PROGRAMAÇÃO III

AULA 2

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Sabe quando você acessa um *site* de compras e seleciona para a página organizar os produtos por preço, do mais barato ao mais caro? Ou então você está trabalhando com uma planilha de nomes de pessoas no Excel e precisa que os dados fiquem ordenados alfabeticamente? Sempre por trás de tudo isso existe um algoritmo de ordenação de dados.

Uma das aplicações mais comuns na área da computação é a necessidade de ordenarmos um conjunto de dados. Dê uma busca rápida na internet pelo termo em inglês *sorting algorithm list* e verá, literalmente, dezenas de algoritmos diferentes para ordenar dados. Isso ocorre porque uma das categorias de algoritmos mais estudadas no meio científico sempre foram os algoritmos de ordenação de dados, e a comunidade científica vem desenvolvendo distintos algorítmos para esse fim.

Cada algoritmo de ordenação apresenta uma complexidade assintótica de tempo e de espaço próprias e, por consequência, aplicações específicas. Existem algoritmos de ordenação específicos para ordenação objetos em cenas de jogos de *videogames*, por exemplo.

Aqui, iremos focar nossos estudos em três algoritmos clássicos para ordenação de dados: ordenação por troca, também conhecida como ordenação bolha (*Bubble Sort*), ordenação por intercalação (*Merge Sort*) e ordenação rápida (*Quick Sort*). Analisaremos seu funcionamento e complexidade.

Os exemplos conduzidos nesta etapa tratarão do uso da linguagem Python, ordenando conjuntos de dados dentro de listas. Apesar dessa escolha, a lógica por trás desses algoritmos é análoga em todas as linguagens de programação e pode também ser aplicada a outros tipos de dados, como arrays e matrizes em C/C++ e Java.

TEMA 1 – *BUBBLE SORT*

O algoritmo do ***Bubble Sort*** é também conhecido como **método da bolha,** ou **ordenação bolha**, ou ainda, **ordenação por troca**. Esse algoritmo é o de mais fácil entendimento e compreensão que temos quando nos referimos à ordenação de dados. Portanto, iniciamos por ele.

O *Bubble Sort* realiza a ordenação de um conjunto de dados por meio da comparação entre dois elementos adjacentes. O método faz uma varredura no

conjunto de dados, do início ao fim em pares. Caso os números estejam em lugares invertidos, troca-se. Caso contrário, mantém-se onde estavam. Está muito confuso o entendimento? Não tem problema. Vamos verificar um exemplo mais completo a seguir.

1.1 *Bubble Sort*: lógica de funcionamento

O algoritmo do *Bubble Sort* é clássico na literatura. Independentemente da linguagem a ser aplicada, sua lógica e funcionamento são imutáveis. Ela consiste em fazer varreduras no conjunto de dados, do início ao fim, pegando dois dados adjacentes e verificando se eles precisam ser trocados de ordem.

A quantidade de varreduras a serem feitas será igual ao tamanho do conjunto de dados. Exemplo, se tivermos um conjunto de dados com cinco dados, varreremos todo o conjunto cinco vezes.

Imagine agora que queremos ordenar o conjunto desornado de dados, contendo: $dados = [5, 8, 3, 4, 1]$, crescentemente. O algoritmo pegará os dois primeiros números (5 e 8) e irá compará-los para efetuar a troca. Uma vez comparado, avança-se para os números 8 e 3 e realiza-se o mesmo processo e, assim, indo até o final do conjunto de dados.

Na tabela a seguir, é apresentado todo o processo de ordenação, iteração por iteração, para as cinco varreduras. Em cinza, estão os dados comparados aos pares naquela iteração. Em azul, está o resultado final para aquela varredura. Vamos do início ao fim no conjunto de dados cinco vezes. Na coluna da esquerda, é apresentado se houve troca, ou não, naquela linha.

Tabela 1 – *Bubble Sort*: ordenando o conjunto de dados: $dados = [5,8,3,4,1]$.

Troca?	**Varredura 1**				
Não	5	8	3	4	1
Sim	5	8	3	4	1
Sim	5	3	8	4	1
Sim	5	3	4	8	1
v = 1	5	3	4	1	8
	Varredura 2				
Sim	5	3	4	1	8
Sim	3	5	4	1	8
Sim	3	4	5	1	8
Não	3	4	1	5	8
v = 2	3	4	1	5	8
	Varredura 3				

Não	3	4	1	5	8
Sim	3	4	1	5	8
Não	3	1	4	5	8
Não	3	1	4	5	8
v = 3	3	1	4	5	8
	Varredura 4				
Sim	3	1	4	5	8
Não	1	3	4	5	8
Não	1	3	4	5	8
Não	1	3	4	5	8
v = 4	1	3	4	5	8
	Varredura 5				
Não	1	3	4	5	8
Não	1	3	4	5	8
Não	1	3	4	5	8
Não	1	3	4	5	8
v = 5	1	3	4	5	8

Note que, na primeira e na segunda varredura, temos três trocas de dados. Na terceira e na quarta, somente uma. Na quinta varredura, nenhuma troca ocorre. O conjunto de dados já ficou ordenado ao final da iteração 4.

