Aula 2 – Complexidade de algoritmos

No estudo de programação de computadores, uma característica fundamental dos algoritmos que vamos implementar em uma linguagem de programação é o seu custo computacional. Em outras palavras, se temos um problema P e dois algoritmos A_1 e A_2 que resolvem P, um dos critérios mais utilizados na escolha de qual a melhor opção é a complexidade computacional do algoritmo, que em termos práticos é uma aproximação para o número de operações que serão executadas pelo algoritmo quando o tamanho da entrada é igual a n. Tipicamente, o número de operações necessárias para um algoritmo resolver um problema é uma função crescente de n (tamanho do problema), uma vez que quanto maior for a entrada, maior o número de operações necessárias.

Na verdade, durante a análise de complexidade de um programa, estamos interessados em duas propriedades básicas:
i. quantidade de memória utilizada
ii. tempo de execução

Quanto menores as quantidades definidas acima, melhor será um algoritmo, sendo que o tempo de execução na verdade é estimado indiretamente a partir do número de instruções a serem executadas.

A quantidade de memória disponível para um programa pode ser aumentada de maneira relativamente simples, de modo que isso não representa um grande gargalo (podemos duplicar a memória RAM e tudo bem). Além disso, a memória utilizada pode ser estimada observando as variáveis utilizadas pelo algoritmo. Quanto mais variáveis e quanto maiores elas forem, no caso de listas, vetores e matrizes, mais memória elas exigem. Ou seja, antes mesmo da execução de um programa, é possível ter noção de quanta memória ele precisará para executar. Porém, no que diz respeito a tempo de execução, há alguns fatores limitantes, dentre os quais:

a) Como estimar o tempo de execução de um programa antes de executá-lo?
b) O tempo de execução de um programa depende do processador, ou seja, um mesmo programa pode levar 5 segundos em um PC novo e 10 segundos em um laptop antigo e velho.
c) Podemos lidar com um algoritmo que demoraria 50 anos para executar. Como esperar?

Veja o exemplo a seguir:

```
import time

def sum_of_n(n):
     start = time.time()
     the_sum = 0
     for i in range(1, n+1):
          the_sum = the_sum + i
     end = time.time()
     return (the_sum, end-start)
```

Se executarmos a função 5 vezes, passando o valor de n como 10000, ou seja:

```
for i in range(5):
     print("Sum is %d required %.7f seconds" % sum_of_n(10000))
```

teremos como resultado os seguintes valores:

```
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
```

```
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
```

Esse tempo depende diretamente do processador utilizado na execução. O que é certo é que se aumentarmos o valor de n para 1 milhão, ou seja, n = 1000000, o tempo gasto será maior. Portanto, há uma relação entre o tamanho da entrada n e o tempo gasto: isso ocorre porque o número de iterações na repetição aumenta, fazendo com que o número de instruções executadas pelo processador aumente.

```
for i in range(5):
     print("Sum is %d required %.7f seconds" % sum_of_n(1000000))

Sum is 500000500000 required 0.1948988 seconds
Sum is 500000500000 required 0.1850290 seconds
Sum is 500000500000 required 0.1809771 seconds
Sum is 500000500000 required 0.1729250 seconds
Sum is 500000500000 required 0.1646299 seconds
```

Para contornar os problemas citados acima, a análise de algoritmos busca fornecer uma maneira objetiva de estimar o tempo de execução de um programa pelo número de instruções que serão executadas. Dessa forma, quanto maior o número de repetições existentes no programa, maior será o tempo de execução. Claramente, calcular de maneira precisa o exato número de instruções necessárias para um programa pode ser extremamente complicado, ou até mesmo impossível. É aqui que entra a notação Big-O. Conforme mencionado previamente, o número de operações a serem executadas em um programa é uma função do tamanho da entrada, ou seja, f(n). Assim, podemos estudar o comportamento assintótico de tais funções, ou seja, o que ocorre com elas quando n cresce muito (tende a infinito).

Notação Big-O

Ao invés de contar exatamente o número de instruções de um programa, é mais tratável matematicamente analisar a ordem de magnitude da função f(n). Vejamos um simples exemplo com a função definida anteriormente. As instruções mais relevantes para esse tipo de análise são atribuições (=) envolvendo operações aritméticas (+, -, * e /). Note que ao entrar na função temos 1 atribuição. Depois disso, o loop é executado n vezes, pois i inicia com 1 e termina com valor menor que n+1, ou seja, termina em n. A nossa função fica definida como $T(n)=n+1$.

Isso significa que para um problema de tamanho n, essa função executa n + 1 instruções. O que ocorre na prática, é que cientistas da computação verificaram que quando n cresce, apenas uma parte dominante da função é importante. Essa parte dominante é o que motiva a definição na notação O(f(n)). No exemplo da função T(n), no que conforme n cresce o termo 1 torna-se desprezível. Por isso, dizemos que O(T(n)) = n, ou seja, esse é uma algoritmo de ordem linear.

