Goals

- Why the choice of algorithms is so critical when dealing with large inputs
- Basic mathematical background
- Review of Recursion
- Review of basic C++ concepts
Selection Problem
Selection Problem

- Find the $k^{th}$-largest number from a group of $N$ numbers
  - How would you solve this?
Selection Problem

- Find the $k^{th}$-largest number from a group of $N$ numbers
  - How would you solve this?

- **Algorithm 1:**
  - Sort the $N$ numbers and Pick the $k^{th}$ one
Selection Problem

- Find the $k^{th}$-largest number from a group of $N$ numbers
  - How would you solve this?

Algorithm 1:
- Sort the $N$ numbers and Pick the $k^{th}$ one
  - Easy to implement, but
    - Sorting requires many comparisons
    - Much work comparing elements that have no chance of being in position $k$
Selection Problem

- Find the $k^{th}$-largest number from a group of $N$ numbers
  - How would you solve this?

**Algorithm 1:**
- Sort the $N$ numbers and Pick the $k^{th}$ one
- Easy to implement, but
  - Sorting requires many comparisons
  - Much work comparing elements that have no chance of being in position $k$
- To sort $N$ elements, need $N \log_2(N)$ comparisons/swaps in general
- Unsorted database with 10,000,000 elements and 1,000 swaps/sec will result in 2.8 hours to finish task
Another strategy
Another strategy

- Algorithm 2: (Better)
  - Sort first $K$ elements in the array.
  - Insert elements $(K+1)$ to $N$, discarding the smallest element each time.
  - Then pick the $k^{th}$ element
Another strategy

Algorithm 2: (Better)
- Sort first $K$ elements in the array.
- Insert elements $(K+1)$ to $N$, discarding the smallest element each time.
- Then pick the $k^{th}$ element

$N = 10,000,000$ and $K = 5,000,000$?
Another strategy

- Algorithm 2: (Better)
  - Sort first $K$ elements in the array.
  - Insert elements $(K+1)$ to $N$, discarding the smallest element each time.
  - Then pick the $k^{th}$ element

- $N = 10,000,000$ and $K = 5,000,000$?

- If $K$ is about as large as $N$, little improvement
- A better algorithm can solve this in a second!
Word Puzzle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>h</td>
<td>i</td>
<td>s</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>a</td>
<td>t</td>
<td>s</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
<td>a</td>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>g</td>
<td>d</td>
<td>t</td>
</tr>
</tbody>
</table>

- **Algorithm 2:**
  - For each combination of letters
    - Check if the word is in the word-list

- What if the word-list is the entire dictionary?
Word Puzzle

- Find all words, given a 2-D array and a word-list.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>this</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>wats</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>oahg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>fgdt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Algorithm 2:**
  - For each combination of letters
    - Check if the word is in the word-list

- What if the word-list is the entire dictionary?
Word Puzzle

- Find all words, given a 2-D array and a word-list.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>h</td>
<td>i</td>
<td>s</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>a</td>
<td>t</td>
<td>s</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
<td>a</td>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>g</td>
<td>d</td>
<td>t</td>
</tr>
</tbody>
</table>

- **Algorithm 1:**
  - For each word in the word list,
    - compare each combination of letters in the puzzle

- **Algorithm 2:**
  - For each combination of letters
    - Check if the word is in the word-list

- What if the word-list is the entire dictionary?
Math Review: Exponents

- $X^A \cdot X^B = X^{A+B}$
- $X^A / X^B = X^{A-B}$
- $(X^A)^B = X^{A\cdot B}$
- $X^A + X^A = 2X^A \neq X^{2\cdot A}$
- $2^N + 2^N = 2 \cdot 2^N = 2^{N+1}$
- $1+2+4+\ldots+2^N = 2^{N+1} - 1$
Logarithms

- All logarithms are to the base 2, unless otherwise specified.

