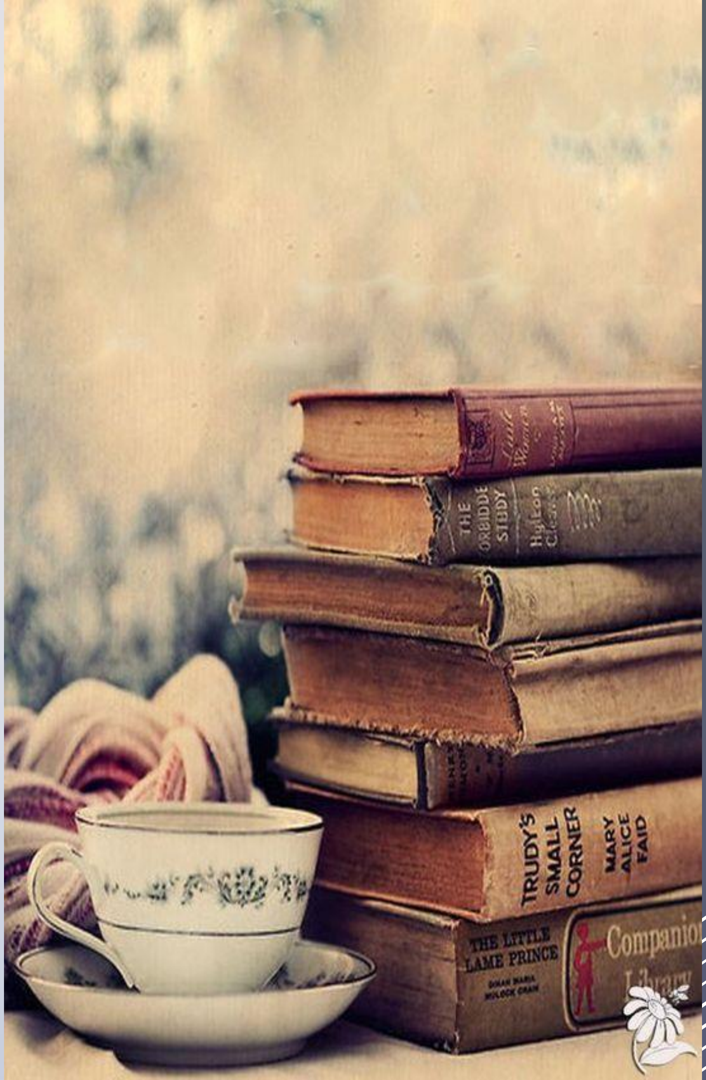
LABORATÓRIO DE PROGRAMAÇÃO II

RECURSIVIDADE



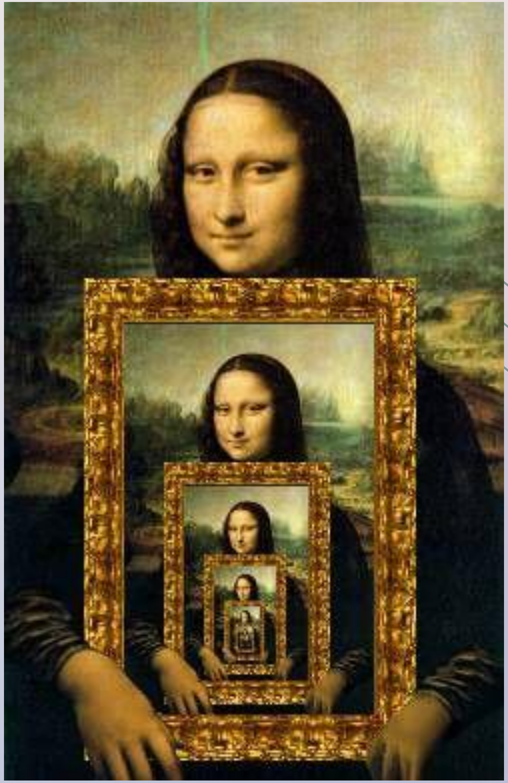
Agenda

Objetivos:

Apresentar conceitos de recursividade, vantagens e aplicações

Agenda:

- Recursividade
- Como o Compilador Executa a Recursão
- Exemplos
- Recursão X Iteração
- Considerações
- Atividade



Recursividade

Uma função que chama a si mesma é dita **recursiva**.

