Dynamic Programming (Ch. 15)

Dynamic programming solutions rely on the optimal substructure property. Usually the recursive solutions to these problems takes exponential time with many redundant calculations. The Floyd-Warshall algorithm used dynamic programming techniques to compute the APSP problem from the bottom up.

Two more dynamic programming examples from Chapter 15 that we will cover in this lecture:

0/1 Knapsack

Longest-Common-Subsequence

0-1 Knapsack Problem

Suppose we use an approach like that used in the Floyd-Warshall APSPs algorithm:

Define subproblems by using a parameter k so that subproblem k is the best way to fill the knapsack using only items from the set \( T_1 \ldots T_k \).

Derive an equation that takes the best solution using only items from \( T_{k-1} \) and considers how to add the item k to that.

Unfortunately, we can find a counter-example to this approach that shows the global solution obtained in this way may actually contain a suboptimal subproblem solution.

0-1 Knapsack Problem

A better approach is to formulate each subproblem as that of computing \( B[k,w] \), which is defined as the maximum total value of a subset of \( T_k \) from among all those having total weight exactly \( w \).

\[
B[k,w] = \begin{cases} 
B[k-1,w] & \text{if } w_k > w \\
\max(B[k-1,w], B[k-1,w - w_k] + b_k) & \text{otherwise}
\end{cases}
\]

The best subset of \( T_k \) that has total weight \( w \) is either the best subset of \( T_{k-1} \) that has total weight \( w \) or the best subset of \( T_{k-1} \) that has total weight \( w - w_k \) plus the benefit of item \( k \).

This solution is simple (only 2 parameters, \( k \) and \( w \)) and it satisfies the sub-problem optimization condition. The problem \( B[k,w] \) is built from \( B[k-1,w] \) or \( B[k-1,w - w_k] \).

Algorithm 0-1 Knapsack

Input: Set \( T \) of \( n \) items, such that item \( i \) has positive benefit \( b_i \) and positive integer weight \( w_i \); positive integer for maximum total weight \( W \).

Output: For \( w = 0, \ldots, W \), maximum benefit \( B[w] \) of a subset of \( T \) with total weight \( w \). \( B \) is an array indexed from 0 to \( W \).

0-1 Knapsack (T, W)

1. for \( w = 0 \) to \( W \)
2. \( B[w] = 0 \)
3. for \( k = 1 \) to \( n \)
4. for \( w = W \) down to \( w_k \)
5. if \( B[w - w_k] + b_k > B[w] \) then
6. \( B[w] = B[w - w_k] + b_k \)
0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

\( T_1, T_2, T_3, T_4 \)

\( (12, 2), (10, 1), (20, 3), (15, 2) \) and \( W = 5 \)

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

\( T_1, T_2, T_3, T_4 \)

\( (12, 2), (10, 1), (20, 3), (15, 2) \) and \( W = 5 \)

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

\( T_1, T_2, T_3, T_4 \)

\( (3, 2), (5, 4), (8, 5), (4, 3) \) and \( W = 20 \)

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

\( T_1, T_2, T_3, T_4 \)

\( (3, 2), (5, 4), (8, 5), (4, 3) \) and \( W = 20 \)
Given the following (benefit, weight) pairs

1. for w = 0 to W do
2. \( B[w] = 0 \)
3. for k = 1 to n do
4. for w = W downto \( w_k \) do
5. if \( B[w - w_k] + b_k > B[w] \) then
6. \( B[w] = B[w - w_k] + b_k \)

0-1 Knapsack Algorithm Execution

0. Given the following (benefit, weight) pairs
1. (3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20
2. \( T_1 = (3,2), T_2 = (5,4), T_3 = (8,5), T_4 = (4,3) \) and \( T_5 = (10,9) \)
3. \( W = 20 \)
4. \( b_k = 8, w_k = 5 \)
5. \( b_k = 4, w_k = 3 \)
6. \( b_k = 10, w_k = 9 \)

5th iteration: \( b_k = 10, w_k = 9 \)

