The Selection Problem and QUICK-SELECT

1 The selection problem

Given a set A of n elements, and an integer k ($1 \leq k \leq n$), we want to find the kth smallest element in A.

SELECT(A, k): returns the k'th smallest element in A

For simplicity we'll assume that the elements in A are distinct and we denote them as $\{a_1, a_2, ..., a_n\}$. (we'll come back to this at the end).

Examples:

- If $k = 1$ we want the *s*mallest element in A
- If $k = 2$ we want the *s*econd smallest element in A
- If $k = n$ we want the nth smallest element in a set of n elements, which is the *largest* element.
- ...
- if $k = \lceil \frac{n}{2} \rceil$ the element in the "middle" of the set is called the *median*. The median has the property that half the elements are smaller, and half are larger.

  For e.g. the median of $A = \{3, 1, 7, 5, 2\}$ is 3. There are two elements smaller than 3 and two elements larger than 3. Element 3 is the $\lceil 5/2 \rceil = 3$ third smallest element in A.

  To be precise, a set of odd size has one median, the element with rank $\lceil \frac{n}{2} \rceil$. A set of even size has two medians, the $\frac{n}{2}$-smallest and $(\frac{n}{2} + 1)$-smallest elements

  For e.g. set $A = \{15, 1, 2, 10, 6, 8\}$ of $n = 6$ elements has two medians: the 3rd smallest element, which is 6, and 4th smallest element, which is 8.

  When we talk about the median of an even set of elements, we can just pick one of them, either the lower or the higher.

Rank of an element: Another term we'll use with the elements of an array is the **rank**: The smallest element is said to have rank=1; the second smallest has rank=2; generally the kth smallest element is said to have rank=k. The rank of an element is the place of that element in sorted order.

2 How to find the kth smallest? Brainstorming

2.1 Selection via sorting

Let's try to come up with an algorithm for Select(A, k). An idea would be to sort the set A, and then return the k^{th} element in sorted order. This is obviously correct, and will run in $O(n \lg n)$ time by using any of the $O(n \lg n)$ sorting algorithms.

Intuitively it seems that finding the k^{th} smallest element is "easier" than sorting, and by sorting we may be doing unnecessary work. Can we do better?

2.2 Selection via repeated Find-Min

- To add to that intuition, note that if $k = 1$ we know how to find the smallest element in $O(n)$ time!
- Same for $k = n$
- Similarly, we can find the second smallest element by finding the smallest twice, in $2 \cdot O(n) = O(n)$
- We can extend this idea and find the kth smallest by repeatedly finding the smallest element, k times. Overall it would take $k \cdot O(n) = O(k \cdot n)$. Finding the median with this approach would take $\frac{n}{2} \cdot O(n) = O(n^2)$
- So....this approach works in linear time is if k is a small constant, but yields quadratic time worst-case
- Can we improve this? An idea that might come to mind is using a better way to find the smallest element rather than traversing the whole array in linear time. Do we know a structure that supports FIND-MIN efficiently?
- Yes, it's the heap!

2.3 Selection with a heap

- We can put all the n elements in A in a heap, and then call Delete-Min k times. The element returned by the last Delete-min is the kth smallest.
- How long would this take? Building a heap can be done ine $O(n)$ time, and Delete-Min in $O(\lg n)$ time, for a total of $O(n + k \cdot \lg n)$ time.
- This is better than quadratic time, and it's in fact better than sorting for small values of k, but not in the worst-case: for $k = n/2$, this approach runs in $O(n \lg n)$ which is same as sorting.

3 Quick-select

Here's another idea for finding the kth smallest element, which is similar, in some sense, to binary search:

- Take for e.g. $A = \{3, 1, 5, 7, 2, 8, 9, 6, 4\}$. Let's say we pick an element of A as pivot, for e.g. we could pick the last element 4 as the pivot. Based on the pivot we can partition the set into two sets: the elements smaller than the pivot, and the elemenst larger than the pivot: $A_1 = \{x_i | x_i < 4\}$ and $A_2 = \{x_i | x_i > 4\}$. In our case, $A_1 = \{3, 1, 2\}$ and $A_2 = \{5, 7, 6, 8, 9\}$
- So we have $A_1 = \{3, 1, 2\}$, the pivot 4, and $A_2 = \{5, 7, 6, 8, 9\}$
- Because the size of A_1 is 3, we know that the pivot is the 4th smallest element.
- So we picked a pivot and partitioned A into A_1 and A_2. What we do next depends on the value of k:
- If $k = 4$, ie if we need to return the 4th smallest element, we return the pivot and we are done!
- If $k < 4$, this means that the element we are looking for must be in A_1. For e.g. if $k = 2$, we know the second smallest element is in A_1. So we recursively find the kth smallest element in A_1 and return it: return $Select(A_1, k)$

- If $k > 4$, this means the element we are looking for must be in A_2. For e.g. if $k = 6$, we know that the 6th smallest element must be among A_2. So we recurse on A_2, but now we must account for all the elements in A_1 and the pivot which we left behind and are smaller. That is, the 6th smallest element in A is the same as the $k - 3 - 1$=2nd smallest element in A_2, which is 6.

