Informe de trabajo:
“Divide y vencerás: Algoritmo de búsqueda de la mediana en un vector desordenado.”

Diseño y Análisis de Algoritmos

Miguel Castro Caraballo
Carlos Troyano Carmona

Introducción

En el campo de la estadística, *mediana* es el nombre que recibe el valor que ocupa el lugar central de un conjunto de datos cuando están ordenados.
A diferencia de la *media (promedio)*, la *mediana* es un parámetro de centralización que se utiliza para dividir la muestra o conjunto de datos en dos partes iguales, es decir, dejando un 50% de los valores a la izquierda y el otro 50 a la derecha.

Para hallar la mediana hay que tener en cuenta el número de valores que constituyen la muestra.

- Si número de valores es impar: La mediana es el valor central que divide la muestra en dos partes iguales.
- Si el número de valores es par: La mediana es el promedio aritmético de los dos términos centrales.

Descripción del problema

Dado un array de n elementos no ordenado, se quiere encontrar la mediana de dicho conjunto de datos mediante la técnica de divide y vencerás.

Este problema se puede resolver de dos formas:

1. Ordenando todos los elementos y hallar la mediana.
2. No ordenar los elementos e intentar hallar la mediana.

Aunque se adelanten acontecimientos, la primera forma de resolver el problema no resultaría en un algoritmo demasiado eficiente comparado con el segundo, pues ordenar los elementos con un algoritmo como el Quicksort nos da un peor caso de $O(n^2)$, un caso promedio de O(n log n) y un mejor caso de O(n log n).
En el siguiente apartado se verán cuáles son los órdenes de complejidad de los otros algoritmos que mejoran el primer tipo de solución y son éstos en los que se centrará este informe.

Algoritmos

Entre los algoritmos más usados para hallar la mediana de un array no ordenado encontramos dos que se usan con mayor frecuencia:

- Selection Algorithm - QuickSelect
- Median of Medians

QuickSelect

El funcionamiento de éste algoritmo es similar al Quicksort, pero como sabemos en qué partición se encuentra nuestro elemento, reducimos el problema en ½ después de cada partición.

El objetivo es realizar llamadas recursivas a las funciones de partición hasta dar con aquel elemento que, después de reordenar los elementos, queda en la mitad (n/2). El pivote a seleccionar puede ser el primer elemento del array, el último o usando valores aleatorios.

En el promedio de casos, como eliminas la mitad de los elementos se tiene que O(n + n/2 + n/4 + …), que es lo mismo que decir O(2 * n) == O(n).

Sin embargo, este no es el mejor de los algoritmos.

Podríamos encontrar un mejor pivote que hiciera más eficiente la búsqueda de la mediana utilizando otro algoritmo.

Median of Medians

Este algoritmo consiste, como dice su nombre, en hallar la mediana de una lista de medianas.

Primero, se divide la lista de valores en grupos de 5 elementos (con la posibilidad de que el último pueda ser más pequeño), después, se halla la mediana en cada uno de esos grupos.
Con esto, obtendremos el que sería el pivote ideal para hallar la mediana.

Si se escogiera el pivote de forma aleatoria (como en el apartado anterior), se esperaría que el algoritmo funcionase con una complejidad de orden lineal. Sin embargo, el peor caso tendría una complejidad de $O(n^2)$. Es por esto que este algoritmo garantiza encontrar un pivote no muy lejano de la verdadera mediana de forma lineal.