1.2 *Bubble Sort*: algoritmo

O *Bubble Sort* consiste em dois laços de repetição aninhados e uma condicional dentro deles. Veja:

```
def bubbleSort(dados):
  tam = len(dados)
  for v in range(0, tam, 1):
    for i in range(0, tam-1, 1):
      if dados[i] > dados[i+1]:
        aux = dados[i]
        dados[i] = dados[i+1]
        dados[i+1] = aux

#Programa Principal
dados = [5, 4, 2, 1, 8]
bubbleSort(dados)
print(dados)
```

O primeiro laço (variável *v*) é quem diz a quantidade de vezes em que a varredura deve ocorrer. Por esse motivo, fazemos um laço *for* iniciando em zero e indo até o tamanho do conjunto de dados, ou seja, de 0 até 5 no exemplo acima. Como na linguagem Python, o laço sempre se encerra no valor

imediatamente anterior ao valor de parada, nosso laço irá ser realizado 5 vezes (de 0 até 4).

O segundo laço, aninhado ao primeiro, inicia 0 e vai até o tamanho dos dados menos um (*tam - 1*). Mas por que essa condição de parada? Porque a última posição da lista é 4. Como as comparações são feitas aos pares, a última comparação feita deverá ser: *if dados[3] > dados[4]*. Ou seja, de maneira genérica, é equivalente a: *if dados[tam-2] > dados[tam-1]*.

Caso a condicional simples resulte em verdadeiro, significa que os dados estão no lugar errado e precisam ser trocados. Dentro dessa condicional, temos então a troca. Note a existência de uma variável auxiliar chamada *aux*. Essa variável serve para, temporariamente, auxiliar na troca dos valores.

Saiba mais

O algoritmo apresentado realiza a ordenação de forma crescente. Então talvez você esteja pensado agora: Como faço se quiser ordenar de maneira decrescente?

Para concretizar isso, basta que inverta a condição utilizada na condicional simples para: if dados[i] < dados[i + 1]. Tente fazer isso em casa e veja se deu certo!

1.3 Complexidade do *Bubble Sort*

Qual a complexidade Big-O da função do *Bubble Sort*? Como temos dois laços aninhados, mas as iterações do segundo laço não se alteram, temos uma PA constante e podemos fazer a propriedade da multiplicação. Sendo assim, o Big-O do ***Bubble Sort* é $O(n^2)$**.

1.4 Melhorando o *Bubble Sort*

Relembre a Tabela 1. Você notou que nosso conjunto de dados já ficou ordenado logo no início da quarta varredura? Será que não podemos aprimorar esse algoritmo para identificarmos quando ele ficou ordenado e interromper o processo a qualquer momento, ganhando em desempenho? A resposta é sim, podemos.

O *Bubble Sort* que você viu até então apresenta complexidade para o pior caso, igual ao melhor, e é $O(n^2)$. Embora nossa principal análise ao longo desta

caminhada seja para a complexidade do pior caso de nossos algoritmos, vamos parar por um instante para analisar o *Bubble Sort* no melhor caso também, para identificarmos como podemos melhorar esse algoritmo fazendo uma simples alteração.

Como já visto, o pior caso de um algoritmo é sempre aquele em que mais instruções são executadas. Sendo assim, o melhor caso é quando temos menos instruções executadas.

No *Bubble Sort* atual, se você passar como parâmetro para a função um conjunto de dados já ordenado (melhor caso), o algoritmo irá executar todas as iterações, gastando muito tempo desnecessário. Ou seja, o pior caso e o melhor caso são idênticos e igualmente ruins.

Podemos melhorar e corrigir isso inserindo uma variável que atue como uma *flag* no programa. Assim, essa *flag* pode identificar quando os dados já estão ordenados e interromper a ordenação precocemente, ganhando em desempenho. Veja a implementação dessa melhoria a seguir:

```
def bubbleSort(dados):
  tam = len(dados)
  for v in range(0, tam, 1):
    flag = 0
    for i in range(0, tam-1, 1):
      if dados[i] > dados[i+1]:
        aux = dados[i]
        dados[i] = dados[i+1]
        dados[i+1] = aux
        #se houve uma troca, coloca a flag em 1
        flag = 1
    #encerra precocemente se não houve troca
    if flag == 0:
      return
```

Assim, a complexidade do *Bubble Sort* melhorado é, para o pior caso, ainda $O(n^2)$, mas seu melhor caso agora fica somente $\Omega(n)$.

TEMA 2 – *MERGE SORT*

O **algoritmo de ordenação por intercalação**, ou ***Merge Sort***, usufrui da estratégia de dividir para conquistar. O *Merge Sort* realiza a ordenação dividindo um conjunto de dados em metades iguais e reorganizando essas metades. É um algoritmo que opera de maneira recursiva, dividindo de maneira contínua o conjunto de dados até eles tornarem-se indivisíveis.

Sempre que duas metades estão ordenadas, um processo chamado de mesclagem (ou intercalação) é performado. A intercalação é um processo que pega dois conjuntos de dados ordenados e os combina em um só, ordenando, resultando em novo conjunto de dados. Vamos agora compreender o funcionando desse algoritmo mais detalhadamente.

2.1 *Merge Sort*: lógica de funcionamento

Como temos um funcionamento pelo princípio de dividir para conquistar, precisamos entender qual é o problema que precisamos resolver e qual o caso-base.

