Recursividade

Prof. Luiz Gustavo Almeida Martins

Definições

- **Definição de recursão / recursividade:**
  - **Matemática:** o ato de definir um objeto (geralmente uma função) em termos do próprio objeto

Definições

- **Definição de recursão / recursividade:**
  - **Matemática:** o ato de definir um objeto (geralmente uma função) em termos do próprio objeto
  - **Computação:** o ato de um algoritmo chamar a si mesmo para resolver um dado problema
    - Um ou mais passos do algoritmo invocam sua repetição
    - A recursão pode ser **direta ou indireta**

Definições

- **Definição de recursão / recursividade:**
  - **Matemática:** o ato de definir um objeto (geralmente uma função) em termos do próprio objeto
  - **Computação:** o ato de um algoritmo chamar a si mesmo para resolver um dado problema
    - Um ou mais passos do algoritmo invocam sua repetição
    - A recursão pode ser **direta ou indireta**
- Um **procedimento** que utiliza-se da recursão é dito **recursivo**
  - Qualquer **objeto** resultante desse tipo de procedimento também é dito **recursivo**

Definições

- A **recursão** pode ser usada para obter **sequências infinitas** a partir de **componentes finitos**
  - Permite pensar no "**passo chave**" **da resolução** sem a necessidade de integrá-lo aos demais

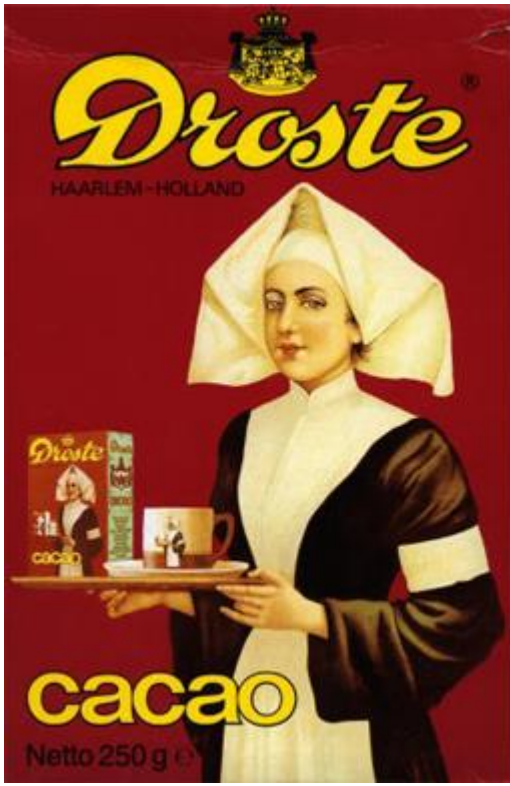
- **Exemplo:**

O conjunto dos números naturais pode ser definido por:

- Seja 0 um número natural

- Cada número natural ***n*** tem um sucessor ***n + 1***, o qual também é um número natural.

Recursividade visual



Fonte: http://beachpackagingdesign.com/boxvox/droste-effect-p

Recursão e computação

- **Propriedades dos problemas com estrutura recursiva:**
  - Só sabe resolver o caso mais simples (**solução trivial**)
  - Cada instância do problema é resolvida a partir de uma instância menor do mesmo problema

Recursão e computação

- **Propriedades dos problemas com estrutura recursiva:**
  - Só sabe resolver o caso mais simples (**solução trivial**)
  - Cada instância do problema é resolvida a partir de uma instância menor do mesmo problema

- **Resolução de problemas recursivos:**
  - Se a instância é **simples**, resolva-a diretamente

Recursão e computação

- **Propriedades dos problemas com estrutura recursiva:**
  - Só sabe resolver o caso mais simples (**solução trivial**)
  - Cada instância do problema é resolvida a partir de uma instância menor do mesmo problema

- **Resolução de problemas recursivos:**
  - Se a instância é **simples**, resolva-a diretamente
  - Senão:
    - Reduza-a a uma **instância menor** do mesmo problema
    - Aplique o método para essa instância menor
    - Utilize o resultado para responder a instância original

Recursão e computação

- **Propriedades dos problemas com estrutura recursiva:**
  - Só sabe resolver o caso mais simples (**solução trivial**)
  - Cada instância do problema é resolvida a partir de uma instância menor do mesmo problema

- **Resolução de problemas recursivos:**
  - Se a instância é **simples**, resolva-a diretamente
  - Senão:
    - Reduza-a a uma **instância menor** do mesmo problema
    - Aplique o método para essa instância menor
    - Utilize o resultado para responder a instância original

- O uso dessa estratégia produz um **algoritmo recursivo**
  - Caracterizado por possuir uma chamada a si mesmo

Etapas de implementação

- **1ª etapa:** definir o **caso base**
  - Parte **não recursiva**
  - Identifica quando a "jornada" já terminou
    - **Critério de parada (caso mais simples)**
  - Instância do problema que tem um **solução trivial**
  - Essa etapa é **muito importante** para evitar **recursividade infinita**

Etapas de implementação

- **2ª etapa:** definir o **passo recursivo**

  - Parte **recursiva**

  - Determina como a solução de um problema é obtido a partir do seu precedente (problema menor)
    - **Solução geral ou genérica** para o problema

  - **Envolve:**
    - Analisar o problema e dividi-lo em **problemas menores**
    - Encontrar uma forma da função **chamar a si mesma**, passando como parâmetro esses problemas menores

Etapas de implementação

- **3ª etapa:** integrar o caso base e o passo recursivo
  - Utiliza uma estrutura condicional (**seleção**)
- **4ª etapa:** verificar a adequação da solução
  - **Envolve:**
    - Verificar a condição de término
    - Verificar a eficiência do algoritmo (memória e tempo)
      - **Árvores de recursão**

Exemplo 1

- **Cálculo do fatorial:**

  Dado um número inteiro positivo *n*, como implementar recursivamente uma função que retorne seu fatorial (*n*!)?

