Recursividade

Programação de Computadores 1

Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

INSTITUTO FEDERAL
Brasília

Sumário

Sumário

Introdução

- A indução matemática é um artifício poderoso para demonstrar propriedades sobre os números naturais.
- Ela consiste em primeiro, demonstrar que o **caso base**, vale. Normalmente provamos que a propriedade vale para $n = 1$.
- Então, assumimos que a propriedade vale para todo $k \leq n$. Isto é conhecido como a **hipótese de indução**.
- Finalmente, se utilizando a hipótese de indução, conseguirmos provar que a propriedade vale para $n + 1$, então ela valerá para todos os naturais. Este último passo é conhecido como **passo de indução**.

Efeito dominó

Figura: https://www.snexplores.org/article/falling-dominoes-speed-friction-physics

Soma dos n primeiros naturias

- Vamos tomar como exemplo a soma dos n primeiros naturais.
- Iremos provar que a soma dos n primeiros naturais, S_n é $S_n = \frac{n+n^2}{2}$.

Soma dos n primeiros naturais

Teorema (Soma dos n primeiros naturais)

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

Soma dos n primeiros naturais

Demonstração

Primeiramente mostraremos o **caso base**.

$$S_1 = \frac{1 + 1^2}{2} = \frac{2}{2} = 1$$

Soma dos n primeiros naturais

Demonstração

Agora vamos assumir que $S_k = \frac{k+k^2}{2}$ para todo $k \leq n$, nossa **hipótese de indução**.

Soma dos n primeiros naturais

Demonstração

Para finalizar, temos que mostrar que $S_{n+1} = \frac{(n+1)+(n+1)^2}{2}$.

Sabemos que a soma dos primeiros $n+1$ naturais é igual a soma dos n primeiros naturais com $n+1$. Em outras palavras, temos que

$$S_{n+1} = S_n + (n+1)$$

Soma dos n primeiros naturais

Demonstração

Como sabemos, pela **hipótese de indução**, que $S_n = \frac{n+n^2}{2}$.
Substituindo:

$$
\begin{aligned}
S_{n+1} &= \frac{n+n^2}{2} + (n+1) \\
&= \frac{n+n^2+2(n+1)}{2} \\
&= \frac{n+n^2+2n+2}{2} \\
&= \frac{(n+1)+n^2+2n+1}{2} \\
&= \frac{(n+1)+(n+1)^2}{2} \quad \square
\end{aligned}
$$

Indução matemática

- Provamos o que queríamos apenas:
  - Provando o caso base.
  - Assumindo que a propriedade vale para todo $k \leq n$.
  - Utilizando a hipótese de indução, mostramos que a propriedade vale para $n + 1$.
- Simplificação do processo de raciocínio. Não precisamos tudo diretamente, basta mostrar que o efeito dominó segue.
- O desafio é mostrar como S_{n+1} pode ser descrito em termos da hipótese de indução.

Recursividade

Podemos utilizar um argumento análogo ao da indução matemática para a programação:

- Mostramos como resolver um caso simples o suficiente (análogo ao caso base).
- Colocamos a solução de um problema sob uma determina entrada em função **da mesma solução** aplicada a uma entrada menor (análogo à hipótese de indução).
- **Recursividade:** Como se trata da mesma solução, existe uma invocação de uma função dentro da mesma.

Sumário

Fatorial

- Vamos usar como exemplo o problema do cálculo do fatorial:

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1$$

- Lembrando que $0! = 1$.

Fatorial

- Modelando em termos da recursão, precisamos definir o **caso base**: quando $n = 0$ ou $n = 1$ a resposta é 1.
- Agora só precisamos modelar a solução de $n!$ em função da solução do mesmo problema, mas aplicado a uma entrada menor.
- Felizmente sabemos que $n! = n \cdot (n-1)!$
- Assim temos que:

$$n! = \begin{cases} 1, & n = 0 \vee n = 1 \\ n \cdot (n-1)!, & n > 1 \end{cases}$$

Fatorial

```c
long int fat(long int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fat(n - 1);
    }
}
```

Fatorial

- Note como, para $n > 1$, a solução de `fat(n)` é escrita em função da solução de `fat(n-1)`.
- Podemos deixar o código um pouco mais compacto:

```
long int fat(long int n) { return n <= 1 ? 1 : n * fat(n - 1); }
```

Recursividade

- A solução obtida é elegante, compacta e clara.
- Permite projetar algoritmos de uma maneira bela e precisa, especialmente quando a solução do problema pode ser modelada em termos de si mesma.

