

Laboratório de Fundamentos em TIC

Ponteiros em C

Prof. Gabriel Resende Machado



gabrielmachado@unifeso.edu.br



<https://www.linkedin.com/in/machadogabriel>



<https://github.com/UNIFESO-Gabriel/fundamentos-em-tic>

Relembrando: O que é Recursividade?

- É uma forma de resolver um problema computacional onde a solução depende de soluções menores do mesmo problema;
- Uma recursão é caracterizada por **três fatores**:
 - é baseada em uma subrotina;
 - contém uma condição de parada;
 - contém uma condição recursiva.



Relembrando: Observações sobre Recursividade

- Recursividade é um maneira de realizar **repetições de forma funcional** (*i.e.* por meio de subrotinas);
- **Única forma de repetição disponível** em algumas linguagens de programação (Prolog, Lisp, Haskell);
- Soluções baseadas em recursividade **podem ser mais simples** do que soluções iterativas convencionais:
 - Exemplos incluem problemas envolvendo estruturas de dados inerentemente recursivas: **grafos e árvores.**
- Contudo, soluções recursivas são, geralmente, **mais complexas e difíceis de analisar**;
- Programas recursivos têm **a pilha (*stack*)** como estrutura de dados principal;
- Programas recursivos tendem a utilizar mais memória e estão sujeitos a um problema de execução chamado ***Stack Overflow***.

Relembrando: Recursividade - Problema 1

- Elabore um programa recursivo que calcule o fatorial de um número inteiro $n \geq 0$.
- Dicas:
 - Considere o caso inicial como **atômico**, *i.e.* que forneça a resposta mais simples possível;
 - Para implementar o caso recursivo, pense em como reduzir o problema em partes mais simples;
 - Utilize os tipo de retorno e os parâmetros da subrotina para realizar a recursão.



Relembrando: Recursividade - Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```

Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

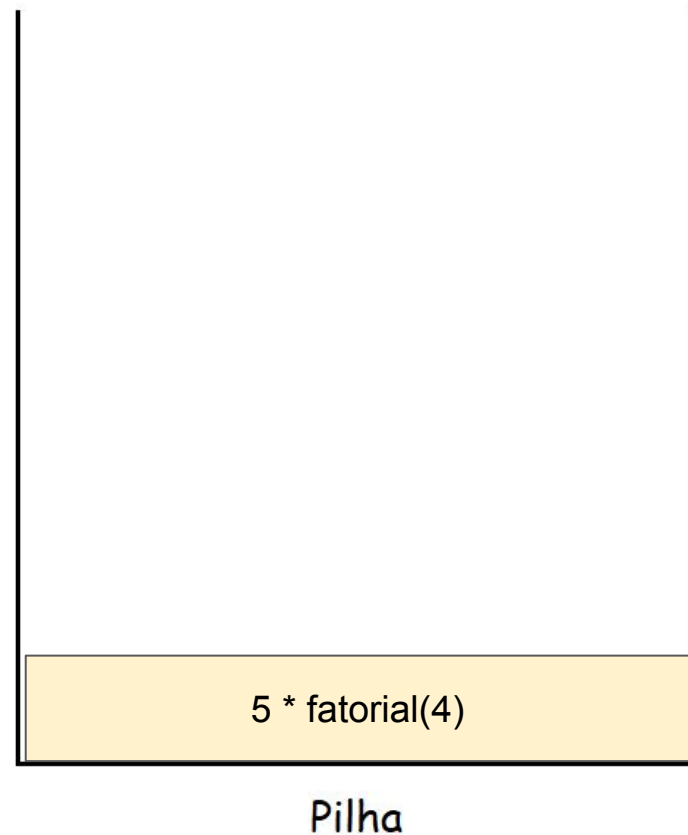
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

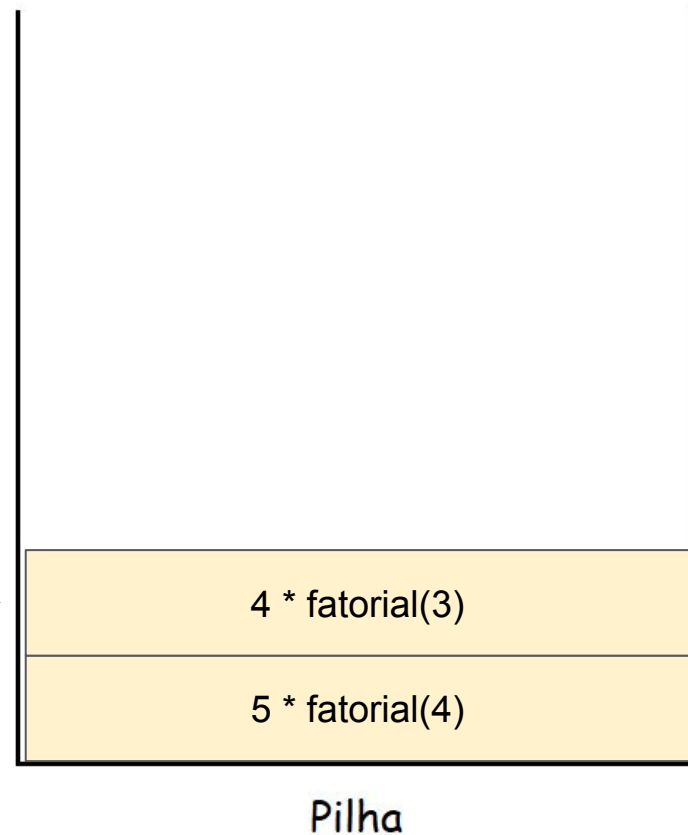
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

