Programación Dinámica

- La técnica de programación dinámica se aplica en general a problemas de optimización. Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamaños menores que son mas fáciles de resolver, pero en este caso se empieza resolviendo los problemas más pequeños. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.
- La idea es evitar resolver más de una vez los mismos problemas.

- Técnica bottom- up

Programación Dinámica

- **Principio de optimalidad** : un problema de optimización satisface el principio de optimalidad si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima. *(es necesario que se cumpla para poder usar la técnica de programación dinámica, no todos los problemas lo cumplen)*

Ejemplos:

- Coeficientes binomiales
- Producto de matrices
- Comparación de secuencias de ADN
- Subsecuencia creciente máxima
- Arbol de búsqueda óptimo
- etc.

Cálculo de coeficientes binomiales

Supongamos que queremos calcular los coeficientes binomiales usando la siguiente función ("divide and conquer"):

Función C(n,k)

si k = 0 **or** k = n **entonces return** 1

sino return C (n-1, k-1) + C (n-1, k)

Es eficiente calcular así?.

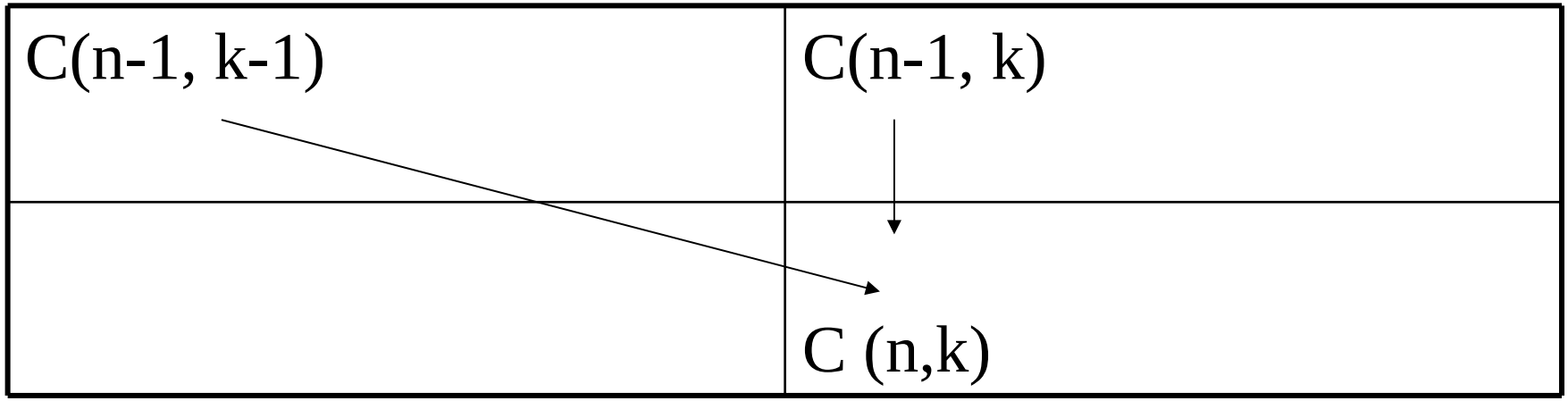
No, los mismos valores de C(n,k) son calculados varias veces.

Cuál es la complejidad de este algoritmo?.

$$\Omega\left(\binom{n}{k}\right)$$

Triángulo de Pascal

	0	1	2	3	…….	k-1	k
0	1	.	.	.	.	.	.
1	1	1	.	.			.
2	1	2	1	.	.		
…….	.	.	.	.	.		
n-1	.	.	.	.	.		
n	.	.	.	..			.



Cuál es la complejidad de este algoritmo?.
Cuánta memoria requiere?. Hay que guardar en memoria toda la tabla?.

- *La “fórmula” que usamos aquí es la misma que en el algoritmo “divide and conquer”, pero acá no calculamos varias veces lo mismo.*

- Tenemos entonces una complejidad de O(nk).

Problema de la multiplicación de n matrices

Queremos ver la forma óptima de calcular

$$M = M_1 M_2 M_3 \ldots\ldots\ldots M_n$$

Por la propiedad asociativa del producto de matrices esto puede hacerse de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias.

Vale el principio de optimalidad acá?

Ejemplo: supongamos que A es de 13 x5, B de 5 x 89 , C de 89 x 3 y D de 3 x 34.

((AB)C)D requiere 10582 multiplicaciones

(AB)(CD) requiere 54201 multiplicaciones

(A(BC))D requiere 2856 multiplicaciones

A((BC)D) requiere 4055 multiplicaciones

A(B(CD)) requiere 26418 multiplicaciones

Para ver cuál es la forma óptima de multiplicar deberíamos analizar todas las formas posibles de poner los paréntesis y ver cual es la mejor.

Sea M =($M_1 M_2 M_3 \ldots\ldots M_i$)($M_{i+1} M_{i+2} \ldots\ldots M_n$)

Sea T(i) la cantidad de formas de poner los paréntesis en el lado derecho y T(n-i) en el lado izquierdo. Entonces para cada i hay T(i)T(n-i) formas de poner los paréntesis para toda la expresión y

$$T(n) = \Sigma_i T(i)\, T(n-i)$$

A partir de que T(1) =1 se puede calcular la expresión para cualquier n. (números catalanes).

n	1	2	3	4	5	10	15
T(n)	1	1	2	5	14	4862	267440

Encontrar de esta forma *(algoritmo de fuerza bruta)* la mejor manera de calcular la cantidad de multiplicaciones que requiere cada posibilidad es del orden $\Omega\ (4^n/n)$.

Dem: ejercicio, usar que $\binom{2n}{n} \geq 4^n / (2n+1)$ y $T(n) = 1/n \binom{2n-2}{n-1}$

Cómo usar Programación Dinámica para resolver este problema?

Construimos una tabla m_{ij} $(1 \leq i \leq j \leq n)$ donde m_{ij} es la cantidad mínima de multiplicaciones necesaria para calcular

$$M_i M_{i+1} M_{i+2} \ldots\ldots M_j .$$

La solución al problema es entonces m_{1n}.

Suponemos que las dimensiones de la matriz M_i están dadas por un vector d_i , $0 \leq i \leq n$, donde la matriz Mi tiene d_{i-1} filas y d_i columnas y armamos la tabla diagonal por diagonal, para cada diagonal s, s = 0,……., n-1

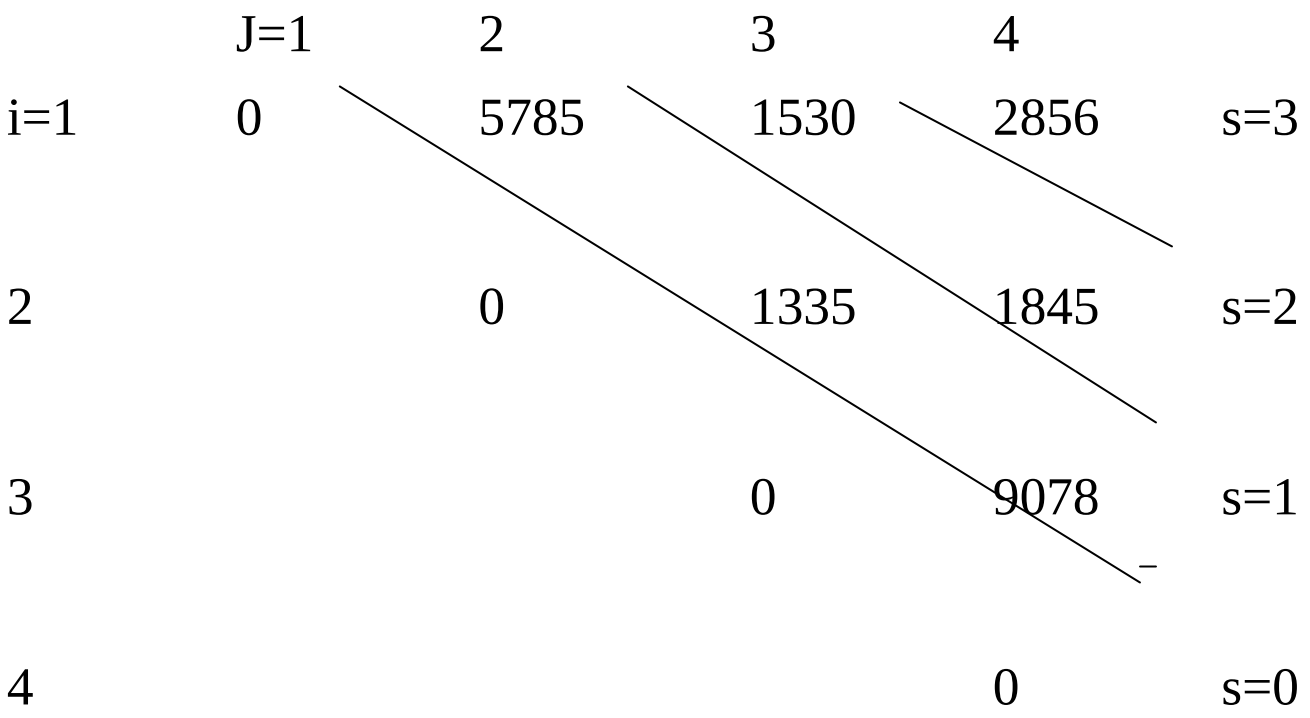
- $s=0 : m_{ii} = 0$ para $i=1,2,\ldots n$
- $s= 1 : m_{i\,i+1} = d_{i-1}d_i d_{i+1}$ para $i=1,2,\ldots n-1$
- $1< s < n : \; m_{i\,i+s} = \min_{i \leq k < i+s} \{m_{ik} + m_{k+1\,i+s} + d_{i-1}d_k d_{i+s}\}$ para $i=1,2,\ldots n-s$

En el ejemplo anterior d= (13,5,89,3,34) y entonces

. Para s =1 m_{12} = 5785 , m_{23} = 1335, m_{34} = 9078

. Para s =2 m_{13} = 1530, m_{24} = 1845

. Para s = 3 m_{14} = 2856



- Cómo guardar la información para saber además cuál es la forma de hacer el producto tener este número de multiplicaciones?.

