Outline

1 Digital Systems
   flip-flops
   registers

2 Intrinsic Operations
   sorting words
   values of numbers given in words

3 queues and stacks
   using Python lists
   towers of Hanoi

4 Summary + Assignments

MCS 260 Lecture 10
Introduction to Computer Science
Jan Verschelde, 17 September 2008
flip-flops and registers
queues and stacks

1. Digital Systems
   - flip-flops
   - registers

2. Intrinsic Operations
   - sorting words
   - values of numbers given in words

3. queues and stacks
   - using Python lists
   - towers of Hanoi

4. Summary + Assignments
Flip-Flops (and latches) are the simplest circuits to store one bit.

D: input line
T: clock line
Q: output line

Its behavior is as follows:

1. When one arrives on the clock line, the output line is set to the value present on the input line.
2. The value at the output line is stored at the flip-flop, until a new one arrives on the clock line.
Flip-Flops (and latches) are the simplest circuits to store one bit.

Its behavior is as follows:

1. When one arrives on the clock line, the output line is set to the value present on the input line.
2. The value at the output line is stored at the flip-flop, until a new one arrives on the clock line.
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

Exercise: represent $\text{NOT } (x \text{ NOR } (\text{NOT } y))$. 

![Logic Gates Diagram]
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

**NOT**

![NOT Truth Table]

**NAND**

![NAND Truth Table]

**NOR**

![NOR Truth Table]

Exercise: represent $\text{NOT} \ ( x \ \text{NOR} \ ( \text{NOT} \ y ))$. 
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

NOT

\[
\begin{array}{c}
0 \rightarrow 1 \\
1 \rightarrow 0
\end{array}
\]

NAND

\[
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0
\end{array}
\]

NOR

\[
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0
\end{array}
\]

Exercise: represent \( \text{NOT} \ ( x \ \text{NOR} \ ( \text{NOT} \ y ) ) \).
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

\[
\begin{array}{c|c}
\text{NOT} & 0 & 1 \\
\hline
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{array}
\quad
\begin{array}{c|c}
\text{NAND} & 0 & 1 \\
\hline
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{array}
\quad
\begin{array}{c|c}
\text{NOR} & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 0 & 1 \\
1 & 0 & 0 \\
\end{array}
\]

Exercise: represent $\text{NOT} ( x \text{ NOR} ( \text{NOT} y ))$. 

Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

Exercise: represent \( \text{NOT } (x \text{ NOR } (\text{NOT } y)) \).
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

\[
\begin{array}{c|c}
\text{NOT} & \text{NAND} & \text{NOR} \\
\hline
0 & 0 & 0 \ 
1 & 1 & 1 \\
\end{array}
\]

Exercise: represent \( \text{NOT} \left( x \ \text{NOR} \left( \text{NOT} \ y \right) \right) \).
Logic Gates

A flip-flop is realized with NOT, NAND, and NOR gates:

Exercise: represent $\text{NOT} \left( x \text{ NOR } ( \text{NOT} y ) \right)$. 
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Realization of a Flip-Flop
one NOT, two NANDs, and two NORs

We simulate the *latching* of a 1 at D to Q, with 1 at Q.

Exercise: verify the effect is the same for 0 at Q.
Behavior in Time
how latching of bits works

The evolution in time is

At $t_1$: 0 at D, 0 at T, and 0 at Q
At $t_2$: 1 at D, but 0 at T and nothing happens
At $t_3$: 1 at T $\Rightarrow$ 1 at D copied to 1 at Q
At $t_4$: 0 at T and nothing happens
At $t_5$: 0 at D, but 0 at T and nothing happens
At $t_6$: 1 at T $\Rightarrow$ 0 at D copied to 0 at Q
Behavior in Time
how latching of bits works

