Problem # 1

Problem Description: Determine the number of ways to achieve S secure doors out of n total doors.
Inputs:

- n = number of doors.
- S = desired number of secure doors.

Outputs: Total number of unique ways the doors can be locked to achieve exactly S secure doors.

Assumptions:

- For a door to be secure it must be locked and it must either be the first door or the door its left must also be locked.
- S and n are integers.

Strategy Overview:
I will use a dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems. Partial solutions will be stored in two tables.
Algorithm Description:
Initialize two 2D arrays, R and Z with dimensions [S][n]. $R[i][j]$ will store the number of ways to achieve i secure doors from n total doors in such a way that the last door is locked. $Z[i][j]$ is the number of ways to achieve i secure doors from n total doors in such a way that the last door is unlocked. Set every value in each array to be 0. Then set $R[i][i] = 1$ and $Z[i][i+1] = 1$. Then we iterate through the two tables row by row filling out solutions and then returning the final answer.

```
for i in range(S):
    for j in range(i+1, n):
        R[s][n] = Z[s][n-2] + R[s-1][n+1]

        if i+1 != j:
            Z[s][n] = Z[s][n-1] + R[s][n-1]

return Z[S][n] + R[Z][n]
```

Problem # 2

Problem Description: Computer the number of ways that K or fewer integers can be selected from a set such that their sum is exactly S.
Inputs:

- A = List containing n integers.
- S = Target sum.
- K = Maximum number of elements that may be selected from A.

Outputs: Total number of unique ways that K or fewer elements from A can be selected such that their sum is exactly S.

Assumptions:

- Items from A cannot be used twice in a given sum.
- Elements of A may be negative.
- S may be negative.

Strategy Overview:
I will use a bottom up dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems.

Algorithm Description: Create a three dimensional array F to store the solutions to subproblems. One dimension should be n, the one should be K, and the third should range from the minimum possible sum in A to the maximum. To find the minimum sum all of the negative elements in A, to find the maximum sum all of the positive elements. For convenience I'll assume that the array is indexed from minimum to maximum even if the minimum is potentially less than 0. Then the value stored at $F[s, k, i]$ is the number of ways to pick k or fewer elements from the first i elements of A to make the sum s.

First fill in the base cases.

1. s =maximum sum
   $F[s, k, i] = 1$ when i is greater than or equal to the index of the last positive number in A and k is greater than or equal to the number of positive numbers in A. Ft every other position the value is 0.
2. s =minimum sum
   $F[s, k, i] = 1$ when i is greater than or equal to the index of the last negative number in A and k is greater than or equal to the number of negative numbers in A. Ft every other position the value is 0.
3. $s = 0$
   If $i = 0$ or $k = 0$ then the value at $F[s, k, i] = 1$.
4. Other s
   If $i = 0$ or $k = 0$ then the value at $F[s, k, i] = 0$.

After the array has been initialized in this manner, we fill in the remaining values.

```
for i in (1, n):
    for k in (1, n):
        for s in (min sum, max sum):
            F[s,k,i] = F[s, k, i-1] + F[s - A[i], k-1, i-1]
```

When this has been completed then the solution to the problem is at $F[S, K, n]$.

Problem # 3

Problem Description: Given a set of boxes, determine how deeply that boxes can be nested within one another.
Inputs:

- B = Set of boxes.

Outputs: The set of boxes that can be nested into the longest chain.
Assumptions:

- Each box b_i contains fields for that box's width, height, and depth.
- Boxes can be rotated in any way.

