BCC202 - Estruturas de Dados I

Aula 8: Recursividade (Parte I)

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

Conteúdo

Conceitos

Visão Geral

- A recursividade é uma estratégia que pode ser utilizada sempre que uma função f pode ser escrita em função dela própria.
- Exemplo: Cálculo do Fatorial:

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

Então: $n! = n * (n-1)!$

Há uma "definição recursiva" do problemas que quero resolver.

Definição

- Dentro do corpo de uma função, chamar novamente a própria função.
  - **Recursão direta**: a função A chama a própria função A.
  - **Recursão indireta**: a função A chama uma função B que, por sua vez, chama A.

Condição de Parada

- Nenhum programa, nem função, pode ser exclusivamente definido por si só:
  - Um programa seria um *loop* infinito.
  - Uma função teria definição circular.

```c
void func(int n ) {
    printf("%d\n", n);
    func(n);
}
```

O que aconteceria?

Condição de Parada

Permite que o procedimento pare de se executar.

```
void func(int n ) {
  printf("%d ", n);
  if(n>0){
    func(n-1);
    printf("* ");
  }
}

```

Condição de Parada: Se n é positivo (caso base).

- n inicialmente positivo.
- O valor de n é decrementado a cada chamada, logo a execução tem um fim.

Se *func*(4), o que seria impresso?

Condição de Parada

Permite que o procedimento pare de se executar.

```c
void func(int n ) {
  printf("%d ", n);
  if(n>0){
    func(n-1);
    printf("* ");
  }
}

```

Condição de Parada: Se *n* é positivo (CASO BASE).

- O valor de *n* é decrementado a cada chamada.
- *n* inicialmente positivo.
  - A execução tem um fim.

Seria impresso:43210****

Além do critério de parada ou caso base ou caso trivial e da chamada recursiva, que visa resolver uma instância menor do mesmo problema, pode existir também o processamento de apoio ou processamento complementar.

Processamento de Apoio ou Processamento Complementar

É formado pelos demais processamentos que acompanham e/ou utilizam o que resultado da chamada recursiva.

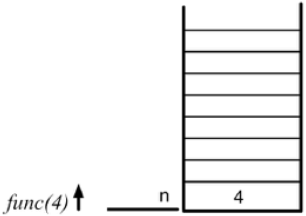
Consumo de Memória

Pilha de Execução

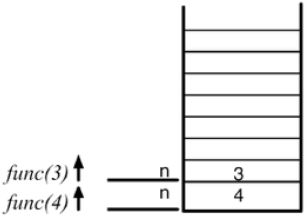
Considere $n = 4$, ou seja, $func(4)$.

```
void func(int n) {
  printf("%d  ", n);
  if(n>0){
    func(n-1);
    printf("* ";)
  }
}

```



(a) Pilha após a chamada func(4)



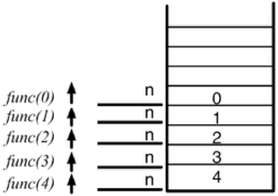
(b) Pilha após a chamada func(3)

Pilha de Execução

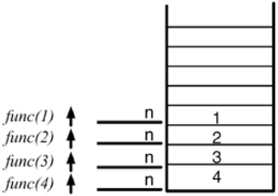
Considere $n = 4$, ou seja, $func(4)$.

```
void func(int n) {
  printf("%d  ", n);
  if(n>0){
    func(n-1);
    printf("* ";)
  }
}

```



(c) Pilha após a chamada func(0)



(d) Pilha após retorno de func(0), no contexto de func(1)

Pilha de Execução

Sobre a execução anterior:

A função é iniciadar com $func(4)$.

