Group 17 Team Members: Gericho Ball, Lily Tang, and My Nguyen
COT 4400, Spring 2018; Due: April 2, 2018

Project 2: Dynamic programming

Question I: How you can break down a large problem instance into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the subproblems and why this breakdown makes sense.
Our overall problem is how many different samplers can Bitman's Confectionary Company (BCC) put together where no two samplers are alike. The samplers contain different types of candies i.e. square and long candies with sizes 1 and 2, respectively. We need to consider that samplers can have the same content but the candies need to be in a different order for it to be different.

Seen below in the Figure 1, we are going to analyze case, where the sampler size is 3, the number of square candies is 2, and the number of long candies is 2. The number of square and long candies are different types of candies such that there are 2 different types of square candies and 2 different types of long candies. In our case, the long candies are colored blue and yellow and the square candies are colored red and purple.

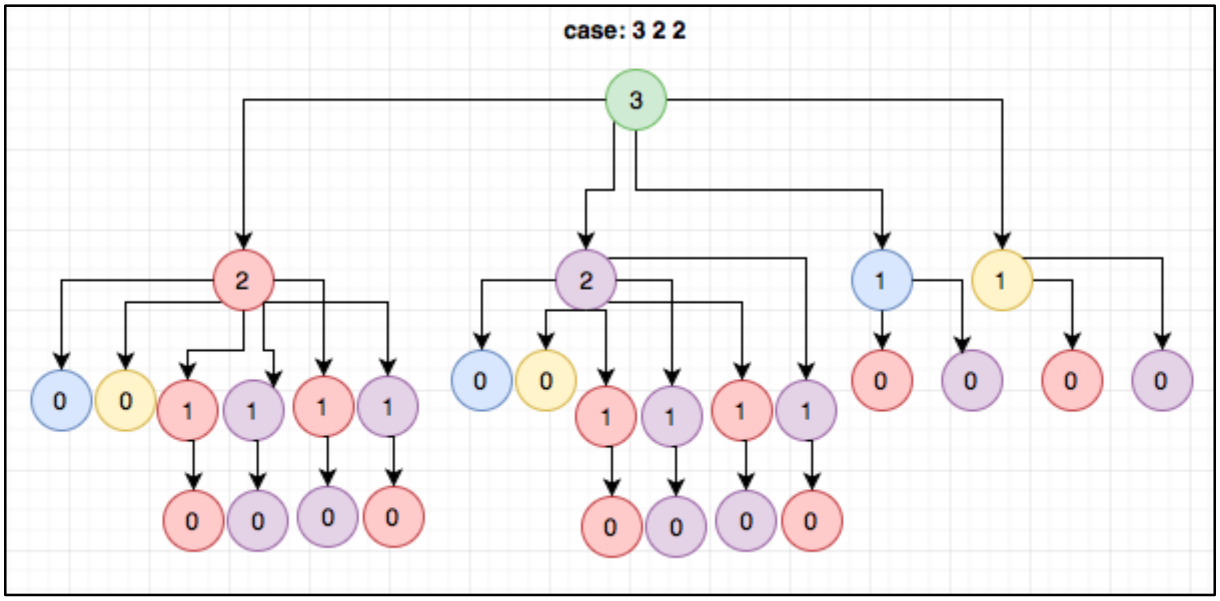

Figure 1: Case: 3 2 2

This large problem instance can be broken down into one or smaller instances because this problem is like the Fibonacci series, where the base cases of Fibonacci are Fibonacci of 0 returns 0 and Fibonacci of 1 returns 1. The breakdown for this candy problem has three cases:

1. s s s
2. s l
3. l s

Case 1: There are two flavors of s for 3 slots, thus, we can make 2^3 = 8 combinations.

When a purple or red square candy is called, a size 2 remains, leaving two options: one for long candy to be called and one for square candy to be called. When a square candy is called, the size will be 1, leaving for another recursive call on another square candy. When a long candy is called, the size will be 0, which is one of our base cases. For the remaining size 1, only a square candy call will occur, with a remaining size 0.

Case 2: With 2 flavors of s with 1 s slot and 2 flavors of l with 1 l slot, we can make 2^1 * 2^1 = 4 combinations. Case 2 and 3 are very similar to each other with the only difference in order.

