Recursividade

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

Licenciamento

Recursividade

Dividir e conquistar

Divida o problema original em subproblemas menores de mesma natureza e combine os resultados obtidos para chegar na solução do problema original

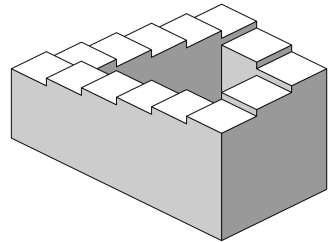
- A divisão de forma recursiva fará um problema grande ficar tão pequeno ao ponto da solução ser trivial, interrompendo a recursão
- Em algoritmos, a **chamada recursiva de função** é quando uma função invoca ela mesma
  - A solução de um problema em algoritmos usando a recursão é equivalente a **indução matemática**[1]

[1] https://pt.wikipedia.org/wiki/Indu%C3%A7%C3%A3o_matem%C3%A1tica

Indução matemática

Exemplo apresentado em (GERSTING, 2016)

- Você está subindo uma escada infinitamente alta
- Como você sabe se será capaz de chegar a um degrau arbitrariamente alto?



Fonte: Wikipedia

Assuma as seguintes hipóteses

1. Você consegue alcançar o primeiro degrau
2. Uma vez chegando a um degrau, você sempre é capaz de chegar ao próximo (é uma condicional)

Indução matemática

Exemplo apresentado em Gersting (2016)

- **Se a proposição 1 e a condicional 2 forem ambos verdadeiros, então**
  - pela proposição 1 você consegue chegar ao primeiro degrau
  - pela proposição 2 você consegue chegar ao segundo degrau
  - novamente, pela proposição 2 você consegue chegar ao terceiro degrau
  - novamente, pela proposição 2 você consegue chegar ao quarto degrau
  - e assim por diante

Indução matemática

Exemplo apresentado em Gersting (2016)

- **Se a proposição 1 e a condicional 2 forem ambos verdadeiros, então**
  - pela proposição 1 você consegue chegar ao primeiro degrau
  - pela proposição 2 você consegue chegar ao segundo degrau
  - novamente, pela proposição 2 você consegue chegar ao terceiro degrau
  - novamente, pela proposição 2 você consegue chegar ao quarto degrau
  - e assim por diante

- Se apenas a proposição 1 for verdadeira, você não teria garantia que passaria do 1º degrau
- Se apenas a proposição 2 for verdadeira, você nunca seria capaz de começar a subida

Indução matemática

Exemplo apresentado em Gersting (2016)

- Em vez de “chegar a um degrau arbitrariamente alto” falamos sobre “inteiro positivo arbitrário tendo essa propriedade *P*”
  - $P(n)$ para indicar que um inteiro positivo `n` tem a propriedade `P`

Como usar a mesma técnica que usamos para subir a escada para provar que, qualquer que seja o inteiro positivo n, temos P(n)?

Indução matemática

Exemplo apresentado em Gersting (2016)

- Em vez de “chegar a um degrau arbitrariamente alto” falamos sobre “inteiro positivo arbitrário tendo essa propriedade P”
  - $P(n)$ para indicar que um inteiro positivo `n` tem a propriedade `P`

Como usar a mesma técnica que usamos para subir a escada para provar que, qualquer que seja o inteiro positivo n, temos P(n)?

1.	$P(1)$ é verdade	1 tem a propriedade `P`
2.	$\forall k, P(k) \textit{verdade} \rightarrow P(k+1) \textit{verdade}$	Se qualquer número tem a propriedade P, então o próximo também tem

Se 1 e 2 puderem ser provadas, então $P(n)$ será válida para qualquer inteiro positivo `n`

Indução matemática

Demonstração por indução

- O estabelecimento da veracidade da proposição 1 é chamado de **base da indução** ou **passo básico** da demonstração por indução
- O estabelecimento da veracidade de $P(k) \rightarrow P(k+1)$ é o **passo indutivo**
- Quando supomos que $P(k)$ é verdade para provar o passo indutivo, $P(k)$ é chamada de **hipótese de indução**

Recursividade

Exemplo: multiplicação de números inteiros positivos

- A multiplicação de `m` por `n` consiste em somar `m`, `n` vezes

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}} \tag{1}$$

Recursividade

Exemplo: multiplicação de números inteiros positivos

- A multiplicação de `m` por `n` consiste em somar `m`, `n` vezes

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}} \tag{1}$$

1. **base da indução**
   - mostrar que o enunciado é válido para $n = 0$ ou $n = 1$ (a depender do enunciado)
2. **passo indutivo**
   - mostrar que, se o enunciado vale para $n = k$, então também vale para $n = k + 1$

