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 do we measure efficiency?
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

• If our goal is to extrapolate out the runtime, we don’t need to know the constants in advance. We can figure them out by running the code.

• For “sufficiently large” inputs, 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
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();
}
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();
}

O(n)

O(n) means “the runtime is proportional to the size of the input.” We’d say that this code runs in *linear time*. 
A More Interesting Example
A More Interesting Example

```c++
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, or CS265 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!
bool linearSearch(const string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}

Worst-Case Runtime: \( O(n) \)
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, or CS265 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, or CS265 for more information!
Three Cheers for Optimism!

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

**Best-Case Runtime:** \(\mathcal{O}(1)\)

\(\mathcal{O}(1)\) means “the runtime doesn’t depend on the size of the input.” In the best case, this code runs in constant time.
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.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$
Some Standard Runtime Complexities
Growth Rates, Part I

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

- $O(n)$
- $O(n \log n)$
- $O(n^2)$

What is this strange $n \log n$? Stay tuned!
Growth Rates, Part III

- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
All Together Now!

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

Exponential runtimes are scary! Avoid them if at all possible.
## Comparison of Runtimes

*(assuming 1 operation = 1 nanosecond)*

<table>
<thead>
<tr>
<th>Size</th>
<th>Runtime (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1 ns</td>
</tr>
<tr>
<td>2000</td>
<td>1 ns</td>
</tr>
<tr>
<td>3000</td>
<td>1 ns</td>
</tr>
<tr>
<td>4000</td>
<td>1 ns</td>
</tr>
<tr>
<td>5000</td>
<td>1 ns</td>
</tr>
<tr>
<td>6000</td>
<td>1 ns</td>
</tr>
<tr>
<td>7000</td>
<td>1 ns</td>
</tr>
<tr>
<td>8000</td>
<td>1 ns</td>
</tr>
<tr>
<td>9000</td>
<td>1 ns</td>
</tr>
<tr>
<td>10000</td>
<td>1 ns</td>
</tr>
<tr>
<td>11000</td>
<td>1 ns</td>
</tr>
<tr>
<td>12000</td>
<td>1 ns</td>
</tr>
<tr>
<td>13000</td>
<td>1 ns</td>
</tr>
<tr>
<td>14000</td>
<td>1 ns</td>
</tr>
</tbody>
</table>
The Story So Far

• Big-O notation is a quantitative measure of how a function’s runtime scales.

• It ignores constants and lower-order terms. Only the fastest-growing terms matter.

• Big-O notation lets us predict how long a function will take to run.

• Big-O notation lets us quantitatively compare algorithms.
Time-Out for Announcements!
Programming Assignments

• Assignment 3 is due on Wednesday.
  • If you’re following our timetable, you should be done with the Sierpinski triangle, Human Pyramids, and Shift Scheduling at this point and should be working on Riding Circuit.
  • Have questions? Stop by the LaIR, email your section leader, or visit Piazza!

• Assignment 4 will go out on Wednesday.
  • We’ll be holding YEAH Hours for this assignment this Wednesday at 7:00PM in room 380-380Y.
big-Oward!
Sorting Algorithms
What is sorting?
One style of “sorting,” but not the one we’re thinking about...
<table>
<thead>
<tr>
<th>Time</th>
<th>Auto</th>
<th>Athlete</th>
<th>Nationality</th>
<th>Date</th>
<th>Venue</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:36.8</td>
<td>Maria Gomers</td>
<td>Netherlands</td>
<td>14 June 1969[7]</td>
<td>Leicester</td>
<td></td>
</tr>
<tr>
<td>4:35.3</td>
<td>Ellen Tittel</td>
<td>West Germany</td>
<td>20 August 1971[7]</td>
<td>Sittard</td>
<td></td>
</tr>
<tr>
<td>4:29.5</td>
<td>Paola Pigni</td>
<td>Italy</td>
<td>8 August 1973[7]</td>
<td>Viareggio</td>
<td></td>
</tr>
<tr>
<td>4:15.61</td>
<td>Paula Ivan</td>
<td>Romania</td>
<td>10 July 1989[7]</td>
<td>Nice</td>
<td></td>
</tr>
</tbody>
</table>

**Problem:** Given a list of data points, sort those data points into ascending / descending order by some quantity.
Suppose we want to rearrange a sequence to put elements into ascending order. What are some strategies we could use? How do those strategies compare? Is there a “best” strategy?
An Initial Idea: *Insertion Sort*
An Initial Idea: *Insertion Sort*

\[
\begin{align*}
7 & \quad 4 & \quad 2 & \quad 1 & \quad 6
\end{align*}
\]
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**

![Insertion Sort Diagram](image-url)
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**

4 7 2 1 6
An Initial Idea: **Insertion Sort**
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**
An Initial Idea: **Insertion Sort**
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**
An Initial Idea: **Insertion Sort**
An Initial Idea: *Insertion Sort*
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**
An Initial Idea: Insertion Sort
An Initial Idea: **Insertion Sort**
An Initial Idea: **Insertion Sort**
An Initial Idea: *Insertion Sort*
An Initial Idea: *Insertion Sort*
An Initial Idea: **Insertion Sort**
An Initial 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]);
        }
    }
}
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?

1  2  4  6  7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

![Insertion Sort Visualization]

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

Work done: $O(n)$
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

7 6 4 2 1
How Fast is Insertion Sort?

7 — 6 — 4 — 2 — 1
How Fast is Insertion Sort?

6

7

4

2

1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

6  7  4  2  1
How Fast is Insertion Sort?

6 7 4 2 1
How Fast is Insertion Sort?

6

4

7

2

1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

4
6
7
2
1
How Fast is Insertion Sort?

4 6 7 2 1
How Fast is Insertion Sort?

4
6
7
2
1
How Fast is Insertion Sort?

4  6  7  2  1
How Fast is Insertion Sort?

4
6
2
7
1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

4  2  6  7  1
How Fast is Insertion Sort?

4  2  6  7  1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2 4 6 7 1
How Fast is Insertion Sort?

2 4 6 7 1
How Fast is Insertion Sort?

2 4 6 7 1
How Fast is Insertion Sort?

2  4  6  1  7
How Fast is Insertion Sort?

2 4 6 1 7
How Fast is Insertion Sort?

2  4  1  6  7
How Fast is Insertion Sort?

2

4

1

6

7
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2 1 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

Work Done: $1 + 2 + 3 + 4$
If we run insertion sort on a sequence of $n$ elements, we might have to do

$$1 + 2 + 3 + 4 + \ldots + (n - 2) + (n - 1)$$

swaps. How many swaps is this?
$1 + 2 + 3 + ... + (n - 2) + (n - 1) = \frac{n(n - 1)}{2}$
The Complexity of Insertion Sort

• In the worst case, insertion sort takes time
  \[ O(n (n - 1) / 2) \]
  \[ = O(n (n - 1)) \]
  \[ = O(n^2 - n) \]
  \[ = O(n^2). \]

• **Fun fact:** Insertion sorting an array of random values takes, on average, \( O(n^2) \) time.
  • Curious why? Come talk to me after class!
Thinking About $O(n^2)$

$T(n) \approx 4T(n)$
Next Time

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