Final exam

- In class, Tuesday 12/18 9am–noon
  - In this room
  - If you have another final at that time, then Monday 12/19 2–5pm
- Exam target: 75 minutes
  - You can bring 2, double-sided pages of notes

- Cumulative, but will focus a bit more on recent topics
  - OOP • dynamic programming • summations • graphs
  - nope: Turing machines • proofs about languages • sorting
Poster session!
<table>
<thead>
<tr>
<th>1</th>
<th>Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Finite-state machines</td>
</tr>
</tbody>
</table>
| 3 | How a Hmmm program runs  
    registers, RAM, stack, etc. |
| 4 | Linear data structures  
    lists, arrays, stacks, queues |
| 5 | The building blocks of functional programming  
    map, filter, foldl, foldr, list comprehensions |
| 6 | Recursion and use-it-or-lose-it |
| 7 | Analysis techniques  
    summations, recurrence relations, asymptotic complexity |
| 8 | Python namespaces |
| 9 | Dynamic programming  
    with tabulation |
| 10 | Object-oriented programming  
    Python vs Java |
| 11 | Trees and graphs  
    Properties and algorithms |
Brainstorm

Idea generators: call out things big and small related to this topic
Scribes: write down the ideas
You can switch roles!

Goal: **quantity, not quality**
we’ll deal with quality next…

- Nothing is too big or too small
- **No judgements:** Nothing is better than anything else
Organize

- How might you start to arrange / organize / explain this topic? Can you make sub-categories?
- What’s most important?
- What’s “good to know”, but not that important?
- What’s unimportant or unrelated?
- What do you feel you can explain clearly?
- What is not yet clear?
Posters!

13 myths about functional programming and one important truth

All My Circuits *an exciting new Netflix show*

How to care for and feed a Python
1. Latch on to the logic

Good Circuits
- Correct
- Simple, easy to read
- Efficient

EVERYONE THINKS YOU NEED ALL 3 TO MAKE A COMPUTER, BUT WE ONLY USED 2!

PUT IT TOGETHER TO MAKE PARTS OF COMPUTERS

S Latch
- Set
- Clear
- Output

SR Latch
- Set
- Reset
- Output

Pass through -> D Latch
- When strobe input is on, remembers last value when off

3-Bit Memory

Truth Table

Each output 1 gets an and gate connected to each input directly or through a nor gate, and then goes to the output.
2. Finite State Machine

$L = \{ w \mid w \text{ is even} \}$

Language

Input

Initial state

Final/Accepting state

Transition function

State

NFA

DFA

Odd

Even

0

1

0

1

NFA \equiv DFA \equiv RE

Kleene's Theorem

< Turing Machines < All Decision Problems
### CPU

**Fetch-Execute Cycle:**
1. Retrieve the instruction at the program counter.
2. Execute the instruction.
3. Increment the program counter.

This is how HMMM executes a program.

**Instruction Types:**
- Computations (add, subtract, etc., in registers)
- GOTOs (jump & various jumps)
- Memory management (move values between registers & RAM)

### Memory

- R13: Return value of function
- R14: Return line (where the program counter goes after function ends)
- R15: Stack pointer (points to top of stack)

### STACK

- **LIFO:** Stores variables throughout program outside of registers.
  - **Push:** Add a value to top of stack
  - **Pop:** Remove most recent value from stack

Stack builds up higher RAM addresses.

### Other Important Skills:
- Tracing code (given code, say what it does)
Python

| lists: [_, _, ..., _] (mutable) |
| tuples: (_, _, _) (immutable) |

[x+2 for x in range(10) if x>5]

Java

ArrayList<Object>

int[]

Arrays: pros: search/ access
cons: resizing
Better for queues (you have to grab the last element)

Racket list (linked)

pros: insertion, deletion, first element

search/access

cons: slow for stacks

Stacks

pop - remove top element
push - add element to top
Functional Programming

Pros:
- Functional vs. Imperative
- Referral Transparency
- Some input -> Same output
- No side-effects (changes in state are not present)
- Easier debugging
- Order of execution is relevant
- Not as much memory

