Análise de Problemas Recursivos

Algoritmos e Estruturas de Dados 2
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

Lembrando de Recursividade

- Procedimento que chama a si mesmo

- Recursividade permite descrever alguns algoritmos de forma mais clara
  - Divisão e Conquista
  - Árvores

- Vamos ver um algoritmo recursivo

Fatorial

- Como computar um fatorial?

Fatorial

- Como computar um fatorial?

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

Fatorial

- Como computar um fatorial?

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

- f(0) = 1
- f(1) = 1
- f(n) = n * f(n-1)

Fatorial em C

- Poucas linhas de código
- Como seria com um for?

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

Fatorial em C

- Poucas linhas de código
- Como seria com um for?
- Qual a complexidade da função ao lado?

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

Exemplo C Tutor

https://goo.gl/v3hrbl

Pilha de Execução

$0! = 1$

$1! = 1 * 0!$

$2! = 2 * 1!$

$3! = 3 * 2!$

$4! = 4 * 3!$

$n! = n * (n - 1)!$: fórmula geral

$0! = 1$: caso-base

Exemplificando as Chamadas

0! = 1

1! = 1 * 0!

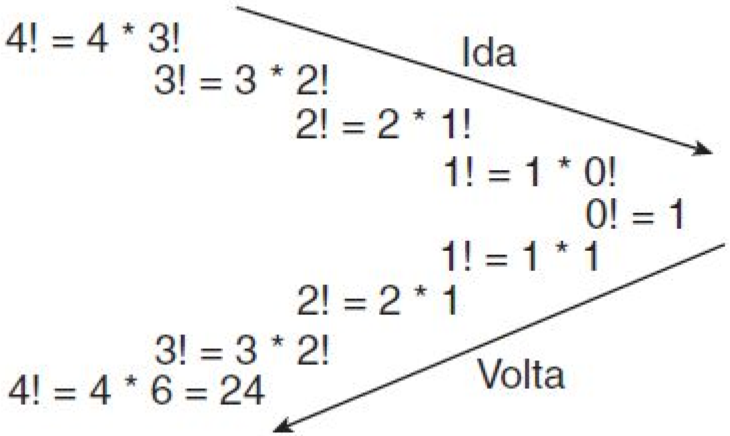
2! = 2 * 1!

3! = 3 * 2!

4! = 4 * 3!

n! = n * (n - 1)! : fórmula geral

0! = 1 : caso-base



Comparando Com e Sem Recursão

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

```c
int fat(int n) {
  if (n <= 0)
    return 1;
  else {
    int i;
    int f = 1;
    for (i = 2; i <= n; i++)
      f = f * i;
    return f;
  }
}
```

Complexidade

- Caso Base (n <= 0):
  - 1 operação

- Caso geral:
  - 1 multiplicação
  - Custo da chamada recursiva

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

Complexidade

- Vamos fazer com n = 7:

- T(7) = 1 + T(6)
- T(6) = 1 + T(5)
- T(5) = 1 + T(4)
- T(4) = 1 + T(3)
- T(3) = 1 + T(2)
- T(2) = 1 + T(1)
- T(1) = 1 + T(0)
- T(0) = 1 //caso base

Complexidade

- Vamos fazer com $n = 7$:

- $T(7) = 1 + T(6)$
- $T(6) = 1 + T(5)$
- $T(5) = 1 + T(4)$
- $T(4) = 1 + T(3)$
- $T(3) = 1 + T(2)$
- $T(2) = 1 + T(1)$
- $T(1) = 1 + T(0)$
- $T(0) = 1$ //caso base

- $T(7) = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$

Recorrência

- Complexidade de funções recursiva é determinada por equações de recorrência. Geralmente vão ser das seguintes formas:

- $T(n) = a * T(n-b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

- ou

- $T(n) = a * T(n/b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

Recorrência

- Complexidade de funções recursiva é determinada por equações de recorrência. Geralmente vão ser das seguintes formas:

- $T(n) = a * T(n-b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

- ou

- $\boldsymbol{T(n) = a * T(n/b) + f(n)}$, ***para o caso geral*** $\boldsymbol{(n > c)}$
- $\boldsymbol{T(n) = k}$, ***para o caso base*** $\boldsymbol{(n <= c)}$

Segundo caso é mais comum. Primeiro é mais simples para o fatorial

Complexidade do Fatorial

- $T(n) = a * T(n\text{-}b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

- a = 1
  - Apenas 1 chamada de fat
  - Afetado por laços geralmente
- b = 1
  - Do código ao lado
- k = 1
  - apenas 1 return
- f(n) = 1
  - 1 multiplicação

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

Complexidade do Fatorial

- $T(n) = a * T(n\text{-}b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

- $T(n) = 1 * T(n\text{-}1) + 1$

$T(n) = T(n\text{-}1) + 1$
$T(n) = T(n\text{-}2) + 1 + 1$
$T(n) = T(n\text{-}3) + 1 + 1 + 1$
…

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

Complexidade do Fatorial

- *T(n) = a * T(n-b) + f(n), para o caso geral (n > c)*
- *T(n) = k, para o caso base (n <= c)*

- *T(n) = 1 * T(n-1) + 1*

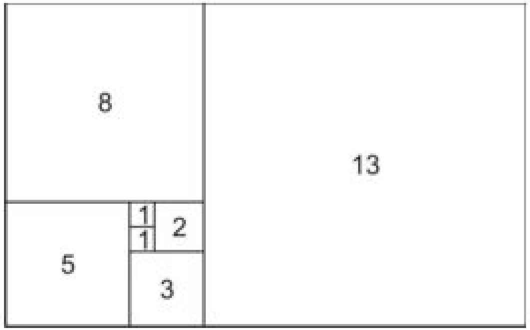
$T(n) = T(n-1) + 1$
$T(n) = T(n-2) + 1 + 1$
$T(n) = T(n-3) + 1 + 1 + 1$
…

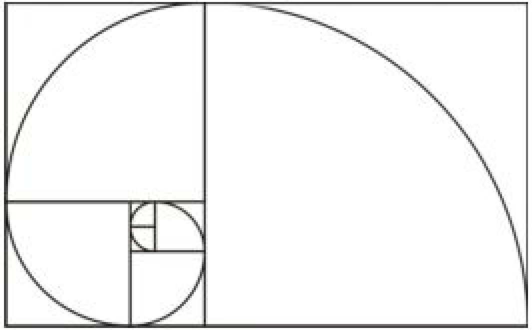
$$\sum_{i=0}^{n} 1 = (n+1) = O(n)$$

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

Outro Exemplo: Fibonacci

- $F(1) = F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$, *quando* $n > 2$





