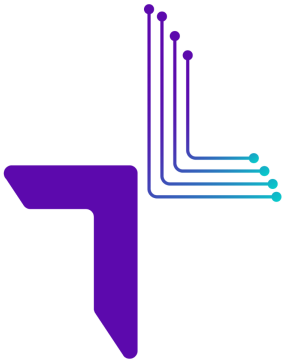
APOSTILA II

PROGRAMAÇÃO ALÉM DO CÓDIGO

REALIZAÇÃO:



Conteúdo

1. Introdução à algoritmos

Bem, antes de partir diretamente para alguma definição ou algum algoritmo em si, é importante entender que esse conceito pode ser aplicado em qualquer sequência de ordens, comandos ou passos que fazemos, independente se está atrelado ou não a questões matemáticas, computacionais ou até mesmo a alguma área de estudo.

Um exemplo que possamos dar e que molda bem a ideia que esse capítulo pretende passar é de que uma simples sequência de ações como acordar, escovar os dentes e tomar café pode ser interpretada não só como um, mas como uma sequência de algoritmos, visto que, entre acordar e escovar os dentes, podemos descrever uma série de passos como abrir os olhos, espreguiçar, se levantar da cama, calçar um par de chinelos, caminhar até a porta do quarto, abrir a porta do quarto, caminhar até o banheiro...

Ou seja, já que um algoritmo, o qual ainda não foi definido, pode ser quebrado em diversas ordens menores e ser tratado como uma sequência entre si, como saberemos quando devemos fazer isso? É com essa questão que eu espero que saiam desse momento inicial, para focá-los especificamente em como entender e aplicar esse conceito nos seus programas.

2. Definindo e interpretando algoritmos

2.1 Definição

Informalmente, um algoritmo pode ser tratado como um procedimento computacional bem definito que toma algum valor ou conjunto de valores como entrada e produz um valor ou um conjunto de valores como saída, sendo assim, uma sequência de etapas computacionais que transformam a entrada numa saída, contudo, nesse curso iremos estudar os algoritmos como forma de resolver problemas, logo, a nossa definição será tratada como: "Uma sequência de passos que nos entrega um resultado", sendo este correto ou não.

O ponto principal é entender que um mesmo algoritmo pode ser construído de diferentes passos, ou outros algoritmos que criam o resultado final, contudo, o uso de uma "lógica"diferente pode resultar em um resultado computacional diferente, seja ele envolvendo a velocidade em que a máquina compila o seu código, ou na alocação de memória necessária para tal.

2.2 Exemplos de algoritmos na solução de problemas mundiais

Agora que existe uma definição, acredito que seja interessante para o leitor deixar a ideia anterior marinando em sua mente, logo, iremos mostrar, superficialmente, como o uso, a compreensão e a aplicação dos algoritmos influencia na nossa sociedade.

Como primeiro caso, irei citar o projeto genoma humano. Esse projeto tinha o objetivo de determinar os pares de bases que compõem o DNA humano e de identificar, mapear e sequenciar todos os genes do genoma humano do ponto de vista físico e funcional. Tal projeto foi iniciado no ano de 1990 e finalizado em 2021, contudo, não pensem que simplesmente deixaram um super computador rodando por 31 anos, pelo contrário, esse é considerado o maior projeto biológico cooperativo do mundo, sendo necessário determinar as sequências dos três bilhões de pares de bases químicas que constroem o DNA humano,

armazenar essas informações em bancos de dados e desenvolver ferramentas para a análise desses dados. Como já citado anteriormente, diferentes algoritmos possuem uma diferente velocidade de compilação e de uso de armazenamento na memória, sendo necessário cada vez mais trabalhar com eficiência, o que explica (em partes) por que um projeto como esse tomou tanto tempo.

