ESTRUTURA DE DADOS

AULA 1

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Já que estamos falando de **Estrutura de Dados**, nada melhor do que iniciarmos conceituando: O que são estruturas de dados?

Na programação, estruturas de dados são maneiras de organizar e colecionar dados. E a forma como os dados ficam organizados dentro da memória, o acesso a eles e suas manipulações caracterizam esse estudo.

Você já estudou programação antes e, certamente, já manipulou algumas estruturas de dados, independentemente da linguagem. Se você já trabalhou com linguagem Python, deve conhecer as estruturas de listas, tuplas e dicionários. No Java, e/ou no C/C++, você deve se lembrar do vetor (*array*), *string*, *struct*, *map*, entre outros. Até então, você sabia utilizar essas estruturas de dados. Neste estudo, você irá aprender a construir tais estruturas do zero – assim como outras –, conhecendo suas caracteristicas, aplicações, vantagens e desvantagens.

Vamos nos aprofundar em programação conhecendo estruturas de dados, estudando, majoritariamente, códigos sendo escritos em linguagem Python.

TEMA 1 – PESQUISA EM UM CONJUNTO DE DADOS

Todo e qualquer problema computacional que possa ser solucionado por meio de um algoritmo apresenta inúmeras soluções distintas. Cada desenvolvedor é capaz de pensar em uma solução ligeiramente diferente da de outro.

O fato de termos diferentes algoritmos possíveis para solucionar um problema nos traz uma pergunta pertinente: Todos os algoritmos terão o mesmo desempenho ao executar? Existem algoritmos mais – ou menos – eficientes que outros? Ademais, quais métricas são empregadas para analisar e comparar os algoritmos?

Vamos tentar responder a todas essas perguntas ao longo dessa etapa. Iniciaremos nosso estudo comparando dois algoritmos distintos para resolver o mesmo problema: **a busca dentro de um conjunto de dados**.

1.1 Pesquisa sequencial

Vamos supor que você e um amigo estão jogando um jogo de adivinhação de números, no qual seu amigo gostaria que você adivinhasse o número que está pensando dentro de um intervalo de 1 a 100 valores. A cada tentativa sua, seu amigo irá indicar se o valor está baixo ou alto em relação ao que ele pensou.

Imagine que o valor pensado por seu amigo é o 99, e você decidiu tentar adivinhar os valores assim: 1, 2, 3, 4, 5 etc. – até atingir o valor correto. A cada tentativa, seu amigo informa que o valor está baixo, e você chuta o próximo valor. Observe a Figura 1.

Figura 1 – Pesquisa sequencial: sequência de tentativas – em laranja, as tentativas; em verde, o acerto

Ao fazer esse tipo de tentativa, você está eliminando da lista apenas um valor por vez. Como o valor correto é 99, você precisa de 99 chances para acertar.

A implementação dessa lógica em um algoritmo nos remete a um simples laço de repetição. Se estivermos manipulando uma estrutura de dados de vetor, faremos uma varredura desse vetor iniciando no índice zero, até o final dele.

A seguir, o código apresenta a função que recebe como parâmetro um vetor e o valor a ser buscado, retornando, como saída, a posição no vetor onde o valor está.

```
"""
Algoritmo que realiza a busca sequencial em um vetor/lista.
Linguagem Python
Autor: Prof. Vinicius Pozzobon Borin
"""

def buscaSequencial (dados, buscado):
  achou = 0
  i = 0

  while (((i < len(dados))) and (achou == 0)):
    if (dados[i] == buscado):
      achou = 1
    else:
      i = i + 1
  if (achou == 0):
    return -1
  else:
    return i + 1

#programa principal
import random

# gerando 10 valores dentro de um intervalo de 0 até 9
dados = random.sample(range(10), 10)
print(dados)
buscado = int(input('Digite o valor que deseja buscar: '))
achou = buscaSequencial(dados, buscado)
if (achou == -1):
  print('Valor não encontrado.')
else:
  print('Valor encontrado na posição {}'.format(achou))
```

1.2 Pesquisa binária

Achou a busca sequencial ineficiente? Será que não conseguiríamos trabalhar com uma lógica mais eficiente para resolver o problema da busca? Certamente.

Agora, imaginemos que você irá tentar iniciar a adivinhar o número pelo valor 50. Seu amigo irá responder: “Está baixo!”. Sabendo disso, você acabou de descobrir que o valor a ser adivinhado está entre 50 e 100, ou seja, acabou de eliminar metade dos números em uma só tentativa.