- Complejidad : en cada diagonal s hay que calcular n-s elementos y para cada uno de ellos hay que elegir entre s posibilidades, entonces la cantidad de operaciones del algoritmo es del orden de:

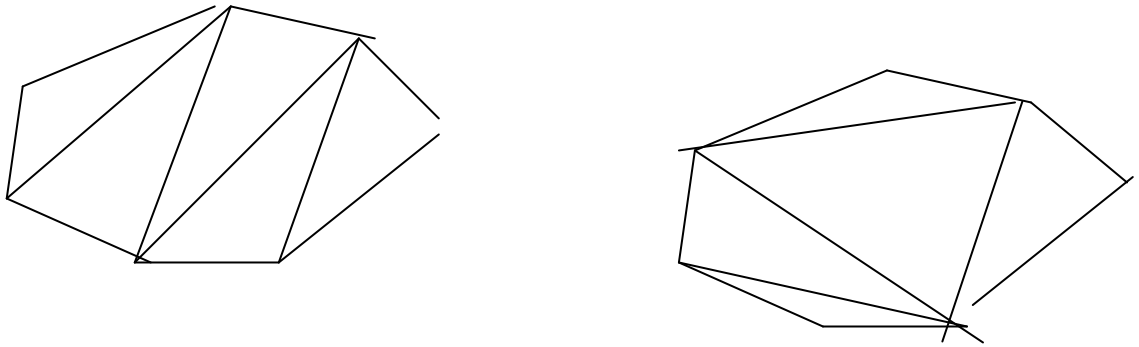
$$\Sigma_s (n-s)s = n \Sigma_s s - \Sigma_s s^2 = n^2 (n-1)/2 - n (n-1) (2n-1)/6$$
$$= (n^3 - n)/6$$

o sea la complejidad del algoritmo es $O(n^3)$.

Triangulación de mínimo peso

Una triangulación de un polígono convexo P = {v_1, v_2.... v_n, v_1} es un conjunto de diagonales que particiona el polígono en triángulos. El peso de la triangulación es la suma de las longitudes de sus diagonales.

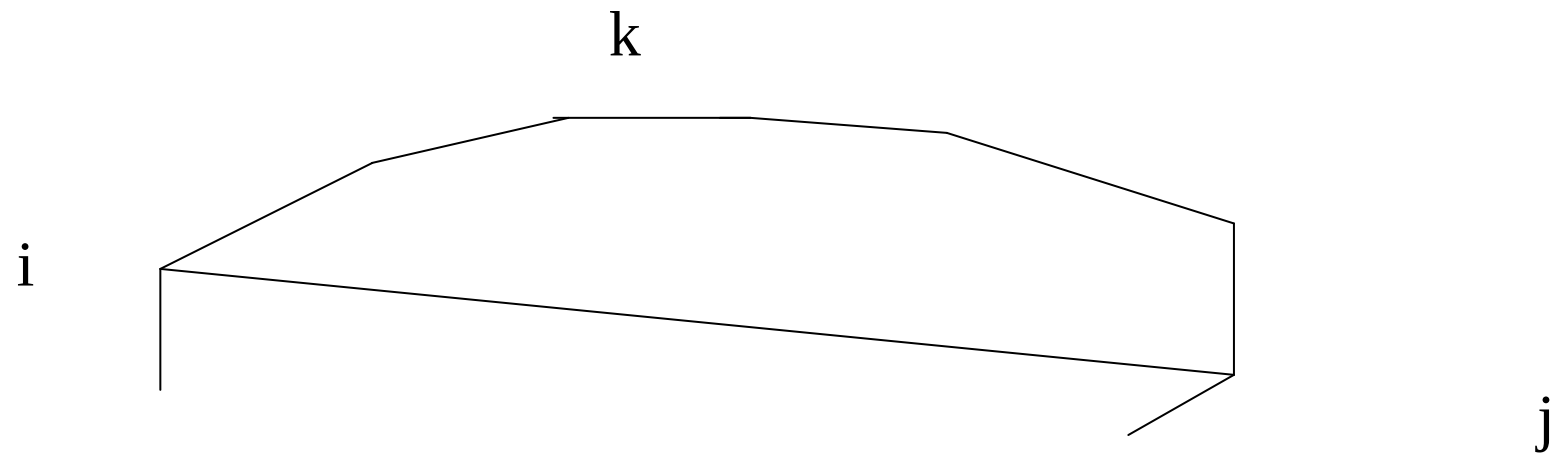
Ejemplo: dos triangulaciones de un mismo polígono.



Observar que cada eje del polígono original está en un solo triángulo.

Dado un segmento que une dos vértices adyacentes, transformarlo en un triángulo implica determinar el tercer vértice. Una vez identificado este vértice al polígono le quedan dos polígonos menores que deben ser a su vez triangulados.

Sea T [i,j] el costo de triangular una de las parte del polígono que queda después de trazar una diagonal desde el vértice v_i al vértice $v_{j,}$, sin tomar en cuenta la longitud d_{ij} de la cuerda que va de v_i a v_j *(para no contarla dos veces).*



$$T[i,j] = \min_{k=i,j} \{T[i,k] + T[k,j] + d_{ik} + d_{kj}\}$$

Caso base, cuando i y j son vecinos:

$$T[i, i+1] = 0$$

Algoritmo para triangulación del polígono

Triangulación-mínima (P)
para i = 1,n -1 hacer
    T[i,i+1] =0
para gap = 1, n-1
    para i = 1, n-gap hacer
        j= i+gap
        T [i,j] = $\min_{k=i,j}$ {T [i,k] + T [k+1,j] + d_{ik} + d_{kj} }
return T[1,n]

==========

- *Complejidad : $O(n^3)$*
- *Complejidad espacial: $O(n^2)$*

==========

- Qué pasa si el polígono no es convexo?.
- Qué pasaría si hubiera puntos en el interior del polígono?.

El número de subregiones en cada paso ya no es 2 y crece exponencialmente y entonces el algoritmo de programación dinámica no es eficiente.

No se conocen algoritmos polinomiales para el problema de triangulación en este caso.

Comparación de secuencias de ADN

Supongamos que tenemos dos secuencias de ADN
GACGGATTAG y GATCGGAATAG

Queremos decidir si son "parecidas" o no.

Para qué se usa esto?

Alineamiento

GA- CGGATTAG
GATCGGAATAG

Objetivo: construir un algoritmo que determine la mejor alineación global entre dos secuencias (que pueden tener distinta longitud).

Asignamos valores a las coincidencias, a las diferencias y a los gaps.

(Las transparencias que siguen sobre este tema, fueron preparadas por alumnos de la materia Biología Computacional, ver página web).

Scores

Ejemplo

T GA–CGGATTAG
S GATCGGAATAG

match = +1
mismatch = -1
gap penalty = -2

Score(T, S) = **9 x 1** + **1 x (-1)** + 1 x (-2) = 6

Algoritmo de Programación dinámica

Needleman & Wunsch (1970)

- Encuentra el mejor scoring para un alineamiento global de dos secuencias

- A partir de allí se determina un alineamiento que tiene ese score óptimo.

