Algoritmos y Estructuras de Datos 3: Práctica 3

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Octubre 2013

Objetivo de la práctica

En la primer parte de la materia, antes de comenzar a estudiar grafos, se explican varias técnicas algorítmicas generales:

- Recursión
- Divide and Conquer
- Algoritmos golosos
- Programación dinámica
- Backtracking

El objetivo en esta práctica es ejercitar el uso de dichas técnicas en la formulación de algoritmos, y el análisis de los mismos.

Objetivo de la práctica

En la primer parte de la materia, antes de comenzar a estudiar grafos, se explican varias técnicas algorítmicas generales:

- Recursión
- Divide and Conquer
- Algoritmos golosos
- Programación dinámica
- Backtracking

El objetivo en esta práctica es ejercitar el uso de dichas técnicas en la formulación de algoritmos, y el análisis de los mismos.

Objetivo de la práctica

En la primer parte de la materia, antes de comenzar a estudiar grafos, se explican varias técnicas algorítmicas generales:

- Recursión
- Divide and Conquer
- Algoritmos golosos
- Programación dinámica
- Backtracking

El objetivo en esta práctica es ejercitar el uso de dichas técnicas en la formulación de algoritmos, y el análisis de los mismos.

División

Si bien los ejercicios están numerados en forma continua y no hay secciones marcadas, la práctica se encuentra dividida implícitamente en cuanto a la temática de los ejercicios siguiendo fielmente las técnicas algorítmicas mencionadas:

- Recursión, 1 y 2
- Divide and Conquer, 3 y 4
- Algoritmos golosos, 5 y 6
- Programación dinámica, 7 al 13
- Backtracking, 14 al 16

Finalmente los ejercicios 17 y 18 son ejercicios generales en los que no se trata un problema particular, sino que se pide analizar los algoritmos ya realizados en ejercicios anteriores de esta y otras prácticas.

Recursión

- Muchos algoritmos útiles son recursivos: Usan soluciones a subproblemas íntimamente relacionados, de la misma naturaleza que el problema principal.
- Varias de las técnicas algorítmicas estudiadas utilizan la recursión (por ejemplo Divide and Conquer, Programación Dinámica y Backtracking), por eso es importante que el alumno entienda la recursión en sí misma antes intentar utilizar técnicas particulares más complejas que hagan uso de la recursión.

Recursión: Ejercicios 1 y 2

- Ejercicio 1: Cálculo de los números de Fibonacci, de manera recursiva e iterativa comparando complejidades.
- Ejercicio 2: Escribir un algoritmo para resolver problema de las torres de Hanoi recursivamente. Analizar complejidad y demostrar correctitud.
- En ambos ejercicios la intención es practicar el diseño de algoritmos recursivos y su análisis de complejidad.

Divide and Conquer

- Muchos algoritmos recursivos responden a un esquema general de divide and conquer: el problema original es *dividido en subproblemas independientes* que se resuelven recursivamente, y luego *se combinan las soluciones* a esos subproblemas para construir una solución al problema completo.
- Como esta técnica está fuertemente basada en la recursión, es natural el paso de ejercicios sobre recursión a ejercicios sobre divide and conquer.

Divide and Conquer: Ejercicio 3

- Este ejercicio pide dar un algoritmo recursivo para encontrar el par de puntos más cercano de un conjunto de puntos en el plano.
- Está marcado con asterisco (mayor dificultad que la mayoría de la práctica).
- Se dan como pista las ideas centrales de un algoritmo eficiente de divide and conquer para resolver el problema.
- Se pide complejidad: Teorema Maestro de AED2

Divide and Conquer: Ejercicio 4

En este problema se pide dar un algoritmo de divide and conquer que arme un fixture para un torneo de n jugadores que se enfrentan todos contra todos una vez.

- La parte a) pide resolver el caso en que n es potencia de 2, para facilitar la división del problema en dos subproblemas de igual tamaño y simplificar el algoritmo resultante.
- La parte b) pide extender el algoritmo de la parte a) para que funcione para cualquier valor de n

Algoritmos golosos

- En general son fáciles de entender conceptualmente, pero no es tan fácil demostrar su correctitud.

Conviene colocar los algoritmos golosos antes de empezar con la sección de programación dinámica, ya que esta última es una técnica más compleja de entender y manejar correctamente:

- Programación Dinámica involucra guardar resultados intermedios y analizar múltiples opciones en cada paso.
- Los algoritmos golosos en cambio utilizan algún criterio para elegir de manera única un candidato en cada paso, sin tener en cuenta posibilidades alternativas para llegar a otra solución.

