CS1 Lecture 32 Apr. 7, 2017

- HW 7 due Monday morning, 9am.
- Make sure to complete the weekly survey
- HW 8 available Monday, last before Exam 2

- HW7
  - when assignment says “You may search the Internet for implementations to use” if means (for the one time in this class!) you can directly copy/download use the code found.
  - most quicksorts found on web are “bad” when used directly as is.

Last time

- Faster sorting methods – merge sort, quicksort

Today

- Sorting summary
- A short discussion of optimization problems and greedy algorithms
- Introduction to graphs and graph algorithms

Last time – discussed how we might we sort faster than insertion sort, selection sort, and other simple sorts – i.e., faster than n^2.

Try a **divide-and-conquer** approach:

- Divide problem into subproblems
- Solve subproblems recursively
- Combine results

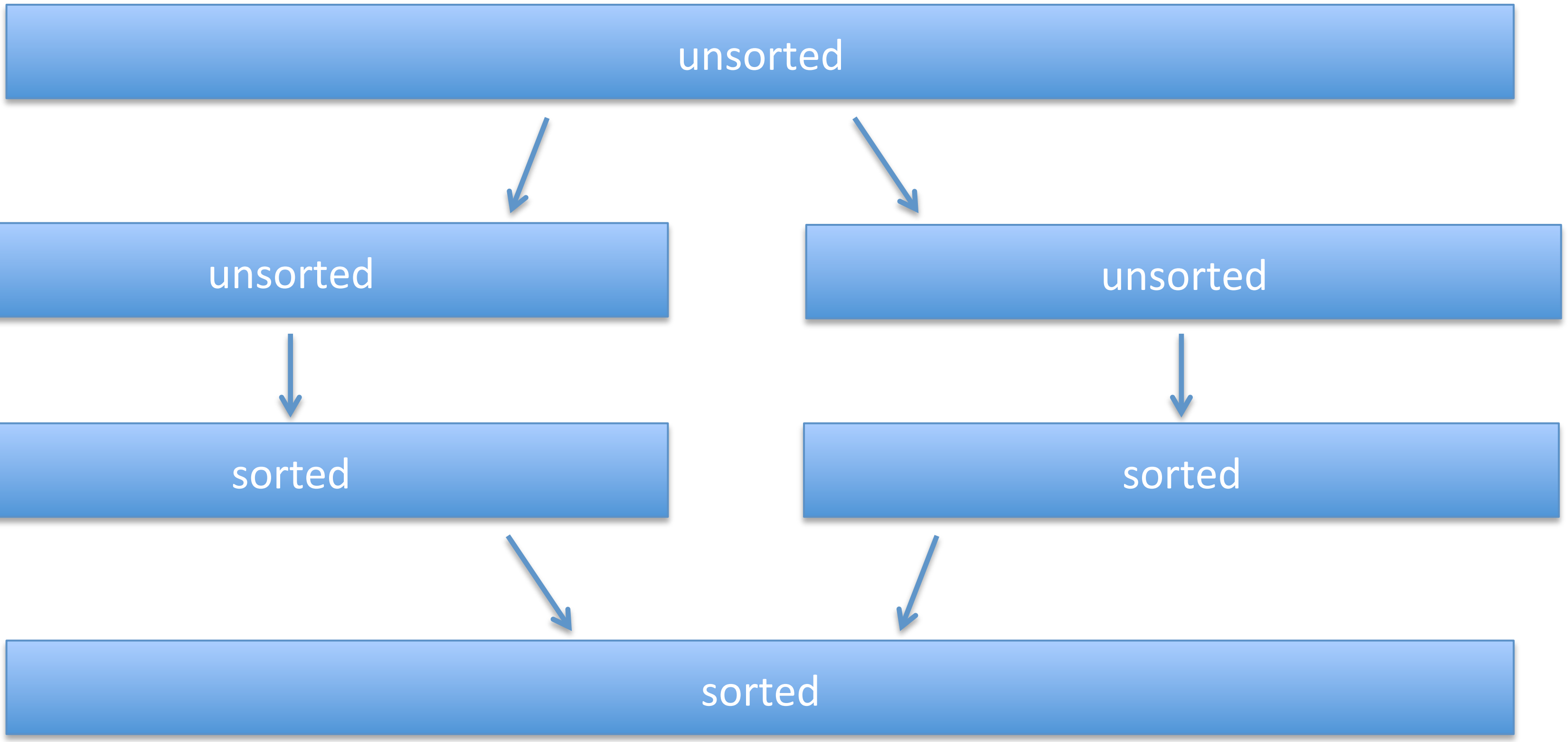
Binary search is a special case of divide and conquer.

