Recursividade

Seções 2.2 e 1.4 do livro Projeto e Análise de Algoritmos

Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Exemplos
  - Algoritmos de “Dividir para Conquistar”
  - Árvores

Exemplo

- Fatorial:
  $n! = n*(n-1)!$ p/ $n>0$
  $0! = 1$

- Em C

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

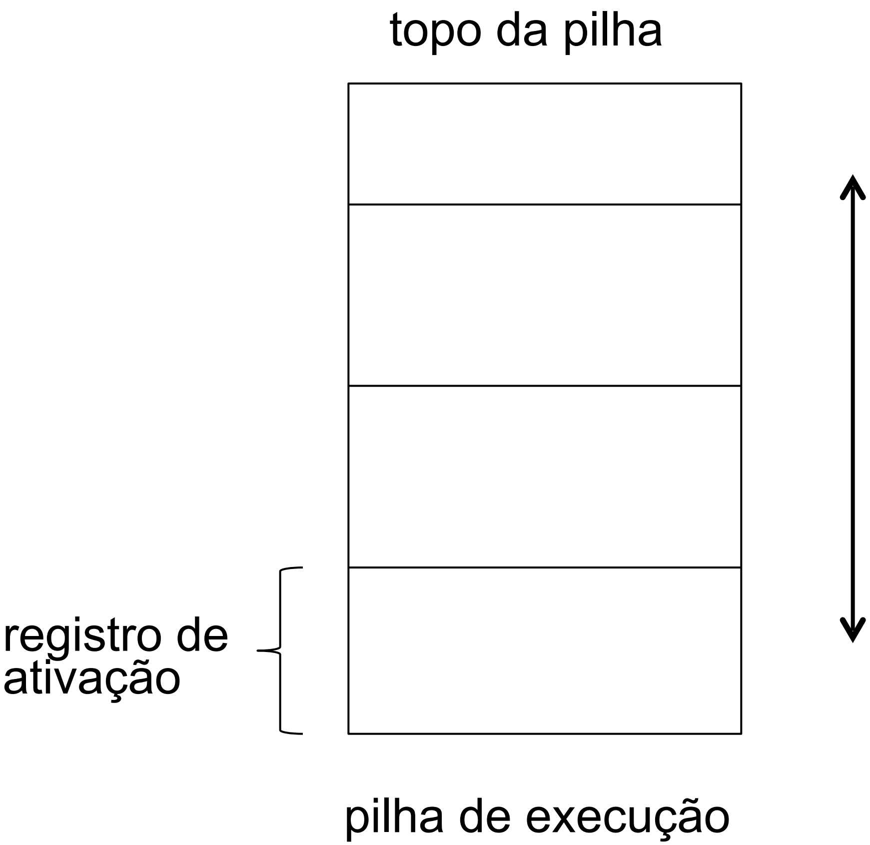
Estrutura

- Normalmente, as funções recursivas são divididas em duas partes
  - Chamada Recursiva
  - Condição de Parada
- A chamada recursiva pode ser direta (mais comum) ou indireta (A chama B que chama A novamente)
- A condição de parada é fundamental para evitar a execução de loops infinitos

Execução

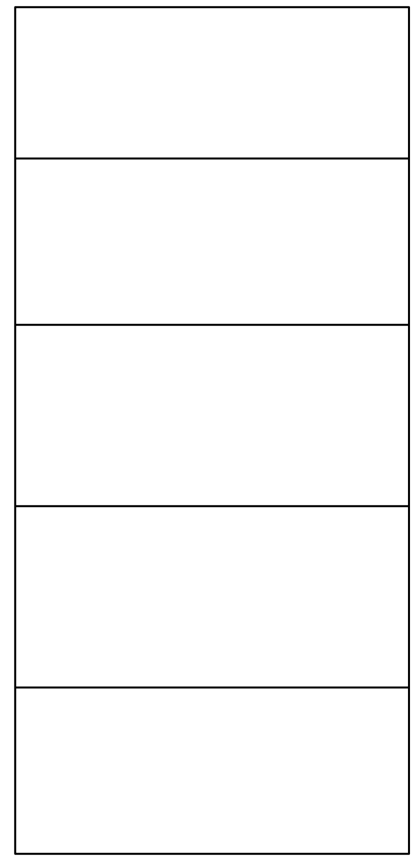
- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação** na **Pilha de Execução** do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função