2.3 Exemplo de algoritmo mais próximo a nossa realidade

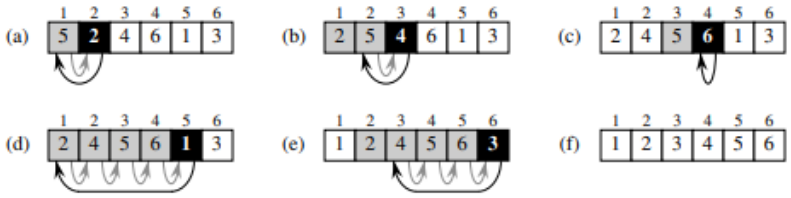
Como segundo caso, podemos citar um exemplo que será trabalhado logo mais, o da ordenação. Imagine que você é um monitor da cadeira de cálculo para a engenharia e o professor lhe entrega uma lista com as notas dos alunos da sala e ele te pede para ordená-las da menor para a maior. Com certeza alguns de vocês já ouviram falar de um ou mais tipos desses algoritmos, seja *bubble*, *selection* ou *insertion sort*, ou talvez você não tenha, mas decide pesquisar na internet sobre como resolver o problema e se depara com dezenas de formas de ordenar uma lista, todas utilizando algoritmos essencialmente diferentes entre si, mas que entregam a mesma entrada e saída. E agora? Como saber qual usar? Bem, essa resposta não será dada agora, mas é importante se manter com a pulga atrás da orelha, pois no próximo capítulo iremos aprender as técnicas que irão nos levar a fazer essa escolha de forma consciente, contudo, por hora, iremos nos contentar em saber que cada algoritmo possui uma lógica diferente e para isso devemos codá-los e testá-los para entender o seu funcionamento e suas diferenças.

2.4 Interpretando um algoritmo

Como dito no título do capítulo, após definirmos e compreendermos o que é um algoritmo, iremos aprender a interpretá-lo, e, para isso, irei utilizar do exemplo de um tipo de ordenação, a ordenação por inserção, um algoritmo eficiente para ordenar um número pequeno de elementos, como seria, no caso acima, uma turma de 60 alunos, ou ordenar as cartas de um baralho. Seguindo a ideia das cartas de um baralho, o algoritmo funciona como a maioria das pessoas iriam ordená-lo, começando com uma pilha vazia e inserindo as cartas na pilha uma por uma na sua posição correta com relação a pilha atual, tal que, sempre estamos ordenando as pequenas parcelas de cartas e reordenando para cada carta seguinte. Tal algoritmo pode ser escrito como:

```
for i in range(1, len(v)):
    chave = v[i]
    j = i - 1
    while j >= 0 and v[j] > chave:
        v[j + 1] = v[j]
        j -= 1
    v[j + 1] = chave
```

Abaixo segue um esquemático do passo a passo de como funciona o algoritmo para a lista [5, 2, 4, 6, 1, 3], retirado do livro Algoritmos: teoria e prática.



Algoritmo de inserção passo a passo.

2.5 Implementação e resolução de problemas

Nas sessões passadas nós definimos o algoritmo, aprendemos a interpretá-lo e agora iremos utilizá-los para solucionar nossos problemas, como o primeiro, que seria construir uma função que nos entrega a soma dos naturais até um certo critério de parada.

Baseado nas aulas passadas, talvez a sua primeira ideia seja de utilizar a função recursiva, ou até mesmo em criar direto um loop, contudo, esta seção está aqui para introduzir uma nova visão de como podemos quebrar um problema em sub rotinas a fim de entender melhor o seu processo.

Antes de mais nada, o algoritmo simples pede que somemos os números, logo, podemos utilizar a seguinte sequência de passos: receber o critério de parada $>$ somar os naturais até o critério $>$ retornar a soma.

Nessa sequência podemos perceber que tanto receber o critério de parada quando retornar a soma são aplicações simples que podem ser feitas de forma direta, logo, devemos pensar em como seria a lógica por trás de somar os naturais até o critério de parada. Podemos usar tanto recursão quanto um loop, mas no caso iremos usar recursão para incentivar o treino dessa técnica:

```
def soma(parada):
    if parada == 1:
        return 1
    else:
        return parada+soma(parada-1)
```

Com esse código temos o resultado que queremos, contudo, como explicado anteriormente, muitos códigos possuem mais de uma maneira de serem resolvidos, no caso, utilizando um artificio matemático conhecido como soma de progressão aritimética, podemos encontrar um valor direto para a soma, sem que a máquina precise realizar cada parcela de cada vez. tal soma é encontrada pelo valor $n(n-1)/2$ a qual nos permite criar o código:

```
def soma(parada):
    soma = parada * (parada - 1) / 2:
        return soma
```

3. Problemas de Algoritmos

Nessa sessão iremos demonstrar diversos problemas em que podemos colocar em prática o que foi ensinado nas sessões anteriores. Peço encarecidamente que o leitor encare cada um desses problemas com a finalidade de tentar resolvê-los antes de partir direto para a solução, visto que além do código haverão instruções para o leitor tentar construir o seu próprio e comparar os resultados.

1) Calcule a soma dos múltiplos de 3 e 5 abaixo de 1000.

