Lecture: Dynamic Programming

- Dynamic Programming
  - What is it? How to use it?
  - Longest Common Subsequence (LCS)
  - The Traveling Salesman Problem (TSP)
  - 0-1-Knapsack

These notes are hand-written, unedited and sketchy. They are primarily used for, and based on my lectures.

If you find any bug, impreciseness, or a rare poor-/mis-interpretation of facts, please let me know. I will be grateful for any additional comments you have that are intended to make the quality of the notes better.

Please note that I will provide my hand-written lecture notes only for a subset of my lectures, not for all lectures. Therefore, it is your responsibility to attend all the lectures, take notes regularly, and ask me and/or the TAs if you have any questions.

Thank you!
--Chittu
- Dynamic Programming

- A general and powerful paradigm for algorithm design.

- "Programming" in "Dynamic Programming" means "Table filling."

- Widely used to solve optimization problems:
  - Finding a solution with the optimal value
  - Optimal = maximization/minimization
Longest Common Subsequence (LCS):

Given: Two sequences
\[ X = \langle x_1, x_2, \ldots, x_m \rangle = X[1 \ldots m] \]
\[ Y = \langle y_1, y_2, \ldots, y_n \rangle = Y[1 \ldots n] \]

Find: the longest subsequence common to both \( X \) and \( Y \).

Examples:

\[ \text{HUMAN} \ 
\text{CHIMPANZEE} \ 
\text{TERMINATOR} \ 
\text{THERMOMETER} \ 
\]

\[ \text{ABC} \ 
\text{BCDAB} \ 
\text{ROMANCE} \ 
\text{TOMATO} \ 
\]

- Common subsequence is \( Z[i_1 \ldots k] \) if
  \[ \forall 1 \leq i_1 \leq i_2 \leq \ldots \leq i_k \leq m \]
  and \( 1 \leq j_1 \leq j_2 \leq \ldots \leq j_k \leq n \)
  s.t. \[ X[i_t] = Y[j_t] = Z[t] \forall t \in \{1, \ldots, k\} \]

- \( Z[i_1 \ldots k] = \text{LCS}(X[1 \ldots m], Y[1 \ldots n]) \) if it is the
  longest common subsequence.

Applications:
- Conserved regions in two protein or DNA sequences
- UNIX diff utility
Brute-Force Algorithm for LCS:

1. Subsequence $S_x$ of $X$
   - Check if $S_x$ is also a subsequence of $Y$
     - Keep track of the longest such $S_x$ and output it

   | Time: $S_x$: | $\_\_\_\_\_\_$ | Size: $O(m)$ |
   | $Y$:         | $\_\_\_\_\_\_$ | Size: $n$    |

   • Checking if $S_x$ is a subsequence of $Y$ takes $O(m+n)$ time using a "two-finger" algorithm.

   • # possible $S_x = 2^m$

   • $\therefore$ Time = $O((m+n)2^m)$, Exponential in $m$.

Perpetual Question: Can we do better?
Dynamic programming is our savior here! $\bigcup$

Four Steps of Dynamic Programming:

1. Define subproblems and characterize the structure of an optimal solution. $\rightarrow$ Optimal Substructure
2. Recursively define the value of an optimal solution. $\rightarrow$ Recursive Formulation
3. Recurse and memoize (top-down)
   - OR Compute the value of an optimal solution, bottom-up, using a table.
4. Construct an optimal solution from the computed information.
Dynamic Programming Solution for LCS:

1. **Optimal Substructure:**
   - Let \( x_i = \text{prefix of } X \text{ of length } i = \langle x_1, \ldots, x_i \rangle \)
   - \( y_i = \text{prefix of } Y \text{ of length } i = \langle y_1, \ldots, y_i \rangle \).