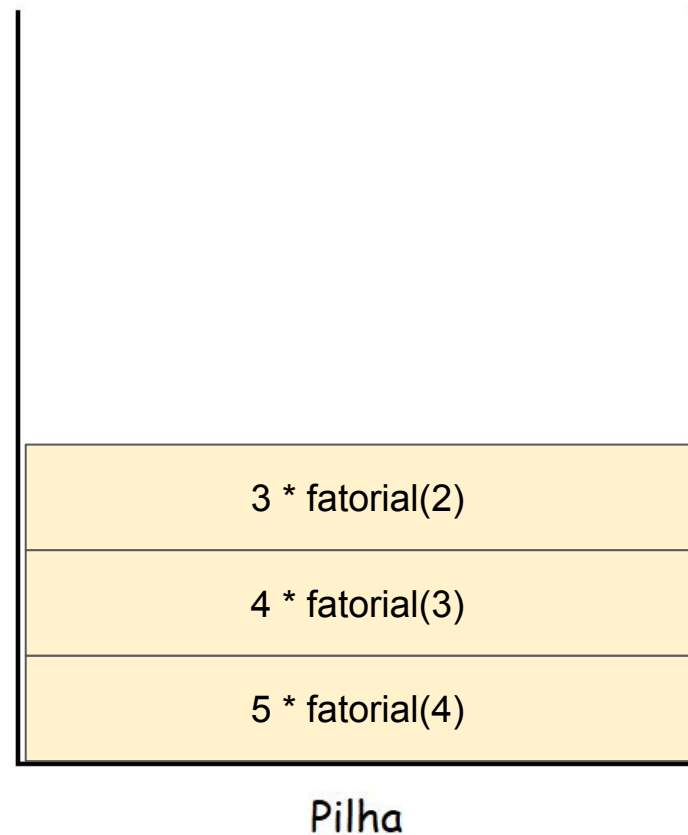
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

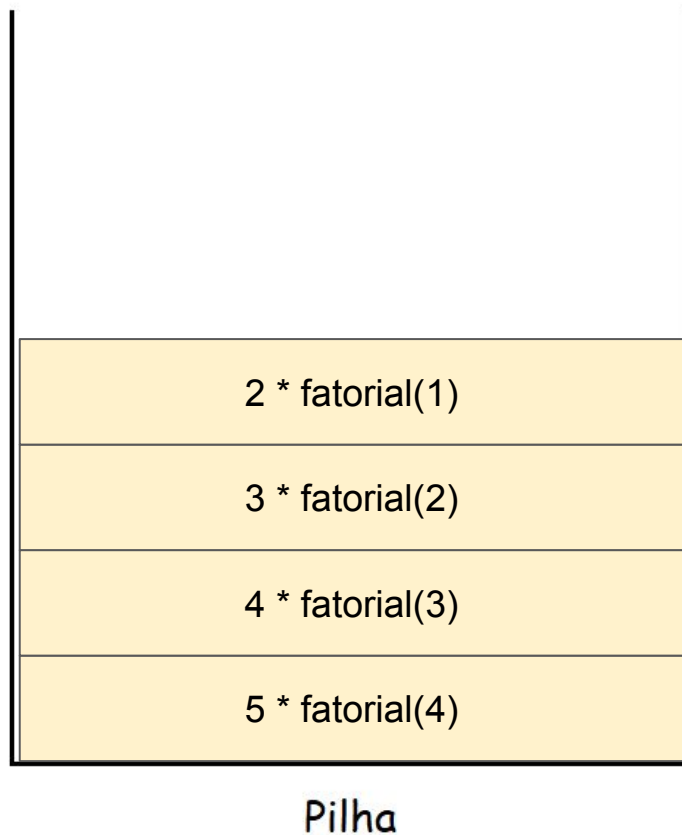
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

