Python

Ordenação e Pesquisa

Prof. Dr. Dieval Guizelini

Analista e Desenvolvedor de Sistemas

Mestre em Bioinformática e Doutor em Ciências-Bioquímica

dieval at ufpr.br / dievalg at gmail.com

Coleções

- Mantém um conjunto de elementos, geralmente, homogêneos (mesmo tipo ou com propriedades em comum).

- Perguntas ou operações que incidem sobre coleções:
  - Busca(coleção, chave/elemento) => posição ou objeto
  - Ordenação(coleção [, comparador])
  - Inserção(coleção, novo_elemento)
  - Exclusão(coleção, elemento/índice)
  - Sucessor(coleção, elemento/índice)
  - Predecessor(coleção, elemento/índice)
  - Menor(coleção)
  - Maior(coleção)

Por quê organizar uma coleção?

1. Para fins de apresentação (requisito de sistema)

2. Para **melhorar o processo de busca**

- **Quais os algoritmos mais utilizados?**
  1. Simples: Inserção / seleção
  2. Eficientes: merge sort / heapsort / quicksort
  3. bubblesort e suas variações do : bubblesort / shellsort / comb sort
  4. Distribuição: counting sort / bucket sort / radix sort

Prática de “criar dados” ou popular coleções

- Premissa: o comportamento de um algoritmo de ordenação ou de busca tende a ser o mesmo independente do tipo
- Para fins de estimativa, as medidas realizadas em coleções com 10, 100, 1000, 10.000 ... elementos são o suficiente para prever o comportamento para massas gigantescas
- O que importa na análise do algoritmo é a “reta” ou “curva” que liga os 3 ou mais pontos (função de crescimento)
- A estimativa visa prever o consumo de tempo e memória.

Exemplo:

- v1 = [1, 3, 5, 7, 9, 2, 8, 4, 6]
- v2 = ['zona', 'Zebra', 'Casa', 'Maça', 'bola', 'leão', 'gato', 'banana', 'açucar', 'baralho']
- Qual o resultado para:

'bola' in v2	True
‘México’ in v2	False
4 in v1	True
13 in v1	False

- Ou

v1.index(4)	7

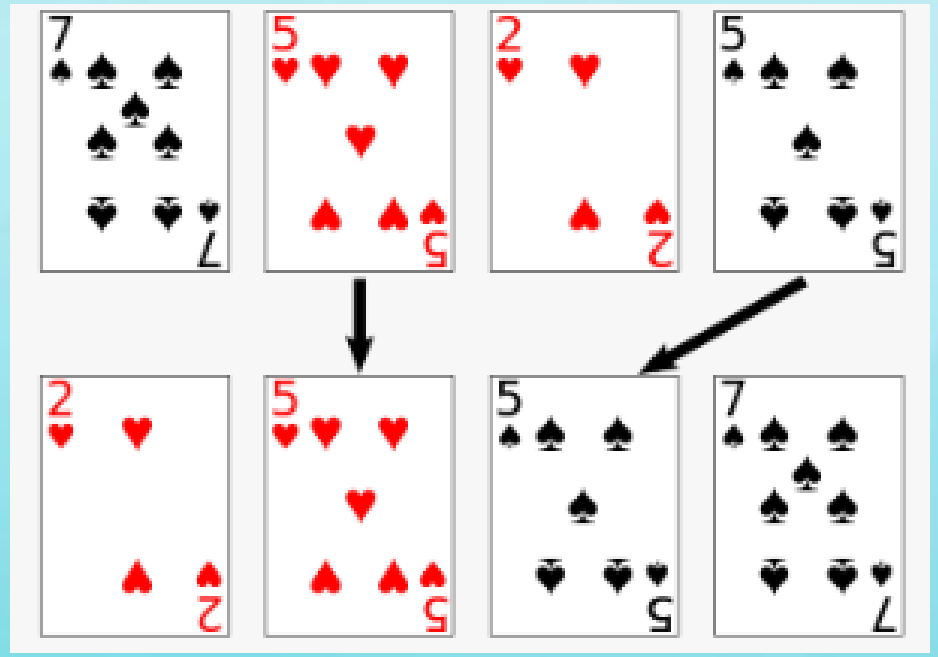
Critérios de comparação dos algoritmos

- Tempo de execução
- Consumo de memória
- Número de comparações
- Número de permutação (swaps)
- Estudos de casos: pior / médio / melhor
- Estabilidade

