LECTURE 10

INTRODUCTION TO DYNAMIC PROGRAMMING
Dynamic Programming

Linear Independent Set:

Input: undirected line graph and weights on vertices.

Output: The max weight independent set in G, i.e. a set of mutually non-adjacent vertices with the maximum total weight.

Ex: A B C D

\{A, D\} \rightarrow 1 t3 \leq 4
\{A\} \rightarrow 1
\{A, C\} \rightarrow 2 1 + 6 = 7

max independent set: \{B, D\} = 5 t3 \geq 8

Naive Alg: Search all $2^n$ possible subsets: Check which one has the highest weight and is independent. $\Theta(2^n)$.

Greedy Alg: max-weight next independent vertex greedy alg:
$\Rightarrow \notin \{C, D\}$ $1 + 6 = 7$
Wrong.

DC Approach: Divide in L & R and find max weight IS and try to combine. $\Rightarrow$ Problem need to handle complicated cases.

Can work, but will be slow.
A New (DP) Approach:

Let $S^*$ be the optimal solution. Reason about what $S^*$ looks like in terms of optimal solutions to sub-problem.

Consider $v_n$.

Simple Claim (Tautology): Only 2 possibilities:

Case 1: $v_n \in S^*$

Case 2: $v_n \notin S^*$

Let:

$G_{n-1} = G \setminus \{v_n\} \Rightarrow S^*_{n-1}$

$G_{n-2} = G \setminus \{v_n, v_{n-1}\} \Rightarrow S^*_{n-2}$

$G_1 = \{v_1\} \Rightarrow S^*_1 = \{v_1\}$

(1) If we are in Case 1:

$S^*_n = S^*_{n-1}$ break $v_n$ by assumption is $\in S^*_n$, so every vertex in $S^*_n \in \{v_1, \ldots, v_{n-1}\}$.

$\Rightarrow S^*_n$ is a solution to $G_{n-1}$ and it must be the max-weight solution.
If we are in Case 2, i.e. $v_n \notin S_n^*$

$S_n = S_{n-1} \cup \{v_n\}$

$S_n = S_n^* - \{v_n\} \Rightarrow$ a solution to $G_{n-1}$

and $S_n^*$ must be the max weight IS

in $G_{n-2}$ blc otherwise weight $(S_{n-2}) \cup \{v_n\} > \text{weight}(S_n^*)$.

If $v_n \notin S_n \Rightarrow S_n = S_{n-1}$

If $v_n \in S_n^* \Rightarrow S_n^* = S_{n-2} \cup \{v_n\}$.

Solution to $G_{n+1}$, i.e $S_n^*$, otherwise $S_n^*$ cannot be the max-weight solution to $G_n$. 

contradicting that $S_n^*$ is the max weight IS in $G_n$. 
A possible recursive algo:
\[ \text{LIS-Rec} \ (G_n \ & \text{weights}) \]

Base case:
\[ S_1 = \text{LIS-Rec} \ (G_{n-1}) \]
\[ S_2 = \text{LIS-Rec} \ (G_{n-2}) \cup \{v_n\} \]
\[ \text{return} \ \max \ (S_1, S_2) \]

Example:
\[ T(n) = T(n-1) + T(n-2) + O(1) \]

Exercise: Show that this is \( \Theta(2^n) \).
Q: # distinct recursive calls?
A: only 

Fix 1: Caching of results also called Memoization
⇒ Exercise (show that this makes the runtime to \(\Theta(n)\)).
⇒ But this not \(\overline{DP}\)?

Fix 2: Bottom-up, Iterative Solution Called Dynamic Program.
A: a solution array of size \(n\).

\[
A[i] = \max \text{ weight is to } G_j.
\]

DP-LIS (\(G_j, \text{weights}\)):
Base cases: \(A[0] = 0; A[1] = w,\)
for \(i = 2 \ldots n\)
\[
A[i] = \max \{A[i-1], A[i-2] + w\};
\]
return \(A[n]\. 

\(\Rightarrow\) run time: \(\Theta(n)\. \)
Correctness: follows from the correctness of the recurrence & base cases. (e.g., a rigorous argument requires induction on $i$).

Space: $O(n)$ but can be reduced to $O(1)$ by storing $A[i-1]$ and $A[i-2]$ at each loop iteration.

Reconstruct the actual IS?

Option 1: For each $A[i]$, store the IS for $G[i]$.  
$\Rightarrow$ $O(n^2)$ space.

Option 2: Backtracking:

Observe:
\[ A = \begin{bmatrix} 1 & 3 & 7 & \cdots & 79 & 83 & 85 & 90 & 90 \end{bmatrix} \]

\[ W = \begin{bmatrix} 7 & 3 & 6 & \cdots & 14 & 5 & 2 & 1 & 7 & 3 \end{bmatrix} \]

\( v_1, v_2, \ldots, v_n \)

Is \( v_n \in E_{S_n} \)?

\( v_{n-1} \in E_{S_n} \)

\( v_{n-2} \in S_n \) or \( S_n^c \)

\( v_{n-3} \neq S_n \)

If \( v_n \in E_{S_n} \), \( v_{n-1} \in E_{S_{n-1}} \) vs \( v_n \in E_{S_n} \) if \( v_n \not\in E_{S_n} \)

\( A[S_n] = A[S_{n-1}] + 90 \leq 85 \)


85 vs 83 + 7 = 90

85 vs 83 + 7 = 90

\( v_n \in E_{S_n} \) or \( S_n^c \)
LIS - Reconstruct Alg (G, weights, A)

let $S^* = \emptyset$

in

while ($i > 0$)

if $(A[i-2] + w_i \geq A[i+1])$

$S^* = S^* \cup \{i\}$

$i = i - 2$

else

$i = i - 1$

else

return $S^*$

Backtracking Alg to reconstruct the LIS.

$\Theta(n^2)$. 
Recipe of a DP Alg:

1. Identify a small # subproblems
   (e.g. US had n subp)
   and represent solution to subproblems
   on a 2D or multi dimensional
   solution array.

2. Quickly solve larger subproblems
   using solutions to smaller subproblem
   (establish a recurrence).

3. Solve all subproblems and easily
   compute the final solution to the
   original problem.