Algorithmic Analysis and Sorting
Part One
Computers use roughly 3% of all the electricity generated in the United States. This electricity generation produces around 826 megatons of CO₂ each year. Reducing the need for computing power – or using that power more wisely – could have a big impact on CO₂ emissions.
**Fundamental Question:**

How can we compare solutions to problems?
One Idea: *Runtime*
Runtime is Noisy

- Runtime is highly sensitive to *which computer you’re using*.
- Runtime is highly sensitive to *which inputs you’re testing*.
- Runtime is highly sensitive to *external factors*.
bool linearSearch(const string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}

Work Done: At most $k_0 n + k_1$
Big Observations

• Don't need to explicitly compute these constants.
  • Whether runtime is $4n + 10$ or $100n + 137$, runtime is still proportional to input size.
  • Can just plot the runtime to obtain actual values.

• Only the dominant term matters.
  • For both $4n + 1000$ and $n + 137$, for very large $n$ most of the runtime is explained by $n$.

• Is there a concise way of describing this?
Big-O
Big-O Notation

• Ignore *everything* except the dominant growth term, including constant factors.

• Examples:
  • $4n + 4 = \mathcal{O}(n)$
  • $137n + 271 = \mathcal{O}(n)$
  • $n^2 + 3n + 4 = \mathcal{O}(n^2)$
  • $2^n + n^3 = \mathcal{O}(2^n)$

For the mathematically inclined:

\[ f(n) = \mathcal{O}(g(n)) \text{ if } \exists n_0 \in \mathbb{R}. \exists c \in \mathbb{R}. \forall n \geq n_0. f(n) \leq c|g(n)| \]
Algorithmic Analysis with Big-O
Algorithmic Analysis with Big-O

def double average(const Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }

    return total / vec.size();
}
Algorithmic Analysis with Big-O

double average(const Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
double average(const Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}

O(n)
A More Interesting Example
A More Interesting Example

```cpp
bool linearSearch(const string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

How do we analyze this?
Types of Analysis

- **Worst-Case Analysis**
  - What's the *worst* possible runtime for the algorithm?
  - Useful for “sleeping well at night.”

- **Best-Case Analysis**
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.

- **Average-Case Analysis**
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!
Types of Analysis

• **Worst-Case Analysis**
  • What's the *worst* possible runtime for the algorithm?
  • Useful for “sleeping well at night.”

**Best-Case Analysis**

What's the *best* possible runtime for the algorithm?
Useful to see if the algorithm performs well in some cases.

**Average-Case Analysis**

What's the *average* runtime for the algorithm?
Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!
A More Interesting Example

```cpp
bool linearSearch(const string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

$O(n)$
Determining if a Character is a Letter
Determining if a Character is a Letter

```cpp
bool isAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
           (ch >= 'a' && ch <= 'z');
}
```
Determining if a Character is a Letter

```cpp
bool isAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
            (ch >= 'a' && ch <= 'z');
}
```

$O(1)$
What Can Big-O Tell Us?

• Long-term behavior of a function.
  • If algorithm A has runtime $O(n)$ and algorithm B has runtime $O(n^2)$, for very large inputs algorithm A will always be faster.
  • If algorithm A has runtime $O(n)$, for large inputs, doubling the size of the input doubles the runtime.
What *Can't* Big-O Tell Us?

- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$
Growth Rates, Part One

- $O(1)$
- $O(\log n)$
- $O(n)$
Growth Rates, Part Two

$O(n)$

$O(n \log n)$

$O(n^2)$
Growth Rates, Part Three

- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
To Give You A Better Sense...

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
## Comparison of Runtimes
*(assuming 1 operation = 1 nanosecond)*

<table>
<thead>
<tr>
<th>Size</th>
<th>1</th>
<th>$\log_2 n$</th>
<th>n</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1ns</td>
<td>9.966ns</td>
<td>1μs</td>
<td>9.966μs</td>
<td>1ms</td>
<td>1s</td>
<td>$3.4 \times 10^{284}$ yr</td>
</tr>
<tr>
<td>2000</td>
<td>1ns</td>
<td>10.966ns</td>
<td>2μs</td>
<td>21.932μs</td>
<td>4ms</td>
<td>8s</td>
<td>Just... wow.</td>
</tr>
<tr>
<td>3000</td>
<td>1ns</td>
<td>11.551ns</td>
<td>3μs</td>
<td>34.652μs</td>
<td>9ms</td>
<td>27s</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>1ns</td>
<td>11.966ns</td>
<td>4μs</td>
<td>47.863μs</td>
<td>16ms</td>
<td>1.067min</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>1ns</td>
<td>12.288ns</td>
<td>5μs</td>
<td>61.439μs</td>
<td>25ms</td>
<td>2.083min</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>1ns</td>
<td>12.551ns</td>
<td>6μs</td>
<td>75.304μs</td>
<td>36ms</td>
<td>3.6min</td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>1ns</td>
<td>12.773ns</td>
<td>7μs</td>
<td>89.412μs</td>
<td>49ms</td>
<td>5.717min</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>1ns</td>
<td>12.966ns</td>
<td>8μs</td>
<td>103.726μs</td>
<td>64ms</td>
<td>8.533min</td>
<td></td>
</tr>
<tr>
<td>9000</td>
<td>1ns</td>
<td>13.136ns</td>
<td>9μs</td>
<td>118.221μs</td>
<td>81ms</td>
<td>12.15min</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1ns</td>
<td>13.288ns</td>
<td>10μs</td>
<td>132.877μs</td>
<td>100ms</td>
<td>16.667min</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>1ns</td>
<td>13.425ns</td>
<td>11μs</td>
<td>147.677μs</td>
<td>121ms</td>
<td>22.183min</td>
<td></td>
</tr>
<tr>
<td>12000</td>
<td>1ns</td>
<td>13.551ns</td>
<td>12μs</td>
<td>162.609μs</td>
<td>144ms</td>
<td>28.8min</td>
<td></td>
</tr>
<tr>
<td>13000</td>
<td>1ns</td>
<td>13.666ns</td>
<td>13μs</td>
<td>177.661μs</td>
<td>169ms</td>
<td>36.617min</td>
<td></td>
</tr>
<tr>
<td>14000</td>
<td>1ns</td>
<td>13.773ns</td>
<td>14μs</td>
<td>192.824μs</td>
<td>196ms</td>
<td>45.733min</td>
<td></td>
</tr>
</tbody>
</table>
Summary of Big-O

