DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estruturas de Dados

Análise de complexidade de algoritmos

Professores: Anisio Lacerda
Wagner Meira Jr.

Análise de complexidade de algoritmos

Nosso objetivo agora é calcular a ordem de complexidade de nossos algoritmos.

- Análise de algoritmos iterativos simples
- Análise de algoritmos recursivos
  - Como obter uma relação de recorrência
  - Como calcular a ordem de complexidade a partir da relação de recorrência

Algoritmos iterativos

- Em algoritmos mais simples calculamos a função de complexidade.

```
void func(int *v, int n) {
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        sum += v[i];

    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            sum += v[j];
}
```

Algoritmos iterativos

```
void func(int *v, int n) {
    int i, j, sum = 0;

   for (i = 0; i < n; i++)
      sum += v[i];

   for (i = 1; i < n; i++)
      for (j = 1; j < n; j++)
         sum += v[j];
}
```

Algoritmos iterativos

```
void func(int *v, int n) {
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        sum += v[i];

    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            sum += v[j];
}
```

n

$$f(n) = n + (n-1)(n-1) = n^2 - n + 1$$

Algoritmos iterativos

```
void func(int *v, int n) {
    int i, j, sum = 0;
```

n

```
    for (i = 0; i < n; i++)
        sum += v[i];
```

$n-1$

```
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            sum += v[j];
}
```

$$f(n) = n + (n-1)(n-1) = n^2 - n + 1$$

Algoritmos iterativos

```
void func(int *v, int n) {
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        sum += v[i];

    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            sum += v[j];
}
```

n

$n-1$

$n-1$

Algoritmos iterativos

```
void func(int *v, int n) {
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        sum += v[i];

    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            sum += v[j];
}
```

n

$n-1$

$n-1$

$$f(n) = n + (n-1)(n-1) = n^2 - n + 1$$

Algoritmos iterativos

- Agora vamos mostrar que a função de complexidade pertence a $O(n^2)$.

$$\exists c, m > 0 \text{ tais que}$$
$$n^2 - n + 1 \leq cn^2, \forall n \geq m$$

$$\text{tome } c = 1 \text{ e } m = 1$$
$$n^2 - n + 1 \leq n^2, \forall n \geq 1$$
$$-n + 1 \leq 0, \forall n \geq 1$$
$$1 \leq n, \forall n \geq 1$$

Algoritmos iterativos

- Vamos observar agora os algoritmos vistos na aula 1.

Algoritmos iterativos

- Vamos observar agora os algoritmos vistos na aula 1.

```
int Max(int *A, int n){
    int i, temp;

   temp = A[0];
   for (i = 1; i < n; i++)
      if (temp < A[i])
        temp = A[i];
    return Temp;
}
```

Algoritmos iterativos

- Vamos observar agora os algoritmos vistos na aula 1.

```
int Max(int *A, int n){
    int i, temp;

   temp = A[0];
   for (i = 1; i < n; i++)
      if (temp < A[i])
         temp = A[i];
    return Temp;
}
```

$n-1$

Algoritmos iterativos

- Vamos observar agora os algoritmos vistos na aula 1.

```
int Max(int *A, int n){
    int i, temp;

   temp = A[0];
   for (i = 1; i < n; i++)
       if (temp < A[i])
          temp = A[i];
    return Temp;
}
```

$$f(n) = n - 1$$

$$n-1$$

Algoritmos iterativos

- Vamos observar agora os algoritmos vistos na aula 1.

```
int Max(int *A, int n){
    int i, temp;

   temp = A[0];
   for (i = 1; i < n; i++)
      if (temp < A[i])
         temp = A[i];
    return Temp;
}
```

$$n-1$$

$$f(n) = n - 1$$

$$f(n) = \Theta(n)$$

Algoritmos iterativos

```
void MaxMin1(int *A, int n, int *Max, int
*Min){
    int i;

    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        if (A[i] < *Min) *Min = A[i];
    }
}
```

Algoritmos iterativos

```
void MaxMin1(int *A, int n, int *Max, int
*Min){
    int i;

    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        if (A[i] < *Min) *Min = A[i];
    }
}
```

$2n - 1$

$$f(n) = 2n - 1$$

Algoritmos iterativos

```
void MaxMin1(int *A, int n, int *Max, int
*Min){
    int i;

    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        if (A[i] < *Min) *Min = A[i];
    }
}
```

$$2n - 1$$

$$f(n) = 2n - 1$$

$$f(n) = \Theta(n)$$

Algoritmos iterativos

```
void MaxMin2(int *A, int n, int *Max, int
*Min){
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

Algoritmos iterativos

```
void MaxMin2(int *A, int n, int *Max, int
*Min){
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

- Melhor caso: $f(n) = n - 1$
- Pior caso: $f(n) = 2(n-1)$
- Caso médio: $f(n) = \frac{3}{2}(n-1)$

Algoritmos iterativos

```
void MaxMin2(int *A, int n, int *Max, int
*Min){
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

- Melhor caso: $f(n) = n - 1$ $\quad f(n) = \Omega(n)$
- Pior caso: $f(n) = 2(n - 1)$
- Caso médio: $f(n) = \frac{3}{2}(n - 1)$

Algoritmos iterativos

```
void MaxMin2(int *A, int n, int *Max, int
*Min){
    int i;
    *Max = A[0]; *Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *Max) *Max = A[i];
        else if (A[i] < *Min) *Min = A[i];
    }
}
```

- Melhor caso: $f(n) = n - 1$ $\quad f(n) = \Omega(n)$
- Pior caso: $f(n) = 2(n-1)$ $\quad f(n) = O(n)$
- Caso médio: $f(n) = \frac{3}{2}(n-1)$

Algoritmos iterativos

- Melhor caso: $f(n) = n - 1$ $f(n) = \Omega(n)$
- Pior caso: $f(n) = 2(n-1)$ $f(n) = O(n)$
- Caso médio: $f(n) = \frac{3}{2}(n-1)$

- Pelo teorema visto na aula passada não precisamos nem analisar o caso médio, podemos concluir que:

$$f(n) = \Theta(n)$$

Algoritmos iterativos

```
int Pesquisa(TipoRegistro *A, int n, int chave) {
   int i;
   i = 0;
   while(i < n) {
      if (A[i].chave == chave) {
         break;
      }
      i++;
   }
   return i;
}
```

Algoritmos iterativos

```
int Pesquisa(TipoRegistro *A, int n, int chave) {
    int i;
    i = 0;
    while(i < n) {
        if (A[i].chave == chave) {
            break;
        }
        i++;
    }
    return i;
}
```

- Melhor caso: $f(n) = 1$
- Pior caso: $f(n) = n$
- Caso médio: $f(n) = \frac{(n+1)}{2}$

Algoritmos iterativos

```
int Pesquisa(TipoRegistro *A, int n, int chave) {
    int i;
    i = 0;
    while(i < n) {
        if (A[i].chave == chave) {
            break;
        }
        i++;
    }
    return i;
}
```

- Melhor caso: $f(n) = 1$ $\quad f(n) = \Omega(1)$
- Pior caso: $f(n) = n$
- Caso médio: $f(n) = \frac{(n+1)}{2}$

Algoritmos iterativos

```
int Pesquisa(TipoRegistro *A, int n, int chave) {
   int i;
   i = 0;
   while(i < n) {
      if (A[i].chave == chave) {
         break;
      }
      i++;
   }
   return i;
}
```

- Melhor caso: $f(n) = 1$ $\quad f(n) = \Omega(1)$
- Pior caso: $f(n) = n$ $\quad f(n) = O(n)$
- Caso médio: $f(n) = \frac{(n+1)}{2}$

Algoritmos iterativos