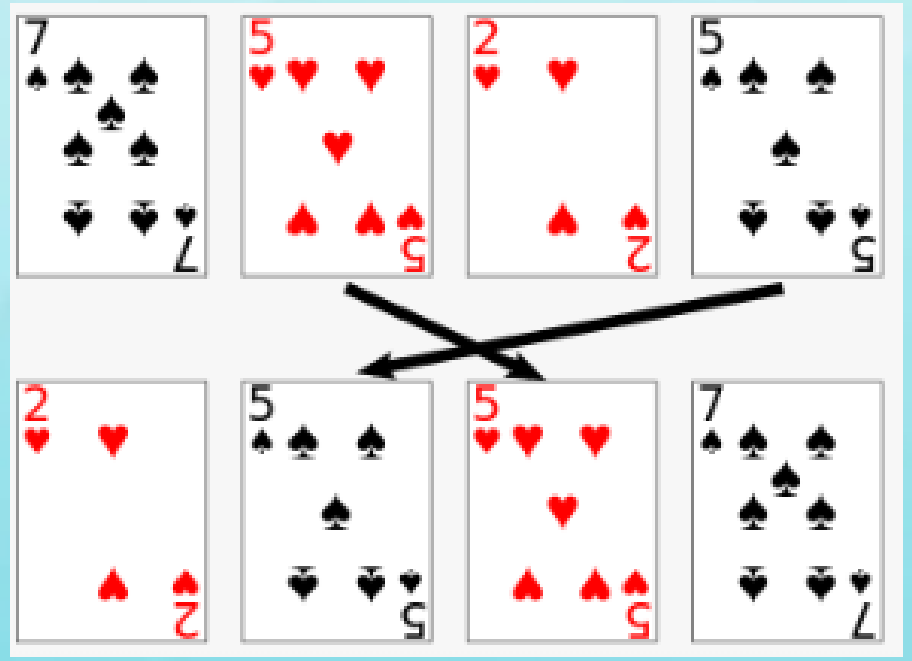
Estabilidade do algoritmo

Não importa a ordem de entrada, o resultado é sempre o mesmo.

Estável



Instável



Complexidade dos algoritmos

Algoritmo	Melhor	Médio	Pior	Memória	Estável	Obs
Bubble	n	n^2	n^2	1x	Sim	Perm.
shellsort	n log n	n $\log^2$ n $N^{5/4}$	n $\log^2$ n	1	Não	Inserção
comb sort	n log n	n^2	n^2	1	Não	Permut.
seleção	n^2	n^2	n^2	1	Não	Seleção
inserção	n	n^2	n^2	1	Sim	inserção
quicksort	n log n	n log n	n^2	log n	Não*	Partic.
merge sort	n log n	n log n	n log n	n / O(1)	Sim	Mescl.
Heapsort	n log n	n log n	n log n	1	Não	Seleção
counting sort		n+r	n^2*k	n . k	Sim	r – range
bucket sort		n+r	n^2*k	n . k	Sim	k-keys
radix sort		n.(k/d)	n.(k/d)	$n+2^d$	Não	d – digitos