Uma convenção comum na análise de algoritmos é justamente desprezar instruções de inicialização, pois elas ocorrem apenas um número fixo e finito de vezes, não sendo função de n. Assim, a função T(n) também pode ser escrita como T(n) = n. Isso ocorre porque constantes possuem ordem zero, ou seja, nesse caso temos que O(1) = 0.

Para definirmos a notação Big-O, vejamos um outro exemplo. Considere o programa em Python a seguir, em que matrix é uma matriz n x n preenchida com números aleatórios.

```
totalSum = 0
for i in range(n):
      rowSum[i] = 0
      for j in range(n):
            rowSum[i] = rowSum[i] + matrix[i,j]
      totalSum = totalSum + rowSum[i]
```

Podemos opcionalmente a partir de agora desprezar as operações de inicializações, caso necessário, pois como elas são em número constante, não fazem diferença para a ordem de magnitude da função f(n). Vamos calcular o número de instruções a serem executadas por esse programa. Note que no loop mais interno (j) temos n iterações. Assim para cada valor de i no loop mais externo, temos n + 2 operações. Como temos n possíveis valores para i, chegamos em:

$$T(n)=n(n+2)=n^2+2n+1$$

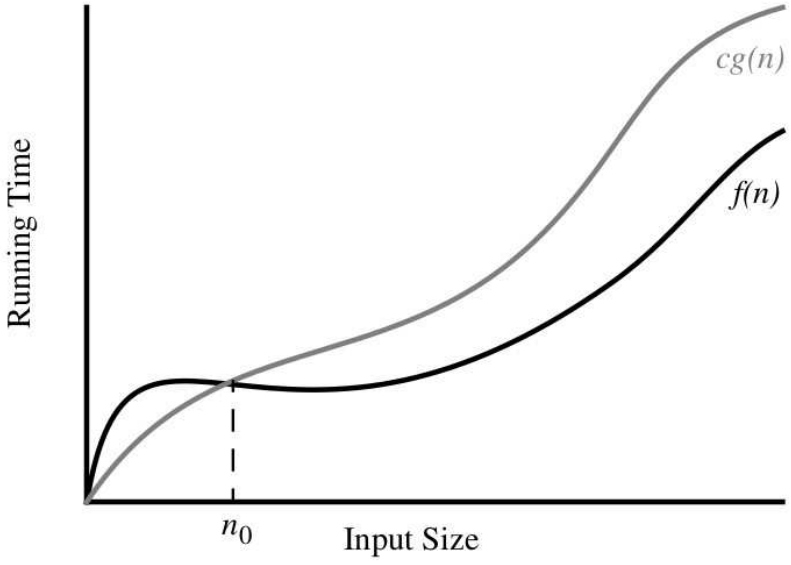
Suponha que exista uma função f(n), definida para todos os inteiros maiores ou iguais a zero, tal que para uma constante c e uma constante m:

$$T(n)\leq c f(n)$$

para todos os valores suficientemente grandes $n\geq m$ (quando n é grande). Então, dizemos que esse algoritmo tem complexidade de O(f(n)). A função f(n) indica a taxa de crescimento na qual a execução de uma algoritmo aumenta, conforme o tamanho da entrada n aumenta. Voltemos ao exemplo anterior. Vimos que $T(n)=n^2+2n+1$. Note que para c = 2 e $f(n)=n^2$, temos:

$n^2+2n+1\leq 2n^2$ para todo n > 2

o que implica em dizer que o algoritmo em questão é $O(n^2)$. A função $f(n)=n^2$ não é a única escolha que satisfaz $T(n)\leq c f(n)$. Por exemplo, poderíamos ter escolhido $f(n)=n^3$, o que nos levaria a dizer que o algoritmo tem complexidade $O(n^3)$. Porém, o objetivo da notação Big-O é o limite superior mais apertado ou justo possível. Trata-se de uma maneira de estudar a taxa de crescimento assintótico de funções.