Execução



Exemplo

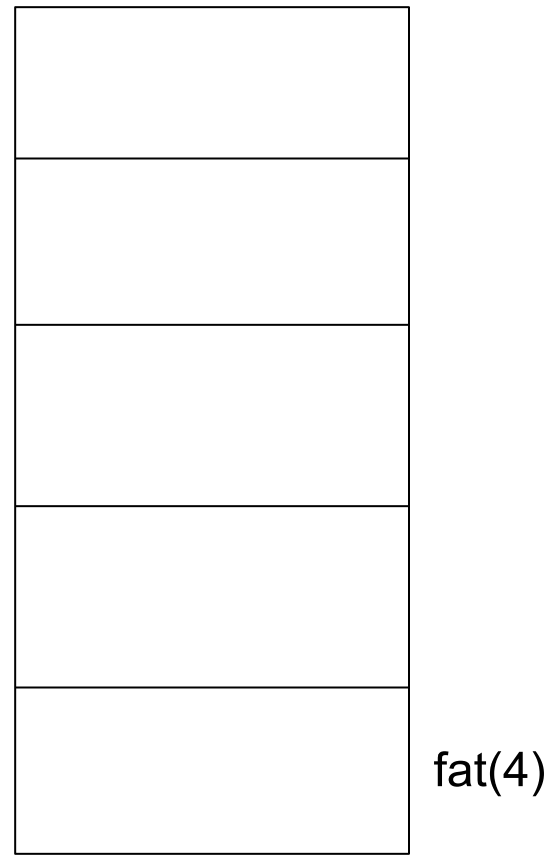
```
Fat (int n) {
    if (n<=0)
        return 1;
    else
        return n * Fat(n-1);
}


main() {
   int f;
   f = fat(4);
   printf("%d",f);
}
```



pilha de execução

Exemplo

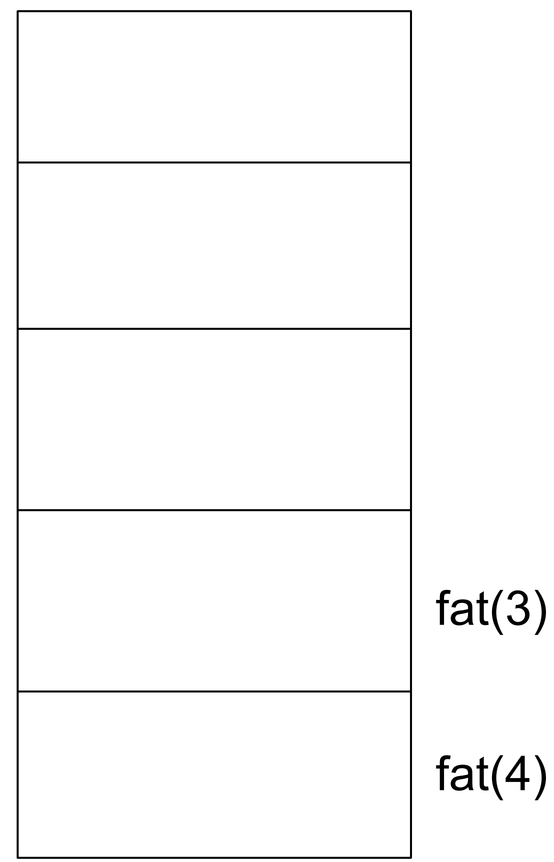
```
Fat (int n) {
    if (n<=0)
        return 1;
    else
        return n * Fat(n-1);
}


main() {
   int f;
   f = fat(4);
   printf(“%d”,f);
}
```