Sumário

Memória

- O que acontece quando invocamos uma função?

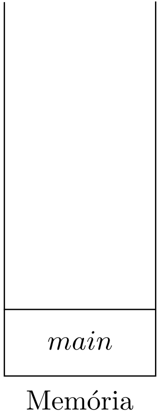
Memória

- O que acontece quando invocamos uma função?
- Os parâmetros dela, juntamente com suas variáveis locais, são empilhados na memória de pilha!

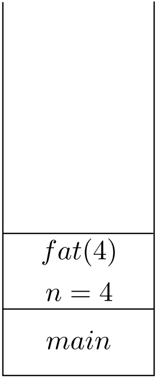
Memória

- O que acontece quando invocamos uma função?
- Os parâmetros dela, juntamente com suas variáveis locais, são empilhados na memória de pilha!
- Ao finalizar, os parâmetros e variáveis locais são desempilhados e o retorno da função é dado no ponto em que ela foi invocada.

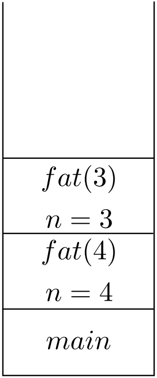
Memória



Memória

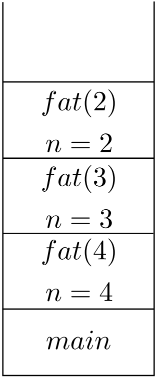


Memória



Memória

Memória



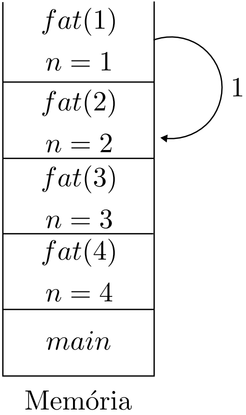
Memória

Memória

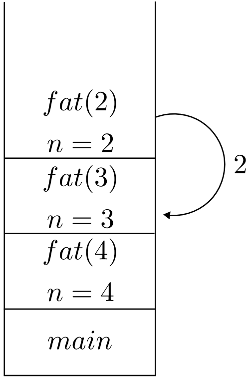
$fat(1)$ $n = 1$
$fat(2)$ $n = 2$
$fat(3)$ $n = 3$
$fat(4)$ $n = 4$
$main$

Memória

Memória

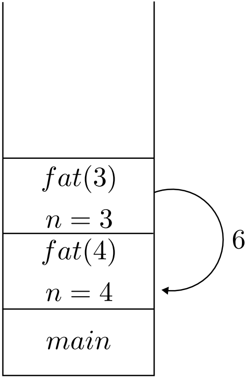


Memória



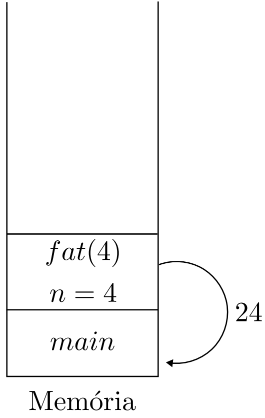
Memória

Memória

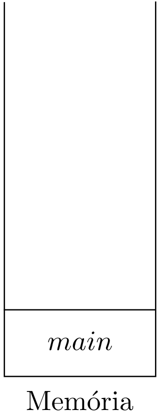


Memória

Memória



Memória



Memória

- Como a memória de pilha tem um limite padrão, 8MB geralmente em sistemas GNU/Linux, existe uma quantidade máxima de chamadas recursivas que pode ser realizada.
- Se o limite é excedido, normalmente nos deparamos com um erro chamado *stack overflow* (estouro de pilha).

Sumário

Recursão x iteração

- Soluções recursivas são mais compactas que as iterativas.
- Soluções recursivas são naturais quando o problema pode ser estruturado recursivamente.
- Soluções iterativas não gastam tempo empilhando funções, parâmetros e variáveis locais na memória de pilha.
- Soluções iterativas, a princípio, não possuem restrições em relação ao número de chamadas.

Sumário

Sumário

Soma de um vetor

Problema

Dado um vetor V de tamanho n, retornar a soma de um vetor através de um algoritmo recursivo

Soma de um vetor

Para resolver este problema podemos formular a seguinte estratégia recursiva:

- Um vetor de tamanho 0 possui soma 0 (caso base).
- A soma de um vetor de tamanho n é igual a soma de um vetor de tamanho $n-1$ (hipótese de indução) adicionado ao elemento $V[n-1]$ (passo de indução).