- Exibe o valor 4, chama $func(3)$
- chama $func(2)$
- chama $func(1)$
- chama $func(0)$
- que retorna sem chamar a função recursivamente pois n não é maior que 0.
- Até aqui, a saída é composta por 4 3 2 1 0

Pilha de Execução

Sobre a execução anterior:

Quando a chamada de *func*(0) retorna, a execução retorna para contexto de *func*(1), que após a chamada recursiva, exibe o $*$ na tela, e retorna. A execução então retorna para o contexto de *func*(2), que também imprime um $*$ e retorna, e assim por diante.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto?
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto?
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Pilha de Execução

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Consumo de Memória - Exemplo: Fatorial

```c
int fat1(int n) {
  int r;
  if(n == 0)
    r = 1;
  else
    r = n * fat1(n-1);
  return r;
}

int fat2(int n) {
  if (n == 0)
    return 1;
  else
    return n * fat2(n-1);
}
```

```c
void main() {
  int f, g;
  f = fat1(4);
  g = fat2(4);
  printf("%d -- %d", f, g);
}
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Consumo de Memória - Exemplo: Fatorial

```c
int fat1(int n) {
  int r;
  if(n == 0)
    r = 1;
  else
    r = n * fat1(n-1);
  return r;
}

int fat2(int n) {
  if (n == 0)
    return 1;
  else
    return n * fat2(n-1);
}
```

```c
void main() {
  int f, g;
  f = fat1(4);
  g = fat2(4);
  printf("%d -- %d", f, g);
}
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Consumo de Memória - Exemplo: Fatorial

```c
int fat1(int n) {
  int r;
  if(n == 0)
    r = 1;
  else
    r = n * fat1(n-1);
  return r;
}

int fat2(int n) {
  if (n == 0)
    return 1;
  else
    return n * fat2(n-1);
}
```

```c
void main() {
  int f, g;
  f = fat1(4);
  g = fat2(4);
  printf("%d -- %d", f, g);
}
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Exemplo com Fatorial

Fatorial: Elementos da Função Recursiva

1^o - **Condição de Parada ou Caso Base**

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

Fatorial: Elementos da Função Recursiva

2º - **Chamada Recursiva para uma Instância Menor do Problema**

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

Fatorial: Elementos da Função Recursiva

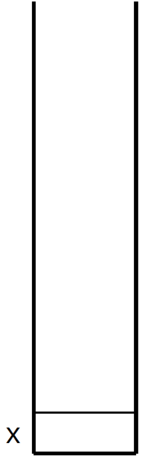
3º - Processamento Complementar

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

Fatorial: Pilha de Recursão

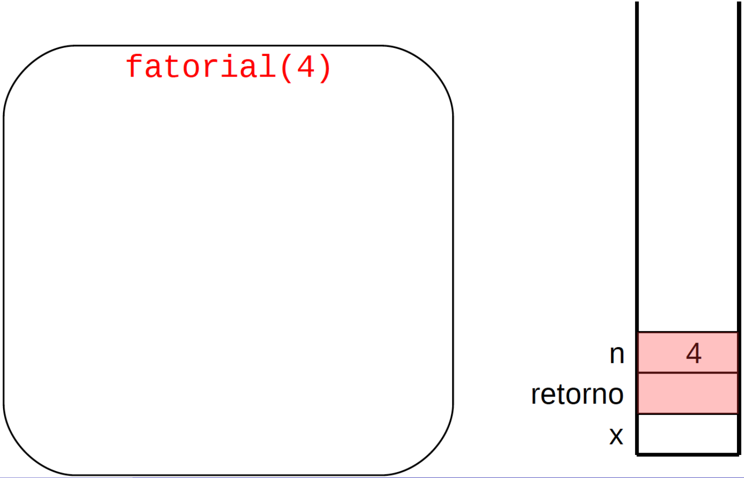
Executando a função recursiva para o cálculo de $intx = fatorial(4)$.

