Design and Analysis of Algorithms

CSE 5311
Lecture 9  Median and Order Statistics

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Medians and Order Statistics

• The $i$th order statistic of $n$ elements $S = \{a_1, a_2, \ldots, a_n\}$: $i$th smallest elements
• Also called selection problem
• Minimum and maximum
• Median, lower median, upper median
• Selection in expected/average linear time
• Selection in worst-case linear time
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?**
  - To compute the maximum, \( n - 1 \) comparisons are necessary and sufficient.
  - The algorithm is optimal with respect to the number of comparisons performed.
  - The same is true for the minimum.

---

**MINIMUM**  
\[
\text{MINIMUM}(A, n) \quad \text{min} \leftarrow A[1] \\
\text{for } i \leftarrow 2 \text{ to } n \\
\text{do if } \text{min} > A[i] \\
\text{then } \text{min} \leftarrow A[i] \\
\text{return } \text{min}
\]

**Can we find the minimum and maximum with less cost?**

Yes:
  - Walk through elements by pairs
  - Compare each element in pair to the other
  - Compare the largest to maximum, smallest to minimum
Order Statistics

• **Simultaneous computation of max and min**
  
  – Maintain the variables min and max. Process the n numbers in pairs.
  
  – For the first pair, set min to the smaller and max to the other. After that, for each new pair, compare the smaller with min and the larger with max.
  
  – Can be done in $3(n-3)/2$ steps

**MAX-AND-MIN(A, n)**

```plaintext
1: max ← A[n]; min ← A[n]
2: for i ← 1 to n/2 do
4:     { if A[2i − 1] > max then
5:       max ← A[2i − 1]
6:     if A[2i] < min then
7:       min ← A[2i] }
8:   else { if A[2i] > max then
9:     max ← A[2i]
10:   if A[2i − 1] < min then
11:     min ← A[2i − 1] }
12: return max and min
```
Example: Simultaneous Max, Min

- \( n = 5 \) (odd), array \( A = \{2, 7, 1, 3, 4\} \)

1. Set \( \text{min} = \text{max} = 2 \)

2. Compare elements in pairs:
   - \( 1 < 7 \Rightarrow \) compare 1 with \( \text{min} \) and 7 with \( \text{max} \)
     \[ \Rightarrow \text{min} = 1, \text{max} = 7 \]
   - \( 3 < 4 \Rightarrow \) compare 3 with \( \text{min} \) and 4 with \( \text{max} \)
     \[ \Rightarrow \text{min} = 1, \text{max} = 7 \]

Total cost: \( 3(n-1)/2 = 6 \) comparisons
Example: Simultaneous Max, Min

- \( n = 6 \) (even), array \( A = \{2, 5, 3, 7, 1, 4\} \)

1. Compare 2 with 5: \( 2 < 5 \)

2. Set \( \text{min} = 2, \text{max} = 5 \)

3. Compare elements in pairs:
   - \( 3 < 7 \Rightarrow \) compare 3 with \( \text{min} \) and 7 with \( \text{max} \)
     \[ \Rightarrow \text{min} = 2, \text{max} = 7 \]
   - \( 1 < 4 \Rightarrow \) compare 1 with \( \text{min} \) and 4 with \( \text{max} \)
     \[ \Rightarrow \text{min} = 1, \text{max} = 7 \]

Total cost: \( 3n/2 - 2 = 7 \) comparisons
$O(n \lg n)$ Algorithm

- Suppose $n$ elements are sorted by an $O(n \lg n)$ algorithm, e.g., MERGE-SORT
  - Minimum: the first element; Maximum: the last element
  - The $i$th order statistic: the $i$th element.
  - Median:
    - If $n$ is odd, then $((n+1)/2)$th element.
    - If $n$ is even,
      - then $\lfloor (n+1)/2 \rfloor$th element, lower median
      - then $\lceil (n+1)/2 \rceil$th element, upper median
- All selections can be done in $O(1)$, so total: $O(n \lg n)$.
  - Selection is a trivial problem if the input numbers are sorted.
  - But using a sorting is more like using a cannon to shoot a fly since only one number needs to be computed.
- Can we do better?
Selection in Expected Linear Time $O(n)$

• Select $i$th element
• A divide-and-conquer algorithm RANDOMIZED-SELECT
• Similar to quicksort, partition the input array recursively
• Unlike quicksort, which works on both sides of the partition, just work on one side of the partition.
  – Called **prune-and-search**, prune one side, just search the other side).
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
Randomized Selection

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

- We will again use a slightly different partition than the book:
  \[ q = \text{RandomizedPartition}(A, p, r) \]
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);

\[ p \leq A[q] \leq A[q] \geq A[q] \leq r \]

Randomized Selection

• Analyzing RandomizedSelect()
  – Worst case: partition always 0:n-1
    \[ T(n) = T(n-1) + O(n) = \ldots = O(n^2) \] (arithmetic series)
    ➢ No better than sorting!
  – “Best” case: suppose a 9:1 partition
    \[ T(n) = T(9n/10) + O(n) = \ldots = O(n) \] (Master Theorem, case 3)
    ➢ Better than sorting!
    ➢ What if this had been a 99:1 split?
Randomized Selection