- Check middle element and create one subproblem of half the size
- This yielded speed-up from O(n) to O(log n)

Many problems benefit from divide and conquer approach

Last time - Merge sort

1. divide list into two (almost) equal halves
2. sort each half recursively!
3. combine two sorted halves into fully sorted result



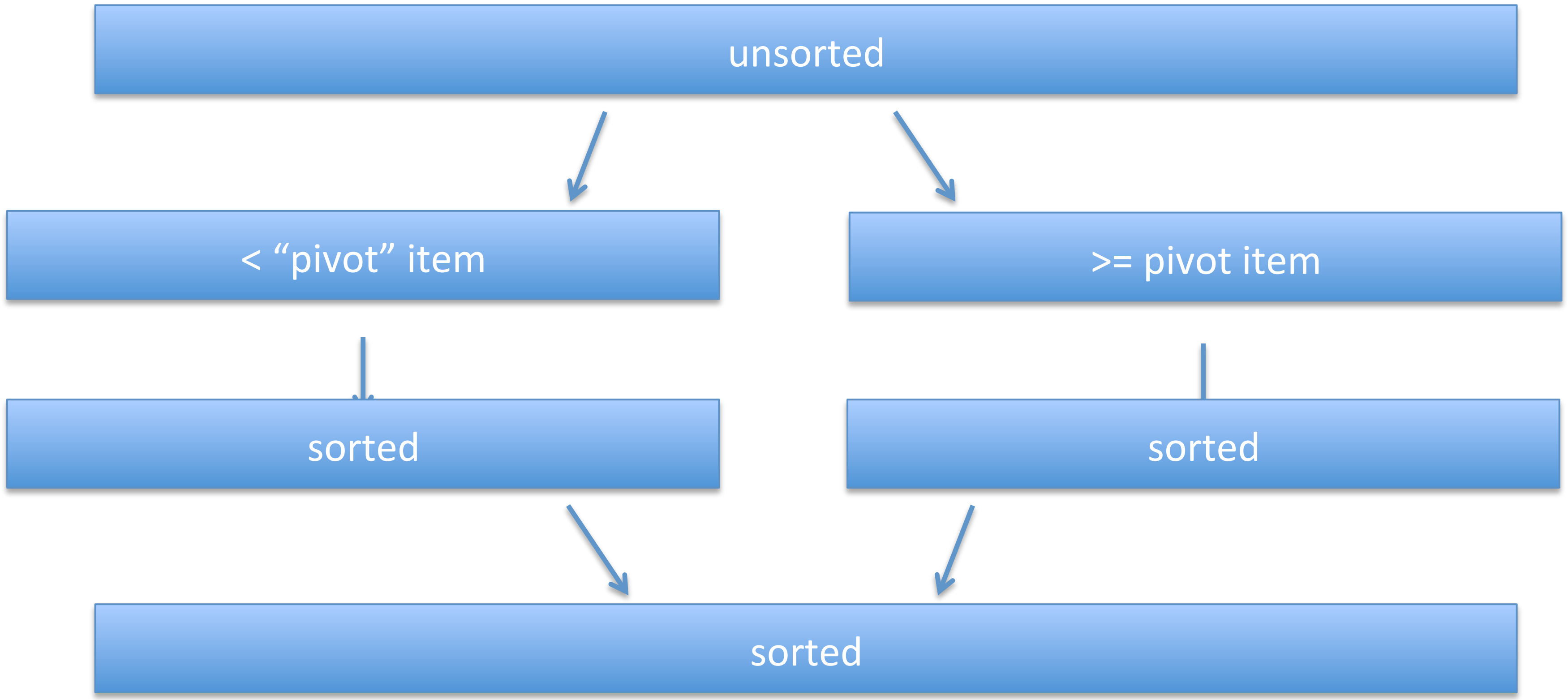
It should be clear how to implement each step (lec31.py)