This verifies what we can confirm through visual inspection of this small set of (benefit, weight) pairs - that the best value for a set that weighs exactly 20 lbs. is the set \( \{1,2,3,5\} \).
0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. \( B[w] = 0 \)
3. for k = 1 to n do
4. for w = W downto \( w_k \) do
5. if \( B[w - w_k] + b_k > B[w] \) then
6. \( B[w] = B[w - w_k] + b_k \)
7. list[w] = list[w - w_k] + k

Complexity of 0-1 Knapsack Solution

Running time is dominated by 2 nested for-loops, where the outer loop iterates n times and the inner one iterates at most W times.

The running time of this algorithm is \( O(nW) \)

The running time of the 0-1 Knapsack algorithm depends on a parameter W that is not proportional to the size of the input.

An algorithm whose running time depends on the magnitude of a number given in the input, not the size of the input set, is called a pseudo-polynomial time algorithm.

Fractional Knapsack Problem (S. 16.2)

**Fractional Knapsack Problem:**
Given items \( T_1, T_2, T_3, \ldots, T_n \) with associated weights \( w_1, w_2, w_3, \ldots, w_n \) and benefit values \( b_1, b_2, b_3, \ldots, b_n \), how can we maximize the total benefit subject to an absolute weight limitation \( W \)?

We can take an amount \( x_i \) of each item \( i \) such that \( 0 \leq x_i \leq w_i \) for each \( i \in T \) and \( \sum x_i \leq W \)

The total benefit is determined by computing the value per unit weight of each item and sorting by that value.

Note that in this problem, unlike the 0-1 Knapsack problem, we are allowed to use arbitrary fractions of an item.

Fractional Knapsack Algorithm

**Input:** Set \( T \) of items (such that each item has a positive benefit and a positive weight) and a positive maximum weight value \( W \).

**Output:** Amount \( x_i \) of each item that maximizes the total benefit while not exceeding the maximum total weight \( W \)

FractionalKnapsack (T, W)
1. for each item \( i \) in \( T \) do
2. \( x_i = 0 \)
3. \( v_i = b_i / w_i \) \{value index of item\}
4. \( w = 0 \)
5. while \( w < W \) do
6. remove from \( S \) item \( i \) with highest \( v_i \)
7. \( a = \min(w, W - w) \)
8. \( x_i = a \)
9. \( w = w + a \)

The running time of the Fractional Knapsack algorithm is \( O(n \log n) \).

Why?
This algorithm uses a greedy approach, not a dynamic programming technique, to find the optimal solution. Which other algorithms did we study that use a greedy approach to find an optimal solution?
01 versus Fractional Knapsack Algorithm

Although these problems are similar, the fractional knapsack problem is solvable in polynomial time using a greedy strategy.

If we use the value per pound greedy strategy to make choices in the 0-1 knapsack problem, we end up with a suboptimal solution.

Longest Common Subsequence Problem (s. 15.4)

Problem: Given $X = <x_1, x_2, ..., x_m>$ and $Y = <y_1, y_2, ..., y_n>$, find the longest common subsequence (LCS) of $X$ and $Y$.

Example:

$X = (A, B, C, B, D, A, B)$
$Y = (B, D, C, A, B)$
$LCS_{XY} = (B, C, B, A)$ (or also $LCS_{XY} = (B, D, A, B)$)

Brute-Force Solution:
1. Enumerate all subsequences of $X$ and check to see if they appear in $Y$.
2. Each subsequence of $X$ corresponds to a subset of the indices $\{1, 2, ..., m\}$ of the elements of $X$ – so there are $2^m$ subsequences of $X$.
3. Clearly, this is not a good approach...time to try dynamic programming!