Na próxima tentativa, se chutar o valor 75, estará novamente quebrando o conjunto de valores ao meio. Se continuar dessa maneira, eliminando metade dos dados, estará sempre reduzindo seu conjunto de dados ao meio, até restar somente o valor buscado. Esse princípio de raciocino lógico é chamado de **dividir para conquistar**.

A Tabela 1 mostra todas as sete tentativas necessárias para localizar o dado 99 dentro do conjunto.

Tabela 1 – Pesquisa binária: total de tentativas e intervalos

Tentativa	Intervalo inicial	Intervalo final	Valor do chute
1	1	100	50
2	50	100	75
3	75	100	87
4	87	100	93
5	93	100	96
6	96	100	98
7	98	100	99

A implementação dessa lógica em um algoritmo será estudada logo a seguir. A função recebe como parâmetro o vetor a ser buscado, o dado a ser localizado e o intervalo dentro do vetor a ser buscado.

No algoritmo, note que ele inicia localizando o valor central do conjunto de dados e verificando se ele é o valor a ser buscado (testes condicionais). A alterações das delimitações dos intervalos ocorrem quando uma condicional resulta em verdadeiro.

```
"""
Algoritmo que realiza a busca binária em um vetor/lista.
Linguagem C/C++.
Autor: Prof. Vinicius Pozzobon Borin
"""

def buscaBinaria (inicio, fim, dados, buscado):
  while(inicio <= fim):
    meio = int((inicio + fim)/2)
    if (buscado > dados[meio]):
      inicio = meio + 1
    elif (buscado < dados[meio]):
      fim = meio - 1
    else:
      return meio
  return -1

#programa principal
import random

#gerando 10 valores dentro de um intervalo de 0 até 9
dados = random.sample(range(10), 10)
dados.sort()
print(dados)
buscado = int(input('Digite o valor que deseja buscar: '))
achou = buscaBinaria(0, len(dados), dados, buscado)
if (achou == -1):
  print('Valor não encontrado.')
else:
  print('Valor encontrado na posição {}'.format(achou))
```

Existe um revés, porém, na busca binária. O fato de o conjunto de dados precisar, obrigatoriamente, estar ordenado (note o método *sort* invocado no algoritmo principal). Caso contrário, será impossível saber se o valor procurado está na primeira, ou na segunda parte do conjunto de dados.

1.3 A importância do conjunto de dados

Talvez você esteja se perguntando: “E se o número a ser adivinhado fosse 1? A busca sequencial não encontrará a resposta na primeira tentativa, sendo melhor que a binária?”. De fato, será. Isso significa que o algoritmo é mais eficiente? Não, pois a ideia de eliminar um só número por vez permanece.

A maneira como os dados estão organizados dentro do conjunto de dados é de suma importância para o desempenho real do algoritmo. Caso o número a ser adivinhado seja 1, isso significa que as condições para o algoritmo sequencial funcionar bem estavam ótimas. Mas isso não significa que, ao compararmos a complexidade de ambos os algoritmos, a busca sequencial é melhor. Trataremos mais desse assunto ao longo dessa etapa, e vamos estudar

como comparar matematicamente ambos os algoritmos, com o objetivo de definir qual tem melhor desempenho.

TEMA 2 – ANÁLISE DE ALGORITMOS

Vamos agora tentar, efetivamente, responder à seguinte questão: Como podemos comparar o desempenho de diferentes algoritmos para uma mesma aplicação? Podemos mensurar isso?

Quando queremos descobrir qual algoritmo é **mais eficiente** para resolver um problema, estamos falando do algoritmo de **menor complexidade**. O objetivo da verificação da complexidade é identificar como o desempenho do algoritmo cresce à medida que o tamanho do conjunto dos dados de entrada (*n*) cresce também. Ou seja, o algoritmo que consumir menos recursos, será de menor complexidade e mais eficiente. E o que é a complexidade do algoritmo?

Temos, portanto, dois tipos de complexidade de algoritmos:

- **Complexidade de tempo**: é o tempo que um algoritmo leva para completar sua execução. A quantidade de instruções do código impacta diretamente em seu desempenho.
- **Complexidade de espaço**: é a quantidade de memória requerida para a execução de um algoritmo. A quantidade de variáveis e seus tamanhos impactam diretamente em seu desempenho.

Muitas vezes, um algoritmo pode ser somente complexo em tempo, mas não em espaço, e vice-versa. De todo modo, nesse material, focaremos nossos estudos somente na complexidade de tempo dos algoritmos, deixando a complexidade de espaço de lado.

