Algorithmic Analysis and Sorting
Part Two
A former student of mine (Montse Cordero) is studying different methods of teaching how to divide fractions. A key step in her analysis is looking at the computational complexity of the different approaches using big-O notation!
Recap from Last Time
Big-O Notation

- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- Works by dropping constants and low-order growth terms.
- For example, this code runs in time O(n):
  ```cpp
  for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
  }
  ```
Big-O Notation

- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- Works by dropping constants and low-order growth terms.
- For example, this code runs in time $O(n^2)$:
  ```cpp
  for (int i = 0; i < vec.size(); i++) {
    for (int j = 0; j < vec.size(); j++) {
      cout << (vec[i] + vec[j]) << endl;
    }
  }
  ```
Sorting Algorithms

- The *sorting problem* is to take in a list of things (integers, strings, etc.) and rearrange them into sorted order.
- Last time, we saw *selection sort*, an algorithm that runs in time $O(n^2)$.
- This means that doubling the objects of elements to sort will (roughly) quadruple the time required.
- Our question for today: can we sort numbers faster than this?
New Stuff!
Another Idea: **Insertion Sort**

7

4

2

1

6
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**

2

4

7

1

6
Another Idea: *Insertion Sort*
Another Idea: **Insertion Sort**
/**
 * Sorts the specified vector using insertion sort.
 * 
 * @param v The vector to sort.
 */

void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        /* Scan backwards until either (1) there is no
         * preceding element or the preceding element is
         * no bigger than us.
         */
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] <= v[j + 1]) break;
            /* Swap this element back one step. */
            swap(v[j], v[j + 1]);
        }
    }
}
# Selection Sort vs Insertion Sort

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
</tr>
</tbody>
</table>
Of insertion sort and selection sort:

1. Which algorithm runs does more work at the start?
2. Which algorithm runs does more work at the end?
Thinking About $O(n^2)$

$T(n) \approx \frac{1}{4}T(n)$

$(n / 2)^2 = n^2 / 4$
The Key Insight: *Merge*

Each step makes a single comparison and reduces the number of elements by one.

If there are $n$ total elements, this algorithm runs in time $O(n)$. 
/**
 * Given two queues of elements in sorted order, merges them together
 * into a single sorted sequence.
 *
 * This can easily be adapted to work with Vectors or other types of
 * sequences, but I thought it was easiest using Queues.
 *
 * @param one The first sorted queue.
 * @param two The second sorted queue.
 * @return A single sorted queue holding all elements of one and two.
 */
Queue<int> merge(Queue<int>& one, Queue<int>& two) {
    Queue<int> result;

    /* Keep comparing the first elements of each queue against one
     * another until one queue is exhausted.
     */
    while (!one.isEmpty() && !two.isEmpty()) {
        if (one.peek() < two.peek()) result.enqueue(one.dequeue());
        else result.enqueue(two.dequeue());
    }

    /* Add all remaining elements of the other queue to the result. */
    while (!one.isEmpty()) result.enqueue(one.dequeue());
    while (!two.isEmpty()) result.enqueue(two.dequeue());

    return result;
}
void splitSort(Vector<int>& v) {
  /* Split the vector in half */
  Vector<int> left, right;
  for (int i = 0; i < v.size() / 2; i++) {
    left += v[i];
  }
  for (int j = v.size() / 2; j < v.size(); j++) {
    right += v[i];
  }

  /* Sort each half. */
  insertionSort(left);
  insertionSort(right);

  /* Merge them back together. */
  merge(left, right, v);
}
## Performance Comparison

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
</tr>
</tbody>
</table>
Time-Out for Announcements!
Assignment 4

- **Reminder:** Assignment 4 is due one week from Friday.
  - Following the assignment timetable we suggested, you should try to be done with the Doctors Without Orders problem by the end of the evening and start working on Disaster Planning.
  - Again, remember that the midterm is coming up right after the assignment due date, so this is probably not a good place to use late days.
Practice Midterm Exam

• We’ll be holding a practice midterm exam on Monday, February 13\textsuperscript{th} from 7PM - 10PM, in Hewlett 200.

• It’s held under realistic conditions and is a fantastic way to prepare for the exam. You should plan to attend this unless you have a compelling reason not to do so.

• We’ll have more details about the midterm exam on Friday.
Honor Code Reminder

- The good news is that the overwhelming majority of you are honest and hardworking.
- The bad news is that some of you turned in Assignment 2 submissions that are clearly not your own work.
- **We take the Honor Code seriously in this course.** We’ll be writing up these cases and submitting them to the Office of Community Standards.
  - Standard sanction for a first offense is many hours of community service, an academic integrity seminar, and a possible suspension.
- Please ask for help if you need it. That’s what we’re here for!
More Assorted Sorts of Sorts!
A Better Idea

- We can speed up insertion sort by almost a factor of two by splitting the array in half, sorting each part independently, and merging the results together.
- So why not split into fourths? That would give a $4 \times$ improvement.
- So why not split into eighths? That would give an $8 \times$ improvement.
- **Question:** What happens if we *never stop splitting*?
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.
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are already sorted. */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    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];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
What is the complexity of mergesort?
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are already sorted. */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    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];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    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];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
A Graphical Intuition

How many levels are there?
Slicing and Dicing

• After zero recursive calls: \( n \)
• After one recursive call: \( n / 2 \)
• After two recursive calls: \( n / 4 \)
• After three recursive calls: \( n / 8 \)

\[ \cdots \]

• After \( k \) recursive calls: \( n / 2^k \)
Cutting in Half

- After \( k \) recursive calls, there are \( n / 2^k \) elements left.
- Mergesort stops recursing when there are zero or one elements left.
- Solving for the number of levels:
  \[
  \frac{n}{2^k} = 1
  \]
  \[
  n = 2^k
  \]
  \[
  \log_2 n = k
  \]
- So mergesort recurses \( \log_2 n \) levels deep.
A Graphical Intuition

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]

\[ O(n \log n) \]
### 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 $O(n \log n)$, which is faster than insertion sort’s $O(n^2)$ runtime.
- Can we do better than this?
  - 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!*
- What might that look like?
An Interesting Observation

- Big-O notation talks about long-term growth, but says nothing about small inputs.
- For small inputs, insertion sort can be faster than mergesort.
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);
    }
}
## Runtime for Hybrid Mergesort

<table>
<thead>
<tr>
<th>Size</th>
<th>Mergesort</th>
<th>Hybrid Mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.063</td>
<td>0.019</td>
</tr>
<tr>
<td>300000</td>
<td>0.176</td>
<td>0.061</td>
</tr>
<tr>
<td>500000</td>
<td>0.283</td>
<td>0.091</td>
</tr>
<tr>
<td>700000</td>
<td>0.396</td>
<td>0.130</td>
</tr>
<tr>
<td>900000</td>
<td>0.510</td>
<td>0.165</td>
</tr>
<tr>
<td>1100000</td>
<td>0.608</td>
<td>0.223</td>
</tr>
<tr>
<td>1300000</td>
<td>0.703</td>
<td>0.246</td>
</tr>
<tr>
<td>1500000</td>
<td>0.844</td>
<td>0.28</td>
</tr>
<tr>
<td>1700000</td>
<td>0.995</td>
<td>0.326</td>
</tr>
<tr>
<td>1900000</td>
<td>1.070</td>
<td>0.355</td>
</tr>
</tbody>
</table>
Hybrid Sorts in Practice

- **Introspective Sort (IntroSort)**
  - Based on three sorting algorithms: quicksort, heapsort, and insertion sort.
  - Quicksort runs in time $O(n \log n)$ on average, but in the worst case runs in time $O(n^2)$.
  - Heapsort runs in time $O(n \log n)$ and is very memory-efficient. You’ll see it on Assignment 5!
  - Uses insertion sort for small inputs.
## Runtime for Introsort

<table>
<thead>
<tr>
<th>Size</th>
<th>Mergesort</th>
<th>Hybrid Mergesort</th>
<th>Introsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>0.063</td>
<td>0.019</td>
<td>0.009</td>
</tr>
<tr>
<td>300000</td>
<td>0.176</td>
<td>0.061</td>
<td>0.028</td>
</tr>
<tr>
<td>500000</td>
<td>0.283</td>
<td>0.091</td>
<td>0.043</td>
</tr>
<tr>
<td>700000</td>
<td>0.396</td>
<td>0.130</td>
<td>0.060</td>
</tr>
<tr>
<td>900000</td>
<td>0.510</td>
<td>0.165</td>
<td>0.078</td>
</tr>
<tr>
<td>1100000</td>
<td>0.608</td>
<td>0.223</td>
<td>0.092</td>
</tr>
<tr>
<td>1300000</td>
<td>0.703</td>
<td>0.246</td>
<td>0.107</td>
</tr>
<tr>
<td>1500000</td>
<td>0.844</td>
<td>0.28</td>
<td>0.123</td>
</tr>
<tr>
<td>1700000</td>
<td>0.995</td>
<td>0.326</td>
<td>0.139</td>
</tr>
<tr>
<td>1900000</td>
<td>1.070</td>
<td>0.355</td>
<td>0.158</td>
</tr>
</tbody>
</table>
Why All This Matters

• Big-O notation gives us a *quantitative way* to predict runtimes.

• Those predictions provide a *quantitative intuition* for how to improve our algorithms.

• Understanding the nuances of big-O notation then leads us to design algorithms that are better than the sum of their parts.
Next Time

- **Designing Abstractions**
  - How do you build new container classes?
- **Class Design**
  - What do classes look like in C++?