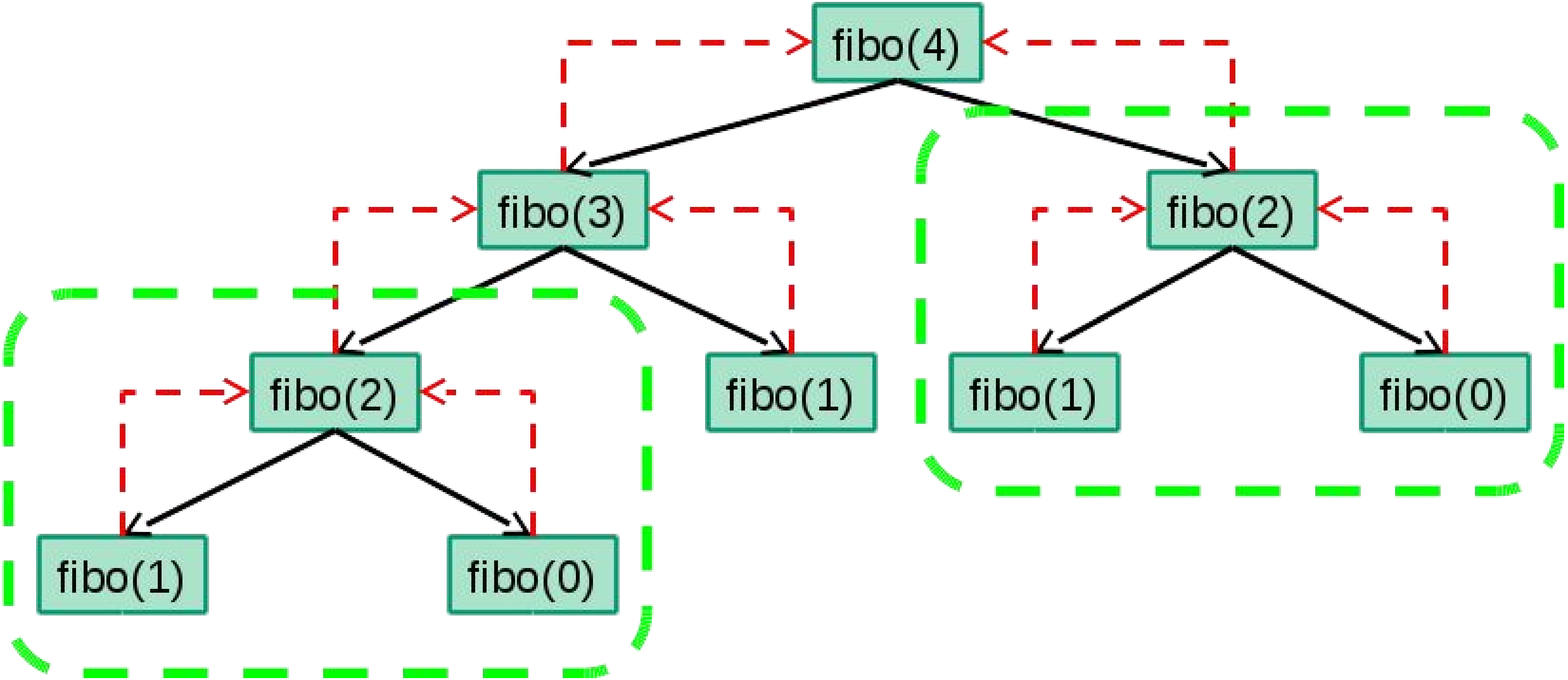
Outro Exemplo: Fibonacci

- *F(1) = F(2) = 1*
- *F(n) = F(n-1) + F(n-2), quando n > 2*

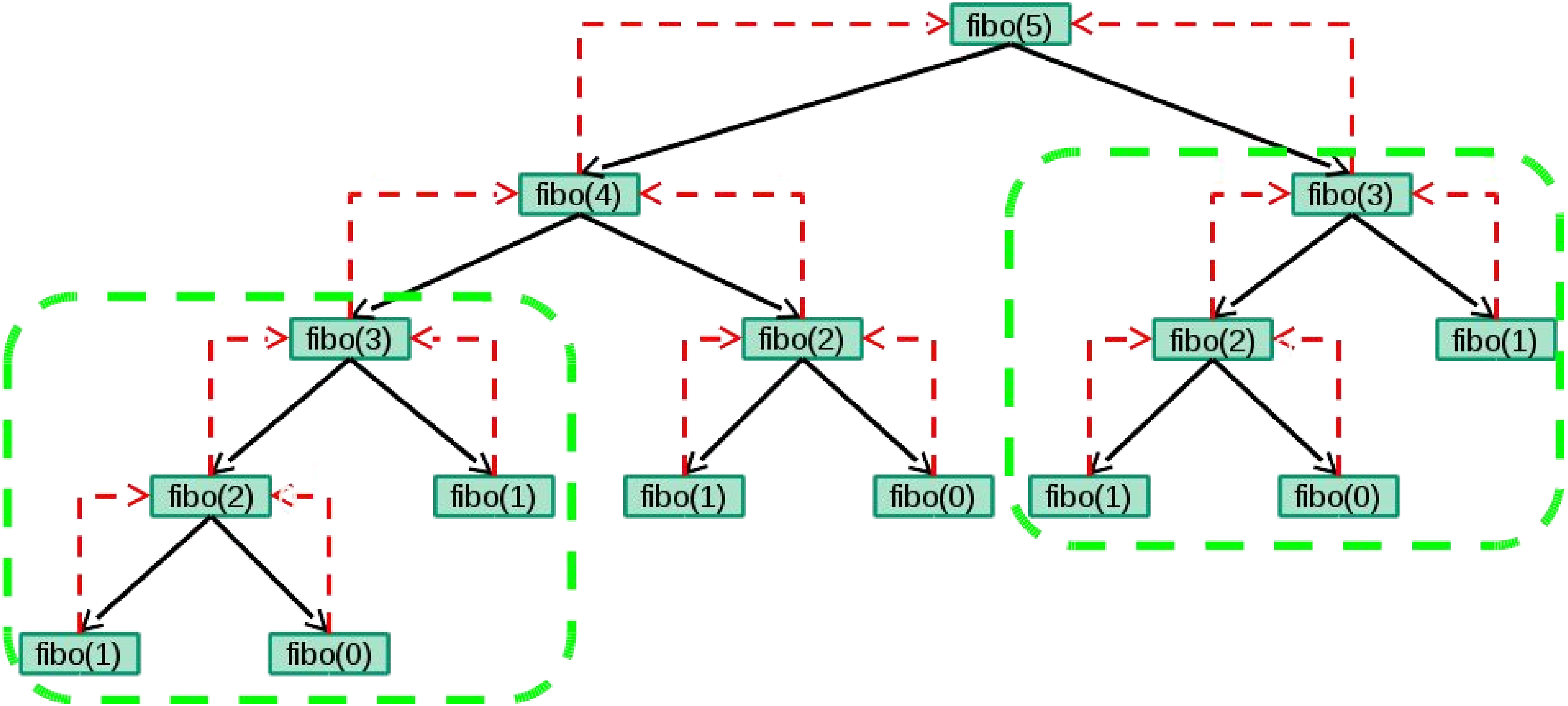
```
int fibo(int n) {
    if (n == 0 || n == 1)
        return n;
    else
        return fibo(n-1) + fibo(n-2);
}
```

Qual o problema deste algoritmo?

Fibo(4)



Fibo(5)


fibo(5)
fibo(4)
fibo(3)
fibo(3)
fibo(2)
fibo(2)
fibo(1)
fibo(2)
fibo(1)
fibo(1)
fibo(0)
fibo(1)
fibo(0)
fibo(1)
fibo(0)

Custo Exponencial

- Não podemos usar as equações de recorrência anteriores

- Custo exponencial

- $T(n) = T(n\text{-}1) + \mathbf{T(n\text{-}2)} + 1$
  $= \mathbf{T(n\text{-}2)} + \mathbf{T(n\text{-}3)} + \mathbf{T(n\text{-}3)} + T(n\text{-}4)$
  $= \ldots.$

- Bastante computação repetida
- Árvore computacional de tamanho aproximado 2^n (ver slide anterior)

