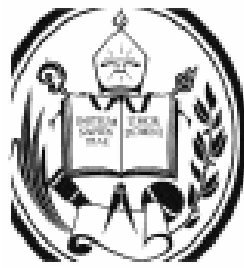


Métodos Ordenamiento

y

Búsqueda de la Mediana



UNIVERSIDAD
DE LOS ANDES



Br. José Gregorio Justo Torres
Departamento de Computación
Diseño y Análisis de algoritmos

Ordenación y Búsqueda

- **Ordenamiento por fusión (MergeSort)**
- **Ordenamiento Rápida (QuickSort)**
- **Búsqueda de la mediana “Selección”**

Introduccion

- Al realizar el ordenamiento de n elementos siempre ha sido un problema a la hora del tiempo de ejecución. Estos problemas los hemos resuelto anteriormente ordenando por selección y por inserción que para el caso medio y peor es de orden cuadrática a diferencia del ordenamiento por montículo (*heapsort*) que tiene un tiempo de $\Theta(n \log n)$. Hay algoritmos de esquema divide y vencerás y vamos a estudiar el **ordenamiento por fusión** y **ordenamiento rápido** además veremos la **búsqueda de la mediana**.
- **Los textos y ejemplos a continuación fueron tomados del libro Fundamentos de Algoritmia. Autor: Brassard G.**

Ordenación por fusión (MergeSort)

- **Este método utiliza la técnica de Divide y Vencerás para realizar la ordenación del vector a . Su estrategia consiste en dividir el vector en dos subvectores, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos subvectores ya ordenados.**

Ordenación por fusión (MergeSort)

Al ordenar un vector mediante las llamadas recursivas y después fusionar las soluciones de cada parte , necesitamos un algoritmo eficiente para fusionar 2 matrices ordenadas en una única matriz cuya longitud sea la suma de las longitudes de las matrices ordenadas. Esto se puede lograr de una forma mas eficiente y sencilla si se dispone de un espacio adicional al final en las matrices ordenadas para utilizarlo como centinela, este centinela es un valor previamente acordado y que se este seguro que no va a ser mayor que cualquier elemento de las matrices ordenadas.

Además otro punto para mejorar la eficiencia del ordenamiento se utiliza como subalgoritmo el ordenamiento por inserción.

Ordenación por fusión (MergeSort)

Tiempos de ejecución

La separación de T en U y V requiere un tiempo lineal ,
es fácil ver que *fucionar*(u ,v ,T) tambien requiere de
un tiempo lineal por lo tanto:

$$t(n)=t(\lfloor n/2 \rfloor)+t(\lfloor n/2 \rfloor) + g(n) , \text{ donde}$$

$$g(n) \in \Theta(n)$$

$$t(n)= 2 t(n/2) + g(n) , \text{ para } n \text{ impares, } g(n) \in \Theta(n).$$

Para n pares es un caso especial por tanto:

$$t(n) \in \Theta(n \log n).$$

Por lo tanto el “merge Sort” en el peor caso tiene un
tiempo igual al *Ordenamiento por montículo*.

Implementación del MergeSort

Procedimiento fusionar (Entero [] Vector U, Entero [] Vector V , Entero [] Vector T Entero m, Entero n)	
{ pre: m , n > 0 }	{ pos: }
<pre> 1 Entero i, j =1,1 2 U[m+1], V[n+1] = ∞ 3 (si(U[i] < V[j]) entonces T[k] = U[i] i=i+1 sino T[k] = V[j] j=j+1 f_si) k = 1 ; k < m+n ; k++ </pre>	<p>i , j : iteradores para recorrer vectores U y V de tipo entero.</p> <p>T : vector final de tipo entero.</p> <p>U y V : vectores de tipo entero con tamaños n y m respectivamente.</p> <p>k : iterador del repita para hasta m + n.</p> <p>∞: centinela al final de los vectores U y V .</p>

Implementación del MergeSort

Procedimiento ordenarporfusión (Entero [] Vector T, Entero n)	
{pre: n > 0}	{pos: }
<pre> 1 Si n es suficientemente pequeño entonces insertar(T) 2 sino Entero U[1..n/2] = T[1...n/2] Entero V[1..n/2] = T[1+ [n/2]...n] ordenarporfusión(U[1..n/2] , n/2) ordenarporfusión(V[1..n/2] , n/2) fusionar(U , V , T) f_si </pre>	<p>n: tamaño del vector T de tipo Entero.</p> <p>U, V: vectores del tipo Entero con la mitad de los valores de T cada uno.</p> <p>insertar: metodo de ordenamiento por inserción , como subalgoritmo.</p> <p>fusionar: metodo que une los 2 vectores ordenados U y V en T.</p>

Ejemplo: Algoritmo de ordenación (merge-sort)

3 - 1 - 4 - 1 - 5 - 9 - 2 - 6 - 5 - 3 - 5 - 8 - 9

3 - 1 - 4 - 1 - 5 - 9

2 - 6 - 5 - 3 - 5 - 8 - 9

1 - 1 - 3 - 4 - 5 - 9

2 - 3 - 5 - 5 - 6 - 8 - 9

1 - 1 - 2 - 3 - 3 - 4 - 5 - 5 - 5 - 6 - 8 - 9 - 9



Ordenación rápida (QuickSort)

