], l[i]
3
4  def selection_sort(lista):
5    for i in range(len(lista)-1):
6      menor_pos = i
7      for j in range(i+1, len(lista)):
8        if(lista[j]<lista[menor_pos]):
9            menor_pos = j
10     #swap
11     swap(lista, i, menor_pos)
12   return lista
```

- Número de comparações:

$$(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = \frac{n^2-n}{2}$$

- Número de trocas: $n-1$
  - Muito bom quando trocas são muito caras

Ordenação por Seleção

```
1  def swap(l, i, j)
2    l[i], l[j] = l[j], l[i]
3
4  def selection_sort(lista):
5    for i in range(len(lista)-1):
6      menor_pos = i
7      for j in range(i+1, len(lista)):
8        if(lista[j]<lista[menor_pos]):
9            menor_pos = j
10     #swap
11     swap(lista, i, menor_pos)
12   return lista
```

- Número de comparações:

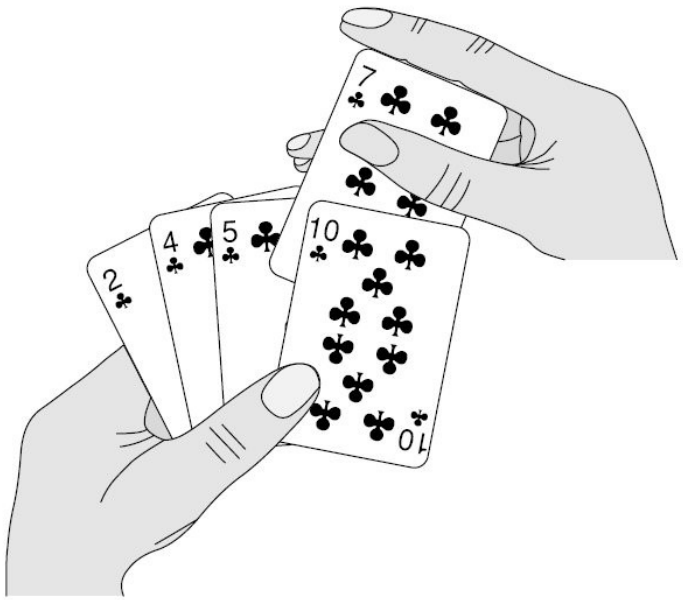
$$(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = \frac{n^2-n}{2}$$

- Número de trocas: $n-1$
  - Muito bom quando trocas são muito caras

Ordenação por Seleção

Existem algumas variações para o Selection Sort:

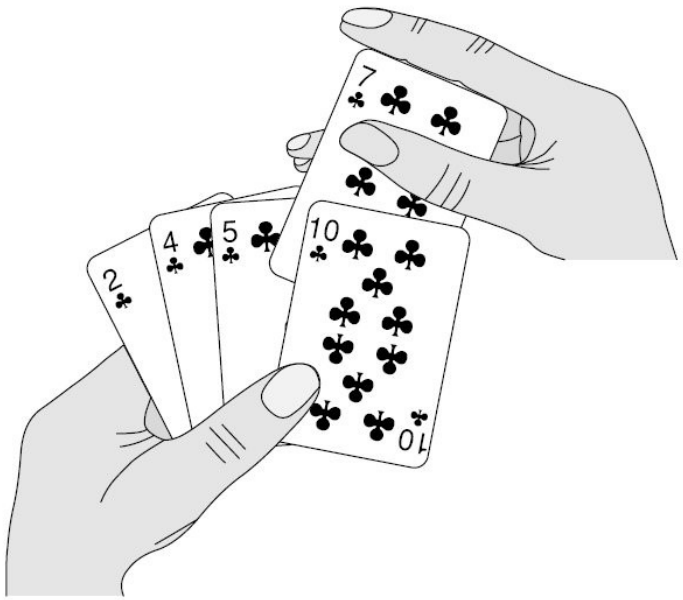
- Busca invertida pelo maior valor
- Mesmo número de comparações: $\frac{n^2-n}{2}$



Ordenação por Seleção

Existem algumas variações para o Selection Sort:

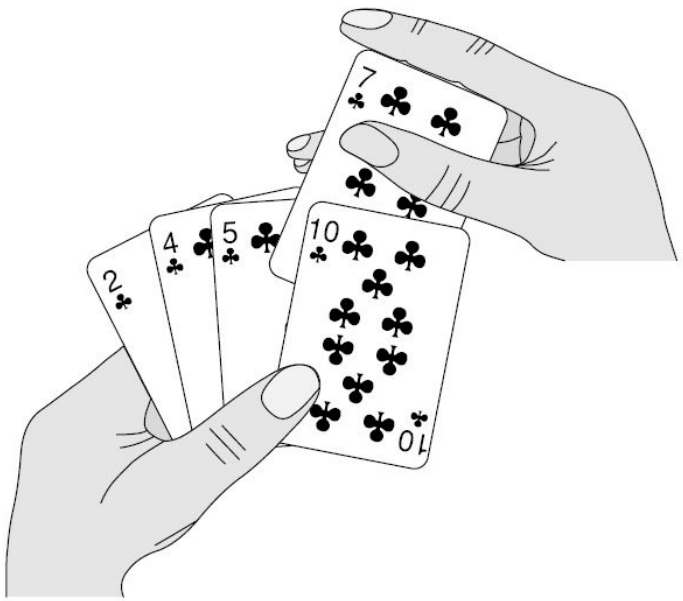
- Busca invertida pelo maior valor
- Mesmo número de comparações: $\frac{n^2-n}{2}$



Roteiro

Bubble Sort

Bubble Sort:



Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

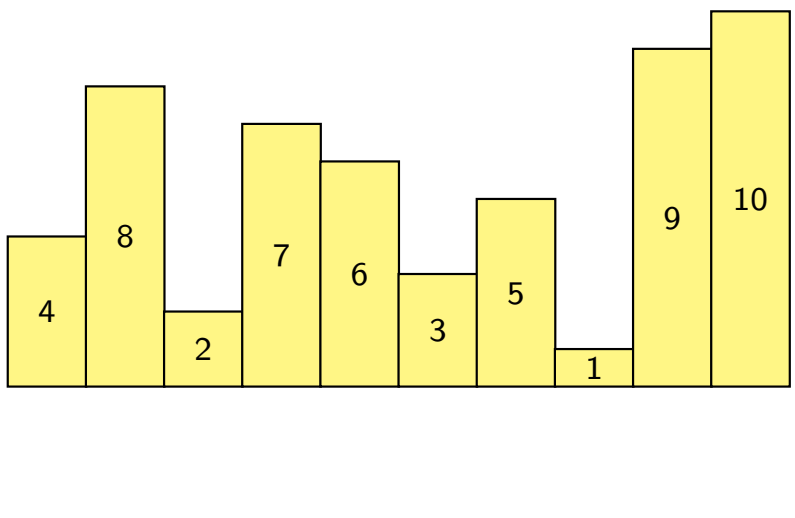
```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

