Ayudantía 9 - Estructura de Datos y Algoritmos ELO320

Algoritmos de ordenamiento

Werner Creixell, Felipe Vera

29 de agosto del 2013

1. Ordenamiento en tiempo lineal

Los algoritmos vistos hasta ahora (merge, quick, heap sort) tienen un rendimiento de $O(n \log n)$ en el caso típico. Dado que están basado en comparaciones, no se puede mejorar ese rendimiento asintótico. Sin embargo el hecho de que estén basados en comparaciones sirve para poder ordenar números decimales.

Hay estrategias para números enteros que pueden ser más eficientes si el rango de los números es pequeño. Estas son:

1.1. Counting Sort

Este método de ordenamiento está basado en elaborar un histograma y luego, usando ese histograma, rellenar el arreglo resultante usando números ordenados.

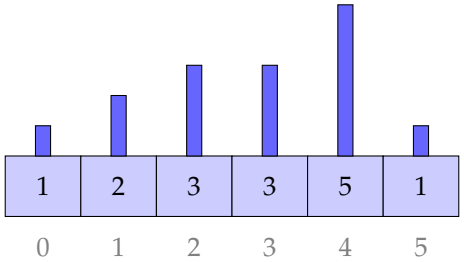
Ejemplo Se tiene un arreglo de números del 0 al 5.

3	3	4	4	4	2	1	3	2	1	4	4	2	0	5

Por lo tanto se crea una matriz histograma de elementos entre 0 y 5, con todos sus elementos inicializados en cero.

0	0	0	0	0	0
0	1	2	3	4	5

Se recorre toda la matriz y se suma 1 a cada elemento de la matriz donde corresponda, armando así el histograma.



Y por último, se reemplaza la matriz inicial en base al histograma:

- Se coloca un cero al principio del arreglo.
- A continuación dos unos.
- Luego tres veces el dos.
- Etc.

Con ello se tiene la matriz ordenada.

0	1	1	2	2	2	3	3	3	4	4	4	4	4	5

1.2. Radix Sort

Representa una mejora al Counting Sort en términos tanto de memoria como de tiempo de ordenamiento para rangos de números más grandes, en donde tener una matriz histograma con demasiados elementos sería un desperdicio de memoria, y recorrerla entera para construir el histograma sería una pérdida de tiempo.

La base es la misma pero con algunos cambios. Como aquí no se debe perder información del resto de la cifra, no se puede hacer Counting-Sort directamente a cada dígito. Se explicará con más detalle en un ejemplo.

Ejemplo Se debe ordenar este arreglo.

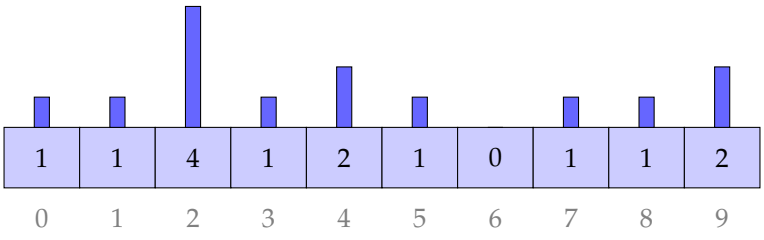
754	812	595	31	53	489	432	192	209	108	867	660	902	454
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Se debe ordenar partiendo desde el dígito menos significativo, es decir, la unidad. Como estos elementos son de base 10, se crea un histograma de 10 elementos.

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Y se llenan de acuerdo a los dígitos de la unidad.

Cabe destacar que este histograma es de **frecuencia acumulada**. En consecuencia el histograma resultante es el siguiente:



Y el acumulado, este (nótese que cada término del acumulado es una suma de todos los anteriores del otro histograma):

0	1	2	6	7	9	10	10	11	12
0	1	2	3	4	5	6	7	8	9

Por lo tanto, de acuerdo al histograma se debe seccionar el nuevo arreglo ordenado a la unidad. Este histograma indica en qué posición del arreglo se almacenará cada número con el último dígito dado. Volvamos al arreglo original.

Almacenemos el 754 (unidad = 4). Se busca en el histograma verde a qué posición corresponde el dígito 4, encontrando que debe almacenarse en la séptima casilla ($s[6]$)...

754	812	595	31	53	489	432	192	209	108	867	660	902	454
0	1	2	3	4	5	6	7	8	9	10	11	12	13

El resultante:

-	-	-	-	-	-	754	-	-	-	-	-	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Y se aumenta 1 a la casilla del histograma correspondiente.

0	1	2	6	8	9	10	10	11	12
0	1	2	3	4	5	6	7	8	9

Este proceso se repite para las unidades, decenas y centenas para lograr tener un arreglo ordenado.

Es muy útil en el caso no sólo de números, sino de ordenar cadenas de caracteres en orden alfabético también.

2. Tablas hash

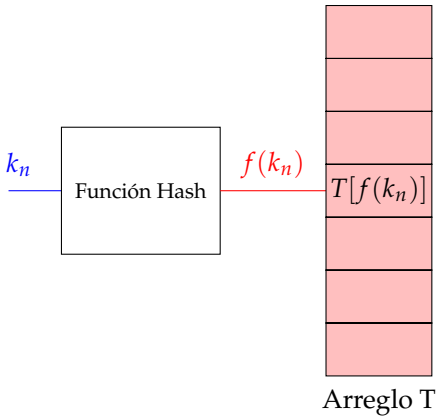
Las tablas Hash son un método para *asociar* cada elemento de un arreglo (que llamaremos T) a una clave (que llamaremos k_n). Todas estas claves están contenidas en una **tabla Hash**.