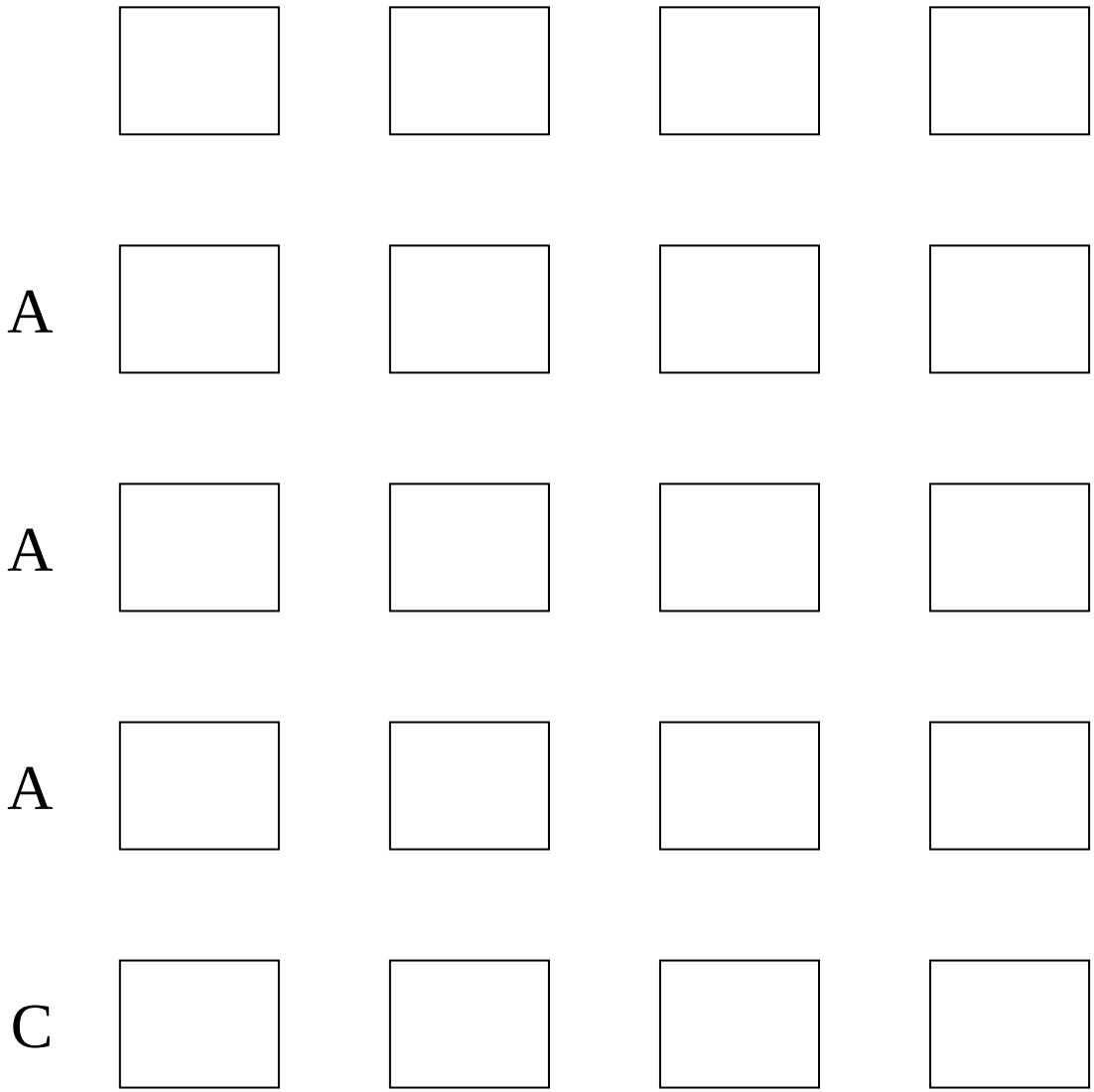
De acuerdo a los parámetros elegidos la alineación óptima puede variar.

- Se quiere alinear las secuencias S y T.
- Sea n el tamaño de S y m el de T.
- Hay n+1 posibles prefijos de S y m+1 posibles prefijos de T (incluyendo el string vacío).
- Se genera una matriz F de (n+1)x(m+1) elementos.
- El elemento F(i, j) tendrá la mejor similitud entre las subsecuencias S[1...i] y T[1...j].

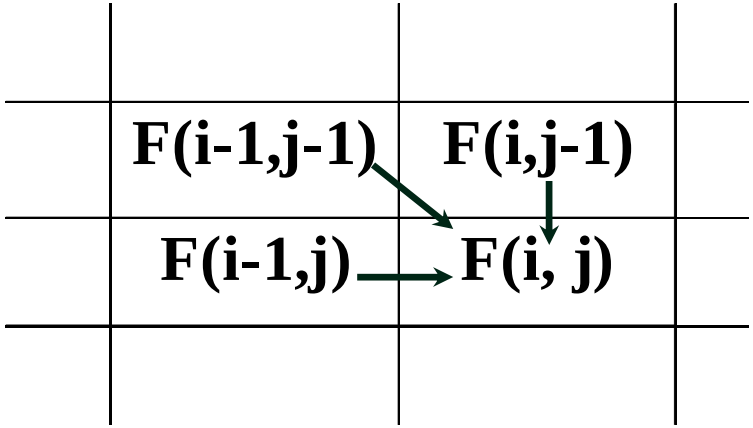
Dadas las secuencias:

S = AAAC

T = AGC



Cálculo de la similitud óptima



$$F(i,j) = \max \begin{cases} F(i-1,j-1) + p(i,j) \\ F(i-1,j) - w \\ F(i,j-1) - w \end{cases}$$

w=Penalización del Gap

Llenado de la Matriz

- Se llena la matriz la primera fila y la primera columna con múltiplos de la penalidad por espacio.
- Se llenan los elementos interiores

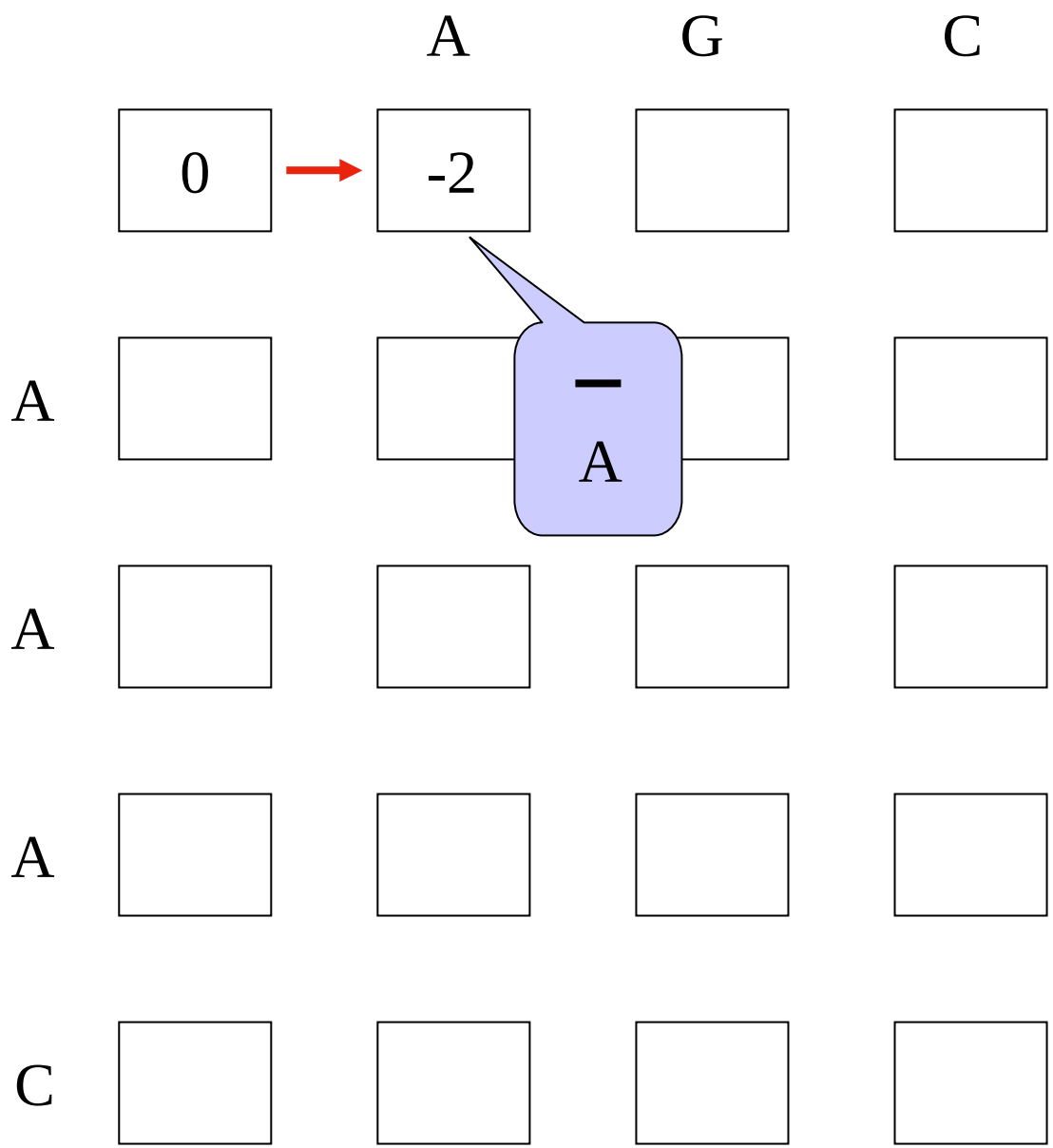
$$\mathbf{F[i,j]} = \mathbf{max} \begin{cases} F[i, j-1] + g \\ F[i-1, j-1] + p(i,j) \\ F[i-1, j] + g \end{cases}$$

- g = penalidad por espacio, p(i,j) = valor del matching entre S(i) y T(j)
- Sigue cualquier orden que permita tener los valores F[i, j-1], F[i-1, j-1], F[i-1, j]. Como fila por fila (de izquierda a derecha) o columna por columna (de arriba hacia abajo).

