

Aulas 11 E 12

Métodos de Ordenação



Algoritmos e Estrutura de Dados II

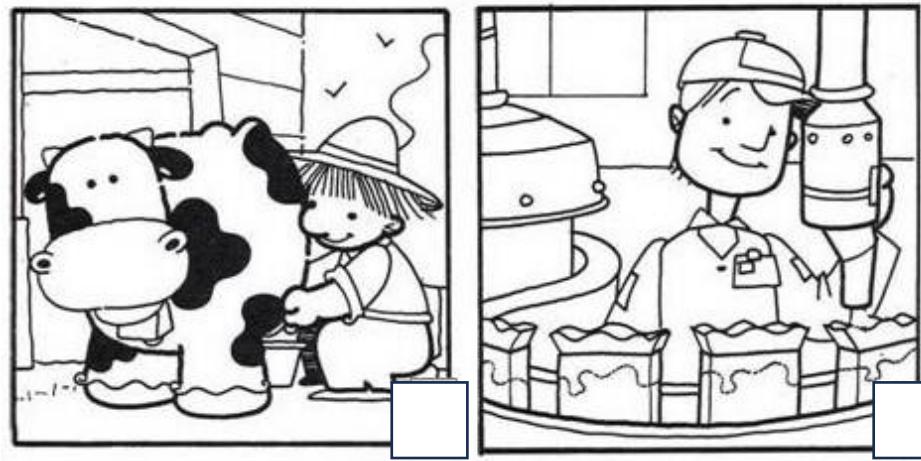
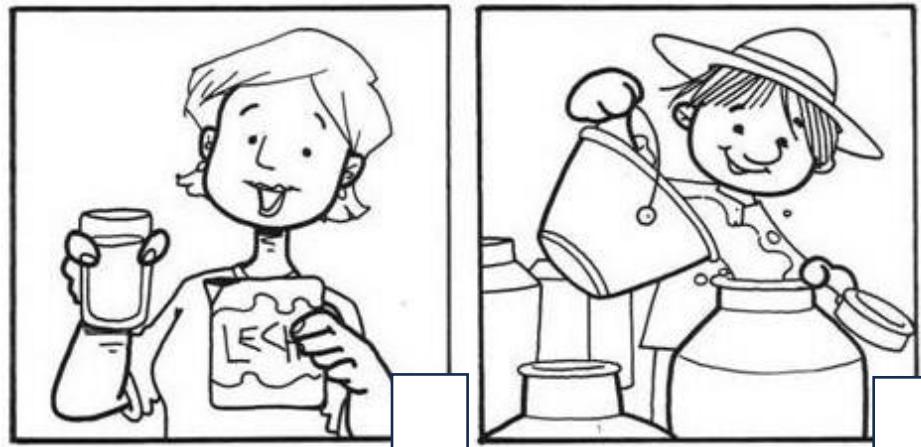


2º Semestre – CDN

Conteúdo Programático - Planejamento

| Semana | Data | Temas/Atividades |
|--------|-------|---|
| 1 | 07/08 | Acolhimento e Boas-vindas! Introdução a Disciplina. Formas de Avaliação e Percurso Pedagógico. |
| 2 | 14/08 | Tipo de dado abstrato. Introdução a Estrutura de Dados. |
| 3 | 21/08 | Complexidade de Algoritmos |
| 4 | 28/08 | Vetores não-Ordenados e busca sequencial |
| 5 | 04/09 | Vetores Ordenados e busca binária |
| 6 | 11/09 | Revisão de Programação Orientada a Objetos (POO) |
| 7 | 18/09 | Pilhas |
| 8 | 25/09 | Filas |
| 9 | 02/10 | Listas encadeadas |
| 10 | 09/10 | Recursão |
| 11 | 16/10 | Primeira Avaliação Formal (P1). Correção da Avaliação após o intervalo. |
| 12 | 18/10 | Algoritmos de Ordenação |
| 13 | 23/10 | Algoritmos de Ordenação |
| 14 | 30/10 | Arvores |
| 15 | 06/11 | Grafos |
| 16 | 13/11 | Segunda Avaliação Formal (P2). Correção da Avaliação após o intervalo |
| 17 | 27/11 | Apresentação PI do curso de CDN |
| 18 | 04/12 | Tabela Hash (tabela de espalhamento) – Tópico extra. |
| 19 | 11/12 | Exame / Avaliação Substitutiva. Correção da Avaliação após o intervalo. Finalização Disciplina |
| 20 | 18/12 | Finalização da disciplina. |

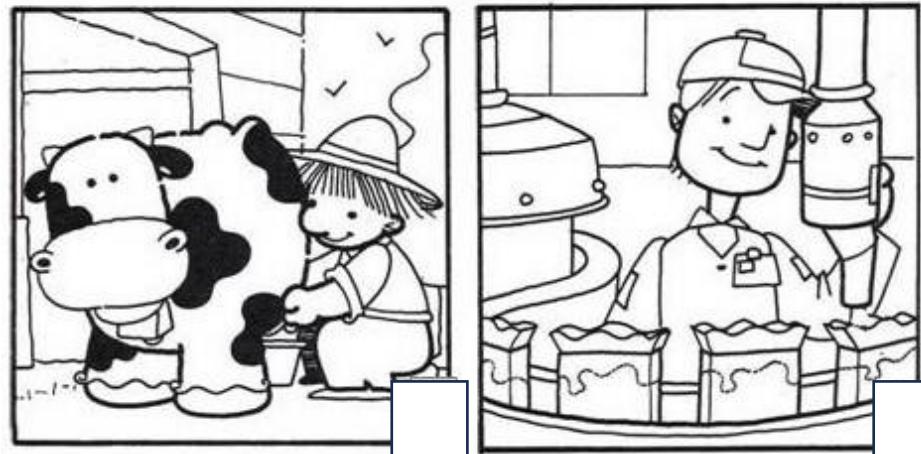
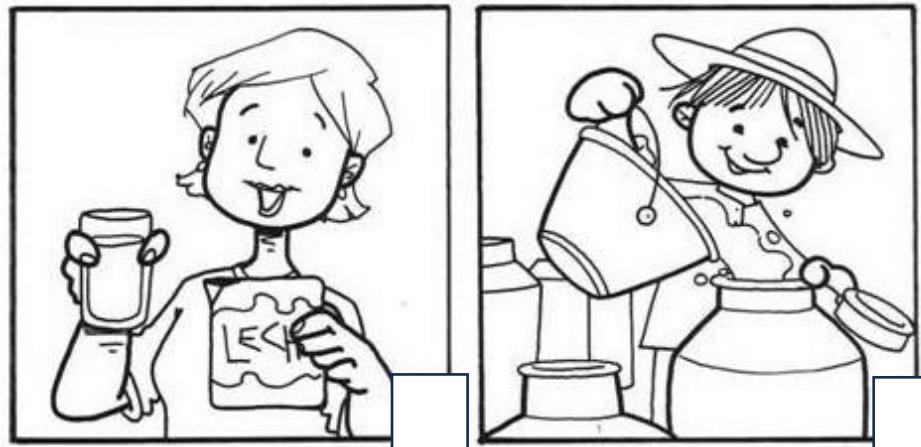
Qual a ordem correta?



