Medians & Selection

CS16: Introduction to Data Structures & Algorithms
Seny Kamara - Spring 2017
Outline

- Medians
- Selection
- Randomized Selection
- Median-of-Medians Selection
Medians

- The median of a collection of numbers
  - is the middle element
  - half of the numbers smaller and half larger
- Used to summarize the collection
- The mean or average can also be used…
  - …but averages can be corrupted by outliers
- What are the mean & median of
  - \[9, 5, 4, 6, 5, 7, 10000, 6, 4, 8\]
    - mean 1005.4 & median 6
- Finding median is easy: sort list and pick middle element
  - \(O(n \log n)\)...can we do better?
Medians

- If we can find median quickly we could use it in Quicksort
- Recall that Quicksort picks pivot at random
  - if pivot is max or min element every time
  - runtime is $O(n^2)$
- If we could find pivot in $O(n)$
  - Quicksort would have $O(n \log n)$ worst-case runtime
  - Same as Merge Sort
Selection

- Let's consider a more general problem
- Selection
  - given a list \( L \) and an integer \( k \)
  - output the \( k \)th smallest element in list
- Median with selection
  - Selection with \( k = n/2 \)
Selection

- Divide and conquer
  - divide: pick random element (called pivot) and partition set into
    - \( L \): elements less than \( x \)
    - \( E \): elements equal to \( x \)
    - \( G \): elements larger than \( x \)
  - recur:
    - if \( k \leq |L| \): call select\( (L,k) \)
    - if \( |L| < k \leq |L| + |E| \): return \( x \)
    - if \( k > |L| + |E| \): call select\( (G, k - (|L| + |E|)) \)
  - conquer: return
Hoare’s Selection Pseudo-Code

```python
select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []     E = []     G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size)
        return pivot
    else
        return select(G, k - (L.size + E.size))
```
Selection

```python
select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []   E = []   G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size):
        return pivot
    else
        return select(G, k - (L.size + E.size))
```

Activity #1+2
Selection

select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []   E = []   G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size)
        return pivot
    else
        return select(G, k – (L.size + E.size))
Selection

```python
select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []     E = []     G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size):
        return pivot
    else
        return select(G, k - (L.size + E.size))
```
**Selection**

```python
def select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []  E = []  G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size)
        return pivot
    else
        return select(G, k – (L.size + E.size))
```

Activity #1+2
select(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []     E = []     G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return select(L, k)
    else if k <= (L.size + E.size):
        return pivot
    else
        return select(G, k – (L.size + E.size))
Selection Analysis

- When selection uses random pivot…
  - …it is known as Hoare’s selection

- How fast is Hoare’s selection?
  - like quicksort it has $O(n^2)$ run time if we pick the wrong pivots

- We consider expected runtime since algorithm is randomized

- We assume all elements are distinct (worst-case scenario)
  - if list has more than one copy of pivot,
  - it would shrink the sub-lists and improve runtime
Selection Analysis

- Each pivot has equal probability of being chosen
- Each pivot splits list in two lists
  - one of size $i$
  - and one of size $n-1-i$
- Recurrence relation now has form
  \[
  \mathbb{E}[T(n)] = (n - 1) + \frac{1}{n} \left( \sum_{j=1}^{k-1} T(n-i) + \sum_{j=k+1}^{n} T(i-1) \right)
  \]
- which is $O(n)$
Selection in Quicksort

- Selection runtime
  - Expected $O(n)$
  - Worst-case is still $O(n^2)$
- If we use selection in Quicksort
  - we still have $O(n \log n)$ expected time
  - and $O(n^2)$ worst-case
- But we’re not done!
Outline

- Medians
- Selection
- Randomized Selection
- Median-of-Medians Selection
Median-of-Medians Select

- General Strategy
  - pick pivot that is “always good”
  - i.e., between 25th and 75th percentile
- Do this by picking the median of medians (mom)
  - partition list into $\frac{n}{5}$ lists of size 5
  - sort each list in $O(1)$ time
  - choose the median of each list
  - call mom-select recursively on list of $\frac{n}{5}$ medians
- Use the mom as pivot and continue w/ selection algorithm
Median-of-Medians Select

```
momSelect(list, k)
    if list.size = 5:
        sort5(list)    // in O(1) b/c list always size 5

    miniLists = divide list into n/5 lists of size 5
    medians = []
    for miniList in miniLists:
        medians.append(momSelect(miniList, 3))
    pivot = momSelect(medians, medians.size/2)

    L = []    E = []    G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: R.append(x)
    if k <= L.size:
        return momSelect(L, k)
    else if k <= (L.size + E.size)
        return pivot
    else
        return momSelect(G, k – (L.size + E.size))
```
Median-of-Medians Select

- Sorting a list of numbers from 1 to 50

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>7</td>
<td>15</td>
<td>22</td>
<td>17</td>
<td>25</td>
<td>20</td>
<td>6</td>
<td>34</td>
</tr>
<tr>
<td>26</td>
<td>37</td>
<td>35</td>
<td>32</td>
<td>29</td>
<td>30</td>
<td>33</td>
<td>40</td>
<td>45</td>
<td>47</td>
</tr>
<tr>
<td>27</td>
<td>50</td>
<td>42</td>
<td>49</td>
<td>44</td>
<td>38</td>
<td>39</td>
<td>41</td>
<td>46</td>
<td>48</td>
</tr>
</tbody>
</table>

- Guaranteed to be larger than red numbers and larger than blue numbers which is between 25th and 75th percentiles. **Great pivot choice!**

- How many elements will pivot eliminate? What is area of blue or red region?
  - **height:** 3
  - **width:** \((\frac{n}{5})/2\)
  - **area:** \(3n/10\)
  - leaves problem of size at most \(7n/10\)
Median-of-Medians Select

Activity #3

3 min
Median-of-Medians Select

Activity #3

3 min
Median-of-Medians Select

Activity #3

2 min
Median-of-Medians Select

1 min

Activity #3
Median-of-Medians Select

Activity #3

0 min
Median-of-Medians Select

- With median-of-medians as pivot...
- ...the select recurrence relation is:

\[ T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n) \]

From recursing on list of \( n/5 \) medians to find median of median
From recursing on list guaranteed to be at most \( 7/10 \) size of original

- This is \( O(n) \) (see Epstein for proof)
Summary

- We can perform select in worst-case $O(n)$
- …which means we can find medians in worst-case $O(n)$
- …which means we can perform Quicksort in worst-case $O(n \log n)$
- In practice, Quicksort w/ random pivots is faster
- Quicksort w/ median of medians is useful when worst-case performance is crucial
- We don’t expect you to implement median of medians when implementing Quicksort
Readings

- Dasgupta et al.
  - Section 2.4: analysis of median finding algorithms
- Wocjan’s analysis of Selection w/ random pivot
  - http://www.eecs.ucf.edu/courses/cot5405/fall2010/chapter1_2/QuickSelAvgCase.pdf
- Proof that median of median is $O(n)$ (Epstein reading)