Selection in $O(n)$ worst case time

Given an array A of n elements and an integer k ($1 \leq k \leq n$), we want to find the kth smallest element in A. Last time we saw that QUICK-SELECT runs in $O(n)$ expected. Today we introduce an elegant and ingenious algorithm that runs in $O(n)$ worst-case.

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

- The idea is to find a pivot element which guarantees that the partition is relatively balanced.
- How? Chosing a good pivot relies on the following claim:

  Claim: Divide A into groups of 5 and find median of each group. The median of medians, call it x, has the property that at least $3n/10$ elements are guaranteed to be smaller than x; and at least $3n/10$ elements are guaranteed to be larger than x.

  We'll prove this claim a little later.
- Assuming the claim above is true. We'll use it as follows: we'll find a good pivot x, use it to partition, and then do what we were doing before. Here's the resulting algorithm:

SMARTSELECT(set A of n elements, k)

- Divide A into groups of 5 and find median of each group. Copy these medians into an array B (note that B has $\lceil \frac{n}{5} \rceil$ elements).
- call SMARTSELECT(B, $n/10$) recursively on B to find median x of B (note that x is the median of B, or the median of medians).
- Use x as pivot to partition A into A_1 and A_2
- Recurse as before:
  - If $k == |A_1| + 1$: return x
  - If $k < |A_1| + 1$: return SMARTSELECT(A_1, k)
  - If $k > |A_1| + 1$: return SMARTSELECT($A_2, k - |A_1| - 1$)

- ANALYSIS: The running time $T(n)$ consists of:
  - Computing array B with the $n/5$ medians; this can be done in $\Theta(n)$ time.
  - Computing recursively the median of B; this takes $T(\lceil \frac{n}{5} \rceil)$ time.
  - Recursing on A_1 or A_2; this takes $T(n')$ time, where n' is how many elements are in A_1 or A_2, respectively

  So the recurrence for the worst case running time is $T(n) = \Theta(n) + T(\frac{n}{5}) + T(n')$.

  To solve it we need to estimate n': the maximeum number of elements on one side of the partition.

- From the **claim** we know that the number of elements in A_1 is at least $3n/10$; from here it follows that the maximum number of elements in A_2 is at most $7n/10$.

  Similarly, based on the claim we know that the number of elements in A_2 is at least $3n/10$; from here it follows that the maximum number of elements in A_1 is at most $7n/10$.

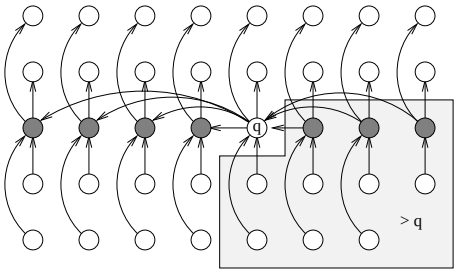
  Therefore the maximum number of element in either A_1 or A_2 is $n' = 7n/10$.

- So SmartSelect(i) runs in :
  $T(n) = \Theta(n) + T(\frac{n}{5}) + T(\frac{7}{10}n)$

- It can be shown that the solution to this recurrence is $T(n) = O(n)$
  (the insight here is that $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$ and the recurrence is "equivalent" to $T(n) = T(9n/10) + \Theta(n)$)

Proof of Claim:

- We split the n elements in groups of 5, and for each group of 5 elements, we find their median
- The groups are drawn below. An arrow between element e_1 and e_2 indicates that $e_1 > e_2$
- For each group, the median is drawn solid
- In each group, the median of that group is guaranteed to be smaller than two elements, and larger than two elements in that group
- The median of medians is labeled as q in the figure (note: should be x).



- How many elements **are guaranteed to be $> q$?**
  The two elements in q' s group, plus 3 elements in each of the groups whose medians are larger than q.
  That's at least $3 + 3 \times \frac{1}{2}\lceil\frac{n}{5}\rceil > \frac{3n}{10}$
- Similarly, the number of elements guaranteed to be $< q$ is at least $\frac{3n}{10}$

Final notes on Selection

- What's magic about groups of 5? will other choices work? see exercises.
- **Selection and Quicksort:** Recall that the running time of Quicksort depends on how good the partition is, which depends on how well the chosen pivot splits the input. We could use SELECT() to find the **median** in $O(n)$ time, and use the median as pivot in PARTITION. Thus we could make quick-sort run in $\Theta(n \log n)$ time worst case!

  In practice, although a worst-case time $\Theta(n \lg n)$ algorithm is possible, it is not used because the constant factors in SELECT are too high. RANDOMIZED-QUICKSORT remains the fastest sort in practice.
- **A brief history of the selection problem:**

  The fast randomized select is due to Hoare, in 1961.

  The worst-case select is due to Blum, Floyd, Pratt, Rivest and Tarjan, in 1973.

  If you think the selection algorithms are hard/smart/elegant, you are right. Hoare (known for Quicksort) got the Turing award in 1980. Rivest is the R in RSA and got the Turing award in 2002. Manual Blum got the Turing award in 1995. Tarjan got the Turing award in 1986. Floyd got the Turing award in 1978.

Study questions and practice problems

1. In the SELECT() algorithm described above, the input elements are divided into groups of 5. Will the SELECT() algorithm work in linear time if they are divided into groups of 7?
2. Argue that SELECT does not run in linear time of groups of 3 or 4 are used.
3. Let A be a list of n (not necessarily distinct) integers. Describe an $O(n)$-algorithm to test whether any item occurs more than $\lceil n/2 \rceil$ times in A. Your algorithm should use $O(1)$ additional space. A general solution should not make any additional assumptions about the integers.
4. (CLRS 9-1) Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms on terms of n and i.

   (a) Sort the numbers, and list the i largest.

   (b) Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

   (c) Use a SELECT algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

Appendix: Selection when the elements are not unique

So far the assumption has been that the elements (keys) in A are unique. For e.g. imagine we have a set of elements such that the smallest element is 10, and that there are 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) should be the element in position k of the sorted array (to be precise $k-1$ if array is indexed at 0). We see that duplicate keys will mean that several calls to Select() will return the same element.

Quick-Select() and Smart-Select() work correctly with duplicate elements. For e.g. let $A = [3, 1, 1, 2, 1]$. Partition chooses pivot=1, and let's say it creates $A_1 = [1, 1]$ and $A_2 = [3, 2]$. Let's say we want the 2nd smallest element. It will check thee rank of the pivot, which is 3, and conclude we need the second element in $A_1 = [1, 1]$. Works.

A more efficient way to handle duplicates: Note however that this it would be more efficient to notice that there are three 1's, and since we look for the second smallest, return 1 right away. See below: The algorithm can be adapted rather easily to handle frequencies. The idea is, when selecting a pivot, **count** the frequency of the pivot and take it into account when deciding to recurse left or right of the partition. Denote by l the number of elements $< x$. For e.g. if the frequency of pivot x is 3:

- If $k \leq l$: we recurse left (that is, on all elements $< x$);
- If k is $l+1, l+2$ or $l+3$: return x
- If $k > l+3$: recurse right (that is, on all elements $> x$), looking for $k-l-3$