Geralmente as funções recursivas são divididas em duas partes:

(i) chamada recursiva, e

(ii) condição de parada (também chamada de Caso Base) para evitar ***loop*** infinito.

```
<tipo> nome_func(<lista de parâmetros>){
   ...
   if(<critério de parada>)//parada
        return<func(<argumentos>)>;//chamada recursiva
   ...
}
```

Recursividade

Como o compilador executa a recursão?

Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um registro de ativação na **Pilha de Execução do programa**.

O registro de ativação armazena os parâmetros e variáveis locais da função, bem como o "**registro de retorno**".

Ao final da execução dessa função, o registro é desempilhado e a execução é retomada de acordo com a informação armazenada no "registro de retorno"

Recursividade

A ideia básica da recursão é **dividir um problema maior em um conjunto de problemas menores**, que são então resolvidos de forma independente e depois combinados para gerar a solução final.

DIVIDIR E CONQUISTAR

Isso fica evidente no cálculo do fatorial.

O fatorial de um número **n** é o **produto de todos os números inteiros entre 1 e n**. Por exemplo, o fatorial de **3** é igual a **1*2*3**, ou seja, **6**.

Assim, o fatorial desse mesmo número 3 pode ser definido em termos do fatorial de 2, ou seja, **3! = 3* 2!**

Recursividade

Exemplo

Realizar o fatorial de um número inteiro positivo:

Sendo n um número de inteiros: $n! = \begin{cases} 1 \text{ se } n = 0 \\ n * n - 1 \text{ se } n > 0 \end{cases}$

ou seja,

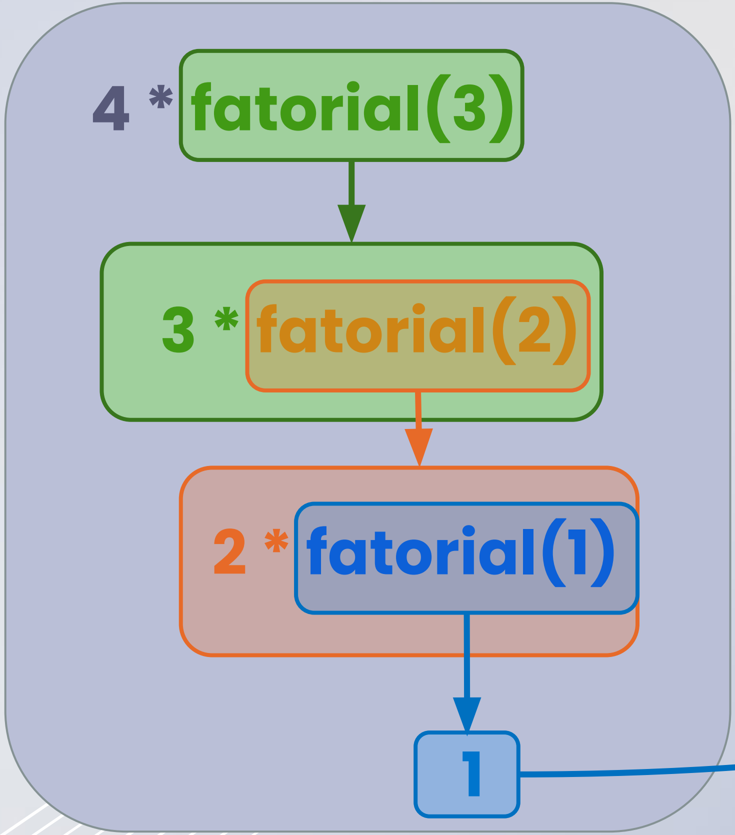
$0! = 1$

$n! = n*(n-1)!$

Recursividade

Exemplo

4! =



fatorial(4)= 4 * 6 = 24

fatorial(3)= 3 * 2 = 6

fatorial(2)= 2 * 1 = 2

fatorial(1)= 1

Recursividade

Exemplo

Fatorial(4) = 4 * Fatorial(3)

Fatorial(3) = 3 * Fatorial(2)

Fatorial(2) = 2 * Fatorial(1)

