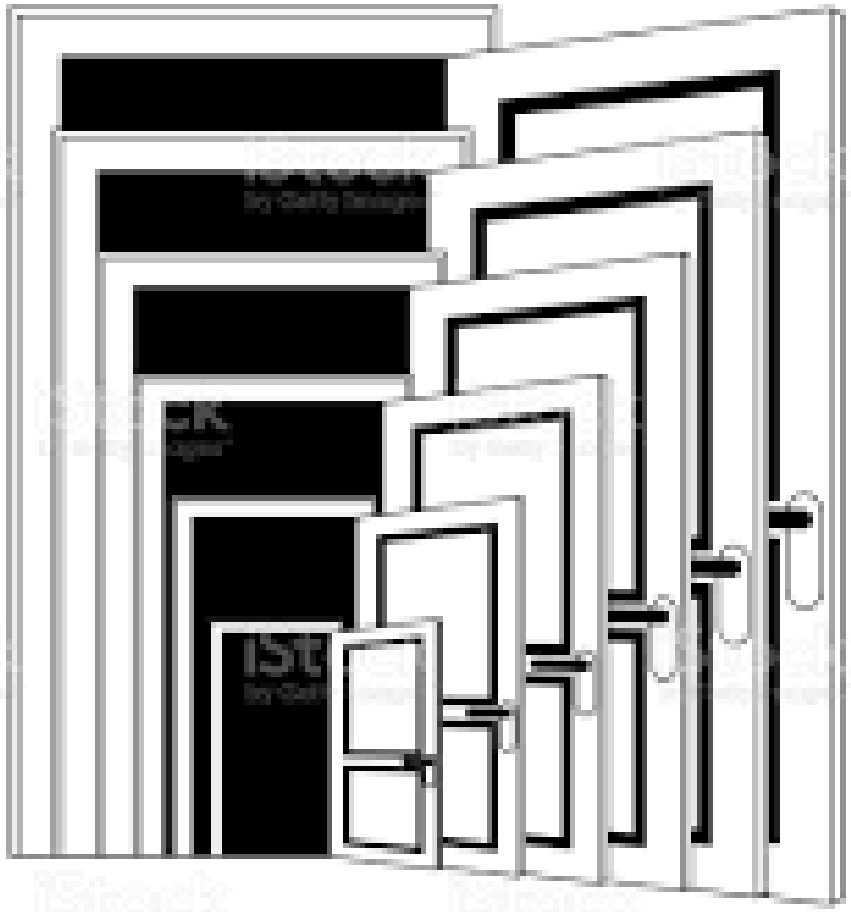
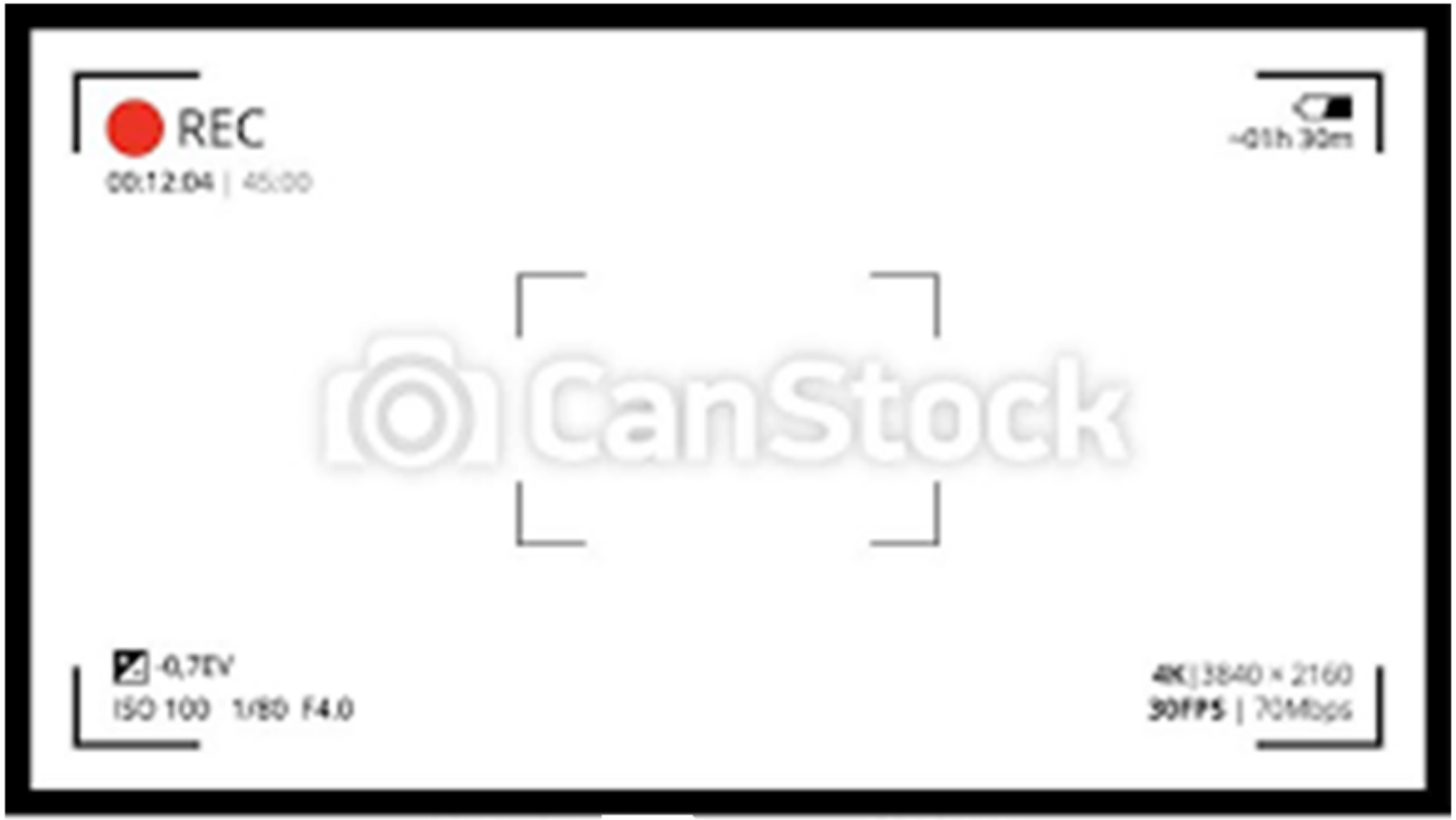
