CS 171: Introduction to Computer Science II

Mergesort
Announcements/Reminders

• Hw3 grades distributed: median 93
• Quiz2 grades distributed: median 81
• Hw4 due Friday
Roadmap

• MergeSort
  – Recursive Algorithm (top-down)
  – Improvements
  – Non-recursive algorithm (bottom-up)

• Runtime analysis of recursive algorithms

• QuickSort
MergeSort

• Uses a divide and conquer approach
• Splits an input array to two sub-arrays, recursively sort each half, and merge them
MergeSort

- Recursion overhead for tiny subarrays
- Merging cost even when the array is already sorted
- Requires additional memory space for auxiliary array (quicksort later)
Mergesort: practical improvements

Use insertion sort for small subarrays.
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 7 \) items.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
    int mid = lo + (hi - lo) / 2;
    sort (a, aux, lo, mid);
    sort (a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```
Mergesort: practical improvements

Stop if already sorted.

• Is biggest item in first half \(\leq\) smallest item in second half?
• Helps for partially-ordered arrays.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```
MergeSort Improvements

• MergeX.java
• *Use insertion sort for small subarrays*
  – improve the running time by 10 to 15 percent.
• *Test whether array is already in order*
  – reduce merging cost for sorted subarrays
• *Eliminate the copy to the auxiliary array*
  – Switches the role of the input array and the auxiliary array at each level: one that sorts an input array and puts the sorted output in the auxiliary array; the other sorts the auxiliary array and puts the sorted output in the given array
Mergesort: visualization
Roadmap

• MergeSort
  – Recursive Algorithm (top-down)
  – Improvements
  – Non-recursive algorithm (bottom-up)
• Runtime analysis of recursive algorithms
• QuickSort
Recursive MergeSort (top-down)
Non-Recursive MergeSort (bottom-up)
Bottom-up mergesort

Basic plan.
- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ....

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(merge(a, 0, 0, 1))</th>
<th>(merge(a, 2, 2, 3))</th>
<th>(merge(a, 4, 4, 5))</th>
<th>(merge(a, 6, 6, 7))</th>
<th>(merge(a, 8, 8, 9))</th>
<th>(merge(a, 10, 10, 11))</th>
<th>(merge(a, 12, 12, 13))</th>
<th>(merge(a, 14, 14, 15))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
<td>MERGESAVERTEXAMPLE</td>
</tr>
<tr>
<td>2</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td>EGMRRESSORETXAMPLEL</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td>EGMRRSSAOETXELMP</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td>EEEMORRSSAOETXELMP</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td>AEEEEEGLMMMOPRRSTX</td>
<td></td>
</tr>
</tbody>
</table>

Bottom line. No recursion needed!
Bottom-up mergesort: visual trace

2

4

8

16

32
Roadmap

• MergeSort
  – Recursive Algorithm (top-down)
  – Improvements
  – Non-recursive algorithm (bottom-up)
• Runtime analysis of recursive algorithms
• QuickSort
Recursive Mergesort

```java
public class Merge {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        /* as before */
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```
Mergesort Cost

• Mergesort
  – Recursively sort 2 half-arrays
  – Merge 2 half-arrays into 1 array

• What’s the cost for merging?
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    assert isSorted(a, lo, mid); // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi); // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)
        aux[k] = a[k]; // copy

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
        { // merge
            if (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }

    assert isSorted(a, lo, hi); // postcondition: a[lo..hi] sorted
}
Mergesort Cost

• Mergesort
  – Recursively sort 2 half-arrays
  – Merge 2 half-arrays into 1 array
Cost Analysis for Recursive Algorithms

- Define the cost function $T(N)$ using recurrence relation and define base cases
- Solve the recurrence relation
- Derive Big O function
Defining Recurrence Relation

• A recurrence relation is an equation that recursively defines a sequence: each term of the sequence is defined as a function of the preceding terms

• Examples:
  \[ T(n) = T(n-1) + 1 \]
Mergesort Cost

• Mergesort
  – Recursively sort 2 half-arrays
  – Merge 2 half-arrays into 1 array (cost: N)
• Suppose cost function for sorting N items is $T(N)$
  – $T(N) = 2 \times T(N/2) + N$
Solving Recurrence Relations

• Rewrite $T(N)$, $T(N-1)$, $T(N-2)$, ..., with the recurrence formula
• Discover the patterns and find an expression
• Check the correctness
  – Substitute solution in initial conditions
  – Substitute solutions in the recurrence relation
Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming $N$ is a power of 2]

\[
\begin{align*}
D(N) &= 2D(N/2) + N \\
D(N)/N &= 2D(N/2)/N + 1 \\
&= D(N/2)/(N/2) + 1 \\
&= D(N/4)/(N/4) + 1 + 1 \\
&= D(N/8)/(N/8) + 1 + 1 + 1 \\
&\vdots \\
&= D(N/N)/(N/N) + 1 + 1 + \ldots + 1 \\
&= \lg N
\end{align*}
\]
Mergesort: empirical analysis

Running time estimates:

- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Binary Search Example

- Binary Search Cost Function
- $T(n) = T(n/2) + 1$
- $T(1) = 1$
Binary Search Example: Solution

- Binary Search Cost Function
  - $T(n) = T(n/2) + 1$
  - $T(1) = 1$

- $N = 2^k$
  - $T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 1 + 1 + ...$
  - $T(2^k) = T(2^0) + k = 1 + k$
  - $T(N) = 1 + \log N$
The Tower of Hanoi

• How many steps does the solution take to solve a N-disk problem?
• Use recursive formula
  \[ T(N) = T(N-1) + 1 + T(N-1) = 2*T(N-1) + 1 \]
• So how to solve this?
  \[ T(N) = 2^{N-1} + 2^{N-2} + 2^{N-3} +...+1 = 2^N - 1 \]
• So it takes an exponential number of steps!
When is the world going to end?

- Takes 585 billion years for $N = 64$ (at rate of 1 disc per second).