Last time - Merge sort

- Merge sort is very famous sorting algorithm. Classic example of power of divide and conquer. Yields “optimal” O(n log n) sort. Much faster than O(n^2) algorithms.
- Can be implemented non-recursively (but most people find the rec. version much easier to implement)
- One possible concern? Standard implementations require O(n) additional space
- Python’s built-in sort is (in most Python implementations) Timsort (named after Tim Peters), a hybrid drawing from mergesort and insertion sort. Has O(n log n) average and worst-case time (like mergesort) and O(n) best case time (like insertion sort)

Another divide-and-conquer style sorting algorithm: quicksort

1. Divide ("partition") list into two parts, one containing all elements < some number ("pivot"), the other containing all elements >= that number
2. Recursively sort parts
3. combine two sorted halves into fully sorted result



Is it clear this time how to implement each step? (the "partition" step is a bit tricky to get right)

Last time - Quicksort

- Quicksort has worst case running time of $O(n^2)$
- *BUT* average case O(n log n) and if we choose pivot properly we can make worst case very very unlikely. (The detailed mathematical analysis of this is not so easy).
- Unlike mergesort, is "in place" – does not require O(n) extra space).
- When implemented properly is excellent and very commonly used sorting method in the real world.
- Be careful if you implement it yourself. *Easy to get slightly wrong*. E.g. the code at the first (and second) result returned when I googled - quicksort python – is c*orrect* (in that it properly sorts) but a poor implementation because it chooses pivot badly (try it on an already sorted list of, say, 10000 or more elements, or even a list containing many many identical elements!?) – qsbad.py
- Good standard pivot choice – "median of three" – choose as pivot the median value among first, middle, and last elements in the given list. E.g. for [15, 4, 2, 99, 6, 3, 25, 26, 8], choose median of 15, 6, 8, which is 8
  - In this case, good – 4, 2, 3, 6 will be in "less-than" partition, 15, 99, 25, 26 will be in "greater-than" partition.

Last time - Merge sort and Quicksort are both allocate work differently

- Divide step
  - mergesort: easy – just cut in half O(1)
  - quicksort: takes work – scan through entire list putting elements in “less than” or “greater than” (vs pivot) part O(n)
- Combine step
  - Merge sort: takes work – step through subproblem solutions, merging them O(n)
  - quicksort: easy! we’re actually already done! Just put parts together O(1)
- Subproblems
  - mergesort: the two subproblems always half the size
  - quicksort: subproblem size depends of value of item you choose as pivot. What pivot would yield half-sized subproblems? Can you find that pivot item easily?
    - poor pivot choices can yield poor sorting performance
    - good practical pivot choices: 1) median of first, middle, last items, 2) random item

Sorting summary

Simple sorts:

- Understand the main step of:
  - Selection sort
  - Insertion sort
- Both are O(n^2) but insertion has very good behavior on sorted/nearly sorted data, and is often used in hybrid algorithms to finish the job

Faster sorts:

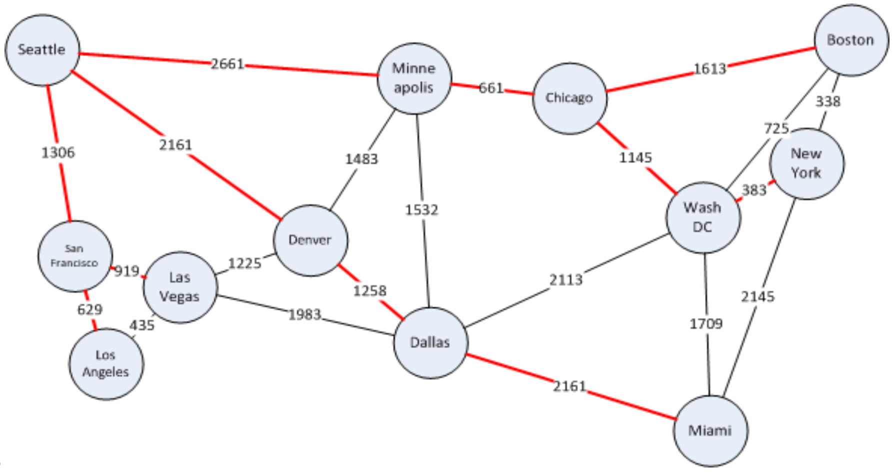
- Merge sort and quicksort are two divide-and-conquer style algorithms that yield O(n log n) behavior, enabling sorting much much larger datasets than the simple sorts