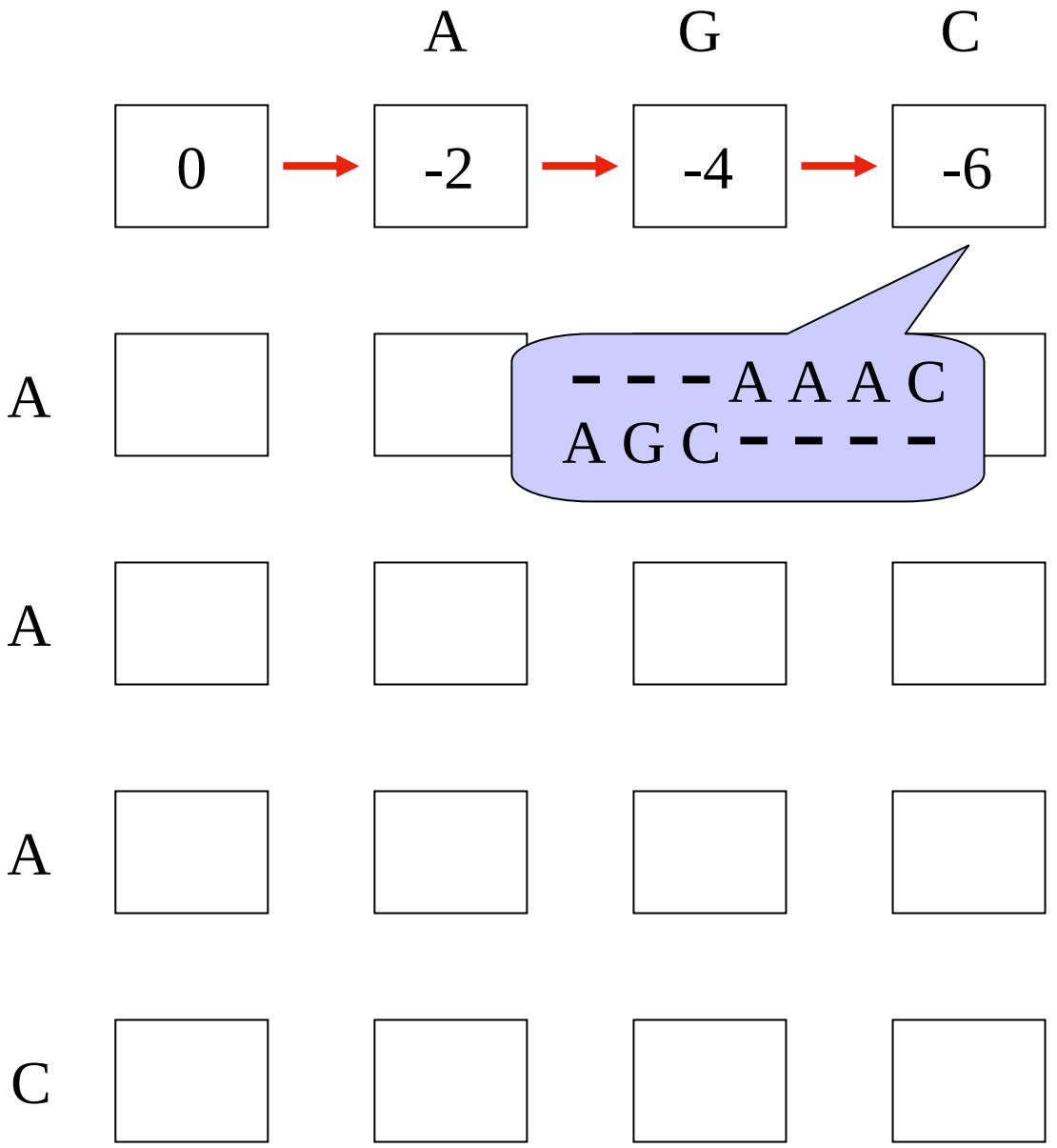
Algoritmo

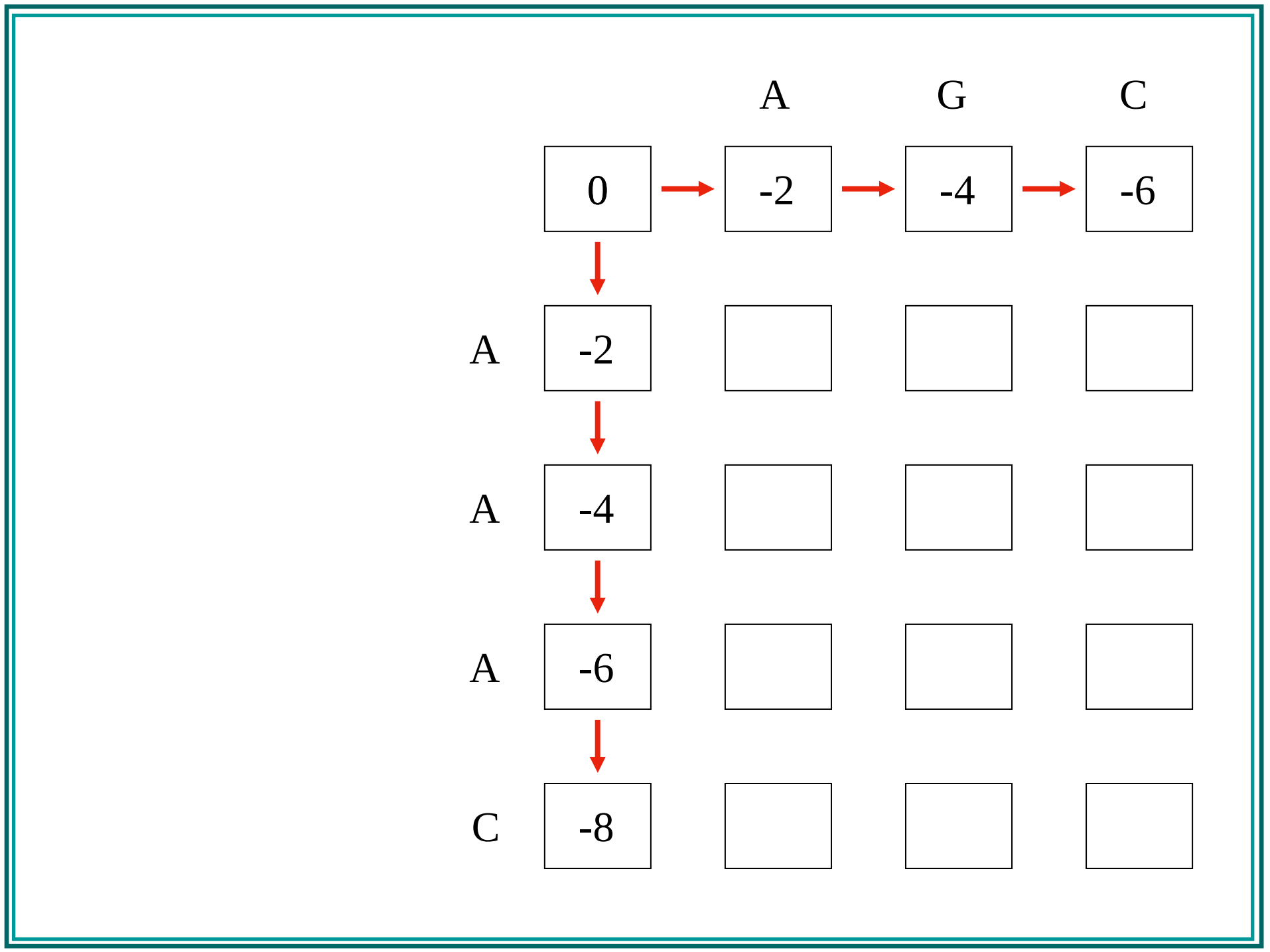
```
n = long(S)
m = long(t)
Para i = 0 hasta n hacer
        F[i, 0] = i  g
Para i = 0, m hacer
        F[0, i] = i  g
Para i = 1, n hacer
        Para j = 1, m hacer
                F[i, j] = max (F[i-1, j] + g, F[i-1, j-1]
                        + p(i,j), F[i, j-1] + g)
Return F
```

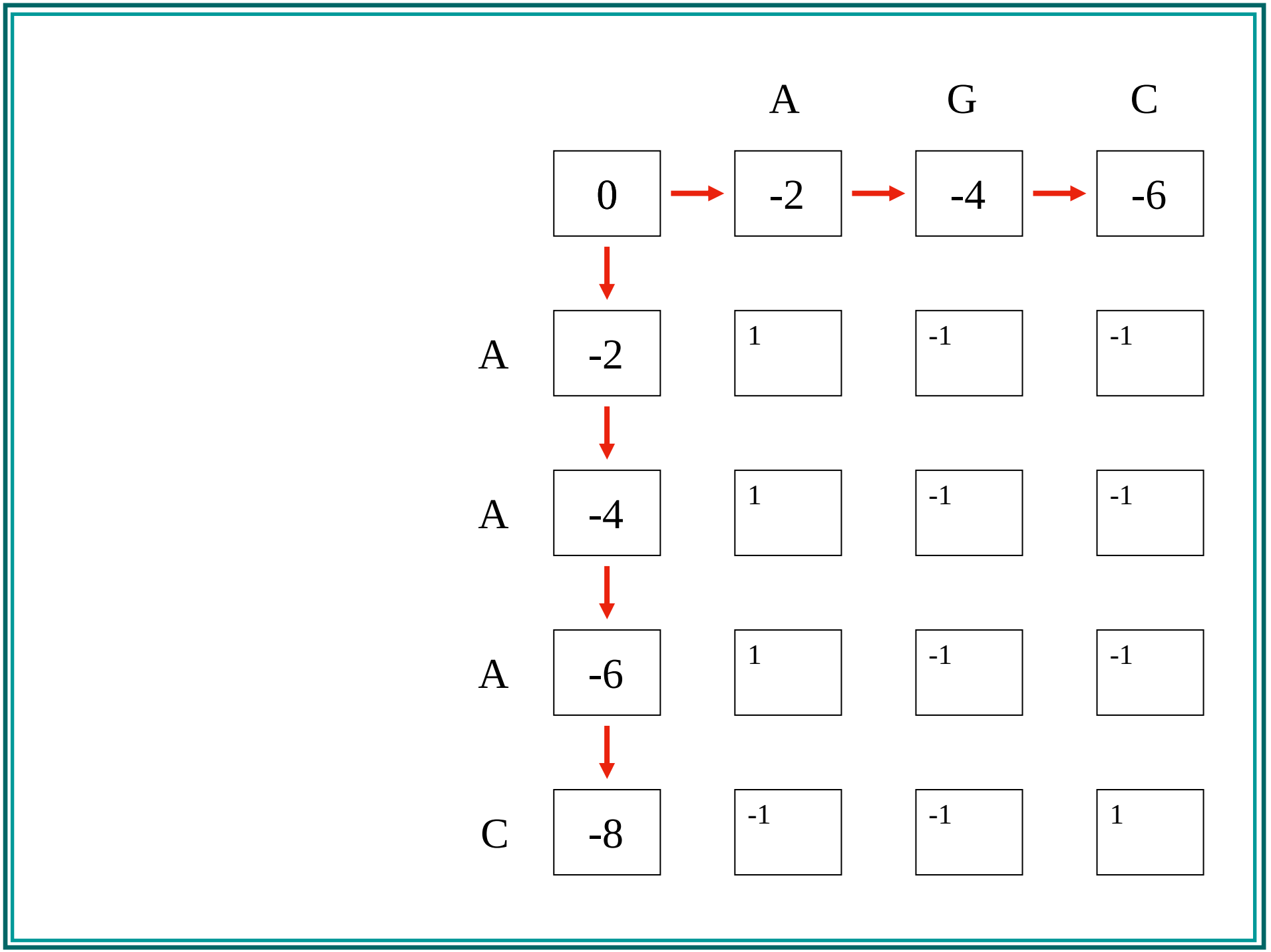
Inicio de la matriz:



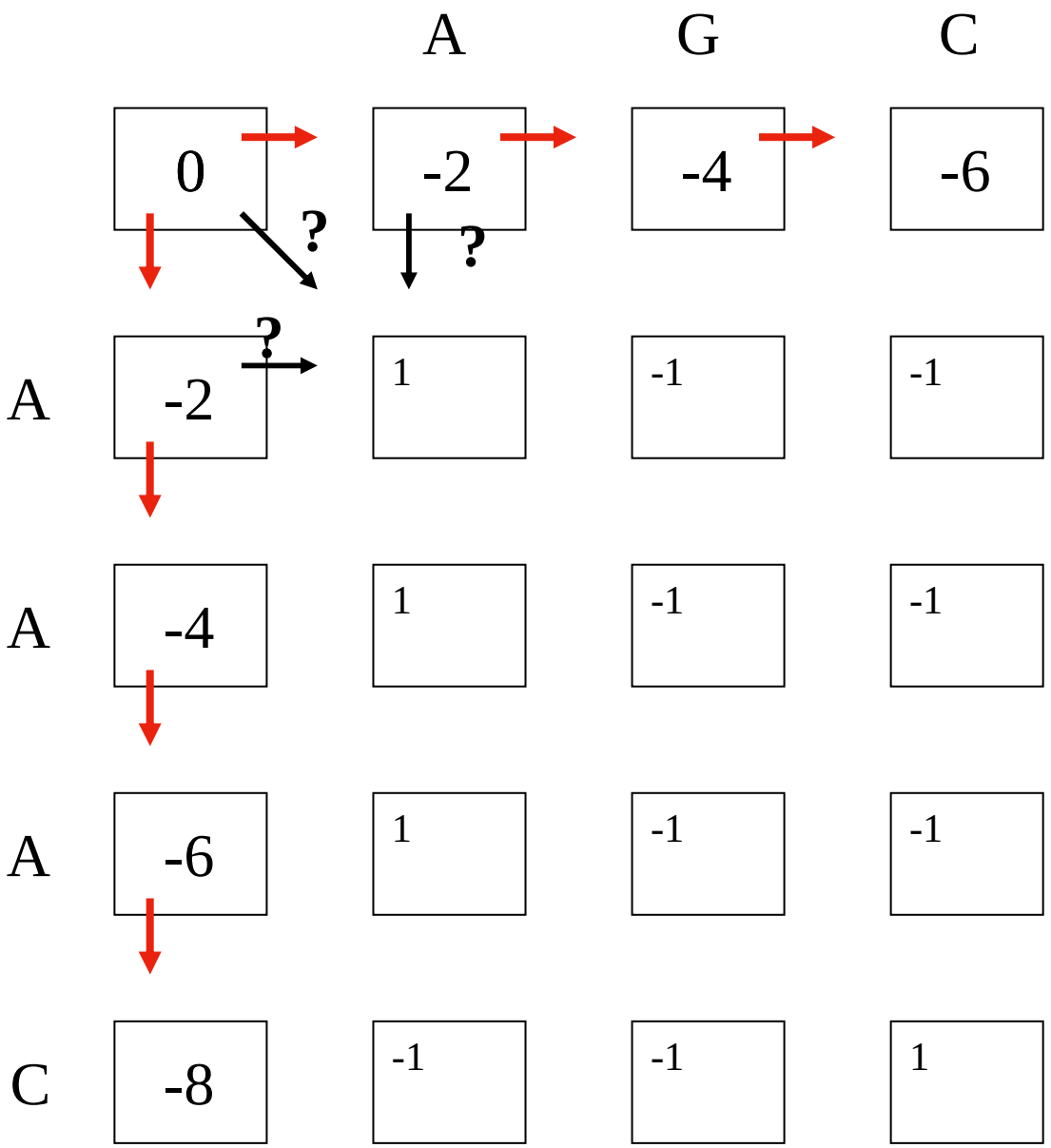
Inicio de la matriz:



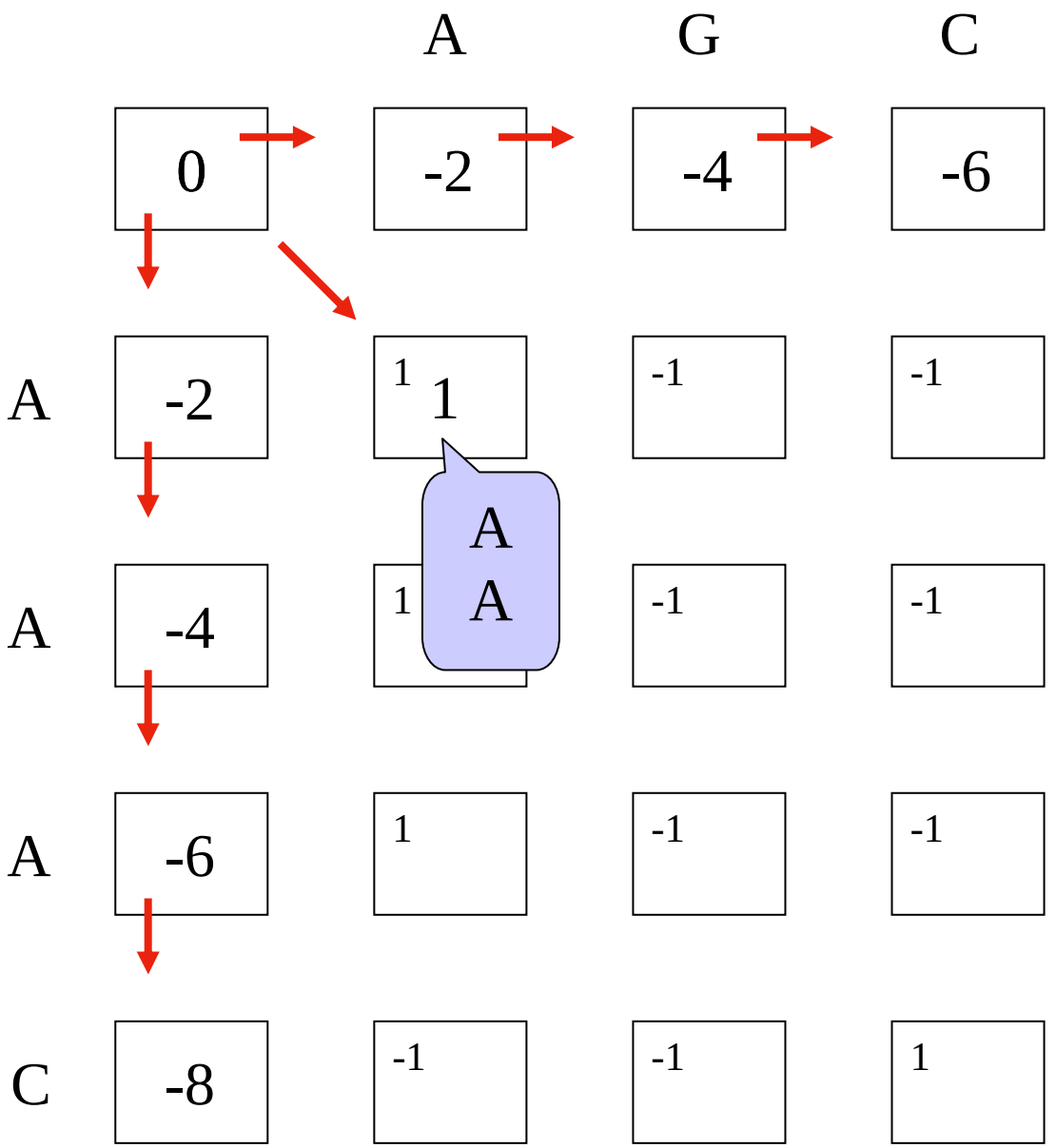

A
G
C
0
-2
-4
-6
A
-2
A
-4
A
-6
C
-8


A
G
C
0
-2
-4
-6
A
-2
1
-1
-1
A
-4
1
-1
-1
A
-6
1
-1
-1
C
-8
-1
-1
1

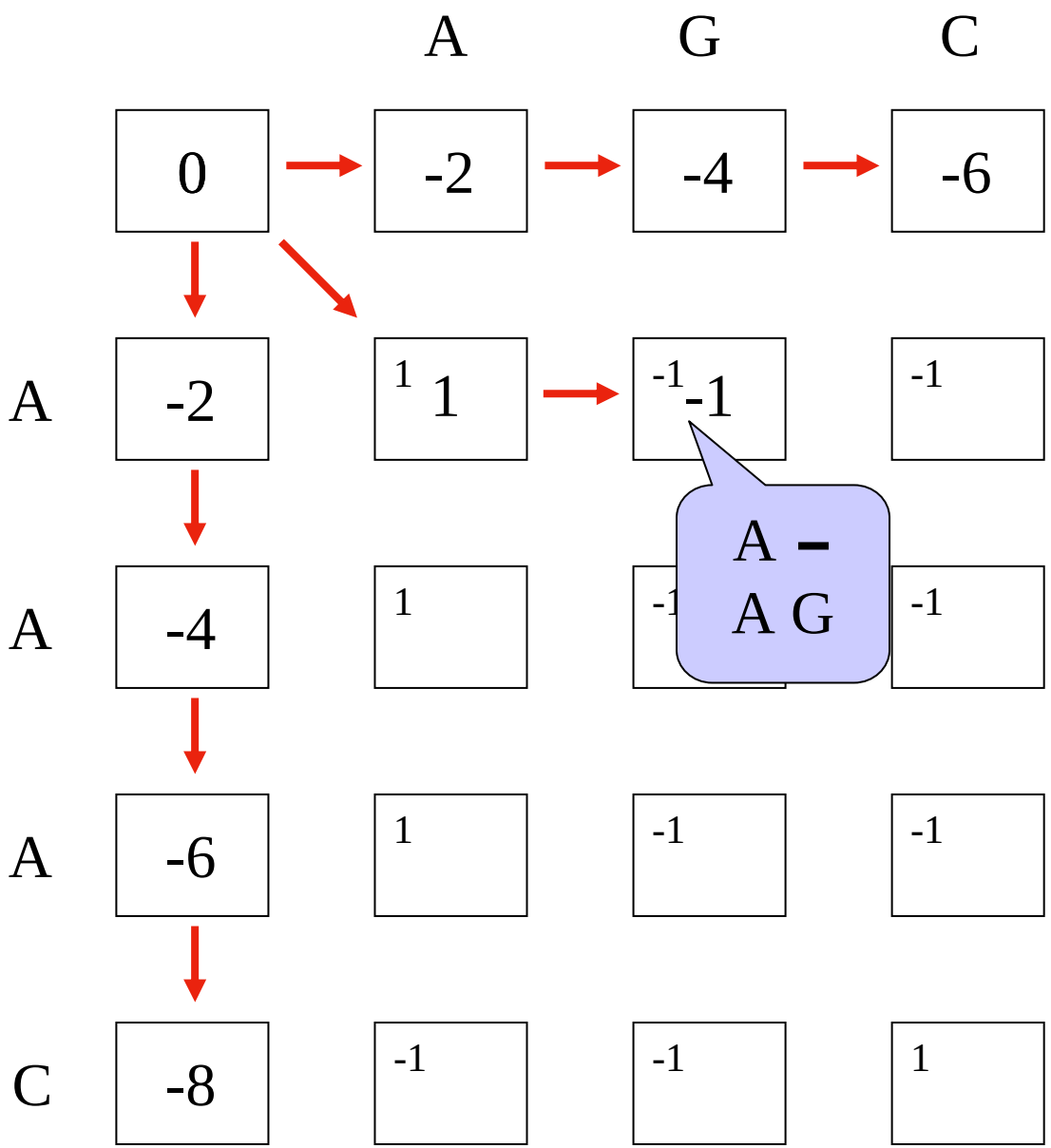
$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(i,j) \\ F(i-1, j) - w \\ F(i, j-1) - w \end{cases}$$



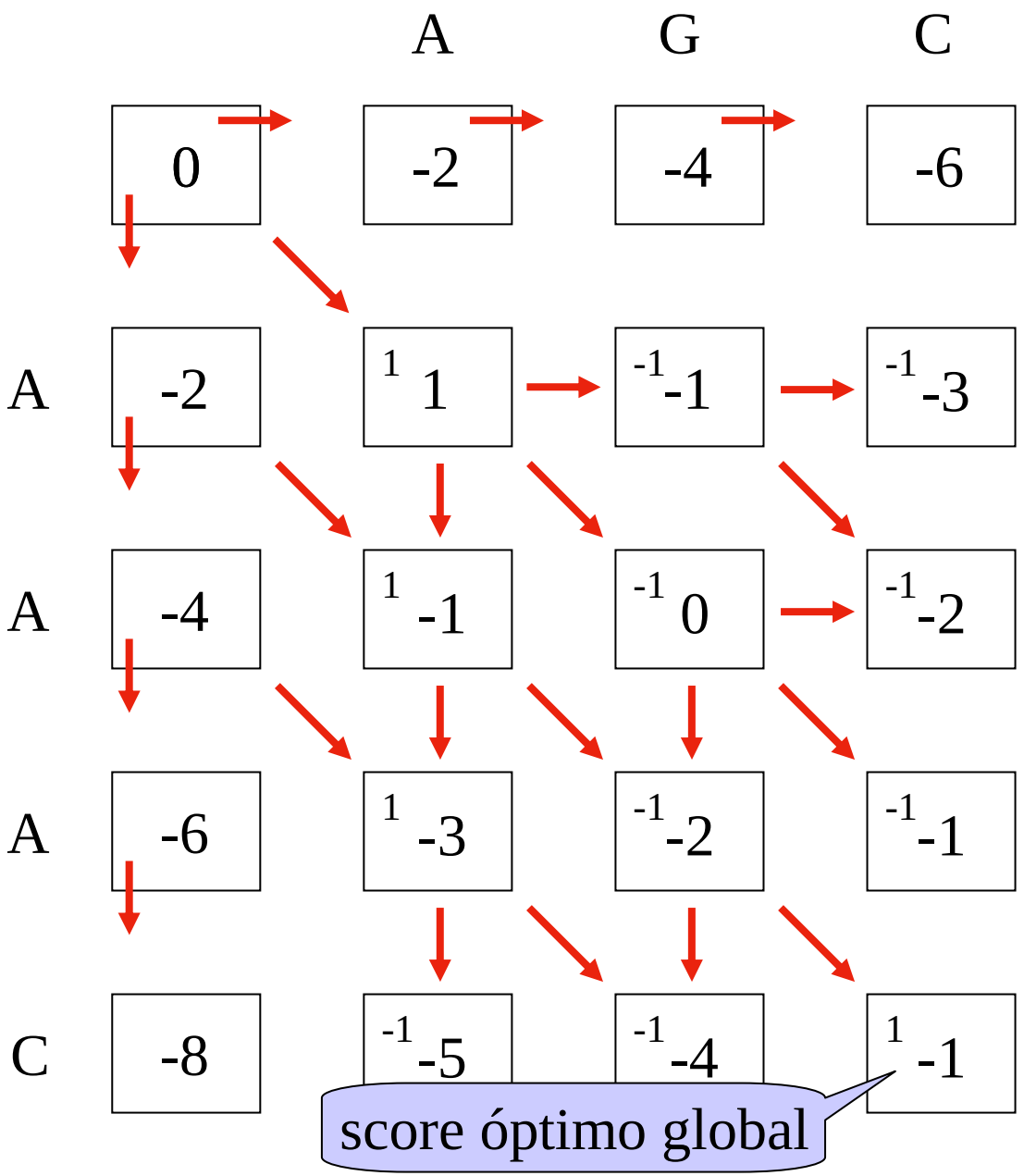
$$F(i,j) = \max \begin{cases} F(i-1,j-1)+s(i,j) \\ F(i-1,j)-w \\ F(i,j-1)-w \end{cases}$$



$$F(i,j) = \max \begin{cases} F(i-1,j-1)+s(i,j) \\ F(i-1,j)-w \\ F(i,j-1)-w \end{cases}$$