Fibonacci Linear

- Fácil de fazer com um for apenas

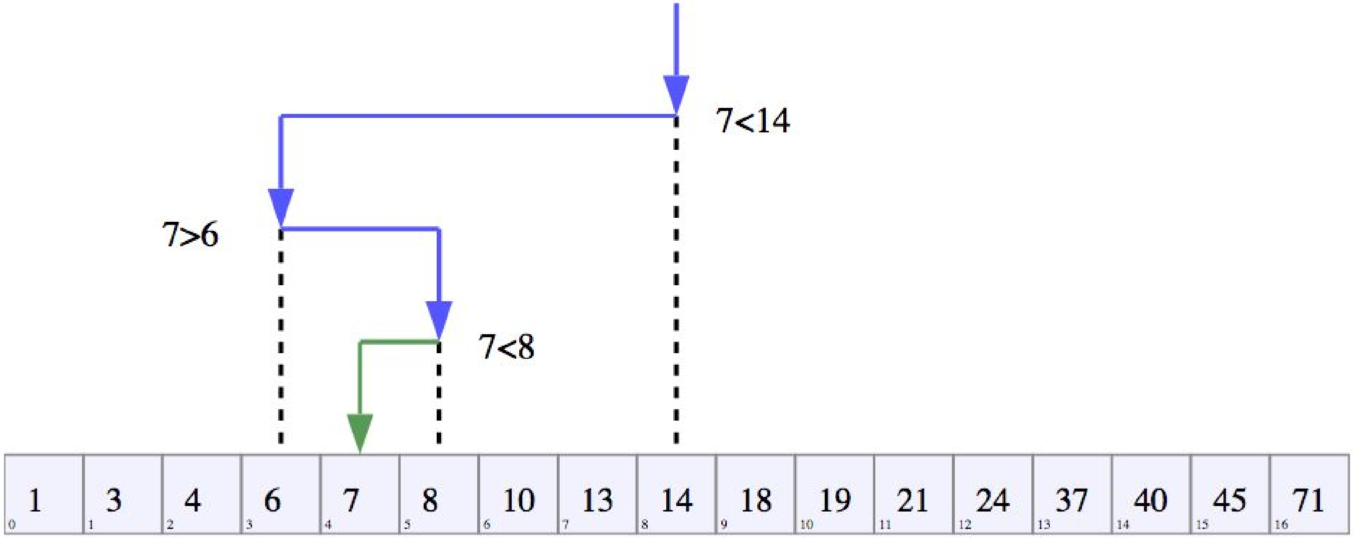
- Custo Linear

- [Lição] Recursividade é elegante mas nem sempre é eficiente
    - Queremos quebrar em subproblemas menores
    - Fibonacci recursivo não faz isto

- $T(n) = a * T(n/b) + f(n)$, *para o caso geral* $(n > c)$
    - Problema menor de tamanho n/b que é chamado a vezes
- $T(n) = k$, *para o caso base* $(n <= c)$

Mais um Problema

- Busca Binária
- Achar um elemento em um vetor ordenado
- https://goo.gl/IzE8HW



```c
int buscaBinariaLimites(int dados[], int limInf, int limSup, int alvo) {
  if (limSup < limInf) {
    return -1;
  }
  int meio = (limInf + limSup) / 2;

  if (alvo == dados[meio]) { //Achamos o elemento alvo
    return meio;
  }
  else {
    if (alvo < dados[meio]) {
      return buscaBinariaLimites(dados, limInf, meio - 1, alvo); //Problema N/2
    }
    else {
      return buscaBinariaLimites(dados, meio + 1, limSup, alvo); //Problema N/2
    }
  }
}
```

Complexidade

- $T(n) = a * T(n/b) + f(n)$, *para o caso geral* $(n > c)$
- $T(n) = k$, *para o caso base* $(n <= c)$

- $k = 1$
  - *Retornamos -1 caso não encontramos o elemento*
- $f(n)$
  - *Custo constante c*
  - *Basta olhar o código, sem laços apenas ifs*
- $a = 1$
  - *1 chamada recursiva*
- $b = 2$
  - *Dividimos o problema na metade a cada chamada*

Complexidade

- *T(n) = a * T(n/b) + f(n), para o caso geral (n > c)*
- *T(n) = k, para o caso base (n <= c)*

- *T(n) = T(n/2) + c*

$$T(n) = T(n/2) + c$$
$$T(n/2) = T(n/4) + c$$
$$T(n/4) = T(n/8) + c$$
$$\vdots$$
$$T(n/2^{k-1}) = T(n/2^k) + c$$

Complexidade

- Como resolver?
    - Faça $n = 2^x$
    - Logo $x = log_2 n$
    - Intuitivamente: quantas vezes eu consigo dividir n por 2 até chegar em 1?

- $T(2^x) = T(2^{x-1}) + c$
  $T(2^{x-1}) = T(2^{x-2}) + c$
  $T(2^{x-2}) = T(2^{x-3}) + c$
  $T(2^{x-3}) = T(2^{x-4}) + c$
  $T(2^{x-4}) = T(2^{x-5}) + c$
  $\ldots$
  $T(2^2) = T(2^1) + c$
  $T(2^1) = T(2^0) + c$

Complexidade

- Como resolver?
  - Faça $n = 2^x$
  - Logo $x = log_2 n$
  - Intuitivamente: quantas vezes eu consigo dividir n por 2 até chegar em 1?