Case 3: Looking at the right-hand side of 3, there is a blue long candy and a yellow long candy. When a long candy is called, a size of 1 remaining. From that size, only a square candy, red or purple, can be called. *Note*: Since there are two different types of square and long candies, thus, the algorithm should repeat for the two different types of candies.

Therefore, iterative recursive dynamic programming was used to call a long candy or a square a candy, solving the original problem with subproblems. From our analysis in Figure 1, we found our base cases to be: if **n = 0**, return *1*; if **n = 1**, return *s*; and if **n = 2**, return *s*s + l*.

Question II: What recurrence can you use to model this problem using dynamic programming? What are the base cases of this recurrence?
Based on the solution given in Question I, the total number of samplers are based on the time complexity of the square and long candies. The recurrence can be modelled as follows:

$$sampler(n,s,l) = \begin{cases} 1 & if\ n = 0 \\ s & if\ n = 1 \\ s^2 + l & if\ n = 2 \\ s * sampler(n-1,s,l) + l * sampler(n-2,s,l)\ if\ otherwise & \end{cases}$$

The base cases of this recurrence would return *1* when **n = 0**, return *s* when **n = 1**, and return *s^2 + l* if **n = 2**. When **n = 0**, the empty set is returned, which is needed for the recurrence relation to work. When **n = 1**, then a square candy can be returned. When **n = 2**, either a long candy or 2 square candies will be returned. Otherwise, we can calculate the number of samplers based on the recurrence model given above.

We were able to get the base cases from looking at the diagram above in Figure 1. By following the trace on (3,2,2) and observing the values in the circles after a new candy is added to any given arrangement, the base case is shown to be when **n = 1** and **n = 2**. We added a base case of **n = 0** returning 1 just in case. At this point, there are is no any room for mixed combinations of candies, new arrangements can only be made up of 1 square, or 2 squares / 1 long respectively. If you track up from the base cases shown above, you can see how the algorithm adds either a square or long candy, and adjusts *n* accordingly.

If **n = 1**, only a square candy can fit, and subsequently the number of new combinations is the number of different types of squares. After these new square pieces are added, then **n = 0** and no new combinations are possible.

If **n = 2**, we know that only 2 combinations for the sizes are possible, a single long or two squares. For handling the cases where the combination is a single long, the number of new combinations is the number of different longs available and **n = 0** for that possible combination. If there are squares available, the algorithm can handle it the same way as it did with **n = 1**, finding the possible combinations of one square, and repeating for the other until **n = 0** and there are no longer any new combinations available.

Question III: What data structure would you use to store the partial solutions to this problem? Justify your answer.

To store the partial solutions to this problem, there are a couple of key issues that need to be considered. First, looking at the test output, we know that the total number of combinations becomes very large. Second, we know that we will need to access every partial solution that we find because the total count will need to include every possibility. So, it would be smartest to use a data structure that can access the elements efficiently. Finally, we know that we do not need to search, or delete any past elements assuming we consider them accounted for. For these reasons, something simple, and efficient would work well. Assuming we have enough memory available to use we can use an associative map to store the partial solutions. We can use a three-dimensional map to account for *n*, *l*, and *s* and store them as needed. Accesses to any element would be cheap, and there would be no need to reconstruct it after doing so initially.

Question IV: Describe a pseudocode algorithm that uses memoization to compute the number of samplers.

Input: *n*: number of "space" units available for the box of chocolates
Input: *l*: number of different types of long chocolate available for the box
Input: *s*: number of different types of square chocolates available for the box
Output: total number of different chocolate combination for the given box

```
1 Algorithm: ChocCount
2 unsigned long long count[n+1];
3 Initialize array to NULL;
4 Return Arrange(n, s, l);
```

```
1 Algorithm: Arrange(c, sq, ln)
2 if count[n] != NULL then
3         return count[n];
4 else if n = 0
5         return count[n] = 1;
6 else if n = 1
7         return count[n] = sq;
```

```
8 else if $n$ = 2
9       return count[$n$] = $sq$*$sq$ + $ln$;
10 else
11      count = $sq$*count[i - 1] + $ln$*count[i - 2];
12 return count;
```

We used map initialized to *unsigned long long* to make sure that all combinations can be accounted for such as outputs with more than 12 digits. In addition, we had to make sure to account for overflow, thus *unsigned long long* was chosen. We had to ensure that the solution will be able to cover 2^32 and less than 2^64 digits.

Question V: What is the time complexity of your memoized algorithm?
Seen in Figure 2, we have case (4 1 2) where the sampler size is 4, the number of square candies is 1, and the number of long candies is 2. The number of square and long candies are different types of candies such that there is 1 different type of square candies and 2 different types of long candies. In our case, the long candies are colored blue and yellow and the square candy is colored red.

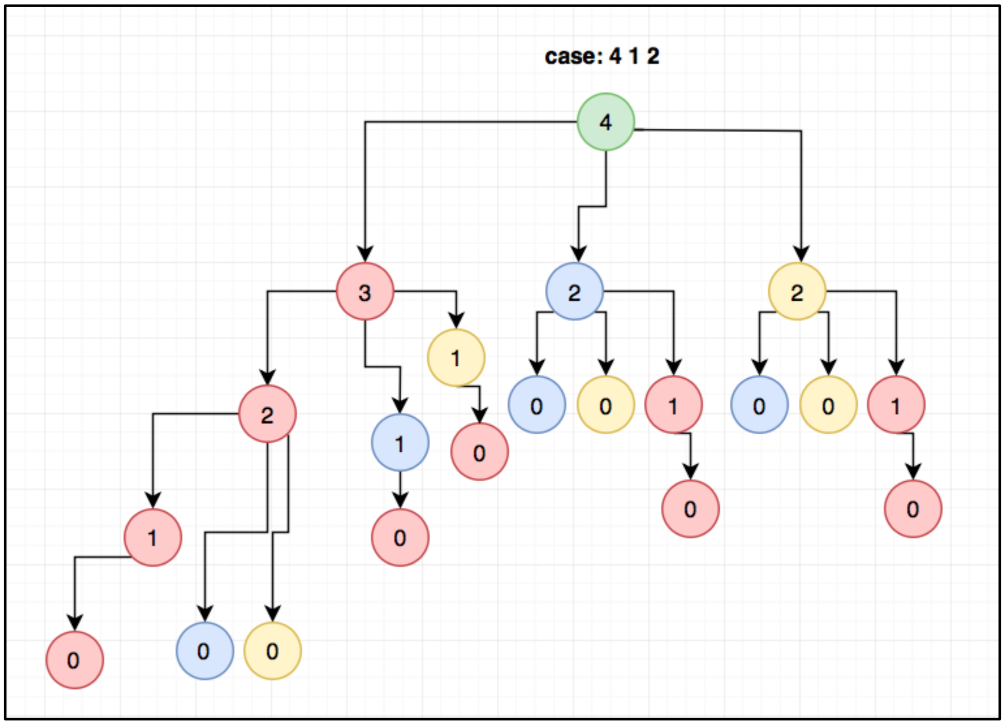

Figure 2: Case: 4 1 2

We used our memoized algorithm to reduce the time complexity of calculating the number of different samplers created by returning a stored value that was already computed. Based on memoization and time complexity as T = s*T(n-1) + l*T(n-2), we compute ChocCount(4) = 1*ChocCount(3) + 2*ChocCount(2) = 1*5 + 2*3 = 11 to get the number of sampler. With our memoized algorithm, the time complexity is O(n) because we are returning the stored value.

Group 17 Team Members: Gericho Ball, Lily Tang, and My Nguyen
COT 4400, Spring 2018; Due: April 2, 2018

Question VI: Describe an iterative algorithm for this problem.
Input: *n*: number of “space” units available for the box of chocolates
Input: *l*: number of different types of long chocolate available for the box
Input: *s*: number of different types of square chocolates available for the box
Output: total number of different chocolate combination for the given box

```
Algorithm: Sampler
unsigned long long count[n+1];
count[0] = 1;
count[1] = s;
count[2] = s*s + l;
for i = 3 to n do
    count[i] = s*count[i-1] + l*count[i-2];
end
return count[n];
```

Question VII: Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm? Justify your answer.
Space complexity is the memory usage. The space complexity of the iterative algorithm is the same as the memoized algorithm of O(n). Our space complexity would be O(n) because we already have work done through stored values of already computed numbers. However, we can improve on this by using an optimized iterative algorithm by saving only the last two values instead of using an array to store all previous answers. In such a case the space complexity would be O(1) because each cell only needs the previous two values, and old values that are not useful are thrown away.