Estrutura de Dados

Aula 09

Prof. Luiz Antonio Schalata Pacheco, Dr. Eng.

Instituto Federal de Santa Catarina
Câmpus Garopaba
Curso Superior de Tecnologia em Sistemas para Internet

schalata@ifsc.edu.br

30/05/2022

Algoritmos de Ordenação

Motivação

- Inserir um dado em um vetor ordenado gera um Big(O) elevado, pois tem de fazer o remanejamento dos itens
- Ter os dados ordenados é um passo preliminar para pesquisá-los:
  - pesquisa linear (lenta)
  - pesquisa binária
- Logo, muitas vezes é necessário ordenar uma base de dados já construída:
  - Nome em ordem alfabética
  - Alunos por nota
  - Vendas por preço

Vários métodos de ordenação

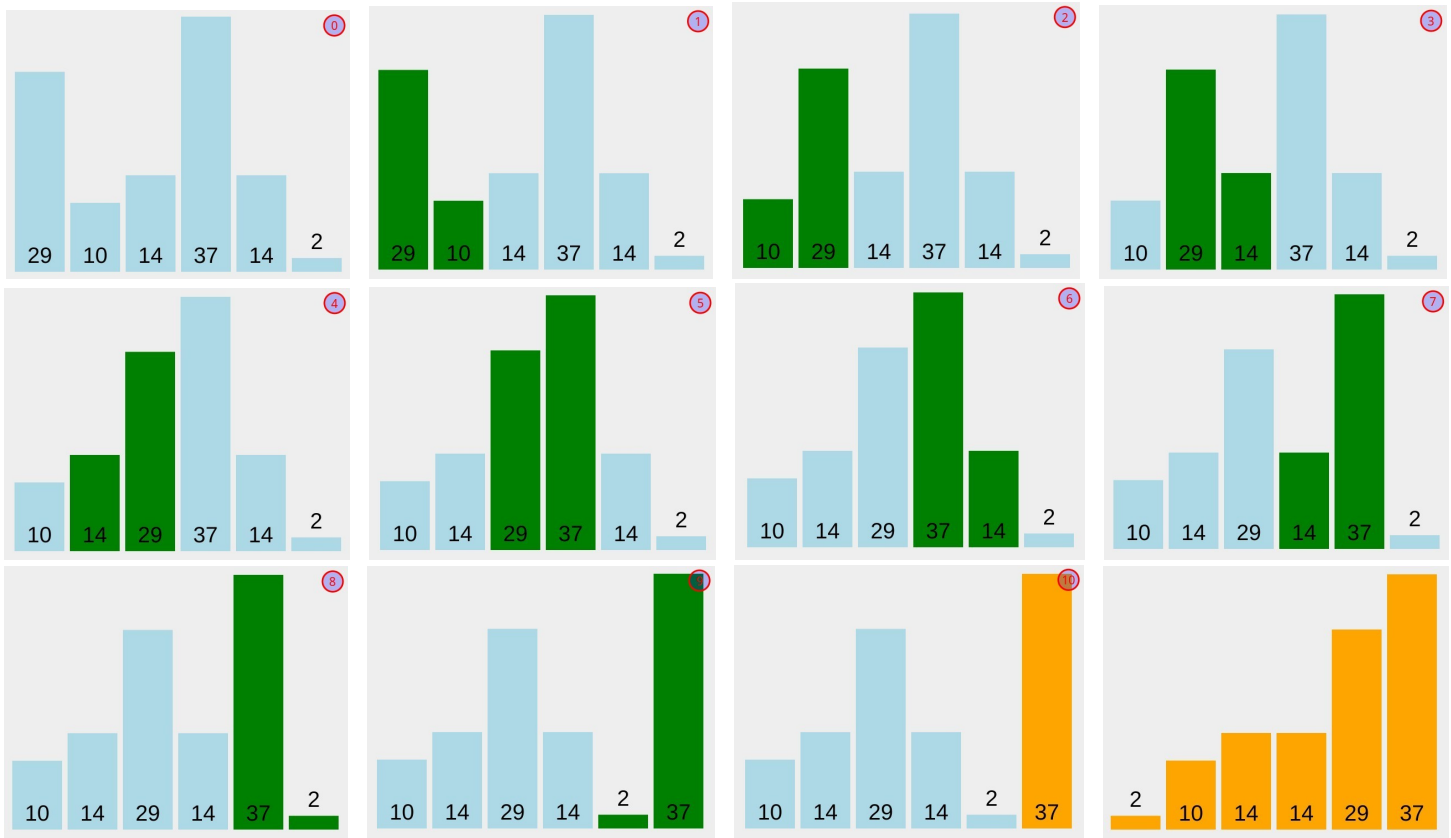
- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort

Bubble Sort

Bubble Sort - Conceito

- É o algoritmo de ordenação mais lento e mais simples
- Funcionamento:
  - Compara dois números
  - Se o da esquerda for maior, os elementos devem ser trocados
  - Desloca-se uma posição à direita
  - A medida que algoritmo avança, os itens maiores "surgem como uma bolha" na extremidade superior do vetor
  - Exige várias rodadas

Bubble Sort em funcionamento



- https://visualgo.net/en/sorting

Análise da complexidade

- O algoritmo para ordenação de 6 números faz:
  - 5 comparações na primeira passagem
  - 4 comparações na segunda passagem
  - 3 comparações na terceira passagem
  - 2 comparações na quarta passagem
  - 1 comparações na quinta passagem
- Para 6 itens: 15 comparações, ou seja, $(n-1)+(n-2)+(n-3)+(n-4)+(n-5)$
- Generalizando: $(n-1)+(n-2)+...+(n-(n-1))$, onde n é o número de itens a ordenar
- O número de trocas depende de como os dados estão organizados
- Cada troca leva 3 passos

