Today: Sorting, continued

- Quicksort
- Selection
- Distribution counting
- Radix sorts

Next topic readings: *Data Structures*, Chapter 9.

Public Service Announcement: Residential Computing is hiring. Be an RCC and get paid for your computer skills. Flexible Hours, Work Study: $11.97/hour Past RCC's have gone on to places like Google, Apple, Microsoft and eBay. For more information visit:

http://rescomp.berkeley.edu/rcchiring.
Quicksort: Speed through Probability

Idea:

- *Partition* data into pieces: everything $> a$ *pivot* value at the high end of the sequence to be sorted, and everything $\leq$ on the low end.
- Repeat recursively on the high and low pieces.
- For speed, stop when pieces are “small enough” and do insertion sort on the whole thing.
- Reason: insertion sort has low constant factors. By design, no item will move out of its piece [why?], so when pieces are small, #inversions is, too.
- Have to choose pivot well. E.g.: *median* of first, last and middle items of sequence.
Example of Quicksort

• In this example, we continue until pieces are size \( \leq 4 \).

• Pivots for next step are starred. Arrange to move pivot to dividing line each time.

• Last step is insertion sort.

\[
\begin{align*}
16 & \, 10 & \, 13 & \, 18 & \, -4 & \, -7 & \, 12 & \, -5 & \, 19 & \, 15 & \, 0 & \, 22 & \, 29 & \, 34 & \, 1^* \\
-4 & \, -5 & \, -7 & \, -1 & \, 18 & \, 13 & \, 12 & \, 10 & \, 19 & \, 15 & \, 0 & \, 22 & \, 29 & \, 34 & \, 16^* \\
-4 & \, -5 & \, -7 & \, -1 & \, 15 & \, 13 & \, 12^* & \, 10 & \, 0 & \, 16 & \, 19 & \, 22 & \, 29 & \, 34 & \, 18 \\
-4 & \, -5 & \, -7 & \, -1 & \, 10 & \, 0 & \, 12 & \, 15 & \, 13 & \, 16 & \, 18 & \, 19 & \, 29 & \, 34 & \, 22
\end{align*}
\]

• Now everything is “close to” right, so just do insertion sort:

\[
\begin{align*}
-7 & \, -5 & \, -4 & \, -1 & \, 0 & \, 10 & \, 12 & \, 13 & \, 15 & \, 16 & \, 18 & \, 19 & \, 22 & \, 29 & \, 34
\end{align*}
\]
Performance of Quicksort

- Probabalistic time:
  - If choice of pivots good, divide data in two each time: $\Theta(N \lg N)$
    with a good constant factor relative to merge or heap sort.
  - If choice of pivots bad, most items on one side each time: $\Theta(N^2)$.
  - $\Omega(N \lg N)$ in best case, so insertion sort better for nearly ordered input sets.

- Interesting point: randomly shuffling the data before sorting makes $\Omega(N^2)$ time very unlikely!
Quick Selection

The Selection Problem: for given $k$, find $k^{th}$ smallest element in data.

- Obvious method: sort, select element $#k$, time $\Theta(N \lg N)$.
- If $k \leq$ some constant, can easily do in $\Theta(N)$ time:
  - Go through array, keep smallest $k$ items.
- Get probably $\Theta(N)$ time for all $k$ by adapting quicksort:
  - Partition around some pivot, $p$, as in quicksort, arrange that pivot ends up at dividing line.
  - Suppose that in the result, pivot is at index $m$, all elements $\leq$ pivot have indicies $\leq m$.
  - If $m = k$, you’re done: $p$ is answer.
  - If $m > k$, recursively select $k^{th}$ from left half of sequence.
  - If $m < k$, recursively select $(k - m - 1)^{th}$ from right half of sequence.
Selection Example

Problem: Find just item #10 in the sorted version of array:

Initial contents:

<table>
<thead>
<tr>
<th>51</th>
<th>60</th>
<th>21</th>
<th>-4</th>
<th>37</th>
<th>4</th>
<th>49</th>
<th>10</th>
<th>40</th>
<th>59</th>
<th>0</th>
<th>13</th>
<th>2</th>
<th>39</th>
<th>11</th>
<th>46</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looking for #10 to left of pivot 40:

<table>
<thead>
<tr>
<th>13</th>
<th>31</th>
<th>21</th>
<th>-4</th>
<th>37</th>
<th>4*</th>
<th>11</th>
<th>10</th>
<th>39</th>
<th>2</th>
<th>0</th>
<th>40</th>
<th>59</th>
<th>51</th>
<th>49</th>
<th>46</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looking for #6 to right of pivot 4:

| -4 | 0 | 2 | 4 | 37 | 13 | 11 | 10 | 39 | 21 | 3* | 40 | 59 | 51 | 49 | 46 | 60 |
|----|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 4  |

Looking for #1 to right of pivot 31:

| -4 | 0 | 2 | 4 | 21 | 13 | 11 | 10 | 31 | 39 | 37 | 40 | 59 | 51 | 49 | 46 | 60 |
|----|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 9  |

Just two elements; just sort and return #1:

| -4 | 0 | 2 | 4 | 21 | 13 | 11 | 10 | 31 | 37 | 39 | 40 | 59 | 51 | 49 | 46 | 60 |
|----|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 9  |

Result: 39
Selection Performance

- For this algorithm, if \( m \) roughly in middle each time, cost is

\[
C(N) = \begin{cases} 
1, & \text{if } N = 1, \\
N + C(N/2), & \text{otherwise.}
\end{cases}
\]

\[
= N + N/2 + \ldots + 1 \\
= 2N - 1 \in \Theta(N)
\]

- But in worst case, get \( \Theta(N^2) \), as for quicksort.