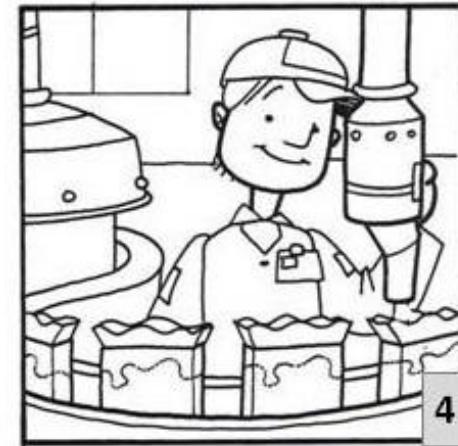
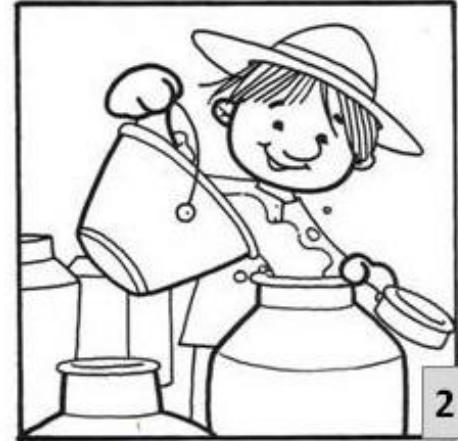
Qual a ordem correta?

Dica:

O ciclo de vida do leite!!



Qual a ordem correta?

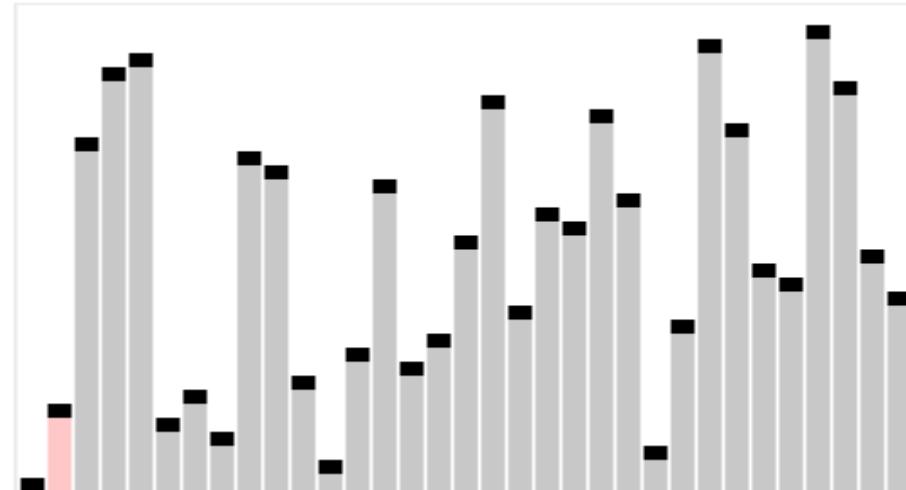


Algoritmos de ordenação de dados

Ser capaz de ordenar os elementos de um conjunto de dados é uma das tarefas básicas mais requisitadas por aplicações computacionais.

Como exemplo, podemos citar a busca binária, um algoritmo de busca muito mais eficiente que a simples busca sequencial. Buscar elementos em conjuntos ordenados é bem mais rápido do que em conjuntos desordenados.

Existem diversos algoritmos de ordenação, sendo alguns mais eficientes do que outros. Vamos ver alguns deles: Bubblesort, Selectionsort, Insertionsort, Shellsort, Quicksort e Mergesort.



Bubble Sort

O algoritmo Bubblesort é uma das abordagens mais simplistas para a ordenação de dados.

A ideia básica consiste em percorrer o vetor diversas vezes, em cada passagem fazendo flutuar para o topo da lista (posição mais a direita possível) o maior elemento da sequência.

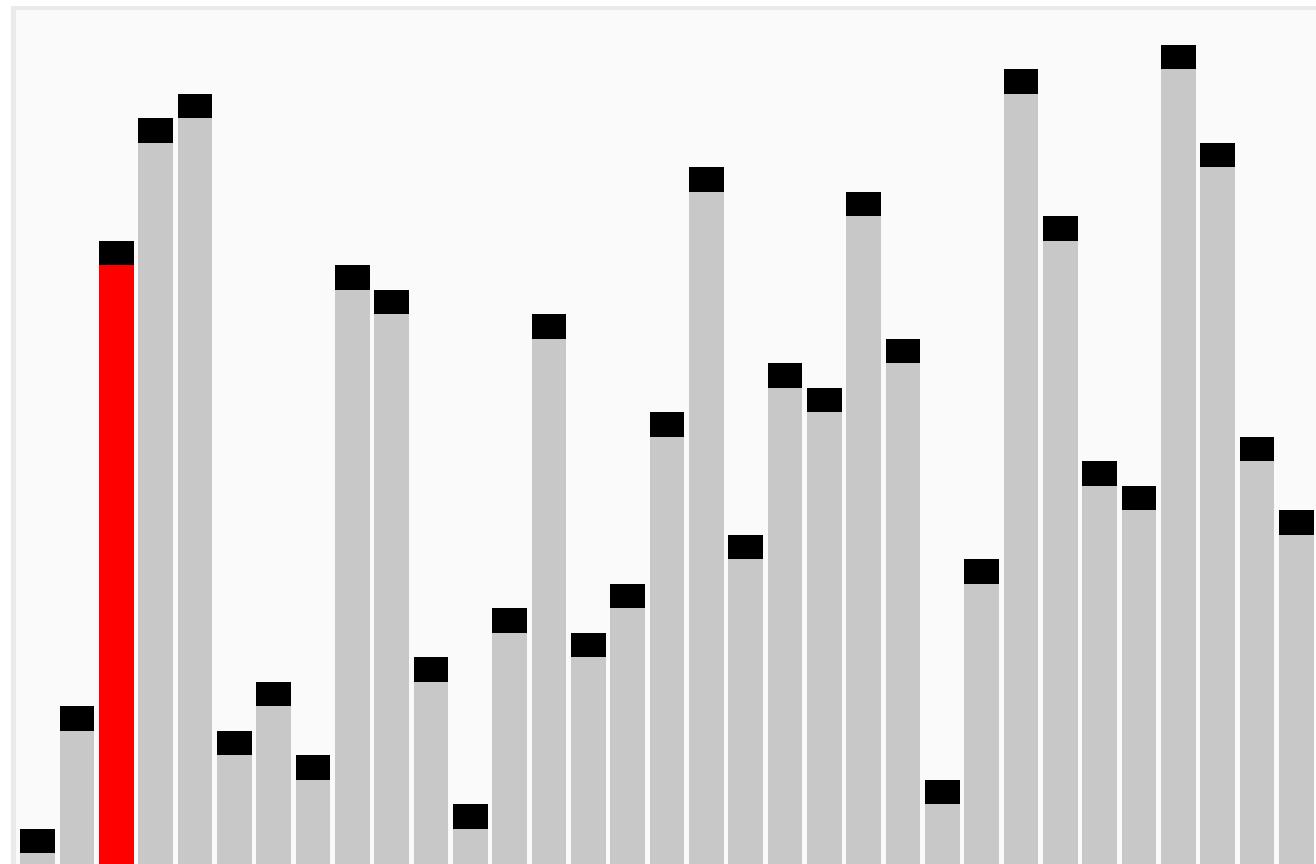
Esse padrão de movimentação lembra a forma como as bolhas em um tanque procuram seu próprio nível, e disso vem o nome do algoritmo (também conhecido como o método bolha)