```
int Pesquisa(TipoRegistro *A, int n, int chave) {
   int i;
   i = 0;
   while(i < n) {
      if (A[i].chave == chave) {
         break;
      }
      i++;
   }
   return i;
}
```

- Melhor caso: $f(n) = 1$ $f(n) = \Omega(1)$
- Pior caso: $f(n) = n$ $f(n) = O(n)$
- Caso médio: $f(n) = \frac{(n+1)}{2}$ $f(n) = O(n)$

Algoritmos iterativos

```
int Fatorial(int n) {
    int r = n;
    n--;
    while(n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

Algoritmos iterativos

```
int Fatorial(int n) {
    int r = n;
    n--;
    while(n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

$n-1$

$$f(n) = n - 1$$

$$f(n) = \Theta(n)$$

DCC

Algoritmos recursivos

- Mas e se implementarmos a função fatorial conforme o código abaixo? Como calculamos a complexidade assintótica?

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

Revisão: Algoritmos recursivos

- Primeiramente vamos nos lembrar como é a forma de um algoritmo recursivo. Eles são tipicamente divididos em duas partes.

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

Revisão: Algoritmos recursivos

- A primeira é a **condição de parada** ou **caso base**. Neste caso a função não chama a si mesma, geralmente retorna algum valor.

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

Revisão: Algoritmos recursivos

- A segunda é a **chamada recursiva** ou **caso recursivo**. Este é o caso o qual a função chama a si mesma.

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

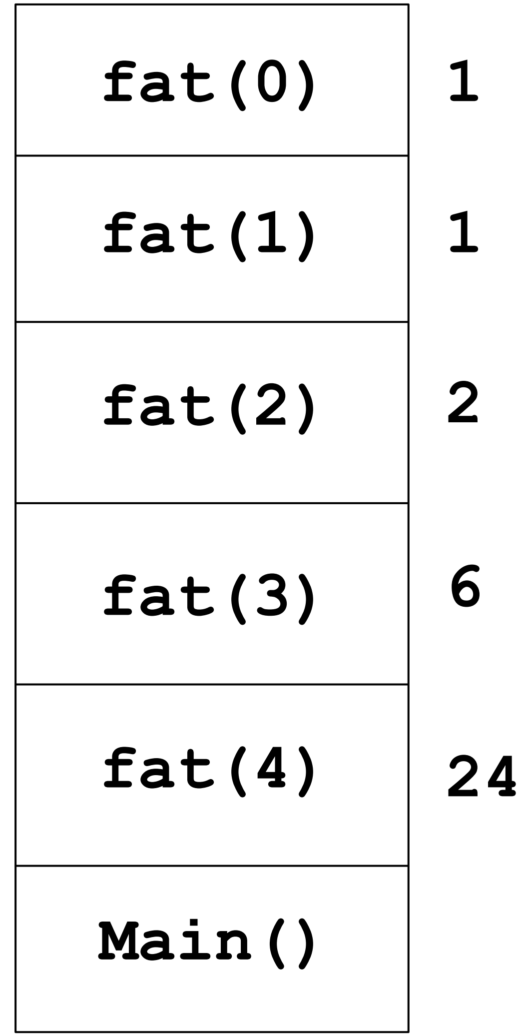
Revisão: Algoritmos recursivos

- É importante lembrar que funções recursivas não necessariamente precisam ser funções que chamam a si próprias.
- Por exemplo podemos ter uma função que *A* chama uma função *B*, e a função *B* chama *A* novamente.
- Também é importante se lembrar que cada chamada recursiva ocupa espaço na memória, criando o que chamamos de **pilha de execução**.

Pilha de execução

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


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

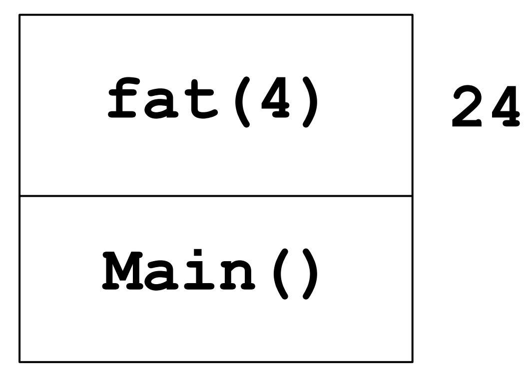


pilha de execução

Pilha de execução

```
int fat (int n) {
    int r = n;
    n--;
    while(n > 0){
        r*= n;
        n--;
    }
}


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



pilha de execução

Relações de recorrência

- Para calcular o tempo de execução de um algoritmo recursivo utilizamos um recurso matemático chamado **relações de recorrência**.
  - Estruturalmente relações de recorrência se assemelham muito aos algoritmos recursivos. Elas também possuem um caso base e um caso recursivo.
- Iremos definir então uma relação de recorrência *T(n)* que descreve o tempo de execução do nosso algoritmo para uma entrada *n*.

Relações de recorrência

- Qual é o caso mais simples do nosso algoritmo fatorial?

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

Relações de recorrência

- Qual é o caso mais simples do nosso algoritmo fatorial?
- Esse caso conseguimos calcular o custo.

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

$$T(n) := \begin{cases} c, & \text{se } n <= 0 \end{cases}$$

Relações de recorrência

- Qual é o caso mais simples do nosso algoritmo fatorial?
- Esse caso conseguimos calcular o custo.
- Considere *c* uma constante positiva.
- Calculamos então o caso base da nossa relação *T(n)*.

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

$$T(n) := \begin{cases} c, & \text{se } n <= 0 \end{cases}$$

Relações de recorrência

- Agora vamos pensar no custo do caso recursivo.

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

Relações de recorrência

- Agora vamos pensar no custo do caso recursivo.
- Existe um custo constante *d* do produto.

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

Relações de recorrência

- Agora vamos pensar no custo do caso recursivo.
- Existe um custo constante *d* do produto.
- Se *T(n)* é o tempo gasto para calcular fatorial de *n*, então o tempo necessário para calcular fatorial de *n-1* é *T(n-1)*.

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

Relações de recorrência

- Agora vamos pensar no custo do caso recursivo.
- Existe um custo constante *d* do produto.
- Se *T(n)* é o tempo gasto para calcular fatorial de *n*, então o tempo necessário para calcular fatorial de *n-1* é *T(n-1)*.
- Concluindo então o nosso caso recursivo.

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

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

Relações de recorrência

- Como calcular a ordem de complexidade de uma relação de recorrência?

Relações de recorrência

- Como calcular a ordem de complexidade de uma relação de recorrência?
- A ideia gira em torno de expandir os termos do caso recursivo e concluir algo a respeito disso.

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

Relações de recorrência

- Como calcular a ordem de complexidade de uma relação de recorrência?
- A ideia gira em torno de expandir os termos do caso recursivo e concluir algo a respeito disso.

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

$$T(n) = d + T(n-1)$$
$$T(n-1) = d + T(n-2)$$
$$T(n-2) = d + T(n-3)$$
$$\cdots$$
$$T(1) = d + T(0)$$

Relações de recorrência

- Substituindo estes termos, vamos reescrever *T(n)*.

