>> Programação Dinâmica

Fib(n)= {Fib(n-1) + Fib(n-2) , n > 1
{ 1 , n = 1
{ 0 , n = 0

SI(n) = {SI(n-1) + SI(n-2) + 2 , n > 1
{1 , n = 1
{1 , n = 0

SI(n) = Ω(2^(n/2))

*Vimos que em alguns casos de Fib, executamos funções que já havíamos executado antes, desperdiçando tempo.
Para solucionar isso vamos trabalhar com a Tabela de Soluções

---

>> Tabela de Soluções

*Estrutura de mapeamento entre sub-instancias e suas soluções.

Nesse caso, vamos usar um vetor como tabela de soluções para armazenar
todas as solucões que formos encontrando para cada sub-instancia.

OBS.: As tabelas precisam representar um mapeamento e nos permitir
inserir novos mapeamentos e diferenciar entre mapeamentos existentes/inexistentes.

Algoritmo: FibMemo(n, T)
Entrada: natural n, tabela de soluções T
Saída: valor do n-esimo termo de Fibonacci.

```
se n = 0 ou n = 1 entao:
|  retorne n
se T[n – 1] = -1  entao:
|  T[n – 1] ← FibMemo(n – 1,T)
Se T[n-2] = -1 entao:
|  T[n – 2] ← FibMemo(n – 2,T)
retorne T[n – 1]+T[n – 2]
```

*Neste caso fomos preenchendo durante a execução do algoritmo.
Agora, vamos calcular tudo antes.

Algoritmo: FibPD(n)
Entrada: natural n
Saída: valor do n-esimo termo de Fibonacci.

T ← NovoVetor(n + 1)
T[0] ← 0
T[1] ← 1
para i ← 2,3,4...,n faca:
| T[i] ← T[i – 1]+T[i – 2]
retorne T[n]

*Chamamos essa técnica de BottomUp.

>> Problemas de Otimização

Problemas computacionais que descrevem as soluções desejadas através de propriedades estruturais e métricas de comparação entre as soluções.
Podemos imaginar duas etapas:

- Quais regras descrevem o conjunto de estruturas que estamos interessados.
- Quais regras determinam como comparar essas estruturas.

Estamos interessados na "melhor" estrutura.

> Temos dois conceitos de solução:

- Solução viável: todas as estruturas que respeitem as regras estruturais definidas no problema.
- Solução ótima: dentre as soluções viáveis, aquelas que as métricas de comparação consideram as melhores soluções.

Sub Estrutura Ótima
-Soluções ótimas incorporam soluções ótimas para sub instancias relacionadas.
-Somos capazes de definir como encontrar uma solução ótima através das soluções ótimas de sub instancias.

Problemas de otimização que apresentam sub estrutura ótima aceitam soluções de divisão e conquista e programação dinâmica.

Corte de Hastes

tamanho:	1	2	3	4	5
valor:	5	7	10	3	8

-Imaginando uma haste, qual o melhor modo de cortá-la? (Sendo cada índice correspondente a um tamanho)

Teorema: O problema corte de hastes apresenta subestrutura ótima.

Demonstração: Supondo uma instancia do problema com tabela de preços P e tamanho de haste n.

Seja S* uma solução ótima para essa instancia e S*[j] o tamanho do j-ésimo pedaço em S*.

Seja S' a subsequencia obtida de S* excluindo-se o primeiro pedaço. Perceba que S' é uma solução viável para a instância <P, n-S*[1]>. Por absurdo, suponha que S' não seja ótima.

Assim, existe outra solução S para <P, n - S*[1]> que é melhor que S'.

Imaginando que P(S'), P(S) e P(S*) representam os valores das respectivas soluções, temos que:

P(S) > P(S') (I)

Observe que S*[1] * S também é uma solução viável para <P,n> e que S*[1] * S' = S*.

Dai temos

P(S*[1] * S) <= P(S*) (II)

Porém,

P(S*[1] * S) = P[S*[1]] + P(S)

P(S*) = P[S*[1]] + P(S')

Por (I), temos que

P(S*[1] * S) = P[S*[1]] + P(S) > P[S*[1]] + P(S') = P(S*)

Um absurdo contra (II). Logo, S' é ótima.

