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.

```
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 dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems.
Algorithm Description:

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 0s. Then iterate through S filling in solutions using the recurrence that

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

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 dynamic programming approach that constructs an overall solution by looking at solutions to smaller subproblems.
Algorithm Description:
