Heuristics for choosing a subproblem

Because dynamic Programming is an algorithmic design technique, and not an algorithm, it requires a lot of practice to master. Getting started on a dynamic programming solution requires a bit of intuition for choosing the right subproblem. Below are some heuristics to consider for choosing your subproblem. These are by no means exhaustive, but they are common patterns found in a variety of dynamic programming problems and may be useful to keep in mind.

Numerical inputs

If your input is an integer, the subproblem usually considers smaller integers. Consider the rod-cutting problem, where the goal is to find the maximum value you can get form a rod of size $L$. The subproblem $r(j)$ is defined as the maximum value you can get from a rod of size $j$. Another good example of this is the Knapsack problem (the version where you consider replacements).

When you have a sequence or list of items

If the problem is to find the maximum value over some input sequence $A$, there are three common approaches:

Prefix approach $V(j) = \max\text{ over list } A[1: j]$ (first $j$ elements of $A$)

Suffix approach $V(j) = \max\text{ over list } A[j :]$ (last $j$ elements of $A$)

Interval approach $V(i, j) = \max\text{ over list } A[i : j]$ (elements $i$ to $j$ of $A$)

Often, if you can use the prefix approach to solve a problem, you can also use the suffix approach (try it with the Longest Common Subsequence problem!)

The coin-collecting problem can be seen as a 2-dimensional example of using prefixes. The Edit Distance problem and the Longest Common Subsequence problem also use the prefix/suffix approach, though they keep track of prefixes for two separate sequences.

Two good examples of the interval approach are the Matrix Chain Multiplication problem (from lecture 21) and the Placing Parentheses problem (from recitation 21).

There is another variation on prefixes where you consider the sequence $A[1: j]$, but you are forced to include item $j$. An example is the longest increasing subsequence problem found here: https://people.cs.clemson.edu/~bcdean/dp_practice/dp_3.swf

Graph inputs

For graphs, your subproblem might choose a node and then consider only paths that end on that node, like in the longest paths in a DAG problem. Alternatively, you can do only paths that start on that node. For the dynamic programming solution for finding shortest paths in a graph, the subproblem also had another parameter $k$, which was the max number of edges allowed in the path.

If the input is a tree, another common subproblem takes a node $u$ and considers the subtree rooted at $u$. 
Keeping track of additional state

In some problems, you may have to keep track of some extra state. This can be achieved by adding an extra variable. A good example of this is problem 6-1 from pset 6, Fall 2014.

Combining strategies

Sometimes you have to combine multiple of these strategies together. For example, the Knapsack problem without replacement has both a numerical input (the knapsack’s capacity) and a list of items. Not surprisingly, the subproblem has two variables: one corresponds to the integer maximum weight (capacity) and the other corresponds to a prefix of the list of items.

Edit Distance

Given two strings \( A[1, 2, \ldots, n] \) and \( B[1, 2, \ldots, m] \), the edit distance between \( A \) and \( B \) is defined as the minimum number of edits we must make to \( A \) to get \( B \). The allowed edits are insertion, in which we insert a new character in the middle of \( A \); deletion, in which we delete a character of \( A \); and substitution, in which we replace a character of \( A \) with a different one.

As an example, if \( A \) is the string “cat” and \( B \) is the string “goat”, the edit distance between \( A \) and \( B \) is 2, because we can use the following operations to get from \( A \) to \( B \):

1. insert a ‘g’ at the beginning, giving “gcat”
2. substitute ‘o’ in place of the second character, giving “goat”.

It is interesting to note that the edit distance from \( A \) to \( B \) is the same as the edit distance from \( B \) to \( A \). This is because if we have a way of editing \( A \) to get \( B \), we can undo those edits to go from \( B \) to \( A \) (using deletes in place of insert and vice versa).

The edit distance has many applications, including measuring the similarity between gene sequences, automatically correcting spelling mistakes, and catching students who copy each others’ pset code.

Computing the Edit Distance

Given \( A \) and \( B \), how can we compute the edit distance between them? We can use dynamic programming, similar to the longest common subsequence problem.