Além do tempo... o espaço

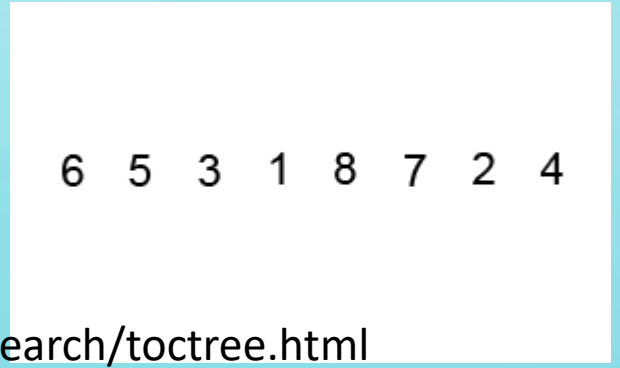
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Fonte: https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889

BubbleSort

```
v = [ 6, 5, 3, 1, 8, 7, 2, 4 ]
    [5, 3, 1, 6, 7, 2, 4, 8]
    [3, 1, 5, 6, 2, 4, 7, 8]
    [1, 3, 5, 2, 4, 6, 7, 8]
    [1, 3, 2, 4, 5, 6, 7, 8]
    [1, 2, 3, 4, 5, 6, 7, 8]
    [1, 2, 3, 4, 5, 6, 7, 8]
    [1, 2, 3, 4, 5, 6, 7, 8]
```

```
def bubbleSort(alist):
        for passos in range(len(alist)-1,0,-1):
                for i in range( passos ):
                        if alist[i]>alist[i+1]:
                                temp = alist[i]
                                alist[i] = alist[i+1]
                                alist[i+1] = temp
                print(alist);
```



http://interactivepython.org/runestone/static/pythonds/SortSearch/toctree.html

Seleção (SelectionSort)

```
def selectionSort(alist):
        for tamanhoLogico in range(len(alist)-1,0,-1):
                posMaior=0
                for ind in range(1,tamanhoLogico+1):
                        if alist[ind]>alist[posMaior]:
                                posMaior = ind

                temp = alist[tamanhoLogico]
                alist[tamanhoLogico] = alist[posMaior]
                alist[posMaior] = temp
                print( alist );
```

```
>>> selectionSort(ml)
[1, 8, 7, 6, 5, 4, 3, 2, 9]
[1, 2, 7, 6, 5, 4, 3, 8, 9]
[1, 2, 3, 6, 5, 4, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Inserção (insertion sort)