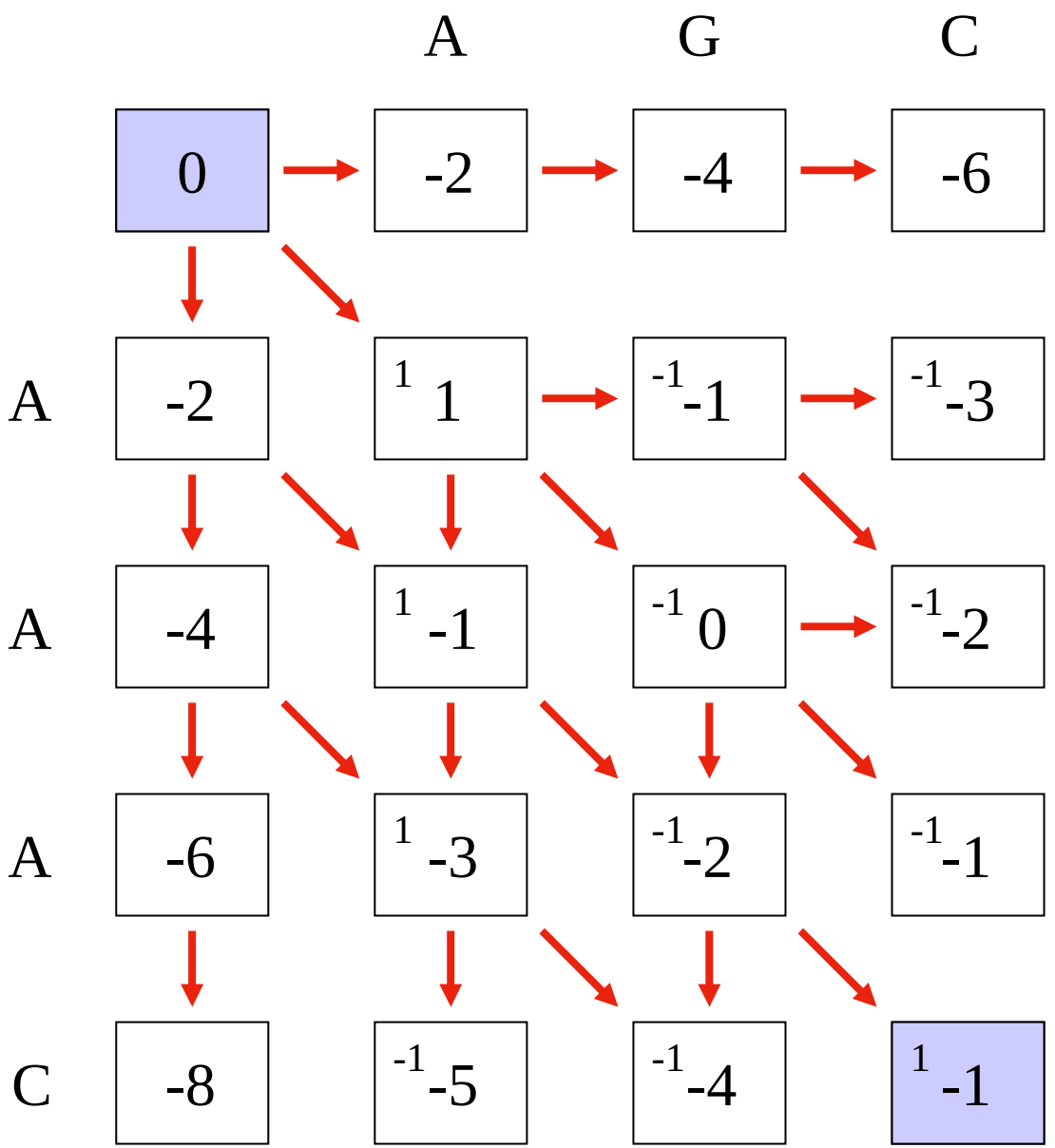


$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(i,j) \\ F(i-1, j) - w \\ F(i, j-1) - w \end{cases}$$



Tenemos el valor de la similitud entre las secuencias.

Ahora necesitamos reconstruir el alineamiento

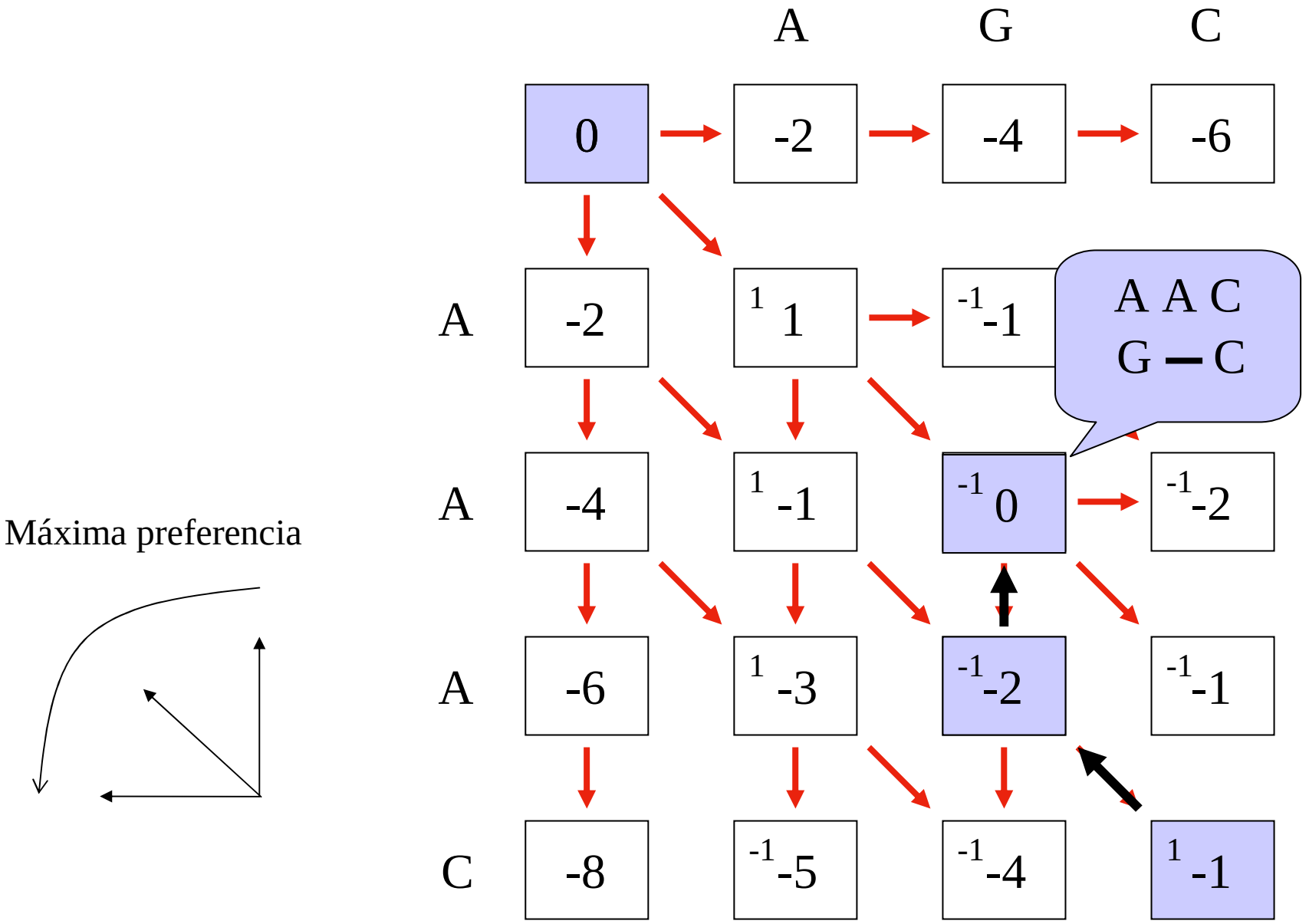


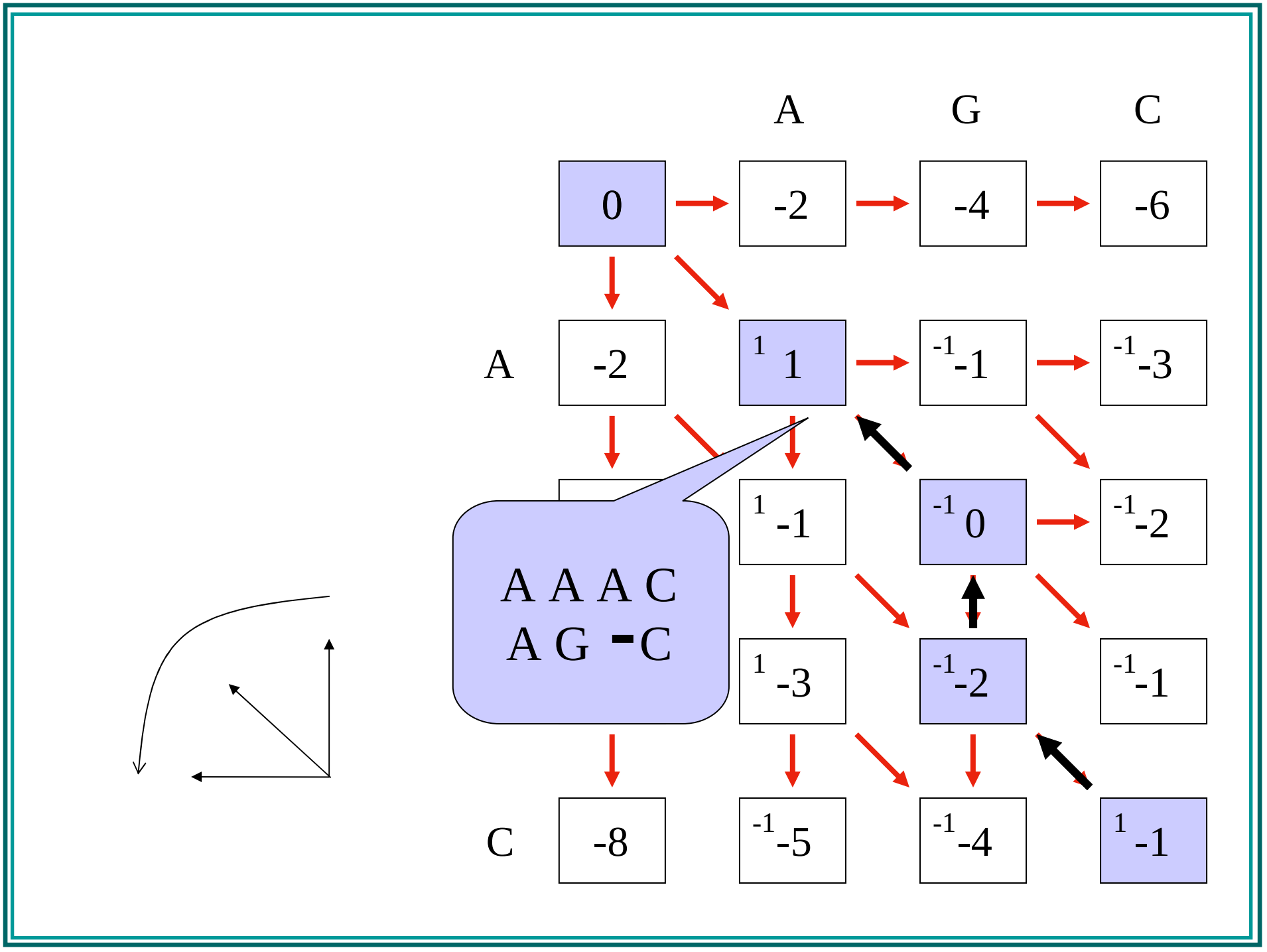
Determinación del alineamiento óptimo

- Se comienza desde el valor de F[n,m] y se realiza el camino inverso hasta llegar al valor F[0,0].
- Por cada valor F[i,j] se testea si provino de F[i-1, j-1] + p(i,j) o F[i-1, j] – g o F[i, j-1] – g.

Hay que tomar la última coincidencia del alineamiento y comenzar a buscar un camino hacia atrás que maximice la función, es decir siguiendo los valores más altos.

Puede haber empates. En ese caso cualquier camino que se tome conducirá a la formación de un alineamiento de score óptimo.


A
G
C
0
-2
-4
-6
A
-2
1 1
-1 -1
A A C
G — C
A
-4
1 -1
-1 0
-1 -2
A
-6
1 -3
-1 -2
-1 -1
C
-8
-1 -5
-1 -4
1 -1
Máxima preferencia


A
G
C
0
-2
-4
-6
A
-2
1 1
-1 -1
-1 -3
1 -1
-1 0
-1 -2
A A A C
A G -C
1 -3
-1 -2
-1 -1
C
-8
-1 -5
-1 -4
1 -1

Algoritmo de programación dinámica para el problema de la mochila

Trabajaremos con la siguiente versión del problema de la mochila.

Problema P(n, k): dado un entero k y n items i de diferentes tamaños $k_{i,}$ encontrar un subconjunto de los items tal que sumen exactamente K o determinar que no existe tal conjunto.

Cómo se expresa el principio de optimalidad acá?.

Algoritmo knapsack

Input: S un array con los tamaños de cada item y K.

Output: Un arreglo de dos dimensiones tal que P[i,k].exist = true si existe una solución con los primeros i elementos para una mochila de tamaño k y P[i,k].belong = true si el elemento i pertenece a esa solución.

Algoritmo

Empezar

```
P[0,0] exist = true
para k = 1,K hacer
    P[0,k].exist = false
para i =1,n hacer
    para k = 0,K hacer
      P[i,k]. exist = false
      if P[i-1,k]. exist  then
           P[i,k]. exist = true
          P[i,k]. belong = false
      else if  k-S[i] ≥ 0
         if P[i-1,k-S[i]]. exist  then
             P[i,k]. exist = true
             P[i,k]. belong = true
```

end

Ejemplo: aplicar el algoritmo anterior a 4 items de tamaño 2,3,5 y 6 tamaños de la mochila desde 2 a 16.

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
5	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
6	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

Referencias de la tabla:

I : se encontró una solución que contiene este item

O: se encontró una solución sin este item

- : no hay solución hasta el momento (si este símbolo aparece en la última línea quiere decir que no hay solución para este tamaño de mochila)

Ejercicio: Pensar un algoritmo de programación dinámica con ideas similares a este para el problema de la mochila más general, es decir cuando queremos resolver

$$\text{Max} \ \ \Sigma_j \, c_j \, x_j$$

sujeto a que

$$\Sigma_j \, a_j \, x_j \leq b$$

$$x_j \in \{0,1\}$$

Subsecuencia creciente más larga

Determinar la subsecuencia creciente más larga de una sucesión de números.

Ejemplo:

S = { 9,5,2,8,7,3,1,6,4)

Las subsecuencias más largas son {2,3,4} o {2,3,6 }

Para construir un algoritmo de programación dinámica definimos:

l_i = *longitud de la secuencia creciente más larga que termina con el número* s_i

p_i = *predecesor de* s_i *en la secuencia creciente más larga que termina con el número* s_i

Vale el principio de optimalidad en este caso?. Cómo se expresaría?.

Relación de recurrencia:

$$l_0 = 0$$

$$l_i = \max_{j<i} \; l_j \; +1, \qquad \text{con } s_j < s_i$$

La solución al problema la da

$$\max_{1 \le i \le n} l_i$$

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3
predecesor	-	-	-	2	2	2	-	3	3

Complejidad?

$O(n^2)$ (se podría implementar en O(nlogn))

Cuál es la complejidad espacial?.

Cómo hacemos para tener también la sucesión y no sólo la longitud?.

Arboles de búsqueda óptimos

- árbol binario de búsqueda: *cada nodo contiene una clave, y el valor contenido en cada nodo interno es mayor o igual (numérica o lexicográficamente) que él de sus descendientes por la rama izquierda y menor o igual que los de la derecha.*

- Cuántos árboles binarios de búsqueda distintos puede haber con un número dado n de palabras código (o clave)?.

- Cómo podemos determinar el árbol binario que minimiza el número promedio de comparaciones necesarias para encontrar una clave.

- Supongamos que tenemos un conjunto $c_1 < c_2 < \ldots.. < c_n$ de claves con probabilidad p_i de ser buscadas.

  Si la clave c_i está en el nivel d_i entonces hacen falta $d_i + 1$ comparaciones para encontrarla.

- Suponemos $\sum p_i = 1$

Entonces para un árbol dado la cantidad de comparaciones que tenemos que hacer en media para buscar una clave es

$$C = \sum_i p_i (d_i + 1)$$

Dado un conjunto de claves y sus probabilidades queremos construir un árbol que minimice esta función, usando programación dinámica.

Se cumple el principio de optimalidad?.

“ en un árbol óptimo todos sus subárboles son óptimos para las claves que contienen”

Ejercicio: pensar una forma de determinar un árbol óptimo usando programación dinámica.

Hay otros métodos para resolver este problema (Algoritmo de Huffman)

Relación de recurrencia?

Sea:

- $m_{ij} = \sum_{k=i}^{j} p_k$

(probabilidad de que una clave esté en la sucesión $c_i, c_{i+1}, \ldots\ldots c_j$)

- C_{ij} número de comparaciones promedio que se hacen en un subárbol óptimo que contiene las claves $c_i, c_{i+1}, \ldots\ldots c_j$, cuando buscamos en el árbol completo.

- $C_{ij} = 0$ para $j = i-1$

- C^{k}_{ij} número de comparaciones promedio en un subárbol que contiene las claves $c_i, c_{i+1}, \ldots\ldots c_j$, y tiene a la clave k en la raíz. (*siempre pensando que buscamos una clave en todo el árbol)*

Si la clave k es la raíz de ese subárbol podemos calcular:

$$C^{k}_{ij} = m_{ij} + C_{i,k-1} + C_{k+1,j}$$

o sea dado un subárbol con raíz k:

- *la probabilidad de que la clave buscada se una de las del subárbol o sea alguna de las c_i, c_{i+1} ,……c_j es m_{ij}*
- *una comparación se hace con c_k y el resto en el árbol derecho o en el árbol izquierdo*

Y si queremos que el subárbol sea optimo la raíz tiene que ser elegida de modo a minimizar

$$C_{ij} = m_{ij} + \min_{i \le k \le j} \{C_{i,k-1} + C_{k+1,j}\}$$

Ejemplo:

i	1	2	3	4	5
p_i	0.30	0.05	0.08	0.45	0.12

0.30	0.35	0.43	0.88	1
	0.05	0.13	0.58	0.70
		0.08	0.53	0.65
			0.45	0.57
				0.12

Calculamos la matriz de los m_{ij}

A partir de esto podemos calcular los C_{ij} usando la fórmula de recurrencia y obtener el mínimo número de comparaciones promedio necesarias del árbol óptimo.

0.30	0.40	0.61	1.49	1.73
0	0.05	0.18	0.76	1.00
	0	0.08	0.61	0.85
		0	0.45	0.69
			0	0.12

Cómo hacemos para determinar el árbol óptimo?

- *La técnica de Programación Dinámica se aplica también a problemas con variables continuas*

  (los ejemplos que vimos acá fueron todos con variables discretas).

- *En el caso de variables continuas puede ser necesario resolver en el marco de las ecuaciones de recurrencia, problemas de maximización o minimización continuos, usando las herramientas habituales del cálculo u otras.*