PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

7 de novembro de 2019

Avisos

Site da disciplina:

- http://www.decom.ufop.br/marco/

Lista de e-mails:

- pcc104@googlegroups.com

Para solicitar acesso:

- http://groups.google.com/group/pcc104

Na aula de hoje

1. Programação Dinâmica Parte II – Exemplos Clássicos
   - Problema da Mochila 0-1
   - Longest Increasing Subsequence
   - Coin Change
   - Maximum-Subarray Problem
   - Max 2d Range Sum
   - String Alignment

Avisos

O Problema da Mochila 0-1

Dadas uma mochila de capacidade W e uma lista de n itens distintos e únicos (enumerados de 0 a $n-1$), cada um com um peso $w_0, w_1, \ldots, w_{n-1}$ e um valor $v_0, v_1, \ldots, v_{n-1}$, maximizar o valor carregado na mochila, respeitando sua capacidade. Para cada item, devemos escolher se ele estará incluso na solução ou não.

Exemplo

$W = 10$, $n = 4$, w=[8, 1, 5, 4] e V=[500, 1000, 300, 210].

A resposta é 1510, selecionando os itens 1, 2 e 3, com peso total 10.

Programação Dinâmica

Recorrência da Busca Completa

Consideramos dois parâmetros para caracterizar um estado/subproblema: o índice i do item atual e o peso remanescente W na mochila:

- mochila(i, 0) = 0; (Não cabe mais nada na mochila);
- mochila(n, W) = 0; (Já analisamos todos os itens);
- se $w_i > W$, o item i não cabe na mochila e deve ser ignorado, então, mochila(i, W) = mochila(i+1, W);
- se $w_i \leq W$, então podemos levar o item i ou não: mochila(i, W) = max(mochila(i+1, W), v_i + mochila($i + 1$, $W - w_i$)).

Versão *Top-Down*

Adicionamos à recorrência descrita uma tabela bidimensional que armazena o valor de uma solução para derivarmos a formulação por programação dinâmica *top-down*.

Versão *Top-Down*

Mochila(i, W)
Entrada: Índice i, Peso remanescente W
se $i = n$ ***ou*** $W = 0$ **então**
| **retorna** *0*;
fim
se $tabela[i][W] \neq -1$ **então**
| **retorna** $tabela[i][W]$;
fim
se $w_i > W$ **então**
| **retorna** $tabela[i][W] \leftarrow$ ***Mochila***$(i + 1, W)$;
fim
retorna
$tabela[i][W] \leftarrow max($***Mochila***$(i+1, W), v_i +$ ***Mochila***$(i+1, W - w_i)$;

Exercício

Preencha a tabela abaixo com a execução da formulação.

(w_i, v_i) / W	1	2	3	4	5	6	7	8	9	10
(8, 500)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(1, 1000)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(5, 300)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(4, 210)	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Versão *Bottom-Up*

Mochila()

//Aumentamos uma linha e um coluna dummies na tabela

//A primeira linha e a primeira coluna da tabela são zeradas previamente

para $i \leftarrow 1$ ***até*** $n + 1$ **faça**

  para $j \leftarrow 1$ ***até*** $W + 1$ **faça**

    se $w_i > j$ **então**

      $tabela[i][j] \leftarrow tabela[i-1][j];$

    senão

      $tabela[i][j] \leftarrow max(tabela[i-1][j], v_i + tabela[i-1][j-w_i];$

    fim

  fim

fim

retorna $tabela[n][W];$

Exercício

Preencha a tabela abaixo com a execução da formulação.

(w_i, v_i) / W	$\emptyset$	1	2	3	4	5	6	7	8	9	10
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0
(8, 500)	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(1, 1000)	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(5, 300)	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
(4, 210)	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Complexidade

Apesar dos subproblemas repetidos, há apenas $O(nW)$ subproblemas distintos, uma vez que existem n itens e o peso pode variar entre 0 e W.

Podemos calcular a solução de cada problema em $O(1)$ e, portanto, a complexidade da formulação por programação dinâmica é $O(nW)$.

Bottom-Up vs. *Top-Down*

Para este problema específico, claramente a versão *Top-Down* é mais rápida, uma vez que não preenche todas as entradas da tabela $n \times W$, ao contrário da versão *Bottom-Up*.

