Aula 3 - 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. Neste curso, iremos apresentar alguns deles: Bubblesort, Selectionsort, Insertionsort, Quicksort e Mergesort.

Bubblesort

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. A seguir veremos uma implementação em Python desse algoritmo.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Bubblesort
def BubbleSort(L):
    # Percorre cada elemento da lista L
    for i in range(len(L)-1, 0, -1):
        # Flutua o maior elemento para a posição mais a direita
        for j in range(i):
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
BubbleSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Exemplo: mostre o passo a passo necessário para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem (levar maior elemento para última posição)

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6] em amarelo, não troca
[2, 5, 13, 7, -3, 4, 15, 10, 1, 6] em azul, troca
[2, 5, 7, 13, -3, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 13, 4, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 15, 10, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 15, 1, 6]
[2, 5, 7, -3, 4, 13, 10, 1, 15, 6]

[5, 2, 7, -3, 4, 13, 10, 1, 6, 15]

2ª passagem (levar segundo maior para penúltima posição)

[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, 7, -3, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 7, 4, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 13, 10, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 13, 1, 6, 15]
[2, 5, -3, 4, 7, 10, 1, 13, 6, 15]

[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]

3ª passagem (levar terceiro maior para antepenúltima posição)
[2, 5, -3, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 5, 4, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 10, 1, 6, 13, 15]
[2, -3, 4, 5, 7, 1, 10, 6, 13, 15]

[2, -3, 4, 5, 7, 1, 6, 10, 13, 15]

4ª passagem

[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 7, 1, 6, 10, 13, 15]
[-3, 2, 4, 5, 1, 7, 6, 10, 13, 15]

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

5ª passagem

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]

[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 5, 1, 6, 7, 10, 13, 15]
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]

6ª passagem

[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]
[-3, 2, 4, 1, 5, 6, 7, 10, 13, 15]
[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]

7ª passagem

[-3, 2, 1, 4, 5, 6, 7, 10, 13, 15]
[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

8ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

9ª passagem

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Análise da complexidade do Bubblesort

Observando a função definida anteriormente, note que no pior caso o segundo loop vai de zero a i, sendo que na primeira vez i = n – 1, na segunda vez i = n – 2 e até i = 0. Sendo assim, o número de operações é dado por:

T(n) = (n + (n – 1) + (n - 2) + … + 1)

Já vimos na aula anterior que o somatório 1 + 2 + … + n é igual a n(n + 1)/2. Assim, temos:

$$T(n)=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$

o que nos leva a $O(n^2)$.

No melhor caso, é possível fazer uma pequena modificação no Bubblesort para contar quantas inversões (trocas) ele realiza. Dessa forma, se uma lista já está ordenada e o Bubblesort não realiza nenhuma troca, o algoritmo pode terminar após o primeiro passo. Com essa modificação, se o algoritmo encontra uma lista ordenada, sua complexidade é O(n), pois ele percorre a lista de n

elementos uma única vez. Porém, para fins didáticos, a versão apresentada acima tem complexidade $O(n^2)$ mesmo no melhor caso, pois não faz a checagem de quantas inversões são realizadas.

Finalmente, no caso médio, precisamos calcular a média dos custos entre todos os casos possíveis. O primeiro caso (de menor custo) ocorre quando o laço mais externo realiza uma única iteração e o último caso (de maior custo) ocorre quando o laço externo realiza o número máximo de iterações, isto é, n-1 iterações. Como todos os casos são igualmente prováveis, isto é, todas as diferentes configurações do vetor de entrada têm a mesma probabilidade de ocorrer (distribuição uniforme), então a probabilidade de cada caso é de 1/(n – 1) e a média é dada por:

$$\frac{1}{n-1}\sum_{k=1}^{n-1} f(k)$$

onde f(k) denota o número de execuções do loop mais externo para a k-ésima configuração. Lembrando que quando k = 1 o Bubblesort realiza uma única passagem na lista L, quando k = 2 o Bubblesort realiza duas passagens na lista L, e assim sucessivamente. Dessa forma, podemos definir a função f(k) de maneira a contar o número de trocas em cada configuração possível como:

$$f(k)=\sum_{i=1}^{k}\sum_{j=1}^{i} 1$$

uma vez que o loop mais interno realiza i trocas, sendo que esse número de trocas depende de qual configuração estamos, ou sjea, quanto maior for o k, mais vezes o loop mais interno será executado.