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

$$T(n) = d + T(n-1)$$
$$T(n-1) = d + T(n-2)$$
$$T(n-2) = d + T(n-3)$$
$$\ldots$$
$$T(1) = d + T(0)$$

Relações de recorrência

- Substituindo estes termos, vamos reescrever *T(n)*.

$$T(n) = d + d + d + \cdots + d + c$$

$$
\begin{aligned}
T(n) &= d + T(n-1)\\
T(n-1) &= d + T(n-2)\\
T(n-2) &= d + T(n-3)\\
&\ldots\\
T(1) &= d + T(0)
\end{aligned}
$$

Relações de recorrência

- Substituindo estes termos, vamos reescrever *T(n)*.

$$T(n) = \underbrace{d + d + d + \cdots + d}_{n \text{ vezes}} + c$$

$$\begin{aligned}
T(n) &= d + T(n-1) \\
T(n-1) &= d + T(n-2) \\
T(n-2) &= d + T(n-3) \\
&\dots \\
T(1) &= d + T(0)
\end{aligned}$$

Relações de recorrência

- Substituindo estes termos, vamos reescrever *T(n)*.

$$T(n) = \underbrace{d + d + d + \cdots + d}_{n \text{ vezes}} + c$$

$$T(n) = dn + c$$

$$T(n) = d + T(n-1)$$
$$T(n-1) = d + T(n-2)$$
$$T(n-2) = d + T(n-3)$$
$$\ldots$$
$$T(1) = d + T(0)$$

Relações de recorrência

- Substituindo estes termos, vamos reescrever *T(n)*.

$$T(n) = \underbrace{d + d + d + \cdots + d}_{n \text{ vezes}} + c$$

$$T(n) = dn + c$$

$$T(n) = O(n)$$

$$T(n) = d + T(n-1)$$
$$T(n-1) = d + T(n-2)$$
$$T(n-2) = d + T(n-3)$$
$$\ldots$$
$$T(1) = d + T(0)$$

Relações de recorrência

Vamos analisar agora o seguinte algoritmo visto na aula passada:

- Recebe como entrada um vetor com *n* inteiros.
  - Imprima o primeiro elemento do vetor.
  - Se n > 1, divida o vetor em dois vetores de tamanho n/2 e chame a função recursivamente para segunda metade.

Relações de recorrência

- Nosso caso base acontece quando o tamanho do vetor é menor ou igual a 1.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \end{cases}$$

Relações de recorrência

- Nosso caso base acontece quando o tamanho do vetor é menor ou igual a 1.
- O caso recursivo divide o vetor na metade, mas faz a chamada para apenas uma das metades!

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

Relações de recorrência

- Note que agora é um pouco mais complicado utilizar o mesmo procedimento que fizemos com fatorial.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

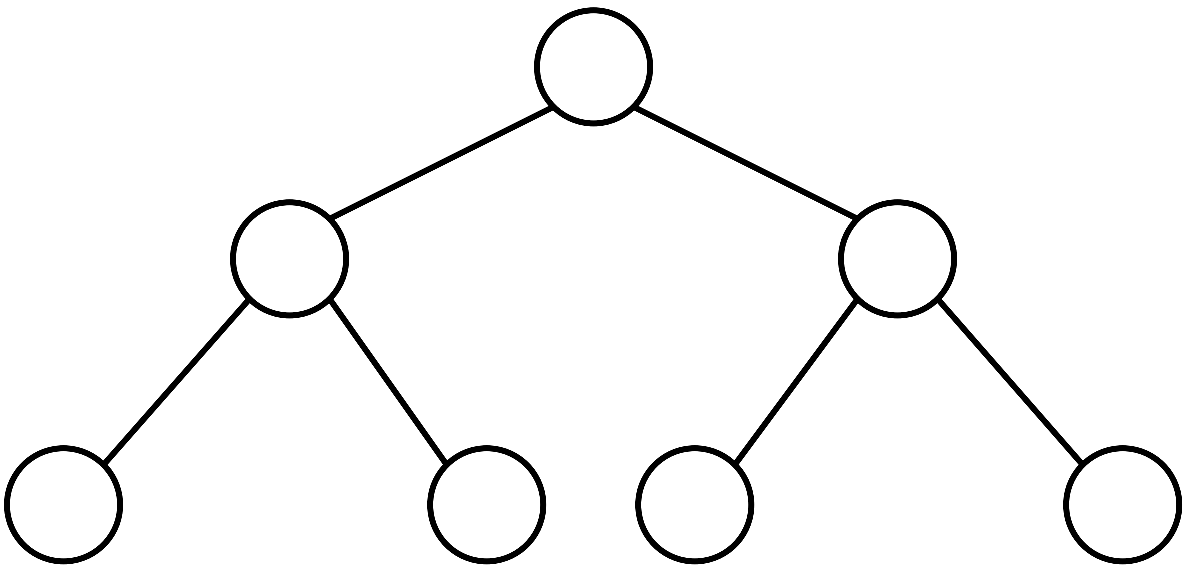
Relações de recorrência

- Note que agora é um pouco mais complicado utilizar o mesmo procedimento que fizemos com fatorial.
- Iremos utilizar uma outra abstração para fazer este cálculo, chamada **árvore de recursão**.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

