Programação Dinâmica

Por: Jogi Suda Neto

História

- É tanto um método matemático de otimização, quanto um método de programação.
- Inventado por Richard Bellman, em 1950.

História

- É tanto um método matemático de otimização, quanto um método de programação.
- Inventado por Richard Bellman, em 1950.
- Bellman trabalhava na época para a RAND (uma entidade que faz pesquisa para o departamento de defesa dos EUA) fazendo pesquisa. Bellman inventou o termo "Programação Dinâmica" para esconder o fato de que ele fazia pesquisa matemática na época.

História

- Conceitos do que conhecemos hoje por Prog. Dinâmica já haviam surgido antes, como Von Neumann e Morgenstein (1944 - seminário sobre Teoria dos Jogos).

História

- Conceitos do que conhecemos hoje por Prog. Dinâmica já haviam surgido antes, como Von Neumann e Morgenstein (1944 - seminário sobre Teoria dos Jogos).
- Apesar do nome complicado, basicamente é a resolução inteligente de alguns problemas recursivos com um uso de "memória" para armazenar subproblemas já resolvidos.

Sumário

- Conceitos iniciais
- Exemplos
    1. Fibonacci.
    2. Problema binário da mochila.
    3. Subsequência máxima comum.
- Resumo.

Quando usar?

- Utilizado em problemas que contenham soluções recursivas (ou seja, cada problema pode ser dividido em subproblemas).

Quando usar?

- Utilizado em problemas que contenham soluções recursivas (ou seja, cada problema pode ser dividido em subproblemas).
- Esses subproblemas contém **repetições.**

Quando usar?

- Utilizado em problemas que contenham soluções recursivas (ou seja, cada problema pode ser dividido em subproblemas).
- Esses subproblemas contém **repetições**.
- A solução ótima do problema é constituída por soluções ótimas de subproblemas.

Quando usar?

- Utilizado em problemas que contenham soluções recursivas (ou seja, cada problema pode ser dividido em subproblemas).
- Esses subproblemas contém **repetições**.
- A solução ótima do problema é constituída por soluções ótimas de subproblemas.
- Soluções devem ser independentes (evitar ciclos na solução - recursão infinita).

Fibonacci

- Solução recursiva.
- Memoização.
- Bottom-up.
- Complexidade.
- Análise de corretude.

Definição

$$Fibonacci(n) = \left\{ \begin{array}{cc} 1 & , se\ n = 1,2 \\ Fibonacci(n-1) + Fibonacci(n-2) & , se\ n > 2 \end{array} \right\}$$

Estratégia

- Solução recursiva.
- Podemos melhorar? Com quais estratégias?

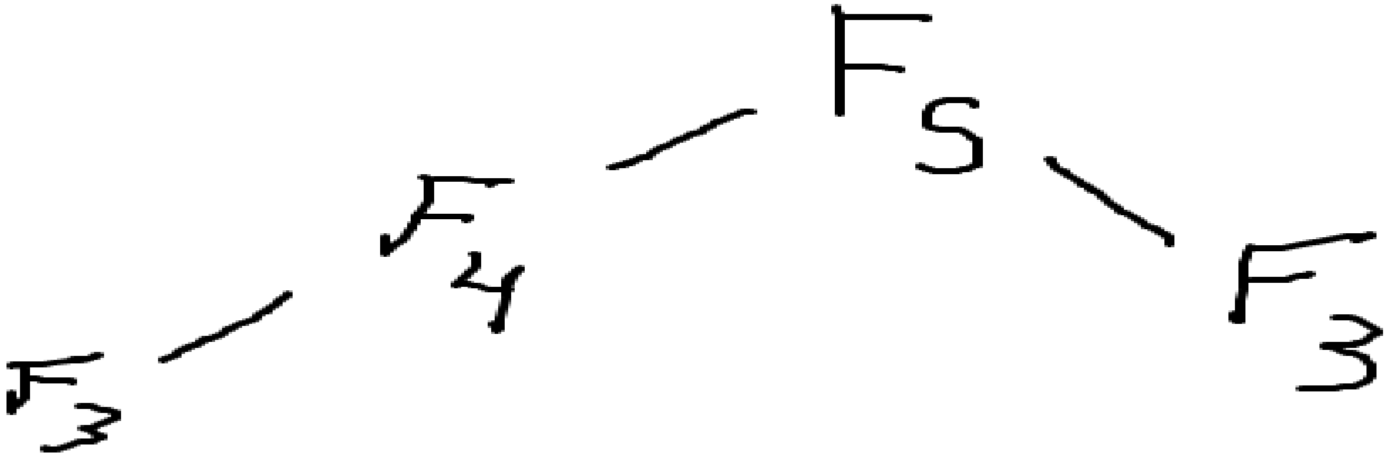
Solução recursiva (ingênua)

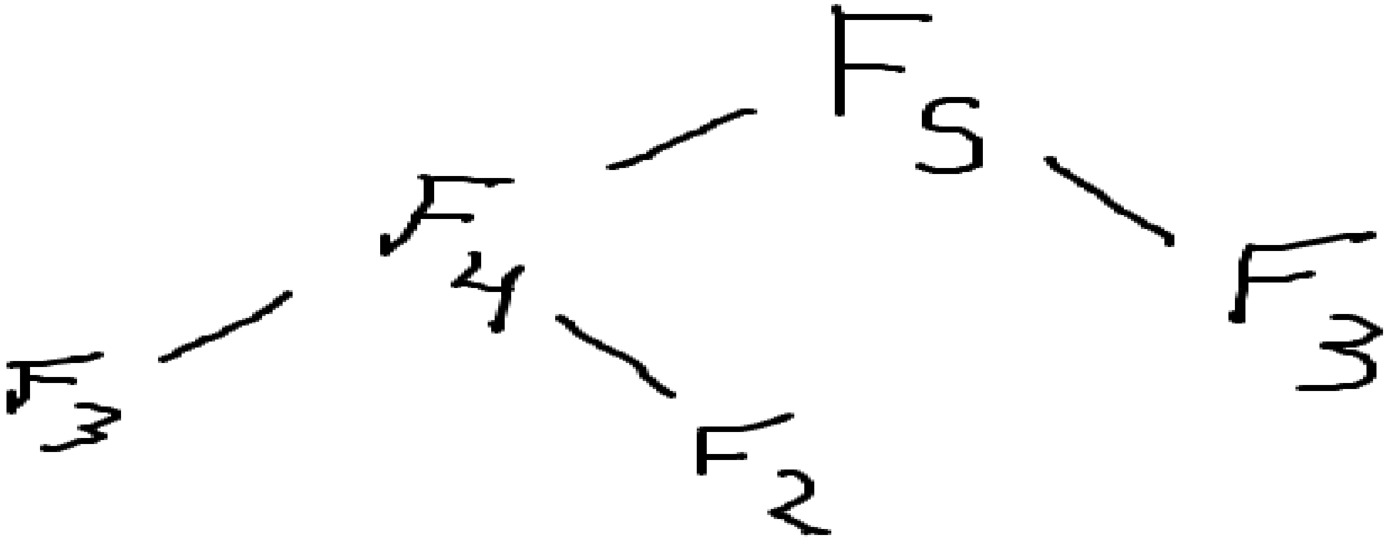
```
int fibonacci(n){
    if (n <= 2)
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
}
```

