Programación Dinámica

- Es aplicada típicamente a problemas de optimización, donde puede haber muchas soluciones, cada una tiene un valor asociado y prentendemos obtener la solución con valor óptimo. Al igual que "dividir y conquistar", el problema es dividido en subproblemas de tamanños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problemas original.

Principio de optimalidad

Principio de optimalidad:
Un problema de optimización satisface el principio de **optimalidad de Bellman** si en una sucesión óptima de decisiones o elecciones, cada subsucesión es a su vez óptima. Es decir, si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución.

Ejemplos donde se cumple o no el principio de optimalidad

- Camino mínimo

Ejemplos donde se cumple o no el principio de optimalidad

- Camino mínimo
- Camino máximo

Programación Dinámica: ejemplos

- Coeficientes binomiales $\binom{n}{k}$
- Producto de matrices
- Subsecuencia creciente máxima
- Comparación de secuencias de ADN
- etc.

Coeficientes binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Función recursiva ("dividir y conquistar")
  complejidad

Coeficientes binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Función recursiva ("dividir y conquistar")
  complejidad
  $\Omega(\binom{n}{k})$
- Programación dinámica.
  La misma función recursiva pero SIN repetir los calculos.

Coeficientes binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Función recursiva ("dividir y conquistar")
  complejidad
  $\Omega(\binom{n}{k})$
- Programación dinámica.
  La misma función recursiva pero SIN repetir los calculos.
- Hay que guardar toda la matriz ? cuánta memoria requiere ?
  qué complejidad tiene ?

Coeficientes binomiales

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Función recursiva ("dividir y conquistar")
  complejidad
  $\Omega(\binom{n}{k})$
- Programación dinámica.
  La misma función recursiva pero SIN repetir los calculos.
- Hay que guardar toda la matriz ? cuánta memoria requiere ?
  qué complejidad tiene ?
- Complejidad $O(nk)$

Multiplicación de n matrices

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

Debido a la propiedad asociativa del producto de matrices, puede hacer la multiplicación de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5, B de 5×89, C de 89×3 y D de 3×34. Tenemos

Multiplicación de n matrices

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

Debido a la propiedad asociativa del producto de matrices, puede hacer la multiplicación de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5, B de 5×89, C de 89×3 y D de 3×34. Tenemos

- $((AB)C)D$ requiere 10582 multiplicaciones.

Multiplicación de n matrices

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

Debido a la propiedad asociativa del producto de matrices, puede hacer la multiplicación de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5, B de 5×89, C de 89×3 y D de 3×34. Tenemos

- $((AB)C)D$ requiere 10582 multiplicaciones.
  $13 * 5 * 89 + 13 * 89 * 3 + 13 * 3 * 34 = 5785 + 3471 + 1326$

Multiplicación de n matrices

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

Debido a la propiedad asociativa del producto de matrices, puede hacer la multiplicación de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5, B de 5×89, C de 89×3 y D de 3×34. Tenemos

- $((AB)C)D$ requiere 10582 multiplicaciones.
  $13 * 5 * 89 + 13 * 89 * 3 + 13 * 3 * 34 = 5785 + 3471 + 1326$
- $(AB)(CD)$ requiere 54201 multiplicaciones.

Multiplicación de n matrices

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

Debido a la propiedad asociativa del producto de matrices, puede hacer la multiplicación de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias. Por ejemplo: las dimensiones de A es de 13×5, B de 5×89, C de 89×3 y D de 3×34. Tenemos

- $((AB)C)D$ requiere 10582 multiplicaciones.
  $13 * 5 * 89 + 13 * 89 * 3 + 13 * 3 * 34 = 5785 + 3471 + 1326$
- $(AB)(CD)$ requiere 54201 multiplicaciones.
  $13 * 5 * 89 + 89 * 3 * 34 + 13 * 89 * 34 = 5785 + 9078 + 39338$
- $(A(BC))D$ requiere 2856 multiplicaciones.
- $A((BC)D)$ requiere 4055 multiplicaciones.
- $A(B(CD))$ requiere 26418 multiplicaciones.

