SCC0214 – Projeto de Algoritmos

Recursão

Exercício

- Implemente uma função para calcular o fatorial de um número inteiro positivo

Definição

- Uma função é dita *recursiva* quando é definida em seus próprios termos, direta ou indiretamente
  - Dicionário Michaelis: *ato ou efeito de recorrer*
    - Recorrer: *Correr de novo por; tornar a percorrer. Repassar na memória; evocar.*
- É uma função como qualquer outra

3

Exercício

- Implemente uma função recursiva para calcular o fatorial de um número inteiro positivo

4

Exemplo

- Função que imprime os elementos de um vetor

```
void imprime(int v[], int tamanho) {
    int i;
    for (i=0;i<tamanho;i++)
        printf("%d ",v[i]);
}
```

5

Exercício

- Faça a versão recursiva da função

6

Exercício

- Faça a versão recursiva da função

```
void imprime(int v[], int tamanho, int indice_atual) {
    if (indice_atual<tamanho) {
       printf("%d ",v[indice_atual]);
       imprime(v,tamanho,indice_atual+1);
    }
}
```

7

Efeitos da recursão

- A cada chamada
  - Empilham-se na memória os dados locais (variáveis e parâmetros) e o endereço de retorno
    - A função corrente só termina quando a função chamada terminar
  - Executa-se a nova chamada (que também pode ser recursiva)
  - Ao retornar, desempilham-se os dados da memória, restaurando o estado antes da chamada recursiva

8

Exercício

- Simule a execução da função para um vetor de tamanho 3 e mostre a situação da memória a cada chamada recursiva

```
void imprime(int v[], int tamanho, int indice_atual) {
    if (indice_atual<tamanho) {
       printf("%d ",v[indice_atual]);
       imprime(v,tamanho,indice_atual+1);
    }
}
```

9

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {
       printf("%d ",v[0]);
       imprime1(v,...);
    }
}

void imprime1(int v[], ...) {
       printf("%d ",v[1]);
       imprime2(v,...);
    }
}
```

```
void imprime2(int v[], ...) {
       printf("%d ",v[2]);
       imprime3(v,...);
    }
}

...
```

10

Alternativa: tem o mesmo efeito?

```
void imprime0(int v[], ...) {
        printf("%d ",v[0]);
        imprime1(v,...);
    }
}

void imprime1(int v[], ...) {
        printf("%d ",v[1]);
        imprime2(v,...);
    }
}
```

```
void imprime2(int v[], ...) {
        printf("%d ",v[2]);
        imprime3(v,...);
    }
}

...
```

Mesmo resultado, com diferença de haver duplicação de código

11

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
  - O que isso quer dizer? Funções recursivas são sempre melhores do que funções não recursivas?

12

Efeitos da recursão

- *Mesmo resultado, com diferença de haver duplicação de código*
  - O que isso quer dizer? Funções recursivas são sempre melhores do que funções não recursivas?
    - Depende do problema, pois nem sempre a recursão é a melhor forma de resolver o problema, já que pode haver uma versão simples e não recursiva da função (que não duplica código e não consome mais memória)

13

Recursão

- Quando usar: quando o problema pode ser definido recursivamente de forma natural
- Como usar
  - 1º ponto: definir o problema de forma recursiva, ou seja, em termos dele mesmo
  - 2º ponto: definir a condição de término (ou *condição básica*)
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
    - Caso contrário, qual o problema?

14

Recursão

- Problema do fatorial
  - 1º ponto: definir o problema de forma recursiva
    - n! = n * (n-1)!
  - 2º ponto: definir a condição de término
    - n=0
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
    - A cada chamada, n é decrementado, ficando mais próximo da condição de término

15

Recursão vs. iteração

- Quem é melhor?

```
//versão recursiva
int fatorial(int n) {
   int fat;

   if (n==0) fat=1;
   else fat=n*fatorial(n-1);

   return(fat);
}
```

```
//versão iterativa
int fatorial(int n) {
   int i, fat=1;

   for (i=2;i<=n;i++)
      fat=fat*i;

   return(fat);
}
```

16

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
  - 1º ponto: definir o problema de forma recursiva
  - 2º ponto: definir a condição de término
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término

17

Exercício

- Implemente uma função recursiva para calcular o enésimo número de Fibonacci
  - 1º ponto: definir o problema de forma recursiva
    - f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2) para n>=2
  - 2º ponto: definir a condição de término
    - n=0 e/ou n=1
  - 3º ponto: a cada chamada recursiva, deve-se tentar garantir que se está mais próximo de satisfazer a condição de término
    - n é decrementado em cada chamada

18

Recursão vs. iteração

- Quem é melhor? Simule a execução

```
//versão recursiva
int fib(int n) {
   int resultado;

   if (n<2) resultado=n;
   else resultado=fib(n-1)+fib(n-2);

   return(resultado);
}
```

```
//versão iterativa
int fib(int n) {
   int i=1, k, resultado=0;

   for (k=1;k<=n;k++) {
      resultado=resultado+i;
      i=resultado-i;
   }

   return(resultado);
}
```

19

Recursão vs. iteração

- Quem é melhor? Simule a execução

```
//versão recursiva
int fib(int n) {
   int resultado;

   if (n<2) resultado=n;
   else resultado=fib(n-1)+fib(n-2);

   return(resultado);
}
```

Certamente mais elegante, mas duplica muitos cálculos!

