"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan
Course Objectives

Algorithms:
- Design – strategies for algorithmic problem solving
  - advanced data structures
  - and their use in algorithms
- Reasoning about algorithm correctness
- Analysis of time and space complexity
- Implementation – create an implementation that respects the runtime analysis

Algorithmic Approaches:
- Divide and Conquer
- Greedy
- Dynamic programming

Problem Classes:
- P: Polynomial, NP: Non deterministic Polynomial
Grading

Quizzes  10%

Assignments

  Programming  20 %
  Written      20 %

Exams

  Midterm 1  15 %
  Midterm 2  15 %
  Final      20 %
Implementation

Programs will be written in Python:

- Concise and powerful
- Powerful **data structures**
  - tuples, dictionaries, arraylists
- Simple, easy to learn syntax
- Highly readable, compact code
- Supports object oriented and functional programming
- An extensive standard library
- Strong support for integration with other languages (C, C++, Java)
Python vs. Java

What makes Python different from Java?

Java is **statically** typed

Variables are bound to types at compile time.

This decreases run time errors, but makes java programs more rigid.

Python is **dynamically** typed

A variable takes on some type at run time, and its type can change. A variable can be of one type somewhere in the code and of another type somewhere else.

This makes python programs more flexible, but can cause strange run time errors.
Does anyone else use Python?
Our approach to problem solving

Formulate it with precision (usually using mathematical concepts, such as sets, relations, and graphs)

Design an algorithm and its main data structures

Prove its correctness

Analyze its complexity (time, space)
   Improve the initial algorithm (in terms of complexity), preserving correctness

Implement it, preserving the complexity

(Of ten these steps are repeated multiple times.)
Representative Problems
Remember the problem solving paradigm

1. Formulate it with precision (usually using mathematical concepts, such as sets, relations, and graphs, costs, benefits, optimization criteria)

2. Design an algorithm

3. Prove its correctness, e.g. in terms of pre and post conditions

4. Analyze its complexity

5. Implement respecting the derived complexity

Often, steps 2-5 are repeated, to improve efficiency
Interval Scheduling

You have a resource (hotel room, printer, lecture room, telescope, manufacturing facility...)

There are requests to use the resource in the form of start time $s_i$ and finish time $f_i$, such that $s_i < f_i$

Objective: grant as many requests as possible. Two requests $i$ and $j$ are compatible if they don't overlap, i.e.

$$f_i \leq s_j \text{ or } f_j \leq s_i$$
Interval Scheduling

Input. Set of jobs with start times and finish times.
Goal. Find maximum cardinality subset of compatible jobs.

What happens if you pick the first starting (a)?, the smallest (c)? What is the optimum?
Algorithmic Approach

The interval scheduling problem is amenable to a very simple solution.

Now that you know this, can you think of it?

Hint: Think how to pick a first interval while preserving the longest possible free time...
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of compatible jobs.
Bipartite Matching

Stable matching was defined as matching elements of two disjoint sets.

We can express this in terms of graphs.

A graph is **bipartite** if its nodes can be partitioned in two sets $X$ and $Y$, such that the edges go from an $x$ in $X$ to a $y$ in $Y$.
Bipartite Matching

**Input.** Bipartite graph.

**Goal.** Find maximum cardinality matching.

Matching in bipartite graphs can model assignment problems, e.g., assigning jobs to machines, where an edge between a job \( j \) and a machine \( m \) indicates that \( m \) can do job \( j \), or professors and courses.

How is this different from the stable matching problem?
Independent Set

**Input.** Graph.

**Goal.** Find maximum cardinality independent set: subset of nodes such that no two are joined by an edge.

Can you formulate interval scheduling as an independent set problem? If so, how could you solve the interval scheduling problem?
Independent set problem

- There is no known efficient way to solve the independent set problem.

- But we just said: we can formulate interval scheduling as independent set problem..... ???

- What does "no efficient way" mean?

- The only solution we have so far is trying all sub sets and finding the largest independent one.

- How many sub sets of a set of n nodes are there?
Looking ahead…

- **Interval scheduling**: $n \log(n)$ greedy algorithm.

- **Weighted interval scheduling**: $n \log(n)$ dynamic programming algorithm.