The evolution in time is

At $t_1$: 0 at D, 0 at T, and 0 at Q

At $t_2$: 1 at D, but 0 at T and nothing happens

At $t_3$: 1 at T $\Rightarrow$ 1 at D copied to 1 at Q

At $t_4$: 0 at T and nothing happens

At $t_5$: 0 at D, but 0 at T and nothing happens

At $t_6$: 1 at T $\Rightarrow$ 0 at D copied to 0 at Q
The evolution in time is

At $t_1$: 0 at D, 0 at T, and 0 at Q
At $t_2$: 1 at D, but 0 at T and nothing happens
At $t_3$: 1 at T $\Rightarrow$ 1 at D copied to 1 at Q
At $t_4$: 0 at T and nothing happens
At $t_5$: 0 at D, but 0 at T and nothing happens
At $t_6$: 1 at T $\Rightarrow$ 0 at D copied to 0 at Q
Behavior in Time
how latching of bits works

The evolution in time is

At $t_1$: 0 at D, 0 at T, and 0 at Q
At $t_2$: 1 at D, but 0 at T and nothing happens
At $t_3$: 1 at T $\Rightarrow$ 1 at D copied to 1 at Q
At $t_4$: 0 at T and nothing happens
At $t_5$: 0 at D, but 0 at T and nothing happens
At $t_6$: 1 at T $\Rightarrow$ 0 at D copied to 0 at Q
Behavior in Time
how latching of bits works

The evolution in time is

At \(t_1\): 0 at D, 0 at T, and 0 at Q
At \(t_2\): 1 at D, but 0 at T and nothing happens
At \(t_3\): 1 at T \(\Rightarrow\) 1 at D copied to 1 at Q
At \(t_4\): 0 at T and nothing happens
At \(t_5\): 0 at D, but 0 at T and nothing happens
At \(t_6\): 1 at T \(\Rightarrow\) 0 at D copied to 0 at Q
Behavior in Time

how latching of bits works

The evolution in time is

At $t_1$: 0 at D, 0 at T, and 0 at Q
At $t_2$: 1 at D, but 0 at T and nothing happens
At $t_3$: 1 at T $\Rightarrow$ 1 at D copied to 1 at Q
At $t_4$: 0 at T and nothing happens
At $t_5$: 0 at D, but 0 at T and nothing happens
At $t_6$: 1 at T $\Rightarrow$ 0 at D copied to 0 at Q
flip-flops and registers
queues and stacks

1. Digital Systems
   flip-flops
   registers

2. Intrinsic Operations
   sorting words
   values of numbers given in words

3. queues and stacks
   using Python lists
   towers of Hanoi

4. Summary + Assignments
An 8-bit register is realized with 8 flip-flops.

eight input lines and eight clock lines

eight output lines
Copy Bits using 2 AND gates

We want to copy a bit from one latch to the other. The bit is at the output line of the first latch, at Q1 and has to get to the output line of the second latch, at Q2.

The copy is activated by the control signal. For synchronization, another signal from the system clock copies the bit from the input line at D2 to Q2.
Copy Bits
using 2 AND gates

We want to copy a bit from one latch to the other. The bit is at the output line of the first latch, at Q1 and has to get to the output line of the second latch, at Q2.

![Diagram of copy bits](https://example.com/diagram.png)

The copy is activated by the control signal. For synchronization, another signal from the system clock copies the bit from the input line at D2 to Q2.
We want to copy a bit from one latch to the other. The bit is at the output line of the first latch, at Q1 and has to get to the output line of the second latch, at Q2.

The copy is activated by the control signal. For synchronization, another signal from the system clock copies the bit from the input line at D2 to Q2.
Copy Bits using 2 AND gates

We want to copy a bit from one latch to the other. The bit is at the output line of the first latch, at Q1 and has to get to the output line of the second latch, at Q2.

