Problem 1603: Design Parking System

Problem Information

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

Problem Description

Design a parking system for a parking lot. The parking lot has three kinds of parking spaces: big, medium, and small, with a fixed number of slots for each size.

Implement the

ParkingSystem

class:

ParkingSystem(int big, int medium, int small)

Initializes object of the

ParkingSystem

class. The number of slots for each parking space are given as part of the constructor.

bool addCar(int carType)

Checks whether there is a parking space of

carType

for the car that wants to get into the parking lot.

carType

can be of three kinds: big, medium, or small, which are represented by

1

,

2

, and

3

respectively.

A car can only park in a parking space of its

carType

. If there is no space available, return

false

, else park the car in that size space and return

true

.

Example 1:

Input

["ParkingSystem", "addCar", "addCar", "addCar", "addCar"] [[1, 1, 0], [1], [2], [3], [1]]

Output

[null, true, true, false, false]

Explanation

ParkingSystem parkingSystem = new ParkingSystem(1, 1, 0); parkingSystem.addCar(1); // return true because there is 1 available slot for a big car parkingSystem.addCar(2); // return true because there is 1 available slot for a medium car parkingSystem.addCar(3); // return false because there is no available slot for a small car parkingSystem.addCar(1); // return false because there is no available slot for a big car. It is already occupied.

Constraints:

0 <= big, medium, small <= 1000

carType

is

1

,

2

, or

3

At most

1000

calls will be made to

addCar

Code Snippets

C++:

```
class ParkingSystem {
public:
ParkingSystem(int big, int medium, int small) {

}

bool addCar(int carType) {

}
};

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem* obj = new ParkingSystem(big, medium, small);
* bool param_1 = obj->addCar(carType);
*/
```

Java:

```
class ParkingSystem {

public ParkingSystem(int big, int medium, int small) {

}

public boolean addCar(int carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = new ParkingSystem(big, medium, small);
* boolean param_1 = obj.addCar(carType);
*/
```

Python3:

```
class ParkingSystem:

def __init__(self, big: int, medium: int, small: int):
```

```
def addCar(self, carType: int) -> bool:



# Your ParkingSystem object will be instantiated and called as such:
# obj = ParkingSystem(big, medium, small)
# param_1 = obj.addCar(carType)
```

Python:

```
class ParkingSystem(object):

def __init__(self, big, medium, small):
"""
:type big: int
:type medium: int
:type small: int
"""



def addCar(self, carType):
"""
:type carType: int
:rtype: bool
"""




# Your ParkingSystem object will be instantiated and called as such:
# obj = ParkingSystem(big, medium, small)
# param_1 = obj.addCar(carType)
```

JavaScript:

```
/**
* @param {number} big
* @param {number} medium
* @param {number} small
*/
var ParkingSystem = function(big, medium, small) {

};
```

```
/**
* @param {number} carType
* @return {boolean}
*/
ParkingSystem.prototype.addCar = function(carType) {

};

/**
* Your ParkingSystem object will be instantiated and called as such:
* var obj = new ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

TypeScript:

```
class ParkingSystem {
constructor(big: number, medium: number, small: number) {

}

addCar(carType: number): boolean {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* var obj = new ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

C#:

```
public class ParkingSystem {

public ParkingSystem(int big, int medium, int small) {

}

public bool AddCar(int carType) {
```

```
}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = new ParkingSystem(big, medium, small);
* bool param_1 = obj.AddCar(carType);
*/
```

C:

```
typedef struct {

} ParkingSystem;


ParkingSystem* parkingSystemCreate(int big, int medium, int small) {

}

bool parkingSystemAddCar(ParkingSystem* obj, int carType) {

}

void parkingSystemFree(ParkingSystem* obj) {

}

/**
* Your ParkingSystem struct will be instantiated and called as such:
* ParkingSystem* obj = parkingSystemCreate(big, medium, small);
* bool param_1 = parkingSystemAddCar(obj, carType);

* parkingSystemFree(obj);
*/
```

Go:

```
type ParkingSystem struct {

}


func Constructor(big int, medium int, small int) ParkingSystem {

}


func (this *ParkingSystem) AddCar(carType int) bool {

}


/**
* Your ParkingSystem object will be instantiated and called as such:
* obj := Constructor(big, medium, small);
* param_1 := obj.AddCar(carType);
*/
```

Kotlin:

```
class ParkingSystem(big: Int, medium: Int, small: Int) {

fun addCar(carType: Int): Boolean {

}

}

/**
* Your ParkingSystem object will be instantiated and called as such:
* var obj = ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

Swift:

```

class ParkingSystem {

init(_ big: Int, _ medium: Int, _ small: Int) {
```

```
}

func addCar(_ carType: Int) -> Bool {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* let obj = ParkingSystem(big, medium, small)
* let ret_1: Bool = obj.addCar(carType)
*/
```

Rust:

```
struct ParkingSystem {

}


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

fn new(big: i32, medium: i32, small: i32) -> Self {

}

fn add_car(&self, car_type: i32) -> bool {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* let obj = ParkingSystem::new(big, medium, small);
* let ret_1: bool = obj.add_car(carType);
*/
```

Ruby:

```
class ParkingSystem

=begin
:type big: Integer
:type medium: Integer
:type small: Integer
=end
def initialize(big, medium, small)

end



=begin
:type car_type: Integer
:rtype: Boolean
=end
def add_car(car_type)

end



end

# Your ParkingSystem object will be instantiated and called as such:
# obj = ParkingSystem.new(big, medium, small)
# param_1 = obj.add_car(car_type)
```

PHP:

```
class ParkingSystem {
/**
* @param Integer $big
* @param Integer $medium
* @param Integer $small
*/
function __construct($big, $medium, $small) {

}

/**
```

```
* @param Integer $carType
* @return Boolean
*/
function addCar($carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* $obj = ParkingSystem($big, $medium, $small);
* $ret_1 = $obj->addCar($carType);
*/
```

Dart:

```
class ParkingSystem {

ParkingSystem(int big, int medium, int small) {

}

bool addCar(int carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = ParkingSystem(big, medium, small);
* bool param1 = obj.addCar(carType);
*/
```

Scala:

```
class ParkingSystem(_big: Int, _medium: Int, _small: Int) {

def addCar(carType: Int): Boolean = {

}

}
```

```
/**
* Your ParkingSystem object will be instantiated and called as such:
* val obj = new ParkingSystem(big, medium, small)
* val param_1 = obj.addCar(carType)
*/
```

Elixir:

```
defmodule ParkingSystem do
@spec init_(big :: integer, medium :: integer, small :: integer) :: any
def init_(big, medium, small) do

end

@spec add_car(car_type :: integer) :: boolean
def add_car(car_type) do

end
end

# Your functions will be called as such:
# ParkingSystem.init_(big, medium, small)
# param_1 = ParkingSystem.add_car(car_type)

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

Erlang:

```
-spec parking_system_init_(Big :: integer(), Medium :: integer(), Small ::
integer()) -> any().
parking_system_init_(Big, Medium, Small) ->
.

-spec parking_system_add_car(CarType :: integer()) -> boolean().
parking_system_add_car(CarType) ->
.


%% Your functions will be called as such:
%% parking_system_init_(Big, Medium, Small),
```

```
%% Param_1 = parking_system_add_car(CarType),

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

Racket:

```
(define parking-system%
(class object%
(super-new)

; big : exact-integer?
; medium : exact-integer?
; small : exact-integer?
(init-field
big
medium
small)

; add-car : exact-integer? -> boolean?
(define/public (add-car car-type)
)))

;; Your parking-system% object will be instantiated and called as such:
;; (define obj (new parking-system% [big big] [medium medium] [small small]))
;; (define param_1 (send obj add-car car-type))
```

Solutions

C++ Solution:

```
/*
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class ParkingSystem {
public:
ParkingSystem(int big, int medium, int small) {

}

bool addCar(int carType) {

}
};

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem* obj = new ParkingSystem(big, medium, small);
* bool param_1 = obj->addCar(carType);
*/
```

Java Solution:

```
/**
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class ParkingSystem {

public ParkingSystem(int big, int medium, int small) {

}

public boolean addCar(int carType) {

}
}
```

```
/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = new ParkingSystem(big, medium, small);
* boolean param_1 = obj.addCar(carType);
*/
```

Python3 Solution:

```
"""
Problem: Design Parking System
Difficulty: Easy
Tags: general

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

class ParkingSystem:

def __init__(self, big: int, medium: int, small: int):


def addCar(self, carType: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class ParkingSystem(object):

def __init__(self, big, medium, small):
"""
:type big: int
:type medium: int
:type small: int
"""


def addCar(self, carType):
"""
```

```
:type carType: int
:rtype: bool
"""



# Your ParkingSystem object will be instantiated and called as such:
# obj = ParkingSystem(big, medium, small)
# param_1 = obj.addCar(carType)
```

JavaScript Solution:

```
/**
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} big
* @param {number} medium
* @param {number} small
*/
var ParkingSystem = function(big, medium, small) {

};

/**
* @param {number} carType
* @return {boolean}
*/
ParkingSystem.prototype.addCar = function(carType) {

};

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

```
* var obj = new ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

TypeScript Solution:

```
/**
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class ParkingSystem {
constructor(big: number, medium: number, small: number) {

}

addCar(carType: number): boolean {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* var obj = new ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

C# Solution:

```
/*
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
```

```
* Space Complexity: O(1) to O(n) depending on approach
*/

public class ParkingSystem {

public ParkingSystem(int big, int medium, int small) {

}

public bool AddCar(int carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = new ParkingSystem(big, medium, small);
* bool param_1 = obj.AddCar(carType);
*/
```

