

# Problem 307: Range Sum Query - Mutable

## Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

Given an integer array

nums

, handle multiple queries of the following types:

Update

the value of an element in

nums

Calculate the

sum

of the elements of

nums

between indices

left

and

right

inclusive

where

$\text{left} \leq \text{right}$

Implement the

NumArray

class:

NumArray(int[] nums)

Initializes the object with the integer array

nums

void update(int index, int val)

Updates

the value of

nums[index]

to be

val

```
int sumRange(int left, int right)
```

Returns the

sum

of the elements of

nums

between indices

left

and

right

inclusive

(i.e.

```
nums[left] + nums[left + 1] + ... + nums[right]
```

).

Example 1:

Input

```
["NumArray", "sumRange", "update", "sumRange"] [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
```

Output

```
[null, 9, null, 8]
```

Explanation

```
NumArray numArray = new NumArray([1, 3, 5]); numArray.sumRange(0, 2); // return 1 + 3 + 5  
= 9 numArray.update(1, 2); // nums = [1, 2, 5] numArray.sumRange(0, 2); // return 1 + 2 + 5 =  
8
```

Constraints:

$1 \leq \text{nums.length} \leq 3 * 10$

4

$-100 \leq \text{nums}[i] \leq 100$

$0 \leq \text{index} < \text{nums.length}$

$-100 \leq \text{val} \leq 100$

$0 \leq \text{left} \leq \text{right} < \text{nums.length}$

At most

$3 * 10$

4

calls will be made to

update

and

sumRange

## Code Snippets

C++:

```

class NumArray {
public:
NumArray(vector<int>& nums) {

}

void update(int index, int val) {

}

int sumRange(int left, int right) {

};

/***
* Your NumArray object will be instantiated and called as such:
* NumArray* obj = new NumArray(nums);
* obj->update(index,val);
* int param_2 = obj->sumRange(left,right);
*/

```

### **Java:**

```

class NumArray {

public NumArray(int[] nums) {

}

public void update(int index, int val) {

}

public int sumRange(int left, int right) {

};

/***
* Your NumArray object will be instantiated and called as such:
* NumArray obj = new NumArray(nums);
* obj.update(index,val);
*/

```

```
* int param_2 = obj.sumRange(left,right);  
*/
```

### Python3:

```
class NumArray:  
  
    def __init__(self, nums: List[int]):  
  
        def update(self, index: int, val: int) -> None:  
  
            def sumRange(self, left: int, right: int) -> int:  
  
                # Your NumArray object will be instantiated and called as such:  
                # obj = NumArray(nums)  
                # obj.update(index,val)  
                # param_2 = obj.sumRange(left,right)
```

### Python:

```
class NumArray(object):  
  
    def __init__(self, nums):  
        """  
        :type nums: List[int]  
        """  
  
        def update(self, index, val):  
            """  
            :type index: int  
            :type val: int  
            :rtype: None  
            """  
  
            def sumRange(self, left, right):  
                """
```

```

:type left: int
:type right: int
:rtype: int
"""

# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)

```

### JavaScript:

```

/**
 * @param {number[]} nums
 */
var NumArray = function(nums) {

};

/**
 * @param {number} index
 * @param {number} val
 * @return {void}
 */
NumArray.prototype.update = function(index, val) {

};

/**
 * @param {number} left
 * @param {number} right
 * @return {number}
 */
NumArray.prototype.sumRange = function(left, right) {

};

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)

```

```
* obj.update(index,val)
* var param_2 = obj.sumRange(left,right)
*/
```

### TypeScript:

```
class NumArray {
constructor(nums: number[]) {

}

update(index: number, val: number): void {

}

sumRange(left: number, right: number): number {

}
}

/**
* Your NumArray object will be instantiated and called as such:
* var obj = new NumArray(nums)
* obj.update(index,val)
* var param_2 = obj.sumRange(left,right)
*/

```

### C#:

```
public class NumArray {

public NumArray(int[] nums) {

}

public void Update(int index, int val) {

}

public int SumRange(int left, int right) {

}
```

```
}
```

```
/**
```

```
* Your NumArray object will be instantiated and called as such:
```

```
* NumArray obj = new NumArray(nums);
```

```
* obj.Update(index,val);
```

```
* int param_2 = obj.SumRange(left,right);
```

```
*/
```

C:

```
typedef struct {
```

```
}
```

```
NumArray* numArrayCreate(int* nums, int numsSize) {
```

```
}
```

```
void numArrayUpdate(NumArray* obj, int index, int val) {
```

```
}
```

```
int numArraySumRange(NumArray* obj, int left, int right) {
```

```
}
```

```
void numArrayFree(NumArray* obj) {
```

```
}
```

```
/**
```

```
* Your NumArray struct will be instantiated and called as such:
```

```
* NumArray* obj = numArrayCreate(nums, numsSize);
```

```
* numArrayUpdate(obj, index, val);
```

```
* int param_2 = numArraySumRange(obj, left, right);
```

```
* numArrayFree(obj);
*/
```