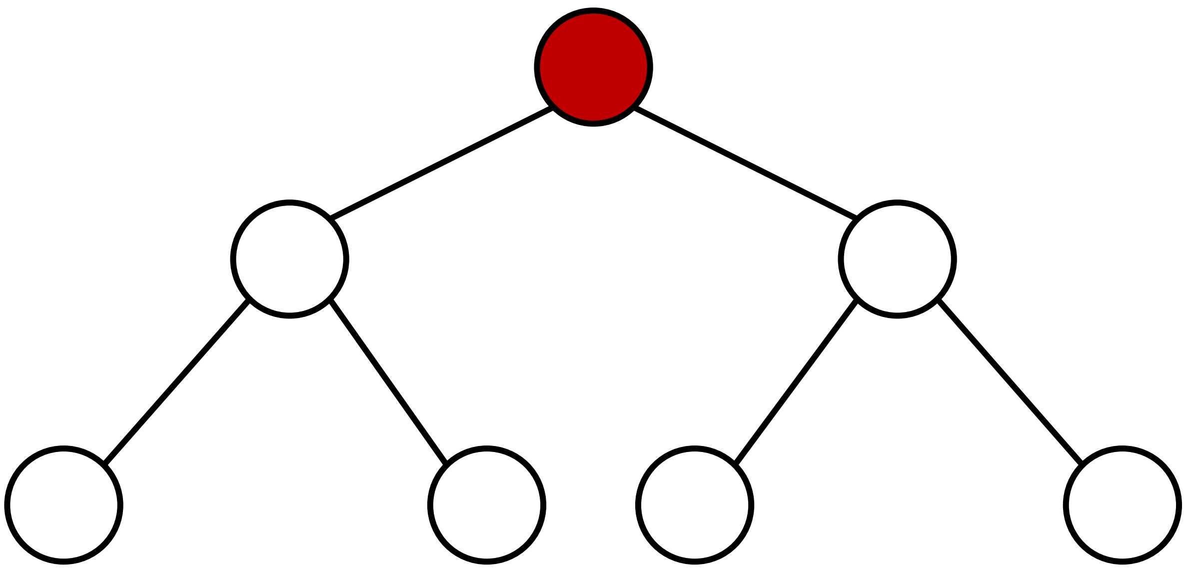
Relações de recorrência

- Nessa abstração cada nó representa uma chamada da nossa função



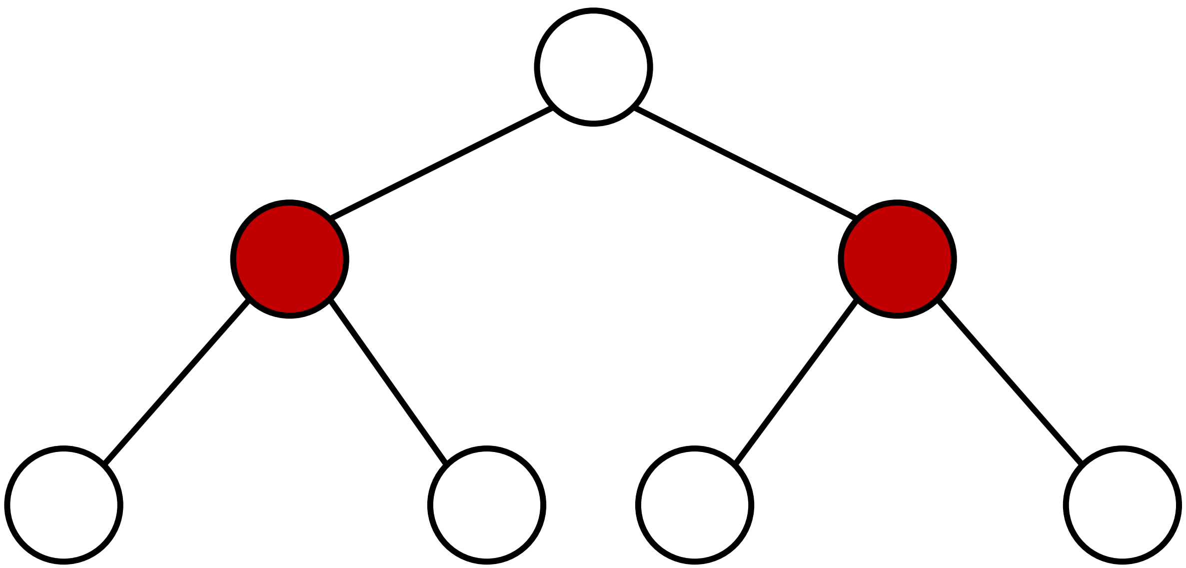
Relações de recorrência

- A primeira é nossa chamada original, com uma instância de tamanho *n*.



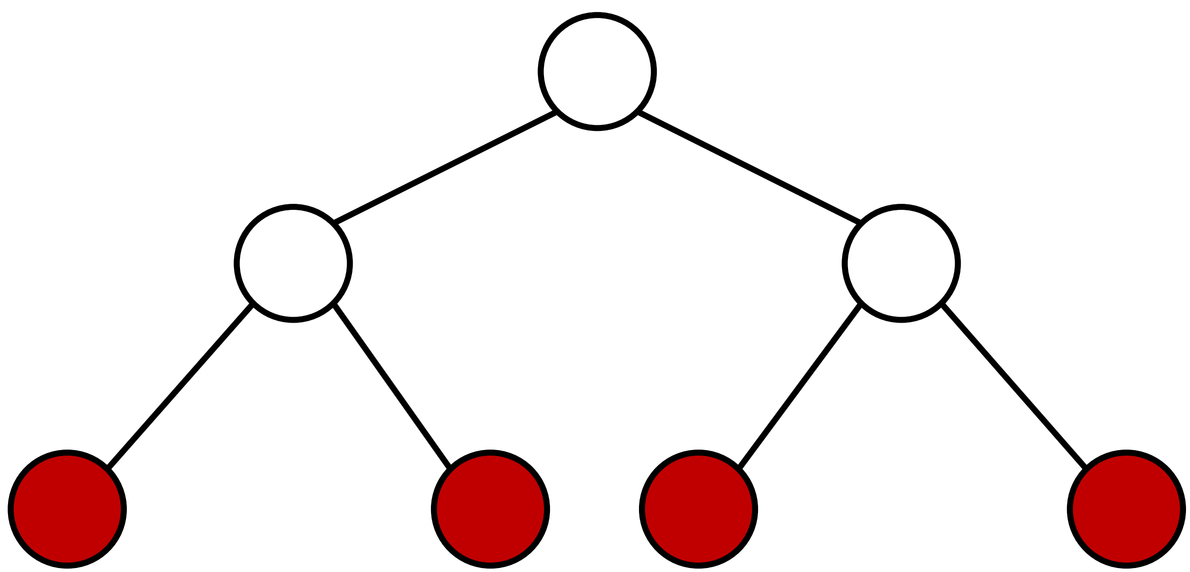
Relações de recorrência

- Como nosso algoritmo divide o vetor pela metade, a próxima chamada tem tamanho *n/2*.



Relações de recorrência

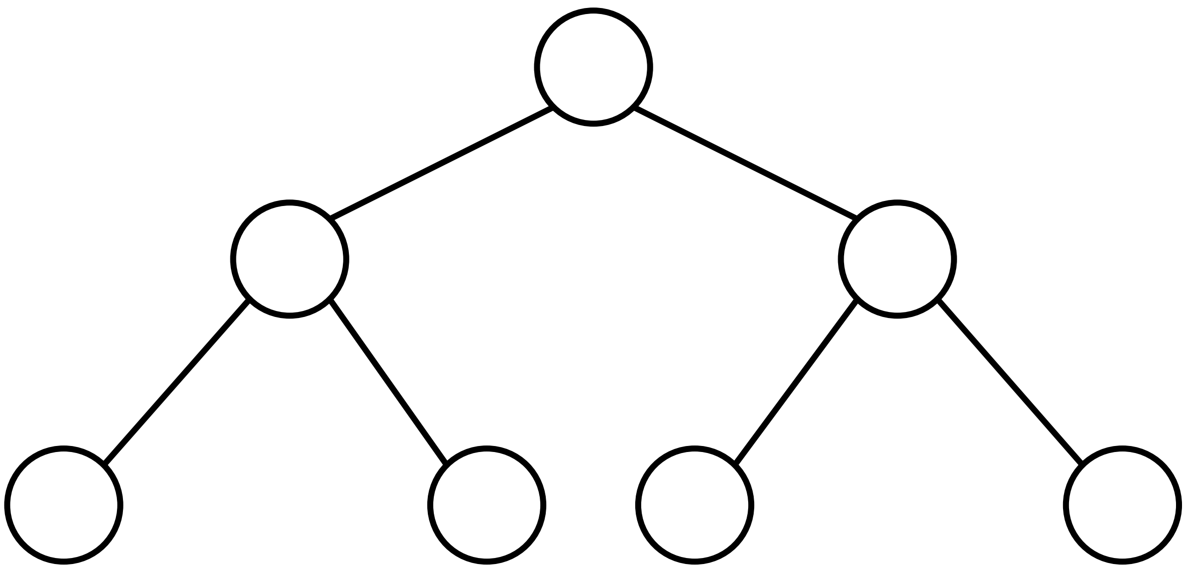
- A chamada seguinte tem tamanho *n/4* e assim por diante até chegarmos ao caso base.