1

Fatorial(1) = 1

Fatorial(2) = 2 * 1 = 2

Fatorial(3) = 3 * 2 = 6

Fatorial(4) = 4 * 6 = 24

Recursividade

Exemplo

Realizar o fatorial de um número inteiro positivo

Solução Iterativa

```
int fatorial(int n)
{
    if(n <= 0)
        return 1;
    else {
        int i, f=1;
        for(i=2; i<=n; i++)
            f*=i;
        return f;
}
```

Solução Recursiva

```
int fatorial(int n)
{
    if(n <= 1)//caso base
        return 1;
    else
        return n*fatorial(n-1);
}
```

O fatorial de um número inteiro não negativo n, escrito como n!, é o produto de todos os números inteiros entre n e 1 (ou entre 1 e n):

n! = n x (n-1) x (n-2) x (n-3) x... x 1 ou n! = n x (n-1)!

Recursividade

Exemplo

4! É o produto 4 * 3 * 2 * 1 = 24

Empilha (Chamada, num)		Desempilha (Retorno)
4	2*FAT(2-1)	2*1
3	3*FAT(3-1)	3*2
2	4*FAT(4-1)	4*6
1	FAT(4)	24

Solução Recursiva

```
int fatorial(int n)
{
    if(n <= 1)//caso base
        return 1;
    else
        return n*fatorial(n-1);
}

int main()
{
    printf("%d", FAT(4));
    return 0;
}
```

Recursividade

Exemplo

Em geral, as formas recursivas dos algoritmos são consideradas "**mais elegantes**" do que suas formas iterativas.

Isso **facilita a interpretação do código**.

Porém, esses algoritmos apresentam **maior dificuldade na detecção de erros** e podem ser **ineficientes**.

Todo cuidado é pouco ao se fazer funções recursivas, pois duas coisas devem ficar bem estabelecidas:

- O critério de parada e,
- O parâmetro da chamada recursiva.

Recursividade

Exemplo

Solução Recursiva

```
int fatorial(int n)
{
    if(n <= 1)//caso base
        return 1;
    else
        return n*fatorial(n-1);
}

int main()
{
    printf("%d", FAT(4));
    return 0;
}
```

Critério de parada determina quando a função deverá parar de chamar a si mesma. Se ela não existir, a função irá executar infinitamente.

Parâmetro da chamada recursiva: quando chamamos a função dentro dela mesmo, devemos sempre **mudar o valor do parâmetro passado**, de forma que a recursão chegue a um término.

Recursividade

Exemplo

Realizar a soma dos 5 primeiros números naturais.

Solução Iterativa

```
res = SOMAR(5);
int SOMAR(int num){
   int j, soma=0;

    for(j=0; j<num; j++)
        soma+=j;

    return soma;
}
```

Solução Recursiva

```
res = SOMAR(1);
int SOMAR(int num)
{
    if(num < 5)
        return num+SOMAR(num+1);
    else
        return 0;
}
```

O fatorial de um número inteiro não negativo n, escrito como n!, é o produto de todos os números inteiros entre n e 1 (ou entre 1 e n):

n! = n x (n-1) x (n-2) x (n-3) x... x 1 ou n! = n x (n-1)!

Recursividade

Exemplo

Cada chamada à função recursiva, bem como seus parâmetros e variáveis locais, são armazenados em uma pilha, tal que o retorno à função ocorra na ordem LIFO

	Empilha (Chamada, num)	Desempilha (Retorno)
5	4+SOMAR(4+1)	4+0
4	3+SOMAR(3+1)	3+4
3	2+SOMAR(2+1)	2+7
2	1+SOMAR(1+1)	1+9
1	SOMAR(1)	10

Solução Recursiva

```
int SOMAR(int num){
    if(num < 5)
        return num+SOMAR(num+1);
    else
        return 0;
}

int main()
{
    printf("%d", SOMAR(1));
    return 0;
}
```

Recursividade

Sempre que chamamos uma função, é necessário um **espaço de memória para armazenar os parâmetros**, **variáveis locais** e **endereço de retorno da função**.

Numa função recursiva, essas informações são armazenadas para **cada chamada da recursão**, sendo, portanto a memória necessária para armazená-las **proporcionalmente ao número de chamadas da recursão**.

Além disso, todas essas tarefas de alocar e liberar memória, copiar informações, dentre outros, envolvem **tempo computacional**, de modo que uma função recursiva gasta mais tempo que sua versão iterativa (sem recursão).

