Análise de Algoritmos

PROGRAMAÇÃO DINÂMICA

Bacharelado em Ciências da Computação

Flávia Coelho
flaviacoelho@ufersa.edu.br

Atualizado em Janeiro de 2014

UNIVERSIDADE FEDERAL
UFERSA
RURAL DO SEMI-ÁRIDO

Sumário

- Motivação
- Fundamentos da Programação Dinâmica
- Exemplo de Utilização: Multiplicação de Matrizes
- Quando Aplicar Programação Dinâmica?
- Leitura Recomendada

Motivação

- Vamos considerar os números de Fibonacci
  - **Entrada**: Um número inteiro n
  - **Saída**: O número de Fibonacci F_n, definido da seguinte forma

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \; para \; n \geqslant 2$$

- A solução clássica utiliza recursão!!!

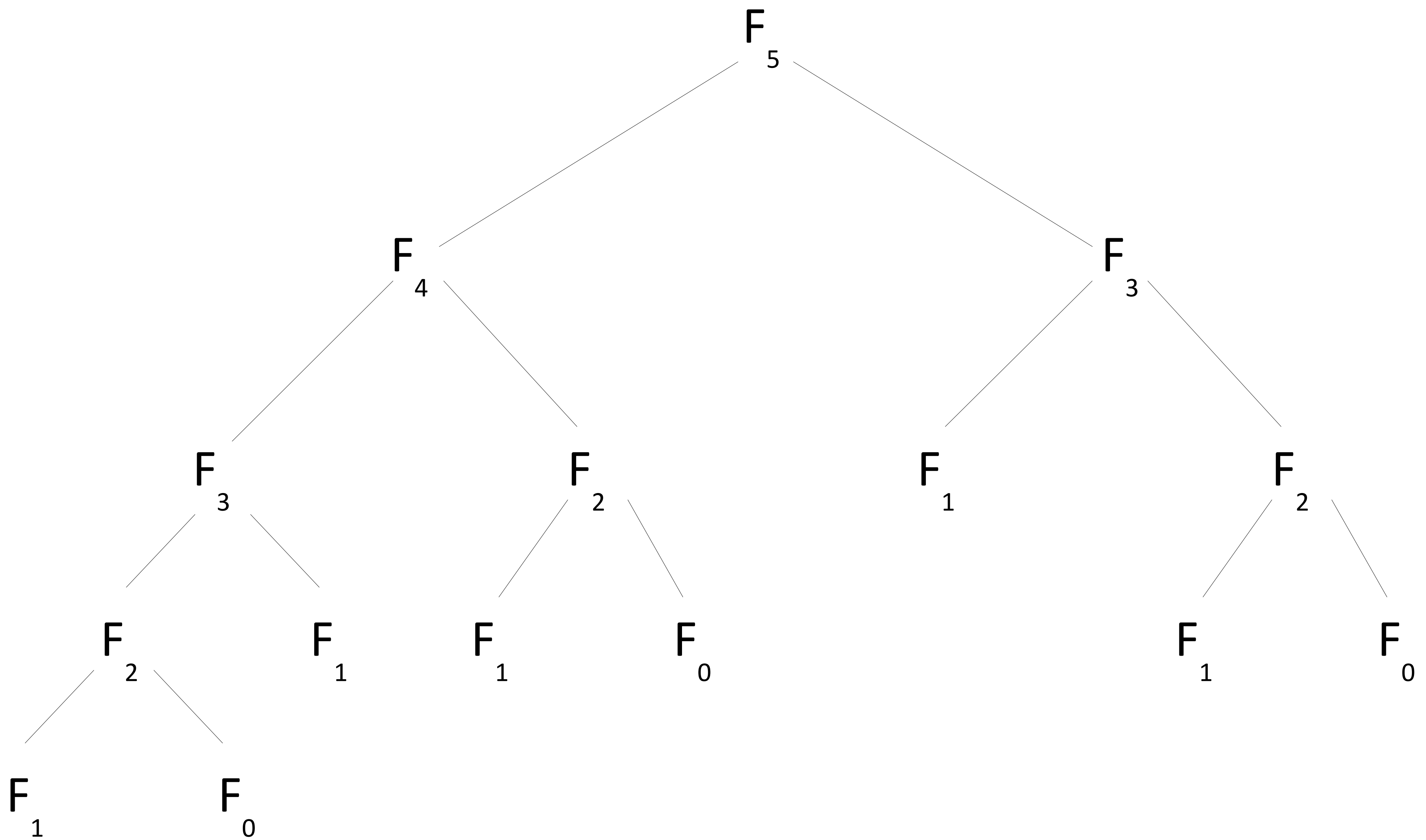
Exemplo

Número de Fibonacci usando Recursão

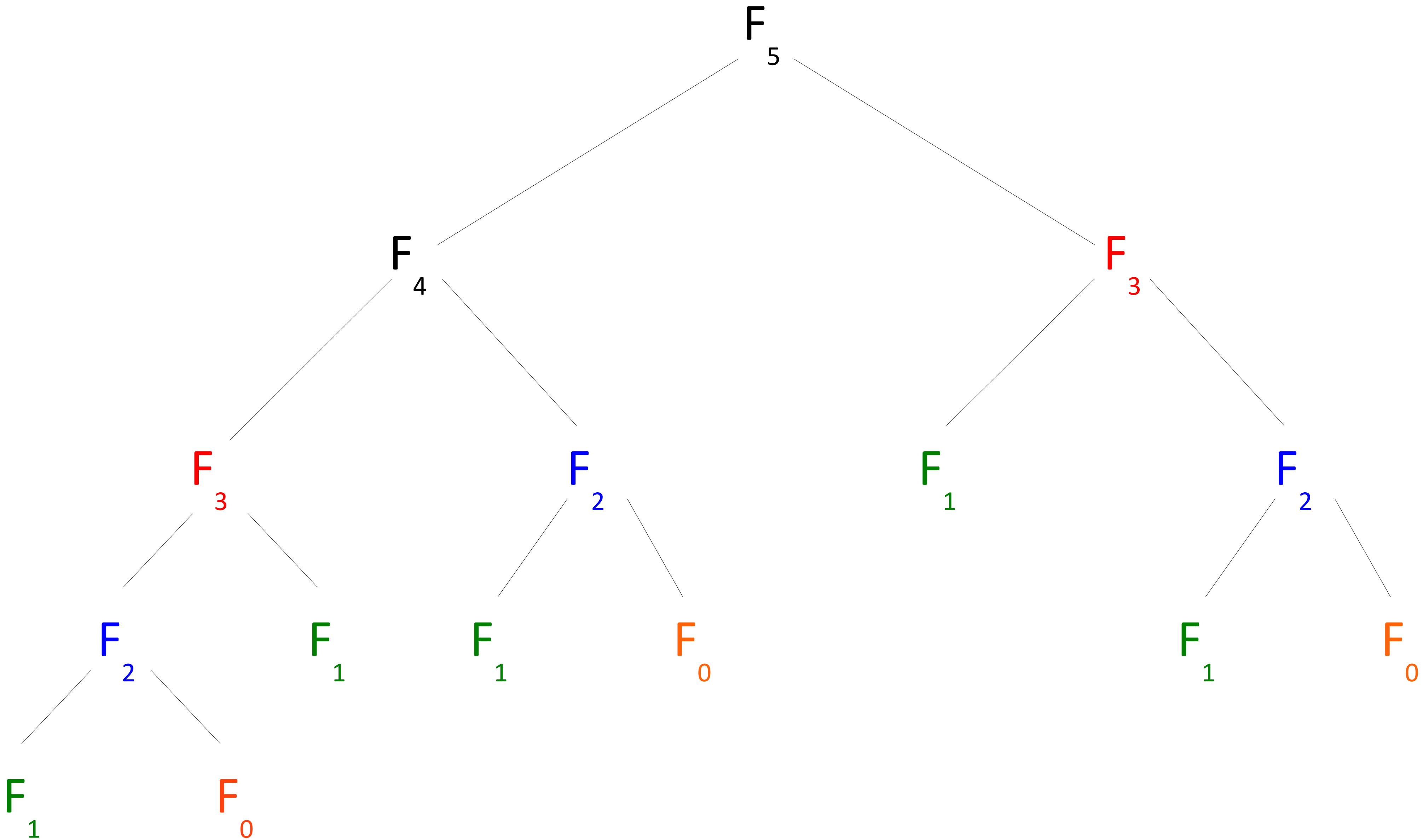
```
fib(n){
    if n ≤ 1 then
        return n
    return fib(n – 1) + fib(n – 2)
}
```

- Análise da complexidade
  - $T(n) = T(n-1) + T(n-2) + c \rightarrow O(2^n)$
  - Solução ineficiente!

Qual é o Motivo da Ineficiência?



Repetição Desnecessária de Cálculos



Dica

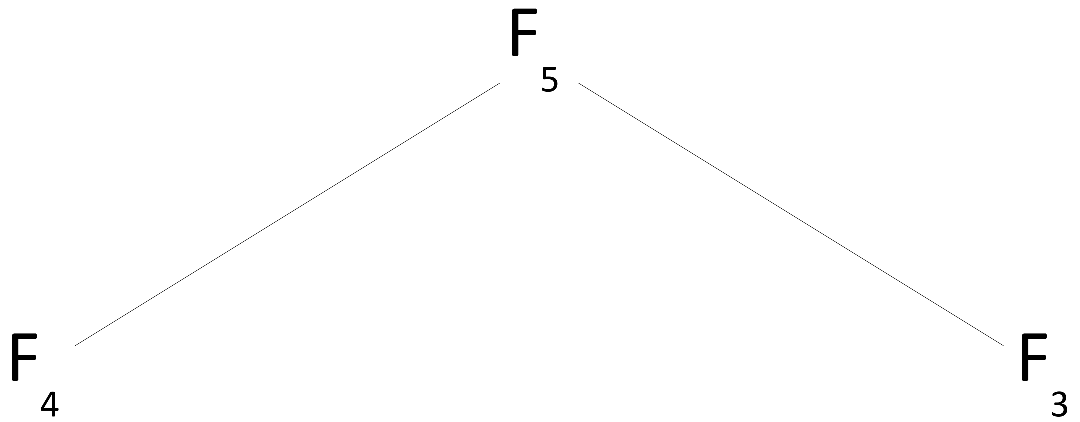
- Recursividade é útil quando o problema a ser resolvido pode ser dividido em subproblemas a um baixo custo e os subproblemas podem ser mantidos pequenos
- Quando a soma dos tamanhos dos subproblemas é O(n), então é provável que o algoritmo recursivo tenha complexidade polinomial
- Quando a divisão de um problema de tamanho n resulta em n subproblemas de tamanho (n – 1), então é provável que o algoritmo recursivo tenha complexidade exponencial