Para um mesmo algoritmo, o tempo levado para ele executar depende, majoritariamente, de três fatores:

1. **Tamanho do conjunto de dados de entrada (n)**: quanto mais dados temos para manipular, mais tempo.
2. **Disposição dos dados dentro do conjunto**: a ordem com que os dados estão organizados no conjunto implicada em distintas situações.
3. **Quantidade de instruções a serem executadas**: entendemos por instrução cada linha de código de programação de alto nível, independentemente do *hardware*.

Note que os dois primeiros itens independem do algoritmo escolhido para resolver o problema. Porém, a quantidade de instruções é diretamente proporcional à eficiência do algoritmo construído.

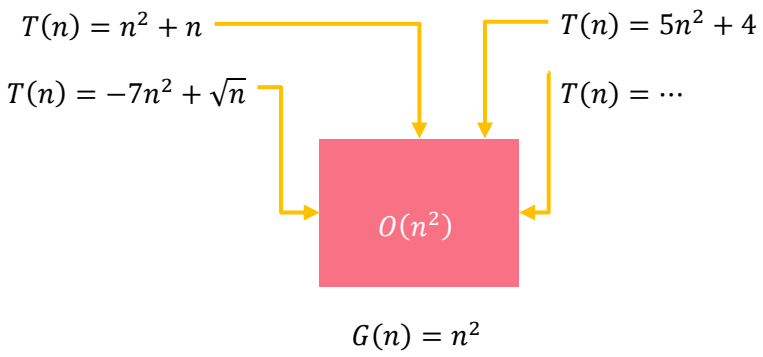
2.1 Notação *Big-O*

Para podermos comparar a complexidade dos algoritmos, podemos analisá-los matematicamente. A notação mais comum adotada na literatura para comparar algoritmos e dizer o quão rápido um algoritmo é, é a **notação *Big-O*** (ou “Grande-O”). Para compararmos a complexidade de algoritmos, devemos encontrar a função matemática que descreve cada um, e compará-las.

Assumindo que um conjunto T(n) de funções é tido como todas as funções que contêm ordem de grandeza menor ou igual a G(n), quando falamos de notação *Big-O*, dizemos que $O(G(n))$ é um conjunto de funções T(n), que pertencem à ordem de grandeza G(n).

Por exemplo: se $G(n) = n^2$, isso significa que a ordem de grandeza de todas as funções que pertencem a esse conjunto é n^2. A seguir, podemos ver diferentes funções que pertencem a esse conjunto.

Figura 2 – Conjunto de funções T(n) que pertencem à ordem de grandeza de $G(n) = n^2$



Podemos dizer que cada equação T(n) apresentada na Figura 2 é um algoritmo distinto para resolver um problema. Todos esses algoritmos apresentam a mesma ordem de grandeza e, portanto, a mesma **complexidade assintótica** (notação *Big-O*).

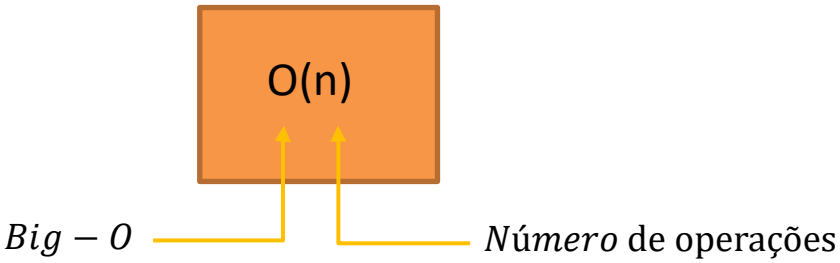
Desse modo, para sabermos a complexidade de um algoritmo utilizando a notação *Big-O*, basta encontrarmos o termo de maior grau da equação que o descreve.

A notação *Big-O* também é tida como o **limite inferior (pior caso)** de um algoritmo. Ou seja, o pior cenário que o algoritmo pode enfrentar para executar. Desse modo, com o *Big-O* teremos um parâmetro que indica que o algoritmo nunca será pior do que aquele cenário.

Há outras notações que podem ser investigadas. A que indica o **limite superior (melhor caso – Big-Ω)**, e o **caso médio (Big-θ)**. Essas notações não serão trabalhadas nesse estudo.

A notação *Big-O* é sempre representada pela letra "O" maiúscula. Dentro dos parênteses, temos o termo de maior grau da equação, o qual aprenderemos a calcular ainda nessa etapa. Note, na Figura 3, como se dá essa representação.