´

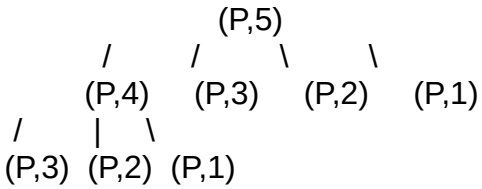
Exemplo de um vetor:

1	{ P[1] + P(<n-1>)
2	{ P[2] + P(<n-2>)
k	{ P[k] + P(<n-k>)
n-1	{ P[n-1] + P(<n-(n-1)>)
n	{ P[n] + P(<n-n>)

Algoritimo: CorteHastesDC(P, n)
Entrada: Tabela de preços P, tamanho n
Saída: valor do ganho máximo ao vender uma haste de tamanho n em pedaços
Requisitos: |P| >= n, n > 0

```
se n=1 entao:
|  retorne P[1]
maximo ← P[n]
para i ← 1,...,n-1 faca:
|  ganho ← P[i] + CorteHastesDC(P,n - i)
|  Se ganho > maximo entao:
|  |  maximo ← ganho
retorne maximo
```

Arvore de Recorrência



Vemos que temos valores de (P,x) se repetindo. Podemos usar Tabela de Solução!

Novo Algoritmo (Com Tabela de Soluções):

```
CorteHastesMemo(P, n, T)
se n=1 entao:
|  retorne P[1]
maximo ← P[n]
para i <- 1,...,n-1 faca:
|  se T[n – i] = y então:
|  |  T[n – i] ← CorteHastesMemo(P, n – i, T)
|  ganho ← P[i] + T[n – i]
|  Se ganho > maximo entao:
|  |  maximo ← ganho
T[n] ← maximo
retorne T[n]
```

Novo Algoritmo (Com programação dinâmica):

```
CorteHastesPD(P, n)
T ← NovoVetor(n)
T[1] <- P[1]
para j ← 2,...,n faca:
|  maximo ← P[j]
|  para i ← 1,2,...,j-1 faca:
|  |  ganho ← P[i] + T[j – i]
|  |  Se ganho > maximo entao:
|  |  |  maximo ← ganho
|  T[j] ← maximo
retorne T[n]
```

>> Problema: Dada uma sequencia X, com |x| = n, determinar uma subsequencia y<=x crescente de tamanho máximo.

Definição: Dada uma sequencia

$x = <x_1, x_2, x_3, \ldots, x_n>$,

Uma sequencia $y = <y_1, y_2, \ldots, y_k>$ é dita subsequencia de x, denotado por y contido em X. Se y pode ser obtida a partir de x pela remoção de zero ou mais elementos, porem sem alterar a ordem relativa dos elementos restantes.

Exemplo:

* x = < 5, 7, 10, 8, 5, 4 >
  y = < 7, 10, 5> ----------> (Y contido em X)

* x = < 5, 7, 10, 8, 5, 4 >
  y = < 5, 5, 7 > ----------> (Y não esta contido em x)

Etapas para PD

1. Caracterizar a estrutura de uma solução ótima.
   - Imaginar como se estruturar os dados de uma solução ótima.
2, Definir recursivamente o valor de uma solução ótima.
   - Aqui deve ocorrer subestrutura ótima.
3. Computar o valor de uma solução ótima algoritmicamente
   - D&C -> Memorização -> PD
4. Construir uma solução ótima a partir de informações computadas sobre as decisões tomadas pelo algoritmo anterior.
   - Qual a menor escolha tomada para cada sub-instancia?

Exemplo de Instancia

x = < 10, 8, 5, 15, 9, 19>

possibilidade 1: x_1 está na solução

$x^1 = <15,19>$

possibilidade 2: x_1 não esta na solução

$x^2 = <8,5,15,9,19>$

MSC(X,i) = Tamanho da maior subsequencia crescente em X[i...|x|]

MSC(X, i) = { 1 , se i = |x|
{ max(1 + MSC(y, 1), MSC(x, i + 1))

OBS.: y é a subsequencia de x compatível com o elemento xi.