Recursive Solution to LCS Problem

The recursive LCS Formulation:
- Let $C[i][j]$ = length of the LCS of $X_i$ and $Y_j$, where $X_i = (x_1, x_2, ..., x_i)$ and $Y_j = (y_1, y_2, ..., y_j)$.
- Our goal: $C[m][n]$ (consider entire $X$ and $Y$).
- Basis: $C[0][j] = 0$ and $C[i][0] = 0$.
- $C[i][j]$ is calculated as shown below (two cases):
  
  **Case 1:** $x_i = y_j$.
  In this case, we can increase the size of the LCS of $X_{i-1}$ and $Y_{j-1}$ by one by appending $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$, i.e.,
  $$C[i][j] = C[i-1][j-1] + 1$$

  **Case 2:** $x_i \neq y_j$.
  In this case, we take the LCS to be the longer of the LCS of $X_{i-1}$ and $Y_j$ and the LCS of $X_i$ and $Y_{j-1}$, i.e.,
  $$C[i][j] = \max(C[i][j-1], C[i-1][j])$$

Top-Down DP Solution to LCS Problem

- init $C[i][0] = C[0][j] = 0$ for $i = 0...m$ and $j = 1...n$.
- init $C[i][j] = \text{NIL}$ for $i = 1...m$ and $j = 1...n$.

$LCS(i, j)$
1. if $C[i, j] = \text{NIL}$
2. if $x_i = y_j$ then
3. $C[i, j] = LCS(i-1, j-1) + 1$
4. else
5. $C[i, j] = \max(LCS(i, j-1), LCS(i-1, j))$
6. return $C[i, j]$.

Bottom-Up DP Solution to LCS Problem

We now want to figure out the “right” order to solve the subproblems.

To compute $C[i][j]$, we need the solutions to:
- $C[i-1][j-1]$ (when $x_i = y_j$).
- $C[i-1][j]$ and $C[i][j-1]$ (when $x_i \neq y_j$).

If we fill in the $C$ array in row major order, these dependencies will be satisfied.

$LCS(X, Y)$
1. $m = \text{length}(X)$
2. $n = \text{length}(Y)$
3. for $i = 0$ to $m$ do $C[i][0] = 0$
4. for $j = 0$ to $n$ do $C[0][j] = 0$
5. for $i = 1$ to $m$ do
6. for $j = 1$ to $n$ do
7. if $x_i = y_j$ then
8. $C[i][j] = C[i-1][j-1] + 1$
9. else
10. $C[i][j] = \max(C[i][j-1], C[i-1][j])$
11. return $C[m][n]$.

Bottom-Up LCS DP

Running time = $O(mn)$ (constant time for each entry in $C[i][j]$).

This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?

We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).

If $x_i = y_j$, the answer came from the upper left (diagonal).
If $x_i \neq y_j$, the answer came from above or to the left, whichever value is larger (if equal, default to above).
**Bottom-Up DP Solution to LCS Problem**

**Idea:** Save a pointer to find the path representing the longest common subsequence. Use a 2-dimensional array \( B[i,j] \) to store the pointers (initially this array will be all NIL).

```plaintext
LCS(X, Y)
1. \( m = \text{length}(X) \)
2. \( n = \text{length}(Y) \)
3. for \( i = 0 \) to \( m \) do \( C[i, 0] = 0 \)
4. for \( j = 0 \) to \( n \) do \( C[0, j] = 0 \)
5. for \( i = 1 \) to \( m \) do
6.   for \( j = 1 \) to \( n \) do
7.     if \( x_i = y_j \) then \( C[i, j] = C[i-1, j-1] + 1 \)
8.     \( B[i, j] = "\uparrow" \)
9.   else if \( C[i-1, j] \geq C[i, j-1] \) then
10.    \( C[i, j] = C[i-1, j] \)
11.    \( B[i, j] = "\uparrow" \)
12. else \( C[i, j-1] \)
13.    \( B[i, j] = "\leftarrow" \)
```

**Complexity of LCS Algorithm**

The running time of the LCS algorithm is \( O(mn) \), since each table entry takes \( O(1) \) time to compute.

The running time of the Print-LCS algorithm is \( O(m + n) \), since one of \( m \) or \( n \) is decremented in each stage of the recursion.