Dynamic Programming: Partition

We have seen the partition problem before. The input is a collection, C, of integers, and we are interested in a subset whose sum is exactly half of the total sum of C. The problem is NP-hard, and we know that split-and-merge can solve instances with about 40 numbers.

But what if we have n = 500 numbers? Obviously, split-and-merge does not work anymore. We need some extra information if want to have any hope of solving this in reasonable time. Suppose that that we do have some information – we know that the sum of all the numbers is at most N = 10000. This little detail makes the problem solvable in O(nN) time.

In the spirit of dynamic programming, we will create a boolean array T of size N+1. After the algorithm has finished, T[x] will be true if and only if there is a subset of the numbers that has sum x. Once we have that, we can simply return T[N/2]. If it is true, then there is a subset that adds up to half the total sum.

To build this array, we set every entry to false to start with. Then we set T[0] to true – we can always build 0 by taking an empty set. If we have no numbers in C, then we are done! Otherwise, we pick the first number, C[0]. We can either throw it away or take it into our subset. This means that the new T[] should have T[0] and T[C[0]] set to true. We continue by taking the next element of C.

In general, suppose that we have already taken care of the first i elements of C. Now we take C[i] and look at our table T[]. It has a 1 in every location that corresponds to a sum that we can make from the numbers we have already processed. Now we add the new number, C[i]. What should the table look like? First of all, we can simply ignore C[i]. This means that no ones should disappear from T[] – we can still make all those sums. Now consider some location of T[j] that has a 1 in it. It corresponds to some subset of the previous numbers that adds up to j. If we add C[i] to that subset, we will get a new subset with total sum j+C[i]. So we should set T[j+C[i]] to true as well. That's all.

Partition: simple version

```
bool T[10240];
bool partition( vector< int > C ) {
    // compute the total sum
    int n = C.size();
    int N = 0;
    for( int i = 0; i < n; i++ ) N += C[i];

    // initialize the table
    T[0] = true;
    for( int i = 1; i <= N; i++ ) T[i] = false;

    // process the numbers one by one
    for( int i = 0; i < n; i++ )
        for( int j = N - C[i]; j >= 0; j-- )
            if( T[j] ) T[j + C[i]] = true;

    return T[N / 2];
}
```

Note one important detail in the j loop. It goes through the table from right to left. We have to do this in order to avoid double-counting C[i]. For each true entry in the table, we set another entry to the

right of it to true. We wouldn't want to stumble across that new entry and process it again in the same loop – that would correspond to adding C[i] to the subset twice.

We can optimize this code quite a bit. First of all, it doesn't really make sense to scan the whole table every time. All we care about is finding all the true entries. At the beginning, only the 0th entry is true. If we keep the location of the rightmost true entry in a variable, we can always start at that spot and go left instead of starting at the right end of the table. To take full advantage of this, we can also sort C[] first. That way, the rightmost true entry will move to the right as slowly as possible. Finally, we don't really care about what happens in the right half of the table (after T[N/2]) because if T[x] is true, then T[N-x] must also be true eventually – it corresponds to the complement of the subset that gave us x. Here is the optimized version.

Partition: optimized version

```
bool T[10240];
bool partition( vector< int > C ) {
    // compute the total sum and sort C
    int n = C.size();
    int N = 0;
    for( int i = 0; i < n; i++ ) N += C[i];
    sort( C.begin(), C.end() );

    // initialize the table
    T[0] = true;
    for( int i = 1; i <= N; i++ ) T[i] = false;
    int R = 0;    // rightmost true entry

    // process the numbers one by one
    for( int i = 0; i < n; i++ ) {
        for( int j = R; j >= 0; j-- )
            if( T[j] ) T[j + C[i]] = true;
        R = min( N / 2, R + C[i] );
    }

    return T[N / 2];
}
```

After the optimizations, the running time is still O(nN), but we have avoided doing a lot of useless work.

Dealing with Unlimited Copies

What if we make the j loop run left to right? What will happen in this case? Suppose we are looking at C[i] and T[j] is true. Then we will set $T[j+C[i]]$ to true as well. And then we will set $T[j+2*C[i]]$, $T[j+3*C[i]]$, etc. In essence, we are allowed to use as many copies of C[i] as we want.

Suppose that you have an unlimited supply of various coins (pennies, nickels, dimes, quarters, loonies and toonies). They have the corresponding values of 1, 5, 10, 25, 100 and 200 cents each. How many ways can you make change for $5?

Here we are interested in the number of ways of making a set that sums to 500 cents. The idea is exactly the same. Only this time, our DP table is not boolean. Instead, it will store in T[x] the number of ways we can make change for x cents. We will start off by setting everything to 0, except of T[0], which is 1. Then we will proceed like before, only this time for each C[i], we will say the following. We can either ignore C[i], or we can add C[i] to x. If before we had T[x] ways of making change for x, then now will will have T[x] new ways of making change for x + C[i].

Coin changer

```
int T[10240];
int coins( vector< int > C, int N ) {
    // initialize the table
    T[0] = 1;
    for( int i = 1; i <= N; i++ ) T[i] = 0;

    // process the numbers one by one
    for( int i = 0; i < n; i++ )
        for( int j = 0; j + C[i] <= N; j++ )
            T[j + C[i]] += T[j];

    return T[N];
}
```

The function returns the number of ways of making change for N cents, given the values of the coins that we have. Note that our optimizations do not help anymore.

Partition with Duplicate Elements

What if our collection of integers looks like this: $(C_1, C_1, C_1, C_2, C_3, C_3, C_3, C_3, C_4, C_4, ...)$? If we have a lot of duplicate integers, we can avoid scanning the whole table once for each copy. Suppose that we have D[1] coins with value C[1], D[2] coins with value C[2], etc. There are n different coins. Then we can still have our algorithm run in O(nN) time if we use a little trick.

Once again, we have to scan the table right-to-left for each C[i], but this time, whenever we see a 1, we will go to the right and set D[i] locations to 1. In this coin problem, we are looking for a subset of the coins that make up half the total value. The following function will tell you whether such a partition is possible.

Partition with duplicates

```
bool T[10240];
bool partition( vector< int > C, vector< int > D ) {
    // compute the total sum (value)
    int n = C.size();
    int N = 0;
    for( int i = 0; i < n; i++ ) N += C[i] * D[i];

    // initialize the table
    T[0] = true;
    for( int i = 1; i <= N; i++ ) T[i] = false;
    int R = 0;  // rightmost true entry
```

```
        // process the numbers one by one
        for( int i = 0; i < n; i++ ) {
            for( int j = R; j >= 0; j-- ) if( T[j] )
                for( int k = 1; k <= D[i] && j + k * C[i] <= N / 2; k++ )
                    T[j + k * C[i]] = true;
            R = min( N / 2, R + C[i] * D[i] );
        }

        return T[N / 2];
    }
```

The 3 nested `for` loops need some attention. The i loop simply goes through each of the different coins. The j loop scans the table from right to left, looking for ones. When it finds a 1, the k loop will try to add coin C[i] to our growing subset. We can add at most D[i] copies. k in the innermost loop is the number of copies of C[i] that we are adding.

So far, this runs in O(N*z) time, where z is the total number of coins. We would like to reduce it to N times the number of different coins. To do that, we add the following line inside the k loop:

```
if( T[j + k * C[i]] ) break;
```

When we hit a 1 at T[j + k * C[i]] (the place where we're trying to place a 1 of our own), we know that we've already been there. There is no need to continue further to the right. This line will ensure that each cell in the table is only set to true at most once. And each cell will be looked at at most 2n times (once when going left in the j loop and once when going right in the k loop). The total running time is then O(nN).

There are lots of different variations of this problem. Sometimes, instead of a sum, we are interested in the product, and our coins are prime numbers. Then this algorithm is somewhat similar to the sieve of Erastosthenes. Other times, we might want to find a subset of a particular size k, whose sum is m. In this case, we might want to make the table two-dimensional, k-by-n, and use each row to generate the next one.