ACH2002

Aula 6

Técnicas de Desenvolvimento de Algoritmos - Recursividade

(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

EACH USP

Aulas passadas

- Prova de corretude de algoritmos iterativos:
  – Prova por indução de invariantes
- Análise de complexidade (ex: tempo)
  – Testes empíricos
  – Notação assintótica (O, o, Ω, ω, Θ)

Aula de hoje e próximas

- Técnicas de programação
- Hoje: recursividade – tem a ver com indução (fraca)!!!

Recursividade

- O que é recursividade?

Recursividade

- O que é recursividade?
  - *sf* (*recursivo+i+dade*) *Ling* Propriedade sintática pela qual um elemento pode aparecer um número infinito de vezes numa derivação, introduzido sempre pela mesma regra.
  - Ex: Significado da sigla GNU:

“GNU is Not Unix”

EACH USP

Recursividade é uma técnica de programação baseada em indução (fraca)

- Objetivo da indução (fraca): provar que uma determinada propriedade (P) é válida para todos os elementos de um conjunto potencialmente infinito

- 3 partes:

- Base da indução: prova/mostra que P é verdadeira para o primeiro elemento (0, ou 1, ou 2, …) desse conjunto

- Hipótese da indução: assume que P é verdadeira para o n-ésimo elemento desse conjunto (é na verdade o enunciado de P)

- Passo da indução: usa a hipótese para provar que P é verdadeira para o (n+1)-ésimo elemento desse conjunto

EACH USP

Prova por indução

Ex: prova por indução que $$1+2+3+\cdots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

Base: P é verdadeira para n = 1: $$1=\frac{1(1+1)}{2}$$

Hipótese: P é verdadeira para um dado $$1+2+3+\cdots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

Passo: Dado que P vale para n, P é verdadeira para n+1:

$$\begin{aligned} 1+2+3+\cdots+n+n+1 &= \frac{n(n+1)}{2}+n+1 \\ &= \frac{n^2+n+2n+2}{2} \\ &= \frac{n^2+3n+2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

Note que um loop é uma série...

Um exemplo visual...

Bonecas russas

https://pt.wikipedia.org/wiki/Matriosca

Um exemplo visual...

Bonecas russas

https://www.bonecarussa.com.br/novidade/matrioshkas-matrioskas-no-brasil

Recursividade

- Em computação:
  - ação na qual um procedimento (método, função, ...) chama a si mesmo.
  - geralmente seu uso permite descrição mais clara e concisa dos algoritmos, principalmente quando o problema considerado tem natureza recursiva.
  - Exemplo:
    - calcular fatorial de um número natural

$0! = 1$

$1! = 1$

$2! = 2$

$n! = 2 * 3 * 4 * ... * (n-1) * n, n > 2$

Para $n > 1$:

$n! = (n-1)! * n$

Recursividade

- Exemplo:
  - O cálculo do fatorial pode ser implementado de duas formas:
    - iterativo
    - recursivo

$1! = 1$

Para $n > 1$:

$n! = (n-1)! * n$

Recursividade

- Exemplo:
  - Algoritmo iterativo:

$1! = 1$

Para $n > 1$:

$n! = (n-1)! * n = 1 * 2 * 3 * \ldots * n$

Recursividade

- Exemplo:
  - Algoritmo iterativo:

```
fatorial (n)
 fat = 1
  para i = 2 até n
            fat = fat * i
 fim para
 retorna fat
```

1! = 1
Para n > 1:
n! = (n-1)! * n = 1 * 2 * 3 * … * n

Recursividade

- Exemplo:
  - Algoritmo iterativo:

Complexidade?

```
fatorial (n)
  fat = 1
  para i = 2 até n
            fat = fat * i
  fim para
  retorna fat
```

O(1)

n * O(1) = O(n)

O(1)

1! = 1

Para n > 1:

n! = (n-1)! * n = 1 * 2 * 3 * … * n

Recursividade

- Exemplo:
  - Algoritmo iterativo:

Complexidade: **O(n)**

```
fatorial (n)
  fat = 1
   para i = 2 até n
            fat = fat * i
  fim para
  retorna fat
```

O(1)

n * O(1) = O(n)

O(1)

1! = 1
Para n > 1:
n! = (n-1)! * n = 1 * 2 * 3 * … * n

Recursividade

- Exemplo:
  - Algoritmo recursivo:

fatorial (n)

Indução fraca:
Base da indução
Hipótese da indução
Passo da indução

1! = 1
Para n > 1:
n! = (n-1)! * n

Recursividade

- Exemplo:
  - Algoritmo recursivo:

```
fatorial (n)
  se n < 2
        retorna 1
```

Indução fraca:
Base da indução
Hipótese da indução
Passo da indução

Base da recursão !

