Estructuras de datos
Clase teórica 11

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN

Contenido

- Tablas hash
- Análisis amortizado
- Problemas candidatos

Material elaborado por: Julián Moreno

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Eficiencia de las tablas hash

Teorema de Carter & Wegman: Todas las operaciones tienen tiempo de ejecución *O(1)*

Carter, L., Wegman, M. (1979). Universal Classes of Hash Functions. Journal of Computer and System Sciences, 18(2), 143–154

Letra pequeña: *valor esperado para cualquier conjunto de datos arbitrario*

Para analizar este teorema concentrémonos en la operación “más costosa”: una búsqueda insatisfactoria

Eficiencia de las tablas hash

Sea *S* el conjunto de datos en la tabla hash y consideremos una búsqueda de *x* que no esté en *S*:

Eficiencia = $\underbrace{O(1)}_{\text{Tiempo de evaluación de } h(x)} + \underbrace{O(\text{longitud de la lista en } A[h(x)])}_{\text{Llamémosla } L}$

L es una variable aleatoria que depende de *h*

Analizar el valor esperado de esta variable es muy complejo, por esta razón utilizaremos el principio de descomposición estadística (analizar una variable en términos de otras más simples que la componen)

Para $y \in S$ (por tanto $y \neq x$), sea $Z_y = \begin{Bmatrix} 1\ si\ h(y) = h(x) \\ 0\ en\ caso\ contrario \end{Bmatrix}$

Siendo así, tenemos que $L = \sum_{y \in S} Z_y$

Luego, por linealidad $E[L] = \sum_{y \in S} E[Z_y]$

Pero $E[Z_y] = 0 * \Pr[Z_y = 0] + 1 * \Pr[Z_y = 1] = \Pr[h(x) = h(y)]$

La clase pasada vimos que $\Pr[h(x) = h(y)] = 1/n$, por tanto $E[L] = \sum_{y \in S} \frac{1}{n} = \frac{|S|}{n}$

A este valor se le conoce como la densidad α de la tabla hash y representa la relación entre la cantidad de claves en la tabla y la cantidad de espacios del arreglo donde se almacenan.

Si por ejemplo *n* = 2*|S|, se tendría que α = 0.5

Eficiencia de las tablas hash

Momento, momento !Paren todo!

en el mundo real |S| no es constante!

y ¿qué es esa vaina de “análisis amortizado”?

El análisis de que la eficiencia de la tabla hash es igual a *O(1)* se fundamenta en que $n = c*|S|$ (para ser más estrictos deberíamos decir que $n \geq c*|S|$*).* Pero si la tabla es dinámica (en la mayoría de los problemas es así) llegaría un punto que, al ir insertando elementos, dicha relación se violaría.

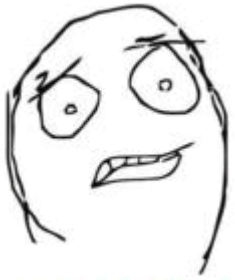
El efecto de ello sería que las posiciones del arreglo se verían más pobladas de lo “saludable” y la consecuencia sería que esa eficiencia se iría a la basura.

Eficiencia de las tablas hash

La solución para este problema es bastante simple: de tanto en tanto (cuando α excede un valor constante α_{max} para ser exactos) se debe aumentar el tamaño del arreglo y los elementos almacenados deben ser "re-hasheados" a sus nuevas posiciones.

Cada operación de redimensionamiento toma *O(n)* siendo *n* el tamaño final. Siendo así la eficiencia de una operación de inserción en el peor escenario no sería O(*1*) si no O(*n*).

El "truco" es que, pese a este peor escenario, la inserción de *n* elementos toma solamente O(*n*). Esto se debe a que decimos que una inserción tiene un **tiempo amortizado** *O(1)* porque en promedio es así.



Uhm?

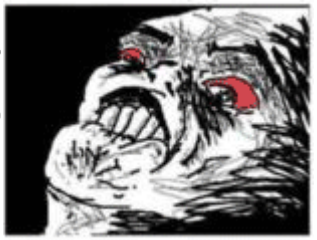
Eficiencia de las tablas hash

Para ver el por qué de la afirmación anterior supongamos que insertamos *m* elementos en una tabla hash y que duplicamos el tamaño del arreglo cada que α supere un α_{max} = 0,5.

Si comenzamos con un tamaño del arreglo *m*=2, todos los "rehashings" ocurrirían en potencias de 2 (la primera vez cuando *m*=|*S*| = 1, luego cuando m=2, luego m=4 y así sucesivamente).