2. **Theorem:**
   - Let \( x = \langle x_1, \ldots, x_m \rangle \) and \( y = \langle y_1, \ldots, y_n \rangle \) be two sequences.
   - Let \( z = \langle z_1, \ldots, z_k \rangle = \text{LCS} (X, Y) \). Then,
     1. \( x_m = y_n \Rightarrow z_k = z_m = y_n \land z_{k-1} = \text{LCS} (x_{m-1}, y_{n-1}) \).
     2. \( x_m \neq y_n \Rightarrow \begin{cases} 
(\circlearrowleft) & z_k \neq x_m \Rightarrow z = \text{LCS} (x_{m-1}, y_n) \\
(\circlearrowright) & z_k \neq y_n \Rightarrow z = \text{LCS} (x_m, y_{n-1}) \\
(\circlearrowright) & z_k \neq x_m \neq y_n \Rightarrow z = \text{LCS} (x_{m-1}, y_{n-1})
\end{cases} \)

**Proof:** By contradiction. Exercise!

1. \( \langle x_1, x_2, \ldots \rangle \)
   \( \langle y_1, y_2, \ldots \rangle \)
   \( \langle z_1, \ldots, z_{k-1} \rangle \)
   \( \text{LCS} (x_m, y_n) = \text{LCS} (x_{m-1}, y_{n-1}) \cdot x_m \)

2. \( \langle x_1, x_2, \ldots \rangle \)
   \( \langle y_1, y_2, \ldots \rangle \)
   \( \langle z_1, \ldots, z_k \rangle \)
   \( z_k \neq x_m \Rightarrow \text{LCS} (x_m, y_n) = \text{LCS} (x_{m-1}, y_n) \)

2. \( \langle x_1, x_2, \ldots \rangle \)
   \( \langle y_1, y_2, \ldots \rangle \)
   \( \langle z_1, \ldots, z_{k-1} \rangle \)
   \( z_k \neq y_n \Rightarrow \text{LCS} (x_m, y_n) = \text{LCS} (x_m, y_{n-1}) \)

2. \( \langle x_1, x_2, \ldots \rangle \)
   \( \langle y_1, y_2, \ldots \rangle \)
   \( \langle z_1, \ldots, z_k \rangle \)
   \( z_k \neq x_m \neq y_n \Rightarrow \text{LCS} (x_m, y_n) = \text{LCS} (x_{m-1}, y_{n-1}) \)
Bottom Line: \[
\text{LCS}(X_i, Y_j) = \text{prefix}(\text{LCS}(X, Y)) \]

1 \leq i \leq m, 1 \leq j \leq n

i.e. an optimal solution to a problem (here LCS) instance contains optimal solutions to some of the subproblems, which we solve.

2. Recursive Formulation

- Let \[
L[i, j] = |\text{LCS}(X_i, Y_j)|
\]

- Goal: To compute \[
L[m, n].
\]

- \[
L[i, j] = \begin{cases} 
0 & \text{if } i=0 \text{ or } j=0 \quad \text{(Case Cases)} \\
L[i-1, j-1] + 1 & \text{if } i, j > 0 \land x_i = y_j \quad \text{(Case 1 of RC)} \\
\max \left( \frac{L[i-1, j]}{2a}, \frac{L[i, j-1]}{2b}, \frac{L[i-1, j-1]}{2c} \right) & \text{if } i, j > 0 \land x_i \neq y_j \quad \text{(Case 2 of RC)}
\end{cases}
\]

- Observation: \[
\max \left( \frac{L[i-1, j]}{2a}, \frac{L[i, j-1]}{2b}, \frac{L[i-1, j-1]}{2c} \right) = \max \left( \frac{L[i-1, j]}{2a}, \frac{L[i, j-1]}{2b} \right)
\]

i.e. \(2c\) is already included in \(2a\) and \(2b\).

\[
L[i, j] = \begin{cases} 
0 & \text{if } i=0 \text{ or } j=0 \\
L[i-1, j-1] + 1 & \text{if } i, j > 0 \land x_i = y_j \\
\max \left( \frac{L[i-1, j]}{2a}, \frac{L[i, j-1]}{2b}, \frac{L[i-1, j-1]}{2c} \right) & \text{if } i, j > 0 \land x_i \neq y_j
\end{cases}
\]

Recursion!
Recursion Tree:

- Observation: Many repeated (i.e. overlapping) subproblems.
- Idea: Instead of recomputing, compute once, and memoize, i.e. store in a table for future look up.

- More Observation: What is the size of the table?
  Answer: #distinct LCS subproblems for two strings of lengths m and n = mn.
  ⇒ having a 2D table of dimension m x n works!
3. Recurse and Memoize:
Our first try is a "Recurse Only" algorithm.

