## Dynamic Programming

Dynamic programming is a technique that appears in many forms in lots of vastly different situations. The core idea is a simple application of divide-and-conquer. First, we split the problem into several parts, all of which are smaller instances of the same problem. This gives us a recursive solution. And then we simply run the recursive solution, making sure never to solve the same sub-problem twice. This is best shown with an example (or a few).

## Catalan Numbers and Memoization

How many binary search trees are there that contain n different elements? Remember that a binary search tree always obeys the rule of having smaller elements to the left of larger ones. For example, there are 2 BST's with 2 nodes (2 choices for the root) and 5 BST's with 3 nodes. Let us call the number of BST's with n nodes $C_n$ .

Whenever you have a problem involving trees, it's always a good idea to think about a recursive solution. Here is one. We will pick one of the n nodes to be the root. If the nodes are numbered from 1 to n, imagine picking node number i. This splits the problem nicely into 2 sub-problems. In the left subtree, we have a BST on i-1 nodes, and there are $C_{i-1}$ ways of building those. In the right subtree, we have n-i nodes and so, $C_{n-i}$ ways of building a BST. The two subtrees are independent of each other, so we simply multiply the two numbers.

That is if we have chosen node i to be the root. Picking a different root is sure to give us a different tree, so to get all the possible trees, we pick all the possible root nodes and multiply the numbers for the two subtrees that we get. This gives us a neat, recursive formula.

$$C_n = \sum_{i=1}^{n} C_{i-1} C_{n-i}$$

Of course, we need some base cases. Simply setting $C_0$ to be 1 is enough – there is just one way to make a tree with zero nodes.

A mathematician would now try to get a closed form solution to this recurrence, but we have computers. What will happen if we simply write a recursive function to calculate $C_n$ ? Here it is.

```
int C( int n ) {
    if( n == 0 ) return 1;
    int ans = 0;
    for( int i = 1; i <= n; i++ )
        ans += C( i - 1 ) * C( n - i );
    return ans;
}
```

This works fine, but there is a slight problem – it's ridiculously slow. It's not just exponential in time; it's worse. C(n) makes 2n recursive calls. It is even worse than factorial! This seems wrong. After all, $C_n$ only depends on the numbers $C_0$ through $C_{n-1}$ , and there are n of these numbers. There must be a lot of repeating recursive calls to C(i) that are responsible for the horrible running time.

Here is a simple way to fix that. We will keep a table of previously computed values of $C_i$ . Then if the function C() is called with parameter i, and we have already computed C(i) once before, we will simply look up its value in the table and avoid spending a lot of running time (again). We will need to initialize the cells of this table with some dummy values that would tell us whether we have computed the value of that cell already or not. This technique is called memoization, and here is how it would change the code.

```
int Cval[1024];
int C( int n ) {
    if( Cval[n] ) != -1 ) return Cval[n];
    int ans = 0;
    if( n == 0 ) ans = 1;
    for( int i = 1; i <= n; i++ )
        ans += C( i - 1 ) * C( n - i );
    return Cval[n] = ans;
}

int main() {
    for( int i = 0; i < 1024; i++ ) Cval[i] = -1;
    // call C(n) for any n up to 1023
}
```

The running time of C(n) is now quadratic in n because to compute C(n), we will need to compute all of the C(i) values between 0 and n-1, and each one will be computed exactly once, in linear time. Unfortunately, we will not be able to compute C(1000) this easily because it is huge and will cause an overflow, but that's a minor detail. :-)

The $C_n$ are called Catalan numbers, and there is actually a nice formula for them.

$$C_n = \frac{(2n)!}{n!(n+1)!}$$

They appear all over the place, including the number of ways of triangulating a convex polygon and the number of ways of putting brackets into an expression. You can find lots of cool facts here: http://mathworld.wolfram.com/CatalanNumber.html

## Lucky Numbers

This technique of remembering previously computed answers in a table, thus trading off memory for a gain in computation time, is called memoization. [I don't know anyone who has a good explanation of why there is no 'r' in "memoization".] Here is a fun little problem that can be solved using the same technique.

13 is an unlucky number. In fact, any number that contains 13 is also unlucky. For example, 130, 213 234513235 and 2451325 are all unlucky. All of the other numbers are lucky by default. How many lucky numbers are there with n digits?

We will try the same approach here. First, we will find a recursive solution, and then add a memoization table to it in order to avoid calling the recursive function twice with the same parameters.

In an n-digit number, the first digit can be anything between 1 and 9, but all subsequent digits can be anything between 0 and 9. This difference is a little inconvenient, so we will let the first digit be 0 as well. This way, instead of counting numbers with n digits, we will count all the numbers with up to n digits. Then if T[n] is the number of lucky numbers with up to n digits, we can count those with exactly n digits by subtracting T[n-1] from T[n], eliminating all those with fewer than n digits.

