1 Busca sequencial

Algoritmos de busca verificam se um certo elemento ocorre ou não em uma determinada sequência. Em Python, podemos simplesmente utilizar o comando **in** para verificar se um elemento pertence a uma sequência:

```python
l = [3,16,5,8,9,2,1,15,6]
x = int(input("Digite um numero: "))

if x in l:
    print("Encontrei o elemento!")
else:
    print("Ops... ta aqui nao!")
```

Entretanto, isso só pode ser utilizado para elementos simples. Imagine que a sequência contém tuplas indicando o nome e cpf dos usuários:

```python
l = [ ("Joao", "123.456.789-00"), ("Jose", "234.567.891-00"),
      ("Maria", "345.678.912-00"), ("Ana", "456.789.123-00") ]
```

Nesse caso, seria incorreto utilizar `if cpf in l` para buscar um cpf na lista. Por isso, precisamos entender como funciona a busca por um elemento, para podermos criar funções específicas para as estruturas de dados que formos utilizar.

A forma mais fácil de procurar por um elemento é percorrer toda a sequência, comparando o elemento com cada um de seus itens. No código abaixo, ao invés de retornar verdadeiro ou falso, retornamos a posição da sequência em que o elemento ocorre (ou -1, caso não ocorra):

```python
def busca1(x, l):
    # Nesta funcao, l eh uma lista de tuplas, e o elemento
    # eh procurado sempre na segunda posicao de cada tupla.
    for i in range(len(l)):
        if l[i][0] == x:
            return i

    return -1
```

A busca também pode ser feita de forma recursiva:

```python
def busca2(x, l, pos=0):
    if pos == len(l):
        return -1
    else:
        if x==l[pos]:    # if x==l[pos][1] se forem tuplas
            return pos
        else:
```

```
            return busca2(x, l, pos+1)
```

No melhor caso, a função irá comparar apenas um elemento (caso já encontre o elemento na primeira tentativa). No pior caso, para uma sequência de n elementos, a função irá varrer toda a sequência e comparar x com todos os n elementos.

2 Busca binária

Um algoritmo de busca mais eficiente assume que os elementos da sequência já foram pré-ordenados por algum motivo. Neste caso, ao invés de testarmos os elementos sequencialmente, podemos fazer o seguinte:

- Considere o elemento M no meio da lista.
- Caso x seja igual a M, encontramos o elemento e a busca terminou.
- Caso x for menor que M, então x deve ser procurado apenas na primeira metade da lista, já que ela está ordenada.
- Caso x for maior que M, então x deve ser procurado apenas na segunda metade da lista, já que ela está ordenada.

No exemplo abaixo, queremos encontrar o elemento 57. No primeiro passo, temos:

```
 0   1   2   3   4   5   6   7   8   9   10  11  12
 --------------------------------------------------
 12  13  15  19  24  28  39  57  59  63  67  69  74
 |                       |                       |
início                  meio                    fim
```

No segundo passo, temos:

```
 0   1   2   3   4   5   6   7   8   9   10  11  12
 --------------------------------------------------
 12  13  15  19  24  28  39  57  59  63  67  69  74
                             |       |           |
                          início    meio        fim
```

Em seguida:

```
 0    1    2    3    4    5    6    7    8    9    10   11   12
---------------------------------------------------------------
12   13   15   19   24   28   39   57   59   63   67   69   74
                                   |    |
                                início  fim
                                 meio
```

Por fim, no último passo, o valor na posição “meio” é igual a 57 e, portanto, encontramos o resultado.

Este algoritmo é conhecido como **busca binária** pois metade da sequência é eliminada da busca a cada iteração. Se tivermos 1024 elementos e fizermos uma busca sequencial, pode ser que todos os 1024 elementos sejam testados antes do algoritmo terminar. No caso da busca binária, o primeiro teste elimina 512 elementos, o segundo outros 256, o terceiro elimina outros 128, e depois 64, 32, 16, 8, 4, 2, até que a lista contenha apenas 1 elemento. Dessa forma, ao invés de 1024, apenas 10 elementos (ou $log_2(len(l))$) precisam ser testados.

```python
def buscaBin(x, l):
    esquerda, direita = 0, len(l)-1
    while esquerda <= direita:
        meio = (esquerda + direita) // 2
        if l[meio] == x:
            return meio
        elif l[meio] > x:
            direita = meio - 1
        else:
            esquerda = meio + 1
    return -1

def buscaBinRec(x, l, esquerda, direita):
    if direita < esquerda: return -1

    meio = (esquerda + direita) // 2
    if l[meio] == x: return meio
    if l[meio] > x: return buscaBinRec(x, l, esquerda, meio-1)
    else: return buscaBinRec(x, l, meio+1, direita)
```