Portanto, podemos escrever:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\sum_{i=1}^{k}\sum_{j=1}^{i} 1\right)$$

Note que o último somatório resulta em i, o que nos permite escrever:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\sum_{i=1}^{k} i\right)$$

Sabemos que o somatório 1 + 2 + 3 + … + k = k(k+1)/2, o que nos leva a:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\frac{k(k+1)}{2}$$

Aplicando a distributiva, temos:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left(\frac{1}{2}(k^2+k)\right)$$

Como um somatórios de somas é igual as somas dos somatórios, podemos escrever:

$$T(n)=\frac{1}{2(n-1)}\left[\sum_{k=1}^{n-1} k^2+\sum_{k=1}^{n-1} k\right] \quad (*)$$

Vamos chamar o primeiro somatório de A e o segundo de B. O valor de B é facilmente calculado pois sabemos que 1 + 2 + 3 + … + n – 1 = n(n-1)/2. Vamos agora calcular o valor de A, dado por:

$$A=\sum_{k=1}^{n-1} k^2$$

Para isso, iremos utilizar o conceito de soma telescópica. Lembre-se que:

$$(k+1)^3=k^3+3k^2+3k+1$$

de modo que podemos escrever

$$(k+1)^3-k^3=3k^2+3k+1$$

Aplicando somatório de ambos os lados, temos:

$$\sum_{k=1}^{n-1}\left[(k+1)^3-k^3\right]=3\sum_{k=1}^{n-1}k^2+3\sum_{k=1}^{n-1}k+\sum_{k=1}^{n-1}1$$

Note que o somatório do lado esquerdo é uma soma telescópica, ou seja, vale $n^3 – 1$ (é a diferença entre o último elemento e o primeiro, uma vez todos os termos intermediários se cancelam:

$$n^3-1=3\sum_{k=1}^{n-1}k^2+3\frac{n(n-1)}{2}+n-1$$

Dessa forma, isolando o somatório desejado, temos:

$$\sum_{k=1}^{n-1}k^2=\frac{n^3-1-n+1-\frac{3n(n-1)}{2}}{3}=\frac{n^3-n}{3}-\frac{n(n-1)}{2}=\frac{n(n^2-1)}{3}-\frac{n(n-1)}{2}=\frac{n(n-1)(n+1)}{3}-\frac{n(n-1)}{2}$$

Voltando para a equação (*), podemos escrever:

$$T(n)=\frac{1}{2(n-1)}\left[\frac{n(n-1)(n+1)}{3}-\frac{n(n-1)}{2}+\frac{n(n-1)}{2}\right]$$

Note que os dois últimos termos da equação se cancelam pois são idênticos, o que nos leva a:

$$T(n)=\frac{1}{2(n-1)}\left(\frac{n(n-1)(n+1)}{3}\right)$$

Cancelando os termos n – 1, finalmente chegamos em:

$$T(n)=\frac{1}{6}n(n+1)=\frac{1}{6}(n^2+n)$$

o que mostra que a complexidade do caso médio é $O(n^2)$.

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.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Selectionsort
def SelectionSort(L):
    # Percorre todos os elementos de L
    for i in range(len(L)):
        menor = i
        # Encontra o menor elemento
        for k in range(i+1, len(L)):
            if L[k] < L[menor]:
                menor = k
        # Troca a posição do elemento i com o menor
        L[menor], L[i] = L[i], L[menor]

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
SelectionSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Análise da complexidade do Selectionsort

Observando a função definida anteriormente, note que no pior caso o segundo loop vai de i +1 até n, sendo que na primeira vez i = 1, na segunda vez i = 2 e até i = n - 1. Sendo assim, o número de operações é dado por:

$$T(n)=\sum_{i=0}^{n-1}\left(2+\sum_{j=i+1}^{n-1}1\right)=\sum_{i=0}^{n-1}2+\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}1$$