Complexidade

É importante salientar que o problema da mochila 0-1 é NP-Difícil, e este é um algoritmo **pseudo-polinomial**, ou seja, é polinomial no valor numérico da entrada, porém, exponencial no comprimento da codificação mesma.

Especificamente, W não é polinomial em relação ao comprimento da sua codificação.

Por exemplo, seja $W =$ 1.000.000.000.000: são necessários 40 bits para representar este número, porém, o tempo de execução considera o fator 1.000.000.000.000, que é $O(2^{40})$, ou seja, $O(2^{logW})$.

Na verdade, a complexidade do algoritmo é mais precisamente expressa por $O(n2^{logW})$, ou seja, exponencial.

Complexidade

Por exemplo, suponhamos uma instância com $W = 2$, o que exige 2 bits para representar.

Se alterarmos o valor de W para 4, a representação aumentou em um bit, mas a complexidade computacional dobrou.

Se aumentarmos W para 1024 na mesma instância, precisaremos de 10 bits. Entretanto, a complexidade aumenta por um fator de 512.

Dada esta complexidade, em casos que $nW >> 1M$, mesmo a versão *on-the-fly* não é viável.

Longest Increasing Subsequence

Dada uma sequência $\{x_0, x_1, \ldots, x_{n-1}\}$, é necessário determinar a maior subsequência crescente. Note que a subsequência não necessariamente é contígua.

Exemplo

$N = 8$ e a sequência {-7, 10, 9, 2, 3, 8, 8, 1}.

A solução é {-7, 2, 3, 8} com comprimento 4.

Solução

Seja LIS(i) a maior subsequência crescente terminando no índice i (inicialmente, todas possuem valor 1), então temos:

- **LIS**$(0) = 1$;
- **LIS**$(i) = max(LIS(j) + 1)$, $\forall\ j \in [0..i-1]$ e $x_j < x_i$.

Versão *Bottom-Up*

```
LISBottomUp(X, n)
Entrada: Vetor X, Inteiro n
para i ← 1 até n faça
|   ans ← 0;
|   para j ← 0 até i faça
|   |   se X[i] > X[j] e LIS[j] > ans então
|   |   |   ans ← LIS[j];
|   |   fim
|   fim
|   LIS[i] ← ans + 1;
fim
ans ← 0;
para i ← 0 até n faça
|   se LIS[i] > ans então  ans ← LIS[i] ;
fim
retorna ans;
```

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

As soluções para o problema podem ser reconstruídas seguindo as setas.

Programação Dinâmica

Longest Increasing Subsequence

Considerando $n = 8$ e a sequência {-7, 10, 9, 2, 3, 8, 8, 1}, temos:

- **LIS**(0) = 1, o próprio número -7;
- **LIS**(1) = 2, {-7, 10};
- **LIS**(2) = 2, {-7, 10}. Podemos formar {-7, 9}, mas não {-7, 10, 9}, que não é crescente;
- **LIS**(3) = 2, {-7, 10} ou {-7, 9}. Podemos formar {-7, 2}, mas não {-7, 10, 2} ou {-7, 9, 2}, que não são crescentes;
- **LIS**(4) = 3, {-7, 2, 3};
- **LIS**(5) = 4, {-7, 2, 3, 8};
- **LIS**(6) = 4, {-7, 2, 3, 8};
- **LIS**(7) = 2, {-7, 1}.

Longest Increasing Subsequence

A complexidade de tempo do algoritmo anterior é determinada pelos laços de repetição: $O(n^2)$.

Depois de preenchida a tabela, determinar o maior elemento é feito em tempo $O(n)$.

A complexidade de espaço é $2 \times n$, ou $O(n)$, dado que temos um vetor para a entrada e outro de mesmo tamanho para a tabela.

Análise – *Longest Increasing Subsequence*

Claramente, há vários subproblemas sobrepostos ao determinarmos a maior subsequência crescente.

Entretanto, há apenas N estados, ou seja, a maior subsequência crescente que termina no índice i $(0 \leq i \leq n-1)$.

Programação Dinâmica

Coin Change

Dada uma quantia V e uma lista de denominações de n moedas, qual é o número mínimo de moedas que devemos utilizar para obter a quantia V?