Figura 3 – Representação da notação *Big-O*



O termo de maior grau colocado na notação não representa diretamente a equação do tempo que o algoritmo irá levar para executar, pois o tempo irá depender do *hardware*, e estamos tentando abstraí-lo nessa análise.

Apesar disso, ainda podemos comparar tempos se fizermos algumas suposições. Imaginemos que temos um algoritmo que realiza a soma de *10* valores de um vetor (*n = 10*). Assumindo que cada somatório leva 1 ms para acontecer, quanto tempo O(n) e O(n²) levarão para executar essa soma?

$$O(n) = 10 * 1ms = 10ms$$
$$O(n^2) = 10^2 * 1ms = 100ms$$

O O(n²) levará bem mais tempo para concluir o cálculo. Portanto, ele é mais complexo em tempo de execução, ou seja, menos eficiente.

TEMA 3 – ENCONTRANDO O *BIG-O* DE ALGORITMOS

Como efetivamente encontramos os *Big-O* dos algoritmos? Precisamos agora estudar e analisar diferentes situações de algoritmos para descobrir.

Podemos iniciar imaginando um programa sem laços de repetição, nem funções recursivas. Considere um programa qualquer que somente some dois valores simples. Esse tipo de algoritmo irá executar o código uma única vez e se encerrar. Dizemos que um programa assim não tem dependência do conjunto de dados de entrada, desse modo, seu *Big-O* é dito como constante: **O(1)**.

3.1 Laço simples

Vamos agora investigar um algoritmo com um laço de repetição simples. O código a seguir faz a varredura de um vetor e imprime, na tela, um valor. O que está sendo executado dentro do laço é irrelevante para nossa análise.

É importante compreender o que acontece quando um *loop* é executado. A cada iteração de um laço, todas as instruções dentro de sua estrutura serão executadas *n* vezes. Neste exemplo, *n,* é o tamanho de nosso vetor. Se o vetor/lista for de dimensão 10, o laço ocorre 10 vezes.

```
def boo(dados):
  for i in dados:
    print(i)
```

Na análise *Big-O*, sabemos que a expressão matemática que define o algoritmo representa a quantidade de instruções a serem executadas. A complexidade assintótica desse algoritmo será, portanto, **O(n).**

3.2 Propriedade da adição

Vamos agora analisar uma propriedade matemática da notação *Big-O*. Sempre que houver um trecho de código seguido por outro que é independente, somamos suas notações *Big-O*. A seguir, veja dois laços de repetição simples, sendo executados um após o outro.

```
def boo(dadosA, dadosB):
  for i in dadosA:
    print(i)
  for i in dadosB:
    print(i)
```

Em ações consecutivas, fazemos adição:

$$T(n) = T_{laço1}(n) + T_{laço2}(n) = O(n) + O(n) = O(2n)$$

No *Big-O*, só nos interessa o termo de maior grau da equação. Portanto, negligenciamos o multiplicador dois na complexidade assintótica, ficando:

$$\boldsymbol{T(n) = O(n)}$$

3.3 Propriedade da multiplicação

Agora, sempre que houver um trecho de código aninhado a outro, ou seja, com relação de dependência entre eles, multiplicaremos suas notações. A seguir, observe dois laços de repetição simples sendo executados aninhados.

```
def boo(dadosA, dadosB):
  for i in dadosA:
    for i in dadosB:
      print(i + j)
```

Em ações aninhadas, fazemos multiplicação:

$$T(n) = T_{laço1}(n) * T_{laço2}(n) = O(n) * O(n) = O(n^2)$$

$$\boldsymbol{T(n) = O(n^2)}$$

3.4 Progressão aritmética (PA)

Vamos agora investigar um algoritmo com dois laços de repetição. Nesse caso, a análise é um pouco mais complexa. A seguir temos uma função na qual um laço está aninhado em outro. As instruções dentro dos laços são irrelevantes ao problema.

```
def boo(dadosA, dadosB):
  for i in dadosA:
    for j in dadosB:
      print(i, j)
```

Quando temos um aninhamento de laços, precisamos analisar a quantidade de iterações geradas com base na dependência entre os laços. A maneira como essa dependência existe irá representar a quantidade de vezes que o laço interno irá executar, caracterizando uma progressão matemática.

