Índice

1. Notas

1.1. Brute Force

1.1.1. Orden

A simple vista, calcular si un elemento es o no el 4to es O(n), ya que se fija cuántos son menores a él entre todos los otros. Como lo hacemos potencialmente para todos los elementos, este método es $O(n^2)$.

1.1.2. Sugerencia

Para almacenar los elementos menores al que se está analizando, no convendría directamente usar una lista y utilizar append? Cada elemento nuevo es **siempre** O(1) (a diferencia del set, donde es en promedio O(1), y seguramente es una estructura más sencilla y más rápida para estos fines.

1.1.3. Mejor caso

Cuando el k-ésimo elemento es el primero en la lista. Ejemplo: k = 4 l = [4,15,2,1,0,14,6,11,8,9,3,13,12,7,5,10] O(n)

1.1.4. Peor caso

Cuando el k-ésimo elemento es el último en la lista. Ejemplo: k = 4 l = [10,15,2,1,0,14,6,11,8,9,3,13,12,7,5,4] O(n^2)

1.2. Sort and Select

1.2.0.1. Orden.

El timsort (algoritmo de ordenamiento usado por python) es O(n logn) en el peor caso (lineal en el mejor). Una vez que está hecho el orden solo se toma el elemento k de la lista en O(1). Por lo tanto, este algoritmo sería O(n logn).

1.2.1. Potencial Problema

La lista indexa desde 0. Debería devolver el elemento de índice k-1.

1.2.2. Mejor caso

El mejor caso del Timsort se da cuando la entrada ya está ordenada. Ejemplo: k = cualquiera l = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] O(n)

1.2.3. Peor caso

TODO O(n log n)

1.3. k-selecciones

En la selección se analizan los n elementos de la lista y se pone en primer lugar al más pequeño. Luego, sobre los n-1 restantes se repite el proceso. Luego sobre n -2 y así k veces en total. Cada selección, sobre una lista de n elementos, es O(n) (debe recorrerlos todos para ver el mínimo).

Este algoritmo tiene k selecciones, con lo cual es entonces kO(n) (tanto k como n son parte de la entrada, ninguna es constante). Ya que k < n, esto seguramente sea menor a O(nˆ2). Salvo que k sea menor a log(n), este algoritmo es superado por el Sort and Select.

Ver: En estos casos estamos tomando k como 0-based, por lo que aquí, la selección se haría "k+1" veces.

1.3.1. Potencial Problema

El mismo que en Sort and Select.

1.3.2. Mejor caso

La complejidad del algoritmo depende de 'k', por lo que el mejor caso se da cuando k = 0 (realiza la selección parcial 1 sola vez), y esto se da con cualquier entrada de cualquier tamaño 'n'.

1.3.3. Peor caso

Por el mismo argumento anterior, el peor caso se da cuando k = n

1.4. k-heapsort

Un heapsort es un ordenamiento de selección, solo que se usa un heap de mínimo para obtener el mínimo en O(logn) en lugar de O(n). Si bien observar el mínimo de un heap es O(1), como lo quitaremos debe hacerse un upheap o float, que es O(logn).

Entonces, el heapsort consiste en dos etapas:

1. Heapify. Se reordena el array para que sea un heap. Esto es O(n).
2. Las n extracciones del mínimo O(n logn).

Por lo tanto, un heapsort normalmente es O(n) + O(n logn) = O(n logn).

Aquí, al igual que con el k-selecciones, quitaremos los primeros k elementos. Por esto, aquí se hará el heapify, sucedido de k extracciones del mínimo. El orden sería O(n + k logn) con k < n.

1.5. HeapSelect

Esta era una función de python que se encarga de algo parecido: ### Nota sobre heapq.nsmallest

Este algoritmo de python que se encuentra en [https://hg.python.org/cpython/file/3.4/Lib/heapq.py#l195] consiste en lo siguiente:

1. Toma los primeros k elementos de la lista y los convierte en un heap de máximo con heapify. Esto es O(k).
2. Mantiene los k mínimos en el heap. Para esto recorre cada elemento de la lista y, si es menor al máximo lo agrega al heap y quita el nuevo máximo. Si es mayor al máximo, no lo agrega. Para cada elemento cuesta O(logk) con lo cual este paso completo es O(n logk).
3. Ordena el heap de k elementos, lo cual es O(klogk) y devuelve esa lista.

El tiempo entonces es O (k + n logk + k logk) = O(n logk).

Como explica la documentación, esta función es rápida par k pequeño. Si k se acerca a n, termina siendo más conveniente directamente ordenar la lista, ya que de todos modos tendremos que ordenar en el 3er paso la mayor parte de los elementos en el paso 3. Esta comparación de cuándo conviene cada una es justamente una comparación entre este algoritmo y el Order and Select.

1.5.1. Solución Diseñada

No nos conviene usar el heapq.nsmallest, ya que el 3er paso ordena los k más pequeños, siendo que nosotros solo necesitamos obtener el máximo de estos últimos. Nos basaremos en esta documentación para implementarlo, pero variando el paso 3. En aquel paso solo observaremos el máximo, que justamente es la cabeza del heap. Esto es O(1) y es lo que se devuelve.

El orden queda entonces: O(k + n logk + 1) = O(n logk). Es básicamente la función de python, pero sin el orden del final, lo cual reduce sustancialmente el tiempo.

1.6. QuickSelect

Se usa la lógica del Quicksort: se define un pivote y se ponen todos los elementos mayores “a la derecha” y todos los menores “a la izquierda”. Según la posición p del pivote: - Si p == k: se devuelve l[p] - Si p < k: se hace QuickSelect sobre la parte derecha (l[p+1..n]) - Si p > k: se hace QuickSelect sobre la parte izquierda(l[0..p]) Esto entonces, en el caso óptimo de que el pivote siempre queda a la mitad tiene un tiempo de T(n) = T(n/2) + O(n) [n es lo que tarda en poner todo en su lado correspondiente del pivote].

Utilizando el Teorema Maestro, esto es O(n).

1.7. Nota general sobre k

k <= n. Sería absurdo pedir algo como el n+1 menor elemento.

Si k == n, entonces el k mínimo es el máximo, y puede encontrarse en O(n) más rápido que con cualquiera de los otros. De hecho, todos los algoritmos antes descriptos pueden ser mejorados sustancialemnte con este razonamiento. Si k > n/2, entonces el k-mínimo es el n-k-máximo y será más barato buscar al k-máximo con el mismo algoritmo.