So how do we compute T[n] – the number of lucky numbers with up to n digits? Let's look at the first digit. It can be anything; we have 10 choices for it. However, when it is 1, then the second digit can not be 3 (or the number would start with "13"). Here is a recursive function for T[n].

$$T[n]=10\times T[n-1]-1\times 1\times T[n-2]$$

The first term ( $10\times T[n-1]$ ) says, "pick any of the 10 values for the first digit and make sure the rest of the number is lucky." This counts all the lucky numbers with up to n digits, but we get a few unlucky ones there, too. Namely, we also count the numbers that start with 13 and have no other 13s appearing anywhere. The second term subtracts all the "almost lucky" numbers we have erroneously counted. There are exactly T[n-2] of them because they all look like 13xxx, where xxx is an (n-2)-digit lucky number.

Since T[n] depends on both T[n-1] and T[n-2], we need two base cases here. T[0] is 1, and T[1] is 10. Here is a simple recursive implementation that uses a memoization table to store the values of T[n].

```
int T[1024];
int lucky( int n ) {
    if( T[n] == -1 )
        T[n] = 10*lucky( n - 1 ) - lucky( n - 2 );
    return T[n];
}

int main() {
    T[0] = 1;
    T[1] = 10;
    for( int i = 2; i < 1024; i++ ) T[i] = -1;
    // Call lucky(x) to compute the answer for x
}
```

Once again, we use -1 to mark the spaces in the memoization table that are still uncomputed. This time, we take care of the base cases in the main() function, which makes lucky() extremely short. Note that we have to actually use recursive calls to lucky(n-1) and lucky(n-2) in order for this to work. We can not simply use T[n-1] and T[n-2]. This time, the function works in linear time, computing T[i] for all the values between 2 and the target number, n. Each value takes constant time to compute and is computed exactly once. Without memoization, the lucky() function would require exponential time – $2^n$ recursive calls. The difference in running time is huge.

## Hamiltonian Path

A lot of hard problems can be solved using memoization. Sometimes, it is easy to come up with a $O(n!)$ brute force solution to a problem, and in many cases, adding memoization will improve the running time to $O(2^n)$ , which means an improvement from n=10 or 11 to n=25 or 30.

Recall the Hamiltonian path problem. In an unweighted graph, find a path from s to t that visits each vertex exactly once. The naive backtracking solution would start at s and try all of its neighbours recursively, making sure never to visit the same vertex twice. Branch-and-bound can improve the running time considerably, on average, but in the worst case, this approach is still n factorial in time. The algorithm would look something like this.

```
bool seen[32];
void hpath( int u ) {
    if( u == t ) { /* Check that we have seen all vertices. */ }
    else {
        for( int v = 0; v < n; v++ ) if( !seen[v] && graph[u][v] ) {
            seen[v] = true;
            hpath( v );
            seen[v] = false;
        }
    }
}
```

Note, however, that if we have a partial path from s to u using vertices $s=v_1, v_2, \ldots, v_k=u$ , then we don't care about the order in which we visited these vertices in order to figure out which vertex to visit next. All that we need to know is the set of vertices we have seen (the seen[] array) and which vertex we are at right now (u). There are $2^n$ possible sets of vertices and n choices for u. In other words, there are $2^n$ possible seen[] arrays and n different parameters to hpath(). What hpath() does during any particular recursive call is completely determined by the seen[] array and the parameter, u.

What we can do is change the seen[] array to be a single integer, using one bit per vertex. Then we can pass 'seen' as a second parameter to hpath(). Now if hpath() ever gets called with the same parameters twice, we will not do the same work again. Instead, we will store the answer in a memoization table and simply return it. Since there are only $n2^n$ different parameter pairs, and each recursive call requires $O(n)$ time (one 'for' loop), we will get a total running time of $O(n^2 2^n)$ – much better than $O(n!)$ .

```
bool memo[20][1 << 20];
void hpath( int u, int seen ) {
    if( memo[u][seen] ) return;
    memo[u][seen] = true;

    if( u == t ) { /* check that seen == (1<<n)-1 (seen every vertex) */ }
    else {
        for( int v = 0; v < n; v++ )
            if( !( seen & ( 1 << v ) ) && graph[u][v] )
                hpath( v, seen | ( 1 << v ) );
    }
}
```

Initially, we would call hpath() with u=s and seen=(1<<s) indicating that we are at s and have seen s already. The first two lines make sure that hpath() is never called twice with the same values of u and seen – there is no reason to do the same work twice. memo[][] is assumed to be initialised to false to start with.

Note again that we are trading memory (20 MB in this case) for an improvement in running time. This is the main advantage of memoization. Dynamic programming (DP) is a technique for filling out the memoization table iteratively, instead of using recursion, and we will look at it next. Many people use the terms DP and memoization interchangeably.