Strategy Overview: I will use a dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems.
Algorithm Description:
First replace B with a list of all the possible rotations of all of the boxes by permuting the parameters of the given boxes. Sort this list by decreasing volume. At this point the problem has been reduced to finding the longest subsequence of B subject to the condition that each box must fit into the box that was previously selected. Now create an array S with the same length as B. Each position $S[i]$ will store the length of the longest chain of boxes that has $B[i]$ as its innermost box. Initialize S to contain all 1s. Then iterate through S filling in solutions using the recurrence that

$$S[j] = \max(\{S[i] | i < h, B[i].width < B[j].width, B[i].height < B[j].height, B[i].depth < B[j].depth\})$$

If the set generated above of which the maximum is found is empty then simply make no change to the value of $S[j]$. Then the deepest chain is the maximum value in S. Next we must reconstruct the actual set of boxes used in this chain. If the maximum value in S is located at $S[k]$ then we know that box $B[k]$ is the final box in the chain. Then scan backwards through S until you find a $S[l]$ such that $S[l] = S[k]$ and all of the parameters of $B[l]$ are larger than the corresponding parameters of $B[k]$. Then add $B[l]$ to the output and continue scanning using the same process until the beginning of the chain is reached.

The algorithm has a running time of $\Theta(n^2)$ where n is the total number of boxes and their orientations.

Problem # 4

Problem Description: Given a set of ski runs and the amount of time each takes, find the minimum number of days to ski all of the runs and the particular way to do so that minimizes the dissatisfaction over time wasted.
Inputs:

- R = List of runs.
- n = Number of runs.
- L = Minutes of skiing available in a day.
- m = Maximum number of minutes that can be wasted without significant dissatisfaction.
- C = Constant used to quantify dissatisfaction.

Outputs: A skiing schedule that minimizes the number of days needed to ski. If there are multiple ways to do that then it outputs the way that also minimizes the time wasted dissatisfaction function.
Assumptions:

- Each run must fit into a single day.
- Time wasted dissatisfaction is measured by $twd(t)$.

Strategy Overview: I will use a bottom up dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems.
Algorithm Description: The first step is to find the minimum number of days D that can be used to complete the runs. This is done with the same greedy algorithm developed in the roadtrip problem from an earlier homework. Next create a two dimenional array A. One dimension will have a length equal to the minimum number of days just calculated. The second dimension will have a length equal to the total number of runs. Then the value stored at $A[d, i]$ is the minimum value of the twd function if run i is completed on day d.

Initialization
We know that we will never need more days than the number of runs. Therefore we can initialize every $A[d, i]$ where $i > d$ to be infinity or some other placeholder value. $A[1, i] = twd(L - r_1 - r_2 - \cdots - r_i)$ as long as $L - r_1 - \cdots - r_i \geq 0$. If it's not greater than 0 then the value is irrelevant and can be set to infinity. If $d = i \neq 1$ then we know that we did one run per day and the value of $A[d, i]$ is equal to

$$A[d - 1, i - 1] + twd(L - r_i)$$

The final case is when $d < i$. There a few different things that could happen to lead to this situation. Either run i happened by itself on day d, or run i happened with run $i - 1$ on day d, or runs $i, i - 1, i - 2$ all happened on day d,

etc. This goes on until runs $i, i-1, \ldots, d$ happen on day d. It's impossible that run $d-1$ happens on day d in the most efficient solution. Thus

$$A[d,i] = \min_{\substack{j \leq i \\ j > d}} twd(L - r_i - r_{i-1} - \cdots - r_j) + A[d-1, j-1]]$$

subject to the condition that $L - r_i - r_{i-1} - \cdots - r_j \geq 0$. If the restrictions mean there are no values to take the minimum of then we can default to infinity because that square in the grid will be irrelevant to the overall solution. In addition, in order to make reconstructing the actual run simpler, we can store the successful j in a serperate table B.
After the grid has been filled in we can look in $A[D, n]$ to find the minimized value of the twd function. We know that r_n is the last run on day D. From there simply consider the value stored in the corresponding location in table B. Then all of the runs from j to n happened on the last day. Now look at location $A[d-1, j-1]$ and preform the same process. All of the runs from $r_{B[d-1,j-1]}$ to j were done on day $d-1$. Repeat until the entire run has been reconstructed.