Median Calculation

Just like Linear-Time Sorting

Based on slides by *David Luebke*

Order Statistics

- The *i*th *order statistic* in a set of *n* elements is the *i*th smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the *n*th order statistic
- The *median* is the *n*/2 order statistic
  - If *n* is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes:
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
  - Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the “pivot” element separating the two subarrays
- *How do you suppose we implement this?*

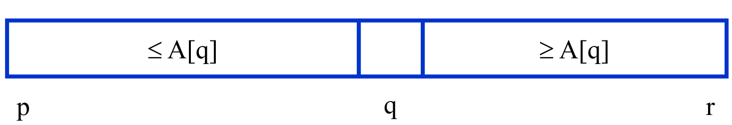
Partition In Words

- Partition(A, p, r):
  - Select an element to act as the “pivot” (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Randomized Order Stat

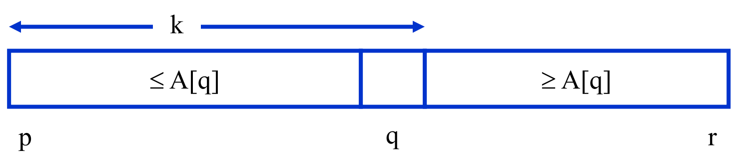
- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will use the randomized partition:

  q = RandomizedPartition(A, p, r)



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Randomized Selection

- Analyzing **`RandomizedSelect()`**
  - Worst case: partition always 0:n-1

    T(n) $= T(n-1) + O(n)$ = ???

    $= O(n^2)$ (arithmetic series)

    - No better than sorting!
  - "Best" case: suppose a 9:1 partition

    T(n) $= T(9n/10) + O(n)$ = ???

    $= O(n)$ (Master Theorem, case 3)

    - Better than sorting!

Randomized Selection

- Average case
  - For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

What happened here?

  - Let's show that T(n) = O(n) by substitution

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$ ***The recurrence we started with***

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$ ***Substitute T(n) ≤ cn for T(k)***

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$ ***"Split" the recurrence***

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$ ***Expand arithmetic series***

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ ***Multiply it out***

Randomized Selection

- Assume $T(n) \le cn$ for sufficiently large c:

$$
\begin{aligned}
T(n) &\le c(n-1) - \frac{c}{2}\left(\frac{n}{2} - 1\right) + \Theta(n) && \textit{The recurrence so far} \\
&= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n) && \textit{Multiply it out} \\
&= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n) && \textit{Subtract c/2} \\
&= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right) && \textit{Rearrange the arithmetic} \\
&\le cn \quad \text{(if c is big enough)} && \textit{What we set out to prove}
\end{aligned}
$$

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element *x*

Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide n elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median x of the $\lfloor n/5 \rfloor$ medians
  4. Partition the n elements around x. Let $k = \text{rank}(x)$
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find ith smallest element in first partition

     else (i > k) use Select() recursively to find (i-k)th smallest element in last partition

Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are $\leq x$?*
  - At least 1/2 of the medians $= \lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are $\leq x$?*
  - At least $3 \lfloor n/10 \rfloor$ elements
- For large n, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements $\leq x$
- Similarly: at least $n/4$ elements $\geq x$

Worst-Case Linear-Time Selection

- Thus after partitioning around x, step 5 will call Select() on at most $3n/4$ elements
- The recurrence is therefore:

$$
\begin{aligned}
T(n) &\le T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) \\
&\le T(n/5) + T(3n/4) + \Theta(n) && \lfloor n/5 \rfloor \le n/5 \\
&\le cn/5 + 3cn/4 + \Theta(n) && \textit{Substitute } T(n) = cn \\
&= 19cn/20 + \Theta(n) && \textit{Combine fractions} \\
&= cn - (cn/20 - \Theta(n)) && \textit{Express in desired form} \\
&\le cn \quad \text{if } c \text{ is big enough} && \textit{What we set out to prove}
\end{aligned}
$$

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

Linear-Time Median Selection

- Given a "black box" O(n) median algorithm, what can we do?
  - *i*th order statistic:
    - Find median *x*
    - Partition input around *x*
    - if ($i \leq (n+1)/2$) recursively find *i*th element of first half
    - else find ($i - (n+1)/2$)th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$
  - *Can you think of an application to sorting?*

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - $T(n) = 2T(n/2) + O(n) = O(n \lg n)$