* Inefficient Recursive Solution:  
  \[
  \text{RECURSE-DO NOT-MEMOIZE-LCS-LENGTH}(X, Y) \\
  1. m \leftarrow X.\text{length}, \ n \leftarrow Y.\text{length} \\
  2. \text{if } m = 0 \text{ or } n = 0 \\
     \text{return} \\
  3. \text{else return } \text{RDM-LCS-LENGTH}(X, Y, m, n)
  \]

\[
\text{RDM-LCS-LENGTH}(X, Y, i, j) \\
1. \text{if } i = 0 \text{ or } j = 0 \\
2. \text{return } 0 \\
3. \text{else if } x_i = y_j \\
4. \text{return } \text{RDM-LCS-LENGTH}(X, Y, i-1, j-1) + 1 \\
5. \text{else return } \max(\text{RDM-LCS-LENGTH}(X, Y, i-1, j), \text{RDM-LCS-LENGTH}(X, Y, i, j-1))
\]

Time: \text{EXponential! } O(n^2^m)  
- visits all nodes of the recursion tree.
- Does repeat the same subproblem computation over and over again.

Next: Try Recurse and Memoize.
- Memoization saves re-computation.
* Efficient Recurse and MEMOIZE Algorithm: Top-down Recursive with Memoization

**RECURSE-MEMOIZE-LCS-LENGTH**(x, y)
1. m ← x.length, n ← y.length
2. if m = 0 or n = 0
   return
3. for i ← 0 to m
   
   
   Table has dimension L[0...m, 0...n]
   
   Time: \(\Theta(mn(n+1)) = \Theta(mn)\)
4. for j ← 0 to n
5.   \(L[i,j] ← -1\)
6. return **RM-LCS-LENGTH**(x, y, m, n, L)

**RM-LCS-LENGTH**(x, y, i, j, L)
1. if i = 0 or j = 0
2. \(L[i,j] ← 0\)
3. if \(x_i = y_j\)
4. \(L[i,j] ← **RM-LCS-LENGTH**(x, y, i-1, j-1, L) + 1\)
5. else \(L[i,j] ← \max\left(RM-LCS-LENGTH(x, y, i-1, j, L)\right)\)

A simple modification can bring down the time complexity from exponential to polynomial, i.e. \(\Theta(mn)\).

3. if \(x_i = y_j\)
   → 3a. if \(L[i,j] < 0\)
   3b. if \(x_i = y_j\)

**Time:**

1. Step 3 executed at most \(mn\) times.
2. In the worst case, step 6 always executes and makes two recursive calls for each execution of step 3.
3. Therefore, \(\text{Time} = \Theta(2mn) = \Theta(mn)\).
3. Recurse and Memoize vs. Build Dynamic Programming
   Table Bottom-up, i.e., Bottom-up Dynamic Programming:
   - Iterative
   - Table holds all information we need to
     - Print the LCS itself along with its length
     - Contains information on other possible solutions
     - Table need not be initialized fully
     - Tricks to reduce table size can be used

\[
LCS-LENGTH(X, Y) \leftarrow \text{iterative, Bottom-up, Table look-up}
\]

1. \( m \leftarrow X\text{'s length, } n \leftarrow Y\text{'s length } \)
2. for \( i \leftarrow 1 \) to \( m \) \{ \)
3. \( L[i, 0] \leftarrow 0 \)
4. for \( j \leftarrow 1 \) to \( n \)
5. \( L[0, j] \leftarrow 0 \)
6. for \( i \leftarrow 1 \) to \( m \)
7. for \( j \leftarrow 1 \) to \( n \)
8. if \( x_i = y_j \)
9. \( L[i, j] \leftarrow L[i-1, j-1] + 1 \)
10. \( M[i, j] \leftarrow \rightarrow \)
11. else if \( L[i-1, j] \geq L[i, j-1] \)
12. \( L[i, j] \leftarrow L[i-1, j] \)
13. \( M[i, j] \leftarrow \uparrow \)
14. else \( L[i, j] \leftarrow L[i, j-1] \)
15. \( M[i, j] \leftarrow \leftarrow \)
16. Return \( L \) and \( M \)

Time: \( O(mn) \), since each table entry takes \( O(1) \) time to compute.
Space: \( O(mn) \). The table itself.
Example:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

L and M matrices:

\[
x: A B C B D A B
\]

\[
y: B D C A B A
\]

\[
\therefore x_i = y_j
\]

4. Construct the Optimal Solution (i.e. here construct an LCS):

PRINT-LCS \( M, X, i, j \)

1. if \( i = 0 \) or \( j = 0 \)
2. return
3. if \( M[i,j] = \downarrow \)
4. PRINT-LCS \( M, X, i-1, j-1 \)
5. print \( x_i \)
6. else if \( M[i,j] = \uparrow \)
7. PRINT-LCS \( M, X, i-1, j \)
8. else PRINT-LCS \( M, X, i, j-1 \) // i.e. \( \leftarrow \)

Initial Call: PRINT-LCS \( M, X, X \cdot \text{length}, Y \cdot \text{length} \).