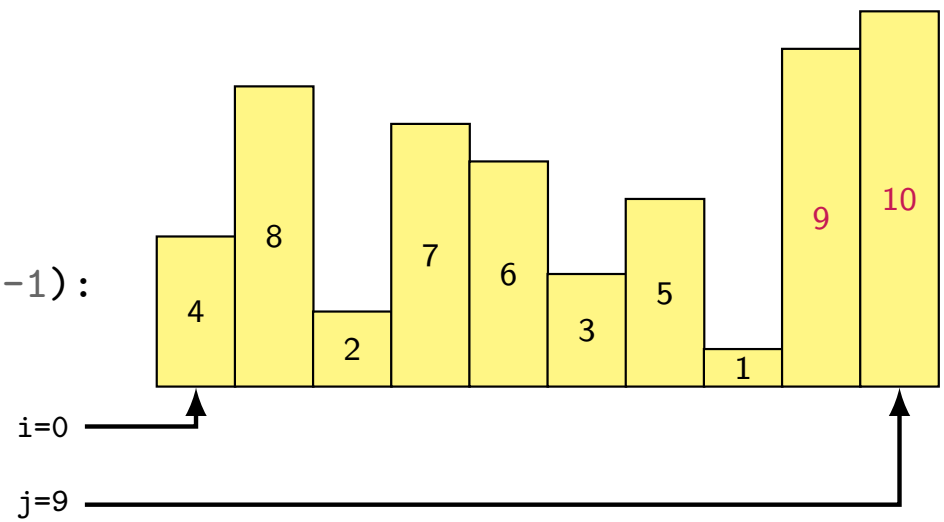


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

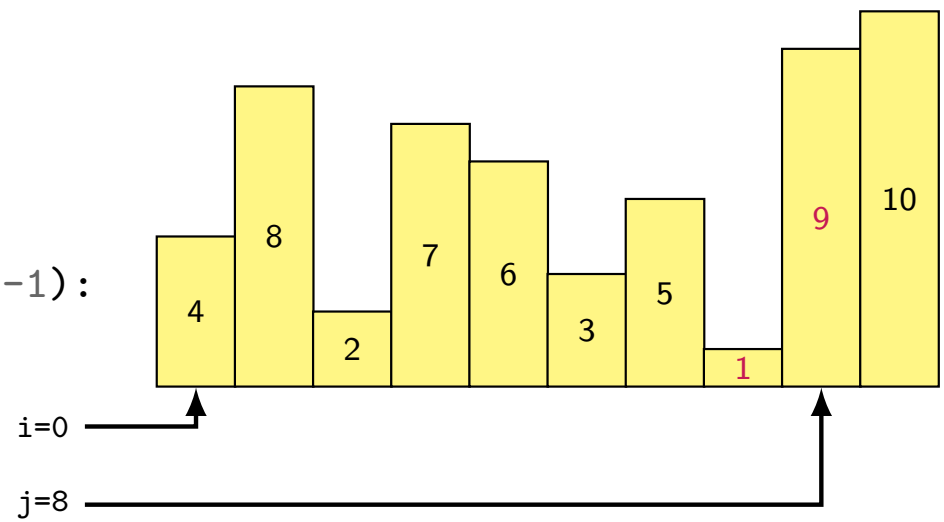


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

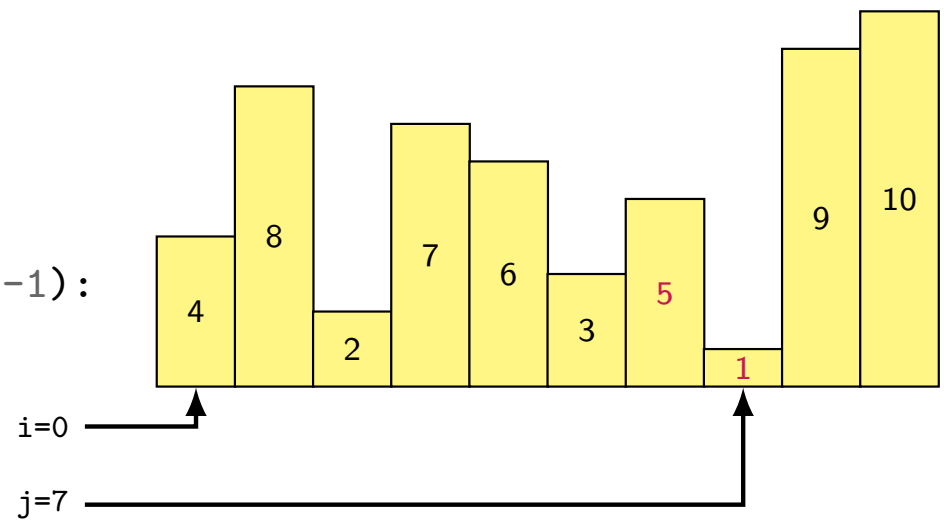


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

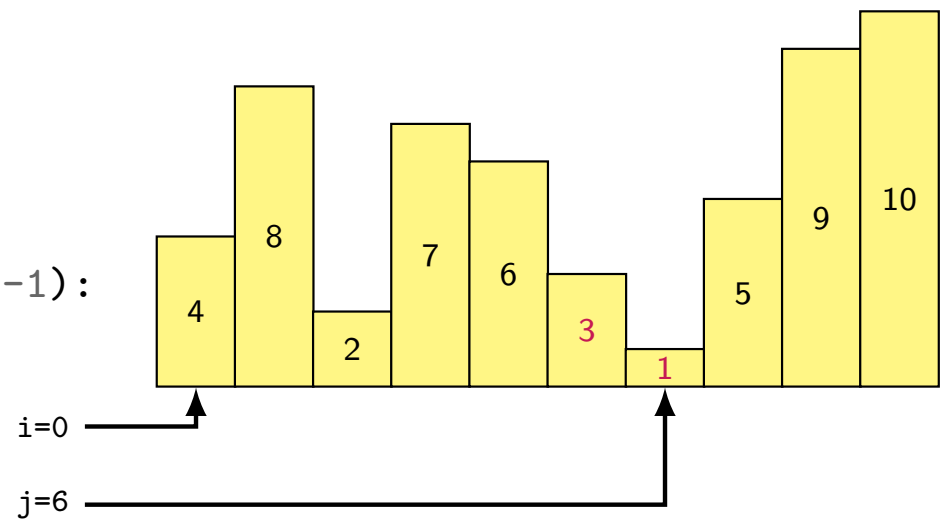


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

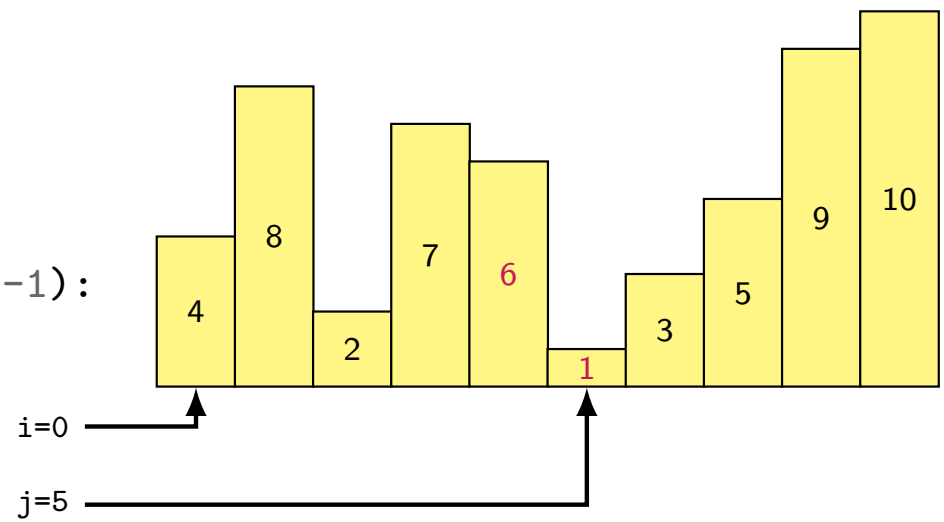


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

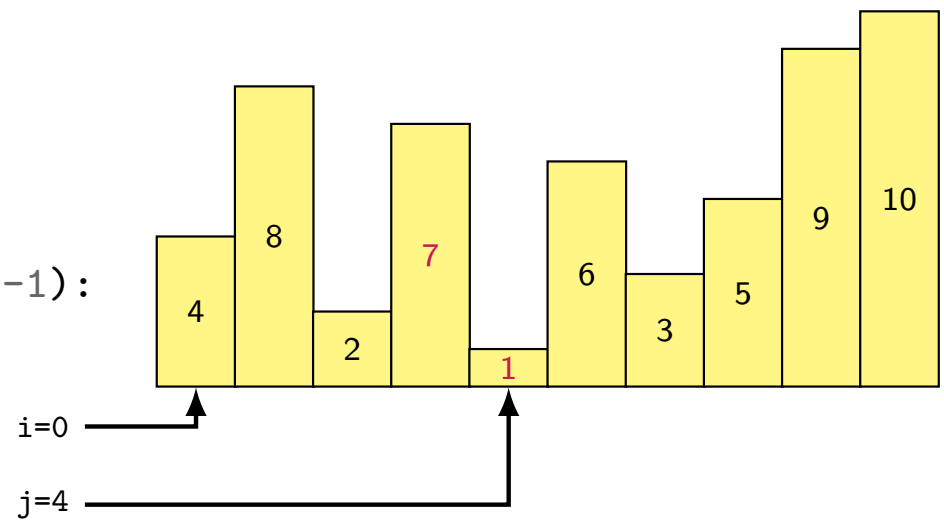


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