Exemplo 1

- **Cálculo do fatorial:**

  Dado um número inteiro positivo n, como implementar recursivamente uma função que retorne seu fatorial (n!)?

$$n! = \begin{cases} 1 & \text{se } n = 0 \text{ (}\textbf{caso base}\text{)} \\ n \times (n-1)! & \text{se } n > 0 \text{ (}\textbf{passo recursivo}\text{)} \end{cases}$$

Exemplo 1

- **Cálculo do fatorial:**

  Dado um número inteiro positivo *n*, como implementar recursivamente uma função que retorne seu fatorial (*n*!)?

$$n! = \begin{cases} 1 & \text{se } n = 0 \text{ (}\textbf{caso base}\text{)} \\ n \times (n-1)! & \text{se } n > 0 \text{ (}\textbf{passo recursivo}\text{)} \end{cases}$$

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Exemplo 2

- **Gramáticas de linguagens de programação:**
  - Na especificação de uma gramática, a estrutura de expressões pode ser modelada como:

<*expr*> *::=* <*numero*>

| (<*expr*> * <*expr*>)

| (<*expr*> + <*expr*>)

Exemplo 2

- **Gramáticas de linguagens de programação:**
  - Na especificação de uma gramática, a estrutura de expressões pode ser modelada como:

    <expr> ::= <numero>
    | (<expr> * <expr>)
    | (<expr> + <expr>)

  - A referência recursiva à ***<expr>*** permite a formação de **expressões complexas**
    - Uso de mais de um produto ou soma em uma única expressão
    - **Ex:** (5 * ((3 * 6) + 8))

Exemplo 2

- **Gramáticas de linguagens de programação:**
  - Na especificação de uma gramática, a estrutura de expressões pode ser modelada como:

    <expr> ::= <numero>

    | (<expr> * <expr>)

    | (<expr> + <expr>)

  - A referência recursiva à ***<expr>*** permite a formação de **expressões complexas**
    - Uso de mais de um produto ou soma em uma única expressão
    - **Ex:** (5 * ((3 * 6) + 8))
  - **Questões:**
    - Qual é o caso base e o passo recursivo?

Exemplo 2

- **Gramáticas de linguagens de programação:**
  - Na especificação de uma gramática, a estrutura de expressões pode ser modelada como:

    <expr> ::= <numero> **caso base**

    | *(<expr> * <expr>)*

    | *(<expr> + <expr>)* **passos recursivos**

  - A referência recursiva à ***<expr>*** permite a formação de **expressões complexas**
    - Uso de mais de um produto ou soma em uma única expressão
    - **Ex:** (5 * ((3 * 6) + 8))
  - **Questões:**
    - Qual é o caso base e o passo recursivo? O que eles representam?

Exemplo 3

- O que faz a função ***xyz( )*** ?

```
int xyz (int n)
{
   if (n == 0)
      return 0;
   return n + xyz (n-1);
}
```

Exemplo 3

- O que faz a função ***xyz( )*** ?

```
int xyz (int n)
{
   if (n == 0)
      return 0;
   return n + xyz (n-1);
}
```

Implementa o somatório: $\sum_{i=0}^{n} i$

Implementação de Recursividade

- O que acontece na chamada da função?

Implementação de Recursividade

- O que acontece na chamada da função?
  - **Passagem de parâmetros:**
    - Uma cópia local de cada argumento é criada

Implementação de Recursividade

- O que acontece na chamada da função?
  - **Passagem de parâmetros:**
    - Uma cópia local de cada argumento é criada
  - **Alocação e inicialização das variáveis locais:**
    - Variáveis declaradas e temporárias

Implementação de Recursividade

- O que acontece na chamada da função?
  - **Passagem de parâmetros:**
    - Uma cópia local de cada argumento é criada
  - **Alocação e inicialização das variáveis locais:**
    - Variáveis declaradas e temporárias
  - **Transferência do controle:**
    - Passagem do endereço de retorno
    - Desvio do controle para a função

Implementação de Recursividade

- O que acontece no retorno da função?

Implementação de Recursividade

- O que acontece no retorno da função?
  - **Recuperação do endereço de retorno:**
    - Uma cópia é realizada para um “**local seguro**”

Implementação de Recursividade

- O que acontece no retorno da função?
  - **Recuperação do endereço de retorno:**
    - Uma cópia é realizada para um "**local seguro**"
  - **Liberação da área de dados da função:**
    - Argumentos e variáveis locais

Implementação de Recursividade

- O que acontece no retorno da função?
  - **Recuperação do endereço de retorno:**
    - Uma cópia é realizada para um "**local seguro**"
  - **Liberação da área de dados da função:**
    - Argumentos e variáveis locais
  - **Devolução do controle:**
    - Desvio à rotina de chamada
    - Volta ao ponto imediatamente depois da chamada

Implementação de Recursividade

- Como é implementada uma recursão?

Implementação de Recursividade

- Como é implementada uma recursão?

  - O compilador implementa um procedimento recursivo através de uma **pilha**

Implementação de Recursividade

- Como é implementada uma recursão?

  - O compilador implementa um procedimento recursivo através de uma **pilha**

  - Armazena os **dados usados** pela função
    - Parâmetros, variáveis locais e endereço de retorno

Implementação de Recursividade

- **Fluxo de execução:**
  - A função começa a execução do seu primeiro comando cada vez que é chamada
  - **Novas e distintas** cópias dos parâmetros passados por valor e variáveis locais são criadas
  - A **posição** da chamada da função **é colocada em estado de espera**
  - O nível gerado recursivamente é executado

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

[4 * < 3 * { 2 * fatorial (1) } >]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
   else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

[4 * < 3 * { 2 * fatorial (1) } >]

[4 * < 3 * { 2 * 1 } >]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
   else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

[4 * < 3 * { 2 * fatorial (1) } >]

[4 * < 3 * { 2 * 1 } >]

[4 * < 3 * 2 >]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
   else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

[4 * < 3 * { 2 * fatorial (1) } >]

[4 * < 3 * { 2 * 1 } >]

[4 * < 3 * 2 >]

[4 * 6]