Time: \( \Theta(m+n) \). Each step either decrements \( i \) or \( j \) or both.
More Thoughts:

- Can we improve the time complexity?
  - No. \# distinct subproblems = mn in the worst case!

- Can we improve the space complexity?
  - If asked to compute the length of LCS only,
    - Only two rows need to be memorized;
      therefore, space required is \(2 \min(m,n) + O(1)\).
    - With a little more work you can show space: \(\min(m,n) + O(1)\).
  - In 1975, Dan Hirschberg showed how to compute LCS string and its length in \(O(mn)\) time and \(O(m+n)\) space.
    The idea is a clever divide-and-conquer to split the two sequences, and then doing dynamic programs on the fragments.

Variants and Related Problems:

- Find the length of LCS. \?
- Find the LCS string. \{ We've done it! \}
- Find all LCS strings.
- \textbf{DIFF} between two sequences \(X\) and \(Y\): e.g. UNIX diff
  
  \[
  \text{EDIT-DISTANCE} (X,Y) = (|X| - \text{LCS}(X,Y)) \quad \text{DELETES}
  + (|Y| - \text{LCS}(X,Y)) \quad \text{INSERTS}.
  \]
The Travelling Salesman Problem: (TSP)

Given: \( n \) cities \( V = \{1, 2, \ldots, n\} \)

distance/cost between every pair of cities

\[
d_{ij} = \begin{cases} 
0 & \text{if } i = j \\
\neq 0 & \text{if } i \neq j 
\end{cases}
\]

Find: a tour that starts with city 1

visit each city exactly once, and

ends with the city 1 so that

the total distance travelled (i.e., total cost) is the minimum.

Note on \( d_{ij} \):

- metric? Euclidean distance, \( L_p \) distance
- Symmetric?
- Other variants...

Distance/cost matrix \( D = [d_{ij}]_{n \times n} \).

\[
D = \begin{bmatrix}
  d_{11} & d_{12} & \cdots & d_{1n} \\
  d_{21} & d_{22} & \cdots & d_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  d_{n1} & d_{n2} & \cdots & d_{nn}
\end{bmatrix}
\]

TSP graph is a clique (i.e., complete graph)

\[
D = \begin{bmatrix}
  0 & 2 & 4 & 1 & 2 \\
  2 & 0 & 3 & 2 & 3 \\
  4 & 3 & 0 & 4 & 2 \\
  1 & 2 & 4 & 0 & 2 \\
  2 & 3 & 2 & 2 & 0
\end{bmatrix}
\]
Our TSP Variant:
- \( V = \{1, 2, \ldots, n\} \)
- \( \forall i, j \in V, \quad d_{ij} = \begin{cases} 0 & \text{if } i = j \\ > 0 & \text{otherwise} \end{cases} \)
- Start at city 1, visit all the cities exactly once, return to city 1.

Goal: Minimize the total cost of the tour,

\[
\text{i.e. minimize } \sum_{i=1}^{n-1} d_{\pi(i), \pi(i+1)} + d_{\pi(n-1), \pi(0)}.
\]

\( \pi: V \rightarrow V \) is a permutation of the cities traveled in order.

Route Force Algorithm:
- Evaluate every possible permutation of vertices, where each permutation is considered as a tour order.
- Pick a permutation with the minimum total cost.

Analysis:
\[
\text{# permutations (cyclic)} = (n-1)!.
\]
\[
\Rightarrow \text{Time} = O(n!).
\]

Perpetual Question: Can we do better?
The Dynamic Programming Solution: Held and Karp

- We'll see a faster, although exponential time algorithm for TSP.