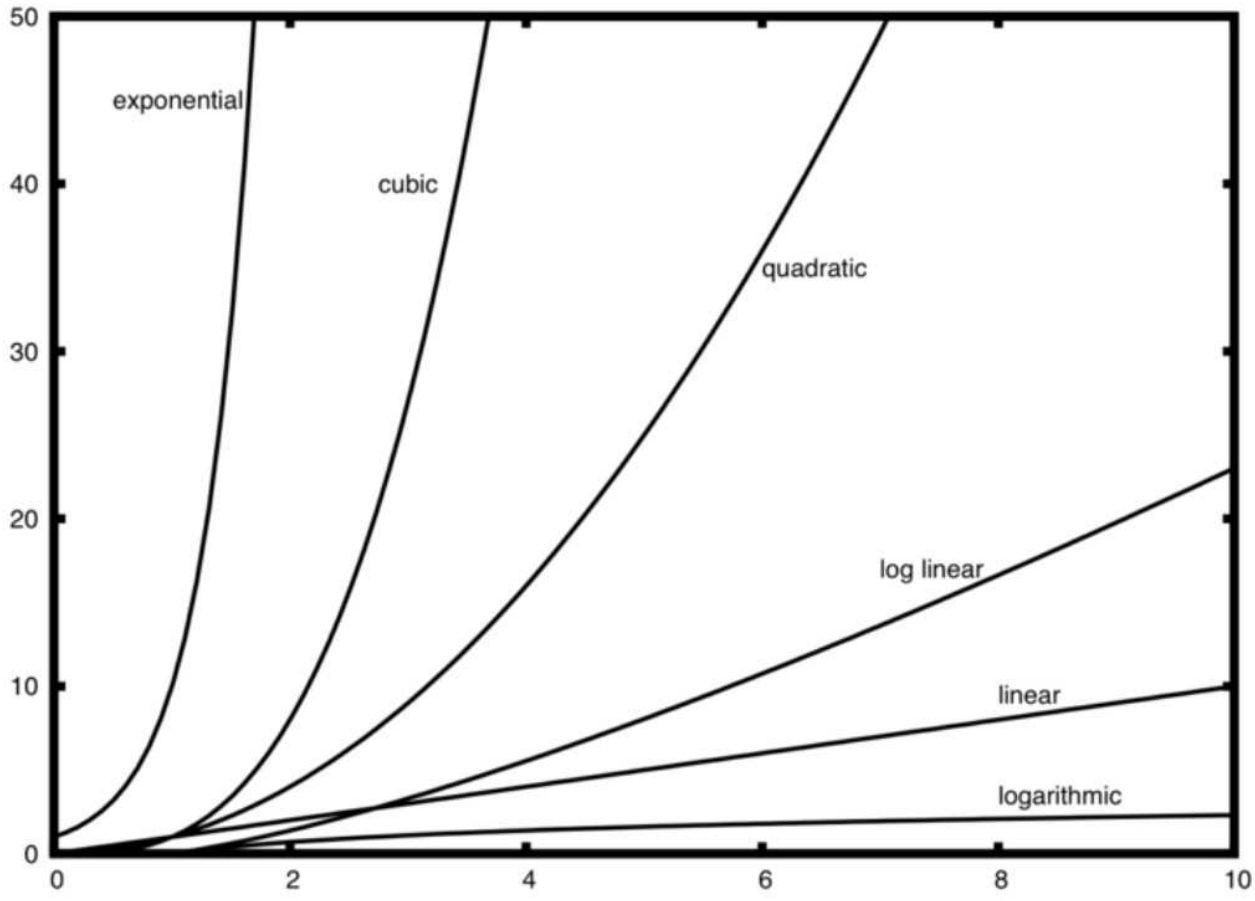


As vezes, o desempenho de um algoritmo não depende apenas do tamanho da entrada, mas também dos elementos que compõem o vetor/matriz. Veremos isso no caso dos algoritmos de ordenação. Se o vetor de entrada está quase ordenado, o algoritmo leva menos tempo, ou seja, é mais rápido. Em

cenários como esse, podemos realizar a análise do algoritmo em três situações distintas: melhor caso (seria o vetor já ordenado), caso médio (valores aleatórios) e pior caso (o vetor em ordem decrescente, totalmente desordenado). Costuma ignorar o melhor caso, pois ele é muito raro de acontecer na prática. Em geral, realizamos as análises no caso médio ou pior caso. Costuma-se dividir os algoritmos nas seguintes classes de complexidade:

f(n)	**Classe**
1	Constante
log n	Logarítmica
n	Linear
n log n	Log-linear
n^2	Quadrática
n^3	Cúbica
2^n	Exponencial

O comportamento assintótico dessas funções é muito diferente. De modo geral, é raro um algoritmo ter complexidade constante (mas há estruturas de dados em que a inserção ou remoção de elementos possui tempo constante), sendo que a classe logarítmica é quase sempre o melhor que podemos obter. Quando um algoritmo é da classe exponencial, ele é praticamente inviável, pois para n = 100, já temos uma valor extremamente elevado de operações, o que seria suficiente para fazer o tempo de execução superar dezenas de anos nos computadores mais rápidos do planeta. A figura a seguir mostra a taxa de crescimento dessas funções.



Construindo T(n)

Ao invés de contar o número exato de comparações lógicas ou operações aritméticas, avaliamos um algoritmo por meio de instruções básica. Para os nossos propósitos, uma instrução básica é toda operação que envolve uma atribuição. Por exemplo, se temos a seguinte linha de código:

```
x = a*b + c*d + e*f
```

então ela toda equivale a uma instrução básica, mesmo envolvendo 3 multiplicações e 2 adições. Assume-se que cada instrução básica possui tempo constante, ou seja, possui O(1). O número total de operações de um algoritmo é dado pela somatória dos tempos individuais $f_i(n) = 1$ sequencialmente, ou seja:

$$T(n)=f_1(n)+f_2(n)+...+f_k(n)$$

Somatórios

Séries e somatórios aparecem com frequência em diversos problemas da matemática e da computação. Na análise de algoritmos é bastante comum termos que resolver somatórios.