Análise da complexidade

- Big-O: $O(n^2)$
- São realizadas cerca de $n^2/2$ comparações
- Ocorrem menos trocas que comparações, pois os elementos só serão trocados se não estiverem em ordem
- A quantidade de trocas necessária é estimada em $n^2/4$ considerando dados aleatórios
- Pior caso: dados ordenados de modo inverso

Implementação

- Implementar e testar o algoritmo de ordenação Bubble Sort

Implementação: Bubble Sort

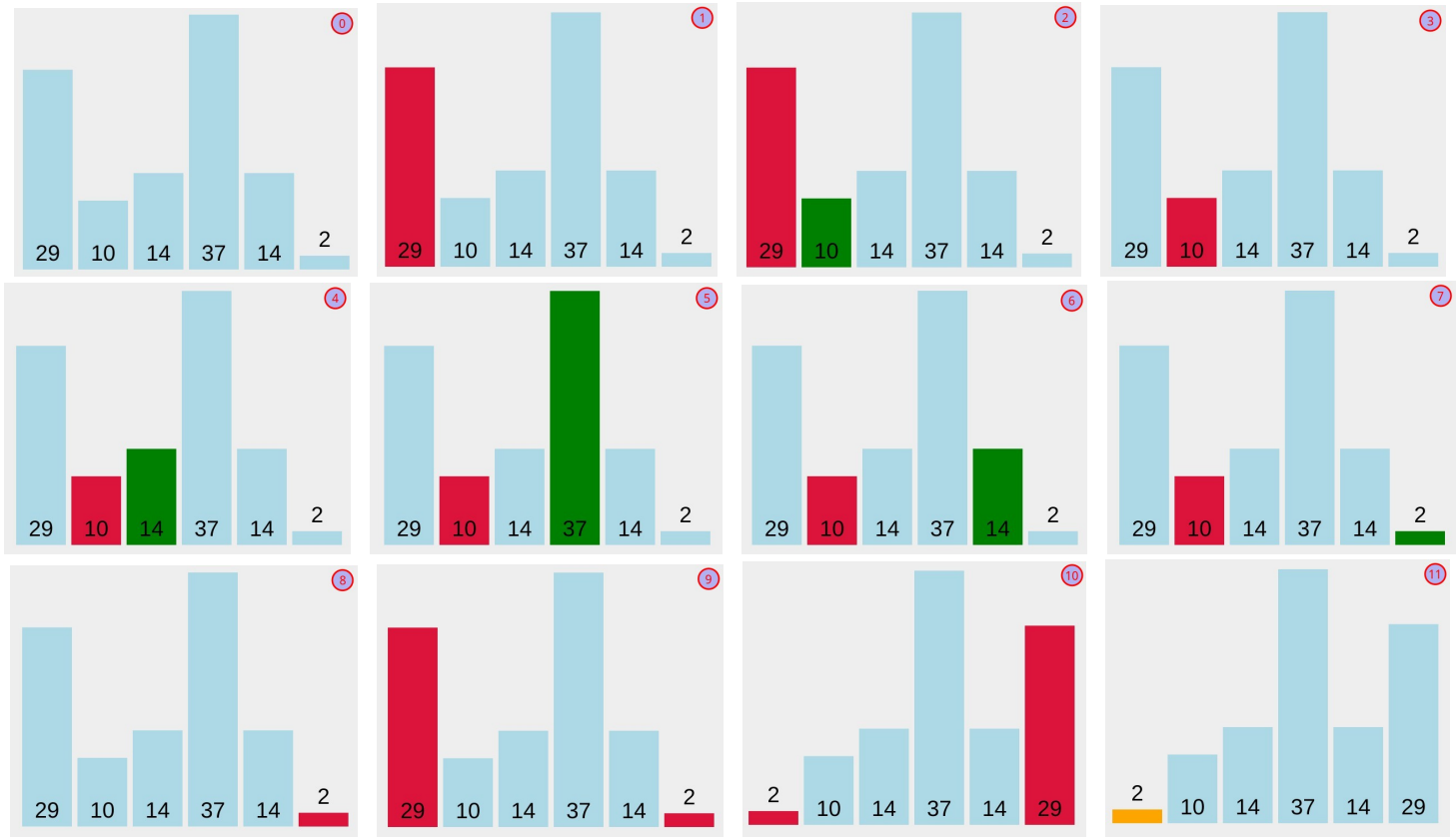
```
1  import numpy as np
2
3  def bubble_sort(vetor): # Recebe um vetor nao ordenado
4    n = len(vetor) # Tamanho do vetor
5
6    for i in range(n):
7      # Executa a ordenacao ate o valor que ja esta
       ordenado, por isso tem que usar (n - i) - 1 no range
8      for j in range(0, (n - i) - 1):
9        # Se o elemento da esquerda for maior que o da
       direita, entao faz a troca dos valores
10       if vetor[j] > vetor[j + 1]:
11         temp = vetor[j]
12         vetor[j] = vetor[j + 1]
13         vetor[j + 1] = temp
14   return vetor
15
16   print(bubble_sort(np.array([29, 10, 14, 37, 14, 2])))
```

Selection Sort

Selection Sort - Conceito

- Reduz o número de trocas de N^2 para N
- O número de comparações permanace $N^2/2$
- Funcionamento:
  - Percorre todos os elementos e seleciona o menor
  - O menor elemento é trocado com o elemento da extremidade esquerda do vetor
  - Os elementos ordenados acumulam-se na esquerda

Selection Sort em funcionamento



- https://visualgo.net/en/sorting

Análises

- Mesmo número de comparações do Bubble Sort
- Para 6 itens: 15 comparações
- Big-O: $O(n^2)$
- Geralmente uma troca é feita a cada passagem. Então para 6 elementos são requeridas menos de 6 trocas.
- Se tiver 100 itens, são requiridas 4950 comparações, mas menos de 100 trocas