![Diagram of copy bits using 2 AND gates]

The copy is activated by the control signal. For synchronization, another signal from the system clock copies the bit from the input line at D2 to Q2.
flip-flops and registers
queues and stacks

1 Digital Systems
   flip-flops
   registers

2 Intrinsic Operations
   sorting words
   values of numbers given in words

3 queues and stacks
   using Python lists
towers of Hanoi

4 Summary + Assignments
Input/Output problem statement:

**Input:** string with words separated by spaces.  
**Output:** string with alphabetically sorted words.

Running `sortwords.py` at the command prompt $:$

$ python sortwords.py  
Give words : this is my sentence  
Sorted words : is my sentence this  
$
Input/Output problem statement:

**Input**: string with words separated by spaces.

**Output**: string with alphabetically sorted words.

Running `sortwords.py` at the command prompt $:$

```
$ python sortwords.py
Give words : this is my sentence
Sorted words : is my sentence this
$ 
```
### Sorting Words

in the Python shell

```python
>>> s = 'this is my sentence'
>>> L = s.split()
>>> L
['this', 'is', 'my', 'sentence']

Methods are different from functions:

```python
>>> max(L)
'this'
>>> min(L)
'is'
```
Sorting Words
in the Python shell

>>> s = 'this is my sentence'
>>> L = s.split()
>>> L
['this', 'is', 'my', 'sentence']

Methods are different from functions:
>>> max(L)
'this'
>>> min(L)
'is'

>>> L.sort()
>>> L
['is', 'my', 'sentence', 'this']

>>> ''.join(L)
'is my sentence this'
Sorting Words
in the Python shell

>>> s = 'this is my sentence'
>>> L = s.split()
>>> L
['this', 'is', 'my', 'sentence']

Methods are different from functions:

>>> max(L)
'this'
>>> min(L)
'is'

>>> L.sort()
>>> L
['is', 'my', 'sentence', 'this']

>>> ''.join(L)
'is my sentence this'
Some Intrinsic Operations
on strings and lists: split, join, and sort

To sort a list \( L \):

\[
>>> L.sort()
\]

A compare function can be given as argument in \((\text{ })\).

To split a string \( s \) into a list \( L \) of strings:

\[
>>> L = s.split()
\]

The default separator is a space, another separator (like a comma or colon) can be given as string in \((\text{ })\).

To join a list of strings \( L \) into one string \( s \):

\[
>>> s = ' '.join(L)
\]

The separator used between the strings in \( L \) is the string to which the method is applied to.
Some Intrinsic Operations
on strings and lists: split, join, and sort

To sort a list $L$:

```python
>>> L.sort()
```

A compare function can be given as argument in (`) .

To split a string $s$ into a list $L$ of strings:

```python
>>> L = s.split()
```

The default separator is a space, another separator (like a comma or colon) can be given as string in (`) .

To join a list of strings $L$ into one string $s$:

```python
>>> s = ' '.join(L)
```

The separator used between the strings in $L$ is the string to which the method is applied to.
Some Intrinsic Operations
on strings and lists: split, join, and sort

To sort a list \( L \):

```python
>>> L.sort()
```

A compare function can be given as argument in \((\)\).

To split a string \( s \) into a list \( L \) of strings:

```python
>>> L = s.split()
```

The default separator is a space, another separator (like a comma or colon) can be given as string in \((\)\).

To join a list of strings \( L \) into one string \( s \):

```python
>>> s = ' '.join(L)
```

The separator used between the strings in \( L \) is the string to which the method is applied to.
The program `sortwords.py`:

```python
# L-10 MCS 260 Wed 17 Sep 2008 sort words
#
# This program shows intrinsic operations on strings and lists, to sort words,
# given as a raw input string by the user.
#
s = raw_input('Give words : ');
L = s.split()  # spaces separate the words
L.sort()       # sort list alphabetically
s = ' '.join(L) # join the sorted list
print 'Sorted words :', s
```
The program `sortwords.py`:

```python
# L-10 MCS 260 Wed 17 Sep 2008 sort words
#
# This program shows intrinsic operations
# on strings and lists, to sort words,
# given as a raw input string by the user.
#
s = raw_input('Give words : ');
L = s.split() # spaces separate the words
L.sort() # sort list alphabetically
s = ' '.join(L) # join the sorted list
print 'Sorted words :', s
```
flip-flops and registers
queues and stacks

1. Digital Systems
   - flip-flops
   - registers

2. Intrinsic Operations
   - sorting words
   - values of numbers given in words

3. queues and stacks
   - using Python lists
e   - towers of Hanoi

4. Summary + Assignments
Value of a Number

given in words, problem statement

Problem Statement:

**Input:** string with *at most* two words, separated by *exactly one* space.

**Output:** value of the number represented by the string.

Running the Python code `write_values.py`:

```
$ python write_values.py
give a number in words : forty seven
the value of forty seven is 47
```

Note: reverse of `write_numbers.py` of Lecture 9.
Value of a Number

given in words, problem statement

Problem Statement:

**Input:** string with *at most* two words, separated by *exactly one* space.

**Output:** value of the number represented by the string.

Running the Python code `write_values.py`:

```
$ python write_values.py
give a number in words : forty seven
the value of forty seven is 47
```

Note: reverse of `write_numbers.py` of Lecture 9.
Value of a Number

given in words, problem statement

Problem Statement:

Input: string with at most two words, separated by exactly one space.

Output: value of the number represented by the string.

Running the Python code write_values.py:

$ python write_values.py

give a number in words : forty seven
the value of forty seven is 47

Note: reverse of write_numbers.py of Lecture 9.
The Dictionary
to translate words into values

The keys are strings representing words, the values are the corresponding numbers.

```python
d = {
    'zero': 0, 'one': 1, 'two': 2, 'three': 3, 
    'four': 4, 'five': 5, 'six': 6, 'seven': 7, 
    'eight': 8, 'nine': 9, 'ten': 10, 
    'eleven': 11, 'twelve': 12, 'thirteen': 13, 
    'fourteen': 14, 'fifteen': 15, 'sixteen': 16, 
    'seventeen': 17, 'nineteen': 19, 'twenty': 20, 
    'thirty': 30, 'forty': 40, 'fifty': 50, 
    'sixty': 60, 'seventy': 70, 'eighty': 80, 
    'ninety': 90, 'hundred': 100
}
```

The dictionary solves 28 cases.
Ingredients of the Code

```python
def has_key(s):
    return s in ['zero', 'one', ...

>>> d = { 'zero':0, 'one':1, ...
... }
>>> s = 'forty seven'
>>> has_key(s)
False
>>> L = s.split(' ')
>>> L
['forty', 'seven']
>>> has_key(L[0])
True
>>> d[L[0]]
40
>>> d[L[1]]
7
>>> v = d[L[0]] + d[L[1]]
>>> v
47
```
Ingredients of the Code

>>> d = { 'zero':0, 'one':1, ... 
    ... }
>>> s = 'forty seven'
>>> d.has_key(s)
False
>>> L = s.split()
>>> L
['forty', 'seven']
>>> d.has_key(L[0])
True
>>> d[L[0]]
40
>>> d[L[1]]
7
>>> v = d[L[0]] + d[L[1]]
>>> v
47
Ingredients of the Code

```python
>>> d = { 'zero':0, 'one':1, ...
     ... }
>>> s = 'forty seven'
>>> d.has_key(s)
False
>>> L = s.split(' ')
>>> L
['forty', 'seven']
>>> d.has_key(L[0])
True
>>> d[L[0]]
40
>>> d[L[1]]
7
>>> v = d[L[0]] + d[L[1]]
>>> v
47
```
Ingredients of the Code

