GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: INTRODUCCIÓN A LA PROGRAMACIÓN DINÁMICA

1. Introducción

Entre las diferentes áreas de conocimientos que debe conocer un concursante de programación competitiva está la *Programación Dinámica* la cual es una técnica que combina la corrección de la búsqueda completa y la eficiencia de los algoritmos golosos. A dicha técnica y sus principales elementos estará dedicada la siguiente guía de aprendizaje a manera de introducción en esta área de conocimiento.

2. Conocimientos previos

2.1. Búsqueda completa

La búsqueda completa es un método general que se puede utilizar para resolver casi cualquier problema de algoritmo. La idea es generar todas las posibles soluciones al problema utilizando la fuerza bruta, y luego seleccionar la mejor solución o contar el número de soluciones, dependiendo del problema.

2.2. Algoritmos golosos (*Greedy*)

Un algoritmo goloso construye una solución al problema al tomar siempre una decisión que se ve mejor en este momento. Un algoritmo goloso nunca recupera sus opciones, pero construye directamente la solución final. Por esta razón, los algoritmos golosos suelen ser muy eficientes.

La dificultad para diseñar algoritmos golosos es encontrar una estrategia codiciosa que siempre produzca una solución óptima al problema. Las opciones localmente óptimas en un algoritmo goloso también deben ser globalmente óptimos. A menudo es difícil argumentar que funciona un algoritmo goloso.

2.3. Función recursiva

La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre. Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

3. Desarrollo

La programación dinámica es una técnica que combina la corrección de la búsqueda completa y la eficiencia de los algoritmos golosos. Hay dos usos para la programación dinámica:

1. **Encontrar una solución óptima:** Queremos encontrar una solución que sea tan grande lo más posible o lo más pequeño posible de acuerdo a la situación que nos plantea el problema o ejercicio.
2. **Contar el número de soluciones:** Queremos calcular el número total de soluciones posibles.

La programación dinámica (a partir de ahora abreviada como DP) es quizás de las más desafiante técnica de resolución de problemas entre los paradigmas presentes en la programación de concursos ya que compinas algunos de estos como son la búsqueda completa, algoritmos golosos y divide y vencerás. Las habilidades clave que debe desarrollar para dominar DP son las habilidades para determinar los estados del problema y determinar las relaciones o transiciones entre los estados problema y sus subproblemas.

Ahora como poder definir si un problema puede ser solucionado aplicando DP. Para esto el problema original debe tener:

1. **Propiedad óptima de la subestructura:** La solución óptima al problema contiene soluciones óptimas a los subproblemas.
2. **Propiedad de subproblemas superpuestos:** Accidentalmente recalculamos el mismo problema dos veces o más.

Hay 2 tipos de DP: podemos construir soluciones de subproblemas de pequeños a grande (de abajo hacia arriba, *Bottom-Up*) o podemos guardar los resultados de las soluciones de los subproblemas en una tabla (de arriba hacia abajo + memorización, *Top-Down+memoization*).

3.1. Top-Down DP

En el enfoque de *Top-Down DP*, implementamos la solución naturalmente utilizando la recursión, pero la modificamos para guardar la solución de cada subproblema en una matriz o tabla hash. Este enfoque primero verificará si ha resuelto previamente el subproblema. En caso afirmativo, devuelve el valor almacenado y guarda más cálculos. De lo contrario, el enfoque de arriba hacia abajo calculará las soluciones de subproblemas de la manera habitual. Decimos que es la versión memoizada de una solución recursiva, es decir, recuerda qué resultados se han calculado anteriormente.

3.2. Bottom-Up DP

Hay otra forma de implementar una solución DP a menudo denominada *Bottom-Up DP*. Esta es en realidad la forma verdadera de DP, ya que DP se conocía originalmente como el método tabular (técnica de cálculo que involucra una tabla). Los pasos básicos para construir una solución *Bottom-Up DP* son como sigue:

1. Determine el conjunto requerido de parámetros que describen de manera única el problema (el estado). Este paso es similar a lo que hemos discutido en retroceso recursivo y *Top-Down DP* anteriormente.
2. Si se requieren N parámetros para representar a los estados, prepare una matriz n dimensional (tabla DP), con una entrada por estado. Esto es equivalente a la tabla de memo en *Top-Down DP*. Sin embargo, hay diferencias. En Bottom-Up DP, solo necesitamos inicializar algunas celdas de la tabla DP con valores iniciales conocidos (los casos base). Recuerde que en *Top-Down DP*, inicializamos la tabla de memo completamente con valores ficticios (generalmente -1) para indicar que aún no hemos calculado los valores.

3. Ahora, con las células/estados de casos base en la tabla DP ya llena, determine las celdas/estados que se pueden llenar a continuación (las transiciones). Repita este proceso hasta que se complete la tabla DP. Para el *Bottom-Up DP*, esta parte generalmente se logra a través de iteraciones, utilizando bucles.