```
def insertionSort(alist, novo):
        pos = len(alist)
        for index in range(0,len(alist)):
                if alist[index] > novo:
                        pos = index
                        break
        alist.insert(len(alist),0)
        for index in range(len(alist)-1,pos,-1):
                alist[index] = alist[index-1];
        alist[pos] = novo
        print(alist)
```

Inserção (insertion sort) - Variação

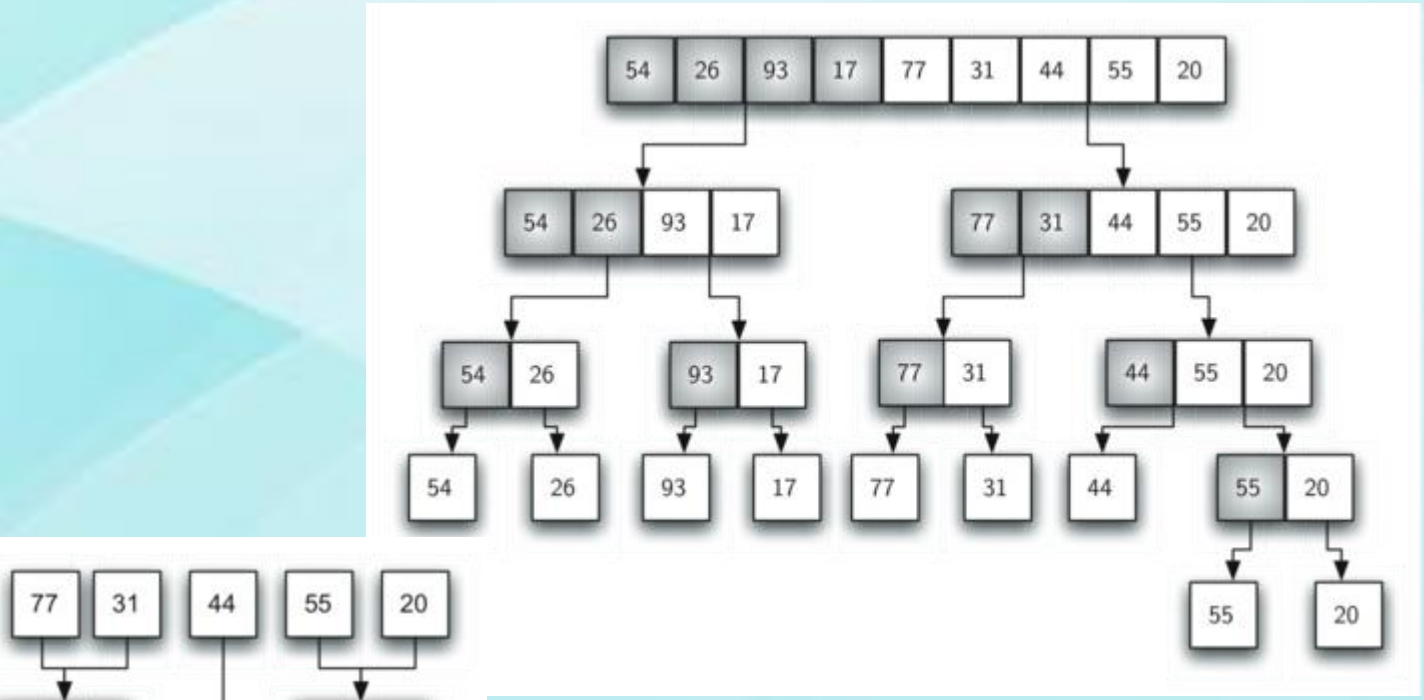
```
def insertionSort(alist):
        for index in range(1,len(alist)):
                currentvalue = alist[index]
                position = index
                while position>0 and alist[position-1] >
currentvalue:
                        alist[position]=alist[position-1]
                        position = position-1
                alist[position]=currentvalue
                print(alist)
```

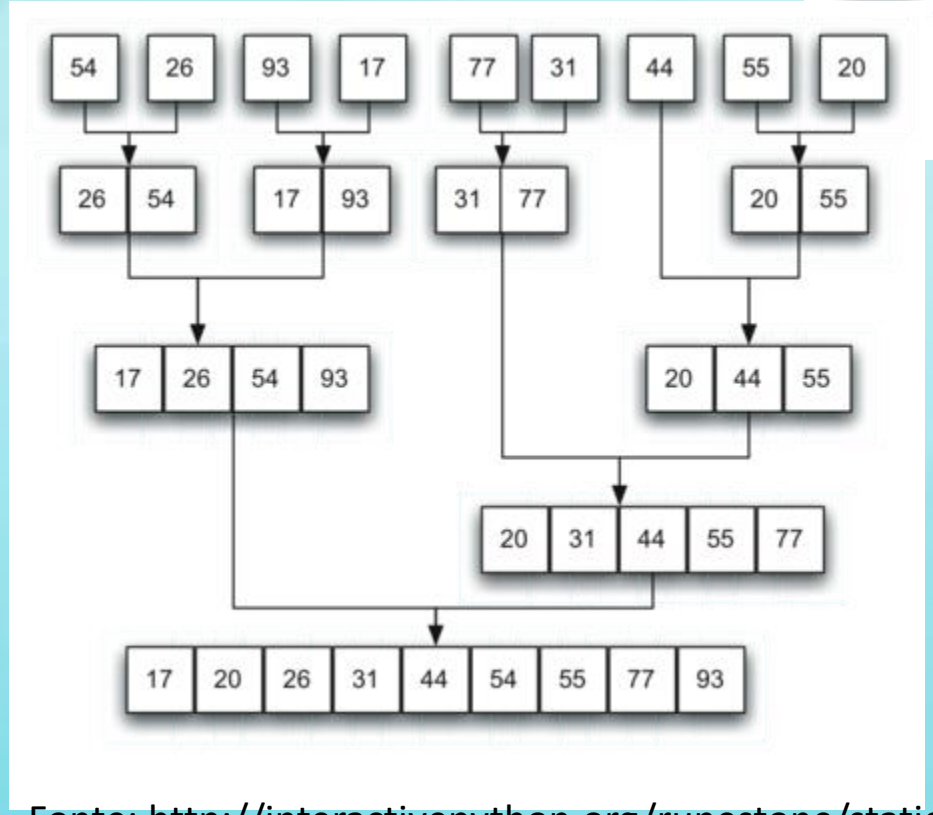
ShellSort