```python
>>> d = { 'zero':0, 'one':1, ...
     ... }
>>> s = 'forty seven'
>>> d.has_key(s)
False
>>> L = s.split(' ')
>>> L
[ 'forty', 'seven' ]
>>> d.has_key(L[0])
True
>>> d[L[0]]
40
>>> d[L[1]]
7
>>> v = d[L[0]] + d[L[1]]
>>> v
47
```
s = raw_input('give a number in words : ')
outcome = 'the value of ' + s + ' is '
if d.has_key(s):
    outcome += str(d[s])
else:
    L = s.split(' ')
    outcome += str(d[L[0]] + d[L[1]])
print outcome

Alternative: do first L = s.split(' ')
and then test on len(L) == 1 to determine outcome.
Python code continued for write_values.py

```python
s = raw_input('give a number in words : ')
outcome = 'the value of ' + s + ' is '
if d.has_key(s):
    outcome += str(d[s])
else:
    L = s.split(' ')
    outcome += str(d[L[0]] + d[L[1]])
print outcome

Alternative: do first L = s.split(' ') and then test on len(L) == 1 to determine outcome.
```
flip-flops and registers
queues and stacks

1. Digital Systems
   flip-flops
   registers

2. Intrinsic Operations
   sorting words
   values of numbers given in words

3. queues and stacks
   using Python lists
   towers of Hanoi

4. Summary + Assignments
Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a *queue*
think of a normal waiting list

**FILO:** First in Last Out, a *stack*
think of a pile of papers on a desk

Intrinsic operations on a list $L$:

- $L$.append(<item>) appends <item> to $L$
- <item> = $L$.pop() removes last item added to $L$
- <item> = $L$.pop(0) removes first item added to $L$
- $L$.insert(0,<item>) inserts <item> to front of $L$

All these operations modify $L$!

How to select from $L$, *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks
use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO**: First In First Out, a *queue*
think of a normal waiting list

**FILO**: First in Last Out, a *stack*
think of a pile of papers on a desk

Intrinsic operations on a list $L$:

- $L$.append(<item>)  appends <item> to $L$
- <item> = $L$.pop()  removes last item added to $L$
- <item> = $L$.pop(0) removes first item added to $L$
- $L$.insert(0,<item>)  inserts <item> to front of $L$

All these operations modify $L$!

How to select from $L$, *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks
use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a *queue*
think of a normal waiting list

**FILO:** First in Last Out, a *stack*
think of a pile of papers on a desk

Intrinsic operations on a list \( L \):

- \( L.append(<item>) \) **appends** \(<item>\) to \( L \)
- \(<item> = L.pop()\) **removes** last item added to \( L \)
- \(<item> = L.pop(0)\) **removes** first item added to \( L \)
- \( L.insert(0,<item>)\) **inserts** \(<item>\) to front of \( L \)

All these operations modify \( L \)!

How to select from \( L \), *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks
use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a queue
think of a normal waiting list

**FILO:** First in Last Out, a stack
think of a pile of papers on a desk

Intrinsic operations on a list $L$:

- `L.append(<item>)` appends `<item>` to $L$
- `<item> = L.pop()` removes last item added to $L$
- `<item> = L.pop(0)` removes first item added to $L$
- `L.insert(0, <item>)` inserts `<item>` to front of $L$

All these operations modify $L$!

How to select from $L$, *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks

use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a *queue*
think of a normal waiting list

**FILO:** First In Last Out, a *stack*
think of a pile of papers on a desk

Intrinsic operations on a list $L$:

- $L$.append(<item>)  
  appends <item> to $L$
- <item> = L.pop()  
  removes last item added to $L$
- <item> = L.pop(0)  
  removes first item added to $L$
- L.insert(0,<item>)  
  inserts <item> to front of $L$

All these operations modify $L$!

How to select from $L$, *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks
use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a queue
think of a normal waiting list

**FILO:** First in Last Out, a stack
think of a pile of papers on a desk

Intrinsic operations on a list $L$:

- $L$.append($<$item$>$) \hspace{1cm} appends $<$item$>$ to $L$
- $<$item$>$ = $L$.pop() \hspace{1cm} removes last item added to $L$
- $<$item$>$ = $L$.pop(0) \hspace{1cm} removes first item added to $L$
- $L$.insert(0,$<$item$>$) \hspace{1cm} inserts $<$item$>$ to front of $L$

All these operations modify $L$!

How to select from $L$, *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Queues and Stacks
use of Python lists

Two protocols to retrieve elements sequentially:

**FIFO:** First In First Out, a queue
think of a normal waiting list

**FILO:** First in Last Out, a stack
think of a pile of papers on a desk

Intrinsic operations on a list \( L \):

- \( L.append(<item>) \) appends \(<item>\) to \( L \)
- \(<item> = L.pop() \) removes last item added to \( L \)
- \(<item> = L.pop(0) \) removes first item added to \( L \)
- \( L.insert(0,<item>) \) inserts \(<item>\) to front of \( L \)

All these operations modify \( L \)!

How to select from \( L \), *without modifications*?