- **Esta es probablemente la técnica más rápida conocida. Fue desarrollada en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). El algoritmo fundamental es el siguiente:**

Ordenación rápida (QuickSort)

- **Eliges un elemento del vector. Puede ser cualquiera.Lo llamaremos elemento de división(pivote).**
- **Acomodas los elementos del vector a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores. En este momento el elemento de división separa el vector en dos subvectores.**
- **Realizas esto de forma recursiva para cada subvector mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados.**

Ordenación rápida (QuickSort)

- **Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo.**

Ordenación rápida (QuickSort)

Tiempo de ejecución

El QuickSort que conocemos todos el cual recorre el vector con 2 iteradores para el caso medio tiene un tiempo de $\Theta(n \log n)$ y en el peor de los casos es $\Theta(n^2)$ para ordenar n elementos , por otro lado tomando una nueva forma de elegir el pivote seria el algoritmo *pivotebis*($T[1..j]$, p , var k , l) el cual particiona a T en 3 secciones:

Ordenación rápida (QuickSort)

Tiempo de ejecución

Los elementos de $T[i...k]$ son los menores que p , los $T[k+1 \dots l-1]$ son los iguales a p y los $T[l.... j]$ los mayores que p , con esta modificación , la ordenación de una matriz de elementos iguales requiere un tiempo lineal, ahora tenemos un tiempo tanto para el caso medio como el peor de $\Theta(n \log n)$ pero no es la mejor forma si se selecciona como pivote la mediana de $T[i...j]$ ya que hay que evitarla porque la constante oculta asociada a esta versión <<mejorada>> de quicksort es tan grande que da lugar a un algoritmo , peor que ordenación por montículo en todos los casos.

Implementación del QuickSort

(Relizado por Br. Justo José)

Procedimiento pivotebis(Entero Vector [] arreglo , Entero left , Entero righth, const Entero & p , Entero & k , Entero & l)

{ pre: left , righth , p , k , l > 0 }

{pos: }

```
1   Entero i = left
2   queue<Entero> iz , c , de
3   [ si ( arreglo[i] == p )
      c.push(p);
      sino
        si ( arreglo[i] < p )
          iz.push(arreglo[i]);
        sino
          de.push(arreglo[i]);
        i++;
    ] i <= righth
4   Entero j=left;
5   [ arreglo[j++]=iz.front();
      iz.pop();
    ] !iz.empty()
6   k=j-1;
7   [ arreglo[j++]=c.front();
      c.pop();
    ] !c.empty()
8   l=j-1;
9   [ arreglo[j++]=de.front();
      de.pop();
    ] !de.empty()
```

i , j : iterador para recorrer el vector arreglo de tipo entero.

arreglo : vector de tipo entero.

iz , c , de : colas de la STL de tipo Entero donde se almacena los menores , iguales y mayores que p.

left , righth : variables de tipo entero las cuales almacenan posición izquierda y derecha del arreglo.

k , l : posiciones con la que se van a hacer las siguientes particiones.

Implementación del QuickSort

Procedimiento quick_sort(Entero Vector [] arreglo , constante Entero & left , constante Entero & righth)

{ pre: left , righth > 0 }

{pos: }

```

1 Si ( righth -left es suficientemente pequeño entonces
   Insertion_sort( arreglo , left , righth+1 )
2 sino

   Entero k , l;

   Entero p= arreglo[righth];
   pivotebis(arreglo ,left ,righth , p , k , l);
   quick_sort(arreglo , left , k);
   quick_sort(arreglo , l+1 , righth );
   f_si
    
```

arreglo : vector de tipo entero.

left , righth : variables de tipo entero las cuales almacenan posición izquierda y derecha del arreglo.

k , l : posiciones con la que se van a hacer las siguientes particiones.

Insertion_sort : subalgoritmo para ordenar el arreglo.

p: variable de tipo entero que funciona como pivote del arreglo.

pivotebis : metodo que divide el arreglo en tres parte menores , iguales y mayores que p (pivote).

Ejemplo: Algoritmo de ordenación (QuickSort)

5 - 3 - 7 - 6 - 2 - 1 - 4

5 - 3 - 7 - 6 - 2 - 1 - 4

Implementación del método pivotebis

3 - 2 - 1 - 4 - 5 - 7 - 6

Implementación del método insertar

1 - 2 - 3

5 - 6 - 7

1 - 2 - 3 - 4 - 5 - 6 - 7



Búsqueda de la mediana

Sea $T[1 \dots n]$ una matriz de enteros y sea s un entero entre 1 y n . Se define el s -ésimo menor elemento de T como aquel elemento que se encontraría en la s -ésima posición si se ordenara T en orden no decreciente. Dados T y s , el problema de encontrar el s -ésimo elemento de T se conoce como el problema de selección. En particular se define la mediana de T como su $\lfloor n/2 \rfloor$ -ésimo elemento.

