14/12/2023

Algoritmia y Estructuras de Datos Avanzadas

Práctica - 3

G.1281 - P3

Joaquín Abad Díaz - Carlos García Santa

Algoritmia y Estructuras de datos, 2023-2024
Práctica-3

I-D. Cuestiones sobre QuickSelect y QuickSort

1. Argumentar en primer lugar que MergeSort ordena una tabla de 5 elementos en a lo sumo 8 comparaciones de clave. 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?

En el caso de querer ordenar una tabla de 5 elementos con mergesort, este hará cómo máximo 8 comparaciones de clave por lo siguiente: MergeSort divide la tabla en dos partes, una subtabla de 2 elementos y otra subtabla de 3 elementos. Para ordenar la primera se necesita 1 comparación entre los dos elementos. Sin embargo, para la segunda (que será a su vez dividida en otras dos subtablas, una de un elemento y otra de dos elementos) se necesitarán en el peor caso 3 comparaciones para que quede ordenada, 1 para ordenar la subtabla de 2 elementos y 2 para insertar el tercer elemento en el sitio adecuado. Tenemos un total de 4 comparaciones para ordenar las subtablas. En la fase de fusión de esas subtablas, en el peor de los casos, por ejemplo, si el primer elemento de la subtabla de 2 elementos es mayor que el primer elemento de la subtabla de 3 elementos, pero menor que el segundo, se necesitará otra comparación. De manera similar, podría necesitarse una cuarta comparación para el segundo elemento de la subtabla de 2 elementos. Esto nos deja con un total de 8 comparaciones.

En el caso de encontrar la mediana en una tabla de 5 elementos de quickselect5, dado que el arreglo tiene 5 elementos, se creará un solo subarreglo. A la hora de buscar la mediana en ese arreglo, si recurrimos a un algoritmo de ordenación para después elegir el elemento del medio como la media, no hay ningún algoritmo que nos garantice menos de 8 comparaciones en el peor caso. Sin embargo, podemos usar un método que nos garantice menos comparaciones. Por ejemplo, si sé que tengo 5 elementos puedo inicialmente comparar pares de elementos (el primero con el segundo y el tercero con el cuarto, 2 comparaciones) para sacar valores máximos y mínimos, después comparamos el máximo del primer par con el máximo del segundo y hacemos lo mismo con los mínimos (otras 2 comparaciones); esto con el objetivo de descartar el menor de los mínimos y el mayor de los máximos, finalmente comparamos el elemento restante con los dos números restantes (otras 2 comparaciones) para averiguar cual es la mediana.

Ejemplo: Si tenemos [2,8,3,5,7]

-min(2,8) = 2
-max(2,8) = 8 Esto se puede implementar con una sola comparación
-min(3,5) = 3

-max(3,5) =5 Esto se puede implementar con una sola comparación

-min(2,3) = 2 Por lo tanto nos olvidamos del 2. Tercera comparación.

-max(5,8) = 8 Por lo tanto nos olvidamos del 8. Cuarta comparación.

Si 7 > 3 Quinta comparación.

Si 7 < 5 Sexta comparación.