Embora no melhor caso esse algoritmo necessite de apenas n operações relevantes, onde n representa o número de elementos no vetor, no pior caso são feitas n^2 operações.

Portanto, diz-se que a complexidade do método é de **ordem quadrática**. Por essa razão, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

Bubble Sort

A ideia básica consiste em percorrer o vetor diversas vezes, em cada passagem fazendo flutuar para o topo da lista (posição mais a direita possível) o maior elemento da sequência.



Bubble Sort

A implementação é bem simples...

```
def bubble_sort(vetor):
    n = len(vetor)

    for i in range(n):
        for j in range(0, n - i - 1):
            if vetor[j] > vetor[j + 1]:
                vetor[j], vetor[j + 1] = vetor[j+1], vetor[j]

    return vetor
```

VAMOS PARA A PRÁTICA ?!!!



Selection Sort

A ordenação por seleção é um método baseado em se passar o menor valor do vetor para a primeira posição mais a esquerda disponível, depois o de segundo menor valor para a segunda posição e assim sucessivamente, com os $n - 1$ elementos restantes.

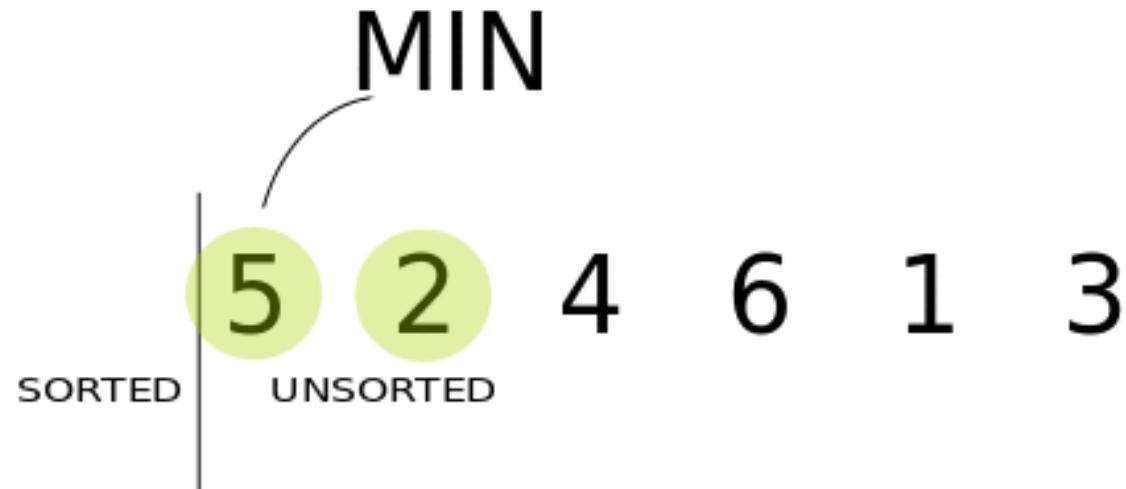
Esse algoritmo compara a cada iteração um elemento com os demais, visando encontrar o menor. A complexidade desse algoritmo será sempre de ordem quadrática, isto é o número de operações realizadas depende do quadrado do tamanho do vetor de entrada.

Algumas vantagens desse método são: é um algoritmo simples de ser implementado, não usa um vetor auxiliar e portanto ocupa pouca memória, é um dos mais velozes para vetores pequenos.

Como desvantagens podemos citar o fato de que ele não é muito eficiente para grandes vetores.

Selection Sort

Processo: passar o menor valor do vetor para a primeira posição mais a esquerda disponível, depois o de segundo menor valor para a segunda posição e assim sucessivamente, com os $n - 1$ elementos restantes.



Selection Sort

A implementação é a seguinte:

```
def selection_sort(vetor):
    n = len(vetor)

    for i in range(n):
        id_minimo = i
        for j in range(i + 1, n):
            if vetor[id_minimo] > vetor[j]:
                id_minimo = j
        vetor[i], vetor[id_minimo] = vetor[id_minimo], vetor[i]

    return vetor
```

VAMOS PARA A PRÁTICA ?!!!



Insertion Sort

Insertion sort, ou **ordenação por inserção**, é o algoritmo de ordenação que, dado um vetor inicial constrói um vetor final com um elemento de cada vez, uma inserção por vez.

Assim como algoritmos de ordenação quadráticos, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Podemos fazer uma comparação do Insertion sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas.



Insertion Sort

Podemos fazer uma comparação do Insertion sort com o modo de como algumas pessoas organizam um baralho num jogo de cartas.

Imagine que você está jogando cartas.

Você está com as cartas na mão e elas estão ordenadas.



Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.

A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta.

Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas.

Então, você recebe outra carta e repete o mesmo procedimento.

Então outra carta, e outra, e assim por diante, até você não receber mais cartas.

Insertion Sort

Esta é a ideia por trás da ordenação por inserção. Percorra as posições do vetor, começando com o índice zero. Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto no sub-vetor ordenado à esquerda daquela posição.

6 5 3 1 8 7 2 4

Insertion Sort

A implementação 1 é a seguinte:

```
# insertion sort 1
def insertion_sort1(vetor):
    n = len(vetor)

    for i in range(1, n):
        marcado = vetor[i] # elemento que vamos posicionar na parte ordenada

        j = i - 1
        # desloca elementos maiores que ‘marcado’ uma posição a frente
        while j >= 0 and marcado < vetor[j]:
            vetor[j + 1] = vetor[j]
            j -= 1

        # Insere o elemento marcado na posição correta (após sair do loop,
        # j estará em -1 ou em posição com elemento menor ou igual ao marcado).
        vetor[j + 1] = marcado

    return vetor
```

Insertion Sort

A implementação 2 é a seguinte:

```
# insertion sort 2
def insertion_sort2(vetor):
    # Percorre cada elemento do vetor a partir do segundo
    for i in range(1, len(vetor)):
        k = i
        # vai trocando o vetor[k] com seu vizinho à esquerda
        # enquanto for menor
        while k > 0 and vetor[k] < vetor[k-1]:
            vetor[k], vetor[k-1] = vetor[k-1], vetor[k]
            k = k - 1
```

VAMOS PARA A PRÁTICA ?!!!



