Departamento de Computación
FCEFQyN, Universidad Nacional de Río Cuarto
Asignaturas: Diseño de Algoritmos - Algoritmos II
Primer Cuatrimestre de 2023

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. Éstos pueden accederse a partir del classroom de github Diseno-de-Algoritmos-Algoritmos-II-2023.

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 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. 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.

5. 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 Java o 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.

6. 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 *Memoization.* Compare empiricamente este algoritmo con la versión recursiva original. Construya además una versión utilizando *Programación Dinámica* .

7. 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 o Haskell, un algoritmo que resuelva este problema, de forma óptima, 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.

9. El problema de calcular el número mínimo de saltos para alcanzar una posición en un arreglo consiste en lo siguiente. Dado un arreglo de enteros, en el cual cada elemento representa el número máximo de pasos que se puede realizar hacia adelante desde esa posición, encontrar el número mínimo de saltos para llegar a la posición destino desde una posición origen (menor o igual a la primera). El siguiente programa resuelve este problema de forma recursiva:

```
int minJumps(int arr[], int l, int h ) {
  if (h == l)
        return 0;
  if (arr[l] == 0)
        return Integer.MAX_VALUE;
  int min = Integer.MAX_VALUE;
  for (int i = l+1; i <= h && i <= l + arr[l]; i++) {
        int jumps = minJumps(arr, i, h);
        if (jumps != Integer.MAX_VALUE && jumps + 1 < min)
              min = jumps + 1;
  }
  return min;
}
```

si el origen == 0, desde ahí no se puede ir a ningún lado así que se le asigna el numero Integer mas grande de Java.

   Esta solución es sumamente ineficiente. Mejore esta solución de manera tal que diferentes subproblemas comunes que surjan en la resolución del mismo no sean computadas repetidamente.

10. Averigüe en qué consiste el problema de la mochila (*knapsack*), planteado en el capítulo 8 de *Introduction to the Design and Analysis of Algorithms*. Dado que puede existir en general más de una solución optimal para este problema, diseñe un algoritmo que determine si existe una única solución óptima para el problema.

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.