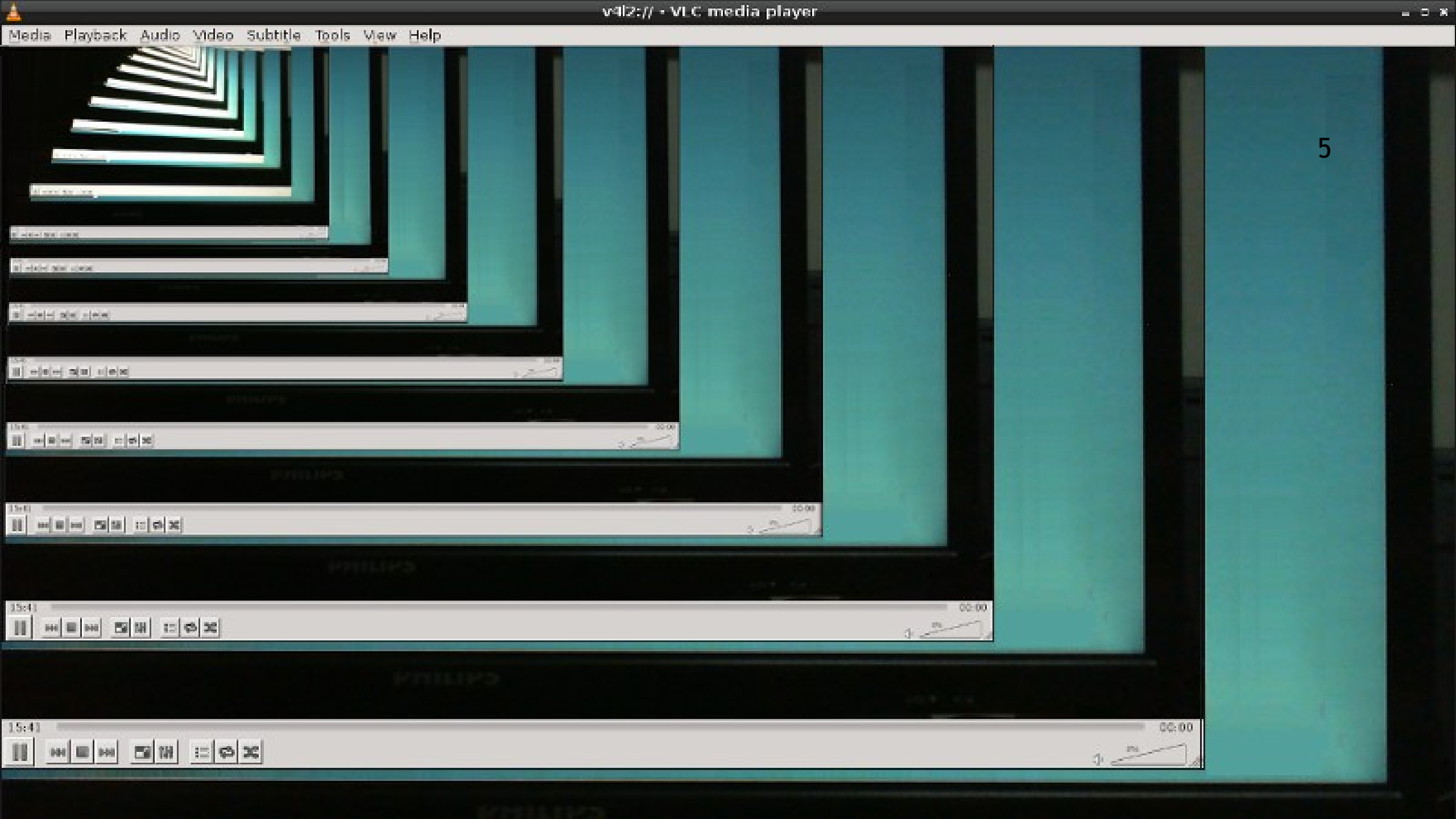
recorrência/recursividade