Note que:

$$\sum_{i=2}^{5} 1 = (5-2)+1 = 4$$

Então, podemos escrever:

$$T(n) = 2n + \sum_{i=0}^{n-1} (n-1-(i+1)+1) = 2n + \sum_{i=0}^{n-1} (n-i-1)$$

Podemos decompor o somatório em três, de modo que:

$$T(n) = 2n + \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 = 2n + n^2 - \frac{n(n-1)}{2} - n = n^2 + n - \frac{n^2-n}{2} = \frac{1}{2}n^2 + \frac{3}{2}n$$

o que resulta em $O(n^2)$. Uma desvantagem do algoritmo Selectionsort é que mesmo no melhor caso, para encontrar o menor elemento, devemos percorrer todo o restante do vetor no loop mais interno. Isso significa que, mesmo no melhor caso, a complexidade é $O(n^2)$ Isso implica que no caso médio, a complexidade também é $O(n^2)$.

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6]
2ª passagem: [-3, 2, 13, 7, 5, 4, 15, 10, 1, 6] → [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6]
3ª passagem: [-3, 1, 13, 7, 5, 4, 15, 10, 2, 6] → [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6]
4ª passagem: [-3, 1, 2, 7, 5, 4, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]
5ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6]
6ª passagem: [-3, 1, 2, 4, 5, 7, 15, 10, 13, 6] → [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7]
7ª passagem: [-3, 1, 2, 4, 5, 6, 15, 10, 13, 7] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]
8ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]
9ª passagem: [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

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. 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. 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.

```
import numpy as np
import time

# Implementação em Python do algoritmo de ordenação Insertionsort
def InsertionSort(L):
    # Percorre cada elemento de L
    for i in range(1, len(L)):
        k = i
        # Insere o pivô na posição correta
        while k > 0 and L[k] < L[k-1]:
            L[k], L[k-1] = L[k-1], L[k]
            k = k - 1

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
InsertionSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

Análise da complexidade do Insertionsort

Observando a função definida anteriormente, note que no pior caso a posição correta do pivô será sempre em k = 0 de modo que o segundo loop vai ter de percorrer todo vetor (de i até 0), sendo que na primeira vez i = 1, na segunda vez i = 2 e até i = n - 1. Sendo assim, o número de operações é dado por:

$$T(n)=\sum_{i=1}^{n-1}\left(1+\sum_{j=0}^{i}2\right)$$

Expandindo os somatórios, temos:

$$T(n)=\sum_{i=1}^{n-1}1+2\sum_{i=1}^{n-1}(i+1)=n-1+2\sum_{i=1}^{n-1}i+2\sum_{i=1}^{n-1}1$$

O valor do primeiro somatório é n(n-1)/2 e o valor do segundo somatório é n – 1, o que nos leva a:

$$T(n)=n-1+2\frac{n(n-1)}{2}+2(n-1)=3(n-1)+n^2-n=n^2+2n-3$$

o que mostra que a complexidade é $O(n^2)$.

Para o melhor caso, note que o pivô sempre está na posição correta, sendo que o loop mais interno não será executado nenhuma vez. Assim temos que dentro do loop mais externo apenas uma instrução será executada, o que nos leva a:

$$T(n)=\sum_{i=1}^{n-1}1=n-1$$

mostrando que a complexidade é linear, ou seja, $O(n)$. Para o caso médio, podemos aplicar uma estratégia muito similar àquela adotada na análise do Bubblesort. A ideia consiste em considerar que todos os casos são igualmente prováveis e calcular uma média de todos eles. Assim, temos:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}\left[\sum_{i=1}^{k}\left(1+\sum_{j=0}^{i}2\right)\right]$$

Do pior caso, sabemos que a soma entre colchetes pode ser simplificada, o que nos leva a:

$$T(n)=\frac{1}{n-1}\sum_{k=1}^{n-1}(k^2+2k-3)$$

Deixaremos o restante dos cálculos para o leitor (basta seguir os mesmos passos algébricos da análise do caso médio do Bubblesort). É possível verificar então que a complexidade é $O(n^2)$.

Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1ª passagem: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]
2ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 13, 7, -3, 4, 15, 10, 1, 6]
3ª passagem: [2, 5, 13, 7, -3, 4, 15, 10, 1, 6] → [2, 5, 7, 13, -3, 4, 15, 10, 1, 6]
4ª passagem: [2, 5, 7, 13, -3, 4, 15, 10, 1, 6] → [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6]
5ª passagem: [-3, 2, 5, 7, 13, 4, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]
6ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6]
7ª passagem: [-3, 2, 4, 5, 7, 13, 15, 10, 1, 6] → [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6]
8ª passagem: [-3, 2, 4, 5, 7, 10, 13, 15, 1, 6] → [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6]
9ª passagem: [-3, 1, 2, 4, 5, 7, 10, 13, 15, 6] → [-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Fim: garantia de que vetor está ordenado só é obtida após todos os passos.

Recursão

Dizemos que uma função é recursiva se ela é definida em termos dela mesma. Em matemática e computação uma classe de objetos ou métodos exibe um comportamento recursivo quando pode ser definido por duas propriedades:

1. Um caso base: condição de término da recursão em que o processo produz uma resposta.

2. Um passo recursivo: um conjunto de regras que reduz todos os outros casos ao caso base.

A série de Fibonacci é um exemplo clássico de recursão, pois:

F(1) = 1 (caso base 1)
F(2) = 1 (caso base 2)

Para todo n > 1, F(n) = F(n - 1) + F(n – 2)

A função recursiva a seguir calcula o n-ésimo termo da sequência de Fibonacci.

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

n = int(input('Enter com o valor de n: '))
resultado = fib(n)

print('Fibonacci(%d) = %d' %(n, resultado))
```