Anonymous functions

Data structures:
- List
- Tree

HOFs:
- fold
- map
- filter

```
(map (adder 1) L)
```

```
(let (LCS 42) (body expr))
```
Recursion

- Solving a problem by breaking it into smaller problems (ex: sum of a list)

Use it or Lose it
- A specific type of recursion
- Used by making a template

Benefits
- Intuitive
- Gets right answer
- Useful for inductive structures

Drawbacks
- Slow
- Inefficient, redundant computations
Analysis Techniques

What: Measuring the performance of a program
Why: Real-world programs have finite resources

Concepts:
- Cost metric: A measure of cost, depends on situation
  - Empirical: measured
  - Theoretical: derived

Summation:
- \( T(N) = \text{cost given input size } N \)
- For \( j \) in range \( N \)
  - \( \sum_{k=0}^{N} k = \frac{N(N+1)}{2} \) if \( k \) is constant

Recurrence Relations:
- Define \( T(N) = \text{cost given input size } N \)
  - Base case: \( T(0) = 0 \)
  - \( T(1) = k \) for small \( k \)
  - \( T(N) = N T(1) \) for large \( N \)

Repeated Memories:
- Used to find time complexity of recursive functions
- \( T(N) \) can be recursive

Benefits:
- Intuitive
- Gets right
- Useful for structure
Namespaces are one honking great idea

When we import a module, a new namespace gets created.

We refer to names in an imported namespace as module.x

#evaluates to 5

How are names looked up?

1) Local
2) Global
3) Built-in

ERROR

```python
main.py
import module

x = 42
c = Car(10)

class Car:
    def __init__(self, sp):
        self.car_sp = sp
    def get_sp() -> float:
        return self.car_sp

class HondaCar:
    def __init__(self, sp):
        super().__init__(sp)
        self.type = "Honda"
    def get_type() -> str:
        return self.type
```

Dynamic Programming

With Tabulation

Why: Recursive functions include a lot of repeated calculations.

What: By keeping track of old calculations, you can avoid doing repeat work, making a program run faster.

Memoization:
- When solving a problem, check to see if it has already been computed.
  - If a in memo return memo[a]
  - Else: recursion

Tabulation:
- Use previous values in a generated table starting at the N=0 case.

Example: $Fib(N) = Fib(N-1) + Fib(N-2)$

Recurrence Relation:
<table>
<thead>
<tr>
<th>Public vs Private</th>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(global vs local)</td>
<td>The user uses the public variables and methods (functions).</td>
<td>How it is done (What's in the blackbox)</td>
</tr>
<tr>
<td><strong>Objects</strong></td>
<td>Each kind of object has a defined set of operations. The user interacts with instances of objects via their interface.</td>
<td><strong>Java</strong></td>
</tr>
</tbody>
</table>
| - data structures  |                                                      |  * Private variables
| - classes          |                                                      |  * Public variables
| **Inheritance**    | Subclasses can have more variables and methods available to the user than their superclasses. Each instance of a subclass can work as an instance of its superclass, **"Is-a"** | **Python** |
|                    |                                                      |  * _notational_ (private) (global)
|                    |                                                      |  * new Object (initialization)
|                    |                                                      |  * This
|                    |                                                      |  * knows about itself
|                    |                                                      |  * constructor
|                    |                                                      |  * to strings, casts, etc.
|                    |                                                      |  * `__init__`
|                    |                                                      |  * Extends, Implements
|                    |                                                      |  * Class Name (superclass)
|                    |                                                      |  * Implement the super class
Graphs

Def: A data structure of nodes and edges
- Edges connect nodes and can have direction and weight

Uses: Modeling things (i.e., internet)

Types:
- Connected vs. nonconnected
- Complete vs. noncomplete
- Directional vs. nondirectional
- Weighted vs. nonweighted
- Dense vs. sparse
- Cyclic vs. acyclic

Trees
- Specific type of connected, directional, nonweighted, acyclic graph
- Edges connect roots and leaves

Binary Search Tree
- Specific type of tree that makes it easier to store and find things

Subtree
Height