Python's built-in sort is Timsort, a merge/insertion hybrid

Visualizations and comparison of properties of sorting algorithms at http://www.sorting-algorithms.com

Optimization and graph problems

- Many computing tasks these days involve solving **optimization problems** – finding the smallest, biggest, best, cheapest of something
- In general, optimization problems are expressed in terms of two components
  - An *objective function* that is to be minimized/maximized (e.g. airfare, travel distance, travel time)
  - *A set of constraints* that must be met (e.g. route must include these intermediate cities, departure must be after 8am, arrival must be before noon)
- Many optimization problems can be addressed as problems on **graphs** (the computer science kind, consisting of nodes/vertices and edges/connections, not the 2D x-y *plots* you have been using to visualize running time behavior.)



- HW 8 will focus on graphs
- Before we get to graphs, a quick look at other optimization problems …

A simple optimization problem

Suppose you need to give someone n cents in change (given US coins – penny, nickel, dime, quarter). How do you do it with the minimum number of coins?

- “greedily” give as many large value coins as possible first, then next largest size, etc. That is, first as many quarters as possible, then as many dimes, …
- E.g. for 56¢: 2 quarters, 1 nickel, 1 penny

What if we replace nickel (5c) with 3 cent and 4 cent coins? Does same greedy approach work?

- for 56¢ greedy approach yields 25, 25, 4, 1, 1 but 25, 25, 3, 3 is fewer coins!

For US coins, the algorithm works. It is an example of a broad class of “greedy algorithms” – if you take Algorithms (CS3330), you will likely study more about greedy algorithms.

Greedy algorithms

- Generally, a greedy algorithm is one that proceeds as follows:
  - At each step, choose the "locally"/"apparently"/"immediately" best option (e.g. the one that seems like to make the most progress toward a solution)
- The idea (hope!) behind greedy algorithms is that by making many locally optimal choices we end up with overall optimal solution.
- But, as we saw, *doesn't always succeed!* Sometimes need to work harder.
- Greedy algorithm for Travelling Salesperson problem? No, does not always yield optimal solution
- Greedy algorithm for shortest driving route between two cities? (E.g. driving directions in Google maps). Yes, Dijkstra's algorithm.
- Greedy algorithms are very important and useful. But you need to think carefully about whether greedy approach indeed gives you an optimal solution (or, if not, a good enough one)
- for more, see http://en.wikipedia.org/wiki/Greedy_algorithm

Egyptian fractions

3/4 -> sum of (different) fractions all with 1 as numerator

E.g. 3 / 4 -> 1 / 2 + 1 / 4

Greedy algorithm?

Try in sequence ½, 1/3, ¼, 1/5, …

see code in: egypt.py (including implementation using division that does not work well!)

Another optimization problem

	Value	Weight
clock	175	10
painting	90	9
radio	20	4
vase	50	2
book	10	1
computer	200	20

Burglar with a knapsack in a home full of valuable items

- Objective function: fill knapsack with maximum value
- Constraint: knapsack can only hold 20 pounds

Algorithm to solve this?

Burglar filling knapsack

	Value	Weight	Val/wt
clock	175	10	17.5
painting	90	9	10
radio	20	4	5
vase	50	2	25
book	10	1	10
computer	200	20	10

Constraint: Knapsack can hold up to 20lbs

- Greedy approach says to pick "best" at each step. What rule could we use for best here?
  - Highest value -> computer only -> $200 total
  - Lowest weight -> book, vase, radio painting, -> $170 total
  - Highest value/weight -> vase, clock, book, radio -> $255 total

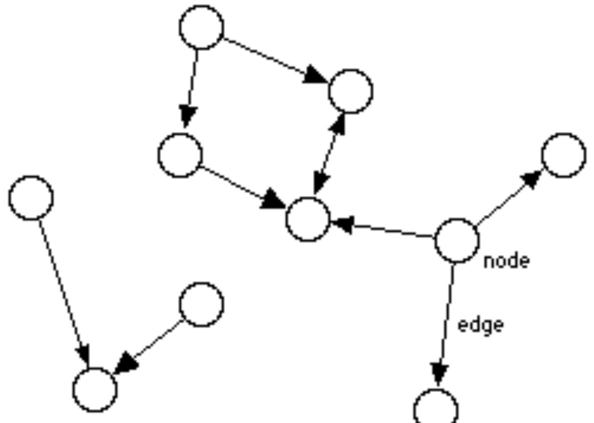
None of these criteria produce the optimal solution for this particular situation (best total is $275 via clock, painting, book).