- Definition:
  - $\log_x A = B$ is equivalent to $x^B = A$
Properties of logarithms
Properties of logarithms

Base-change:

\( \log_A B = \frac{\log_C B}{\log_C A} \); \( A, B, C > 0, A \neq 1 \)
Properties of logarithms

- Base-change:
  \[ \log_A B = \frac{\log_C B}{\log_C A} \quad ; \quad A, B, C > 0, \ A \neq 1 \]

- Proof: Let \( Z = \log_C A \) and \( Y = \log_A B \)
Properties of logarithms

- **Base-change:**
  \[ \log_A B = \frac{\log_C B}{\log_C A} \quad ; \quad A, B, C > 0, \ A \neq 1 \]

- **Proof:** Let \( Z = \log_C A \) and \( Y = \log_A B \)
  Then \( A = C^Z \) and \( B = A^Y \)
Properties of logarithms

- **Base-change:**
  \[ \log_A B = \frac{\log_C B}{\log_C A} \quad ; \quad A, B, C > 0, \ A \neq 1 \]

- **Proof:** Let \( Z = \log_C A \) and \( Y = \log_A B \)
  - Then \( A = C^Z \) and \( B = A^Y \)
  - Implies \( B = (C^Z)^Y = C^{Z \cdot Y} \)
Properties of logarithms

- **Base-change:**
  \[ \log_A B = \frac{\log_C B}{\log_C A} ; \quad A, B, C > 0, \ A \neq 1 \]

- **Proof:** Let \( Z = \log_C A \) and \( Y = \log_A B \)
  - Then \( A = C^Z \) and \( B = A^Y \)
  - Implies \( B = (C^Z)^Y = C^{Z \cdot Y} \)
  - Hence \( \log_C B = Z \cdot Y = \log_C A \cdot \log_A B \)
Logarithms (contd.)

- \( \log(A \cdot B) = \log(A) + \log(B); \ A, B > 0 \)
- **Proof:**
  - Let \( X = \log(A) \) and \( Y = \log(B) \)
  - Hence \( 2^X = A \) and \( 2^Y = B \)
  - Combining the above, \( A \cdot B = 2^X \cdot 2^Y = 2^{X+Y} \)
  - Implies, \( \log(A \cdot B) = X + Y = \log(A) + \log(B) \)
  - Hence proved!
Logarithms (contd.)
Logarithms (contd.)

- More results
Logarithms (contd.)

- More results
  - \( \log(A/B) = \log(A) - \log(B) \)
Logarithms (contd.)

- More results
  - \( \log\left(\frac{A}{B}\right) = \log(A) - \log(B) \)
  - \( \log(AB) = B \log(A) \)
Logarithms (contd.)

- More results
  - $\log(A/B) = \log(A) - \log(B)$
  - $\log(AB) = B \log(A)$
  - $\log(X) < X$ for all $X > 0$
Logarithms (contd.)

More results

- \( \log(A/B) = \log(A) - \log(B) \)
- \( \log(AB) = B \log(A) \)
- \( \log(X) < X \) for all \( X > 0 \)
- \( \log(1) = 0, \log(2) = 1, \log(1,024) = 10 \).
Logarithms (contd.)

- More results
  - $\log(A/B) = \log(A) - \log(B)$
  - $\log(AB) = B \log(A)$
  - $\log(X) < X$ for all $X > 0$
  - $\log(1)=0$, $\log(2)=1$, $\log(1,024)=10$, $\log(1,048,576)=20$
Geometric Series
Geometric Series

\[ \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]
Geometric Series

\[ \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]

\[ \sum_{i=0}^{N} A^i = \]
Geometric Series

\[ \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]

\[ \sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \]

- If \( 0 < A < 1 \), \( \sum_{i=0}^{N} A^i \leq \)
Geometric Series

- \[ \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]

- \[ \sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \]

- If \( 0 < A < 1 \), \[ \sum_{i=0}^{N} A^i \leq \frac{1}{A - 1} \]

- If \( N \to \infty \), \[ \sum_{i=0}^{N} A^i = \frac{1}{A - 1} \]
Arithmetic Series
Arithmetic Series

\[ \sum_{i=0}^{N} i = \]
Arithmetic Series

\[
\sum_{i=0\ldots N} i = \frac{N(N+1)}{2}
\]
Arithmetic Series

\[ \sum_{i=0\ldots N} i = \frac{N(N+1)}{2} \]

