Introdução

- O que é recorrência?

  É uma **descrição recursiva** de uma função, ou em outras palavras, uma descrição de uma função em termos de si mesmo. Como todas as estruturas recursivas, uma recorrência consiste em **um ou mais casos bases (critério de parada)** e **um ou mais casos recursivos**.

$$T(n) = \begin{cases} T(n-1)+1, & n > 0 \\ 1 \quad\quad , & n = 0 \end{cases}$$

- Com base na função T(n) acima, define-se:
  - **Caso base**: é o critério de parada da função de recorrência (toda função recursiva tem um critério de parada); neste caso, o custo será 1 quando n = 0 => T(0) = 1;
  - **Caso recursivo**: é a chamada recursiva da função. Observa-se que há uma chamada T(n – 1), ou seja, a chamada para a própria função T.

- Outro ponto importante é que a função acima define o custo de execução quando da chamada recursiva, neste caso, custo 1.

Quais são os Métodos/Técnicas?

- Há diversos métodos que resolvem as relações de recorrências. A ordem abaixo reflete a "preferência" da literatura:

- Método Backward Substitution (ou Derivation)
  - Supõe-se a forma da solução: expande, substitui, generaliza e encontra as constantes e gera a função.

- Método Recursion Tree
  - É uma árvore em que cada nó representa um custo de um certo subproblema recursivo. Soma-se os custos de cada nó para obter o custo de todo problema, generalizando a solução.

- Método Master Theory
  - É uma fórmula para resolver recorrências da forma $T(n) = aT(n/b) + f(n)$, onde $a >= 1$ e $b > 1$, e $f(n)$ é assintoticamente positivo.

- Método de Contagem (para funções iterativas)

Indução: pode-se provar por indução se a função complexidade de tempo encontrada nos métodos acima representa de forma correta a recorrência.

Exemplo – "Torres de Hanoi"

Método de contagem

- Reconhecer a função de recorrência a partir do código:

```
public class Hanoi {
   public static void main(String[] args) {
      hanoi(4, "A", "B", "C");
   }
   public static void hanoi(int n, String start, String auxiliary,  String end) {
      if (n > 0) {
         hanoi(n - 1, start, end, auxiliary);
         System.out.println(start + " -> " + end);
         hanoi(n - 1, auxiliary, start, end);
      }
   }
}
```

```
T(n)
1
T(n-1)
1
T(n-1)
```

- Montar a função de recorrência:

$$T(n) = \begin{cases} 2T(n-1) + 1, & n > 0 \\ 1 \quad , & n = 0 \end{cases}$$

- Alguns custos (aritmético, por exemplo) foram omitidos para simplificar a análise de complexidade e por não influenciarem na ordem assintótica, pois são constantes. Ainda, o correto seria 2T(n - 1) + 2, porém, foi simplificado para 2T(n - 1) + 1.

Hanoi (cont.)

$$T(n) = \begin{cases} 2T(n-1)+1, n > 0 \\ 1 \qquad\qquad\quad , n = 0 \end{cases}$$

3. Calculando a complexidade:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ T(n-1) &= 2T(n-2)+1 \\ T(n-2) &= 2T(n-3)+1 \\ T(n-3) &= 2T(n-4)+1 \\ &\cdots \end{aligned}$$

4. Técnica **backward substitution:**

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

$$T(n) = 2(2T(n-2)+1)+1 \Rightarrow 2^2T(n-2)+2+1$$

$$T(n) = 4(2T(n-3)+1)+2+1 \Rightarrow 2^3T(n-3)+4+2+1$$

$$T(n) = 8(2T(n-4)+1)+4+2+1 \Rightarrow 2^4T(n-4)+8+4+2+1$$

...

Olha o "k" aparecendo

5. Generalização:

$$T(n) = 2^kT(n-k)+2^{k-1}+2^{k-2}+\cdots+2^2+2^1+1$$

Hanoi (cont.)

$$T(n) = \begin{cases} 2T(n-1)+1, n>0 \\ 1 \qquad\qquad\quad , n=0 \end{cases}$$

6. Assume-se que n – k = 0, logo, **k = n.**

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

$$T(n) = 2^k T(0) + \underbrace{2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2^1 + 1}_{2^k - 1}$$

