Algorithmic Analysis and Sorting
Part Three
Friday Four Square!

4:15PM at Gates Computer Science
Midterm Logistics

- Midterm is next **Tuesday, May 7 from 7PM - 10PM**.
- We are in five different rooms, organized by last name:
  - A – G: Go to **Nvidia Auditorium** (right here!)
  - H – L: Go to **Skilling Auditorium**
  - M – U: Go to **Gates B01** or **Gates B03**
  - V – W: Go to **Huang 018** (next door!)
  - X – Z: Go to **Hewlett 101**
- Review session **Sunday 7PM-9PM** in **320-105**.
- Exam is open-book, open-note, but closed-computer.
- Covers material up through and including Wednesday's lecture.
Announcements

- Assignment 3 due right now.
- Assignment 4 (**Boggle**) out, due Monday, May 13 at 2:15PM.
  - Play around with exhaustive recursion and recursive backtracking!
  - Write a computer program that can beat you at your own game. 😊
Previously on CS106B...
Big-O Notation

- Notation for summarizing the long-term growth rate of some function.
- Useful for analyzing runtime:
  - $O(n)$: The runtime grows linearly.
  - $O(n^2)$: The runtime grows quadratically.
  - $O(2^n)$: The runtime grows exponentially.
High-Level Idea

• A recursive sorting algorithm!

• **Base Case:**
  • An empty or single-element list is already sorted.

• **Recursive step:**
  • Break the list in half and recursively sort each part.
  • Use `merge` to combine them back into a single sorted list.

• This algorithm is called `mergesort`. 

●
A Graphical Intuition

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n log n) \]
## Mergesort Times

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
<th>Mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
<td>0.006</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
<td>0.010</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
<td>0.017</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
<td>0.021</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
<td>0.028</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
<td>0.035</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
<td>0.041</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
<td>0.042</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
<td>0.048</td>
</tr>
</tbody>
</table>
Can we do Better?

• Mergesort is $O(n \log n)$.
• This is asymptotically better than $O(n^2)$.
• Can we do better?
  • In general, no: comparison-based sorts cannot have a worst-case runtime better than $O(n \log n)$.

• In the worst case, we can only get faster by a constant factor!
And now... new stuff!
A Trivial Observation
A Trivial Observation
A Trivial Observation
A Trivial Observation
A Trivial Observation
So What?

- This idea leads to a particularly clever sorting algorithm.

- Idea:
  - Pick an element from the array.
  - Put the smaller elements on one side.
  - Put the bigger elements on the other side.
  - Recursively sort each half.

- But how do we do the middle two steps?
Partitioning

- Pick a **pivot element**.
- Move everything less than the pivot to the left of the pivot.
- Move everything greater than the pivot to the right of the pivot.
- Good news: $O(n)$ algorithm exists!
- Bad news: it's a bit tricky...
The Partition Algorithm
The Partition Algorithm

6 3 8 2 9 1 4 5 7 10
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm

6  3  8  2  9  1  4  5  7  10

Diagram of the partition algorithm with numbers sorted into groups.
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
The Partition Algorithm
Code for Partition
Code for Partition

```c++
int partition(Vector<int>& v, int low, int high) {
  int pivot = v[low];
  int left = low + 1, right = high;

  while (left < right) {
    while (left < right && v[right] >= pivot) --right;
    while (left < right && v[left] < pivot) ++left;
    if (left < right) swap(v[left], v[right]);
  }

  if (pivot < v[right]) return low;
  swap(v[low], v[right]);
  return right;
}
```
A Partition-Based Sort

- Idea:
  - Partition the array around some element.
  - Recursively sort the left and right halves.
- This works extremely quickly.
- In fact... the algorithm is called quicksort.
Quicksort
void quicksort(Vector<int> &v, int low, int high) {
    if (low >= high) return;

    int partitionPoint = partition(v, low, high);
    quicksort(v, low, partitionPoint - 1);
    quicksort(v, partitionPoint + 1, high);
}

Quicksort
How fast is quicksort?
It depends on our choice of pivot.
Suppose we get lucky...
Suppose we get lucky...
Suppose we get lucky...
Suppose we get lucky...
Suppose we get lucky...

\[ O(n \log n) \]
Suppose we get *sorta* lucky...
Suppose we get *sorta* lucky...
Suppose we get *sorta* lucky...
Suppose we get *sorta* lucky...
Suppose we get *sorta* lucky...
Suppose we get *sorta* lucky...