Principio de optimalidad

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

$$(A(BC))D$$

Principio de optimalidad

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

$$(A(BC))D$$

$$\mathbf{(A(BC))}\ \ D$$

- El subproblema es: multiplicar las matrices ABC de la mejor manera posible.

Principio de optimalidad

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

$$(A(BC))D$$

$$\mathbf{(A(BC))}\ \ D$$

- El subproblema es: multiplicar las matrices ABC de la mejor manera posible.
- Subsolución ?:

Principio de optimalidad

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

$$(A(BC))D$$

$$\mathbf{(A(BC))}\ \ D$$

- El subproblema es: multiplicar las matrices ABC de la mejor manera posible.
- Subsolución ?: $(A(BC))$. Es óptima para el subproblema?

Principio de optimalidad

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

$$(A(BC))D$$

$$\mathbf{(A(BC))}\ \ D$$

- El subproblema es: multiplicar las matrices ABC de la mejor manera posible.
- Subsolución ?: $(A(BC))$. Es óptima para el subproblema?
- Vale el principio de optimalidad?

Programacion dinámica

- 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

Programacion dinámica

- 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

- $\omega\ (4^n/n)$.

Programacion dinámica

- 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

- $\omega\ (4^n/n)$.
- Sea $M = ((M_1M_2)((M_3M_4)(M_5 \cdots M_i)))(M_{i+1}M_{i+2}(\cdots M_n))$

Programacion dinámica

- 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

- $\omega\ (4^n/n)$.
- Sea $M = ((M_1M_2)((M_3M_4)(M_5 \cdots M_i)))(M_{i+1}M_{i+2}(\cdots M_n))$
- Sea $M = \mathbf{(}((M_1M_2)((M_3M_4)(M_5 \cdots M_i)))\mathbf{)(}M_{i+1}M_{i+2}(\cdots M_n)\mathbf{)}$

Programacion dinámica

- 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

- $\omega\ (4^n/n)$.
- Sea $M = ((M_1M_2)((M_3M_4)(M_5 \cdots M_i)))(M_{i+1}M_{i+2}(\cdots M_n))$
- Sea $M = \mathbf{(}((M_1M_2)((M_3M_4)(M_5 \cdots M_i)))\mathbf{)(}M_{i+1}M_{i+2}(\cdots M_n)\mathbf{)}$

- Función:
- m_{ij} es la cantidad mínima de multiplicaciones necesaria para calcular $M_iM_{i+1}M_{i+2} \cdots M_j$.

Suponemos que las dimensiones de la matriz M_i están dadas por un vector d_i , $0 \leq i \leq n$, donde la matriz M_i tiene d_{i-1} filas y d_i columnas.

- Funcion de prog. dinámica:

Suponemos que las dimensiones de la matriz M_i están dadas por un vector d_i , $0 \leq i \leq n$, donde la matriz M_i tiene d_{i-1} filas y d_i columnas.

- Funcion de prog. dinámica:
- Para cada $2 \leq s \leq n-1$, para $i = 1, 2, n-s$
  $m_{i\ i+s} =$
  $= min_{i \leq k \leq i+s-1} \{\ m_{ik} + m_{(k+1)(i+s)} + d_{i-1} \times d_k \times d_{i+s}\ \}$

Suponemos que las dimensiones de la matriz M_i están dadas por un vector d_i , $0 \leq i \leq n$, donde la matriz M_i tiene d_{i-1} filas y d_i columnas.

- Funcion de prog. dinámica:
- Para cada $2 \leq s \leq n-1$, para $i = 1, 2, n-s$
  $m_{i\ i+s} =$
  $= min_{i \leq k \leq i+s-1} \{\ m_{ik} + m_{(k+1)(i+s)} + d_{i-1} \times d_k \times d_{i+s}\ \}$
- La solucion al problema es: 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 M_i tiene d_{i-1} filas y d_i columnas.