Algoritmos golosos: Ejercicios 5 y 6

- Ejercicio 5: Se pide analizar tres estrategias golosas posibles de ordenamiento de los programas en una cinta para minimizar el tiempo promedio de carga del programa.

La idea del ejercicio es mostrar con ejemplos que dos de las estrategias pueden dar resultados muy lejanos al óptimo en casos desfavorables, y demostrar la optimalidad de la tercera estrategia.

- Ejercicio 6: Problema de dar el vuelto con mínima cantidad de monedas.

Se pide demostrar que el goloso funciona con cierto conjunto de monedas bien elegidas, pero que no da el óptimo con otro conjunto.

Programación Dinámica

- La técnica de programación dinámica consiste en resolver un problema recursivamente, pero calculando la respuesta para cada valor posible de los parámetros del problema una única vez, y reutilizándo este valor ya calculado cuando sea necesario.
- Permite eliminar algunas complejidades exponenciales en recursiones (tipo fibonacci)
- Aprovecha **superposición de subproblemas**.
- Resulta generalmente más difícil de entender que las anteriores, ya que para resolver un ejercicio hay que lograr plantear el espacio de estados posibles del algoritmo, y establecer una relación de recurrencia que permita calcular el resultado de unos estados en función de estados anteriores.
- Este “estado” no siempre se desprende fácilmente del enunciado del problema en sí, sino que debe incorporar información intermedia propia de la solución que se quiere construir.

Programación Dinámica : Ejercicios 7,8 y 9

- Ejercicio 7: Cálculo de coeficientes binomiales (triángulo de Pascal). Ilustra superposición de subproblemas, y se sugiere repensar el ejercicio de Fibonacci usando esta idea.
- Ejercicio 8: Camino mínimo cruzando una matriz de esquina a esquina. También ilustra superposición de subproblemas y el uso del **principio del óptimo** para obtener una solución al problema. Al pedir *el camino* y no solo su *valor*, este ejercicio ejercita un punto importante de la programación dinámica, que es la reconstrucción de soluciones.
- Ejercicio 9: Encontrar una secuencia de operaciones para obtener un resultado fijo (si es posible). Sigue la línea anterior pero con un ejemplo más complejo , la elección del espacio de subproblemas es más difícil y menos evidente del enunciado.

Programación Dinámica : Ejercicio 10

- Se plantea una situación con números concretos, donde se fabrican radios mensualmente, pero se puede imaginar un esquema general del problema con cantidad arbitraria de meses y valores distintos de los costos. Se pide obtener un plan óptimo de producción usando PD.

Este ejercicio está marcado con un asterisco. Tiene más complejidad que otros debido a que hay mucha información en juego (más parámetros y datos diferentes que en otros problemas):

- La especificación de un subproblema involucra el mes que se está considerando actualmente y la cantidad de radios actualmente en stock.
- Hay que considerar todas las posibles cantidades de radios a fabricar durante el mes actual a la hora de escribir la relacion de recurrencia.
- Para esto último entran en juego la cantidad de radios pedidas en el mes actual, el costo de producir en el mes actual, el costo de stock y el costo fijo de iniciar producción.

Programación Dinámica : Ejercicios 11, 12 y 13

- Ejercicio 11: Problema de dar el vuelto con la menor cantidad de monedas. Se pide dar un algoritmo de programación dinámica y analizar en qué casos funciona. Permite comparar la técnica con el algoritmo goloso del ejercicio 6.
- Ejercicio 12: Distancia de edición entre dos palabras. Este ejemplo suele contarse brevemente en la teórica, pero aquí se pide dar el algoritmo preciso, incluyendo la reconstrucción de una secuencia de operaciones que transforme una palabra en otra.
- Ejercicio 13: Este es un ejercicio teórico sobre programación dinámica y por eso está al final, cuando se supone el alumno ya ha pensado o resuelto todos los ejercicios del tema.
- En este último ejercicio se pide dar ejemplos donde **no** valga el principio de optimalidad, de manera que no se pueda usar directamente programación dinámica. Esto también sirve como puente para llegar a backtracking, que al ser un algoritmo de búsqueda exhaustiva será aplicable en esos casos.

Backtracking

- La técnica propone una alternativa a la fuerza bruta.
- En lugar de probar todas las soluciones posibles individualmente, se va construyendo una solución de manera incremental.
- Se detiene el proceso (backtrack) cuando se detecta que esta solución parcial no se va a completar a una solución global.
- Generalmente da lugar a algoritmos exponenciales, y se la usa para tratar problemas computacionalmente difíciles.