Recursão
Recursão
Recursão
Recursão
Recursão
Recursão




REC
00:12:04 | 45:00
~01h 30m
-0,7EV
ISO 100 1/80 F4.0
4K | 3840 × 2160
30FPS | 70Mbps


v4l2:// - VLC media player
Media Playback Audio Video Subtitle Tools View Help
5

Definição recorrente

Como definir algo em torno de si mesmo?

1. Uma base, ou condição básica, onde alguns casos simples (pelo menos um) do item que está sendo definido são dados explicitamente.

2. Um passo de indução ou recorrência, onde novos casos do item está sendo definido são dados em função dos casos anteriores.

- O item 1 nos dá o começo, fornecendo casos simples e concretos.
- O item 2 nos permite construir novos casos, a partir dos simples e ainda outros casos a partir desses novos e assim por diante.

- SEQUÊNCIA RECURSIVA:

É uma lista de objetos (uma sequência), enumerados segundo uma ordem S(k) que denota o k-ésimo elemento da sequência.

Uma sequência recursiva explicita seu primeiro valor (ou primeiros valores) e define outros valores na sequência em termos dos valores iniciais.

EXEMPLO:

Sequência definida recursivamente

1. S(1) = 2
2. S(n) = 2S(n1) para n ≥ 2

O primeiro valor da sequência é explicitamente dado, S(1) = 2, os valores seguintes são sucessivamente calculados com base na definição recursiva:

- S(2) = 2S(1) = 2*2 = 4
- S(3) = 2S(2) = 2*4 = 8
- S(4) = 2S(3) = 2*8 = 16

Escreva os primeiros 5 valores da Sequência T definida por:

1. T(1) = 1 .
2. T(n) = T(n1)+3 para n ≥ 2

Solução

- $T(1) = 1$
- $T(2) = T(1) + 3 = 1 + 3 = 4$
- $T(3) = T(2) + 3 = 4 + 3 = 7$
- $T(4) = T(3) + 3 = 7 + 3 = 10$
- $T(5) = T(4) + 3 = 10 + 3 = 13$

ENTÃO:

RELAÇÃO DE RECORRÊNCIA

É uma regra que define um valor da sequência a partir de um ou mais valores anteriores.

- A sequência de Fibonacci pode ser definida recursivamente como se segue:

F(1) = 1

F(2) = 2

F(3) = F(n2) + F(n1) para n > 2

F(3) = 1 + 2 = 3

F(4) = F(n2) + F(n3) = 2 + 3 = 5

F(5) = F(n3) + F(n4) = 3 + 5 = 8

Na Ciência da Computação

⇩

recursividade

⇩

é uma sub-rotina

⇩

que se invoca durante

⇩

processo de execução de um programa.

ESTE TIPO DE ALGORITMO PODE SER EXTREMAMENTE PODEROSO EM ALGUNS CENÁRIOS, MAS EM OUTROS NEM TANTO, PODENDO SER CONSIDERADO ATÉ INEFICIENTE.

- Um exemplo muito claro do que é a recursão, é quando estudamos na matemática o cálculo do fatorial de um número qualquer. Exemplo:

factorial(1) = 1

factorial(2) = 2 * 1

factorial(3) = 3 * 2 * 1

*factorial(4) = 4 * 3 * 2 * 1*

Se pararmos para analisar, a fórmula para este processo sempre será a seguinte:

factorial(n) = n * factorial(n-1)

- Vemos que a função chama à ela mesma, mas agora passando o valor de ***n - 1***, para poder calcular o valor do fatorial de ***n***.
- Logo, se quiséssemos calcular o fatorial de 4, precisaríamos antes saber o valor do fatorial de 3 e assim por diante.

- Se tentarmos desenvolver o algoritmo do cálculo do fatorial de um número em uma determinada linguagem de programação, você perceberá que não necessariamente existe apenas uma maneira de desenvolver esse algoritmo.

Mas sim por dois caminhos, um de maneira recursiva e outro de maneira iterativa.

- Primeiro, vamos discutir como esse algoritmo se comportaria de maneira **recursiva**:

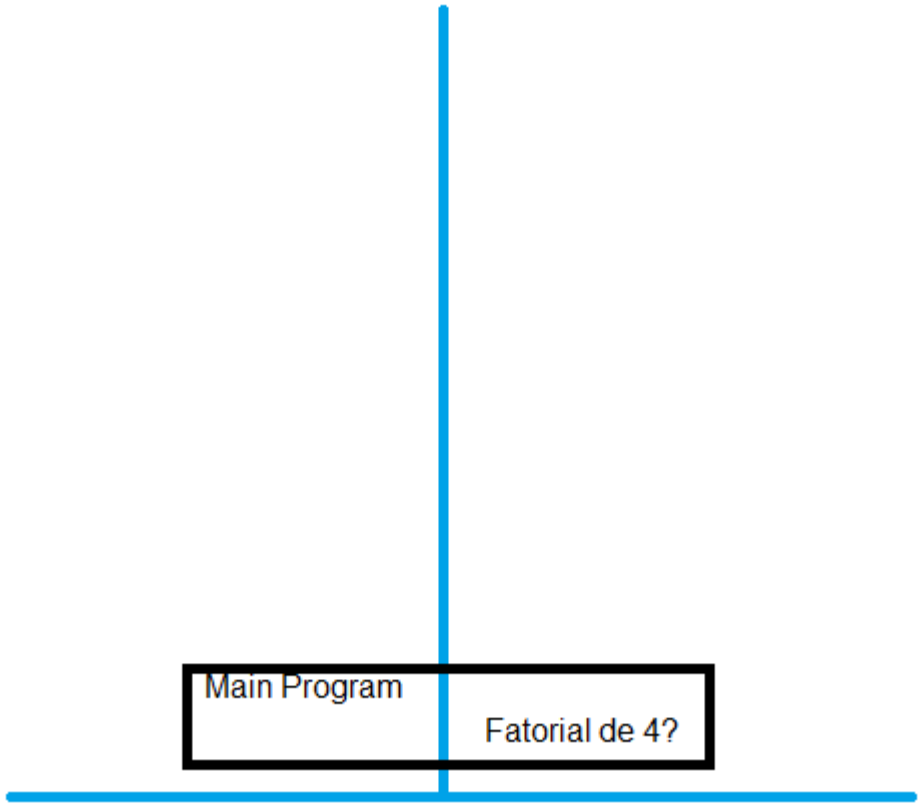
```
int factorial (int n) {
if (n == 1) return 1;
else
 return n * factorial(n-1);
}
```

Conforme vemos acima, quando retornamos a chamada da função ***factorial(n)***, estamos voltando para o mesmo pedaço de código.