pilha de execução

Exemplo

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


main() {
   int f;
   f = fat(4);
   printf(“%d”,f);
}
```



pilha de execução

Exemplo

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


main() {
   int f;
   f = fat(4);
   printf("%d",f);
}
```

pilha de execução

Complexidade

- A complexidade de tempo do fatorial recursivo é O(n). (Em breve iremos ver a maneira de calcular isso usando **equações de recorrência**)
- Mas a complexidade de espaço também é O(n), devido a pilha de execução
- Já no fatorial não recursivo a complexidade de espaço é O(1)

Complexidade

- A complexidade de tempo do fatorial recursivo é O(n). (Em breve iremos ver a maneira de calcular isso usando **equações de recorrência**)
- Mas a complexidade de espaço também é O(n), devido a pilha de execução
- Já no fatorial não recursivo a complexidade de espaço é O(1)

```
Fat (int n)  {
int f;
    f = 1;
    while(n > 0){
      f = f * n;
      n = n - 1;
    }
    return f;
}
```

Recursividade

- Portanto, a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

Fibonacci

- Outro exemplo: **Série de Fibonacci:**
  - $F_n = F_{n-1} + F_{n-2}$ $\quad n > 2,$
  - $F_1 = F_2 = 1$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

```
Fib(int n) {
    if (n < 3)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}
```

Análise da função Fibonacci

- Ineficiência em Fibonacci
  - Termos F_{n-1} e F_{n-2} são computados independentemente
  - Custo para cálculo de F_n
    - $O(\phi^n)$ onde $\phi = (1 + \sqrt{5})/2 = 1{,}61803...$
    - *Golden ratio*
    - Exponencial!!!

Fibonacci não recursivo

```
int FibIter(int n) {
    int fn1 = 1, fn2 = 1;
        int fn, i;

        if (n < 3) return 1;

    for (i = 3; i <= n; i++) {
        fn += fn2 + fn1;
            fn2 = fn1;
            fn1 = fn;
    }
    return fn;
}
```

- Complexidade: O(n)
- Conclusão: não usar recursividade cegamente!

Quando vale a pena usar recursividade

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

Dividir para Conquistar

- Duas chamadas recursivas
  - Cada uma resolvendo a metade do problema
- Muito usado na prática
  - Solução eficiente de problemas
  - Decomposição
- Não produz recomputação excessiva como fibonacci
  - **Porções diferentes do problema**

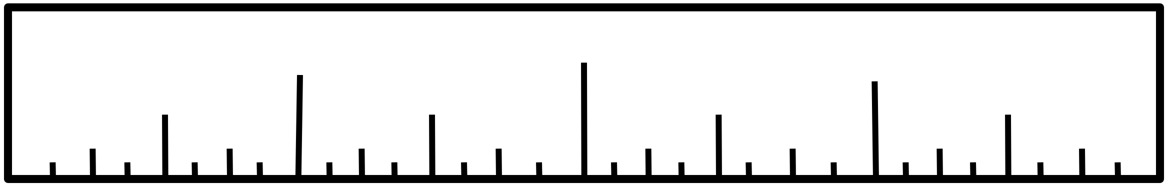
Exemplo: encontrar o máximo

```
void max(int *v, int e, int d){
int u, v;
int m = (e+d)/2;

      if (e == d) return v[e];
      u = max(v, e, m);
      v = max(v, m+1, d);
      if (u > v)
         return u;
      else
         return v;
}

max(v, 0, n-1);
```