Backtracking (cont)

La técnica no es muy difícil de entender conceptualmente, pero suele ser difícil implementar buenos algoritmos de backtracking:

- Como generalmente se utiliza recursión en la implementación, requiere un buen manejo de recursión.
- Hay que identificar y mantener en un estado consistente durante toda la ejecución la información relevante de la solución que se está construyendo, que va mutando permanentemente.

Teniendo en cuenta esto y que se la suele aplicar a problemas computacionalmente difíciles, es razonable poner esta técnica al final.

Backtracking : Ejercicios 14, 15 y 16

- Ejercicio 14: Problema de las 8 reinas en un tablero de ajedrez.
- Ejercicio 15: Problema de la mochila. Este problema es un ejemplo de problema NP completo que se verá más adelante en la materia.
- Ejercicio 16: Problema de la suma de subconjuntos, también NP completo.

En todos los casos se pide comparar contra el algoritmo de fuerza bruta, para enfatizar que si bien son todos algoritmos exponenciales, hay una enorme diferencia de eficiencia entre la fuerza bruta y la técnica de backtracking.
Al dar problemas computacionalmente difíciles, se va mostrando que la técnica de backtracking es aplicable a estos problemas, aunque resulte en complejidades exponenciales.

Ejercicios teóricos generales

Finalmente, terminan la práctica dos ejercicios generales para clasificar y analizar los algoritmos ya realizados en ejercicios anteriores.

Ejercicios teóricos generales : Ejercicio 17

En este ejercicio se pide determinar cuales ejercicios de las prácticas 2 y 3 son algoritmos de divide and conquer, y cuales son algoritmos recursivos. Además se pregunta si siempre es mejor usar una versión recursiva de un algoritmo o viceversa.
En base a las complejidades obtenidas en ejercicios anteriores, el alumno debería poder concluir que no siempre es mejor una versión recursiva, y tampoco es siempre mejor una versión no recursiva. Se puede ver como ejemplos los ejercicios 1 y 3 de esta práctica: En uno es mejor un algoritmo recursivo, y en otro es mejor uno iterativo.

Ejercicios teóricos generales : Ejercicio 18

Este ejercicio pide clasificar todos los ejercicios de esta práctica en “buenos” y “malos”. En las teóricas se define un buen algoritmo como un algoritmo de complejidad temporal polinomial. Por lo tanto, revisando los análisis de complejidad de los distintos algoritmos ya propuestos, el alumno debería poder realizar esta clasificación.

Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Septiembre 2013

Soluciones recursivas a problemas

- Muchos algoritmos de utilidad son recursivos: para resolver un problema, se utilizan las soluciones a subproblemas fuertemente relacionados.
- En estos algoritmos, se divide el problema en varios subproblemas que luego se resuelven y se combinan las soluciones obtenidas para resolver el original.
- Un ejemplo de técnica recursiva de diseño de algoritmos es la técnica de Divide and Conquer, vista en algoritmos 2.

¿En qué consiste la programación dinámica?

- La programación dinámica es una técnica de solución de problemas recursiva.
- Al igual que Divide and Conquer, la técnica propone descomponer el problema a resolver en **subproblemas más pequeños de la misma especie**, para resolverlos recursivamente y combinar esas soluciones en una solución al problema original.
- La diferencia esencial que lo contrasta con Divide and Conquer, es que mientras que en esta técnica los subproblemas que se resuelven son independientes entre sí y se resuelven individualmente, la programación dinámica es aplicable cuando los subproblemas **no son independientes**.
- En estos casos, un algoritmo de Divide and Conquer realizaría el mismo trabajo múltiples veces, ya que la solución a un mismo subproblema puede ser **recalculada** muchas veces si se la reutiliza como parte de varios subproblemas más grandes.

¿En qué consiste la programación dinámica? (2)

- La solución que propone la técnica de programación dinámica es **almacenar** las soluciones a subproblemas ya calculadas, de manera de calcularlas una sola vez, y luego leer el valor ya calculado cada vez que se lo vuelve a necesitar.
- Uno de los usos más importantes de esta técnica es en problemas de **optimización**: En estos problemas interesa encontrar la solución que maximiza un cierto puntaje u objetivo, en un espacio de soluciones posibles.
- Un indicador central de la aplicabilidad de la técnica lo constituye el **principio del óptimo**. Este principio afirma que **las partes de una solución óptima** a un problema, deben ser **soluciones óptimas de los correspondientes subproblemas**, y es lo que permite obtener una solución óptima al problema original a partir de soluciones óptimas de los subproblemas.