```
def shellSort(alist):
    meio = len(alist)//2    # div por inteiro
    while meio > 0:
        for ini_pos in range(meio):
            for i in range(ini_pos+meio,len(alist),meio):
                valorAtual = alist[i]
                pos = i
                while pos>=meio and alist[pos-meio]>valorAtual:
                        alist[pos]=alist[pos-meio]
                        pos = pos-meio
                alist[pos]=valorAtual
             print( alist );
        meio = meio // 2
```

```
>>> ml = [ 9, 8, 7, 6, 5, 4, 3, 2, 1
>>> shellSort(ml)
[1, 8, 7, 6, 5, 4, 3, 2, 9]
[1, 4, 7, 6, 5, 8, 3, 2, 9]
[1, 4, 3, 6, 5, 8, 7, 2, 9]
[1, 4, 3, 2, 5, 8, 7, 6, 9]
[1, 4, 3, 2, 5, 8, 7, 6, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Merge Sort





Fonte: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

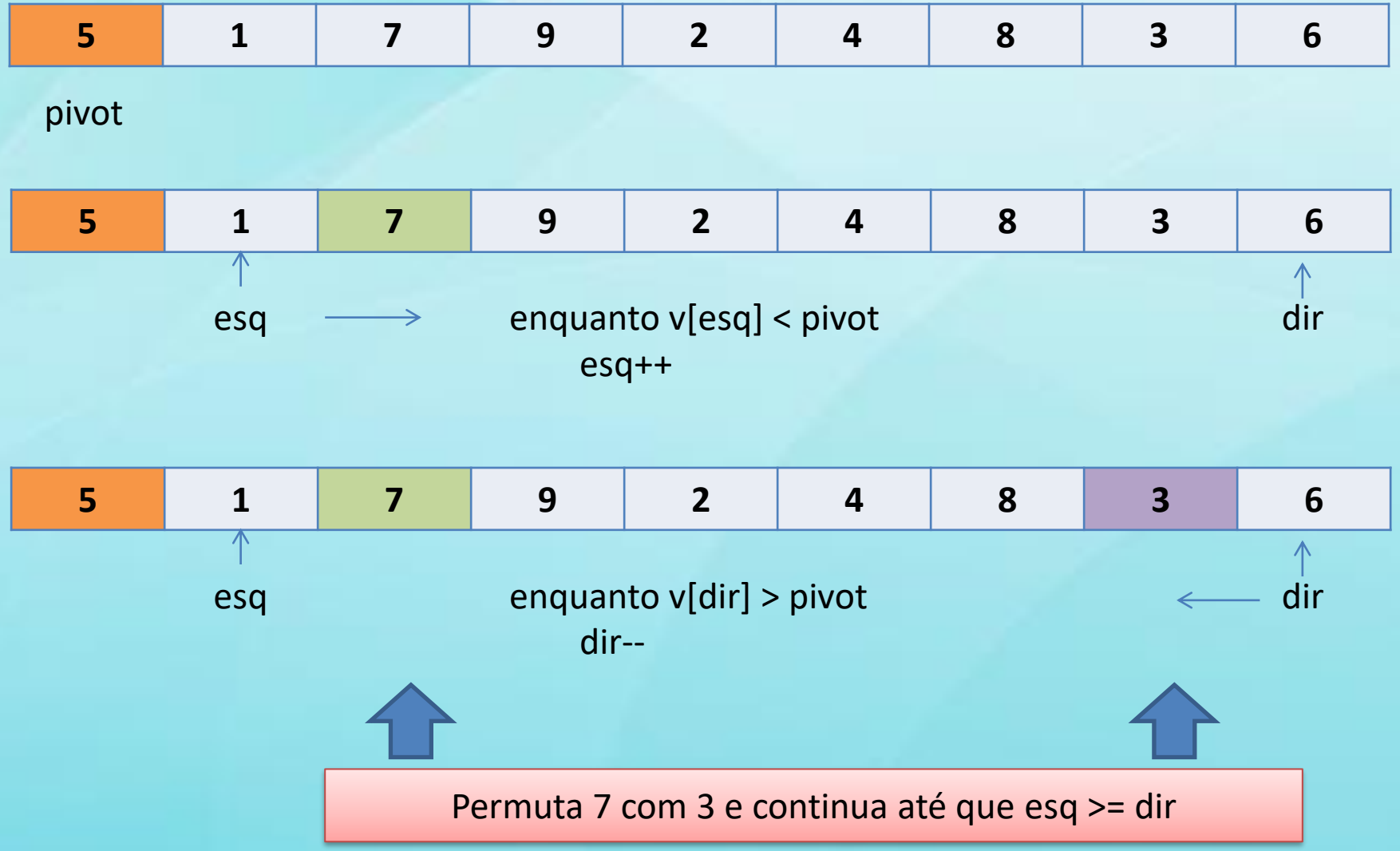
```
def mergeSort(alist):
      # dividindo a lista
    if len(alist)>1:
        meio = len(alist)//2
        metadeEsq = alist[:meio]
        metadeDir = alist[meio:]
        # chamada recursiva
        mergeSort(metadeEsq)
        mergeSort(metadeDir)
        i=0
        j=0
        k=0
        while i < len(metadeEsq) and j < len(metadeDir):
            if metadeEsq[i] < metadeDir[j]:
                alist[k]=metadeEsq[i]
                i=i+1
            else:
                alist[k]=metadeDir[j]
                j=j+1
            k=k+1
```

```
        while i < len(metadeEsq) and j < len(metadeDir):
            if metadeEsq[i] < metadeDir[j]:
                alist[k]=metadeEsq[i]
                i=i+1
            else:
                alist[k]=metadeDir[j]
                j=j+1
            k=k+1