Nosso problema consiste em ordenar um conjunto inteiro de dados, que em nossos exemplos serão listas em linguagem Python. Já nossos subproblemas, ou seja, aquele em que devemos dividir consiste em dividirmos o conjunto de dados sempre ao meio e ordenarmos as duas metades, atingindo o caso-base.

O primeiro passo do algoritmo de ordenação por intercalação é, portanto, encontrar o ponto central do nosso conjunto de dados, assim saberemos onde dividir nosso conjunto. Encontramos o meio do conjunto de dados pela equação:

$$meio = int\left(\frac{inicio + fim}{2}\right)$$

Em que *início* é a posição inicial do conjunto de dados, *fim* é a posição final do conjunto de dados e *int* indica que devemos ficar somente com a parte inteira do resultado da divisão. Em linguagem Python, podemos escrever a equação anterior de uma maneira simplificada, como:

$$meio = len(dados)//2$$

Em que *len* representa o tamanho do conjunto de dados sendo manipulado, e as duas barras // o resultado inteiro da divisão.

Com a equação anterior, **dividimos** os dados. Em seguida, **conquistamos** recursivamente os conjuntos de dados menores, até atingirmos os casos-base, que aqui corresponde à menor unidade indivisível de um conjunto de dados (conjunto menor do que dois elementos). Por fim, **combinamos** mesclando os subconjuntos de volta em um único conjunto.

Queremos agora ordenar o conjunto desordenado de dados, contendo: $dados = [54, 26, 93, 17, 77, 31, 44, 55]$. Vejamos o passo a passo:

0	1	2	3	4	5	6	7
54	26	93	17	77	31	44	55

Acima de cada número, temos o índice do conjunto de dados utilizando a nomenclatura da linguagem Python, que sempre inicia um conjunto de dados, como uma lista, no índice zero. Qual o ponto central para dividirmos esse conjunto?

$$meio = int\left(\frac{0+7}{2}\right) = int(3{,}5) = 3$$

Quebraremos nosso conjunto de dados no índice 3, deixando 4 valores para cada lado:

0	1	2	3
54	26	93	17
CONJUNTO DA ESQUERDA			

4	5	6	7
77	31	44	55
CONJUNTO DA DIREITA			

Só iremos parar de dividir quando nossos conjuntos tiverem tamanhos menores do que dois, certo? Portanto, continuamos e quebramos cada subconjunto em outros dois. Para tal, precisamos localizar o meio de cada um deles novamente:

$$meio_{esquerda} = int\left(\frac{0+3}{2}\right) = int(1{,}5) = 1$$

$$meio_{direita} = int\left(\frac{4+7}{2}\right) = int(5{,}5) = 5$$

Nossos quatro conjuntos, serão:

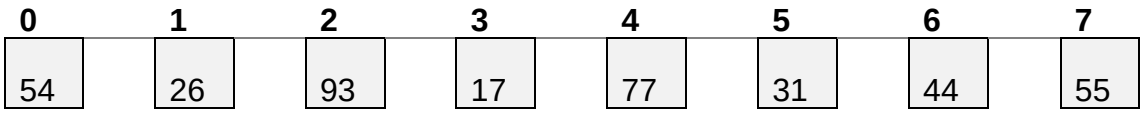
0	1
54	26
Esquerda	

2	3
93	17
Direita	

4	5
77	31
Esquerda	

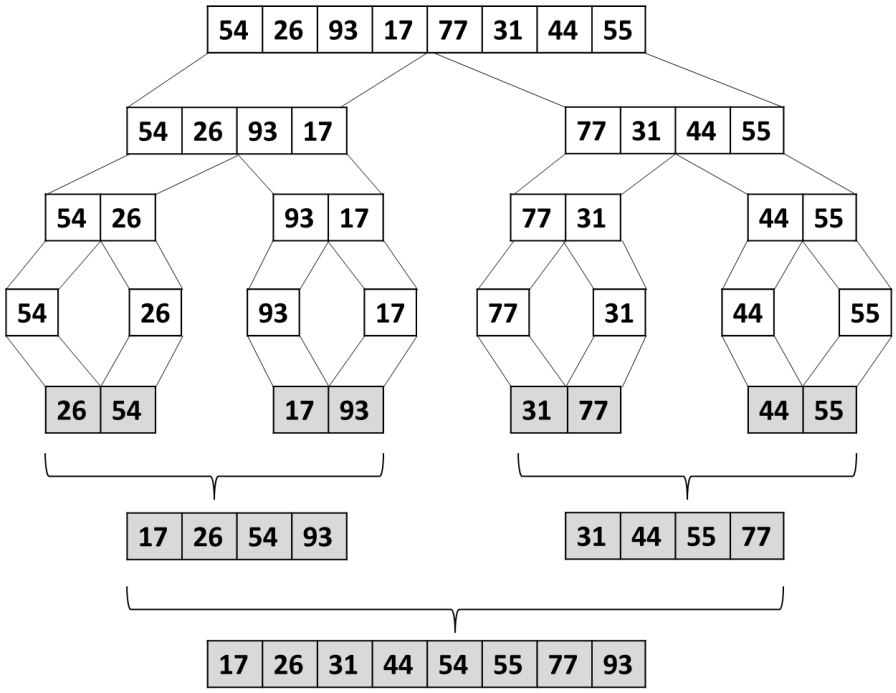
6	7
44	55
Direita	

Temos que dividir novamente porque nossos conjuntos têm tamanho dois. Ficamos com:



Atingimos oito conjuntos indivisíveis e de tamanhos unitários. Agora, iremos mesclar os conjuntos de volta até atingir um único conjunto novamente. Faremos isso mesclando da mesma forma que dividimos. Ao mesclar, verificamos a ordem dos dados dos subconjuntos. A seguir, temos todo o processo de divisão e a posterior mesclagem dos dados do conjunto.