Nota: Nesta versão do problema consideramos ter um suprimento infinito de moedas de qualquer denominação.

Exemplo 1

$V = 10, n = 2$ e denominação = {1, 5}, podemos usar:

- Dez moedas de 1 centavo = 10 moedas;
- Uma moeda de 5 centavos + cinco moedas de 1 centavo = 6 moedas;
- Duas moedas de 5 centavos = 2 moedas.

Lembrete

Para algumas denominações de moedas podemos utilizar algoritmos gulosos com sucesso.

Exemplo 2

$V = 7, n = 4$ e denominação = {1, 3, 4, 5}.

A solução gulosa seria 3 (uma moeda de 5 centavos e duas moedas de 1 centavo).

Entretanto a solução ótima é 2 (uma moeda de 3 centavos e uma moeda de 4 centavos).

Solução

Usamos a recorrência de busca completa:

- **moeda**(0) = 0;
- **moeda**(< 0) = ∞;
- **moeda**($valor$) = 1 + min(**moeda**($valor-$denominação$[i]$)) $(\forall\ i \in [0..n-1])$.

A resposta é determinada por **moeda**(V).

Versão *Top-Down*

CoinChange(V)
Entrada: Inteiro V
se $troco[V] \neq \infty$ ***ou*** $V < 0$ **então**
  retorna $troco[V]$;
senão
  para $i \leftarrow 0$ ***até*** n **faça**
    $troco[V] \leftarrow min(troco[V]$, 1+**CoinChange**$(V - D[i]))$;
  fim
  retorna $troco[V]$;
fim

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2,
coinValue= {1, 5}

Tabela para a formulação por programação dinâmica para o problema *Coin Change*.

Coin Change

Considerando $V = 10, n = 2$ e denominação = {1, 5}, temos:

- **moeda**(<0) = ∞;
- **moeda**(0) = 0, o caso base;
- **moeda**(1) = 1, obtido de 1 + **moeda**(1-1). **moeda**(1-5) é inviável;
- **moeda**(2) = 2, obtido de 1 + **moeda**(2-1). **moeda**(2-5) é inviável;

  $\vdots$
- **moeda**(5) = 1, obtido de 1 + **moeda**(5-5), menor do que 1+**moeda**(5-1) = 5;

  $\vdots$
- **moeda**(10) = 2, obtido de 1 + **moeda**(10-5), menor do que 1+**moeda**(10-1) = 6.

Análise – *Coin Change*

Novamente, há vários subproblemas sobrepostos ao determinarmos o número mínimo de moedas.

Entretanto, há apenas $O(V)$ estados, ou seja, V quantias possíveis.

Como testamos $O(n)$ moedas por estado, a complexidade geral é $O(nV)$.

Maximum-Subarray Problem, ou *Max 1d Range Sum*

Dada uma vetor de n inteiros diferentes que zero, determine o intervalo de maior soma.

Exemplo

Consideremos a sequência -2, 1, -3, 4, -1, 2, 1, -5, 4. A subsequência contígua de números de soma máxima é 4, -1, 2, 1, cuja soma é 6.

Observação

Existem $O(n^2)$ possibilidades de intervalos, porém, vimos um algoritmo de divisão e conquista com tempo $O(n\ log\ n)$.

Entretanto, o algoritmo de *Kadane* resolve este problema em tempo $O(n)$.

Algoritmo de *Kadane*

O princípio do algoritmo de *Kadane* é manter a soma dos elementos já analisados e reiniciar o valor em 0 caso a soma se torne negativa.

Isto porque é melhor começar de zero do que continuar de uma soma "ruim".

Embora possa também ser interpretado como um algoritmo guloso, este algoritmo possui características de programação dinâmica na medida em que a cada passo possui duas opções: incrementa a soma do intervalo anterior ou inicia um novo intervalo.

Exercício

Execute o algoritmo de *Kadane* para a entrada A=[4, -5, 4, -3, 4, 4, -4, 4, -5] e $n = 9$.

