Recursive sorting

CSC148, Introduction to Computer Science
Fall 2016
Diane Horton
Recursion isn’t always structural

• Often we recurse over a recursive structure.
  – sum over a nested list of ints
  – height of a tree
  – print all files starting “under” a given folder

• Sometimes we *build* a recursive structure as we recurse.
  – building an AbstractFileTree from a given folder

• But sometimes we do neither.
  – tiling exercise from Lab 8
  – recursive binary search
Sorting

• We can only do binary search on a *sorted* list.

• Sorting algorithms you may have seen:
  – Bubblesort: float the small items up in the list
  – Insertion sort: Insert each item into your growing sorted list
  – Selection sort: Pick the smallest and put it at front; pick second smallest, . . .

• All are iterative

• Aside: For each, the “Loop invariant” is critical.

• **Loop invariant**: a boolean condition that is true at the beginning and ending of every iteration.
Efficiency of these iterative sorts

• None of these is very fast vs other algorithms.

• Consider selection sort:
  – How many iterations of the main loop?
  – For each one of these, how many iterations?
  – Amount of work per iteration?
  – Therefore, efficiency is:

• They are all in that same big-oh complexity class.

• Can we do better?
Recursive sorting

- Mergesort

- Quicksort
Mergesort and quicksort: in common?

• What do these algorithms have in common?
  1. Split the list into two parts.
  2. Recursively sort each
  3. Combine the two parts into a single, sorted, list.

• We call this a divide and conquer approach.

• Notice that we’ve switched from talking about general ADTS to general algorithms. algorithms.

• Both are very important for a computer scientist to know.
Where does the hard work happen?

• **Mergesort:**
  – Splitting is trivial
  – Combining is some work

• **Quicksort:**
  – Splitting is some work
  – Combining is trivial
Efficiency of Mergesort
Efficiency of Mergesort

• Consider the tree of calls to mergesort.
• How many levels are in the tree?
  \[ \sim \log_2 n \]
• What is the total cost at each level?
  \[ \sim n \]
• So mergesort is \( O(n \log n) \)
  – Much, much better than \( O(n^2) \)
• The time does not depend on list contents
  – Worst-case and best-case running times are the same
Efficiency of Quicksort
Efficiency of Quicksort

• Consider the tree of calls to quicksort.
• Same analysis?
  – Yes, if we pick a pivot that divides the list into equal halves
  – That is, if we pick the median value
• So quicksort’s best case running time is $O(n \log n)$
• What’s the worst pivot we could pick?
  – The smallest (or largest) value!
• In that worst case, quicksort is $O(n^2)$
So Mergesort is better than Quicksort?

- Best-case running time is a tie:
  - Both are $O(n \log n)$

- Mergesort has better worst-case running time:
  - Mergesort: $O(n \log n)$
  - Quicksort: $O(n^2)$

- But *average* case running time is a tie:
  - Both are $O(n \log n)$
Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mergesort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>O(n^2)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
</tr>
</tbody>
</table>

**In practise, quicksort is faster on average.**

- **Reason:** Its non-recursive parts are faster than mergesort’s.
Time analysis of recursive code

• We were quite informal.
• Also, we unwound all the recursive calls.
• You don’t have to unwind it to do the analysis.
• (Just like you don’t have to unwind it to write the code.)
• In csc236, you’ll use recurrence relations to analyze the time efficiency of recursive code.