La utilidad de las tablas Hash se hace brillar al momento de implementar **diccionarios** [1]. A diferencia de Python y PHP, C no soporta diccionarios directamente, pero se puede usar Tablas Hash para poder acceder a los elementos de un arreglo usando una cadena de caracteres.

2.1. Funcionamiento

Se tiene un conjunto de claves k_n que son las entradas a la tabla Hash (como palabras en un diccionario). Se calcula el índice usando una **función Hash** $f(k_n)$ a partir de la palabra entregada y el dato del arreglo $T[f(k_n)]$ buscado contiene el significado de esa palabra.

[1] Un diccionario es un arreglo a cuyos elementos se acceden mediante una cadena de caracteres en lugar de un índice.



2.2. Funciones Hash

Son funciones matemáticas que computan, a partir de una clave k_n un índice del arreglo T asociado. La gracia es que esta computación se calcule en un tiempo constante para cualquier clave $O(1)$.

Los índices resultantes de estas funciones Hash, lógicamente, deben ser números naturales (incluído el 0). ¿Se puede pasar una clave no natural? Claro, si se interpreta como un número natural. Por ejemplo el caracter `'0'` tiene un valor numérico de 48.

Hashes criptográficos Existen varias funciones Hash que calculan hashes a partir de cadenas; y se ocupan en encriptación (como guardar contraseñas en bases de datos sin que el administrador de la página o cualquiera que pueda interceptar la conexión de internet puedan saberlas). Estas son MD5, SHA-1, SHA-2, SHA-3 entre otras.

Existen dos métodos matemáticos bastante comunes para diseñar estas funciones Hash.

2.2.1. Método divisivo

Supongamos que ingresamos una clave k número natural, hasheando a un arreglo T de M elementos (Se puede asignar valores desde $T[0]$ a $T[M-1]$). La función hash, por método divisivo es:

$$f(k) = k \text{ módulo } M$$

En otras palabras, el residuo de la división entre k y M. Este método es bastante rápido de calcular, dando una complejidad de $O(1)$.

Hay valores de m en que este hash tiene un **mal rendimiento**, como cuando M es una potencia exacta de 2; y un **buen rendimiento**, cuando m es un número primo lejano a una potencia de 2, como $M = 701$. La razón de esto es que, binariamente, un módulo de una potencia de 2 es obtener sólo una cantidad de bits determinada para obtener el hash, lo cual le brinda una gran desventaja si se quiere implementar un hash criptográfico difícil de revertir.

2.2.2. Método multiplicativo

Supongamos que ingresamos una clave k número natural, hasheando a un arreglo T de M elementos (Se puede asignar valores desde $T[0]$ a $T[M-1]$) y se elige una constante C entre 0 y 1. La función hash, por método multiplicativo es:

$$f(k) = \lfloor M \cdot (kC) \text{ módulo } 1 \rfloor$$

Es decir, se extrae la parte fraccional de $k \cdot C$ y se multiplica por el número de elementos M que tenga el arreglo hasheado.

La ventaja es que con este método se puede implementar el hash con M potencia de 2 sin problemas, sin embargo la eficiencia de este hash varía de acuerdo a la constante C elegida.

Implementación de Knuth Es una implementación bastante buena y barata de implementar en un computador, al no usar divisiones sino corrimientos de bits.

Se tienen como datos:

- k: Entrada de la función Hash.
- M: Tamaño del arreglo a hashear. Debe ser una potencia de 2, de la forma $M = 2^p$.
- p: $\log_2 M$
- w: Tamaño en bits del tipo de dato `int`. Generalmente es 32.

El algoritmo es como sigue:

1. Se debe escoger un número s tal que:
$$A = \frac{s}{2^w}$$
Knuth recomienda que A sea el número áureo $\frac{1}{2}\left(\sqrt{5}-1\right)$, para mayor eficiencia. Con ello, si $w = 32$, se obtendría que $s =$ 2,654,435,769.
2. Se toman los w menores bits de la multiplicación $k \cdot s$. (Esta multiplicación da como resultado un número de $2w$ bits. Esto se hace con un `and`.
$$V = (k \cdot s) \text{ módulo } (2^{32} - 1)$$
3. Se toman los p bits más significativos de V, y esa es la salida de la función Hash.
$$h(k) = p \text{ bits más significativos de } V$$

2.3. Resolución de colisiones

Estas funciones idealmente deberían ser *inyectivas* (es decir, a todas las claves distintas les corresponden índices distintos). Sin embargo en la práctica no es así debido a limitaciones en los tamaños de tipos de datos. (Por ejemplo, en una cadena de 128 caracteres se pueden almacenar hasta 8^{128} valores distintos. Es un número muy grande pero las posibilidades de claves distintas pueden ser mayores (si ocupamos por ejemplo el libro entero de Don Quijote como clave de Hash y pensamos en cuántas combinaciones podemos tener si cambiamos cada letra de ese libro...)

La forma de resolver las colisiones es sencilla: en lugar de tener un arreglo T de números, se hace un arreglo T de listas enlazadas, para que $T[h(k)]$ sea capaz de enlazar a más de un elemento.