Algoritmos e Estrutura de Dados II (AE23CP-3CP)

Aula #05 - Paradigmas de Projeto de Algoritmos
- Divisão e Conquista
- Programação Dinâmica

Profª Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Paradigmas de Projeto de Algoritmos

- Divisão e Conquista

Idéias básicas da estratégia de divisão e conquista

- **Divisão:**

  Divida o problema em duas ou mais partes, criando subproblemas menores.

- **Conquista:**

  Os subproblemas são resolvidos recursivamente usando divisão e conquista. Caso os subproblemas sejam suficientemente pequenos resolva-os de forma direta.

- **Combina:**

  Tome cada uma das partes e junte-as todas de forma a resolver o problema original.

Divisão e Conquista

Algoritmo genérico

Algoritmo 1 *Divisao e Conquista*(x)

if x é pequeno ou simples **then**
    return resolver(x)
else
    decompor x em conjuntos menores $x_0, x_1, \ldots x_n$
    for $(i \leftarrow 0$ até $n)$ **do**
        $y_i \leftarrow Divisao\ e\ Conquista(x_i)$
        $i \leftarrow i + 1$
    end for
    combinar y_i's
end if
return y

Exemplo 1 - Maior Valor

O problema consiste em encontrar o maior elemento de um array $A[1..n]$

Solução Ingênua

Algoritmo 2 $Maxim(A[1..n])$

```
max ← A[1]
for i ← 2 até n do
    if A[i] > max then
        max ← A[i]
    end if
end for
return max
```

Análise:

Pior Caso e caso médio: $O(n)$

Exemplo 1 - Maior Valor

Solução Divisão e Conquista

Algoritmo 3 $Maxim(A[1..n])$

if $y - x \leq 1$ **then**
  return $max(A[x], A[y])$
else
  $m \leftarrow (x + y)/2$
  $v1 \leftarrow Maxim(A[x..m])$
  $v2 \leftarrow Maxim(A[m + 1..y])$
end if
return $max(v1, v2)$

Análise:

Pior Caso: $O(n)$

Exemplo 2 - Potenciação

Solução Ingênua

Algoritmo 4 $Pot(a, n)$

```
p ← a
for i ← 2 até n do
    p ← p × a
end for
return p
```

Análise:

Pior Caso: $O(n)$

Solução Divisão e Conquista

Algoritmo 5 $Pot(a, n)$

```
if n = 0 then
    return 1
else if n = 1 then
    return a
else
    x ← Pot(a, ⌊n/2⌋)
    if n é par then
        return x ∗ x
    else
        return a ∗ x ∗ x
    end if
end if
```

Análise:

Pior Caso: $O(logn)$

Exemplo 3 - Mergesort

Caso o tamanho do vetor seja maior que 1

1. **Divisão:** divida o vetor ao meio
2. **Conquista:** ordene a primeira metade recursivamente
3. **Conquista:** ordene a segunda metade recursivamente
4. **Combinação:** intercale as duas metades

Senão

- devolva o elemento

```
function MergSort(A, p, r))
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergSort(A, p, q)
        MergSort(A, q + 1, r)
        Intercala(A, p, q, r)
    end if
end function
```

Exemplo 3 - Complexidade do Mergesort

Relembrando: o objetivo é rearranjar $A[p..r]$, com $p \leq r$, em ordem crescente.

```
1: function MergSort(A, p, r))
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MergSort(A, p, q)
5:         MergSort(A, q + 1, r)
6:         Intercala(A, p, q, r)
7:     end if
8: end function
```

p				q				r
66	33	55	44	99	11	77	22	88

Exemplo 3 - Complexidade do Mergesort

Relembrando: o objetivo é rearranjar $A[p..r]$, com $p \leq r$, em ordem crescente.

function *MergSort*$(A, p, r))$
  if $p < r$ **then**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MergSort(A, p, q)
    MergSort$(A, q+1, r)$
    Intercala(A, p, q, r)
  end if
end function

p				q				r
33	44	55	66	99	11	77	22	88

Exemplo 3 - Complexidade do Mergesort

Relembrando: o objetivo é rearranjar $A[p..r]$, com $p \leq r$, em ordem crescente.

```
function MergSort(A, p, r))
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergSort(A, p, q)
        MergSort(A, q + 1, r)
        Intercala(A, p, q, r)
    end if
end function
```

p				q				r
33	44	55	66	99	11	22	77	88

Exemplo 3 - Complexidade do Mergesort

Relembrando: o objetivo é rearranjar $A[p..r]$, com $p \leq r$, em ordem crescente.

```
function MergSort(A, p, r))
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergSort(A, p, q)
        MergSort(A, q + 1, r)
        Intercala(A, p, q, r)
    end if
end function
```

p				q				r
11	22	33	44	55	66	77	88	99

Exemplo 3 - Complexidade do Mergesort

```
function MergSort(A, p, r))
    if p < r then
        q ← ⌊(p + r)/2⌋
        MergSort(A, p, q)
        MergSort(A, q + 1, r)
        Intercala(A, p, q, r)
    end if
end function
```

Qual é a complexidade do MergeSort?

Seja $T(n) :=$ o consumo de tempo máximo (pior caso) em função de $n = r - p + 1$

Exemplo 3 - Complexidade do Mergesort

```
1: function MergSort(A, p, r))
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MergSort(A, p, q)
5:         MergSort(A, q + 1, r)
6:         Intercala(A, p, q, r)
7:     end if
8: end function
```

linha	consumo de tempo
2	?
3	?
4	?
5	?
6	?

$$T(n) = ?$$

Exemplo 3 - Complexidade do Mergesort

```
1: function MergSort(A, p, r))
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MergSort(A, p, q)
5:         MergSort(A, q + 1, r)
6:         Intercala(A, p, q, r)
7:     end if
8: end function
```

linha	consumo de tempo
2	$\Theta(1)$
3	$\Theta(1)$
4	$T(\lceil n/2 \rceil)$
5	$T(\lfloor n/2 \rfloor)$
6	$\Theta(n)$