F_S

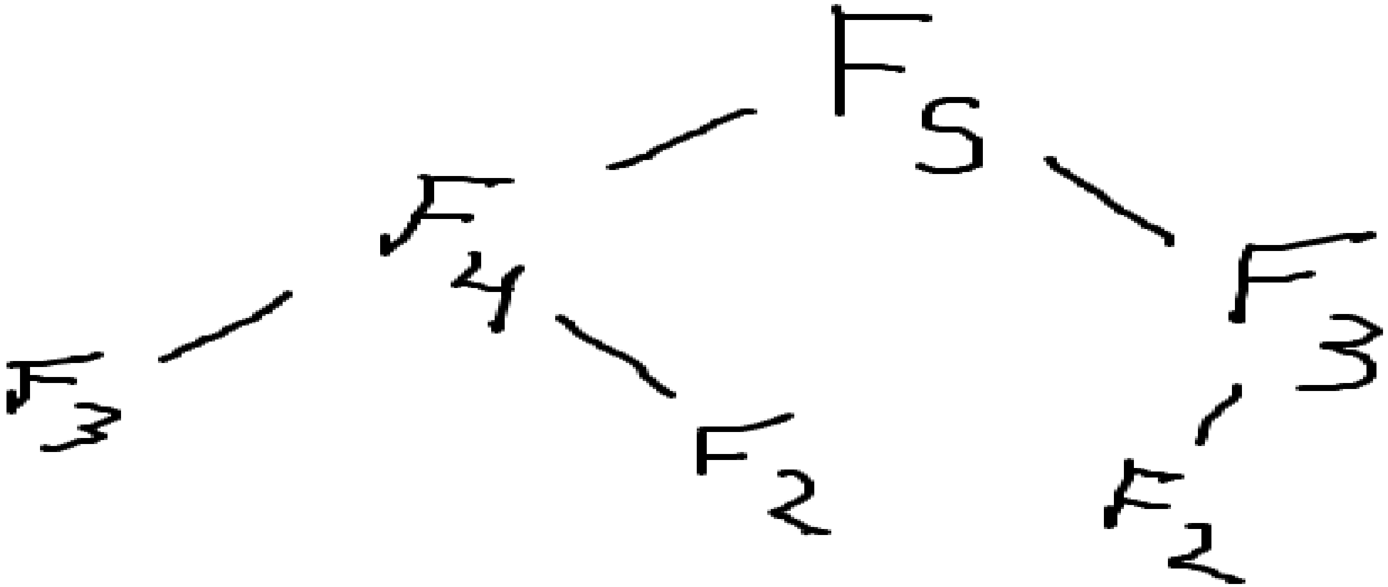
F_4 — F_5

$$\begin{tikzcd} & F_5 \arrow[dl, dash] \arrow[dr, dash] & \\ F_4 & & F_3 \end{tikzcd}$$