## Go:

```
type NumArray struct {

}

func Constructor(nums []int) NumArray {

}

func (this *NumArray) Update(index int, val int) {

}

func (this *NumArray) SumRange(left int, right int) int {

}

/**
 * Your NumArray object will be instantiated and called as such:
 * obj := Constructor(nums);
 * obj.Update(index,val);
 * param_2 := obj.SumRange(left,right);
 */

```

## Kotlin:

```
class NumArray(nums: IntArray) {

    fun update(index: Int, `val`: Int) {

    }

    fun sumRange(left: Int, right: Int): Int {
```

```
}
```

```
}
```

```
/**
```

```
* Your NumArray object will be instantiated and called as such:
```

```
* var obj = NumArray(nums)
```

```
* obj.update(index,`val`)
```

```
* var param_2 = obj.sumRange(left,right)
```

```
*/
```

### Swift:

```
class NumArray {
```

```
    init(_ nums: [Int]) {
```

```
    }
```

```
    func update(_ index: Int, _ val: Int) {
```

```
    }
```

```
    func sumRange(_ left: Int, _ right: Int) -> Int {
```

```
    }
```

```
}
```

```
/**
```

```
* Your NumArray object will be instantiated and called as such:
```

```
* let obj = NumArray(nums)
```

```
* obj.update(index, val)
```

```
* let ret_2: Int = obj.sumRange(left, right)
```

```
*/
```

### Rust:

```
struct NumArray {
```

```
}
```

```

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

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

    fn update(&self, index: i32, val: i32) {
        }

    fn sum_range(&self, left: i32, right: i32) -> i32 {
        }
    }

    /**
     * Your NumArray object will be instantiated and called as such:
     * let obj = NumArray::new(nums);
     * obj.update(index, val);
     * let ret_2: i32 = obj.sum_range(left, right);
     */
}

```

## Ruby:

```

class NumArray

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

end

=begin
:type index: Integer
:type val: Integer

```

```

:rtype: Void
=end

def update(index, val)

end

=begin
:type left: Integer
:type right: Integer
:rtype: Integer
=end

def sum_range(left, right)

end

end

# Your NumArray object will be instantiated and called as such:
# obj = NumArray.new(nums)
# obj.update(index, val)
# param_2 = obj.sum_range(left, right)

```

## PHP:

```

class NumArray {

    /**
     * @param Integer[] $nums
     */
    function __construct($nums) {

    }

    /**
     * @param Integer $index
     * @param Integer $val
     * @return NULL
     */
    function update($index, $val) {

    }
}

```

```

/**
 * @param Integer $left
 * @param Integer $right
 * @return Integer
 */
function sumRange($left, $right) {

}

/**
 * Your NumArray object will be instantiated and called as such:
 * $obj = NumArray($nums);
 * $obj->update($index, $val);
 * $ret_2 = $obj->sumRange($left, $right);
 */

```

## Dart:

```

class NumArray {

    NumArray(List<int> nums) {

    }

    void update(int index, int val) {

    }

    int sumRange(int left, int right) {

    }

}

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = NumArray(nums);
 * obj.update(index,val);
 * int param2 = obj.sumRange(left,right);
 */

```

## Scala:

```
class NumArray(_nums: Array[Int]) {  
  
  def update(index: Int, `val`: Int): Unit = {  
  
  }  
  
  def sumRange(left: Int, right: Int): Int = {  
  
  }  
  
}  
  
/**  
 * Your NumArray object will be instantiated and called as such:  
 * val obj = new NumArray(nums)  
 * obj.update(index,`val`)  
 * val param_2 = obj.sumRange(left,right)  
 */
```

## Elixir:

```
defmodule NumArray do  
  @spec init_(nums :: [integer]) :: any  
  def init_(nums) do  
  
  end  
  
  @spec update(index :: integer, val :: integer) :: any  
  def update(index, val) do  
  
  end  
  
  @spec sum_range(left :: integer, right :: integer) :: integer  
  def sum_range(left, right) do  
  
  end  
  
  end  
  
# Your functions will be called as such:  
# NumArray.init_(nums)
```

```

# NumArray.update(index, val)
# param_2 = NumArray.sum_range(left, right)

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

```

### Erlang:

```

-spec num_array_init_(Nums :: [integer()]) -> any().
num_array_init_(Nums) ->
.

-spec num_array_update(Index :: integer(), Val :: integer()) -> any().
num_array_update(Index, Val) ->
.

-spec num_array_sum_range(Left :: integer(), Right :: integer()) ->
integer().
num_array_sum_range(Left, Right) ->
.

%% Your functions will be called as such:
%% num_array_init_(Nums),
%% num_array_update(Index, Val),
%% Param_2 = num_array_sum_range(Left, Right),

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