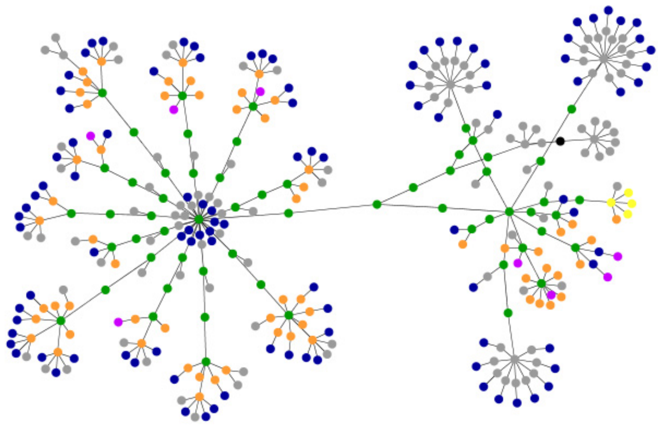
We could easily write an algorithm that always find the best by trying every subset of items. However, this solution is in very efficient for many knapsack-like problems. If have n items, how many subsets? 2^n, so exhaustive search potentially very slow

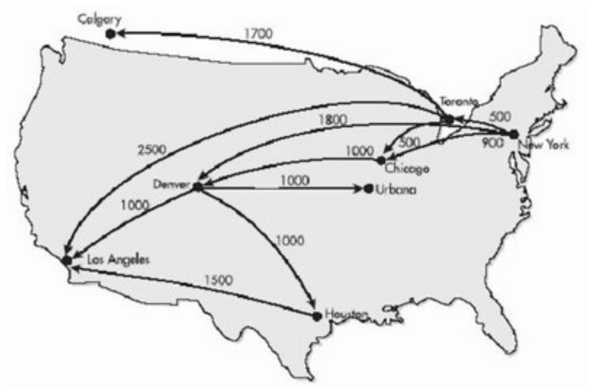
At present, no known efficient algorithm for knapsack problems

Graphs and optimization problems based on graphs

- *Many* important real-world problems can be modeled as optimization problems on **graphs**
- A graph is:
  - A set of nodes (vertices)
  - A set of edges (arcs) representing connections between pairs of nodes
- There are several types of graphs:
  - **Directed**. Edges are "one way" from source to destination)
  - **Undirected**. Edges have no particular direction – can travel either way, "see" each node from other, etc.
  - **Weighted**. Edges have associated numbers called weights that can be used to represent cost, time, flow capacity, etc.
- See Ch. 2 of follow-up book to our text – Think Complexity - http://greenteapress.com/complexity/html/thinkcomplexity003.html