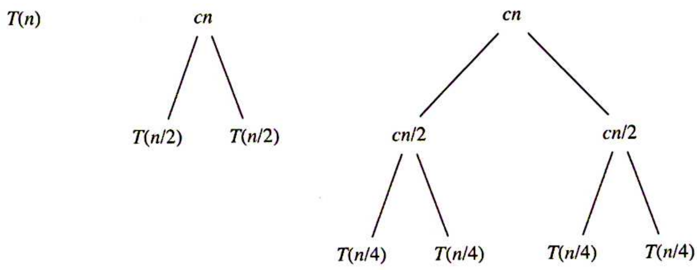
$$T(n) = T(\lceil n/2 \rceil) + \Theta(n) + \Theta(2)$$

Exemplo 3 - Complexidade do Mergesort

Ao aplicar o paradigma da **divisão-e-conquista**, chega-se a um algoritmo recursivo, cuja complexidade $T(n)$ é uma **fórmula de recorrência** (i.e., uma fórmula definida em termos de si mesma):

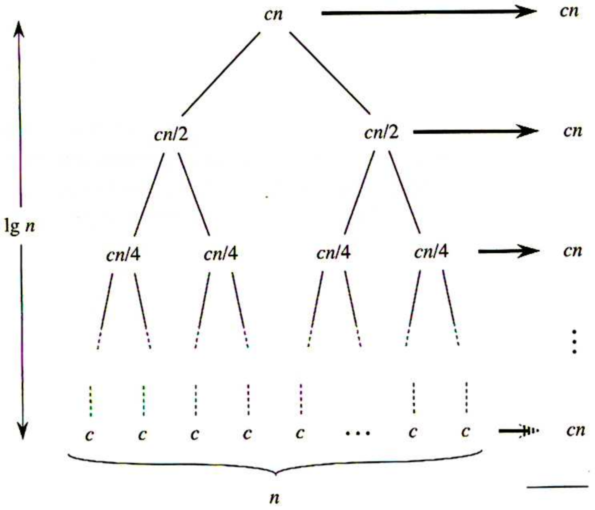
$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \text{ para n } = 2, 3, 4, \ldots$$

A árvore de recursão da recorrência resulta $T(n) = 2T(n/2) + cn$



Exemplo 3 - Complexidade do Mergesort

Custo total: $cnlog\ n + cn = \Theta(n\ log\ n)$



Conclusão

- Serve para resolver problemas nos quais subproblemas são versões menores do problema original
  - Isso leva a soluções eficientes e elegantes, em especial quando é utilizado recursivamente.
- Procurar sempre manter o **balenceamento** na subdivisão de um problema em partes menores.
- Devemos nos preocupar com a questão de eficiência tanto de espaço quanto de tempo.
- Devemos evitar o uso de recursividade quando existe uma solução óbvia por iteração (Ex.: sequência de Fibonacci)
- Desvantagens:
  - Dificuldade para encontrar erros;
  - Podem ser ineficientes.

Paradigmas de Projeto de Algoritmos

- Programação Dinâmica

Idéias Básicas

Construir por etapas uma resposta ótima combinando respostas já obtidas para partes menores.

- Inicialmente, a entrada é decomposta em partes mínimas, para as quais são obtidas respostas.
- Em cada passo, sub-resultados são combinados dando respostas para partes maiores, até que se obtenha uma resposta para o problema original.
- A decomposição é feita uma única vez e, além disso, os casos menores são tratados antes dos maiores.

Assim, esse método é chamado ascendente, ao contrário dos métodos recursivos, que são chamados descendentes.

Exemplo 1: Número de Fibonacci

Entrada: Um número inteiro n.
Saída: O número de Fibonacci F_n, definido da seguinte forma:

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \text{ para } n \geq 2.$$

Solução clássica utiliza recursão

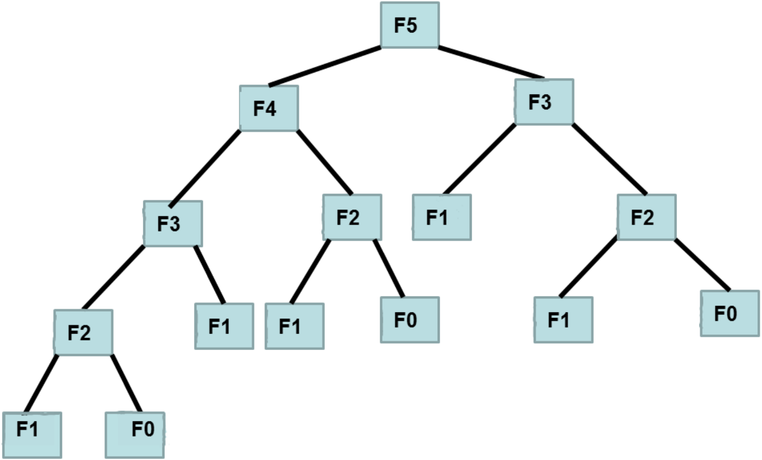
Algoritmo 6 $Fib(n)$

```
if n ≤ 1 then
    return n
else
    return Fib(n − 1) + Fib(n − 2)
end if
```

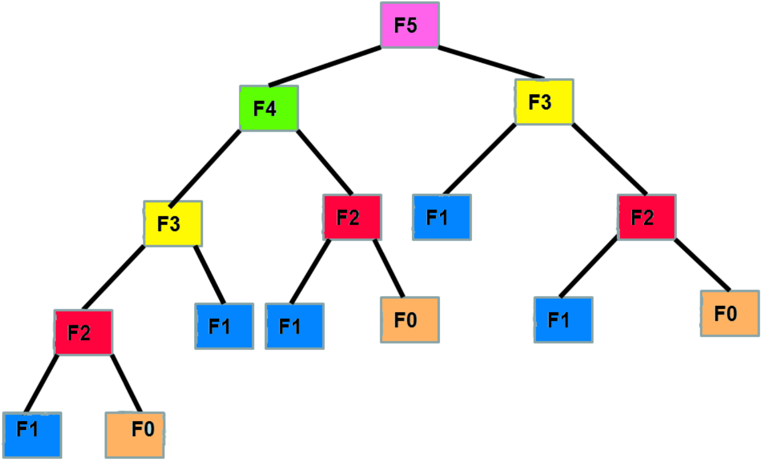
Análise: a resolução da recorrência nos dá:

- $T(n) = T(n-1) + T(n-2) + c \Rightarrow O(2^n)$

Conclusão: Solução ineficiente (solução exponencial, intratável)


F5
F4
F3
F3
F2
F1
F2
F2
F1
F1
F1
F0
F1
F0
F1
F0

Solução Apresentada



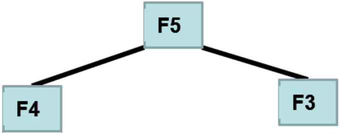
Solução alternativa com programação dinâmica

- Utilizar um array $f[0, ..., n]$ para guardar os valores calculados.
- Inicialmente, f contém apenas símbolos especiais ∞.

Algoritmo 7 $Fib1(n)$

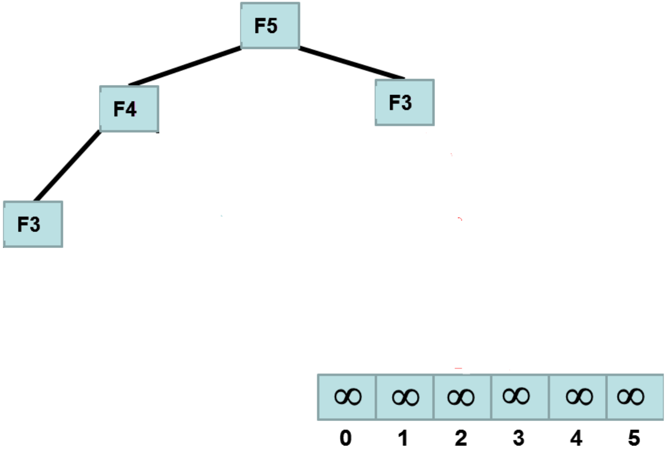
if $f[n] \neq \infty$ **then**
  return $f[n]$
end if
if $n \leq 1$ **then**
  return $f[n] \leftarrow n$
end if
return $f[n] \leftarrow Fib1(n-1) + Fib1(n-2)$