1! = 1
Para n > 1:
n! = (n-1)! * n

Recursividade

- Exemplo:
  - Algoritmo recursivo:

Indução fraca:
Base da indução
Hipótese da indução
Passo da indução

```
fatorial (n)
  se n < 2
        retorna 1
  senão
        retorna n * fatorial (n-1)
  fim se
```

Base da recursão !

Hipótese da indução
Passo da indução

$1! = 1$
Para $n > 1$:
$n! = (n-1)! * n$

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4
retorna n * fatorial (3)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4
retorna n * fatorial (3)
```

```
n = 3
retorna n * fatorial (2)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4
retorna n * fatorial (3)
```

```
n = 3
retorna n * fatorial (2)
```

```
n = 2
retorna n * fatorial (1)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
  se n < 2
       retorna 1
  senão
       retorna n*fatorial (n-1)
  fim se
```

Base da recursão !

```
n = 5
retorna n * fatorial (4)

    n = 4
    retorna n * fatorial (3)

        n = 3
        retorna n * fatorial (2)

            n = 2
            retorna n * fatorial (1)

                n = 1
                retorna 1
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)

    n = 4
    retorna n * fatorial (3)

        n = 3
        retorna n * fatorial (2)

            n = 2
            retorna n * fatorial (1)
```

1

Recursividade

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
n = 5
retorna n * fatorial (4)

    n = 4
    retorna n * fatorial (3)

        n = 3
        retorna n * fatorial (2)

            n = 2            1
            retorna 2 * fatorial (1)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
        retorna 1
 senão
        retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4
retorna n * fatorial (3)
```

```
n = 3
retorna n * fatorial (2)
```

2

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4
retorna n * fatorial (3)
```

n = 3

retorna **3** * ~~fatorial (2)~~ **2**

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4                6
retorna n * fatorial (3)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna n * fatorial (4)
```

```
n = 4                6
retorna 4 * fatorial (3)
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

24

```
n = 5
retorna n * ~~fatorial (4)~~
```

Recursividade

- Exemplo:
  - Algoritmo recursivo:
  - Simulação para n = 5

```
fatorial (n)
 se n < 2
      retorna 1
 senão
      retorna n*fatorial (n-1)
 fim se
```

```
n = 5
retorna 5 * fatorial (4)
```

24

RESULTADO = 120

Recursividade

- Implementação na linguagem C:

```
int fatorial (int n)
 {
   if (n < 2)
     return 1;
   else
     return n * fatorial (n - 1);
 }
```

Recursividade

- Testando... na linguagem C:

```
#include <stdio.h>

int fatorial (int n)
 {
   if (n < 2)
     return 1;
   else
     return n * fatorial (n - 1);
 }

void main()
{
  int n;
  do {
    printf ("Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:");
    scanf ("%i", &n);
    if (n >= 0)
      printf ("Fatorial de n = %d é %d\n", n, fatorial (n));
  } while (n >= 0);
}
```

EACH USP

Recursividade

- Testando... na linguagem C:

```
(base) ariane@rainbow:~/ACH2002-2022-2/codigos/tecnicas$ gcc -o recursao.exe recursao.c
(base) ariane@rainbow:~/ACH2002-2022-2/codigos/tecnicas$ ./recursao.exe
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:0
Fatorial de n = 0 é 1
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:1
Fatorial de n = 1 é 1
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:2
Fatorial de n = 2 é 2
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:3
Fatorial de n = 3 é 6
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:4
Fatorial de n = 4 é 24
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:5
Fatorial de n = 5 é 120
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:6
Fatorial de n = 6 é 720
Você quer saber o fatorial de quanto? Se não quiser calcular mais digite um número negativo:-1
(base) ariane@rainbow:~/ACH2002-2022-2/codigos/tecnicas$
```

Recursividade

- Como implementar recursividade:
  - Compilador usa uma ***pilha***
    - estrutura de dados que armazena dados usados em cada chamada de um procedimento que ainda não terminou de processar;
    - o último dado a entrar é o primeiro a sair (LAST IN FIRST OUT);
    - o espaço de variáveis e parâmetros alocado para um método implementado em uma determinado linguagem é chamado de ***registro de ativação***;
    - o ***registro de ativação*** é desalocado quando termina um método.
    - mais detalhes de pilha serão vistos na disciplina AED
  - Deve ser considerado o problema de **terminação**
    - no algoritmo deve existir uma condição que encerre o processo de empilhamento e comece desempilhar os dados armazenados (**base da recursão**)

Indução Matemática

- Algoritmos recursivos podem ser definidos e estudados a partir da indução matemática.
- Seja *T* um teorema a ser provado
  - Consideremos ***T*** como tendo um número natural como parâmetro (***n***)
  - Em vez de tentar provar que *T* é válido para todos valores de ***n***, basta provar duas condições:

1. *T* é válido para *n = 1; //ou outro valor base*
2. Para todo *n* > 1:
   - se *T* é válido para *n* ⇒ *T* é válido para *n+1*

EACH USP

Indução Matemática

- Exemplo: fatorial
- Para facilitar a indução, modificaremos um pouco o método usado anteriormente para calcular o fatorial (base = 0) facilitando a definição do passo base:

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

Recursividade e indução

- Definindo o cálculo do fatorial usando indução:

  - Qual é o passo base?

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

Recursividade e indução

- Definindo o cálculo do fatorial usando indução:

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

  - Qual é o passo base?

    Se n = 0, então o fatorial = 1

Recursividade e indução

- Definindo o cálculo do fatorial usando indução:

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

  - Qual é o passo base?

    Se n = 0, então o fatorial = 1

  - Qual é o passo indutivo?

    n * fatorial (n - 1)

Recursividade e indução

- Provando que o algoritmo (recursivo) está **correto**:

  (com base em seus retornos)

  - Base da indução:

    Se n = 0, a função retorna 1 (o valor correto)

  - Passo da indução

    - Assumindo que a função retorne o valor correto ((n-1)!) para (n-1) > 0
    - fatorial(n) irá retornar n * fatorial(n - 1) = n * (n - 1)! = (n)!
    - Logo, fatorial(n) retorna o valor **correto**!

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

Recorrências

- Recorrências são **equações ou inequações** que descrevem uma **função** em termos de seus valores com entradas menores
- Ex: F: função fatorial

  Equação de recorrência:

$$F(n) = \begin{cases} 1 & , \text{ se } n = 0 \\ n * F(n-1) & , \text{ se } n > 0 \end{cases}$$

Equações de recorrências

- A complexidade de algoritmos **recursivos** é normalmente definida por **equações de recorrência**
- Seja *T(n)* a complexidade de tempo do algoritmo para uma entrada de "*tamanho*" n
- Como seria a complexidade de nosso algoritmo recursivo **fatorial**?

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

Equações de recorrências

- A complexidade de algoritmos **recursivos** é normalmente definida por **equações de recorrência**
- Seja *T(n)* a complexidade de tempo do algoritmo para uma entrada de "*tamanho*" n
- Como seria a complexidade de nosso algoritmo recursivo **fatorial**?

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

$$T(n) = \begin{cases} O(1) & \text{, se } n = 0 \\ & \text{, se } n > 0 \end{cases}$$

Equações de recorrências

- A complexidade de algoritmos **recursivos** é normalmente definida por **equações de recorrência**
- Seja *T(n)* a complexidade de tempo do algoritmo para uma entrada de "*tamanho*" n
- Como seria a complexidade de nosso algoritmo recursivo **fatorial**?

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

$$T(n) = \begin{cases} O(1) & \text{, se } n = 0 \\ O(1) + T(n-1) & \text{, se } n > 0 \end{cases}$$

$$T(n) = \underbrace{O(1) + O(1) + \ldots + O(1)}_{n+1 \text{ vezes}} = ?$$

Equações de recorrências

- A complexidade de algoritmos **recursivos** é normalmente definida por **equações de recorrência**
- Seja *T(n)* a complexidade de tempo do algoritmo para uma entrada de "*tamanho*" n
- Como seria a complexidade de nosso algoritmo recursivo **fatorial**?

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

$$T(n) = \begin{cases} O(1) & \text{, se } n = 0 \\ O(1) + T(n-1) & \text{, se } n > 0 \end{cases}$$

$$T(n) = \underbrace{O(1) + O(1) + \ldots + O(1)}_{n+1 \text{ vezes}} = O(n)$$

(igual à versão iterativa)

Equações de recorrências

- Prove por indução que o algoritmo fatorial tem complexidade descrita pela equação de recorrência abaixo:
  - Base: se n = 0, há só a comparação e o return 1 → O(1)
  - Hipótese da indução: para n > 0, fatorial leva tempo T(n) = O(n)
  - Passo da indução: Para n+1, multiplica (n+1) pelo retorno de fatorial(n), levando então tempo = O(1) + T(n) = O(1) + O(n) = O(n)

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

$$T(n) = \begin{cases} O(1) & \text{, se } n = 0 \\ O(1) + T(n-1) & \text{, se } n > 0 \end{cases}$$

$$T(n) = \underbrace{O(1) + O(1) + \ldots + O(1)}_{\text{n+1 vezes}} = O(n)$$

Busca Sequencial

- Solução iterativa:

Recebe um número x e um vetor $v[0\,..\,n{-}1]$ com $n \geq 0$
e devolve k no intervalo $0\,..\,n{-}1$ tal que $v[k] = x$.
Se tal k não existe, devolve -1.