Pseudocódigo

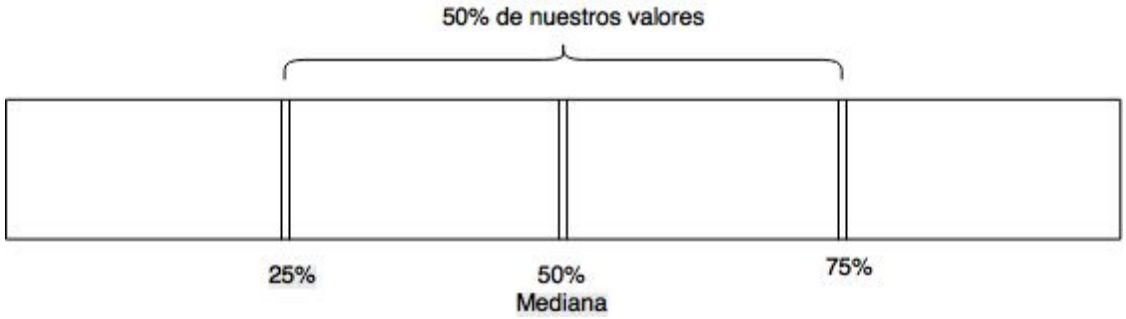
```
function select(V, left, right, n)
      if left = right
          return V[left]
      k := partition(V, left, right)
     lenght := k - izquierda + 1
      if lenght = k
         return V[k]
      else if n < k
         return select(V, n, left, k-1)
      else
         return select(V, n - leght, k + 1, right)

function partition(list, left, right)
    pivotValue := pivot(list, left, right)
    while(left < rigth)
        while(V[left] < pivotValue)    left++;
        while(V[right] > pivotValue)    right++;
     if(V[left] == V[right]) left++;
     else if swap V[left] and V[right]
return right;
```

```
function pivot(V, left, right)
    // for 5 or less elements just get median
    if right - left + 1 < 9:
        sort(V)
        return V[V.length / 2]
    tmp as array;
    medians as array;
    medianIndex := 0;
    while (left <= right)
        tmp.size := MIN(5, right - left -1)
          for i from 0 to tmp.size AND left <= right
            tmp[i] := V[left];
            left++;
        sort(tmp)
        medians[medianIndex] = tmp[tmp.size / 2];
        medianIndex++;
return pivot(meidans, 0, medians.size -1);
```

Análisis del algoritmo

El algoritmo QuikSelect tiene un tiempo de O(N²). Mejorando la selección el pivote puede mejorarse bastante. Vemoas que pasaría si hiciéramos de manera aleatoria la selección del pivote.



Un buen pivote se considera que está cerca de la mediana. Si el recuadro de arriba fuese nuestro conjunto de datos un buen pivote estará situado lo más cerca del medio posible. Diremos que la zona que va desde el 25% al 75% es un buen pivote la otra zona es un mal pivote.Hagamos un análisis del tiempo de ejecución esperado de esto.

Tenemos el 50% de probabilidades de escoger un buen pivote y el 50% de no hacerlo por lo que representaremos dicha situación como ½. Por lo tanto. Si el tiempo esperado es **$T_e(N)$** entonces:
$T_e(N) <= ½ T_e(¾ N) + ½ T_e(N) + O(n)$
Restamos $½ T_e(N)$ =
$½ T_e(N) <= ½ T_e(¾ N) + O(n)$
Multiplicamos por 2 =
$T_e(N) <= T_e(¾ N) + O(n)$
Por el teorema maestro sabemos que el tiempo esperado es el caso 1 O(n).

Con esto vemos que si elegimos un buen pivote tenemos un caso lineal pero tenemos un 50% de probabilidades de escoger un mal pivote. Por eso el algoritmo la media de las medias mejora la selección del pivote con un tiempo lineal para poder después ejecutarlo en un tiempo lineal.
Lo primero que haremos será dividir el conjunto de datos S de entrada en un subconjuntos