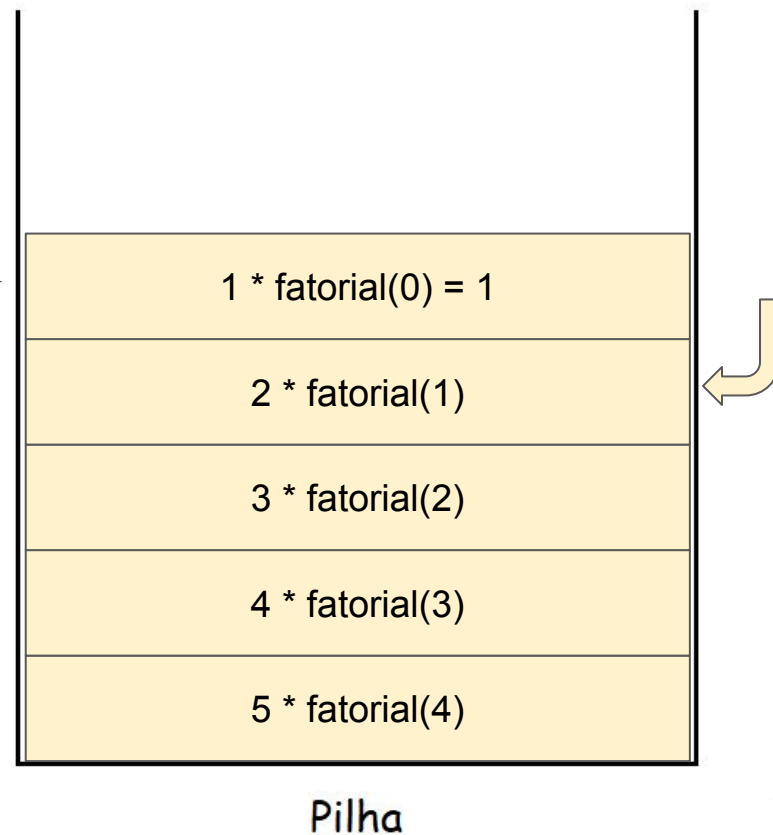
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

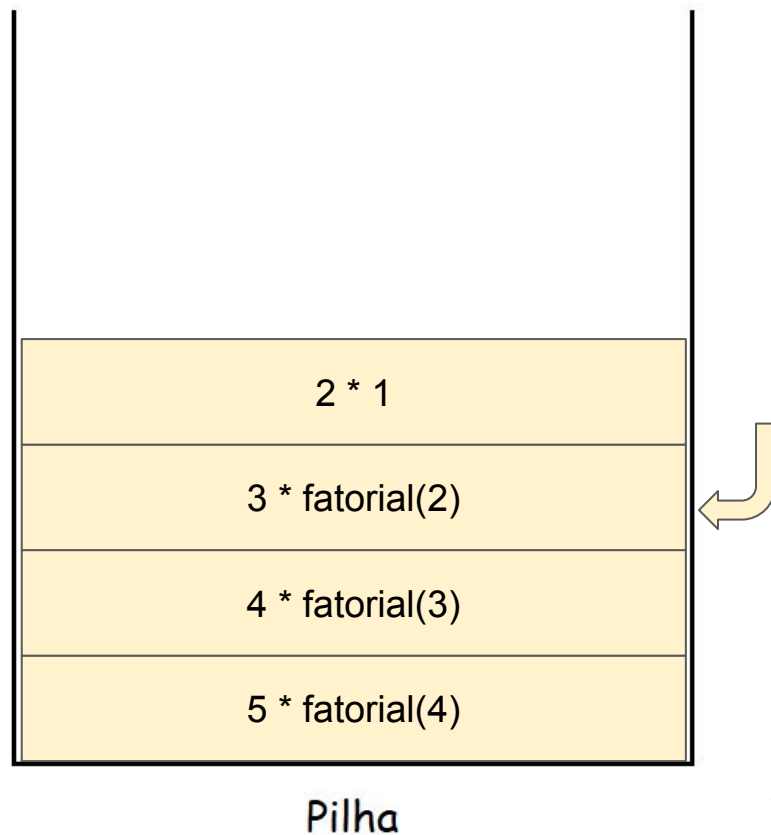
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

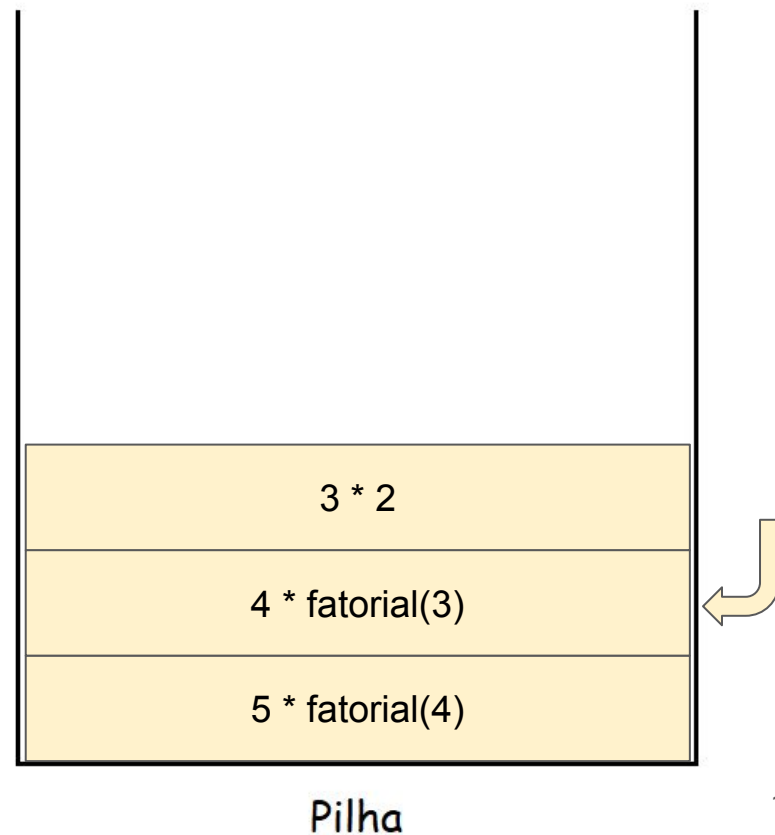
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