```
int Busca (int x, int v[], int n) {
    int k;
    k = n - 1;
    while (k >= 0 && v[k] != x)
        k -= 1;
    return k;
}
```

Busca Sequencial

- Solução **recursiva**:

Recebe x, v e $n \geq 0$ e devolve k tal que $0 \leq k < n$ e $v[k] = x$.
Se tal k não existe, devolve -1.

```
int BuscaR (int x, int v[], int n) {
```

Busca Sequencial

- Solução **recursiva**:

Recebe x, v e $n \geq 0$ e devolve k tal que $0 \leq k < n$ e $v[k] = x$.
Se tal k não existe, devolve -1.

```
int BuscaR (int x, int v[], int n) {
    if (n == 0) return -1;
    if (x == v[n-1]) return n - 1;
    return BuscaR (x, v, n - 1);
}
```

Sequência de Fibonacci

- Números de Fibonacci, definidos pela seguinte equação de recorrência:

$$\left\{ \begin{array}{c} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{array} \right\}$$

Sequência de Fibonacci

- Implemente um procedimento recursivo para calcular a sequência de Fibonacci, dado ***n***

$$\left\{ \begin{array}{c} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{array} \right\}$$

Sequência de Fibonacci

- Implemente um procedimento recursivo para calcular a sequência de Fibonacci, dado ***n***

$$\left\{ \begin{array}{c} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{array} \right\}$$

```
fibonacci(n)
 se n < 2
     retorna n
 senão
     retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```

Sequência de Fibonacci

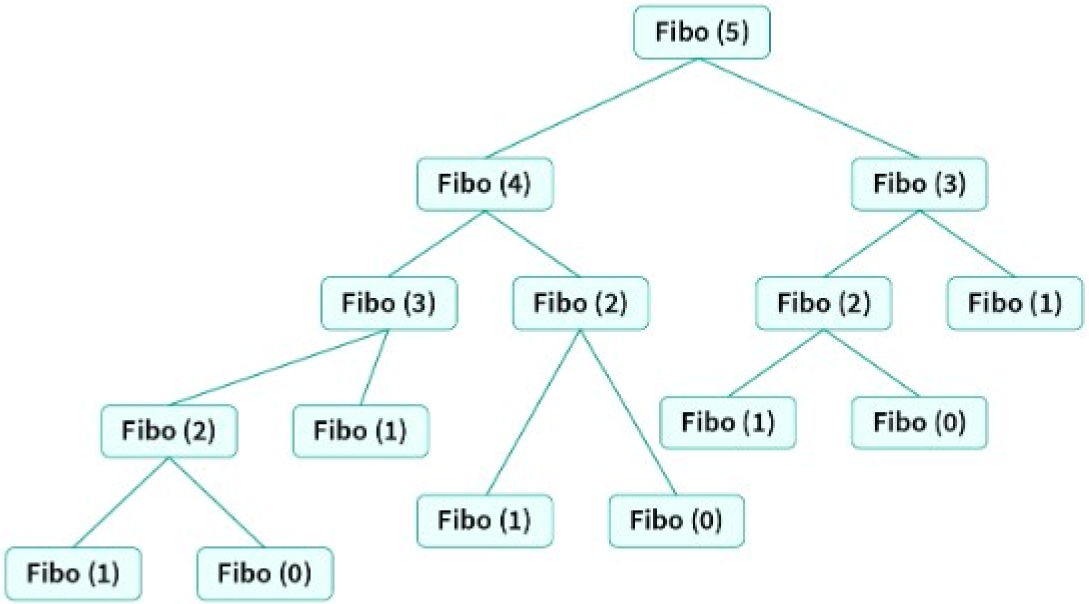
- Complexidade?

```
fibonacci(n)
 se n < 2
     retorna n
 senão
     retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```

Sequência de Fibonacci

- Complexidade?

```
fibonacci(n)
 se n < 2
      retorna n
 senão
      retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```

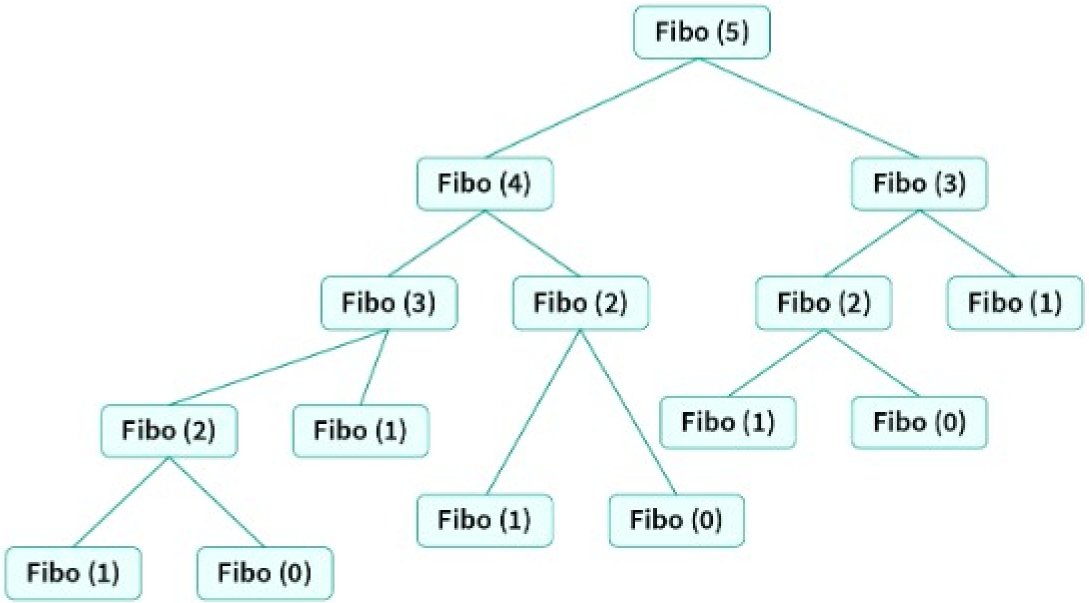


O que você chutaria?
(tempo e espaço)

Sequência de Fibonacci

- Complexidade?

```
fibonacci(n)
 se n < 2
      retorna n
 senão
      retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```



O que você chutaria?

Espaço: Linear (O(n)) – na memória só tem um ramo dessa árvore
Tempo: Exponencial!

Veremos técnicas de como calcular na aula que vem

Sequência de Fibonacci

- Complexidade?

```
fibonacci(n)
 se n < 2
     retorna n
 senão
     retorna fibonacci(n-1) + fibonacci(n-2)
 fim se
```

- Para cada iteração, chama o procedimento recursivamente 2 vezes → **aumenta absurdamente a complexidade do algoritmo**

Sequência de Fibonacci

- Como seria a versão iterativa? (C)

$$\begin{Bmatrix} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{Bmatrix}$$

Sequência de Fibonacci

- Como seria a versão iterativa? (C)

$$\left\{ \begin{array}{c} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{array} \right\}$$

```c
int fibonacci (int n)
{
  int fib, fib_1, fib_2; /* fib_1 significa 'fib-1' */
  if (n < 2)
    return n;
  else
  {
    fib_2 = 0;
    fib_1 = 1;
    for (int i = 2; i <= n; i++)
    {
      /* i-esimo elemento da série sendo calculado */
      /* INVARIANTE: */
      /* fib_1 tem o elemento anterior, e fib_2 tem o anterior do anterior */
      fib = fib_1 + fib_2;
      fib_2 = fib_1;
      fib_1 = fib;
    }
    return fib;
  }
}
```

Sequência de Fibonacci

- Como seria a versão iterativa? (C)

$$\begin{cases} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{cases}$$

