Greedy Algorithms
Greed

• *Locally optimal* decisions are called *greedy*
  ▪ Short sighted strategy (e.g. thinking only one move ahead in a game), usually easy to implement
  ▪ Generally efficient, but…
  ▪ May not lead to a *globally optimal* solution
    • Sometimes, close enough to globally optimal anyway
• What are some examples of greedy algorithms?
  ▪ For the MST problem: Prim’s and Kruskal’s algorithms
  ▪ For the SSSP problem: Dijkstra’s algorithm
    • Remember, Dijkstra only works for graphs with no negative edge weights… why?
Greedy Technique

- Basic steps to finding efficient greedy algorithms:
  - Start by finding a dynamic programming style solution
  - Prove that at each step of the recursion, the min/max can be satisfied by a “greedy choice” (*greedy substructure*)
  - Show that only one recursive call needs to be made once the greedy choice is assumed. This is often natural when all the recursive calls are made by the min/max.
  - Find the recursive solution using the greedy choice
  - Convert to an iterative algorithm if possible

- More generally, taking the direct approach:
  - Show the problem is reduced to a subproblem via a greedy choice
  - Prove there is an optimal solution containing the greedy choice
  - Prove that combining the greedy choice with an optimal solution for the remaining subproblem yields an optimal solution
Examples: Activity Selection

• Given a set of activities $S = \{a_1, \ldots, a_n\}$, where $a_i$ starts at time $s_i \geq 0$ and finishes at time $f_i > s_i$, find a maximal subset $A \subseteq S$ such that, for distinct activities $a_i, a_j \in A$, either $s_i \geq f_j$ or $s_j \geq f_i$
  - In other words, you want to find a largest possible subset of “compatible” activities (that do not overlap)

• Convenient notations:
  - Let $a_0$ be an imaginary activity finishing at time 0
  - Let $a_{n+1}$ be an imaginary activity starting at time $\infty$
  - $S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \}$
  - Observe that $S_{0, n+1}$ contains all activities
Recursive Sol’n to AS Problem
(as in Dynamic Programming)

• Assume activities sorted in increasing order of $f_i$
  ▪ If they are not sorted, can do it in $O(n \lg n)$ time
• Let $c[i, j]$ be the number of activities in the maximal solution for the subset $S_{ij}$
  ▪ In other words, the largest number of compatible activities starting from time $f_i$ finishing before time $s_j$
• Recursively, $c[i,j] =$
  ▪ $0$ if $S_{ij} = \emptyset$, e.g. if $j \geq i$
  ▪ $\max_{i < k < j} \{ c[i,k] + c[k,j] + 1 \}$ otherwise
• What is the runtime of this solution?
Greedy Substructure in AS

• Let $f_m = \min\{f_k : a_k \in S_{ij}\}$. That is, activity $a_m$ has the earliest finishing time in $S_{ij}$
  ▪ Claim 1: $a_m$ is used in some maximal solution for the activities in $S_{ij}$
    • Proof sketch: Suppose $a_k$ is the first activity in some maximal solution. It can safely be removed, and replaced with $a_m$
  ▪ Claim 2: $S_{im} = \emptyset$
    • Proof sketch: Nothing else starting after $a_i$ finishes before $a_m$

• Thus, always safe to include $a_m$, and solve the remaining problem for $S_{mj}$ only
Greedy Sol’n to AS Problem

Recursive-AS( S_{ij} )

// Assumes S_{ij} sorted in order of increasing f_k
if S_{ij} = ∅
    return ∅

m = first activity in S_{ij}
return \{a_m\} \cup \text{Recursive-AS}( S_{mj} )

• Want to compute \text{Recursive-AS}( S_{0,n+1} ) \ldots
  ▪ What is the runtime? Is recursion necessary?
Greedy Sol’n to AS Problem
(Iterative Version)

Greedy-AS( S )
// Assumes S sorted in order of increasing $f_k$
A = \{a_1\} ; \quad i = 1
for m = 2 to n
  if $s_m \geq f_i$ // $a_m$ is compatible with $a_i$
    A = A \cup \{a_m\} ; \quad i = m
return A

• What is the runtime?
Examples: Optimal Prefix Codes

- A prefix code is a prefix-free encoding of a character set (easily represented using a tree)
- Prefix-free binary encodings correspond to placing the characters into leaves on some binary tree
- The number of bits required to encode a string from character set C using a binary prefix code T is given by:
  \[ B(T) = \sum_{c \in C} f(c)d_T(c) \]
  - \( f(c) \) is the number of times character c occurs in the string
  - \( d_T(c) \) is the depth of the leaf containing c
- An optimal binary prefix code for a string is given by a tree T such that the value of \( B(T) \) is minimal for that string
Huffman Codes

• A greedy solution to find the optimal prefix code:

\[
\text{Huffman}( \ C[1 .. n] \ )
\]

// Each entry \( C[i] \) has an associated priority \( f[ i ] \)

\[
Q \leftarrow C \quad // \ Q \text{ is a min-priority queue}
\]

for \( i = 1 \) to \( n-1 \)

allocate a new tree node \( z \)

\[
\text{left}[z] = \text{Extract-Min}(Q)
\]

\[
\text{right}[z] = \text{Extract-Min}(Q)
\]

\[
 f[ z ] = f[ \text{left}[z] ] + f[ \text{right}[z] ]
\]

\[
\text{Insert}(Q, z)
\]

return \( \text{Extract-Min}(Q) \)

• See CLRS for proof of correctness (sketched in class)
Pitfalls: The Knapsack Problem

• The 0-1 knapsack problem: A thief has knapsack that holds at most $W$ lbs. He can steal from a jewelry collection containing $n$ items where the $i$-th item is worth $v_i$ dollars and weighs $w_i$ lbs. What is the most valuable way to pack the knapsack?
  - If the thief is greedy, and packs the most valuable items first, will he get away with the most valuable loot?
  - Can certainly be solved with dynamic programming

• The fractional knapsack problem: As above, but the thief can steal a fraction of each item (e.g. gold dust).
  - Think about dollars per lb, for each kind of item…
  - Is this easier or harder to solve than before? Why?