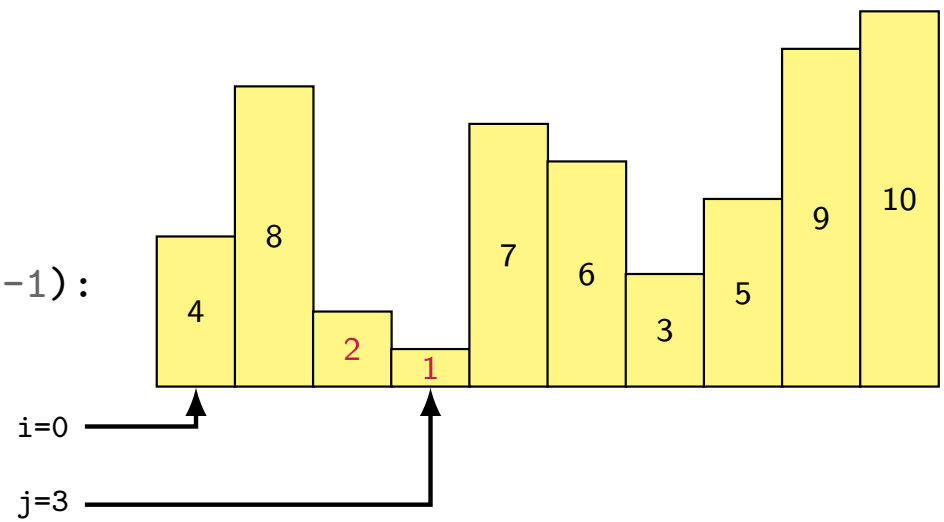


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

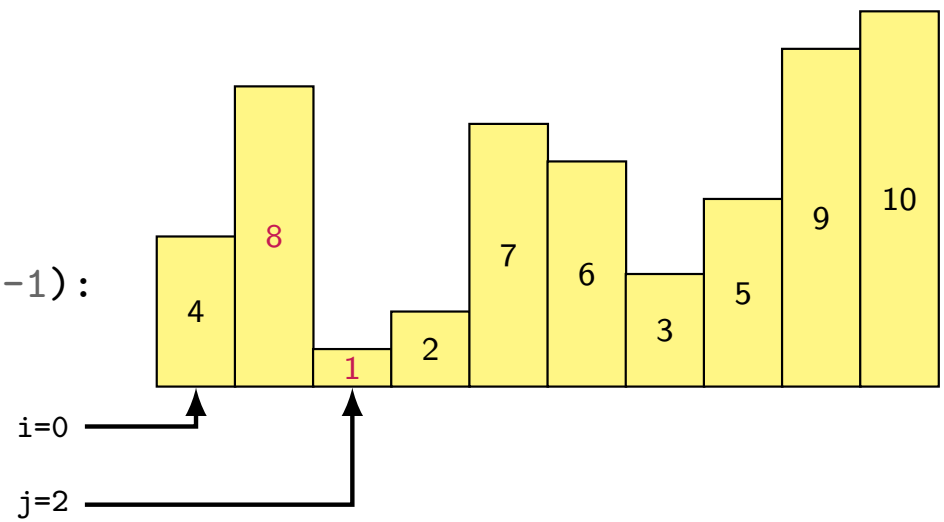


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

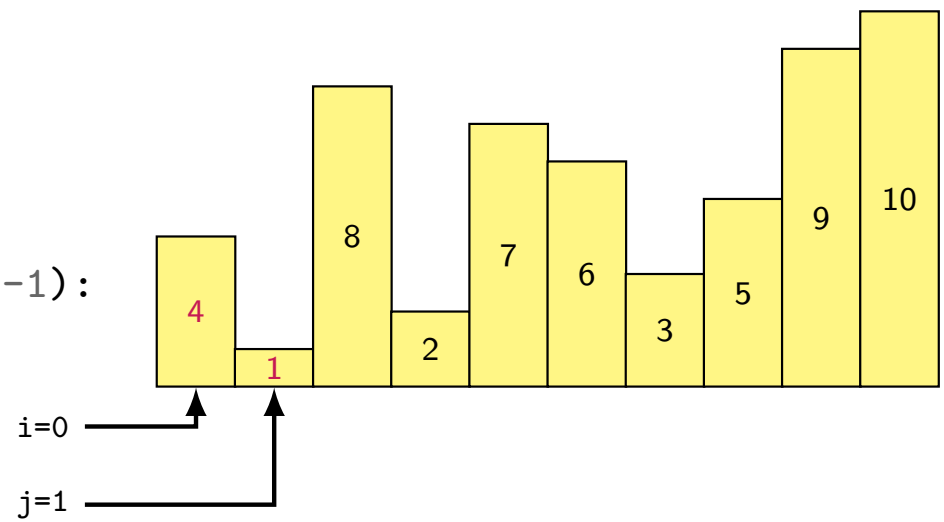


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

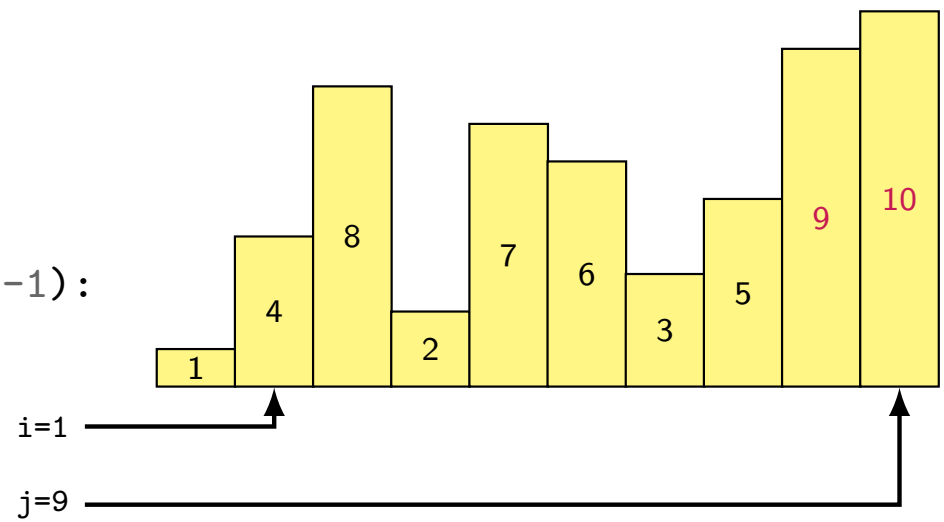


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

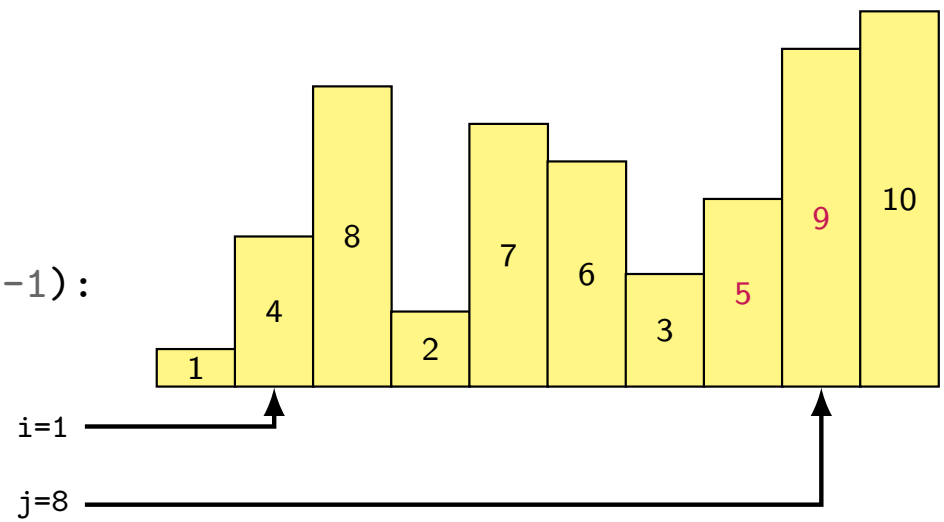


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

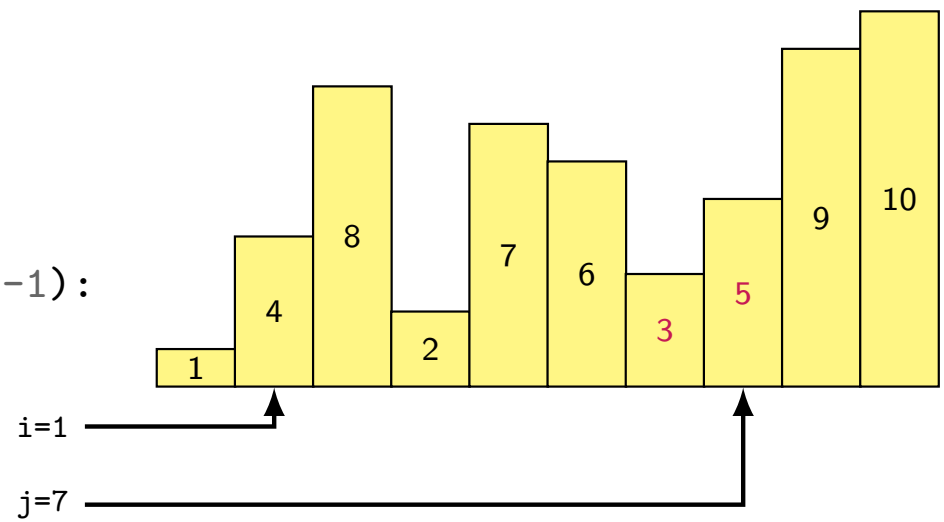


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

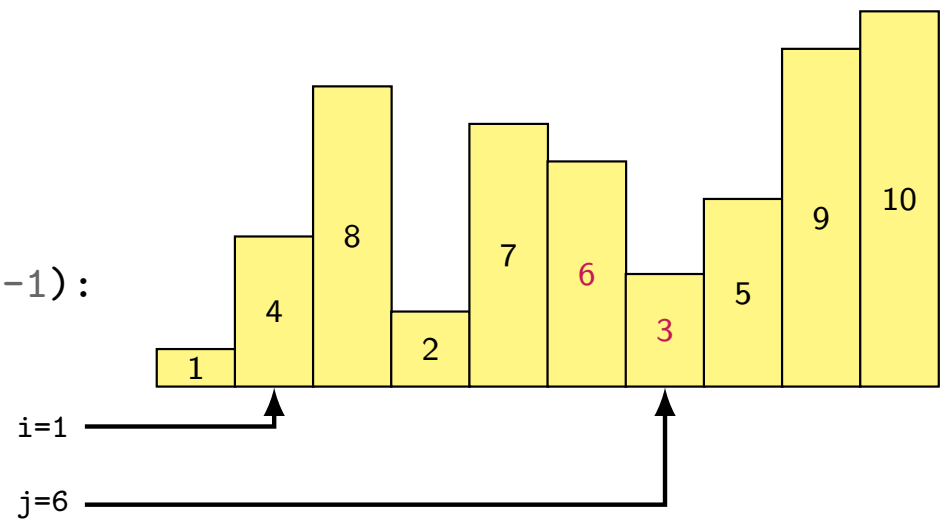