```c
int fibonacci (int n)
{
  int fib, fib_1, fib_2; /* fib_1 significa 'fib-1' */
  if (n < 2)
    return n;
  else
  {
    fib_2 = 0;
    fib_1 = 1;
    for (int i = 2; i <= n; i++)
    {
      /* i-esimo elemento da série sendo calculado */
      /* INVARIANTE: */
      /* fib_1 tem o elemento anterior, e fib_2 tem o anterior do anterior */
      fib = fib_1 + fib_2;
      fib_2 = fib_1;
      fib_1 = fib;
    }
    return fib;
  }
}
```

Complexidade:

Sequência de Fibonacci

- Como seria a versão iterativa? (C)

$$\begin{cases} f_0 = 0, f_1 = 1, \\ f_n = f_{n-1} + f_{n-2}, n \geq 2 \end{cases}$$

```
int fibonacci (int n)
{
  int fib, fib_1, fib_2; /* fib_1 significa 'fib-1' */
  if (n < 2)
    return n;
  else
  {
    fib_2 = 0;
    fib_1 = 1;
    for (int i = 2; i <= n; i++)
    {
      /* i-esimo elemento da série sendo calculado */
      /* INVARIANTE: */
      /* fib_1 tem o elemento anterior, e fib_2 tem o anterior do anterior */
      fib = fib_1 + fib_2;
      fib_2 = fib_1;
      fib_1 = fib;
    }
    return fib;
  }
}
```

Complexidade:
Espaço: O(1)
Tempo: O(n)

(diferente da versão iterativa!)

Recursividade versus Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas, gerando programas menores e mais simples.
- Soluções iterativas em geral usam espaço definido de memória, enquanto soluções recursivas solicitam memória à medida que precisam.
- Cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas.
- Programas recursivos que possuem chamadas recursivas no final do código são ditos terem recursividade de cauda. São facilmente transformáveis em uma versão não recursiva (exemplos: fatorial, números de Fibonacci)
- Projetista de algoritmos deve levar consideração a complexidade (temporal e espacial), bem como os outros custos (e.g., facilidade de manutenção) para decidir por qual solução utilizar.

Recursividade

- Quando usar e não usar recursividade:
  - Algoritmos recursivos são adequados quando o problema ou os dados a serem tratados são definidos em termos recursivos
  - Mas isso não garante que a solução recursiva seja a melhor solução.
  - Problemas para os quais devem ser evitados algoritmos recursivos podem ser caracterizados por: (B = condição, S = comandos, P = função)

    P≡ se B então (S, P)

  - Esses programas são facilmente transformados em não recursivos, fazendo-se:

    P≡ (enquanto B faça S).

Ex: busca sequencial em vetor

EACH USP

Exercícios

1. Forneça uma solução recursiva para o problema de busca binária.
2. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor de inteiros *v[0..n-1]*. O problema faz sentido quando *n* é igual a 0? Quanto deve valer a soma nesse caso?
3. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor *v[ini..fim-1]*. O problema faz sentido quando *ini* é igual a *fim*? Quanto deve valer a soma nesse caso?
4. Escreva uma função recursiva *max* que calcule o valor de um elemento máximo de um vetor *v[0..n-1]*. Quantas comparações envolvendo os elementos do vetor a sua função faz?
5. Escreva uma função recursiva que calcule a soma dos dígitos de um inteiro positivo *n*. A soma dos dígitos de 132, por exemplo, é 6.
6. Escreva uma função recursiva *potencia*(*base*, *expoente*) que calcula base elevado a expoente utilizando multiplicações. Considere base e expoente >= 0. Considere $0^0 = 1$.
7. Escreva uma função recursiva que resolva a seguinte equação de recorrência:

   R(x) = 2 * R(x - 1) – 4, para x > 0

   R(0) = 2

EACH USP

Referências

- C. Camarão & L. Figueiredo. Programação de Computadores em Java. Livros Técnicos e Científicos Editora, 2003.
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004. Cap 2.2
- Notas de aula – Prof. Delano Beder – EACH-USP
- Paulo Feofiloff. Algoritmos em C. Cap 2 e 3 (tem exercícios!!!) https://www.ime.usp.br/~pf/algoritmos-livro/

EACH USP