```

### Racket:

```

(define num-array%
  (class object%
    (super-new)

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

    ; update : exact-integer? exact-integer? -> void?
    (define/public (update index val)

```

```

)
; sum-range : exact-integer? exact-integer? -> exact-integer?
(define/public (sum-range left right)
 ))

;; Your num-array% object will be instantiated and called as such:
;; (define obj (new num-array% [nums nums]))
;; (send obj update index val)
;; (define param_2 (send obj sum-range left right))

```

## Solutions

### C++ Solution:

```

/*
 * Problem: Range Sum Query - Mutable
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

class NumArray {
public:
    NumArray(vector<int>& nums) {

    }

    void update(int index, int val) {

    }

    int sumRange(int left, int right) {

    }
};

/***

```

```
* Your NumArray object will be instantiated and called as such:  
* NumArray* obj = new NumArray(nums);  
* obj->update(index,val);  
* int param_2 = obj->sumRange(left,right);  
*/
```

### Java Solution:

```
/**  
 * Problem: Range Sum Query - Mutable  
 * Difficulty: Medium  
 * Tags: array, tree  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(h) for recursion stack where h is height  
 */  
  
class NumArray {  
  
    public NumArray(int[] nums) {  
  
    }  
  
    public void update(int index, int val) {  
  
    }  
  
    public int sumRange(int left, int right) {  
  
    }  
}  
  
/**  
 * Your NumArray object will be instantiated and called as such:  
 * NumArray obj = new NumArray(nums);  
 * obj.update(index,val);  
 * int param_2 = obj.sumRange(left,right);  
 */
```

### Python3 Solution:

```

"""
Problem: Range Sum Query - Mutable
Difficulty: Medium
Tags: array, tree

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class NumArray:

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

        def update(self, index: int, val: int) -> None:
            # TODO: Implement optimized solution
            pass

```

## Python Solution:

```

class NumArray(object):

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

    def update(self, index, val):
        """
        :type index: int
        :type val: int
        :rtype: None
        """

    def sumRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: int

```

```
"""
# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)
```

### JavaScript Solution:

```
/**
 * Problem: Range Sum Query - Mutable
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * @param {number[]} nums
 */
var NumArray = function(nums) {

};

/**
 * @param {number} index
 * @param {number} val
 * @return {void}
 */
NumArray.prototype.update = function(index, val) {

};

/**
 * @param {number} left
 * @param {number} right
 * @return {number}

```

```

/*
NumArray.prototype.sumRange = function(left, right) {

};

/** 
* Your NumArray object will be instantiated and called as such:
* var obj = new NumArray(nums)
* obj.update(index,val)
* var param_2 = obj.sumRange(left,right)
*/

```

### TypeScript Solution:

```

/** 
* Problem: Range Sum Query - Mutable
* Difficulty: Medium
* Tags: array, tree
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class NumArray {
constructor(nums: number[]) {

}

update(index: number, val: number): void {

}

sumRange(left: number, right: number): number {

}

}

/** 
* Your NumArray object will be instantiated and called as such:
* var obj = new NumArray(nums)

```

```
* obj.update(index,val)
* var param_2 = obj.sumRange(left,right)
*/
```

### C# Solution:

```
/*
 * Problem: Range Sum Query - Mutable
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

public class NumArray {

    public NumArray(int[] nums) {

    }

    public void Update(int index, int val) {

    }

    public int SumRange(int left, int right) {

    }
}

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * obj.Update(index,val);
 * int param_2 = obj.SumRange(left,right);
 */
```

### C Solution:

```

/*
 * Problem: Range Sum Query - Mutable
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

typedef struct {

} NumArray;

NumArray* numArrayCreate(int* nums, int numsSize) {

}

void numArrayUpdate(NumArray* obj, int index, int val) {

}

int numArraySumRange(NumArray* obj, int left, int right) {

}

void numArrayFree(NumArray* obj) {

}

/**
 * Your NumArray struct will be instantiated and called as such:
 * NumArray* obj = numArrayCreate(nums, numsSize);
 * numArrayUpdate(obj, index, val);
 *
 * int param_2 = numArraySumRange(obj, left, right);
 *
 * numArrayFree(obj);
 */

```

## Go Solution:

```
// Problem: Range Sum Query - Mutable
// Difficulty: Medium
// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

type NumArray struct {

}

func Constructor(nums []int) NumArray {

}

func (this *NumArray) Update(index int, val int) {

}

func (this *NumArray) SumRange(left int, right int) int {

}

/**
 * Your NumArray object will be instantiated and called as such:
 * obj := Constructor(nums);
 * obj.Update(index,val);
 * param_2 := obj.SumRange(left,right);
 */