        while i < len(metadeEsq):
            alist[k]=metadeEsq[i]
            i=i+1
            k=k+1

        while j < len(metadeDir):
            alist[k]=metadeDir[j]
            j=j+1
            k=k+1
    print("mescalndo ",alist)
```

QuickSort


5 1 7 9 2 4 8 3 6
pivot
5 1 7 9 2 4 8 3 6
esq
enquanto v[esq] < pivot
esq++
dir
5 1 7 9 2 4 8 3 6
esq
enquanto v[dir] > pivot
dir--
dir
Permuta 7 com 3 e continua até que esq >= dir

QuickSort

QuickSort

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
    if first<last:
        splitpoint = partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
```

QuickSort

```
def partition(alist,first,last):
    pivotvalue = alist[first]
    leftmark = first+1
    rightmark = last
    done = False
    while not done:
        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1
        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark -1
        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp
    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp
    return rightmark
```

Busca binária

```
def buscaBinaria(alist, item):
        ini = 0
        fim = len(alist)-1
        meio = (ini+fim)//2
        while ini<=fim:
                meio = (ini+fim+1)//2
                if alist[meio] == item:
                        return( meio );
                else:
                        if alist[meio] < item:
                                fim = meio-1
                        else:
                                ini = meio+1
        return(-1)
```

Referência

- THOMAS H. CORMEN, CHARLES ERIC LEISERSON, RONALD RIVEST, RONALD L. RIVEST E CLIFFORD STEIN. Algoritmos: Teoria e Prática. MIT Press. 2012