Soma de um vetor

- Em outras palavras, temos:

$$soma(V, n) = \begin{cases} 0, & n = 0 \\ V[n-1] + soma(V, n-1), & n > 0 \end{cases}$$

Soma de um vetor

```c
#include <stdio.h>

int soma_vetor(int *v, int n) {
    if (n == 0) {
        return 0;
    }
    return soma_vetor(v, n - 1) + v[n - 1];
}

int main(){
    int v[] = {1,2,3,4,5};
    printf("Soma = %d\n",soma_vetor(v,5));
    return 0;
}
```

Sumário

Inverso de uma string

Problema

Dado uma string S, imprimir o inverso dela, sem precisar invertê-la.

Inverso de uma string

- Podemos utilizar a recursão para avançar na string e imprimir na ordem em que as funções são desempilhadas.
- Caso base: se chegarmos ao fim da string não imprimimos nada.
- Passo de indução: avançamos para o próximo caractere e, apenas após imprimi-lo, imprimimos o caractere corrente.

Inverso de uma string

```c
#include <stdio.h>

void imprime_inverso_string(const char* str, size_t i){
    if(str[i]=='\0')
        return;
    imprime_inverso_string(str,i+1);
    printf("%c",str[i]);
}

int main(void){
    char* str = "abracadabra";
    imprime_inverso_string(str,0);
    return 0;
}
```

Sumário

3 Exemplos
- Soma de um vetor
- Inverso de uma string
- Busca linear
- Fibonacci

Busca linear

Problema

Dado um vetor de inteiros V, de tamanho n e um elemento k, retornar a posição em que k ocorre em V. Se k não ocorre em V, -1 deve ser retornado.

Busca linear

- Caso base: a busca em um vetor vazio retorna -1.
- Passo de indução: se o elemento a ser buscado está na posição i então, retorne a posição i, senão, proceda recursivamente para a posição $i+1$.

Busca linear

```c
#include <stdio.h>

int busca_linear(int *v, size_t n, size_t i, int k) {
    if (i == n)
        return -1;
    return v[i] == k ? i : busca_linear(v, n, i + 1, k);
}

int main(void){
    int v[] = {1,5,2,3,4};
    printf("%d\n",busca_linear(v,5,0,3));
    printf("%d\n",busca_linear(v,5,0,6));
    return 0;
}
```

Sumário

Fibonacci

Problema

Dado um inteiro n, calcular o n-ésimo número da sequência de Fibonacci.

Fibonacci

- A sequência de Fibonacci naturalmente possui uma caracterização recursiva:

$$fib(n) = \begin{cases} 1, & n \leq 2 \\ fib(n-1) + fib(n-2), & n > 2 \end{cases}$$

Fibonacci

```c
long int fib(int n) { return n <= 2 ? 1 : fib(n - 1) + fib(n - 2); }
```

Fibonacci

- O problema dessa recursão é que ela explode exponencialmente.
- Cada chamada gera duas chamadas recursivas no caso geral.
- Consequência: muito tempo ou memória de pilha excedida.

Fibonacci

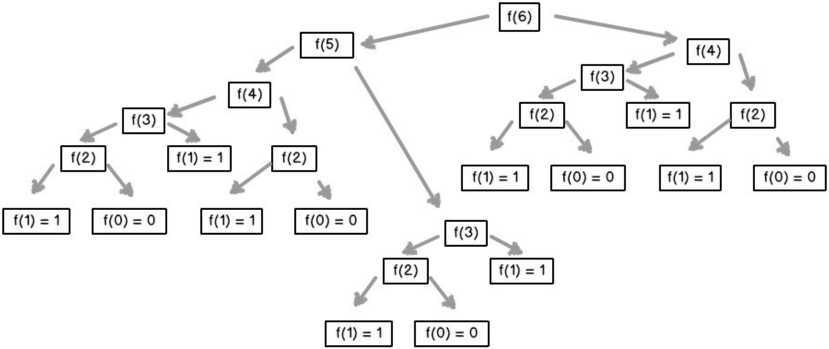


Figura: `https://medium.com/launch-school/recursive-fibonnaci-method-explained-d82215c5498e`

Fibonacci

- Podemos fazer melhor se passarmos os últimos dois termos computados por parâmetro.

Fibonacci

```
long int fib(int n, int cur, int prev) {
    if (n == 1)
        return prev;
    return fib(n - 1, cur + prev, cur);
}
```