```

## Kotlin Solution:

```
class NumArray(nums: IntArray) {

    fun update(index: Int, `val`: Int) {
```

```

}

fun sumRange(left: Int, right: Int): Int {

}

/** 
* Your NumArray object will be instantiated and called as such:
* var obj = NumArray(nums)
* obj.update(index,`val`)
* var param_2 = obj.sumRange(left,right)
*/

```

### Swift Solution:

```

class NumArray {

    init(_ nums: [Int]) {

    }

    func update(_ index: Int, _ val: Int) {

    }

    func sumRange(_ left: Int, _ right: Int) -> Int {

    }
}

/** 
* Your NumArray object will be instantiated and called as such:
* let obj = NumArray(nums)
* obj.update(index, val)
* let ret_2: Int = obj.sumRange(left, right)
*/

```

## Rust Solution:

```
// Problem: Range Sum Query - Mutable
// Difficulty: Medium
// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

struct NumArray {

}

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

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

    fn update(&self, index: i32, val: i32) {
        }

    fn sum_range(&self, left: i32, right: i32) -> i32 {
        }
    }

    /**
     * Your NumArray object will be instantiated and called as such:
     * let obj = NumArray::new(nums);
     * obj.update(index, val);
     * let ret_2: i32 = obj.sum_range(left, right);
     */
}
```

## Ruby Solution:

```

class NumArray

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

end

=begin
:type index: Integer
:type val: Integer
:rtype: Void
=end
def update(index, val)

end

=begin
:type left: Integer
:type right: Integer
:rtype: Integer
=end
def sum_range(left, right)

end

end

# Your NumArray object will be instantiated and called as such:
# obj = NumArray.new(nums)
# obj.update(index, val)
# param_2 = obj.sum_range(left, right)

```

## PHP Solution:

```

class NumArray {
/**
 * @param Integer[] $nums

```

```

        */
    function __construct($nums) {

    }

    /**
     * @param Integer $index
     * @param Integer $val
     * @return NULL
     */
    function update($index, $val) {

    }

    /**
     * @param Integer $left
     * @param Integer $right
     * @return Integer
     */
    function sumRange($left, $right) {

    }
}

/**
* Your NumArray object will be instantiated and called as such:
* $obj = NumArray($nums);
* $obj->update($index, $val);
* $ret_2 = $obj->sumRange($left, $right);
*/

```

## Dart Solution:

```

class NumArray {

    NumArray(List<int> nums) {

    }

    void update(int index, int val) {

```

```

}

int sumRange(int left, int right) {

}

/** 
* Your NumArray object will be instantiated and called as such:
* NumArray obj = NumArray(nums);
* obj.update(index,val);
* int param2 = obj.sumRange(left,right);
*/

```

### Scala Solution:

```

class NumArray(_nums: Array[Int]) {

def update(index: Int, `val`: Int): Unit = {

}

def sumRange(left: Int, right: Int): Int = {

}

/** 
* Your NumArray object will be instantiated and called as such:
* val obj = new NumArray(nums)
* obj.update(index,`val`)
* val param_2 = obj.sumRange(left,right)
*/

```

### Elixir Solution:

```

defmodule NumArray do
@spec init_(nums :: [integer]) :: any
def init_(nums) do

```

```

end

@spec update(index :: integer, val :: integer) :: any
def update(index, val) do

end

@spec sum_range(left :: integer, right :: integer) :: integer
def sum_range(left, right) do

end
end

# Your functions will be called as such:
# NumArray.init_(nums)
# NumArray.update(index, val)
# param_2 = NumArray.sum_range(left, right)

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

```

### Erlang Solution:

```

-spec num_array_init_(Nums :: [integer()]) -> any().
num_array_init_(Nums) ->
.

-spec num_array_update(Index :: integer(), Val :: integer()) -> any().
num_array_update(Index, Val) ->
.

-spec num_array_sum_range(Left :: integer(), Right :: integer()) ->
integer().
num_array_sum_range(Left, Right) ->
.

%% Your functions will be called as such:
%% num_array_init_(Nums),
%% num_array_update(Index, Val),
%% Param_2 = num_array_sum_range(Left, Right),

```

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

### Racket Solution:

```
(define num-array%
  (class object%
    (super-new)

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

    ; update : exact-integer? exact-integer? -> void?
    (define/public (update index val)
      )
    ; sum-range : exact-integer? exact-integer? -> exact-integer?
    (define/public (sum-range left right)
      )))

;; Your num-array% object will be instantiated and called as such:
;; (define obj (new num-array% [nums nums]))
;; (send obj update index val)
;; (define param_2 (send obj sum-range left right))
```