Advanced Dynamic Programming Technique

1 Bitmasks in DP

Consider the following example: suppose there are several balls of various values. Each ball may be one of three different colours: red, green, and blue. You want to package the balls together such that each package contains exactly three balls, one of each colour. The value of a package is defined to be the sum of the values of the three balls and you want to find the maximum value of the package possible. The state can be defined by the tuple (which ball we are currently considering, what colours are already present in the package). The recurrence relation essentially decide whether we will add the current ball we are considering. How exactly do we encode which colours are already present in the package? Bitmask of course! Code as follows:

```
int memo[128][8]; // initialized to -1

int max_value(int cur, int bitmask) {
  // base case
  if (bitmask == 7) return 0;
  if (cur >= num_balls) return -INF;
  if (memo[cur][bitmask] != -1) return memo[cur][bitmask];


  // don't add the current ball to the package
  int ans = max_value(cur + 1, bitmask)
  if (((1 << colour[cur]) & bitmask) == 0) {
    // try adding the current ball to the package
    int temp = value[cur] +
      max_value(cur + 1, bitmask | (1 << colour[cur]));
    ans = max(ans, temp);
  }
  return memo[cur][bitmask] = ans;
}
```

2 Iterative vs Recursive DP

In general, there are two ways to implement a DP algorithm. So far, all of the examples presented is recursive (hence recursive DP). This is because once we have found the state and recurrence relation, coding up a recursive implementation is a straightforward process. While this is easy, there are a few drawbacks of recursive DP. First, recursion involves pushing and popping stack frames for each function call and so is slower. Secondly, one must be careful about the depth of recursive calls. The DP algorithm may be sound, but if it involves making 100,000 recursive calls, you can be sure that the program will crash due to stack overflow.

Iterative DP, as the name implies, involves no recursion. Instead, we simply fill in the correct answer for each state ”in the right order”. What we mean by the ”right order” is that when you are trying to find the answer for a state, all the other states that the current state depends on are already filled in. This is best illustrated by an example. Consider the coin changing problem, coded iteratively:

```
int memo[128];

// find the minimum number of coins needed to make N
int min_coin(int N) {
  // base case
  memo[0] = 0;

  // fill in memo array
  for (int i = 1; i <= N; ++i) {
    memo[i] = INF;
    for (int j = 0; j < num_denominations; ++j) {
      if (denomination[j] <= i)
        memo[i] = min(memo[i], memo[i - denomination[j]);
    }
  }
  return memo[N];
}
```

In the above example, we noted that the recurrence relation for n depended on state whose value is less than n. So if we fill in the answer in increasing order, we will be guaranteed that the states we depend on has already been filled in.

3 Circular Recurrence Relation

When we first introduce DP, we noted that circular recurrence relation is a BIG no no. However, this does not mean that you should immediately discard any circular recurrence relation. In some (very special) cases, it is possible to rearrange the recurrence relation so that it's not circular. For example, suppose we are given the following recurrence $f(n) = \sum_{i=1}^{n} i * f(i)$. On the first look, the recurrence relation is circular since it depends on itself. However, we can rearrange the equation:

$$
\begin{aligned}
f(n) &= \sum_{i=1}^{n} i * f(i) \\
f(n) - n * f(n) &= \sum_{i=1}^{n-1} i * f(i) \\
f(n) &= \frac{1}{1-n} \sum_{i=1}^{n-1} i * f(i)
\end{aligned}
$$

Note that the recurrence relation is no longer circular. For the given example, it is simple to see that we can rearrange the recurrence. However, other recurrence may not be so obvious. So it's a good idea to keep this in mind when you are designing a DP algorithm.

4 Reducing the State Space

Sometimes, you have this really nice DP idea that will run in time. However, the state space is too large so the algorithm is infeasible. Again, in very special cases, all might not be lost! Consider the following variant of the coin changing problem: suppose you only have a K types of coins and you want to make change for the amount N. For coin type i, the denomination is `denom[i]` and the number of coins you have is `coin[i]`. A DP algorithm defines the state using the tuple (n, i), specifying the least number of coins required to make amount n using only the first i coins (or INF if impossible). The recurrence relation is easy to see and here's the pseudocode for the recursive implementation:

```
int memo[128][128]; // initialized to -1

int min_coin(int n, int i) {
  if (n == 0) return 0;
  if (i >= K) return INF;
  if (memo[n][i] != -1) return memo[n][i];

  int ans = INF;
  for (int j = 0; j < coin[i] && j * denom[i] <= n; ++j) {
    ans = min(ans, j + min_coin(n - j*denom[i], i+1)
  }
  return memo[n][i] = ans;
}
```

First, let's note that we can code this iteratively:

```
int min_coin(int N) {
  memset(memo, 0x3f3f3f3f, sizeof(memo));

  for (int i = 0; i < K; ++i) {
    memo[0][i] = 0;
    for (int j = 0; j < coin[i]; ++j) {
      int amt = j * denom[i];
      for (int n = amt; n <= N; ++n) {
        memo[n][i] = min(memo[n][i], j + memo[n-amt][i-1]);
      }
    }
  }
  return memo[N][K-1];
}
```

Note that in the above code, `memo[n][i]` only depends on the values `memo[*][i-1]`. So is there a point keeping track of `memo[*][i-2]`? Of course not! In fact, we can reduce the second dimension of the `memo` array! Instead, we will keep overwriting the same array (in a "smart way") so that we don't overwrite informations we may need later. In this case, the trick is to iterate n backward from N to `coin[i]`. This reduces the state space from $O(N * K)$ to $O(N)$.

```
int min_coin(int N) {
  memset(memo, 0x3f3f3f3f, sizeof(memo));
  memo[0] = 0;

  for (int i = 0; i < K; ++i) {
    for (int j = 0; j < coin[i]; ++j) {
      int amt = j * denom[i];
      for (int n = N; n >= amt; --n)
          memo[n] = min(memo[n], j+memo[n-amt]);
    }
  }
  return memo[N];
}
```