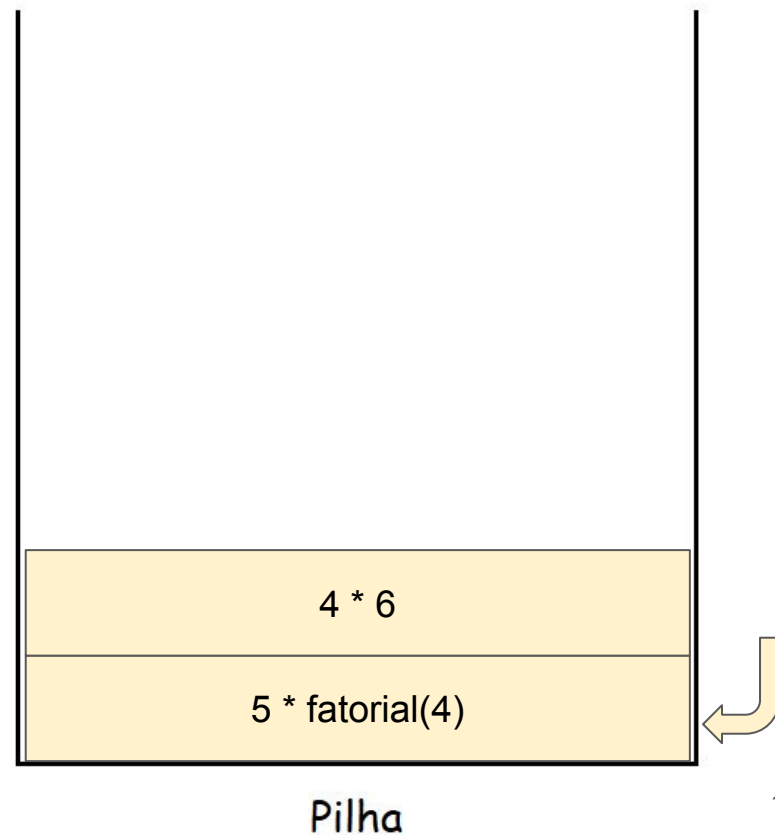
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

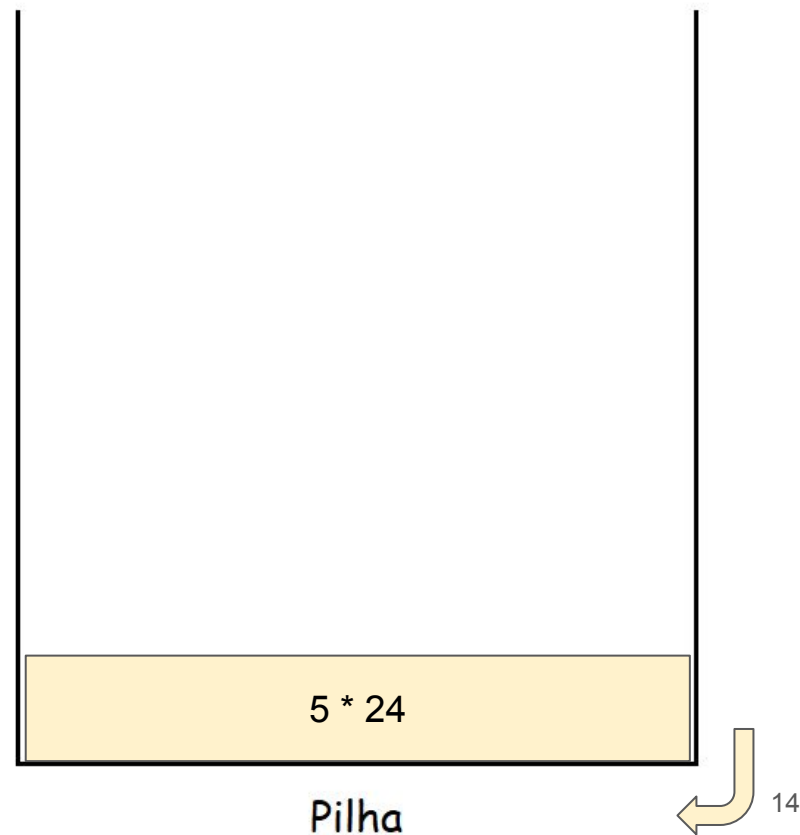
    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```