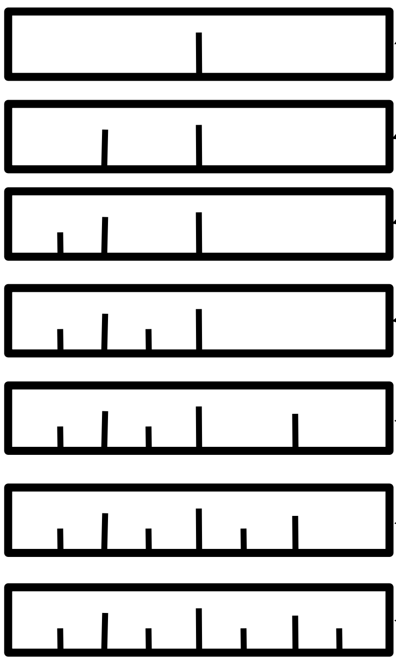
0	1	2	3	4	5	6
E	**X**	**E**	**M**	**P**	**L**	**O**

Exemplo: régua

```
void regua(int l, r, h){
int m;

      if (h > 0) {
         m = (l + r) / 2;
         marca(m, h);
         regua(l, m, h - 1);
         regua(m, r, h - 1);
      }
}
```

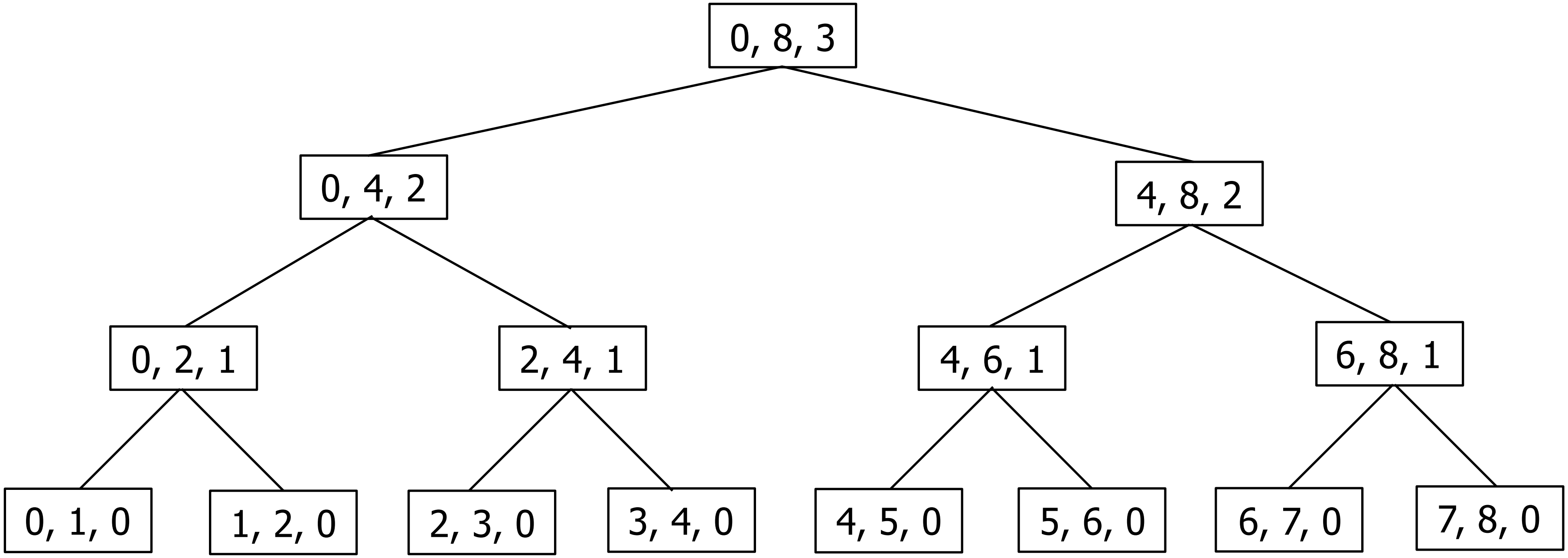


Execução: régua



```
regua(0, 8, 3)
  marca(4, 3)
  regua(0, 4, 2)
    marca(2, 2)
    regua(0, 2, 1)
      marca(1, 1)
      regua(0, 1, 0)
      regua(1, 2, 0)
    regua(2, 4, 1)
      marca(3, 1)
      regua(2, 3, 0)
      regua(3, 4, 0)
  regua(4, 8, 2)
    marca(6, 2)
    regua(4, 6, 1)
      marca(5, 1)
      regua(4, 5, 0)
      regua(5, 6, 0)
    regua(6, 8, 1)
      marca(7, 1)
       regua(6, 7, 0)
       regua(7, 8, 0)
```