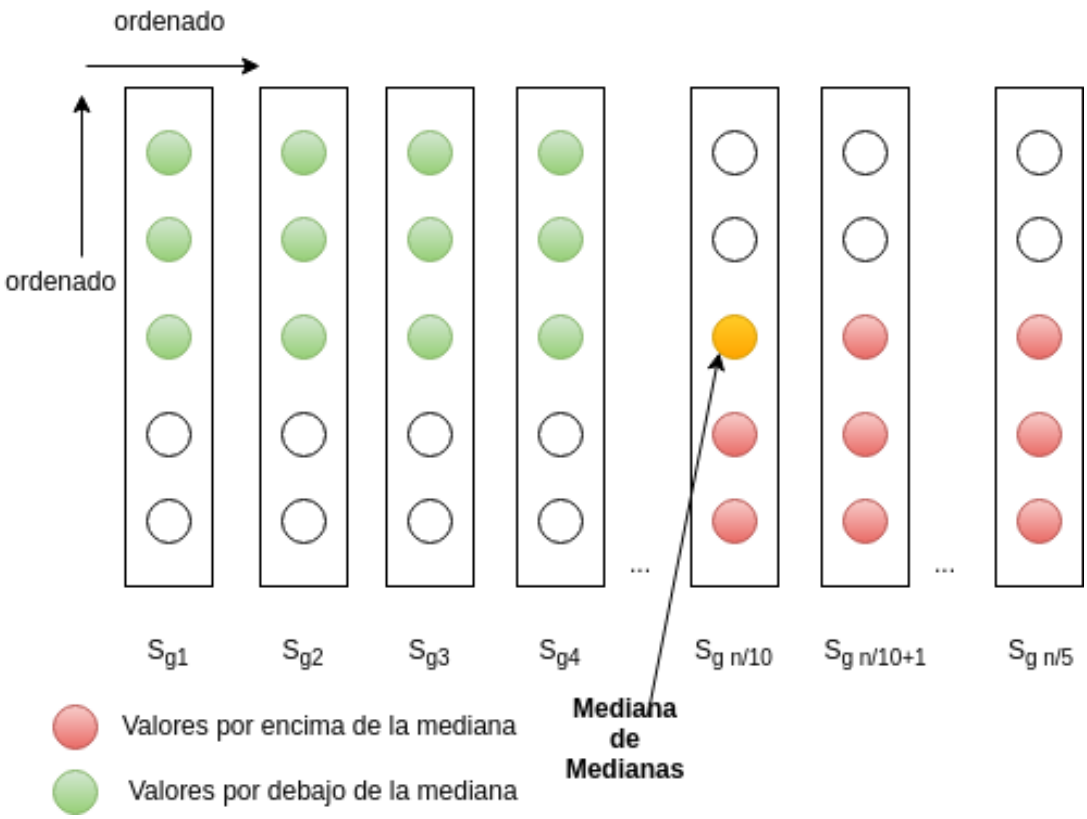
$S_{s1}=\{x_1,x_2,x_3,x_4,x_5\}$

$S_{s2}=\{x_6,x_7,x_8,x_9,x_{10}\}$

...

$S_{sn/5}=\{x_{n-1},x_n\}$

De tal manera que tendremos n/5 subgrupos. Buscar las medianas iniciales en los grupos de 5 elementos supondrá un tiempo constante O(1). Para ello se ordenaran dichos subconjuntos y se retornara su valor central (Mediana).
Buscar la mediana de las medianas recursivamente tendrá un coste de O(n/5) ya que el coste se dividirá por el número de grupos en relación con el total de datos de la entrada.
Por otro lado la búsqueda de del QuickSelect recursivo se reduce ya que parte de los elementos están ordenados y no se crean las ramas correspondientes en el árbol de recursión.



Como vemos en la figura de arriba la mitad de los conjuntos en su totalidad se quedan fuera de la recursión. Por lo que cuando realicemos las particiones iniciales derecha e izquierda sabemos una cosa sobre ellas 3 elementos de estas no se comprarán en la recursión por lo tanto sabemos que las mitades serán:

$|S_{left}|$ >= 3 n/10

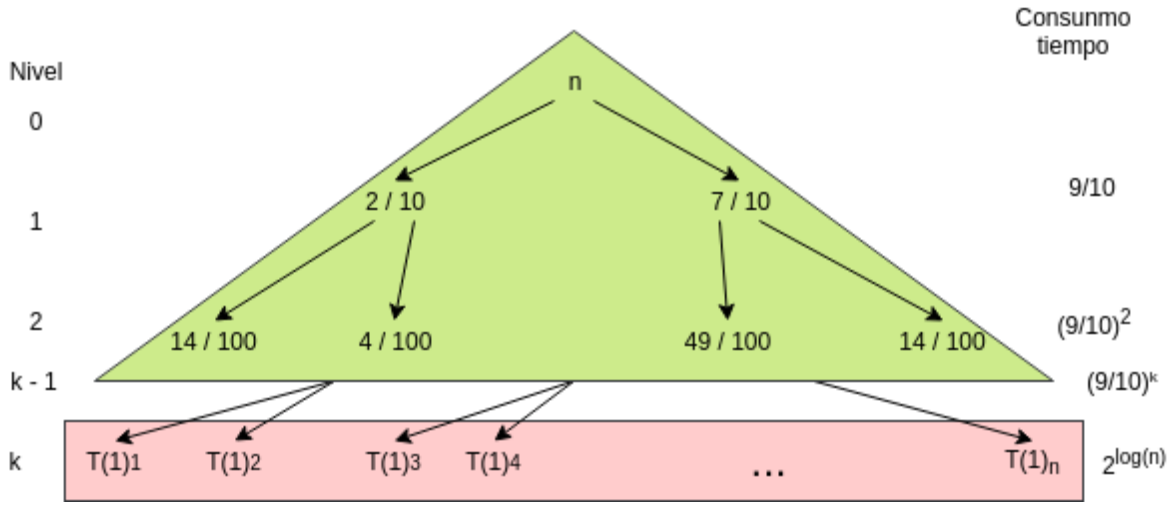
$|S_{right}|$>= 3 n/10

Combinados : **$|S_{left}|$ <= n - $|S_{right}|$ <= 7/10 n**

Por lo que la recursión tomará una forma tal que **T(n7/10).**

Además el tiempo para dividir el problema en QuickSelect sigue siendo O(n). Por lo que nuestra cuota será: **T(n) = T(n7/10) + T(n/5) + O(n)**

Árbol de recursión



Como vemos el árbol tiene un tamaño de k niveles donde la altura k en el peor caso en n. por lo que la suma de todos los casos base es **$2^{log(n)}C$** y el resto del árbol es $n \sum_{i=0}^{n} (9/10)i$

Resolvemos la progresión aritmética con la fórmula. 1 / 1 - (9 / 10) = 10.

Entonces **T(n) = $2^{log(n)}C$ +10C + O(n) => T(n) = O(n)**

Por lo que llegamos a la conclusión de que el algoritmo es de tiempo polinomial O(n).

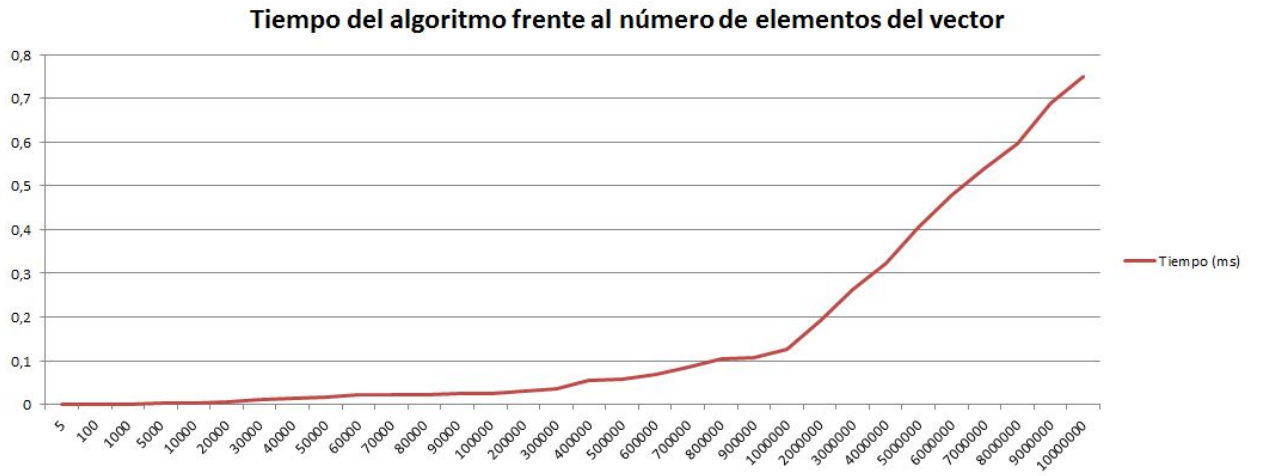
Ahora lo comprobaremos experimentalmente.

