Median-Finding and the Select Algorithm

Calculating the Median

Median: We will define the median of n values as:
the (n+1)/2 smallest if n is odd
the n/2 smallest if n is even
(we're not going to take the average of two numbers when n is even)

A naïve approach to finding the median of n numbers

- We can sort the entire list in $O(n \log n)$ time, and then directly access the median element in $O(1)$ time: overall $O(n \log n)$

- But this certainly feels like it is doing more work than necessary
  - We've sorted the entire list for the sake of one number
  - We don't care about the relative ordering of the elements to the left, only that they are all less than or equal to the median value.
  - Similarly we don't care about the relative ordering of the elements to the right, only that they are all greater than or equal to the median value.

A Reference to Quicksort

Quicksort Algorithm

- Works by selecting a pivot element from the array
  - Could be randomly selected
  - Could be deterministically selected (that is, the index of the pivot element could be deterministically selected)
  - Splits the array into two subarrays: elements less than or equal to the pivot, and elements greater than the pivot
  - Recursively calls Quicksort on the two subarrays
  - Upon return, just glue the two pieces together, as they are already ordered.

  - (we could also define a third subarray to correspond to elements equal to the pivot. They don't need further sorting and get glued in the middle when the other two pieces return.)

Quicksort Performance

Quicksort Algorithm Performance: T(n)

- Best case: the pivot roughly splits the array into two equal halves
  - It takes $O(n)$ time to sort the array into the two halves
  - It takes $T(n/2)$ time to Quicksort one of the subarrays

  - $T(n) = 2T(n/2) + O(n)$
  - We've seen this before - MergeSort: $T(n) = O(n \log n)$

- Worst case: the pivot is always the smallest (or largest) element and one of the subarrays is only one element smaller than the previous step
  - $O(n^2)$ in worst case

Quicksort Performance

Importance of the Choice of Pivot element

- As long as the pivot is chosen such that at least a constant fraction of the data is on each side at each step, it can be shown that Quicksort is $O(n \log n)$.
  - The scaling factor, c, in the $O()$ notation gets worse as the distribution gets worse (farther away from $\frac{1}{2}$)

Using the Idea of a Pivot Element to Compute the Median in O(n)

Quicksort without the sort, and throwing away some of the data

- To compute the median, select a pivot element and split the data as before, but this time into three groups
  - Elements less than the pivot
  - Elements equal to the pivot
  - Elements greater than the pivot

  - If the median element falls in the range of indices that are equal to the pivot, then we're done
  - Else recursively call with the group of elements containing the median

- NOTE: this approach is generally called k-select, and can be used to identify the k^{th} smallest element in an array for any value k, not just the median.

Select Algorithm

Worst Case Running Time: $O(n^2)$

Expected Running Time: $O(n)$

```
SELECT-RAND (A, k)
1.  x = a_i where i = a random number from {1,…,n}
2.  Rearrange A so that all elements smaller than
    x come before x, all elements larger than x
    come after x, and elements equal to x are
    next to x
3. j1,j2 = the first and the last position of
    x in rearranged A
4. if (k < j1) return SELECT-RAND(A[1…j1-1], k)
5. if (k ≥ j1 and k ≤ j2) return x
6. if (k > j2) return SELECT-RAND(A[j2+1…n], k-j2)
```

Improved Select Algorithm

Can Select be improved to guarantee that the worst-case running time is O(n) ?

- We need a way to guarantee that the choice of pivot is always good enough to ensure that we discard a constant fraction of the data at each step.
  - But we also have to be careful that the amount of work we do to find such a pivot doesn't negate the gains.

- Use "median of medians" approach to guarantee that the pivot is good enough

- NOTE – this approach will work for any k^{th} smallest element that we are searching for. Not just the median.

Improved Select Algorithm

Running Time: $O(n)$

```
SELECT (A, k)
0.  If |A| ≤ 5, solve by sorting
1.  split A into n/5 groups of five elements
2.  let b_i be the median of the i-th group
3.  let B = [b_1, b_2, …, b_n/5]
4.  medianB = SELECT (B, B.length/2)
5.  Rearrange A so that all elements smaller than
    medianB come before medianB, all elements
    larger than medianB come after medianB, and
    elements equal to medianB are next to medianB
6.  j1,j2 = the first and the last position of
    medianB in rearranged A
7.  if (k < j1) return SELECT ( A[1…j1-1], k )
8.  if (k ≥ j1 and k ≤ j2) return medianB
9.  if (k > j2) return SELECT ( A[j2+1…n], k-j2 )
```