Shell Sort

Shell sort é o mais eficiente algoritmo de classificação **dentre os de complexidade quadrática**.

É um refinamento do método de inserção (Insertion Sort).

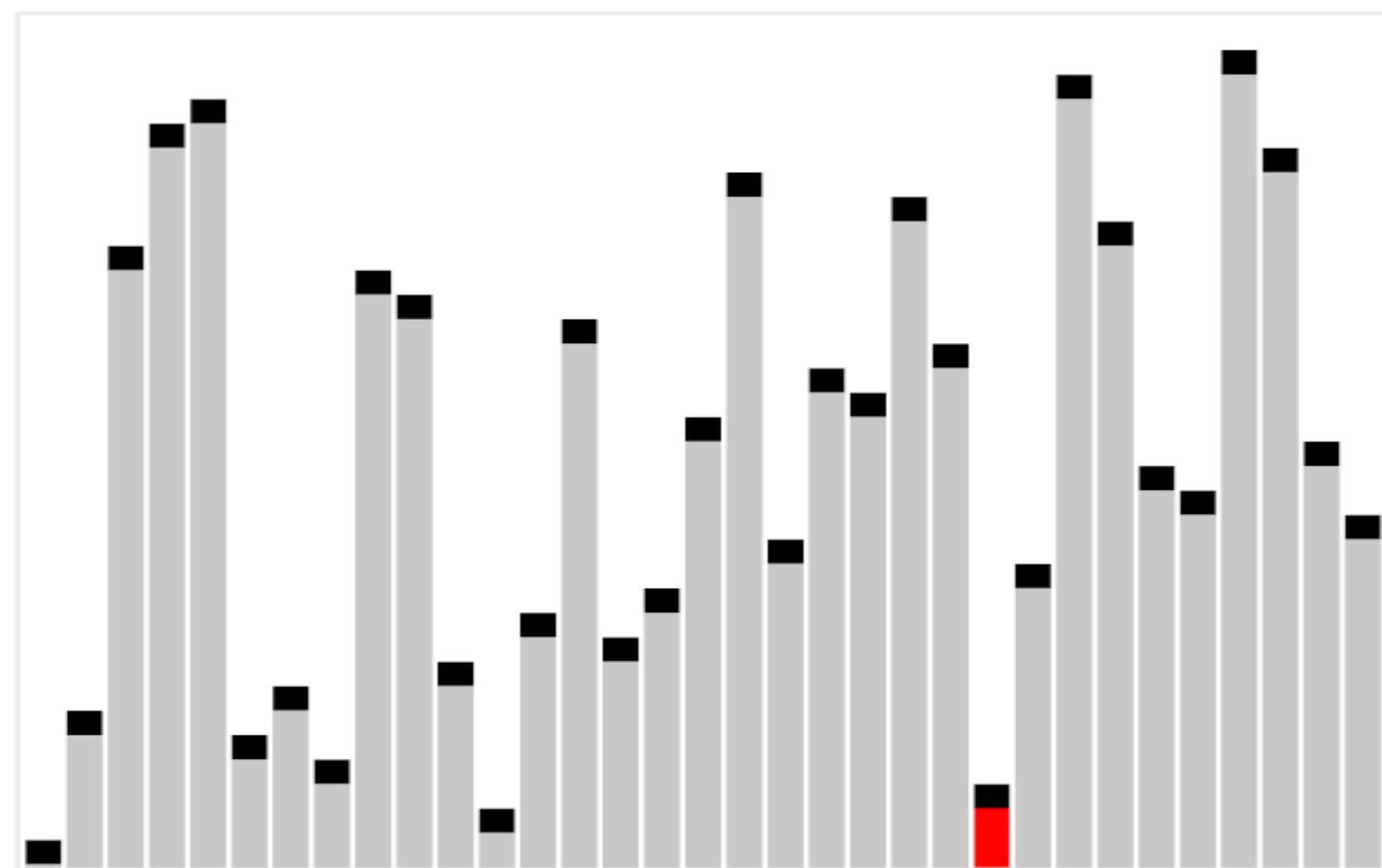
O algoritmo difere do método de inserção pelo fato de no lugar de considerar o array a ser ordenado como um único segmento, ele considera **vários segmentos** sendo aplicado o método de inserção (Insertion Sort) em cada um deles.

Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores.

Nos grupos menores é aplicado o método da ordenação por inserção. Implementações do algoritmo.

Shell Sort

Deve-se escolher um intervalo inicial h . Em seguida, dividir a coleção em vários subgrupos, de forma que elementos em um mesmo subgrupo estão a uma distância h entre si. O passo seguinte é a ordenação de cada subgrupo usando o algoritmo de Ordenação por Inserção. Diminua o valor do intervalo h e repita os passos acima até que o intervalo h seja maior que 0. Ao final, a coleção está ordenada.



Shell Sort

A implementação 1 é a seguinte:

```
def shell_sort1(vetor):
    h = 1
    n = len(vetor)
    while h > 0:
        for i in range(h, n):
            c = vetor[i]
            j = i
            while j >= h and c < vetor[j - h]:
                vetor[j] = vetor[j - h]
                j = j - h
            vetor[j] = c
        h = int(h / 2.2)
    return vetor
```

Shell Sort

A implementação 2 é a seguinte:

```
def shell_sort2(vetor):
    n = len(vetor)
    intervalo = n // 2 # intervalo inicial: metade do tamanho

    while intervalo > 0:
        # percorre do índice intervalo até o final
        for i in range(intervalo, n):
            temp = vetor[i] # elemento a ser inserido na posição correta do subgrupo
            j = i
            # desloca o elemento temp para sua posição correta dentro do subgrupo atual
            while j >= intervalo and vetor[j - intervalo] > temp:
                vetor[j] = vetor[j - intervalo]
                j -= intervalo
            vetor[j] = temp
            # para a próxima interação, reduz o intervalo pela metade
            intervalo //= 2

    return vetor
```

VAMOS PARA A PRÁTICA ?!!!



Quicksort

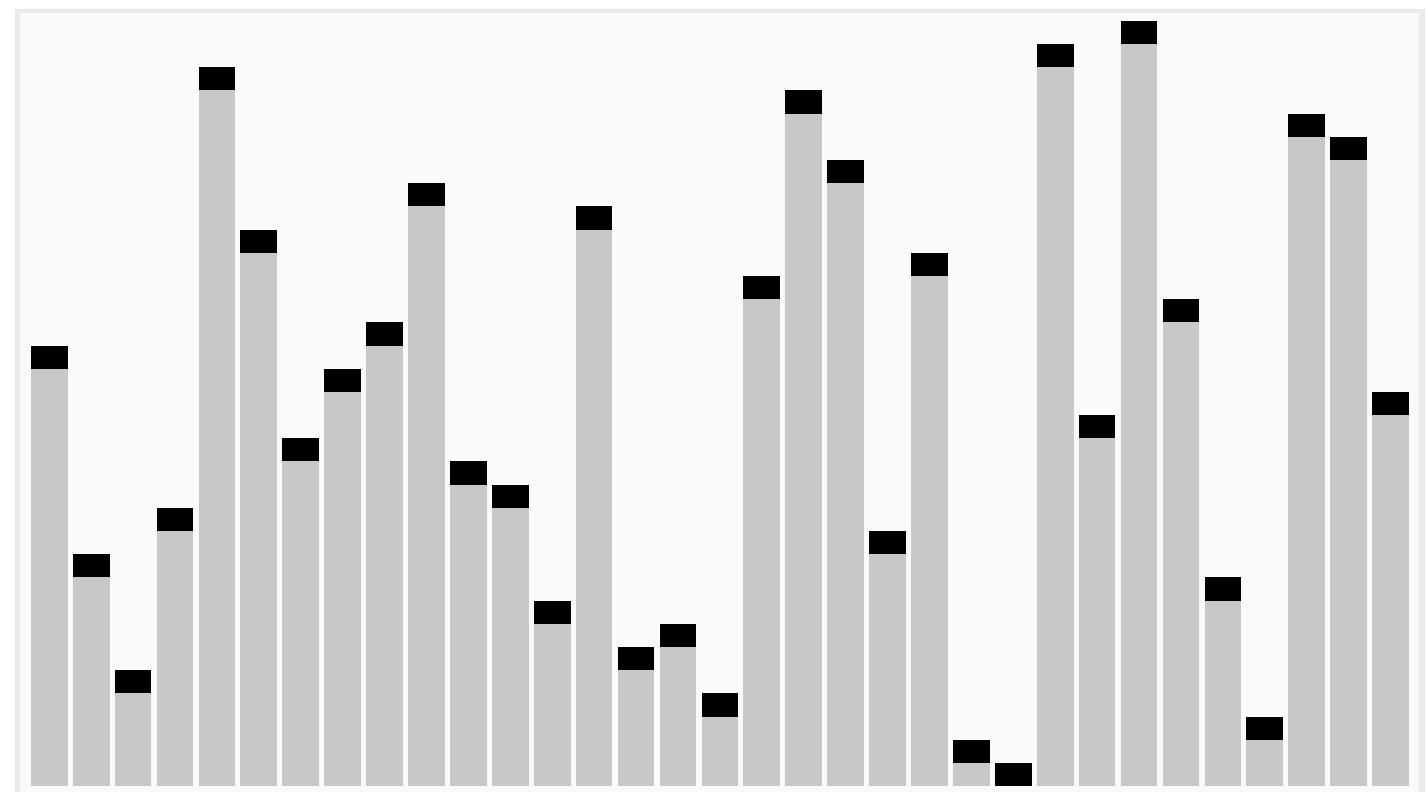
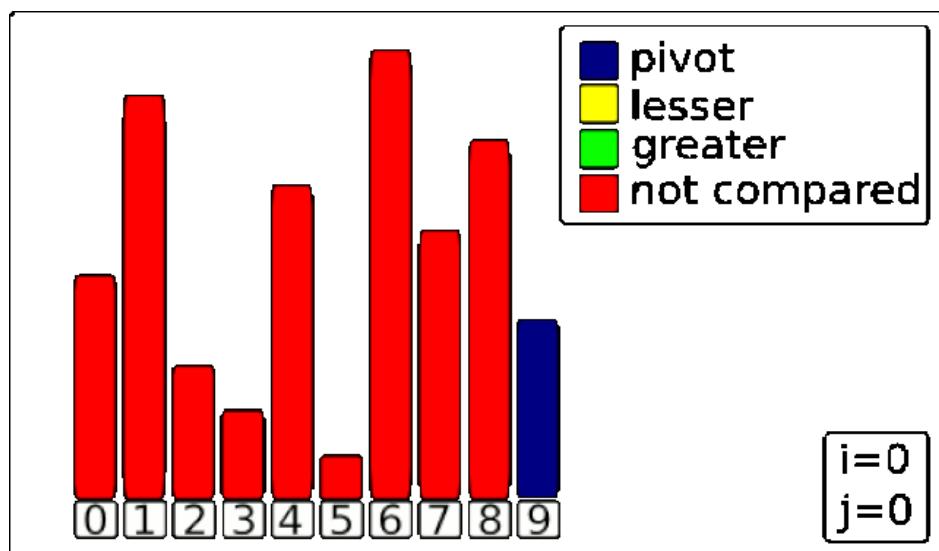
O algoritmo Quicksort segue o paradigma conhecido como “**Dividir para Conquistar**” pois ele quebra o problema de ordenar um vetor em subproblemas menores, mais fáceis e rápidos de serem resolvidos.

Primeiramente, o método divide o vetor original em duas partes: os elementos menores que o pivô (tipicamente escolhido como o primeiro ou último elemento do conjunto). O método então ordena essas partes de maneira recursiva. O algoritmo pode ser dividido em 3 passos principais:

1. Escolha do pivô: em geral, o pivô é o primeiro ou último elemento do conjunto.
2. Particionamento: reorganizar o vetor de modo que todos os elementos menores que o pivô apareçam antes dele (a esquerda) e os elementos maiores apareçam após ele (a direita). Ao término dessa etapa o pivô estará em sua posição final
3. Ordenação: recursivamente aplicar os passos acima aos sub-vetores produzidos durante o particionamento. O caso limite da recursão é o sub-vetor de tamanho 1, que já está ordenado.

Quicksort

Este método escolhe um pivô tipicamente no início ou no final do array. O Particiona recebe como parâmetro dois índices do array, lo e hi , que será a parte do array a ser particionada, então escolhe-se um índice i e percorre-se o array usando outro índice j realizando trocas, quando necessário, a fim de que todos os elementos menores ou iguais ao pivô fiquem antes do índice i e os elementos $i + 1$ até hi , ou $j - 1$, sejam maiores que o pivô.



Quicksort

A implementação é a seguinte:

```
def particao(vetor, inicio, final):
    pivo = vetor[final] # escolhe o último elemento para pivô
    i = inicio - 1      # índice do “limite” dos menores

    for j in range(inicio, final):
        # se o elemento atual é menor ou igual ao pivô
        # incluí-lo na seção dos “menores”
        if vetor[j] <= pivo:
            i += 1
            # troca elemento menor para a frente
            vetor[i], vetor[j] = vetor[j], vetor[i]
    # coloca o pivô na posição correta (i+1)
    vetor[i + 1], vetor[final] = vetor[final], vetor[i + 1]
    return i + 1
```

Quicksort

A implementação é a seguinte:

```
def quick_sort(vetor, inicio=0, final=(len(vetor)-1)):  
    if inicio < final:  
        # realiza a partição e obtém a posição do pivô ordenado  
        posicao = particao(vetor, inicio, final)  
        # Ordena a partição à esquerda do pivô  
        quick_sort(vetor, inicio, posicao - 1)  
        # Ordena a partição à direita do pivô  
        quick_sort(vetor, posicao + 1, final)  
    return vetor
```

VAMOS PARA A PRÁTICA ?!!!



Mergesort (ordenação por intercalação)

O algoritmo Mergesort utiliza a abordagem **Dividir para Conquistar**. A ideia básica consiste em dividir o problema em vários subproblemas e resolver esses subproblemas através da recursividade e depois conquistar, o que é feito após todos os subproblemas terem sido resolvidos através da união das resoluções dos subproblemas menores.

Trata-se de um algoritmo recursivo que divide uma lista continuamente pela metade.

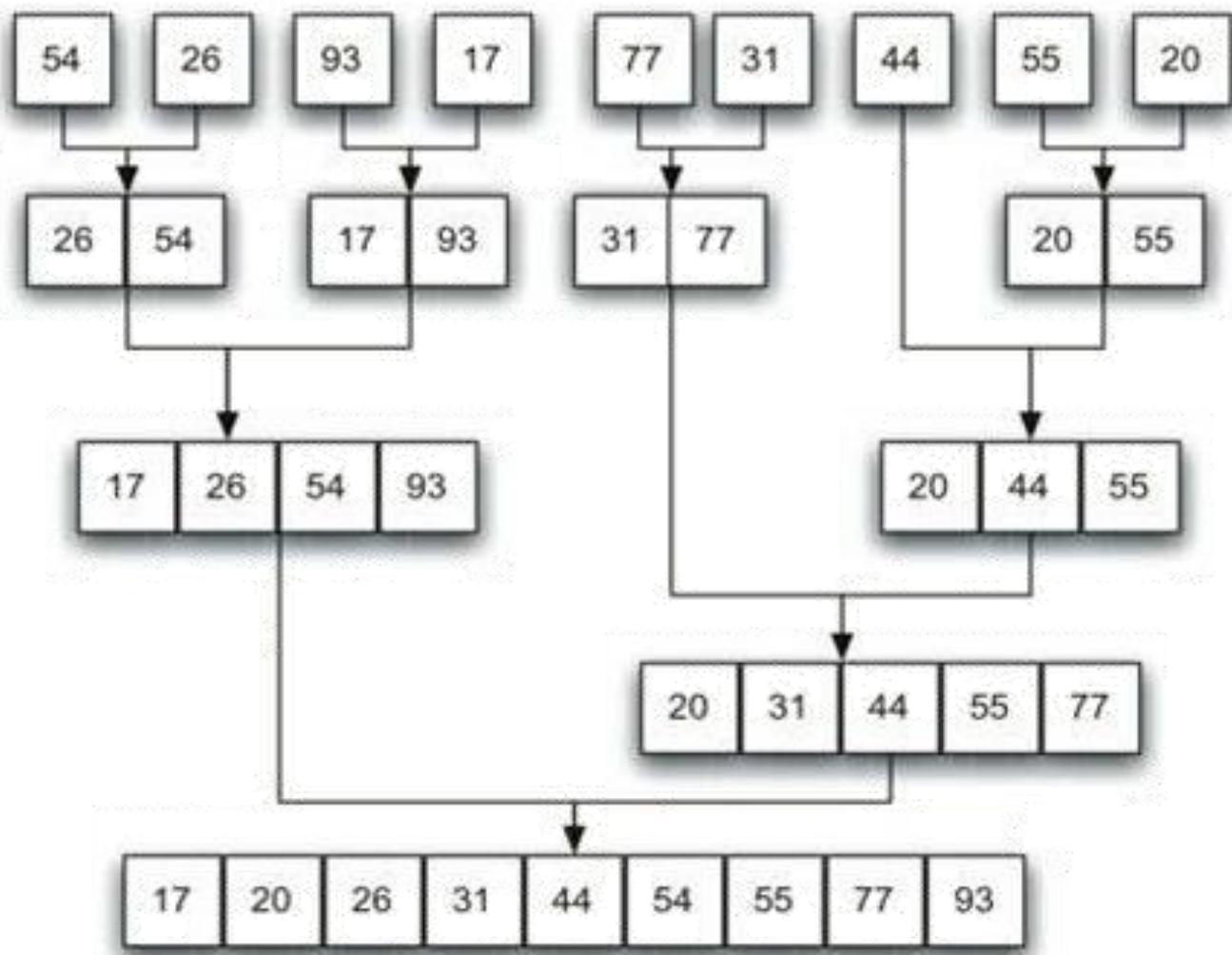
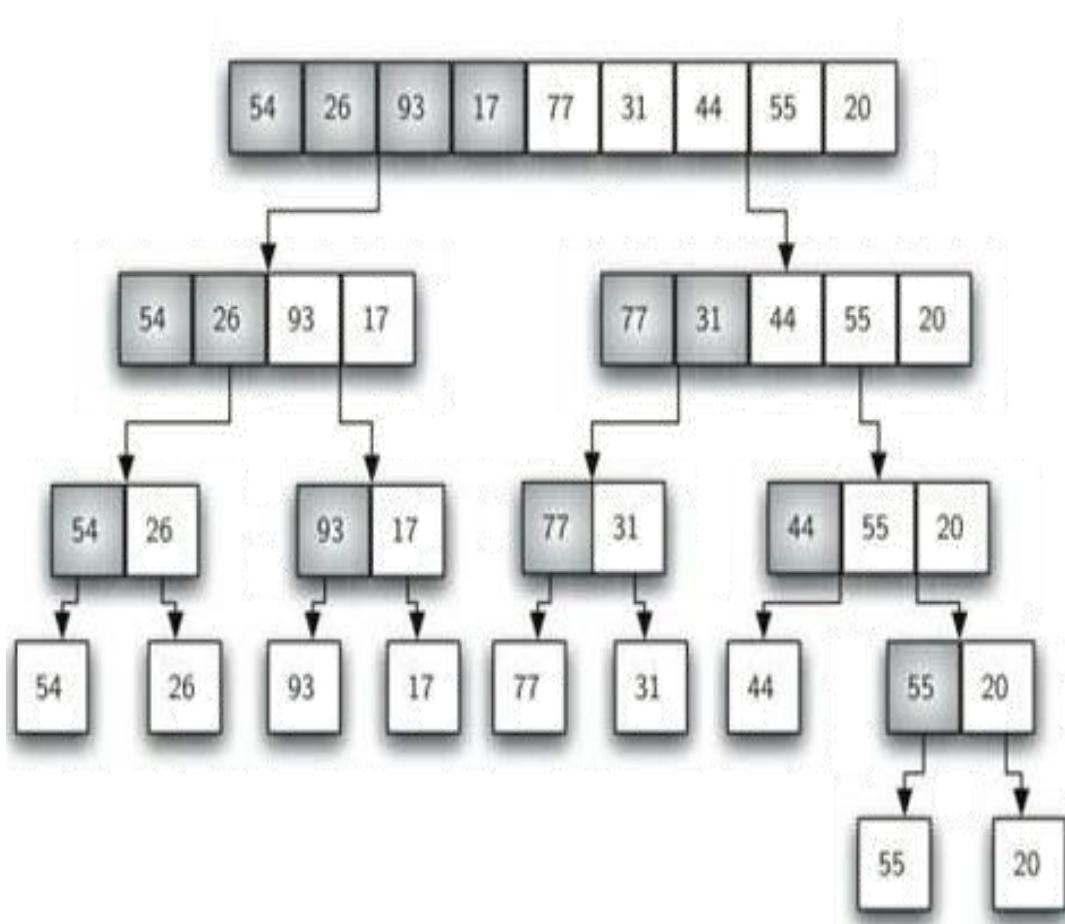
- Se a lista estiver vazia ou tiver um único elemento, ela está ordenada por definição (o caso base).
- Se a lista tiver mais de um elemento, dividimos a lista e invocamos recursivamente um Mergesort em ambas as metades.

Assim que as metades estiverem ordenadas, a operação fundamental, chamada de intercalação, é realizada.

Intercalar é o processo de pegar duas listas menores ordenadas e combiná-las de modo a formar uma lista nova, única e ordenada.

Mergesort (ordenação por intercalação)

A imagem apresenta as duas fases do algoritmo: dividir e intercalar (conquistar)



Mergesort

De forma animada, como funciona o algoritmo mergesort...

6 5 3 1 8 7 2 4

Mergesort

A função MergeSort começa perguntando pelo caso base. Se o tamanho da lista for menor ou igual a um, então já temos uma lista ordenada e nenhum processamento adicional é necessário. Se, por outro lado, o tamanho da lista for maior do que um, então usamos a operação de slice do Python para extrair a metades esquerda e direita. É importante observar que a lista pode não ter um número par de elementos. Isso, contudo, não importa, já que a diferença de tamanho entre as listas será de apenas um elemento.

Quando a função MergeSort retorna da recursão (após a chamada nas metades esquerda e direita), elas já estão ordenadas. O resto da função é responsável por intercalar as duas listas ordenadas menores em uma lista ordenada maior. Note que a operação de intercalação coloca um item por vez de volta na lista original (vetor) ao tomar repetidamente o menor item das listas ordenadas.

Mergesort

A implementação
é a seguinte:

```
def merge_sort(vetor):
    # se o vetor tiver mais de 1 elemento, precisa dividir
    if len(vetor) > 1:
        divisao = len(vetor) // 2
        # divide em dois vetores (direita e esquerda)
        esquerda = vetor[:divisao].copy()
        direita = vetor[divisao:].copy()
        # ordena recursivamente cada metade
        merge_sort(esquerda)
        merge_sort(direita)

        # intercala as 2 metades ordenadas de volta em vetor
        i, j, k = 0, 0, 0

        # Ordena esquerda e direita
        # enquanto há elementos em ambas as listas
        while i < len(esquerda) and j < len(direita):
            if esquerda[i] < direita[j]:
                vetor[k] = esquerda[i]
                i += 1
            else:
                vetor[k] = direita[j]
                j += 1
            k += 1
```

```
# Ordenação final
# se acabarem os elementos
# de 1 das listas, copia o
# restante da outra (ord)
while i < len(esquerda):
    vetor[k] = esquerda[i]
    i += 1
    k += 1
while j < len(direita):
    vetor[k] = direita[j]
    j += 1
    k += 1
return vetor
```

VAMOS PARA A PRÁTICA ?!!!



Comparativo...

In Place
↓

| Algoritmo | Comparações | | | Movimentações | | | Espaço | Estável | In situ | | |
|-----------|-----------------------------------|----------|----------|---------------|----------|------|--------|---------|---------|--|--|
| | Melhor | Médio | Pior | Melhor | Médio | Pior | | | | | |
| Bubble | $O(n^2)$ | | | $O(n^2)$ | | | $O(1)$ | Sim | Sim | | |
| Selection | $O(n^2)$ | | | $O(n)$ | | | $O(1)$ | Não * | Sim | | |
| Insertion | $O(n)$ | $O(n^2)$ | | $O(n)$ | $O(n^2)$ | | $O(1)$ | Sim | Sim | | |
| Merge | $O(n \log n)$ | | | – | | | $O(n)$ | Sim | Não | | |
| Quick | $O(n \log n)$ | | $O(n^2)$ | – | | | $O(n)$ | Não * | Sim | | |
| Shell | $O(n^{1.25})$ ou $O(n (\ln n)^2)$ | | | – | | | $O(1)$ | Não | Sim | | |

* Existem versões estáveis.

$O(\log n)$
Recursão

Comparativo...

| Algoritmos | Tempo de Execução (vetor c/ 5000 números) |
|----------------|--|
| Bubble Sort | 2,57 seg |
| Selection Sort | 1,27 seg |
| Insertion Sort | 1,8 seg |
| Shell Sort | 1,26 seg |
| Quick Sort | 26,5 ms |
| Merge Sort | 18,2 ms |

1. Exercícios... (ENADE)

Julgue os itens a seguir, acerca de algoritmos para ordenação.

- I. O algoritmo de ordenação por inserção tem complexidade $O(n * \log n)$
- II. Um algoritmo de ordenação é dito estável caso ele não altere a posição relativa de elementos de mesmo valor.
- III. No algoritmo **quicksort**, a escolha do elemento pivô influencia o desempenho do algoritmo.
- IV. O **bubble-sort** e o algoritmo de ordenação por inserção fazem, em média, o mesmo número de comparações.

Estão certos apenas os itens:

- A) I e II.
- B) I e III.
- C) II e IV.
- D) I, III e IV.
- E) II, III e IV.