Portanto, **algoritmos recursivos tendem a necessitar de mais tempo e/ou espaço do que algoritmos iterativos.**

Recursividade

Exemplo

Abra o arquivo **a_Recursao.c**

No mesmo arquivo, insira como comentários a ordem de do empilhamento das chamadas recursivas com o valor da variável empilhada, e depois o respectivo valor de cada retorno à chamada, como nos exemplos anteriores. Considere como entrada o valor 4.

Recursividade

Recursão X Iteração

Tanto a iteração como a recursão se baseiam em uma estrutura de controle:

- ✔ A iteração usa uma estrutura de repetição
- ✔ A recursão usa uma estrutura condicional

Os dois métodos envolvem repetições:

- ✔ Iteração usa explicitamente uma estrutura de repetição
- ✔ Recursão obtém repetição por intermédio de chamadas repetidas de funções

Recursividade

Recursão X Iteração

Os dois métodos envolvem um teste de encerramento:

- ✔ A iteração termina quando uma condição de continuação de laço se torna falsa
- ✔ A recursão termina quando um caso básico é reconhecido

Desvantagens da utilização de recursão:

- ✔ A recursão faz repetidas chamadas à função, e isso pode custar caro tanto em tempo de processamento como em espaço de memória, já que o valor das variáveis locais tem de ser mantidos em pilhas

Recursividade

Recursão X Iteração

Vantagens da utilização de recursão

- ✔ Criar versões mais claras e simples de algoritmos

Exemplos: pesquisa binária; algoritmo de ordenação QuickSort (método de ordenação) é muito difícil de implementar de forma iterativa; muitos problemas relacionados com inteligência artificial também são resolvidos mais facilmente utilizando recursão

Escolher recursão quando

- ✔ O problema puder ser resolvido de modo mais natural, resultando em programas mais fáceis de compreender
- ✔ Uma solução iterativa não for fácil de ser encontrada

Recursividade

Considerações

- A técnica de recursividade, que é um paradigma de projeto de algoritmos, permite escrever algoritmos de modo mais claro.
- É usada para resolver problemas complexos e repetitivos; possibilita, sucessivamente, diminuir um problema em um problema menor ou mais simples, até que ele possa ser resolvido sem recorrer a si mesmo.
- Cada chamada à função recursiva, bem como seus parâmetros e variáveis locais, são armazenados em uma pilha tal que o retorno à função ocorra na ordem LIFO (*Last In, First Out*).

Exemplo

Série de Fibonacci

Obs.

Cada elemento da Série é a soma dos dois anteriores:

1, 1, 2, 3, 5, 8, 13, ...

Fib(0) = 0

Fib(1) = 1

Fib(n) = Fib(n-1)+Fib(n-2) para n > 1

Assim, no algoritmo recursivo, para n=0 ou n=1 (que representam o caso base, isto é, quando a função não recorre a si mesma), retornar n.

Caso contrário (n>1), retornar Fib(n-1)+Fib(n-2)

Atividade

FIB(0) = 0 FIB(1) = 1 FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

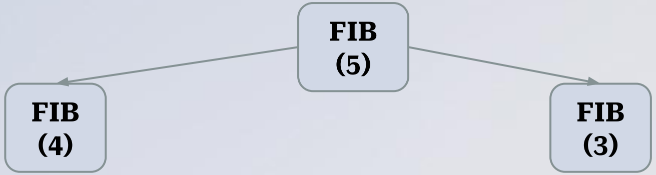
FIB(5) = ?

FIB (5)

Atividade

FIB(0) = 0 FIB(1) = 1 FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

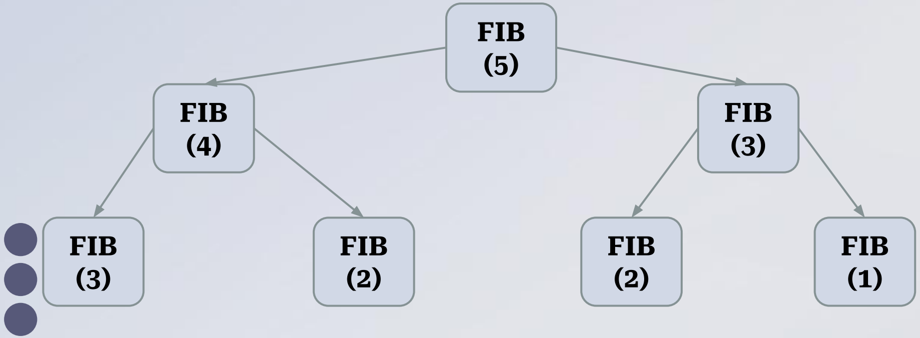
FIB(5) = ?



