a) The dynamic programming subproblems in my situation was that you had to account for the total sum and the optimal solution for each possible subset. With this information, I was able to create two 2D arrays that would store the solutions so I can use it in the recursive formula of OPT(i, j) = min(SUM[i,j] + OPT(I,j-1) , SUM[I, j] + OPT(i+1, j)). This formula considers the optimal solution of each subset from both sides depending on the wall placement. Thus, when you figure out where your boundaries are in terms of your wall, you add its relevant previous weight. In this case, the weight is represented by the total fruit collected from the previous day(s). Each array accommodates the max fruit the apes would get, which will be the min of the two OPT solutions, one from the left side of the wall and one from the right.

b) For my algorithm, I accommodate the fact that dynamic programming involves the solution of subproblems first to solve something bigger. Thus, I realized you must save all possible sums of each subset of the int array to account for every optimal solution scenario. This would be used as the sum of fruit gained from the previous iterations of the current day you were trying to solve. Thus, using the size of the array, I used half of a 2D array to store all possible sums. Of course, this meant that you would still have to find the max number of fruits you can get given specific scenarios. Thus, I would have to solve the fruit gained of a single section first, which was 0 since the humans would get the section regardless. If two sections are left, then the recursive formula would return the min of the two. Lastly, it would always check if the optimal solution was already solved and if it was stored in my optFinal half 2D array. Since my formula would add the sum of the previous days to the optimal solution, it would take into consideration the base cases. Thus, I solved the optimal solutions by getting base cases first. I checked for OPT(I, j) where j = I, and they returned 0. In OPT(I, j) where j= i+1 meaning there is only two sections, it would choose the min of the two. Afterwards, depending on the subsets, it would use the recursive function until it reaches the case where OPT was already solved previously from the base cases. Due to my bottom up approach, I believe I used the information I solved previously efficiently and considered it whenever possible, which resulted in my n^2log(n) time complexity. It took two n^2 to fill up my 2D arrays with every possible subset scenario and use it to calculate my OPT. The log(n) came from selecting the most optimal position of the wall. Simplifying it, I took the half of the array and found out if the left subset or the right subset had the bigger sum. If right, I I would then take the halfway point of that subset, set that point as the right most boundary for the previous left subset, then recalculate the side with the max sum with the updated boundaries. If left, vice versa, but I would use the halfway point as the left most boundary for the previous right subset. Eventually, I get a predicted wall where then I calculate the OPT for the left and right side and take the minimum. However, this is a projected max since we basically only took the OPT of one single wall position. Based on patterns discovered in the preplanning phase, I realized that I could get the true max if I check the OPT when I move the wall left and right. The max can only be updated, meaning that if I move the wall once to the right and its OPT is bigger than my predicted wall, then that would be my new OPT. However, the moment the OPT is not updated, that is our signal to stop since every solution from there on will not be the OPT. This also applies to the left side. Thus, I calculated several times from a predicted midpoint to get my OPT, however, I do not calculate all n points since I have special base cases to prevent that. Thus, by using binary search to reduce the number of predicted wall placements, I end up with a runtime of log(n). In total, my runtime complexity is n^2*log(n)