Solução:
O passo a passo recomendado para construir essa solução seria:

(a) Criar uma função que retorne os múltiplos de um inteiro até um critério de parada n;
(b) Entender a relação matemática entre os múltiplos e que se o mmc dos números buscados for menor que o critério de parada, então esse valor será calculado duas vezes caso apenas somemos as duas listas;
(c) Retirar, de alguma forma, as dobras e somar as listas.

Código exemplo (lembre-se que o seu código final pode usar outros algoritmos para a solução, sendo este apenas uma das formas):

```
def soma(n, v = []):
    for i in range (n):
        if i % 3 == 0 or i % 5 == 0
            v.append(i)
    return sum(v)
```

2) Faça um programa que devolva a soma dos algoritmos de um inteiro.

Solução:
O passo a passo recomendado para construir essa solução seria:

(a) O loop em python permite ser utilizado não só com valores, mas com valores de uma lista, por exemplo, em uma string de caracteres, poderiamos utilizar "for caractere in string", logo, podemos transformar o número desejado em uma string e ter acesso a cada um dos seus algarismos;
(b) Dentro do loop, transformaremos cara caractere numeral em inteiro novamente e adicionaremos a nossa variável de soma, retornando após ela no final.

Código exemplo (lembre-se que o seu código final pode usar outros algoritmos para a solução, sendo este apenas uma das formas):

```
def soma(n, soma = 0):
    for digito in str(n):
        soma = soma + int(digito)
    return soma
```

3) Faça um programa que devolva a soma do fatorial de 100.

Solução:
O passo a passo recomendado para construir essa solução seria:

(a) Definir a função fatorial, a qual pode ser feita tanto através de um loop, quanto recursivamente, embora haja o encorajamento ao leitor de utilizar a recursão como forma de treino;
(b) Após definida a função fatorial, recomendo utilizar a função criada na questão passada para somar o número em questão.

4) Faça um programa que retorne a diferença entre a soma dos naturais até n ao quadrado e a soma do quadrado dos naturais até n.

Solução:
O passo a passo recomendado para construir essa solução seria:

(a) A primeira parte da solução se encontra resolvida no texto dessa apostila, mostrando a soma dos inteiros até um critério de parada n, sendo necessário apenas elevar esse valor ao quadrado;
(b) A segunda parte seria aplicar um conceito extremamente similar ao anterior mas para a soma $1^2 + 2^2 + ... + n^2$, contudo, é recomendado ao leitor pesquisar sobre a forma direta de resolver esse problema com uma só formula, assim como na soma de progressão aritmética.

4. Introdução à complexidade e análise

Agora que já estudamos a base para a lógica de programação e aprendemos a construir e interpretar algoritmos, iremos aprender a analisá-los de uma forma mais objetiva.

Em diversos momentos durante o curso e a apostila foi citado que diferentes algoritmos que resolvem o mesmo problema podem possuir maior ou menor velocidade computacional, além de alocar mais ou menos memória. Para casos pequenos em geral essa mudança é praticamente inexistente, contudo, quando começamos a explorar situações que necessitam de mais e mais potência da nossa máquina, talvez a solução esteja em otimizar o nosso código.

4.1 Definições

Antes que possamos passar diretamente para a matemática envolvida, é necessário estabelecer algumas definições, sendo elas:

- **Eficiência:** A eficiência de um algoritmo é medida com relação a quantidade de recursos que ele utiliza quando executado. Em geral, a eficiência é calculada baseada no tempo de execução, contudo, existe a relevância da questão do espaço (memória alocada);
- **Análise de algoritmo:** A análise é feita baseada no modelo computacional, sendo em sua maioria, adotado o modelo RAM. No modelo RAM, consideramos um único processador trabalhando em uma máquina de *turing* universal, tal que operações aritméticas básicas, atribuições e comparações possuem tempo constante, ou seja, por contextualização, um tempo T para realizar o seguinte código poderia ser calculado como o seguinte algoritmo:

1. Determinar o conjunto I de instruções dadas a máquina;
2. Determinar o tempo t(i) necessário para realizar cada instrução, tal que i pertence a I;
3. Determinar quantas repetições r(i) ocorrem em cada instrução;

4. Realizar o somatório i = 1 a n (r(i) x t(i))

```
a = 4
b = 7
    if a == b:
        print(True)
    else:
        print(False)
```

Nesse código, temos atribuições, uma comparação e um print (que sabemos que funciona como uma cópia de uma variável, ou seja, atribuição), logo, podemos exprimir que o tempo T(n) do algoritmo pode ser dado por 2 (atribuições) + 1 (comparação) + 1 (print) = 5.