Figura 1 – *Merge Sort*: ordenando crescentemente o conjunto de dados: $dados = [54, 26, 93, 17, 77, 31, 44, 55]$

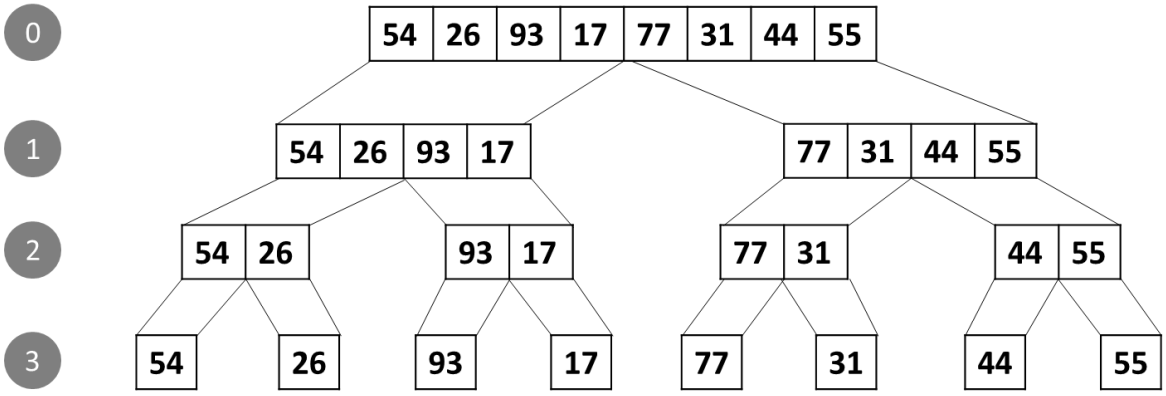


Note que quando atingimos as menores unidades de tamanho do nosso conjunto, iniciamos a grupá-las e ordená-las simultaneamente. Observe, por exemplo, do lado esquerdo os valores unitários 54 e 26. Ao serem mesclados, sua ordem já se altera, pois na ordenação crescente 26 é menor do que 54. Em seguida, mesclando o conjunto *[26, 54]* com *[17, 93]*, todos os quatro dados são verificados entre si e posicionados na ordem correta: *[17, 26, 54, 93]*.

Todas as divisões ocorrem de maneira recursiva e cada subconjunto é uma nova chamada. As funções recursivas, ao serem encerradas, vão originando os conjuntos ordenados de dados.

Observe que a quantidade de chamadas recursivas de um *Merge Sort* depende da quantidade de divisões que ocorreram. Conforme a Figura 2, houve-se a necessidade de quatro subdivisões até atingirmos o caso-base.

Figura 2 – *Merge Sort*: quantidade de níveis/subdivisões



Matematicamente, pode-se dizer que o total de conjuntos por nível é:

NÍVEL	TOTAL
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$

Com isso, a quantidade total de chamadas recursivas da função *Merge Sort* será de:

$$2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15\ chamadas\ recursivas$$

TEMA 3 – *MERGE SORT*: O ALGORITMO

O *Merge Sort* precisa ser analisado com bastante calma, haja vista que seu algoritmo é um pouco extenso. A seguir, apresentamos este código.

```
def mergeSort(dados):
  #Condiçao que indica se os dados ainda precisam ser divididos
  if len(dados) > 1:
    #divide recursivamente
    meio = len(dados)//2
    esquerda = dados[:meio]
    direita = dados[meio:]
    mergeSort(esquerda)
    mergeSort(direita)
    #intercala/mescla os dados
    i = j = k = 0
    while i<len(esquerda) and j<len(direita):
      if esquerda[i]<direita[j]:
        dados[k]=esquerda[i]
        i=i+1
      else:
        dados[k]=direita[j]
        j=j+1
      k=k+1
    while i<len(esquerda):
      dados[k]=esquerda[i]
      i=i+1
      k=k+1
    while j<len(direita):
      dados[k]=direita[j]
      j=j+1
      k=k+1


#Programa Principal
dados = [54, 26, 93, 17, 77, 31, 44, 55]
mergeSort(dados)
print(dados)
```

O nosso algoritmo se inicia verificando se o tamanho do conjunto de dados é maior do que um. Caso seja maior do que um, significa que precisamos ficar dividindo os dados recursivamente, até essa condição não ser mais satisfeita.

Para dividirmos os dados, encontramos o ponto central (variável *meio*) e colocamos cada parte do conjunto em duas listas separadas (variáveis *esquerda* e *direita*). Note que, após dividido o conjunto, a função *mergeSort* é invocada novamente e ficamos nesse processo até obtermos os casos-base.

Em seguida, quando queremos mesclar os dados, fazemos um laço de repetição e dentro dele uma condicional simples. O objetivo desse laço é o de ir verificando a ordem dos elementos e colocando-os ordenados de volta dentro da variável da lista.