Implementação

- Implementar e testar o algoritmo de ordenação Selection Sort

Implementação: Selection Sort

```
1  def selection_sort(vetor):
2    n = len(vetor)
3
4    for i in range(n): # Cada iteracao indica uma rodada
5      id_minimo = i # Posicao onde esta o menor valor
6      for j in range(i + 1, n): # Parte de i + 1 porque os
       elementos ordenados nao precisam ser percorridos
       novamente e eles estao na posicao inicial do vetor
7        if vetor[id_minimo] > vetor[j]:
8          id_minimo = j # Se o valor comparado for menor,
       sua posicao passa a ser a nova posicao de id_minimo
9      # Ao final faz a troca do valores
10     temp = vetor[i]
11     vetor[i] = vetor[id_minimo]
12     vetor[id_minimo] = temp
13
14   return vetor
```

Insertion Sort

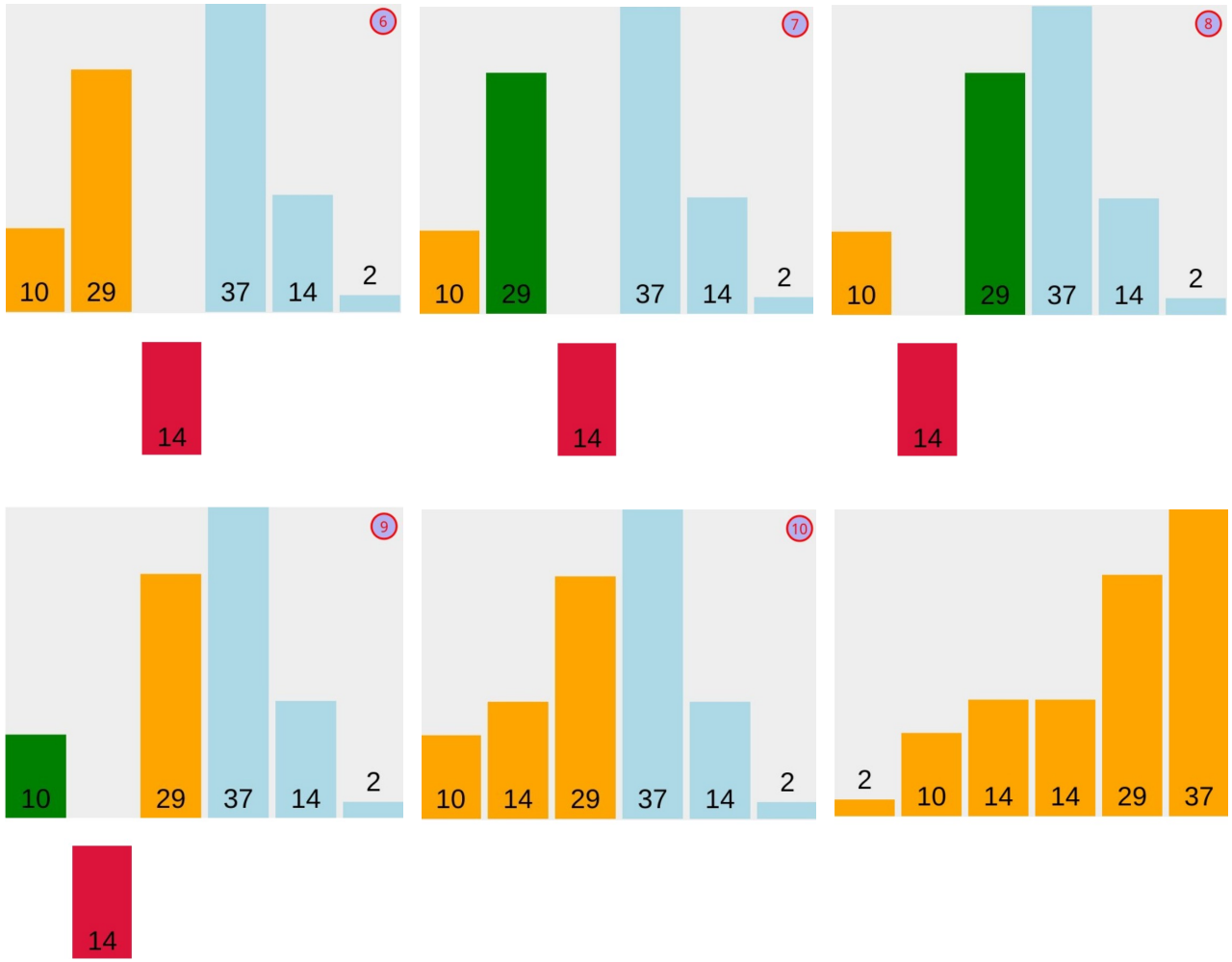
Insertion Sort - Conceito

- É cerca de duas mais rápido que o Bubble Sort e um pouco mais rápido que o Selection Sort (com dados aleatórios!)
- Funcionamento:
  - Há um marcador em algum lugar no meio do vetor
  - Os elementos a esquerda do marcados estão parcialmente ordenados (estão ordenados entre eles, mas não estão em suas posições finais)
  - Os elementos a direita do vetor não estão ordenados

Insertion Sort em funcionamento

- https://visualgo.net/en/sorting

Insertion Sort em funcionamento



Análises

- Na primeira passagem, é comparado no **máximo** um item. Na segunda passagem, no máximo de 2 itens e assim por diante
- Para 6 itens: $1 + 2 + 3 + 4 + 5 = 15$ comparações
- Generalizando: N*(N - 1)/2 comparações
- Como, na média, apenas metade do número máximo de itens é comparado, temos: N*(N - 1)/4 comparações
- O número de cópias (não são trocas!) é aproximadamente o mesmo número de comparações
- É importante conhecer previamente a base de dados a ser ordenada:
  - Para dados pre ordenados esse algoritmo é ainda mais eficiente
  - Para dados em ordem inversa, torna-se mais lento que o Bubble Sort, pois são executadas todas as comparações e deslocamentos

Implementação

- Implementar e testar o algoritmo de ordenação Insertion Sort

Implementação: Insertion Sort

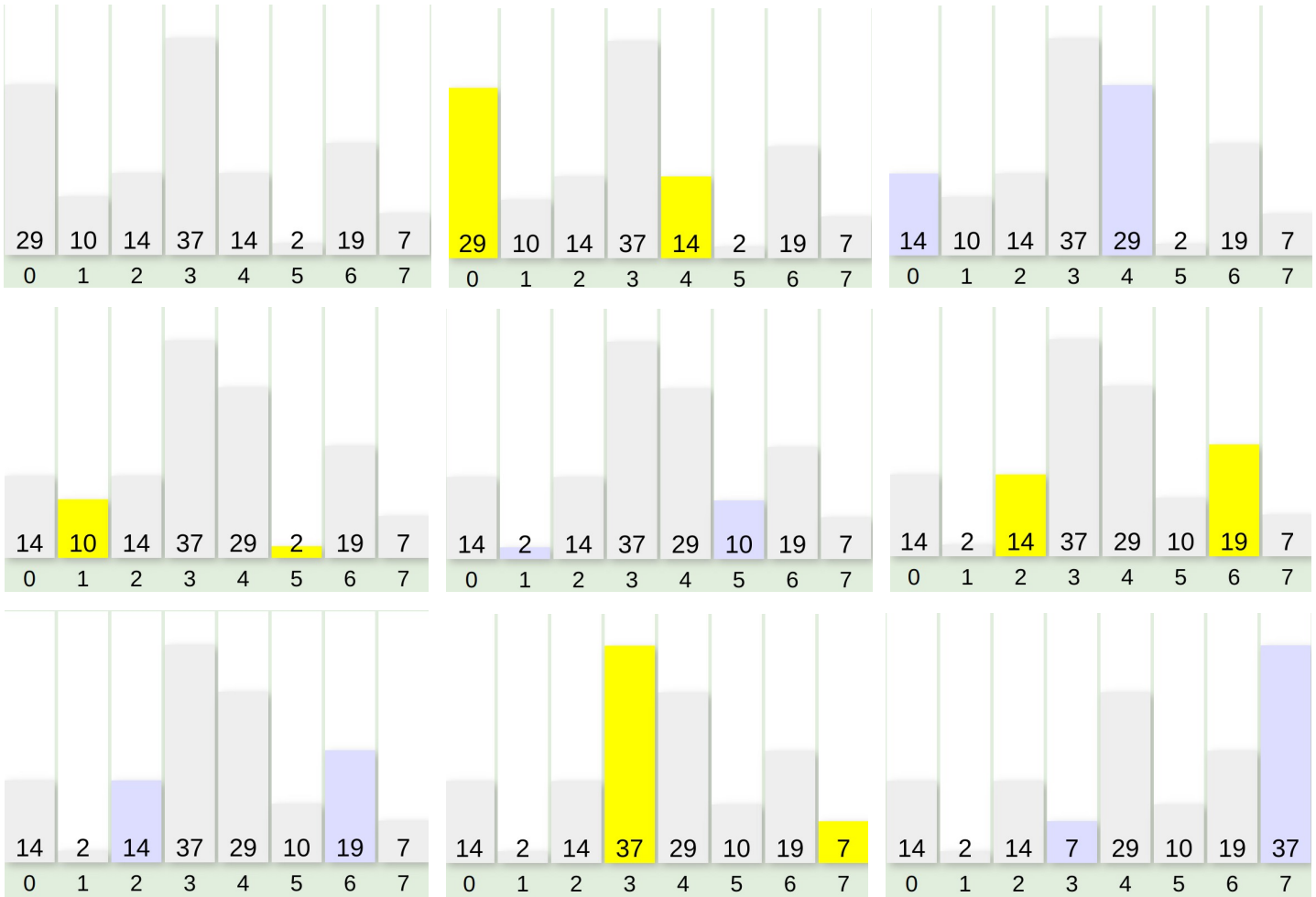
```
1  def insertion_sort(vetor):
2    n = len(vetor)
3
4    for i in range(1, n): # Inicia na 2a posicao do vetor
5      marcado = vetor[i]
6
7      j = i - 1
8      # Faz comparacoes ate o inicio do vetor (j >= 0) e
       somente enquanto o valor marcado for menor que a
       posicao do vetor que esta sendo comparada
9      while j >= 0 and marcado < vetor[j]:
10       vetor[j + 1] = vetor[j] # Copia o elemento uma
         posicao para frente
11       j -= 1
12     vetor[j + 1] = marcado # Copia o elemento marcado na
       posicao correta
13
14   return vetor
```