Atividade

FIB(0) = 0 FIB(1) = 1 FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

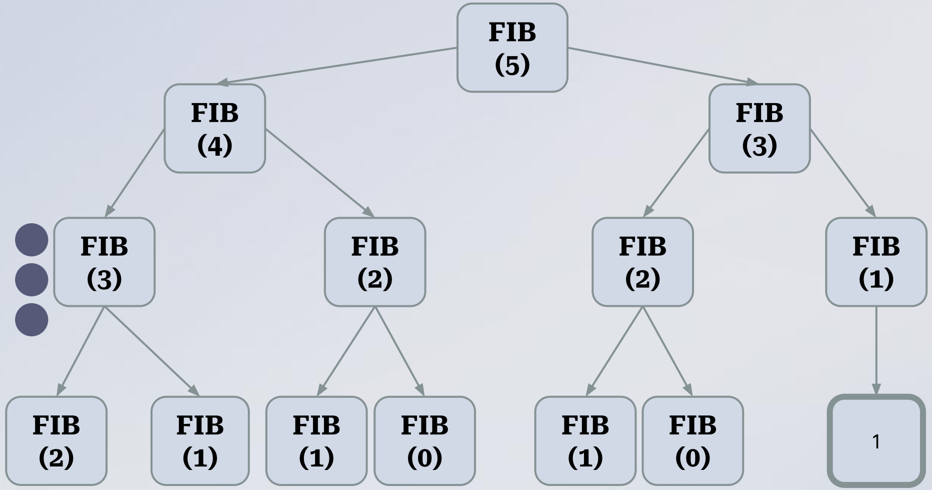
FIB(5) = ?



Atividade

FIB(0) = 0   FIB(1) = 1   FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

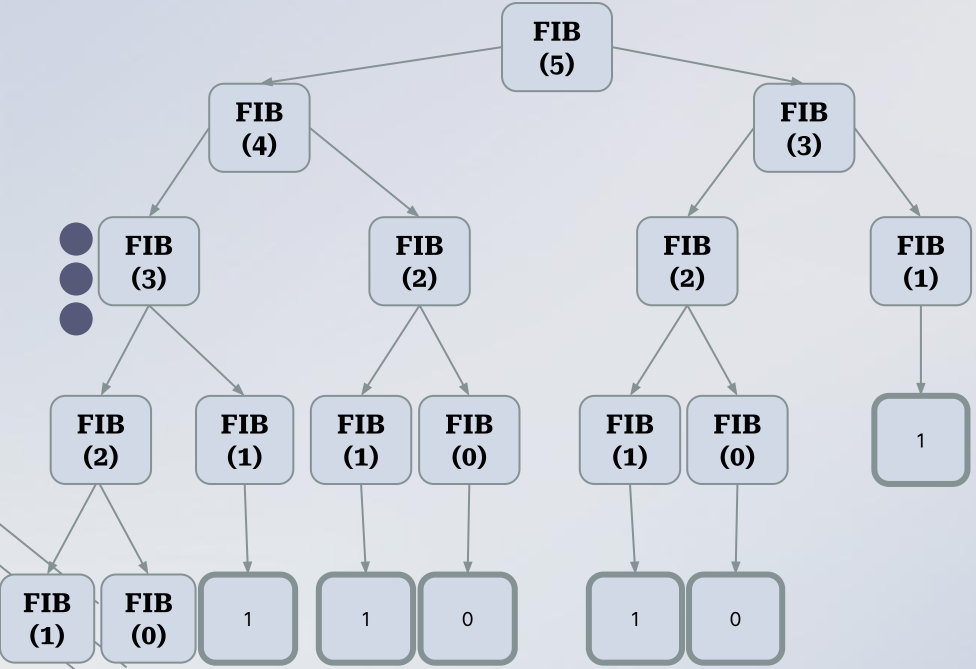
FIB(5) = ?