1. Optimal Substructure:

For a subset of cities \( S \subset V \), with \( 1 \in S \), and given \( j \neq 1 \), \( j \in S \), let

\[
C(S, j) = \text{the cost of the shortest path starting at 1, visiting each city in } S \text{ exactly once, and ending at city } j.
\]

Observation:

1. If \(|S| = 2\), i.e., \( S = \{1, j\} \), for \( j = 2, 3, \ldots, n \), then \( C(S, j) = d_{1j} \).

2. If \(|S| > 2\), then

\[
C(S, j) = \min \left\{ \left\{ \text{Cost of optimal path from 1 to } i, \text{ for some } i \in S - \{j\} \right\} + d_{ij} \right\}
\]

over all \( i \in S - \{j\} \)

i.e. \( C(S, j) = \min \left\{ C(S - \{j\}, i) + d_{ij} \mid i \in S - \{j\} \right\} \)

\[
= \min_{i \in S - \{j\}} \left\{ C(S - \{j\}, i) + d_{ij} \right\}
\]

The min over all \( i \in S - \{j\} \) enforces the checking of all possibilities for finding the best \( i \).
2. **Recursive Formulation:**

\[
C(S, j) = \begin{cases} 
0 & \text{when } S = \{i, j\}, j = 1 \\
\infty & \text{when } |S| > 2, j = 1 \\
\min_{i \in S-i,j} \left\{ C(S-\{i,j\}, i) + dij \right\} & \text{otherwise.}
\end{cases}
\]

3. **Recursively and Memoize:**

- The subproblems are ordered by \( |S| \), i.e., \( 1, 2, \ldots, n \).

\[
\text{TSP}([1, 2, \ldots, n], D) \quad D = [d_{ij}]_{n \times n} \text{ is the distance matrix.}
\]

1. \( C([i, j], 1) \leftarrow 0 \)
2. for \( \beta \leftarrow 2 \) to \( n \)
3. \( \forall \) subset \( S \subseteq [1, 2, \ldots, n] \), with \( |S| = \beta \) and \( 1 \in S \)
4. \( C(S, 1) \leftarrow \infty \)
5. \( \forall j \in S-i,j \)
6. \( C(S, j) \leftarrow \min_{i \in S-i,j} \left\{ C(S-\{i,j\}, i) + dij \right\} \)
7. return \( \min_j \left\{ C([1, 2, \ldots, n], j) + d_{1j} \right\} \)

**Complexity Analysis:**

- **Time:** The two loop in steps 2 and 3 execute \( \sum_{\beta=2}^{n} \beta = O(n^2) \) times.
  - For a fixed \( S \), Step 6 executes \( O(n^2) \) times in the worst case. (Actually, \( O(n^3) \), but \( S = O(n) \), in the worst case).
  - The total time to compute the min cost tour = \( O(n^2 \cdot 2^n) \).
- **Space:** Step 6 requires \( 2^\beta \) to memoize all subsets of \([i, 2, \ldots, n]\) of sizes \( \beta \) and \( \beta-1 \), where \( 2 \leq \beta \leq n \).
  - The space required = \( \Omega(2^n) \).
  - Can you show \#subproblems = \( \Theta(n^2) \)?
0-1-Knapsack:

Given:

<table>
<thead>
<tr>
<th>Item(I)</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$w_1$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>2</td>
<td>$w_2$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$n$</td>
<td>$w_n$</td>
<td>$v_n$</td>
</tr>
</tbody>
</table>

Note: $v_i > 0$, $w_i > 0$, $s_i > 0$.

Knapsack of capacity $W$.

Goal: Determine a subset of items from $I$, that can fit in the knapsack and maximize the total value of items picked. i.e., maximize $\sum_{i=1}^{n} x_i v_i$ subject to $\sum_{i=1}^{n} x_i w_i \leq W$,

where $x_i \in \{0,1\}$. $x_i = 1$ if item $i$ is picked; otherwise $x_i = 0$.

1. Optimal Substructure

Let $M[i,w]$ = the maximum value that a subset of items of $\{1,2,\ldots,i\}$ can have such that the total weight of this subset is $\leq w$.

Then goal is to compute $M[n,w]$.

Observation: Captures the optimal substructure

- If item $i$ is included in $M[i,w]$, then
  
  $M[i,w] = M[i-1, w-w_i] + v_i$

- If item $i$ is not included in $M[i,w]$, then
  
  $M[i,w] = M[i-1, w]$
- Do we know if item \( i \) is included?
  
  No, we do not.
  
  \[ M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w]) \]
  
- What if \( w_i > w \)?
  
