Problem 2241: Design an ATM Machine

Problem Information

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

Problem Description

There is an ATM machine that stores banknotes of

5

denominations:

20

,

50

,

100

,

200

, and

500

dollars. Initially the ATM is empty. The user can use the machine to deposit or withdraw any amount of money.

When withdrawing, the machine prioritizes using banknotes of

larger

values.

For example, if you want to withdraw

$300

and there are

2

$50

banknotes,

1

$100

banknote, and

1

$200

banknote, then the machine will use the

$100

and

$200

banknotes.

However, if you try to withdraw

$600

and there are

3

$200

banknotes and

1

$500

banknote, then the withdraw request will be rejected because the machine will first try to use the

$500

banknote and then be unable to use banknotes to complete the remaining

$100

. Note that the machine is

not

allowed to use the

$200

banknotes instead of the

$500

banknote.

Implement the ATM class:

ATM()

Initializes the ATM object.

void deposit(int[] banknotesCount)

Deposits new banknotes in the order

$20

,

$50

,

$100

,

$200

, and

$500

.

int[] withdraw(int amount)

Returns an array of length

5

of the number of banknotes that will be handed to the user in the order

$20

,

$50

,

$100

,

$200

, and

$500

, and update the number of banknotes in the ATM after withdrawing. Returns

[-1]

if it is not possible (do

not

withdraw any banknotes in this case).

Example 1:

Input

["ATM", "deposit", "withdraw", "deposit", "withdraw", "withdraw"] [[], [[0,0,1,2,1]], [600], [[0,1,0,1,1]], [600], [550]]

Output

[null, null, [0,0,1,0,1], null, [-1], [0,1,0,0,1]]

Explanation

ATM atm = new ATM(); atm.deposit([0,0,1,2,1]); // Deposits 1 $100 banknote, 2 $200 banknotes, // and 1 $500 banknote. atm.withdraw(600); // Returns [0,0,1,0,1]. The machine uses 1 $100 banknote // and 1 $500 banknote. The banknotes left over in the // machine are [0,0,0,2,0]. atm.deposit([0,1,0,1,1]); // Deposits 1 $50, $200, and $500 banknote. // The banknotes in the machine are now [0,1,0,3,1]. atm.withdraw(600); // Returns [-1]. The machine will try to use a $500 banknote // and then be unable to complete the remaining $100, // so the withdraw request will be rejected. // Since the request is rejected, the number of banknotes // in the machine is not modified. atm.withdraw(550); // Returns [0,1,0,0,1]. The machine uses 1 $50 banknote // and 1 $500 banknote.

Constraints:

banknotesCount.length == 5

0 <= banknotesCount[i] <= 10

9

1 <= amount <= 10

9

At most

5000

calls

in total

will be made to

withdraw

and

deposit

.

At least

one

call will be made to each function

withdraw

and

deposit

.

Sum of

banknotesCount[i]

in all deposits doesn't exceed

10

9

Code Snippets

C++:

```
class ATM {
public:
ATM() {

}

void deposit(vector<int> banknotesCount) {

}
vector<int> withdraw(int amount) {

}
};

/**
* Your ATM object will be instantiated and called as such:
* ATM* obj = new ATM();
* obj->deposit(banknotesCount);
* vector<int> param_2 = obj->withdraw(amount);
*/
```

Java:

```
class ATM {

public ATM() {

}

public void deposit(int[] banknotesCount) {

}

public int[] withdraw(int amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* ATM obj = new ATM();
* obj.deposit(banknotesCount);
* int[] param_2 = obj.withdraw(amount);
*/
```

Python3:

```
class ATM:

def __init__(self):
```

```
def deposit(self, banknotesCount: List[int]) -> None:


def withdraw(self, amount: int) -> List[int]:



# Your ATM object will be instantiated and called as such:
# obj = ATM()
# obj.deposit(banknotesCount)
# param_2 = obj.withdraw(amount)
```

Python:

```
class ATM(object):

def __init__(self):


def deposit(self, banknotesCount):
"""
:type banknotesCount: List[int]
:rtype: None
"""


def withdraw(self, amount):
"""
:type amount: int
:rtype: List[int]
"""



# Your ATM object will be instantiated and called as such:
# obj = ATM()
# obj.deposit(banknotesCount)
# param_2 = obj.withdraw(amount)
```