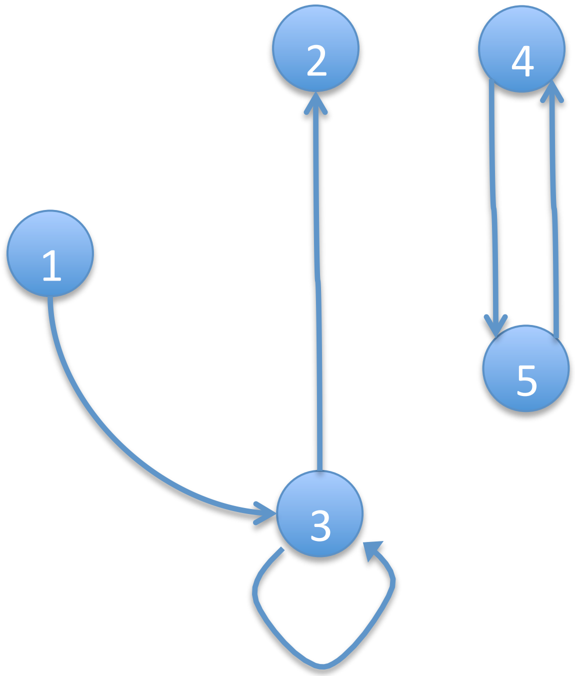


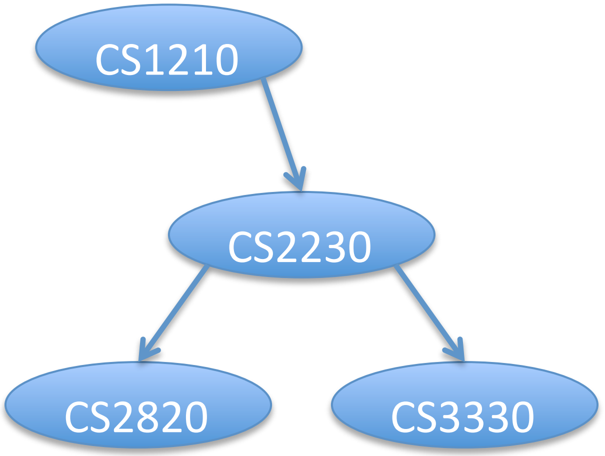




Directed graph

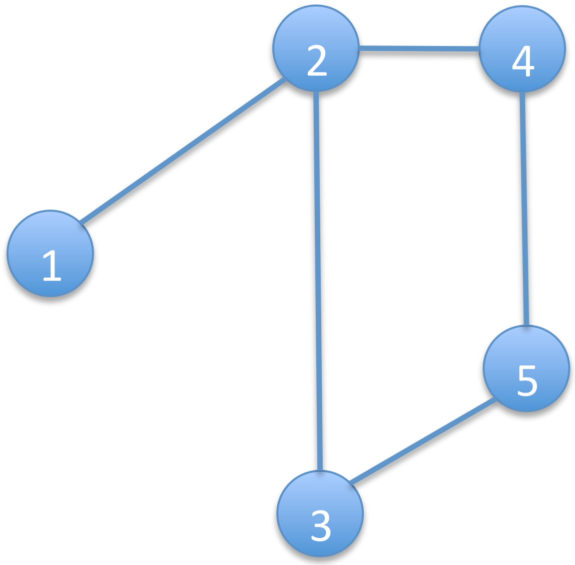
- edges are "one way" from source to destination
- Can have two (one each way) between a pair of nodes
- Node can have edge to self
- Example relationships:
  - course prerequisite
  - hyperlink between web pages
  - street between intersections
  - Twitter follower
  - Infection spread from-to

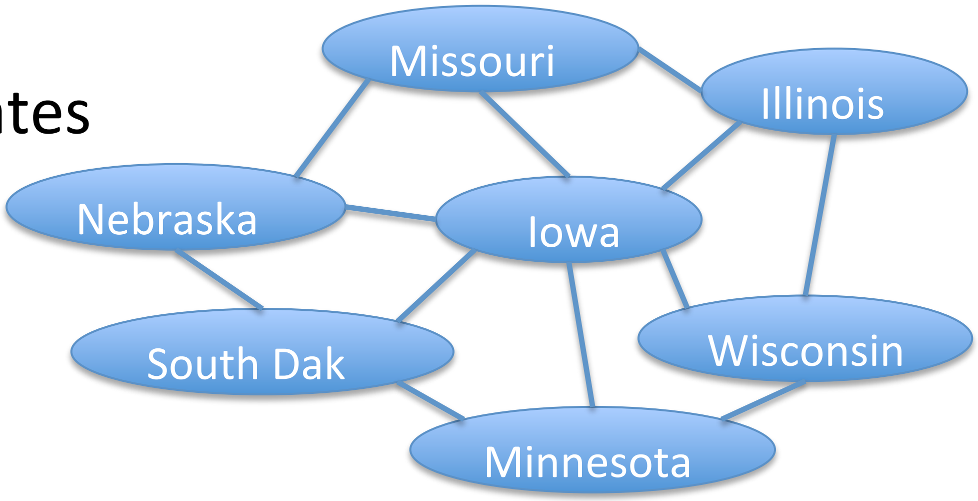




Undirected graph

- Edges have no direction. Can “travel” either direction
- Can have only one edge between a pair of nodes*
- Node cannot have edge to self
- Example relationships:
  - Facebook friend
  - Bordering countries/states