DCC

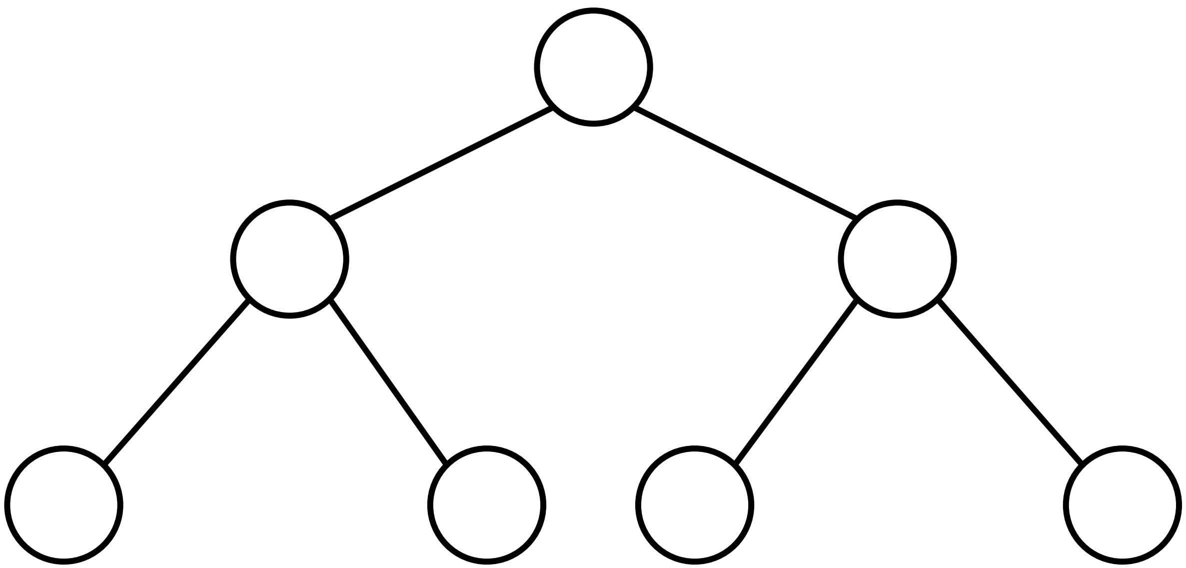
Relações de recorrência

- Note que a quantidade de nós de uma chamada é sempre o dobro da chamada anterior



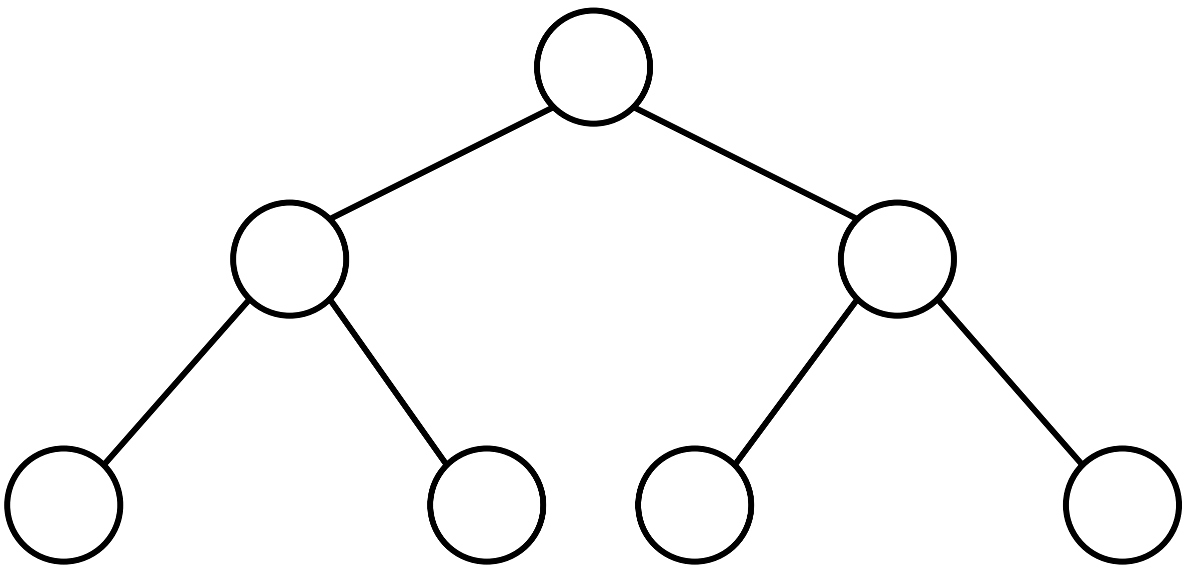
Relações de recorrência

- Então se fizermos *k* chamadas teremos no fim de nossa árvore 2^k nós.



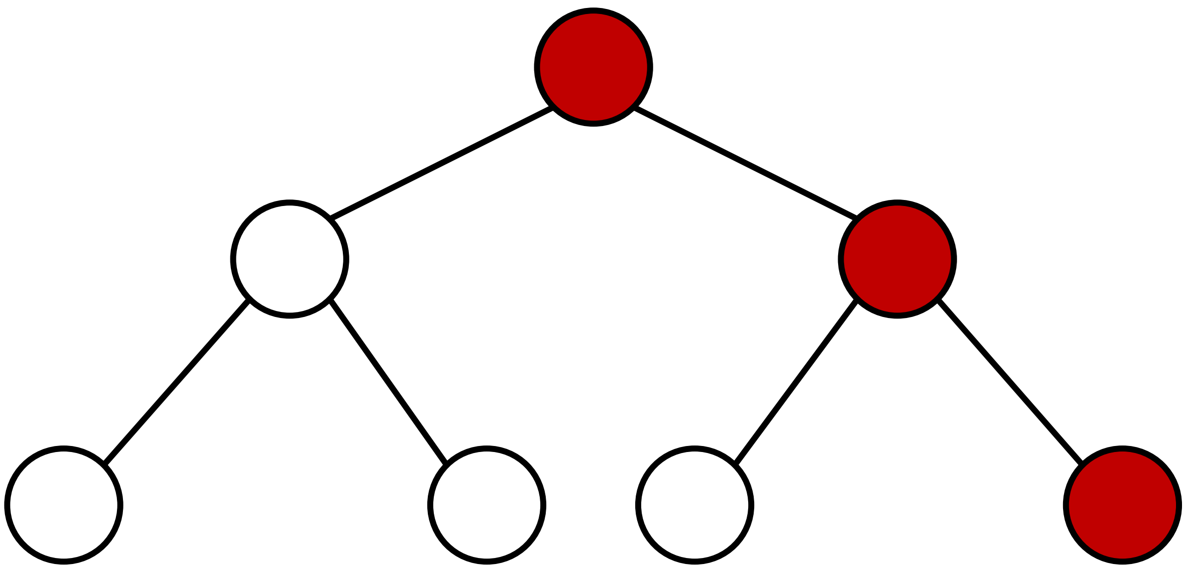
Relações de recorrência

- Note que $log_2 2^k = k$. Essa informação nos será útil em alguns passos.



Relações de recorrência

- Nosso algoritmo faz a chamada recursiva apenas para a segunda metade do vetor, e ele para apenas quando o tamanho do vetor for 1.



Relações de recorrência

- Isso significa que no fim da nossa árvore temos *n* elementos.

Relações de recorrência

- Isso significa que no fim da nossa árvore temos *n* elementos.
- Agora vamos usar $log_2 2^k = k$.

Relações de recorrência

- Isso significa que no fim da nossa árvore temos *n* elementos.
- Agora vamos usar $log_2 2^k = k$.
- Note que *k* é quantas chamadas fizemos.

Relações de recorrência

- Isso significa que no fim da nossa árvore temos n elementos.
- Agora vamos usar $log_2 2^k = k$.
- Note que k é quantas chamadas fizemos
- Se $2^k = n$, então concluímos que fazemos no máximo $log_2 n$ chamadas.

Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

$$T(n) = d + d + d + \cdots + d + c$$

Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

$$T(n) = \underbrace{d + d + d + \cdots + d}_{\textit{log } n \text{ vezes}} + c$$

Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ d + T(\frac{n}{2}), & \text{se } n > 0 \end{cases}$$

$$T(n) = \underbrace{d + d + d + \cdots + d}_{log\, n \text{ vezes}} + c$$

$$T(n) = d(logn) + c$$

$$T(n) = O(logn)$$

Classes de comportamento assintótico

Vamos repetir o processo para este outro algoritmo

- Recebe como entrada um vetor com *n* inteiros.
  - Imprima o maior elemento do vetor.
  - Se n > 1, divida o vetor em dois vetores de tamanho n/2 e chame a função recursivamente para as duas metades.

