CMPSC250
Lecture 6: Maximum Sort

Prof. John Wenskovitch
01/29/2015
Last Time

• Shell sort
• Mergesort
  – Top-Down Mergesort
  – Bottom-Up Mergesort
• Recurrence relations to calculate sort runtime for Mergesort
# Sorting Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Compares</th>
<th>Exchanges/Array Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Sort (Worst)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Average)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Best)</td>
<td>$O(n)$</td>
<td>0</td>
</tr>
<tr>
<td>Shell Sort (Worst)</td>
<td>$O(n^{3/2})$</td>
<td>$O(n^{3/2})$</td>
</tr>
<tr>
<td>Shell Sort (Average)</td>
<td>It depends</td>
<td>It depends</td>
</tr>
<tr>
<td>Top-Down Mergesort (Worst)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Top-Down Mergesort (Average)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Bottom-Up Mergesort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
</tbody>
</table>
Compare-Based Algorithms

• Sorting is a compare-based algorithm.
  – Makes decisions about items only on the basis of comparing keys.
  – Can do an arbitrary amount of calculation between compares, but cannot get any information about a key except by comparing it with another one.

• Theorem: No compare-based sorting algorithm can guarantee to sort N items with fewer than $O(n \times \log(n))$ compares.
Theorem: No compare-based sorting algorithm can guarantee to sort N items with fewer than $O(n \times \log(n))$ compares.

Proof: We will use a binary tree to describe the sequence of compares necessary to sort an item.

- Each node in the tree is either a leaf ($i_0.. i_{n-1}$) that indicates that a solution has been found and the inputs should be ordered as $a[i_0].. a[i_{n-1}]$, ...
- ... or the node could be an internal node ($i: j$) that corresponds to a compare operations between $a[i]$ and $a[j]$.
- Each path from the root to a leaf corresponds to the sequence of compares that the algorithm uses to establish the ordering given in the leaf.
Maximum Sort
Maximum Sort

• Observations:
  – The tree must have at least $n!$ leaves, because there are $n!$ different permutations of $n$ distinct keys.
  – The length of the longest path in the tree (the tree height) since the worst-case number of compares used by the algorithm.
  – We know that a binary tree of height $h$ has no more than $2^h$ leaves.
  – Combining these facts, we know that any compare-based algorithm corresponds to a compare tree of height $h$ with $n! \leq \text{number of leaves} \leq 2^h$.
  – The value of $h$ is the worst-case number of compares, so $\log(n!) \leq \text{number of compares} \leq \log(2^h)$. 
Maximum Sort

• So then, what is \( \log(n!) \)?
  
  \[
  \log(n!) = \log(1 \times 2 \times 3 \times 4 \times \cdots \times n)
  \]
  
  \[
  \log(n!) = \log(1) + \log(2) + \log(3) + \cdots + \log(n)
  \]
  
  \[
  \log(n!) = n \times \log(\text{anything}) = n \times \log(n) \]

\[\blacksquare\]
Conclusions

• No sorting algorithm can guarantee to use fewer that $O(n \times \log(n))$ compares on all inputs.

  – **Corollary:** Mergesort is an *asymptotically optimal* compare based sorting algorithm.

  • “Both the number of compares used by Mergesort in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are $O(n \times \log(n))$.”
Conclusions

• So... are we done with sorting algorithms, now that we found the best we can do?
  – Mergesort is not optimal with respect to space usage.
  – The worst case may not be likely in practice.
  – Operations other than compare may be more important in practice.
  – Can we sort data with using any compares?
public static void sort(Comparable a[]) {
    StdRandom.shuffle(a);
    sort(a, 0, a.length-1);
} //sort

private static void sort(Comparable a[], int lo, int hi) {
    if (hi <= lo) {
        return;
    } //if
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
} //sort
Quicksort

private static int partition(Comparable a[], int lo, int hi) {
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true) {
        while (less(a[++i],v)) {
            if (i == hi) {
                break;
            } //if
        } //while
        while (less(v, a[--j])) {
            if (j == lo) {
                break;
            } //if
        } //while
        if (i >= j) {
            break;
        } //if
        exch(a, i, j);
    } //while
    exch(a, lo, j);
    return j;
} //sort
Quicksort Partition Visual

