Dynamic Programming

- Dynamic Programming is a generic method to design algorithms. It constructs the solution from solutions of “(slightly) smaller” problems.
- E.g. Fibonacci number computation. $F(n) = F(n - 1) + F(n - 2)$.
- However, the recurrence relation is not so obvious in many problems.
- We will consider three examples.
  - Coin Selection
  - Knapsack
  - Longest Common Subsequence
Coin Selection

- **Problem**  Sufficient coins with \( k \) values, \( v_1 < v_2 < \ldots < v_k \). A value \( N \). We want to find a collection of coins so that their values total up to \( N \). There may be multiple coins with the same value in the collection.

- **Naive Solution:**
  For every possible \( 0 \leq c_1, c_2, \ldots, c_k \leq \frac{N}{v_1}, \)
  check whether \( N = c_1 v_1 + \ldots + c_k v_k \).

- Time Complexity: There are \( (\frac{N}{v_1})^k \) possibilities, each requires \( O(k) \) time to check.
  Therefore, the time complexity is \( O \left( (\frac{N}{v_1})^k \times k \right) \).

- Exponential to \( k \). Not a good solution.

- Here, the recurrence relation is not obvious.
Induction:

- Let $S(N)$ be one collection of coins that total up to value $N$.
- Instead of computing a solution $S(N)$ directly, we try to compute $S(N)$ from solutions of $S(i)$, $0 \leq i < N$.

$$S(N) = \text{null}$$

for $j$ from $k$ to 1 do
  if $S(N - v_j) \neq \text{null}$
    $S(N) = S(N - v_j) + (v_j)$
  break

- If $S(i)$ has been correctly computed for $0 \leq i < N$, then the above pseudo-code computes $S(N)$ correctly.
Dynamic Programming:

**Algorithm DP:**
Input: \( v_1, \ldots, v_k; N \).
Output: A collection of values that total up to \( N \).

1. \( S[0] = () \)
2. for \( i \) from 1 to \( N \)
3. \( S[i] = null \)
4. for \( j \) from \( k \) to 1 do
5. if \( i - v_j \geq 0 \) and \( S[i - v_j] \neq null \)
6. \( S[i] = S[i - v_j] + (v_j) \)
7. break
8. output \( S[N] \)

- Correctness of the algorithm:
  1. \( S[0] \) is correct.
  2. If \( S[i] \) is correct for \( 0 \leq i < n \), then \( S[n] \) is also correct.

Therefore, by induction, \( S[i] \) is a correct solution for every \( i \). Hence \( S[N] \) is a correct solution for value \( N \).
• Time complexity:
The major part of the running time is spent on the block consisting of line 5,6,7, which will repeat $kN$ times. Suppose each repeat takes time $T$. The total time complexity is $O(kNT)$.

• Space complexity:
We need a linked list for each $S[i]$, $i = 1, \ldots, N$. Therefore, the space complexity is $N$ linked list.

• The above algorithm and analysis show the basic idea of DP. However, DP is usually done with two steps: one step fills the DP table, and the other step is a backtracking step to construct the solution.
Dynamic Programming with Backtracking

- The following algorithm determines whether a value $N$ can be achieved:

**Algorithm DP:**

Input: $v_1, \ldots, v_k; N$.
Output: true or false.
1. $S[0] \leftarrow true$
2. for $i$ from 1 to $N$
3. $S[i] \leftarrow false$
4. for $j$ from $k$ to 1 do
5. if $i - v_j \geq 0$ and $S[i - v_j]$
6. $S[i] \leftarrow true$
7. break
8. output $S[N]$

- To find the actual collection of coins, we need backtrack in the DP array $S[0..N]$, as follows:
• The Pseudo-code (Suppose $S[N]$ is true.)

**Backtrace**($N$, $S[0..N]$)
1. $i \leftarrow N$
2. while($i > 0$)
3. for $j$ from $k$ to 1 do
4. if $i - v_j \geq 0$ and $S[i - v_j]$
5. print($v_j$)
6. $i \leftarrow i - v_j$
7. break //break for loop

• Correctness:
If $S[i]$ is true, there must be $j$ s.t. $S[i - v_j]$ is true. We print $v_j$ and then the rest of the while loop will print the collection of coins that total up to $i - v_j$. Hence the backtracking is correct if $S[0..N]$ is correct.

• Time Complexity: DP takes $O(kN)$ time. Backtracking takes $O(kN)$ time. In total $O(kN)$ time.

• Space Complexity: DP takes $O(N)$ bits. Backtracking takes $O(1)$. Total space complexity is $O(N)$.

• Here $N$ is not the problem size!!!
Exercise

• Does our algorithm find the minimum number of coins that total up to $N$?
  Positive evidence: we try the coin with the largest value first.
  Negative evidence: using the largest value coin is a good choice at the current step, but it may screw up the future steps.

• If yes, prove it.

• If no, give a counter-example, and try to design an algorithm to find the minimum number of coins.
Knapsack problem

- Knapsack problem (pp. 425 in the textbook) is an extended version of our previous “Coin Selection” problem.

- Suppose a thief enters a store and has a knapsack with a capacity of $W$ (i.e., can hold up to $W$ kilograms).

- There are $n$ items to choose from, with each item having weight $w_i$ and value $v_i$.

- The idea is to fill the knapsack with as much weight as possible and maximize the value of the load. Items cannot be broken down.

