Dynamic Programming

Let us consider a simple question, compute the n^{th} Fibonacci number. The classically bad approach is

```python
def fib1(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

The runtime of this is 2^n, as when we compute for each n we compute two more. But note that we call `fib` on smaller numbers. We call those "subproblems" as they are in the same *format* of the problem we are trying to solve except with smaller inputs.

The above solution is what we call an **divide and conquer** solution as it splits the problem into smaller sub-problems which it then solve. Divide and conquer algorithms work well when all the subproblems are unique, but when there are **overlapping subproblems** as is the case of the Fibonacci's, divide and conquer is doing a lot of extra work that it doesn't have to. To solve this, we use dynamic programming.

Dynamic programming approaches allows us to only solve the **distinct** subproblems and drastically decrease the runtime. There are two ways to do this, one is known as the top-down memoized approach, and the other is the bottom-up tabular approach:

```python
cache = {0: 1, 1: 1}
def fib2(n):
    if n in cache:
        return cache[n]
    else:
        cache[n] = fib2(n - 1) + fib2(n - 2)
        return cache[n]

def fib3(n):
    fibs = [0, 1]
    for i in xrange(2, n)
        fibs.append(fibs[i - 1] + fibs[i - 2])
    return fibs[-1]
```

In the first example we "memoize" or remember the unique subproblems we've solved, and use them to solve new problems, and in the second example we solve all the unique subproblems in a sorted order and store them in a table.

Memoization is often times easier to do, but creates a lot of memory overhead and is often less versatile, as often times we use the structure of the subproblems to our advantage which can only be gained by a tabular structure. To see these in action, let us step through some examples.

Matrix Chain Multiplication

Problem: Given a list of matrices $\langle A_1, A_2, \ldots, A_n\rangle$, in what order should we multiply so that the total number of computations is the lowest.

Subproblem: Given a list of matrices $\langle A_i, A_{i+1}, ..., A_{i+j}\rangle$, how should we parenthesize so that the total number of computations is the lowest.

Substructure: In order to parenthesize $\langle A_i, \ldots, A_{i+j}\rangle$, there exists a k between i and j such that the total cost of multiplication by doing $(A_iA_{i+1} \ldots A_{i+k})(A_{i+k+1} \ldots A_{i+j})$ is the lowest. Note that we know the cost of $A_i \ldots A_{i+k}$ as that is a shorter sequence and we are building a table from the ground up.

Base Case: if $j = 0$ then the cost is 0.

Algorithm:

```python
def min_operations_needed(p, n):
    # the p parameter is a list of length n+1 where the ith
    # matrix (0 indexed) has p p[i] x p[i+1]

    # min_op_lookup[i][j] == minimum number of operations required to
    # compute (A_i ... A_i+j)
    min_op_lookup = []

    for i in xrange(n):
        min_op_lookup.append([0]) # min_op_lookup[i][0] = 0, base case

    for j in xrange(1, n): # j is chain length
        for i in xrange(0, n - j): # compute min_op_lookup[i][j]
            min_val = 1000000 # large number
            for k in xrange(0, j):
                divide_at_k = (min_op_lookup[i][k] +
                               min_op_lookup[i+k+1][j-k-1] +
                               p[i]*p[i+k+1]*p[i+j+1])

                if divide_at_k < min_val:
                    min_val = divide_at_k

            min_op_lookup[i].append(min_val)

    return min_op_lookup[0][-1] # last element in first row
```

0-1 Knapsack Problem

Problem: Given a knapsack of capacity W, a list of n items each with value v_i and weight w_i, find an algorithm to maximize

$$\sum_{i=1}^{n} v_i x_i \text{ subject to } \sum_{i=1}^{n} w_i x_i \text{ and } x_i \in \{0, 1\}$$

Subproblem: maximize the list under the constraint that the total weight is less than j and only use up to item i, store the result of 2d array dp

Substructure: when we compute $dp[i][j]$, we can either include the i^{th} item or not. If $w_i > j$, then it is equal to $dp[i-1][j]$. If it is not, then we have

$$dp[i][j] = \max(m[i-1,j], m[i-1, j-w_i] + v_i)$$

Base case: using no items means we have 0 value

Algorithm:

```
def max_value(v, w, n, W):
    # v = list of values
    # w = list of weights
    # n = number of items
    # W = capacity of knapsack

    dp = [[0 for i in xrange(W)] for j in xrange(n)] # initialize to n x W

    for i in xrange(1, n):
        for j in xrange(W):
            if w[i] > j:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])

    return dp[n][W - 1]
```

Edit Distance

Problem: Given two strings, how many *edits* do we need to change one into the other (edits include insertion, deletion and substitution).

Subproblem: in order to edit one string to fit another, we must edit its substring to fit the substring of the other.

Substructure: at every letter, we can either insert, delete or substitute, we know the cost to get the prefix of the two strings to be the same, so we pick the path that is the easiest.

Base Case: if one string is empty then the edit distance is the length of the other.

Algorithm:

```
int levenshtein_distance(string &a, string &b) {
    // computes the levenshtein edit distance (insertion, deletion and
    // substitution) of two strings
    int dist_lookup[a.size()][b.size()];

    for (int i = 0 ; i < a.size() ; i++) {
        dist_lookup[i][0] = i;
    }

    for (int j = 0 ; j < b.size() ; j++) {
        dist_lookup[0][j] = j;
    }

    for (int i = 1 ; i < a.size() ; i++) {
        for (int j = 1 ; j < b.size() ; j++) {
            int sub_cost = (a[i] == b[j]) ? 0 : 1;
            int cost = dist_lookup[i-1][j] + 1; // deletion
            cost = min(cost, dist_lookup[i][j-1] + 1); // insertion
            // substitution
            cost = min(cost, dist_lookup[i-1][j-1] + sub_cost);
            dist_lookup[i][j] = cost;
        }
    }

    return dist_lookup[a.size() - 1][b.size() - 1];
}
```

Longest Increasing Subsequence

Problem: Given an array of integers, define a subsequence as the result of deleting some elements in the array without changing the order of the remaining elements, what is the longest increasing subsequence of any given array

This problem is slightly more involved, we calculate the longest increasing subsequence of length 1, 2, 3, ... up to the longest one that exists. We also keep track of the values at the endpoints, which are the lowest last number of the LIS of length i. And since we are working with strictly increasing subsequences, the list of endpoints will be sorted, which allows us to perform a binary search, reducing the total runtime to $n \log n$

```
int LIS(vector<int> a) {
    int endpoints[a.size() + 1]; // endpoints[j] is the index of the
   endpoint of lis of length j
    int lis_len = 0;

    for (int i = 0 ; i < a.size() ; i++) {

        // binary search for the largest j such that the endpoint value is
     less
        // than the current value

        int lo = 1;
        int hi = lis_len;

        while (lo <= hi) {
            int mid = (lo + hi + 1) / 2;

            if (a[endpoints[mid]] < a[i]) {
                lo = mid + 1;
            }
            else {
                hi = mid - 1;
            }
        }

        // lo = length of longest prefix of a[i] + 1
        endpoints[lo] = i;

        if (lo > lis_len) {
            lis_len = lo;
        }
    }

    return lis_len;
}
```

Conclusion

These problem are what are known as the classic dynamic programming problems. There are many more as DP is more of a mindset rather than a specific technique. But in general DP can be applied to problems with the following two properties:

- Overlapping Subproblems: the larger problem can be solved by solving the same problem with smaller inputs
- Optimal Substructure: the optimal solution to the large problem is comprised of the optimal solutions of those smaller problems

For problems with those properties, our next step is to find the base case and the recurrence relationship (mathematically), and finally we write up the code.