• Average case
  – For upper bound, assume \( i \)th element always falls in larger side of partition:

\[
T(n) \leq \frac{1}{n} \sum_{k=0}^{n-1} T(\max(k - 1, n - k)) + \Theta(n)
\]

What happened here?

\[
\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + \Theta(n)
\]

– Let’s show that \( T(n) = O(n) \) by substitution

Max(k-1, n-k)=k-1 if \( k > \lceil n/2 \rceil \)
Max(k-1, n-k)=n-k if \( k \leq \lceil n/2 \rceil \)
Randomized Selection

• Assume $T(n) \leq cn$ for sufficiently large $c$:

\[
T(n) \leq \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + \Theta(n)
\]

The recurrence we started with

\[
\leq \frac{2}{n} \sum_{k=n/2}^{n-1} c k + \Theta(n)
\]

Substitute $T(n) \leq cn$ 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

\[
= c(n - 1) \frac{n}{2} - \frac{c}{2} \left( \frac{n}{2} - 1 \right) + \Theta(n)
\]

Multiply it out
Randomized Selection

• Assume $T(n) \leq cn$ for sufficiently large $c$:

$$T(n) \leq c(n-1) - \frac{c}{2} \left( \frac{n}{2} - 1 \right) + \Theta(n)$$

The recurrence so far

$$= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$

Multiply it out

$$= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$$

Subtract $c/2$

$$= cn - \left( \frac{cn}{4} + \frac{c}{2} - \Theta(n) \right)$$

Rearrange the arithmetic

$$\leq cn \quad \text{(if $c$ is big enough)}$$

What we set out to prove
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 the elements into groups of five, where the last group may have less than five elements in case when the input array size is not a multiple of five.
  2. Find median of each group (How? How long?). Ties can be broken arbitrarily
  3. Make a recursive call `Select()` to calculate the median of the medians. Set \( x \) to the median.
  4. Partition the \( n \) elements around \( x \). Let \( k = \text{rank}(x) \)
  5. \textbf{if} (\( i == k \)) \textbf{then} return \( x \)
     \textbf{if} (\( i < k \)) \textbf{then} use `Select()` recursively to find \( i \)th smallest element in first partition
     \textbf{else} (\( i > k \)) use `Select()` recursively to find \((i-k)\)th smallest element in last partition

\( k = \text{rank}(x) \), \( x \) is the \( k \)-th smallest element and there are \( n-k \) elements on the high side of the partition
Example

- Find the \(-11\)th smallest element in array:

\[ A = \{12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47\} \]

1. Divide the array into groups of 5 elements

\[
\begin{array}{cccccc}
12 & 4 & 43 & 2 & 20 & 30 \\
34 & 17 & 82 & 19 & 33 & 3 \\
0 & 32 & 25 & 12 & 16 & 47 \\
3 & 3 & 27 & 5 & 33 & \\
22 & 28 & 34 & 18 & 21 & \\
\end{array}
\]
Example

2. Sort the groups and find their medians

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>25</th>
<th>2</th>
<th>20</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>34</td>
<td>12</td>
<td>21</td>
<td>47</td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>43</td>
<td>19</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>82</td>
<td>18</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

3. Find the median of the medians

12, 12, 17, 21, 34, 30
Example

4. Partition the array around the median of medians (17)

First partition:

\{12, 0, 3, 4, 3, 2, 12, 5, 16, 3\}

Pivot:

17 (position of the pivot is \(q = 11\))

Second partition:

\{34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47\}

To find the 6-th smallest element we would have to recurse our search in the first partition.
Analysis of Running Time

- Step 1: making groups of 5 elements takes $O(n)$

- Step 2: sorting $n/5$ groups in $O(1)$ time each takes $O(n)$

- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time $T(\lceil n/5 \rceil)$

- Step 4: partitioning the n-element array around x takes $O(n)$

- Step 5: recursion on one partition takes

  depends on the size of the partition!!
Worst-Case Linear-Time Selection

• (Sketch situation on the board)
• **How many of the 5-element medians are \( \leq x \)?**
  – At least \( \frac{1}{2} \) of the medians = \( \lfloor n/5 \rfloor / 2 = \lfloor n/10 \rfloor \)
• **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:

$$T(n) \leq T\left(\lfloor n/5 \rfloor\right) + T(3n/4) + \Theta(n)$$

$$\leq T(n/5) + T(3n/4) + \Theta(n)$$

$$\leq cn/5 + 3cn/4 + \Theta(n)$$

Substitute $T(n) = cn$

$$\leq 19cn/20 + \Theta(n)$$

Combine fractions

$$= cn - (cn/20 - \Theta(n))$$

Express in desired form

$$\leq cn$$ if $c$ is big enough

What we set out to prove
### 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
  - \( T(n) = 2T(n/2) + O(n) = O(n \lg n) \)
Summary

• The \( i \)th order statistic of \( n \) elements \( S=\{a_1, a_2, \ldots, a_n\} \): \( i \)th smallest elements:
  – Minimum and maximum.
  – Median, lower median, upper median

• Selection in expected/average linear time
  – Worst case running time
  – Prune-and-search

• Selection in worst-case linear time: