Greedy Algorithms

Review: Dynamic Programming

- *Dynamic programming* is another strategy for designing algorithms
- Use when problem breaks down into recurring small subproblems

Review: Optimal Substructure of LCS

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- Observation 1: Optimal substructure
  - A simple recursive algorithm will suffice
  - *Draw sample recursion tree from c[3,4]*
  - *What will be the depth of the tree?*
- Observation 2: Overlapping subproblems
  - *Find some places where we solve the same subproblem more than once*

Review: Structure of Subproblems

- For the LCS problem:
  - There are few subproblems in total
  - And many recurring instances of each

    (unlike divide & conquer, where subproblems unique)
- *How many distinct problems exist for the LCS of x[1..m] and y[1..n]?*
- A: *mn*

Memoization

- *Memoization* is another way to deal with overlapping subproblems
  - After computing the solution to a subproblem, store in a table
  - Subsequent calls just do a table lookup
- Can modify recursive alg to use memoziation:
  - There are *mn* subproblems
  - *How many times is each subproblem wanted?*
  - *What will be the running time for this algorithm? The running space?*

Review: Dynamic Programming

- *Dynamic programming*: build table bottom-up
  - Same table as memoization, but instead of starting at (m,n) and recursing down, start at (1,1)
- *Least Common Subsequence*: LCS easy to calculate from LCS of prefixes
  - As your homework shows, can actually reduce space to O(min(m,n))
- *Knapsack problem*: we'll review this in a bit

Review: Dynamic Programming

- Summary of the basic idea:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Variations:
  - "Table" could be 3-dimensional, triangular, a tree, etc.

Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Walking to the Corner
    - Playing a bridge hand
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Activity-Selection Problem

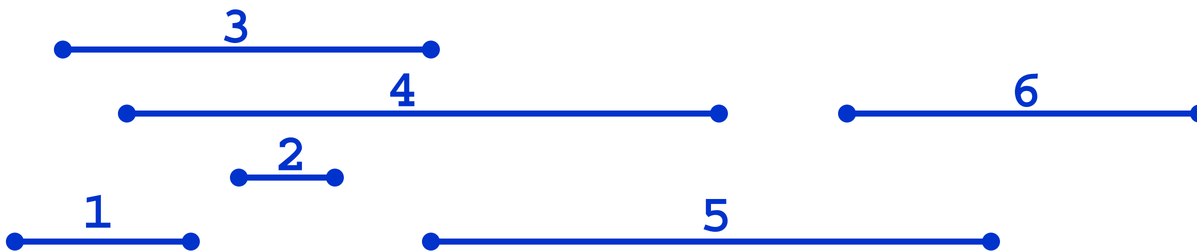
- Problem: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

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



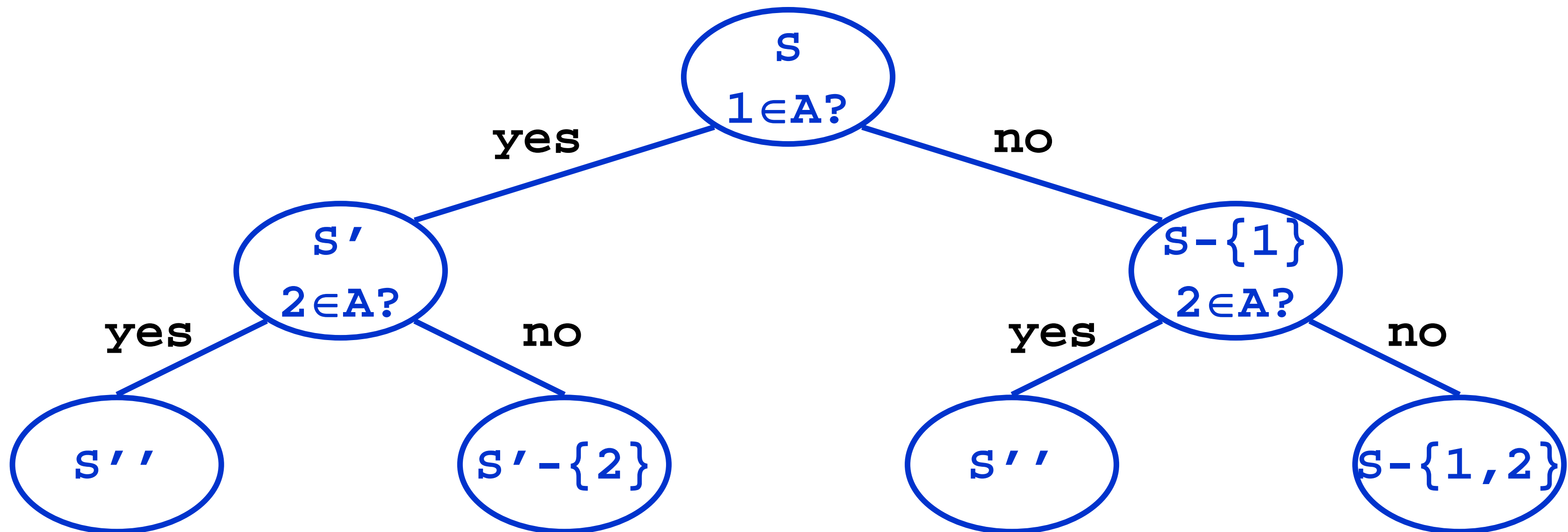
  - Assume (wlog) that $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity Selection: Optimal Substructure

- Let k be the minimum activity in A (i.e., the one with the earliest finish time). Then $A - \{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in S compatible with #1
  - Proof: if we could find optimal solution B' to S' with $|B| > |A - \{k\}|$,
    - Then $B \cup \{k\}$ is compatible
    - And $|B \cup \{k\}| > |A|$

Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



Greedy Choice Property

- 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.

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

Minimum Spanning Tree Revisited

- Recall: MST problem has optimal substructure
  - *Prove it*
- *Is Prim's algorithm greedy? Why?*
- *Is Kruskal's algorithm greedy? Why?*

Review: The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Review: The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The 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
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- 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

Review: The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail*

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - As you have seen, however, it can be solved with dynamic programming