Shell Sort

Shell Sort - Conceito

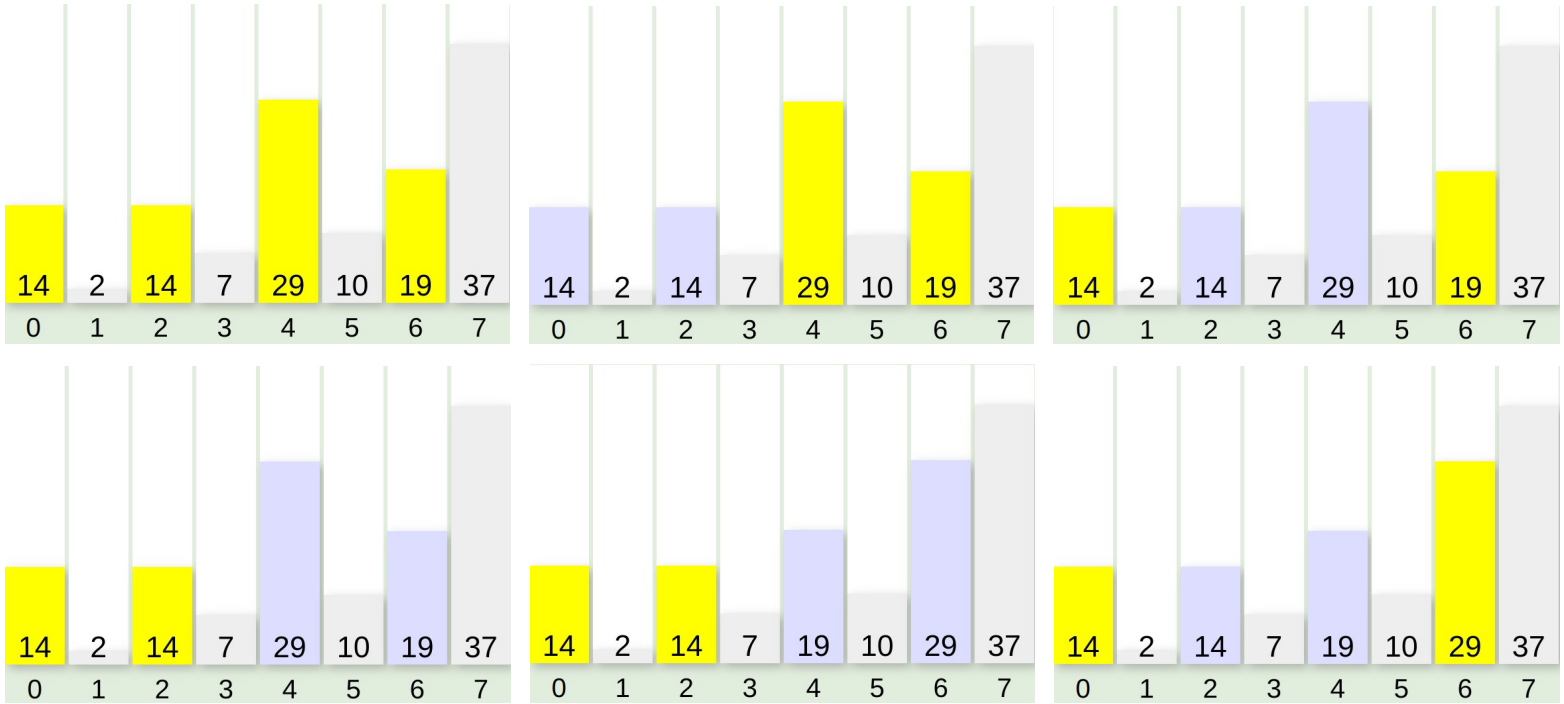
- Implementa melhorias no método Insertion Sort
- Funcionamento:
  - O vetor original é quebrado em subvetores
  - Cada subvetor é ordenado comparando e trocando os valores
  - Ao final de uma rodada, o subvetor é quebrado em mais um subvetor
  - Em um vetor de 20 elementos:
    - Primeira rodada: 10 elementos
    - Segunda rodada: 5 elementos
    - Terceira rodada: 2 ou 3 elementos
    - Terceira rodada: 1 elemento

Shell Sort em funcionamento (N = 4, 2, 1)

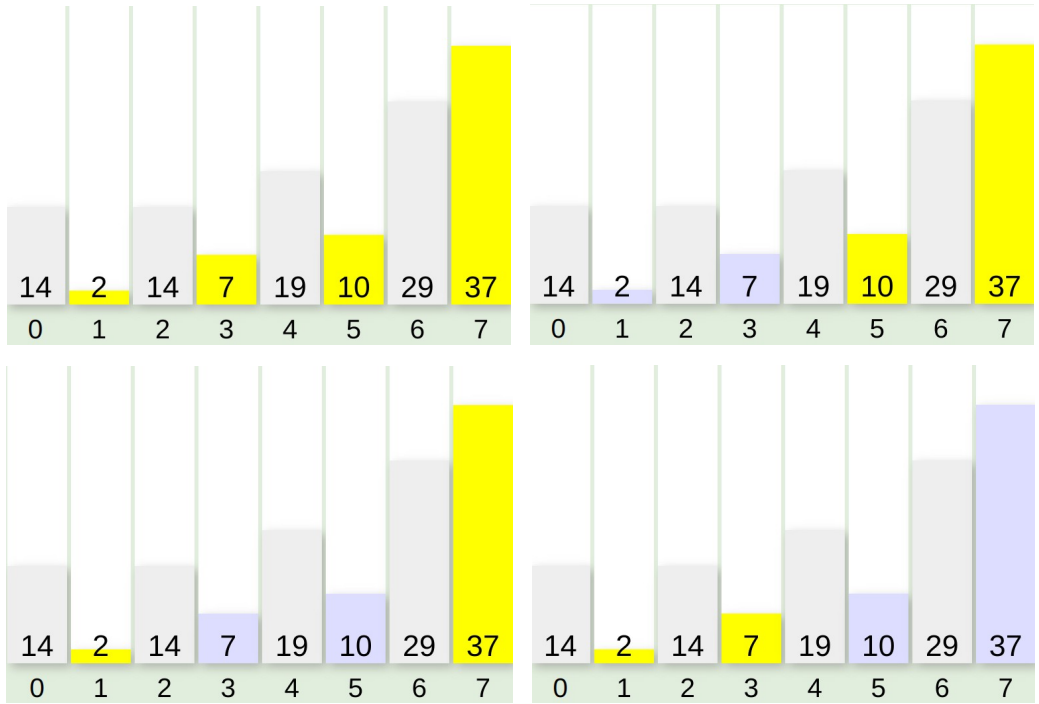


- https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html

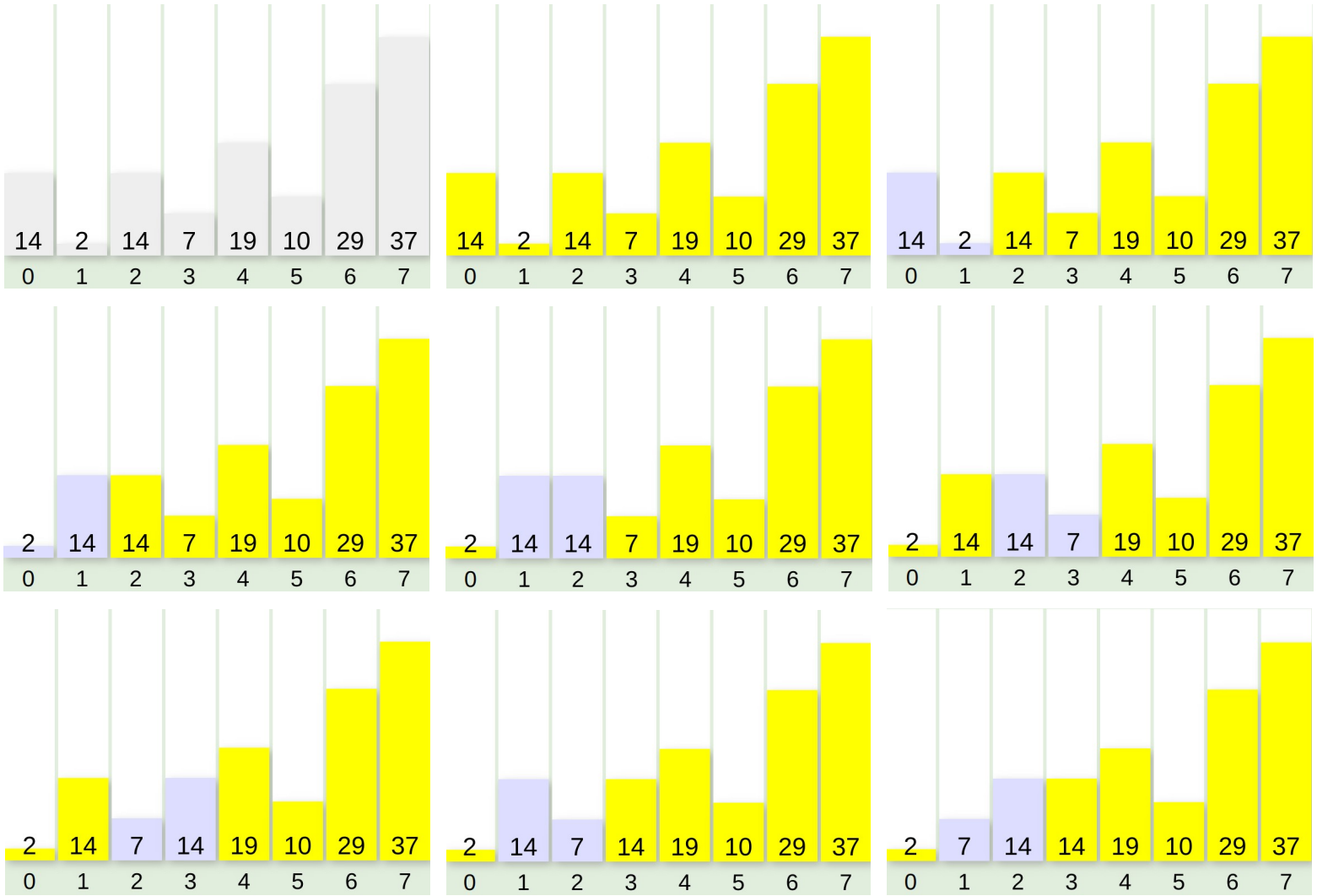
Shell Sort em funcionamento (N = 4, 2, 1)



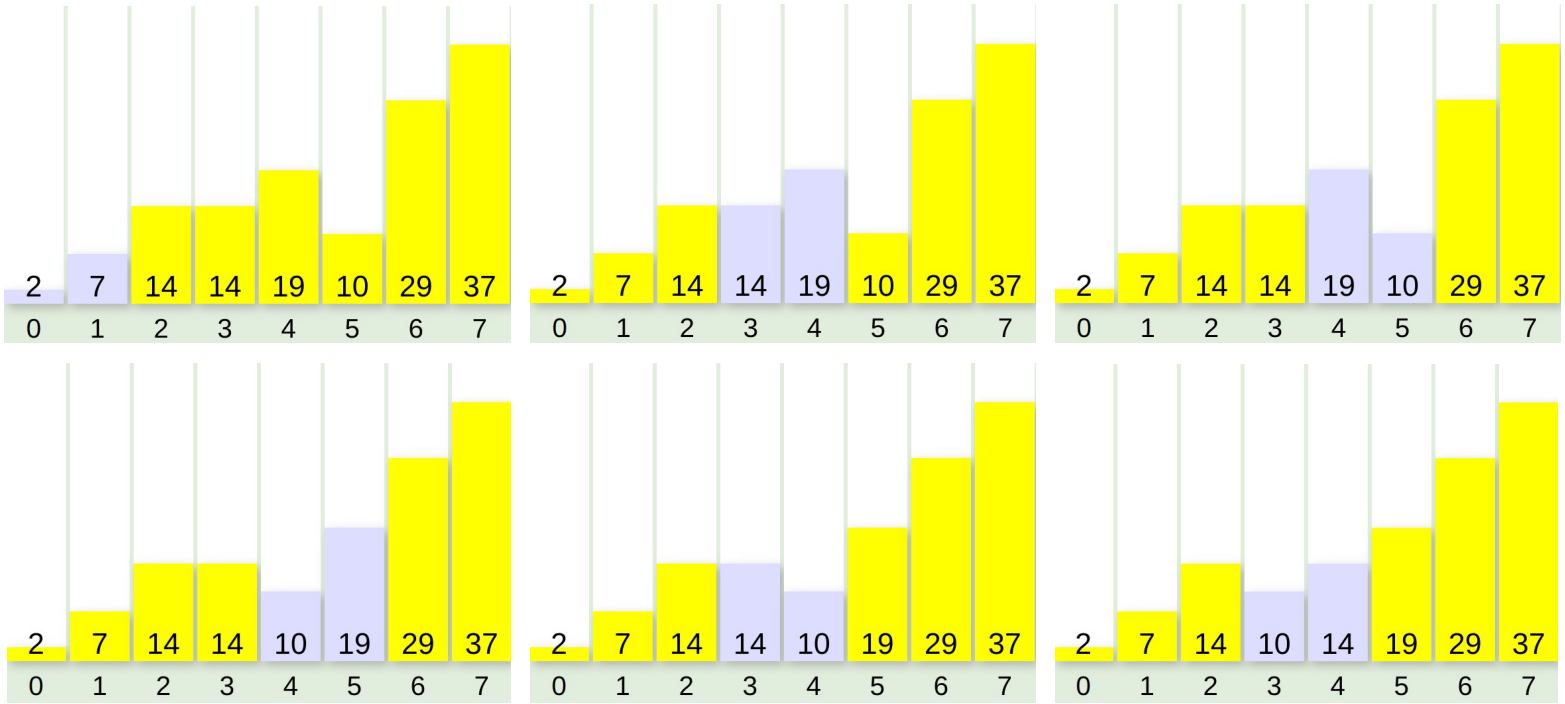
Shell Sort em funcionamento (N = 4, 2, 1)