3.3. Bottom-Up vs Top-Down

Aunque ambos estilos usan matrices o arreglos, la forma en que se llena la matriz o arreglo de *Bottom-Up DP* es diferente a la de la matriz o arreglo de memorizacion de *Top-Down DP*. En el *Top-Down DP*, las entradas de la matriz o arreglo de memorización se llenan según sea necesario a través de la recursividad misma. En el *Bottom-Up DP*, usamos un orden de llenado de matriz o arreglo DP correcto para calcular los valores de modo que ya se hayan obtenido los valores anteriores necesarios para procesar la celda actual.

Para la mayoría de los problemas de DP, estos dos estilos son igualmente buenos y la decisión de usar un el estilo particular de DP es una cuestión de preferencia. Sin embargo, para problemas de DP más difíciles, uno de los estilos pueden ser mejores que el otro. Para ayudarlo a comprender qué estilo debe utilizar cuando se le presenta un problema de DP presentamos la siguiente tabla:

Top-Down DP	**Bottom-Up DP**
Ventajas	
Es una transformación natural de la búsqueda completa recursiva normal	Más rápido si se revisan muchos subproblemas Como no hay sobrecarga de llamadas recursivas
Calcula los subproblemas solo cuando necesario (a veces esto es más rápido)	Puede guardar espacio de memoria con la técnica de *ahorro de espacio*
Desventajas	
Más lento si se revisan muchos subproblemas debido a la sobrecarga de llamadas de función (esto es notablemente penalizado en los concursos de programación)	Para los programadores que se inclinan a la recursión, este estilo puede no ser intuitivo
Si hay M estados, un tamaño de tabla O(M) se requiere, lo que puede conducir a MLE para algunos problema más dificiles	Si hay M estados, *Bottom-Up DP* visita y llena el valor de todos estos estados de M Incluso si muchos de los estados no son necesarios

3.4. Mostrando la solución óptima

Muchos problemas de DP solicitan solo el valor de la solución óptima. Sin embargo, muchos concursantes son atrapados por descuidados cuando también están obligados a imprimir la solución óptima. Somos conscientes de dos formas de hacer esto.

La primera forma se utiliza principalmente en el enfoque Bottom-Up DP (que todavía es aplicable a Top-Down DP) donde almacenamos la información predecesora en cada estado. Si hay más de un predecesor óptimo y tenemos que generar todas las soluciones óptimas, podemos al-

macenar a esos predecesores en una lista. Una vez que tenemos el estado final óptimo, podemos hacer retroceso desde el estado final óptimo y seguir las transiciones óptimas registradas en cada estado hasta llegar a uno de los casos base. Si el problema solicita todas las soluciones óptimas, esta rutina de retroceso las imprimirá todas. Sin embargo, la mayoría de los autores de proeblemas o ejercicios generalmente establecen criterios de salida adicionales para que la solución óptima seleccionada sea única (para juzgar más fácilmente).

La segunda forma es aplicable principalmente al enfoque Top-Down DP, donde utilizamos la fuerza de la recursión y la memorización para hacer el mismo trabajo.

4. Aplicaciones

Si usted encuentra un problema que dice *minimizar esto* o *maximizar aquello* o *contar las formas de haz eso*, entonces hay una probabilidad (alta) de que sea un problema de DP.

5. Complejidad

Comprender la programación dinámica es un hito en la carrera de todo programador competitivo. Si bien la idea básica es simple, el desafío es cómo aplicar la programación dinámica a diferentes problemas. Lograr entender y utilizar este hito te permitirá obtener mejores resultados en competencias y concursos.

6. Ejercicios

Dentro de los ejercicios o problemas que tienen una solución usando DP se puede agrupar o clasificar en dos grupos:

1. **Los clásicos:** Son aquellos problemas que para dicha situación ya se conoce un algoritmo de DP bien definido que le da una solución. Aunque los problemas de tipo programación dinámica son muy popular con una alta frecuencia de aparición en concursos de programación recientes, los problemas clásicos de programación dinámica en su forma pura por lo general ya no aparecen en los IOI o ICPC modernos. A pesar de esto es necesario su estudio ya que nos permite entender la DP y como poder resolver aquellos problema de DP clasificados como **no-clásicos** e incluso nos permite desarrollar nuestras *habilidades de programación dinámica* en el proceso. Dentro de los problemas clásicos de DP podemos mencionar:
    - Suma máxima de rango dentro de un arreglo.
    - Suma máxima de rango dentro de una matriz.
    - Subsecuencia creciente más larga (LIS)
    - 0-1 Mochila
    - Cambio de moneda

- Problema del vendedor ambulante
- Caminos en una cuadrícula
- La distancia de edición o distancia de Levenshtein
- Contando mosaicos
- Multiplicación de matrices
- Subsecuencia común más larga

2. **Los no clásicos:** Es evidente que aquí están los problemas que aún no se cuenta con un algoritmo bien descripto que solucione el problema o que su solución parte de la combinación de varios clásicos. Otro factor por lo que se considera no-clásico es por la frecuencia de aparición de este tipo de problema en los concursos. Es muy común que si la frecuencia de aparición del problema en los concursos y competencia aumente deje ser no clásico y pase a clásico.