\[ O(n \log n) \]
Suppose we get unlucky
Suppose we get unlucky.
Suppose we get unlucky
Suppose we get unlucky
Suppose we get unlucky...
Suppose we get unlucky

\[ n \]

\[ n - 1 \]

\[ n - 2 \]

\[ n - 3 \]

...
Suppose we get unlucky

\[ n \]

\[ n - 1 \]

\[ n - 2 \]

\[ n - 3 \]

\[ \ldots \quad O(n^2) \]
Quicksort is Strange

- In most cases, quicksort has runtime $O(n \log n)$.
- In the worst case, quicksort has runtime $O(n^2)$.
- How can you avoid this?

- **Pick better pivots!**
  - Pick the median.
    - Can be done in $O(n)$, but *expensive* $O(n)$.
  - Pick the “median-of-three.”
    - Better than nothing, but still can hit worst case.
  - Pick randomly.
    - Extremely low probability of $O(n^2)$.
Quicksort is Fast

- Although quicksort is $O(n^2)$ in the worst case, it is one of the fastest known sorting algorithms.
- $O(n^2)$ behavior is extremely unlikely with random pivots; runtime is usually a very good $O(n \log n)$.
- It's hard to argue with the numbers...
## Timing Quicksort

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
<th>Mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
<td>0.006</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
<td>0.010</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
<td>0.017</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
<td>0.021</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
<td>0.028</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
<td>0.035</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
<td>0.041</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
<td>0.042</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
<td>0.048</td>
</tr>
</tbody>
</table>
## Timing Quicksort

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
<th>Mergesort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
<td>0.010</td>
<td>0.002</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
<td>0.017</td>
<td>0.004</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
<td>0.021</td>
<td>0.005</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
<td>0.028</td>
<td>0.006</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
<td>0.035</td>
<td>0.008</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
<td>0.041</td>
<td>0.009</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
<td>0.042</td>
<td>0.009</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
<td>0.048</td>
<td>0.012</td>
</tr>
</tbody>
</table>
An Interesting Observation

- Big-O notation talks about long-term growth, but says nothing about small inputs.
- For small inputs, insertion sort can be better than mergesort or quicksort.

![Graph showing runtime vs input size for insertion sort and mergesort. Insertion Sort line is steeper than Mergesort line for small input sizes.]
Hybrid Sorting Algorithms

• Modify the mergesort algorithm to switch to insertion sort when the input gets sufficiently small.

• This is called a hybrid sorting algorithm.
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        Vector<int> left, right;
        for (int i = 0; i < v.size() / 2; i++)
            left += v[i];
        for (int i = v.size() / 2; i < v.size(); i++)
            right += v[i];
        hybridMergesort(left);
        hybridMergesort(right);
        merge(left, right, v);
    }
}
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        Vector<int> left, right;
        for (int i = 0; i < v.size() / 2; i++)
            left += v[i];
        for (int i = v.size() / 2; i < v.size(); i++)
            right += v[i];

        hybridMergesort(left);
        hybridMergesort(right);

        merge(left, right, v);
    }
}
Hybrid Sorting Algorithms
## Runtime for Hybrid Mergesort

<table>
<thead>
<tr>
<th>Size</th>
<th>Mergesort</th>
<th>Hybrid Mergesort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>0.063</td>
<td>0.019</td>
<td>0.012</td>
</tr>
<tr>
<td>3000000</td>
<td>0.176</td>
<td>0.061</td>
<td>0.060</td>
</tr>
<tr>
<td>5000000</td>
<td>0.283</td>
<td>0.091</td>
<td>0.063</td>
</tr>
<tr>
<td>7000000</td>
<td>0.396</td>
<td>0.130</td>
<td>0.089</td>
</tr>
<tr>
<td>9000000</td>
<td>0.510</td>
<td>0.165</td>
<td>0.118</td>
</tr>
<tr>
<td>1100000</td>
<td>0.608</td>
<td>0.223</td>
<td>0.151</td>
</tr>
<tr>
<td>1300000</td>
<td>0.703</td>
<td>0.246</td>
<td>0.179</td>
</tr>
<tr>
<td>1500000</td>
<td>0.844</td>
<td>0.28</td>
<td>0.215</td>
</tr>
<tr>
<td>1700000</td>
<td>0.995</td>
<td>0.326</td>
<td>0.243</td>
</tr>
<tr>
<td>1900000</td>
<td>1.070</td>
<td>0.355</td>
<td>0.274</td>
</tr>
</tbody>
</table>
Hybrid Sorts in Practice

- Introspective Sort (*Introsort*)
  - Based on quicksort, insertion sort, and *heapsort*.
  - Heapsort is $O(n \log n)$ and a bit faster than mergesort.
  - Uses quicksort, then switches to heapsort if it looks like the algorithm is degenerating to $O(n^2)$.
  - Uses insertion sort for small inputs.
  - Gains the raw speed of quicksort without any of the drawbacks.
## Runtime for Introsort

<table>
<thead>
<tr>
<th>Size</th>
<th>Mergesort</th>
<th>Hybrid Mergesort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.063</td>
<td>0.019</td>
<td>0.012</td>
</tr>
<tr>
<td>300000</td>
<td>0.176</td>
<td>0.061</td>
<td>0.060</td>
</tr>
<tr>
<td>500000</td>
<td>0.283</td>
<td>0.091</td>
<td>0.063</td>
</tr>
<tr>
<td>700000</td>
<td>0.396</td>
<td>0.130</td>
<td>0.089</td>
</tr>
<tr>
<td>900000</td>
<td>0.510</td>
<td>0.165</td>
<td>0.118</td>
</tr>
<tr>
<td>1100000</td>
<td>0.608</td>
<td>0.223</td>
<td>0.151</td>
</tr>
<tr>
<td>1300000</td>
<td>0.703</td>
<td>0.246</td>
<td>0.179</td>
</tr>
<tr>
<td>1500000</td>
<td>0.844</td>
<td>0.28</td>
<td>0.215</td>
</tr>
<tr>
<td>1700000</td>
<td>0.995</td>
<td>0.326</td>
<td>0.243</td>
</tr>
<tr>
<td>1900000</td>
<td>1.070</td>
<td>0.355</td>
<td>0.274</td>
</tr>
</tbody>
</table>
## Runtime for Introsort

<table>
<thead>
<tr>
<th>Size</th>
<th>Mergesort</th>
<th>Hybrid Mergesort</th>
<th>Quicksort</th>
<th>Introsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.063</td>
<td>0.019</td>
<td>0.012</td>
<td>0.009</td>
</tr>
<tr>
<td>300000</td>
<td>0.176</td>
<td>0.061</td>
<td>0.060</td>
<td>0.028</td>
</tr>
<tr>
<td>500000</td>
<td>0.283</td>
<td>0.091</td>
<td>0.063</td>
<td>0.043</td>
</tr>
<tr>
<td>700000</td>
<td>0.396</td>
<td>0.130</td>
<td>0.089</td>
<td>0.060</td>
</tr>
<tr>
<td>900000</td>
<td>0.510</td>
<td>0.165</td>
<td>0.118</td>
<td>0.078</td>
</tr>
<tr>
<td>1100000</td>
<td>0.608</td>
<td>0.223</td>
<td>0.151</td>
<td>0.092</td>
</tr>
<tr>
<td>1300000</td>
<td>0.703</td>
<td>0.246</td>
<td>0.179</td>
<td>0.107</td>
</tr>
<tr>
<td>1500000</td>
<td>0.844</td>
<td>0.28</td>
<td>0.215</td>
<td>0.123</td>
</tr>
<tr>
<td>1700000</td>
<td>0.995</td>
<td>0.326</td>
<td>0.243</td>
<td>0.139</td>
</tr>
<tr>
<td>1900000</td>
<td>1.070</td>
<td>0.355</td>
<td>0.274</td>
<td>0.158</td>
</tr>
</tbody>
</table>
We've spent all of our time talking about **fast** and **efficient** sorting algorithms.
However, we have neglected to find slow and inefficient sorting algorithms.
Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms

Hermann Gruber\textsuperscript{1} and Markus Holzer\textsuperscript{2} and Oliver Ruepp\textsuperscript{2}

\textsuperscript{1} Institut für Informatik, Ludwig-Maximilians-Universität München, Oettingenstraße 67, D-80538 München, Germany
email: gruberh@tcs.ifi.lmu.de

\textsuperscript{2} Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching bei München, Germany
email: \{holzer,ruepp\}@in.tum.de
Introducing Bogosort
Next Time

• **Designing Abstractions**
  • How do you build new container classes?

• **Class Design**
  • What do classes look like in C++?