C Solution:

```
/*
* Problem: Design Parking System
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/




typedef struct {

} ParkingSystem;


ParkingSystem* parkingSystemCreate(int big, int medium, int small) {
```

```
}

bool parkingSystemAddCar(ParkingSystem* obj, int carType) {

}

void parkingSystemFree(ParkingSystem* obj) {

}

/**
* Your ParkingSystem struct will be instantiated and called as such:
* ParkingSystem* obj = parkingSystemCreate(big, medium, small);
* bool param_1 = parkingSystemAddCar(obj, carType);

* parkingSystemFree(obj);
*/
```

Go Solution:

```
// Problem: Design Parking System
// Difficulty: Easy
// Tags: general
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

type ParkingSystem struct {

}


func Constructor(big int, medium int, small int) ParkingSystem {

}


func (this *ParkingSystem) AddCar(carType int) bool {
```

```
}


/**
* Your ParkingSystem object will be instantiated and called as such:
* obj := Constructor(big, medium, small);
* param_1 := obj.AddCar(carType);
*/
```

Kotlin Solution:

```
class ParkingSystem(big: Int, medium: Int, small: Int) {

fun addCar(carType: Int): Boolean {

}

}

/**
* Your ParkingSystem object will be instantiated and called as such:
* var obj = ParkingSystem(big, medium, small)
* var param_1 = obj.addCar(carType)
*/
```

Swift Solution:

```
class ParkingSystem {

init(_ big: Int, _ medium: Int, _ small: Int) {

}

func addCar(_ carType: Int) -> Bool {

}
}

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

```
* let obj = ParkingSystem(big, medium, small)
* let ret_1: Bool = obj.addCar(carType)
*/
```

Rust Solution:

```
// Problem: Design Parking System
// Difficulty: Easy
// Tags: general
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

struct ParkingSystem {

}


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

fn new(big: i32, medium: i32, small: i32) -> Self {

}

fn add_car(&self, car_type: i32) -> bool {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* let obj = ParkingSystem::new(big, medium, small);
* let ret_1: bool = obj.add_car(carType);
*/
```

Ruby Solution:

```
class ParkingSystem

=begin
:type big: Integer
:type medium: Integer
:type small: Integer
=end
def initialize(big, medium, small)

end


=begin
:type car_type: Integer
:rtype: Boolean
=end
def add_car(car_type)

end



end

# Your ParkingSystem object will be instantiated and called as such:
# obj = ParkingSystem.new(big, medium, small)
# param_1 = obj.add_car(car_type)
```

PHP Solution:

```
class ParkingSystem {
/**
* @param Integer $big
* @param Integer $medium
* @param Integer $small
*/
function __construct($big, $medium, $small) {

}

/**
* @param Integer $carType
* @return Boolean
```

```
*/
function addCar($carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* $obj = ParkingSystem($big, $medium, $small);
* $ret_1 = $obj->addCar($carType);
*/
```

Dart Solution:

```
class ParkingSystem {

ParkingSystem(int big, int medium, int small) {

}

bool addCar(int carType) {

}
}

/**
* Your ParkingSystem object will be instantiated and called as such:
* ParkingSystem obj = ParkingSystem(big, medium, small);
* bool param1 = obj.addCar(carType);
*/
```

Scala Solution:

```
class ParkingSystem(_big: Int, _medium: Int, _small: Int) {

def addCar(carType: Int): Boolean = {

}

}
```

```
/**
* Your ParkingSystem object will be instantiated and called as such:
* val obj = new ParkingSystem(big, medium, small)
* val param_1 = obj.addCar(carType)
*/
```

Elixir Solution:

```
defmodule ParkingSystem do
@spec init_(big :: integer, medium :: integer, small :: integer) :: any
def init_(big, medium, small) do

end

@spec add_car(car_type :: integer) :: boolean
def add_car(car_type) do

end
end

# Your functions will be called as such:
# ParkingSystem.init_(big, medium, small)
# param_1 = ParkingSystem.add_car(car_type)

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

Erlang Solution:

```
-spec parking_system_init_(Big :: integer(), Medium :: integer(), Small ::
integer()) -> any().
parking_system_init_(Big, Medium, Small) ->
.

-spec parking_system_add_car(CarType :: integer()) -> boolean().
parking_system_add_car(CarType) ->
.


%% Your functions will be called as such:
%% parking_system_init_(Big, Medium, Small),
```

```
%% Param_1 = parking_system_add_car(CarType),

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

Racket Solution:

```
(define parking-system%
(class object%
(super-new)

; big : exact-integer?
; medium : exact-integer?
; small : exact-integer?
(init-field
big
medium
small)

; add-car : exact-integer? -> boolean?
(define/public (add-car car-type)
)))

;; Your parking-system% object will be instantiated and called as such:
;; (define obj (new parking-system% [big big] [medium medium] [small small]))
;; (define param_1 (send obj add-car car-type))
```