CS 171: Introduction to Computer Science II

Mergesort
Roadmap

• Elementary sorting (quadratic cost)
  – Bubble sort
  – Selection sort
  – Insertion sort
• Recursion (review) and analysis
• Advanced sorting
  – MergeSort
  – QuickSort
MergeSort

• Basic idea
  – Divide an array into two halves
  – Sort each half
  – Merge the two sorted halves into a sorted array

Mergesort overview
Mergesort

- A divide and conquer approach using recursion
  - Partition the original problem into two sub-problems
  - Solve each sub-problem using recursion (sort)
  - Combine the results to solve the original problem (merge)
Merge Two Sorted Arrays

• A **key step** in mergesort
  • Assume subarrays $a[lo..mid]$ (left half) and $a[mid+1 ... Hi]$ (right half) are sorted
  • **copy** $a[]$ to an auxiliary array $aux[]$
  • **merge** the two halves of $aux[]$ to $a[]$ such that it contains all elements and remains sorted

• Example

<table>
<thead>
<tr>
<th>k</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>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>aux[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Merging Two Sorted Arrays

1. Start from the **first** elements of each half;
2. Compare and copy the **smaller** element to a[];
3. Increment indices, and continue;
4. If reaching the end of either half; copy the remaining elements of the other half
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++)
    {
        if (i > mid)  // merge
            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
}
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);
    }
}
```
Assertions

**Assertion.** Statement to test assumptions about your program.

- Helps detect logic bugs.
- Documents code.

**Java assert statement.** Throws an exception unless boolean condition is true.

```java
assert isSorted(a, lo, hi);
```

**Can enable or disable at runtime.** => No cost in production code.

```java
java -ea MyProgram    // enable assertions
java -da MyProgram    // disable assertions (default)
```

**Best practices.** Use to check internal invariants. Assume assertions will be disabled in production code (e.g., don't use for external argument-checking).
First base case encountered
Merge
Mergesort: visualization
Mergesort demo

- Animation

- German folk dance
  - http://www.youtube.com/watch?v=XaqR3G_NVoo
Roadmap

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

• Runtime analysis of recursive algorithms
MergeSort

• Recursion overhead for tiny subarrays
• Merging cost even when the array is already sorted
• Requires copying to auxiliary array
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 ≤ 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
Nerd Sort

• http://www.smbc-comics.com/?db=comics&id=1989
Roadmap

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

• Runtime analysis of recursive algorithms
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=1</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></td>
<td>M</td>
<td>E</td>
<td>R</td>
<td>G</td>
<td>E</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>merge(a, 0, 0, 1)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 2, 2, 3)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 4, 4, 5)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 6, 6, 7)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 8, 8, 9)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 10, 10, 11)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 12, 12, 13)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 14, 14, 15)</td>
<td></td>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz=2</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></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</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>merge(a, 0, 1, 3)</td>
<td></td>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 4, 5, 7)</td>
<td></td>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 8, 9, 11)</td>
<td></td>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</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>
</tr>
<tr>
<td>merge(a, 12, 13, 15)</td>
<td></td>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz=4</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></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>P</td>
</tr>
<tr>
<td>merge(a, 0, 3, 7)</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>merge(a, 8, 11, 15)</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz=8</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></td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td>merge(a, 0, 7, 15)</td>
<td></td>
<td></td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>R</td>
<td>S</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
Recursive Mergesort

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);
    }
}
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++)
        {
            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
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 $T(N)$
  – $T(N) = 2 \times T(N/2) + N$
  – $T(1) = 0$
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 $T(N)$
  – $T(N) = 2 \times T(N/2) + N$
  – $T(1) = 0$

• Big O cost: $N \lg N$
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>
Hw4: Empirical Analysis of Sorting Algorithms

• Implement bubble sort and non-recursive mergesort
• Experimentally compare bubble sort, selection sort, insertion sort, recursive mergesort, and non-recursive mergesort