Os dois próximos algoritmos de ordenação que iremos estudar são exemplos de abordagens recursicas, onde a cada passo, temos um subproblema menor para ser resolvido pela mesma função. Por isso, a função é definida em termos de si própria.

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 (existem várias formas de se fazer essa etapa)

<= c	<= c	<= c	<= c	c	>= c	>=c	>= c	>= c	>=c

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.
Exemplo: mostre os passos necessários para a ordenação do seguinte vetor

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

1º passo: Definir pivô = 6 (último elemento)

2º passo: Particionar vetor (menores a esquerda e maiores a direita)

[5, 2, -3, 4, 1, 6, 13, 7, 15, 10]

3º passo: Aplicar 1 e 2 recursivamente para as metades

a) 2 metades

Metade 1: [5, 2, -3, 4, 1] → pivô = 1

[-3, 1, 5, 2, 4, 6, 13, 7, 15, 10]

Metade 2: [13, 7, 15, 10] → pivô = 10

[-3, 1, 5, 2, 4, 6, 7, 10, 15, 13]

b) 4 metades

Note que a metade 1 possui um único elemento: [-3] → já está ordenada

Metade 2: [5, 2, 4] → pivô = 4

[-3, 1, 2, 4, 5, 6, 7, 10, 15, 13]

Note que a metade 3 possui apenas um único elemento: [7] → já está ordenadas

Metade 4: [15, 13] → pivô = 13

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

c) 4 metades: Note que cada uma das 4 metades restantes contém um único elemento e portanto já estão ordenadas. Fim.

A seguir veremos uma implementação recursiva em Python para o algoritmo *Quicksort*.

```
# Implementação em Python do algoritmo de ordenação Quicksort
def QuickSort(L):
    if len(L) <= 1:
        return L
    # Pivô é o primeiro elemento da lista pode ser último ou do meio)
    m = L[0]
    # Chamada recursiva
    return  QuickSort([x for x in L if x < m]) + \
                      [x for x in L if x == m] + \
            QuickSort([x for x in L if x > m])
```

No Quicksort há significativas diferenças entre o caso melhor e o pior caso. Veremos primeiramente o pior caso.

a) Pior caso: acontece quando o pivô é sempre o maior ou menor elemento, o que gera partições totalmente desbalanceadas:

Na primeira chamada recursiva, temos uma lista de tamanho n, então para criar as novas listas L, teremos n – 1 elementos na primeira lista, o pivô e uma lista vazia.