- Funcion de prog. dinámica:
- Para cada $2 \leq s \leq n-1$, para $i = 1, 2, n-s$
  $m_{i\ i+s} =$
  $= min_{i \leq k \leq i+s-1} \{\ m_{ik} + m_{(k+1)(i+s)} + d_{i-1} \times d_k \times d_{i+s}\ \}$
- La solucion al problema es: m_{1n}

- Llenado de la matriz:

- Llenado de la matriz: Por cada diagonal $s, s = 0, ..., n-1$
  $m_{ii} = 0$, $(s = 0)$, Para $i = 1 \cdots n$
  $m_{i\ i+1}$, $(s = 1)$ $i = 1 \cdots n-1$
- Complejidad:

- Llenado de la matriz: Por cada diagonal $s, s = 0, ..., n-1$
  $m_{ii} = 0$, $(s = 0)$, Para $i = 1 \cdots n$
  $m_{i\ i+1}$, $(s = 1)$ $i = 1 \cdots n-1$
- 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:
  $\sum(n-s)s = n\sum s - \sum s^2 =$
  $n^2(n-1)/2 - n(n-1)(2n-1)/6 = (n^3 - n)/6$
  Es decir, $O(n^3)$

Subsecuencia creciente más larga

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

- Ejemplo: $S = \{9, 5, 2, 8, 7, 9, 3, 1, 6, 4\}$

Subsecuencia creciente más larga

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

- Ejemplo: $S = \{9, 5, 2, 8, 7, 9, 3, 1, 6, 4\}$
- Las subsecuencias más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$ $\{5, 7, 9\}$

Subsecuencia creciente más larga

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

- Ejemplo: $S = \{9, 5, 2, 8, 7, 9, 3, 1, 6, 4\}$
- Las subsecuencias más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$ $\{5, 7, 9\}$
- Vale el ppio de optimalidad?

Subsecuencia creciente más larga

- Para construir un algoritmo de programación dinámica definimos otro problema relacionado:
  l_i = longitud de la secuencia creciente mas larga que termina con s_i

Subsecuencia creciente más larga

- Para construir un algoritmo de programación dinámica definimos otro problema relacionado:
  l_i = longitud de la secuencia creciente mas larga que termina con s_i
- Función:
  $l_0 = 0$
  $l_i = max_{j \leq i-1}\{l_j + 1\}$ para los j tales que $s_j \leq s_{i-1}$
- Solución: Máximo l_i

Subsecuencia creciente más larga

Demostremos que esa funcion calcula efectivamente lo que queremos:
L_i: longitud de la secuencia creciente más larga que termina con s_i
p_i = predecesor de s_i en la secuencia creciente más larga que termina con s_i
Queremos ver que $L_i = l_i$

Subsecuencia creciente más larga

- Complejidad temporal y espacial

Subsecuencia creciente más larga

- Complejidad temporal y espacial
- Como hacemos para tener tambien la sucesión y no solo la longitud?.

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?

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).

Comparación de secuencias de ADN

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

GA -CGGATTAG

GATCGGAATAG

Comparación de secuencias de ADN

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

GA -CGGATTAG

GATCGGAATAG

- coincidencia $= +1$
  diferencia $=$ -1
  gap $=$ -2
- Puntaje(T, S) $= 9\ 1 + 1 \times$ (-1) $+ 1 \times$ (-2) $= 6$

Comparación de secuencias de ADN

- Principio de optimalidad ?

Comparación de secuencias de ADN

- Principio de optimalidad ?
  - GACGGATTAG -

    XXXXXXXXXX

Comparación de secuencias de ADN

- Principio de optimalidad ?
  - GACGGATTAG -

    XXXXXXXXXX
  - GACGGATTAG

    XXXXXXXXX -

Comparación de secuencias de ADN

- Principio de optimalidad ?
  - GACGGATTAG -

    XXXXXXXXXX
  - GACGGATTAG

    XXXXXXXXX -
  - GACGGATTAG GACGGATTA**G**

    XXXXXXXXX X o XXXXXXXXX**G**

Comparación de secuencias de ADN

- Se quiere alinear las secuencias S y T.
- Sea n el tamao de S y m el de T.
  - - - - - - - - - - - - - n
  - - - - - - - - - - - - - - - - - m
- $F(i, j)$: mejor similitud entre las subsecuencias $S[1...i]$ y $T[1...j]$. Sol: $F(n, m)$

Comparación de secuencias de ADN

- $F(i, j) =$

Comparación de secuencias de ADN

- $F(i,j) =$
$$Max \begin{cases} F(i,j-1) + p_g \\ F(i-1,j) + p_g \\ F(i-1,j-1) + p_d \quad (o\ F(i-1,j-1) + p_c) \end{cases}$$
- Donde, p_g, p_c, p_d son los puntos por Gap, coicidencia, y diferencia, resp.