```python
>>> L[0]
>>> L[len(L)-1]
```
Lists as Queues and Stacks

a session in the Python shell

```python
>>> L = 'these are some words'.split()
>>> L
['these', 'are', 'some', 'words']
>>> L.append('and')
>>> L
['these', 'are', 'some', 'words', 'and']
>>> last = L.pop()
>>> last
'and'
>>> first = L.pop(0)
>>> first
'these'
>>> L
['are', 'some', 'words']
```
Lists as Queues and Stacks

a session in the Python shell

```python
>>> L = 'these are some words'.split()
>>> L
['these', 'are', 'some', 'words']
>>> L.append('and')
>>> L
['these', 'are', 'some', 'words', 'and']
>>> last = L.pop()
>>> last
'and'
>>> first = L.pop(0)
>>> first
'these'
>>> L
['are', 'some', 'words']
```
Lists as Queues and Stacks

a session in the Python shell

```python
>>> L = 'these are some words'.split()
>>> L
['these', 'are', 'some', 'words']
>>> L.append('and')
>>> L
['these', 'are', 'some', 'words', 'and']
>>> last = L.pop()
>>> last
'and'
>>> first = L.pop(0)
>>> first
'these'
```
Lists as Queues and Stacks

a session in the Python shell

>>> L = 'these are some words'.split()
>>> L
['these', 'are', 'some', 'words']

>>> L.append('and')
>>> L
['these', 'are', 'some', 'words', 'and']

>>> last = L.pop()
>>> last
'and'

>>> first = L.pop(0)
>>> first
'these'

>>> L
['are', 'some', 'words']

flip-flops and registers
queue and stacks

1. Digital Systems
   flip-flops
   registers

2. Intrinsic Operations
   sorting words
   values of numbers given in words

3. queues and stacks
   using Python lists
   towers of Hanoi

4. Summary + Assignments
The Towers of Hanoi
a mathematical puzzle

Input: a stack of disks, all of varying size, no larger disk sits above a smaller disk, and two other empty stacks.

Task: move the disks from the first stack to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third disk as buffer.

How many moves does it take?
The Towers of Hanoi
a mathematical puzzle

**Input:** a stack of disks, all of varying size, no larger disk sits above a smaller disk, and two other empty stacks.

**Task:** move the disks from the first stack to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third disk as buffer.

How many moves does it take?
The Towers of Hanoi
a mathematical puzzle

Input: a stack of disks, all of varying size, no larger disk sits above a smaller disk, and two other empty stacks.

Task: move the disks from the first stack to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third disk as buffer.

How many moves does it take?
The Towers of Hanoi
a mathematical puzzle

Input: a stack of disks, all of varying size, no larger disk sits above a smaller disk, and two other empty stacks.

Task: move the disks from the first stack to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third disk as buffer.

How many moves does it take?
Towers of Hanoi
for three disks in Python

- three lists A, B, C as stacks
- disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []
move 4 : A = [] B = [3] C = [1, 2]
move 7 : A = [] B = [1, 2, 3] C = []
Towers of Hanoi
for three disks in Python