Búsqueda de la mediana

Para calcular la mediana de todos los elementos de T no se puede hacer con tanta facilidad por lo tanto se ordena la matriz T para extraer el $[n/2]$ -esimo , para ello se puede utilizar la ordenación por montículo u ordenación por fusión , esto requiere un tiempo de $\Theta(n \log n)$ y luego utilizamos el método *pivotebis*(T,p,k,l) el cual como ya se había mencionado particiona la matriz en tres partes.

Búsqueda de la mediana

Para calcular la mediana de todos los elementos de T no se puede hacer con tanta facilidad por lo tanto se ordena la matriz T para extraer el $[n/2]$ -esimo , para ello se puede utilizar la ordenación por montículo u ordenación por fusión , esto requiere un tiempo de $\Theta(n \log n)$ y luego utilizamos el método *pivotebis*(T,p,k,l) el cual como ya se había mencionado particiona la matriz en tres partes.

Búsqueda de la mediana

Para optimizar este algoritmo se utiliza el metodo de la *pseudomediana*($t[1..n]$) el cual logra que el tiempo de busqueda por selección tenga en el peor de los casos un orden de $\Theta(n)$,por lo tanto la mediana se puede hallar en un tiempo lineal en el peor de los casos.

Para esto se sustituye en el pivote la función de la *pseudomediana*(T).

Ejemplo: Algoritmo de Búsqueda de la Mediana

Matriz en la que hay que hallar el 4º elemento mas pequeño

3 - 1 - 4 - 1 - 5 - 9 - 2 - 6 - 5 - 3 - 5 - 8 - 9

Se particiona la matriz como pivote $p = 5$, con pivotebis

3 - 1 - 4 - 1 - 2 - 3 - 5 - 5 - 5 - 9 - 6 - 8 - 9

Se selecciona la parte izquierda ya que en esta parte esta la posición buscada

3 - 1 - 4 - 1 - 2 - 3 - . - . - . - . - . - . - .

Ejemplo: Algoritmo de Búsqueda de la Mediana

Se particiona esta parte utilizando como pivote su mediana $p=2$

1 - 1 - 2 - 3 - 4 - 3 - . - . - . - . - . - . - . - .

Se selecciona la parte derecha ya que en ésta parte esta la posición buscada

. - . - . - 3 - 4 - 3 - . - . - . - . - . - . - . - .

Se particiona esta parte usando como pivote la mediana $p = 3$

. - . - . - 3 - 3 - 4 - . - . - . - . - . - . - . - .

La respuesta es 3 , porque el pivote esta en la 4º posición

Implementación Búsqueda de la Mediana

función seleccion(Entero Vector[] arreglo , constante Entero & rigth , constante Entero &pos)	
{ pre: rigth , pos > 0 }	{pos: p > 0 }
<pre> 1 Entero i=0, j=rigth, k, l; 2 [Entero p=mediana(arreglo , i , j) pivotebis(arreglo , i, j, p, k, l) si(pos <= k) entonces j=k; sino si(pos >= l+1) entonces i=l+1; sino devolver p;]verdadero </pre>	<p>Rigth : tamaño del vector T de tipo Entero .</p> <p>p : pivote del arreglo con el valor de la mediana.</p> <p>k, l: posiciones donde es menor y mayor que p respectivamente.</p> <p>mediana : metodo en el cual ordena el vector y luego selecciona la mediana.</p> <p>pivotebis : metodo que divide el vector en tre partes menores , iguales y mayores que p respectivamente.</p>

Implementación Búsqueda de la Mediana

(Realizado por Br. Justo José)

Function mediana(constante Entero Vector [] arreglo , constante Entero & righth)	
{ pre: righth , pos > 0 } {pos: arreglo[m] > 0 }	
1	Merge_sort(arreglo, 0, right)
2	Entero m=(right/2);
3	devolver arreglo[m];
	Righth : tamaño del vector T de tipo Entero . Merge_sort : metodo para ordenar el arreglo. m : mediana del arreglo

Implementación Búsqueda de la Mediana

Function pseudomediana(Entero Vector [] arreglo , constante Entero & n)	
<pre>{ pre: n > 0 } { pos: selección(Z , [z/2]) > 0 }</pre>	
<pre>1 Si(n <= 5) entonces devolver medianadhoc(arreglo) f_si 2 Entero z=[n/5] 3 Entero Vector Z[1...z] 4 (Z[i]= medianadhoc(arreglo[5i-4...5i])) i =1 ; i<z ; i=i+1 5 devolver seleccion(Z,[z/2])</pre>	<p>n : tamaño del vector T de tipo Entero .</p> <p>medianadhoc: metodo diseñado especialmente para hallar la mediana de un maximo de 5 elementos.</p> <p>Z : arreglo temporal para buscar la mediana de n/5.</p> <p>i: iterador para llenar la matriz Z.</p>

Búsqueda de la mediana

El algoritmo de búsqueda de la mediana (selección) empleado con *pseudomediana* halla el *s*-ésimo elemento más pequeño entre n , con un tiempo hasta en el peor de los casos de $\Theta(n)$, la mediana se puede hallar en un tiempo lineal en el peor de los casos.

Siendo $p = \textit{pseudomediana}(T[i..j])$