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

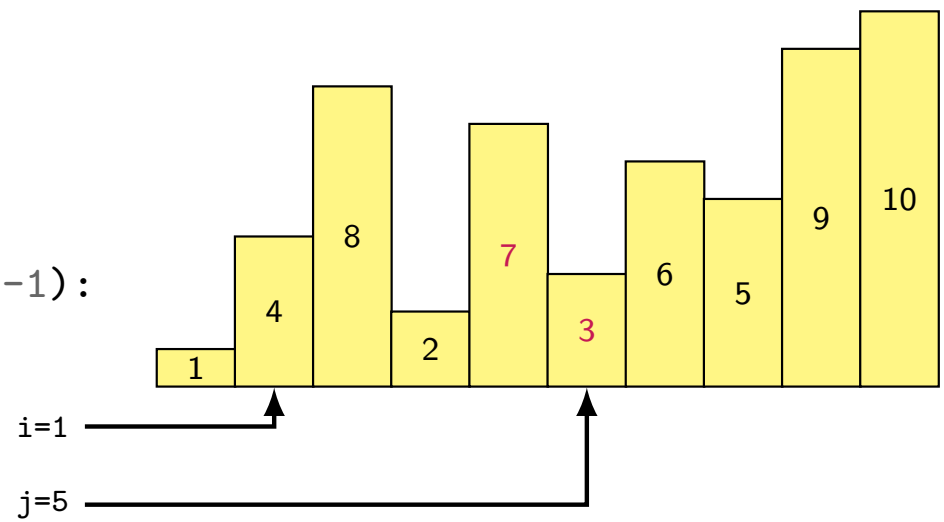


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

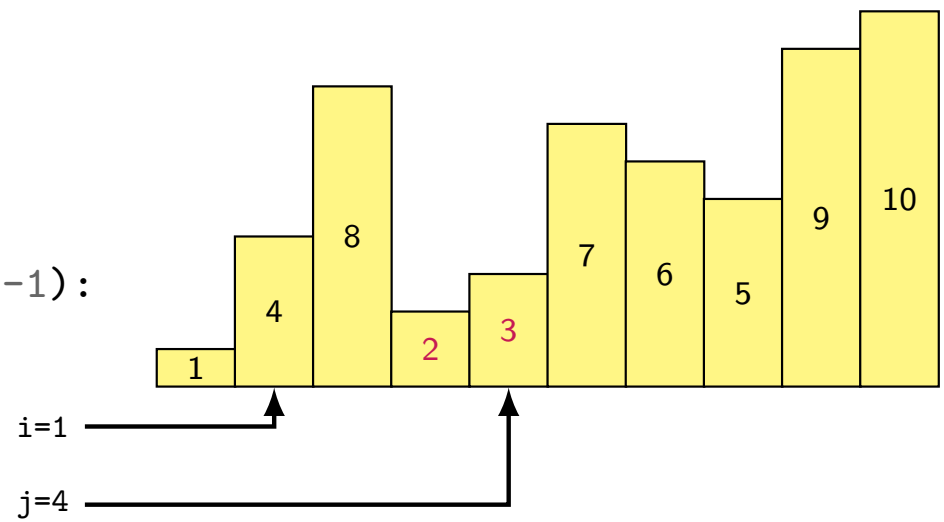


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

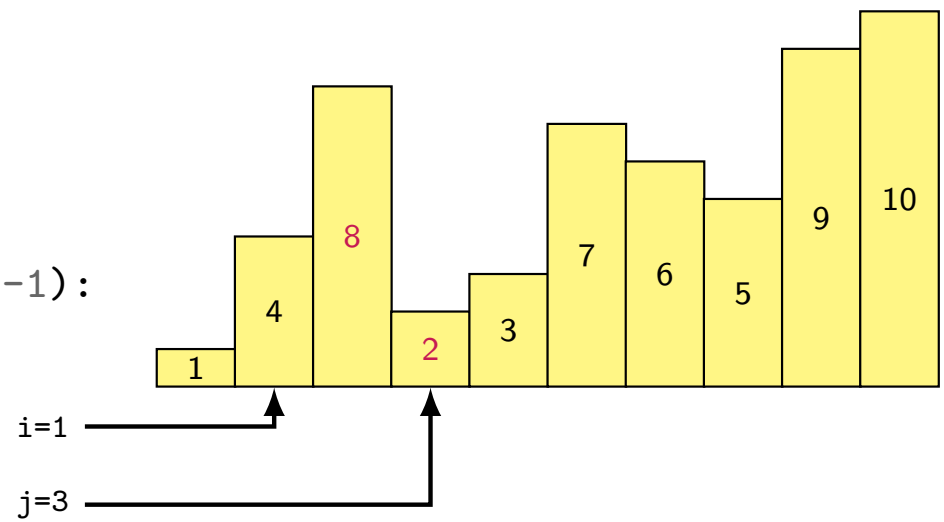


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

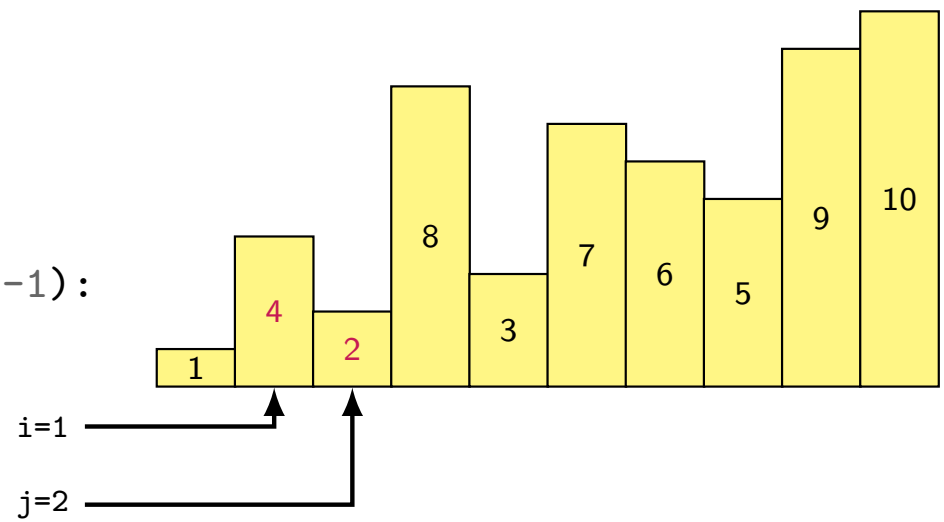


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

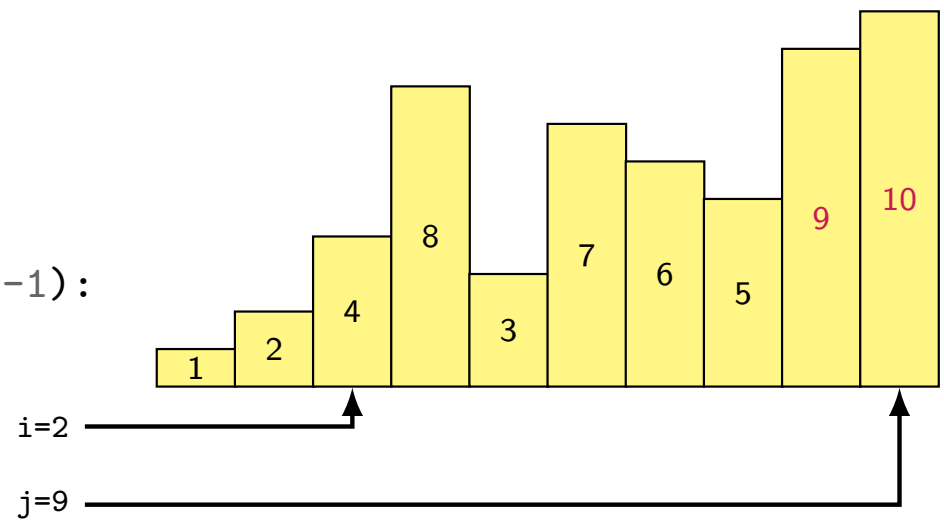


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

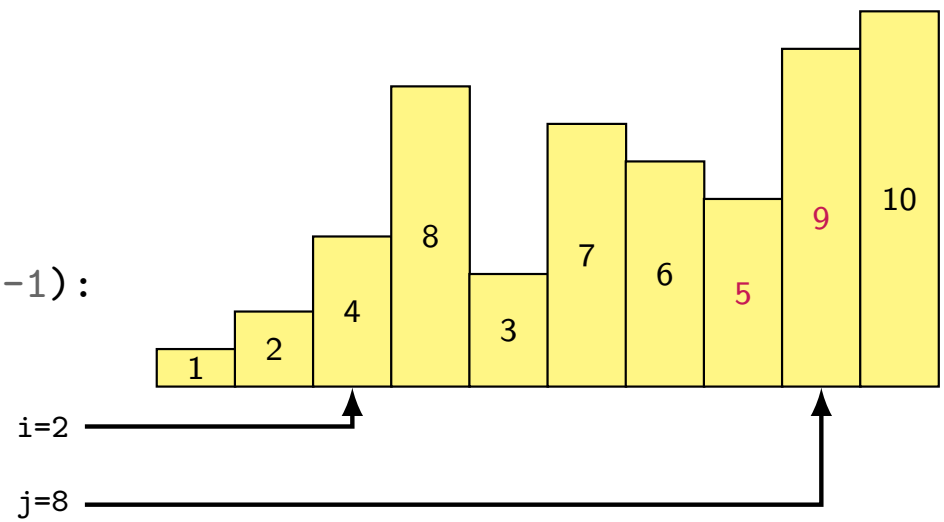