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
   else
        return n*fat(n-1);
}
```

Implementação de Recursividade

- **Exemplo do fluxo de execução:**

fatorial(4)

[4 * fatorial(3)]

[4 * < 3 * fatorial(2) >]

[4 * < 3 * { 2 * fatorial (1) } >]

[4 * < 3 * { 2 * 1 } >]

[4 * < 3 * 2 >]

[4 * 6]

24

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
   else
        return n*fat(n-1);
}
```

Recursão de cauda

- Em uma **recursão comum**, a cada chamada recursiva é necessário armazenar:
  - Cópia dos argumentos de entrada e variáveis locais
  - Posição do código onde foi feita a chamada para continuar posteriormente

Recursão de cauda

- Em uma **recursão comum**, a cada chamada recursiva é necessário armazenar:
  - Cópia dos argumentos de entrada e variáveis locais
  - Posição do código onde foi feita a chamada para continuar posteriormente

- Em uma **recursão de cauda**, a chamada recursiva é a **última operação** que deve ser executada

Recursão de cauda

- Em uma **recursão comum**, a cada chamada recursiva é necessário armazenar:
  - Cópia dos argumentos de entrada e variáveis locais
  - Posição do código onde foi feita a chamada para continuar posteriormente

- Em uma **recursão de cauda**, a chamada recursiva é a **última operação** que deve ser executada
  - Usa menos memória de pilha (**eficiência de memória**)
    - **Reduz** a quantidade de dados armazenados na pilha

Recursão de cauda

- Em uma **recursão comum**, a cada chamada recursiva é necessário armazenar:
  - Cópia dos argumentos de entrada e variáveis locais
  - Posição do código onde foi feita a chamada para continuar posteriormente
- Em uma **recursão de cauda**, a chamada recursiva é a **última operação** que deve ser executada
  - Usa menos memória de pilha (**eficiência de memória**)
    - **Reduz** a quantidade de dados armazenados na pilha
  - Torna a recursão mais rápida (**eficiência de tempo**)
    - Simplifica o empilhamento e desempilhamento

Recursão de cauda

- **Exemplo 1:**

  Faça uma função recursiva que retorne o endereço do nó de uma lista simplesmente encadeada que contenha o elemento informado

```
struct nodo {
        int info;
        struct nodo * prox;
 }

typedef struct nodo Lista;
```

```
Lista* get_nod(Lista *L, int elem)
{
   if (L == NULL || L->info == elem )
       return L;
  else
      return get_nod(L->prox, elem);
}
```

Recursão de cauda

- Facilita conversão entre versões (**recursiva ↔ iterativa**)

```
Lista* get_nod(Lista *L, int elem) {
    if (L == NULL || L->info == elem )
        return L;
   else
       return get_nod(L->prox, elem);
}
```

```
Lista* get_nod(Lista *L, int elem) {
   while (L != NULL && L->info != elem)
        L = L->prox;
   return L;
}
```

conversão

Recursão de cauda

- **Exemplo 2:** Função recursiva do fatorial

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Recursão comum

Recursão de cauda

- **Exemplo 2:** Função recursiva do fatorial

```
int fat(int n) {
    if (n == 1 || n == 0)
        return 1;
    else
        return n*fat(n-1);
}
```

Recursão comum

```
int fat_cauda(int n, int parcial) {
    if (n == 1 || n == 0)
        return parcial;
    else
        return fat_cauda(n-1, parcial*n);
}

int fat(int n) {
    return fat_cauda(n, 1);
}
```

Recursão de cauda

Recursão de cauda

- Conversão entre versões (**recursiva ↔ iterativa**)

```
int fat_cauda(int n, int parcial) {
    if (n == 1 || n == 0)
        return parcial;
    else
        return fat_cauda(n-1, parcial*n);
}

int fat(int n) {
    return fat_cauda(n, 1);
}
```

Recursão de cauda

- Conversão entre versões (**recursiva ↔ iterativa**)

```
int fat(int n) {
    int parcial = 1;

    while (n > 1) {
        parcial *= n;
        n = n-1;
    }

  return parcial;
}
```

```
int fat_cauda(int n, int parcial) {
    if (n == 1 || n == 0)
        return parcial;
    else
        return fat_cauda(n-1, parcial*n);
}

int fat(int n) {
    return fat_cauda(n, 1);
}
```

conversão

Recursão vs. iteração

Versão recursiva	Versão iterativa
Usa estruturas de seleção (*if*, *if-else* ou *switch*)	Usa estruturas de repetição (*for, while* ou *do-while*)
Repetição implícita através das chamadas à função	Repetição explícita
Termina ao atingir o caso base	Termina quando o teste do laço falha
Pode ocorrer infinitamente	Pode ocorrer infinitamente
Lento	Rápido
Implementação mais simples e de fácil manutenção	Implementação mais elaborada que pode dificultar a manutenção

Fonte: notas de aula do Prof. Rodrigo de Oliveira

Cuidados com as funções recursivas

- Garantir a **parada da recursão**:
  - O caso base deve estar implementado no código da função (**existência**)

Cuidados com as funções recursivas

- Garantir a **parada da recursão**:

  - O caso base deve estar implementado no código da função (**existência**)

  - A condição de parada deve ser atingível em algum momento (**corretude**)

Cuidados com as funções recursivas

- Garantir a **parada da recursão**:
  - O caso base deve estar implementado no código da função (**existência**)
  - A condição de parada deve ser atingível em algum momento (**corretude**)
- Evita a **recursão infinita**
  - Na prática, o programa **não** executará infinitamente
    - Ocorre um **estouro de memória** provocado pela limitação no tamanho da pilha

Cuidados com as funções recursivas

- **Parada da recursão**:

```
int fat(int n) {
   return (n*fat(n-1));
}
```

∄ critério de parada

Cuidados com as funções recursivas

- **Parada da recursão**:

```
int fat(int n) {
   return (n*fat(n-1));
}
```

∄ critério de parada

```
int fat(int n) {
    if (n == 0)
        return (1);
   else return (n*fat(n));
}
```

Critério de parada inatingível

Cuidados com as funções recursivas

- **Parada da recursão**:

```
int fat(int n) {
   return (n*fat(n-1));
}
```

∄ critério de parada

```
int fat(int n) {
   if (n == 0)
       return (1);
   else return (n*fat(n));
}
```

