Question 2 (cont’d)

Now we will define the subproblems differently. Assume the \( v_i \)'s are positive integers.

For \( i = 0,1, \ldots, n \), let \( V_i = \sum_{t=1}^{i} v_t \). \((V_0 = 0.)\)

For \( i = 0,1, \ldots, n \), and \( v = 0,1, \ldots, V_i \), \( W(i, v) = \) the \textbf{minimum weight} of a subset of items \( \{1,2, \ldots, i\} \) whose \textbf{value} is \( \geq v \).

Compare to the subproblems we defined before: For \( i = 0,1, \ldots, n \), and \( c = 0,1, \ldots, C \), \( K(i, c) = \) the \textbf{maximum value} of a subset of items \( \{1,2, \ldots, i\} \) whose \textbf{weight} is \( \leq c \).
Question 2 (cont’d)

For $i = 0, 1, \ldots, n$, and $v = 0, 1, \ldots, V_i$,
$W(i, v) =$ the minimum weight of a subset of items
{1, 2, \ldots, i} whose value is $\geq v$.

• Give a recursive formula to compute the
  subproblems.

• Describe your DP algorithm in pseudocode.

• Analyze the running time of your algorithm.

• Modify the algorithm to find the actual set of items
  of maximum value whose weight does not exceed
  the knapsack capacity $C$. 
Question 2 — answer (cont’d)

- Recursive formula to compute the subproblems.

**Case 1:** \( v > V_{i-1} \). (Lightest set of items of value \( \geq v \) must use item \( i \).)

\[
W(i, v) = W(i - 1, \max(0, v - v_i)) + w_i
\]

**Case 2:** \( v \leq V_{i-1} \). (Lightest set of items of value \( \geq v \) may or may not use item \( i \).)

\[
W(i, v) = \min(W(i - 1, v), W(i - 1, \max(0, v - v_i)) + w_i)
\]
Describe your DP algorithm in pseudocode.

\[ V[0] := 0; \text{for } i := 1 \text{ to } n \text{ do } V[i] := V[i - 1] + v_i \]
\[ \text{for } i := 0 \text{ to } n \text{ do } W[i, 0] := 0 \]
\[ \text{for } v := 1 \text{ to } V[n] \text{ do } W[0, v] := 0 \]
\[ \text{for } i := 1 \text{ to } n \text{ do } \]
\[ \quad \text{for } v := 1 \text{ to } V[i] \text{ do } \]
\[ \quad \quad \text{if } v > V[i - 1] \text{ then } \]
\[ \quad \quad \quad W[i, v] := W[i - 1, \max(0, v - v_i)] + w_i \]
\[ \quad \quad \text{else } \]
\[ \quad \quad \quad W[i, v] := \min(W[i - 1, v], W[i - 1, \max(0, v - v_i)] + w_i) \]
\[ \text{return } \max\{v: W[n, v] \leq C\} \]
• Analyze the running time of your algorithm.

\[ \Theta(n \cdot \sum_{i=1}^{n} v_i) \]

This is pseudopolynomial.

This version of the algorithm (with subproblems based on value rather than weight) is the basis for a polynomial-time approximation algorithm for knapsack that we will see at the end of the course.
Approximation local search algorithm for max cut
Graph with 8 nodes and 11 edges
A cut of the graph
This cut has 3 cross edges
Node 5 has more **internal** edges (1) than **cross** edges (0).

Increase the number of cross edges by moving it to the blue side.
This cut has 4 cross edges.
Node 7 has more internal edges (2) than cross edges (1).

Increase the number of cross edges by moving it to the yellow side.
This cut has 5 cross edges
Node 3 has more **internal** edges (3) than **cross** edges (1).

Increase the number of cross edges by moving it to the yellow side.
This cut has 7 cross edges.

No local improvement is possible.

But as we will see, it is not a max cut!
Back to the original cut with 3 cross edges.

Now move nodes in a different order.
Node 7 has more internal edges (2) than cross edges (1).

Improve the number of cross edges by moving it to the yellow side (instead of moving node 5 to the blue side, as before).
This cut has 4 cross edges
Node 2 has more internal edges (2) than cross edges (1).

Improve the number of cross edges by moving it to the yellow side.
This cut has 5 cross edges
Node 7 has more internal edges (2) than cross edges (1).

Improve the number of cross edges by moving it to the blue side.

NB: Moving back!
This cut has 6 cross edges
Node 1 has more internal edges (2) than cross edges (0).

Improve the number of cross edges by moving it to the yellow side.
This cut has 8 cross edges
Node 5 has more internal edges (1) than cross edges (0).

Improve the number of cross edges by moving it to the blue side.
This cut has 9 cross edges.

This is a max cut.

Why?

Hint: Disjoint triangles 1,3,8 and 4,6,7.
Approximation algorithm for metric TSP
Step 1: Find a MST of the graph
Step 1: Find a MST of the graph
Step 2: Do a DFS of the MST
(each edge of the MST is visited twice: once when discovered and once again when backtracking)
Step 2: Do a DFS of the MST
Record the sequence of nodes in the order visited

1, 2, 1, 3, 1, 5, 4, 5, 6, 8, 6, 7, 6, 5, 1
Step 3: Keep only the **first** occurrence of each node, then back to the first node

"Trail" produced by the DFS of the MST

Tour produced by the algorithm
Step 3: This is the algorithm's tour
(its cost is at most twice the cost of the optimal tour)
Metric TSP approximation algorithm

1. Find a MST $T^*$ of the graph
2. Do a DFS of $T^*$
3. $S' :=$ sequence of nodes in the order visited by the DFS
   # $S'$ is not a tour
4. $S :=$ subsequence of $S'$ containing only the first occurrence of each node, followed by the first node
   # $S$ is a tour
5. return $S$