The sorting problem

- Given a list of $n$ items, place the items in a given order
  - Ascending or descending
    - Numerical
    - Alphabetical
    - etc.
boolean less(Comparable v, Comparable w) {
    return (v.compareTo(w) < 0);
}

void swap(Object[] a, int i, int j) {
    Object temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
Bubble sort

- Iterate through the array comparing adjacent pairs of items, swap them if they are out of relative order
  - Repeat until you make it through the array with 0 swaps

```java
void bubbleSort(Comparable[] a) {
    boolean swapped;
    do {
        swapped = false;
        for(int j = 1; j < a.length; j++) {
            if (less(a[j], a[j-1])) {
                swap(a, j-1, j); swapped = true;
            }
        }
    } while(swapped);
}
```
Bubble sort example

SWAPPED!

1  3  4  5  10
void bubbleSort(Comparable[] a) {
    boolean swapped;
    int to_sort = a.length;
    do {
        swapped = false;
        for(int j = 1; j < to_sort; j++) {
            if (less(a[j], a[j-1])) {
                swap(a, j-1, j); swapped = true;
            }
        }
        to_sort--;
    } while(swapped);
}
How bad is it?

Runtime: $O(n^2)$

"Although the techniques used in the calculations [to analyze the bubble sort] are instructive, the results are disappointing since they tell us that the bubble sort isn't really very good at all."

Donald Knuth

*The Art of Computer Programming*
What is the most efficient way to sort a million 32-bit integers?

I think the bubble sort would be the wrong way to go.
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);
}
Merge Sort trace

3  5  9  10  12  15  21  25

3  12  15  21

5  9  10  25

12  15

3  21

9  25

5  10

15  12  21  3  9  25  10  5
merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    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++];
    }
}
Merge sort analysis

• Runtime:
  • O(n log n)

• So what’s the catch?
  • Now we need O(n) space available for the aux array
    • Sort does not occur *in-place*
Quick Sort

• Choose a *pivot* value
• Place the pivot in the array such that all items at lower indices are less than pivot, and all higher indices are greater
• Recurse for lesser indices and greater indices

```java
void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
Quick Sort

Quicksort partitioning overview
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;
        while (less(v, a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
Partitioning example

\[ v == 44 \]

\[
\begin{array}{cccccccccc}
12 & 33 & 23 & 43 & 44 & 55 & 64 & 77 & 75 \\
\end{array}
\]
Quick sort analysis

Runtime?

In-place?
This implementation of quick sort is not stable

*Stable* sorting maintains the relative ordering of tied values

<table>
<thead>
<tr>
<th>RAM</th>
<th>Speed</th>
<th>Type</th>
<th>CAS</th>
<th>Modules</th>
<th>Size</th>
<th>Price/GB</th>
<th>Rating</th>
<th>Combo</th>
<th>Prime</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corsair Dominator Platinum</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>7</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$14.37</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$229.99</td>
</tr>
<tr>
<td>G.Skill Trident X</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>7</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.31</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$164.99</td>
</tr>
<tr>
<td>Mushkin Redline</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>7</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$11.44</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$182.99</td>
</tr>
<tr>
<td>Mushkin Redline</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>7</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$11.87</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$189.99</td>
</tr>
<tr>
<td>Crucial Ballistix</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>8</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.31</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$164.99</td>
</tr>
<tr>
<td>Crucial Ballistix</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>8</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.31</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$164.99</td>
</tr>
<tr>
<td>Crucial Ballistix Tactical</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>8</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.00</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$159.99</td>
</tr>
<tr>
<td>Mushkin Redline</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>8</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.62</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$169.99</td>
</tr>
<tr>
<td>Mushkin Redline</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>8</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.19</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$162.99</td>
</tr>
<tr>
<td>A-Data XPG V1.0</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>9</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$9.69</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$154.99</td>
</tr>
<tr>
<td>A-Data XPG V1.0</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>9</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$9.37</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$149.99</td>
</tr>
<tr>
<td>A-Data XPG V2</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>9</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$9.69</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$154.99</td>
</tr>
<tr>
<td>A-Data XPG V2</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>9</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$9.69</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$154.99</td>
</tr>
<tr>
<td>AMD Entertainment Edition</td>
<td>DDR3-1600</td>
<td>240-pin DIMM</td>
<td>9</td>
<td>2x8GB</td>
<td>16GB</td>
<td>$10.12</td>
<td>★★★★★</td>
<td></td>
<td></td>
<td>$161.99</td>
</tr>
</tbody>
</table>
The problem of sorting cannot be solved using comparisons with less than $n \log n$ time complexity.

See Proposition I in Chapter 2.2 of the text.
How can we sort without comparison?

Consider the following approach:

- Look at the least-significant digit
- Group numbers with the same digit
  - Maintain relative order
- Place groups back in array together
  - i.e., all the 0’s, all the 1’s, all the 2’s, etc.
- Repeat for increasingly significant digits
Radix sort analysis

- Runtime?
  - \( n \times (\text{length of items in collection}) \)
  - We'll say \( nk \)
    - How can we compare this to the \( n \log n \) runtime that is optimal for comparison-based sorts?
      - Also, why is it called "Radix sort"?

- In-place?

- Stable?
Further thoughts on Eric Schmidt’s question...

- 1,000,000 32-bit integers don’t take up a whole lot of space
  - 4 MB

- What if we needed to sort 1TB of numbers?
  - Won’t all fit in memory...
  - We had been assuming we were performing *internal* sorts
    - Everything in memory

- We now need to consider *external* sorting
  - Where we need to write to disk
Hybrid merge sort

- Read in amount of data that will fit in memory
- Sort it in place
  - I.e., via quick sort
- Write sorted chunk of data to disk
- Repeat until all data is stored in sorted chunks
- Merge chunks together
External sort considerations

- Should we merge all chunks together at once?
  - Means fewer disk read/writes
    - Each merge pass reads/writes every value
    - But also more disk seeks
- Can we do parallel reads/writes to multiple disks?
- Can we use multiple CPUs/cores to speed up processing
Large scale sorts

- What about when you have 1PB of data?
- In 2008, Google sorted 10 trillion 100 byte records on 4000 computers in 6 hours 2 minutes
- 48,000 hard drives were involved
  - At least 1 disk failed during each run of the sort