O caso mais comum é o de uma **série aritmética**, caracterizada como **uma sequência de números na qual a diferença entre dois termos consecutivos é constante**. Exemplos:

$$3 + 5 + 7 + 9 + 1 \ldots + n$$

$$8 + 4 + 0 - 4 - 8 \ldots + n$$

A soma dos *n* termos da progressão aritmética é dado por:

$$S_n = \frac{(a_1 + a_n).n}{2}$$

Em que S_n é o valor do somatório no enésimo termo, a_n é o enésimo termo da série, a_1é o primeiro termo da série. Observe esse exemplo:

$$1 + 2 + 3 + 4 + \cdots + n = \frac{(1 + n).n}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Para encontrarmos o *Big-O* dessa PA, precisaremos extrair da equação somente o termo de maior grau de crescimento. Para descobrirmos isso, podemos assumir que o *n* tende ao infinito, e substituir na equação, obtendo o resultado. Nesse caso, podemos negligenciar da equação o divisor 2, e também o termo de primeiro grau, ficando somente com n^2.

$$S_n = \frac{n^2}{2} + \frac{n}{2} = \boldsymbol{O(n^2)}$$

Um algoritmo com complexidade O(n²) normalmente terá dois laços de repetição aninhados. O laço da linha 4 depende do laço da linha 3. Se o tamanho do vetor for igual a 10, o segundo laço irá acontecer sempre 10 vezes o tamanho do vetor. Analisando o laço interno, temos uma PA constante, na qual:

$$10 + 10 + 10 + \cdots + \cdots$$

3.5 Progressão geométrica (PG)

Uma série geométrica é uma sequência de números na qual há uma razão entre um número e seu sucessor. Um algoritmo caracterizado por uma PG também irá trabalhar com dois laços aninhados.

Observe o exemplo a seguir: há dois laços aninhados, portanto, teremos O(n²), certo? Errado; precisamos analisar a progressão dos dados. Nesse cenário, o laço interno (linha 4), contém uma condição de parada na qual *j* depende de i *(j < i²)*.

```
def boo(dadosA, dadosB):
  for i in dadosA:
    for j in range(0, j < i * i, 1):
      print(i, j)
```

Nesse caso, o laço interno estará sempre aumentando a quantidade de vezes que irá executar, conforme representado na Tabela 2.

Tabela 2 – Variáveis dos laços que representam as iterações

I	J
0	0
1	1
2	4
3	9
4	16

Vejamos outros exemplos de séries geométricas:

$$1 + 3 + 9 + 27 + \cdots + n$$

$$1 + 2 + 4 + 8 + 16 \ldots + n$$

A soma dos *n* termos da progressão geométrica é dada por:

$$S_n = \frac{a_1 . (q^n - 1)}{q - 1}$$

Em que S_n é o valor do somatório no enésimo termo, a_1é o primeiro termo da série e *q* é a razão da série ($q = {}^{a_n}/_{a_{n-1}}$). Observe este exemplo:

$$1 + 2 + 4 + 8 + 16 + \cdots + n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots 2^n$$

$$q = \frac{a_n}{a_{n-1}} = \frac{2}{1} = 2$$

$$\frac{a_1 . (q^n - 1)}{q - 1} = \frac{1 . (2^n - 1)}{2 - 1} = 2^n - 1 = \boldsymbol{O(2^n)}$$

Assim, uma PG tem $\boldsymbol{O(2^n)}$, enquanto uma PA tem $\boldsymbol{O(n^2)}$.

TEMA 4 – DIVIDIR PARA CONQUISTAR

Uma das maneiras mais clássicas de se pensar racionalmente na solução de um problema chama-se **princípio de dividir para conquistar**. Podemos criar algoritmos que funcionam com esse princípio.

O dividir para conquistar faz com que criemos uma estratégia na qual há um problema de dimensão *n*, e reduzimos esse problema em partes menores, até que ele vire a menor unidade possível daquele problema (problema-base).

Algoritmos recursivos trabalham bastante com tal princípio, mas nem sempre é necessário trabalhar com recursividade para adotar tal estratégia, que consiste em duas etapas:

1. Descobrir o caso-base, que será sempre a menor parte possível para o problema;
2. Descobrir como reduzir o problema para que ele se torne o caso-base.

Vamos relembrar do algoritmo com que iniciamos essa etapa: o algoritmo de busca binária. Nele, pegamos um problema de dimensão *n* (o tamanho do nosso vetor é a dimensão), e dividimos o problema em partes menores, quebrando sempre ao meio, até obtermos a menor unidade possível dentro do vetor (caso-base). Essa menor unidade é a solução desse problema de busca.