Solução Alternativa
Número de Fibonacci

- Utilizar um array f [0, ..., n] para guardar os valores calculados
- Inicialmente, f contém apenas ∝

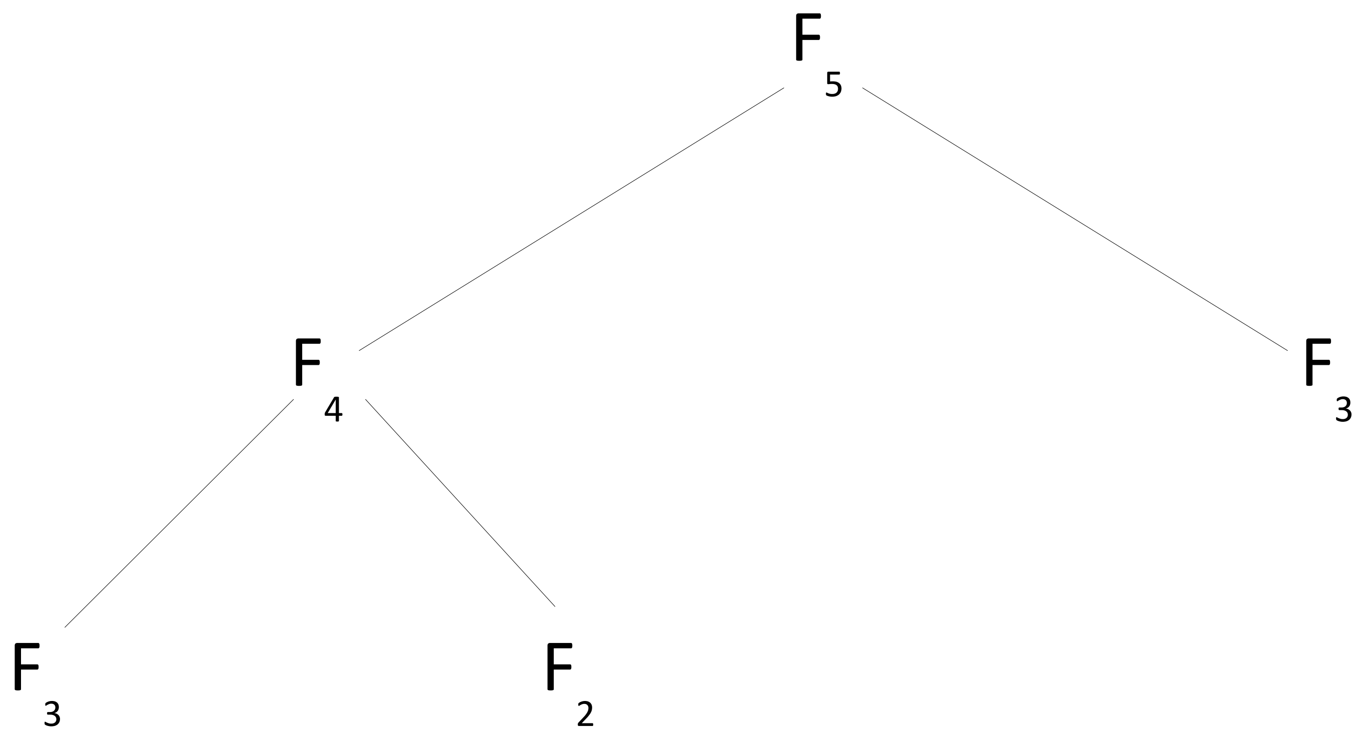
```
fib1(n){
    if f[n] != ∝ then
        return f[n]
    if n ≤ 1 then
        return f[n] = n
    return f[n] = fib1(n-1) + fib1(n-2)
}
```

Executando...



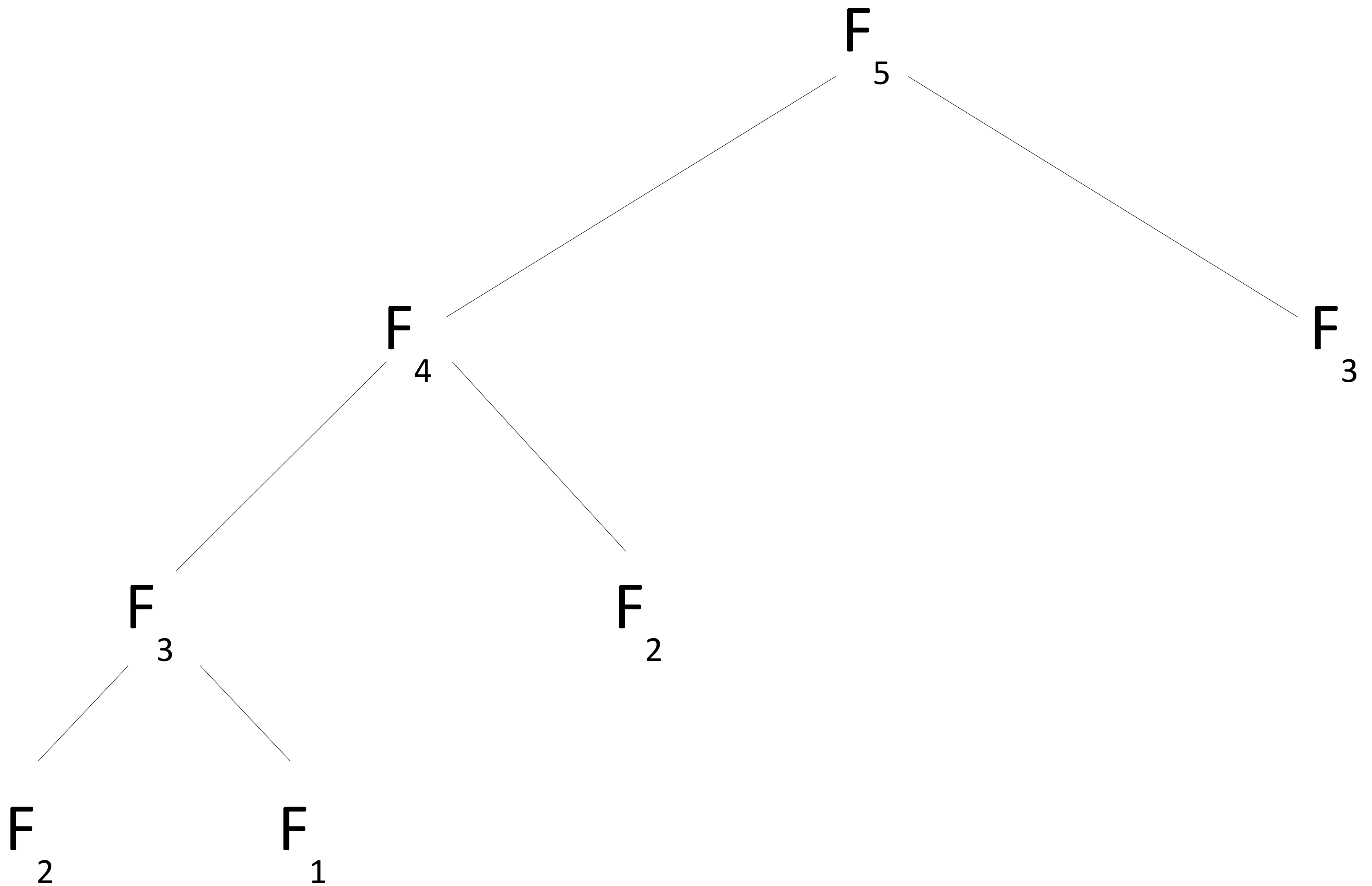
∝	∝	∝	∝	∝	∝

Executando...



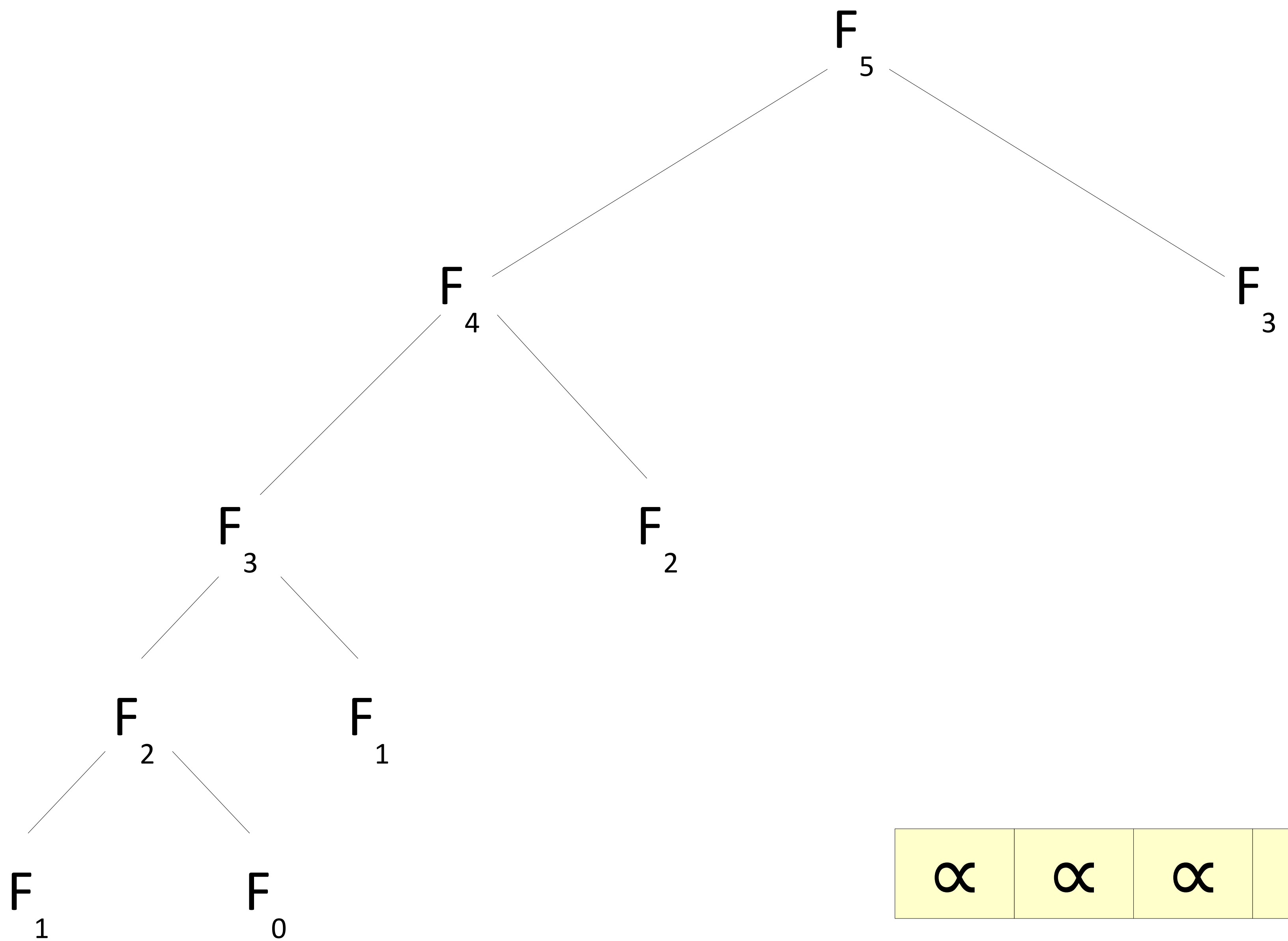
∝	∝	∝	∝	∝	∝

Executando...



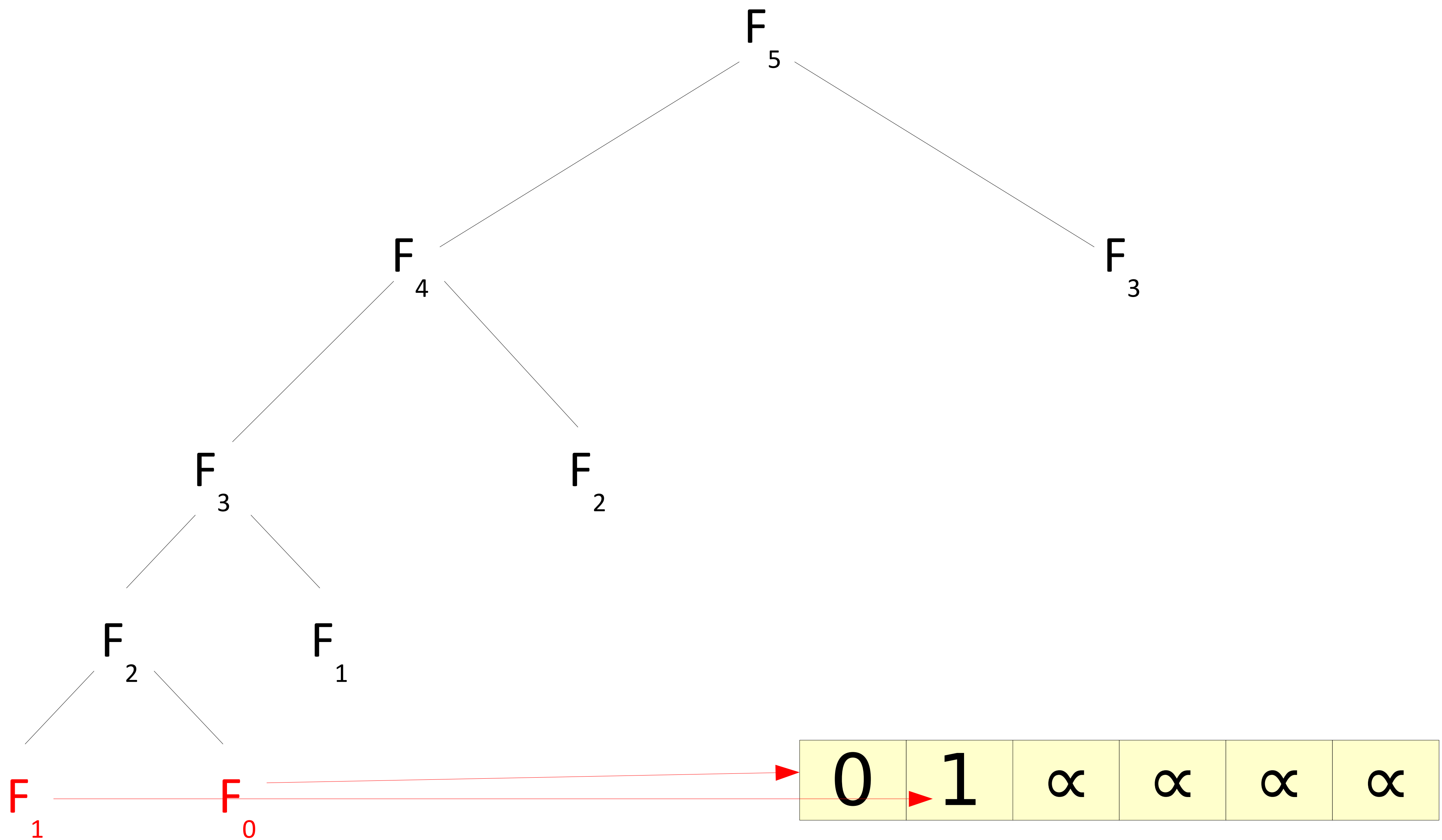
∝	∝	∝	∝	∝	∝

Executando...

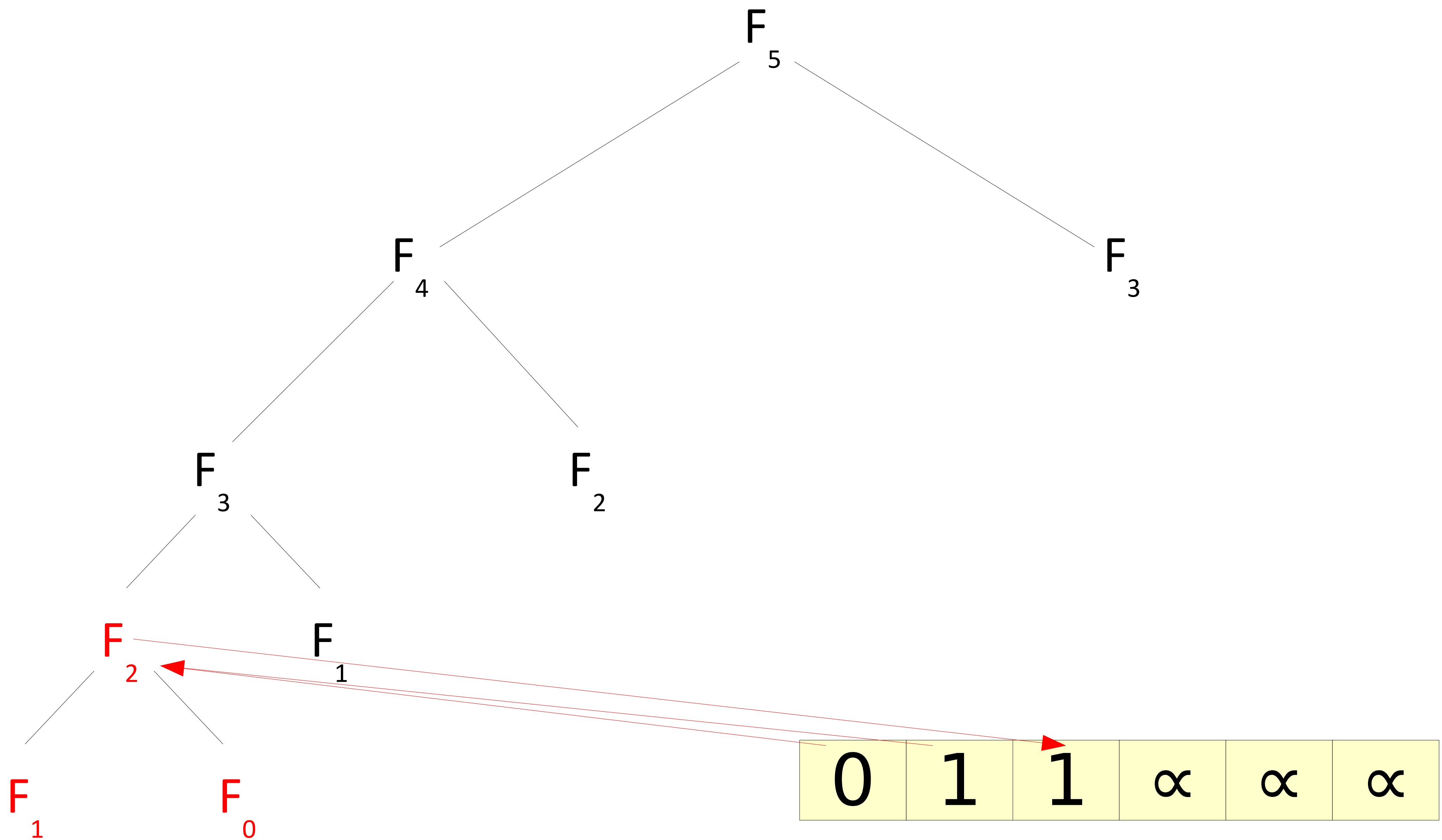


∝	∝	∝	∝	∝	∝

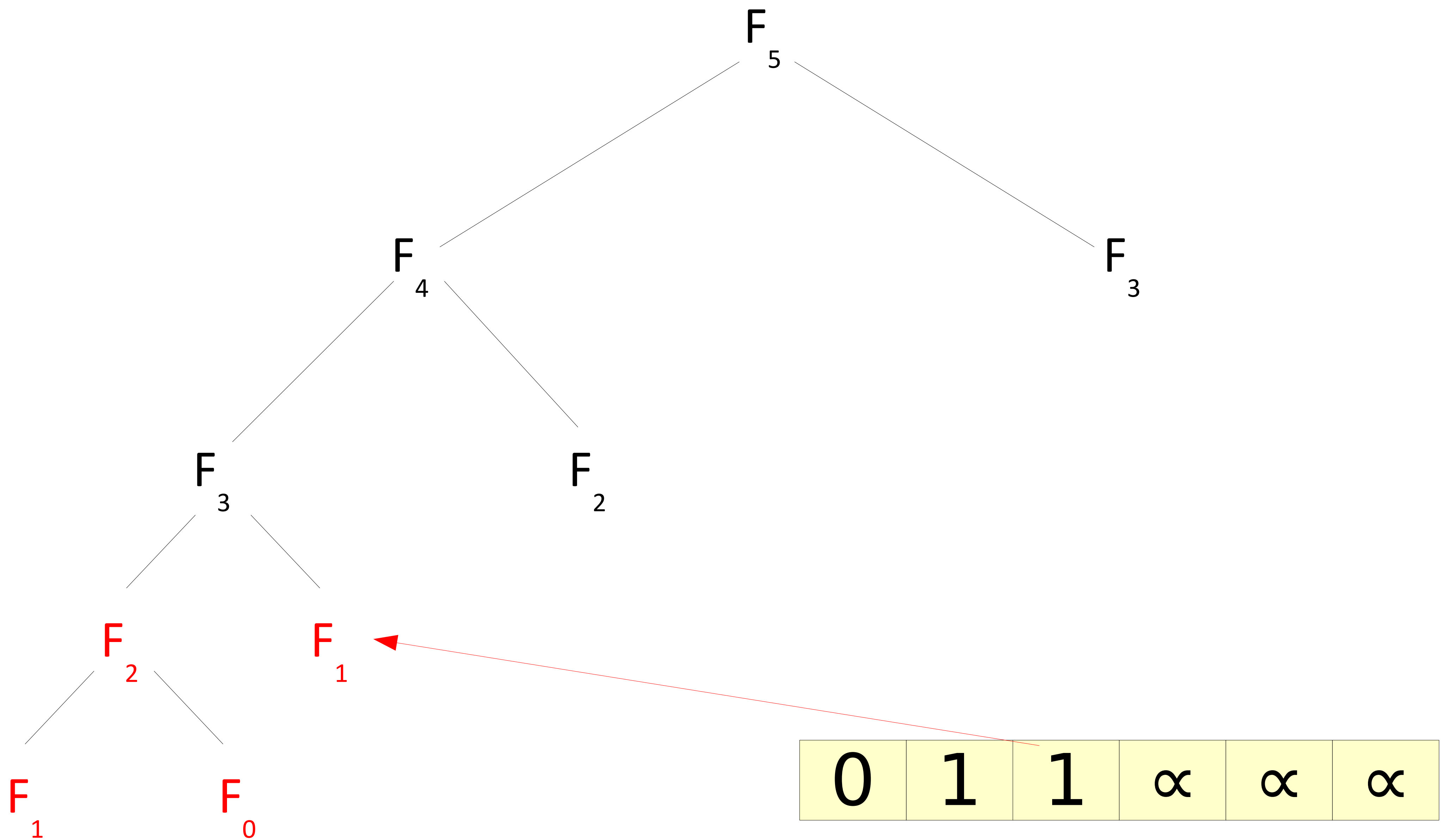
Executando...



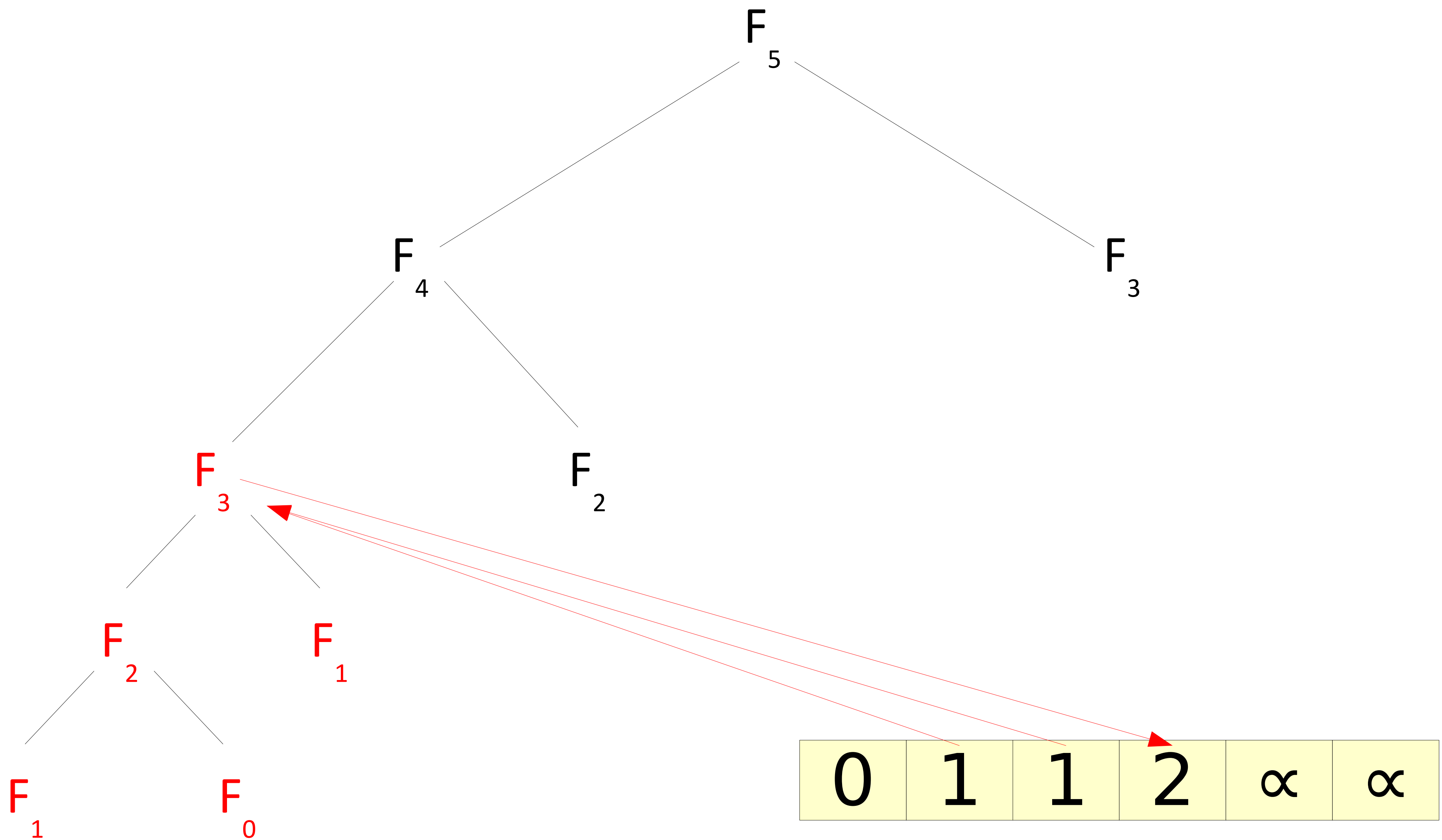
Executando...



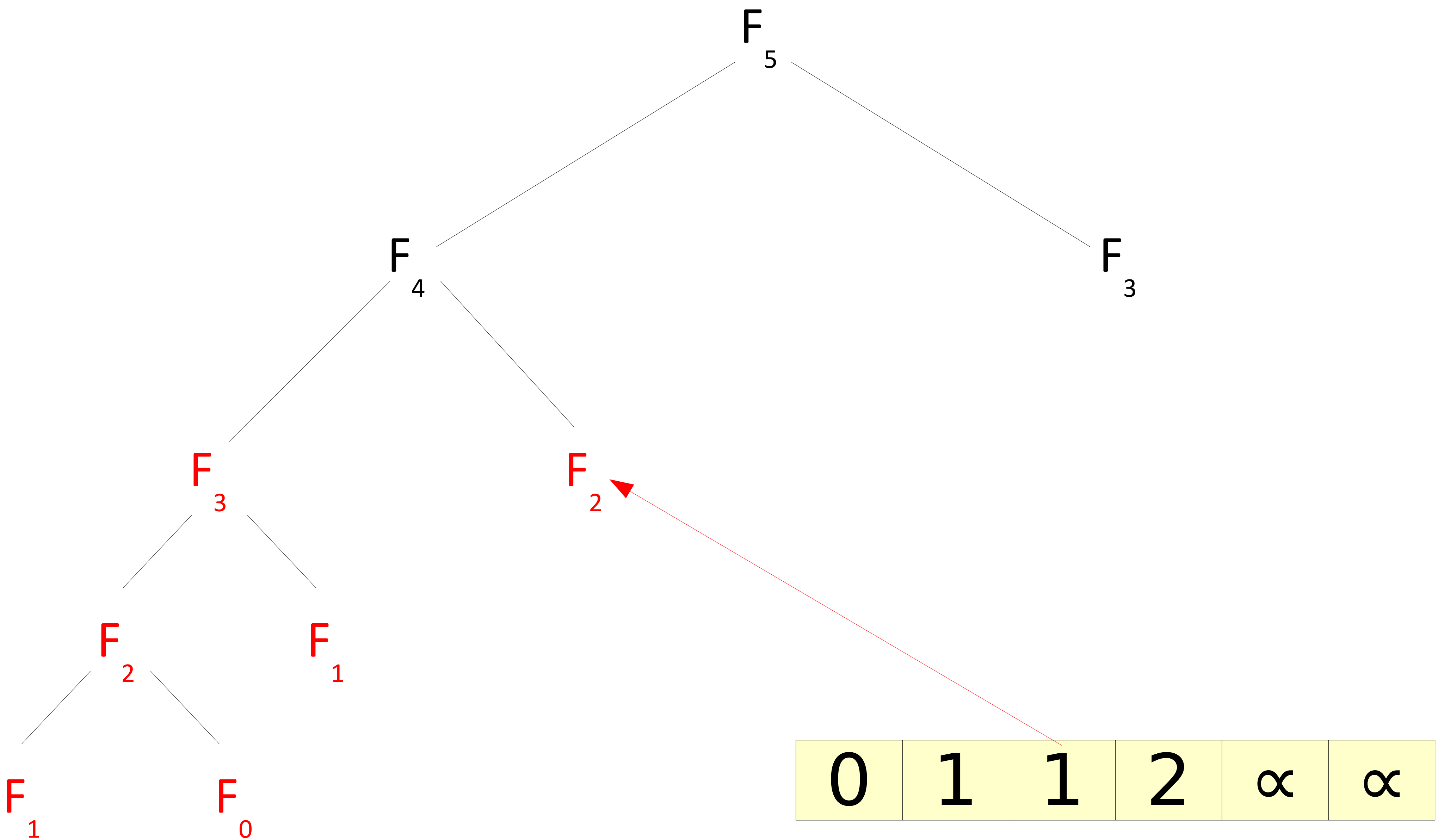
Executando...



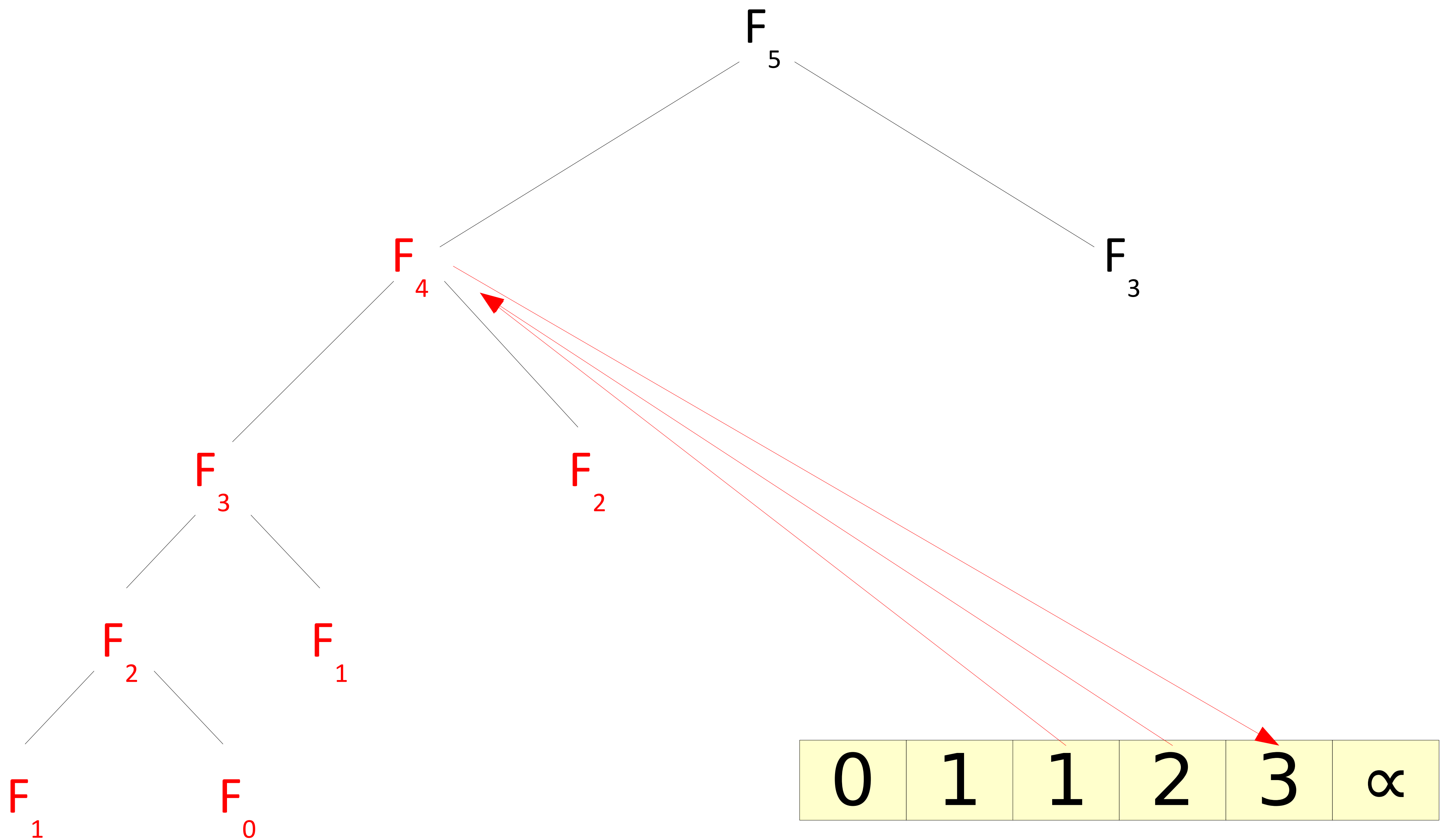
Executando...



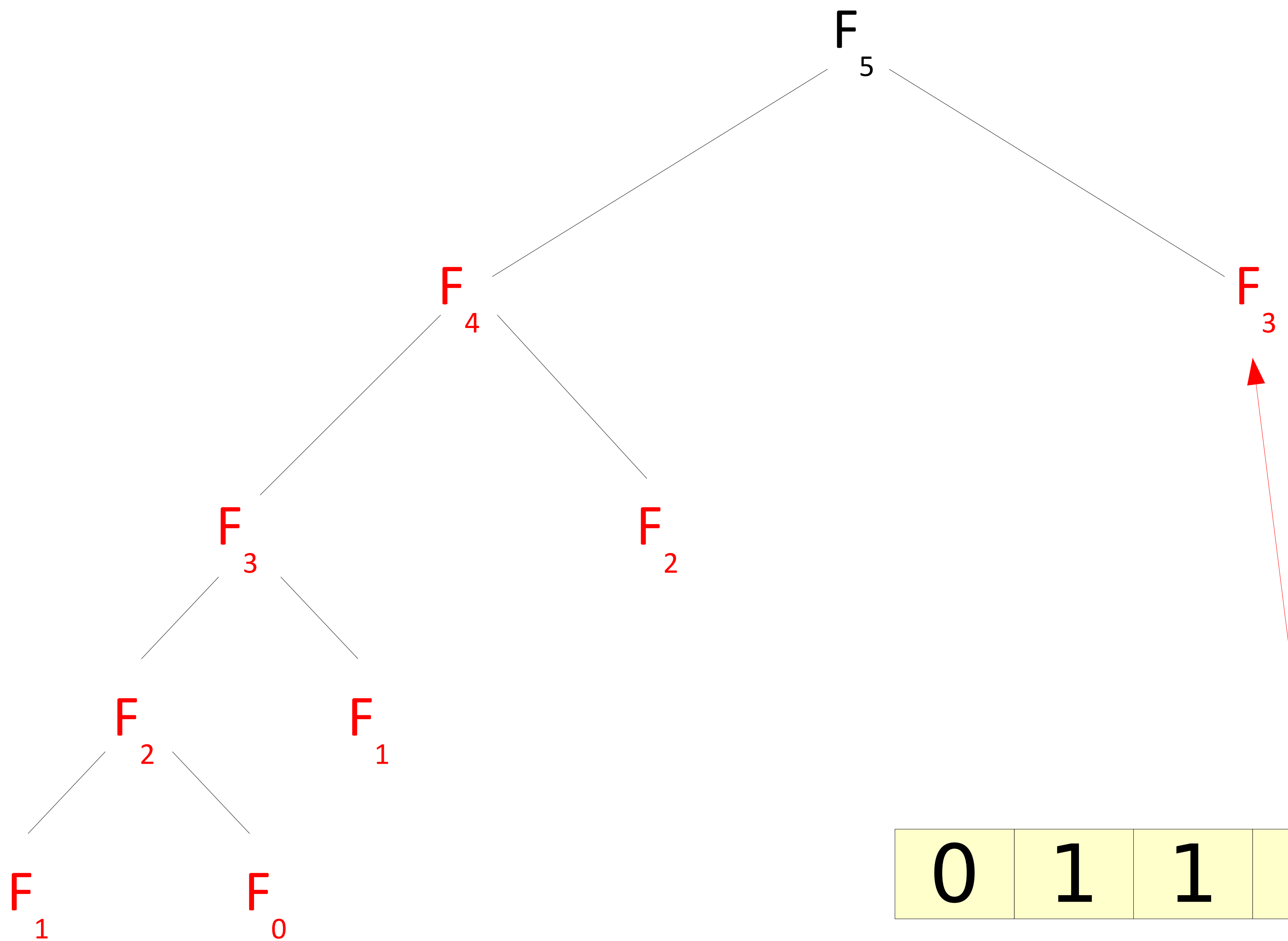
Executando...



Executando...

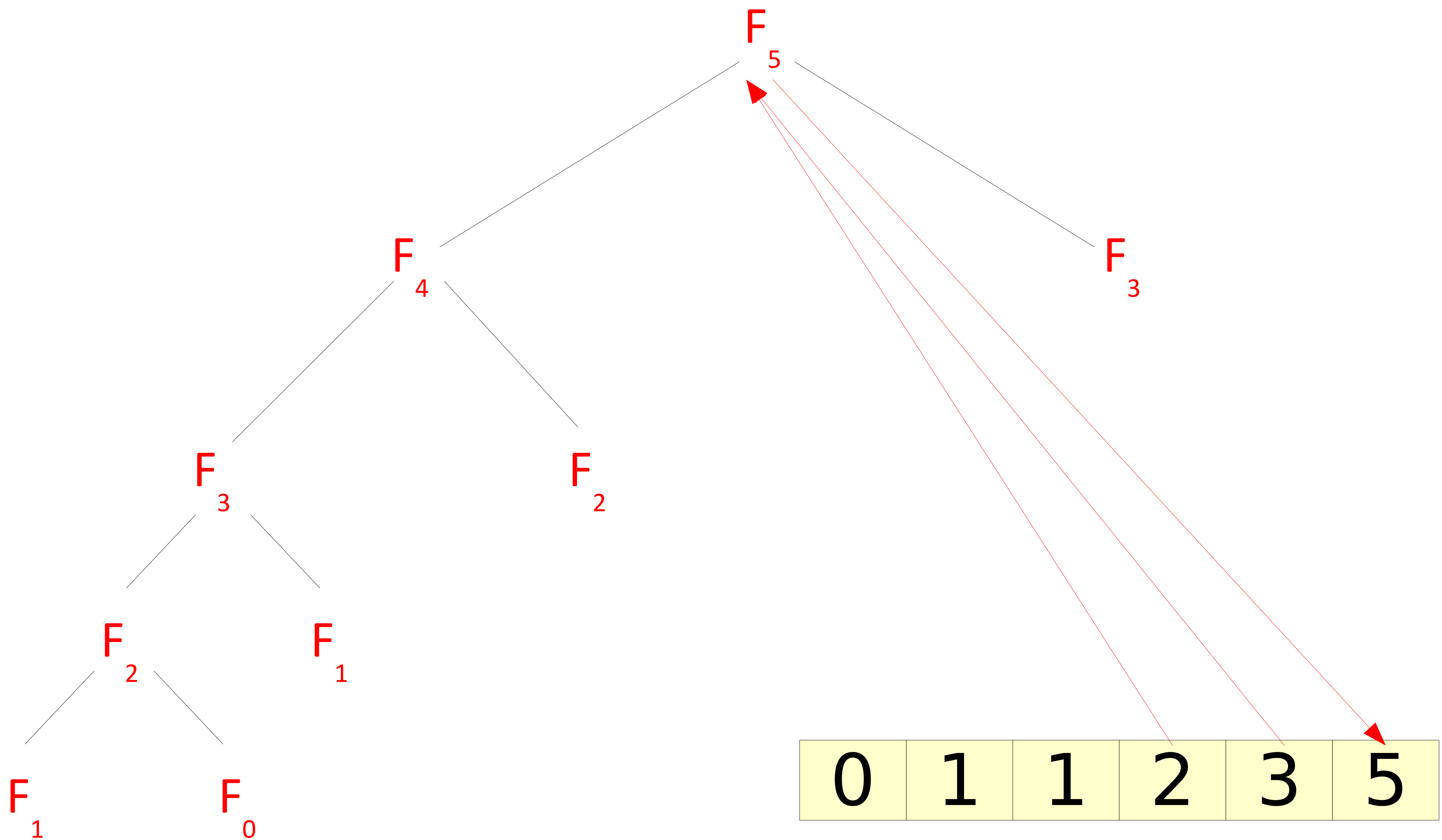


Executando...

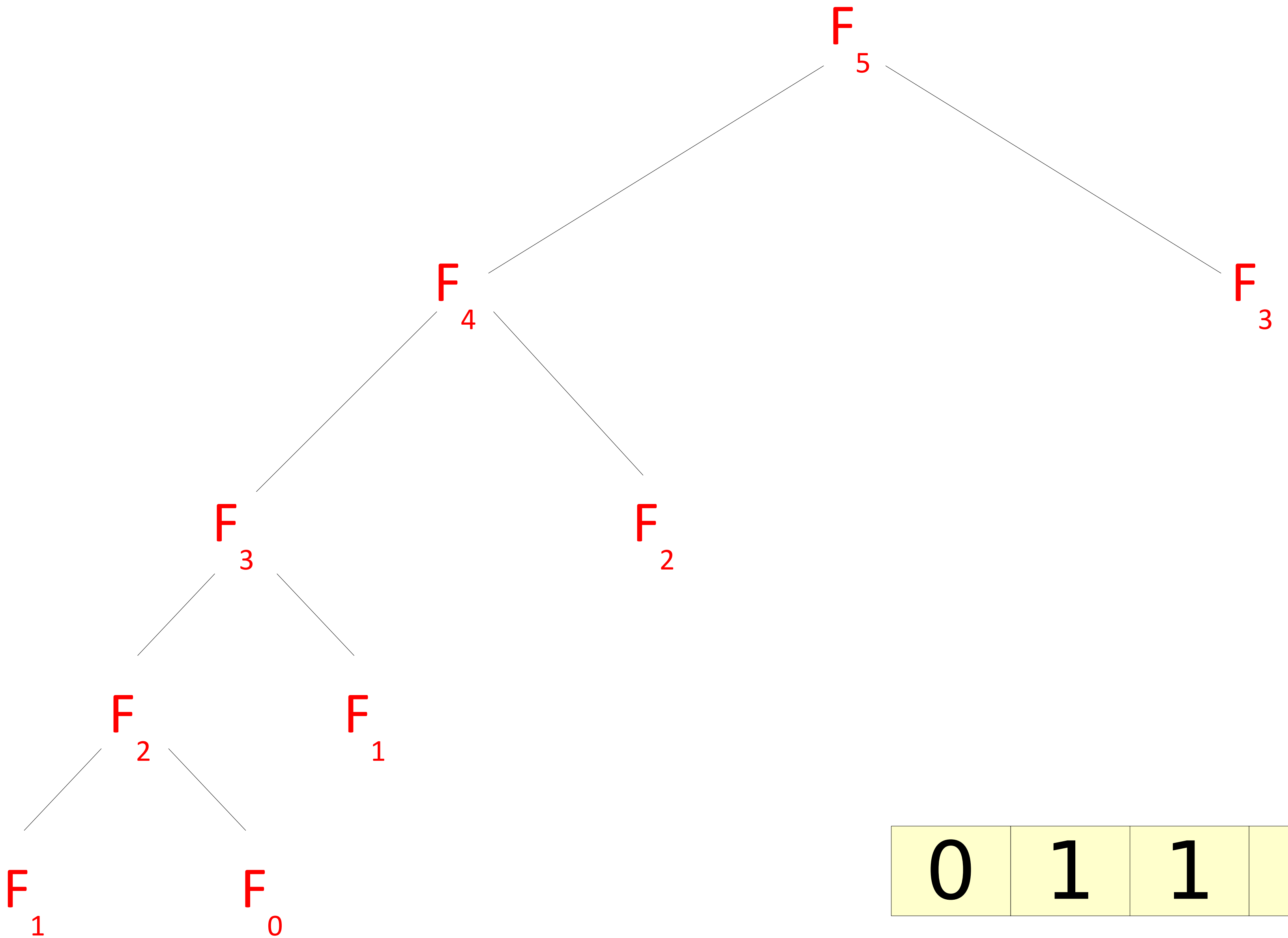


0	1	1	2	3	∝

Executando...



Executando...



0	1	1	2	3	5

Outra Solução Alternativa

- Elimina as chamadas recursivas
- Utiliza um arranjo para armazenar os dados calculados
- Estratégia *bottom-up*

```
fib2(n){
    f[0] = 0
    f[1] =1
    for i = 2 to n do
        f[i] = f[i-1] + f[i-2]
    return f[n]
}
```

Outra Solução Alternativa

- Complexidade computacional é O(n)
- Abordagem utilizada
  - Adicionar memorização para armazenar resultados de subproblemas
  - Elaborar uma versão *bottom-up*, iterativa

```
fib2(n){
    f[0] = 0
    f[1] =1
    for i = 2 to n do
        f[i] = f[i-1] + f[i-2]
    return f[n]
}
```

fib2 é Programação Dinâmica!!!

Sumário

- Motivação
- Fundamentos da Programação Dinâmica
- Exemplo de Utilização: Multiplicação de Matrizes
- Quando Aplicar Programação Dinâmica?
- Leitura Recomendada

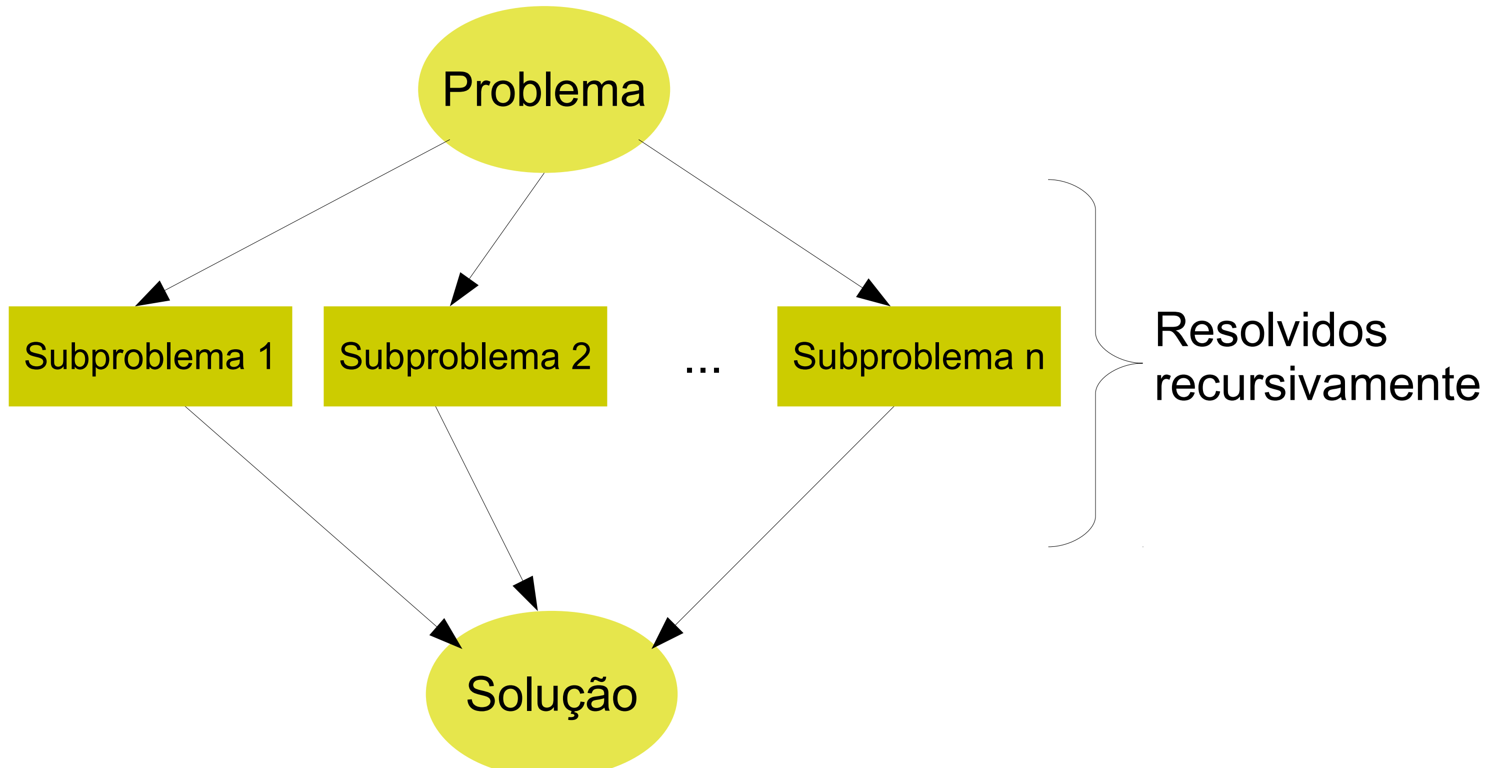
Programação Dinâmica

- Técnica exata para o projeto de algoritmos eficientes
- Aplicável a problemas de otimização combinatória
  - Problemas baseados em uma série de escolhas (ou de decisões) para atingir uma solução ótima
- Aplicado quando a recursão produz repetição dos mesmos subproblemas
  - PD resolve um problema combinando as soluções para subproblemas, armazenando as respostas em uma área de armazenamento auxiliar

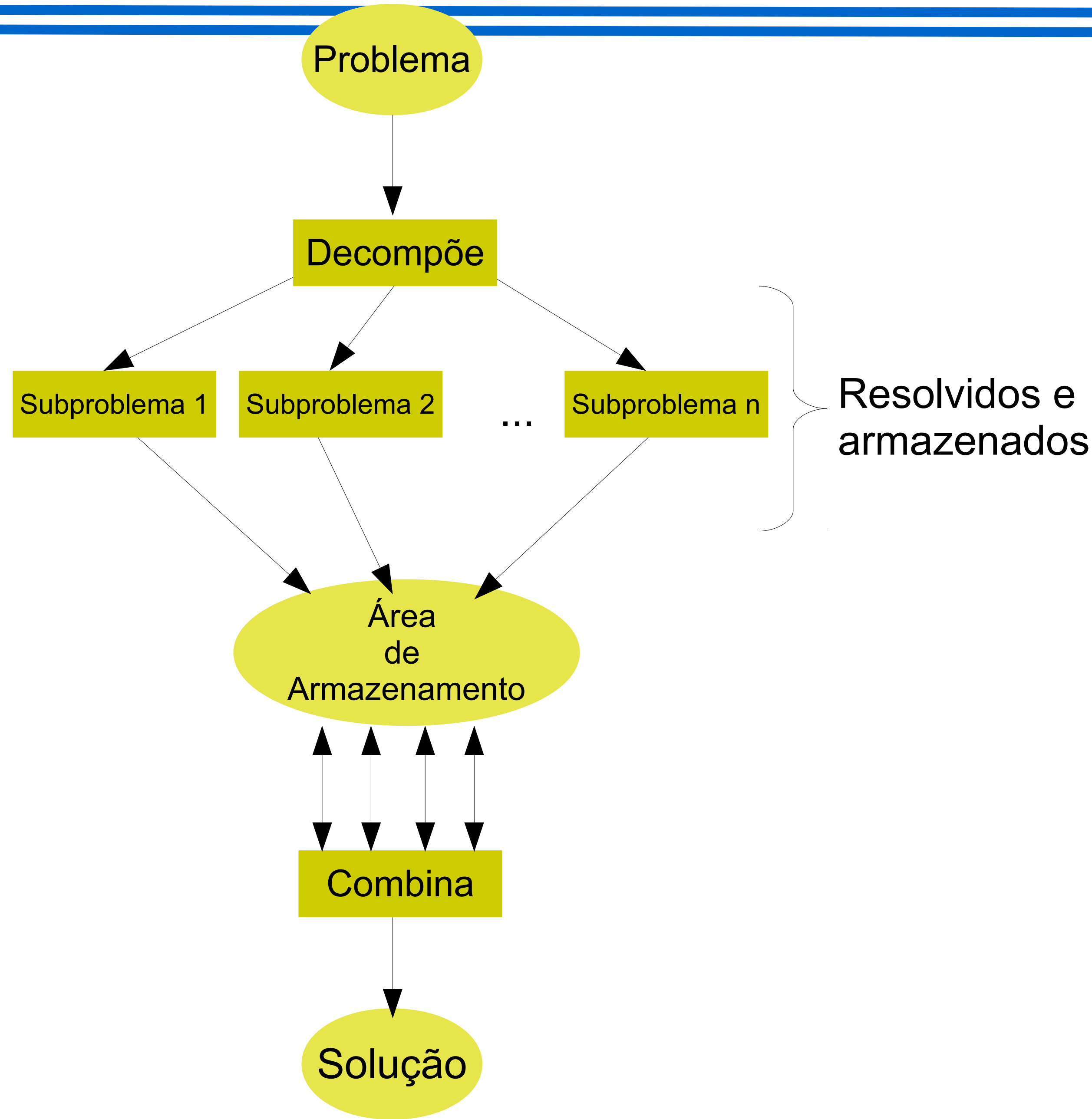
Divisão-e-Conquista vs PD

- Divisão-e-conquista particiona o problema em subproblemas independentes, resolvendo-os de modo recursivo e combinando as soluções para resolver o problema original
- PD particiona o problema em subproblemas dependentes (subproblemas ”compartilham” subproblemas), resolvendo cada subproblema apenas uma vez e armazenando as soluções em uma estrutura de dados para resolver o problema original

Divisão-e-Conquista



Programação Dinâmica



Divisão-e-Conquista vs PD
Concluindo...

PROGRAMAÇÃO DINÂMICA = DIVISÃO-E-CONQUISTA + MEMÓRIA – RECURSIVIDADE

Características da PD

- Busca **uma** solução ótima para um problema
  - Atenção! Podem existir várias soluções que alcançam o valor ótimo esperado
- O desenvolvimento de algoritmos de PD compreende quatro etapas
  - Caracterizar a estrutura de uma solução ótima
  - Definir o valor de uma solução ótima
  - Calcular o valor de uma solução ótima em um processo de baixo-para-cima (*bottom-up*)
  - Construir uma solução ótima a partir de informações calculadas

Sumário

- Motivação
- Fundamentos da Programação Dinâmica
- Exemplo de Utilização: Multiplicação de Matrizes
- Quando Aplicar Programação Dinâmica?
- Leitura Recomendada

Exemplo de Utilização

- Multiplicação de matrizes
  - Dadas as matrizes M_1, M_2, ... M_n, calcular o produto

    $M = M_1 \times M_2 \times \ldots \times M_n$
- A multiplicação de matrizes é associativa
  - Todas as associações resultam no mesmo produto
  - Exemplos
    - $M_1 \times M_2 \times M_3$
    - $M_1 \times (M_2 \times M_3)$
    - $(M_1 \times M_2) \times M_3$

Associações são distintas, mas o resultado é o mesmo!

Multiplicando Matrizes

- Podemos multiplicar duas matrizes $A_{p \times q}$ e $B_{q \times r}$, somente se elas forem compatíveis
  - O número de colunas de A é igual ao número de linhas de B → válido
  - O número de operações (multiplicações) necessárias (custo) é p.q.r
- As associações aplicadas podem impactar drasticamente o custo de obtenção do produto

Multiplicando Matrizes

- Vejamos um exemplo
  - $A_{50x20} \times B_{20x1} \times C_{1x10} \times D_{10x100}$

Organização dos Parênteses	Computação do Custo	Custo
A x ((B x C) x D)	20.1.10 + 20.10.100 + 50.20.100	120.200
(A x (B x C)) x D	20.1.10 + 50.20.10 + 50.10.100	60.200
(A x B) x (C x D)	50.20.1 + 1.10.100 + 50.1.100	7.000

- Ao aplicar PD, neste caso, buscamos determinar a ordem de multiplicação de matrizes que forneça o custo mais baixo (ótimo)

Complexidade Computacional

- A verificação exaustiva de todas as possíveis ordens (colocação de parênteses) para o cálculo não é eficiente!
- Seja P(n) o número de alternativas para colocar os parênteses em uma sequência de n matrizes
  - Quando $n = 1$, é trivial (apenas 1 modo)
  - Quando $n \geq 2$, um produto de matrizes totalmente entre parênteses é o produto de dois subprodutos de matrizes totalmente entre parênteses
- P(n) é $\Omega(2^n)$

Solução

- Etapas da programação dinâmica
  - Etapa 1: caracterizar a estrutura de uma colocação ótima de parênteses
  - Etapa 2: definir uma solução ótima
  - Etapa 3: calcular os custos ótimos
  - Etapa 4: construir uma solução ótima

Etapa 1: Estrutura de uma Colocação Ótima dos Parênteses

- Vamos adotar a notação $M_{i..j}$, onde $i \leq j$ para a matriz-resultado do produto entre $M_i M_{i+1} \ldots M_j$
- Para algum valor k, onde $i \leq k < j$, primeiro calculamos as matrizes $M_{i..k}$ e depois $M_{k+1..j}$ e depois multiplicamos os dois para gerar o produto final $M_{i..j}$
- O custo da colocação de parênteses é portanto o custo de calcular a matriz $M_{i..k}$ mais o custo de calcular $M_{k+1..j}$, mais o custo de multiplicá-las uma pela outra

Etapa 1: Estrutura de uma Colocação Ótima dos Parênteses

- Considere que uma colocação de parênteses de $M_i M_{i+1} ... M_j$ divida o produto entre $M_{i..k}$ e $M_{k+1..j}$
  - Então, a colocação dos parênteses de $M_i M_{i+1} ... M_k$ dentro dessa colocação ótima dos parênteses de $M_i M_{i+1} ... M_j$ deve ser uma colocação ótima
- Podemos construir uma solução ótima para uma instância do problema dividindo o problema em 2 subproblemas (por meio da colocação ótima dos parênteses de $M_i M_{i+1} ... M_k$ e $M_{k+1} M_{k+2} ... M_j$), encontrando as soluções ótimas para os subproblemas e combinando as soluções
- Atenção! Quando procurarmos o lugar correto para dividir o produto, precisamos considerar **todos os lugares possíveis**

Etapa 2: Definir uma Solução

- Definimos o custo de uma solução ótima em termos das soluções ótimas dos subproblemas
- Para o problema de multiplicação de matrizes, os subproblemas buscam determinar o custo mínimo de uma colocação de parênteses de $M_i M_{i+1} \ldots M_j$ para $1 \leq i \leq j \leq n$
- Considere m[i,j], o número mínimo de multiplicações necessárias para calcular a matriz $M_{i..j}$ para o problema original
  - A matriz $M_{1..n}$, o custo é dado por m[1,n]

Etapa 2: Uma Solução

- Vamos definir m[i,j]
  - Se $i = j$, o problema é trivial ($M_{i..i} = M_i$)
    - nenhuma operação de multiplicação é necessária
    - Portanto, $m[i,i] = 0$, para i = 1, 2, 3, ..., n

Etapa 2: Uma Solução

- Vamos definir m[i,j]
  - Se i < j, tiramos proveito da estrutura de uma solução ótima da Etapa 1
    - Supomos que a colocação ótima dos parênteses divida o produto $M_i M_{i+1} \ldots M_j$ entre M_k e M_{k+1}, onde $i \leq k < j$
      - Então m[i,j] é igual ao custo mínimo para calcular os subprodutos $M_{i..k}$ e $M_{k+1..j}$ mais o custo de multiplicar as 2 matrizes
    - Sendo que cada M_i é p_{i-1} x p_i (dimensão), o cálculo do produto das matrizes $M_{i..k} M_{k+1..j}$ exige $p_{i-1}.p_k.p_j$ operações de multiplicação
    - Portanto, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j$

Etapa 2: Uma Solução

- A equação $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j$ pressupõe que conhecemos o valor de k → *o que não ocorre!*
- Na verdade, há j-i valores possíveis para k (k=i, i+1,...,j-1)
  - Como a colocação ótima dos parênteses deve usar um desses valores para k, precisamos verificar todos eles para encontrar o melhor!
- Portanto, a definição para o custo mínimo de colocar entre parênteses o produto $M_i M_{i+1} \ldots M_j$ é dada por

$$m[i,j] = \begin{cases} 0, \text{ se } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j\}, \text{ se } i < j \end{cases}$$

Etapa 3: Como Calcular os Custos Ótimos

- Buscamos o custo mínimo m[1,n] para multiplicar $M_i M_{i+1} \ldots M_j$
- Neste momento, temos relativamente poucos subproblemas: um problema para cada opção de i e j que satisfaz a $1 \leq i \leq j \leq n$
- Vamos aplicar a Etapa 3 da PD
  - Calculando o custo ótimo usando uma abordagem tabular de baixo para cima (*bottom-up*)

Etapa 3: Como Calcular os Custos Ótimos

- Vamos estudar um pseudocódigo, considerando que M_i tem dimensões $p_{i-1} \times p_i$ para $i = 1, 2, ..., n$
- A entrada é uma sequência $p = \langle p_0, p_1, ..., p_n \rangle$, cujo tamanho é $(n + 1)$
- Usa-se uma matriz auxiliar m[1..n, 1..n] para armazenar os custos de m[i,j]
- Usa-se uma outra matriz auxiliar s[1..n,1..n] que registra qual índice de k atingiu um custo ótimo no cálculo de m[i,j]
  - s será usada para construir uma solução ótima

Etapa 3: Como Calcular os Custos Ótimos

```
n = comprimento[p] – 1
for i = 1 to n
    m[i,i] = 0 //custo para cadeias de tamanho 1
for l = 2 to n
    for i = 1 to (n – l) + 1
        j = i + l – 1
        m[i,j] = ∞
        for k = i to j – 1
            q = m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j
            if q < m[i,j] then
                m[i,j] = q
                s[i,j] = k
return m, s
```

Etapa 3: Como Calcular os Custos Ótimos

```
n = comprimento[p] – 1
for i = 1 to n
    m[i,i] = 0 //custo para cadeias de tamanho 1
for l = 2 to n
    for i = 1 to (n – l) + 1
        j = i + l – 1
        m[i,j] = ∞
        for k = i to j – 1
            q = m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j
            if q < m[i,j] then
                m[i,j] = q
                s[i,j] = k
return m, s
```

O custo m[i,j] depende apenas das entradas m[i,k] e m[k+1,j]

Etapa 3: Como Calcular os Custos Ótimos

- Vejamos um exemplo para n = 6
- Com os seguintes valores

Matriz	Dimensão
M_1	30 x 35
M_2	35 x 15
M_3	15 x 5
M_4	5 x 10
M_5	10 x 20
M_6	20 x 25

	1	2	3	4	5	6	
M_1	0	15750	7875	9375	11875	15125	1
M_2		0	2625	4375	7125	10500	2
M_3			0	750	2500	5375	3
M_4				0	1000	3500	4
M_5					0	5000	5
M_6						0	6

Matriz m

j

2	3	4	5	6	i
1	1	3	3	3	1
	2	3	3	3	2
		3	3	3	3
			4	5	4
				5	5

Matriz s

Etapa 3: Como Calcular os Custos Ótimos

j

s

i \ j	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

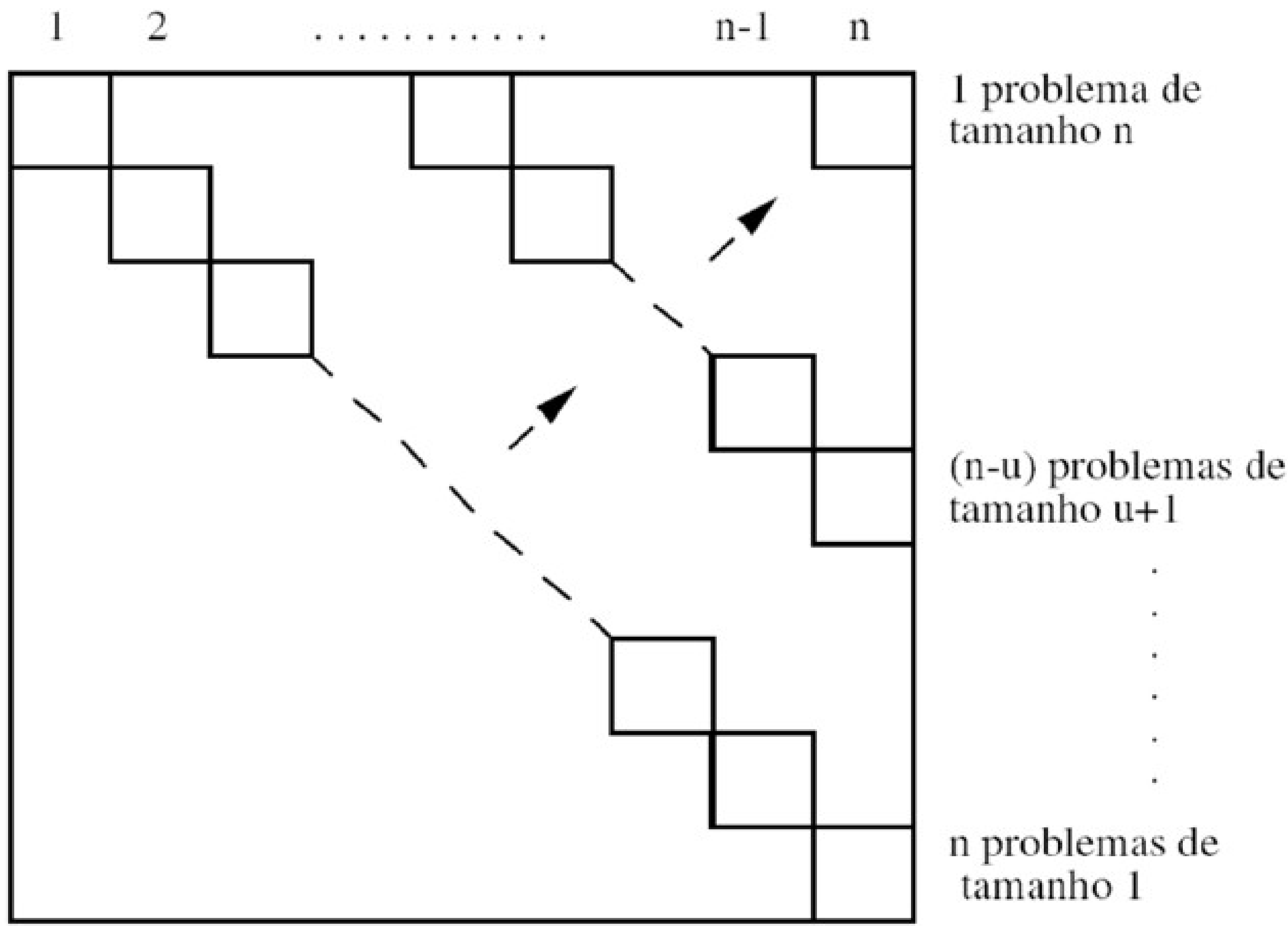
m

	i \ j	1	2	3	4	5	6
M_1	1	0	15750	7875	9375	11875	15125
M_2	2		0	2625	4375	7125	10500
M_3	3			0	750	2500	5375
M_4	4				0	1000	3500
M_5	5					0	5000
M_6	6						0

$$m[i,j] = \begin{cases} 0, \text{ se } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j\}, \text{ se } i < j \end{cases}$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1.p_2.p_5 = 0 + 2500 + 35.15.20 = 13000 \\ \mathbf{m[2,3] + m[4,5] + p_1.p_3.p_5 = 2625 + 1000 + 35.5.20 = 7125} \\ m[2,4] + m[5,5] + p_1.p_4.p_5 = 4375 + 0 + 35.10.20 = 11375 \end{cases}$$

Etapa 3: Como Calcular os Custos Ótimos



Etapa 4: Como Construir uma Solução Ótima

```
PRINT-OPTIMAL-PARENS(s, i, j)
    if i = j
        then print M_i
        else print "("
            PRINT-OPTIMAL-PARENS(s, i, s[i,j])
            PRINT-OPTIMAL-PARENS(s, s[i,j] + 1, j)
            print ")"
```

j **s**

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

i

Para o nosso exemplo, seria
$((M_1(M_2M_3))((M_4M_5)M_6)$

s[i,j] registra o valor de k tal que a colocação ótima dos parênteses divide o produto entre M_k e M_{k+1}

Exemplo
Complexidade Computacional

- Uma inspeção simples na estrutura de laços aninhados de MATRIX-ORDER fornece um tempo de execução $O(n^3)$
  - Os laços estão aninhados com profundidade 3
  - Cada índice (l, i e k) considera no máximo n valores

Sumário

- Motivação
- Fundamentos da Programação Dinâmica
- Exemplo de Utilização: Multiplicação de Matrizes
- Quando Aplicar Programação Dinâmica?
- Leitura Recomendada

Quando Aplicar PD?

- Aplicar em problemas que requerem muito tempo para serem resolvidos (em geral, de complexidade exponencial ou pior)
- Principais características
  - Princípio da Otimalidade (subestrutura ótima/subproblemas ótimos) → o valor ótimo global pode ser definido em termos dos valores ótimos dos subproblemas
  - Sobreposição (*overlap*) de Subproblemas → os subproblemas não são independentes
    - Existe um *overlap* entre eles (devem ser construídos *bottom-up*)

Padrão para Descoberta de uma Subestrutura Ótima

- Mostrar que uma solução para o problema consiste em fazer escolhas
  - Fizemos escolhas de índices das matrizes, por exemplo!
- Supor que, para um dado subproblema, você tem a escolha que conduz a uma solução ótima
- Dada a escolha, determinar quais subproblemas resultam dela e como caracterizar melhor o espaço de subproblemas resultante
- Mostrar que as soluções para os subproblemas usados dentro da solução ótima devem elas próprias serem ótimas

Observações Importantes

- O tempo de execução depende do produto de 2 fatores
  - Número de subproblemas globais
  - Quantidade de escolhas observadas para cada problema
  - Na multiplicação de matrizes, havia $\Theta(n^2)$ subproblemas globais e, em cada um deles, no máximo n-1 escolhas, resultando em $O(n^3)$

Observações Importantes

- PD emprega a subestrutura ótima *bottom-up*
  - Primeiro, encontramos soluções ótimas para subproblemas para encontrar, depois, a solução ótima para o problema
- Geralmente, o custo da solução do problema é igual ao custo dos subproblemas mais um custo diretamente atribuído à escolha em si
  - Por exemplo, na multiplicação de matrizes, determina-se a colocação ótima dos parênteses e depois escolhe-se a matriz M_k que divide o problema
    - O custo atribuído à escolha propriamente dita é $p_{i-1}.p_k.p_j$

Subproblemas Superpostos

- O espaço de subproblemas deve ser pequeno
  - Um algoritmo recursivo, neste caso, resolve os mesmos subproblemas repetidas vezes, ao invés de gerar novos subproblemas
- Quando um algoritmo examina o mesmo problema inúmeras vezes, afirma-se que o problema possui subproblemas superpostos
  - PD tira proveito disso! Resolve cada subproblema apenas uma vez e armazena a solução!!!

Aplicabilidade da PD

- Subsequência crescente mais longa
- Caminhos mínimos em grafos
- Caixeiro viajante
- Entre outros...

Referências

Material didático elaborado por Jorge Cesar Abrantes de Figueiredo, UFCG (Universidade Federal de Campina Grande)

Leitura Recomendada

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Algoritmos. Teoria e Prática. Tradução da Segunda Edição Americana.* Campus, 2002, pp. 259-280

N. Ziviani. *Projeto de Algoritmos com Implementações em Java e C++.* Thomson Learning, 2007, pp. 68-72