Relembrando: Recursividade - Análise da Solução do Problema 1

N = 5

```
#include <stdio.h>

int fatorial(int n) {

    if (n == 0 || n == 1)
        return 1; // caso base

    return n * fatorial(n - 1); // caso recursivo
}

int main() {

    printf("Digite um numero inteiro >= 0: ");
    int n, fatorial_n; scanf("%d", &n);

    fatorial_n = fatorial(n);
    printf("%d! = %d", n, fatorial_n);
    return 0;
}
```

120

Pilha

Recursividade - Problema 2

- Elabore um programa recursivo que localize o índice de um número em um *array* em ordem crescente. Caso não encontre o número, o programa deve retornar -1.
- Dicas:
 - Considere o caso inicial como **atômico**, *i.e.* que forneça a resposta mais simples possível;
 - Para implementar o caso recursivo, pense em como reduzir o problema em partes mais simples;
 - Utilize os tipo de retorno e os parâmetros da subrotina para realizar a recursão.



Recursividade - Solução do Problema 2

```
int busca_binaria(int* lista, int inicio, int fim, int numero_procurado) {  
  
    if (inicio > fim)  
        return -1;  
  
    int mid = (inicio + fim) / 2;  
  
    if (lista[inicio] == numero_procurado)  
        return inicio;  
  
    if (lista[fim] == numero_procurado)  
        return fim;  
  
    if (lista[mid] == numero_procurado)  
        return mid;  
  
    if (lista[mid] < numero_procurado)  
        return busca_binaria(lista, mid+1, fim, numero_procurado);  
  
    if (lista[mid] > numero_procurado)  
        return busca_binaria(lista, inicio, mid-1, numero_procurado);  
  
}
```

Recursividade - Problema 3

- Elabore um programa recursivo que retorne o n -ésimo termo da sequência de Fibonacci.
- Dicas:
 - Considere o caso inicial como **atômico**, *i.e.* que forneça a resposta mais simples possível;
 - Para implementar o caso recursivo, pense em como reduzir o problema em partes mais simples;
 - Utilize os tipo de retorno e os parâmetros da subrotina para realizar a recursão.



Recursividade - Solução do Problema 3

```
#include <stdio.h>
#include <stdlib.h>

int fibonacci(int termo) {
    if (termo == 0)
        return 0;

    if (termo == 1)
        return 1;

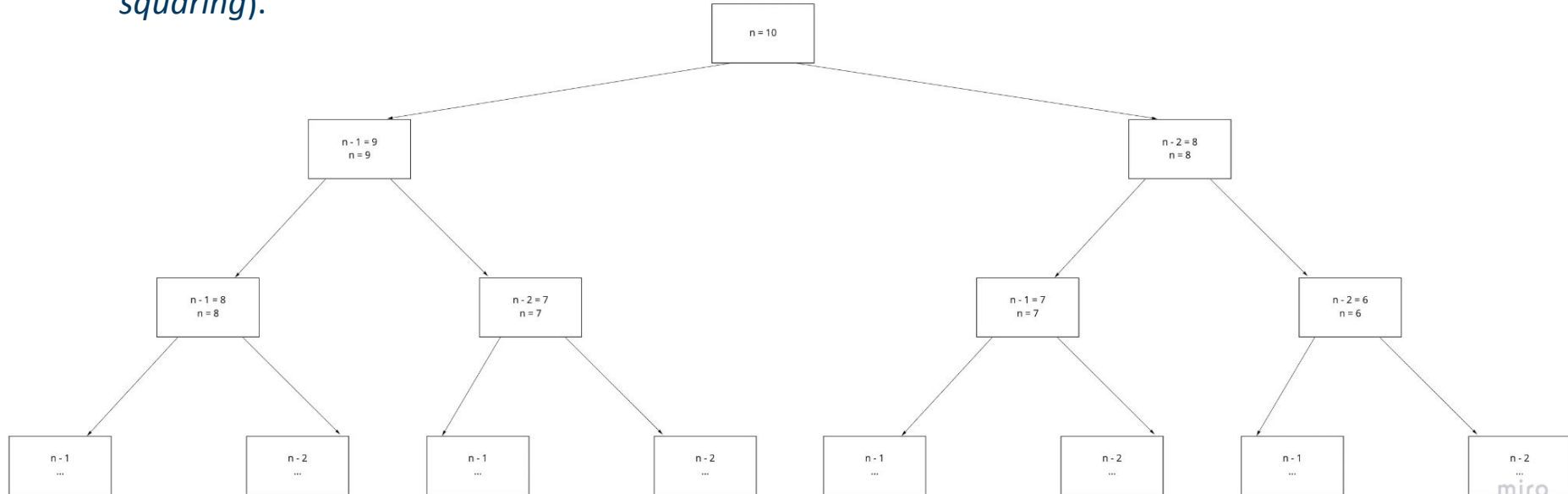
    return fibonacci(termo - 1) + fibonacci(termo - 2);
}

int main() {

    printf("Digite o termo de Fibonacci: ");
    int termo; scanf("%d", &termo);
    printf("O termo %d de Fibonacci eh %d", termo, fibonacci(termo));
    return 0;
}
```

Recursividade - Discussão do Problema 3

- Algoritmo de complexidade exponencial: $O(2^n)$;
- Repetição de soluções já computadas em passos anteriores;
- Versão iterativa muito mais eficiente: $O(n)$, porém é possível chegar a $O(\log_2 n)$ (*recursive squaring*).