Critério de parada inatingível

```
int fat(int n)
{
    if (n == 0)
        return (1);
    else return (n*fat(n-1));
}
```

Implementação correta (critério de parada definido e atingível)

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

- **Exemplo:**

fat(4)

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

- **Exemplo:**

$$\boldsymbol{fat(4)} = \underbrace{4 * \boldsymbol{fat(3)}}$$

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

- **Exemplo:**

fat(4)

*4 * **fat(3)***

*3 * **fat(2)***

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

- **Exemplo:**

fat(4)

*4 * **fat(3)***

*3 * **fat(2)***

*2 * **fat(1)***

Cuidados com as funções recursivas

- Implementar o **passo recursivo**:
  - O tamanho do problema deve diminuir
  - A cada recursão o problema deve se aproximar do caso base

- **Exemplo:**

fat(4)

*4 * **fat(3)***

*3 * **fat(2)***

*2 * **fat(1)***

1 (**caso base**)

Cuidados com as funções recursivas

- A **ordem** das chamadas recursivas afeta o resultado

Cuidados com as funções recursivas

- A **ordem** das chamadas recursivas afeta o resultado

implementação 1

```
void recursiveFunction(int num) {
    if (num < 5) {
        printf("%d\n", num);
        recursiveFunction(num+1);
    }
}
```

1	recursiveFunction (0)				
2	printf (0)				
3		recursiveFunction (0+1)			
4		printf (1)			
5			recursiveFunction (1+1)		
6			printf (2)		
7				recursiveFunction (2+1)	
8				printf (3)	
9					recursiveFunction (3+1)
10					printf (4)

execução para *num* = 0

Cuidados com as funções recursivas

- A **ordem** das chamadas recursivas afeta o resultado

implementação 1

```
void recursiveFunction(int num) {
    if (num < 5) {
      printf("%d\n", num);
      recursiveFunction(num+1);
    }
}
```

1	recursiveFunction (0)
2	printf (0)
3	recursiveFunction (0+1)
4	printf (1)
5	recursiveFunction (1+1)
6	printf (2)
7	recursiveFunction (2+1)
8	printf (3)
9	recursiveFunction (3+1)
10	printf (4)

implementação 2

execução para *num* = 0

```
void recursiveFunction(int num) {
    if (num < 5) {
      recursiveFunction(num+1);
      printf("%d\n", num);
    }
}
```

1	recursiveFunction (0)
2	recursiveFunction (0+1)
3	recursiveFunction (1+1)
4	recursiveFunction (2+1)
5	recursiveFunction (3+1)
6	printf (4)
7	printf (3)
8	printf (2)
9	printf (1)
10	printf (0)

Cuidados com as funções recursivas

- Tentar limitar a **profundidade da recursão**:

Cuidados com as funções recursivas

- Tentar limitar a **profundidade da recursão**:

- A quantidade das chamadas recursivas deve ser **finita e pequena**
  - Evitar **problemas de memória**

Cuidados com as funções recursivas

- Tentar limitar a **profundidade da recursão**:

- A quantidade das chamadas recursivas deve ser **finita e pequena**
  - Evitar **problemas de memória**

- **Exemplo:**

  Qual a profundidade da recursão da função para calcular o fatorial de *n*?

Cuidados com as funções recursivas

- Tentar limitar a **profundidade da recursão**:

- A quantidade das chamadas recursivas deve ser **finita e pequena**
  - Evitar **problemas de memória**

- **Exemplo:**

  Qual a profundidade da recursão da função para calcular o fatorial de *n*?

  linear: *O(n)*

Cuidados com as funções recursivas

- Evitar uma **"explosão" exponencial das chamadas** recursivas:
  - Crescimento rápido do número de chamadas
    - **Limita o tamanho da entrada**
  - **Geram retrabalho** (repetição de cálculos)
    - Instâncias do problema são resolvidas mais de uma vez
  - **Fato:** alguns problemas não são eficientes quando implementados recursivamente
    - **Exemplo:** cálculo da série de *Fibonacci*

Série de *Fibonacci*

- **Definição informal:**
  - Sequência numérica iniciada com 0 e 1 e cujo os demais termos são obtidos pela soma dos dois anteriores

- Alguns termos da série de *Fibonacci*:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- **Definição recursiva:**

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \text{ } (\textbf{caso base}) \\ 1 & \text{se } n = 1 \text{ } (\textbf{caso base}) \\ f(n-1) + f(n-2) & \text{se } n > 1 \text{ } (\textbf{passo recursivo}) \end{cases}$$

Série de *Fibonacci*

- **Exemplos de aplicação:**
  - Inicialmente relacionada com a velocidade de reprodução dos coelhos

Série de *Fibonacci*

- **Exemplos de aplicação:**
  - Inicialmente relacionada com a velocidade de reprodução dos coelhos
  - É observada em muitos fenômenos biológicos

