Linear Time Selection

(CLRS 9)

The selection problem is the following: Given an array A of n elements and a value i ($1 \leq i \leq n$), find the ith smallest element in an array. This element is called the element of $rank = i$.

For simplicity, we assume the elements are distinct.

SELECT(A, i): returns the i'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.

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 $\frac{n}{2} + 1$.

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 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
- 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 $O(n)$ time algorithm for general i?

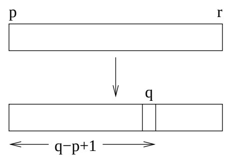
Selection in $O(n)$ expected time via partitioning

- The crucial observation is that once we computed a partition (like in Quicksort), that helps us SELECT.
- Basically, we recursively look for the element in one of the sides of the partition:

```
SELECT($A, p, r, i$)

base case: IF $p = r$ then return $A[p]$

$q$=PARTITION($A, p, r$)
```



```
$k = q - p + 1$ //k is the rank of pivot A[q]
IF $i == k$ then
    //we found it!
    return A[q]
IF $i < k$ then
    //recurse on the left of pivot
    return SELECT($A, p, q, i$)
ELSE
    //recurse on the right of pivot
    return SELECT($A, q + 1, r, i - k$)
```

To find the ith element in A, call SELECT$(A, 0, n-1, i)$

- CORECTNESS: This is pretty clever, but why does this work? We'll see that for efficiency it is crucial that the recursion happens *only on one side* of the partition (not on both). This is correct, because once we partitioned, the pivot is in the right place , 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. $i < k$), 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? The trick is that we only recurse on one side.
  - If the partition was perfect ($q = n/2$) we have:

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

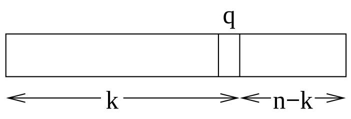
  - If the partition was bad: $T(n) = T(n-1) + n = \Theta(n^2)$
- Thus, the algorithm runs in $O(n^2)$ time in the worst case. We could argue that on average we get $O(n)$ time, assuming that we ran the algorithm on a set of inputs such that all input permutations were equally likely. This assumption is often not true though...

- But, we could use the randmonized version of partition, RANDOMIZED-PARTITION, that gives a balanced partition on the avarege. The resulting selection algorithm is referred to as RANDOMIZED-SELECT and has expected running time $O(n)$ (on any set of inputs).
- Even though a worst-case $O(n)$ selection algorithm exists (see below), in practice RANDOMIZED-SELECT is preferred.

Selection in $O(n)$ worst-case

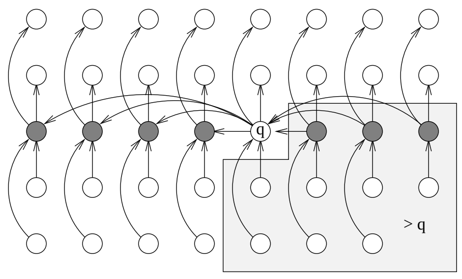
- It turns out that we can modify the algorithm and get $T(n) = \Theta(n)$ in the worst case
- The idea is to find a pivot element x such that we always eliminate a fraction of the elements:

```
SELECT($i$)
  – Divide $n$ elements into groups of 5
  – Select median of each group ($\Rightarrow \lceil \frac{n}{5} \rceil$ selected elements)
  – Use SELECT recursively to find median $x$ of selected elements
  – Partition all elements using $x$ as pivot
```



```
  – Use SELECT recursively to find $i$'th element
      * If $i == k$ then return $x$
      * If $i \leq k$ then use SELECT($i$) on the $k$ elements before the pivot
      * If $i > k$ then use SELECT($i-k$) on the $n-k$ elements after the pivot
```

- ANALYSIS: How long does this take, in the worst case? If n' is the maximal number of elements we recurse on in the last step of the algorithm, the running time is given by $T(n) = \Theta(n) + T(\lceil \frac{n}{5} \rceil) + \Theta(n) + T(n')$
- 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 $T(n) = \Theta(n) + T(\lceil\frac{n}{5}\rceil) + T(\frac{7}{10}n + 6)$
- It can be shown that the solution to $T(n) = n + T(\lceil\frac{n}{5}\rceil) + T(\frac{7}{10}n + 6)$ is $T(n) = O(n)$.
- Comments:
  - It is important that we chose every 5'th element, not all other choices will work.
  - This algorithm gives $\sim 16n$ comparisons. Best know $\sim 2.95n$. Best lower bound $> 2n$.

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-QUICKSOT remains the fastest sort in practice.