Isso nos permite escrever a seguinte relação de recorrência:

$$T(n)=T(n-1)+T(0)+n=T(n-1)+n$$

pois estamos decompondo um problema de tamanho n em um de tamanho zero e outro de tamanho n – 1, mas para realizar a divisão da lista L em sublistas, utilizamos n operações (uma vez que a lista L tem n elementos)

Note que:

T(n) = n + T(n – 1)
T(n - 1) = n - 1 + T(n – 2)
T(n - 2) = n - 2 + T(n – 3)
T(n - 3) = n - 3 + T(n – 4)
…
T(2) = 2 + T(1)
T(1) = 1 + T(0)

onde T(0) = 0.

Somando todas as parcelas, temos o somatório (1 + 2 + 3 + … + n – 1) + n, cuja resposta é:

$$T(n)=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$

o que nos leva a $O(n^2)$.

b) Melhor caso: acontece quando as partições tem exatamente o mesmo tamanho, ou seja, são n/2 elementos, o pivô e mais n/2 - 1 elementos.

Podemos escrever a seguinte relação de recorrência:

$$T(n)=T\left(\frac{n}{2}\right)+T\left(\frac{n}{2}\right)+n=2T\left(\frac{n}{2}\right)+n$$

pois estamos decompondo um problema de tamanho n em dois problemas de tamanho n/2, mas para realizar a divisão da lista L em sublistas, utilizamos n operações (uma vez que a lista L tem n elementos). Expandindo a recorrência temos:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$
$$T\left(\frac{n}{2}\right)=2T\left(\frac{n}{4}\right)+n$$
$$T\left(\frac{n}{4}\right)=2T\left(\frac{n}{8}\right)+n$$
$$T\left(\frac{n}{8}\right)=2T\left(\frac{n}{16}\right)+n$$

…

Voltando com as substituições, temos:

$$T(n)=2\left[2\left[2\left[2T\left(\frac{n}{16}\right)+n\right]+n\right]+n\right]+n=2^4T\left(\frac{n}{2^4}\right)+4n$$

Generalizando para um valor k arbitrário, podemos escrever:

$$T(n)=2^kT\left(\frac{n}{2^k}\right)+kn$$

Para que tenhamos T(1), é preciso que $n = 2^k$, ou seja, $k=\log_2 n$. Quando $k=\log_2 n$, temos:

$$T(n)=2^{\log_2 n}T(1)+n\log_2 n=nT(1)+n\log_2 n$$

Como T(1) é constante (pois não depende de n), temos finalmente que a complexidade do Quicksort no melhor caso é $O(n\log_2 n)$.

Veremos a seguir que o fato de que o pior caso é $O(n^2)$, não é um problema, pois o caso médio está muito mais próximo do melhor caso do que do pior caso. Para considerar o caso médio, suponha que alternemos as recorrências de melhor caso, M(n), com pior caso, P(n). Então, iniciando com o melhor caso:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$

Expandindo agora o pior caso, temos:

$$T(n)=2\left[T\left(\frac{n}{2}-1\right)+\frac{n}{2}\right]+n$$

Simplificando a expressão acima, podemos escrever:

$$T(n)=2T\left(\frac{n}{2}-1\right)+O(n)$$

que é uma relação de recorrência da mesma forma que a obtida para o melhor caso. A solução da recorrência implica que a complexidade do caso médio é $O(n\log_2 n)$.

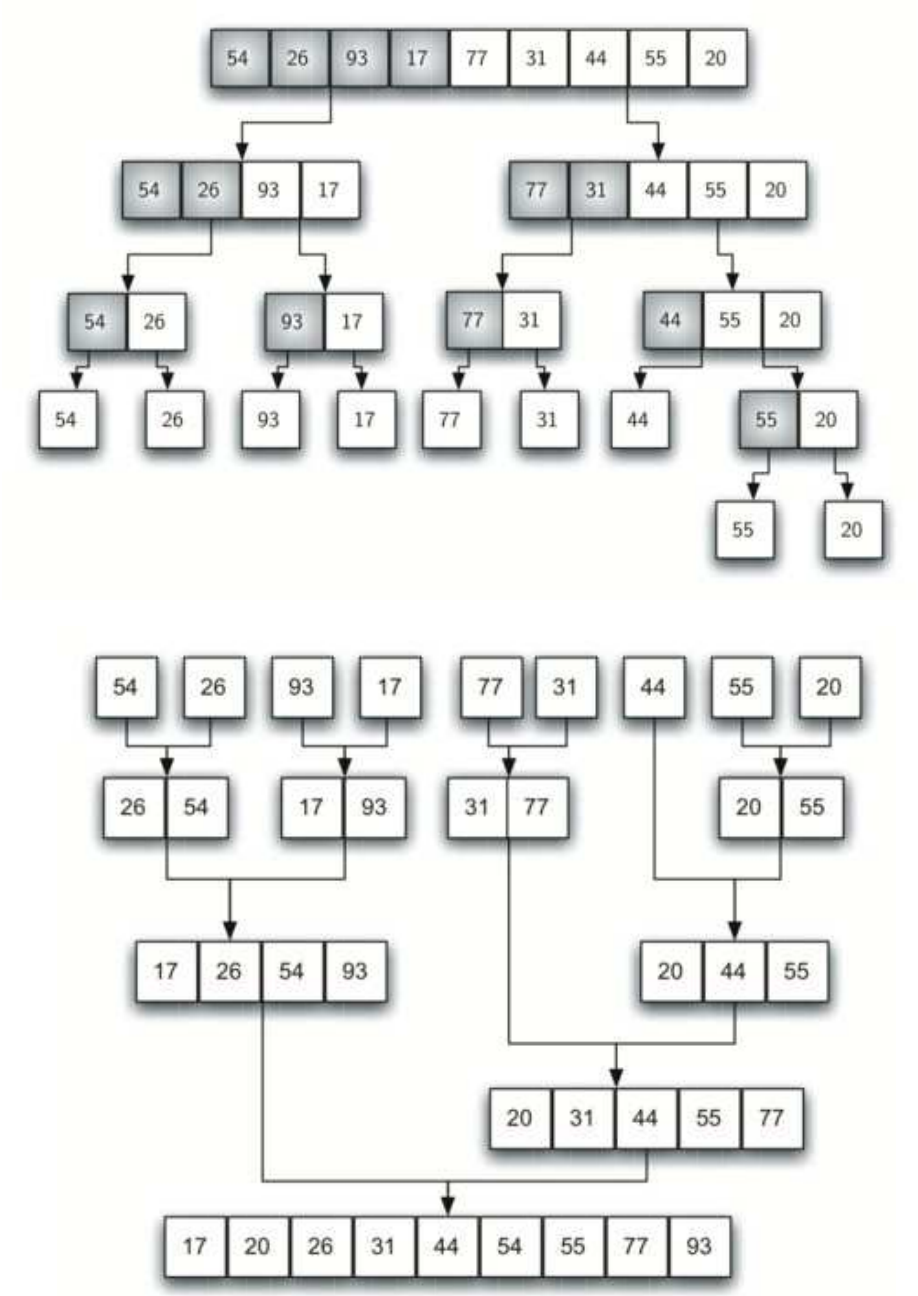
Uma estratégia empírica que, em geral, melhora o desempenho do algoritmo Quicksort consiste em escolher como pivô a mediana entre o primeiro elemento, o elemento do meio e o último elemento.

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.

A figura a seguir ilustra as duas fases principais do algoritmo Mergesort: a divisão e a intercalação.



A seguir apresentamos um código em Python para implementar o algoritmo MergeSort.

```
# Implementação em Python do algoritmo de ordenação MergeSort
def MergeSort(L):

    if len(L) > 1:

        meio = len(L)//2
        LE = L[:meio]   # Lista Esquerda
        LD = L[meio:]   # Lista Direita

        # Aplica recursivamente nas sublistas
        MergeSort(LE)
        MergeSort(LD)

        # Quando volta da recursão inicia aqui!
        i, j, k = 0, 0, 0
        # Faz a intercalação das duas listas (merge)
        while i < len(LE) and j < len(LD):
            if LE[i] < LD[j]:
                L[k] = LE[i]
                i += 1
            else:
                L[k] = LD[j]
                j += 1
            k += 1

        while i < len(LE):
            L[k] = LE[i]
            i += 1
            k += 1

        while j < len(LD):
            L[k] = LD[j]
            j += 1
            k += 1

# Início do script
n = 5000
X = list(np.random.random(n))

# Imprime vetor
print('Vetor não ordenado: ')
print(X)
print()

# Aplica Bubblesort
inicio = time.time()
MergeSort(X)
fim = time.time()

# Imprime vetor ordenado
print('Vetor ordenado: ')
print(X)
print()
print('Tempo de processamento: %.3f s' %(fim - inicio))
```