- By another, non-obvious algorithm, can get \( \Theta(N) \) worst-case time for all \( k \) (take CS170).
Better than $N \lg N$?

- Can prove that *if all you can do to keys is compare them* then sorting must take $\Omega(N \lg N)$.
- Basic idea: there are $N!$ possible ways the input data could be scrambled.
- Therefore, your program must be prepared to do $N!$ different combinations of move operations.
- Therefore, there must be $N!$ possible combinations of outcomes of all the if tests in your program (we’re assuming that comparisons are 2-way).
- Since each if test goes two ways, number of possible different outcomes for $k$ if tests is $2^k$.
- Thus, need enough tests so that $2^k > N!$, which means $k \in \Omega(\lg N!)$.
- Using Stirling’s approximation, 
  \[ m! \in \sqrt{2\pi m} \left(\frac{m}{e}\right)^m \left(1 + \Theta\left(\frac{1}{m}\right)\right), \]
  this tells us that 
  \[ k \in \Omega(N \lg N). \]
Beyond Comparison: Distribution Counting

- But suppose can do more than compare keys?
- For example, how can we sort a set of \( N \) integer keys whose values range from 0 to \( kN \), for some small constant \( k \)?
- One technique: count the number of items < 1, < 2, etc.
- If \( M_p = \#\text{items with value} < p \), then in sorted order, the \( j^{\text{th}} \) item with value \( p \) must be \( \#M_p + j \).
- Gives linear-time algorithm.
Distribution Counting Example

- Suppose all items are between 0 and 9 as in this example:

```
  7 0 4 0 9 1 9 1 9 5 3 7 3 1 6 7 4 2 0
```

```
  0 1 2 3 4 5 6 7 8 9
```

```
  0 3 6 7 9 11 12 13 16 16
```

```
  0 0 0 1 1 1 2 3 3 4 4 5 6 7 7 7 9 9 9
```

- “Counts” line gives # occurrences of each key.
- “Running sum” gives cumulative count of keys ≤ each value...
- ...which tells us where to put each key:
- The first instance of key \( k \) goes into slot \( m \), where \( m \) is the number of key instances that are \( < k \).
Radix Sort

Idea: Sort keys one character at a time.

- Can use distribution counting for each digit.
- Can work either right to left (LSD radix sort) or left to right (MSD radix sort).
- LSD radix sort is venerable: used for punched cards.

Initial: set, cat, cad, con, bat, can, be, let, bet

Pass 1 (by char #2)

```
be  cad  can  set
\_'  'd'  'n'  't'
```

be, cad, con, can, set, cat, bat, let, bet

Pass 2 (by char #1)

```
bet  bat  let
con  can  cat
\_'  'a'  'e'  'o'
```

Pass 3 (by char #0)

```
bet  bat  can
con  cad  let  set
\_'  'b'  'c'  'l'  's'
```

bat, be, bet, cad, can, cat, con, let, set
**MSD Radix Sort**

- A bit more complicated: must keep lists from each step separate
- But, can stop processing 1-element lists

<table>
<thead>
<tr>
<th>$A$</th>
<th>posn</th>
</tr>
</thead>
<tbody>
<tr>
<td>set, cat, cad, con, bat, can, be, let, bet</td>
<td>0</td>
</tr>
<tr>
<td>bat, be, bet / cat, cad, con, can / let / set</td>
<td>1</td>
</tr>
<tr>
<td>bat / be, bet / cat, cad, con, can / let / set</td>
<td>2</td>
</tr>
<tr>
<td>bat / be / bet / cat, cad, con, can / let / set</td>
<td>1</td>
</tr>
<tr>
<td>bat / be / bet / cat, cad, can / con / let / set</td>
<td>2</td>
</tr>
<tr>
<td>bat / be / bet / cad / can / cat / con / let / set</td>
<td>1</td>
</tr>
</tbody>
</table>
Performance of Radix Sort

- Radix sort takes $\Theta(B)$ time where $B$ is total size of the key data.
- Have measured other sorts as function of #records.
- How to compare?
- To have $N$ different records, must have keys at least $\Theta(lg N)$ long [why?]
- Furthermore, comparison actually takes time $\Theta(K)$ where $K$ is size of key in worst case [why?]
- So $N \lg N$ comparisons really means $N(lg N)^2$ operations.
- While radix sort takes $B = N \lg N$ time.
- On the other hand, must work to get good constant factors with radix sort.
And Don't Forget Search Trees

Idea: A search tree is in sorted order, when read in inorder.

• Need balance to really use for sorting [next topic].

• Given balance, same performance as heapsort: \( N \) insertions in time \( \lg N \) each, plus \( \Theta(N) \) to traverse, gives

\[
\Theta(N + N \lg N) = \Theta(N \lg N)
\]
Summary

- Insertion sort: $\Theta(Nk)$ comparisons and moves, where $k$ is maximum amount data is displaced from final position.
  
  - Good for small datasets or almost ordered data sets.

- Quicksort: $\Theta(N \lg N)$ with good constant factor if data is not pathological. Worst case $O(N^2)$.

- Merge sort: $\Theta(N \lg N)$ guaranteed. Good for external sorting.

- Heapsort, treesort with guaranteed balance: $\Theta(N \lg N)$ guaranteed.

- Radix sort, distribution sort: $\Theta(B)$ (number of bytes). Also good for external sorting.