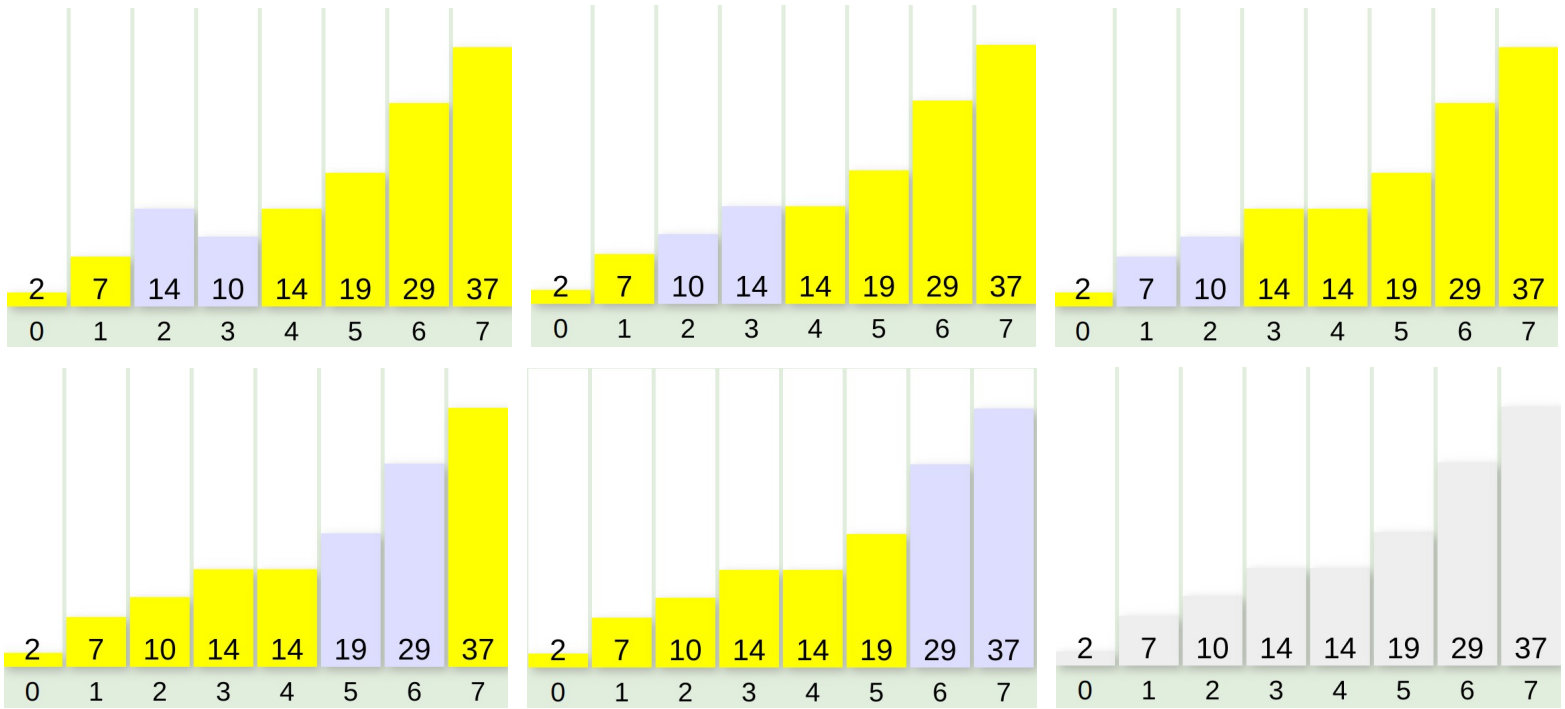
Shell Sort em funcionamento (N = 4, 2, 1)



Shell Sort em funcionamento (N = 4, 2, 1)



Shell Sort em funcionamento (N = 4, 2, 1)



Análises

- É possível definir a escolha dos intervalos, com isso o algoritmo pode se comportar de maneiras diferente
- Logo, a complexidade do algoritmo pode ser alterada
  - Pior caso: $O(n^2)$
  - Melhor caso: $O(n * \log n)$
- É melhor que o Selection Sort e que o Bubble Sort, pois ambos tem complexidade $O(n^2)$