JavaScript:

```
var ATM = function() {

};

/**
* @param {number[]} banknotesCount
* @return {void}
*/
ATM.prototype.deposit = function(banknotesCount) {

};

/**
* @param {number} amount
* @return {number[]}
*/
ATM.prototype.withdraw = function(amount) {

};

/**
* Your ATM object will be instantiated and called as such:
* var obj = new ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

TypeScript:

```
class ATM {
constructor() {

}

deposit(banknotesCount: number[]): void {

}

withdraw(amount: number): number[] {

}
}
```

```
/**
* Your ATM object will be instantiated and called as such:
* var obj = new ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

C#:

```
public class ATM {

public ATM() {

}

public void Deposit(int[] banknotesCount) {

}

public int[] Withdraw(int amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* ATM obj = new ATM();
* obj.Deposit(banknotesCount);
* int[] param_2 = obj.Withdraw(amount);
*/
```

C:

```
typedef struct {

} ATM;
```

```
ATM* aTMCreate() {

}

void aTMDeposit(ATM* obj, int* banknotesCount, int banknotesCountSize) {

}

int* aTMWithdraw(ATM* obj, int amount, int* retSize) {

}

void aTMFree(ATM* obj) {

}

/**
* Your ATM struct will be instantiated and called as such:
* ATM* obj = aTMCreate();
* aTMDeposit(obj, banknotesCount, banknotesCountSize);

* int* param_2 = aTMWithdraw(obj, amount, retSize);

* aTMFree(obj);
*/
```

Go:

```
type ATM struct {

}


func Constructor() ATM {

}


func (this *ATM) Deposit(banknotesCount []int) {

}
```

```


func (this *ATM) Withdraw(amount int) []int {

}



/**
* Your ATM object will be instantiated and called as such:
* obj := Constructor();
* obj.Deposit(banknotesCount);
* param_2 := obj.Withdraw(amount);
*/
```

Kotlin:

```
class ATM() {

fun deposit(banknotesCount: IntArray) {

}

fun withdraw(amount: Int): IntArray {

}

}

/**
* Your ATM object will be instantiated and called as such:
* var obj = ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

Swift:

```


class ATM {

init() {

}
func deposit(_ banknotesCount: [Int]) {

}

func withdraw(_ amount: Int) -> [Int] {

}
}

/**
* Your ATM object will be instantiated and called as such:
* let obj = ATM()
* obj.deposit(banknotesCount)
* let ret_2: [Int] = obj.withdraw(amount)
*/
```

Rust:

```
struct ATM {

}



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

fn new() -> Self {

}

fn deposit(&self, banknotes_count: Vec<i32>) {

}

fn withdraw(&self, amount: i32) -> Vec<i32> {

}
}
```

```
/**
* Your ATM object will be instantiated and called as such:
* let obj = ATM::new();
* obj.deposit(banknotesCount);
* let ret_2: Vec<i32> = obj.withdraw(amount);
*/
```

Ruby:

```
class ATM
def initialize()

end



=begin
:type banknotes_count: Integer[]
:rtype: Void
=end
def deposit(banknotes_count)

end



=begin
:type amount: Integer
:rtype: Integer[]
=end
def withdraw(amount)

end



end

# Your ATM object will be instantiated and called as such:
# obj = ATM.new()
# obj.deposit(banknotes_count)
# param_2 = obj.withdraw(amount)
```

PHP:

```
class ATM {
/**
*/
function __construct() {

}

/**
* @param Integer[] $banknotesCount
* @return NULL
*/
function deposit($banknotesCount) {

}

/**
* @param Integer $amount
* @return Integer[]
*/
function withdraw($amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* $obj = ATM();
* $obj->deposit($banknotesCount);
* $ret_2 = $obj->withdraw($amount);
*/
```

Dart:

```
class ATM {

ATM() {

}

void deposit(List<int> banknotesCount) {

}

List<int> withdraw(int amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* ATM obj = ATM();
* obj.deposit(banknotesCount);
* List<int> param2 = obj.withdraw(amount);
*/
```

Scala:

```
class ATM() {

def deposit(banknotesCount: Array[Int]): Unit = {

}

def withdraw(amount: Int): Array[Int] = {

}

}

/**
* Your ATM object will be instantiated and called as such:
* val obj = new ATM()
* obj.deposit(banknotesCount)
* val param_2 = obj.withdraw(amount)
*/
```

Elixir:

```
defmodule ATM do
@spec init_() :: any
def init_() do

end
```

```
@spec deposit(banknotes_count :: [integer]) :: any
def deposit(banknotes_count) do

end

@spec withdraw(amount :: integer) :: [integer]
def withdraw(amount) do

end
end

# Your functions will be called as such:
# ATM.init_()
# ATM.deposit(banknotes_count)
# param_2 = ATM.withdraw(amount)

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

Erlang:

```
-spec atm_init_() -> any().
atm_init_() ->
.

-spec atm_deposit(BanknotesCount :: [integer()]) -> any().
atm_deposit(BanknotesCount) ->
.

-spec atm_withdraw(Amount :: integer()) -> [integer()].
atm_withdraw(Amount) ->
.



%% Your functions will be called as such:
%% atm_init_(),
%% atm_deposit(BanknotesCount),
%% Param_2 = atm_withdraw(Amount),

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

Racket:

```
(define atm%
(class object%
(super-new)

(init-field)

; deposit : (listof exact-integer?) -> void?
(define/public (deposit banknotes-count)
)
; withdraw : exact-integer? -> (listof exact-integer?)
(define/public (withdraw amount)
)))

;; Your atm% object will be instantiated and called as such:
;; (define obj (new atm%))
;; (send obj deposit banknotes-count)
;; (define param_2 (send obj withdraw amount))
```

Solutions

C++ Solution:

```
/*
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class ATM {
public:
ATM() {

}

void deposit(vector<int> banknotesCount) {
}

vector<int> withdraw(int amount) {

}
};

/**
* Your ATM object will be instantiated and called as such:
* ATM* obj = new ATM();
* obj->deposit(banknotesCount);
* vector<int> param_2 = obj->withdraw(amount);
*/
```

Java Solution:

```
/**
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class ATM {

public ATM() {

}

public void deposit(int[] banknotesCount) {

}

public int[] withdraw(int amount) {

}
}
```

```
/**
* Your ATM object will be instantiated and called as such:
* ATM obj = new ATM();
* obj.deposit(banknotesCount);
* int[] param_2 = obj.withdraw(amount);
*/
```

Python3 Solution:

```
"""
Problem: Design an ATM Machine
Difficulty: Medium
Tags: array, greedy

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class ATM:

def __init__(self):


def deposit(self, banknotesCount: List[int]) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class ATM(object):

def __init__(self):


def deposit(self, banknotesCount):
"""
:type banknotesCount: List[int]
:rtype: None
"""
```

```
def withdraw(self, amount):
"""
:type amount: int
:rtype: List[int]
"""



# Your ATM object will be instantiated and called as such:
# obj = ATM()
# obj.deposit(banknotesCount)
# param_2 = obj.withdraw(amount)
```

JavaScript Solution:

```
/**
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/



var ATM = function() {

};

/**
* @param {number[]} banknotesCount
* @return {void}
*/
ATM.prototype.deposit = function(banknotesCount) {

};

/**
```

```
* @param {number} amount
* @return {number[]}
*/
ATM.prototype.withdraw = function(amount) {

};

/**
* Your ATM object will be instantiated and called as such:
* var obj = new ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

TypeScript Solution:

```
/**
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class ATM {
constructor() {

}

deposit(banknotesCount: number[]): void {

}

withdraw(amount: number): number[] {

}
}

/**
```

```
* Your ATM object will be instantiated and called as such:
* var obj = new ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

C# Solution:

```
/*
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class ATM {

public ATM() {

}

public void Deposit(int[] banknotesCount) {

}

public int[] Withdraw(int amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* ATM obj = new ATM();
* obj.Deposit(banknotesCount);
* int[] param_2 = obj.Withdraw(amount);
*/
```

C Solution:

```c
/*
* Problem: Design an ATM Machine
* Difficulty: Medium
* Tags: array, greedy
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/




typedef struct {

} ATM;


ATM* aTMCreate() {

}

void aTMDeposit(ATM* obj, int* banknotesCount, int banknotesCountSize) {

}

int* aTMWithdraw(ATM* obj, int amount, int* retSize) {

}

void aTMFree(ATM* obj) {

}

/**
* Your ATM struct will be instantiated and called as such:
* ATM* obj = aTMCreate();
* aTMDeposit(obj, banknotesCount, banknotesCountSize);

* int* param_2 = aTMWithdraw(obj, amount, retSize);

* aTMFree(obj);
*/
```

Go Solution:

```
// Problem: Design an ATM Machine
// Difficulty: Medium
// Tags: array, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type ATM struct {