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

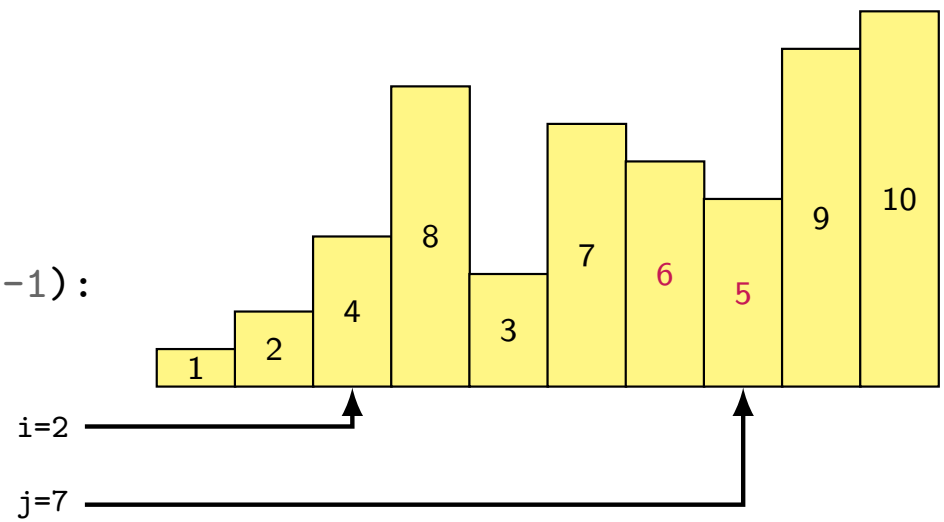


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

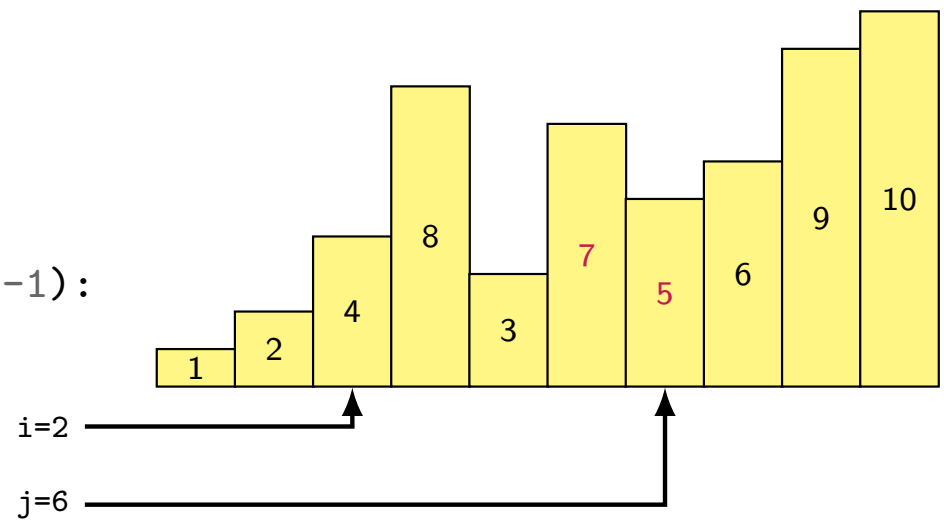


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

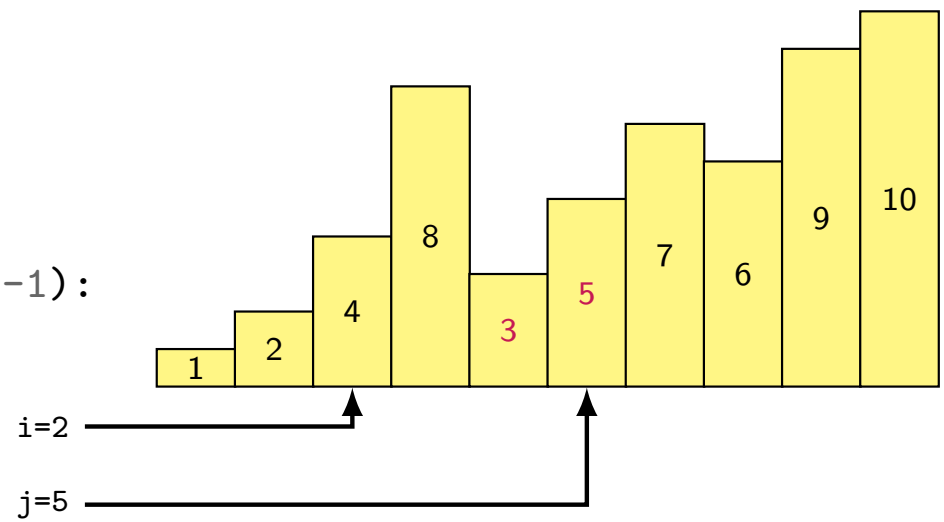


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

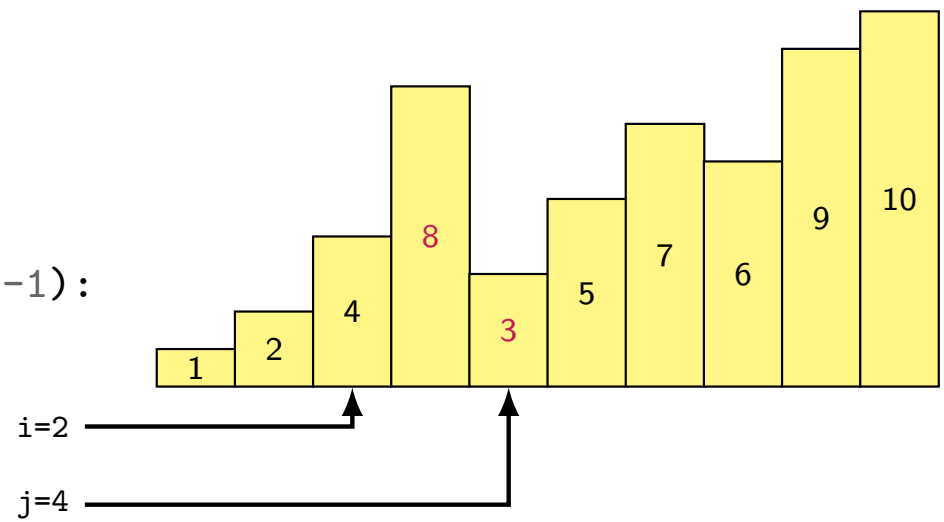


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

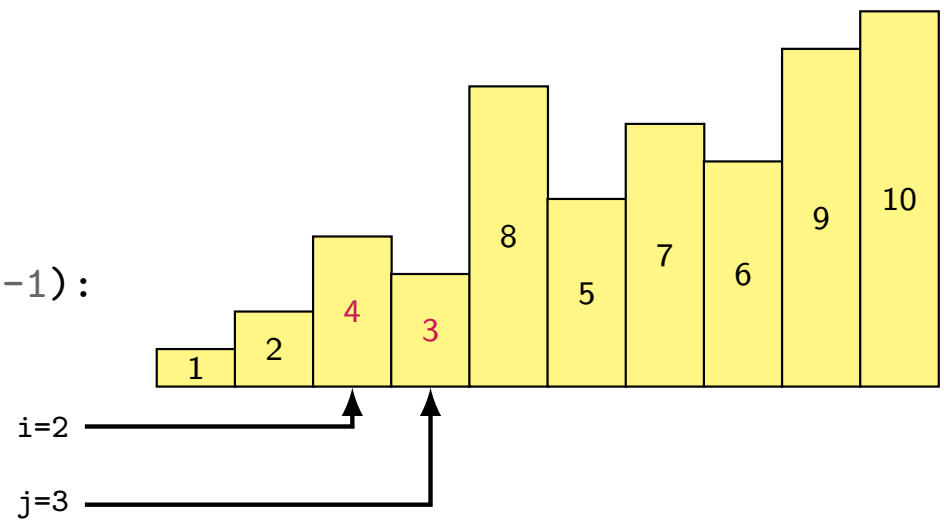


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

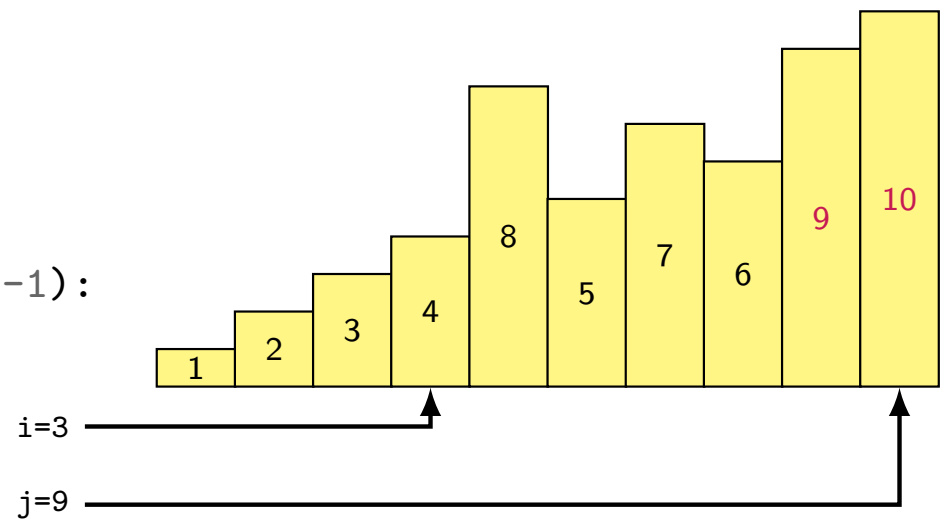