Comparación de secuencias de ADN

- $F(i,j) =$
  $$Max \begin{cases} F(i,j-1) + p_g \\ F(i-1,j) + p_g \\ F(i-1,j-1) + p_d \quad (o\ F(i-1,j-1) + p_c) \end{cases}$$
- Donde, p_g, p_c, p_d son los puntos por Gap, coicidencia, y diferencia, resp.
- Llenado de la matriz: Una matriz de $(n+1) \times (m+1)$
  $F(i,0) = p_g \times i$
  $F(0,j) = p_g \times j$

Comparación de secuencias de ADN

- Rearmar la solución
- Complejidad Temporal:

Comparación de secuencias de ADN

- Rearmar la solución
- Complejidad Temporal:
  $O(mn)$ en armar la matriz
  $O(m+n)$ en buscar en la matriz (recorrer la secuencia resultante, que a lo sumo tiene $n+m$ elementos).
- Complejidad Espacial: $O(mn)$ por el espacio necesario para la matriz

Problema de la mochila

- Problema $M(n, k)$: Sea k un entero, y n items, cada uno de peso p_i. Encontrar un subconjunto de items tal que su peso sume exáctamente k, si existe.
  Es decir, $\sum_j p_j x_j$, $x_j \in \{0, 1\}$
- Principio de optimalidad:

Problema de la mochila

- Problema $M(n, k)$: Sea k un entero, y n items, cada uno de peso p_i. Encontrar un subconjunto de items tal que su peso sume exáctamente k, si existe.
  Es decir, $\sum_j p_j x_j$, $x_j \in \{0, 1\}$
- Principio de optimalidad: dada una solución del problema, $a_{i_1}, \cdots a_{i_s}$ entonces $a_{i_1}, \ldots a_{i_{s-1}}$ .....

Problema de la mochila

- $P(i, j)$: Vale 1 si existe una solución entre los primeros i elementos, que sumen j y 0 en caso contrario

Problema de la mochila

- $P(i, j)$: Vale 1 si existe una solución entre los primeros i elementos, que sumen j y 0 en caso contrario
- Función: $M(i, j) = Max\{M(i-1, j), M(i-1, j-a_i)\}$

Problema de la mochila

- $P(i, j)$: Vale 1 si existe una solución entre los primeros i elementos, que sumen j y 0 en caso contrario
- Función: $M(i, j) = Max\{M(i-1, j), M(i-1, j-a_i)\}$
- Llenado de la matriz

Problema de la mochila

- $P(i, j)$: Vale 1 si existe una solución entre los primeros i elementos, que sumen j y 0 en caso contrario
- Función: $M(i, j) = Max\{M(i-1, j), M(i-1, j-a_i)\}$
- Llenado de la matriz $M(1, j) = \{a_1 = j\}$
  $M(i, 1) = Max\{M(i-1, 1), \{a_i = 1\}\}$

Problema de la mochila

- $P(i, j)$: Vale 1 si existe una solución entre los primeros i elementos, que sumen j y 0 en caso contrario
- Función: $M(i, j) = Max\{M(i-1, j), M(i-1, j-a_i)\}$
- Llenado de la matriz $M(1, j) = \{a_1 = j\}$
  $M(i, 1) = Max\{M(i-1, 1), \{a_i = 1\}\}$
- Complejidad

Problema de la mochila

- Ejercicio: Pensar un algoritmo de programacion dinamica con ideas similares a este para el problema de la mochila mas general:
  Max $\sum_j c_j x_j$
  sujeto a que
  $\sum_j a_j xj \leq k$
  $x_j \in \{0, 1\}$