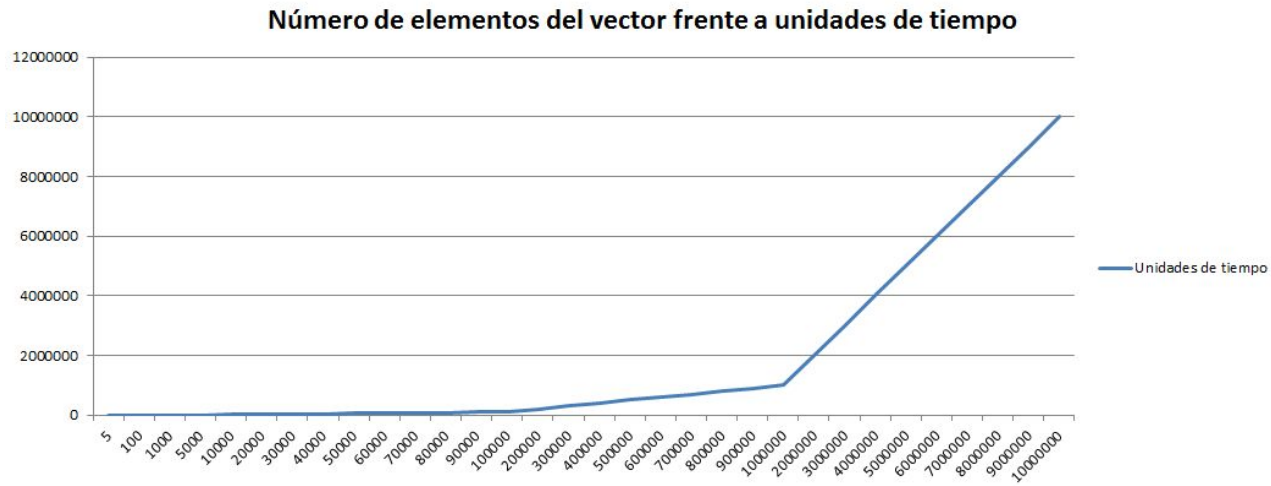
Evaluación experimental

Se ha probado a ejecutar el algoritmo con 32 tamaños distintos para el vector de números. El rango de los intervalos varía para mostrar de forma más amplia la eficiencia del tiempo del algoritmo con respecto a los posibles números de elementos que pudiera tener el vector.

- Los primeros 4 elementos representan cantidades comunes: 5, 100, 1000, 5000
- Los siguientes 10 elementos representan pruebas con incrementos de 10.000 elementos (desde 10.000 hasta 100.000)
- A continuación, los próximos 10 elementos representan pruebas con incrementos de 100.000 de elementos (desde 100.000 hasta 1.000.000)
- Los últimos 10 elementos son pruebas con incrementos de 1 millón en 1 millón hasta llegar a los 10 millones.

Número de elementos	Tiempo (ms)
5	0
100	0,001
1000	0,001
5000	0,002
10000	0,004
20000	0,007
30000	0,011
40000	0,014
50000	0,017
60000	0,022
70000	0,021
80000	0,022
90000	0,024
100000	0,026
200000	0,031
300000	0,037
400000	0,056
500000	0,059
600000	0,068
700000	0,084
800000	0,103
900000	0,107
1000000	0,126
2000000	0,193
3000000	0,262
4000000	0,324
5000000	0,404
6000000	0,479
7000000	0,538
8000000	0,597
9000000	0,689
10000000	0,75





Conclusiones

A la vista de los resultados obtenidos se pueden extraer las siguientes conclusiones:

- Las pruebas demuestran de forma empírica que el algoritmo “Median of Medians” resuelve el problema en tiempo lineal. En la misma gráfica se aprecia la tendencia de los valores a formar una línea recta.
- El algoritmo “Median of Medians” es un claro ejemplo de algoritmo que, dada una gran cantidad de datos, obtiene un resultado en un tiempo eficiente. Esto demuestra una vez más la importancia de hallar algoritmos eficientes.

Bibliografía

- https://en.wikipedia.org/wiki/Quickselect (Wikipedia)
- https://en.wikipedia.org/wiki/Median_of_medians (Wikipedia)
- http://cs.indstate.edu/~spitla/presentation.pdf (Indiana State University)
- http://www.vitutor.com/estadistica/descriptiva/a_9.html (Vitutor)