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

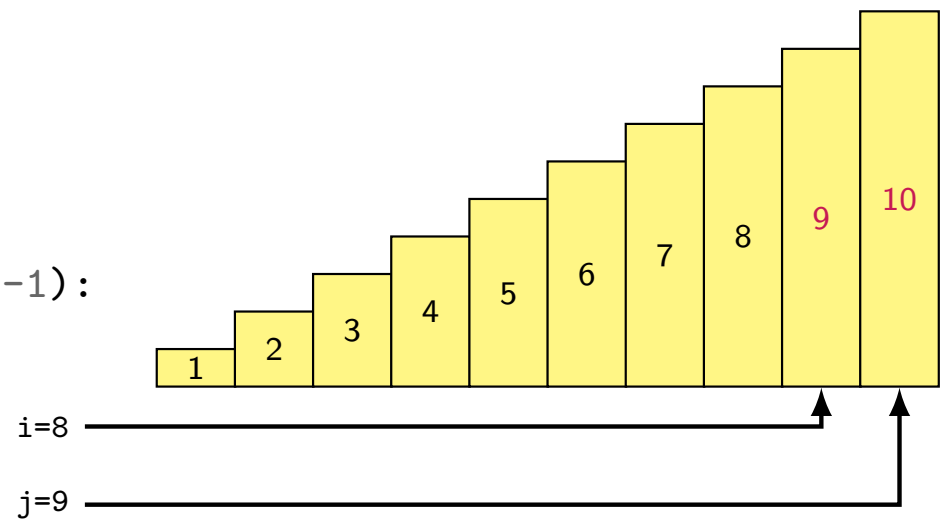


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```

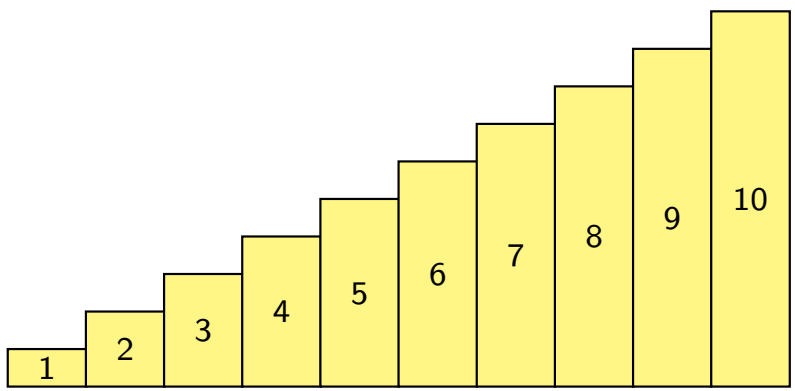


Bubble Sort

Ideia do algoritmo:

- Percorre a `lista` *n* vezes, trocando pares **invertidos**
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem à esquerda

```
1 def swap(l, i, j):
2   l[i], l[j] = l[j], l[i]
3
4 def bubble_sort(lista):
5   for i in range(len(lista)):
6     for j in range(len(lista)-1, i, -1):
7       if(lista[j]<lista[j-1]):
8         swap(lista, j, j-1)
9   return lista
```



