Algorithmic Analysis and Sorting
Part One
Announcements

• Solutions to warm-up recursion problems have been posted.

• Midterm is next **Tuesday, May 7 from 7PM - 10PM**.
  • Location TBA.
  • More details next time.
  • Please email Dawson ASAP if you have a conflict with the exam time.
Fundamental Question:

How can we compare solutions to problems?
One Idea: Runtime
Why Runtime Isn't a Good Metric

- Fluctuates between computer to computer and from run to run.
- Fluctuates based on inputs.
- Doesn't predict behavior for larger inputs.
 bool linearSearch(string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}

Work Done: At most $k_0n + k_1$
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:
  - $4n + 4 = O(n)$
  - $137n + 271 = O(n)$
  - $n^2 + 3n + 4 = O(n^2)$
  - $2^n + n^3 = O(2^n)$
Algorithmic Analysis with Big-O
Algorithmic Analysis with Big-O

def double average(Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
double average(Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
double average(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(string& str, char ch) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] == ch) {
            return true;
        }
    }
    return false;
}
```

return false;
A More Interesting Example

```cpp
bool linearSearch(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(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

```c
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 Three

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

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
Once More with Logarithms

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
### Comparison of Runtimes

(1 operation = 1 microsecond)

<table>
<thead>
<tr>
<th>Size</th>
<th>1</th>
<th>lg n</th>
<th>n</th>
<th>n log n</th>
<th>n^2</th>
<th>n^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1μs</td>
<td>7μs</td>
<td>100μs</td>
<td>0.7ms</td>
<td>10ms</td>
<td>&lt;1min</td>
</tr>
<tr>
<td>200</td>
<td>1μs</td>
<td>8μs</td>
<td>200μs</td>
<td>1.5ms</td>
<td>40ms</td>
<td>&lt;1min</td>
</tr>
<tr>
<td>300</td>
<td>1μs</td>
<td>8μs</td>
<td>300μs</td>
<td>2.5ms</td>
<td>90ms</td>
<td>1min</td>
</tr>
<tr>
<td>400</td>
<td>1μs</td>
<td>9μs</td>
<td>400μs</td>
<td>3.5ms</td>
<td>160ms</td>
<td>2min</td>
</tr>
<tr>
<td>500</td>
<td>1μs</td>
<td>9μs</td>
<td>500μs</td>
<td>4.5ms</td>
<td>250ms</td>
<td>4min</td>
</tr>
<tr>
<td>600</td>
<td>1μs</td>
<td>9μs</td>
<td>600μs</td>
<td>5.5ms</td>
<td>360ms</td>
<td>6min</td>
</tr>
<tr>
<td>700</td>
<td>1μs</td>
<td>9μs</td>
<td>700μs</td>
<td>6.6ms</td>
<td>490ms</td>
<td>9min</td>
</tr>
<tr>
<td>800</td>
<td>1μs</td>
<td>10μs</td>
<td>800μs</td>
<td>7.7ms</td>
<td>640ms</td>
<td>12min</td>
</tr>
<tr>
<td>900</td>
<td>1μs</td>
<td>10μs</td>
<td>900μs</td>
<td>8.8ms</td>
<td>810ms</td>
<td>17min</td>
</tr>
<tr>
<td>1000</td>
<td>1μs</td>
<td>10μs</td>
<td>1000μs</td>
<td>10ms</td>
<td>1000ms</td>
<td>22min</td>
</tr>
<tr>
<td>1100</td>
<td>1μs</td>
<td>10μs</td>
<td>1100μs</td>
<td>11ms</td>
<td>1200ms</td>
<td>29min</td>
</tr>
<tr>
<td>1200</td>
<td>1μs</td>
<td>10μs</td>
<td>1200μs</td>
<td>12ms</td>
<td>1400ms</td>
<td>37min</td>
</tr>
<tr>
<td>1300</td>
<td>1μs</td>
<td>10μs</td>
<td>1300μs</td>
<td>13ms</td>
<td>1700ms</td>
<td>45min</td>
</tr>
<tr>
<td>1400</td>
<td>1μs</td>
<td>10μs</td>
<td>1400μs</td>
<td>15ms</td>
<td>2000ms</td>
<td>56min</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.
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**

4 1 2 7 6
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**

```
1  2  4  7  6
```
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**

```
1  2  4  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**

1, 2, 4, 7, 6
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.)
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}

int indexOfSmallest(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 $n + (n - 1) + (n - 2) + \ldots + 1$. 

\[ \text{Work} = n + (n - 1) + (n - 2) + \ldots + 1. \]
\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
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 \( O(n^2) \).
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

• Selection sort has runtime $O(n^2)$ in the worst case.

• How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
Notes on Selection Sort

- Selection sort has runtime $O(n^2)$ in the worst case.
- How about the best case?
- Also $O(n^2)$
- Selection sort always takes $O(n^2)$ time.
- Notation: Selection sort is $\Theta(n^2)$. 
Thinking About $O(n^2)$
Thinking About $O(n^2)$

| 14 | 6  | 3  | 9  | 7  | 16 | 2  | 15 |
Thinking About $O(n^2)$
Thinking About $O(n^2)$
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?