Disso, podemos expandir essa ideia quando tratamos, por exemplo, de um laço, supondo um while ou um for que tenha "n"loops com uma quantidade fixa k de operações básicas, o tempo calculado seria $T(n) = k \cdot n$.

Contudo, mesmo com essa definição, ainda se torna complicado analisar situações que dependam de vários tipos de atribuições, com diversos condicionais, breaks e etc, principalmente quando falamos de casos como os famosos e já citados algoritmos de sort, em que a lista desejada já pode vir na forma ordenada, necessitando de um mínimo poder computacional, ou até mesmo se encontrar da forma mais desordenada possível, logo, nós iremos fazer duas considerações para o cálculo do que iremos chamar de complexidade de um algoritmo:

1) A complexidade de um algoritmo estará condicionada ao termo do polinômio com maior expoente. O motivo disso vem da ideia de que para poucos valores não há necessidade de pensar em otimização, e, nos casos com um número "n"bem alto, podemos puxar do crescimento de funções. Uma ($g(n) = n^2$) cresce muito mais rápido que a função $f(n) = n$, podemos até pensar em limites, se temos dois termos em uma divisão n^m/n^{m+1}, seu limite com n tendendo ao infinito é zero;
2) Como alguns algoritmos possuem diversos(até infinitos) casos, iremos apenas trabalhar com três, o melhor caso, o pior caso e o caso médio. Primordialmente, em casos em que o caso médio pode ser facilmente encontrado, não há problema em trabalhar diretamente com ele, contudo, nem sempre será possível fazer esse cálculo, ou até mesmo ele é, mas apresenta uma extensa prova matemática, o que foge do que estamos procurando. Iremos chamar de Big O (ou O(n)) a notação que representa um limite assintótico acima do "pior caso", ou seja, pensando que um certo algoritmo possua T(n) = $n^2 + 5n + 6$, sabemos que existe c pertencente aos reais tal que $c \cdot n^2 > T(n)$ para todo n, logo, essa será considerada a nossa notação. Além do Big O. Essencialmente, os outros casos, tanto para provar, quanto para uso, não costumam ser muito usados além do Theta, o qual é chamado do limite exato, visto que o O(n) da função passada também poderia ser O(n) = n^k tal que k é natural tal que $k > 1$, disso, podemos considerar sempre que falarmos de Big O, estaremos falando do limite superior mais próximo possível da função original.

4.2 Como utilizar as ferramentas apresentadas

Dado o caminhão de informações que foi passado, basicamente iremos unir as duas definições e usá-las para rapidamente identificar em um algoritmo qual a sua "complexi-

dade", por exemplo, citando qualquer algoritmo que possua dois laços um dentro do outro, independente do que acontece dentro deles(considerando operações básicas), sabemos que um dos loops será condicionado ao um número "n"e a outro número "m", logo, poderíamos falar q a complexidade é O(n,m) = n*m ou, para facilitar, O(n) = n^2 e daí segue.

Para facilitar a fixação, serão dados alguns exemplos, contudo, devo deixar claro que a intenção desse curso não é saber calcular diversos O(n), visto que fora casos simples, a matemática envolvida pode ser extremamente complexa, envolvendo conhecimentos que estão envoltos a cursos como da matemática e de áreas nichadas da ciências da computação, logo, a preparação existe para que ao se deparar com um algoritmo e das suas especificações, saber reconhecer os casos, visto que para qualquer algoritmo já conhecido e determinado, nós já temos catalogado a sua complexidade.

4.3 Exemplos

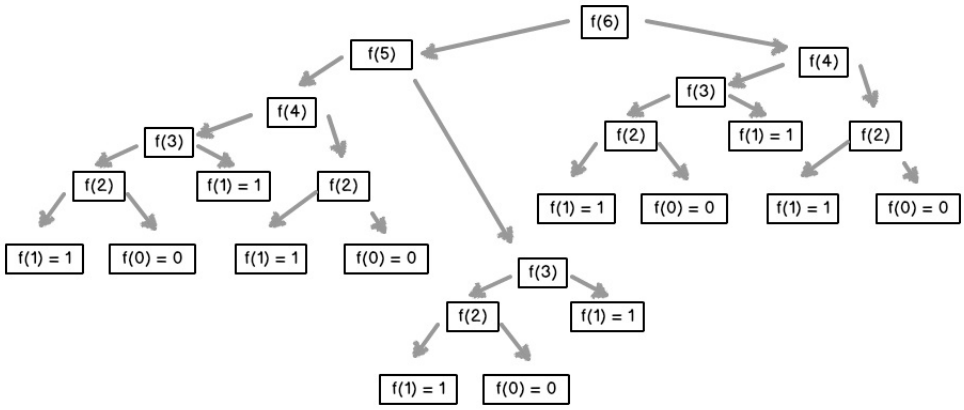
Iremos focar inicialmente em algoritmos de ordenação, especificamente, no insertion sort, bubble sort e selection sort. Como já foi coberta durante o curso e na apostila a aplicação de dois desses algoritmos, insertion e bubble, apenas iremos falar sobre eles sem citar novamente como foram feitos e reutilizando o seu código para sua análise, já o selection, iremos explicar o seu conceito e deixamos para o leitor utilizar os seus conhecimentos e tentar deduzir o seu algoritmo através de uma breve explicação de como ele funciona.