Parando quando não há mais trocas

Também existem variações do Bubble Sort

- Se não aconteceu nenhuma troca, podemos **parar o algoritmo**:

```
1  def swap(l, i, j):
2    l[i], l[j] = l[j], l[i]
3
4  def bubble_sort_v2(lista):
5    for i in range(len(lista)):
6      trocou = False
7      for j in range(len(lista)-1, i, -1):
8        if(lista[j]<lista[j-1]):
9          swap(lista, j, j-1)
10         trocou = True
11     if not trocou:
12       break
13   return lista
```

No pior caso, cada comparação gera uma troca:

- Número de comparações: $n(n-1)/2 = \frac{n^2-n}{2}$
- Número de trocas: $n(n-1)/2 = \frac{n^2-n}{2}$

Parando quando não há mais trocas

Também existem variações do Bubble Sort

- Se não aconteceu nenhuma troca, podemos **parar o algoritmo**:

```
1  def swap(l, i, j):
2    l[i], l[j] = l[j], l[i]
3
4  def bubble_sort_v2(lista):
5    for i in range(len(lista)):
6      trocou = False
7      for j in range(len(lista)-1, i, -1):
8        if(lista[j]<lista[j-1]):
9          swap(lista, j, j-1)
10         trocou = True
11     if not trocou:
12       break
13   return lista
```

No pior caso, cada comparação gera uma troca:

- Número de comparações: $n(n-1)/2 = \frac{n^2-n}{2}$
- Número de trocas: $n(n-1)/2 = \frac{n^2-n}{2}$

Roteiro

Comparação de Desempenho

Como comparar a eficiência de dois algoritmos?

- Podemos analisar o custo teórico (análise matemática).
  - Mas e quando os dois tempo o mesmo custo?
  - Podemos medir o **tempo prático**.

Vimos na aula passada o módulo `time` do **Python**.

- `time`:

```
1 >>> import time
2 >>> time.time() # retorna o quantos segundos passados desde 01/01/1970
3 1621450935.0984983
```

01/01/1970: início dos tempos para sistemas UNIX.

Comparação de Desempenho

Como comparar a eficiência de dois algoritmos?

- Podemos analisar o custo teórico (análise matemática).
  - Mas e quando os dois tempo o mesmo custo?
  - Podemos medir o **tempo prático**.

Vimos na aula passada o módulo `time` do **Python**.

- `time`:

```
1 >>> import time
2 >>> time.time() # retorna o quantos segundos passados desde 01/01/1970
3 1621450935.0984983
```

01/01/1970: início dos tempos para sistemas UNIX.

Comparação de Desempenho

Como comparar a eficiência de dois algoritmos?

- Podemos analisar o custo teórico (análise matemática).
  - Mas e quando os dois tempo o mesmo custo?
  - Podemos medir o **tempo prático**.

Vimos na aula passada o módulo `time` do **Python**.

- `time`:

```
1 >>> import time
2 >>> time.time() # retorna o quantos segundos passados desde 01/01/1970
3 1621450935.0984983
```

01/01/1970: início dos tempos para sistemas UNIX.

Comparação de Desempenho

Como comparar a eficiência de dois algoritmos?

- Podemos analisar o custo teórico (análise matemática).
  - Mas e quando os dois tempo o mesmo custo?
  - Podemos medir o **tempo prático**.

Vimos na aula passada o módulo `time` do **Python**.

- `time`:

```
1 >>> import time
2 >>> time.time() # retorna o quantos segundos passados desde 01/01/1970
3 1621450935.0984983
```

01/01/1970: início dos tempos para sistemas UNIX.

Comparação de Desempenho

Como **medir o tempo** de um algoritmo com o módulo `time`?

```
>>> import time
>>> antes = time.time()
>>> #algoritmo()
>>> depois = time.time()
>>>
>>> print(depois-antes, "segundos")
```

Comparação de Desempenho

Vamos criar um módulo com os algoritmos de ordenação vistos em aula:

ordenadores.py

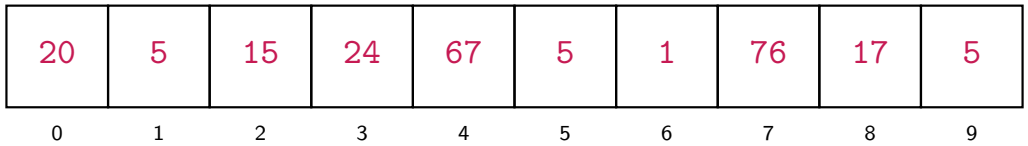
```
1  def swap(l, i, j):
2    l[i], l[j] = l[j], l[i]
3
4  def selection_sort(l):
5    for i in range(len(l)-1):
6      menor_pos = i
7      for j in range(i+1, len(l)):
8        if(l[j]<l[menor_pos]):
9            menor_pos = j
10     swap(l, i, menor_pos)
11   return l
12
13 def bubble_sort(l):
14   for i in range(len(l)):
15     for j in range(len(l)-1, i, -1):
16       if(l[j]<l[j-1]):
17         swap(l, j, j-1)
18   return l
```

ordenadores.py

```
1  def bubble_sort_v2(l):
2    for i in range(len(l)):
3      trocou = False
4      for j in range(len(l)-1, i, -1):
5        if(l[j]<l[j-1]):
6          swap(l, j, j-1)
7          trocou = True
8      if not trocou:
9        break
10   return l
```

Comparação de Desempenho

Testar com lista:




```
1 def tempo(algoritmo, lista):
2   import time
3   antes = time.time()
4   algoritmo(lista)
5   depois = time.time()
6   return depois-antes
```

```
1 >>> import ordenadores as ord
2 >>>
3 >>> lista = [20, 5, 15, 24, 67, 5, 1, 76, 17, 5]
4 >>> tempo(ord.selection_sort, lista)
5 5.91278076171875e-05 # muito pequeno
```

Em **Python** podemos criar *alias* para nomes de funções.

Comparação de Desempenho

Precisamos gerar listas maiores!!

- Na aula passada vimos o módulo `random` do **Python**:

```
>>> import random
>>> random.randint(3, 9) # retorna um valor entre 3 e 9
4
```

- Vamos adicionar no módulo `ordenadores.py` a seguinte função:

```
def lista_aleatoria(n = 10, m = 100):
  import random
  l = []
  for i in range(n):
    l.append(random.randint(1, m))
  return l
```

```
>>> ord.lista_aleatoria()
[90, 64, 23, 96, 88, 96, 55, 29, 78, 45]
```

Comparação de Desempenho

- Relembrando:

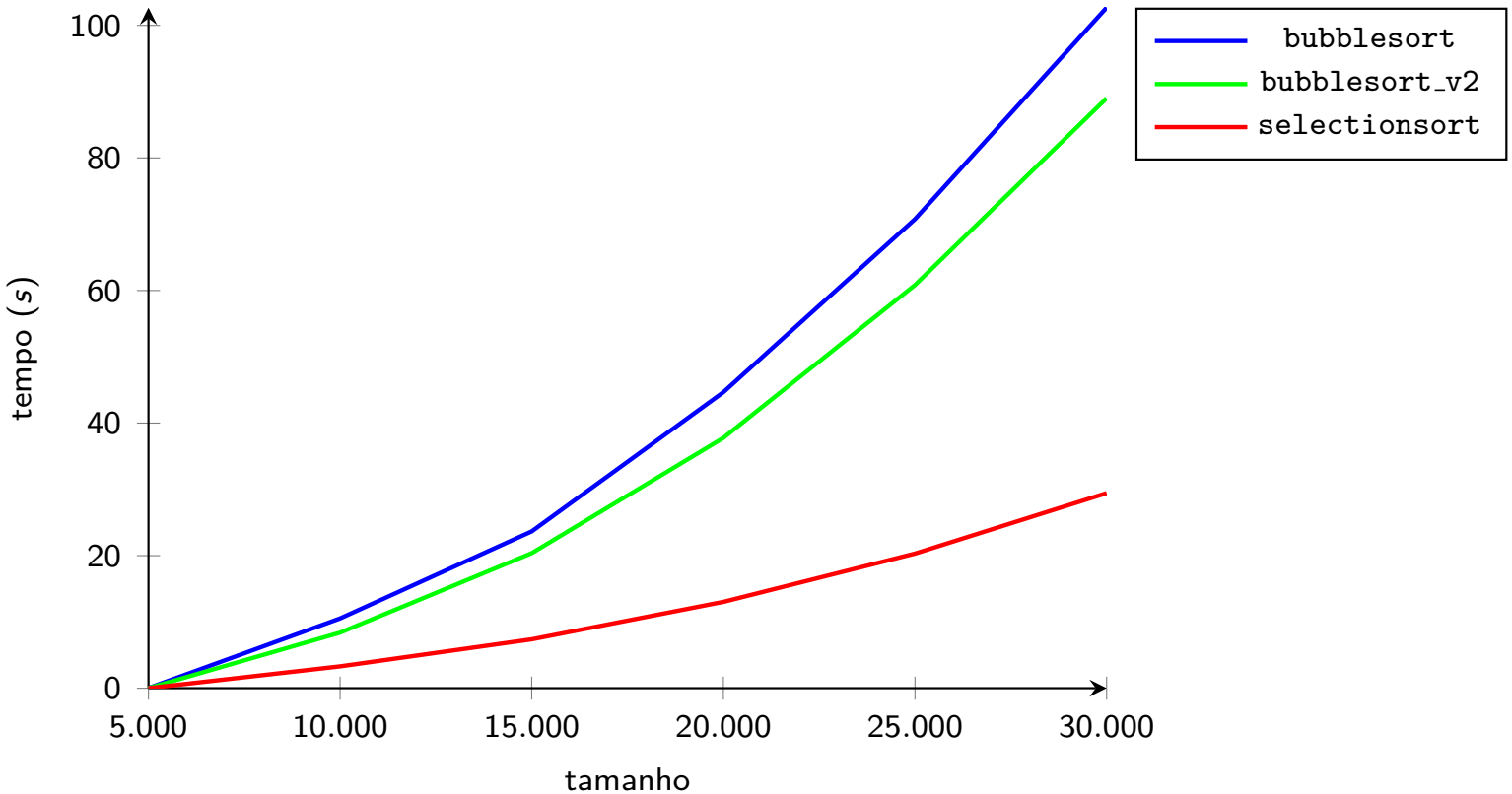
```
>>> import importlib
>>> importlib.reload(ord)
>>>
>>> lista = ord.lista_aleatoria(10000)
```