A função MergeSort mostrada acima começa perguntando pelo caso base. Se o tamanho da lista form 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, LE, e direita, LD), elas já estão ordenadas. O resto da função (linhas 11-31) 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 (`L`) ao tomar repetidamente o menor item das listas ordenadas.

A seguir apresentamos um exemplo ilustrativo passo a passo da aplicação do MergeSort.

Ex: Mostre o passo a passo da ordenação do vetor a seguir pelo algoritmo MergeSort

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

Passo 1: Dividir em subproblemas

1ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 10 // 2 = 5

2ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 5 // 2 = 2

3ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 2 // 2 = 1 ou meio = 3 // 2 = 1

4ª divisão: [5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

meio = 2 // 2 = 1

Parte 2: Intercalar listas (Merge) – as últimas a serem divididas serão as primeiras a fazer o merge

[5, 2, 13, 7, -3, 4, 15, 10, 1, 6]

[5, 2, 13, -3, 7, 4, 15, 10, 1, 6]

[2, 5, -3, 7, 13, 4, 15, 1, 6, 10]

[-3, 2, 5, 7, 13, 1, 4, 6, 10, 15]

[-3, 1, 2, 4, 5, 6, 7, 10, 13, 15]

Análise da complexidade do MergeSort

Os três passos úteis dos algoritmos de dividir para conquistar, ou *divide and conquer*, que se aplicam ao MergeSort são:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante O(1);
2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho n/2, o que contribui com T(n/2) + T(n/2) para o tempo de execução;
3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo O(n);

Assim, podemos escrever a relação de recorrência como:

$$T(n)=\begin{cases} O(1), & se \quad n=1 \\ 2T\left(\frac{n}{2}\right)+O(n), & se \quad n>1 \end{cases}$$

Trata-se da mesma recorrência resolvida no algoritmo Quicksort. Note que expandindo a recorrência, temos:

$$T(n)=2T\left(\frac{n}{2}\right)+n$$
$$T\left(\frac{n}{2}\right)=2T\left(\frac{n}{4}\right)+n$$
$$T\left(\frac{n}{4}\right)=2T\left(\frac{n}{8}\right)+n$$
$$T\left(\frac{n}{8}\right)=2T\left(\frac{n}{16}\right)+n$$

…

Voltando com as substituições, podemos escrever:

$$T(n)=2\left[2\left[2\left[2T\left(\frac{n}{16}\right)+n\right]+n\right]+n\right]+n=2^4T\left(\frac{n}{2^4}\right)+4n$$

Generalizando para um valor k arbitrário, podemos escrever:

$$T(n)=2^kT\left(\frac{n}{2^k}\right)+kn$$

Para que tenhamos T(1), é preciso que $n = 2^k$, ou seja, $k=\log_2 n$. Quando $k=\log_2 n$, temos:

$$T(n)=2^{\log_2 n}T(1)+n\log_2 n=nT(1)+n\log_2 n$$

Como T(1) é O(1), temos que a complexidade do Quicksort no melhor caso é $O(n\log_2 n)$.

Uma das maiores limitações do algoritmo MergeSort é que esse método passa por todo o longo processo mesmo se a lista L já estiver ordenada. Por essa razão, a complexidade de melhor caso é idêntica a complexidade de pior caso, ou seja, $O(n\log n)$.

Para o caso de n muito grande, e listas compostas por números gerados aleatoriamente, pode-se mostrar que o número médio de comparações realizadas pelo algoritmo MergeSort é aproximadamente αn menor que o número de comparações no pior caso, onde:

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$$

Em resumo, a tabela a seguir faz uma comparação das complexidades dos cinco algoritmos de ordenação apresentados aqui, no pior caso, caso médio e melhor caso.

