

# Problem 3242: Design Neighbor Sum Service

## Problem Information

Difficulty: [Easy](#)

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

You are given a

$n \times n$

2D array

grid

containing

distinct

elements in the range

$[0, n$

$2$

$- 1]$

.

Implement the

NeighborSum

class:

NeighborSum(int [][]grid)

initializes the object.

int adjacentSum(int value)

returns the

sum

of elements which are adjacent neighbors of

value

, that is either to the top, left, right, or bottom of

value

in

grid

.

int diagonalSum(int value)

returns the

sum

of elements which are diagonal neighbors of

value

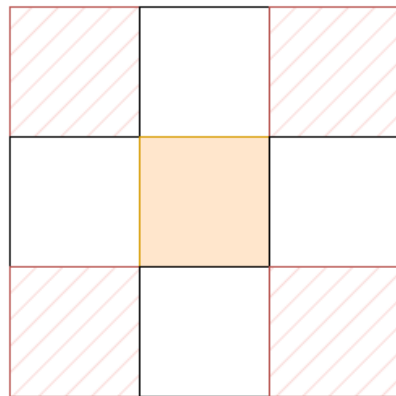
, that is either to the top-left, top-right, bottom-left, or bottom-right of

value

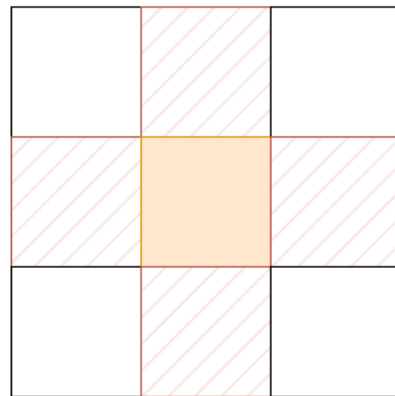
in

grid

.



**Diagonal  
to  
Middle  
Cell**



**Adjacent  
to  
Middle  
Cell**

Example 1:

Input:

["NeighborSum", "adjacentSum", "adjacentSum", "diagonalSum", "diagonalSum"]

[[[0, 1, 2], [3, 4, 5], [6, 7, 8]], [1], [4], [4], [8]]

Output:

[null, 6, 16, 16, 4]

Explanation:

0	1	2
3	4	5
6	7	8

The adjacent neighbors of 1 are 0, 2, and 4.

The adjacent neighbors of 4 are 1, 3, 5, and 7.

The diagonal neighbors of 4 are 0, 2, 6, and 8.

The diagonal neighbor of 8 is 4.

Example 2:

Input:

["NeighborSum", "adjacentSum", "diagonalSum"]

[[[1, 2, 0, 3], [4, 7, 15, 6], [8, 9, 10, 11], [12, 13, 14, 5]], [15], [9]]

Output:

[null, 23, 45]

Explanation:

1	2	0	3
4	7	15	6
8	9	10	11
12	13	14	5

The adjacent neighbors of 15 are 0, 10, 7, and 6.

The diagonal neighbors of 9 are 4, 12, 14, and 15.

Constraints:

$3 \leq n == \text{grid.length} == \text{grid}[0].\text{length} \leq 10$

$0 \leq \text{grid}[i][j] \leq n$

2

- 1

All

`grid[i][j]`

are distinct.

value

in

adjacentSum

and

diagonalSum

will be in the range

$[0, n$

$2$

$- 1]$

.

At most

$2 * n$

$2$

calls will be made to

adjacentSum

and

diagonalSum

.

## Code Snippets

### C++:

```
class NeighborSum {
public:
    NeighborSum(vector<vector<int>>& grid) {

    }

    int adjacentSum(int value) {

    }

    int diagonalSum(int value) {

    }
};

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum* obj = new NeighborSum(grid);
 * int param_1 = obj->adjacentSum(value);
 * int param_2 = obj->diagonalSum(value);
 */
```

### Java:

```
class NeighborSum {

    public NeighborSum(int[][] grid) {

    }

    public int adjacentSum(int value) {

    }

    public int diagonalSum(int value) {

    }

}
```

```

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = new NeighborSum(grid);
 * int param_1 = obj.adjacentSum(value);
 * int param_2 = obj.diagonalSum(value);
 */

```

### Python3:

```

class NeighborSum:

    def __init__(self, grid: List[List[int]]):

    def adjacentSum(self, value: int) -> int:

    def diagonalSum(self, value: int) -> int:


# Your NeighborSum object will be instantiated and called as such:
# obj = NeighborSum(grid)
# param_1 = obj.adjacentSum(value)
# param_2 = obj.diagonalSum(value)

```

### Python:

```

class NeighborSum(object):

    def __init__(self, grid):
        """
        :type grid: List[List[int]]
        """

    def adjacentSum(self, value):
        """
        :type value: int
        :rtype: int

```

```

"""

def diagonalSum(self, value):
    """
    :type value: int
    :rtype: int
    """

# Your NeighborSum object will be instantiated and called as such:
# obj = NeighborSum(grid)
# param_1 = obj.adjacentSum(value)
# param_2 = obj.diagonalSum(value)

```

## JavaScript:

```

/**
 * @param {number[][]} grid
 */
var NeighborSum = function(grid) {

};

/**
 * @param {number} value
 * @return {number}
 */
NeighborSum.prototype.adjacentSum = function(value) {

};

/**
 * @param {number} value
 * @return {number}
 */
NeighborSum.prototype.diagonalSum = function(value) {

};

/**

```

```
* Your NeighborSum object will be instantiated and called as such:  
* var obj = new NeighborSum(grid)  
* var param_1 = obj.adjacentSum(value)  
* var param_2 = obj.diagonalSum(value)  
*/
```

### TypeScript:

```
class NeighborSum {  
  constructor(grid: number[][]) {  
  
  }  
  
  adjacentSum(value: number): number {  
  
  }  
  
  diagonalSum(value: number): number {  
  
  }  
}  
  
/**  
 * Your NeighborSum object will be instantiated and called as such:  
 * var obj = new NeighborSum(grid)  
 * var param_1 = obj.adjacentSum(value)  
 * var param_2 = obj.diagonalSum(value)  
 */
```

### C#:

```
public class NeighborSum {  
  
  public NeighborSum(int[][] grid) {  
  
  }  
  
  public int AdjacentSum(int value) {  
  
  }  
  
  public int DiagonalSum(int value) {  
  
  }  
}
```

```

}
}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = new NeighborSum(grid);
 * int param_1 = obj.AdjacentSum(value);
 * int param_2 = obj.DiagonalSum(value);
 */

```

**C:**

```

typedef struct {

} NeighborSum;

NeighborSum* neighborSumCreate(int** grid, int gridSize, int* gridColSize) {

}

int neighborSumAdjacentSum(NeighborSum* obj, int value) {

}

int neighborSumDiagonalSum(NeighborSum* obj, int value) {

}

void neighborSumFree(NeighborSum* obj) {

}

/**
 * Your NeighborSum struct will be instantiated and called as such:
 * NeighborSum* obj = neighborSumCreate(grid, gridSize, gridColSize);
 * int param_1 = neighborSumAdjacentSum(obj, value);
 */

```

```

* int param_2 = neighborSumDiagonalSum(obj, value);

* neighborSumFree(obj);
*/

```

## Go:

```

type NeighborSum struct {

}

func Constructor(grid [][]int) NeighborSum {

}

func (this *NeighborSum) AdjacentSum(value int) int {

}

func (this *NeighborSum) DiagonalSum(value int) int {

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * obj := Constructor(grid);
 * param_1 := obj.AdjacentSum(value);
 * param_2 := obj.DiagonalSum(value);
 */

```

## Kotlin:

```

class NeighborSum(grid: Array<IntArray>) {

    fun adjacentSum(value: Int): Int {

    }

}

```

```

fun diagonalSum(value: Int): Int {

}

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * var obj = NeighborSum(grid)
 * var param_1 = obj.adjacentSum(value)
 * var param_2 = obj.diagonalSum(value)
 */

```

### Swift:

```

class NeighborSum {

    init(_ grid: [[Int]]) {

    }

    func adjacentSum(_ value: Int) -> Int {

    }

    func diagonalSum(_ value: Int) -> Int {

    }
}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * let obj = NeighborSum(grid)
 * let ret_1: Int = obj.adjacentSum(value)
 * let ret_2: Int = obj.diagonalSum(value)
 */

```

### Rust:

```

struct NeighborSum {

```

```

}

/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl NeighborSum {

fn new(grid: Vec<Vec<i32>>) -> Self {

}

fn adjacent_sum(&self, value: i32) -> i32 {

}

fn diagonal_sum(&self, value: i32) -> i32 {

}
}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * let obj = NeighborSum::new(grid);
 * let ret_1: i32 = obj.adjacent_sum(value);
 * let ret_2: i32 = obj.diagonal_sum(value);
 */

```

## Ruby:

```

class NeighborSum

  =begin
  :type grid: Integer[][]
  =end

  def initialize(grid)

  end

  =begin

```

```

:type value: Integer
:rtype: Integer
=end
def adjacent_sum(value)

end

=begin
:type value: Integer
:rtype: Integer
=end
def diagonal_sum(value)

end

end

# Your NeighborSum object will be instantiated and called as such:
# obj = NeighborSum.new(grid)
# param_1 = obj.adjacent_sum(value)
# param_2 = obj.diagonal_sum(value)

```

## PHP:

```

class NeighborSum {
    /**
     * @param Integer[][] $grid
     */
    function __construct($grid) {

    }

    /**
     * @param Integer $value
     * @return Integer
     */
    function adjacentSum($value) {

    }
}

```

```

/**
 * @param Integer $value
 * @return Integer
 */
function diagonalSum($value) {

}

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * $obj = NeighborSum($grid);
 * $ret_1 = $obj->adjacentSum($value);
 * $ret_2 = $obj->diagonalSum($value);
 */

```

#### Dart:

```

class NeighborSum {

  NeighborSum(List<List<int>> grid) {

  }

  int adjacentSum(int value) {

  }

  int diagonalSum(int value) {

  }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = NeighborSum(grid);
 * int param1 = obj.adjacentSum(value);
 * int param2 = obj.diagonalSum(value);
 */

```

#### Scala:

```

class NeighborSum(_grid: Array[Array[Int]]) {

  def adjacentSum(value: Int): Int = {

  }

  def diagonalSum(value: Int): Int = {

  }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * val obj = new NeighborSum(grid)
 * val param_1 = obj.adjacentSum(value)
 * val param_2 = obj.diagonalSum(value)
 */

```

## Elixir:

```

defmodule NeighborSum do
  @spec init_(grid :: [[integer]]) :: any
  def init_(grid) do

  end

  @spec adjacent_sum(value :: integer) :: integer
  def adjacent_sum(value) do

  end

  @spec diagonal_sum(value :: integer) :: integer
  def diagonal_sum(value) do

  end

end

# Your functions will be called as such:
# NeighborSum.init_(grid)
# param_1 = NeighborSum.adjacent_sum(value)
# param_2 = NeighborSum.diagonal_sum(value)

```

```
# NeighborSum.init_ will be called before every test case, in which you can
do some necessary initializations.
```

## Erlang:

```
-spec neighbor_sum_init_(Grid :: [[integer()]]) -> any().
neighbor_sum_init_(Grid) ->
.

-spec neighbor_sum_adjacent_sum(Value :: integer()) -> integer().
neighbor_sum_adjacent_sum(Value) ->
.

-spec neighbor_sum_diagonal_sum(Value :: integer()) -> integer().
neighbor_sum_diagonal_sum(Value) ->
.

%% Your functions will be called as such:
%% neighbor_sum_init_(Grid),
%% Param_1 = neighbor_sum_adjacent_sum(Value),
%% Param_2 = neighbor_sum_diagonal_sum(Value),

%% neighbor_sum_init_ will be called before every test case, in which you can
do some necessary initializations.
```

## Racket:

```
(define neighbor-sum%
  (class object%
    (super-new)

    ; grid : (listof (listof exact-integer?))
    (init-field
      grid)

    ; adjacent-sum : exact-integer? -> exact-integer?
    (define/public (adjacent-sum value)
      )

    ; diagonal-sum : exact-integer? -> exact-integer?
    (define/public (diagonal-sum value)
      ))))
```

```
// Your neighbor-sum% object will be instantiated and called as such:  
// (define obj (new neighbor-sum% [grid grid]))  
// (define param_1 (send obj adjacent-sum value))  
// (define param_2 (send obj diagonal-sum value))
```

## Solutions

### C++ Solution:

```
/*  
 * Problem: Design Neighbor Sum Service  
 * Difficulty: Easy  
 * Tags: array, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
class NeighborSum {  
public:  
    NeighborSum(vector<vector<int>>& grid) {  
  
    }  
  
    int adjacentSum(int value) {  
  
    }  
  
    int diagonalSum(int value) {  
  
    }  
};  
  
/**  
 * Your NeighborSum object will be instantiated and called as such:  
 * NeighborSum* obj = new NeighborSum(grid);  
 * int param_1 = obj->adjacentSum(value);  
 * int param_2 = obj->diagonalSum(value);  
 */
```

```
*/
```

## Java Solution:

```
/**
 * Problem: Design Neighbor Sum Service
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class NeighborSum {

    public NeighborSum(int[][] grid) {

    }

    public int adjacentSum(int value) {

    }

    public int diagonalSum(int value) {

    }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = new NeighborSum(grid);
 * int param_1 = obj.adjacentSum(value);
 * int param_2 = obj.diagonalSum(value);
 */
```

## Python3 Solution:

```
"""
Problem: Design Neighbor Sum Service
Difficulty: Easy
```

Tags: array, hash

Approach: Use two pointers or sliding window technique

Time Complexity:  $O(n)$  or  $O(n \log n)$

Space Complexity:  $O(n)$  for hash map

"""

```
class NeighborSum:
```

```
def __init__(self, grid: List[List[int]]):
```

```
def adjacentSum(self, value: int) -> int:
```

```
# TODO: Implement optimized solution
```

```
pass
```

## Python Solution:

```
class NeighborSum(object):
```

```
def __init__(self, grid):
```

```
"""
```

```
:type grid: List[List[int]]
```

```
"""
```

```
def adjacentSum(self, value):
```

```
"""
```

```
:type value: int
```

```
:rtype: int
```

```
"""
```

```
def diagonalSum(self, value):
```

```
"""
```

```
:type value: int
```

```
:rtype: int
```

```
"""
```

```
# Your NeighborSum object will be instantiated and called as such:
# obj = NeighborSum(grid)
# param_1 = obj.adjacentSum(value)
# param_2 = obj.diagonalSum(value)
```

## JavaScript Solution:

```
/**
 * Problem: Design Neighbor Sum Service
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

/**
 * @param {number[][]} grid
 */
var NeighborSum = function(grid) {

};

/**
 * @param {number} value
 * @return {number}
 */
NeighborSum.prototype.adjacentSum = function(value) {

};

/**
 * @param {number} value
 * @return {number}
 */
NeighborSum.prototype.diagonalSum = function(value) {

};

/**
```

```
* Your NeighborSum object will be instantiated and called as such:
* var obj = new NeighborSum(grid)
* var param_1 = obj.adjacentSum(value)
* var param_2 = obj.diagonalSum(value)
*/
```

## TypeScript Solution:

```
/**
 * Problem: Design Neighbor Sum Service
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class NeighborSum {
  constructor(grid: number[][]) {

  }

  adjacentSum(value: number): number {

  }

  diagonalSum(value: number): number {

  }
}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * var obj = new NeighborSum(grid)
 * var param_1 = obj.adjacentSum(value)
 * var param_2 = obj.diagonalSum(value)
 */
```

## C# Solution:

```

/*
 * Problem: Design Neighbor Sum Service
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

public class NeighborSum {

    public NeighborSum(int[][] grid) {

    }

    public int AdjacentSum(int value) {

    }

    public int DiagonalSum(int value) {

    }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = new NeighborSum(grid);
 * int param_1 = obj.AdjacentSum(value);
 * int param_2 = obj.DiagonalSum(value);
 */

```

## C Solution:

```

/*
 * Problem: Design Neighbor Sum Service
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map

```

```

*/

typedef struct {

} NeighborSum;

NeighborSum* neighborSumCreate(int** grid, int gridSize, int* gridColSize) {

}

int neighborSumAdjacentSum(NeighborSum* obj, int value) {

}

int neighborSumDiagonalSum(NeighborSum* obj, int value) {

}

void neighborSumFree(NeighborSum* obj) {

}

/**
 * Your NeighborSum struct will be instantiated and called as such:
 * NeighborSum* obj = neighborSumCreate(grid, gridSize, gridColSize);
 * int param_1 = neighborSumAdjacentSum(obj, value);
 *
 * int param_2 = neighborSumDiagonalSum(obj, value);
 *
 * neighborSumFree(obj);
 */

```

### Go Solution:

```

// Problem: Design Neighbor Sum Service
// Difficulty: Easy
// Tags: array, hash

```

```
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type NeighborSum struct {

}

func Constructor(grid [][]int) NeighborSum {

}

func (this *NeighborSum) AdjacentSum(value int) int {

}

func (this *NeighborSum) DiagonalSum(value int) int {

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * obj := Constructor(grid);
 * param_1 := obj.AdjacentSum(value);
 * param_2 := obj.DiagonalSum(value);
 */
```

### Kotlin Solution:

```
class NeighborSum(grid: Array<IntArray>) {

    fun adjacentSum(value: Int): Int {

    }

    fun diagonalSum(value: Int): Int {

    }

}
```

```

}

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * var obj = NeighborSum(grid)
 * var param_1 = obj.adjacentSum(value)
 * var param_2 = obj.diagonalSum(value)
 */

```

### Swift Solution:

```

class NeighborSum {

    init(_ grid: [[Int]]) {

    }

    func adjacentSum(_ value: Int) -> Int {

    }

    func diagonalSum(_ value: Int) -> Int {

    }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * let obj = NeighborSum(grid)
 * let ret_1: Int = obj.adjacentSum(value)
 * let ret_2: Int = obj.diagonalSum(value)
 */

```

### Rust Solution:

```

// Problem: Design Neighbor Sum Service
// Difficulty: Easy

```

```

// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct NeighborSum {

}

/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl NeighborSum {

fn new(grid: Vec<Vec<i32>>>) -> Self {

}

fn adjacent_sum(&self, value: i32) -> i32 {

}

fn diagonal_sum(&self, value: i32) -> i32 {

}
}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * let obj = NeighborSum::new(grid);
 * let ret_1: i32 = obj.adjacent_sum(value);
 * let ret_2: i32 = obj.diagonal_sum(value);
 */

```

### Ruby Solution:

```

class NeighborSum

```

```

=begin
:type grid: Integer[][]
=end
def initialize(grid)

end

=begin
:type value: Integer
:rtype: Integer
=end
def adjacent_sum(value)

end

=begin
:type value: Integer
:rtype: Integer
=end
def diagonal_sum(value)

end

end

# Your NeighborSum object will be instantiated and called as such:
# obj = NeighborSum.new(grid)
# param_1 = obj.adjacent_sum(value)
# param_2 = obj.diagonal_sum(value)

```

### PHP Solution:

```

class NeighborSum {
/**
 * @param Integer[][] $grid
 */
function __construct($grid) {

```

```

}

/**
 * @param Integer $value
 * @return Integer
 */
function adjacentSum($value) {

}

/**
 * @param Integer $value
 * @return Integer
 */
function diagonalSum($value) {

}

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * $obj = NeighborSum($grid);
 * $ret_1 = $obj->adjacentSum($value);
 * $ret_2 = $obj->diagonalSum($value);
 */

```

### Dart Solution:

```

class NeighborSum {

  NeighborSum(List<List<int>> grid) {

  }

  int adjacentSum(int value) {

  }

  int diagonalSum(int value) {

  }

}

```

```

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * NeighborSum obj = NeighborSum(grid);
 * int param1 = obj.adjacentSum(value);
 * int param2 = obj.diagonalSum(value);
 */

```

### Scala Solution:

```

class NeighborSum(_grid: Array[Array[Int]]) {

  def adjacentSum(value: Int): Int = {

  }

  def diagonalSum(value: Int): Int = {

  }

}

/**
 * Your NeighborSum object will be instantiated and called as such:
 * val obj = new NeighborSum(grid)
 * val param_1 = obj.adjacentSum(value)
 * val param_2 = obj.diagonalSum(value)
 */

```

### Elixir Solution:

```

defmodule NeighborSum do
  @spec init_(grid :: [[integer]]) :: any
  def init_(grid) do

  end

  @spec adjacent_sum(value :: integer) :: integer
  def adjacent_sum(value) do

```

```

end

@spec diagonal_sum(value :: integer) :: integer
def diagonal_sum(value) do

end

end

# Your functions will be called as such:
# NeighborSum.init_(grid)
# param_1 = NeighborSum.adjacent_sum(value)
# param_2 = NeighborSum.diagonal_sum(value)

# NeighborSum.init_ will be called before every test case, in which you can
do some necessary initializations.

```

### Erlang Solution:

```

-spec neighbor_sum_init_(Grid :: [[integer()]]) -> any().
neighbor_sum_init_(Grid) ->
.

-spec neighbor_sum_adjacent_sum(Value :: integer()) -> integer().
neighbor_sum_adjacent_sum(Value) ->
.

-spec neighbor_sum_diagonal_sum(Value :: integer()) -> integer().
neighbor_sum_diagonal_sum(Value) ->
.

%% Your functions will be called as such:
%% neighbor_sum_init_(Grid),
%% Param_1 = neighbor_sum_adjacent_sum(Value),
%% Param_2 = neighbor_sum_diagonal_sum(Value),

%% neighbor_sum_init_ will be called before every test case, in which you can
do some necessary initializations.

```

### Racket Solution:

```
(define neighbor-sum%  
  (class object%  
    (super-new)  
  
    ; grid : (listof (listof exact-integer?))  
    (init-field  
      grid)  
  
    ; adjacent-sum : exact-integer? -> exact-integer?  
    (define/public (adjacent-sum value)  
      )  
  
    ; diagonal-sum : exact-integer? -> exact-integer?  
    (define/public (diagonal-sum value)  
      )))  
  
;; Your neighbor-sum% object will be instantiated and called as such:  
;; (define obj (new neighbor-sum% [grid grid]))  
;; (define param_1 (send obj adjacent-sum value))  
;; (define param_2 (send obj diagonal-sum value))
```