Solução Apresentada

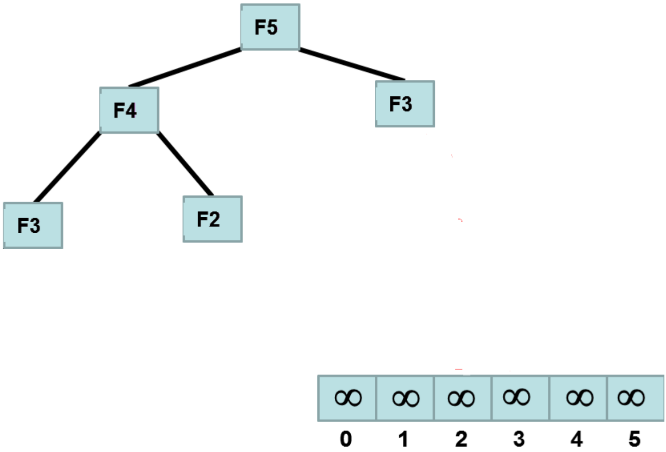


∞	∞	∞	∞	∞	∞
0	1	2	3	4	5

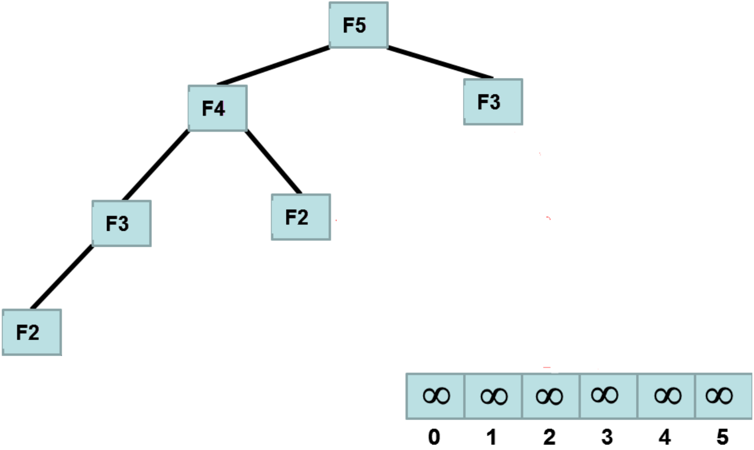
Solução Apresentada



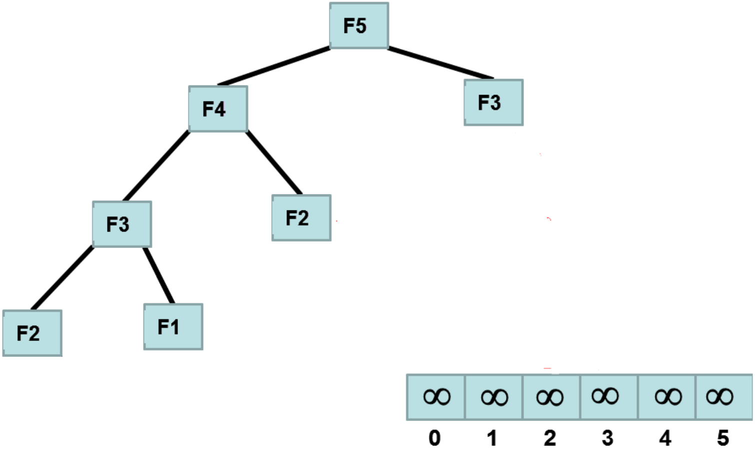
Solução Apresentada



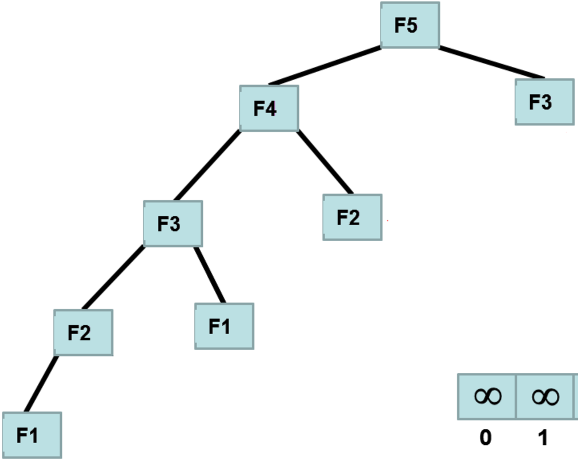
Solução Apresentada



Solução Apresentada

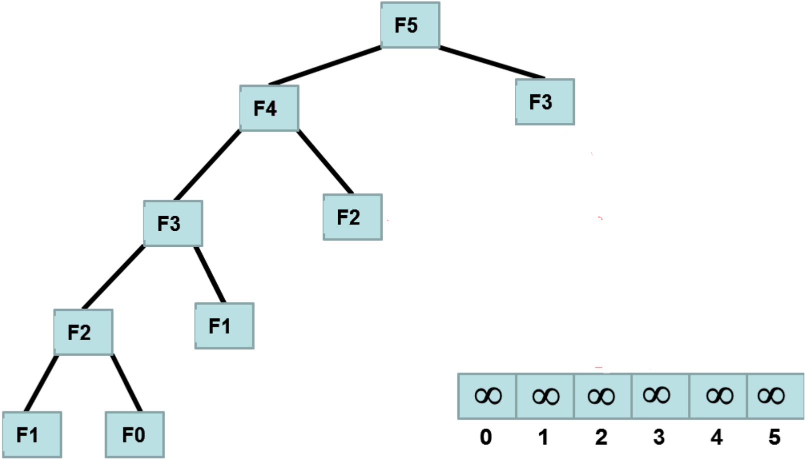


Solução Apresentada

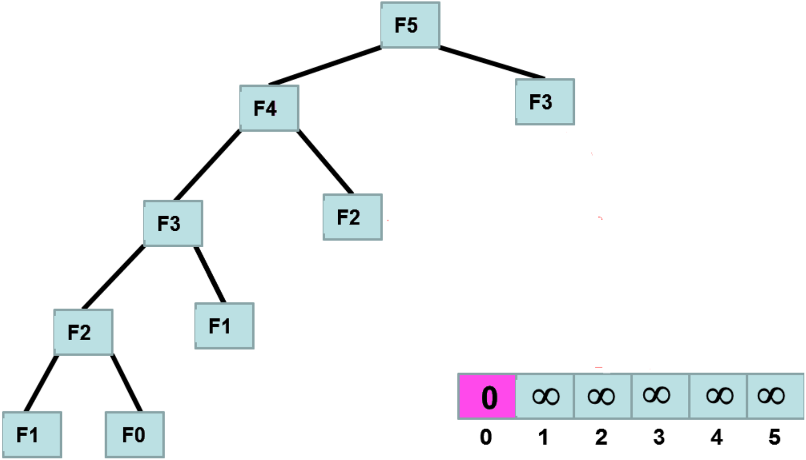


∞	∞	∞	∞	∞	∞
0	1	2	3	4	5

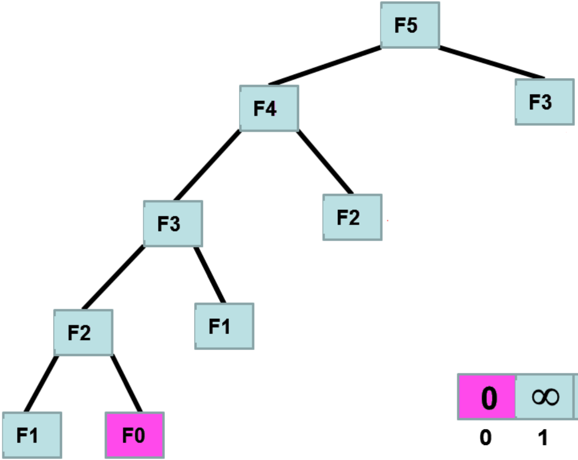
Solução Apresentada



Solução Apresentada

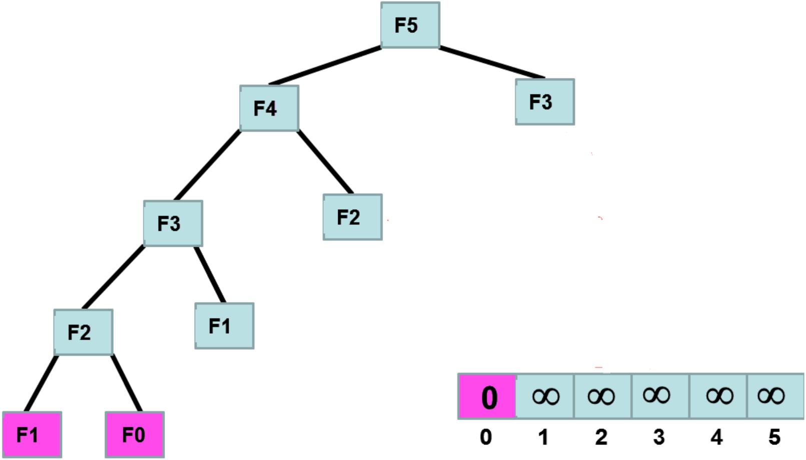


Solução Apresentada

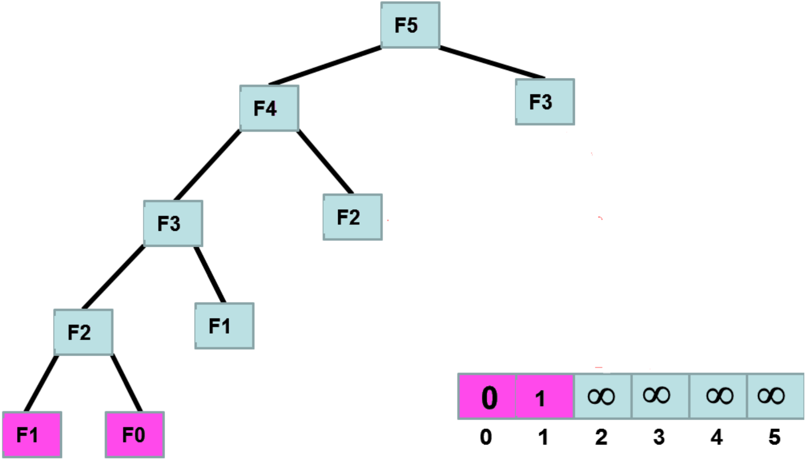


0	1	2	3	4	5
0	∞	∞	∞	∞	∞

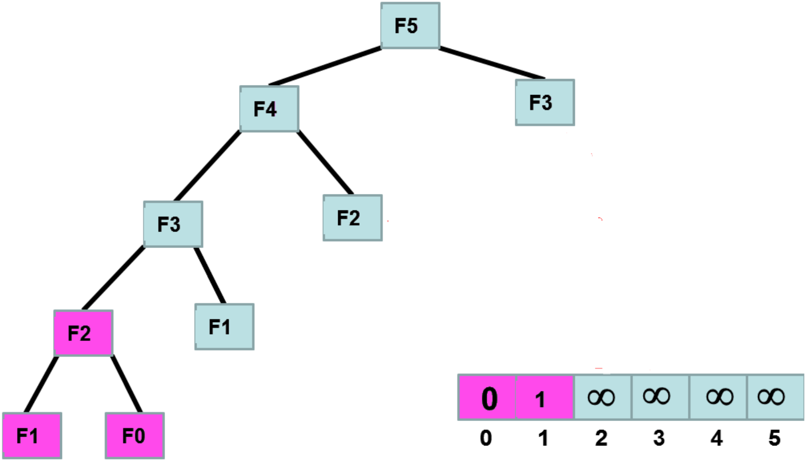
Solução Apresentada



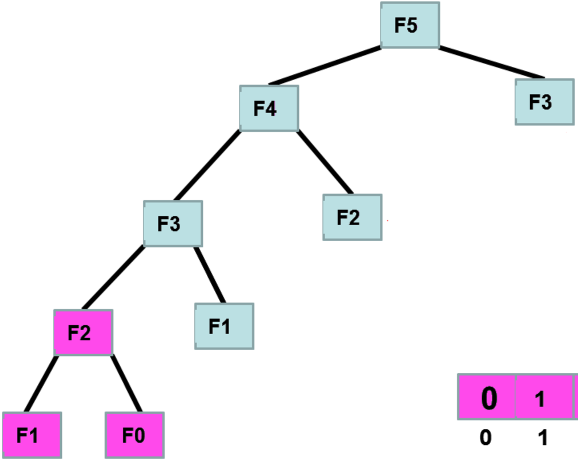
Solução Apresentada



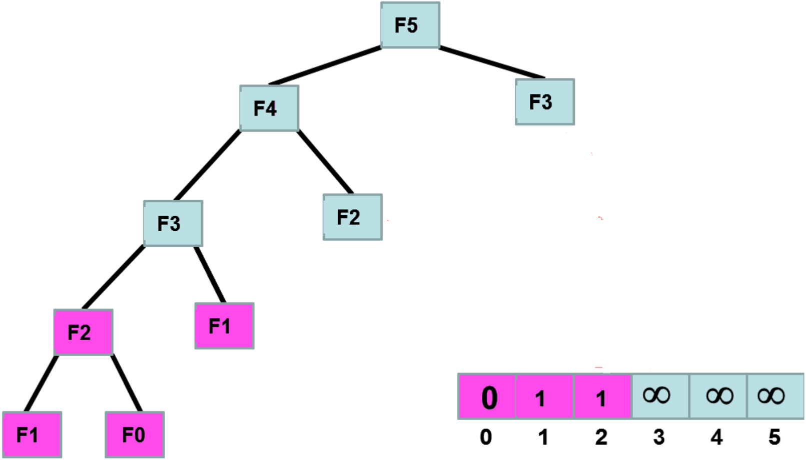
Solução Apresentada



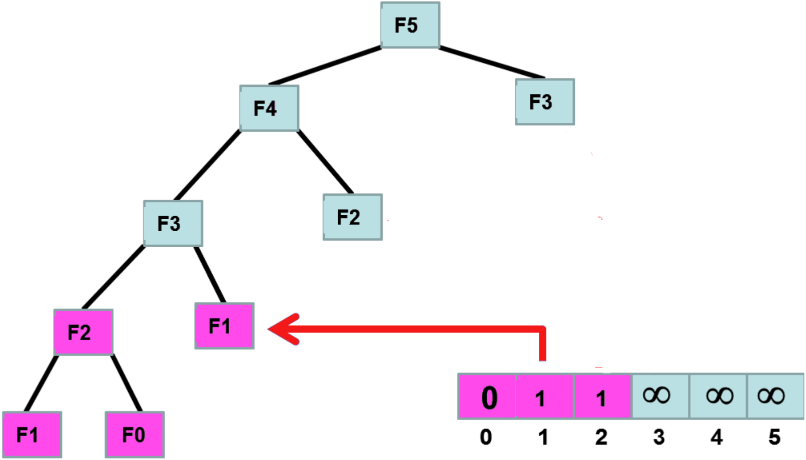
Solução Apresentada



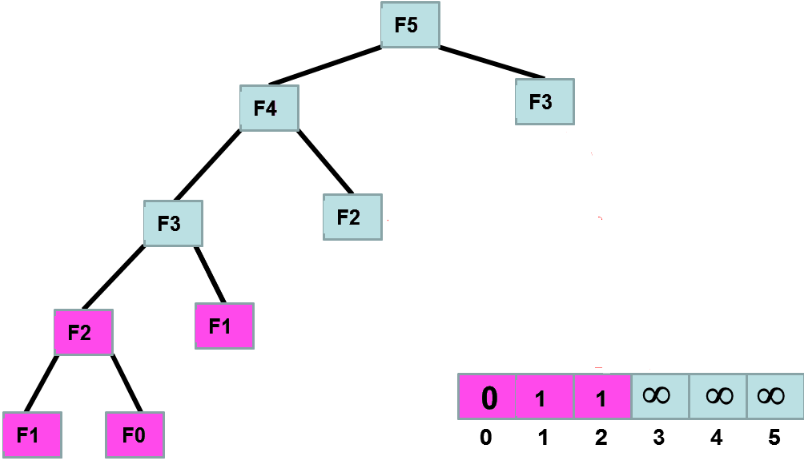
0	1	1	∞	∞	∞
0	1	2	3	4	5


F5
F4
F3
F3
F2
F2
F1
F1
F0
0 1 1 ∞ ∞ ∞
0 1 2 3 4 5

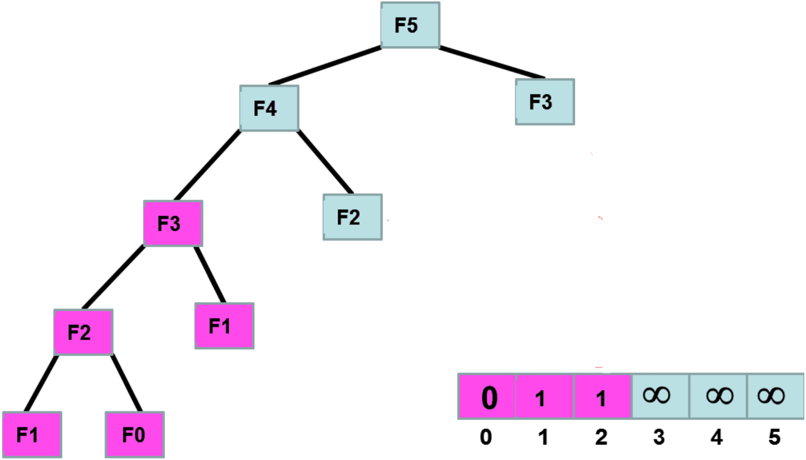
Solução Apresentada



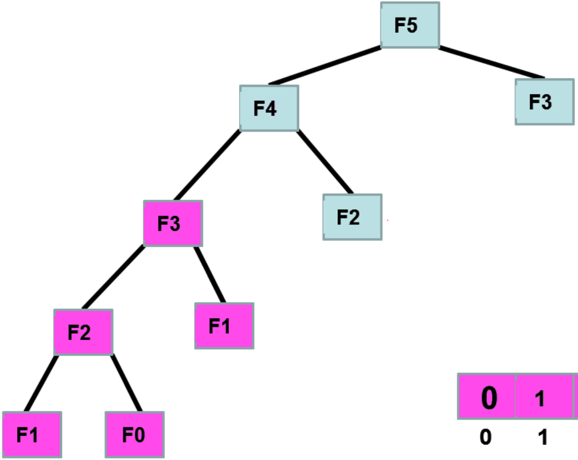
Solução Apresentada



Solução Apresentada

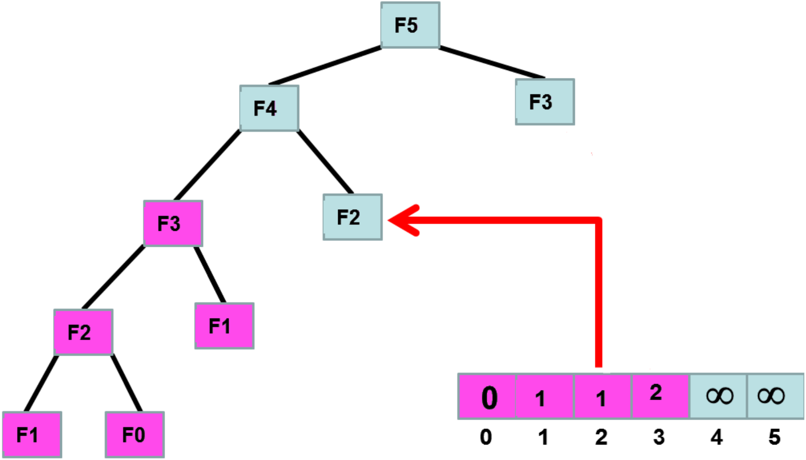


Solução Apresentada

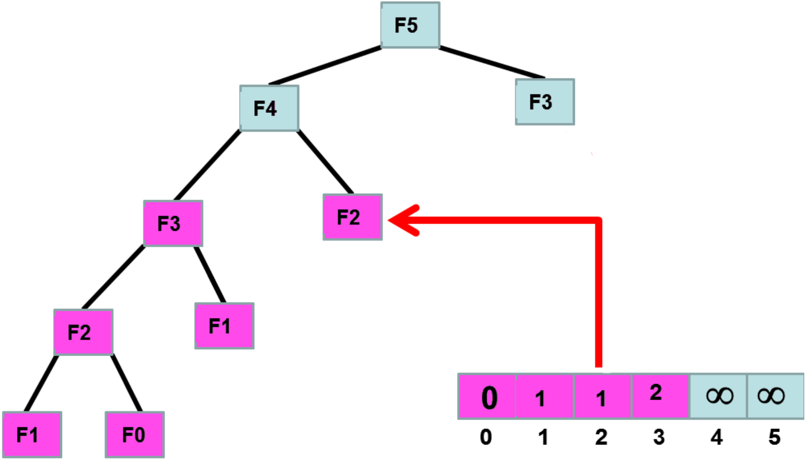


0	1	1	2	∞	∞
0	1	2	3	4	5

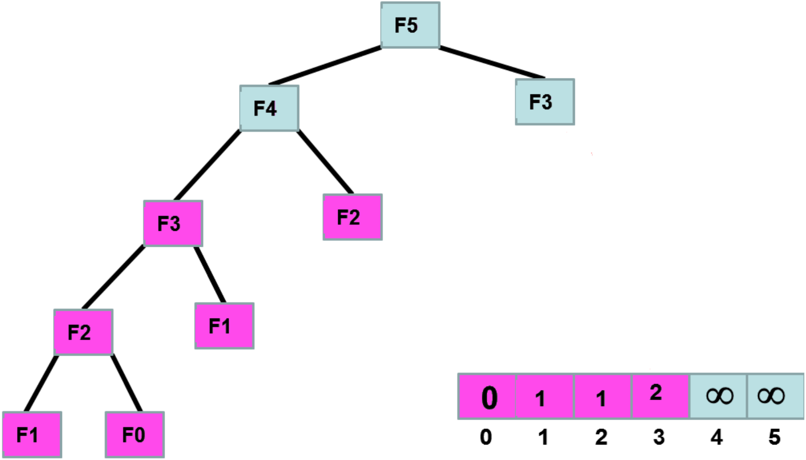
Solução Apresentada



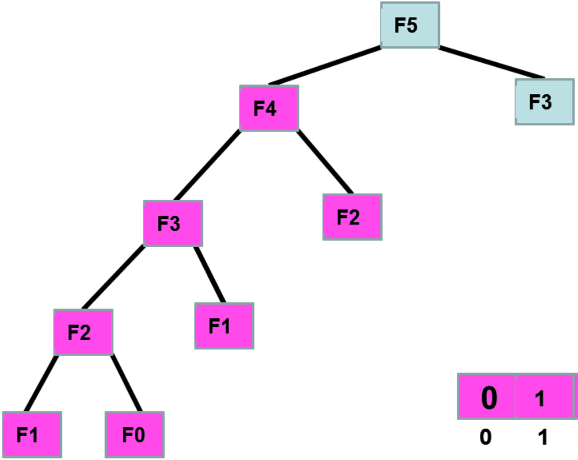
Solução Apresentada



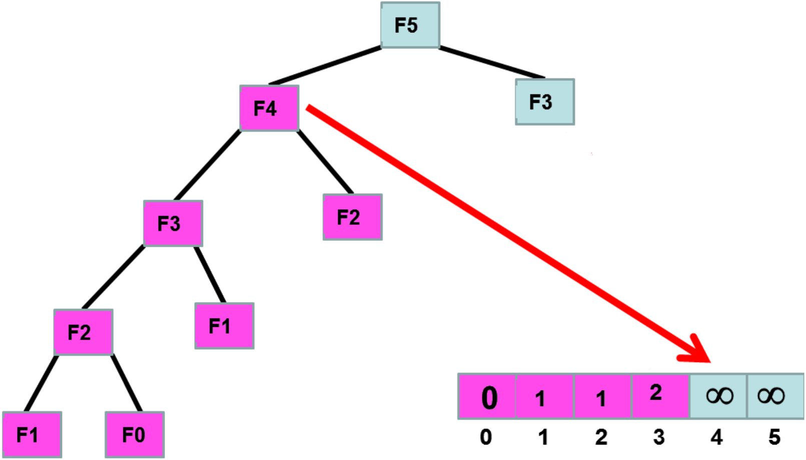
Solução Apresentada



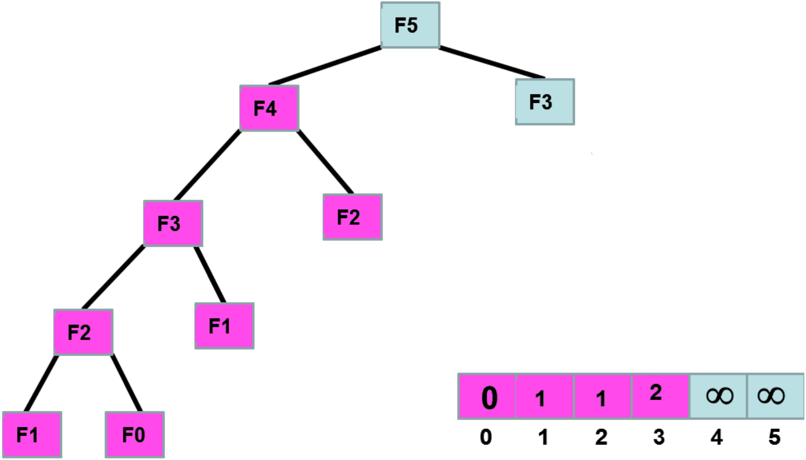
Solução Apresentada



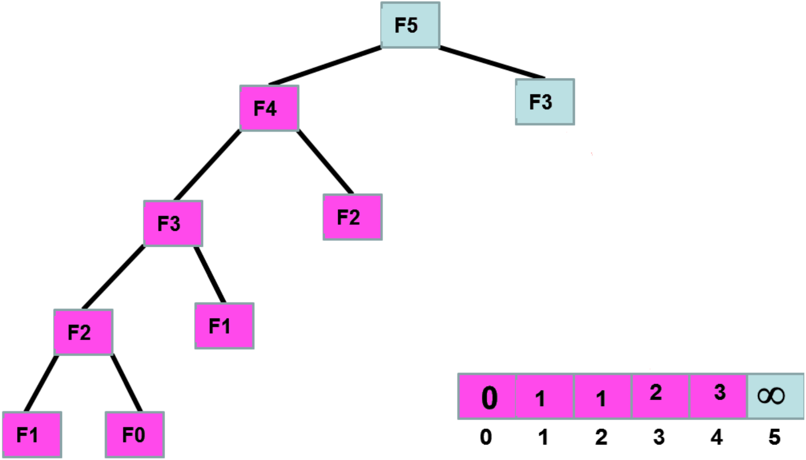
0	1	1	2	∞	∞
0	1	2	3	4	5


F5
F4
F3
F3
F2
F2
F1
F1
F0
0
1
1
2
∞
∞
0
1
2
3
4
5

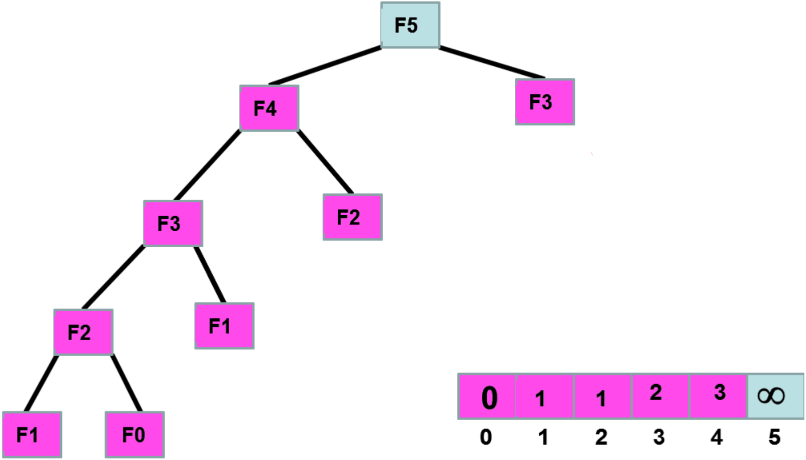
Solução Apresentada



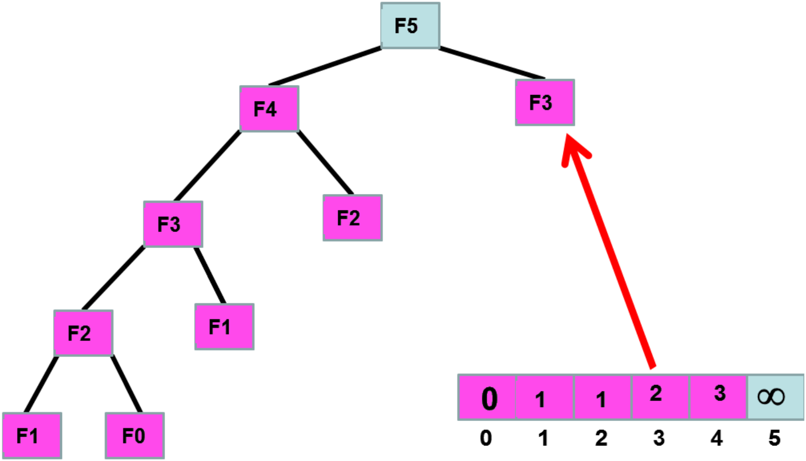
Solução Apresentada



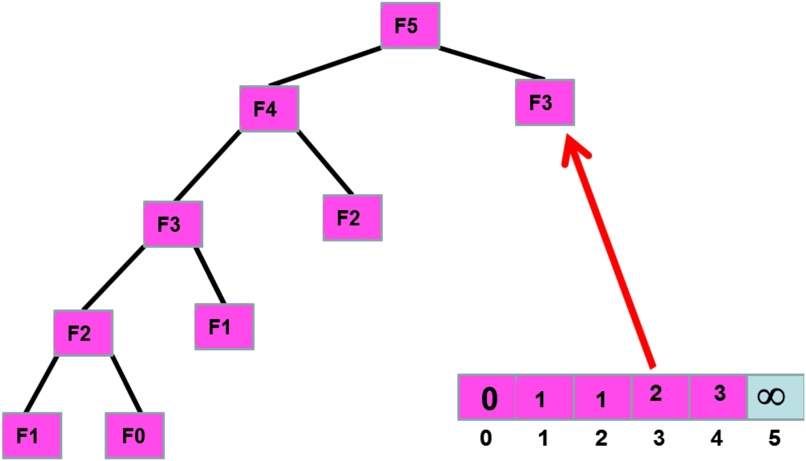
Solução Apresentada



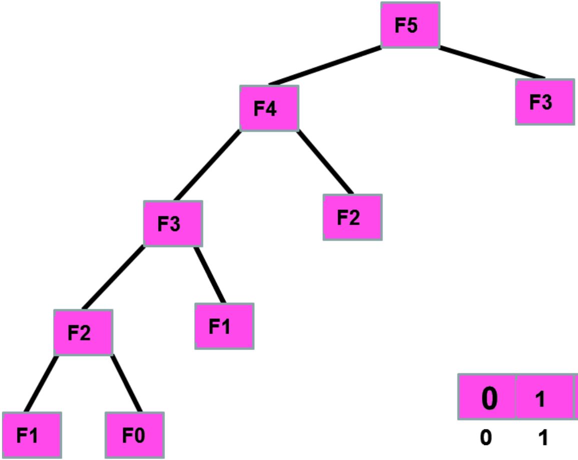
Solução Apresentada



Solução Apresentada

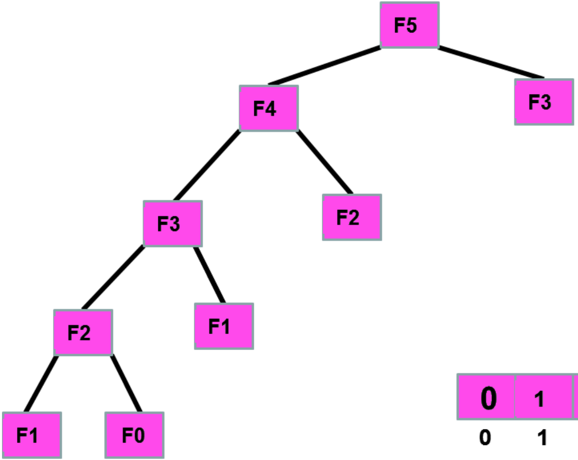


Solução Apresentada



0	1	1	2	3	∞
0	1	2	3	4	5

Solução Apresentada



0	1	1	2	3	5
0	1	2	3	4	5

Programação dinâmica - conclusão

Análise *Fib*1

- *Fib*1 é $O(n)$

Conclusão - como utilizar a abordagem:

- Encontrar uma função recursiva apropriada
- Adicionar **memorização** para armazenar resultados de subproblemas
- Determinar uma versão **bottom-up**, **iterativa**

Referências

- Cormen et. al. *Algoritmos - Teoria e Prática*.
- Nivio Ziviani. *Projeto de Algoritmos com Implementações em Pascal e C*.
- Toscani e Veloso. *Complexidade de Algoritmos*.