Comparação de Desempenho

- Finalmente:

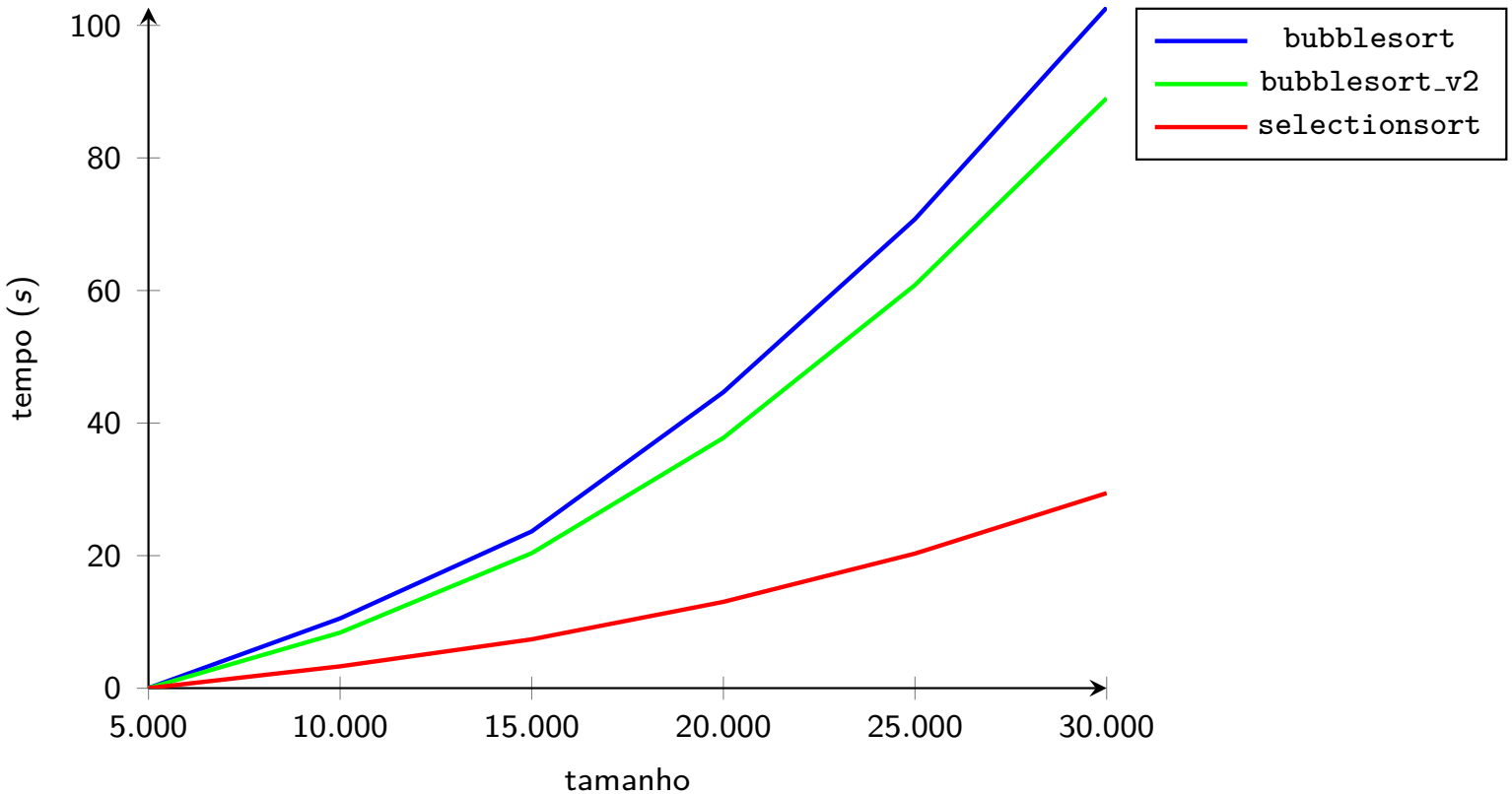
```
>>> lista = ord.lista_aleatoria(10000)
>>>
>>> l1 = lista[:]
>>> print("{:.2f} segundos".format(tempo(ord.selection_sort, l1)))
9.59 segundos
>>>
>>> l2 = lista[:]
>>> print("{:.2f} segundos".format(tempo(ord.bubble_sort, l2)))
22.30 segundos
>>>
>>> l3 = lista[:]
>>> print("{:.2f} segundos".format(tempo(ord.bubble_sort_v2, l3)))
20.21 segundos
```

Comparação de Desempenho



- O tempo de cada algoritmo é quadrático: quando n dobra, o tempo **quadriplica**.

Comparação de Desempenho



- O tempo de cada algoritmo é quadrático: quando n dobra, o tempo **quadriplica**.

Conclusão

Vimos dois algoritmos **quadráticos** para ordenação:

- `Bubble Sort`: na pratica é o pior dos dois, raramente usado
- `Selection Sort`: o melhor dos dois na prática.

Existem outros algoritmos melhores, vamos ver em outros cursos..

- `Insertion Sort`, `Merge Sort` e `QuickSort`

Conclusão

Vimos dois algoritmos **quadráticos** para ordenação:

- `Bubble Sort`: na pratica é o pior dos dois, raramente usado
- `Selection Sort`: o melhor dos dois na prática.

Existem outros algoritmos melhores, vamos ver em outros cursos..

- `Insertion Sort`, `Merge Sort` e `QuickSort`

Conclusão

Vimos dois algoritmos **quadráticos** para ordenação:

- `Bubble Sort`: na pratica é o pior dos dois, raramente usado
- `Selection Sort`: o melhor dos dois na prática.

Existem outros algoritmos melhores, vamos ver em outros cursos..

- `Insertion Sort`, `Merge Sort` e `QuickSort`

Conclusão

Vimos dois algoritmos **quadráticos** para ordenação:

- `Bubble Sort`: na pratica é o pior dos dois, raramente usado
- `Selection Sort`: o melhor dos dois na prática.

Existem outros algoritmos melhores, vamos ver em outros cursos..

- `Insertion Sort`, `Merge Sort` e `QuickSort`

Conclusão

Vimos dois algoritmos **quadráticos** para ordenação:

- `Bubble Sort`: na pratica é o pior dos dois, raramente usado
- `Selection Sort`: o melhor dos dois na prática.

Existem outros algoritmos melhores, vamos ver em outros cursos..

- `Insertion Sort`, `Merge Sort` e `QuickSort`

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.
2. panda.ime.usp.br/pensepy/static/pensepy/index.html