Tarea 1 Diseño y análisis de Algoritmos

Diego Muñoz, Sebastian Ramos

14 abril 2016

0.1 Introducción

El K-esimo y el top-M son algoritmos de selección bastante eficaces, ya que su solución sencilla, es de orden $O(n*log(n))$ y aún lo son mas sus soluciones mejoradas, de orden $O(n)$ como ya demostraremos en el informe.

Estos algoritmos se caracterizan por usar la técnica de programación "Divide and Conquer", la cual nos ayuda a reducir el tiempo de ejecución, cuando debemos encontrar elementos en listas por ejemplo.

Conocer de estos algoritmos resulta muy útil ya que al enfrentarse a algoritmos que deban usar entradas de muchos elementos, podremos saber como hacerlo de una forma mas eficiente.

EL algoritmo K-esimo básicamente nos sirve para cualquier caso en el que necesitemos encontrar el k elemento mas grande. Una aplicación para este algoritomo es en la aplicaciones microeconómicas

El algoritmo top-m nos sirve para cualquier caso en el que necesitemos encontrar los primeros m elementmos de un arreglo. Una aplicación de este algoritmo es al momento de elegir a los empleados del mes en el área de ventas, en el cual por una base de datos, se seleccionan a los que mas ganancias han aportado a sus empresas.

0.2 Descripcion de algoritmos

K-esimo simple:

El algoritmo K-esimo en su solución simple, ordena los elementos para luego entregar la posición k-1 de la lista. Utiliza un algoritmo de ordenamiento mergesort, el cual usando la técnica "dividir y conquistar, tiene un orden $O(n * log(n))$ convirtiéndose este en el orden de nuestra función, debido a la simplesa del resto de la función que es tiempo lineal.

```
1  def k_esimosimple(k,array):
2          if k == 0 :
3              print "k es invalido"
4          else :
5              nuevo=merge_sort(array)
6              return nuevo[k-1]
```

Como podemos ver la linea 2, 3, 4 y 6, cumplen un tiempo constante, y la linea 5 cumple un tiempo, en el peor caso de, $O(n * log(n))$

K-esimo Mejorado:

El algoritmo K-esimo en su solución mejorada, al igual que el método simple, ordena los elementos para luego entregar la posición k-1 de la lista, pero a diferencia de la solución simple, este utiliza un algoritmo llamado "Median of Medians" o bien, "Mediana de Medianas, que hace exactamente lo que dice, encuentra la mediana de las medianas, subdividiendo el problema en arreglos de tamaño 5, encontrando las medianas de estos, y calculando la mediana de las medianas, devolviendo una posición ideal para utilizar como pivote ya que las particiones seran equivalentes, resultando el ordenamiento, en el peor caso $O(n)$

```
1  def k_esimosimple(k,array):
2          if k == 0 :
3              print "k es invalido"
4          else :
5              nuevo=mediana(array,0,len(array)-1,k)
6              return nuevo[k-1]
```

Al igual que la solucion simple, las lineas tienen el mismo tiempo excepto por la quinta, que es corresponde al metodo de ordenamiento en donde el mejorado, es inferior al simple ya que ordenar con mejorado nos toma $O(n)$ y el simple $O(n * log(n))$

Top-m simple:

El algoritmo Top-m en su solucion simple, ordena los elementos para luego entregar la posicion m-1 de la lista. Utiliza un algoritmo de ordenamiento mergesort, el cual tiene orden $O(n * log(n))$ y al igual que nuestra funcion K-esimo simple y mejorada, su código es simple ya que depende de su algoritmo de ordenamiento. Nota: tambien usamos esta libreria para implementar nuestro mergesort luegode haber tenido problemas con la anterior implementación.

```
1  def topmSimple(m, array):
2          if m == 0:
3                  print "m es invalido"
4          else :
5              nuevo=merge_sort(array)
6              return nuevo[:m]
```

Igual que el K-esimo, la linea 2, 3, 4 y 6, cumplen un tiempo constante, y la linea 5 cumple un tiempo, en el peor caso de, $O(n * log(n))$

Top-m Mejorado:

El algoritmo Top-m en su solucion mejorada, al igual que el metodo simple, ordena los elementos para luego entregar la posicion m-1 de la lista. A diferencia de la solucion simple, usamos un algoritmo conocido, llamado Heapsort, que python tiene en sus librerias, importando "heapq". Este algoritmo tambien conocido como cola de prioridad, se comporta como arbol binario y su raiz siempre es el elemento mas pequeño.

```
1def topmMejorado(m, array):
2       arrayO = array[:]
3       heapq.heapify(arrayO)
4       return [heapq.heappop(arrayO) for i in range(m)]
```

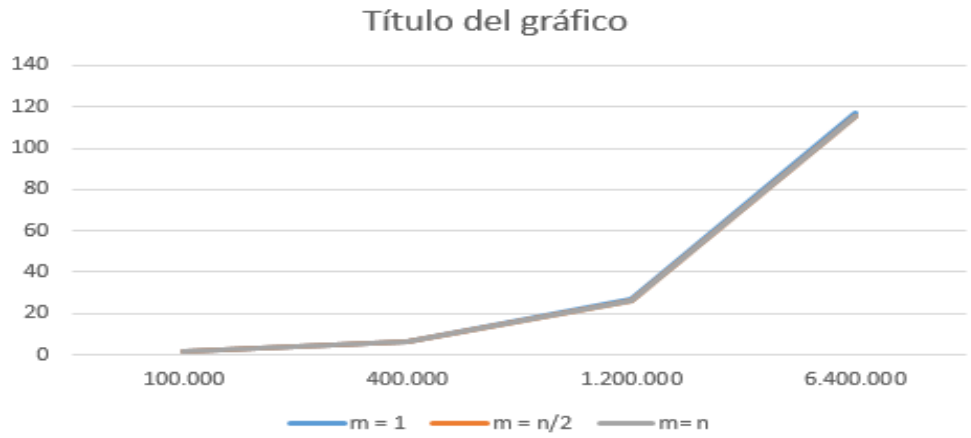
Primero, pasamos la lsit a una variable, luego heapify() acomoda la lista a orden de heap, para luego retornar con heappop() los m elementos. Este algoritmo utiliza tiempo lineal para encontrar su resultado, osea, $O(n)$ el cual es menor que la solucion simple, que es $O(n * log(n))$.