Recursividade vs. Iteração - Quando Usar?

- **Recursão:**

- Quando o problema pode ser dividido em subproblemas menores e similares;
- Quando o problema naturalmente exibe auto-similaridade, como fractais;
- Quando a solução se beneficia da clareza e simplicidade em detrimento de soluções iterativas.

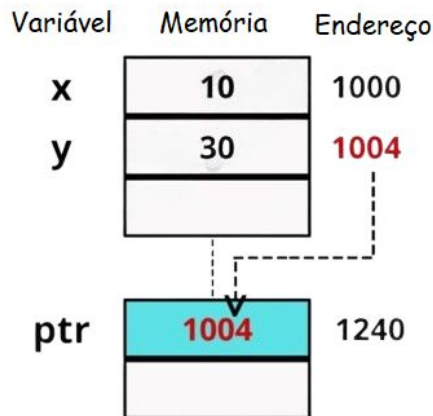
- **Iteração:**

- Quando o *overhead* de desempenho e memória são fatores de preocupação;
- Quando a profundidade da recursão pode ser potencialmente muito grande;
- Quando o problema pode ser resolvido de forma mais natural usando *loops*.

Ponteiros em C - O que São?

- Como já visto, uma **variável** é o nome de uma **posição na memória** que pode **armazenar um valor de um determinado tipo**;
- Um **ponteiro** é uma **variável** que pode armazenar o **endereço de uma posição na memória**;
- Ponteiros podem também apontar para variável alguma, a partir do valor **NULL**;
- A maioria das linguagens de programação utilizam ponteiros direta ou indiretamente em seus compiladores/interpretadores.

```
Ponteiros em C
#include <stdio.h>
int main()
{
    int x=10, y=30;
    int *ptr;
    ptr = &y;
    return 0;
}
```



Exemplo de Ponteiros em C

```
int main() {  
  
    int num = 5; // variável inteira.  
    int *ptr_num = NULL; // ponteiro para variável inteira.  
  
    // 'ptr_num' recebe o endereço de memória de 'num'.  
    ptr_num = &num;  
  
    // exibe o valor armazenado em 'num'.  
    printf("Valor de num: %d\n", num);  
  
    // exibe os endereços de memória de 'num' a partir de '&'.  
    printf("Endereco de num: %p\n", &num);  
  
    // exibe o endereço de memória armazenado em 'ptr_num'.  
    printf("Valor de ptr_num: %p\n", ptr_num);  
  
    // desreferencia o ponteiro 'ptr_num'.  
    printf("Valor da variavel apontada por ptr_num: %d\n", *ptr_num);  
  
    // exibe o endereço de memória de 'ptr_num'.  
    printf("Endereco de ptr_num: %p\n", &ptr_num);  
  
    return 0;  
}
```



```
Valor de num: 5  
Endereco de num: 0x7ffd2c7fc18c  
Valor de ptr_num: 0x7ffd2c7fc18c  
Valor da variavel apontada por ptr_num: 5  
Endereco de ptr_num: 0x7ffd2c7fc180
```

Passagem por Valor e Referência

- Até o momento do curso, os parâmetros declarados nas subrotinas receberam uma cópia dos argumentos passados nas chamadas dessas subrotinas;
- Esse mecanismo é conhecido como **passagem por valor**;
 - Como é realizada uma cópia de cada argumento, **esse processo pode ser custoso**;
- Quando se deseja (i) alterar o valor da variável diretamente na subrotina e/ou (ii) evitar a cópia dos argumentos para os parâmetros, utiliza-se a **passagem por referência**;
 - **Nesses casos, faz-se uso da funcionalidade dos ponteiros**: os parâmetros são definidos como variáveis ponteiro (*), enquanto que, na chamada da subrotina, são passados os endereços dos argumentos (&).