```
//versão iterativa
int fib(int n) {
   int i=1, k, resultado=0;

   for (k=1;k<=n;k++) {
      resultado=resultado+i;
      i=resultado-i;
   }

   return(resultado);
}
```

20

Recursão vs. iteração

- Quem é melhor?
  - Estimativa de tempo para Fibonacci (Brassard e Bradley, 1996)

n	*10*	*20*	*30*	*50*	*100*
Recursão	8 ms	1 s	2 min	21 dias	10^9 anos
Iteração	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

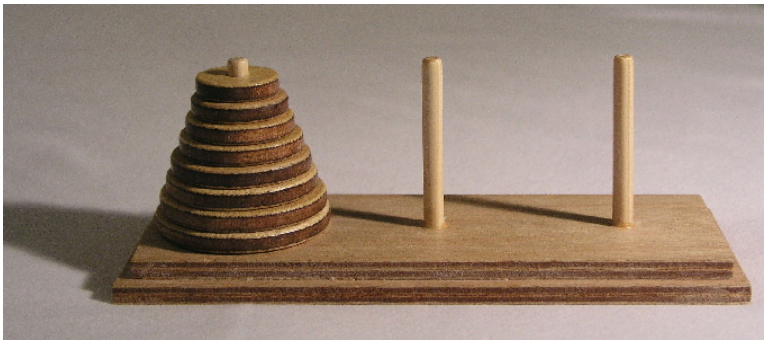
21

Recursão vs. iteração

- Programas recursivos que possuem chamadas ao final do código são ditos terem **recursividade de cauda**
  - São mais facilmente transformáveis em programas iterativos
    - A recursão pode virar uma condição

22

Torres de Hanoi



23

Torres de Hanoi

- Jogo
  - Tradicionalmente com 3 hastes: Origem, Destino, Temporária
  - Número qualquer de discos de tamanhos diferentes na haste Origem, dispostos em ordem de tamanho: os maiores embaixo
  - Objetivo: usando a haste Temporária, movimentar um a um os discos da haste Origem para a Destino, sempre respeitando a ordem de tamanho
    - Um disco maior não pode ficar sobre um menor!

24

Torres de Hanoi

- Simulação

25

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi

26

Torres de Hanoi

- Implemente uma função recursiva que resolva o problema das Torres de Hanoi
  - Mover n-1 discos da haste Origem para a haste Temporária
  - Mover o disco n da haste Origem para a haste Destino
  - Recomeçar, movendo os n-1 discos da haste Temporária para a haste Destino

27

Torres de Hanoi

```
#include <stdio.h>

void mover(int, char, char, char);

int main(void) {
    mover(3,'O','T','D');
    system("pause");
    return 0;
}

void mover(int n, char Orig, char Temp, char Dest) {
     if (n==1) printf("Mova o disco 1 da haste %c para a haste %c\n",Orig,Dest);
     else {
          mover(n-1,Orig,Dest,Temp);
          printf("Mova o disco %d da haste %c para a haste %c\n",n,Orig,Dest);
          mover(n-1,Temp,Orig,Dest);
     }
}
```

Torres de Hanoi

- Exercício para casa
  - Tente fazer a versão não recursiva do programa

29

Exercício

- Em duplas, para entregar
  - Escolha 1 deles para implementar e mostrar a "árvore de recursão" para um pequeno exemplo
    - Dado um arranjo v de n números inteiros, implemente uma função recursiva que retorne o maior elemento do arranjo
    - Dado um arranjo v de n números inteiros, implemente uma função recursiva que retorne a soma dos elementos do arranjo

30

Recursão

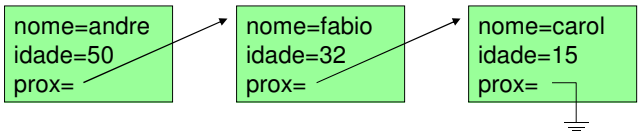
- Muito útil para lidar com estruturas de dados mais complexas
  - Listas sofisticadas, árvores, etc.

Exercício

- Imagine que você tem declarado vários blocos de memória para a seguinte estrutura

```
struct bloco {
        char nome[50];
        int idade;
        struct bloco *prox;
}
```

em que cada bloco aponta para o endereço do próximo bloco alocado. Por exemplo:



- Faça um programa que leia esses dados do usuário, armazenando-os nos blocos alocados dinamicamente, e, depois, chame uma função recursiva que imprima os dados armazenados