Progressão Geométrica!
Observe que o intervalo é k-1 a 0. Por isso, 2^k-1. Original é $2^{k+1} - 1$

$$T(n) = 2^k * 1 + 2^k - 1$$

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

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

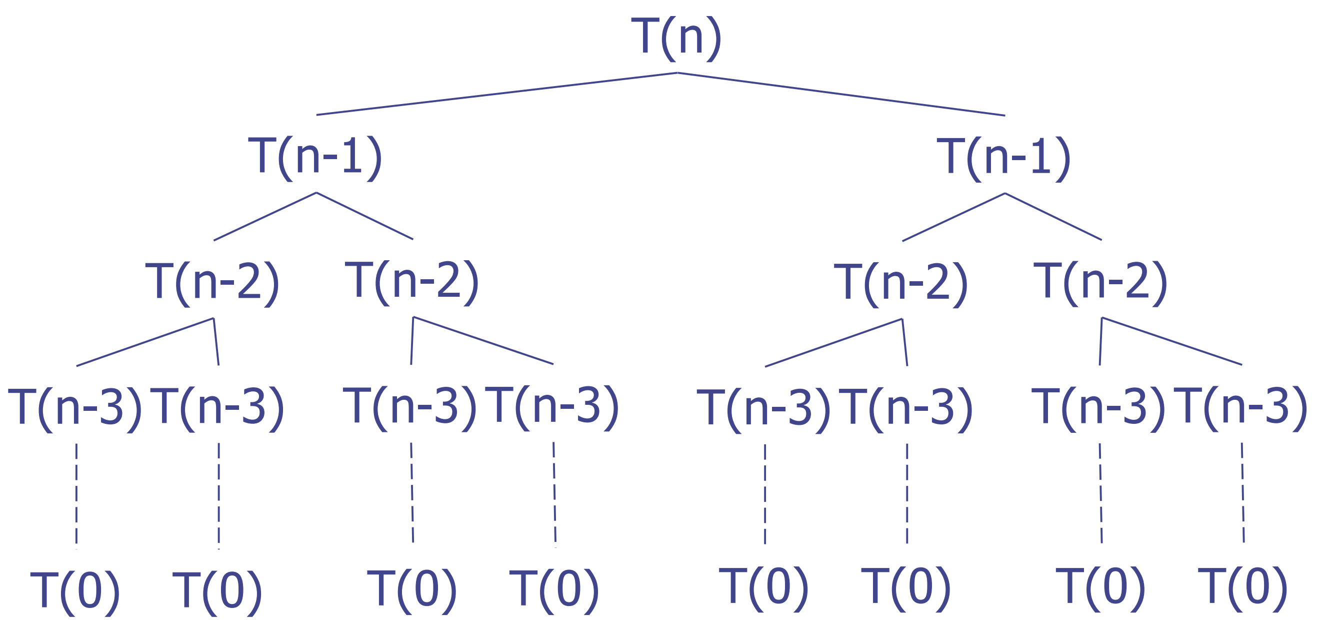
Objetivo é chegar aqui!

Logo, o comportamento assintótico é exponencial, **$O(2^n)$**.

Hanoi - Recursive Tree

$$T(n) = \begin{cases} 2T(n-1) + 1, n > 0 \\ 1 \quad\quad\quad\quad\quad , n = 0 \end{cases}$$

- Como calcular a complexidade usando a técnica **Recursive Tree. Ótimo exemplo de uso da técnica (duas chamadas recursivas).**



custo 1 = 2^0

custo 2 = 2^1

custo 4 = 2^2

custo 8 = 2^3

custo 2^{k-1}

custo 2^k

$$\underbrace{\quad\quad\quad}_{2^{k+1} - 1}$$

Olha o "k" aparecendo

- Assume-se que n – k = 0, logo, **k = n.**

$$2^{n+1} - 1$$

Logo, o comportamento assintótico é exponencial, **$O(2^n)$**.

Hanoi – Prova baseada na execução

Como os movimentos são $\mathbf{2^n - 1}$, pode-se fazer uma prova baseada na execução do algoritmo. Observe que há mais um $\mathbf{2^n}$ que é a execução para quando o n = 0 (sem movimento). Ex.:

discos = 3 => 7 movimentos	**discos = 4 => 15 movimentos**
A -> C A -> B C -> B A -> C B -> A B -> C A -> C	A -> B A -> C B -> C A -> B C -> A C -> B A -> B A -> C B -> C B -> A C -> A B -> C A -> B A -> C B -> C

Simplificações Matemáticas

Progressão Aritmética

$$\frac{n(n+1)}{2} = 1 + 2 + 3, \dots, n - 1 + n$$

$$\Downarrow$$

$$\frac{n^2 + n}{2}$$

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

$a_1 = primeiro\ termo$

$a_n = termo\ geral$

$n = total\ de\ termos$

$$\log(n!) = \log(1) + \log(2) + \log(3) + \cdots + \log(n-1) + \log(n)$$
$$\log(n!) \leq \log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n)$$
$$\log(n!) \leq \mathrm{nlog(n)}$$

Progressão Geométrica

$$2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

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

$a_1 = primeiro\ termo$

$q = quociente$

$n = total\ de\ termos$