Exercício: Demonstração de subestrutura ótima do MSC.
Algoritmo: MSC-DC (X, i)
Entrada: Sequencia X, índice i
Saída: Valor de MSC(X,i)

```
se i > |x| entao:
| retorne 0
se i = |x| entao:
| retorne 1
sem_i ← MSC-DC(X, i + 1)
y ← NovaSequencia()
para cada e pertencente a X[i+1...|x|] faca:
| se e > X[i] entao:
| | y ← y U <e>
com_i ← 1 + MSC-DC(y, 1)
retorne max(sem_i, com_i)
```

```
Algoritmo MSC-Memo(X, i, T)
Se i > |x| entao:
| retorne 0
se i = |x| ent o:
| retorne 1
se T[x[i+1...|x|]] ← MSC-Memo(x,i + 1)
sem_i ← T[x[i+1...|x|]]
y <- NovaSequencia()
para cada e pertencente a X[i+1...|x|] faca:
| se e > X[i] entao:
| | y ← y U <e>
se T[y] = vazio entao:
| T[y] <- MSC-Memo(y, 1)
com_i <- T[y]
retorne max(sem_i, com_i)
```

$x = \langle x_1,x_2,x_3,\ldots,x_n \rangle$

```
   |      |
   |      '----.
   |           \
```

$y = \langle y_1,y_2,\ldots,y_i,x_3 \rangle$

Vamos verificar a subsequencia de cada elemento considerando o elemento como o ultimo elemento da subsequencia.

Vamos raciocinar sob a ótica de determinar uma maior subsequencia crescente que termina em um dado elemento.

```
1 2 3 4 5 6 7 8 9          --índices
x = <1,3,2,0,4,7,8,5,10>   --chaves
1 2 2 1 3                  --maior subsequencia de cada índice salva em outro vetor
```

Com Tabela de Soluções

```
Algoritmo: MSC-Memo(X, i, T)
Entrada: Sequencia numérica x, índice i, tabela de solução T
Saída: MSC de x que obrigatoriamente termina com x.

max ← 0
para j ←1,2,...,i – 1 faca:
| se x[j] < x[i] faca:
| | se T[j] = vazio entao:
| | | T[j] ← MSCiMemo(X,j)
| | se T[j] > max entao:
| | | max ← T[j]
retorne 1 + max
```

Com Programação Dinâmica

```
Algoritmo: MSC-PD(x)
Saída: Tamanho de uma maior subsequencia crescente de x

T ← NovoVetor()
T[1] ← 1
sol ← 1
para i ← 2,3,...,|x| faca:
| max ← 0
| para j ← 1,2,3,...,i-1 faca:
| | se x[j] < x[i] e t[j]>max
| | max ← t[j]
| T[i] ← 1 + max
| se T[i] > sol entao:
| | sol ← T[i]
retorne sol
```

Agora, retornando também a tabela de decisões

```
Algoritmo: MSC-PD(x)
Saída: Tamanho de uma maior subsequencia crescente de x, tabela de decisões S

T ← NovoVetor()
T[1] ← 1
S[1] ← 0
para i ← 2,3,...,|x| faca:
| S[i] ← 0
| max ← 0
| para j ← 1,2,3,...,i – 1 faca:
| | se x[j] < x[i] e t[j]>max
| | max ← T[j]
| | S[i] ← j
| T[i] ← 1+max
retorne max(T), S, k            – k é o índice do final da maior sequencia
```

Algoritmo Final

Algoritmo: MSC-Sol(x)
Entrada: (...)
Saída: Uma maior subsequencia crescente em x

sol, S, k ← MSC(x)
L <- NovaLista()
faça
| InserirInicio(L, x[k])
| k <- S[k]
enquanto k>0
retorne L

Portanto, e melhor seguir sempre essa sequencia no desenvolvimento do algoritmo, para que possamos desenvolver a melhor solução possível com mais precisão e melhor compreensão. Complexidades: Programação Dinâmica = $\Theta(n^2)$, Força Bruta = $\Theta(2^n)$

>> Problema: Dada uma cadeia de matrizes $<A_1, A_2, A_3, \ldots, A_n>$, em que para i = [1, 2, …, n], A_i tem dimensões $[p_{i-1} \times p_i]$, determine os parênteses do produto $A_1 * A_2 * A_3 * \ldots * A_n$ para minimizar a quantidade de multiplicações escalares.