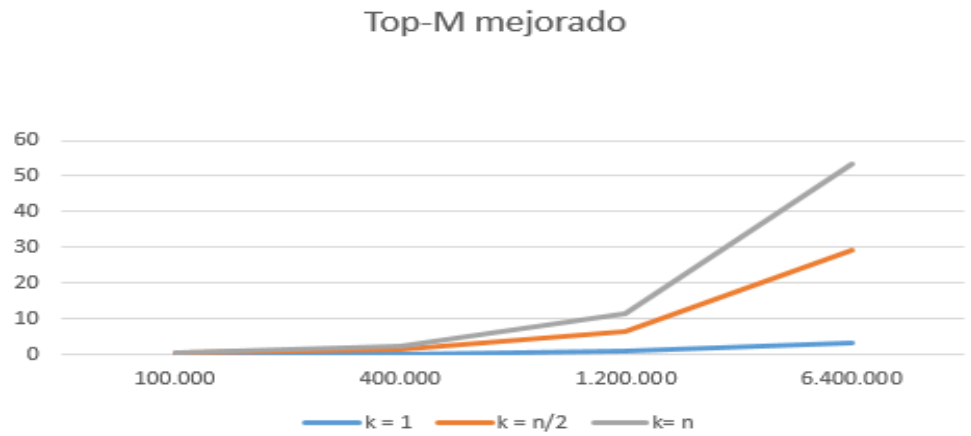
0.3 Experimentación

Para comprobar que nuestra solucion mejorada era mejor empiricmente, hicimos los siguientes graficos. Estos deberian evidenciar una gran diferencia en tiempo, ya que uno tiene orden $O(n * log(n))$ y el otro $O(n)$

Como podemos apreciar en nuestros graficos, revelan lo que pensabamos, el k-esimo mejorado toma mucho menos tiempo que el simple. Cuando nuestro peor tiempo del mejorado es 34 se. aprox. en el simple es 109 seg aprox. lo cual es una notable diferencia. En el gráfico del k-ésimo simple, el tiempo es casi el mismo cuando se pide encontrar el primer elemento, n/2 o n, por ende hay solo una curva debido a que estos se superposicionan.

En estos graficos podremos ver las diferencias entre algoritmos top-m. Al igual que en el analisis anterior, esperamos ver una notoria en los tiempos de ejecucion.





A simple vista se puede dar cuenta de la similitud con el grafico anterior. Cuando nuestro peor caso de la solución mejorada es 53 seg aprox. la solucion simple toma un valor de 115 seg aprox.
En el gráfico del top-m simple, el tiempo es casi el mismo cuando se piden encontrar 1, n/2 o n elementos, por ende hay solo una curva debido a que estos se superposicionan.
Con esto podemos afirmar que las funciones mejoradas son realmente mas rápidas. Destacamos que para las pruebas utilizamos un algoritmo que creará archivos de entrada con una lista desordenada y sin ni un elemento repetido.

0.4 Conclusión

Luego de haber realizado las comprobaciones de forma empírica con distintas entradas, podemos concluir que la tecnica "Divide and Conquer" resulta muy útil al resolver problemas de ordenamiento en forma recursiva. En el k-esimo utilizamos el algoritmo Mediana de Medianas para mejorar nuestra solución de $O(n*log(n))$, el cual es fácil identificar en su código, que tomamos prestado, el uso de dividir y conquistar, ya que divide el problema en 5 para resolverlo. En el Top-m al utilizar la función heapq.hipify utiliza el método ya mencionado, divide and conquer, el cual nos ayuda a resolver el problema de manera mas eficaz.

0.5 Bibliografía

Algoritmo de sottolo:
-http://terminus.ignaciocano.com/k/2011/03/29/desordenando-listas-en-python/

Manejo de ficheros python:
-http://blog.zerial.org/ficheros/Informe$_Ordenamiento.pdf$

Mediana de medianas:
-http://www.ardendertat.com/2011/10/27/programming-interview-questions-10-kth-largest-element-in-array/

Merge sort con heapq
-https://rosettacode.org/wiki/Sorting$_algorithms/Merge_sortPython$

Mediana de medianas explicacion:
-http://functionspace.com/articles/19/Median-of-Medians

Funcion heapq
-http://docs.python.org.ar/tutorial/2/stdlib2.html?highlight=heap

-https://docs.python.org/2/library/heapq.html

Aplicaciones microeconómicas
-https://books.google.cl/books?id=pMdDCQAAQBAJpg=
PA720lpg=PA720dq=aplicacion+para+el+k-esimo
source=blots=zSuoR2bpRqsig=Q7TYrBtfyfE9GWUDVe3GJ
6EbJrQhl=essa=Xved=0ahUKEwidtbmCwY$_M AhVKSSYKHeu$
$4AO84ChDoAQgeMAIv = onepageq = aplicacion$