Implementação: Shell Sort

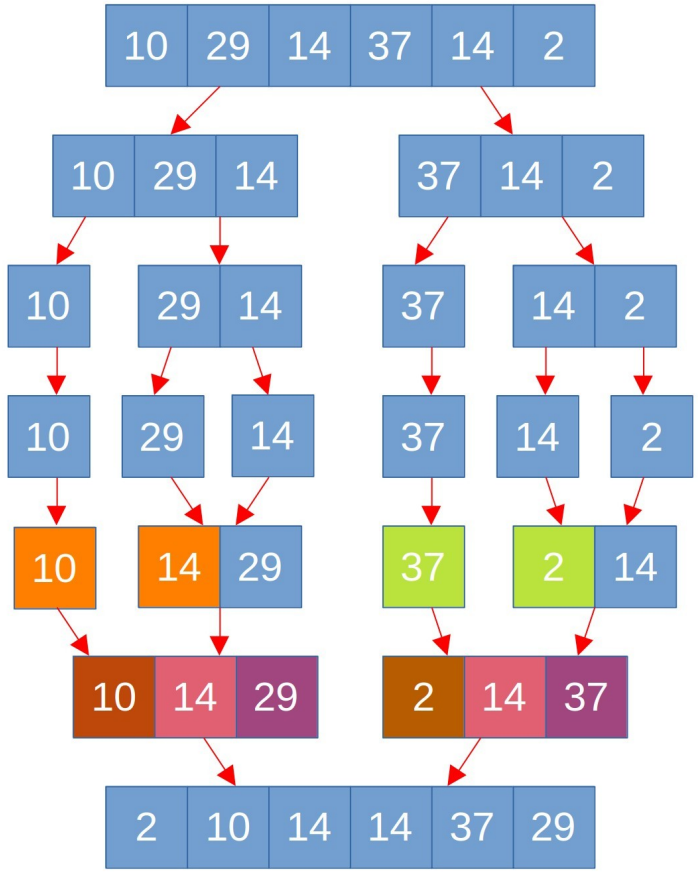
```python
import numpy as np

def shell_sort(vetor):
  intervalo = len(vetor) // 2

  while intervalo > 0:
    for i in range(intervalo, len(vetor)):
      temp = vetor[i]
      j = i
      while j >= intervalo and vetor[j - intervalo] > temp:
        vetor[j] = vetor[j - intervalo]
        j -= intervalo
      vetor[j] = temp
    intervalo //= 2

  return vetor
```


Merge Sort

Merge Sort

- Divisão do problema em subproblemas (dividir e conquistar)
- Divide o vetor continuamente pela metade, ordena e combina (merge)
- Implementação mais complexa que os algoritmos vistos anteriormente

Merge Sort em funcionamento



- https://visualgo.net/en/sorting

Implementação: Merge Sort

```
1  import numpy as np
2
3  def merge_sort(vetor):
4      # Separa os elementos do vetor, usando chamada recursiva
5      if len(vetor) > 1:
6          divisao = len(vetor) // 2
7          esquerda = vetor[:divisao].copy()
8          direita = vetor[divisao:].copy()
9
10         merge_sort(esquerda)
11         merge_sort(direita)
```

Implementação: Merge Sort - continuação

```
13          i = j = k = 0
14
15          # Ordena esquerda e direita
16          while i < len(esquerda) and j < len(direita):
17              if esquerda[i] < direita[j]:
18                  vetor[k] = esquerda[i]
19                  i += 1
20              else:
21                  vetor[k] = direita[j]
22                  j += 1
23              k += 1
```


Implementação: Merge Sort - continuação

```
25          # Ordenacao final
26          while i < len(esquerda):
27              vetor[k] = esquerda[i]
28              i += 1
29              k += 1
30          while j < len(direita):
31              vetor[k] = direita[j]
32              j += 1
33              k += 1
34
35      return vetor
36
37  if __name__ == '__main__':
38      print(np.array([10, 29, 14, 37, 14, 2]))
39      print(merge_sort(np.array([10, 29, 14, 37, 14, 2])))
```

Merge Sort - Complexidade

- Pior caso: O(n*log n)
- Melhor caso: O(n*log n)
- Lembrando que:
  - Bubble Sort: $O(n^2)$
  - Selection Sort: $O(n^2)$
  - Shell Sort: $O(n^2)$ no pior caso e O(n*log n) em média

Quick Sort

Quick Sort

- Ocorre a divisão em subvetores que são chamados recursivamente para ordenar os elementos
- Estratégia da divisão e conquista. Ideia de dividir a tarefa em subtarefas para obter melhor resultados.
- `https://visualgo.net/en/sorting`

Implementação: Quick Sort

```
1  import numpy as np
2
3  def particao(vetor, inicio, final):
4    pivo = vetor[final]
5    i = inicio - 1
6
7    for j in range(inicio, final):
8      if vetor[j] <= pivo:
9        i += 1
10       vetor[i], vetor[j] = vetor[j], vetor[i]
11   vetor[i + 1], vetor[final] = vetor[final], vetor[i + 1]
12   return i + 1
```


Implementação: Quick Sort

```
14 def quick_sort(vetor, inicio, final):
15   if inicio < final:
16     posicao = particao(vetor, inicio, final)
17     # Esquerda
18     quick_sort(vetor, inicio, posicao - 1)
19     # Direito
20     quick_sort(vetor, posicao + 1, final)
21   return vetor
22
23 vetor = np.array([10, 29, 14, 37, 14, 2])
24 print(quick_sort(vetor, 0, len(vetor) - 1))
```


Quick Sort - Complexidade

- Pior caso: O(n^2), que ocorre quando o pivô é o maior ou menor elemento
- Melhor caso: O(n*log n), o que indica que em média é mais rápido que os outros algoritmos

Atividade Avaliativa

Questão 01

- Realizar um comparativo do tempo de execução dos algoritmos de ordenação estudados
  - Criar um vetor aleatório de 5000 elementos inteiros com valores entre 0 e 1000 (utilize a biblioteca random)
  - Ordenar através de cada um dos métodos apresentados (algoritmos de ordenação já estão prontos!!)
  - Medir os tempos de cada ordenação (usar timeit)
  - ATENÇÃO: o mesmo vetor deve ser utilizado como entrada em cada algoritmo

Questão 02

- Comparar um vetor ordenado com os métodos de ordenação
  - Preencher um vetor ordenado com os elementos do vetor da questão anterior, medindo o tempo de execução
  - Comparar o tempo obtido com o tempo de execução dos algoritmos de ordenação
- O trabalho deverá ser apresentado na aula do dia 18/07/2022