Sendo A, B e C matrizes

$(A_{[3x2]} * B_{[2x4]}) * C_{[4x5]}$ -> 3*2*4 + 3*4*5 = 84
$A_{[3x2]} * (B_{[2x4]} * C_{[4x5]})$ -> 2*4*5 + 3*2*5 = 70

onde: $M_{[i \times j]} * N_{[j \times k]}$ -> total de produtos entre escalares = i * j * k

Sendo D um vetor e $<A_1, A_2, \ldots, A_n>$ matrizes :
-D[0] = dim. de linhas de A_1
-D[i] = dim. de colunas da matriz A_i

* Para $<A_{[3x2]}, B_{[2x4]}, C_{[4x5]}>$ temos:
D[0] = 3 D[1] = 2
D[2] = 4 D[3] = 5

Dependendo da ordem em que realizamos os produtos em uma sequencia de multiplicação de matrizes, podemos realizar mais ou menos produtos entre escalares.

Dadas as dimensões das matrizes que queremos calcular, determinar em que ordem devemos multiplicá-las para obtermos o menor número de produtos entre escalares.

$M = A_1 * A_2 * A_3 * A_4 * A_5 * \ldots * A_n$

Vamos receber como entrada um vetor de dimensões D de forma que, para o produto acima:
-D[0] = dim. de linhas de A_1
-D[i] = dim. de colunas da matriz A_i
-(|D| = n + 1)

O conjunto de soluções viáveis para esse problema é o conjunto das permutações da sequencia <p1, p2,...,pn-1>

supondo o conjunto abaixo que é uma solução ótima
S* = <s1,s2,s3,...,sn-1)

Teorema: MCM possui subestrutura ótima.
Demonstração: Suponha que S* é solução ótima para o produto $A_1 * A_2 * A_3 * A_4 * ... * A_n$ com vetor de dimensões D.

Perceba que se reorganizarmos S* na solução S* de forma que os produtos que dizem respeito as matrizes $A_1...A_{n-1}$ ocorrem a partir do início da mesma ordem relativa que em S*, os demais seguindo esses, temos que S* também é ótima.

Seja i o índice em S* que marca o ultimo produto com relação as matries $A_1,...,A_{sn-1}$

```
S* = <s1,s2,...,si,...,sn-1,sn>
     \         / \        /
      \       /   \      /
      A1,...,Asn-1  \    /
                 Asn-1+1,...,An
```

Queremos mostrar que S*[1...i] é solução ótima de calcular A1...As e S*[i+1...n-2] é solução ótima para $A_{sn-1+1}*...*A_n$.

Perceba que S*[1...i] nada mais é que uma permutação dos produtos p1,...,p(sn-1)-1.

Portanto, S*[1...i] é viável para a subinstancia correspondente. Supondo, por absurdo, que ela não seja otina, seja S' uma solução ótima para $A_1*...*A_{sn-1}$.

Isso significa que c(S') < c(S*[[1...i]) sendo c a função que calcula o custo de uma solução viável. Observe que se concatenarmos S' com S*[i+1...n-1], terremos uma solução viável para $A_1*...*A_n$.

Além disso
c(S') concatenado a S*[i+1..n-1] = c(S') + c(S[i+1..n-1])
c(S*) = c(S*[1..i] + c(S*[i+1...n-1])

Daí, temos que (S' * S*[i+1...n-1]) < c(S*), um absurdo.

$A_1 * A_2 * A_3 * A_4$ -> (1,4)
depende de (1,1);(2,4)
(1,2);(3,4)
(1,3);(4,4)

As repetições acontecem em metade da matriz. A linha diagonal central e os números acima bastam ser calculados uma vez apenas. Veremos um tipo de solução desse problema, trabalhando apenas com os números da diagonal central e os que estão acima na matriz resultante.

MCM (D,i,j) = min{MCM(D, i, k) + MCM(D, k+1, j) + D[i – 1] * D[k] * D[j]} , i <= k < j

Algoritmo: MCM_PD(D)

Entrada: Vetor de dimensões D
Saída: menor quantidade de produtos entre escalares para calcular o produto das matrizes com dimensões em D.
Requisito: #matriz = |D| - 1; D[0] = #linhas(A1); D[i] = #colunas(Ai).

```
N ← |D| - 1
T ← NovaMatriz(n, n)
S ← NovaMatriz(n, n)
para i ←1,2,...,n faca:
| T[i][i] ← 0                          --preencher a diagonal da matriz
| S[i][i] ← 0
para i ←n-1,n-2,...,1 faca:
| para j ← i+1,i+2,...,n faca:        --só precisamos trabalhar com metade da matriz
| | minimo ← +infinito
| | para k ← i,i+1,...,j-1 faca:
| | | valor ← T[i][k] + T[k+1][j] + D[i-1]*D[k]*D[j]
| | | se valor < minimo entao:
| | | | minimo ← valor
| | | | S[i][j] ← k
| | T[i][j] ← minimo
retorne T[1][n], S
```

