UNIVERSIDAD AUTÓNOMA DE MADRID

ALGORITMOS Y ESTRUCTURAS DE DATOS AVANZADAS

Práctica 3: Quick Select. Programación Dinámica

Sergio Hidalgo Gamborino
Miguel Ibáñez González

Grupo 1292
Pareja 05

I. EL PROBLEMA DE SELECCIÓN

I-D. Cuestiones sobre QuickSelect y QuickSort

Contestar razonadamente a las siguientes cuestiones incluyendo gráficas si fuera preciso.
1. Argumentar en primer lugar que MergeSort ordena una tabla de 5 elementos en a lo sumo 8 comparaciones de clave.

Para una tabla de 5 elementos: a,b,c,e,f

abc|ef-> máximo 4 comparaciones
ab|c -> máximo 2 comparaciones
e|f -> máximo 1 comparaciones
a|b -> máximo 1 comparaciones

El máximo de comparaciones es 8.

Pero, en realidad, en qsel_5 solo queremos encontrar la mediana de una tabla de 5 elementos, pero no ordenarla.
¿Podríamos reducir así el número de comparaciones de clave necesarias? ¿Cómo?

- Ordenamos las primeras dos parejas, [a b c d e] -> a >? b, c >? d (2 comparaciones)
- Sacamos el menor [a < b c < d e] -> a >? c (1 comparación)
- Eliminamos el menor
- Comparamos el siguiente elemento con el elemento de la pareja eliminada b >? e (1 comparación)
- Volvemos a comprobar el menor y lo lo eliminamos a [b c < d e] -> b <? e , b <? c (2 comparaciones)
- Comparamos el elemento que se queda solo con el menor de la otra
- pareja. Esta será la mediana-> a b [c < d e] -> c <? e (1 comprobación)

En total se hacen 6 comprobaciones

2. ¿Que tipo de crecimiento cabría esperar en el caso peor para los tiempos de ejecución de nuestra función qsort_5? Intenta justificar tu respuesta experimentalmente.

El problema a evitar es elegir un pivote que sea el primero o último elemento de la tabla ordenada. En este caso, cada iteración deberá realizar n-1 operaciones, lo que conlleva un coste de: $(n^2 - n) / 2 \rightarrow O(n^2)$

Caso peor -> {6, 5, 4, 3, 2} -> {5, 4, 3, 2} -> {4, 3, 2} -> {4, 3} -> {3}

II. PROGRAMACIÓN DINÁMICA

II-C. Cuestiones sobre las funciones de programación dinámica Contestar razonadamente a las siguientes cuestiones.

1. ¿Cual es el coste en espacio de los algoritmos PD para el problema del cambio y de la mochila 0-1? Si en dichos problemas solo queremos conocer los valores óptimos y no la composición de las soluciones, ¿hay alguna manera de reducir el coste en memoria? ¿Como?

Problema del cambio:
El algoritmo PD para el cambio tiene un coste espacial de O(n * c), donde 'n' es el número de monedas disponibles y 'c' es la cantidad a cambiar. Esto se debe a la matriz creada para almacenar los resultados parciales.

Problema de la mochila 0-1:
El algoritmo PD para este problema tiene un coste espacial de O(n * bound), donde 'n' es el número de elementos y 'bound' es la capacidad de la mochila. Esto se debe a la matriz creada para almacenar los valores óptimos.

Reducción del coste en memoria:
En el caso del problema del cambio, en lugar de almacenar toda la matriz, podríamos utilizar dos vectores, uno para el resultado actual y otro para el resultado anterior. Al iterar sobre las monedas, actualizamos estos vectores para el cálculo del resultado final.
Para el problema de la mochila 0-1 podemos mantener sólo dos filas de la matriz para calcular los valores óptimos.

2. Una variante del problema de la mochila 0-1 consiste en suponer que de cada elemento si hay cualquier número de copias. Desarrollar en detalle las fórmulas de una solución PD para esta variante

Si dp un array de tamaño (n+1) x (bound+1) inicializado con ceros
La fórmula para la iteración sería:
dp[i][w] = max(dp[i-1][w], dp[i][w-weight[i]] + value[i])

En donde dp[i][w] representa el valor óptimo para el peso 'w' utilizando los primeros 'i' elementos.
weight[i] representa el peso del elemento 'i'.
value[i] representa el valor del elemento 'i'.

Con estos cambios se puede implementar la variante donde hay múltiples copias de cada elemento en el problema de la mochila 0-1 mediante programación dinámica.