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial values</td>
<td>0</td>
<td>16</td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>scan left, scan right</td>
<td>1</td>
<td>12</td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>exchange</td>
<td>1</td>
<td>12</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>scan left, scan right</td>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>exchange</td>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>scan left, scan right</td>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>exchange</td>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>scan left, scan right</td>
<td>6</td>
<td>5</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>final exchange</td>
<td>6</td>
<td>5</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>result</td>
<td>5</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
</tbody>
</table>
# Quicksort Sort Visual

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
</tbody>
</table>

**Initial values**

- **random shuffle**

| 0 | 5 | 15 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
|---|---|----|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | 3 | 4  | E | C | A | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 0 | 2 | 2  | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 0 | 0 | 1  | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 1 | 1  | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 4 | 4  | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 6 | 6  | 15 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| 7 | 9  | 15 | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| 7 | 7  | 8  | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| 8 | 8  | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| 10 | 13 | 15 | A | C | E | E | I | K | L | M | O | P | S | Q | R | T | U | X |
| 10 | 12 | 12 | A | C | E | E | I | K | L | M | O | P | P | Q | R | S | T | U | X |
| 10 | 11 | 11 | A | C | E | E | I | K | L | M | O | P | P | Q | R | S | T | U | X |
| 10 | 10 | 10 | A | C | E | E | I | K | L | M | O | P | P | Q | R | S | T | U | X |
| 14 | 14 | 15 | A | C | E | E | I | K | L | M | O | P | P | Q | R | S | T | U | X |
| 15 | 15 | A | C | E | E | I | K | L | M | O | P | P | Q | R | S | T | U | X |

**Result**

- **no partition for subarrays of size 1**

- **final order**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>
Quicksort vs. Mergesort

• In Mergesort, we always split the array in half (as best we could). In Quicksort, we split the array depending on input.
  – Makes sense that this would improve things – worry about what the input we’re sorting is rather than making it arbitrary.

• In Mergesort, we did our recursive calls before we touched the whole array in the same operation. In Quicksort, our recursive calls come after the whole array in partitioned.
  – This also seems like an improvement – instead of merging things that are far apart, let’s partially order the array first.
Why Shuffle the Input?

• Quicksort is a **randomized** algorithm.
  – After each `partition()` call, each subarray is in what is essentially a random order.
  – This random order turns out to be important in predicting the run time of Quicksort.
  – It then follows that we want to select keys randomly. We could either shuffle the array at the beginning, or we could pick a random key from the input instead of always picking the first key.
Quicksort Performance Characteristics

- Inner partition loop increments an index and compares an array entry against a fixed value. Mergesort and Shell Sort also do data movement in their inner loops.
- Quicksort doesn’t use many compares – the efficiency of the sort depends on how well the data is partitioned into subarrays, which hence depends on the choice of keys.
  - **Best case:** Each partitioning stage splits the array perfectly in half. \( C(N) = 2C(N/2) + N = O(n \times \log(n)) \)
  - **Worst case:** Each partitioning stage picks the worst possible key, so that every data item needs to be exchanged. (what’s this complexity?)
QuickSort Performance Characteristics

- Wait, so the best case of QuickSort is the average case of Mergesort. How is this better?
  - Mergesort used $n \times \log(n)$ compares and $6n \times \log(n)$ array accesses.
  - QuickSort uses $2n \times \log(n)$ compares and $\frac{1}{3}n \times \log(n)$ exchanges.
Quicksort Improvements

• Cutoff to Insertion Sort
• “Median-of-Three Partitioning” – Pick a few random items from the subarray, take the median, and use that as the pivot.
• “Entropy-Optimal Sorting” – In arrays with large numbers of duplicates, we’ll run into subarrays that don’t need to be sorted. Partition into three pieces – keys less than, keys greater than, and keys equal to the pivot.
• Quicksort is widely used today because it outperforms all other sorting algorithms in “practical applications.”
Any Questions?

http://goo.gl/forms/KrKuiDH7WT