3 Noções de complexidade

A eficiência de um algoritmo costuma ser medida de acordo com seu tempo de execução, que varia de acordo com a entrada. Em geral, o tempo de execução aumenta quando o tamanho da entrada aumenta. O caso médio costuma ser difícil de encontrar, então normalmente tentamos saber o pior caso que o algoritmo pode demorar. Podemos analisar o tempo de execução de duas formas:

1. Análise empírica: Executamos o programa para entradas com tamanhos e valores variados, e calculamos a média de tempo. Nem sempre é conclusiva.

2. Análise teórica: Calcula o tempo máximo de execução em função do número de chamadas a cada instrução do código. Permite avaliar a performance de forma independente do hardware/software.

Na análise teórica, fazemos estimativas de quantas vezes cada instrução será executada. Com isto, temos uma noção de quanto da complexidade em função do tamanho da entrada. Por exemplo, no algoritmo a seguir:

```python
i = 0
while i < n:
    print(i)
    i += 1
```

poderíamos estimar que o tempo gasto é:

```
T(n) = (tempo gasto para fazer uma atribuição) +
       N * (tempo gasto por uma comparação entre i e n) +
       N * (tempo gasto para imprimir um número inteiro) +
       N * (tempo gasto para incrementar i)
```

No caso acima, o tempo da atribuição é irrelevante na medida em que o tamanho da entrada aumenta. O que importa, é que há 3 operações que são executadas n vezes. Quando analisamos o tempo de execução de um algoritmo e encontramos algo como

$$T(n) = 10n^2 + 150n + 30,$$

nesse caso o termo quadrático $10n^2$ é mais importante que os outros, pois é ele que irá dominar o crescimento do tempo de execução na medida em que aumentarmos n. Para fins de estimativa, podemos simplificá-la para $T(n) = 10n^2$. Outro fator menos importante é a constante que multiplica a variável. Independente da fórmula ser $T(n) = 10n^2$ ou $T(n) = 1000n^2$, dobrar o tamanho do n significa que o tempo será quadruplicado. Dizemos que o código acima tem uma taxa de crescimento proporcional a n^2, tem complexidade igual a n^2, ou simplesmente que ele é $O(n^2)$.

Se analisarmos o número de vezes em que as instruções na busca sequencial são executadas, vemos que o algoritmo é $O(n)$. Por isso, também chamamos a busca sequencial de busca linear, já que o tempo de execução cresce linearmente em função do aumento do número de elementos. Já a busca binária é $O(log_2 n)$, ou apenas $O(lg\ n)$. Por exemplo, se $n = 2^{30} \approx 1.000.000.00$, então a busca sequencial irá executar pelo menos alguma instrução em torno de 1 bilhão de vezes, enquanto a busca binária irá executar cada instrução no máximo 30 vezes.

4 Bubble Sort

Existem diversos algoritmos para colocar os elementos de uma lista em ordem. Um dos mais simples e mais implementados é o **Bubble Sort**. Nele, trocamos todos os elementos adjacentes que estejam fora de ordem:

```
Primeira passada pelos elementos:
  32   13   25   19   24   18
   \   /
  trocar!
  ---------------------------
  13   32   25   19   24   18
        \   /
       trocar!
  ---------------------------
  13   25   32   19   24   18
             \   /
            trocar!
  ---------------------------
  13   25   19   32   24   18
                  \   /
                 trocar!
  ---------------------------
  13   25   19   24   32   18
                       \   /
                      trocar!
  ---------------------------
  13   25   19   24   18 | 32
```

Ao final do primeiro passo, temos certeza que o maior elemento da lista está no final da lista. Portanto, temos que ordenar os $n-1$ elementos que restaram no início da lista:

```
Segunda passada pelos elementos:
  13   25   19   24   18 | 32
   \   /
não trocar!
  ---------------------------
```

```
  13   25   19   24   18 | 32
        \   /
       trocar!
  ---------------------------
  13   19   25   24   18 | 32
             \   /
            trocar!
  ---------------------------
  13   19   24   25   18 | 32
                  \   /
                 trocar!
  ---------------------------
  13   19   24   18 | 25   32
```

Agora, o penúltimo elemento é, garantidamente, o segundo maior elemento da lista.

```
Terceira passada pelos elementos:
  13   19   24   18 | 25   32
   \   /
não trocar!
  ---------------------------
  13   19   24   18 | 25   32
        \   /
     não trocar!
  ---------------------------
  13   19   24   18 | 25   32
             \   /
            trocar!
  ---------------------------
  13   19   18 | 24   25   32
```

```
Quarta passada pelos elementos:
  13   19   18 | 24   25   32
   \   /
não trocar!
  ---------------------------
  13   19   18 | 24   25   32
        \   /
       trocar!
  ---------------------------
  13   18 | 19   24   25   32
```

```
Quinta e última passada pelos elementos:
  13   18 | 19   24   25   32
   \   /
não trocar!
  ---------------------------
  13 | 18   19   24   25   32
```

Em Python:

```python
def bubbleSort(l):
    n = len(l)

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

Ao invés disso, podemos parar a execução do algoritmo quando percebermos que ele já está ordenado:

```python
def bubbleSort2(l):
    n = len(l)

    for i in range(n):
        trocou = False
        for j in range(0, n-i-1):
            if l[j] > l[j+1] :
                l[j], l[j+1] = l[j+1], l[j]
                trocou = True

        if not trocou: return
```

A complexidade do Bubble Sort é $O(n^2)$

5 Selection Sort

Para colocar os elementos de uma lista em ordem crescente, o método **Selection Sort** vai selecionando o menor elemento da lista e inserindo-o na frente dos demais, repetidamente. O algoritmo mantém a lista dividida em duas partes ao longo de sua execução:

- Os elementos já ordenados, no início da lista.

- O restante da lista a ser ordenada.

Exemplo:

```
Encontrando o menor elemento:
  32  13  25  19  24  18
 |     ^                |
--------------------------
  13  32  25  19  24  18
```

```
Encontrando o segundo menor elemento na sublista restante:
  13  32  25  19  24  18
     |                ^ |
--------------------------
  13  18  25  19  24  32
```

```
Encontrando o segundo menor elemento na sublista restante:
  13  18  25  19  24  32
         |     ^        |
--------------------------
  13  18  19  25  24  32
```

```
Encontrando o segundo menor elemento na sublista restante:
  13  18  19  25  24  32
             |     ^    |
--------------------------
  13  18  19  24  25  32
```

```
Encontrando o segundo menor elemento na sublista restante:
  13  18  19  24  25  32
                 | ^    |
--------------------------
  13  18  19  24  25  32
```

Em Python:

```python
def selectionSort(l):
    for i in range(len(l)):
        posMenor = i
        for j in range(i+1, len(l)):
            if l[posMenor] > l[j]:
                posMenor = j

        l[i], l[posMenor] = l[posMenor], l[i]
```

Uma vantagem deste método é que ele **nunca** troca dois elementos de posição mais do que $O(n)$ vezes.

6 Insertion Sort

O método **Insertion Sort** é um método simples de ordenação, que simula a forma como ordenamos as cartas de um baralho:

1. Inicialmente, escolho um elemento qualquer e insiro numa nova sequência, que por enquanto está ordenada.
2. Em seguida, escolho um outro elemento qualquer dentre os que faltam, e insiro-o ordenadamente na lista que contém os elementos já ordenados.
3. Caso ainda haja elementos a serem inseridos, volte ao passo 2.

Exemplo:

```
| ordenados | não ordenados:

| 32 | 13  25  19  24  18
| 13  32 | 25  19  24  18
| 13  18  32 | 25  19  24
| 13  18  19  32 | 25  24
| 13  18  19  24  32 | 25
| 13  18  19  24  25  32 |
```

Em Python:

```python
def insertionSort(l):
    for k in range(1, len(l)):
        elem = l[k]
        pos = k-1

        while pos >=0 and l[pos] > elem:
                l[pos+1] = l[pos]
                pos = pos - 1

        l[pos+1] = elem
```

7 Merge Sort

O método de ordenação **Merge Sort** é um algoritmo recursivo mais eficiente que o Bubble, Selection e Insertion Sort. A complexidade dele é $\Theta(n \cdot lg\ n)$, enquanto os outros três são $O(n^2)$. Quando a complexidade de um algoritmo é $\Theta(f(n))$, significa que o tempo é *sempre* proporcional a $f(n)$ (no melhor caso, no pior caso e na média).

O algoritmo vai dividindo a sequência ao meio, até que haja apenas 1 elemento. Depois, vai mesclando as subsequências de forma que se mantenham ordenadas.

```
[32,  13,  25,  19,  24,  18,  12,  20]
-----------------------------------------> dividindo ao meio:
[32,  13,  25,  19] [24,  18,  12,  20]
-----------------------------------------> dividindo ao meio:
[32,  13] [25,  19] [24,  18] [12,  20]
-----------------------------------------> dividindo ao meio:
[32] [13] [25] [19] [24] [18] [12] [20]
  \   /     \   /     \   /     \   /
-----------------------------------------> mesclando duas a duas:
[13,  32] [19,  25] [18,  24] [12,  20]
       \   /               \   /
-----------------------------------------> mesclando duas a duas:
[13,  19,  25,  32] [12,  18,  20,  24]
                 \   /
-----------------------------------------> mesclando duas a duas:
[12,  13,  18,  19,  20,  24,  25,  32]
```

Resumindo, os passos acima, a ideia do algoritmo é a seguinte:

1. Se houver no máximo um elemento na sequência, ela está ordenada.
2. Se não, encontre o meio da sequência e divida-a em duas metades.
3. Recursivamente, ordene a metade da esquerda usando o próprio mergesort.
4. Recursivamente, ordene a metade da direita usando o próprio mergesort.
5. Junte as duas metades, de forma que o resultado se matenha ordenado.

O algoritmo em Python é:

```python
def mergeSort(l):
    if len(l) > 1:
        meio = len(l) // 2
        lEsq = l[:meio]
```

```
        lDir = l[meio:]

        mergeSort(lEsq)
        mergeSort(lDir)

        merge(l, lEsq, lDir)
```

Para que ele funcione, é necessário criar uma função auxiliar que recebe duas listas ordenadas, e salva seus elementos em uma sequência ordenada:

```
def merge(l, lEsq, lDir):
    i = 0
    j = 0
    k = 0

    while i < len(lEsq) and j < len(lDir):
        if lEsq[i] < lDir[j]:
            l[k] = lEsq[i]
            i += 1
        else:
            l[k] = lDir[j]
            j += 1
        k += 1

    while i < len(lEsq):
        l[k] = lEsq[i]
        i += 1
        k += 1

    while j < len(lDir):
        l[k] = lDir[j]
        j += 1
        k += 1
```

8 Quick Sort

Assim como o Merge Sort, o **Quick Sort** também divide o problema em problemas menores. Como vantagem, ele não necessita de armazenamento extra para criar sublistas. Como desvantagem, sua eficiência pode ser prejudicada se a escolha do **pivô** for ruim.

O algoritmo seleciona um elemento qualquer como o pivô. Em seguida, divide a sequência entre os menores que o pivô à sua esquerda e os maiores à direita. Em seguida, ordena cada sublista recursivamente.

No exemplo a seguir, fazemos a primeira separação entre menores e maiores.

```
23   13   25   19   24   18   12   20
-----------------------------------------> pivô: 23
23 | 13   25   19   24   18   12   20
      i                            j
-----------------------------------------> procurando troca
23 | 13   25   19   24   18   12   20
           i                  j
-----------------------------------------> trocar j e i
23 | 13   12   19   24   18   25   20
                i         j
-----------------------------------------> j deve ser trocado
23 | 13   12   19   24   18   25   20
                     i    j
-----------------------------------------> trocar j e i
23 | 13   12   19   18   24   25   20
                     j    i
-----------------------------------------> fim: trocar j e pivô
18   13   12   19 | 23 | 24   25   20
------------------      -------------
   ordenar cada uma recursivamente
```

Em Python:

```python
def quickSort(l, inf, sup):
    if inf < sup:
        pos = particao(l, inf, sup)
        quickSort(l, inf, pos-1)
        quickSort(l, pos+1, sup)
```

Para que o algoritmo funcione, a função auxiliar "particao()" realiza a separacao entre menores e maiores, e retorna o índice do pivô após a partição:

```python
def particao(l, inf, sup):
    pivot = l[inf]
    i = inf+1
    j = sup

    while i <= j:
        while i <= j and l[i] <= pivot:
            i += 1
        while j >= i and l[j] > pivot:
            j -= 1
```

```
            if i < j: l[i], l[j] = l[j], l[i]

    l[inf], l[j] = l[j], l[inf]
    return j
```

9 Comparação entre os métodos

Método	*Melhor caso*	*Média*	*Pior caso*	Espaço extra
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick	$O(n \cdot lg\ n)$	$O(n \cdot lg\ n)$	$O(n^2)$	$O(1)$
Merge	$\Theta(n \cdot lg\ n)$			$O(n)$

Tabela 1: Comparação entre os métodos de ordenação.