Departamento de Computación
FCEFQyN, Universidad Nacional de Río Cuarto
Asignaturas: Diseño de Algoritmos - Algoritmos II
Primer Cuatrimestre de 2025

Guía Práctica No. 3: Programación Dinámica y Memoization

Esta guía práctica abarca las técnicas Programación Dinámica y Memoization. Encontrará material relacionado en el capítulo 8 de *Introduction to the Design and Analysis of Algorithms* (Levitin 2003), que es lectura recomendada para el desarrollo de la práctica.

Parte de esta práctica cuenta con esquemas de programación y tests de asistencia a su resolución.

Esta práctica no tiene entrega formal. Su resolución es opcional.

1. Qué tienen en común y en qué se diferencian las técnicas *Divide & Conquer/Decrease & Conquer* y *Programación Dinámica*?

2. Dado un problema P , y una solución recursiva S (correcta) para el mismo. Siempre se puede llevar S a una solución basada en Programación Dinámica? Justifique su respuesta.

3. Implemente en Java el algoritmo basado en *Programación Dinámica* para calcular el n-ésimo número de Fibonacci.

4. Considere el problema de calcular el número combinatorio $\binom{n}{m}$. El algoritmo recursivo natural para el cálculo de $\binom{n}{m}$ sufre de algunos problemas de eficiencia. Intente mejorar este algoritmo utilizando *Programación Dinámica*. Compare empiricamente este algoritmo con la versión recursiva original.

5. El n-ésimo número de *Catalan* está definido de la siguiente manera:

$$C_n = \sum_{k=1}^{n} (C_{k-1} \times C_{n-k})$$

   con la salvedad de que $C_0 = 1$. Escriba un programa recursivo en java que calcule el n-ésimo número de Catalan. Corra su programa utilizando los tests provistos, si su solución sufre de problemas de eficiencia, escriba una solución que mejore este algoritmo utilizando *Programación Dinámica*. ¿Es su solución polinomial con respecto a n?.

6. Considere el problema de determinar el orden óptimo en el cual conviene multiplicar una secuencia de n matrices $A_1, A_2..A_n$, cuyas dimensiones son $d_0 \times d_1, d_1 \times d_2..d_{n-1} \times d_n$ respectivamente. El objetivo es encontrar la forma de parentizar el producto de manera tal que minimice el número total de multiplicaciones necesarias.

   El costo de multiplicar una matriz de dimensión $p \times q$ con una matriz de dimensión $q \times r$ es $p \times q \times r$

   Ejemplo: Sea A_1, A_2 y A_3, 3 matrices de las siguientes dimensiones:

   $A_1 : 10 \times 100$
   $A_2 : 100 \times 5$
   $A_3 : 5 \times 50$

   Podemos parentizar de do maneras distintas:

   (a) $((A_1 \times A_2) \times A_3)$ $\quad costo = 7500$
   (b) $(A_1 \times (A_2 \times A_3))$ $\quad costo = 75000$

(a) Se computa $A_1 \times A_2$ con costo $10 \times 100 \times 5 = 5000$ y se obtiene una matriz A_i de dimensiones 10×5, luego se computa $A_i \times A_3$ con costo $10 \times 5 \times 50 = 2500$, por lo tanto el costo total es $5000 + 2500 = 7500$

(b) Se computa $A_2 \times A_3$ con costo $100 \times 5 \times 50 = 25000$ y se obtiene una matriz A_i de dimensiones 100×50, luego se computa $A_1 \times A_i$ con costo $10 \times 100 \times 50 = 50000$, por lo tanto el costo total es $25000 + 50000 = 75000$

A partir de una solución recursiva para este problema, diseñe un algoritmo para resolver este problema utilizando la técnica de *Programación Dinámica*. Implemente este algoritmo en Java, y compare empiricamente su programa con la solución original.

7. Retome los ejercicios 4 y 5 e implemente versiones utilizando *Memoization*.

8. Sea $M = \{m_0, m_1, ..., m_n\}$ un conjunto finito de valores de monedas, tales que $m_i \in Nat \wedge m_i > 0$. Suponiendo que la cantidad disponible de monedas de cada tipo es ilimitada, se desea determinar la forma óptima, con respecto al número mínimo de monedas, de dar vuelto por $C > 0$ centavos. Diseñe un algoritmo *Divide & Conquer* que resuelva el problema, y en caso de ser necesario (por razones de eficiencia) diseñe una solución usando *Programación Dinámica* alternativa. Escriba test para probar y comparar empiricamente sus soluciones.

9. Sean $p_1, p_2, .., p_n$ los precios iniciales de n vinos, acomodados en un estante de una vinoteca, donde el vino i-ésimo se corresponde con p_i. Sin embargo, el precio de los vinos aumenta con el tiempo. Cada año se puede vender solamente un vino, o el primero (el más a la izquierda), o el último (el más a la derecha) vino de la lista. Supongamos que es el año 1, en el año Y, la ganancia del i-ésimo vino será $Y * p_i$. Se desea calcular la ganancia máxima que se puede obtener vendiendo todos los vinos.

   Implemente, usando Java, un algoritmo que resuelva este problema, de forma óptima, utilizando *Memoization*. Escriba test para probar su solución.

10. En un examen de Historia, se pide a los alumnos que ordenen un conjunto de eventos en orden cronológico. Decidir si el orden en una solución de un estudiante es el correcto es simple: sólo hay que compararlo con la solución correcta provista por el profesor, y comprobar si coinciden.

    Para los alumnos que *no* ordenaron todos los eventos correctamente, el profesor quiere darles un puntaje menor al del ejercicio correcto, pero que "crezca" a medida que la solución más se acerque a la solución correcta. Propone entonces dar como puntaje la longitud más larga de eventos cuyo órden relativo es el correcto. Por ejemplo, si los eventos ordenados correctamente son $[1, 2, 3, 4]$ y la solución del alumno es $[1, 3, 2, 4]$, entonces el puntaje obtenido es 3 (las secuencias $[1, 2, 4]$ y $[1, 3, 4]$ son las dos secuencias más largas de eventos cuyo orden relativo es el correcto).

    Implemente, usando Haskell, un algoritmo que resuelva este problema. La entrada del algoritmo será una secuencia de n números enteros positivos, sin repetidos y con valores entre 1 y n, que representa la solución provista por el alumno. La solución correcta es esa misma secuencia, ordenada.

    Es su solución polinomial? Justifique. Si su solución no fuera polinomial diseñe e implemente una solución polinomial.

11. Implemente en Java un programa que calcule la distancia de Damerau-Levenshtein entre dos cadenas de caracteres. En caso de que su solución sufra de problemas de ineficiencia, intente salvarlos diseñando e implementando un algoritmo basado en *Memoization* para resolver el problema.

12. Diseñe un algoritmo que resuelva el problema de la mochila (*knapsack*), planteado en el capítulo 8 de *Introduction to the Design and Analysis of Algorithms*, utilizando un enfoque *bottom-up*.