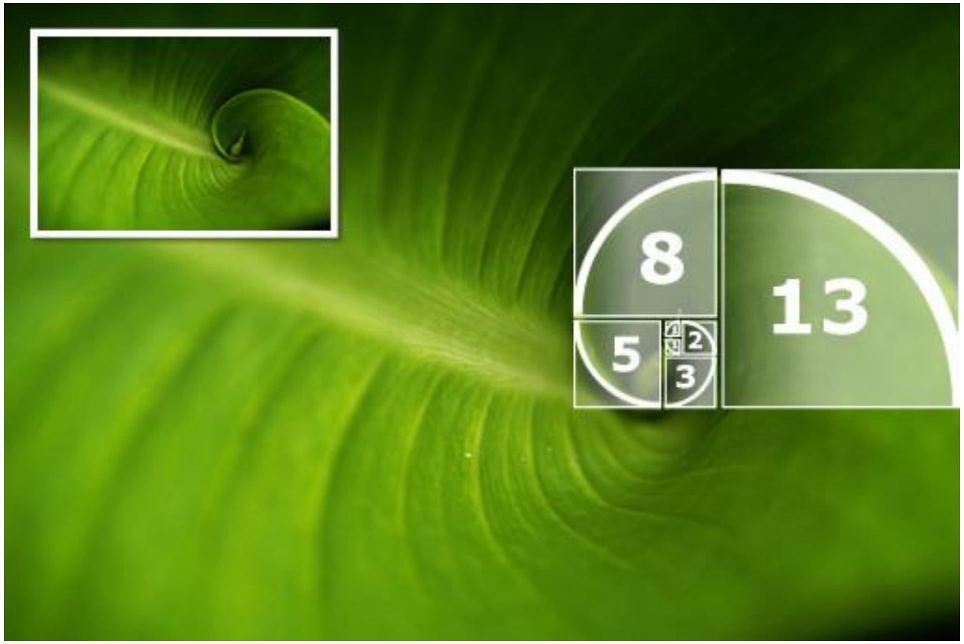


Figura: Evolução da espiral da folha da bromélia

Série de *Fibonacci*

- **Exemplos de aplicação:**
  - Inicialmente relacionada com a velocidade de reprodução dos coelhos
  - É observada em muitos fenômenos biológicos
  - Possui aplicações em computação, matemática, teoria dos jogos, artes e música

Conversão (aproximada) de milhas para km:

5 milhas ≈ 8 km

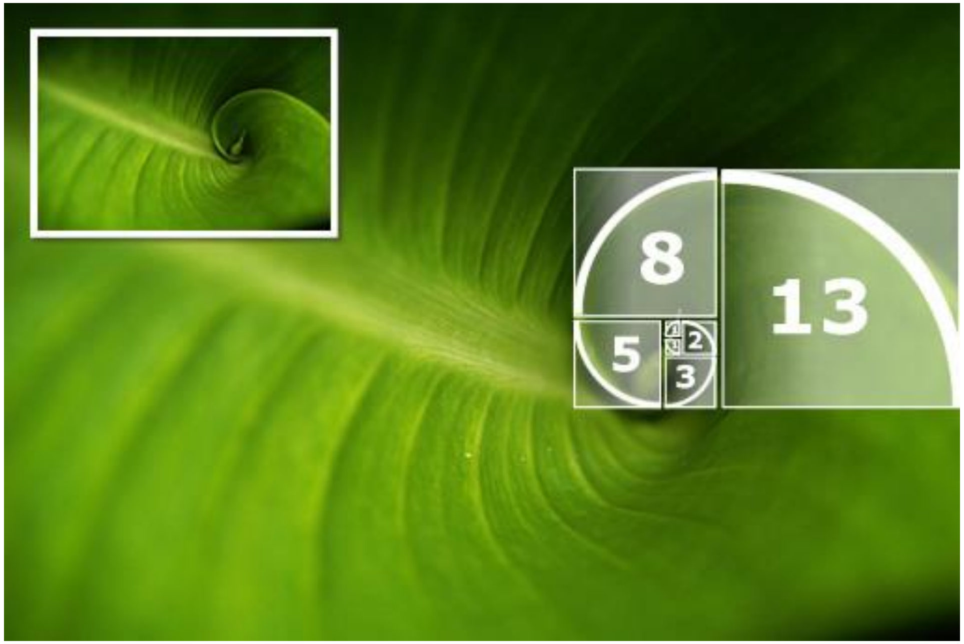


Figura: Evolução da espiral da folha da bromélia

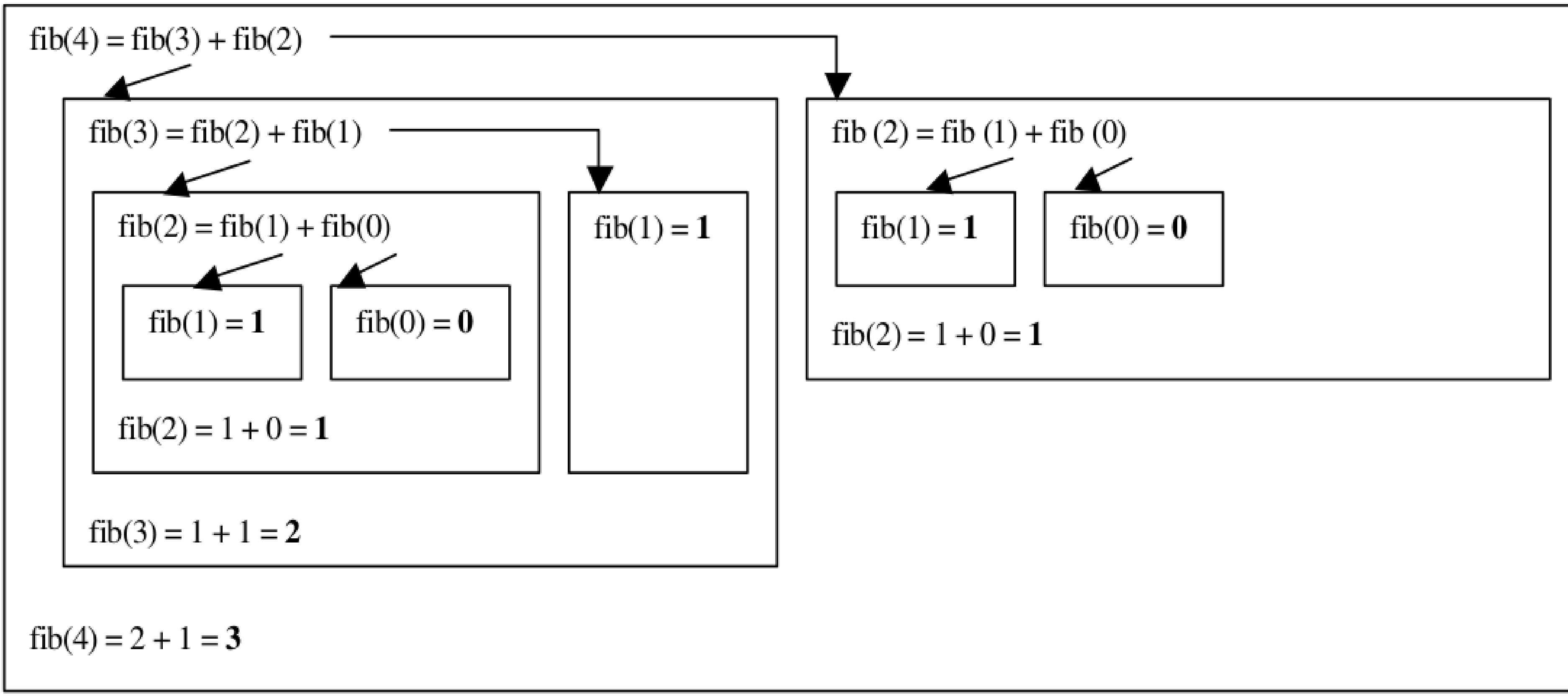
Série de *Fibonacci*

- **Implementação recursiva:**
  - Rápido para o cálculo de poucos termos
  - Processamento começa a demorar para valores de entrada maiores (***n > 40***)

```
int fib(int n)
{
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

Série de *Fibonacci*

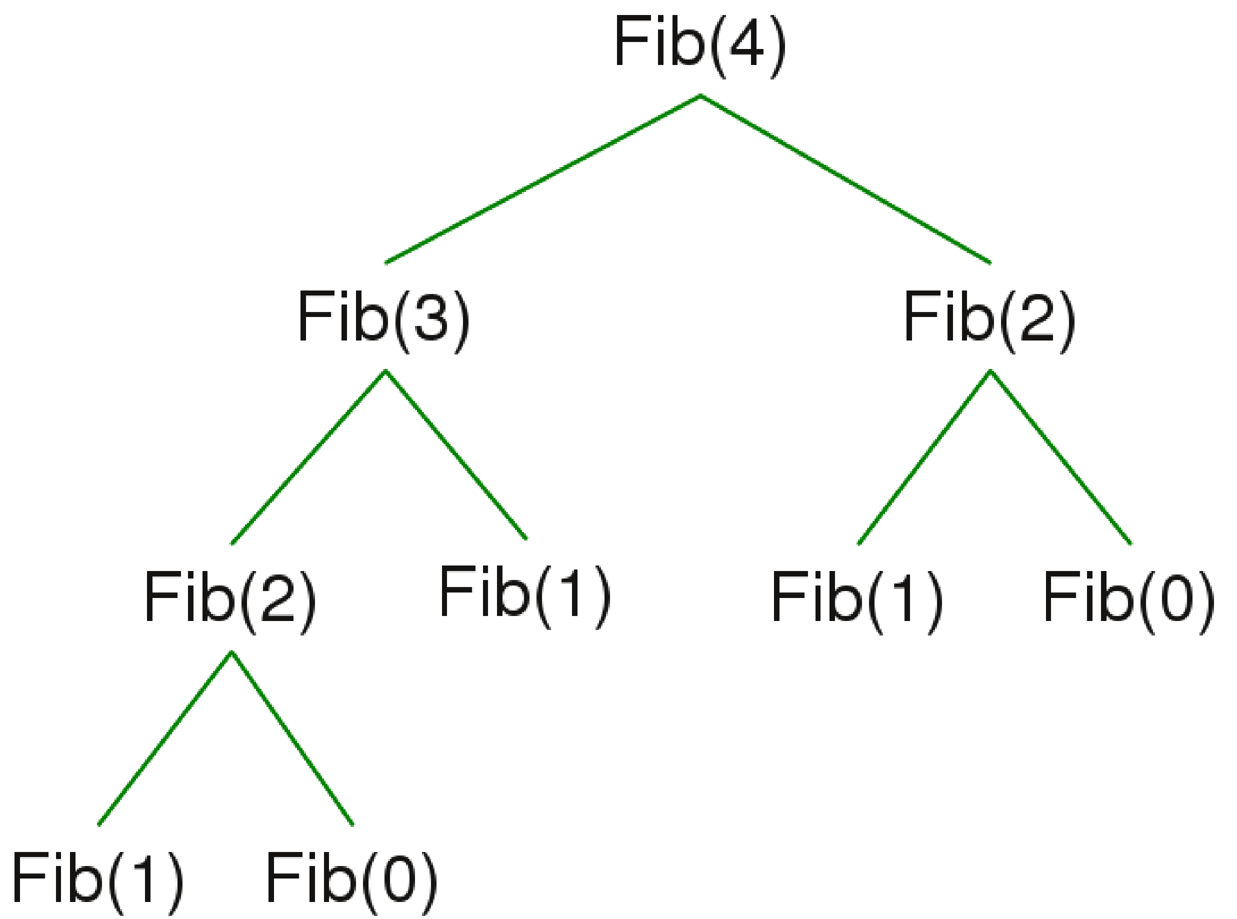
- Esquema das chamadas recursivas para n = 4 é:



Fonte: notas de aula do Prof. Rodrigo de Oliveira

Série de *Fibonacci*

- **Árvore de recursão:**



Análise de complexidade

- Analisar a complexidade de uma função recursiva **não** é uma tarefa trivial
  - Envolve a **resolução de uma recorrência**

Análise de complexidade

- Analisar a complexidade de uma função recursiva **não** é uma tarefa trivial
  - Envolve a **resolução de uma recorrência**

- **Recorrência** é uma expressão que define uma função em termos dos seus valores "anteriores"
  - **Ex:** $f(n) = f(n-1) + 5n - 8$

Análise da complexidade

- Analisar a complexidade de uma função recursiva **não** é uma tarefa trivial
  - Envolve a **resolução de uma recorrência**

- **Recorrência** é uma expressão que define uma função em termos dos seus valores "anteriores"
  - **Ex:** $f(n) = f(n-1) + 5n - 8$

- Resolver uma recorrência é:

> encontrar uma **fórmula fechada** que dê o valor diretamente em termos de seu parâmetro.

  - Geralmente resulta em uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais, etc.

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

  - **Caso base:**
    - **2 instruções:** comparação e retorno
    - Custo constante: ***O(1)***

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

  - **Caso base:**
    - **2 instruções:** comparação e retorno
    - Custo constante: ***O(1)***
  - **Passo recursivo:**
    - **5 instruções:** comparação, 2 chamadas recursivas, soma e retorno

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

  - **Caso base:**
    - **2 instruções:** comparação e retorno
    - Custo constante: ***O(1)***
  - **Passo recursivo:**
    - **5 instruções:** comparação, 2 chamadas recursivas, soma e retorno
    - Custo 1º passo (k = 1):

$T(n) = T(n-1) + T(n-2) + 5$

$\approx 2T(n-1) + 5,$ se considerármos $\boldsymbol{T(n-1) \approx T(n-2)}$

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

  - **Caso base:**
    - **2 instruções:** comparação e retorno
    - Custo constante: ***O(1)***
  - **Passo recursivo:**
    - **5 instruções:** comparação, 2 chamadas recursivas, soma e retorno
    - Custo 1º passo ($k = 1$):

      $T(n) = T(n-1) + T(n-2) + 5$

      $\approx 2T(n-1) + 5,$ se considerármos $\boldsymbol{T(n-1) \approx T(n-2)}$

    - Agregando o custo do 2º e 3º passos ($k = 2$ e $K = 3$), temos:

      $T(n) \approx 2(\boldsymbol{2T(n-2) + 5}) + 5 \approx 4T(n-2) + 15$ $\quad (\boldsymbol{K = 2})$

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

