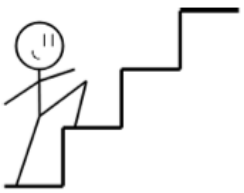
123. Climbing Stairs

You are currently climbing a staircase where n steps are needed to reach the top. When you climb the stairs, you can either choose to move up one step or move up two steps as shown below:

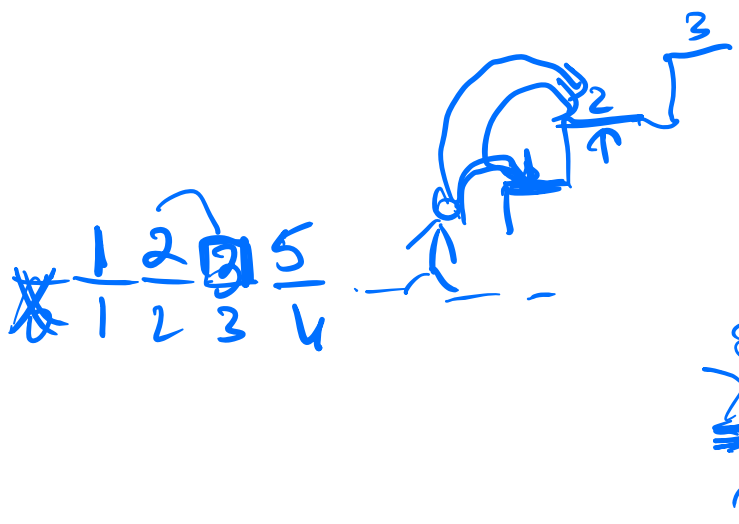


Implement the following function, which takes in the value of n and determines the number of distinct ways one can climb to the top of the staircase, given that they can either move up one step or two steps with each move.

Complexity: $O(n)$ time and $O(n)$ auxiliary memory.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
int climb_stairs(int n) {
  vector<int> memo(n);
  memo[0] = 1;
  memo[1] = 2;
  for (int i = 2; i < n; ++i) {
    memo[i] = memo[i - 1] + memo[i - 2];
  }
  return memo[n - 1];
}
```



127. Dice Throw

You are given N dice, each with M faces that are numbered from 1 to M. Implement a function that returns the number of ways to get a sum of X, where X is the summation of values on each face after all of the dice are thrown.

Example: Given $N = 2$ (two dice), $M = 6$ (each die has 6 sides), and $X = 12$ (the target sum), your function should return 1, since there is only one way to get a total of 12 (by rolling a 6 on each die).

Complexity: $O(MNX)$ time and $O(NX)$ auxiliary space.

Implementation: Implement your solution below. You may use anything from the STL. Line limit: 15 lines of code.

$F(N, M, X) = F(N-1, M, X-1) + F(N-1, M, X-2) + \dots + F(N-1, M, X-M)$

$x=12$: $F(2, 6, 12) = 1$

$F(1, 6, 11) = 0$
$+ F(1, 6, 10)$
$+ F(1, 6, 9)$
$+ F(1, 6, 8)$
$+ F(1, 6, 7) = 0$
$+ F(1, 6, 6) = 1$
$F(1, 6, 5)$

$F(1, 6, 11) =$
$F(0, 6, 10)$
$+ F(0, 6, 9)$

$F(1, 6, 6) =$
$F(0, 6, 5)$
$\vdots$
$+ F(0, 6, 1)$
$+ F(0, 6, 0)$

M X

```
if (N==0 && X==0) {
    return 1;
else if (N==0)
    return 0;
}
```

```
for (i=1; i<=M; ++i){
    F(N-1, M, X-i);
```

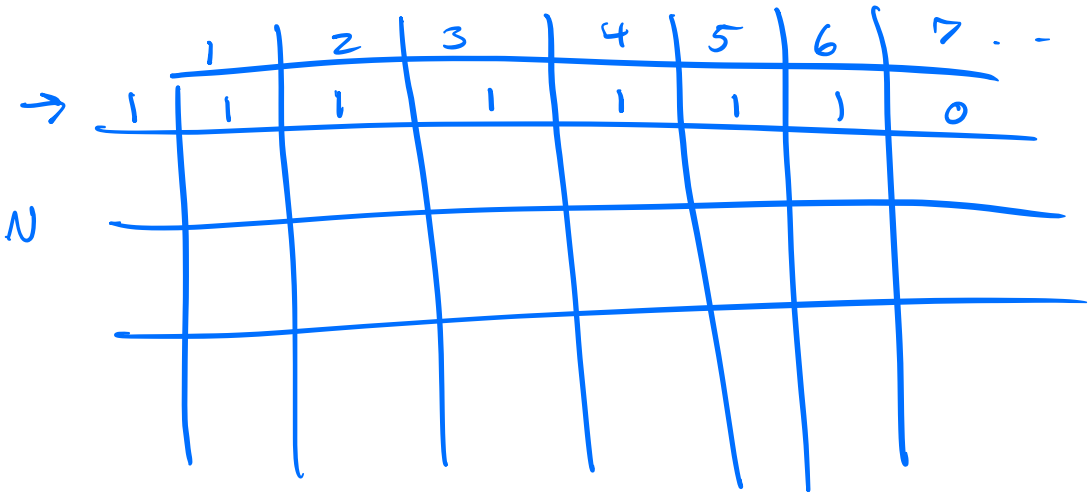
M dices

N dice