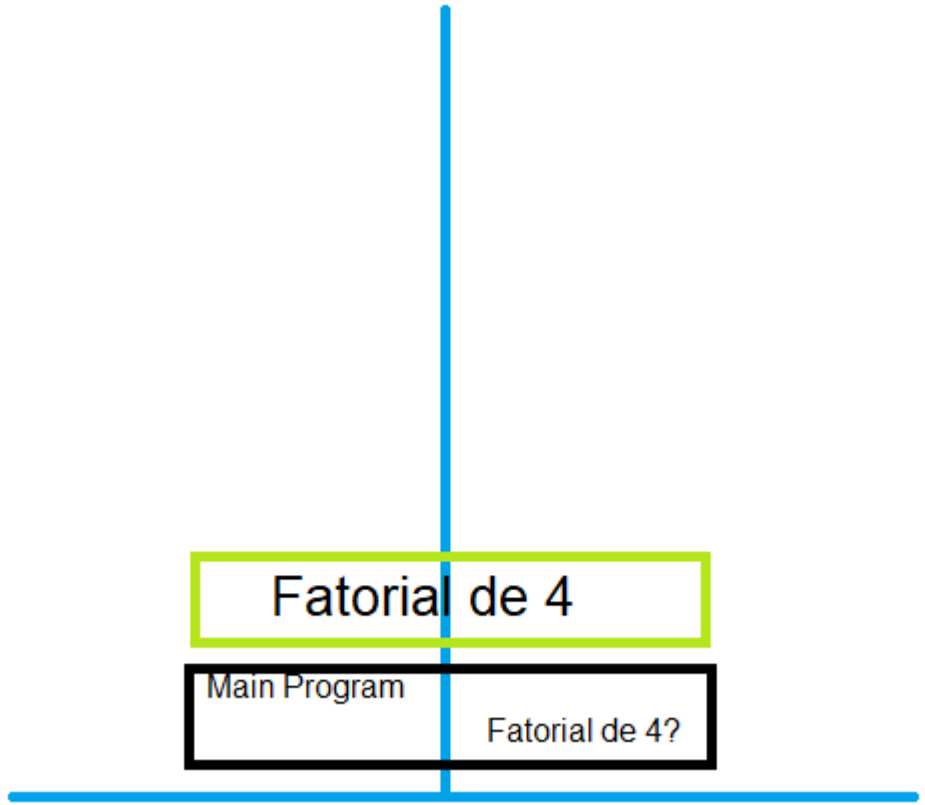
- Logo, chamamos o parâmetro ***n*** diversas vezes, mas em todas ***n*** vezes sempre será diferente. Então teremos que manusear e separar esse parâmetro toda vez que chamarmos a função.

Assim, em um algoritmo como este, não temos apenas um ***n*** como parâmetro sendo armazenado na memória.
E para podermos manusear e separar apropriadamente, precisamos então de uma **Stack** (pilha).

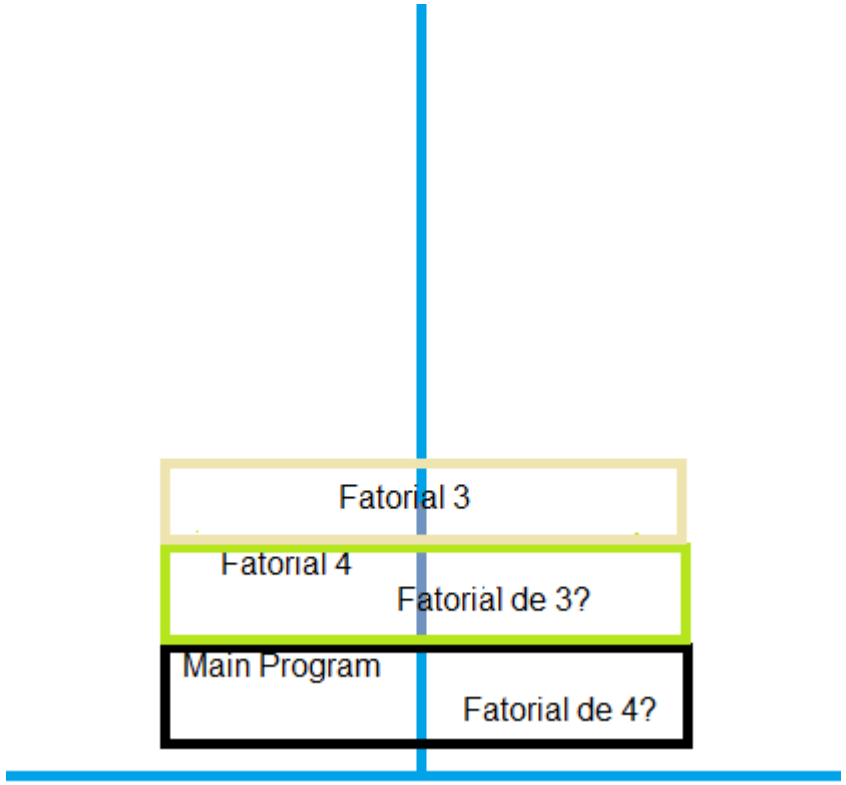
- Imagine que vamos calcular o fatorial do número 4. A primeira coisa que irá acontecer vai ser mandarmos esse algoritmo para nossa pilha.