Após o primeiro laço, existem outros dois laços de repetição. Esses dois laços servem para preencher as lacunas que irão faltar de dados sobrando nos vetores.

Saiba mais

Como estão ordenados esse algoritmo de maneira decrescente?

A alteração ocorre somente em uma linha. Dentro do primeiro laço, temos uma condicional, certo? Experimente alterar o sinal de menor para o sinal de maior, ao comparar a parte esquerda com a direita e veja o que acontece.

3.1 Complexidade do *Merge Sort*

Qual a complexidade Big-O da função do *Merge Sort*? A análise aqui deve ser mais cautelosa do que o que fizemos no *Bubble Sort*. Lembre-se do princípio de funcionamento desse algoritmo. Ele opera com o princípio de dividir para conquistar, isso significa que teremos uma complexidade O(logn) atrelada a esse algoritmo. Porém, não paramos por aqui.

No algoritmo, após o dividir para conquistar, temos ainda laços de repetição posicionados um após o outro. Como os laços não estão aninhados, utilizamos o princípio da adição: O(n) + O(n) + O(n) = O(n). Por fim, utilizamos o princípio da multiplicação, para agregarmos as duas partes do algoritmo em uma só complexidade:

$$O(logn).O(n) = \boldsymbol{O_{MergeSort}(n.logn)}$$

TEMA 4 – *QUICK SORT*

O algoritmo de ordenação rápida, ou *Quick Sort*, também usufrui da estratégia de dividir para conquistar como o seu irmão *Merge Sort*. O *Quick Sort* é talvez o algoritmo de ordenação mais famoso e também o mais utilizado entre todos, pois funciona muito bem para grande maioria dos casos de ordenação.

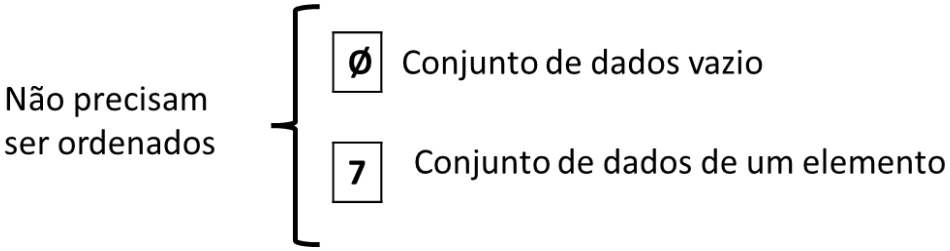
Todavia, a maneira como o *Quick Sort* trabalha com o dividir para conquistar é ligeiramente diferente, pois usufrui das características de desempenho dessa estratégia, mas sem precisar utilizar memória adicional como o *Merge Sort*. Como desvantagem perante a ordenação por intercalação,

é que nem sempre o conjunto de dados será divido exatamente ao meio e, quando isso ocorrer, teremos um certo impacto no seu desempenho.

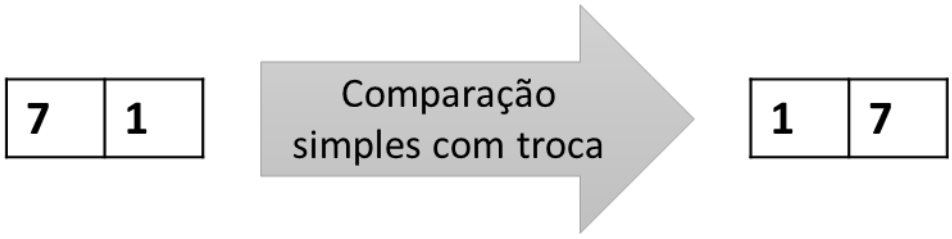
Vamos agora compreender o funcionando desse algoritmo mais detalhadamente.

4.1 *Quick Sort*: lógica de funcionamento

Toda nossa análise será realizada para ordenação de dados crescente (do menor para o maior). Antes de sairmos ordenando um conjunto de dados com diversos elementos, vamos iniciar nossa análise por algo mais simples, certo? Qual é o conjunto de dados mais simples que um algoritmo de ordenação pode ordenar? Veja bem. Existem conjuntos que não precisam ser ordenados. Veja quais são:



Um conjunto de dados com dois elementos é bastante simples de ser ordenado. Basta verificar se o primeiro elemento é menor do que o segundo e trocá-los de lugar, caso necessário.



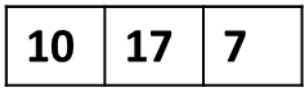
Até o momento, nenhuma dificuldade ou novidade, certo? Mas e um conjunto com três elementos? Como o *Quick Sort* ordenaria? Veja bem, esse algoritmo trabalha com dividir para conquistar. Portanto, precisamos atingir o caso-base. Eis a lógica desse algoritmo.

1. Escolhemos um elemento dentro do conjunto de dados, o qual chamamos de **pivô**. Em tese, o pivô pode ser qualquer elemento do conjunto. Porém, normalmente na literatura é escolhido o elemento central (Ascencio, 2030) ou então o primeiro elemento do conjunto (Bhargava, 2039).

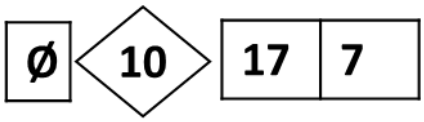
2. Em seguida, realizamos um processo chamado de **particionamento**. Onde encontram-se os elementos menores do que o pivô e também os elementos maiores.
3. Com isso, particionamos o conjunto de dados em três partes. Em que deixamos do lado esquerdo os valores menores que o pivô, e a direita, os maiores.