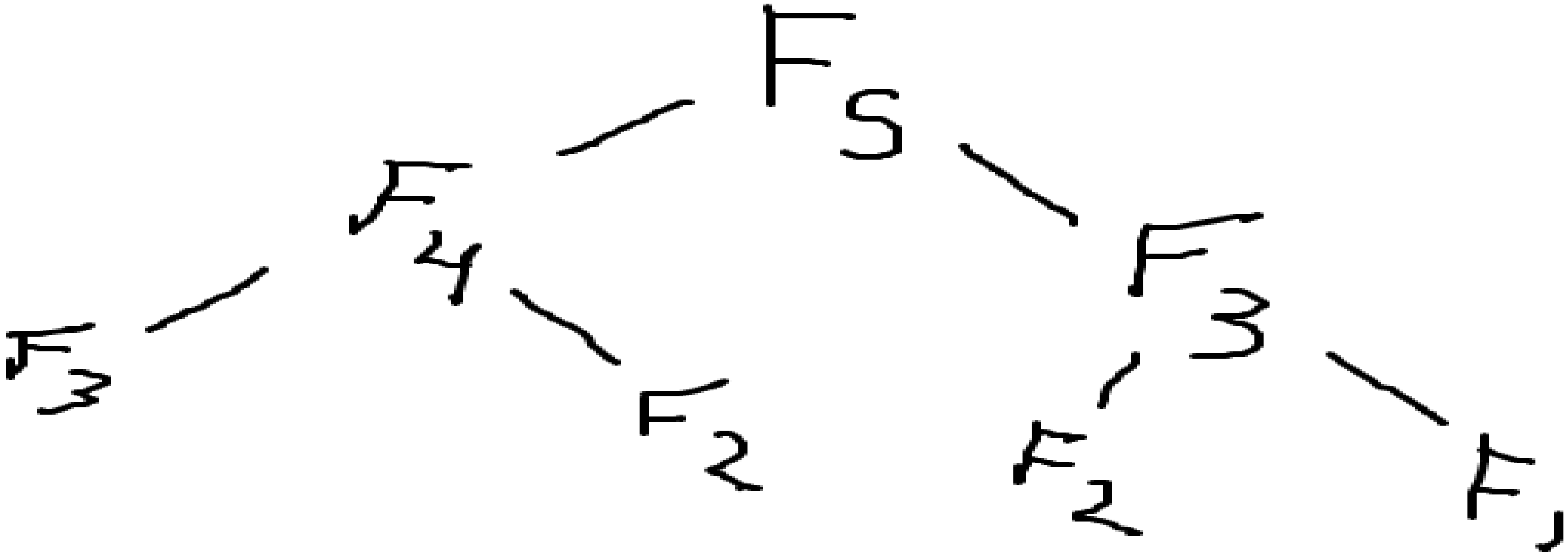

F_5
F_4
F_3
F_3

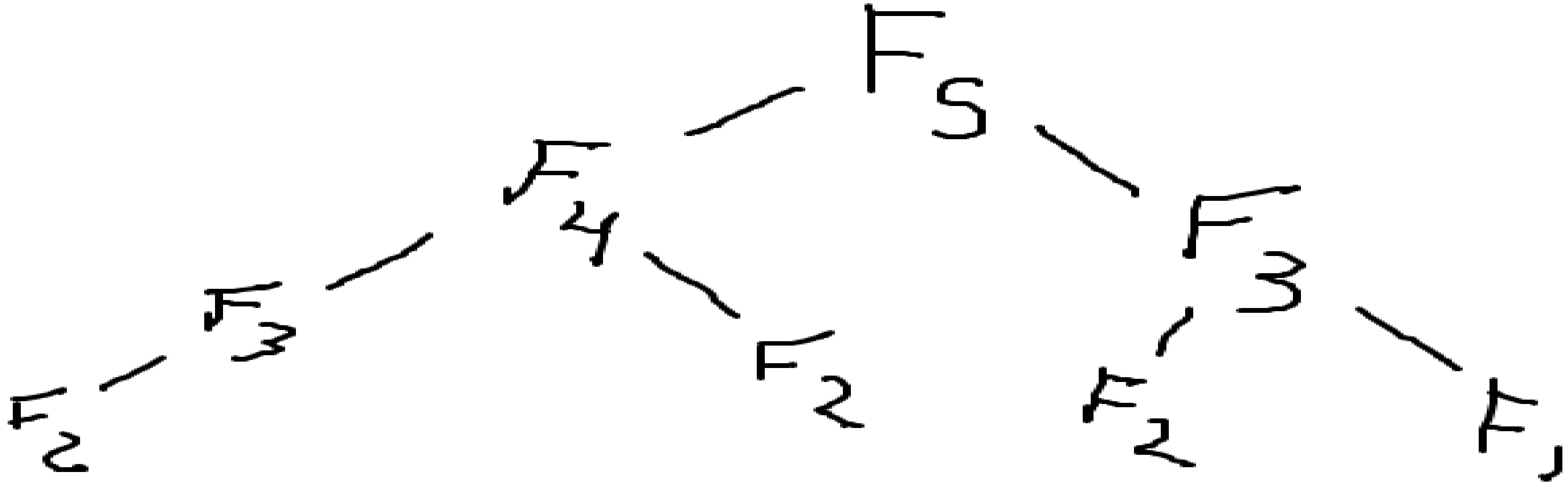


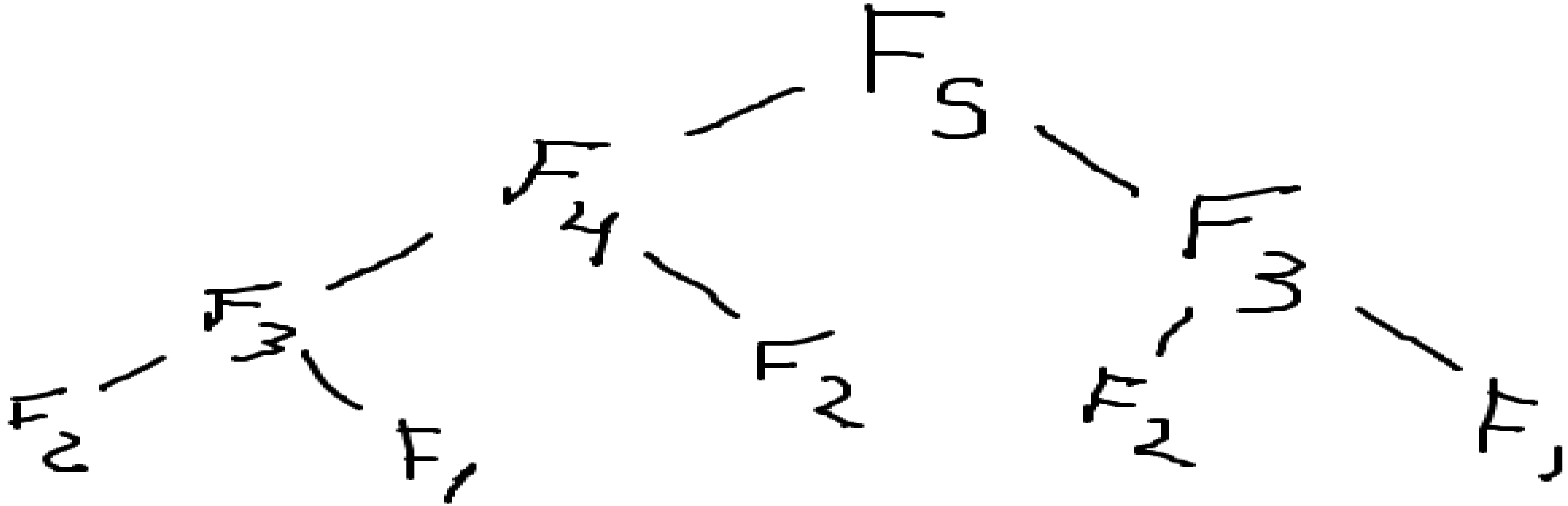
F_5
F_4
F_3
F_3
F_2



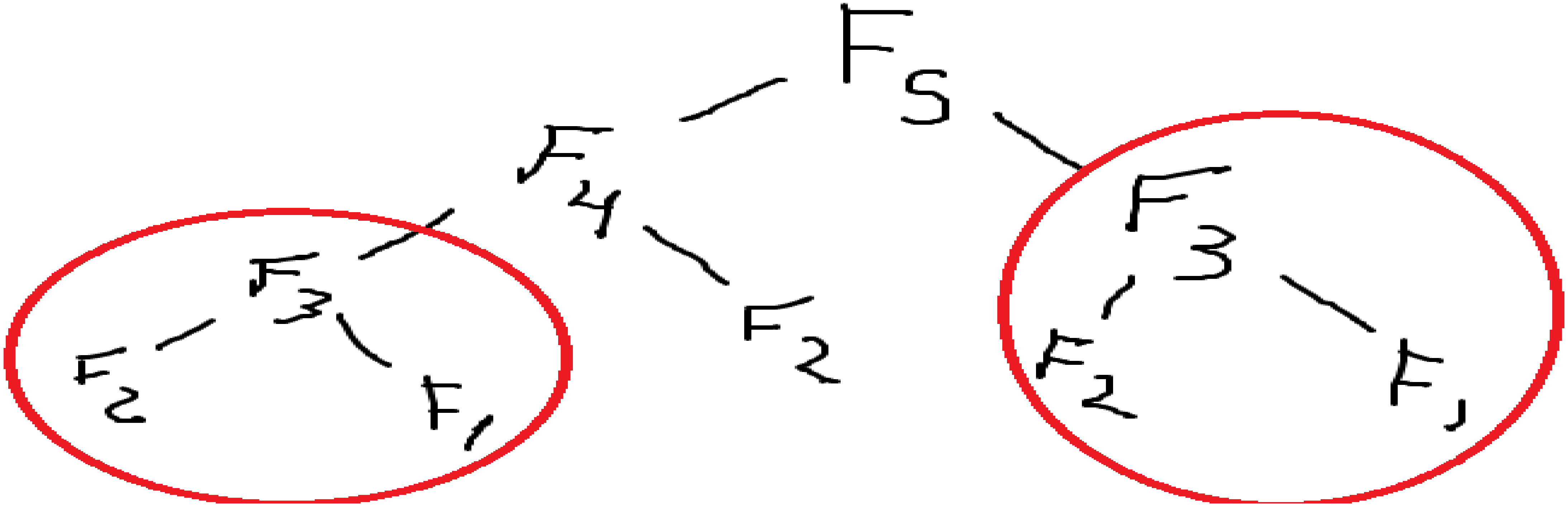
F_5
F_4
F_3
F_3
F_2
F_2


F5
F4
F3
F3
F2
F2
F1

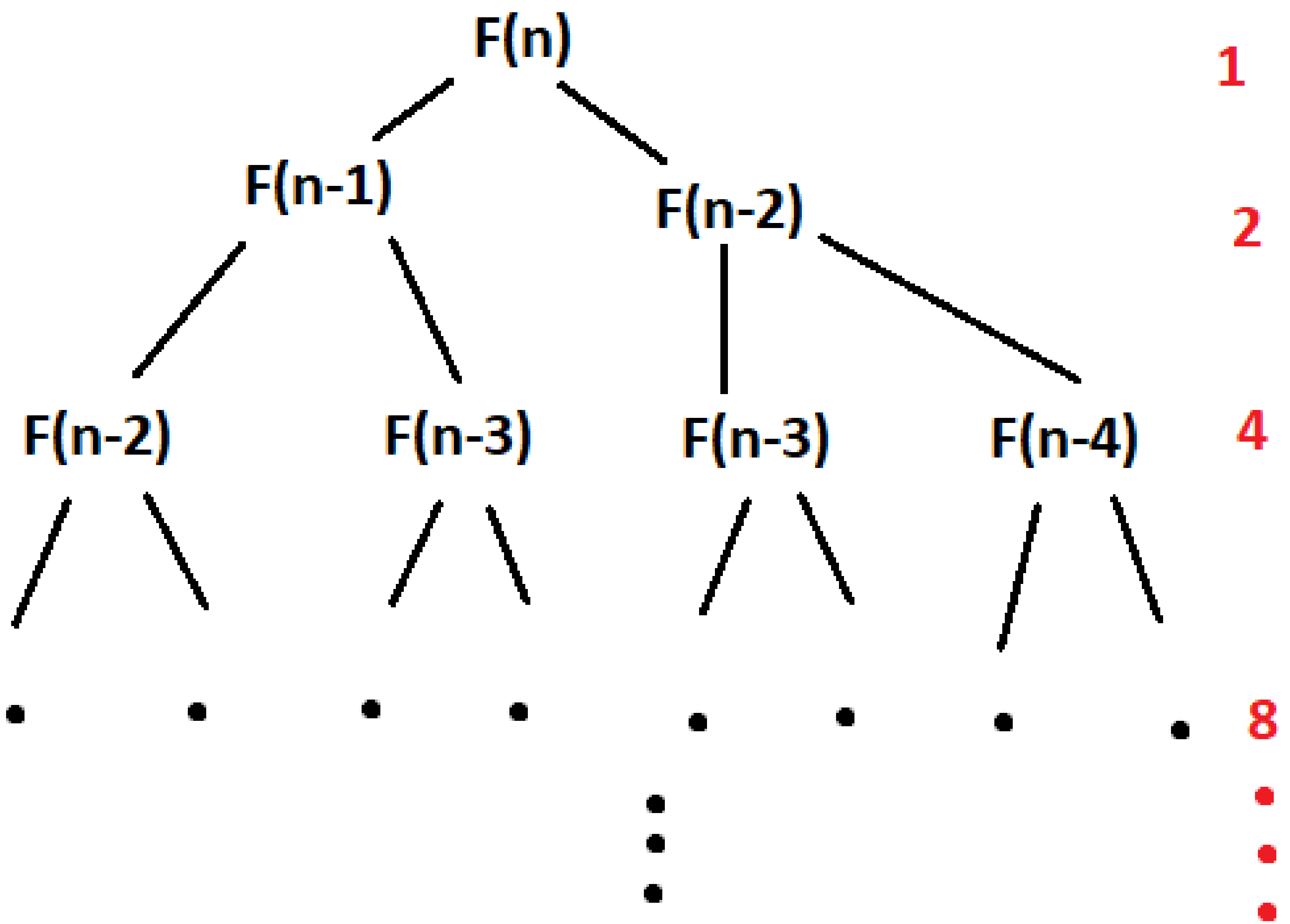

F5
F4
F3
F3
F2
F2
F2
F1



F_5
F_4
F_3
F_3
F_2
F_2
F_1
F_2
F_2
F_1



F5
F4
F3
F2
F1
F2
F3
F2
F1


F(n)
F(n-1)
F(n-2)
F(n-2)
F(n-3)
F(n-3)
F(n-4)
1
2
4
8

Estratégia 1 - memoização

- Basicamente, recursão com uso de uma "memória".

Estratégia 1 - memoização

- Basicamente, recursão com uso de uma "memória".
- Seguir os mesmo passos na construção de um algoritmo recursivo, apenas checando antes de tudo se o resultado que queremos calcular não está em nossa "memória" (vetor, matriz ou, de maneira geral, qualquer tensor).

Estratégia 1 - memoização

- Basicamente, recursão com uso de uma "memória".
- Seguir os mesmo passos na construção de um algoritmo recursivo, apenas checando antes de tudo se o resultado que queremos calcular não está em nossa "memória" (vetor, matriz ou, de maneira geral, qualquer tensor).
- Se estiver, retornamos o resultado, evitando recursões repetidas e, portanto, desnecessárias.