Relações de recorrência

- O caso base é similar a do algoritmo anterior.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \end{cases}$$

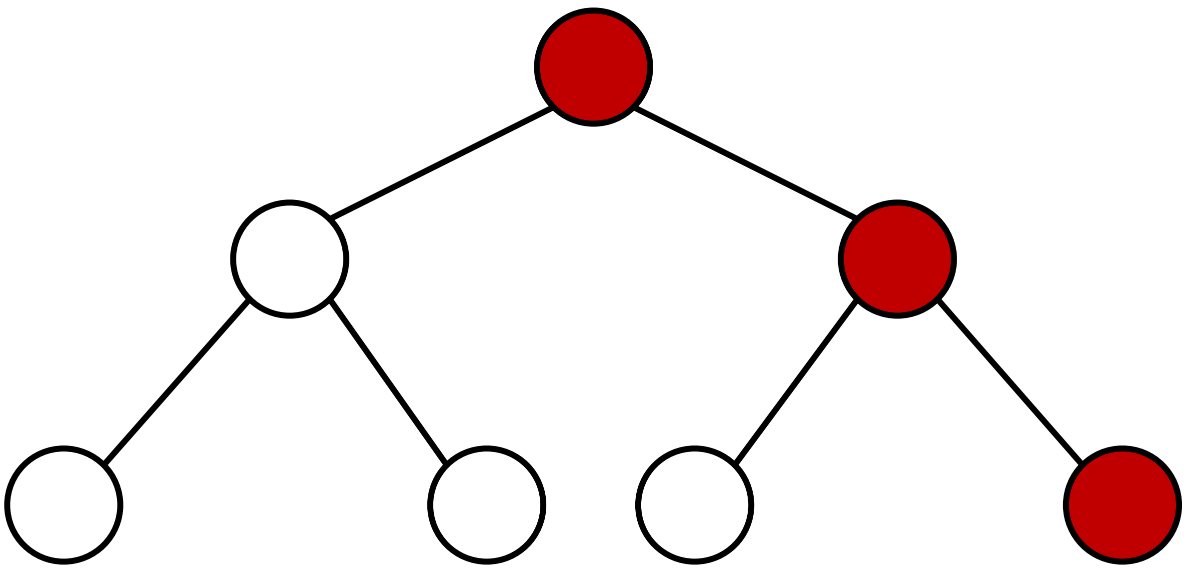
Relações de recorrência

- O caso base é similar a do algoritmo anterior.
- No entanto no caso recursivo devemos encontrar o maior elemento do vetor e fazer a chamada recursiva para **as duas** metades do vetor.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ n + 2T(\frac{n}{2}), & \text{se } n > 1 \end{cases}$$

Relações de recorrência

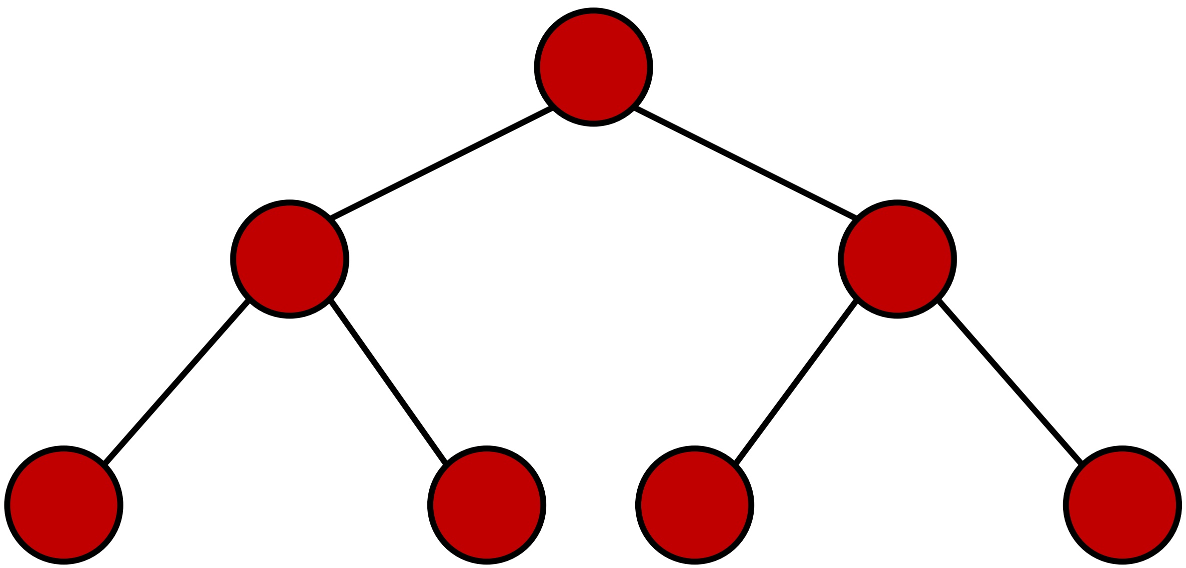
- O algoritmo anterior fazia operações de custo constante em apenas uma das metades.



DCC

Relações de recorrência

- Este algoritmo faz operações de linear nas duas metades.



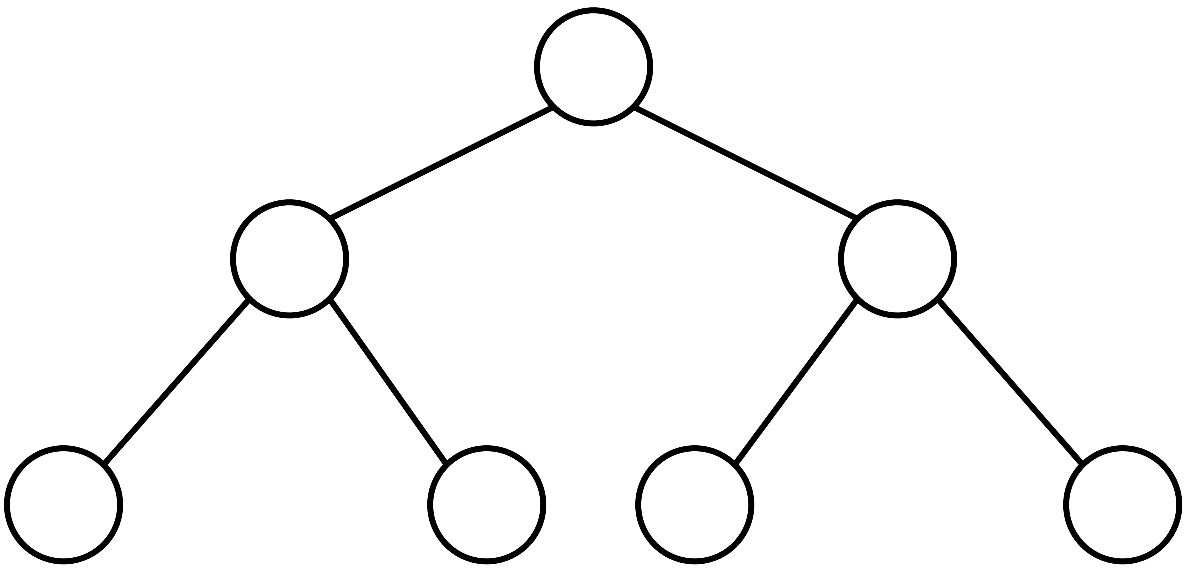
Relações de recorrência

- Nós sabemos que a quantidade de chamadas é limitada por $log_2 n$.
- O que devemos observar é como se comporta o termo não recursivo da nossa relação.

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$
$$T\left(\frac{n}{2}\right) = \frac{n}{2} + 2T\left(\frac{n}{4}\right)$$
$$T\left(\frac{n}{4}\right) = \frac{n}{4} + 2T\left(\frac{n}{8}\right)$$
$$\cdots$$

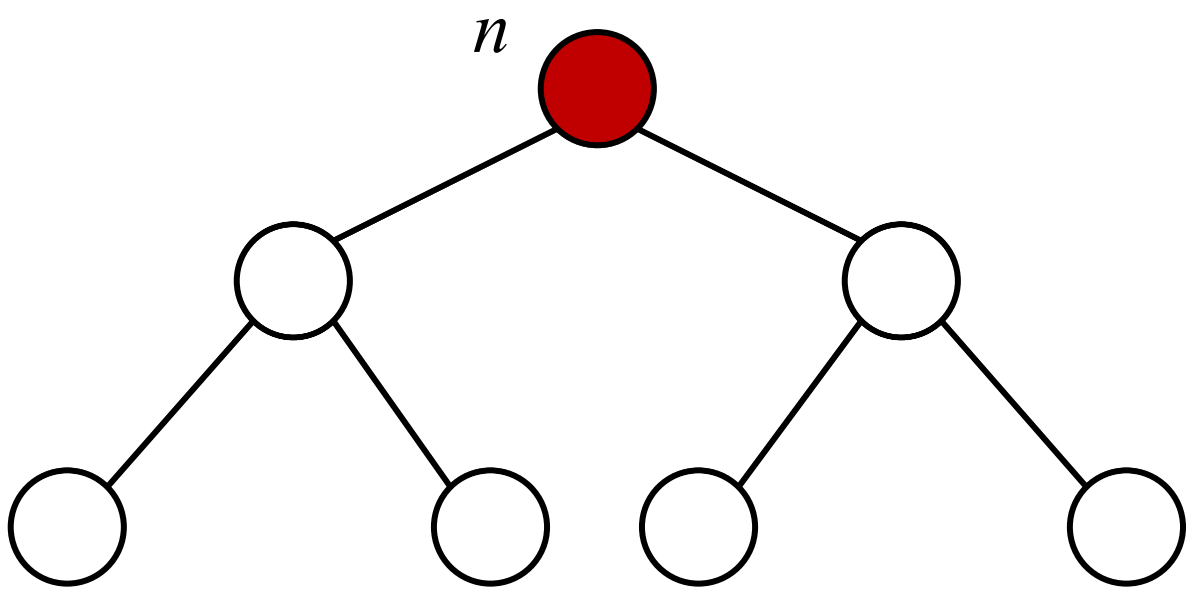
Relações de recorrência

- Se somarmos os de todas as "metades" vamos perceber que para cada chamada o custo total são de *n* operações.



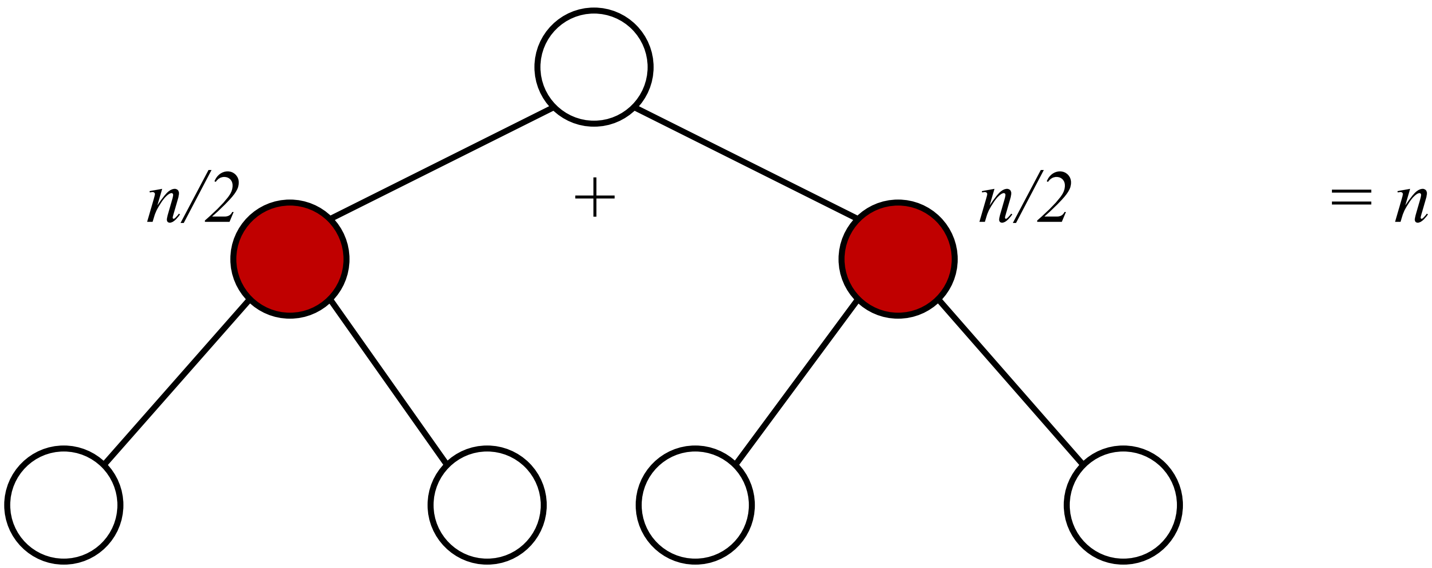
Relações de recorrência

- Se somarmos os de todas as "metades" vamos perceber que para cada chamada o custo total são de *n* operações.



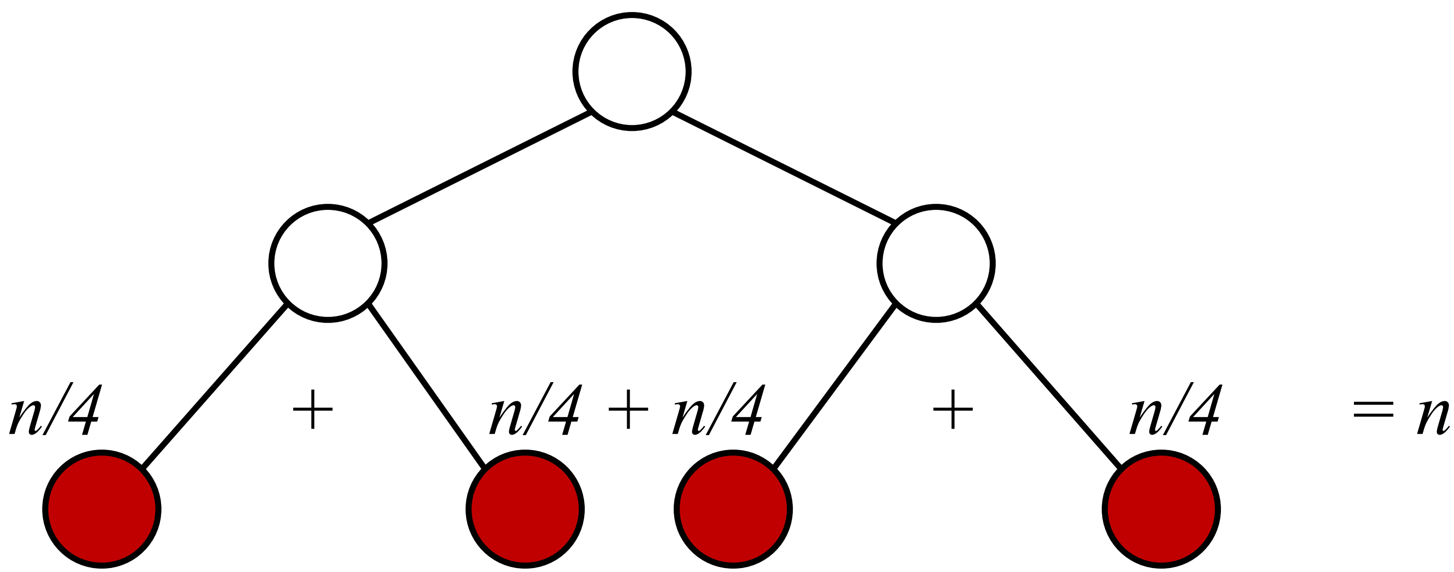
Relações de recorrência

- Se somarmos os de todas as “metades” vamos perceber que para cada chamada o custo total são de *n* operações.