Algoritmo	**Tempo**		
	Melhor	**Médio**	**Pior**
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Como pode ser visto, fica claro que os dois melhores algoritmos são O QuickSort e o MergeSort.

Existem vários outros algoritmos de ordenação que não apresentaremos aqui. Dentre eles, podemos citar o HeapSort, o ShellSort e RadixSort. Para os interessados em aprender mais sobre o assunto, a internet contém diversos materiais sobre algoritmos de ordenação.

Há várias animações que demonstram o funcionamento dos algoritmos de ordenação. A seguir indicamos alguns links interessantes:

15 sorting algorithms in 6 minutes (Animações sonorizadas muito boas para visualização)
https://www.youtube.com/watch?v=kPRA0W1kECg

Animações passo a passo dos algoritmos
https://visualgo.net/en/sorting

Comparação em tempo real dos algoritmos
https://www.toptal.com/developers/sorting-algorithms

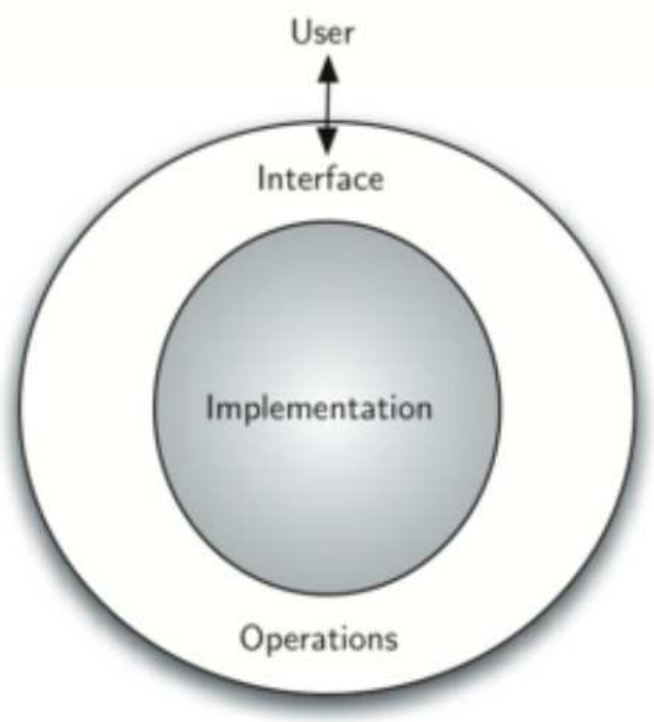
Algoritmos de ordenação como danças
https://www.youtube.com/user/AlgoRythmics

Aula 4 - Programação orientada a objetos em Python

Na programação de computadores, o conceito de abstração é fundamental para o desenvolvimento de software de alto nível. Um exemplo são as funções, que são abstrações de processos. Uma vez que uma função é criada, toda lógica é encapsulada do usuário. Sempre que o programador necessitar, basta invocar a função, sem que ele precise conhecer os detalhes da implementação. Conforme a complexidade dos programas aumenta, torna-se necessário definir abstrações para dados. É para essa finalidade que foram criados os Tipos Abstratos de Dados (TAD's), assunto que iremos explorar em detalhes a seguir.

Tipos Abstratos de Dados (TAD's)

Um Tipo Abstrato de Dados, ou TAD, é um tipo de dados definido pelo programador que especifica um conjunto de variáveis que são utilizadas para armazenar informação e um conjunto de operações bem definidas sobre essas variáveis. TAD's são definidos de forma a ocultar a sua implementação, de modo que um programador deve interagir com as variáveis internas a partir de uma interface, definida em termos do conjunto de operações. A Figura a seguir ilustra essa ideia.



Um exemplo de TAD implementado nativamente pela linguagem Python são as listas. Uma lista em Python consiste basicamente de uma variável composta heterogênea (pode armazenar informações de tipos de dados distintos) utilizada para armazenar as informações mais um conjunto de operações para manipular essa variável.