- QUICK-SELECT(A, k)

  1. Base case: If A has 1 element: return it // k must be 1
  2. Pick a pivot element p from A (could be the last one). Split A into two subarrays A_1 and A_2 by comparing each element to p. While we are at it, count the number l of elements going into A_1.
  3. (a) if $k \leq l$: return QUICK-SELECT $(A_1,\ k)$
     (b) if $k == l + 1$: return p
     (c) if $k > l + 1$: return QUICK-SELECT $(A_2,\ k - l - 1)$

- Why does this work? All the elements in A_1 are $<$ pivot, and all the elements in A_2 are $>$ pivot. So if the rank of the element we are looking for is smaller than the rank of the pivot (i.e. $k < l + 1$), then the element must be smaller than the pivot, so we can "throw away" A_2 and recurse only on A_1. Similarly, if the rank of the element we are looking for is $>$ the rank of the pivot, then the element must be in A_2. The kth smallest element in A is the same as the $(k - l - 1)$th smallest element in $A - \{p + A_1\} = A_2$.

- This algorithm may remind you of binary search, in that it only recurses in one side of the partition. However binary search does $\Theta(1)$ work to decide where to recurse, whererea this algorithm needs to partition, so that's $\Theta(n)$ work before deciding which side to recurse. The other difference is that unfortunately picking an arbitrary element as pivot does not guarantee a half-half split.

- **Analysis:**

  - If the partition was perfect i.e. $(|A_1| = |A_2| = n/2)$ at each step we'd have:

    $$T(n) = T(n/2) + \Theta(n)$$

    which solves to $T(n) = \Theta(n)$
  - However, it is possible that the pivot is the smallest/largest element in A, which leads to $T(n) = T(n-1) + \Theta(n)$, which solves to $\Theta(n^2)$

- Thus QUICK-SELECT runs in $O(n^2)$ time in the worst case.

- We could argue (similar to Quicksort analysis) that the partition is balanced on the average if we assume that all input permutations are equally likely. If this is true then QUICK-SELECT runs in $O(n)$ average time. This is only partially useful because this assumption is often not true.

- Solution? Pick the pivot randomly. This has the effect of "randomizing" the imput and has an expected $O(n)$ time algorithm **no matter what the input distribution is**.

  The resulting selection algorithm is referred to as RANDOMIZED-SELECT or QUICK-SELECT and has expected worst-case running time $O(n)$ irrespective of the input distribution.

- Even though a worst-case $O(n)$ selection algorithm exists in practice RANDOMIZED-SELECT is preferred.

4 Selection when the elements are not unique

So far the assumption has been that the elements (keys) in A are unique. This assumption is not necessary and QUICK=-SELECT works perfectly fine if there are duplicate elements.

Consier the following array which contains 3 elements with key 10; a different way to say this is that the *frequency* of 10 is 3.

$$A = \{15, 19, 10, 12, 10, 18, 10, 21\}$$

One of these 10s will have rank 1, one will have rank 2, and one will have rank 3. Therefore calling SELECT(A, 1), SELECT (A, 2) and SELECT(A, 3) should all return 10. Another way to think about this is to imagine the array in sorted order.

$$A = \{10, 10, 10, 12, 15, 18, 19, 21\}$$

The element returned by SELECT(A, k) is the kth element in the sorted array. Having duplicate keys means that several calls to Select() will return the same element.

So what happens when we call SELECT on an array with duplicates? QUICK-SELECT works perfectly fine even when there are duplicate elements. Consider a few examples to convince yourself. Suppose we have $A = [3, 1, 1, 2, 1]$ and we call Select(A, 2). Partition chooses pivot=1, and creates $A_1 = [1, 1]$ and $A_2 = [3, 2]$. Select will check the rank of the pivot, which is 3, and conclude it needs the second element in $A_1 = [1, 1]$. This is correct.

QUICK-SELECT() works with duplicate elements no matter how partitioning handles the elements equal to the pivot (depending on the exact partition algorithm, they will be put in A_1, in A_2, or both — all will work).