```
int fib(int n) {
    if (n < 2)          // caso base
      return n;
    else                // passo recursivo
      return (fib(n-1) + fib(n-2));
}
```

  - **Caso base:**
    - **2 instruções:** comparação e retorno
    - Custo constante: ***O(1)***
  - **Passo recursivo:**
    - **5 instruções:** comparação, 2 chamadas recursivas, soma e retorno
    - Custo 1º passo (k = 1):

      $T(n) = T(n-1) + T(n-2) + 5$

      $\approx 2T(n-1) + 5$, se considerármos $\boldsymbol{T(n-1) \approx T(n-2)}$
    - Agregando o custo do 2º e 3º passos (k = 2 e K = 3), temos:

      $T(n) \approx 2(\mathbf{2T(n-2) + 5}) + 5 \approx 4T(n-2) + 15$ ($\mathbf{K = 2}$)

      $T(n) \approx 4(\mathbf{2T(n-3) + 5}) + 15 \approx 8T(n-3) + 35$ ($\mathbf{K = 3}$)

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):
  - **Passo recursivo:**
    - Analisando a evolução obtemos a **generalização da recorrência**:

$T(n) \approx 2T(n-1) + c \qquad (\boldsymbol{K = 1})$

$T(n) \approx 4T(n-2) + 3c \qquad (\boldsymbol{K = 2})$

$T(n) \approx 8T(n-3) + 7c \qquad (\boldsymbol{K = 3})$

...

$\boldsymbol{T(n) \approx 2^K T(n-K) + (2^K-1)c}$

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):

  - **Passo recursivo:**
    - Analisando a evolução obtemos a **generalização da recorrência**:

$$T(n) \approx 2T(n-1) + c \qquad \qquad \boldsymbol{(K = 1)}$$
$$T(n) \approx 4T(n-2) + 3c \qquad \qquad \boldsymbol{(K = 2)}$$
$$T(n) \approx 8T(n-3) + 7c \qquad \qquad \boldsymbol{(K = 3)}$$
$$\ldots$$
$$\boldsymbol{T(n) \approx 2^K T(n-K) + (2^K-1)c}$$

    - A complexidade é obtida quando ***T(n)*** é dado em função de ***T(0)*** (**último nível da recursão**):

$$n-K = 0 \;\equiv\; \boldsymbol{K = n}$$

$$\boldsymbol{T(n) \approx 2^n T(0) + (2^n-1)c \approx O(2^n)}$$

Análise da complexidade

- Série de *Fibonacci* (**versão recursiva**):
  - **Passo recursivo:**
    - Analisando a evolução obtemos a **generalização da recorrência**:

$T(n) \approx 2T(n-1) + c \quad (\mathbf{K = 1})$

$T(n) \approx 4T(n-2) + 3c (\mathbf{K = 2})$

$T(n) \approx 8T(n-3) + 7c (\mathbf{K = 3})$

...

$\boldsymbol{T(n) \approx 2^K T(n-K) + (2^K-1)c}$

    - A complexidade é obtida quando ***T(n)*** é dado em função de ***T(0)*** (**nível + profundo da recursão**):

$n\text{-}K = 0 \equiv \mathbf{K = n}$

$\boldsymbol{T(n) \approx 2^n T(0) + (2^n-1)c \approx O(2^n)}$

exponencial:
$O(1) + O(2^n) = O(2^n)$

Série de *Fibonacci*

- **Algoritmo iterativo:**

```
int Fib_iter(int n) {
    int k, i = 1, F = 0;
    for (k = 1; k <= n; k++) {
        F = F + i;
        i = F - i;
    }
    return F;
}
```

Análise da complexidade

- Série de *Fibonacci* (**versão iterativa**):

```
int Fib_iter(int n) {
    int k, i = 1, F = 0;                    constante: O(1)
    for (k = 1; k <= n; k++) {
        F = F + i;
        i = F - i;
    }
    return F;                               constante: O(1)
}
```

Análise da complexidade

- Série de *Fibonacci* (**versão iterativa**):

```
int Fib_iter(int n) {
    int k, i = 1, F = 0;
    for (k = 1; k <= n; k++) {
        F = F + i;
        i = F - i;
    }
    return F;
}
```

constante: ***O(1)***

linear: ***O(n)***

constante: ***O(1)***

Análise da complexidade

- Série de *Fibonacci* (**versão iterativa**):