  Obviously, we cannot include it in the knapsack.
  
  \[ M[i, w] = M[i-1, w] \quad \text{if } w_i > w \]
  
- Base case:
  
  When \( i = 0 \) (i.e. no item is there to pick)
  
  or \( W = 0 \) (i.e. nothing can fit into the knapsack)
  
  \[ M[i, w] = 0 \]

We are now good to give a recursive formulation.

2. Recursive formulation

\[
M[i, w] = \begin{cases} 
  0 & \text{if } i = 0 \text{ or } w = 0 \\
  M[i-1, w] & \text{if } w_i > w \\
  \max(M[i-1, w-w_i] + v_i, M[i-1, w]) & \text{otherwise}
\end{cases}
\]

3. Recurse and Memoize

\[0-1\text{-KNAPSACK}(w, v, n, W) \quad // w = w_1, \ldots, w_n \quad v = v_1, \ldots, v_n\]

1. if \( n = 0 \) and \( W = 0 \) return
2. return \( \text{RMO1K}(w, v, i, j, M) \)
3. for \( i \leftarrow 0 \) to \( n \)
4. for \( j \leftarrow 0 \) to \( W \)
5. \( M[i, j] \leftarrow -1 \)
6. return \( \text{RMO1K}(w, v, n, W, M) \)

\[ \text{RMO1K}(w, v, i, j, M) \]

1. if \( i = 0 \) or \( j = 0 \)
2. \( M[i, j] \leftarrow 0 \)
3. if \( M[i, j] < 0 \)
4. if \( w_i > j \)
5. return \( \text{RMO1K}(w, v, i-1, j, M) \)
6. else return \( \max(\text{RMO1K}(w, v, i-1, j-w_i, M) + v_i, \text{RMO1K}(w, v, i-1, j, M)) \)
3. **Bottom-up Dynamic Programming**

0-1-KNAPSACK \((w, v, n, W)\)  
// \(w = w_1, w_2, \ldots, w_n\), \(v = v_1, v_2, \ldots, v_n\)

1. if \(n = 0\) or \(W = 0\)
2. return
3. for \(i \leftarrow 0\) to \(n\)
4. \(M[i, 0] \leftarrow 0\)
5. for \(w \leftarrow 0\) to \(W\)
6. \(M[0, w] \leftarrow 0\)
7. for \(i \leftarrow 1\) to \(n\)
8. for \(w \leftarrow 1\) to \(W\)
9. if \(w_i \leq w\)
10. \(M[i, w] \leftarrow M[i-1, w]\)
11. \(T[i, w] \leftarrow N\)  // \(N = \text{Not included}\)
12. else \(M[i, w] \leftarrow \max(M[i-1, w-w_i] + v_i, M[i-1, w])\)
13. if \(M[i, w] = M[i-1, w]\)
14. \(T[i] \leftarrow N\)  // \(N = \text{Not included}\)
15. else \(T[i] \leftarrow I\)  // \(I = \text{Included}\)
16. return \(M\) and \(T\)

4. **Construct the Optimal Solution**

0. Print the content of the knapsack.

PRINT-KNAPSACK \((i, w, T)\)
1. if \(i > 0\) and \(w > 0\)
2. if \(T[i, w] = I\)
3. PRINT-KNAPSACK \((i-1, w-w_i, T)\)
4. Print \(i\)
5. else PRINT-KNAPSACK \((i-1, w, T)\)

To print the content of the knapsack invoke PRINT-KNAPSACK \((0, W, T)\).
**Complexity Analysis:**

**0-1-KNAPSACK**

- **Time:** Size of the table $T$ to be filled is $(n+1)(W+1) = O(nW)$.
  - Each entry takes $O(1)$ time to compute.
  - $\therefore$ Time $= O(nW)$.

- **Space:** Size of the table + constant amount of extra space $= O(nW)$.

**PRINT-KNAPSACK**

- **Time:** $O(n)$ since every recursive call decrements $n$ by 1.

**More Thoughts:**

- What is the time (and space) complexity of 0-1-KNAPSACK in terms of the size (#bits) of the knapsack capacity $W$?
  - Size of $W$ is $\lceil \log W \rceil$ bits.
  - Therefore, 0-1-KNAPSACK runs in exponential time (and space)!

**Variants of 0-1-KNAPSACK:**

- Multiple copies of an item is available i.e. REPETITION allowed.
- Coin-change problem
- Subset-Sum problem
- Partition Problem: $W = \frac{1}{2} \sum_{i=1}^{n} W_i$