Exemplo de Código: Passagem por Valor

```
#include <stdio.h>

void passagem_por_valor(int num1, int num2) {

    printf("Valores iniciais dos parametros:\n");
    printf("num1: %d, num2: %d\n\n", num1, num2);
    num1 += 100;
    num2 += 100;
    printf("Valores finais dos parametros:\n");
    printf("num1: %d, num2: %d\n\n", num1, num2);
}

int main() {

    int numero1 = 1, numero2 = 2;
    printf("Valores iniciais dos argumentos:\n");
    printf("numero1: %d, numero2: %d\n\n", numero1, numero2);

    passagem_por_valor(numero1, numero2);

    printf("Valores finais dos argumentos:\n");
    printf("numero1: %d, numero2: %d\n\n", numero1, numero2);
    return 0;
}
```



Valores iniciais dos argumentos:
numero1: 1, numero2: 2

Valores iniciais dos parametros:
num1: 1, num2: 2

Valores finais dos parametros:
num1: 101, num2: 102

Valores finais dos argumentos:
numero1: 1, numero2: 2

Exemplo de Código: Passagem por Referência

```
#include <stdio.h>

void passagem_por_referencia(int *num1, int *num2) {

    printf("Valores iniciais dos parametros:\n");
    printf("num1: %d, num2: %d\n\n", *num1, *num2);
    *num1 += 100;
    *num2 += 100;
    printf("Valores finais dos parametros:\n");
    printf("num1: %d, num2: %d\n\n", *num1, *num2);
}

int main() {

    int numero1 = 1, numero2 = 2;
    printf("Valores iniciais dos argumentos:\n");
    printf("numero1: %d, numero2: %d\n\n", numero1, numero2);

    passagem_por_referencia(&numero1, &numero2);

    printf("Valores finais dos argumentos:\n");
    printf("numero1: %d, numero2: %d\n\n", numero1, numero2);
    return 0;
}
```



Valores iniciais dos argumentos:
numero1: 1, numero2: 2

Valores iniciais dos parametros:
num1: 1, num2: 2

Valores finais dos parametros:
num1: 101, num2: 102

Valores finais dos argumentos:
numero1: 101, numero2: 102


Alocação Dinâmica de Memória

- Nem sempre é possível saber *a priori* o quantidade de memória necessária em **tempo de compilação**;
- A alocação dinâmica de memória permite ao programador **alocar memória em tempo de execução**, proporcionando flexibilidade no gerenciamento de recursos e às mudanças de requisitos e tamanhos de dados;
- **São extremamente úteis para:**
 - **A criação de estruturas de dados**, como filas, pilhas, listas encadeadas, árvores, etc;
 - **Economia de recursos**, alocando exatamente a quantidade de memória necessária para a execução do programa.
- Contudo, é necessário liberar a memória alocada manualmente para evitar vazamentos, por meio da subrotina **free**;
- A principal subrotina do C para essa tarefa é **malloc**. Há também a **calloc** e a **realloc**.


Malloc em Detalhes

```
int *aposta = (int *)malloc(num_dezenas * sizeof(int));
```


Conversão de tipo
(CAST)



Queremos
num_dezenas ints



sizeof(int) retorna a
quantidade de memória
que int precisa



Obs.: **malloc** retorna **NULL** caso não seja possível alocar a memória solicitada.

Exemplo de Código: Alocação Dinâmica de um *Array*

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    int num_dezenas;
    printf("Digite quantas dezenas deseja apostar: ");
    scanf("%d", &num_dezenas);

    printf("Digite as %d dezenas de sua aposta: ", num_dezenas);

    // utiliza a função malloc para alocar
    // num_dezenas * sizeof(int) espaços na memória.
    int *aposta = (int *)malloc(num_dezenas * sizeof(int));

    for (int i = 0; i < num_dezenas; i++)
        scanf("%d", &aposta[i]);

    printf("Verifique sua aposta:\n");
    for (int i = 0; i < num_dezenas; i++)
        printf("%d ", aposta[i]);

    // libera a memória alocada.
    free(aposta);
    return 0;
}
```



```
Digite quantas dezenas deseja apostar: 6

Digite as 6 dezenas de sua aposta:
44 32 55 60 12 23

Verifique sua aposta:
44 32 55 60 12 23
```

Exemplo de Código: Alocação Dinâmica de uma Matriz

```
int **apostas;
// aloca memória para um vetor de int *.
apostas = (int **)malloc(num_jogos * sizeof(int *));

for (int i = 0; i < num_jogos; i++) {
    printf("\nDigite as %d dezenas de sua Aposta %d:\n", num_dezenas, i+1);
    apostas[i] = (int *)malloc(num_dezenas * sizeof(int));
    for (int j = 0; j < num_dezenas; j++) {
        // aloca memória para um vetor de int.
        scanf("%d", &apostas[i][j]);
    }
}

printf("\nVerifique suas apostas:\n");
for (int i = 0; i < num_jogos; i++) {
    printf("Aposta %d: ", i+1);
    for (int j = 0; j < num_dezenas; j++) {
        printf("%d ", apostas[i][j]);
    }
    printf("\n");
}

// libera toda a memória alocada.
for (int i = 0; i < num_jogos; i++)
    free(apostas[i]);
free(apostas);
```



```
Digite quantos jogos deseja apostar: 2
Digite quantas dezenas deseja apostar: 6