Atividade

FIB(0) = 0 FIB(1) = 1 FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

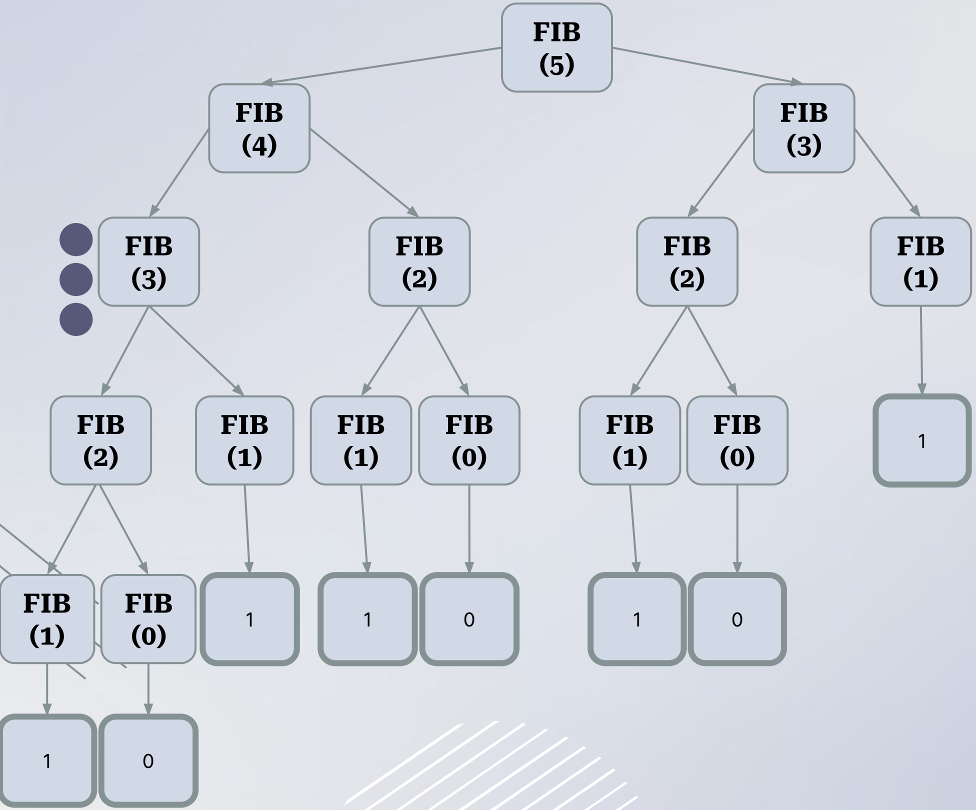
FIB(5) = ?



Atividade

FIB(0) = 0   FIB(1) = 1   FIB(n) = FIB(n - 1) + FIB(n - 2) para n > 1

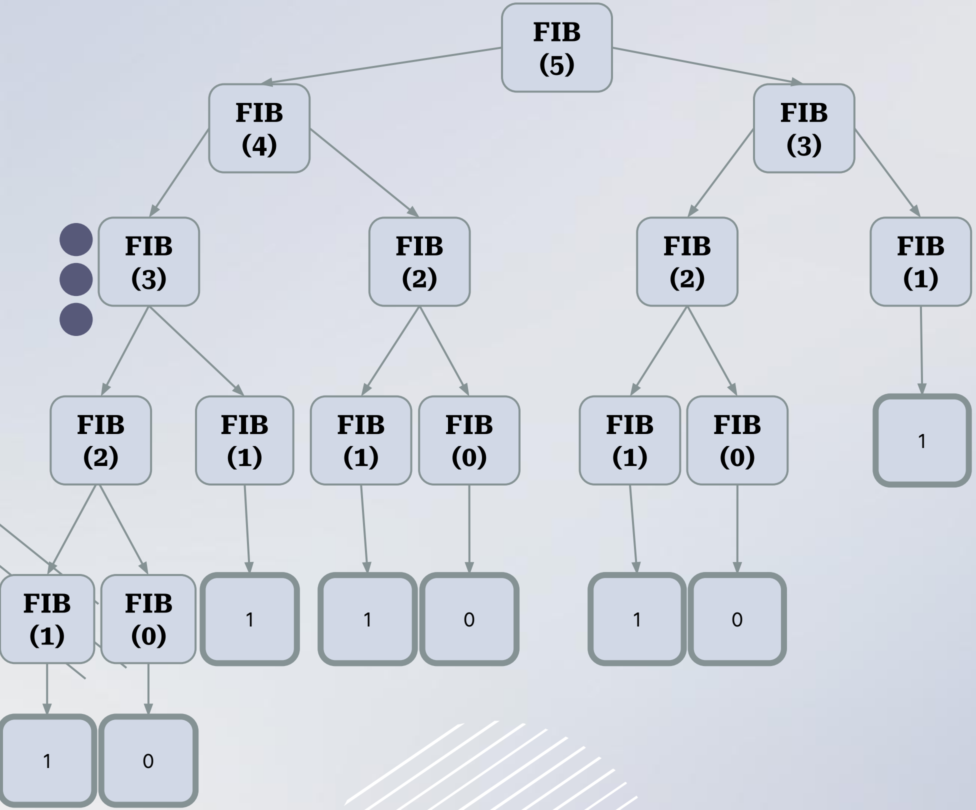
FIB(5) = ?

