Problem 2312: Selling Pieces of Wood

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given two integers

m

and

n

that represent the height and width of a rectangular piece of wood. You are also given a 2D integer array

prices

, where

prices[i] = [h

i

, w

i

, price

i

]

indicates you can sell a rectangular piece of wood of height

h

i

and width

w

i

for

price

i

dollars.

To cut a piece of wood, you must make a vertical or horizontal cut across the

entire

height or width of the piece to split it into two smaller pieces. After cutting a piece of wood into some number of smaller pieces, you can sell pieces according to

prices

. You may sell multiple pieces of the same shape, and you do not have to sell all the shapes. The grain of the wood makes a difference, so you

cannot

rotate a piece to swap its height and width.

Return

the

maximum

money you can earn after cutting an

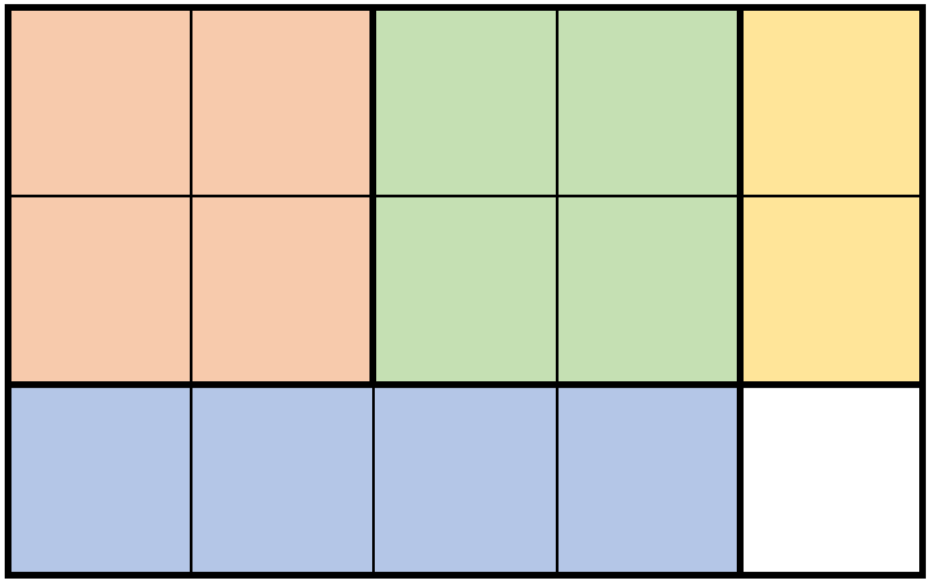
m x n

piece of wood

.

Note that you can cut the piece of wood as many times as you want.

Example 1:



Input:

m = 3, n = 5, prices = [[1,4,2],[2,2,7],[2,1,3]]

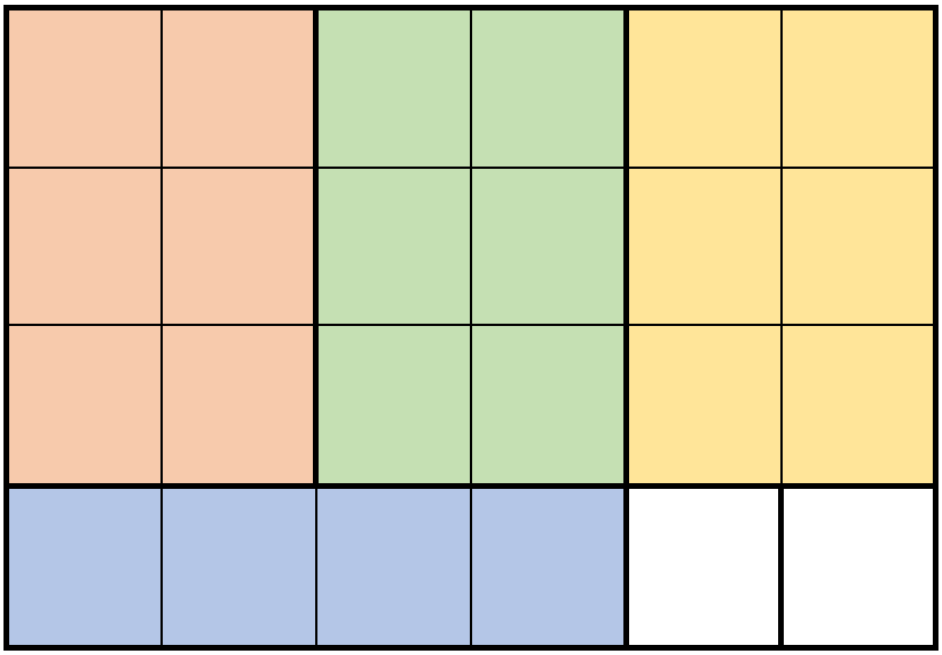
Output:

19

Explanation:

The diagram above shows a possible scenario. It consists of: - 2 pieces of wood shaped 2 x 2, selling for a price of 2 * 7 = 14. - 1 piece of wood shaped 2 x 1, selling for a price of 1 * 3 = 3. - 1 piece of wood shaped 1 x 4, selling for a price of 1 * 2 = 2. This obtains a total of 14 + 3 + 2 = 19 money earned. It can be shown that 19 is the maximum amount of money that can be earned.

