Linear Time Selection

(CLRS 9)

The selection problem is the following: Given an array A of n elements (assume a 1-based array with elements $A[1]$ through $A[n]$) and a value i ($1 \leq i \leq n$), find the ith smallest element in an array.

For simplicity, we assume the elements are distinct.

$\boxed{\textsc{Select}(A, i)\text{: returns the } i\text{'th smallest element in A}}$

Note that for $i = 1$ we want the *m*inimum element, and for $i = n$ we want the *maximum* element.

Rank: The ith smallest element is said to be the element of rank i. The rank of an element x tells us how many elements are smaller than x: the element x has rank i if precisely $i - 1$ elements are smaller than x (assuming elements are unique). The rank of an element is its index in sorted order.

The element of rank $i = n/2$ is called the *median*. 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 elements with rank $\frac{n}{2}$ and $\frac{n}{2} + 1$. 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.

1 Selection via sorting

Of course we can find the ith smallest element by first sorting A, and then returning the ith element in the sorted array. This approach will run in $O(n \lg n)$ time.

The question is: can we do better? Selecting one element seems easier than sorting so we should be able to do it faster. To add to that intuition, let's look at some special cases of $\textsc{Select}(i)$:

- Minimum or maximum can easily be found in $n - 1$ comparisons: Scan through elements maintaining minimum/maximum.
- Second largest/smallest element can be found in $(n - 1) + (n - 2) = 2n - 3$ comparisons: Find and remove minimum/maximum. Find minimum/maximum again.
- Median:
  - Using the above idea repeatedly we can find the median in time $\sum_{i=1}^{n/2}(n - i) = n^2/2 - \sum_{i=1}^{n/2} i = n^2/2 - (n/2 \cdot (n/2 + 1))/2 = \Theta(n^2)$
  - So....this approach does not scale.

Can we design an $O(n)$ time algorithm for general i?

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

- Recall the PARTITION function employed by Quicksort.
- The idea is to notice that once we PARTITION(A), we can tell which side of the partition has the element we are looking for, just by looking at their sizes.
- For instance, if we are looking for the 53-rd element in an array of 100 elements, and after partitioning the left side of the partition has 30 elements and the right side has 69 elements, then we just need to find the 53 - 30 -1 = 22nd element in the right side of the partition. And if the left side had exactly 52 elements, then we would just return the pivot.
- 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 Partition does not guarantee a half-half split.
- Here's is the generic algorithm:

```
QUICK-SELECT($A, i$)

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

- CORECTNESS: After we partition, all the elements before it are $\leq$ pivot, and all the elements to the right of it are $\geq$ pivot. So if we know that the rank of the element we are looking for is smaller than the rank of the pivot (i.e. $i \leq k$), then we can "throw away" the right side of the partition and recurse only on the left side.
- ANALYSIS: How fast is this, in the worst case? It is crucial that we only recurse on one side.
  - If the partition was perfect ($q = n/2$) we have:

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= n + n/2 + n/4 + n/8 + \cdots + 1 \\ &= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\log n} (\frac{1}{2})^i \leq n \cdot \sum_{i=0}^{\infty} (\frac{1}{2})^i = \Theta(n) \end{aligned}$$

  - If the partition was bad: $T(n) = T(n-1) + n = \Theta(n^2)$

- Thus, the algorithm runs in $O(n^2)$ time in the worst case. We could argue that on average we get $O(n)$ time, assuming that we ran the algorithm on a set of inputs such that all input permutations are equally likely. This is only partially useful, as this assumption is often not true.
- We could use the same trick we used for Quicksort: we make Partition pick the pivot randomly. This is essentially the same as randomizing the input beforehands. It can be shown that picking the pivot

randomly gives a balanced partition on the average. 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.

2.1 A more elegant implementation

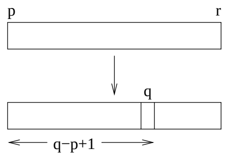
- To avoid copying arrays, we use indices p and r to specify the slice of the array that we are looking at.
- We'll think of the rank i as relative to the current array, that is, to the current slice $p..r$ that we are in. That is, if we are looking at slice $p..r$ of the array, then i must be such that $1 \leq i \leq r - p + 1$.
- Our notation will be the fololwing:

  Quick-Select(A, p, r, i): returns the ith smallest element in $A[p..r]$.

- Example: Quick-Select$(A, 4, 6, 2)$ will return the 2nd smallest element among $A[4..6]$.
- To find the ith element in A, we call Quick-Select$(A, 1, n, i)$

```
Quick-Select($A, p, r, i$)

