Linear-Time Selection (page 1)

• Problem statement:
  – Given an array $A$ of $n$ numbers $A[1..n]$ and an integer $i$ ($1 \leq i \leq n$), find the $i^{th}$ smallest number in $A$.
  – Definition: The median of $A$ is the $\left\lfloor n/2 \right\rfloor^{th}$ element in $A$.
    • Example: If $A = (7, 4, 8, 2, 4)$; then $|A| = 5$ and the 3rd smallest element (and median) is 4.

– Trivial solutions for our problem:
  • General approach: Sort the array and find the $i^{th}$ element.
    – Execution time: $\Theta(n \log n)$
  • If $i$ is a small constant (or $n$ minus a small constant) we can easily design a linear-time scanning algorithm (similar to finding a minimum or maximum in the array).
• Strategy: Partition-based (divide and conquer) selection
  – Choose one element $p$ from array $A$ (pivot element)
  – Split input into three sets:
    • LESS: elements from $A$ that are smaller than $p$
    • EQUAL: elements from $A$ that are equal to $p$
    • MORE: elements from $A$ that are greater than $p$
  – We then have three cases:
    • $i \leq |LESS|$: implies the element we are looking for is also the $i^{th}$ smallest number in LESS,
    • $|LESS| < i \leq |LESS| + |EQUAL|$: implies the element we are looking for is $p$,
    • $|LESS| + |EQUAL| < i$: implies the element we are looking for is also the $(i - |LESS| - |EQUAL|)^{th}$ smallest element in MORE.
function SELECT(A, i)
// find i-th element in array A
p := choose_pivot(A);
// partition A into LESS, EQUAL, MORE
create new arrays LESS, EQUAL, MORE;
for i := 1 to size(A) do
    if A[i] < p then add A[i] to LESS;
    if A[i] = p then add A[i] to EQUAL;
    if A[i] > p then add A[i] to MORE;
// decide which case to pursue
if(size(LESS) >= i) then
    return SELECT(LESS, i);
else if(size(LESS) + size(EQUAL) >= i) then
    return p;  // No recursive call
else
    return SELECT(MORE, i - size(LESS) - size(EQUAL));
Linear-Time Selection (page 4)

- Choice of pivot:
  - Option 1: Choose an arbitrary element (for example, the first element).
    - If we were given a sorted array and we are looking for the \( n^{th} \) smallest element, then execution time would be \( \Omega(n^2) \).
  - Option 2: Choose a random element.
    - This is better and gives us a “randomized” algorithm.
    - It can be shown that in this case the expected running time is \( \Theta(n) \), while worst-case running time is still \( \Theta(n^2) \).
      - See [CLRS, 9.2], if interested.
Another choice of pivot:

- Option 3: Use “grouping by fives” to select the pivot:
  1. Split the array $A[1..n]$ into $n/5$ groups each with 5 elements.
  2. From each group select the third smallest element (i.e., take median of each of the groups).
     - Denote the set of these elements as MEDIANS.
  3. Recursively call SELECT to obtain the median of MEDIANS.
     - (i.e.: the $\lceil n/2 \rceil^{th}$ smallest element of MEDIANS).
  4. Take the resulting element as pivot $p$. 
• Lemma:
  – At least 1/4 of the elements in $A$ are smaller than or equal to $p$ (so $|\text{MORE}| \leq 3n/4$).

• Proof:
  – Imagine sorting elements in each of the groups from smallest to largest and ordering groups by their median.
    • This is not done by the algorithm.
  – Let us represent the whole set $A$ by a table where each group is depicted as a single column and the columns are ordered by their medians.
  – Then the following figure demonstrates the claim (for the case where $n$ is a multiple of 5):
Linear-Time Selection (page 7)

At least half the columns:

More than half the rows:

○ is $p = \text{median of medians}$

The arrows indicate that an element is greater than another element.
• Similar lemma:
  – At least 1/4 of the elements in $A$ are greater than or equal to $p$ (so $|\text{LESS}| \leq 3n/4$).

• In summary:
  – If we use $p$ as pivot in SELECT, then arrays LESS and MORE each have at most $3n/4$ elements.
Linear-Time Selection (page 9)

• Run time analysis:
  – The running time $T(n)$ of the SELECT algorithm with the “group by fives” selection of pivot can be derived as follows:
    – Divide phase: $\Theta(n)$
    – Conquer phase:
      • To select “median of medians” we need time: $T(\lceil n/5 \rceil)$.
      • To run selection on one of the arrays: LESS or MORE, we need time: $\leq T(\lfloor 3n/4 \rfloor)$.
    – Combine phase: There is no combine work.
  – Thus we have: $T(n) = T(\lceil n/5 \rceil) + T(\lfloor 3n/4 \rfloor) + \Theta(n)$, with $T(1) \in \Theta(1)$. 
The running time of the SELECT algorithm is $O(n)$ because: $T(n) \leq cn$ (constant $c$ to be determined later).

Proof is by induction on $n$ (substitution method):

- Base case:
  - For $n < 40$ the claim clearly holds as long as $c$ is large enough.

- Induction step:
  - Assume that $n \geq 40$ and that for all $n_0 < n$, $T(n_0) \leq cn_0$. Then:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{3n}{4} \right\rceil\right) + kn$$

$$\leq c\left(\frac{n}{5} + 1\right) + \frac{3cn}{4} + kn \leq \frac{cn}{5} + \frac{cn}{40} + \frac{3cn}{4} + kn$$

$$\leq \frac{39cn}{40} + kn \leq cn \quad \text{as long as } k \leq c/40, \quad \text{i.e. } c \geq 40k.$$
Quick Sort Revisited

• Recall QuickSort:
  1. Select pivot element \( p \).
  2. Split the array into two parts: elements smaller than \( p \) and elements larger than \( p \).
  3. These can be sorted separately.
     – Hopefully, whenever we split the array, we get sub-arrays of approximately the same size to achieve \( \Theta(n \log n) \) running time.
       • However, if we are unlucky, we get \( \Omega(n^2) \) running time.
     – Idea: What if we use our SELECT algorithm to select pivot \( p \) to be the median?
       • Then every time we split, we guarantee “almost equal” splits thus having \( \Theta(n \log n) \) worst-case running time.
       • This is quite slow in practice (the constant associated with running SELECT is too large).
       • It is better to just select a random element (it can be proved that then we get \( \Theta(n \log n) \) expected running time).
Making Divide and Conquer Faster

• In practice:
  – Divide and conquer algorithms have often large multiplicative and additive constant overhead (for recursion, etc.) which makes them slower for small size data sets than the trivial algorithm.
  – Running time of a divide and conquer algorithm can be reduced if we solve small subproblems by some trivial algorithm (instead of dividing them further).
  – When to “divide" and when to use the trivial algorithm needs to be determined empirically.
An Example

function SELECT(A,i)
* if size(A)<100 then
* sort elements of A;
* return A[i];
else
// find i-th element in array A:
(The same code we studied earlier).