```
int find_ways(int M, int N, int X) {
  vector<vector<int>> memo(N + 1, vector<int>(X + 1));
  for (int j = 1; j <= M && j <= X; ++j) {
    memo[1][j] = 1;
  }
  for (int i = 2; i <= N; ++i) {
    for (int j = 1; j <= X; ++j) {
      for (int k = 1; k <= M && k < j; ++k) {
        memo[i][j] += memo[i - 1][j - k];
      }
    }
  }
  return memo[N][X];
}
```

X

M = 6



Winter 2020 Final Exam Question 25: Super DP Bros.

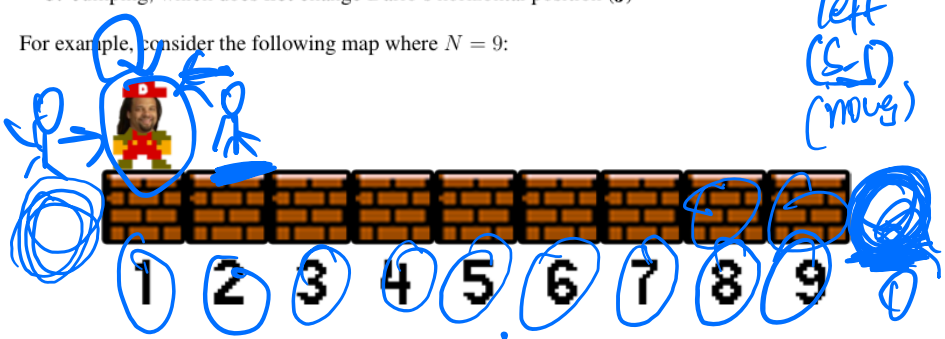
For the following problem, you will be able to submit your solution to Gradescope for automatic feedback. Please see Canvas to download the starter files for this problem.

The video game company that you work for, *Cin-tendo*, is developing an exciting new game for release next year: *Super DP Bros.*! This game details the lives of two Italian plumbers, Dario and Paoluigi (the DP Brothers) and their quest to save Princess Pear from a group of evil lobsters who are holding her hostage in a faraway castle codenamed "EECS280."

In the final level of this game, the evil lobsters require Dario to complete a challenge: given an integer M and a one-dimensional map of N tiles numbered 1 to N, Dario must discover the total number of ways he can make **exactly** M moves and end up back at his starting position S. The following three actions count as a single move:

1. Moving left by 1 tile (**L**)
2. Moving right by 1 tile (**R**)
3. Jumping, which does **not** change Dario's horizontal position (**J**)

For example, consider the following map where $N = 9$:

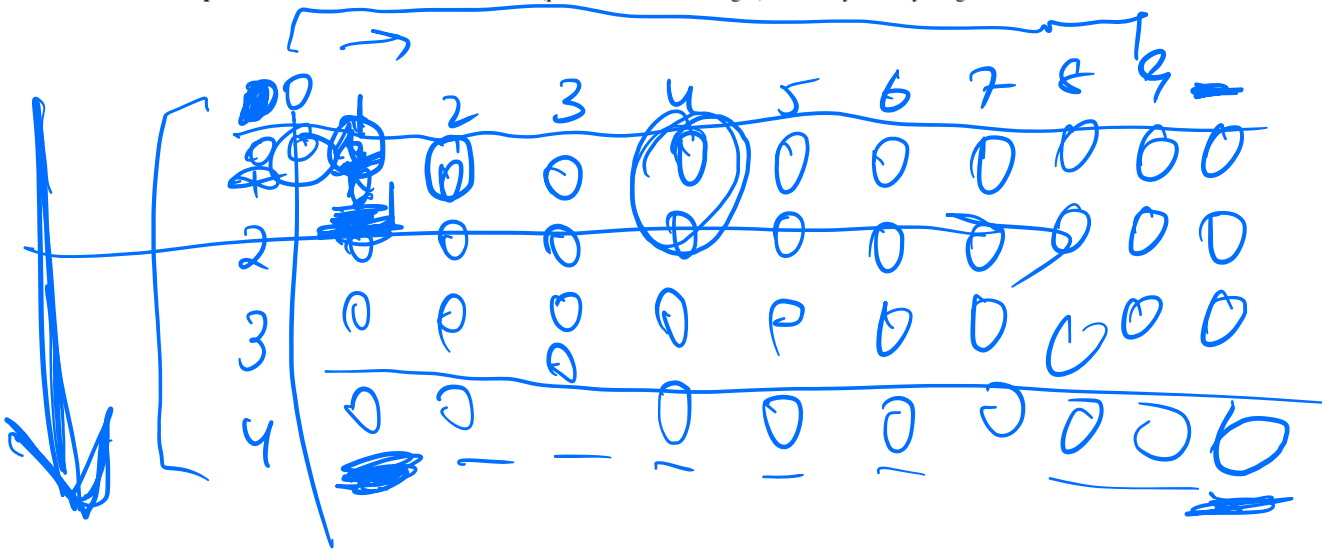


If the evil lobsters were to give Dario the values $M = 4$ and $S = 1$, Dario would need to return the value **9**, since there are 9 possible sequences of length 4 that Dario can make if he wants to start and end on tile 1: **JJJJ**, **JJRL**, **JRJL**, **JRLJ**, **RLRL**, **RRLL**, **RLJJ**, **RJJL**, and **RJLJ**. The tile positions on the map are 1-indexed, and Dario cannot jump off the edge or loop around.

Dario doesn't have much time, and he needs your help! Implement the `count_possible_moves()` function, which takes in three values, `num_moves`, `num_tiles`, and `start_pos`, and returns the total number of ways Dario can exhaust all `num_moves` moves and finish on tile `start_pos`. The fate of Princess Pear lies in your hands!

Complexity: $O(MN)$ time and auxiliary space, where $M =$ `num_moves` and $N =$ `num_tiles`. You may assume that `num_moves` > 0 and `num_tiles` > 1, and that `start_pos` will always be a valid position on the map.

Implementation: Limit: 30 lines of code (points deducted if longer). You may use anything from the STL.



Implement your solution to "Super DP Bros." below:

```
int count_possible_moves(int num_tiles, int num_moves, int start_pos) {
  vector<vector<int>> memo =
    vector<vector<int>>(num_moves + 1, vector<int>(num_tiles + 2, 0));
  memo[0][start_pos] = 1;
  for (int m = 1; m <= num_moves; ++m)
    for (int n = 1; n <= num_tiles; ++n)
      memo[m][n] = memo[m - 1][n - 1] + memo[m - 1][n] + memo[m - 1][n + 1];
  return memo[num_moves][start_pos];
}
```

Alternative solution to "Super DP Bros.":

```
int count_possible_moves(int num_tiles, int num_moves, int start_pos) {
  vector<vector<int>> memo(num_moves + 1, vector<int>(num_tiles + 2, -1));
  for (int i = 0; i <= num_tiles; ++i)
    memo[0][i] = (i == start_pos) ? 1 : 0;
  return helper(num_tiles, num_moves, start_pos, memo);
}

int helper(int num_tiles, int num_moves, int start_pos,
           vector<vector<int>> &memo) {
  if (start_pos == 0 || start_pos == num_tiles + 1)
    return 0;
  if (memo[num_moves - 1][start_pos - 1] == -1)
    memo[num_moves - 1][start_pos - 1] =
      helper(num_tiles, num_moves - 1, start_pos - 1, memo);
  if (memo[num_moves - 1][start_pos] == -1)
    memo[num_moves - 1][start_pos] =
      helper(num_tiles, num_moves - 1, start_pos, memo);
  if (memo[num_moves - 1][start_pos + 1] == -1)
    memo[num_moves - 1][start_pos + 1] =
      helper(num_tiles, num_moves - 1, start_pos + 1, memo);
  return memo[num_moves - 1][start_pos - 1] +
         memo[num_moves - 1][start_pos] +
         memo[num_moves - 1][start_pos + 1];
}
```

124. Jump Game

You are given a vector of non-negative integers, and you are initially positioned at the first index of the vector. Each element in the array represents your maximum jump length at that position.

Write a program that returns whether you are able to reach the last index.

Example 1: Given the following vector:

```
[2, 3, 1, 1, 4]
```

return **`true`**. You can reach the last index by jumping 1 step from index 0 to index 1, then jumping 3 steps to the last index.

Example 2: Given the following vector:

```
[3, 2, 1, 0, 4]
```

return **`false`**. You will always arrive at index 3 no matter what. Since your maximum jump length at index 3 is 0, it is impossible to reach the last index.

Complexity: $O(n)$ time and $O(n)$ auxiliary space, where n is the number of elements in the vector.

Implementation: Implement your solution below. You may use anything from the STL. Line limit: 15 lines of code.

```cpp
bool can_jump_out(vector<int> &nums) {
  if (!nums.empty() && !nums[0]) {
    return nums.size() == 1;
  }
  if (nums.size() > 1) {
    vector<int> dp(nums.size(), nums[0]);
    for (int i = 1; i < static_cast<int>(nums.size()) - 1; ++i) {
      dp[i] = max(nums[i], dp[i - 1] - 1);
      if (!dp[i]) {
        return false;
      }
    }
  }
  return true;
}
```