```
int x = fatorial(4);
```



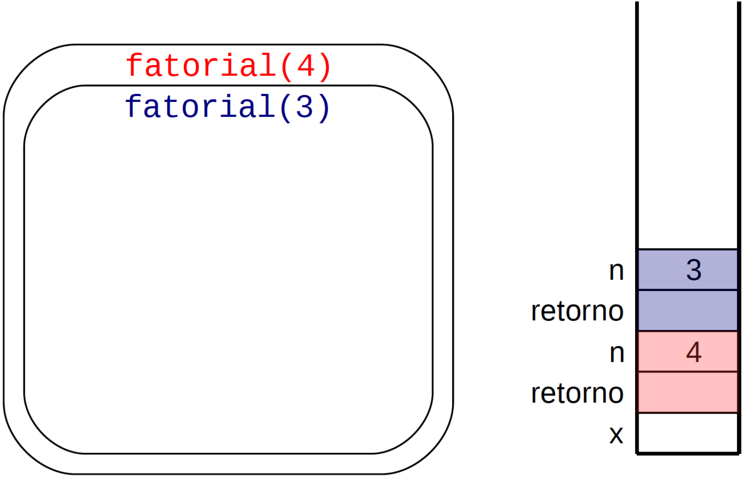
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



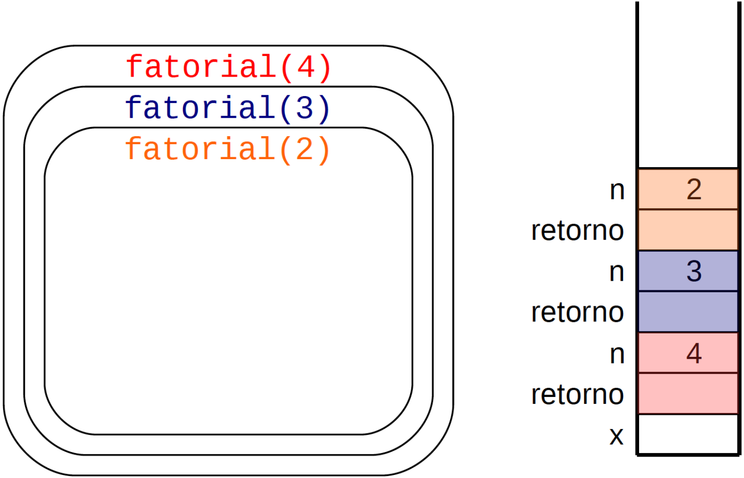
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



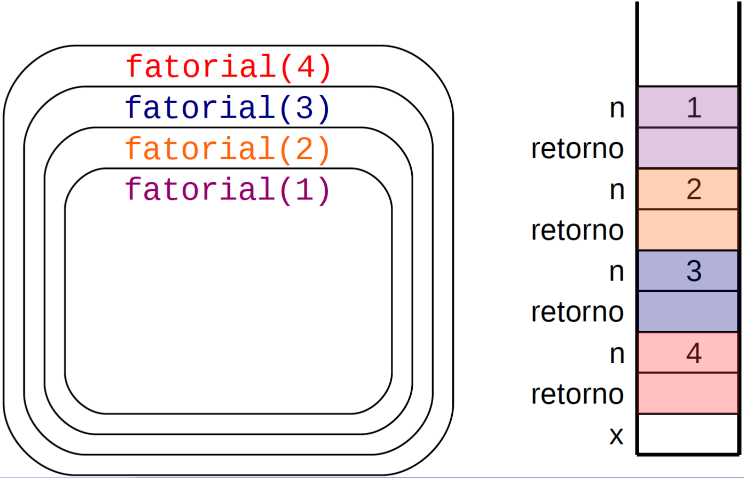
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



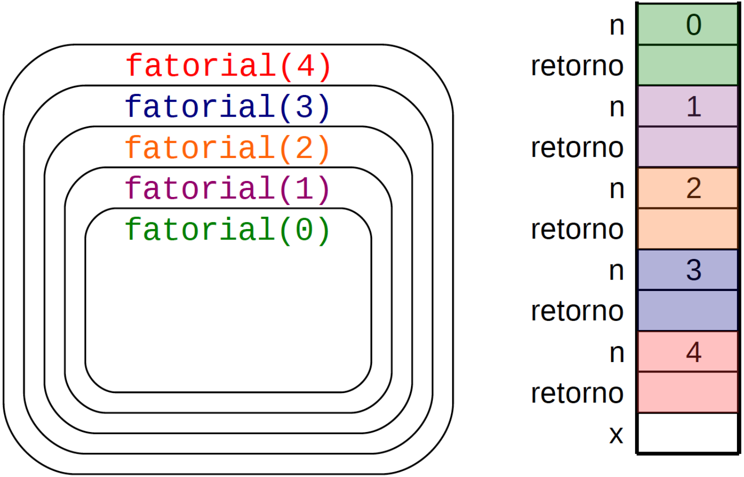
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $intx = fatorial(4)$.



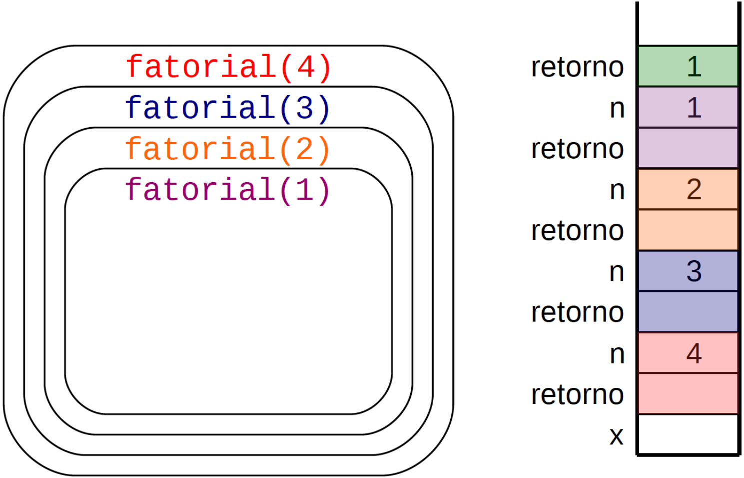
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



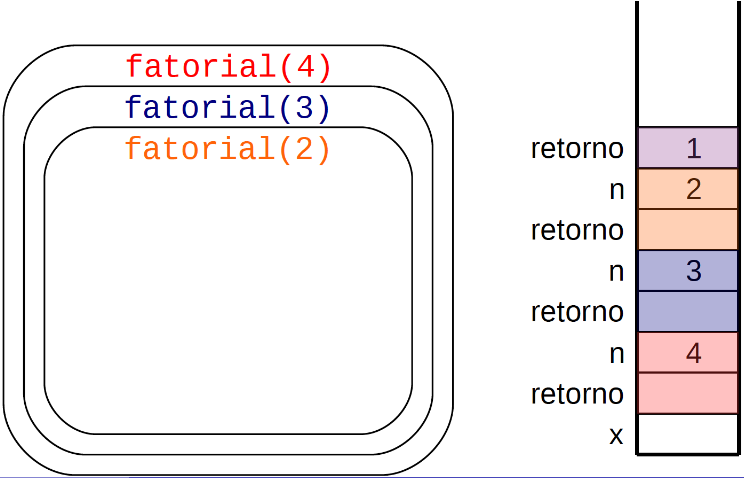
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $intx = fatorial(4)$.



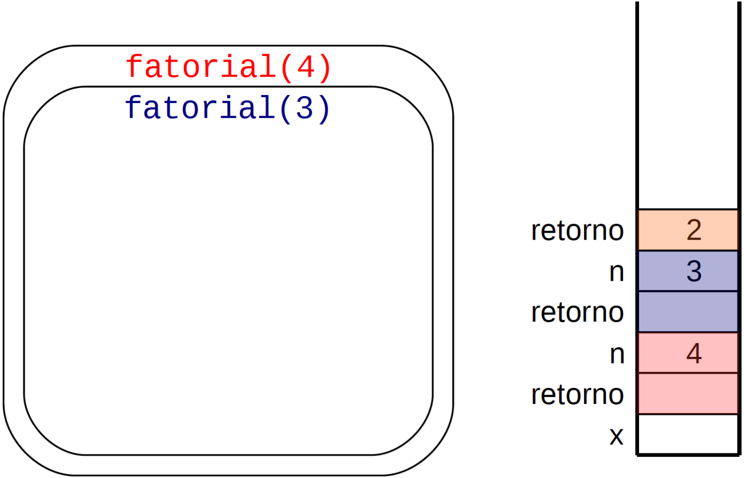
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $intx = fatorial(4)$.



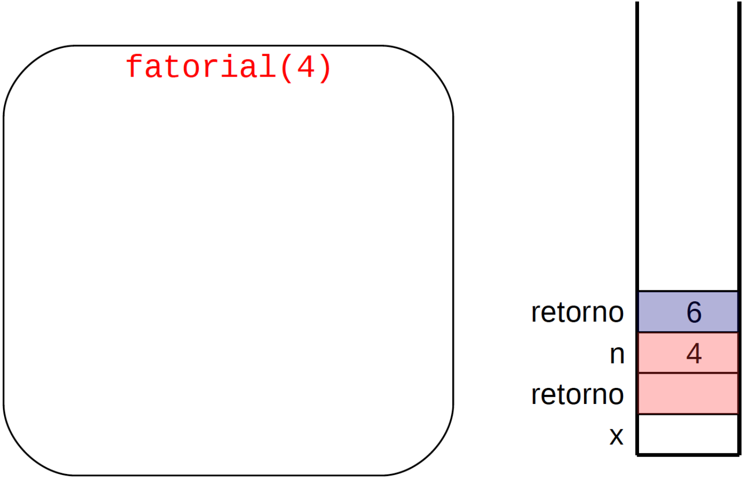
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $intx = fatorial(4)$.



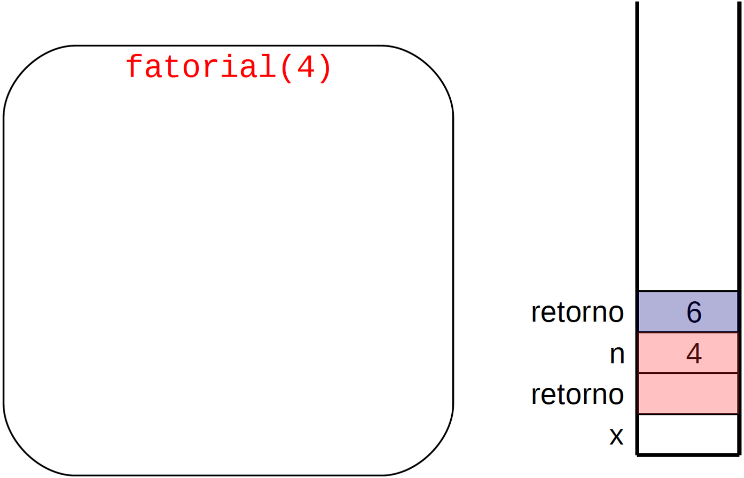
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.



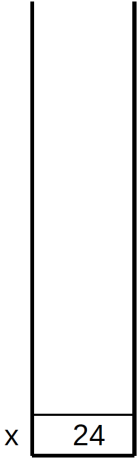
Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $intx = fatorial(4)$.

Fatorial: Pilha de Recursão

Executando a função recursiva para o cálculo de $int x = fatorial(4)$.

```
int x = fatorial(4);
```



Consumo de Memória - Exemplo: Fatorial

- A complexidade de tempo do fatorial recursivo é $O(n)$ (veremos como definir isto através de **equação de recorrência**, em breve).
- Mas a complexidade de espaço também é $O(n)$, devido à pilha de execução.
- Já no fatorial não recursivo a complexidade de espaço é $O(1)$.

```
int fatIter(int n) {
  int f = 1;
  while(n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Importante

- Portanto, podemos concluir que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos.
- Além disso, pode-se afirmar que:
  - Todo algoritmo recursivo tem uma versão não recursiva.
  - A questão é: **vale a pena implementar a versão não-recursiva?**

Exemplo: Série de Fibonacci

- Outro exemplo clássico de recursividade é a **Série de Fibonacci**, definida pela expressão:

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

- Originando a série: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```c
int fibR(int n) {
  if(n == 0)
    return 0;
  else if(n == 1)
    return 1;
  else
    return fibR(n-1) +
           fibR(n-2);
}
```

Solução Iterativa:

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

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```c
int fibR(int n) {
  if(n == 0)
    return 0;
  else if(n == 1)
    return 1;
  else
    return fibR(n-1) +
           fibR(n-2);
}
```

Solução Iterativa:

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

- Um mesmo ***n*** é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803...$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```c
int fibR(int n) {
  if(n == 0)
    return 0;
  else if(n == 1)
    return 1;
  else
    return fibR(n-1) +
           fibR(n-2);
}
```

- Um mesmo n é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803...$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Solução Iterativa:

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

- Custo: $O(n)$
- **Complexidade Linear**!

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```c
int fibR(int n) {
  if(n == 0)
    return 0;
  else if(n == 1)
    return 1;
  else
    return fibR(n-1) +
           fibR(n-2);
}
```

- Um mesmo n é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803\ldots$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Solução Iterativa:

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

- Custo: $O(n)$
- **Complexidade Linear**!

Conclusão

Não se deve utilizar recursividade cegamente!!!

Quando vale a pena usar recursividade?

- Recursividade vale a pena para algoritmos complexos, cuja implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha.
- Exemplos:
  - Dividir para Conquistar (Ex. Quicksort).
  - Caminhamento em Árvores (pesquisa, *backtracking*).

Dividir para Conquistar

Muitos algoritmos recursivos seguem uma abordagem dividir para conquistar

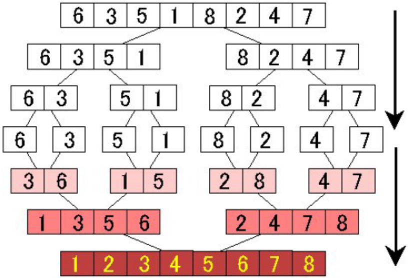
Divisão
Divida o problema em vários subproblemas mais simples.

Conquista
Conquiste os subproblemas recursivamente.

Combinação
Combine as soluções intermediárias.

Não se reduz trivialmente como Fatorial.

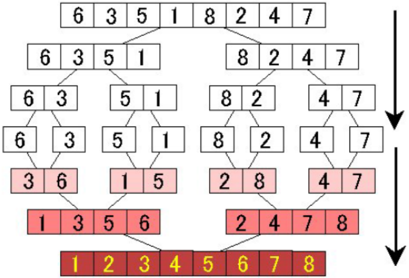
MergeSort



Onde está cada um dos passos?

- Dividir
- Conquistar
- Combinar

MergeSort

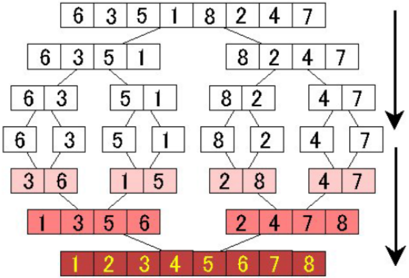


Onde está cada um dos passos?

- Dividir
- Conquistar
- Combinar

- Dividir: Divide a lista de n elementos em duas listas de $n/2$ elementos cada.

MergeSort

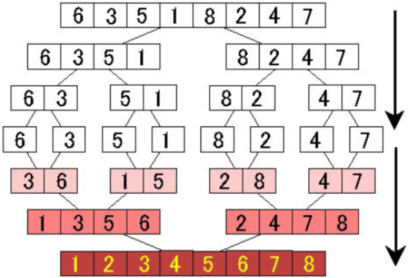


Onde está cada um dos passos?

- Dividir
- Conquistar
- Combinar

- Dividir: Divide a lista de n elementos em duas listas de $n/2$ elementos cada.
- Conquistar: Ordena cada subsequência recursivamente.

MergeSort



Onde está cada um dos passos?

- Dividir
- Conquistar
- Combinar

- Dividir: Divide a lista de n elementos em duas listas de $n/2$ elementos cada.
- Conquistar: Ordena cada subsequência recursivamente.
- Combinar: Combina as subsequências ordenadas.

Conclusão

Conclusão

- Conceitos importantes sobre **recursividade**:
  - critério de parada
  - chamadas de funções recursivas para instâncias menores
  - processamento de apoio
  - pilha de execução de funções recursivas e consumo de memória
- Poderoso paradigma de programação: **dividir para conquistar**.

Continuação de recursividade.

Exercícios

Exercício 01

- Crie uma função recursiva que calcula a potência de um número.
  - Qual a condição de parada?
  - Qual a complexidade de sua função? Apresente a equação de recorrência e resolva-a.