Relações de recorrência

- Se somarmos os de todas as “metades” vamos perceber que para cada chamada o custo total são de *n* operações.



Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ n + 2T(\frac{n}{2}), & \text{se } n > 1 \end{cases}$$

$$T(n) = n + n + n + \cdots + n + c$$

DCC

Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ n + 2T(\frac{n}{2}), & \text{se } n > 1 \end{cases}$$

$$T(n) = \underbrace{n + n + n + \cdots + n}_{\log n \text{ vezes}} + c$$

Relações de recorrência

- Agora conseguimos finalizar a expansão dos termos da relação de recorrência.

$$T(n) := \begin{cases} c, & \text{se } n <= 1 \\ n + 2T(\frac{n}{2}), & \text{se } n > 1 \end{cases}$$

$$T(n) = \underbrace{n + n + n + \cdots + n}_{\textit{log n} \text{ vezes}} + c$$

$$T(n) = n(logn) + c$$

$$T(n) = O(nlogn)$$

Relações de recorrência

- Estes foram exemplos simples de relações de recorrência as quais o tamanho da entrada é dividido por um fator.
- As abstrações se tornam cada vez mais complexas, tornando o processo de determinar a complexidade assintótica de relações desse tipo bem longo e exaustivo.
- Na próxima aula veremos uma maneira mais simples porém menos intuitiva de resolver recorrências parecidas com os últimos dois exemplos.

Relações de recorrência

- Por fim, vamos ver o exemplo de calcular o *n*-ésimo termo da sequencia de fibonacci de forma recursiva.

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

Relações de recorrência

- O primeiro passo é determinar a relação de recorrência do algoritmo.

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

Relações de recorrência

- O primeiro passo é determinar a relação de recorrência do algoritmo.

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

$$T(n) := \begin{cases} c, & \text{se } n < 3 \\ T(n-1) + T(n-2), & \text{se } n \geq 3 \end{cases}$$

Relações de recorrência

- A observação principal neste caso é a seguinte desigualdade:

$$T(n-2) < T(n-1)$$

Relações de recorrência

- A observação principal neste caso é a seguinte desigualdade:

$$T(n-2) < T(n-1)$$

- Agora podemos limitar superiormente *T(n)* por termos mais simples de calcular

$$T(n) = T(n-1) + T(n-2) < 2T(n-1)$$

Relações de recorrência

- A observação principal neste caso é a seguinte desigualdade:

$$T(n-2) < T(n-1)$$

- Agora podemos limitar superiormente *T(n)* por termos mais simples de calcular

$$T(n) = T(n-1) + T(n-2) < 2T(n-1)$$
$$T(n) < 2T(n-1)$$

Relações de recorrência

- Agora basta utilizar o mesmo raciocínio que fizemos nos outros algoritmos.

Relações de recorrência

- Agora basta utilizar o mesmo raciocínio que fizemos nos outros algoritmos.

$$T(n) < 2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 \cdot c$$

Relações de recorrência

- Agora basta utilizar o mesmo raciocínio que fizemos nos outros algoritmos.

$$T(n) < \underbrace{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}_{n \text{ vezes}} \cdot c$$

Relações de recorrência

- Agora basta utilizar o mesmo raciocínio que fizemos nos outros algoritmos.

$$T(n) < \underbrace{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}_{n \text{ vezes}} \cdot c$$

$$T(n) < c2^n$$

$$T(n) = O(2^n)$$

Relações de recorrência

- Neste caso, como usamos um limite superior mais "folgado" no nosso cálculo, o resultado não é o mais apertado o possível.
- Utilizando cálculos um pouco mais complicados (determinar as raízes do polinômio característico) conseguimos obter um limite apertado para esta relação:

$$T(n) = O(\varphi^n)$$

Fibonacci recursivo X iterativo

- Vamos comparar a performance com uma maneira iterativa de resolver o problema:

```
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;
}
```

Fibonacci recursivo X iterativo

- Vamos comparar a performance com uma maneira iterativa de resolver o problema:

```
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;
}
```

$\theta(n)$

Fibonacci recursivo X iterativo

- A diferença de performance é significativa.
- Existe uma grande repetição de subproblemas que a implementação recursiva não trata.
- Enquanto isso a implementação iterativa não repete nenhum subproblema.
- É possível calcular o *n*-ésimo termo em tempo logarítmico utilizando exponenciação rápida de matrizes, mas este algoritmo está fora do nosso escopo.

Algoritmos recursivos X iterativos

- Qual é a melhor alternativa? O código recursivo ou o código iterativo?

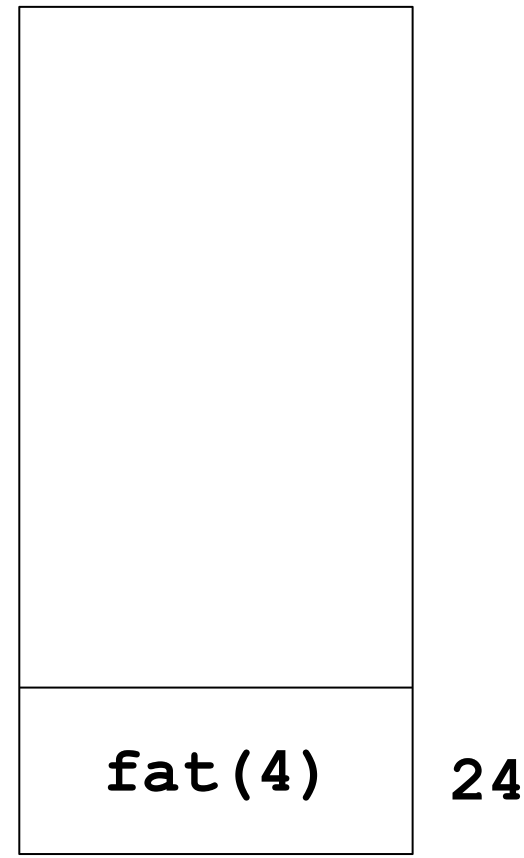
- Vamos olhar o que acontece com a complexidade de espaço...

Exemplo de execução

Versão Iterativa:

```
int fat (int n) {
    int r = n;
    n--;
    while(n > 0){
        r*= n;
        n--;
    }
}


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



pilha de execução

Exemplo de execução

Versão Recursiva:

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


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

fat(0)	1
fat(1)	1
fat(2)	2
fat(3)	6
fat(4)	24

pilha de execução

Análise de Algoritmos Recursivos

- Para a abordagem recursiva complexidade de espaço é O(n), devido a pilha de execução
- Já na abordagem iterativa complexidade de espaço é O(1)
- Novamente, vemos que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos