# Problem 1140: Stone Game II

## Problem Information

**Difficulty:** Medium
**Acceptance Rate:** 0.00%
**Paid Only:** No

## Problem Description

Alice and Bob continue their games with piles of stones. There are a number of piles

arranged in a row

, and each pile has a positive integer number of stones

piles[i]

. The objective of the game is to end with the most stones.

Alice and Bob take turns, with Alice starting first.

On each player's turn, that player can take

all the stones

in the

first

X

remaining piles, where

1 <= X <= 2M

. Then, we set

M = max(M, X)

. Initially, M = 1.

The game continues until all the stones have been taken.

Assuming Alice and Bob play optimally, return the maximum number of stones Alice can get.

Example 1:

Input:

piles = [2,7,9,4,4]

Output:

10

Explanation:

If Alice takes one pile at the beginning, Bob takes two piles, then Alice takes 2 piles again. Alice can get

2 + 4 + 4 = 10

stones in total.

If Alice takes two piles at the beginning, then Bob can take all three piles left. In this case, Alice get

2 + 7 = 9

stones in total.

So we return 10 since it's larger.

Example 2:

Input:

piles = [1,2,3,4,5,100]

Output:

104

Constraints:

1 <= piles.length <= 100

1 <= piles[i] <= 10

4

## Code Snippets

**C++:**

```
class Solution {
public:
int stoneGameII(vector<int>& piles) {

}
};
```

**Java:**

```
class Solution {
public int stoneGameII(int[] piles) {

}
}
```

**Python3:**

```
class Solution:
def stoneGameII(self, piles: List[int]) -> int:
```

**Python:**

```
class Solution(object):
def stoneGameII(self, piles):
"""
:type piles: List[int]
:rtype: int
"""
```

**JavaScript:**

```
/**
* @param {number[]} piles
* @return {number}
*/
var stoneGameII = function(piles) {

};
```

**TypeScript:**

```
function stoneGameII(piles: number[]): number {

};
```

**C#:**

```
public class Solution {
public int StoneGameII(int[] piles) {

}
}
```

**C:**

```
int stoneGameII(int* piles, int pilesSize) {

}
```

**Go:**

```
func stoneGameII(piles []int) int {
```

```
}
```

**Kotlin:**

```
class Solution {
fun stoneGameII(piles: IntArray): Int {

}
}
```

**Swift:**

```
class Solution {
func stoneGameII(_ piles: [Int]) -> Int {

}
}
```

**Rust:**

```
impl Solution {
pub fn stone_game_ii(piles: Vec<i32>) -> i32 {

}
}
```

**Ruby:**

```
# @param {Integer[]} piles
# @return {Integer}
def stone_game_ii(piles)

end
```

**PHP:**

```
class Solution {

/**
* @param Integer[] $piles
* @return Integer
*/
```

```
function stoneGameII($piles) {

}
}
```

**Dart:**

```
class Solution {
int stoneGameII(List<int> piles) {

}
}
```

**Scala:**

```
object Solution {
def stoneGameII(piles: Array[Int]): Int = {

}
}
```

**Elixir:**

```
defmodule Solution do
@spec stone_game_ii(piles :: [integer]) :: integer
def stone_game_ii(piles) do

end
end
```

**Erlang:**

```
-spec stone_game_ii(Piles :: [integer()]) -> integer().
stone_game_ii(Piles) ->
.
```

**Racket:**

```
(define/contract (stone-game-ii piles)
(-> (listof exact-integer?) exact-integer?)
)
```

## Solutions

### C++ Solution:

```
/*
* Problem: Stone Game II
* Difficulty: Medium
* Tags: array, dp, math
*
* 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:
int stoneGameII(vector<int>& piles) {

}
};
```

### Java Solution:

```
/**
* Problem: Stone Game II
* Difficulty: Medium
* Tags: array, dp, math
*
* 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 int stoneGameII(int[] piles) {

}
}
```

### Python3 Solution:

```
"""
Problem: Stone Game II
Difficulty: Medium
Tags: array, dp, math

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 stoneGameII(self, piles: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

### Python Solution:

```
class Solution(object):
def stoneGameII(self, piles):
"""
:type piles: List[int]
:rtype: int
"""
```

### JavaScript Solution:

```
/**
* Problem: Stone Game II
* Difficulty: Medium
* Tags: array, dp, math
*
* 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[]} piles
* @return {number}
*/
var stoneGameII = function(piles) {
```

```
};
```

### TypeScript Solution:

```
/**
* Problem: Stone Game II
* Difficulty: Medium
* Tags: array, dp, math
*
* 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 stoneGameII(piles: number[]): number {

};
```

### C# Solution:

```
/*
* Problem: Stone Game II
* Difficulty: Medium
* Tags: array, dp, math
*
* 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 int StoneGameII(int[] piles) {

}
}
```

### C Solution:

```
/*
* Problem: Stone Game II
* Difficulty: Medium
```

```
* Tags: array, dp, math
*
* 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
*/

int stoneGameII(int* piles, int pilesSize) {

}
```

### Go Solution:

```
// Problem: Stone Game II
// Difficulty: Medium
// Tags: array, dp, math
//
// 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 stoneGameII(piles []int) int {

}
```

### Kotlin Solution:

```
class Solution {
fun stoneGameII(piles: IntArray): Int {

}
}
```

### Swift Solution:

```
class Solution {
func stoneGameII(_ piles: [Int]) -> Int {

}
}
```

### Rust Solution:

```
// Problem: Stone Game II
// Difficulty: Medium
// Tags: array, dp, math
//
// 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 stone_game_ii(piles: Vec<i32>) -> i32 {

}
}
```

### Ruby Solution:

```
# @param {Integer[]} piles
# @return {Integer}
def stone_game_ii(piles)

end
```

### PHP Solution:

```
class Solution {

/**
* @param Integer[] $piles
* @return Integer
*/
function stoneGameII($piles) {

}
}
```

### Dart Solution:

```
class Solution {
int stoneGameII(List<int> piles) {
```

```
}
}
```

### Scala Solution:

```
object Solution {
def stoneGameII(piles: Array[Int]): Int = {

}
}
```

### Elixir Solution:

```
defmodule Solution do
@spec stone_game_ii(piles :: [integer]) :: integer
def stone_game_ii(piles) do

end
end
```

### Erlang Solution:

```
-spec stone_game_ii(Piles :: [integer()]) -> integer().
stone_game_ii(Piles) ->
.
```

### Racket Solution:

```
(define/contract (stone-game-ii piles)
(-> (listof exact-integer?) exact-integer?)
)
```