How about \[ 2 + 5 + 8 + \ldots + (3k-1) \]?
Modular Arithmetic
Modular Arithmetic

- We say that \( A \) is congruent to \( B \) modulo \( N \)
  - Written as \( A \equiv B \pmod{N} \)
  - If \( N \) divides \( A - B \)
Modular Arithmetic

- We say that \( A \) is congruent to \( B \) modulo \( N \)
  - Written as \( A \equiv B \pmod{N} \)
  - If \( N \) divides \( (A-B) \)

- In other words, the remainder is the same if either \( A \) or \( B \) are divided by \( N \)
  - E.g. \( 81 \equiv 61 \equiv 1 \pmod{10} \)
Modular Arithmetic

- We say that **A is congruent to B modulo N**
  - Written as $A \equiv B \pmod{N}$
  - If $N$ divides $(A-B)$

- In other words, the remainder is the same if either $A$ or $B$ are divided by $N$
  - E.g. $81 \equiv 61 \equiv 1 \pmod{10}$

- Similarly,
  - $A+C \equiv B+C \pmod{N}$ and $AD \equiv BD \pmod{N}$
Proof by Induction

- Given a theorem
- First prove a **base case**
  - Show the theorem is true for some small degenerate values
- Next assume an **inductive hypothesis**
  - Assume the theorem is true for all cases up to some limit $k$
- Then prove that the theorem holds for the next value ($k+1$)
Proof by Induction - example

- Fibonacci Series
  - \( F_0 = 1, F_1 = 1, F_i = F_{i-1} + F_{i-2}, \text{ for } i > 1 \)

- Show that
  - \( F_i < (5/3)^i, \text{ for } i \geq 1 \)

- Base case: \( F_0 = 1 = F_1 < 5/3 \)

- Inductive Hypothesis
  - \( F_i < (5/3)^i, \text{ for } i = 1, 2, \ldots, k \)
Proof by Induction - example (contd.)
Proof by Induction - example (contd.)

Now prove that $F_{k+1} < (5/3)^{k+1}$
Proof by Induction - example (contd.)

- Now prove that $F_{k+1} < (5/3)^{k+1}$
- From definition of Fibonacci Sequence
  $F_{k+1} < F_k + F_{k-1}$
Proof by Induction - example (contd.)

- Now prove that $F_{k+1} < (5/3)^{k+1}$
- From definition of Fibonacci Sequence
  $$F_{k+1} < F_k + F_{k-1}$$
- Using inductive Hypothesis
Proof by Induction - example (contd.)

Now prove that $F_{k+1} < (5/3)^{k+1}$

From definition of Fibonacci Sequence

$F_{k+1} < F_k + F_{k-1}$

Using inductive Hypothesis

$F_{k+1} < (5/3)^k + (5/3)^{k-1}$

$< (5/3)^k + (5/3)^{k-1}$

$< (5/3)^k + (3/5 + (3/5)^2)$

$< (5/3)^k + (24/25)$

$< (5/3)^k + 1$
Proof by Induction - Exercise
Proof by Induction - Exercise

\[ \sum_{i=0..N} i = \]

\[ \sum_{i=0..N} i = \]
Proof by Induction - Exercise

$$\sum_{i=0..N} i = \frac{N(N+1)}{2}$$

$$\sum_{i=0..N} i =$$
Proof by Induction - Exercise

\[ \sum_{i=0}^{N} i = \frac{N(N+1)}{2} \]

\[ \sum_{i=0}^{N} i = \frac{N(N+1)(2N+1)}{6} \]
Other types of proofs
Other types of proofs

- *Proof by counter-example*
Other types of proofs

- Proof by counter-example
  - The statement $F_k < k^2$ in the Fibonacci series is false
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
    - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
    - Then show that some known property would be false as well
Other types of proofs

**Proof by counter-example**
- The statement \( F_k < k^2 \) in the Fibonacci series is false
  - Proof: \( F_{11} = 144 > 11^2 \)