Para iniciar com um exemplo simples, suponha que desejamos somar todas as potências de 2, iniciando em 2^1 e terminando em 2^{10}. A maneira explícita de escrever essa soma é:

$$2+2^2+2^3+2^4+...+2^{10}$$

É possível expressar esse somatório como:

$$\sum_{k=1}^{10} 2^k$$

A seguir veremos diversas propriedades úteis na manipulação e resolução de somatórios.

1) Substituição de variáveis

Seja o seguinte somatório:

$$\sum_{k=1}^{n} 2^k$$

Definindo i = k – 1, temos que k = i + 1. Se k inicia em 1, i deve iniciar em zero. Para o limite superior, vale o mesmo. Se k vai até n, i deve ir até n – 1. Dessa forma, podemos expressar o somatório como:

$$\sum_{i=0}^{n-1} 2^{i+1}$$

2) Distributiva: para toda constante c

$$\sum_{k\in A} c\,f(k)=c\left(\sum_{k\in A} f(k)\right)$$

Em outras palavras, é possível mover as constantes para fora do somatório colocando-as em evidência.

3) Associativa: somatórios de somas é igual a somas de somatórios

$$\sum_{k\in A}(f(k)+g(k))=\sum_{k\in A}f(k)+\sum_{k\in A}g(k)$$

4. Decomposição de domínio: Seja A_1 e A_2 uma partição de A. Então,

$$\sum_{k\in A}f(k)=\left[\sum_{k\in A_1}f(k)\right]+\left[\sum_{k\in A_2}f(k)\right]$$

Em outras palavras, podemos decompor o somatório em dois somatórios menores, desde que cada valor de índice apareça no domínio de uma dos subconjuntos. Por exemplo, se A é o conjunto dos naturais, A_1 é o conjunto dos pares e A_2 é o conjunto dos ímpares, todo índice em A_1 não está em A_2.

5. Comutatividade: se p(.) é uma permutação do domínio A, então

$$\sum_{k\in A}f(k)=\sum_{k\in A}f(p(k))$$

ou seja, podemos embaralhar os termos de um somatório que o valor final não muda.

6. Somas telescópicas: considere uma sequencia de números reais $x_1, x_2, x_3, \ldots, x_n, x_{n+1}$. Então, a identidade a seguir é válida:

$$\sum_{k=1}^{n}(x_{k+1}-x_k)=x_{n+1}-x_1$$

ou seja, o valor do somatório das diferenças é igual a diferença entre o último elemento e o primeiro

Prova:

1. Pela propriedade associativa (3), temos:

$$S=\sum_{k=1}^{n}(x_{k+1}-x_k)=\sum_{k=1}^{n}x_{k+1}-\sum_{k=1}^{n}x_k$$

2. Por substituição de variáveis (1), temos:

$$S=\sum_{i=2}^{n+1}x_i-\sum_{k=1}^{n}x_k$$

3. Removendo o último termo do primeiro somatório e o primeiro termo do segundo, temos:

$$S=\sum_{i=2}^{n}x_i+x_{n+1}-x_1-\sum_{k=2}^{n}x_k=x_{n+1}-x_1$$

A prova está concluída.

Analisando códigos em Python

Considere o exemplo a seguir, com duas estruturas de repetição.

```
def ex2(n):
     count = 0
     for i in range(n):
          count += 1
     for j in range(n):
          count += 1
     return count
```

Note que temos uma atribuição inicial (1) e logo dois loops com n iterações. Cada um deles, contribui com n para o total, de modo que no total temos T(n) = 2n + 1, o que resulta em uma complexidade O(n).

Ex: Considere o algoritmo a seguir:

```
def ex3(n):
     count = 0
     for i in range(n):
          for j in range(n):
               count += 1
     return count
```

Nesse caso, o loop interno tem n operações. Como o loop externo é executado n vezes, e temos uma inicialização, o total de operações é $T(n)=n^2+1$, o que resulta em $O(n^2)$.

Note que nem todos os loops aninhados possuem custo quadrático. Considere o código a seguir:

```
def ex3(n):
     count = 0
     for i in range(n):
          for j in range(10):
               count += 1
     return count
```

O loop mais interno é executado 10 vezes (número constante de vezes). Sendo assim, o total de operações é T(n) = 10n + 1, o que resulta em O(n).

Ex: Considere esse código em que o loop interno executa um número variável de vezes.

```
def ex5(n):
     count = 0
     for i in range(n) :
          for j in range(i+1) :
               count += 1
     return count
```

Note que quando i = 0, o loop interno executa uma vez, quando n = 1, o loop interno executa duas vezes, quando n = 2, o loop interno executa 3 vezes, e assim sucessivamente. Assim, o número de vezes que a variável count é incrementada é igual a: 1 + 2 + 3 + 4 + + … n

Devemos resolver esse somatório para calcular a complexidade dessa função.

