Selection: Finding the kth smallest element

Module 3: Efficient Sorting and Selection

Overview

We introduce a new problem, called the *selection* problem: Given a set S of n elements, $\{x_1, x_2, ..., x_n\}$ and an integer k $(1 \leq k \leq n)$, find the kth smallest element in S. We describe several ideas for solving this problem, culminating with an elegant and ingenious algorithm that runs in $O(n)$ worst-case.

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 i'th smallest element in A

For simplicity, we assume the elements in A are distinct and we denote them as $\{a_1, a_2, ..., a_n\}$. (we'll come back to considering how duplicates affect selection 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.
- and so on
- if $k = \lceil \frac{n}{2} \rceil$ the element in the "middle" of the set is called the *median.* This element 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 because 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 a set 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 basically where that element fits in the sorted order of the set.

2 Towards a selection algorithm

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 $2 \cdot O(n) = O(n)$ time
- 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 whether there is any better way to find the smallest element rather than traversing the set 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.

In the next sections we'll explore a new idea and move towards an $O(n)$ time algorithm.

3 Selection via partitioning in $O(n)$ expected time

Here's another idea of how one might approach finding the kth smallest element; the idea 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 we 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 $k \le l$: 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, the algorithm 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—assuming that all input permutations are equally likely—-and we get $O(n)$ average time. This is only partially useful, as this assumption is often not true.
- Solution? Pick the pivot randomly! This has the effect of "randomizing" the imput, and it can be shown gives a balanced partition on the average, and an expected $O(n)$ time algorithm **no matter what the input distribution is**.

  The resulting selection algorithm is referred to as RANDOMIZED-SELECT or QUICKSELECT and has expected worst-case running time $O(n)$ irrespective of the input distribution.
- Even though a worst-case $O(n)$ selection algorithm exists (see below), in practice RANDOMIZED-SELECT is preferred.

4 Selection in $O(n)$ worst-case

- The approach behind Quick-Select() can be extended to get $T(n) = \Theta(n)$ in the worst case
- 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 :

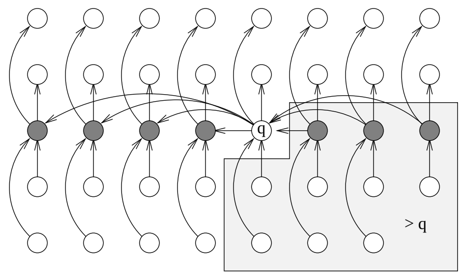
  $T(n) = \Theta(n) + T(\frac{n}{5}) + T(\frac{7}{10}n)$

- 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 $\frac{3n}{10}$

5 Selection when the elements are not unique

So far the assumption has been that the elements (keys) in A are unique.

Defining Select(A,k) when elements are not 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.

SELECT with duplicates: The algorithms we've seen work perfectly fine even when there are duplicate elements. Consider a few examples to convince yourself. Suppose $A = [3, 1, 1, 2, 1]$. 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 $A_1 = [1, 1]$. Yes!

Quick-Select() and Smart-Select() work with duplicate elements, no matter how PARTITION 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).

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:

A SELECT algorithm optimized to handles duplicates: The algorithms for SELECT 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 $k \le l$: 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 $k - l - 3$

6 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.