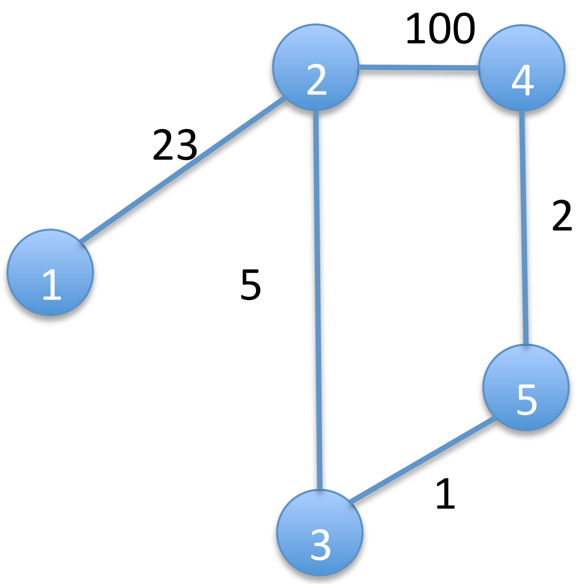


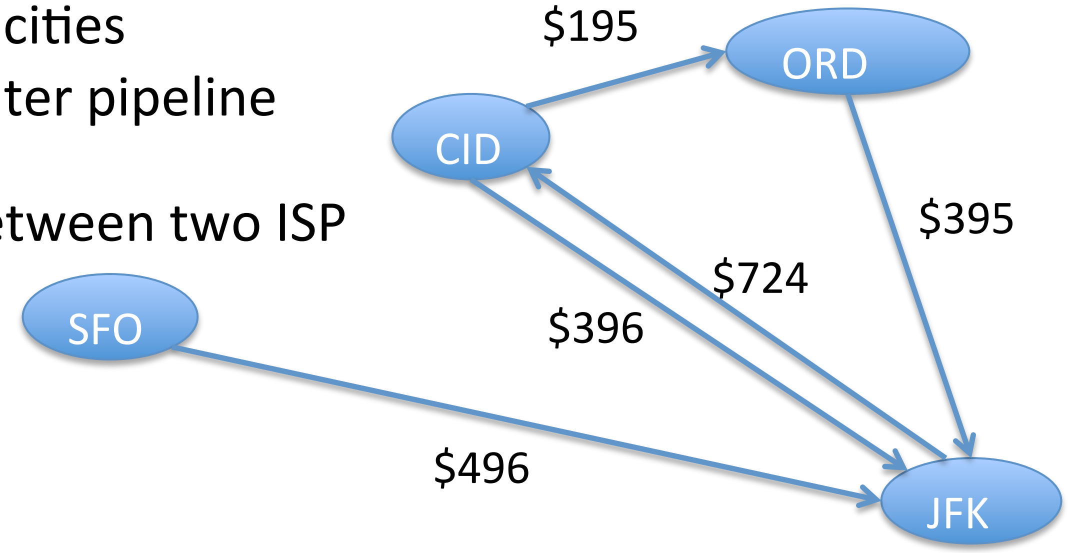


*another kind of graph – multigraph – relaxes this rule

Weighted graphs

- Variant of both directed and undirected graphs in which each edge has an associate number called a **weight** or cost
- Edge weight provides additional information about the relationship between the nodes.
- Example relationships:
  - Airfare between two cities
  - Distance between two cities
  - Flow capacity of oil/water pipeline between two points
  - Network bandwidth between two ISP nodes