Recursividade

Exemplo: multiplicação de números inteiros positivos

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}}$$

- **base da indução**
  - $m \times n = 0$, se $n == 0$
- **passo indutivo**
  - $m \times n = m + (m \times (n - 1))$, se $n > 0$

$$m \times n = \begin{cases} m \times n & \text{se } n == 0 \\ m \times n = m + (m \times (n - 1)) & \text{se } n > 0 \end{cases} \quad (2)$$

Recursividade

1. Defina a solução para a **base da indução** (onde não há mais necessidade de recursão)
2. Defina a solução para o **passo indutivo** (fazendo uso de soluções para entradas menores do problema)

Recursividade

Exemplo: multiplicação de números inteiros positivos em C

- **Implementação iterativa**

```c
int mult_ite(int m, int n){
    int resultado = 0;
    for(int i=0; i < n; i++){
        resultado += m;
    }
    return resultado;
}
```

Recursividade

Exemplo: multiplicação de números inteiros positivos em C

Implementação iterativa

```
1 int mult_ite(int m, int n){
2     int resultado = 0;
3     for(int i=0; i < n; i++){
4         resultado += m;
5     }
6     return resultado;
7 }
```

Implementação recursiva

```
1 int mult_rec(int m, int n){
2     if (n == 0){
3         return 0;
4     }
5     return (m + mult_rec(m, n-1));
6 }
```

$$m \times n = \begin{cases} m \times n & \text{se } n == 0 \\ m \times n = m + (m \times (n-1)) & \text{se } n > 0 \end{cases}$$

Multiplicação de números inteiros positivos em C

Implementação iterativa

```c
int mult_ite(int m, int n){
    int r = 0;
    for(int i=0; i < n; i++){
        r += m;
    }
    return r;
}

int main(){
    int resultado = mult_ite(2, 3);
}
```

Teste de mesa

#	i	r	m	n	Total de variáveis
-	-	0	2	3	4

Multiplicação de números inteiros positivos em C

Implementação iterativa

```c
int mult_ite(int m, int n){
    int r = 0;
    for(int i=0; i < n; i++){
        r += m;
    }
    return r;
}

int main(){
    int resultado = mult_ite(2, 3);
}
```

Teste de mesa

#	i	r	m	n	Total de variáveis
-	-	0	2	3	4
1	0	2	2	3	4

Multiplicação de números inteiros positivos em C

Implementação iterativa

```c
int mult_ite(int m, int n){
    int r = 0;
    for(int i=0; i < n; i++){
        r += m;
    }
    return r;
}

int main(){
    int resultado = mult_ite(2, 3);
}
```

Teste de mesa

#	i	r	m	n	Total de variáveis
-	-	0	2	3	4
1	0	2	2	3	4
2	1	4	2	3	4

Multiplicação de números inteiros positivos em C

Implementação iterativa

```c
int mult_ite(int m, int n){
    int r = 0;
    for(int i=0; i < n; i++){
        r += m;
    }
    return r;
}

int main(){
    int resultado = mult_ite(2, 3);
}
```

Teste de mesa

#	i	r	m	n	Total de variáveis
-	-	0	2	3	4
1	0	2	2	3	4
2	1	4	2	3	4
3	2	6	2	3	4

Multiplicação de números inteiros positivos em C

Implementação iterativa

```c
int mult_ite(int m, int n){
    int r = 0;
    for(int i=0; i < n; i++){
        r += m;
    }
    return r;
}

int main(){
    int resultado = mult_ite(2, 3);
}
```

Teste de mesa

#	i	r	m	n	Total de variáveis
-	-	0	2	3	4
1	0	2	2	3	4
2	1	4	2	3	4
3	2	6	2	3	4
4	3	6	2	3	4

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
2	`mult(2, 2)`	4 variáveis
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
3	`mult(2, 1)`	6 variáveis
2	`mult(2, 2)`	4 variáveis
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
4	`mult(2, 0)`	8 variáveis
3	`mult(2, 1)`	6 variáveis
2	`mult(2, 2)`	4 variáveis
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
4	**0**	
3	`mult(2, 1)`	6 variáveis
2	`mult(2, 2)`	4 variáveis
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
3	**2**	
2	`mult(2, 2)`	4 variáveis
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

#	Chamada	
2	**4**	
1	`mult(2, 3)`	2 variáveis

Multiplicação de números inteiros positivos em C

Implementação recursiva

```
1  int mult(int m, int n){
2      if (n == 0){
3          return 0;
4      }
5      return (m + mult(m, n-1));
6  }
7
8  int main(){
9      int resultado = mut(2, 3);
10 }
```