KadaneOnTheFly(A, n)
Entrada: Vetor A, Inteiro n
$soma \leftarrow 0$;
$resp \leftarrow 0$;
para $i \leftarrow 0$ ***até*** n **faça**
    $soma \leftarrow soma + A[i]$;
    $resp \leftarrow max(resp, soma)$;
    se $soma < 0$ **então**
        $soma \leftarrow 0$;
    fim
fim
retorna $resp$;

Max 2d Range Sum

Dada uma matriz $n \times n$ de inteiros, determine a submatriz de maior valor.

Exemplo

$$\begin{array}{cccc} 0 & -2 & -7 & 0 \\ \color{red}{9} & \color{red}{2} & -6 & 2 \\ \color{red}{-4} & \color{red}{1} & -4 & 1 \\ \color{red}{-1} & \color{red}{8} & 0 & -2 \end{array}$$

9+2-4+1-1+8=15

Estratégia Ingênua

Podemos tentar uma estratégia de busca completa, mostrada a seguir.

```
MaximumSumN6(M, n)
Entrada: Matriz M, inteiro n
maxSubRect ← −∞;
para i ← 0 até n faça
    para j ← 0 até n faça
        para k ← i até n faça
            para l ← j até n faça
                subRect ← 0;
                para a ← i até k faça
                    para b ← j até l faça
                        subRect ← subRect + M[a][b];
                    fim
                fim
                maxSubRect ← max(maxSubRect, subRect);
            fim
        fim
    fim
fim
retorna maxSubRect;
```

Max 2d Range Sum

Claramente, o algoritmo anterior possui complexidade $\Theta(n^6)$, e para uma matriz de ordem 100 realizaria $100^6 = 1.000.000.000.000$ (um trilhão) de operações.

Existem várias soluções por Programação Dinâmica conhecidas para problemas de tamanhos estáticos como este.

Para este problema, quando computamos uma submatriz maior, definitivamente estamos também computando submatrizes menores, ou seja, subproblemas sobrepostos.

Max 2d Range Sum

Uma possível solução por Programação Dinâmica é transformar a matriz `entrada[n][n]` em uma matriz `soma[n][n]` em que `soma[i][j]` não contém o valor de `entrada[i][j]`, mas sim a soma de todos os elementos na submatriz delimitada pelos índices (0, 0) e (i, j).

Podemos computar a segunda matriz no exato momento em que lemos a entrada, e a complexidade da operação se manterá $\Theta(n^2)$.

```
SubMatrixSum()
Leia n;
para i ← 0 até n faça
    para j ← 0 até n faça
        Leia o elemento soma[i][j];
        //se possível, adiciona os valores do topo
        se i > 0 então
            soma[i][j] ← soma[i][j] + soma[i − 1][j];
        fim
        //se possível, adiciona os valores da esquerda
        se j > 0 então
            soma[i][j] ← soma[i][j] + soma[i][j − 1];
        fim
        //evita contagens duplicadas, princípio da inclusão-exclusão
        se i > 0 e j > 0 então
            soma[i][j] ← soma[i][j] − soma[i − 1][j − 1];
        fim
    fim
fim
```

Max 2d Range Sum

O procedimento **SubMatrixSum** transforma a matriz de entrada (à esquerda) na matriz de soma (à direita).

Com esta estratégia, podemos determinar a soma de qualquer submatriz delimitada por (i, j) e (k, l) em $O(1)$.

Por exemplo, se quisermos determinar a soma da submatriz delimitada por (2, 3) e (4, 4) (elementos $soma[1][2]$ e $soma[3][3]$) inclusive, podemos dividir a matriz original em 4 partes e calcular `soma[3][3] - soma[0][3] - soma[3][1] + soma[0][1]` = -3-(-9)-13+(-2) = -9.

Exemplo

$$\begin{matrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{matrix} \qquad \begin{matrix} 0 & -2 & -9 & -9 \\ 9 & 9 & -4 & -2 \\ 5 & 6 & -11 & -8 \\ 4 & 13 & -4 & -3 \end{matrix}$$

Max 2d Range Sum

Apesar de podermos determinar a soma de uma submatriz específica em $O(1)$, ainda falta determinar a maior soma de uma submatriz.

O método apresentado a seguir, *bottom-up*, requer complexidade $O(n^4)$, ou 100.000.000 operações para uma matriz de dimensões 100.