1. Exercícios... (ENADE)

- **Item I: FALSO.** A complexidade do algoritmo de ordenação por **inserção** é $O(n^2)$ no pior caso, e não $O(n \times \log n)$, que é a complexidade de algoritmos como **merge sort** ou **quicksort** no caso médio e no melhor caso.
- **Item II: VERDADEIRO.** Um algoritmo de ordenação é dito **estável** se ele preserva a ordem relativa de elementos com valores iguais. Isso significa que, se dois elementos possuem o mesmo valor, eles permanecerão na mesma ordem após a ordenação.
- **Item III: VERDADEIRO.** No algoritmo **quicksort**, a escolha do pivô é fundamental para o desempenho. Um pivô mal escolhido (por exemplo, sempre o menor ou o maior valor) pode levar o algoritmo a um desempenho $O(n^2)$ no pior caso.
- **Item IV: VERDADEIRO.** Tanto o **bubble-sort** quanto o algoritmo de **inserção** realizam, em média, um número semelhante de comparações, ambos com complexidade $O(n^2)$, embora o algoritmo de inserção geralmente tenha melhor desempenho na prática.
- Portanto, os itens II, III, e IV estão corretos, levando à resposta E.

2. Exercícios... (ENADE)

Considere o algoritmo que implementa o seguinte processo: uma coleção desordenada de elementos é dividida em duas metades e cada metade é utilizada como argumento para a reaplicação recursiva do procedimento. Os resultados das duas reaplicações são, então, combinados pela intercalação dos elementos de ambas, resultando em uma coleção ordenada. Qual é a complexidade desse algoritmo?

Alternativas:

- A) $O(n^2)$
- B) $O(n^{2n})$
- C) $O(2^n)$
- D) $O(\log n \times \log n)$
- E) $O(n \times \log n)$

2. Exercícios... (ENADE)

O algoritmo descrito no enunciado é o **merge sort**, que é um algoritmo de ordenação por divisão e conquista. Ele funciona dividindo a coleção de elementos ao meio, chamando recursivamente o processo para cada metade e, em seguida, intercalando as duas partes já ordenadas.

- **Divisão recursiva:** A cada chamada recursiva, a lista é dividida ao meio, resultando em uma árvore de recursão com $\log n$ níveis.
- **Intercalação:** A cada nível da recursão, todos os n elementos são processados para serem combinados (intercalados).
- Portanto, a complexidade do algoritmo é composta por duas partes:
 - 1.**Número de níveis de recursão:** $\log n$ (porque a lista é dividida repetidamente ao meio).
 - 2.**Intercalação dos elementos:** Cada nível da recursão requer $O(n)$ operações para intercalar os elementos.

Multiplicando essas duas partes, obtemos a complexidade total de $O(n \times \log n)$. Assim, a resposta correta é a alternativa E.

3. Exercícios... (ENADE)

Considere os seguintes algoritmos de ordenação e suas respectivas características:

- I. O **merge sort** é um algoritmo de ordenação estável com complexidade $O(n \log n)$ no pior caso.
- II. O **quick sort** tem complexidade $O(n^2)$ no pior caso, mas seu desempenho médio é $O(n \log n)$.
- III. O **selection sort** tem complexidade $O(n^2)$ no pior caso e não é um algoritmo estável.
- IV. O **insertion sort** tem complexidade $O(n^2)$ no pior caso, mas pode ser eficiente em listas quase ordenadas, com complexidade próxima de $O(n)$.

Estão corretos os itens:

- A) I e II
- B) II e III
- C) I, II e IV
- D) I, II, III e IV
- E) II e IV

3. Exercícios... (ENADE)

- **Item I: Verdadeiro.** O **merge sort** é um algoritmo de ordenação estável, o que significa que ele preserva a ordem relativa de elementos iguais. Sua complexidade é $O(n \log n)$ em todos os casos, seja no melhor, médio ou pior cenário.
- **Item II: Verdadeiro.** O **quick sort** tem um pior caso com complexidade $O(n^2)$ quando o pivô escolhido é o menor ou maior elemento da lista. No entanto, na maioria dos casos, o desempenho é $O(n \log n)$.
- **Item III: Verdadeiro.** O **selection sort** é $O(n^2)$ no pior caso, e não é um algoritmo estável, pois a ordem de elementos iguais pode ser alterada durante a execução.
- **Item IV: Verdadeiro.** O **insertion sort** tem complexidade $O(n^2)$ no pior caso, mas quando a lista está quase ordenada, pode ter um desempenho muito mais eficiente, aproximando-se de $O(n)$.
- **Assim, todos os itens estão corretos, o que leva à alternativa D).**

4. Exercícios... (ENADE)

Considere dois algoritmos de ordenação: **bubble sort** e **quick sort**. Considere que você tenha um vetor de 10.000 elementos, onde todos os elementos estão em ordem decrescente. Sobre a execução desses algoritmos nesse cenário, julgue os itens a seguir:

- I. O **bubble sort** terá desempenho $O(n^2)$, fazendo um grande número de comparações e trocas.
- II. O **quick sort** pode ter desempenho $O(n^2)$, caso o pivô seja escolhido de forma ineficiente, como o primeiro ou o último elemento da lista.
- III. O **bubble sort** é mais eficiente que o **quick sort** em listas completamente desordenadas.
- IV. O **quick sort** pode ser mais rápido que o **bubble sort**, se uma boa escolha de pivô for feita, mesmo em uma lista inicialmente em ordem decrescente.

Estão corretos os itens:

- A) I e II
- B) II e IV
- C) I, II e IV
- D) II e III
- E) I e III

4. Exercícios... (ENADE)

- **Item I: Verdadeiro.** O **bubble sort** tem uma complexidade $O(n^2)$, já que ele percorre a lista várias vezes, comparando e trocando elementos adjacentes. Em um cenário de ordem decrescente, ele fará o máximo de trocas e comparações possíveis, resultando em um desempenho muito ruim.
- **Item II: Verdadeiro.** O **quick sort** pode ter um pior desempenho de $O(n^2)$ se o pivô for escolhido de maneira desfavorável, como sendo sempre o primeiro ou o último elemento da lista. Nesse caso, a divisão das sublistas será muito desigual, resultando em um tempo de execução quadrático.
- **Item III: Falso.** O **bubble sort** não é mais eficiente que o **quick sort** em listas desordenadas. Na verdade, o **bubble sort** é um dos algoritmos de ordenação menos eficientes, enquanto o **quick sort** geralmente tem um desempenho médio de $O(n \log n)$, o que o torna superior na maioria dos casos.
- **Item IV: Verdadeiro.** O **quick sort** pode ser muito mais rápido que o **bubble sort**, especialmente se o pivô for escolhido de maneira eficiente (por exemplo, usando a mediana), o que divide a lista de forma equilibrada, mesmo em casos de listas ordenadas ou parcialmente ordenadas.

Portanto, os itens I, II, e IV estão corretos, o que leva à alternativa C).

Próxima Aula



- Ler o capítulo 10 do livro “Estrutura de Dados com Python”



Boa semana e bons estudos!!