LEGO

HACKERRANK LEGO BLOCKS

By: Abderahim Salhi, Siddhanth Patel, Yakov Kazinets

The Problem

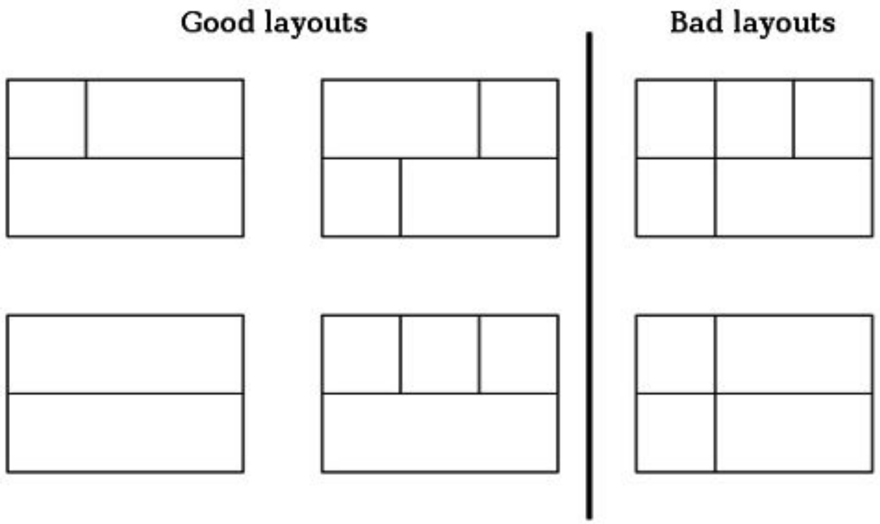
Given 4 sizes of blocks with each having a depth and height of 1 and width ranging from 1-4 create every possible wall that has a height N and width M . The bricks must be laid flat and there can be no holes in the wall. The wall must also be one solid structure such that there is no vertical lines that break across across all the rows.

The return result is number of valid wall formations modulo (10ˆ9)+7

https://www.hackerrank.com/challenges/lego-blocks/problem?h_r=internal-search

Example

Given sized wall of n=2 and m=3 here are good and bad layouts for context with the vertical line violation



These are not all of the valid permutations. There are 9 valid permutations in all.

Test Cases and Solutions

Our Test Cases were created with stress testing in mind to ensure that with running large and medium sized numbers the algorithm did not see any significant slow down.

100
271 700
418 840
570 364
623 795
174 848
432 463
683 391
293 792
58 116
522 158
575 492
948 952
232 22
538 741
55 31
99 326
82 517
517 3
{truncated for presentation format}

820286798
336082008
487244999
940472633
891909552
41392131
539075773
534496193
638706983
75175175
316908099
115618020
297994145
82228217
335734596
939848822
238272159
463545918
{truncated for presentation format}

Initial Approach and Shortcomings

Description of our initial approach:

Try to find a mathematical formula to find the total number of blocks then subtract the number of wrong builds.

Failed attempts or realizations:

We tried to do simple exponentiation of different sorts like Width ˆ Height – 1. Or (Width – x) ˆ Height

We also tried the combinations formula but that also failed

Combination Formula

$$ {}^{n}C_{r} = \frac{n!}{(n-r)!r!} $$

Final approach:

Two Steps:

Step 1: Find total number of layouts using tetranacci numbers (basically fibonacci but with 4 number sequences instead of 2)

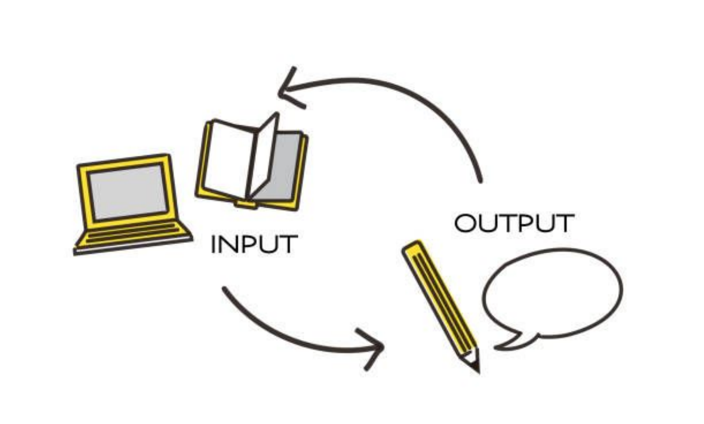
Step 2: Find the number of solid wall layouts by reversing the equation used to get total number of layouts and isolating just the solid layouts.

Code Implementation

Implemented in Python

Input and Output

```
if __name__ == "__main__":
    next(sys.stdin) # skip first line of input file
    for line in sys.stdin: # for each test case
        if not line.strip(): # if there is no new line then break
            break
        (Height, Width) = line.split() #split the line to get width and height
        (Height, Width) = (int(Height), int(Width)) #cast string values to int
```

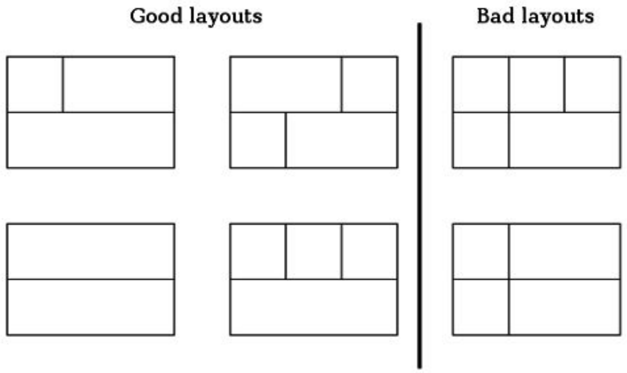


Main Algorithm – Dynamic Programming

Step 1: Find how many M*N walls can we build if we don't worry about making it a solid wall.

We can treat each row separately, and then multiply the counts since they are independent.

There is only one way to tile a 0*1 or a 1*1 wall, and the number of ways to tile an n*1 is the total of the number of ways to tile {n-1}*1...{n-4}*1-sized walls (these walls can be obtained by removing the last tile of the n*1 wall).



These are not all of the valid permutations. There are 9 valid permutations in all.

```
T(X)   = X > 0: T(X-1)+T(X-2)+T(X-3)+T(X-4)
         X = 0: 1
         X < 0: 0
```

```
A(W,H) = T(W)^H
```

The tetranacci numbers are a generalization of the Fibonacci numbers defined by $T_0 = 0$, $T_1 = 1$, $T_2 = 1$, $T_3 = 2$,

$$T_n = T_{n-1} + T_{n-2} + T_{n-3} + T_{n-4}$$

```python
Total = [0 for x in range(0, 1001)]
Solid = [0 for x in range(0, 1001)]
memo = {}
def totallegoblocks(Height, Width):
    track = [1]
    cur_width = 1 # First find out all possible ways to build a single row of a wall of a certain Width.
    while (cur_width < Width + 1):
        if cur_width >= len(track):
            if cur_width - 4 >= 0:
                track.append((track[cur_width - 4] + track[cur_width - 3] + track[cur_width - 2] + track[cur_width - 1]) % 1000000007)
            elif cur_width - 3 >= 0:
                track.append((track[cur_width - 3] + track[cur_width - 2] + track[cur_width - 1]) % 1000000007)
            elif cur_width - 2 >= 0:
                track.append((track[cur_width - 2] + track[cur_width - 1]) % 1000000007)
            elif cur_width - 1 >= 0:
                track.append((track[cur_width - 1]) % 1000000007)
            else:
                track.append(0)
        Total[cur_width] = power_memoized(track[cur_width], Height) #Get the number of ways to build a wall of certain Width and Height
        cur_width+=1

#function that returns the power of num ^ exponent. Memoized to speed up power function
def power_memoized(num, exp):
    key = str(num) +"," + str(exp) #store key as "num,exponent" in memo dictionary
    if key in memo: #if key already exists, return the value of the key
        return memo[key]
    else: #if key doesn't already exist, find the power and store in memo[key] by dividing and conquering the exponent
        if exp <= 2: # if exponent is small like 2 just find the power mod 1000000007 and store it
            memo[key] = (num ** exp) % 1000000007
        else: #otherwise, recursively call power_memoized function to divide up the powers and store it
            if exp % 2 == 0:
                memo[key] = (power_memoized(num, exp / 2) ** 2) % 1000000007
            else: #if power is odd, multiply num to the result of recursive call
                memo[key] = ((power_memoized(num, exp / 2) ** 2) * num) % 1000000007
        return memo[key]
```

Main Algorithm – Dynamic Programming

Step 2: Now we can count the number of Solid Walls from the total number of walls we just found.

Branch on the leftmost place where the wall is not connected. The number of all W*H walls is the number of Solid X*H walls times the number of all {W-X}*H walls, summed across all possible values of X, plus the number of Solid W*H walls.

```
S(W,H) = A(W,H) - sum_x( S(X,H)*A(W-X,H) )
```

```
cur_width = 1
while (cur_width < Width + 1): #Finally, using the array of Total Walls, we must figure out the total solid/unbreakable walls
    off = (1000000007 * 1000000007) * cur_width
    Solid[cur_width] = Total[cur_width] + off
    for i in range(1, cur_width):
        Solid[cur_width] = ((Solid[cur_width]) - Solid[i] * Total[cur_width - i])
    Solid[cur_width] = Solid[cur_width] % 1000000007
    cur_width+=1
print(Solid[Width])
```

Main Algorithm – Dynamic Programming

Analysis: The algorithm should run on a O(W^2), where W is the width, to find the number of solid blocks.

Pretty slow, but it's the best possible DP solution to the problem.

CODE DEMONSTRATION

https://www.hackerrank.com/challenges/lego-blocks

Special Thanks

-StackOverflow (https://stackoverflow.com/): algorithm understanding and debugging

-GeekforGeeks (https://www.geeksforgeeks.org/): syntax help