$$\sum_{k=1}^{n} k$$

Primeiramente, note que

$$(k+1)^2 = k^2 + 2k + 1$$

o que implica em

$$(k+1)^2 - k^2 = 2k + 1$$

Assim, $\sum_{k=1}^{n} [(k+1)^2 - k^2] = \sum_{k=1}^{n} [2k+1]$. Porém, o lado esquerdo é uma soma telescópica e temos:

$$\sum_{k=1}^{n} [(k+1)^2 - k^2] = (n+1)^2 - 1$$

Dessa forma, podemos escrever:

$$(n+1)^2 - 1 = \sum_{k=1}^{n} [2k+1]$$

Aplicando a propriedade associativa, temos:

$$(n+1)^2 - 1 = 2\sum_{k=1}^{n} k + \sum_{k=1}^{n} 1$$

o que nos leva a:

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

Finalmente, colocando n em evidência e dividindo por 2, finalmente temos:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Portanto, T(n) é igual a:

$$T(n) = \frac{1}{2}(n^2 + n) + 1$$

o que resulta em $O(n^2)$.

Ex: Considere a função em Python a seguir.

```
def ex6(n):
     count = 0
     i = n
     while i > 1:
          count += 1
          i = i // 2       # divisão inteira
     return count
```

Essa função calcula quantas vezes o número pode ser dividido por 2. Por exemplo, considere a entrada n = 16. Em cada iteração esse valor será dividido por 2, até que atinja o zero.

16 → 8 → 4 → 2 → 1

A variável count termina a função valendo 4, pois $2^4 = 16$.

Se n = 25, temos:

25 → 12 → 6 → 3 → 1

A variável count termina a função valendo 4, pois $2^4 < 25 < 2^5$

Se n = 40, temos:

40 → 20 → 10 → 5 → 2 → 1

A variável count termina a função valendo 5, pois $2^5 < 40 < 2^6$

Portanto, o número de iterações do loop é $\log_2 n$. Dentro do loop existem duas instruções, portanto neste caso teremos:

$T(n) = 1 + 2\lfloor \log_2 n \rfloor$, onde a função piso(x) retorna o maior inteiro menor que x.

o que resulta em $O(\log_2 n)$.

Ex: Considere a função em Python a seguir.

```
def ex7(n):
     count = 0
     for i in range(n):
          count += ex6(n)
     return count
```

Note que, como a função ex6(n) tem complexidade logarítmica, e o loop tem n iterações, temos que a complexidade da função em questão é $O(n \log_2 n)$.

Ex: Verifique a complexidade do algoritmo em Python a seguir, computando primeiramente o número de instruções a serem executadas.

```
a = 5
b = 6
c = 10
for i in range(n):
     for j in range(n):
```

```
          x = i * i
          y = j * j
          z = i * j
for k in range(n):
     w = a * k + 45
     v = b * b
d = 33
```

Iniciamos com os loops aninhados (i e j), onde temos 3 operações de ordem constante, resultando em $3n^2$ operações, pois são n operações no loop mais interno vezes as n vezes do loop mais externo. No segundo loop (k), temos 2 operações de ordem constante, o que resulta em 2n operações. Por fim, há 4 operações de tempo constante fora dos loops. Sendo assim, temos:

$$T(n)=3n^2+2n+4$$

Quando temos uma expressão polinomial, é fácil perceber que o termo com o maior grau domina os demais. Neste caso, o termo dominante é quadrático, portanto a complexidade do algoritmo em questão é $O(n^2)$.

Ex: Escreva duas funções para encontrar o menor elemento de uma lista, uma que compara cada elemento com cada outro elemento da lista e outra que percorre a lista uma única vez. Calcule a complexidade de cada um deles, analisando o pior caso.

```
def menor_A(L):
     n = len(L)
     for i in range(n):
          x = L[i]
          menor = n*[0]
          for j in range(n):
               if x <= L[j]:
                    menor[j] = 1
          if sum(menor) == n:
               return (i, x)
```

```
def menor_B(L):
     pos = 0
     n = len(L)
     menor = L[pos]
     for i in range(n):
          if L[i] < menor:
               pos = i
               menor = L[i]
     return (pos, menor)
```

a) Vamos analisar o algoritmo menor_A: note que, para cada elemento x da lista L, ele verifica se x é menor ou igual a todos os demais. Ele faz a marcação com o número 1 na posição de x na lista menor. Se x for menor ou igual a todos os elementos de L, teremos exatamente n 1's na lista menor, o que fará com que a soma dos elementos de L seja igual a n.

Pior caso: o menor elemento está na última posição de L

Loop mais interno (j) tem n execuções de um comando
Loop mais externo (i) tem n execuções de 2 comandos e do loop mais interno
Inicialização de n é 1 comandos

Assim, temos:

$T(n) = 1 + n(2 + n) = n^2 + 2n + 1$

o que resulta em $O(n^2)$.

b) Vamos analisar o algoritmo menor_B: note que iniciamos o menor elemento como o primeiro elemento da lista L, Então, percorremos a lista verificando se o elemento atual é menor que atual menor. Se ele for, então atualizamos o menor com esse elemento.

Pior caso: menor elemento está na última posição de L.