  1. Base case: IF $p = r$: return $A[p]$ ($i$ must be 1)
  2. $q$=Partition($A, p, r$)
```



```
  3. Let $k = q - p$ //$k$ is the size of the left side of the partition
  4. If $i \leq k$: return Quick-Select($A, p, q - 1, i$) //recurse on left side
  5. If $i == k + 1$: return $A[q]$
  6. If $i > k+1$: return Quick-Select($A, q+1, r, i-k-1$) //recurse on right
     side
```

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

- It turns out that the idea above can be extended to get $T(n) = \Theta(n)$ in the worst case
- The quadratic worst-case running time is caused by imbalanced partitions. The idea is to find a pivot element x which guarantees that the partition is relatively balanced.
- Chosing a good pivot x relies on the following 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 below.

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

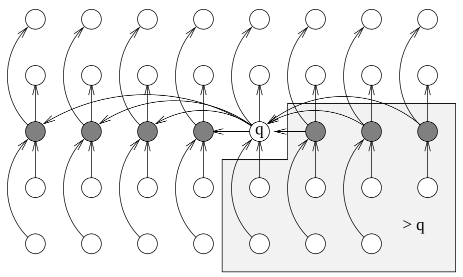
```
SmartSelect(array $A$ of $n$ elements, index $i$)
  – Divide $A$ into groups of 5 and find median of each group. Copy these medians into an array
    $B$ (note that $B$ has $\lceil \frac{n}{5} \rceil$ elements).
  – call SmartSelect(B, $n/10$) recursively on $B$ to find median $x$ of $B$ (median of medians).
  – Partition $A$ using $x$ as pivot: $q$=Partition($A$) //note: rank of $x$ is $q$
  – Recurse as before:
      ∗ If $i == q$: return $x$
      ∗ If $i < q$: return SmartSelect(the elements before the pivot, $i$)
      ∗ If $i > q$: return SmartSelect(the elements after the pivot, $i - q$)
```

- ANALYSIS: The running time $T(n)$ consists of:
  1. Computing the $n/5$ medians takes $\Theta(n)$ time.
  2. Computing recursively the median of B takes $T(\lceil \frac{n}{5} \rceil)$ time.
  3. Recursing on a side pf the partition takes $T(n')$ time, where n' is how many elements fall on that side of the partition.

  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 maximal number of elements on one side of the partition.

- Estimation of n':
  - Consider the following figure of the groups of 5 elements
  - An arrow between element e_1 and e_2 indicates that $e_1 > e_2$
  - The $\lceil \frac{n}{5} \rceil$ selected elements are drawn solid (q is median of these)
  - Elements $> q$ are shown inside the box



  - Number of elements $> q$ is at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil - 2) \geq \frac{3n}{10} - 6$
    - We get 3 elements from each of $\frac{1}{2}\lceil \frac{n}{5} \rceil$ columns except possibly the one containing q and the last one.
  - Similarly, the number of elements $< q$ is at least than $\frac{3n}{10} - 6$

    $\Downarrow$

    We recurse on at most $n' = n - (\frac{3n}{10} - 6) = \frac{7}{10}n + 6$ elements

- So SELECT(i) runs in time $T(n) = \Theta(n) + T(\frac{n}{5}) + T(\frac{7}{10}n + 6)$
- It can be shown that the solution is $T(n) = O(n)$.

4 Selection and Quicksort

- Recall that the running time of Quicksort depends on how good the partition is
  - Best case ($q = n/2$): $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$.
  - Worst case ($q = 1$): $T(n) = T(1) + T(n-1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$.
- How balanced the partition is depends on well the chosen pivot splits the input
- We could call SELECT to find the median in $O(n)$ time, and use the median as pivot in PARTITION.
  - We could just exchange e with last element in A in beginning of PARTITION and thus make sure that A is always partition in the middle
- 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.

5 Selection when the elements are not unique

So far the assumption has been that the elements (keys) are unique. For example, assume that we have a set of elements such that the smallest element is 10, and that there are 3 elements with key 10 (the frequency of 10 is 3). One of these 10s will have rank 1, one will have rank 2, and one will have rank 3. We see that the element returned by SELECT(i) should be the element in position i of the sorted array. and that possibly calls to SELECT(i) with different values of i may return the same element.

The algorithms for SELECT can be adapted rather easily to handle frequencies. The idea is, when selecting a pivot, to count the frequency of the pivot and take it into account when deciding to recurse left or right of the partition. That is, assume there are k elements $< x$, and the frequency of pivot x is 3. If $i \leq k$ we recurse left (that is, on all elements $< x$); if i is $k+1, k+2, k+3$ we return x; if $i > k+3$ we recurse right (that is, on all elements $> x$.

6 How did they come up with this idea?

- Why 5? It is important that we chose every 5'th element; choices ¡ 5 will not work.
- 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.