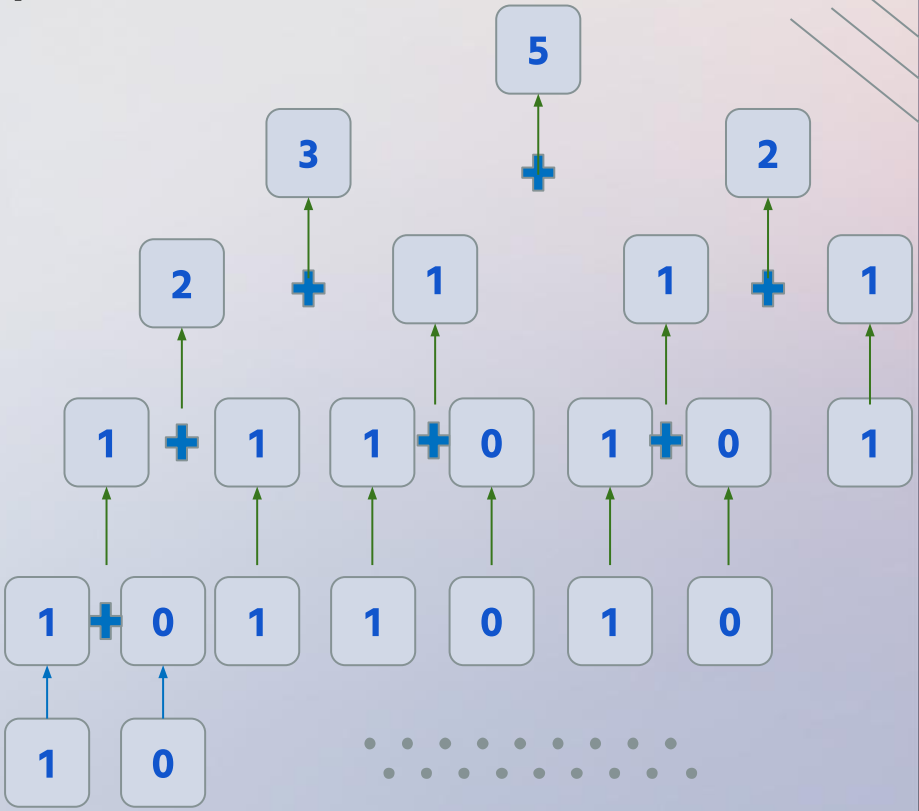


Atividade

$FIB(0) = 0 \qquad FIB(1) = 1 \qquad FIB(n) = FIB(n-1) + FIB(n-2)$ para $n > 1$

FIB(5) = ?





Atividade

Faça uma solução iterativa e uma recursiva para obter o n-ésimo termo da Série de Fibonacci em **b_Fibonacci.c**

Obs.

Cada elemento da Série é a soma dos dois anteriores:

1, 1, 2, 3, 5, 8, 13, ...

Fib(0) = 0

Fib(1) = 1

Fib(n) = Fib(n-1)+Fib(n-2) para n > 1

Assim, no algoritmo recursivo, para n=0 ou n=1 (que representam o caso base, isto é, quando a função não recorre a si mesma), retornar n.

Caso contrário (n>1), retornar Fib(n-1)+Fib(n-2)

Abra o arquivo **c_exerciciosRecursao.c** e faça as atividades solicitadas.