- $T(2^x) = T(2^{x-1}) + c$
  $T(2^{x-1}) = T(2^{x-2}) + c$
  $T(2^{x-2}) = T(2^{x-3}) + c$
  $T(2^{x-3}) = T(2^{x-4}) + c$
  $T(2^{x-4}) = T(2^{x-5}) + c$
  $\ldots$
  $T(2^2) = T(2^1) + c$
  $T(2^1) = T(2^0) + c$

Some os 2 lados
As chamadas recursivas são somas

Complexidade

- $T(2^x) + T(2^{x-1}) + T(2^{x-2}) + \ldots = T(2^{x-1}) + c + T(2^{x-2}) + c + T(2^{x-3}) + c + \ldots + c$

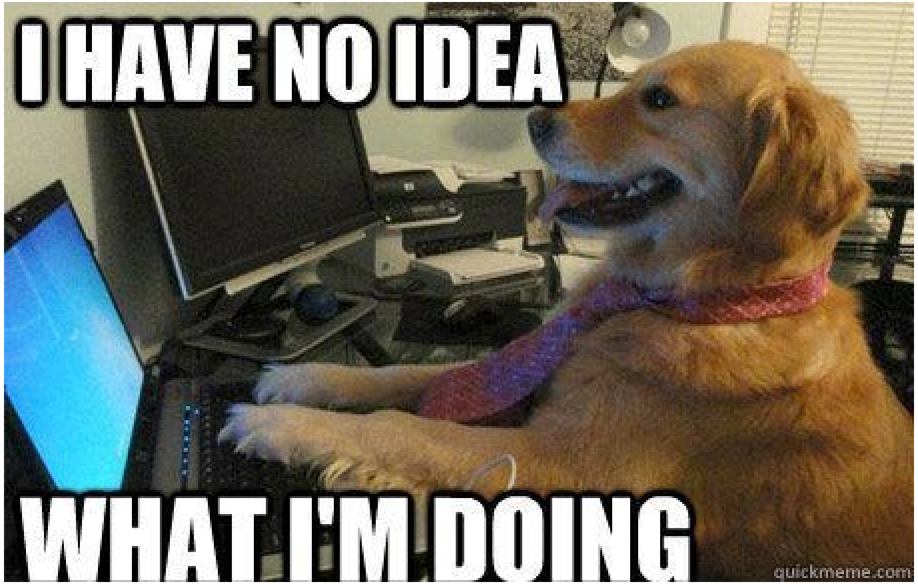
  cortando tudo que é igual

  $T(2^x) = c + cx$

  fazendo $x = log_2 n$

  $T(n) = c + c * log_2 n$
  $T(n) = O(log_2 n)$

Talvez você esteja se sentindo assim....



Normal. Mas existe solução!

Teorema Mestre

- Receita de bolo para os problemas acima

- Serve para recorrências do tipo

  $T(n) = a * T(n-b) + f(n)$, *para o caso geral* $(n > c)$
  $T(n) = k$, *para o caso base* $(n <= c)$

- Nem sempre se aplica
  - Serve em vários casos

Teorema Mestre

- Seja $T(n) = aT(n/b) + f(n)$

$$\text{Se } f(n) = O(n^{\log_b a - \varepsilon}), \text{com } \varepsilon > 0,$$

$$\text{temos } T(n) = \Theta(n^{\log_b a}).$$

$$\text{Se } f(n) = \Theta(n^{\log_b a}),$$

$$\text{temos } T(n) = \Theta(n^{\log_b a} \log n).$$

$$\text{Se } f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ para } \varepsilon > 0 \text{ e se } af(n/b) \le cf(n) \text{ para } c < 1,$$

$$\text{temos } T(n) = \Theta(f(n)).$$

Busca Binária

- Segundo caso

- Vamos aplicar?