- **Bipartite matching**: polynomial max-flow based algorithm.

- **Independent set**: NP (no known polynomial algorithm exists).
Algorithm

Algorithm: effective procedure
- mapping input to output

effective: unambiguous, executable

- Turing defined it as: "like a Turing machine"

- program = effective procedure

Is there an algorithm for every possible problem?
Algorithm: effective procedure

- mapping input to output

**effective**: unambiguous, executable

- Turing defined it as: "like a Turing machine"
- program = effective procedure

Is there an algorithm for every possible problem?

No, the problem must be effectively specified: "how many angels can dance on the head of a pin?" not effective. **Even if** it is effectively specified, there is not always an algorithm to provide an answer. This occurs often for programs analyzing programs (examples?)
Ulam's problem

def f(n):
    if (n==1) return 1
    elif (odd(n)) return f(3*n+1)
    else return f(n/2)
Ulam's problem

def f(n):
    if (n==1) return 1
    elif (odd(n)) return f(3*n+1)
    else return f(n/2)

Steps in running f(n) for a few values of n:
1
2, 1,
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
10, 5, 16, 8, 4, 2, 1

Does f(n) always stop?
Ulam's problem

def f(n):
    if (n==1) return 1
    elif (odd(n)) return f(3*n+1)
    else return f(n/2)

Nobody has found an n for which f does not stop
Nobody has found a proof (so there can be no algorithm deciding this.)
A generalization of this problem has been proven to be undecidable.
A problem P is undecidable, if there is no algorithm that produces P(x) for every possible input x
The Halting Problem is undecidable

Given a program $P$ and input $x$
will $P$ stop on $x$?

We can prove (cs420):
the halting problem is undecidable

i.e. there is no algorithm $\text{Halt}(P,x)$ that for any program $P$ and input $x$ decides whether $P$ stops on $x$. 
Verification/equivalence undecidable

Given any specification $S$ and any program $P$, there is no algorithm that decides whether $P$ executes according to $S$.

Given any two programs $P_1$ and $P_2$, there is no algorithm that decides $\forall x: P_1(x) = P_2(x)$.

Does this mean we should not build program verifiers?
Intractability

Suppose we have a program,
- does it execute a in a reasonable time?
- E.g., towers of Hanoi (cs200).

Three pegs, one with \( n \) smaller and smaller disks, move (1 disk at the time) to another peg without ever placing a larger disk on a smaller

\[ f(n) = \# \text{ moves for tower of size } n \]

Monk: before a tower of Hanoi of size 100 is moved, the world will have vanished
def hanoi(n, from, to):
    if (n>0):
        via = 6 - from - to
        hanoi(n-1,from, via)
        print "move disk", n, " from", from, " to ", to
        hanoi(n-1, via, to);
$f(n)$: #moves in hanoi

$f(n) = 2f(n-1) + 1$, $f(1)=1$

$f(1) = 1$, $f(2) = 3$, $f(3) = 7$, $f(4) = 15$

$f(n) = 2^{n-1}$

How can you show that?

Can you write an iterative Hanoi algorithm?

Was the monk right?

$2^{100}$ moves, say 1 per second.....

How many years?
Is there a better algorithm?
Is there a better algorithm?

Pile(n-1) must be

off peg1

and

on one other peg

before disk n can be moved to its destination

so (inductively) all moves are necessary
Algorithm complexity

Measures in units of **time** and **space**

Linear Search X in dictionary D

\[ i=1 \]

\[ \text{while not at end and } X \neq D[i]; \]

\[ i = i + 1 \]

We don't know if X is in D, and we don't know where it is, so we can only give **worst** or **average** time bounds.

We don't know the time for atomic actions, so we only determine **Orders of Magnitude**.
Linear Search: time and space complexity

Space: $n$ locations in $D$ plus some local variables

Time:
In the worst case we search all of $D$, so the loop body is executed $n$ times

In average case analysis we compute the expected number of steps: i.e., we sum the products of the probability of each option and the time cost of that option. In the average case the loop body is executed about $n/2$ times

$$\sum_{i=1}^{n} \frac{1}{n} \cdot i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n(n+1)/2}{n} \approx \frac{n}{2}$$