- Here all $w_i$ are integers.

- This again can be solved by Dynamic Programming. Consider an optimal solution. If we remove item $j$ from the load, then the remaining load must be the most valuable load possible that uses weight $W - w_j$ from the remaining $n - 1$ items.
• Let $c[i, w]$ be the value of the optimal solution for items 1, ..., $i$ with maximum weight $w$. Then

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

DP:
1. for $i$ from 1 to $n$
2. $c[i, 0] \leftarrow 0$
3. for $w$ from 1 to $W$
4. $c[0, w] \leftarrow 0$
5. for $i$ from 1 to $n$
6. for $w$ from 1 to $W$
7. if $w_i > w$
8. $c[i, w] \leftarrow c[i - 1, w]$
9. else
10. $c[i, w] \leftarrow \max\{v_i + c[i - 1, w - w_i], c[i - 1, w]\}$.
11. output $c[n, W]$. 
Backtracking

- Backtracking assumes $c[i, w]$ has been computed for all valid $i$ and $w$.

- Backtrace($c[0..n, 0..W]$)
  1. $i \leftarrow n$, $w \leftarrow W$.
  2. while $i \neq 0$ and $w \neq 0$
  3.     if $w_i > w$
          // item $i$ is not used
          $i \leftarrow i - 1$
  4.     else if $v_i + c[i - 1, w - w_i] < c[i - 1, w]$
          // item $i$ is not used
          $i \leftarrow i - 1$
  5.     else
          // item $i$ is used
          print($i$)
  6.     $i \leftarrow i - 1$, $w \leftarrow w - w_i$

- Time complexity: Filling DP table $O(nW)$. Backtracking $O(n)$. Total $O(nW)$.

- Space complexity: $O(nW)$.
Longest Common Subsequence

- Are the following two strings similar?
  
  ACCGGTCGAGGCAGCGGAAGCCCGGCCGAA  GTCGTTGGAATGCGTTGGCTCTGTAAGG

- Hamming distance: No, they are not.

  ACCGGTCGAGGCAGCGGAAGCCCGGCCGAA
  || | || || | |
  GTCGTTGGAATGCGTTGGCTCTGTAAGG

- In another sense, using LCS (pp. 390 in the textbook), they are:

  ACCGGTCGAGGCAGCGGAAGCCCGGCCGAA  GCCG  GC  C  G  AA
  || || | || || || | | |
  GTCGTTGGGAATGCGTTGGCTCTGTAAGG

- In Bioinformatics, when people compare two DNA (or protein) sequences, hamming distance cannot be used because one sequence may be resulted from inserting a few letters into another.
Longest Common Subsequence

• Instead, measurements like edit distance, sequence alignment, or longest common subsequence are used. Sequence alignment is the most common.

• We introduce the longest common subsequence since it is easier to understand.

• We are given two sequences of characters, $A$ and $B$, $A = a_1a_2\ldots a_n$, $B = b_1b_2\ldots b_m$. (Assume that the characters come from a finite alphabet.) We want to find the longest common subsequence.

• That is, $a_{i_1}a_{i_2}\ldots a_{i_k} = b_{j_1}b_{j_2}\ldots b_{j_k}; i_1 < i_2 < \ldots < i_k, j_1 < j_2 < \ldots < j_k$; and $k$ is maximized.

• We again use Dynamic Programming to solve this problem.
• Let us look at $a_n$ and $b_m$. There are two cases:
  • Case 1. $a_n = b_m$. Then $LCS(A, B)$ is $LCS(a_1 \ldots a_{n-1}, b_1 \ldots b_{m-1}) + a_n$.
  • Case 2. $a_n \neq b_m$. Then $LCS(A, B)$ is either the $LCS(a_1 \ldots a_{n-1}, B)$ or $LCS(A, b_1 \ldots b_{m-1})$, depending on which one is longer.

• Let $L(i, j)$ be the length of $LCS(a_1 \ldots a_i, b_1 \ldots b_j)$. Then
  \[
  L(i, j) = \begin{cases} 
  1 + L(i-1, j-1), & a_i = b_j \\
  \max\{L(i-1, j), L(i, j-1)\} & a_i \neq b_j
  \end{cases}
  \]

• Pseudo-code
  1. for $i$ from 0 to $n$
  2. \hspace{1cm} $L[i, 0] \leftarrow 0$,
  3. for $j$ from 0 to $m$
  4. \hspace{1cm} $L[0, j] \leftarrow 0$,
  5. for $i$ from 1 to $n$
  6. \hspace{1cm} for $j$ from 1 to $m$
  7. \hspace{2cm} if $a_i = b_j$
  8. \hspace{3cm} $L(i, j) \leftarrow 1 + L(i-1, j-1)$
  9. \hspace{2cm} else
  10. \hspace{3cm} $L(i, j) \leftarrow \max\{L(i-1, j), L(i, j-1)\}$
Exercise

• Write the backtracking pseudo-code for the LCS problem.

• In the textbook (pp. 394), in addition to an array $c[i, j]$ (the same as our $L[i, j]$), an array $b[i, j]$ is used to assist the backtracking. Is it necessary?

• Do we need another array for backtracking?

• The time and space complexity of the DP for LCS.

• Read the other dynamic programming algorithms introduced in Chapter 15 of the textbook.