**Proof by Contradiction**
- Initially assume that the theorem is false
  - Then show that some known property would be false as well
- Example: “There are an infinite number of prime numbers”
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
  - Then show that some known property would be false as well
  - Example: “There are an infinite number of prime numbers”
  - Proof: Assume the theorem is false
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
  - Then show that some known property would be false as well
  - Example: “There are an infinite number of prime numbers”
  - Proof: Assume the theorem is false
    - Let $p_1, p_2, ..., p_k$ be all the primes in increasing order
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
  - Then show that some known property would be false as well
  - Example: “There are an infinite number of prime numbers”
  - Proof: Assume the theorem is false
    - Let $p_1, p_2, \ldots, p_k$ be all the primes in increasing order
    - $N = p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$ is $> p_k$ and so it is not a prime.
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
  - Then show that some known property would be false as well
  - Example: “There are an infinite number of prime numbers”
  - Proof: Assume the theorem is false
    - Let $p_1, p_2, \ldots, p_k$ be all the primes in increasing order
    - $N = p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$ is $> p_k$ and so it is not a prime.
    - But it is also not divisible by any of the listed primes, contradicting the factorization of integers into primes.
Other types of proofs

- **Proof by counter-example**
  - The statement $F_k < k^2$ in the Fibonacci series is false
  - Proof: $F_{11} = 144 > 11^2$

- **Proof by Contradiction**
  - Initially assume that the theorem is false
    - Then show that some known property would be false as well
  - Example: “There are an infinite number of prime numbers”
  - Proof: Assume the theorem is false
    - Let $p_1, p_2, \ldots, p_k$ be all the primes in increasing order
    - $N = p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$ is $> p_k$ and so it is not a prime.
    - But it is also not divisible by any of the listed primes, contradicting the factorization of integers into primes.
    - This is a counter-example
Another type of Proof
Another type of Proof

“Proof by intimidation”
Another type of Proof

- “Proof by intimidation”
- Also known as “Proof by Authority”
Another type of Proof

- “Proof by intimidation”

- Also known as “Proof by Authority”

- Proof:
  - Your prof. says the theorem is true 😊
Recursion
Recursion

- Recursive function
Recursion

- Recursive function
  - Function that is defined in terms of itself
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. $f(0)=0$ and $f(x) = 2*f(x-1) + x^2$
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. $f(0) = 0$ and $f(x) = 2f(x-1) + x^2$
  - Implementation in C or C++
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. $f(0) = 0$ and $f(x) = 2f(x-1) + x^2$
  - Implementation in C or C++
    1. `int f(int x)`
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. $f(0) = 0$ and $f(x) = 2f(x-1) + x^2$
  - Implementation in C or C++
    1. `int f( int x )`
    2. `{`
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. \( f(0)=0 \) and \( f(x) = 2*f(x-1) + x^2 \)
  - Implementation in C or C++
  1. int f( int x )
  2. {
  3. if( x == 0) \( \rightarrow \) Base Case
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. \( f(0) = 0 \) and \( f(x) = 2f(x-1) + x^2 \)
  - Implementation in C or C++
  1. int f(int x)
  2. {
  3.     if (x == 0) \( \rightarrow \) **Base Case**
  4.     return 0;
  }
Recursion

- **Recursive function**
  - Function that is defined in terms of itself
  - E.g. \( f(0)=0 \) and \( f(x) = 2*f(x-1) + x^2 \)
  - Implementation in C or C++
    
    1. `int f( int x )`
    2. `{`
    3. `if( x == 0 ) Base Case`
    4. `return 0;`
    5. `else Recursive Call`
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. $f(0)=0$ and $f(x) = 2*f(x-1) + x^2$
  - Implementation in C or C++
    1. int f( int x )
    2. {
    3.     if( x == 0 ) /*Base Case*/
    4.         return 0;
    5.     else /*Recursive Call*/
    6.         return 2 * f( x - 1 ) + x * x;
Recursion