Representação por árvore

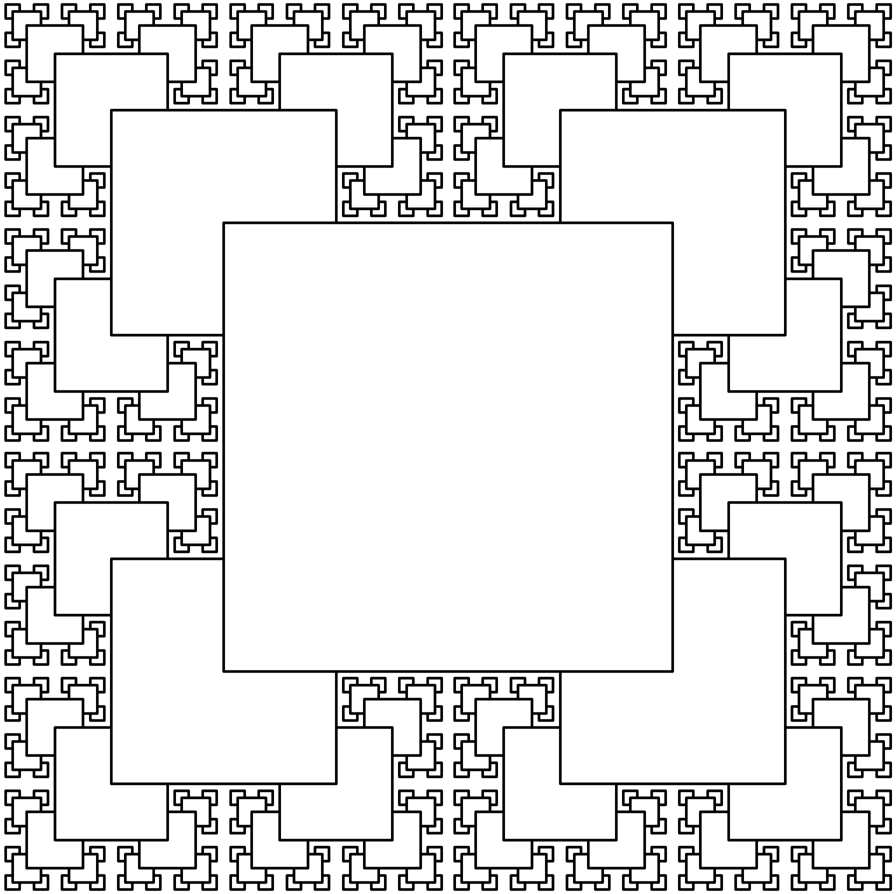


Outros exemplos de recursividade

```
void estrela(int x, y, r)
{
    if (r > 0) {
        estrela(x-r, y+r, r / 2);
        estrela(x+r, y+r, r / 2);
        estrela(x-r, y-r, r / 2);
        estrela(x+r, y-r, r / 2);
        box(x, y, r);
    }
}
```

x e y são as coordenadas do centro.
r o valor da metade do lado

Outros exemplos de recursividade



```
void estrela(int x, y, r)
{
    if (r > 0) {
        estrela(x-r, y+r, r / 2);
        estrela(x+r, y+r, r / 2);
        estrela(x-r, y-r, r / 2);
        estrela(x+r, y-r, r / 2);
        box(x, y, r);
    }
}
```

x e y são as coordenadas do centro.
r o valor da metade do lado