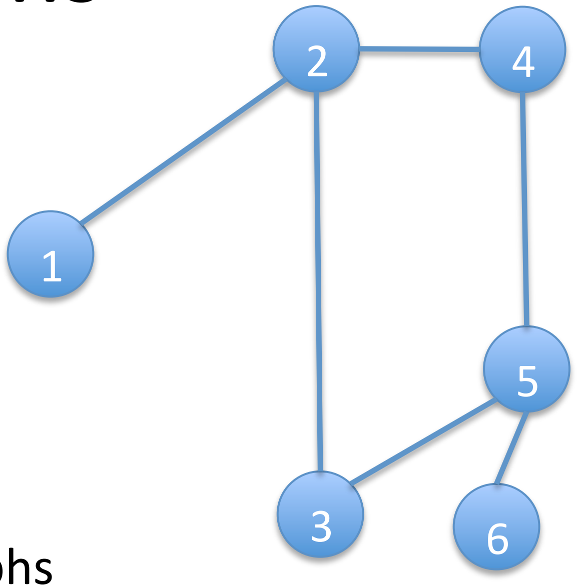


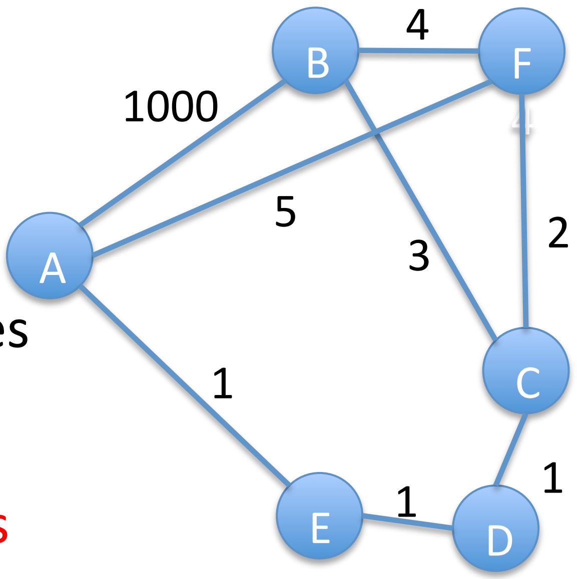


Classic graph problems

- Determine if a graph has a cycle, a path that loops back to start points (e.g. 2-4-5-3-2)
- Find a path (non-branching) that traverses each (undirected) edge exactly once
  - Leonhard Euler and the Bridges of Königsberg
  - Not possible in graph on top right
- Find the shortest path between source s and destination
  - Different algorithms for weighted/unweighted graphs
- Find longest path between source and destination
- Find a path that visits each vertex exactly once
  - A, E, D, C, F, B, A in example on bottom right
- Path of minimum cost that visits each vertex once
  - A, E, D, C, B, F, A (cost 15) in example
- Assign no more than n different colors to vertices under constraint that no pair of connected vertices has the same color

Some of these are easy (have fast algorithms), others hard (no known efficient solution)

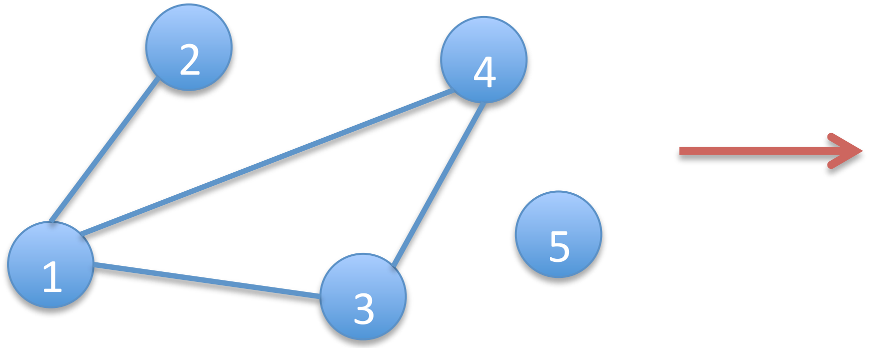




Representing graphs

- How can we represent general graph in Python?
  - Need to keep track of nodes
  - Need to keep track of edges
- Several ways to represent graphs have been developed
    - List of nodes and list of edges
    - Adjacency matrix
    - Adjacency lists
    - Dictionary of dictionaries
  - Efficiency of algorithms that solve graph problems can vary greatly depending on how graph are representated
  - a strong influence on choice is the fact that one of the most common things needed in graph algorithms is access to immediate neighbors of a node (nodes that are destinations of edges for which "current" node is source)

Adjacency matrix



	1	2	3	4	5
1	False	True	True	True	False
2	True	False	False	False	False
3	True	False	False	True	False
4	True	False	True	False	False
5	False	False	False	False	False

- Appealingly simple to understand and implement
- Use, e.g. a list of lists containing True/False, 0/1, or similar
- NOT the most common graph representation for most problems. Can you think of a reason why?
  - Consider representing Facebook friends graph where each node is a FB user and an edge exists between two nodes whenever the two are FB friends.
  - One billion nodes. Adjacency matrix 1B x 1B in size! Your computer doesn't have that much storage. But FB graph ***can*** be represented in computer! How?
  - The 1B x 1B would be mostly False/0 – most people don't have huge number of friends. Should be representable in closer to 1B * median number of friends. Other representations enable this huge memory savings.

Next time

- Adjacency matrix graph representation
- Adjacency list graph representation
- A graph traversal algorithm – breadth first search