Loop tem n execuções com 2 instruções
Inicialização de 3 variáveis

Assim, temos:

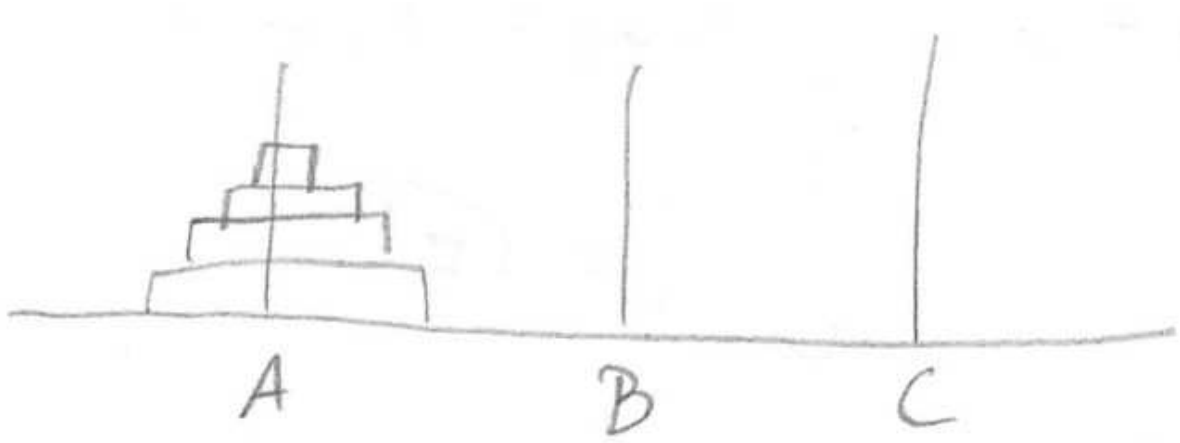
$T(n) = 3 + 2n$

o que resulta em O(n).

Portanto, o algoritmo menor_B é mais eficiente que o algoritmo menor_A.

O problema da torre de Hanói

Imagine que temos 3 hastes (A, B e C) e inicialmente n discos de tamanhos distintos empilhados na haste A, de modo que discos maiores não podem ser colocados acima de discos menores.



O objetivo consiste em mover todos os discos para uma outra haste. Há apenas duas regras:

1. Podemos mover apenas um disco por vez
2. Não pode haver um disco menor embaixo de um disco maior

Vejamos o que ocorre para diferentes valores de n (número de discos).

Se n = 1, basta um movimento: Move A, B

Se n = 2, são necessários 3 movimentos:
Move A, B
Move A, C
Move B, C

Se n = 3, são necessários 7 movimentos:
Move A, B
Move A, C
Move B, C

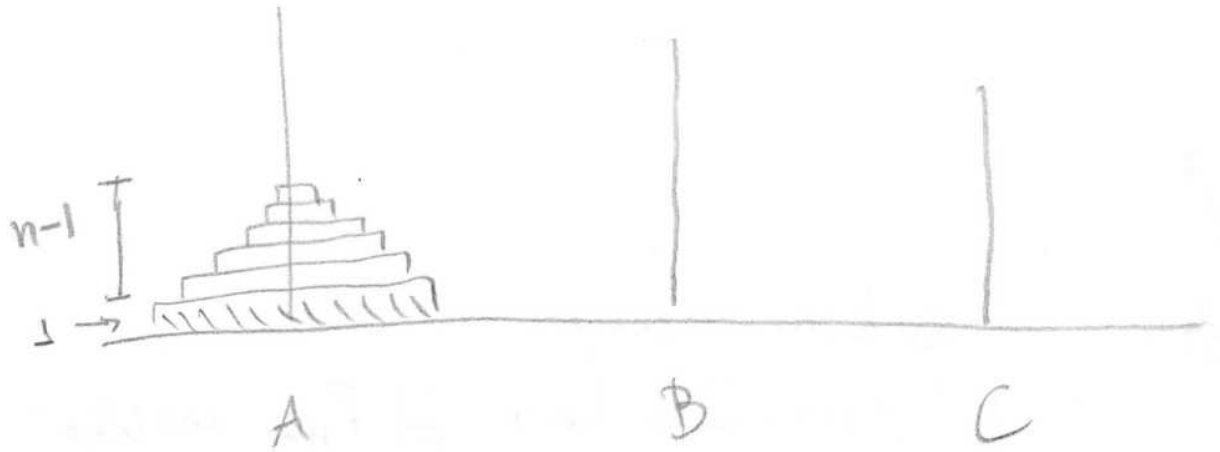
Move A, B
Move C, A
Move C, B
Move A, B

Utilizando uma abordagem recursiva, note que são 3 movimentos para os dois menores discos, 1 para o maior e mais 3 movimentos para os dois menores

Se n = 4, são necessários 15 movimentos: utilizando a abordagem recursiva, temos 7 movimentos para os 3 menores discos, 1 movimento para o maior e mais 7 movimentos para os 3 menores, o que totaliza 7 + 1 + 7 = 15 movimentos

Se n= 5, teremos 15 + 1 + 15 = 31 movimentos

A essa altura deve estar claro que temos a seguinte lógica:



Para mover n – 1 discos menores: T_{n-1} movimentos

Para mover o maior disco: 1 movimento

Para mover de volta os n – 1 discos menores: T_{n-1} movimentos

Assim, a recorrência fica definida como:

$$T_n = 2T_{n-1} + 1$$
$$T_1 = 1$$

onde T_n denota o número de instruções necessárias para resolvermos o problema das n torres de Hanói. Porém, se quisermos descobrir o número de movimentos para n = 100, devemos calcular todos os termos da sequência de 2 até 100.