Nem sempre localizar o caso-base irá finalizar nosso algoritmo. Iremos investigar, ao longo de nossos estudos, diferentes exemplos de dividir para conquistar, como algoritmos de ordenação de dados.

Como sabemos a complexidade de algoritmo que opera com esse princípio? Vamos, a seguir, realizar uma análise matemática utilizando o algoritmo de busca binária.

Assumindo que *n* é o conjunto de dados (tamanho do vetor de busca) e *k* é o número máximo de operações de busca possível, podemos ter:

$$\left.\begin{matrix}16\\8\\4\\2\\1\end{matrix}\right\} n = 16; k = 5 \qquad \left.\begin{matrix}8\\4\\2\\1\end{matrix}\right\} n = 8; k = 4$$

$$\left.\begin{matrix}4\\2\\1\end{matrix}\right\} n = 4; k = 2 \qquad \left.\begin{matrix}2\\1\end{matrix}\right\} n = 2; k = 1 \qquad 1\} n = 1; k = 1$$

Colocando isso em uma tabela, temos:

Tabela 4 – Conjunto de dados (*n*) e tentativas (*k*)

k	*n*	N como potência de 2
5	16	$2^4 = 2^5 - 1$
4	8	$2^3 = 2^4 - 1$
3	4	$2^2 = 2^3 - 1$
2	2	$2^1 = 2^2 - 1$
1	1	$2^0 = 2^1 - 1$

Perceba que o número de tentativas sempre aparece no expoente quando colocado o conjunto de dados como uma base de 2. Portanto podemos generalizar o problema da seguinte maneira:

$$2^{k-1} = n$$
$$k - 1 = \log_2 n$$
$$k = \log_2 n + 1$$

Para a notação *Big-O*, negligenciamos os termos de menor crescimento, ficando com somente:

$$\boldsymbol{O(\log n)}$$

Todo e qualquer algoritmo que opere com o princípio de dividir para conquistar terá uma complexidade logarítmica atrelada a ele.

TEMA 5 – COMPLEXIDADE DA RECURSIVIDADE

A complexidade de algoritmos recursivos demanda uma análise bastante minuciosa. De modo geral, para encontrarmos a complexidade de um algoritmo recursivo, devemos:

1. Calcular a complexidade de uma única chamada da função;
2. Expressar o número de chamadas recursivas por parâmetros de entrada;
3. Multiplicar o número de chamadas recursivas pela complexidade de uma chamada da recursão.

A fatorial é um caso clássico na literatura de recursividade, pois podemos escrever qualquer função recursiva, como nesse exemplo da fatorial de 5:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

$$5! = 5 * 4 * 3 * 2 * 1!$$

$$5! = 5 * 4 * 3 * 2!$$

$$5! = 5 * 4 * 3!$$

$$5! = 5 * 4!$$

Note que a fatorial de 5 depende da fatorial de 4, que, por sua vez, depende da fatorial de 3, e assim por diante. Vejamos o caso do algoritmo de cálculo da fatorial utilizando recursividade.

```
#Fatorial recursivo
def fatorial(n):
  fat = 1
  if n == 0 or n == 1:
    return fat
  else:
    return n * fatorial(n-1)

#programa princinpal
x = fatorial(5)
print(x)
```

Nele, há uma função chamada *fatorial*, que chama ela mesma até que a condição da linha 7 seja satisfeita. Vamos analisar as três etapas colocadas acima:

1. A complexidade de uma única chamada da função será O(1), uma vez que só temos instruções simples a serem executadas;
2. A quantidade de chamadas recursivas depende de quantas vezes a função é chamada novamente para cada instância da função aberta. No exemplo, cada função chama a si mesma uma só vez. Se o conjunto de dados for *n,* teremos *n* chamadas;
3. Assim, faremos: $O(1).n = O(n)$.

5.1 Equação geral da recursividade

Existe uma equação que pode simplificar essa tarefa quando se trata de recursão. Porém, atente-se ao fato de que ela só é válida quando a quantidade

de vezes que uma função recursiva invocar ela mesma duas ou mais vezes. A equação consiste em:

$$O(uma\ chamada) * O(nº\ de\ chamadas^{nº\ níveis})$$

Restrições:

- O número de chamadas deve ser estritamente maior do que 1;
- Se o número de chamadas for igual a 1, teremos uma cadeia linear de chamadas (como na fatorial);
- Se a recursão ocorre em um *loop*, o número de chamadas depende de cada caso específico. Estudaremos isso melhor em breve.