Execução

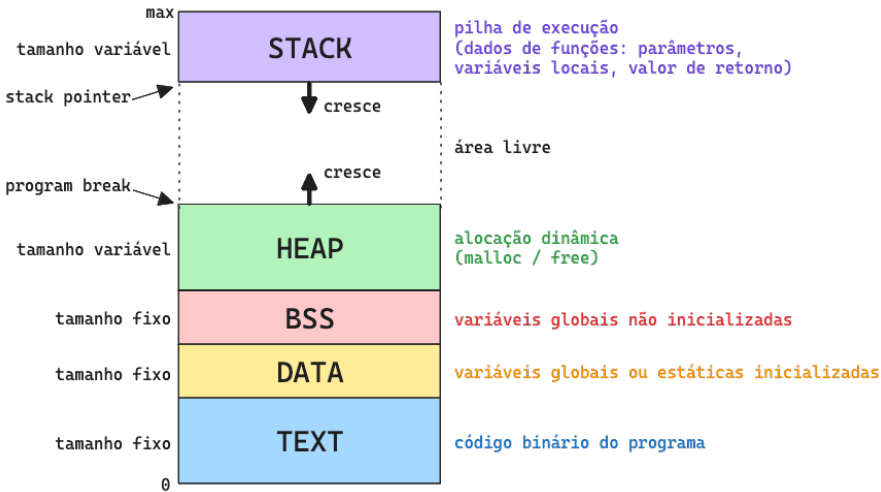
#	Chamada
1	**6**

Recursividade

Vantagens e desvantagens

- **Vantagens**
  - Código mais conciso
  - Código mais claro para problemas de natureza recursiva
- **Desvantagens**
  - Tempo de execução maior se comparado com algoritmos iterativos
  - Dificuldade na depuração em chamadas recursivas profundas
  - Consomem mais memória (chamadas ficam empilhadas até chegar na base) que funções iterativas

Representação da memória de um programa em C



Adaptado de (MAZIERO, 2020)

Alocação de memória

Linguagem C trabalha com alocação estática, automática e dinâmica de memória

- **Alocação estática** (áreas `DATA` ou `BSS`)
  - Espaço da variável é reservado pelo compilador
  - Variáveis globais e variáveis locais estáticas (`static int`)
- **Alocação automática** (área `STACK`)
  - Variáveis locais, parâmetro de funções e valor de retorno
  - Área alocada ao invocar função e liberada ao término da função (conveniente por chamadas recursivas)
- **Alocação dinâmica** (área `HEAP`)
  - Requisitado explicitamente pelo programa
  - Tamanho delimitado pelo ponteiro *program break* (BRK)
    - Memória disponível é exaurida quando o *stack pointer* encontra o *program break*

Stack Overflow e Heap Overflow

- **Stack Overflow**
  - Declaração de grande número de variáveis
  - Vetor muito grande
  - Quando uma **chamada recursiva de função** é muito profunda (cada chamada aloca variáveis locais, parâmetros, etc)
- **Heap Overflow**
  - Quando aloca um grande número de variáveis
  - Quando aloca memória continuamente sem fazer a liberação após o uso
    - *memory leak* – memória continua alocada e não fica disponível para os demais processos

Experimento de alocação de memória na pilha

Créditos: prof. Marcelo Sobral

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

void aloca_memoria_pilha(int quantidade){
    int tamanho = quantidade * (1<<20);
    char memoria[tamanho];
    printf("Vetor de tamanho: %d kB\n", tamanho/1024);
    printf("Endereço da variável tamanho: %p\n",(void*)&tamanho);
    printf("Distância entre vetor e tamanho: %ld\n", (((char*)&tamanho-memoria)/1024));
    bzero(memoria, tamanho);
    printf("zerou memória\n");
}

// passe como argumentos valores como 2, 4, 6, 8, 16, 32, 64, 128 e 1000
int main(int argc, char *argv[]) {
    if (argc > 0)
        aloca_memoria_pilha(strtol(argv[1], NULL, 10));
    return 0;
}
```

Experimento de alocação de memória no heap

Créditos: prof. Marcelo Sobral

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

void aloca_memoria_heap(int quantidade){
    int tamanho = quantidade * (1<<20);
    char *memoria = calloc(tamanho, sizeof (char));
    printf("Vetor de tamanho: %d kB\n", tamanho/1024);
    printf("Endereço da variável tamanho: %p\n",(void*)&tamanho);
    printf("Distância entre vetor e tamanho: %ld\n", (((char*)&tamanho-memoria)/1024));
    bzero(memoria, tamanho);
    printf("zerou memória\n");
    free(memoria);
}
// passe como argumentos valores como 2, 4, 6, 8, 16, 32, 64, 128 e 1000
int main(int argc, char *argv[]) {
    if (argc > 0)
        aloca_memoria_heap(strtol(argv[1], NULL, 10));
    return 0;
}
```

