Slides originais de Gisele Pappa

Recursividade

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

Exemplo

- Fatorial
  - $0! = 1$
  - $n! = n(n\text{-}1)!$
- Fatorial em C

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

Exemplo

- Normalmente, as funções recursivas são divididas em duas partes
  - Condição de parada
  - Chamada recursiva

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

Exemplo

- Normalmente, as funções recursivas são divididas em duas partes
  - Condição de parada
    - Evitar loops infinitos
  - Chamada recursiva
    - Pode ser direta (mais comum) ou indireta

Execução

- Quando uma chamada de função é feita, é criado um **registro de ativação** na pilha de execução do programa
- O registro de ativação guarda
  - Os parâmetros e variáveis locais da função
  - O ponto de retorno da função
- Quando a função termina, o registro de ativação é desempilhado e a execução volta ao subprograma que chamou a função

Exemplo

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

int main() {
    int f = fatorial(5);
    printf("%d", f);
}
```

Complexidade (1)

- A complexidade de tempo do fatorial recursivo é $O(n)$
  - Calculado via equações de recorrência
- Complexidade de espaço também é $O(n)$
- Na implementação não recursiva, a complexidade de espaço é $O(1)$

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

Complexidade (2)

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

Exemplo – Série de Fibonacci

- $F_n = F_{n-1} + F_{n-2}, n > 2$
- $F_1 = F_2 = 1$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

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

Exemplo – Série de Fibonacci

- Ineficiente
  - Termos F_{n-1} e F_{n-2} são computados independentemente e repetidas vezes
  - Número de chamadas recursivas é igual ao número de fibonacci sendo calculado
  - Custo para o cálculo de F_n
    - $O(\varphi^n)$
    - $\varphi = (1 + 5^{1/2})/2 = 1,61803$ é a proporção áurea
    - Complexidade *exponencial*

Exemplo – Série de Fibonacci

- Implementação iterativa:

```
int fib(int n) {
    int f_m2 = 1; int f_m1 = 1;
    int f;
    for(int i = 2; i <= n; i++) {
        f = f_m1 + f_m2;
        f_m2 = f_m1; f_m1 = f;
    }
    return f;
}
```

- Complexidade de tempo: $O(n)$
- Complexidade de espaço: $O(1)$

Quando é útil usar recursividade

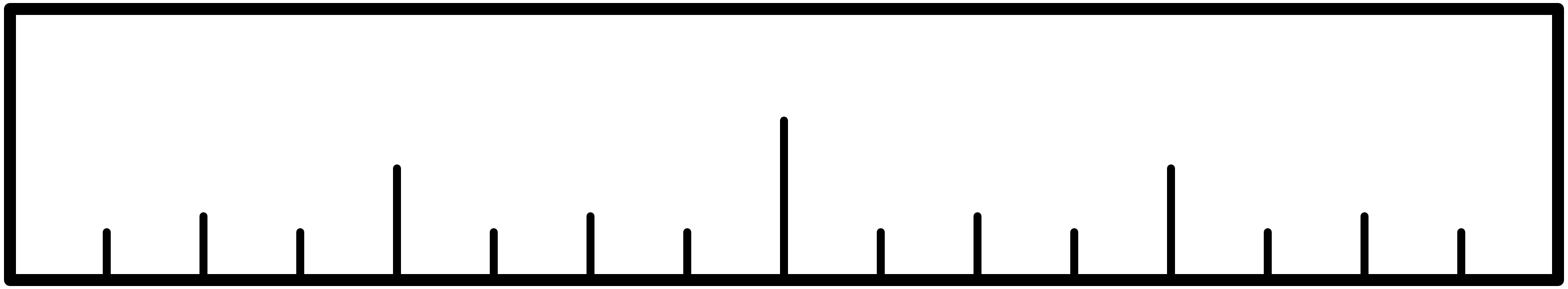
- Problemas cuja implementação iterativa é complexa e requer uso explícito de uma pilha
  - Algoritmos tipo dividir para conquistar (quicksort)
  - Caminhamento em árvores
  - Busca exaustiva

Dividir para conquistar

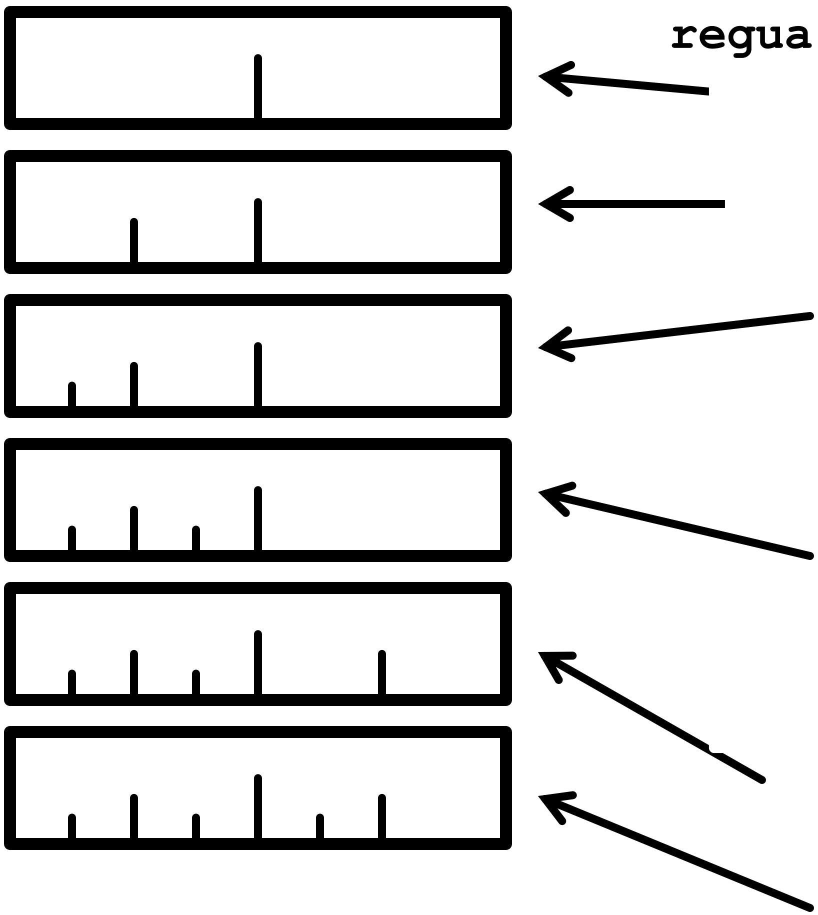
- Duas (ou mais) chamadas recursivas
  - Cada chamada resolve metade do problema
  - Não fazem recomputação excessiva como o exemplo do cálculo da série Fibonacci recursivo
    - Cada função deve operar sobre partes diferentes do problema
- Muito usado na prática
- Problemas de dividir para conquistar não se reduzem trivialmente como fatorial

Exemplo – Régua