- Recursive function
  - Function that is defined in terms of itself
  - E.g. \( f(0) = 0 \) and \( f(x) = 2f(x-1) + x^2 \)
  - Implementation in C or C++
    1. `int f( int x )`
    2. {
    3.     if( x == 0 ) \( \leftarrow \text{Base Case} \)
    4.     return 0;
    5.     else \( \leftarrow \text{Recursive Call} \)
    6.     return 2 * f( x - 1 ) + x * x;
    7. }

Recursion Examples

1. Evaluating $f(x) = 2*f(x-1) + x*x$
   - $f(4) = 2*f(3) + 4*4$
     - $f(3) = 2*f(2) + 3*3$
       - $f(2) = 2*f(1) + 2*2$
         - $f(1) = 2*f(0) + 1*1$
         - $f(0) = 0$
         - $f(1) = 2*0 + 1*1 = 1$
       - $f(2) = 2*1 + 2*2 = 6$
     - $f(3) = 2*6 + 3*3 = 21$
   - $f(4) = 2*21 + 4*4 = 58$

2. Looking up meaning of a word in dictionary
   - Where is the recursion?
   - What is the base case?
Ensuring recursive programs terminate
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?
- Another non-terminating recursive function
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?
- Another non-terminating recursive function
  1. int bad(int n)
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?
- Another non-terminating recursive function
- int bad(int n)
- 

Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?
- Another non-terminating recursive function

1. int bad(int n)
2. {
3.     if(n == 0)
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?

- Another non-terminating recursive function

```c
1. int bad(int n)
2. {
3.     if(n == 0)
4.         return 0;
```
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?

- Another non-terminating recursive function

```c
int bad(int n)
{
    if(n == 0)
        return 0;
    else
```
Ensuring recursive programs terminate

• What happens if you make the call $f(-1)$ in the implementation in slide 19?

• Another non-terminating recursive function

```
1. int bad(int n)
2. {
3.     if(n == 0)
4.         return 0;
5.     else
6.         return bad(n/3 + 1) + n - 1;
```
Ensuring recursive programs terminate

- What happens if you make the call \( f(-1) \) in the implementation in slide 19?

- Another non-terminating recursive function

```c
1. int bad(int n)
2. {
3.     if(n == 0)
4.         return 0;
5.     else
6.         return bad(n/3 + 1) + n - 1;
7. }
```
Ensuring recursive programs terminate

- What happens if you make the call $f(-1)$ in the implementation in slide 19?

- Another non-terminating recursive function

```c
1. int bad(int n)
2. {
3.     if(n == 0) {...
4.         return 0;
5.     else
6.         return bad(n/3 + 1) + n - 1;
7. }
```

What happens on call to bad(1)?
And bad(2)?
And bad(3)?
Ensuring recursive programs terminate (contd.)
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
  - Define a base case
    - This is what terminates recursion
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
  - Define a base case
    - This is what terminates recursion
  - Making progress:
    - At each step make progress towards the base case
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
  - Define a base case
    - This is what terminates recursion
  - Making progress:
    - At each step make progress towards the base case

- More rules for writing clean recursive programs
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
  - Define a base case
    - This is what terminates recursion
  - Making progress:
    - At each step make progress towards the base case

- More rules for writing clean recursive programs
  - Design Rule
    - Assume that all recursive calls work.
Ensuring recursive programs terminate (contd.)

- Two rules for writing correct recursive programs
  - Define a base case
    - This is what terminates recursion
  - Making progress:
    - At each step make progress towards the base case

- More rules for writing clean recursive programs
  - Design Rule
    - Assume that all recursive calls work.
  - Compound Interest Rule:
    - Never duplicate work by making same recursive call twice!
      - E.g. Fibonacci series $f(i) = f(i-1) + f(i-2)$. Why??
For next week...
For next week...

- Reading Assignment
  - Sections 1.5, 1.6 and 1.7
For next week...

- **Reading Assignment**
  - Sections 1.5, 1.6 and 1.7

- **Exercise problems**
  - Problems 1.1 to 1.12 on pages 39 and 40
  - Not for submission/grading.