Curiosidade

- *Stack Overflow*[2] é um site de perguntas e respostas na área de programação de computadores
- O nome foi escolhido após uma votação[3] e é uma referência para quando um programa consome toda memória disponível para sua execução

[2] https://stackoverflow.com
[3] https://blog.codinghorror.com/help-name-our-website/

Exercícios

Exercício 1: Calcular o fatorial de um número inteiro positivo

Desenvolva um algoritmo interativo e um algoritmo recursivo

- $0! = 1$
- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$
- $4! = 4 \times 3 \times 2 \times 1$

1. **Qual é a base da indução?**
2. **Qual é o passo indutivo?**

Exercício 1: Calcular o fatorial de um número inteiro positivo

Desenvolva um algoritmo interativo e um algoritmo recursivo

- $0! = 1$
- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$
- $4! = 4 \times 3 \times 2 \times 1$

1. **Qual é a base da indução?**
   - $0! = 1$
2. **Qual é o passo indutivo?**
   - $n! = n \times (n-1)!$

Exercício 2: Sequência de Fibonacci de tamanho `n`

Desenvolva um algoritmo interativo e um algoritmo recursivo

- Os dois primeiros elementos da sequência de Fibonacci são 0 e 1
- Os elementos subsequentes são resultados da soma dos dois elementos que os precedem na sequência
- Para $n = 8$, a sequência é `0, 1, 1, 2, 3, 5, 8 e 13`

1. **Qual é a base da indução?**
2. **Qual é o passo indutivo?**

Exercício 3: Torre de Hanói com `n` discos

Desenvolva um algoritmo interativo e um algoritmo recursivo

- Mova todos os discos do pino de uma extremidade para o pino da outra extremidade
- Apenas o disco do topo de um pino pode ser movido
- Um disco maior não pode ficar em cima de um disco menor

Crédito: imagem de Casa de Sabiá

Exercício 4

1. Faça a soma de todos os dígitos de um número inteiro positivo
   - Em C, a divisão de um número inteiro por um número inteiro sempre resulta na parte inteira
   - O operador módulo (%) pega o resto da divisão

Ferramentas para ajudar no desenvolvimento em C

Valgrind

- Ferramenta[4] que pode ajudá-los na escrita de algoritmos mais eficientes e corretos
- Pode ser usada para encontrar **vazamento de memória** e acesso a **posições inválidas de memória**
  - Casos típicos quando se está trabalhando com alocação dinâmica de memória e vetores
  - Muito útil para achar a causa de um *Segmentation fault*
- Pode ser usado de forma integrada com o CLion[5], mas é necessário que tenha o valgrind instalado em sua máquina

```
# gerando binário com informações de depuração. Parâmentro -g
gcc -g main.c -o saida

valgrind --leak-check=yes ./saida
```

[4] https://valgrind.org/docs/manual/quick-start.html
[5] https://www.jetbrains.com/help/clion/memory-profiling-with-valgrind.html

GCC

Parâmetros adicionais para encontrar vazamento de memória

```
# -Wall para apresentar todos avisos (warning) ao compilar
# -g inclui informações para depuração (-g0 a -g3)
# -fsanitize=address para detectar diferentes erros referentes a acesso a memória

gcc -Wall -g -fsanitize=address main.c -o saida
```

```c
// Adaptado de https://valgrind.org/docs/manual/quick-start.html
#include <stdlib.h>

void f(void){
    int* x = malloc(100 * sizeof(int));
    x[100] = 0; // problema 1: acesso a posição inválida
}               // problema 2: vazamento de memória - x não é liberado

int main(void){
    for(int i=0; i < 1000; i++){
        f();
    }
    return 0;
}
```

Referências

- CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. LTC, 2012. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788595158092>.
- GERSTING, Judith L. **Fundamentos Matemáticos para a Ciência da Computação**. GEN, 2016. ISBN 9788521633303. Disponível em: <https://app.minhabiblioteca.com.br/#/books/9788521633303>.
- MAZIERO, Carlos. **Sistemas Operacionais: Conceitos e Mecanismos**. Ago. 2020. ISBN 978-85-7335-340-2. Disponível em: <https://wiki.inf.ufpr.br/maziero/doku.php?id=socm:start>.