El esquema general

Los algoritmos de programación dinámica se pueden organizar típicamente en 4 pasos que responden al siguiente esquema general:

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima.
3. Computar el **valor** de una solución óptima. Se calcula de manera *bottom-up*.
4. Construir una solución óptima a partir de la información obtenida en el paso 3

El paso 4 es optativo, ya que si solo nos interesa el valor o puntaje de una solución óptima pero no la solución en sí, este paso de reconstrucción no es necesario.

Problema del recorrido óptimo en una matriz

Ejercicio 3.8, práctica 3 de Algoritmos y Estructuras de Datos 3:

Sea $M \in \mathbb{N}^{m \times n}$ una matriz de números naturales. Se desea obtener un camino que empiece en la casilla superior izquierda $(1, 1)$, termine en la casilla inferior derecha (m, n) y tal que minimice la suma de los valores de las casillas por las que pasa. En cada casilla (i, j) hay dos movimientos posibles: ir hacia abajo (a la casilla $(i + 1, j)$), o ir hacia la derecha (a la casilla $(i, j + 1)$).

a. Diseñar un algoritmo eficiente basado en programación dinámica que resuelva este problema.

b. Determinar la complejidad del algoritmo propuesto (temporal y espacial).

c. Exhibir el comportamiento del algoritmo sobre la matriz que aparece a continuación.

$$\begin{bmatrix} 2 & 8 & 3 & 4 \\ 5 & 3 & 4 & 5 \\ 1 & 2 & 2 & 1 \\ 3 & 4 & 6 & 5 \end{bmatrix}$$

Fórmula recursiva

La matriz de resultados parciales almacena en $best(i, j)$ la mínima longitud de un camino que empiece en (i, j) y llegue a (m, n), haciendo solo movimientos hacia abajo y hacia la derecha.

- $best(i, j) = M_{i,j} + min(best(i + 1, j), best(i, j + 1))$ para $1 \leq i < m$ y $1 \leq j < n$
- $best(i, n) = M_{i,n} + best(i + 1, n)$ para $1 \leq i < m$
- $best(m, j) = M_{m,j} + best(m, j + 1)$ para $1 \leq j < n$
- $best(m, n) = M_{m,n}$ para $i = m$ y $1 \leq j < n$

La longitud del mínimo camino entre esquinas, que constituye la solución al problema, viene dada por $best(1, 1)$. Con esto ya podríamos implementar una solución top-down recursiva:

- Si para calcular un $best(i, j)$ necesitamos un resultado ya calculado, lo usamos directamente.
- Sino, lo calculamos recursivamente, almacenamos su valor en la tabla de resultados y luego lo utilizamos.

Algoritmo bottom-up

```
1  longitudCaminoMinimo(Matriz, m, n):
2      best <- matrizDeEnteros(m,n)
3      best[m,n] = Matriz[m,n]
4      FOR i <- m-1 DOWNTO 1 DO
5          best[i,n] = Matriz[i,n] + best[i+1,n]
6      FOR j <- n-1 DOWNTO 1 DO
7          best[m,j] = Matriz[m,j] + best[m,j+1]
8      FOR i <- m-1 DOWNTO 1 DO
9          FOR j <- n-1 DOWNTO 1 DO
10             best[i,j] = Matriz[i,j] + min(best[i+1,j] + best[i,j+1])
11     RETURN best[1,1]
```

La complejidad del algoritmo resultante es $O(nm)$, tanto espacial como temporal. Se puede bajar la complejidad espacial a $O(min(n, m))$ si no interesa reconstruir el camino sino solo su longitud.

Cálculo en el ejemplo

Matriz de entrada:

$$\begin{bmatrix} \mathbf{2} & 8 & 3 & 4 \\ \mathbf{5} & 3 & 4 & 5 \\ \mathbf{1} & \mathbf{2} & \mathbf{2} & \mathbf{1} \\ 3 & 4 & 6 & \mathbf{5} \end{bmatrix}$$

Matriz de best:

$$\begin{bmatrix} \mathbf{18} & 21 & 15 & 15 \\ \mathbf{16} & 13 & 12 & 11 \\ \mathbf{11} & \mathbf{10} & \mathbf{8} & \mathbf{6} \\ 18 & 15 & 11 & \mathbf{5} \end{bmatrix}$$

En ambas matrices, se indica un camino óptimo en negrita.

Referencias

- Introduction to Algorithms, 2nd Edition. MIT Press. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Página 323: 15 Dynamic Programming