Announcements:
Welcome to 6.0002!

- PS1 is due Monday, October 28th, 9pm. - we hold office hours during the week and will be available on Piazza to answer questions about this pset. Please post questions on Piazza early, even on weekends.
- We have checkoffs due after each pset, you must come to OH before the deadline to get checked off.
- Office hours are usually very busy around deadlines so please plan accordingly!
- Use 6.00 or 6.0002 Stellar to check calendar, download psets & submit psets.
- Do finger exercises on MITx (link on stellar) before every lecture.
- Use 6.0002 Piazza to ask questions about psets or upcoming exams.
- For more personal questions (about grades, extensions, etc) post privately on Piazza.
- Office hours every Monday, Tuesday, Wednesday, Thursday, Friday until 9pm. Check on Stellar home page for details on times and locations.
- Remember, collaboration is allowed (please list collaborators), but sending, looking at, or copying code (including from previous semesters) is not allowed.

1. Graphs

Graphs are nodes or vertices connected by edges. Nodes can store data (numeric values, names, etc.), and edges connect nodes to each other, and may be weighted. A graph may be directed or undirected.

A graph can be represented by an adjacency matrix or adjacency lists.
- Matrix is better when graph is densely populated (most cells are full, little space wasted)
- Lists are better for sparse graphs, though possibly harder to visualize relationships

Trees: An undirected graph in which any two vertices are connected by one simple path, and there are no cycles. The most common tree is a rooted binary tree: there is a root node at the top, every node except for the root has a parent, and each node has at most one left child and one right child. Nodes without children are called leaves. At each level of the tree, number of nodes is at most $2^{level}$ (starting with $2^0$ for the root). For a balanced binary tree, if $h =$ height of tree and $n =$ number of nodes, then what is their relationship?

Binary search tree: Obey the constraint that all nodes in the left subtree of a node have values less than the value at the given node, and all nodes in the right subtree have greater values than that of the given node (no duplicates allowed). What is the time complexity for search, on average and worst case?

General graphs are not constrained to # of children per node, and do not have roots.
Understanding code from lecture 3: Digraph is a directed graph. You can add nodes and edges going from one node to another. Edges (the attribute in Digraph) is a dictionary that maps a node (key) to its adjacency list; ex. \{A: [B, C]\} == A → B, A → C. So Graph is a subclass of Digraph--- how did we do this?

Shortest Path Problems: Given a source node A and a destination node B, find the shortest sequence of edges that will lead from A to B. Edges must be in sequence, where the destination node of one edge is the source node of the next edge.

Breadth-First Search (BFS): Starts at the source node and searches one level at a time (all nodes distance 1 away, then all nodes distance 2 away, etc.) until the destination node is reached.

1. Keep a queue of “start nodes”. First start node is the source node.
2. Iterate through the “start nodes” in order. For each node:
3. Iterate through each child of that node, until:
   - Destination node found
   - No more children of current node
4. Append all children nodes to the end of the “start nodes” list and go back to step 2.
   (Keep a list of visited nodes, since if we’ve already seen them once, skip them if they show up later since that path will definitely be longer.)

Depth-First Search (DFS): Starts at the source node and “drills down” the list of children/descendants until it can’t go any farther. Then it goes back up one level and tries the next child node in line.

1. Start at the source node.
2. Recursively choose a child of the current node, until:
   - Node has already been followed (avoid cycles!)
   - Destination node reached. Store this path as the current shortest path.
   - Node with no children reached
3. When the last two conditions are reached, backtrack and choose the next child node.

DFS is exhaustive search of possible paths, till you find the destination node. You stop searching once you find the destination node. Note that if you only care about getting ANY path from source to destination, DFS will always return at the same time or before shortest-path BFS.

BFS is guaranteed shortest path the first time you visit the destination node! (but only for unweighted graphs)

More Problems
- Say you and your friends decided to represent your friendships as a weighted, undirected graph where each person is a node and edge weights correspond to the degree of friendliness (an integer) between two people. Could you think of a way to convert your graph into an equivalent graph composed of unweighted edges?
- Given a directed graph, how do you find out whether or not there is a path between two nodes?
2. Optimization (Pset 1)

A class of problems in which you want to find the greatest, fastest, or smallest (some "--est").

- Every optimization problem has 2 primary parts
  - **Objective function**: the quantity or metric to be optimized (i.e. maximized or minimized). For example, the cost of a trip through Europe.
  - **Set of constraints** (possibly empty) that must be honored. For example, you want to go to Amsterdam, London, Paris and Madrid, but you don’t want to take more than 2 train rides.

- Algorithmic Approaches:
  - **Brute Force**
    - Enumerate and look at all possible solutions. Can make a **decision tree** to describe all possible solutions or make a **power set** that enumerates all possible solutions
    - Guaranteed to find a globally-optimal solution, if it exists
    - Infeasible for even relatively small problems - brute force algorithms generally have complexity exponential in the number of items under consideration
  - **Greedy**
    - Repeatedly make “best” valid choice in each step of the algorithm until can no longer make any valid choice
    - Different metrics for what the “best” choice is may result in better or worse solutions
    - Typically much faster than the corresponding brute force algorithm
    - Will find a locally-optimal solution, which **may or may not** be the global optimum

Example 1: Class Scheduling
- You want to take as many classes as possible this semester so you can graduate early, but your advisor won’t allow you to schedule conflicting lectures :(
- What are the objective function and the constraints?
- What are some possible greedy algorithms you can use for this problem?
3. Dynamic Programming (Pset 1)

**Dynamic programming (DP)** is useful when you have problems that can be broken down into overlapping subproblems and optimal substructure:

- **optimal substructure** - a globally optimal solution can be found by combining optimal solutions to local subproblems. (Think recursion)
  - For example, merge sort works on the basis that a list can be sorted by first sorting sub-lists individually, and then merging the sub-lists together.

- **overlapping subproblems** - an optimal solution involves solving the same problem multiple times. (Think memoization)
  - Merge sort does not have this property; each merge involves different lists.
  - Fibonacci does have this property: calculating fib(4) and calculating fib(5) both involve calculating fib(1), fib(2), fib(3), and fib(4).

**Memoization** is the process of storing the values for each subproblem in a table so that the next time the subproblem is needed, the program can just return the stored solution instead of recomputing the solution from scratch.

Lots of problems can be improved by using dynamic programming:

- **Bioinformatics:** DNA sequence alignment
- **Mathematical sequences:** Fibonacci (we’ve done this in 6.0001!)

**Example 2: Knapsack Problem**

- Forgetting about DP for a moment, what is the number of possibilities we end up with for n items?
- Possible ways of solving this problem (maybe imperfectly) are:
  - Greedy algorithms: sort the items according to their properties, take the highest item you can take until you run out of space
    - Weight, value or density
    - Is density guaranteed to yield a better solution?
    - Will probably yield a good solution, but can also be far off
  - Brute-force: actually try every possible combination
    - Not possible in fractional knapsack
    - Takes exponential time
  - Dynamic programming: reuse old solutions to save time
    - Runs in pseudo-polynomial time
    - Far faster than the exponential time of brute-force

Does this problem have an optimal substructure?

Does this problem have overlapping subproblems?

Why does dynamic programming work? For what kind of problems is DP effective?