Mediana = 7

Mediana = 5

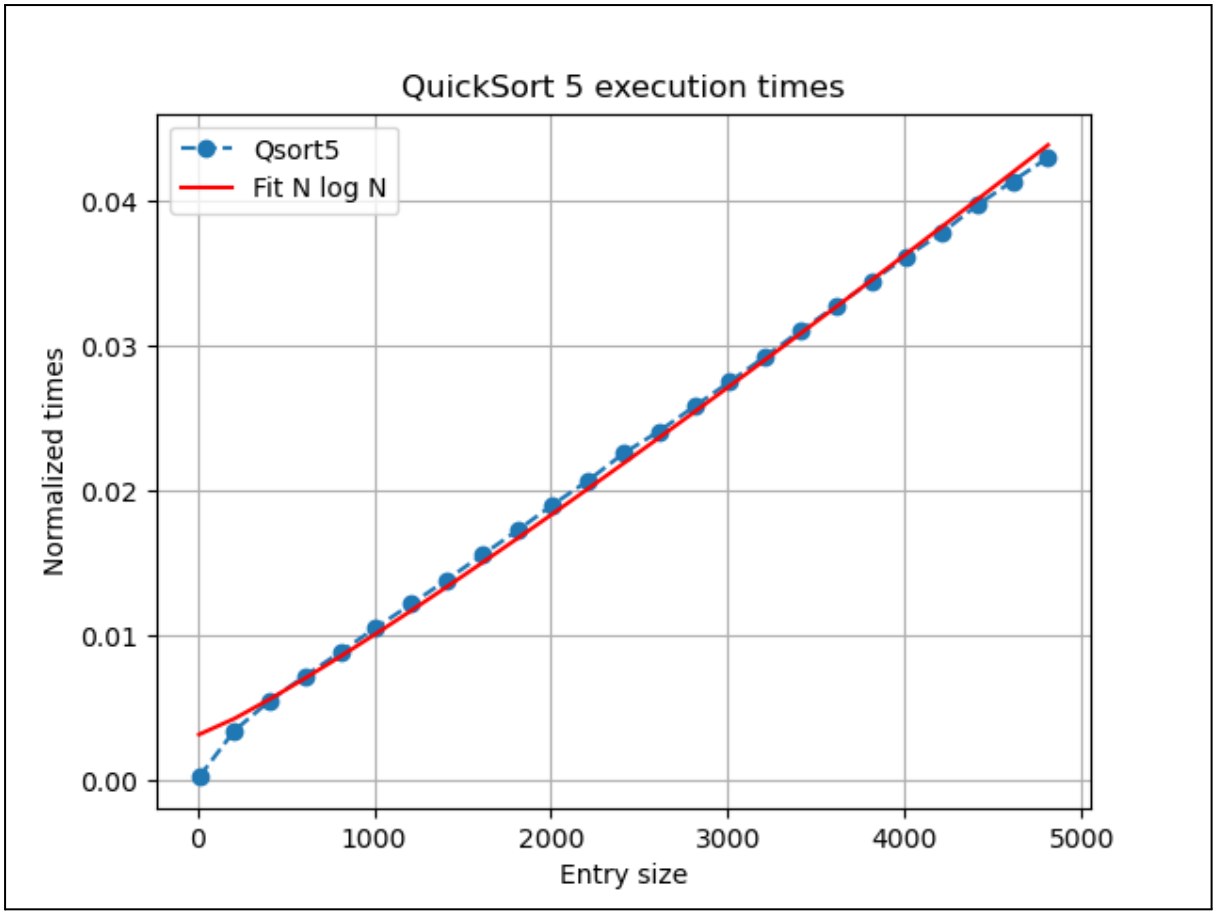
Mediana = 3

Cómo 7 es mayor que 3 pero no es menor que 5, la mediana es 5 que es correcto.

Esta solución requiere 6 comparaciones de clave que son menos que las 8 requeridas como máximo por MergeSort.

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

De los n/5 grupos, la mitad de ellos (1/2 * n/5 = n/10) tienen su mediana menor que el pivote (Mediana de Medianas). Además, la otra mitad de los grupos (nuevamente, 1/2 * n/5 = n/10) tienen su mediana mayor que el pivote. En cada uno de los grupos n/10 con mediana menor que el pivote, hay dos elementos que son menores que sus respectivas medianas, las cuales son menores que el pivote. Así, cada uno de los grupos n/10 tiene al menos 3 elementos que son menores que el pivote. De manera similar, en cada uno de los grupos n/10 con mediana mayor que el pivote, hay dos elementos que son mayores que sus respectivas medianas, las cuales son mayores que el pivote. Por lo tanto, cada uno de los grupos n/10 tiene al menos 3 elementos que son mayores que el pivote. Por consiguiente, el pivote es menor que 3 * n/10 elementos y mayor que otros 3 * n/10 elementos. Así, la mediana de medianas asegura que el pivote divide el conjunto de datos en una proporción que evita los peores casos de particiones muy desiguales, lo que asegura un comportamiento promedio en el peor caso del algoritmo, que en el caso de QuickSort sería O(N log N).



Como podemos ver los resultados experimentales coinciden con nuestra deducción teórica, el caso peor de quickSort con este método de elección de pivote es O(N log N).

II-C. Cuestiones sobre las funciones de programación dinámica

1. ¿Cuál 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 de la Mochila 0-1:

El algoritmo para el problema de la mochila 0-1 utiliza una matriz de tamaño N x M, donde N es el número de elementos y M es el peso límite de la mochila, por lo tanto el coste espacial de este algoritmo es O(N * M) .

Problema del Cambio:

En el caso del problema del cambio, se utiliza una matriz de tamaño N x M, donde N es la cantidad de denominaciones de monedas diferentes y M es la cantidad total para el cual se calcula el cambio. Similar al problema de la mochila, el coste espacial aquí es es O(N * M).

Es posible reducir el coste de memoria para ambos problemas si solo estamos interesados en conocer los valores óptimos y no la composición de las soluciones utilizando un arreglo unidimensional de tamaño cota + 1 en lugar de utilizar una matriz bidimensional, de esta manera, podemos iterar sobre cada elemento (en el caso de la mochila, cada elemento representa un objeto con un peso y un valor; en el caso del cambio, cada elemento representa una denominación de moneda), actualizando el arreglo para reflejar el nuevo estado óptimo considerando el elemento actual.

Para la mochila 0-1: Actualizar arreglo[w] para todos los pesos desde bound hasta el peso del elemento actual, estableciendo arreglo[w] como el máximo entre su valor actual y el valor del elemento actual más arreglo[w - peso_del_elemento_actual].

Para el problema del cambio: Actualizar arreglo[j] para cada sub-valor j desde el valor de la moneda actual hasta c, estableciendo arreglo[j] como el mínimo entre su valor actual y 1 + dp[j - valor_de_la_moneda_actual].

Las complejidades espaciales de ambos problemas pasarían a ser O(M) donde M es el límite clave (peso máximo o cantidad de cambio).

2. Una variante del problema de la mochila 0-1 consiste en suponer que de cada elemento i hay cualquier número de copias.

Desarrollar en detalle las fórmulas de una solución PD para esta variante.

Fórmula: dp[j] = max(dp[j], dp[j - w[i]] + v[i])

1. **dp[j - w[i]] + v[i]**: Esta expresión calcula el valor total si decidimos incluir al menos una instancia del elemento i en nuestra mochila que actualmente tiene un peso total de j.
2. **max(dp[j], dp[j - w[i]] + v[i])**: Esta expresión determina cuál es la mejor opción: incluir o no incluir una instancia adicional del elemento i para alcanzar un peso total de j.