Estratégia 1 - memoização

- Basicamente, recursão com uso de uma "memória".
- Seguir os mesmo passos na construção de um algoritmo recursivo, apenas checando antes de tudo se o resultado que queremos calcular não está em nossa "memória" (vetor, matriz ou, de maneira geral, qualquer tensor).
- Se estiver, retornamos o resultado, evitando recursões repetidas e, portanto, desnecessárias.
- Caso não esteja, seguir normalmente com o algoritmo recursivo, com a diferença que ao final basta atualizar a "memória" para incluir o novo registro calculado.

Solução recursiva (ingênua)

```
int fibonacci(n){
    if (n <= 2)
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
}
```

Utilizando memoização

```
int memo[n+1] = indefinido

int fibonacci(n):
    if(memo[n] != indefinido)
        return memo[n]
    if(n <= 2)
        return 1
    memo[n] = fibonacci(n-1) + fibonacci(n-2)
    return memo[n]
```

complexidade

Na estratégia simples, cada F(n) chama recursivamente dois subproblemas, F(n-1) e F(n-2), sendo que por sua vez, cada uma dessas instâncias chamam recursivamente outras duas. Logo, é simples perceber que a complexidade é exponencial.

Para a versão de Programação Dinâmica, a partir dos casos-base, cada F(i), para i=1,2,3,4,5,..., n será calculado apenas uma vez. Logo temos complexidade O(n), fazendo apenas uma pequena alteração no código!

estratégia ii – bottom-up

Consiste em começar o problema dos subproblemas que são os casos base, e ir construindo a partir disso.

Estratégia iterativa (não utiliza recursão).

Vantagens em relação à memoização: em complexidade de tempo, nenhuma, mas em termos de memória pode ser interessante.

Utilizando bottom-up

```
int fibonacci(n):
    int memo[n+1]

    for(int i=1; i<=n; i++){
        if (i<=2) #casos base
            memo[i] = 1
        else
            memo[i] = memo[i-1] + memo[i-2]
    }
    return memo[n]
```

complexidade - bottom-up

Número de operações dentro do laço de repetição: O(1)

Número de vezes que o laço é iterado: O(n)

complexidade - bottom-up

Número de operações dentro do laço de repetição: O(1)

Número de vezes que o laço é iterado: O(n)

É um pouco mais rápido que a estratégia de memoizar por não depender de recursão, porém a complexidade de tempo é igual em Big O.

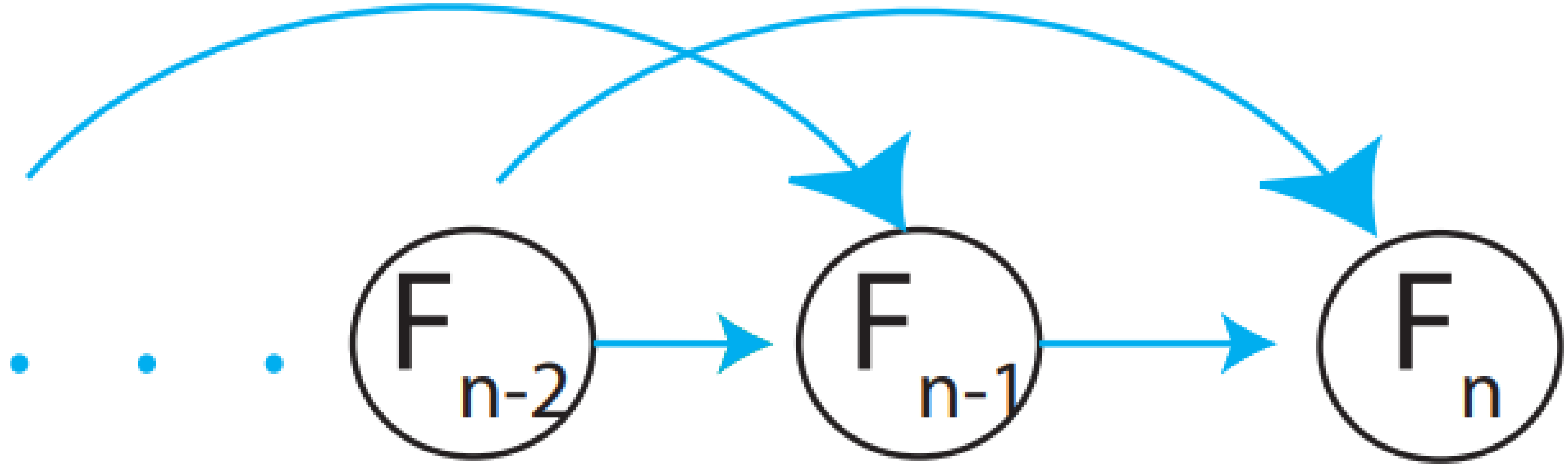
complexidade - bottom-up

Número de operações dentro do laço de repetição: O(1)

Número de vezes que o laço é iterado: O(n)

É um pouco mais rápido que a estratégia de memoizar por não depender de recursão, porém a complexidade de tempo é igual em Big O.

Aqui, podemos economizar complexidade de espaço (memória) em relação à memoização:



F n-2
F n-1
F n

Corretude

- Início: nas duas primeiras iterações, memo[i] conterá os casos base.

Corretude

- Início: nas duas primeiras iterações, memo[i] conterá os casos base.

- Manutenção: Supondo que o algoritmo esteja correto até a iteração i-1, na i-ésima iteração, memo[i] conterá Fibonacci(i).

Corretude

- Início: nas duas primeiras iterações, memo[i] conterá os casos base.

- Manutenção: Supondo que o algoritmo esteja correto até a iteração i-1, na i-ésima iteração, memo[i] conterá Fibonacci(i).

- Término: o laço pára quando i > n, retornando assim memo[n], ou seja, Fibonacci(n), e mantendo as invariantes. Logo, garantimos a corretude.

Corretude

- Sabemos que os casos base estão de acordo com a formulação (retornam o valor correto).

Corretude

- Sabemos que os casos base estão de acordo com a formulação (retornam o valor correto).
- Supomos que Fibonacci(n-1) esteja correto.

Corretude

- Sabemos que os casos base estão de acordo com a formulação (retornam o valor correto).
- Supomos que Fibonacci(n-1) esteja correto.
- Então, Fibonacci(n) depende de Fibonacci(n-1) e Fibonacci(n-2). Logo, Fibonacci(n) está correto!

Cuidados

- É importante apenas considerar que nosso problema não pode apresentar ciclos que levem a uma execução infinita do algoritmo de PD.