Digite as 6 dezenas de sua Aposta 1:
2 4 6 8 10 12

Digite as 6 dezenas de sua Aposta 2:
12 14 46 48 20 33

Verifique suas apostas:
Aposta 1: 2 4 6 8 10 12
Aposta 2: 12 14 46 48 20 33
```

Ponteiros `void`

- Como visto, **variáveis ponteiro** precisam saber *a priori* o tipo esperado da variável que terá seu endereço armazenado;
 - Contudo, isso cria uma **forte dependência de tipagem**.
- Uma alternativa para contornar essa dependência é declarar variáveis ponteiro do tipo `void *`;
- Os ponteiros `void *` permitem, principalmente, **criar subrotinas genéricas** sem que seja necessário repetir sua funcionalidade para cada tipo de variável em C;
- Em C há várias subrotinas que fazem uso dos ponteiros `void`, entre elas a **`qsort`**, a **`malloc`**, a **`calloc`** e a **`realloc`**;
- Variáveis do tipo `void *` não podem ser desreferenciadas diretamente. É preciso realizar um *type casting*.

Ponteiros void

```
#include <stdio.h>

void mostrar_ptr_void(void *var) {

    // realiza o type casting para int *.
    int *numero = (int *)var;

    // desreferencia o ponteiro para exibir seu valor.
    printf("%d\n", *numero);

    // O código abaixo produz um erro.
    // printf("%d\n", *var); // ERRO!
}

int main() {

    int numero = 50;
    mostrar_ptr_void(&numero);
    return 0;
}
```


Ponteiros para Subrotinas

- A partir de ponteiros, é possível adicionar novas funcionalidades a uma subrotina sem alterar sua implementação original;
 - Isso é conhecido como **injeção de dependência**;
- Uma subrotina pode receber como parâmetro uma outra subrotina a partir da utilização de ponteiros;
- A subrotina recebida como parâmetro estende a funcionalidade da subrotina original ao executar uma funcionalidade específica;
- Tem-se total generalização quando usa-se ponteiros do tipo **void**;
- Em C, temos o exemplo principal da subrotina para ordenação de *arrays* **qsort**.

Ponteiros para Subrotinas: Exemplo 1

```
#include <stdio.h>
#include <stdbool.h>

bool is_freezing_celsius();
bool is_freezing_fahrenheit();

// declara uma subrotina que recebe um ponteiro
// como parâmetro de uma outra subrotina, que retorna
// um valor do tipo booleano.
void read_temperature(bool (*is_freezing)()) {
    if (is_freezing())
        printf("IS FREEZING!");
    else
        printf("Is not freezing.");
}

int main() {
    // a subrotina 'read_temperature' pode
    // receber tanto 'is_freezing_fahrenheit'
    // quanto 'is_freezing_celsius' como argumento.
    read_temperature(is_freezing_fahrenheit);
    return 0;
}
```



Type the temperature in Fahrenheit: 32
IS FREEZING!

Ponteiros para Subrotinas: Exemplo 2

```
void qsort(void *base, size_t num_elems, size_t size_elem, int (*compar)(const void *, const void *));
```

```
int crescente(const void *num1, const void *num2) {
    int *n1 = (int *)num1;
    int *n2 = (int *)num2;
    return *n1 - *n2;
}

int decrescente(const void *num1, const void *num2) {
    int *n1 = (int *)num1;
    int *n2 = (int *)num2;
    return *n2 - *n1;
}

int main() {

    int lista[] = {8, 7, 4, 5, 1, 0, 12, -1, 9, 2};
    int num_elementos = sizeof(lista) / sizeof(int);

    // ordena a lista em ordem 'crescente' ou 'decrescente'.
    qsort(lista, num_elementos, sizeof(int), crescente);

    for (int i = 0; i < num_elementos; i++)
        printf("%d ", lista[i]);

    return 0;
}
```



-1 0 1 2 4 5 7 8 9 12

Aritmética de Ponteiros

- A aritmética de ponteiros é uma técnica em linguagens de programação do C que permite **manipular ponteiros usando operadores aritméticos**, como adição, subtração e desreferenciamento;
- Isso é útil para navegar em *arrays* e estruturas de dados de forma eficiente e conveniente.

```
int main() {  
  
    int array[] = {10, 20, 30, 40, 50};  
    int *ptr = array; // ptr aponta para o primeiro elemento do array  
  
    ptr++; // Incrementa ptr para apontar para o próximo elemento  
    printf("%d\n", *ptr); // Saída: 20  
  
    ptr--; // Decrementa ptr para apontar para o elemento anterior  
    printf("%d\n", *ptr); // Saída: 10  
  
    int *ptr1 = &array[0]; // ptr1 aponta para o primeiro elemento do array  
    int *ptr2 = &array[3]; // ptr2 aponta para o quarto elemento do array  
  
    // Saída: número de elementos entre os ponteiros  
    printf("%ld\n", ptr2 - ptr1);  
  
    return 0;  
}
```

Laboratório de Fundamentos em TIC

Ponteiros em C

Prof. Gabriel Resende Machado



gabrielmachado@unifeso.edu.br



<https://www.linkedin.com/in/machadogabriel>



<https://github.com/UNIFESO-Gabriel/fundamentos-em-tic>