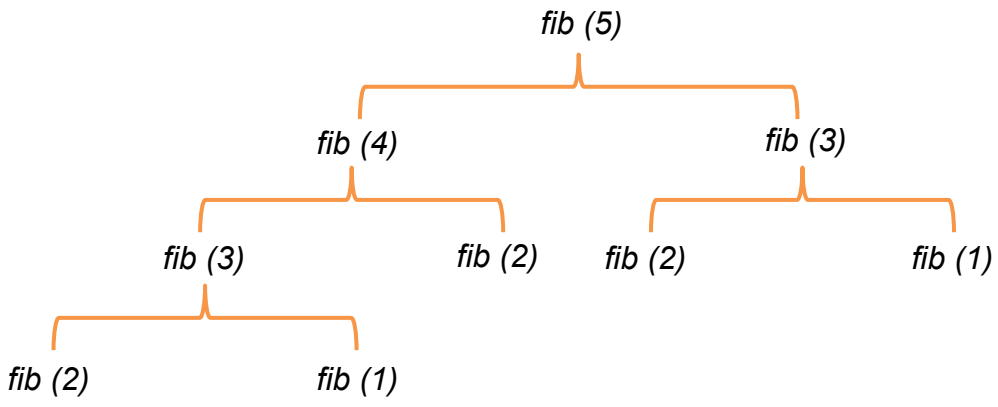
Vejamos agora outro exemplo que envolve a série de Fibonacci implementada recursivamente.

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

#programa princinpal
x = fib(5)
print(x)
```

Note que a função *fib,* ao ser chamada, invoca novamente ela mesma outras duas vezes (linha 6). Dessa maneira, uma só chamada da função *fib* irá resultar em 2^n invocações da função *fib*. Isso nos remete a uma estrutura do tipo árvore binária, e o valor de *n* representa o número de níveis da árvore. Observe o exemplo de *fib(5)*:

Figura 4 – Invocações da função de Fibonacci em árvore binária



Para aplicarmos a equação geral da recursividade, precisamos, primeiro, descobrir a complexidade de uma só chamada recursiva. Será O(1), pois não temos instruções iterativas. A quantidade de nós da árvore será 2^n, pois sempre iremos ramificar cada elemento em até dois outros. Assim, teremos:

$$O(uma\ chamada) = O(1)$$

$$O\left(n^{\underline{o}}\ de\ chamadas^{n^{\underline{o}}\ níveis}\right) = O(2^n)$$

Portanto:

$$O(1).O(2^n) = O(2^n)$$

5.2 Exercícios

Vamos encontrar a complexidade assintótica (*Big-O*) dos seguintes algoritmos escritos em linguagem Python.

Exercício 1:

```
#Exercício 1
def Exercicio1 (dados):
  for i in range(0, len(dados)/2, 1):
    dados[i] = i * 2
```

Solução:

Há um laço único de repetição. Para encontrarmos a complexidade *Big-O*, o que temos dentro do laço não nos interessa. O laço em si inicia em zero e vai até o tamanho da lista, dividido por dois. Ou seja, o *loop* é executado *n/2* vezes. Para o *Big-O*, negligenciamos o valor que está dividindo, assim teremos a complexidade de um laço simples:

$$T(n) = O(n)$$

Exercício 2:

```
#Exercício 2
def Exercicio2 (dados):
  for i in range(0, len(dados), 1):
    dados[i] = i + 1
  for i in range(0, len(dados), 1):
    dados[i] = i - 1
```

Solução:

Temos um laço de repetição e, após ele, outro. Assim, cada laço opera de maneira independente. Isso significa que cada laço representa O(n). Se somarmos ambas as complexidades, teremos O(2n), porém, para o *Big-O*, podemos negligenciar o multiplicador, ficando:

$$T(n) = O(n) + O(n) = O(2n) = O(n)$$

Exercício 3:

```
#Exercício 3
def Exercicio3 (dados):
  for i in range(0, len(dados), 1):
    for j in range(0, len(dados), 1):
      dados[i] = dados[j] + 1
```

Solução:

Temos um laço de repetição e, aninhado a ele, outro. Assim, existe dependência entre os laços. Nesse caso, precisamos verificar se as variáveis envolvidas no laço funcionam como uma PA, ou como uma PG.

O primeiro laço contém a variável *i*, que irá executar uma só vez até o tamanho da lista. Já o laço interno, que contém a variável *j*, irá executar o tamanho da lista para cada execução do laço externo. Se a lista tiver tamanho 10, o primeiro laço executará 10 vezes e, o segundo, 10 vezes 10.

O que mais nos importa aqui é que a cada nova execução do laço contendo a variável *j*, a quantidade de vezes será constante. Isso resultará em uma PA constante, portanto:

$$T(n) = O(n^2)$$

Exercício 4:

```
#Exercício 4
def Exercicio4 (dados):
  for i in range(0, len(dados), 1):
    for j in range(0, len(dados), 1):
      for k in range(0, 9000000, 1):
        dados[i] = dados[j] + 1
```

Solução:

Agora temos 3 laços aninhados. Todos eles executam suas variáveis do laço constantemente (PA). Sendo assim, poderíamos pensar que nossa complexidade seria, com 3 laços aninhados, $O(n^3)$, certo? Na verdade, não.
Note que no terceiro laço, com a variável *k*, a quantidade de iterações independe do conjunto de dados *n*, pois a iteração irá sempre ocorrer 9 milhões de vezes. Portanto:

$$T(n) = n * n * 9000000 = O(9000000.n^2) = O(n^2)$$

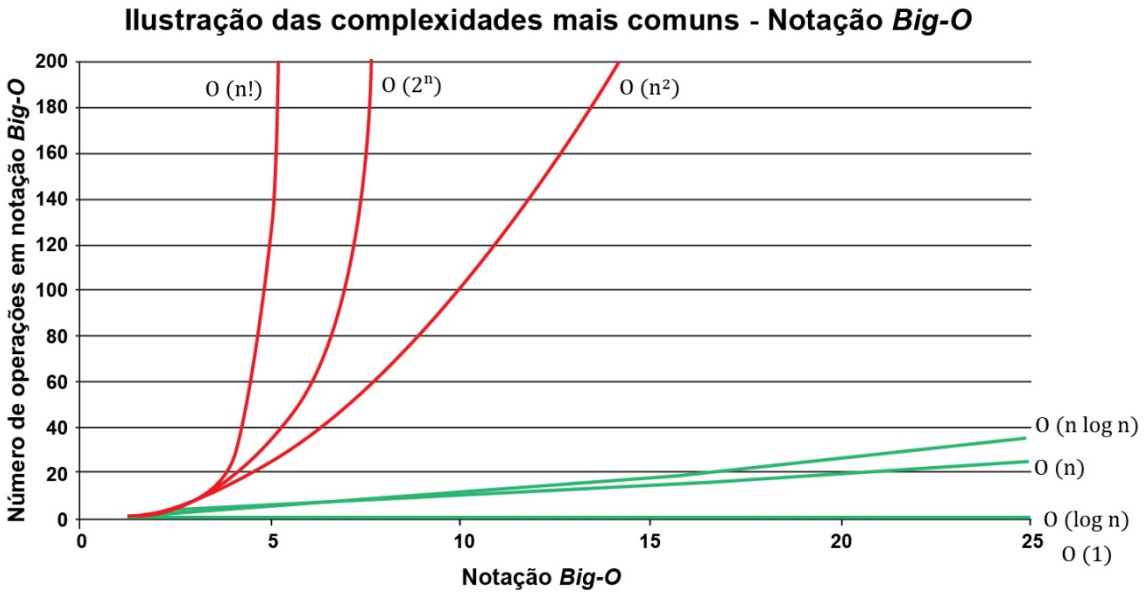
FINALIZANDO

Iniciamos esse estudo aprendendo que diferentes algoritmos para resolver o mesmo problema podem apresentar desempenhos distintos. Estudamos também como calcular o desempenho do algoritmo analisando sua complexidade de tempo de execução. Um algoritmo menos complexo é mais eficiente e, portanto, mais rápido.

Mensuramos a complexidade dos algoritmos pela complexidade assintótica do pior caso desse algoritmo (*Big-O*). Veja agora alguns tipos de algoritmos e suas respectivas complexidades:

- Algoritmo sem iterações nem recursão: O(1);
- Laço iterativo simples: O(n);
- Progressão aritmética (PA): O(n²);
- Progressão geométrica (PG): $O(2^n)$;
- Dividir para conquistar: $O(logn)$;
- Recursão simples: O(n);
- Recursão em árvore binária: $O(2^n)$.

Figura 5 – Principais complexidades de algoritmos



REFERÊNCIAS

DROZDEK, A. **Estrutura de dados e algoritmos em C++**. Tradução da 4. ed. norte-americana. São Paulo: Cengage Learning, 2018.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, abstração, estrutura de dados e projeto usando C++**. Barueri: Grupo GEN, 2008.