Let \( S[i][j] \) be the edit distance between the first \( i \) characters of \( A \) and the first \( j \) characters of \( B \). We write a recurrence for \( S[i][j] \). If \( A[i] = B[j] \), we have \( S[i][j] = S[i-1][j-1] \), because we can edit the first \( i - 1 \) characters of \( A \) to get the first \( j - 1 \) characters of \( B \). Otherwise, we must get the last character of \( B \) by either inserting it (giving \( S[i][j] = S[i][j-1]+1 \)), deleting the last character of \( A \) (giving \( S[i][j] = S[i-1][j]+1 \)), or substituting the last character of \( A \) for the last character of \( B \) (giving \( S[i][j] = S[i-1][j-1]+1 \)). We get
\[ S[i][j] = \begin{cases} 
S[i-1][j-1] & \text{if } A[i] = B[j] \\
1 + \min\{S[i][j-1], S[i-1][j], S[i-1][j-1]\} & \text{otherwise.}
\end{cases} \]

We can then build the table \( S \) using constant time per entry, for a total running time of \( O(mn) \). This dynamic programming solution also uses \( O(mn) \) memory.

**Weighted Edit Distance**

For some applications, we need a different type of edit distance: we may want the operations insert, delete, and substitute to cost different amounts. We may even want those costs to depend on the characters we’re inserting or deleting. For example, perhaps deleting the character ‘a’ costs 5, but deleting the character ‘b’ costs only 3. Can we compute the edit distance between two strings given arbitrary costs?

Suppose the costs are provided as three functions: \( I(x) \) gives the cost of inserting the character \( x \), \( D(x) \) gives the cost of deleting the character \( x \), and \( \text{Sub}(x, y) \) gives the cost of substituting \( x \) with \( y \). We’ll assume all these costs are non-negative. We can then modify the recurrence for \( S[i][j] \) to take into account these new weights. As before, if \( A[i] = B[j] \) we have \( S[i][j] = S[i-1][j-1] \).

If \( A[i] \neq B[j] \), there are three options: either \( B[j] \) will be inserted, or \( A[i] \) will be deleted, or \( A[i] \) will be substituted with \( B[j] \). We get the following recurrence.

\[
S[i][j] = \begin{cases} 
S[i-1][j-1] & \text{if } A[i] = B[j] \\
\min\{S[i][j-1] + I(B[j]), S[i-1][j] + D(A[i]), S[i-1][j-1] + \text{Sub}(A[i], B[j])\} & \text{otherwise.}
\end{cases}
\]

We can compute this recurrence with the same \( O(mn) \) running time as before. The weights didn’t change much!

**LCS with Three Strings**

Now let’s try a different variant: let’s find the longest common subsequence that’s common among three given strings, \( A[1, 2, \ldots, n] \), \( B[1, 2, \ldots, m] \), and \( C[1, 2, \ldots, p] \). Can we use dynamic programming to solve this variant?

**The Recurrence**

The first step in the solution will be to give a recurrence for \( S[i][j][k] \), the length of the longest common subsequence between the first \( i \) characters of \( A \), \( j \) characters of \( B \), and \( k \) characters of \( C \). If the last characters are equal (that is, \( A[i] = B[j] = C[k] \)), we can use that character in the subsequence; we thus get \( S[i][j][k] = S[i-1][j-1][k-1] + 1 \). Otherwise, at least one of \( A[i] \), \( B[j] \), and \( C[k] \) are not used in the longest common subsequence. We conclude
\[
S[i][j][k] = \begin{cases} 
1 + S[i-1][j-1][k-1] & \text{if } A[i] = B[j] = C[k] \\
\max\{S[i][j][k-1], S[i][j-1][k], S[i-1][j][k]\} & \text{otherwise.}
\end{cases}
\]

**Filling the Table**

Now that we have a recurrence, we can use it to fill the three-dimensional table \(S\). We need to make sure that by the time we fill the entry \(S[i][j][k]\), the table entries \(S[i-1][j][k]\), \(S[i][j-1][k]\), \(S[i][j][k-1]\), and \(S[i-1][j-1][k-1]\) have already been filled. One way of doing this is with three nested for loops:

```plaintext
for i = 1..n:
    for j = 1..m:
        for k = 1..p:
            Fill the cell \(S[i][j][k]\)
```

This way of iterating ensures that when the triple \(i, j, k\) is reached, all table entries will smaller values of \(i, j,\) or \(k\) have been filled.

**Running Time**

This solution takes constant time to fill each table entry, so the total running time is \(O(mnp)\). The space used is also \(O(mnp)\).