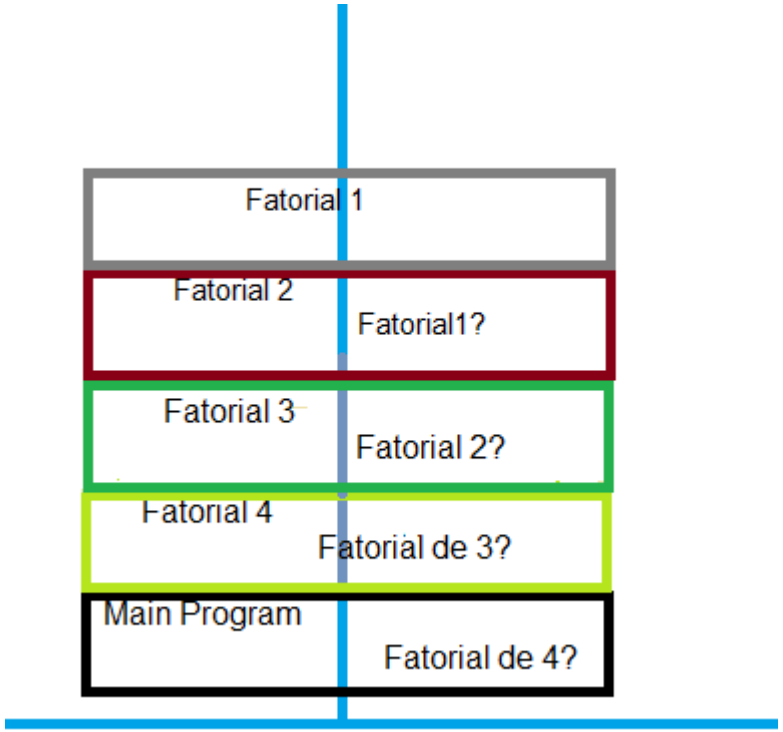
Para isto funcionar corretamente e ter suas próprias variáveis locais, o que inclui todas as diferentes instâncias de ***n***, nós mandamos outro frame para uma outra área da nossa pilha, que será a resposta do fatorial de 4.



Mas o fatorial de 4 exige que tenhamos o resultado do fatorial de 3, caso contrário não é possível obter o valor de 4.



Se seguirmos o mesmo processo para os demais números, ao final nossa pilha estará da seguinte maneira:



Agora, sabemos que o fatorial de 1 é ele mesmo, então o fatorial de 2 que estava aguardando o resultado pode prosseguir com o cálculo, retornando o valor do fatorial de 2 podemos prosseguir com o cálculo do fatorial de 3, e assim por diante, até chegarmos no programa principal com o resultado do **fatorial de 4**, que corresponde ao número **24**.

O que percebemos, é que nessa sequência de múltiplas pendências, cada uma dessas está associada a um valor diferente de ***N***.

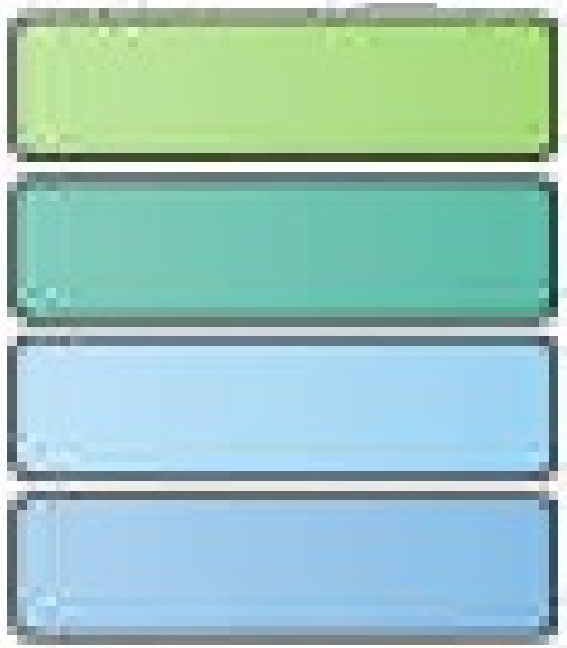
Então quando resolvemos o fatorial de 1, as respostas começam a ser cascatiadas para as demais funções que estavam aguardando na stack.