```
int Fib_iter(int n) {
    int k, i = 1, F = 0;
    for (k = 1; k <= n; k++) {
        F = F + i;
        i = F - i;
    }
    return F;
}
```

constante: ***O(1)***

+

linear: ***O(n)***

+

constante: ***O(1)***

Complexidade: *O(n)*

Série de *Fibonacci*

- **Comparação entre as versões recursiva e iterativa:**

n	**10**	**20**	**30**	**40**	**50**
Recursiva	8 ms	1 s	2 min	21 dias	10^9 anos
Iterativa	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Série de *Fibonacci*

- **Comparação entre as versões recursiva e iterativa:**

n	10	20	30	40	50
Recursiva	8 ms	1 s	2 min	21 dias	10^9 anos
Iterativa	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

- **Conclusão:** versão iterativa costuma ser mais eficiente que a versão recursiva
  - Exceto quando a implementação iterativa não é trivial
  - Exige alguma estrutura linear (pilha ou fila) para armazenamento
    - **Ex:** Percorrimento em profundidade, operações em árvore

Exercícios

1) Faça uma função recursiva que, dado um intervalo [*a*,*b*] (sendo $a \leq b$), calcule o valor do somatório:

$$\sum_{i=a}^{b} i^{2}$$

Exercícios

1) Faça uma função recursiva que, dado um intervalo [*a*,*b*] (sendo $a \leq b$), calcule o valor do somatório:

$$\sum_{i=a}^{b} i^2$$

$$\begin{cases} f(a,b) = a*a & \text{se } a = b \\ f(a,b) = a*a + f(a+1,b) & \text{se } a < b \end{cases}$$

```
int somaQ(int a, int b) {
    if (a == b)
        return (a*a);
    else
        return (a*a + somaQ(a+1,b));
}
```

Exercícios

2) Escreva uma função recursiva que recebe como parâmetros um número real ***x*** e um inteiro ***N*** e retorne o valor de x^N. (**obs: *x*** e ***N*** podem ser negativos).

Exercícios

2) Escreva uma função recursiva que recebe como parâmetros um número real ***x*** e um inteiro ***N*** e retorne o valor de x^N. (**obs: *x*** e ***N*** podem ser negativos).

$$
\begin{cases}
f(x, 0) = 1 & \text{se } N = 0 \\
f(x,N) = x * f(x,N-1) & \text{se } N > 0 \\
f(x,N) = (1/x) * f(x, N+1) & \text{se } N < 0
\end{cases}
$$

```
float exp(float x, int N) {
     if (N==0)
          return 1;
    if (N >0)
         return (x * exp( x ,N-1));
   else
        return(1/x * exp( x, N+1));
}
```

Exercícios

3) Faça um algoritmo recursivo que realize uma contagem regressiva a partir de um valor *n*.

Ex: a *contagem regressiva de 5 mostra na tela:*

5 ... 4 ... 3 ... 2 ... 1 ... Fogo!

Exercícios

3) Faça um algoritmo recursivo que realize uma contagem regressiva a partir de um valor *n*.

Ex: a *contagem regressiva de 5 mostra na tela:*

5 ... 4 ... 3 ... 2 ... 1 ... Fogo!

$$\begin{cases} f(0) = \text{printf(“Fogo!);} \\ f(n) = \{\text{printf(“\%d...”,}n\text{);}\ f(n\text{-}1)\text{;}\} \end{cases}$$

```
void cont_regressiva(int n) {
      if (n < 0)
         exit();
      if (n == 0)
         printf(" Fogo!");
     else {
        printf("%d…", n);
        cont_regressiva(n-1);
     }
}
```

Exercícios

4) Um problema típico em ciência da computação consiste em converter um número decimal para sua forma binária. A forma mais simples de fazer isso é dividir o número sucessivamente por 2. O resto da *i-ésima* divisão é o dígito *i* do binário (da direita para a esquerda).

Ex: O número **12** corresponde ao binário **1100**.

12 / 2 = 6, resto **0** (1º dígito da direita para esquerda)

6 / 2 = 3, resto **0** (2º dígito da direita para esquerda)

3 / 2 = 1, resto **1** (3º dígito da direita para esquerda)

1 / 2 = 0, resto **1** (4º dígito da direita para esquerda)

Escreva uma função recursiva ***void Dec2Bin(int n)*** que, dado um número decimal, escreva na tela sua representação binária corretamente.

Exercícios

$$f(n) = \begin{cases} \text{printf("\%d", n);} & \text{se } n \le 1 \text{ (caso base)} \\ \text{f(n)} = f(n/2)\text{; printf("\%d", n\%2);} & \text{se } n > 1 \text{ (passo recursivo)} \end{cases}$$

```
void Dec2Bin(int n)
{
    if (n <=1 )
        printf("%d", n);
    else {
        Dec2Bin(n/2);
        printf("%d", n%2);
    }
}
```

Exercícios para casa

1. Implemente as versões recursiva e iterativa da função para obter a série de *Fibonacci*. Utilize a biblioteca *time.h* para medir e observar o tempo necessário para calcular *n* = 5, 10, 20 e 30.

2. Implemente uma função recursiva para encontrar o maior elemento de um arranjo. Para isto, encontre o maior valor do arranjo sem o último elemento e depois compare-o com o último.

3. Implemente uma função que exiba todas as *substrings* de uma cadeia com *n* caracteres. Para isto, enumere todas as *substrings* que começam com o 1º caractere (serão *n substrings*). Depois, repita o processo para a string após remover o 1º caractere.

   Ex: para a string UFA temos: U, UF, UFA, F, FA, A

Exercícios para casa

4. Dada a definição da função abaixo, avalie *f(1,10)* através de sua árvore de recursão.

```
double f(double x, double y) {
    if (x >= y)
        return (x+y)/2;
    else
        return (f(x+2,y-1) + f(x+1,y-2))/2;
}
```

5. Determine a complexidade assintótica (pior caso) dessa função.

Bibliografia

- Slides adaptados do material do Prof. Dr. Bruno Travençolo, do Prof. Autran Macêdo, da Profa. Dra. Denise Guliato e do Prof. Dr. Moacir Ponti Jr. (ICMC-USP)

- CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002

- ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2ª ed.), Thomson, 2004

- MORAES, C.R. Estruturas de Dados e Algoritmos: uma abordagem didática (2ª ed.), Futura, 2003

Bibliografia

- FEOFILOFF, P. **Recursão e algoritmos recursivos.** Disponível em: http://www.ime.usp.br/~pf/algoritmos/aula/recu.html

- CALDAS, R. B**. Introdução a Computação**. Disponível em: http://www.dcc.ufam.edu.br/~ruiter/icc/

- de OLIVEIRA, R. **Notas de Aula de Algoritmos e Programação de Computadores**.