• A means of describing the growth rate of a function.
• Ignores all but the leading term.
• Ignores constants.
• Allows for quantitative ranking of algorithms.
• Allows for quantitative reasoning about algorithms.
Time-Out for Announcements!
Assignment 4

- Assignment 4 (*Recursion to the Rescue!*)) goes out today. It’s due next Friday, February 17th at the start of class.
  - We’ve pushed the due date for this assignment back a class period to give you a little more breathing room.
  - You’re encouraged to work in pairs on this assignment. These problems are great to discuss in a group.
  - **Start early!** There’s a suggested timetable on the front of the assignment handout that we think will help you keep on track.
  - **Be careful about taking late days here**. The midterm is on the Tuesday after this assignment is due.
- Anton will be holding YEAH hours tonight from 7PM – 8PM in room 420-040. Highly recommended, as always!
- Assignment 3 was due today at 11:30. Feel free to use a late day if you need to, though keep in mind that you’ll want to get a jump on Assignment 4.
Girl Code @Stanford

• This summer, I’ll be running our fifth iteration of Girl Code @Stanford from July 10th – July 21st.

• We invite high-school girls (primarily from low- to middle-income schools in majority-minority areas) to come to campus for two weeks to learn CS, meet researchers, and talk to folks from industry.

• We’re looking for Stanford students to serve as “Workshop Assistants” during the program. We pay competitively (roughly $3,000 over two weeks).

• Interested? Learn more and apply using this link: https://goo.gl/forms/icYcRiX8PTgoVJ0n1

All current Stanford students are invited to apply. Feel free to forward this link around!
Back to CS106B!
Sorting Algorithms
The Sorting Problem

• Given a list of elements, sort those elements in ascending order.

• There are *many* ways to solve this problem.

• What is the *best* way to solve this problem?

• We'll use big-O to find out!
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort

4  1  2  7  6
An Initial Idea: **Selection Sort**
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: *Selection Sort*

```
1 4 2 7 6
```
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: **Selection Sort**
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort

1  2  4  7  6
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort

[Diagram showing the selection sort process with numbers 1, 2, 4, 7, and 6 in ascending order]
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort

1 2 4 6 7
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: **Selection Sort**
An Initial Idea: Selection Sort
Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position.
- (etc.)
/**
 * Sorts the specified vector using the selection sort algorithm.
 *
 * @param elems The elements to sort.
 */
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}

/**
 * Given a vector and a starting point, returns the index of the smallest
 * element in that vector at or after the starting point
 *
 * @param elems The elements in question.
 * @param startPoint The starting index in the vector.
 * @return The index of the smallest element at or after that point
 *         in the vector.
 */
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex])
            smallestIndex = i;
    }
    return smallestIndex;
}
Analyzing Selection Sort

• How much work do we do for selection sort?
• To find the smallest value, we need to look at all $n$ array elements.
• To find the second-smallest value, we need to look at $n - 1$ array elements.
• To find the third-smallest value, we need to look at $n - 2$ array elements.
• Work is $n + (n - 1) + (n - 2) + \ldots + 1$. 
\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
The Complexity of Selection Sort

\[ O(n \ (n + 1) / 2) \]
\[ = O(n \ (n + 1)) \]
\[ = O(n^2 + n) \]
\[ = O(n^2) \]

So selection sort runs in time \( O(n^2) \).
Thinking About O(n^2)
Thinking About $O(n^2)$
Thinking About $O(n^2)$
Thinking About $O(n^2)$

$T(n)$

\[
\begin{array}{ccccccccccccccc}
14 & 6 & 3 & 9 & 7 & 16 & 2 & 15 & 14 & 6 & 3 & 9 & 7 & 16 & 2 & 15 & 5 & 10 & 8 & 11 & 1 & 13 & 12 & 4 \\
\end{array}
\]
Thinking About $O(n^2)$

T(n)

T(2n) \approx 4T(n)
## Selection Sort Times

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
</tr>
</tbody>
</table>
Next Time

- **Faster Sorting Algorithms**
  - Can you beat $O(n^2)$ time?
- **Hybrid Sorting Algorithms**
  - When might selection sort be useful?