- three lists $A, B, C$ as stacks
- disks are represented as numbers $1 < 2 < 3$

Towers of Hanoi with 3 disks
initially: $A = [1, 2, 3]$ $B = []$ $C = []$


move 4: $A = []$ $B = [3]$ $C = [1, 2]$


move 7: $A = []$ $B = [1, 2, 3]$ $C = []$
Towers of Hanoi
for three disks in Python

• three lists A, B, C as stacks
• disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []
move 4 : A = [] B = [3] C = [1, 2]
move 7 : A = [] B = [1, 2, 3] C = []
• three lists A, B, C as stacks
• disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []
move 4 : A = [] B = [3] C = [1, 2]
move 7 : A = [] B = [1, 2, 3] C = []
Towers of Hanoi
for three disks in Python

- three lists $A$, $B$, $C$ as stacks
- disks are represented as numbers $1 < 2 < 3$

Towers of Hanoi with 3 disks
initially: $A = [1, 2, 3]$, $B = []$, $C = []$


move 3: $A = [3]$, $B = []$, $C = [1, 2]$

move 4: $A = []$, $B = [3]$, $C = [1, 2]$


move 7: $A = []$, $B = [1, 2, 3]$, $C = []$
Towers of Hanoi
for three disks in Python

- three lists A, B, C as stacks
- disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []
move 4 : A = [] B = [3] C = [1, 2]
move 7 : A = [] B = [1, 2, 3] C = []
Towers of Hanoi
for three disks in Python

- three lists A, B, C as stacks
- disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []
move 4 : A = [] B = [3] C = [1, 2]
move 7 : A = [] B = [1, 2, 3] C = []
Towers of Hanoi
for three disks in Python

- three lists A, B, C as stacks
- disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially: A = [1, 2, 3] B = [] C = []
move 4: A = [] B = [3] C = [1, 2]
move 7: A = [] B = [1, 2, 3] C = []
Towers of Hanoi

for three disks in Python

- three lists A, B, C as stacks
- disks are represented as numbers 1 < 2 < 3

Towers of Hanoi with 3 disks
initially : A = [1, 2, 3] B = [] C = []


move 4 : A = [] B = [3] C = [1, 2]


move 7 : A = [] B = [1, 2, 3] C = []
# L-10 MCS 260 Wed 17 Sep 2008  towers of Hanoi
#
# To illustrate stacks we solve the towers
# of Hanoi problem with three disks.
#
print 'Towers of Hanoi with 3 disks'
A = [1,2,3]
B = []
C = []
print 'initially : A =', A, 'B =', B, 'C =', C
Towers of Hanoi

the moves in the Python program for 3 disks

```
B.insert(0,A.pop(0))
print 'move 1 : A =', A, 'B =', B, 'C =', C
C.insert(0,A.pop(0))
C.insert(0,B.pop(0))
B.insert(0,A.pop(0))
A.insert(0,C.pop(0))
print 'move 5 : A =', A, 'B =', B, 'C =', C
B.insert(0,C.pop(0))
B.insert(0,A.pop(0))
print 'move 7 : A =', A, 'B =', B, 'C =', C
```
Summary + Assignments

In this lecture we covered more of

- section 1 in *Computer Science. An Overview*
- chapter 4 & 5 of *Python Power!*

Assignments:

1. Translate the realization diagram for a flip-flop into a logical expression involving the variables D, T, and Q, using NOT, NAND, and NOR.

2. Write a Python program to convert a date like 17 September 2008 into 2008-09-17. Names of the month are written in full.

3. Extend `write_values.py` so it works for all strings representing numbers less than one thousand.

4. Extend the Python code for the towers of Hanoi so that it works for four disks.