Problema binário da mochila

- Solução recursiva.
- Memoização.
- Bottom-up.
- Complexidades.
- Análise de corretude.

Problema binário da mochila - definição

- Temos uma mochila com capacidade C
- Além disso, temos uma lista de N objetos, cada um ocupa um espaço Wn, e tem um valor Vn.
- Qual o número máximo de elementos que conseguimos colocar na mochila de modo a maximizar o **valor total**? Qual é esse preço máximo?

Estratégia inicial

- Devemos considerar todas as 2^n possibilidades de subconjuntos de itens!

Estratégia inicial

- Devemos considerar todas as 2^n possibilidades de subconjuntos de itens!
- Para isto, vamos percorrer a lista toda de maneira recursiva.

Estratégia inicial

- Devemos considerar todas as 2^n possibilidades de subconjuntos de itens!
- Para isto, vamos percorrer a lista toda de maneira recursiva.
- Em cada instância para um item da lista, devemos perceber que existem apenas duas escolhas: colocar ou não o item da instância atual na mochila.

Estratégia inicial

- Devemos considerar todas as 2^n possibilidades de subconjuntos de itens!
- Para isto, vamos percorrer a lista toda de maneira recursiva.
- Em cada instância para um item da lista, devemos perceber que existem apenas duas escolhas: colocar ou não o item da instância atual na mochila.
- Devemos ficar atentos apenas caso o item não caiba na mochila!

Estratégia inicial

- Devemos considerar todas as 2^n possibilidades de subconjuntos de itens!
- Para isto, vamos percorrer a lista toda de maneira recursiva.
- Em cada instância para um item da lista, devemos perceber que existem apenas duas escolhas: colocar ou não o item da instância atual na mochila.
- Devemos ficar atentos apenas caso o item não caiba na mochila!
- Nesse caso, não há o que fazer, apenas passa a função recursivamente para o próximo item.

Casos-base

- Enquanto estivermos iterando a lista de objetos, temos duas possibilidades:

Casos-base

- Enquanto estivermos iterando a lista de objetos, temos duas possibilidades:
- 1- A mochila enche, então não há mais o que analisar, devemos apenas retornar o custo total do que há na mochila.

Casos-base

- Enquanto estivermos iterando a lista de objetos,
temos duas possibilidades:
- 1- A mochila enche, então não há mais o que analisar,
devemos apenas retornar o custo total do que há na mochila.
- 2- A mochila não enche, mas chegamos ao fim da lista de objetos.
Assim, retornaremos o custo total do que há na mochila também.

Formalmente

- Podemos expressar essas condições da seguinte maneira:

Formalmente

- Podemos expressar essas condições da seguinte maneira:

$$mochilaBin(n, C) = \begin{cases} 0, & se\ n = 0\ ou\ C = 0 \\ mochilaBin(n-1, C), & se\ W_n > C \\ max\{mochilaBin(n-1, C), V_n + mochilaBin(n-1, C - W_n)\}, & se\ W_n \leq C \end{cases}$$

Solução recursiva (ingênua)

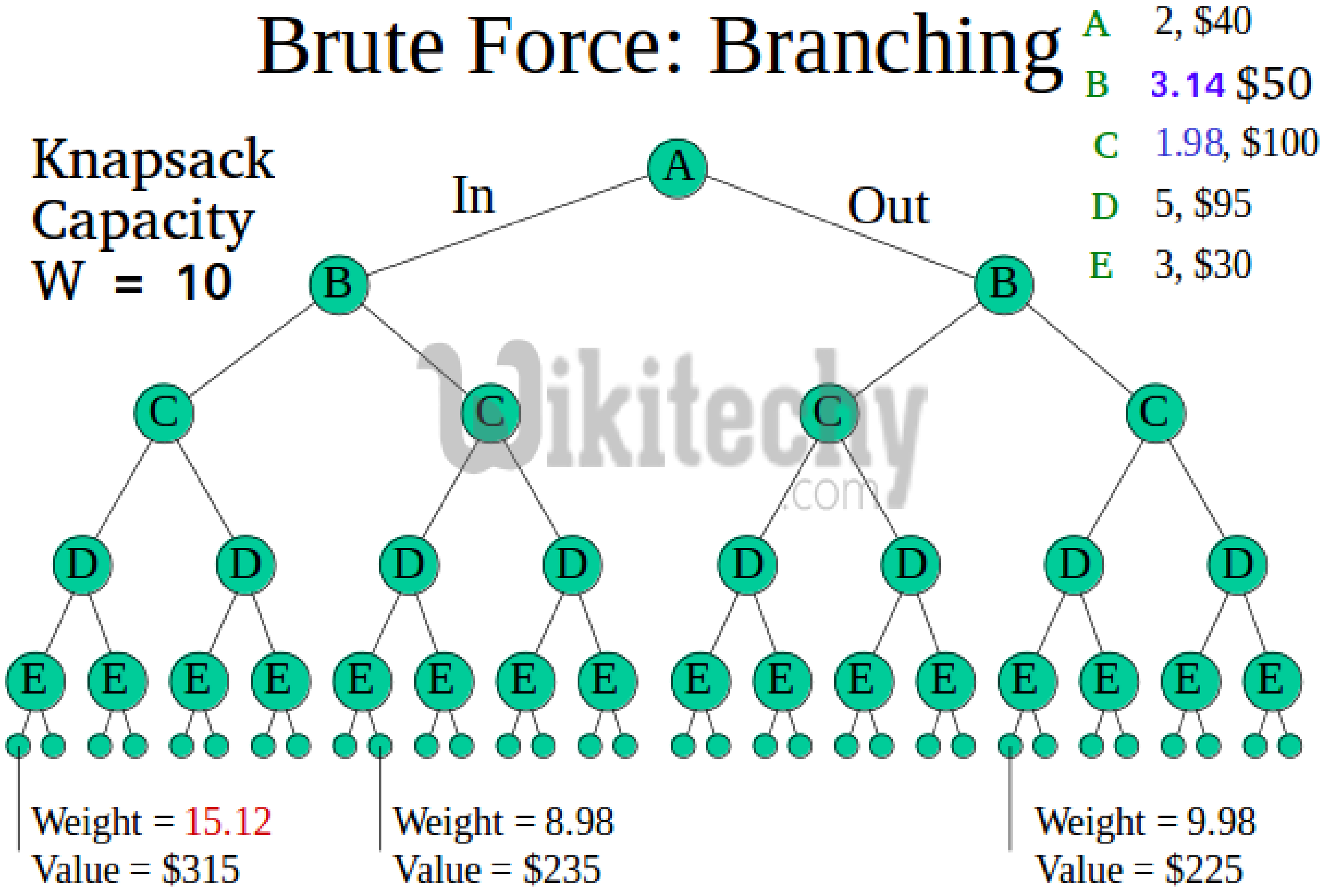
```
int mochilaBinaria(n, C){
    int tmp1
    int tmp2

    if (n == 0 || C == 0)
        result = 0
    else if (w[n] > C) #volume do objeto > capacidade atual da mochila
        result = mochilaBinaria(n-1, C)
    else{
        tmp1 = mochilaBinaria(n-1, C) #não incluindo obj n
        tmp2 = v[n]+mochilaBinaria(n-1, C - w[n]) #caso n seja incluído

        result = max(tmp1, tmp2)
    }

    return result
}
```

Árvore de recursão



Utilizando memoização

```
int memo[n][C] = indefinido
int mochilaBinaria(n, C){
    int tmp1
    int tmp2

    if (memo[n][C] != indefinido)
        return memo[n][C]
    else if (n == 0 || C == 0)
        result = 0
    else if (w[n] > C) #volume do objeto > capacidade atual da mochila
        result = mochilaBinaria(n-1, C)
    else{
        tmp1 = mochilaBinaria(n-1, C) #não incluindo obj n
        tmp2 = v[n]+mochilaBinaria(n-1, C - w[n]) #caso n seja incluído

        result = max(tmp1, tmp2)
    }
    memo[n][C] = result
    return result
}
```

Utilizando bottom-up

```
int mochilaBin(n, C, w, v):
    #memo[i][j] contém o valor máximo que pode ser obtido
    #para uma capacidade j considerando até o i-ésimo obj.
    int memo[n+1][C+1]

    for(c = 0 to C) memo[0, c] = 0 #inicializa casos base
    for(i = 0 to n) memo[i, 0] = 0 #outro caso base (fim da lista)
    for(i = 1 to n)
        for(c=0 to C){
            if(w[i] > c) #não cabe na mochila
                memo[i, c] = memo[i-1, c]
            else #caso caiba
                memo[i, c] = max(memo[i-1,c], v[i] + memo[i-1, c - w[i]])
        }
    return memo[n, C]
```

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

```
for w = 0 to W
        B[0,w] = 0
```

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
for i = 1 to n
        B[i,0] = 0
```

Utilizando bottom-up

```
int mochilaBin(n, C, w, v):
    #memo[i][j] contém o valor máximo que pode ser obtido
    #para uma capacidade j considerando até o i-ésimo obj.
    int memo[n+1][C+1]

    for(c = 0 to C) memo[0, c] = 0 #inicializa casos base
    for(i = 0 to n) memo[i, 0] = 0 #outro caso base (fim da lista)
    for(i = 1 to n)
        for(c=0 to C){
            if(w[i] > c) #não cabe na mochila
                memo[i, c] = memo[i-1, c]
            else #caso caiba
                memo[i, c] = max(memo[i-1,c], v[i] + memo[i-1, c - w[i]])
        }
    return memo[n, C]
```

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
        if b_i + B[i-1,w-w_i] > B[i-1,w]
            B[i,w] = b_i + B[i-1,w- w_i]
        else
            B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=2

w-w_i =0

```
if w_i <= w // item i can be part of the solution
        if b_i + B[i-1,w-w_i] > B[i-1,w]
            B[i,w] = b_i + B[i-1,w- w_i]
        else
            B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Items:

1: (2,3)
2: (3,4)
3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	**7**
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	**0**	**3**	**4**	**5**	

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Prova de Corretude

- Provaremos por indução. Vamos primeiro considerar o seguinte:

Prova de Corretude

- Provaremos por indução. Vamos primeiro considerar o seguinte:
- Dizemos que o par $(i, j) < (i', j')$ se $i < i'$ ou ($i = i'$ e $j < j'$). Por ex:

Prova de Corretude

- Provaremos por indução. Vamos primeiro considerar o seguinte:
- Dizemos que o par $(i, j) < (i', j')$ se $i < i'$ ou ($i = i'$ e $j < j'$). Por ex:
- $(0, 0) < (0, 1) < (0, 2) < \ldots < (1, 0) < (1, 1) < \ldots$

Prova de Corretude

- Provaremos por indução. Vamos primeiro considerar o seguinte:
- Dizemos que o par (i, j) < (i', j') se i < i' ou (i = i' e j < j'). Por ex:
- (0, 0) < (0, 1) < (0, 2) < ... < (1, 0) < (1, 1) < ...
- **Casos base:** trivialmente vemos que memo[i][0] = memo[0][j] = 0 para todo i, j.

Prova de Corretude

- Provaremos por indução. Vamos primeiro considerar o seguinte:
- Dizemos que o par (i, j) < (i', j') se i < i' ou (i = i' e j < j'). Por ex:
- (0, 0) < (0, 1) < (0, 2) < ... < (1, 0) < (1, 1) < ...
- **Casos base:** trivialmente vemos que memo[i][0] = memo[0][j] = 0 para todo i, j.
- **Hipótese indutiva:** vamos supor que todos os elementos memo[i][j] da tabela estão corretos para (i, j) < (i', j'), ou seja, todos os elementos anteriores estão corretos.

Prova de Corretude

- **Passo indutivo:** ao calcularmos memo[i', j'], temos que levar em conta memo[i'-1, j'] e, quando possível, memo[i'-1, j' - Wi']. Pela hipótese anterior, todos estes valores estão corretos, logo, memo[i', j'] também estará.

Complexidade

- Antes: **exponencial!**
- Com programação dinâmica:

Utilizando bottom-up

```
int mochilaBin(n, C, w, v):
    #memo[i][j] contém o valor máximo que pode ser obtido
    #para uma capacidade j considerando até o i-ésimo obj.
    int memo[n+1][C+1]

    for(c = 0 to C) memo[0, c] = 0 #inicializa casos base
    for(i = 0 to n) memo[i, 0] = 0 #outro caso base (fim da lista)
    for(i = 1 to n)
        for(c=0 to C){
            if(w[i] > c) #não cabe na mochila
                memo[i, c] = memo[i-1, c]
            else #caso caiba
                memo[i, c] = max(memo[i-1,c], v[i] + memo[i-1, c - w[i]])
        }
    return memo[n, C]
```

Complexidade

- Antes: **exponencial!**
- Com programação dinâmica: **O(nC)**!

Subsequência Comum Máxima

- Solução recursiva.
- Memoização.
- Bottom-up (opcional).
- Complexidades.
- Análise de corretude.

Subsequência Comum Máxima - definição

- Consideremos um alfabeto de caracteres $\Sigma = \{a, b, c, ..., z\}$.

Subsequência Comum Máxima - definição

- Consideremos um alfabeto de caracteres $\Sigma = \{a, b, c, ..., z\}$.
- Vamos considerar X como uma sequência de elementos de nosso alfabeto. Ex: X = adfyz

Subsequência Comum Máxima - definição

- Consideremos um alfabeto de caracteres $\Sigma = \{a, b, c, ..., z\}$.
- Vamos considerar X como uma sequência de elementos de nosso alfabeto. Ex: X = adfyz
- Também, vamos considerar a seguinte notação:

Subsequência Comum Máxima - definição

- Consideremos um alfabeto de caracteres $\Sigma = \{a, b, c, ..., z\}$.
- Vamos considerar X como uma sequência de elementos de nosso alfabeto. Ex: X = adfyz
- Também, vamos considerar a seguinte notação:
- X_1 = a
- X_2 = ad
- X_3 = adf
- ... X_5 = adfyz

Subsequência Comum Máxima - definição

- Para uma sequência X de nosso alfabeto, temos uma **subsequência** de X qualquer sequência contida em X que mantenha a mesma **ordem relativa** dos elementos, porém, não necessariamente de maneira **contígua.**

Subsequência Comum Máxima - definição

- Para uma sequência X de nosso alfabeto, temos uma **subsequência** de X qualquer sequência contida em X que mantenha a mesma **ordem relativa** dos elementos, porém, não necessariamente de maneira **contígua.**

- **X = adfyz**
- **Y = adfuvwhgj**

Subsequência Comum Máxima - definição

- Para uma sequência X de nosso alfabeto, temos uma **subsequência** de X qualquer sequência contida em X que mantenha a mesma **ordem relativa** dos elementos, porém, não necessariamente de maneira **contígua.**

- **X = adfyz**
- **Y = ajkldmnof**

Subsequência Comum Máxima - definição

- Para uma sequência X de nosso alfabeto, temos uma **subsequência** de X qualquer sequência contida em X que mantenha a mesma **ordem relativa** dos elementos, porém, não necessariamente de maneira **contígua.**

- **X = adfyz**
- **Y = fjkldmnoa**

Subsequência Comum Máxima - definição

- Para uma sequência X de nosso alfabeto, temos uma **subsequência** de X qualquer sequência contida em X que mantenha a mesma **ordem relativa** dos elementos, porém, não necessariamente de maneira **contígua.**

- **X = adfyz**
- **Y = utbadfuvwhgj**

Estratégia inicial

- **X = ……………………… a**
- **Y = ……………………… a**

Estratégia inicial

- **X = ... a**
- **Y = ... a**

- **Se X[n] = Y[m]:**

Estratégia inicial

- **X = ... a**
- **Y = ... a**

- **Se X[n] = Y[m]:**
- **Resultado = 1 + LCS(X, Y, n-1, m-1)**

Estratégia inicial

- **X = …………………… a**
- **Y = …………………… b**

Estratégia inicial

- **X = …………………….. a**
- **Y = …………………….. b**

Estratégia inicial

- **X = …………………… a**
- **Y = …………………… b**

Estratégia inicial

- **X = ... a**
- **Y = ... b**

- **Se X[n] != Y[m]:**

Estratégia inicial

- **X = ... a**
- **Y = ... b**

- **Se X[n] != Y[m]:**
- **tmp1 = LCS(X, Y, n-1, m)**

Estratégia inicial

- **X = ... a**
- **Y = ... b**

- **Se X[n] != Y[m]:**
- **tmp1 = LCS(X, Y, n-1, m)**
- **tmp2 = LCS(X, Y, n, m-1)**

Estratégia inicial

- **X = ... a**
- **Y = ... b**

- **Se X[n] != Y[m]:**
- **tmp1 = LCS(X, Y, n-1, m)**
- **tmp2 = LCS(X, Y, n, m-1)**
- **Resultado = max(tmp1, tmp2)**

Formalmente

$$LCS(X,Y,n,m) = \left\{ \begin{array}{lr} 0, & se\, n = 0\, ou\, m = 0 \\ 1 + LCS(X,Y,n-1,m-1), & se\, X_n = Y_m \\ max\{LCS(X,Y,n-1,m), LCS(X,Y,n,m-1)\}, & se\, X_n != Y_m \end{array} \right\}$$

```
int LCS(X, Y, n, m){
    int result = 0

    if (n == 0 || m ==0) #fim de uma das cadeias
        return 0
    else if(X[n-1] == Y[m-1]) #último carac. igual
        result = 1 + LCS(X, Y, n-1, m-1)
    else
        result = max(LCS(X, Y, n-1, m), LCS(X, Y, n, m-1)) #considerar todas possibilidades
    return result
}
```

Complexidade

- Versão simples: **exponencial.**

```
int LCS(X, Y, n, m){
    int result = 0

    if (memo[n][m] != indefinido)
        return memo[n][m]
    if (n == 0 || m ==0) #fim de uma das cadeias
        return 0
    else if(X[n-1] == Y[m-1]) #último carac. igual
        result = 1 + LCS(X, Y, n-1, m-1)
    else
        result = max(LCS(X, Y, n-1, m), LCS(X, Y, n, m-1)) #considerar todas possibilidades
    memo[n][m] = result
    return result
}
```

```
int LCS(X, Y, n, m){
    int L[n+1][m+1]

    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= m; j++){
            if(i == 0 || j == 0) #casos base
                L[i][j] = 0
            else if(X[i] == Y[j])
                L[i][j] = 1 + L[i-1][j-1]
            else
                L[i][j] = max(L[i-1][j], L[i][j-1])
        }

        return L[n][m]
}
```

	-	A	G	A	C	T	G	T	C
-	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	1	1	1	1
A	0	1	1	1	1	1	1	1	1
G	0	1	2	2	2	2	2	2	2
T	0	1	2	2	2	3	3	3	3
C	0	1	2	2	3	3	3	3	4
A	0	1	2	3	3	3	3	3	4
C	0	1	2	3	4	4	4	4	4
G	0	1	2	3	4	4	5	5	5

Complexidade

- Versão simples: **exponencial.**
- Com programação dinâmica: **O(mn).**

Referências

- [1] Introduction to Algorithms - Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford.
- [2] Dynamic programming: Princeton University press – Bellman, R.