```
void regua(int esq, dir, alt){
    if(alt <= 0) return;
    int m = (esq + dir) / 2;
    marca(m, alt);
    regua(esq, m, alt - 1);
    regua(m, dir, alt - 1);
}
```

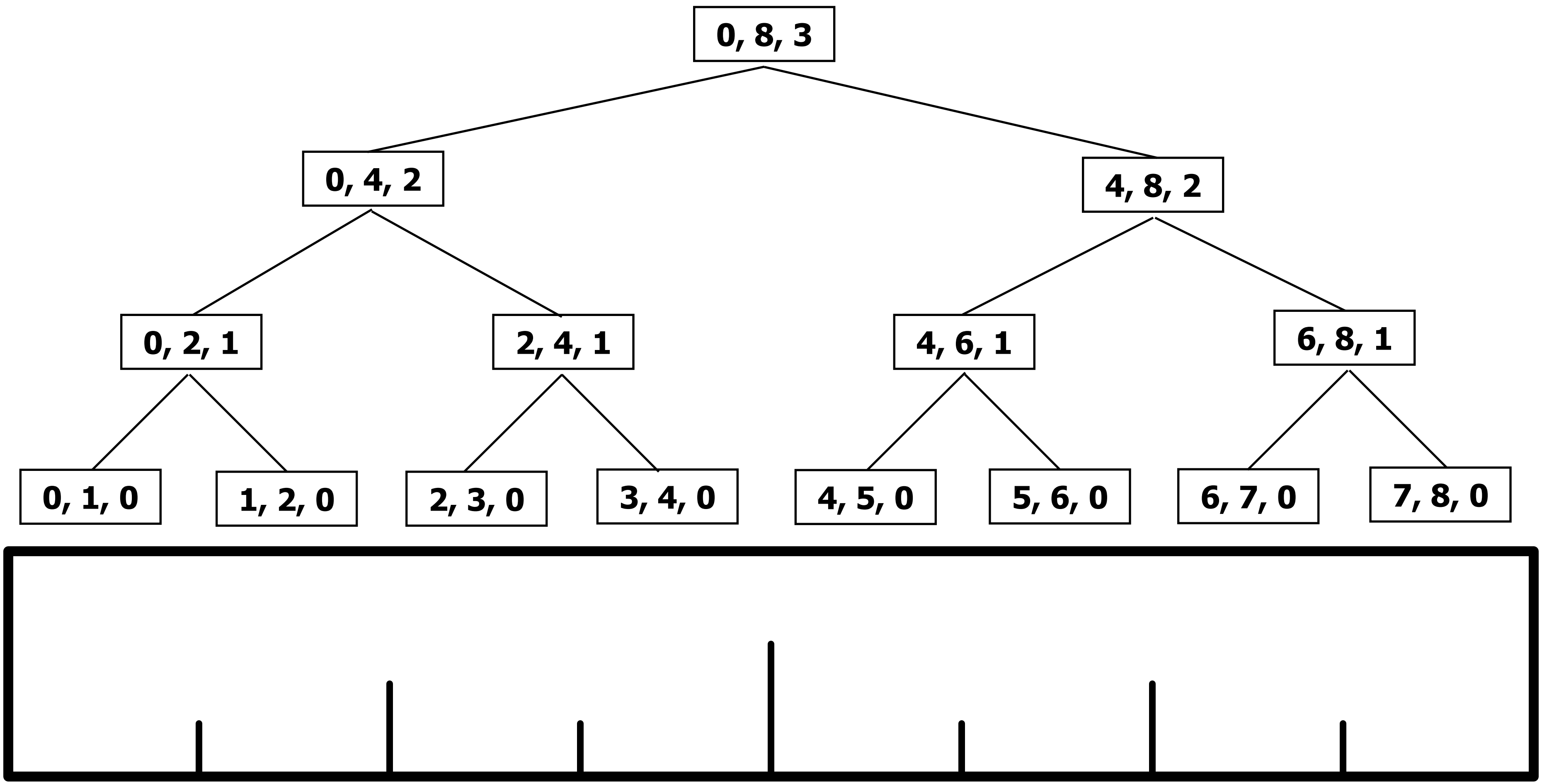


Exemplo – Régua

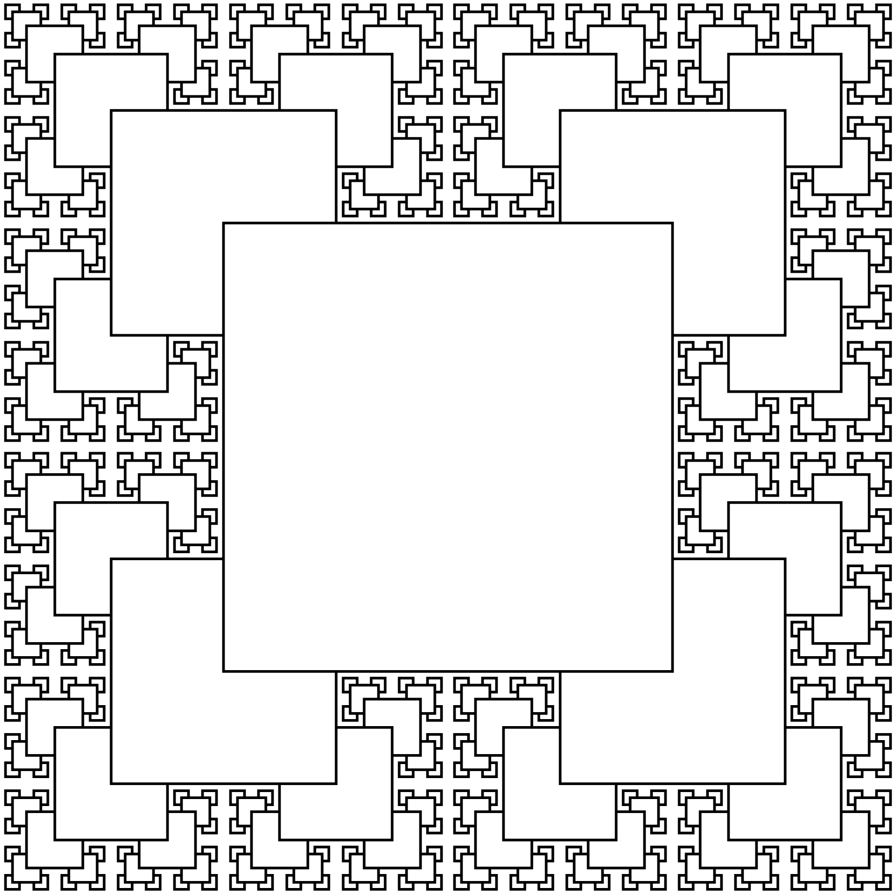


`regua(0, 8, 3)`

Exemplo – Régua
Representação em árvore



Outros exemplos: fractais



```
void fractal(int x, y, r)
{
    if(r <= 0) { return; }
    fractal(x-r, y+r, r / 2);
    fractal (x+r, y+r, r / 2);
    fractal (x-r, y-r, r / 2);
    fractal (x+r, y-r, r / 2);
    quadrado(x, y, r);
}
```

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

Exercícios

- Implemente uma função recursiva para calcular o valor de 2^n

- O que faz a função abaixo?

```
/* considere a > b */
int f(int a, int b) {
    if(b == 0) { return a; }
    return f(b, a % b);
}
```

Respostas

- ```
  int dois_a_n(int n) {
      if(n == 0) { return 1; }
      return 2*dois_a_n(n-1);
  }
  ```

- Algoritmo de Euclides pra cálculo do máximo divisor comum de dois números
  ```