# 1 Doors

For this algorithm, the method will take in three parameters, n (a list of all doors), S (exact number of secure doors needed), and locked ( a Boolean value: True represents current door is locked and False represents unlocked). We will first start with a base case that checks if S is greater than the size of n. If it is, then return 0 because there are 0 unique ways to lock the doors to get S secure doors. The second base case will be if locked is true and the size of n =1. this will just return 1. The next if statement would be if the door we are currently on is true. You need to return two recursive calls where to cover all possibilities (so you would add together two recursive calls and then return). In the first recursive call you would do True, n-1 and S-1 since the current door is locked and you have to check for one less secured door in the next call. For the second recursive call we would have False, n-1, and S (note: not s-1 because this door is unlocked and we do not have one less secured door to take care of). Then, we would have an else statement (meaning locked is set to False) and also return two recursive calls. These recursive calls would be True, S, n-1, and then n-1, S, False. Therefore, these two if statements and four recursive calls will cover every single possibility of the doors and return the total number of unique ways to achieve exactly S secured doors.

# 2 Skiing

We need to initially handle the first part of the problem, where we have to find the optimized scheduling without worrying about the wasted time. So, we need to find the minimum amount of days we can schedule all of the activities. If there is only one schedule that can be done in the minimum amount of days then we simply return that optimized schedule. But, if there are multiple schedule that will fit in the minimum amount of days we found then we have to use dynamic programming to quickly find which schedule wastes the least amount of time. To do this, we need to initialize a list of lists with all zero (called dp), and a list of lists with all False (called solved). Each list in dp will have the amount of time wasted per day for a unique schedule. We then need to populate this list of list with actual time wasted values by looping through each unique optimized schedule and counting the amount of time wasted with twd(t), then setting this specific day to True. The list of list is very important for dynamic programming because with each call to a specific day in a specific schedule we can look at our solved list of list and if it is set to True we have already gone through and calculated the time wasted for a prior day that has the same amount of hours/minutes/seconds of activities. So, we can just then use that time already in our list of list instead of going through and calculating the exact amount of time wasted again. This is the dynamic programming aspect and makes our code a lot faster. Once we have gone through every specific schedule and each day of that schedule, we can then add up every value in each list of dp and compare it to one another. The list with the smallest number in dp is the optimal schedule we will pick.

## 3 2*w board

With a little bit of tweaking, the Fibonacci dynamic programming algorithm from class can be used to satisfy this problem. I know this because of recurrence relations. To fill up a board with 2 times n tiles with dominoes, you have to first tile up 2 * (n-2) tiles then 2 * (n-1) tiles, etc. This can be represented by the recurrence relation $T_n = T_{n-2} + T_{n-1}$ which is the same as the Fibonacci sequence. Therefore, solving this recurrence relation gives a runtime of $\Theta(n)$
So, the pseudocode for the algorithm below will solve this problem:
solution[] needs to be initialized to have all values be -1.

```
def dominoes(int w, long solution[]
    if solution[w] != -1:
        return solution[w]; //return stored value
    long value;
    if w <= 2:
        value = w;//odd but right
    else:
        value = dominoes(w-1, solution) + dominoes(n-2, solution);
    solution[w] = value; //store calculated value
    return value;
```

## 4 4*w board

In order to solve this problem, we cannot exactly copy the Fibonacci sequence like the last question because the board size is now 4 times n, but this will still be $\Theta(n)$ runtime because we are completing the same process of dynamic programming and building up your solution using previous part of your solution. So,to solve this in an efficient way we first need to look at the board vertically, base (which is always 4) times height. You can think of the base case of w being 0 as 0 because there are no possible/unique ways to arrange the dominoes if there is no height. For the base case of w being 1, there is only one possible way/unique arrangement of the dominoes and that is two dominoes lying flat/horizontally. The last base case is w equals 2, and there are 5 possible arrangements of the dominoes. To find the 5 cases I split up the base into two boards of a base of 2. I then drew all the possible combinations and I discovered that there are three arrangements of the dominoes if you put the dominoes horizontally to start, and there are two possible arrangements if you put the dominoes vertically to start. This is where the dynamic programming part of the solution comes in. We have to make a 2-D array storing the value of w and number of unique arrangements at that w, in each list. For every other value of w greater than 2 has to have at least 5 unique cases, so therefore we can use the value at w = 2 in the 2-D array to start "countingthe number of possible arrangements, saving us time and cost. Therefore, for every value of w greater than 2 you can use the previous solutions that we have calculated thus far,

and just build on that. So, you do not have to recount every single combination, because we have the previous amount of unique combinations stored in an accessible 2-d array.