Pergunta: Como calcular uma função T(n) dada a recorrência?

Como resolver essa recorrência, ou seja, obter uma fórmula fechada? Vamos expandir a recorrência.

$T_1 = 1$
$T_2 = 2T_1 + 1$
$T_3 = 2T_2 + 1$

$T_4 = 2T_3 + 1$
$T_5 = 2T_4 + 1$
$T_6 = 2T_5 + 1$
…
$T_{n-2} = 2T_{n-3} + 1$
$T_{n-1} = 2T_{n-2} + 1$
$T_n = 2T_{n-1} + 1$

A ideia consiste em somar tudo do lado esquerdo e somar tudo do lado direito e utilizar a igualdade para chegar em uma expressão fechada. Porém, gostaríamos que a soma fosse telescópica, para simplificar os cálculos. Partindo de baixo para cima, note que para o termo T_{n-1} ser cancelado, a penúltima equação precisa ser multiplicada por 2. Para que o termo T_{n-2} seja cancelado, a antepenúltima equação precisa ser multiplicada por 2^2. E assim sucessivamente, o que nos leva ao seguinte conjunto de equações:

$2^{n-1}T_1 = 2^{n-1}$
$2^{n-2}T_2 = 2^{n-1}T_1 + 2^{n-2}$
$2^{n-3}T_3 = 2^{n-2}T_2 + 2^{n-3}$
$2^{n-4}T_4 = 2^{n-3}T_3 + 2^{n-4}$
$2^{n-5}T_5 = 2^{n-4}T_4 + 2^{n-5}$
$2^{n-6}T_6 = 2^{n-5}T_5 + 2^{n-6}$
…
$2^2T_{n-2} = 2^3T_{n-3} + 2^2$
$2T_{n-1} = 2^2T_{n-2} + 2$
$T_n = 2T_{n-1} + 1$

Somando todas as linhas, temos uma soma telescópica, pois os mesmos termos aparecem do lado esquerdo e direto das igualdades, o que resulta em:

$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + 2^{n-4} + \ldots + 2^2 + 2^1 + 2^0$

Esse somatório pode ser escrito como:

$$T(n)=\sum_{k=0}^{n-1} 2^k$$

Note que $2^{k+1}=2^k 2$, o que implica em $2^{k+1}=2^k+2^k$, e portanto, $2^k=2^{k+1}-2^k$.

O somatório em questão fica definido por uma soma telescópica:

$$T(n)=\sum_{k=0}^{n-1}\left(2^{k+1}-2^k\right)$$

Pela definição de somas telescópicas, temos:

$$T(n)=\sum_{k=0}^{n-1}\left(2^{k+1}-2^k\right)=2^n-2^0=2^n-1$$

Portanto, esse é a fórmula fechada para o número de movimentos necessários para resolver a torre de Hanói com n discos. Trata-se de um algoritmo exponencial. Por exemplo, se n = 100, o número de movimentos a ser executados é:

126765060022822940149670320537 6

Fazendo um rápido teste, a função a seguir mede o tempo gasto pelo Python para executar uma única instrução:

```
import time

def tempo():
     inicio = time.time()
     x = 1 + 2 + 3
     print(x)
     fim = time.time()
     return(fim-inicio)
```

A execução da função em um processador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz demora cerca de 6.556×10^{-5} segundos. Assim, o tempo estimado para resolver esse problema com um programa em Python seria de aproximadamente:

$126765060022822940149670320537 6 \times 6.556 \times 10^{-5} = 8.310 \times 10^{25}$ segundos

o que é igual a

2.308×10^{22} horas = 9.618×10^{20} dias = 2.63×10^{18} anos

Sabendo que a idade do planeta Terra é estimada em 4.543×10^{9} anos, se esse programa tivesse sua execução iniciada no momento da criação do planeta, estaria executando até hoje. Estima-se que o Big Bang (origem do universo) tenha ocorrido a cerca de 13 bilhões de anos. Isso é menos tempo do que seria necessário para resolver a Torre de Hanói com 100 discos. Por essa razão, dizemos que algoritmos exponenciais são inviáveis computacionalmente, pois eles só podem ser executados para valores muito pequenos de n.

Complexidade de algumas funções Python

Em Python, é muito comum utilizarmos funções nativas da linguagem para manipulas listas, vetores e matrizes. A seguir apresentamos uma lista com algumas das principais funções que operam sobre listas e sua respectiva complexidade.

Operação	**Complexidade**	**Descrição**
L[i] = x	O(1)	Atribuição
L.append(x)	O(1)	Insere no final
L.pop()	O(1)	Remove do final (último elemento)
L.pop(i)	O(n)	Remove da posição i
L.insert(i, x)	O(n)	Insere na posição i
x in L	O(n)	Verifica se x pertence a lista (busca)
L[i:j]	O(k)	Retorna sublista dos elementos de i até j-1 (k)
L.reverse()	O(n)	Inverte a lista
L.sort()	O(n log n)	Ordena os elementos da lista

Busca sequencial x Busca binária

Uma tarefa fundamental na computação consiste em dado uma lista e um valor qualquer, verificar se aquele valor pertence a lista ou não. Essa funcionalidade é usada por exemplo em qualquer sistema que exige o login de um usuário (para verificar se o CPF da pessoa está cadastrada). Faça uma função que, dada uma lista de inteiros L e um número inteiro x, verifique se x está ou não em L. A função deve retornar o índice do elemento (posição) caso ele pertença a ele ou o valor lógico False se ele não pertence a L. (isso equivale ao operador in de Python)

```
def busca_sequencial(L, x):
      achou = False
      i = 0
      while i < len(L) and not achou:
            if (L[i] == x):
                  achou = True
                  pos = i
            else:
                  i = i + 1

      if achou:
            return pos
      else:
            return achou
```

Vamos analisar a complexidade da busca sequencial no pior caso, ou seja, quando o elemento a ser buscado encontra-se na última posição do vetor. Por exemplo,

L = [3, 1, 8, 2, 9, 6, 7] e x = 7

Note que o loop executa n - 1 vezes a instrução de incremento no valor de i e uma vez as duas instruções para atualizar os valores de achou e pos.

$T(n) = 2 + n – 1 + 2 = n + 3$

o que resulta em O(n).

A busca binária requer uma lista ordenada de elementos para funcionar. Ela imita o processo que nós utilizamos para procurar uma palavra no dicionário. Como as palavras estão ordenadas, a ideia é abrir o dicionário mais ou menos no meio. Se a palavra que desejamos inicia com uma letra que vem antes, então nós já descartamos toda a metade final do dicionário (não precisamos procurar lá, pois é certeza que a palavra estará na primeira metade.

No algoritmo, temos uma lista com números ordenados. Basicamente, a ideia consiste em acessar o elemento do meio da lista. Se ele for o que desejamos buscar, a busca se encerra. Caso contrário, se o que desejamos é menor que o elemento do meio, a busca é realizada na metade a esquerda. Senão, a busca é realizada na metade a direita. A seguir mostramos um script em Python que implementa a versão recursiva da busca binária.

```
# Função recursiva (ela chama a si própria)
def binary_search(L, x, ini, fim):
    meio = ini + (fim - ini) // 2
    if ini > fim:
        return -1          # elemento não encontrado
    elif L[meio] == x:
        return meio
    elif L[meio] > x:
        print('Buscar na metade inferior')
        return binary_search(L, x, ini, meio-1)
    else:
        print('Buscar na metade superior')
        return binary_search(L, x, meio+1, fim)
```

Uma comparação enter o pior caso da busca sequencial e da busca binária, mostra a significativa diferença entre os métodos. Na busca sequencial, faremos n acessos para encontrar o valor procurado na última posição. Costuma-se dizer que o custo é O(n) (é da ordem de n, ou seja, linear). Na busca binária, como a cada acesso descartamos metade das amostras restantes. Supondo, por motivos de simplificação, que o tamanho do vetor n é uma potência de 2, ou seja, $n = 2^m$, note que:

Acessos		**Descartados**
m = 1	→	n/2
m = 2	→	n/4
m = 3	→	n/8
m = 4	→	n/16

e assim sucessivamente. É possível notar um padrão?

Quantos acessos devemos realizar para que descartemos todo o vetor? Devemos ter $n / 2^m = 1$, o que significa ter $n = 2^m$, o que implica em $m = \log_2 n$, ou seja, temos um custo $O(\log_2 n)$ o que é bem menor do que n quando n cresce muito, pois a função log(n) tem uma curva de crescimento bem mais lento do que a função linear n. Veja que a derivada (taxa de variação) da função linear n é constante e igual a 1 sempre. A derivada da função log(n) é 1/n, ou seja, quando n cresce, a taxa de variação, que é o que controla o crescimento da função, decresce.

Na prática, isso significa que em uma lista com 1024 elementos, a busca sequencial fará no pior caso 1023 acessos até encontrar o elemento desejado. Na busca binária, serão necessários apenas $\log_2 1024 = 10$ acessos, o que corresponde a aproximadamente 1% do necessário na busca sequencial! É uma ganho muito grande.

Porém, na busca binária precisamos gastar um tempo para ordenar a lista! Para isso precisaremos de algoritmos de ordenação, o que é o assunto da nossa próxima aula. Bons estudos e até mais.