IIC2283 — Diseño y Análisis de Algoritmos
Ayudantía 9

Se considerará una lista ligada representada sobre un arreglo de nodos, donde cada nodo está formado por un valor, y el índice del nodo siguiente en la lista.

Además, se cumple que el tamaño del arreglo es significativamente mayor al de la memoria principal, y que los índices *next* de cada nodo pueden apuntar a cualquier elemento del arreglo, mientras se mantenga la estructura de lista.

Como notación, se definirán los siguientes elementos:

- N: tamaño del input
- M: tamaño de la memoria principal
- B: tamaño de un bloque
- n: N/B
- m: M/B
- $scan(N)$: tiempo necesario para recorrer N posiciones contiguas en memoria:

$$O(scan(N)) = O(n)$$

- $sort(N)$: tiempo necesario para ordenar una entrada de tamaño N:

$$O(sort(N)) = O(n \log_m n) \ll O(N)$$

Problema 1: Sumas de elementos contiguos

Dada una lista como la descrita, se busca calcular otra lista que tenga la misma forma, pero donde el valor de cada nodo sea igual a la suma entre el valor del nodo original, y el valor de su antecesor.

Parte 1 Determine una cota superior para el número de accesos a memoria necesarios para resolver el problema.

Solución Una solución posible consiste en copiar la lista en tiempo $O(scan(N))$, y luego recorrer ambas en orden de lista, sumando el valor del nodo i obtenido de la lista original, al nodo $i+1$ de la copia.

Por la estructura de la lista, en el peor caso este algoritmo realiza $O(N)$ accesos a memoria.

Parte 2 Desarrolle y analice un algoritmo que resuelva el problema en tiempo $O(sort(N))$.

Solución Primero se copia la lista en tiempo $O(scan(N))$, y luego se ordena según el índice del sucesor.

Luego, se recorren ambas listas en el orden de memoria, y por cada nodo $(value, next)$ de la lista ordenada, se suma el valor $value$ al nodo en la posición $next$ de la lista original. Como la lista ordenada se recorre en orden secuencial, y las posiciones a las que se suma un valor de la lista original también están ordenadas, se necesitan $O(scan(N))$ accesos a memoria.

Finalmente, el tiempo total es $O(sort(N)) + O(scan(N)) = O(sort(N))$.

Problema 2: Coloración de listas

Con una lista de la misma forma que la anterior, ahora se busca encontrar una 3 coloración de sus nodos.

Parte 1 Describa y analice un algoritmo simple para encontrar una 2-coloración.

Solución Se recorre en el orden de la lista. El primer nodo tiene un color arbitrario, y cada nodo tiene el color contrario a su antecesor. En el peor caso el algoritmo realiza $O(N)$ accesos a memoria secundaria.

Parte 2 Decimos que una sublista es ascendente, si es una sublista contigua en el orden de la lista, tal que las direcciones de memoria de sus nodos son estrictamente crecientes.

Describa un algoritmo de tiempo $O(sort(N))$ que encuentre y coloree las sublistas ascendentes con 2 colores.

Para esto puede utilizar una cola de prioridad, y asumir que las operaciones de la cola toman tiempo:

$$O(PQ(N)) = O\left(\frac{1}{B}\log_m n\right).$$

Las entradas de la cola deberían ser pares entre una dirección y un color.

Solución Se utiliza una cola de prioridad donde se almacenan pares $(dir, color)$, que representan posiciones futuras que deban ser coloreadas con algún color fijo. Esto sucede cuando el nodo es alcanzable desde una posición anterior, donde el color está fijo por que debe ser el contrario al de su antecesor. La cola está organizada por dirección, y prioriza direcciones más pequeñas.

```
function ColorAsc(L)
    for nodo v ∈ L en orden creciente de memoria do
        if v.dir=PQ.min () then                          ▷ El nodo es parte de una sublista ascendente
            (dir,color)←PQ.deleteMin ()
            v.color←color
            if v.next≠ ⊥ and v.next>v.dir then
                PQ.insert({v.next, 1-v.color})           ▷ 1-v.color representa al color contrario a v.color
            end if
        else if v.next>v.dir then                        ▷ El nodo es el inicio de una lista ascendente nueva
            v.color←0                                    ▷ Las listas ascendentes comienzan con color 0
            PQ.insert({v.next, 1})
        end if
    end for
end function
```

En el algoritmo se colorea cada secuencia ascendente con colores 0 o 1, comenzando desde 0. Intuitivamente, el algoritmo asigna color 0 al primer nodo de cada sublista ascendente, y fija el color siguiente de cada nodo como el color contrario, usando la cola de prioridad para manejar varias listas simultáneamente.

El algoritmo requiere $O(scan(N))$ accesos para recorrer la lista, y por cada nodo realiza una cantidad constante de operaciones con la cola de prioridad, por lo que la cantidad de accesos a memoria está dado por:

$$O(scan(N)) + N \cdot O(PQ(N)) = O(scan(N)) + O(sort(N)) = O(sort(N))$$

Parte 3 Adapte el algoritmo anterior para listas descendentes, que se definen de la misma forma, pero donde las direcciones del nodo siguiente son estrictamente decrecientes.

Idea de solución Mismo algoritmo, pero recorriendo la lista desde el final, y modificando la cola de prioridad para priorizar direcciones más altas. También se debe cambiar el sentido de las comparaciones, como por ejemplo revisar que $v.next < v.dir$ en el último **if**.

Parte 4 Describa como generar una 3 coloración, combinando los algoritmos para sublistas ascendentes y descendentes, y analice la cantidad de accesos a memoria secundaria.

Idea de solución Se calculan y colorean las secuencias ascendentes, con colores R, G, y las descendentes con colores B, G. De esta forma, todas las secuencias ascendentes comienzan con un nodo de color R, y las descendentes con un nodo de color B.

Dependiendo del caso, podrían ocurrir conflictos donde el mismo nodo se coloree con dos colores, uno en una secuencia ascendente, y otro en una descendente. Estos conflictos solo pueden suceder entre el final de una lista ascendente con el principio de una descendente, o entre el final de una descendente con el principio de una ascendente, por lo que se pueden resolver usando el color de la lista que comienza en el nodo. Como los colores de inicio de las listas ascendentes y descendentes son distintos, se mantiene la propiedad de 3 coloración, por que asignar ese color no puede generar un conflicto con la lista que termina en el nodo.

El algoritmo realiza $O(sort(N))$ accesos a memoria.