Quick Sort com 3 elementos

Vejamos o exemplo a seguir com três elementos:

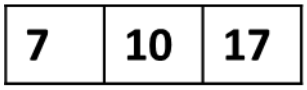


Vamos definir o pivô como sendo o primeiro elemento. Representaremos o pivô com um losango. Ao escolhermos o pivô, particionamos nosso conjunto de dados em dois. Como 10 é o primeiro elemento, ficamos com um conjunto vazio à esquerda:



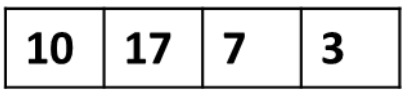
Na sequência, o valor menor do que 10 é colocado à esquerda, e o valor maior do que 10, à direta. Ficamos com:

Nosso conjunto de dados foi ordenado usando *Quick Sort*:

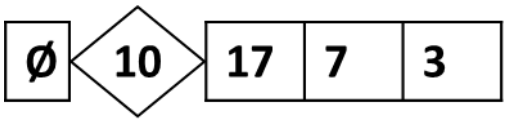


Quick Sort com 4 elementos

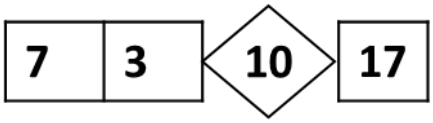
Vejamos o exemplo a seguir com quatro elementos:



Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:

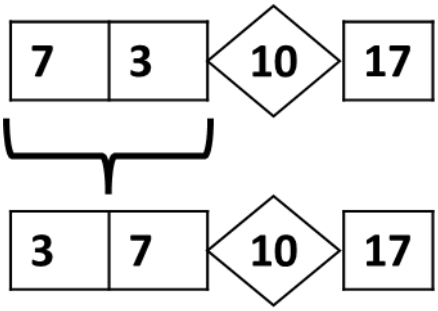


Os valores menores do que 10 são colocados à esquerda, e o valor maior do que 10, à direta. Ficamos com:

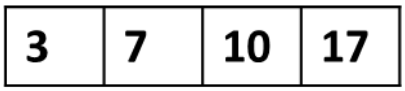


Observe que os valores à esquerda de 10 não estão ordenados. Isso porque o *Quick Sort* inicialmente só se preocupa em separar os valores para seus conjuntos, sem ordenar.

Agora, à esquerda do pivô ficamos com dois valores. Podemos então recursivamente ordenar esses valores, fazendo uma ordenação simples entre dois valores:



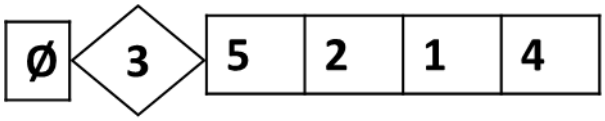
O resultado do vetor ordenado é:



Quick Sort com cinco elementos

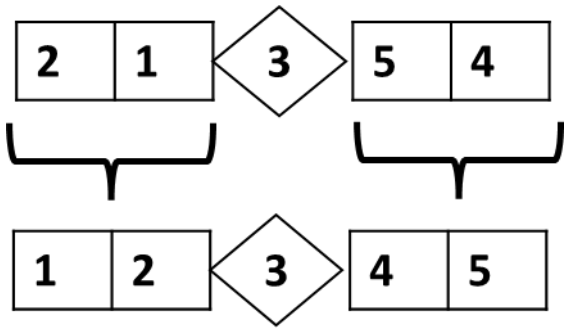
Vejamos o exemplo a seguir com cinco elementos:

Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:

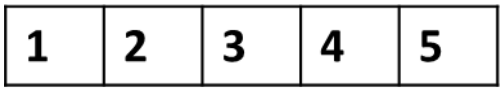


Dividindo os conjuntos:

Temos agora dois conjuntos de dois elementos para cada lado. Podemos fazer uma ordenação simples em cada lado:

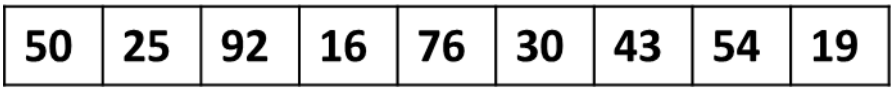


O resultado do vetor ordenado é:

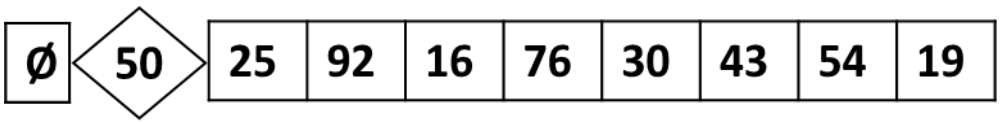


Quick Sort com oito elementos

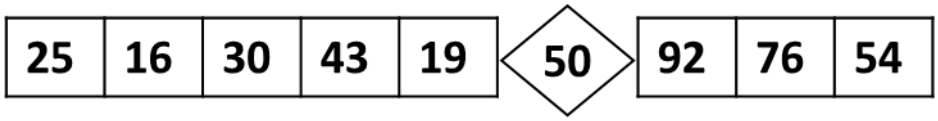
Vamos ao desafio final. Vejamos o exemplo a seguir com nove elementos:



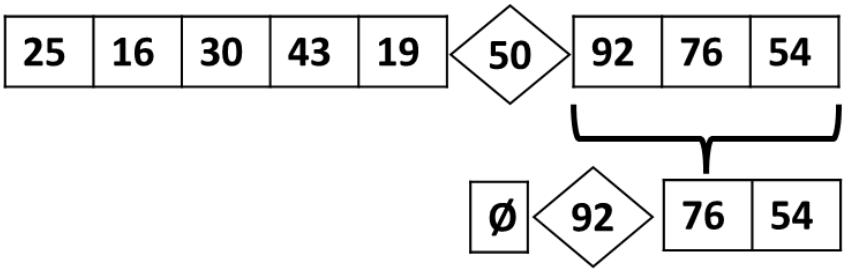
Vamos definir o pivô como sendo o primeiro elemento. Ficamos, portanto:



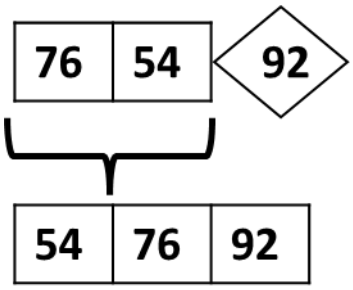
Dividimos os conjuntos:



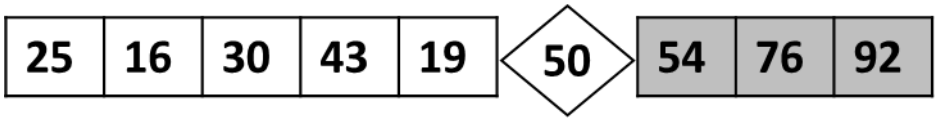
Temos agora dois conjuntos de dois elementos para cada lado. Vamos precisar continuar dividindo os conjuntos até obtermos conjuntos de até dois dados, em que é possível ordenar. Vamos começar pelo lado direito:



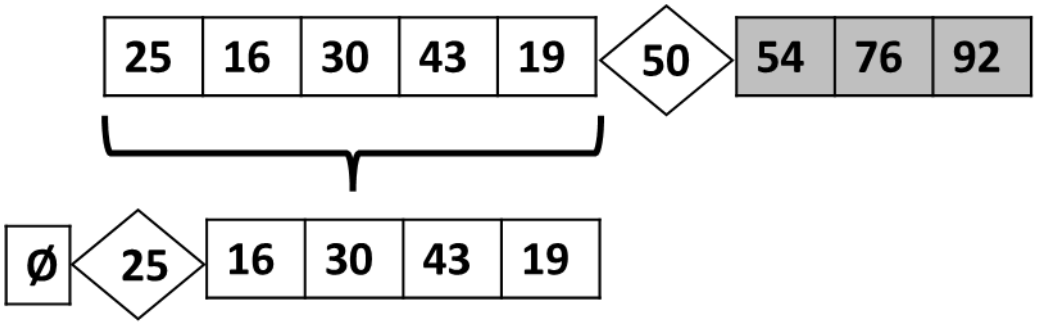
Definimos o pivô do conjunto de três dados como sendo novamente o primeiro. E, agora, conseguimos ordenar o que temos à direita, pois restaram somente dois dados:



Temos o conjunto do lado direito ordenado crescentemente. Deixaremos ele destacado em cinza na imagem e vamos trabalhar agora no lado esquerdo: definimos o pivô do conjunto de três dados como sendo novamente o primeiro. E, agora, conseguimos ordenar o que temos à direita, pois restaram somente dois dados:

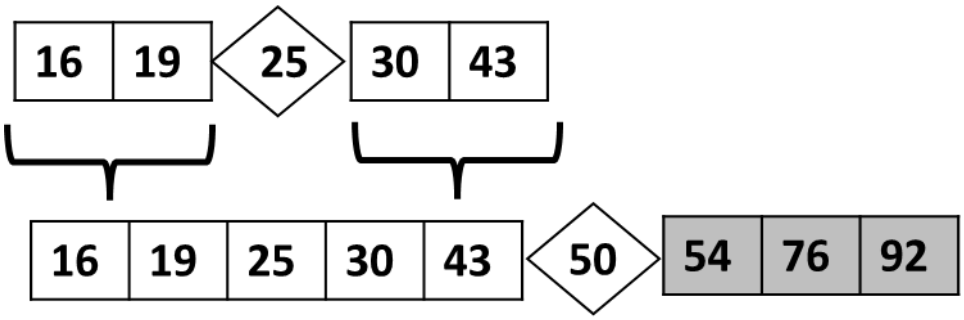


Note que, do lado esquerdo, temos agora um conjunto de cinco dados. Vamos então trabalhar ordenando esse conjunto como já vimos anteriormente. Definimos o pivô e dividimos o conjunto em duas partes:

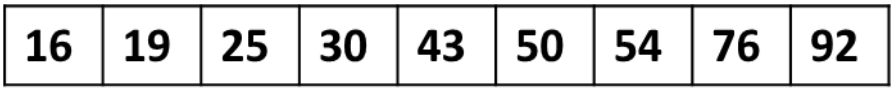


Organizamos os valores menores do que 25, antes dele, e os maiores, após ele:

Nesse momento, ficamos com dois conjuntos de dois dados em ambos os lados. Se notarmos a imagem, os dados já estão ordenados. De todo modo, o algoritmo iria realizar essa verificação para confirmar a ordenação, portanto, teríamos:



Desta forma, nosso conjunto agrupado novamente e ordenado crescentemente ficou:



TEMA 5 – *QUICK SORT*: O ALGORITMO

O *Quick Sort* precisa ser analisado com bastante calma, haja vista que seu algoritmo é um pouco extenso. A seguir, apresentamos esse código.

```
#Versão clássica
def quickSort(dados,inicio,fim):
  if inicio < fim:
    posicao_de_particionamento = partition(dados,inicio,fim)
    quickSort(dados,inicio,posicao_de_particionamento - 1)
    quickSort(dados,posicao_de_particionamento + 1,fim)

def partition(dados,inicio,fim):
  pivo = dados[inicio]
  esq = inicio + 1
  dir = fim
  flag = False
  while not flag:
    while esq <= dir and dados[esq] <= pivo:
      esq = esq + 1
    while dados[dir] >= pivo and dir >= esq:
      dir = dir -1
    if dir < esq:
      flag = True
    else:
      temp = dados[esq]
      dados[esq] = dados[dir]
      dados[dir] = temp
  temp = dados[inicio]
  dados[inicio] = dados[dir]
  dados[dir] = temp
  return dir

#Programa Principal
dados = [50,25,92,16,76,30,43,54,19]
quickSort(dados,0,len(dados)-1)
print(dados)
```

Note que, em nosso algoritmo, temos duas funções separadas. A primeira se chama *Quick Sort*. Ela é a responsável por ficar particionando os conjuntos de dados até atingirmos o caso-base, ou seja, cada conjunto de dados fica unitário.

A outra função chama-se *partition* (partição ou particionamento). Ela é responsável por realizar toda a separação e ordenação dos dados. Inicia-se com ela definindo o pivô como sendo o primeiro dado do conjunto. E, em seguida, varreduras são realizadas para que possamos separar os valores menores e maiores para seus respectivos lados.

Uma implementação alternativa do *Quick Sort* pode ser vista a seguir. A implementação a seguir parece ser bem mais simples do que a anterior, não concorda? Porém, ela só é possível de ser construída devido às características da linguagem Python, em que dentro de cada lista já podemos fazer as varreduras desejadas.

```
def quickSort(dados):
  if len(dados) < 2:
    return dados
  else:
    pivo = dados[0]
    menores = [i for i in dados[1:] if i <= pivo]
    maiores = [i for i in dados[1:] if i >  pivo]
    return quickSort(menores) + [pivo] + quickSort(maiores)


#Programa Principal
dados = [50, 25, 92, 16, 76, 30, 43, 54, 19]
dados = quickSort(dados)
print(dados)
```

O nosso algoritmo se inicia verificando se o tamanho do conjunto de dados é menor do que dois. Caso seja, caímos em um caso que é indivisível (conjunto de dados vazio ou unitário) e, portanto, não precisamos ordenar.

Note que, para o pivô, é sempre atribuído o primeiro dado do conjunto (índice zero). Os valores menores e maiores são calculados na sequência. As variáveis maiores e menores estão sendo definidas em Python como uma lista contendo os respectivos valores.

No *return* existe uma recursividade. Ou seja, antes da função retornar algo ela invoca ela mesma para ambos os lados, esquerda e direita.

Saiba mais

Como, então, ordenamos esse algoritmo de maneira decrescente?

No segundo algoritmo, tente inverter os sinais de comparação das listas de maior e menor. E na abordagem clássica, como ficaria?

5.1 Complexidade do *Quick Sort*

Qual a complexidade Big-O da função do *Quick Sort*? A análise aqui deve ser mais cautelosa do que o que fizemos no *Bubble Sort.* Lembre-se do princípio de funcionamento desse algoritmo. Ele opera com o princípio de dividir para conquistar, isso significa que teremos uma complexidade O (logn) atrelada a esse algoritmo. Analisando o algoritmo clássico, em que o código está mais detalhado, percebemos que a função *quickSort* é quem de fato opera recursivamente.

Ainda temos a função *partition*. Ela contém dois laços de repetição aninhados, o que nos remete a uma PA e, portanto, O(n) * O(n) = O(n²). Por fim, utilizamos o princípio da multiplicação para agregarmos as duas partes do algoritmo em uma só complexidade:

$$O(logn).O(n^2) = O(n^2.logn) = \boldsymbol{O_{QuickSort}(n^2)}$$

Observe que, na complexidade assintótica, ficamos somente com o termo de maior grau. Como *n²* cresce em grau muito mais do que *logn*, ficamos somente com O(n²).

FINALIZANDO

Nesta etapa, aprendemos três algoritmos de ordenação e suas respectivas complexidades. Veja o resumo comparativo.

- Bubble Sort: O(n²).
- Merge Sort: $O(n.logn)$.
- Quick Sort: O(n²).

REFERÊNCIAS

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Halt 3, 2010.

BHARGAVA, A. Y. **Entendendo Algoritmos.** Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. Tradução da 4ª edição norte-americana. Cengage Learning Brasil, 2018.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, Abstração, Estrutura de Dados e Projeto Usando C++.** Grupo GEN, 2008.