Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. In this lecture, we discuss this technique, and present a few key examples. Topics in this lecture include:

- The basic idea of Dynamic Programming.
- Example: Longest Common Subsequence.
- Example: Subset Sum Problem.

Introduction

Dynamic Programming is a powerful technique that can be used to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time. (Usually to get running time below that-if it is possible-one would need to add other ideas as well.) Dynamic Programming is a general approach to solving problems, much like "divide-and-conquer" is a general method, except that unlike divide-and-conquer, the subproblems will typically overlap. This lecture we will present two ways of thinking about Dynamic Programming as well as a few examples. There are several ways of thinking about the basic idea.

**Basic Idea (version 1):** What we want to do is take our problem and somehow break it down into a reasonable number of subproblems (where "reasonable" might be something like $n^2$) in such a way that we can use optimal solutions to the smaller subproblems to give us optimal solutions to the larger ones. Unlike divide-and-conquer (as in mergesort or quicksort) it is OK if our subproblems overlap, so long as there are not too many of them.

**Example 1: Longest Common Subsequence**

**Definition 1** The Longest Common Subsequence (LCS) problem is as follows. We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a
contiguous block) in both strings. For example, consider:

S = ABAZDC  
T = BACBAD

In this case, the LCS has length 4 and is the string ABAD. Another way to look at it is we are finding a 1-1 matching between some of the letters in S and some of the letters in T such that none of the edges in the matching cross each other.

For instance, this type of problem comes up all the time in genomics: given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up. Let’s now solve the LCS problem using Dynamic Programming. As subproblems we will look at the LCS of a prefix of S and a prefix of T, running over all pairs of prefixes. For simplicity, let’s worry first about finding the length of the LCS and then we can modify the algorithm to produce the actual sequence itself.

So, here is the question: say LCS[i,j] is the length of the LCS of S[1···i] with T[1···j]. How can we solve for LCS[i,j] in terms of the LCS’s of the smaller problems?

**Case 1:** what if S[i] ≠ T[j]? Then, the desired subsequence has to ignore one of S[i] or T[j] so we have:

LCS[i, j] = max(LCS[i−1, j], LCS[i, j−1]).

**Case 2:** what if S[i] = T[j]? Then the LCS of S[1···i] and T[1···j] might as well match them up. For instance, if I gave you a common subsequence that matched S[i] to an earlier location in T, for instance, you could always match it to T[j] instead. So, in this case we have:

LCS[i, j] = 1 + LCS[i−1, j−1].

So, we can just do two loops (over values of i and j), filling in the LCS using these rules. Here’s what it looks like pictorially for the example above, with S along the leftmost column and T along the top row.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

We just fill out this matrix row by row, doing constant amount of work per entry, so this takes O(mn) time overall. The final answer (the length of the LCS of S and T) is in the lower-right corner.

How can we now find the sequence? To find the sequence, we just walk backwards through matrix starting the lower-right corner. If either the cell directly above or directly to the right
contains a value equal to the value in the current cell, then move to that cell (if both to, then chose either one). If both such cells have values strictly less than the value in the current cell, then move diagonally up-left (this corresponds to applying Case 2), and output the associated character. This will output the characters in the LCS in reverse order. For instance, running on the matrix above, this outputs DABA.

More on the basic idea, and Example 1 revisited

For example, for the LCS problem, using our analysis we had at the beginning we might have produced the following exponential-time recursive program (arrays start at 1):

```c
LCS(S,n,T,m) {
    if (n==0 || m==0) return 0;
    if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
    else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
    return result;
}
```

This algorithm runs in exponential time. In the memoized version, we store results in a matrix so that any given set of arguments to LCS only produces new work (new recursive calls) once. The memoized version begins by initializing arr[i][j] to unknown for all i,j, and then proceeds as follows:

```c
LCS(S,n,T,m) {
    if (n==0 || m==0) return 0;
    if (arr[n][m] != unknown) return arr[n][m]; // <- added this line (*)
    if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
    else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
    arr[n][m] = result; // <- and this line (**)  
    return result;
}
```

All we have done is saved our work in line (**) and made sure that we only embark on new recursive calls if we haven’t already computed the answer in line (*). In this memoized version, our running time is now just O(mn). One easy way to see this is as follows. First, notice that we reach line (**) at most mn times (at most once for any given value of the parameters). This means we make at most 2mn recursive calls total (at most two calls for each time we reach that line). Any given call of LCS involves only O(1) work (performing some equality checks and taking a max or adding 1), so overall the total running time is O(mn). Comparing bottom-up and top-down dynamic programming, both do almost the same work. The top-down (memoized) version
pays a penalty in recursion overhead, but can potentially be faster than the bottom-up version in situations where some of the subproblems never get examined at all. These differences, however, are minor: you should use whichever version is easiest and most intuitive for you for the given problem at hand.

More about LCS: Discussion and Extensions.

An equivalent problem to LCS is the "minimum edit distance" problem, where the legal operations are insert and delete. (E.g., the unix "diff" command, where S and T are files, and the elements of S and T are lines of text). The minimum edit distance to transform S into T is achieved by doing $|S| - LCS(S, T)$ deletes and $|T| - LCS(S, T)$ inserts. In computational biology applications, often one has a more general notion of sequence alignment. Many of these different problems all allow for basically the same kind of Dynamic Programming solution.

Example 2: Subset Sum Problem

Assume that we are given some integers, such as 7, 3, 2, 5, 8, and we wish to know whether some of these integers sum up to zero. In this example, the answer is 'yes', since the subset of integers -3, -2, 5 corresponds to the sum $(-3) + (-2) + 5 = 0$. The task of deciding whether such a subset with sum zero exists is called the subset sum problem.

As the number of integers that we feed into the algorithm becomes larger, the number of subsets grows exponentially, and thereby implementing a recursive solution takes at least $O(2^n)$ running time complexity.

The recursive solution is provided using the following pseudocode:

```c
isSubsetSum(int set[], int n, int sum) {
    if (sum == 0) {
        return true;
    } //if
    if (n == 0 && sum != 0) {
        return false;
    } //if
    if (set[n-1] > sum) {
        return isSubsetSum(set, n-1, sum);
    } //if
    return isSubsetSum(set, n-1, sum) ||
          isSubsetSum(set, n-1, sum-set[n-1]);
} //isSubsetSum
```
Using dynamic programming and memoization technique, the solution is provided below:

```java
isSubsetSum(int set[], int n, int sum) {
    for (int i=0; i<sum; i++) {
        subsetsum[0][i] = 0;
    } //for
    for (int i=0; i<n; i++) {
        subsetsum[i][0] = 1;
    } //for
    for (int i=1; i<n; i++) {
        for (int j=1; j<sum; j++) {
            subsetsum[i][j] = max(
                subsetsum[i-1][j],
                subsetsum[i-1][j-set[j]]
            );
        } //for
    } //for
    return subsetsum[n][sum];
} //isSubsetSum
```

An example of Subset Sum Problem is shown below:

<table>
<thead>
<tr>
<th>sum</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

```

0 1 2 3 4 5 6 7
0 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0
2 1 1 1 1 0 0 0
3 1 1 1 1 1 1 1
4 1 1 1 1 1 1 1
```