Example 2:



Input:

m = 4, n = 6, prices = [[3,2,10],[1,4,2],[4,1,3]]

Output:

32

Explanation:

The diagram above shows a possible scenario. It consists of: - 3 pieces of wood shaped 3 x 2, selling for a price of 3 * 10 = 30. - 1 piece of wood shaped 1 x 4, selling for a price of 1 * 2 = 2. This obtains a total of 30 + 2 = 32 money earned. It can be shown that 32 is the maximum amount of money that can be earned. Notice that we cannot rotate the 1 x 4 piece of wood to obtain a 4 x 1 piece of wood.

Constraints:

1 <= m, n <= 200

1 <= prices.length <= 2 * 10

4

prices[i].length == 3

1 <= h

i

<= m

1 <= w

i

<= n

1 <= price

i

<= 10

6

All the shapes of wood

(h

i

, w

i

)

are pairwise

distinct

.

Code Snippets

C++:

```
class Solution {
public:
long long sellingWood(int m, int n, vector<vector<int>>& prices) {

}
};
```

Java:

```
class Solution {
public long sellingWood(int m, int n, int[][] prices) {

}
}
```

Python3:

```
class Solution:
def sellingWood(self, m: int, n: int, prices: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def sellingWood(self, m, n, prices):
"""
:type m: int
:type n: int
:type prices: List[List[int]]
```

```
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} m
* @param {number} n
* @param {number[][]} prices
* @return {number}
*/
var sellingWood = function(m, n, prices) {

};
```

TypeScript:

```
function sellingWood(m: number, n: number, prices: number[][]): number {

};
```

C#:

```
public class Solution {
public long SellingWood(int m, int n, int[][] prices) {

}
}
```

C:

```
long long sellingWood(int m, int n, int** prices, int pricesSize, int*
pricesColSize) {

}
```

Go:

```
func sellingWood(m int, n int, prices [][]int) int64 {

}
```

Kotlin:

```
class Solution {
fun sellingWood(m: Int, n: Int, prices: Array<IntArray>): Long {

}
}
```

Swift:

```
class Solution {
func sellingWood(_ m: Int, _ n: Int, _ prices: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn selling_wood(m: i32, n: i32, prices: Vec<Vec<i32>>) -> i64 {

}
}
```

Ruby:

```
# @param {Integer} m
# @param {Integer} n
# @param {Integer[][]} prices
# @return {Integer}
def selling_wood(m, n, prices)

end
```

PHP:

```
class Solution {

/**
* @param Integer $m
* @param Integer $n
* @param Integer[][] $prices
* @return Integer
```

```
*/
function sellingWood($m, $n, $prices) {

}
}
```

Dart:

```
class Solution {
int sellingWood(int m, int n, List<List<int>> prices) {

}
}
```

Scala:

```
object Solution {
def sellingWood(m: Int, n: Int, prices: Array[Array[Int]]): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec selling_wood(m :: integer, n :: integer, prices :: [[integer]]) ::
integer
def selling_wood(m, n, prices) do

end
end
```

Erlang:

```
-spec selling_wood(M :: integer(), N :: integer(), Prices :: [[integer()]])
-> integer().
selling_wood(M, N, Prices) ->
.
```

Racket:

```
(define/contract (selling-wood m n prices)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
long long sellingWood(int m, int n, vector<vector<int>>& prices) {

}
};
```

Java Solution:

```
/**
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public long sellingWood(int m, int n, int[][] prices) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Selling Pieces of Wood
Difficulty: Hard
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def sellingWood(self, m: int, n: int, prices: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def sellingWood(self, m, n, prices):
"""
:type m: int
:type n: int
:type prices: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

/**
* @param {number} m
* @param {number} n
* @param {number[][]} prices
* @return {number}
*/
var sellingWood = function(m, n, prices) {

};
```

TypeScript Solution:

```
/**
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function sellingWood(m: number, n: number, prices: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
```

```
public long SellingWood(int m, int n, int[][] prices) {

}
}
```

C Solution:

```
/*
* Problem: Selling Pieces of Wood
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

long long sellingWood(int m, int n, int** prices, int pricesSize, int*
pricesColSize) {

}
```

Go Solution:

```
// Problem: Selling Pieces of Wood
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func sellingWood(m int, n int, prices [][]int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun sellingWood(m: Int, n: Int, prices: Array<IntArray>): Long {
```

```
}
}
```

Swift Solution:

```
class Solution {
func sellingWood(_ m: Int, _ n: Int, _ prices: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Selling Pieces of Wood
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn selling_wood(m: i32, n: i32, prices: Vec<Vec<i32>>) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer} m
# @param {Integer} n
# @param {Integer[][]} prices
# @return {Integer}
def selling_wood(m, n, prices)

end
```

PHP Solution:

```
class Solution {
```

```
/**
* @param Integer $m
* @param Integer $n
* @param Integer[][] $prices
* @return Integer
*/
function sellingWood($m, $n, $prices) {

}
}
```

Dart Solution:

```
class Solution {
int sellingWood(int m, int n, List<List<int>> prices) {

}
}
```

Scala Solution:

```
object Solution {
def sellingWood(m: Int, n: Int, prices: Array[Array[Int]]): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec selling_wood(m :: integer, n :: integer, prices :: [[integer]]) ::
integer
def selling_wood(m, n, prices) do

end
end
```

Erlang Solution:

```
-spec selling_wood(M :: integer(), N :: integer(), Prices :: [[integer()]])
-> integer().
```

```
selling_wood(M, N, Prices) ->
.
```

Racket Solution:

```
(define/contract (selling-wood m n prices)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```