Algorithms

NP Completeness

Review: Dynamic Programming

- When applicable:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Basic approach:
    - Build a table of solved subproblems that are used to solve larger ones
    - *What is the difference between memoization and dynamic programming?*
    - *Why might the latter be more efficient?*

Review: Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
    - Yes: fractional knapsack problem
    - No: playing a bridge hand
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Review: Activity-Selection Problem

- The *activity selection problem*: get your money’s worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, starting and ending at different times
  - Your goal: ride as many rides as possible
- Naïve first-year CS major strategy:
  - Ride the first ride, when get off, get on the very next ride possible, repeat until carnival ends
- *What is the sophisticated third-year strategy?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity i f_i = finish time of activity i
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is optimal substructure for this problem?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity i f_i = finish time of activity i
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is optimal substructure for this problem?*
  - A: If k is the activity in A with the earliest finish time, then A - $\{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$

Review: Greedy Choice Property For Activity Selection

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal sol'n
  - Them 17.1: if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ such that $\{1\} \in A$
    - Sketch of proof: if $\exists$ optimal solution B that does not contain $\{1\}$, can always replace first activity in B with $\{1\}$ (*Why?*). Same number of activities, thus optimal.

Review: The Knapsack Problem

- The *0-1 knapsack problem*:
  - A thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust
- *What greedy choice algorithm works for the fractional problem but not the 0-1 problem?*

NP-Completeness

- Some problems are *intractable*:
  as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time? Standard working definition: *polynomial time*
  - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: every algorithm we've studied provides polynomial-time solution to some problem
  - We define **P** to be the class of problems solvable in polynomial time
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

NP-Complete Problems

- The *NP-Complete* problems are an interesting class of problems whose status is unknown
  - No polynomial-time algorithm has been discovered for an NP-Complete problem
  - No suprapolynomial lower bound has been proved for any NP-Complete problem, either
- We call this the *$P = NP$ question*
  - The biggest open problem in CS

An NP-Complete Problem: Hamiltonian Cycles

- An example of an NP-Complete problem:
  - A *hamiltonian cycle* of an undirected graph is a simple cycle that contains every vertex
  - The hamiltonian-cycle problem: given a graph G, does it have a hamiltonian cycle?
    - Draw on board: dodecahedron, odd bipartite graph
  - *Describe a naïve algorithm for solving the hamiltonian-cycle problem. Running time?*

P and NP

- As mentioned, **P** is set of problems that can be solved in polynomial time
- **NP** (*nondeterministic polynomial time*) is the set of problems that can be solved in polynomial time by a *nondeterministic* computer
  - *What the hell is that?*

Nondeterminism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - If a solution exists, computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution
  - So: **NP** = problems *verifiable* in polynomial time

P and NP

- Summary so far:
  - **P** = problems that can be solved in polynomial time
  - **NP** = problems for which a solution can be verified in polynomial time
  - Unknown whether **P = NP** (most suspect not)
- Hamiltonian-cycle problem is in **NP**:
  - Cannot solve in polynomial time
  - Easy to verify solution in polynomial time (*How?*)

NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve hamiltonian-cycle in O(n^{100}) time, you've proved that **P = NP**. Retire rich & famous.

Reduction

- The crux of NP-Completeness is *reducibility*
  - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
    - *What do you suppose "easily" means?*
    - This rephrasing is called *transformation*
  - Intuitively: If P reduces to Q, P is "no harder to solve" than Q

Reducibility

- An example:
  - P: Given a set of Booleans, is at least one TRUE?
  - Q: Given a set of integers, is their sum positive?
  - Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if $x_i =$ TRUE, $y_i = 0$ if $x_i =$ FALSE
- Another example:
  - Solving linear equations is reducible to solving quadratic equations
    - *How can we easily use a quadratic-equation solver to solve linear equations?*

Using Reductions

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Complete:
  - If P is NP-Complete, $P \in$ **NP** and all problems R are reducible to P
  - Formally: $R \leq_p P \ \forall\ R \in$ **NP**
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete
  - This is the *key idea* you should take away today

Coming Up

- Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  - Graph coloring (= register allocation)
  - Hamiltonian cycle
  - Hamiltonian path
  - Knapsack problem
  - Traveling salesman
  - Job scheduling with penalities
  - Many, many more