Viéndolo de otra manera si $m = 2^k$, el último "rehashing" ocurre con los *m* elementos, el penúltimo con *m/2,* el antepenúltimo con *m/4* y así sucesivamente. Siendo así la cantidad total de operaciones debido al "rehasing" sería $m + m/2 + m/4 + m/8 + ... = m(1 + 1/2 + 1/4 + 1/8 + ...) \leq 2m$

En otras palabras el costo real de insertar *m* elementos es O(*m*) + O(*2m*), es decir, la constante es mayor, pero constante al fin y al cabo.



Problema candidato: 2 suma S

Dejemos la teoría y vamos a la práctica!

Entrada: Un arreglo *A* no ordenado de *n* números enteros no repetidos y un valor *S*

Salida: cantidad de duplas A_i, A_j tal que $A_i + A_j = S$, $i \leq j$

Ejemplo: A = {1, 2, 3, 4, 5, 6, 7, 8, 9} y S = 10

Respuesta: **5**

En general para este problema ¿cuántas duplas diferentes hay (considerando que se puede evaluar *i=j*)? *n(n+1)/2*

2 suma S

Solución #1: fuerza bruta

//Todo lo siguiente es pseudocódigo, no código de Java

```
function twoSumS(A,n,S){
   c = 0
   for i=0:n-1{
      for j=i:n-1{
         if A[i] + A[j] = S
            c++
      }
   }
   return c
}
```

¿Cuál es la eficiencia de este algoritmo? $O(n^2)$, nada impresionante

2 suma S

Solución #2: preOrdenamiento

```
function twoSumS(A,n,S){
   c = 0
   sort(A)
   for i=0:n-1{
      if (binarySearch(A,n,S-A[i])
         c++
   }
   return c
}
```

¿Cuál es la eficiencia de este algoritmo?

$O(n * \log(n))$ para el ordenamiento más n veces búsqueda binaria que es $O(\log(n))$, esto da $O(2 * n * \log(n))$ = $O(n * \log(n))$ … mejor pero no lo suficiente

2 suma S

Solución #3: tabla hash

```
function twoSumS(A,n,S){
   ht = new hashMap
   for i=0:n-1{
       ht.put(A[i], A[i])
   }
   c = 0
   for i=0:n-1{
      if (ht.containsKey(S-A[i]))
         c++
   }
   return c
}
```

¿Cuál es la eficiencia de este algoritmo? $2n = O(n)$, mucho mejor

3 suma S

Compliquemos un poco el problema anterior:

Entrada: Un arreglo *A* no ordenado de *n* números enteros no repetidos y un valor *S*

Salida: cantidad de ternas A_i, A_j, A_k tal que $A_i + A_j + A_k = S$

Ejemplo: A = {1, 2, 3, 4, 5, 6, 7, 8, 9} y S = 10

Respuesta: **20**

En general para este problema ¿cuántas ternas diferentes hay (considerando que se puede evaluar *i=j=k*)? *n(n+1)(n-1)/6*

Se podrá generalizar el algoritmo anterior basado en la tabla hash para obtener un mejor resultado?

3 suma S

```
function twoSumS(A,n,S){
   ht = new hashMap
   for i=0:n-1{
       ht.put(A[i], A[i])
   }
   c = 0
   for i=0:n-1{
      for j=i:n-1{
         if (ht.containsKey(S-A[i]-A[j]))
            c++
      }
   }
   return c
}
```

¿Cuál es la eficiencia de este algoritmo? $O(n) + O(n^2) = O(n^2)$

Intersección de k arreglos

Entrada: *k* arreglos no ordenados, todos de tamaño *n* (en otras palabras, una matriz de *k* x *n*)

Salida: lista que corresponde a la intersección (elementos comunes) de todos los arreglos

Ejemplo: {1,2,3,4; 2,3,4,5; 3,4,5,6}

Respuesta: **{3,4}**

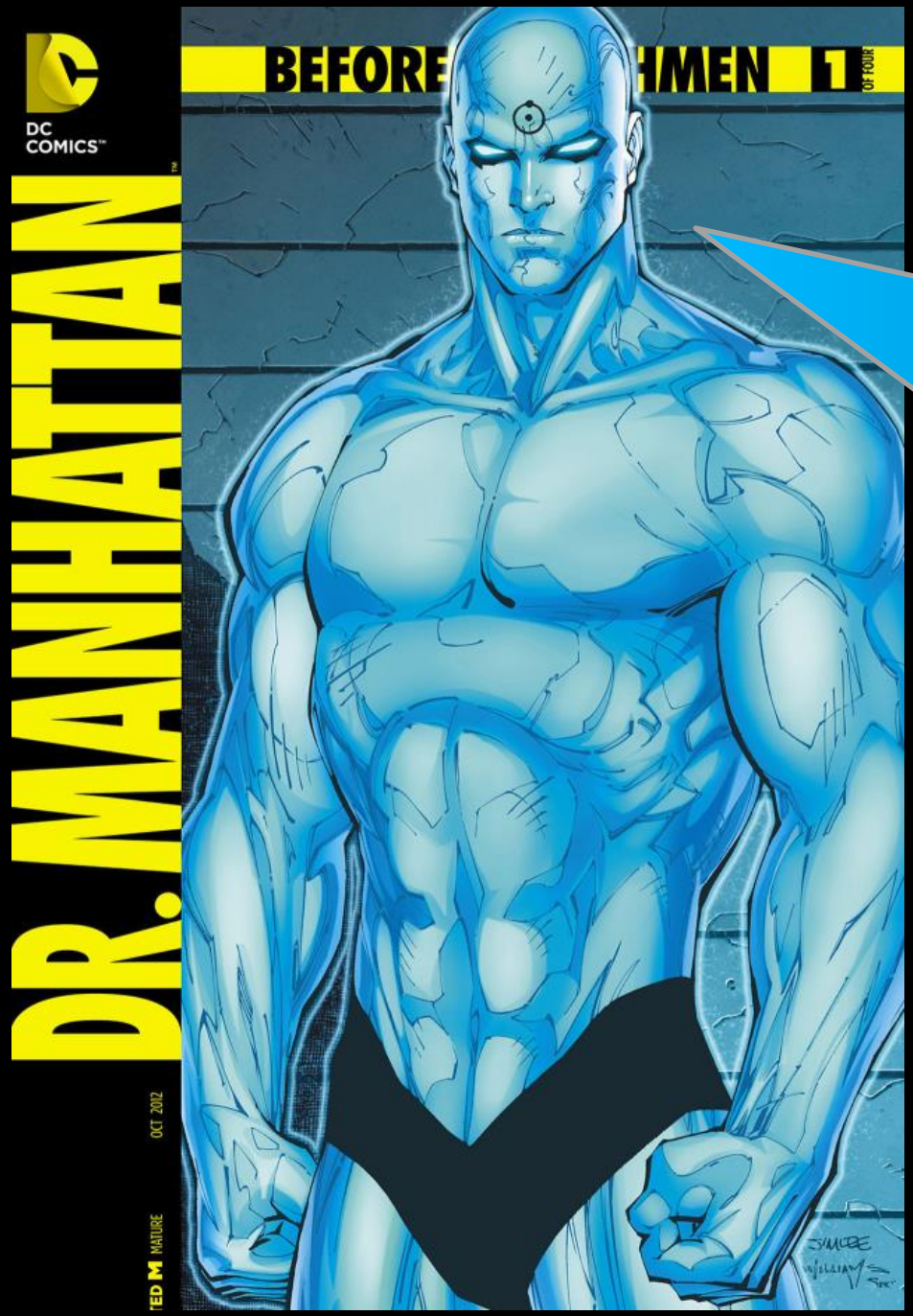
De manera similar a los problemas anteriores, podemos imaginarnos tres alternativas de solución:

- Alternativa 1: Fuerza bruta, $O(n^2k)$
- Alternativa 2: Preordenamiento de las filas 1 a *n-1* y búsqueda por bisección, $O(nk * log(n))$
- Alternativa 3: Arreglo de tablas hash, $O(nk)$

```
function arrayIntersection(A,n){//recibe los arreglos como una matriz

   HashMap<Integer, Integer> ht[] = new HashMap[k-1];
   LinkedList<Integer> b = new LinkedList();
   for j=0:k-1{
      ht[j] = new HashMap();
      for i=0:n-1{
         ht[j].put(A[j][i], A[j][i])
      }
   }
   for i=0:n-1{
      c = 0
      for j=1:k-1{
         if (ht[j].containsKey(A[0][i])
            c++
      }
      if c = k
         b.add(A[0][i])
   }
   return b

}
```



Resto de estructuras
... tiemblen ante mí