Primeiro, vamos iniciar com o insertion sort. Nele, temos catalogado que sua função "tempo"é dada por $T(n) = 5n^2 + 8n - 11$, na qual, aplicando o conceito de Big O para o melhor caso mais próximo ao algoritmo, temos que ele possui complexidade O(n) = n^2.

```
for i in range(1, len(v)):
    chave = v[i]
    j = i - 1
    while j >= 0 and v[j] > chave:
        v[j + 1] = v[j]
        j -= 1
    v[j + 1] = chave
```

No algoritmo podemos ver que há dois loops interligados um dentro do outro. Disso, com uma rápida análise podemos concluir que o algoritmo possui pelo menos O(n) = n^2. Os outros algoritmos de ordenação servirão como treino na sessão de problemas.

Outra ferramenta importante de ser estudada é como funciona a recursão, e, para isso, devemos nos atentar a uma das mais famosas e perigosas. A esse momento do curso o nome “bad fibonacci” já deve soar conhecido por vocês, a famosa forma de fazer a sequência utilizando um método de recursão direto chamando duas diferentes instâncias da função, como podemos ver na figura a seguir:



Bad fibonacci.

Dela podemos perceber que as instâncias não são "salvas"em memória, precisando diversas vezes encontrar um valor de função como f(2), disso, fazendo as somas das operações básicas e trabalhando com indução matemática, podemos provar que essa sequência recursiva cria um algoritmo com O(n) = $\frac{(1+\sqrt{5})^n}{2}$, crescendo de forma exponencial, diferente do “good fibonacci”, que é equivalente a um loop, possuindo O(n) = n de complexidade.

4.4 Memória

Como citado no começo, o mesmo conceito para velocidade serve para memória, e é aí que vemos todo o brilho em utilizar recursão. Nesse tipo de técnica, nós estamos sempre "puxando"uma função nova a partir da outra, ou seja, apenas realizando operações e não criando ou modificando variáveis(a não ser no caso das pilhas de chamada, o que não precisa ser estudado), o que explica por que vale mais a pena utilizar um good fibonacci recursivo, ou até mesmo fatorial, mesmo eles possuindo a mesma complexidade da sua versão com um loop.

5. Problemas de Complexidade, Análise

Nessa sessão iremos focar em trabalhar com problemas já passados em que o código já foi apresentado, como os da aula passada do módulo de algoritmos.

1) O primeiro problema será para fazer uma comparação entre os dois métodos discutidos de resolver a 4a questão da lista anterior, sobre as somas dos quadrados e o quadrado da soma. Qual a complexidade dos dois?
R = O(n) e O(1).

2) Desenvolva um algoritmo para encontrar os números primos e estude a sua complexidade. Após isso, pesquise as diferentes maneiras de se encontrar um número primo, como o Crivo de Eratóstenes e estude sua complexidade.

3) Conhecendo sobre algoritmos de inserção, sabendo que o bubble sort é um algoritmo que funciona percorrendo uma lista fazendo trocas, caso necessário, entre os números adjacentes, programe o algoritmo estude sua complexidade e em que casos ele pode ser mais interessante que o insertion sort.

4) Conhecendo sobre algoritmos de inserção, sabendo o que o selection sort é um algoritmo que funciona percorrendo uma lista e descobrindo seu menor valor e o ordenando em relação aos demais, no caso, após a primeira iteração, procurando o segundo menor valor e assim por diante, programe o algoritmo e estude sua complexidade, além de em que casos ele pode ser mais ou menos interessante que insertion sort ou bubble sort.

5) Aplicando os conceitos de busca binária apresentada anteriormente, tente deduzir(baseado em como explicamos bad fibonacci) sua complexidade e estude a diferença entre a recursiva e a iterativa.