Veja o comportamento desta função.

```
(factorial 4)
(* 4 (factorial 3))
(* 4 (* 3 (factorial 2)))
(* 4 (* 3 (* 2 (factorial 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

Se notarmos, podemos ver que o fluxo tem um comportamento gráfico de *"ida e volta"*, ou seja, temos as múltiplas pendências e o retorno dos resultados dessas múltiplas pendências.

- Isto está relacionado com a necessidade de memória, pois a medida que vamos fazendo novas chamadas, e aguardamos o retorno dessas chamadas para continuar a computação, precisamos fazer empilhamentos na Stack, conforme vimos anteriormente.



- Agora como faríamos esse algoritmo de maneira **iterativa**?

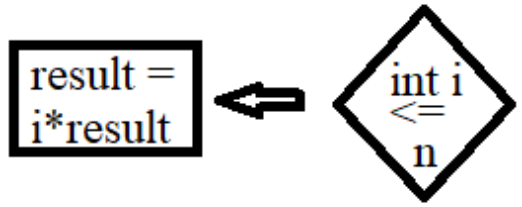
Veja o código abaixo:

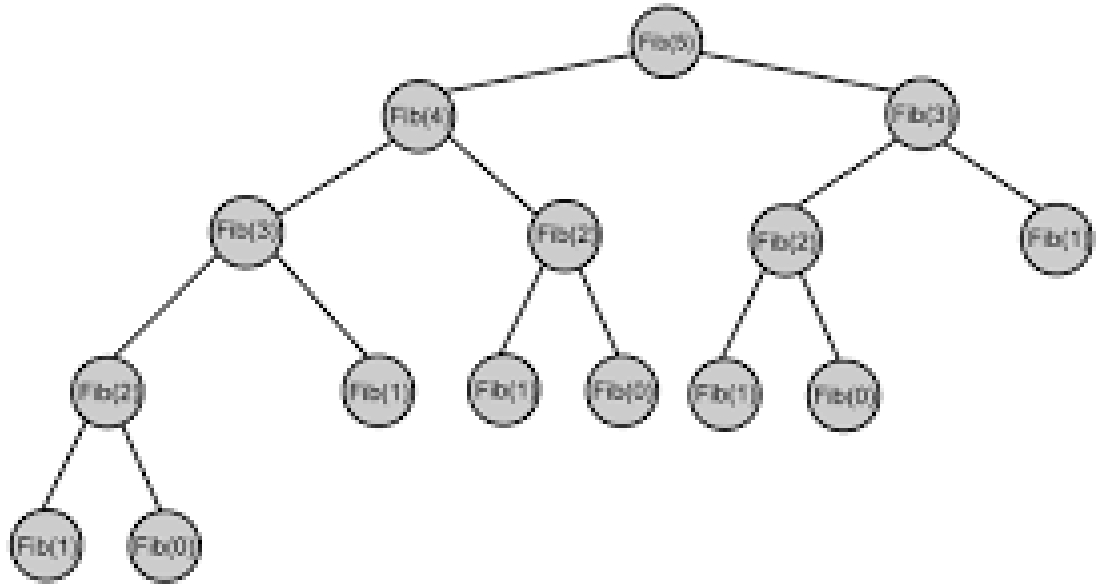
```
int factorial (int n) {
 int result = 1;
 for (int i = 1; i <= n; i++) {
    result = i * result;
 }
}
```

```
int factorial (int n) {
 int result = 1;
 for (int i = 1; i <= n; i++) {
    result = i * result;
 }
}
```

O que fizemos aqui foi primeiro criar uma variável que inicia com o valor de 1 (esta que representa nosso resultado final) e depois criar um loop que a cada volta nós armazenamos na nossa variável ***result*** o valor do contador multiplicado pelo valor dela mesma. Ao final do algoritmo é retornado o valor da variável ***result,*** que no caso é o nosso resultado.

- Com este tipo de procedimento, nós não temos a necessidade de *"ir e voltar"*, pois não tem computação sendo executada no retorno do processo.
- Logo, não temos necessidade de voltar todo o empilhamento feito na Stack.





Recursão em Árvore

- A recursão em árvore, é um outro tipo de shape (forma) de execução que encontramos em alguns algoritmos recursivos. Ela é muita conhecida por seu fluxo ser moldado em uma estrutura de árvore.

Para exemplificar, vamos utilizar um algoritmo que nos retorna um número que pertence à sequência de Fibonacci e está na posição correspondente ao valor do nosso parâmetro:

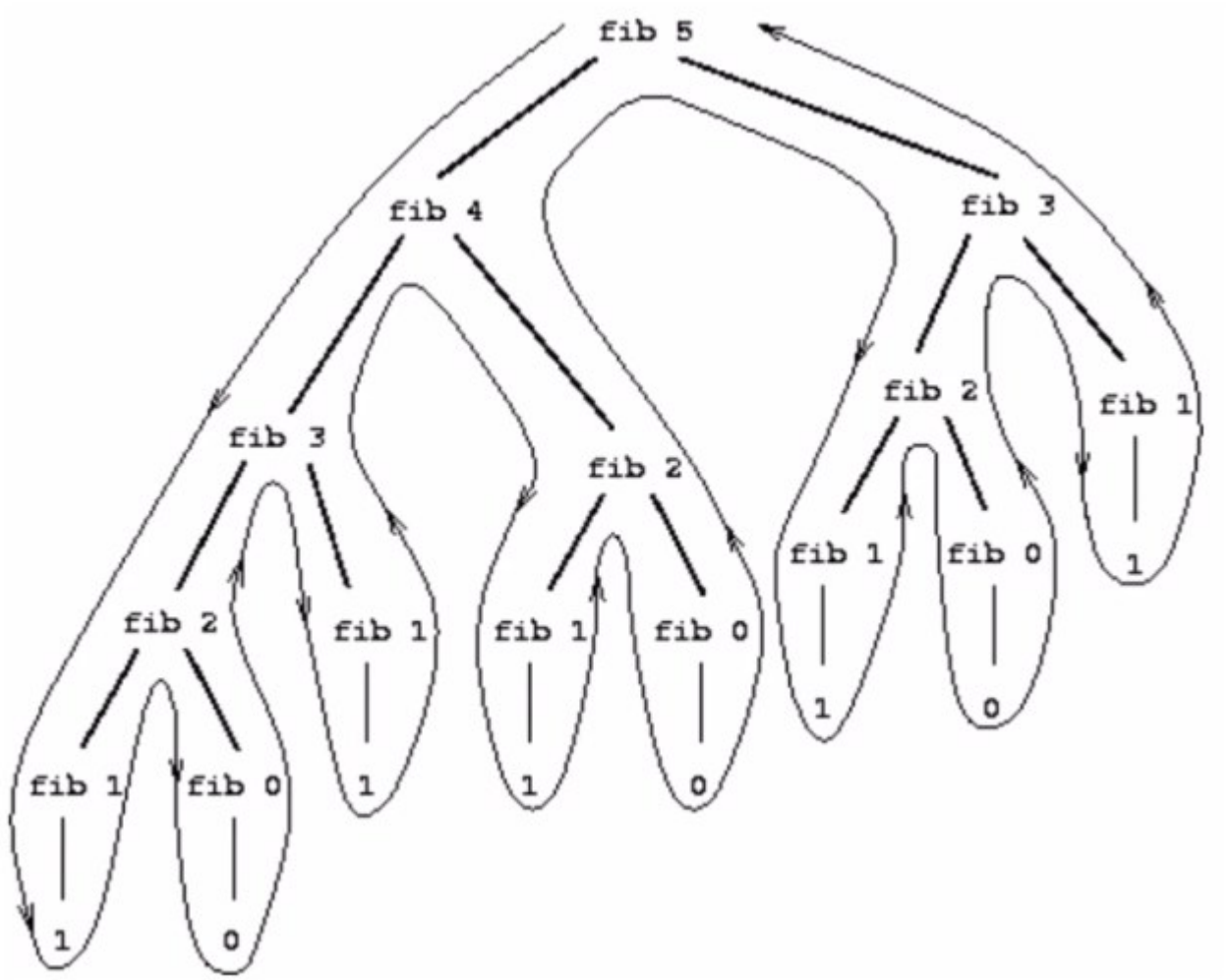
```
int fibonacci (int n) {
 if (n == 0) return 0;
 if (n == 1) return 1;
 return fibonacci(n - 1) + fibonacci(n - 2);
}
```

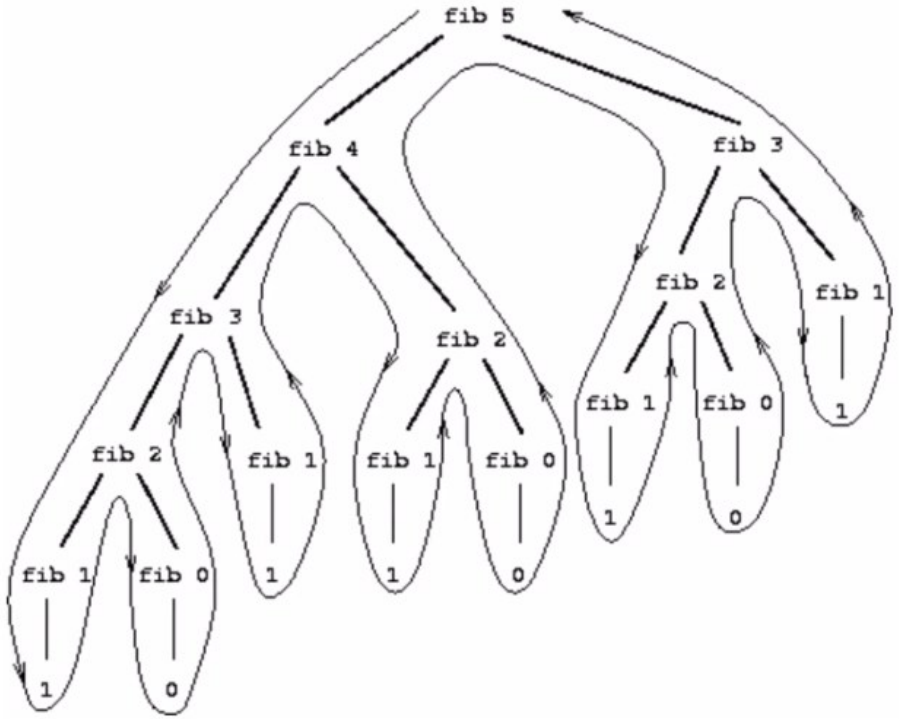
Se tentarmos executar este código passando o parâmetro como valor 5, a nossa função nos retornará o número 5 pois ele é o quinto número da sequência de Fibonacci após o zero **(lembrando que começamos a contar do 0)**:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

Sequência de Fibonacci

Logo, seu shape de execução terá o seguinte formato:





Por mais que isso seja interessante e nosso código no final fique enxuto e simples, uma abordagem como esta é ineficiente, pois se você reparar na imagem, verá que temos diversos galhos de execução repetidos, como por exemplo o fibonacci do número 1 (fib 1), o que acarreta um desperdício e compromete o tempo de execução do nosso algoritmo.

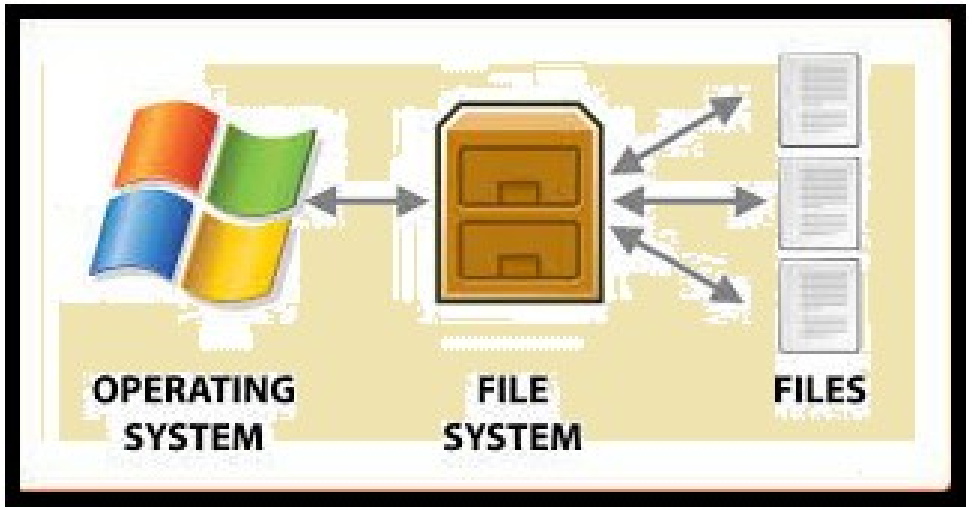
É importante tentarmos migrar um shape em árvore como este, para uma abordagem mais econômica. Neste caso podemos utilizar um loop para termos um shape de execução iterativo:

```
int fibonacci(int n) {
  int x = 0;
  int y = 1;
  int res = 0;

  for (int i = 0; i <= n; i++) {
    res = x;
    x = y;
    y = res + y;
  }

  return res;
}
```

A recursão em árvore não é sempre ruim, em alguns cenários ela é até necessária, por exemplo em file system de sistemas operacionais.



Lembre-se que o fato do seu código estar enxuto e simples não significa que ele está eficiente, então sempre é bom entender a complexidade dos algoritmos que você está escrevendo para não causar desperdício na hora da execução.

Relações de Recorrência

- Portanto:

Uma relação de recorrência é uma **equação em que cada termo de uma sequência é definido em função dos elementos anteriores**.

Resolver uma relação de recorrência é encontrar uma fórmula explícita que dá o termo geral da sequência, de preferência usando funções elementares ou outras relações de recorrência mais simples.

Vamos imaginar agora uma sequência

$$\frac{2}{3}, \frac{4}{9}, \frac{8}{27}, \frac{16}{81}, \frac{32}{243} \ldots =$$

Utilizando a recorrência teríamos:

$T_1 = \frac{2}{3}$ e o próximo termo pode ser calculado como: $T_2 = \frac{2.2}{3.3} = \frac{4}{9}$

$$T_3 = \frac{4.2}{9.3} = \frac{8}{27}$$

Então podemos escrever está recorrência da seguinte maneira:

$$\frac{2}{3}, \frac{4}{9}, \frac{8}{27}, \frac{16}{81}, \frac{32}{243} \quad \ldots =$$

$$\begin{cases} T_{n+1} = Tn \,.\, \frac{2}{.3} \\ T_1 = \frac{2}{3} \end{cases}$$

Agora podemos definir qualquer termo da sequência

Exemplo

Seja a fórmula de recorrência:

$$\begin{cases} T_{n+2} = T_{n+1} + T_n \\ T_1 = 3 \\ T_2 = 4 \end{cases}$$

Assim podemos definir a nossa sequência

(3, 4 , 7,11, 18,...)

Determine o termo geral da série abaixo.
E defina uma fórmula de recorrência para a sequência.

a) $\frac{1}{1}, \frac{2}{2}, \frac{3}{4}, \frac{4}{8}, \frac{5}{16} \ldots =$

Solução

$$\frac{1}{1}, \frac{5}{2}, \frac{25}{4}, \frac{125}{8}, \frac{625}{16} \ldots =$$

T(1)= 1/1

$$\begin{cases} T(n) = F(n-1).\frac{5}{2} \end{cases}$$

Exercícios

1) De maneira recursiva, considerando $n \in N^*$determine o quinto termo da sequência definida por:

$$\begin{cases} f(1) = \frac{14}{3} \\ f(n+1) = f(n) + \frac{2}{3} \end{cases}$$

Após definido os 5 primeiros termos da sequência, determine uma expressão matemática que calcule qualquer termo desta sequência de forma fechada.

2) Aplicando a recursividade resolva o problema abaixo.

$$F(2n+1) = 10.F(n) - 3 \quad onde \quad F(31) = 0 \quad qual\ o\ valor\ de\ \ F(0)?$$

3) Encontre a fórmula fechada das seguintes relações de recorrência:

(a)

$$\begin{cases} a_n &= 3a_{n-1} \\ a_0 &= 1 \end{cases}$$

(b)

$$\begin{cases} a_n &= -2a_{n-1} \\ a_0 &= 3 \end{cases}$$

(c)

$$\begin{cases} a_n = a_{n-1} + 2^n \\ a_0 = 1 \end{cases}$$

4) Determine uma função recursiva onde dado um valor um valor qualquer para x (inteiro positivo) possamos calcular o próximo elemento.

exemplo; se $x = 2$ teremos: $1, 2, 4, 8, \ldots$.

Se x = 3 teremos 1, 3,9,27,...

Se x = 4 teremos 1, 4,16,64,...

Bibiliografia

- https://www.linkapi.solutions/blog/o-que-realmente-e-um-algoritmo-recursivo
- https://www.ime.usp.br/~leo/mac/mac122/introducao_recursividade.html
- https://stringfixer.com/pt/Divide-and-conquer_method
- https://www.youtube.com/watch?v=qG5WxGGxrZQ&list=PLrVGp617x0hAttp3LQVBBF2td1uHjfhbr&index=3