}


func Constructor() ATM {

}


func (this *ATM) Deposit(banknotesCount []int) {

}


func (this *ATM) Withdraw(amount int) []int {

}


/**
* Your ATM object will be instantiated and called as such:
* obj := Constructor();
* obj.Deposit(banknotesCount);
* param_2 := obj.Withdraw(amount);
*/
```

Kotlin Solution:

```
class ATM() {

fun deposit(banknotesCount: IntArray) {


}

fun withdraw(amount: Int): IntArray {

}

}

/**
* Your ATM object will be instantiated and called as such:
* var obj = ATM()
* obj.deposit(banknotesCount)
* var param_2 = obj.withdraw(amount)
*/
```

Swift Solution:

```

class ATM {

init() {

}

func deposit(_ banknotesCount: [Int]) {

}

func withdraw(_ amount: Int) -> [Int] {

}
}

/**
* Your ATM object will be instantiated and called as such:
* let obj = ATM()
* obj.deposit(banknotesCount)
* let ret_2: [Int] = obj.withdraw(amount)
*/
```

Rust Solution:

```
// Problem: Design an ATM Machine
// Difficulty: Medium
// Tags: array, greedy
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct ATM {

}



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

fn new() -> Self {

}

fn deposit(&self, banknotes_count: Vec<i32>) {

}

fn withdraw(&self, amount: i32) -> Vec<i32> {

}
}

/**
* Your ATM object will be instantiated and called as such:
* let obj = ATM::new();
* obj.deposit(banknotesCount);
* let ret_2: Vec<i32> = obj.withdraw(amount);
*/
```

Ruby Solution:

```
class ATM
def initialize()

end



=begin
:type banknotes_count: Integer[]
:rtype: Void
=end
def deposit(banknotes_count)

end



=begin
:type amount: Integer
:rtype: Integer[]
=end
def withdraw(amount)

end



end

# Your ATM object will be instantiated and called as such:
# obj = ATM.new()
# obj.deposit(banknotes_count)
# param_2 = obj.withdraw(amount)
```

PHP Solution:

```
class ATM {
/**
*/
function __construct() {

}

/**
* @param Integer[] $banknotesCount
* @return NULL
*/
function deposit($banknotesCount) {

}

/**
* @param Integer $amount
* @return Integer[]
*/
function withdraw($amount) {

}
}

/**
* Your ATM object will be instantiated and called as such:
* $obj = ATM();
* $obj->deposit($banknotesCount);
* $ret_2 = $obj->withdraw($amount);
*/
```

Dart Solution:

```
class ATM {

ATM() {

}

void deposit(List<int> banknotesCount) {

}

List<int> withdraw(int amount) {

}
}

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

```
* ATM obj = ATM();
* obj.deposit(banknotesCount);
* List<int> param2 = obj.withdraw(amount);
*/
```

Scala Solution:

```
class ATM() {

def deposit(banknotesCount: Array[Int]): Unit = {

}

def withdraw(amount: Int): Array[Int] = {

}

}

/**
* Your ATM object will be instantiated and called as such:
* val obj = new ATM()
* obj.deposit(banknotesCount)
* val param_2 = obj.withdraw(amount)
*/
```

Elixir Solution:

```
defmodule ATM do
@spec init_() :: any
def init_() do

end

@spec deposit(banknotes_count :: [integer]) :: any
def deposit(banknotes_count) do

end

@spec withdraw(amount :: integer) :: [integer]
def withdraw(amount) do
```

```
end
end

# Your functions will be called as such:
# ATM.init_()
# ATM.deposit(banknotes_count)
# param_2 = ATM.withdraw(amount)

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

Erlang Solution:

```
-spec atm_init_() -> any().
atm_init_() ->
.

-spec atm_deposit(BanknotesCount :: [integer()]) -> any().
atm_deposit(BanknotesCount) ->
.

-spec atm_withdraw(Amount :: integer()) -> [integer()].
atm_withdraw(Amount) ->
.


%% Your functions will be called as such:
%% atm_init_(),
%% atm_deposit(BanknotesCount),
%% Param_2 = atm_withdraw(Amount),

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

Racket Solution:

```
(define atm%
(class object%
(super-new)
(init-field)

; deposit : (listof exact-integer?) -> void?
(define/public (deposit banknotes-count)
)
; withdraw : exact-integer? -> (listof exact-integer?)
(define/public (withdraw amount)
)))

;; Your atm% object will be instantiated and called as such:
;; (define obj (new atm%))
;; (send obj deposit banknotes-count)
;; (define param_2 (send obj withdraw amount))
```