Algoritmo: Sol_MCM(S, i, j, P, prim, ult)
Entrada: Tabela de decisões S, Vetor de soluções P, índices de subproblema i,j, índices prim e ult

```
se i < j entao:
| P[ult] ← S[i][j]
| n_prods ← P[ult] - i
| Sol_MCM(S, i, P[ult], P, prim, prim + n_prods - 1)
| Sol_MCM(S, P[ult] + 1 , j, P, prim + n_prods, ult)
```

>>Problema: Maior Subsequencia Comum

$X = \langle x_1, x_2, \ldots, x_m \rangle$
$Y = \langle y_1, y_2, \ldots, y_n \rangle$
$Z = \langle z_1, z_2, \ldots, z_k \rangle$

Se Z c X e Z c Y, Z é subsequencia comum a X e Y.

Ex: X = <5, 7, 3, 4, 8>, Y = <5, 8, 4, 3, 10, 8>, Z = <5, 3, 8> ou Z = <5, 4, 8>

Teorema: Dadas duas sequencias X e Y, com |X| = m e |Y| = n, e uma MSComum Z , com |Z| = k.

1. Se $X_m = Y_n$, então $Z_k = X_m = Y_n$ e Z[1 …. k – 1] é MSComum a X[1 … m – 1] e Y[1 … n – 1]

2. Se X_m != Y_n temos:
   2.1. Z_k != X_m implica que Z é MSComum a X[1 … m – 1] e Y
   2.2. Z_k != Y_n implica que Z é MSComum a X e Y[1 … n – 1]

Demonstração:

(1) Suponha que $X_m = Y_n$ e suponha por absurdo que $Z_k != M_x$.

Perceba que Z, por ser um MSComum, inclui elementos apenas de X[1 … m – 1] e Y[1 … n – 1], portanto, se tomarmos a sequencia Z * <X_m> temos uma nova sequencia comum a X e Y em tamanho maior que Z, um absurdo.

Suponha que Z[1 … k – 1] não seja ótima, por absurdo. Assim, seja W uma MSComum a X[1 … m – 1] e Y[1 … n – 1]. Nesse caso, |W| > k – 1.

Perceba que W * <X_m> deve ser subsequencia comum a X e Y, por construção. Dai temos que |W * <X_m>| > (k – 1) + 1 = |Z|, um absurdo. Logo, (1) é verdadeira.

(2.1) Suponha que $X_m != Y_n$ e que $Z_k != X_m$. Logo, $Z_k != X_m$ deve existir um índice i < m tal que $X_i = Z_k$. Sendo assim, descartar X_m não impede de afirmarmos que Z é comum a X[1 … m – 1] e Y.

Supondo, por absurdo que Z não seja ótima e que W seja ótima, podemos perceber que W é comum a X e Y, dado que por construção W não considera X_m. Porém, como |W| > |Z| e ambas são comuns a X e Y, temos um absurdo á hipótese de Z ser ótima para X e Y.

(2.2) Análoga a 2.1

MSComum (X, i, Y, j) = { 1 + MSComum(X, i – 1, Y, j – 1), caso i, j >= 1 e X[i] = Y[j]
{ max{ MSComum(X, i – 1, Y, j), MSComum(X, i, Y, j – 1),
caso i, j >= 1 e X[i] != Y[j]
{ 0, caso i = 0 ou j = 0

Algoritmo: MSComum(X, i, Y, j)
Entrada: Sequencias X e Y, índices respectivos
Saída: Tamanho da maior sequencia comum à X e Y

```
a = |X|, b = |Y|
T ← CriarMatrizIndiceZero(a , b)
para i ← 0, …, a faca:
    para j ← 0, …, b faca:
        T[i][j] ← 0

para i ← 1, …, a faca:
    para j ← 1, …, b faca:
        se X[i] == X[j] entao:
            T[i][j] = T[i – 1][j – 1] + 1
        senao:
            T[i][j] ← max(T[i – 1][j], T[i][j – 1])

retorne T[a][b]
```