Entretanto, há uma maneira de resolver o problema em $O(n^3)$, ou 1.000.000 operações.

```
MaximumSumN4(soma, n)
Entrada: Matriz soma, inteiro n
maxSubRect ← −∞;
para i ← 0 até n faça
    para j ← 0 até n faça
        para k ← i até n faça
            para l ← j até n faça
                subRect ← soma[k][l];//soma de (0,0) a (k, l): O(1)
                se i > 0 então
                    subRect ← subRect − soma[i − 1][l];//O(1)
                fim
                se j > 0 então
                    subRect ← subRect − soma[k][j − 1];//O(1)
                fim
                se i > 0 e j > 0 então
                    subRect ← subRect + soma[i − 1][j − 1];//O(1)
                fim
                maxSubRect ← max(maxSubRect, subRect);//O(1)
            fim
        fim
    fim
```

Parâmetros Comuns

Após resolver uma quantidade suficiente de formulações por programação dinâmica e *backtracking*, é possível notar parâmetros comumente selecionados para representar os subproblemas.

Alguns destes parâmetros são descritos a seguir.

Programação Dinâmica

Parâmetros Comuns 1

Parâmetro: índice i em um arranjo, i.e., $[x_0, x_1, \ldots, x_i, \ldots]$.
Transição: Aumentar o arranjo $[0..i]$ (ou $[i..n\text{-}1]$), processar i, levar o item i ou não, etc.
Exemplos: Maximum subarray problem, Longest Increasing Subsequence, parte da mochila 0-1, Caixeiro Viajante, etc.

Parâmetros Comuns 2

Parâmetro: índices (i, j) em dois arranjos, i.e., $[x_0, x_1, \ldots, x_i]$+$[y_0, y_1, \ldots, y_i]$.
Transição: aumentar i, j ou ambos.
Exemplos: Alinhamento de Strings/Distância de Edição, Longest Common Subsequence, etc.

Parâmetros Comuns 3

Parâmetro: parâmetro estilo mochila (capacidade).
Transição: diminuir (ou aumentar) o valor atual até zero (ou outro limite).
Exemplos: Mochila 0-1, Subset Sum, Coin Change, etc.

Parâmetros Comuns 4

Parâmetro: subarranjo (i, j) em um arranjo, i.e., $[\ldots, x_i, x_{i+1}, \ldots, x_j, \ldots]$.
Transição: dividir (i, j) em (i, k)+$(k+1, j)$ ou em $(i, i+k)$+$(i+k+1, j)$.
Exemplos: Multiplicação de matrizes em cadeia.

Programação Dinâmica

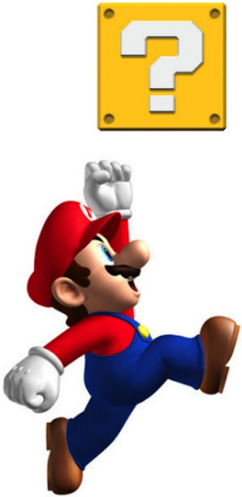
Parâmetros Comuns 5

Parâmetro: vértice em um grafo direcionado acíclico.
Transição: processar os vizinhos do vértice.
Exemplos: caminhos mais longos, caminhos mais curtos, enumerar caminhos, etc.

Parâmetros Comuns 6

Parâmetro: conjunto (pequeno), normalmente usando *bitmask*.
Transição: marcar um ou mais elementos no conjunto como ligado ou desligado, presente ou ausente, etc.
Exemplos: problemas de emparelhamento, caixeiro viajante (como subrotina que verifica as cidades já visitadas) ou qualquer outra formulação de programação dinâmica que use *bitmask*.

Dúvidas?



Alinhamento de *Strings*

O Problema de Alinhamento de *Strings* (ou Distância de Edição) nos pede que, dadas duas *strings* A e B, alinhe-as[a] com máxima pontuação de alinhamento (ou o mínimo de operações de edição).

Após alinharmos A e B, existem 4 possibilidades para os caracteres $A[i]$ e $B[i]$ $\forall$ índice i:

1. Os caracteres $A[i]$ e $B[i]$ **coincidem** (pontuação +2): não fazemos nada;
2. Os caracteres $A[i]$ e $B[i]$ **descoincidem** (pontuação -1): substituímos $A[i]$ por $B[i]$;
3. Inserimos um espaço em $A[i]$ (pontuação -1); ou
4. Removemos o caractere em $A[i]$ (pontuação -1).

[a] O alinhamento é o processo de inserir espaços nas *strings* A ou B tal que elas tenham o mesmo número de caracteres.

Exemplo

```
A='ACAATCC' →'A _ C A A T C C'
B='AGCATGC' →'A G C A T G C _ '
             2 - 2 2 - - 2 -
```

Pontuação de alinhamento: $4 \times 2 + 4 \times -1 = 4$.

Alinhamento de *Strings*: *Needleman-Wunsch*

O algoritmo de *Needleman-Wunsch* é um famoso algoritmo de programação dinâmica *bottom-up* para resolver este problema.

Consideremos duas *strings* $A[1 \ldots n]$ e $B[1 \ldots m]$:

- Definimos $V(i, j)$ como sendo a pontuação ótima do alinhamento entre $A[1 \ldots i]$ e $B[1 \ldots j]$;
- Definimos que **pontuação**(a, b) é a pontuação do alinhamento dos caracteres a e b.

Alinhamento de *Strings*

Casos base:

- $V(0,0) = 0$ (não há pontuação para alinharmos duas *strings* vazias);
- $V(i,0) = i\times$ **pontuação**$(A[i], \_)$, deletamos a string $A[1..i]$ para realizar o alinhamento ($i>0$);
- $V(0,j) = j\times$ **pontuação**$(\_, B[j])$, inserimos espaços em $B[1..j]$ para realizar o alinhamento ($j>0$).

Alinhamento de *Strings*

Recorrências para $i > 0$ e $j > 0$:

- $V(i, j) = max$ (*opção1*, *opção2*, *opção3*), em que:
  - *opção1* $= V(i-1, j-1)$+**pontuação**($A[i]$, $B[j]$) (coincidência ou não);
  - *opção2* $= V(i-1, j)$+**pontuação**($A[i]$, _) (remoção de $A[i]$);
  - *opção3* $= V(i, j-1)$+**pontuação**(_, $B[j]$) (inserção em $B[j]$).

Alinhamento de *Strings*

Esta formulação se concentra nas três possibilidades para o último par de caracteres: uma coincidência/descoincidência, uma remoção ou uma inserção.

Embora não saibamos qual é a melhor, podemos tentar todas as possibilidades e evitar a computação redundante de subproblemas.

Exemplos

```
xxx...xx     xxx...xx          xxx...x_
       |            |                 |
xxx...yy     yyy...y_          yyy...yy
```

Coincidência/descoincidência, remoção e inserção.

Alinhamento de *Strings*

Usando o exemplo de função de custo em que uma coincidência pontua +2, uma descoincidência, uma remoção e uma inserção pontuam -1, a figura a seguir demonstra a pontuação de alinhamento para `A='ACAATCC'` e `B='AGCATGC'`.

A pontuação de alinhamento é 7 (canto inferior direito). Seguindo as setas vermelhas, partindo do canto inferior direito podemos reconstruir a solução.

As setas em diagonal representam uma coincidência ou uma descoincidência (por exemplo, o último 'C').

As setas verticais representam uma remoção (por exemplo, `...CAT..` para `...C_T..`).

As setas horizontais representam uma inserção (por exemplo, `A_C` para `AGC..`).

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2		Base Cases					
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

Casos base para o alinhamento das *strings* A = ‘`ACAATCC`’ e B = ‘`AGCATGC`’.

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3				
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

Exemplo de alinhamento de *strings* para o alinhamento das *strings* A = ‘ACAATCC’ e B = ‘AGCATGC’.

Programação Dinâmica

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

Alinhamento com pontuação 7 ($5 \times 2 + 3 \times -1 = 7$).

A = ‘A_CAAT[C]C’

B = ‘AGC_AT[G]C’

Alinhamento de *Strings*

É necessário preencher todos os elementos da matriz $n \times n$.

Cada elemento pode ser computado em $O(1)$.

A complexidade de tempo e de espaço é $O(nm)$ – exatamente o tamanho da tabela da Programação Dinâmica.

A busca completa testaria todas as $\binom{n+m}{m}$ combinações.

Apenas alterando o valor das pontuações, podemos resolver também o *Longest Common Subsequence*, um problema similar.

Dúvidas?