Problem 3387: Maximize Amount After Two Days of Conversions

Problem Information

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

Problem Description

You are given a string

initialCurrency

, and you start with

1.0

of

initialCurrency

.

You are also given four arrays with currency pairs (strings) and rates (real numbers):

pairs1[i] = [startCurrency

i

, targetCurrency

i

]

denotes that you can convert from

startCurrency

i

to

targetCurrency

i

at a rate of

rates1[i]

on

day 1

.

pairs2[i] = [startCurrency

i

, targetCurrency

i

]

denotes that you can convert from

startCurrency

i

to

targetCurrency

i

at a rate of

rates2[i]

on

day 2

.

Also, each

targetCurrency

can be converted back to its corresponding

startCurrency

at a rate of

1 / rate

.

You can perform

any

number of conversions,

including zero

, using

rates1

on day 1,

followed

by any number of additional conversions,

including zero

, using

rates2

on day 2.

Return the

maximum

amount of

initialCurrency

you can have after performing any number of conversions on both days

in order

.

Note:

Conversion rates are valid, and there will be no contradictions in the rates for either day. The rates for the days are independent of each other.

Example 1:

Input:

initialCurrency = "EUR", pairs1 = [["EUR","USD"],["USD","JPY"]], rates1 = [2.0,3.0], pairs2 = [["JPY","USD"],["USD","CHF"],["CHF","EUR"]], rates2 = [4.0,5.0,6.0]

Output:

720.00000

Explanation:

To get the maximum amount of

EUR

, starting with 1.0

EUR

:

On Day 1:

Convert

EUR

to

USD

to get 2.0

USD

.

Convert

USD

to

JPY

to get 6.0

JPY

.

On Day 2:

Convert

JPY

to

USD

to get 24.0

USD

.

Convert

USD

to

CHF

to get 120.0

CHF

.

Finally, convert

CHF

to

EUR

to get 720.0

EUR

.

Example 2:

Input:

initialCurrency = "NGN", pairs1 =

[["NGN","EUR"]]

, rates1 =

[9.0]

, pairs2 =

[["NGN","EUR"]]

, rates2 =

[6.0]

Output:

1.50000

Explanation:

Converting

NGN

to

EUR

on day 1 and

EUR

to

NGN

using the inverse rate on day 2 gives the maximum amount.

Example 3:

Input:

initialCurrency = "USD", pairs1 = [["USD","EUR"]], rates1 = [1.0], pairs2 = [["EUR","JPY"]], rates2 = [10.0]

Output:

1.00000

Explanation:

In this example, there is no need to make any conversions on either day.

Constraints:

1 <= initialCurrency.length <= 3

initialCurrency

consists only of uppercase English letters.

1 <= n == pairs1.length <= 10

1 <= m == pairs2.length <= 10

pairs1[i] == [startCurrency

i

, targetCurrency

i

]

pairs2[i] == [startCurrency

i

, targetCurrency

i

]

1 <= startCurrency

i

.length, targetCurrency

i

.length <= 3

startCurrency

i

and

targetCurrency

i

consist only of uppercase English letters.

rates1.length == n

rates2.length == m

1.0 <= rates1[i], rates2[i] <= 10.0

The input is generated such that there are no contradictions or cycles in the conversion graphs for either day.

The input is generated such that the output is

at most

5 * 10

10

.

Code Snippets

C++:

```
class Solution {
public:
double maxAmount(string initialCurrency, vector<vector<string>>& pairs1,
```

```
vector<double>& rates1, vector<vector<string>>& pairs2, vector<double>&
rates2) {

}
};
```

Java:

```
class Solution {
public double maxAmount(String initialCurrency, List<List<String>> pairs1,
double[] rates1, List<List<String>> pairs2, double[] rates2) {

}
}
```

Python3:

```
class Solution:
def maxAmount(self, initialCurrency: str, pairs1: List[List[str]], rates1:
List[float], pairs2: List[List[str]], rates2: List[float]) -> float:
```

Python:

```
class Solution(object):
def maxAmount(self, initialCurrency, pairs1, rates1, pairs2, rates2):
"""
:type initialCurrency: str
:type pairs1: List[List[str]]
:type rates1: List[float]
:type pairs2: List[List[str]]
:type rates2: List[float]
:rtype: float
"""
```

JavaScript:

```
/**
* @param {string} initialCurrency
* @param {string[][]} pairs1
* @param {number[]} rates1
* @param {string[][]} pairs2
* @param {number[]} rates2
* @return {number}
```

```
*/
var maxAmount = function(initialCurrency, pairs1, rates1, pairs2, rates2) {

};
```

TypeScript:

```
function maxAmount(initialCurrency: string, pairs1: string[][], rates1:
number[], pairs2: string[][], rates2: number[]): number {

};
```

C#:

```
public class Solution {
public double MaxAmount(string initialCurrency, IList<IList<string>> pairs1,
double[] rates1, IList<IList<string>> pairs2, double[] rates2) {

}
}
```

C:

```
double maxAmount(char* initialCurrency, char*** pairs1, int pairs1Size, int*
pairs1ColSize, double* rates1, int rates1Size, char*** pairs2, int
pairs2Size, int* pairs2ColSize, double* rates2, int rates2Size) {

}
```

Go:

```
func maxAmount(initialCurrency string, pairs1 [][]string, rates1 []float64,
pairs2 [][]string, rates2 []float64) float64 {

}
```

Kotlin:

```
class Solution {
fun maxAmount(initialCurrency: String, pairs1: List<List<String>>, rates1:
DoubleArray, pairs2: List<List<String>>, rates2: DoubleArray): Double {
```

```
}
}
```

Swift:

```
class Solution {
func maxAmount(_ initialCurrency: String, _ pairs1: [[String]], _ rates1:
[Double], _ pairs2: [[String]], _ rates2: [Double]) -> Double {

}
}
```

Rust:

```
impl Solution {
pub fn max_amount(initial_currency: String, pairs1: Vec<Vec<String>>, rates1:
Vec<f64>, pairs2: Vec<Vec<String>>, rates2: Vec<f64>) -> f64 {

}
}
```

Ruby:

```
# @param {String} initial_currency
# @param {String[][]} pairs1
# @param {Float[]} rates1
# @param {String[][]} pairs2
# @param {Float[]} rates2
# @return {Float}
def max_amount(initial_currency, pairs1, rates1, pairs2, rates2)

end
```

PHP:

```
class Solution {

/**
* @param String $initialCurrency
* @param String[][] $pairs1
* @param Float[] $rates1
* @param String[][] $pairs2
```

```
* @param Float[] $rates2
* @return Float
*/
function maxAmount($initialCurrency, $pairs1, $rates1, $pairs2, $rates2) {

}
}
```

Dart:

```
class Solution {
double maxAmount(String initialCurrency, List<List<String>> pairs1,
List<double> rates1, List<List<String>> pairs2, List<double> rates2) {

}
}
```

Scala:

```
object Solution {
def maxAmount(initialCurrency: String, pairs1: List[List[String]], rates1:
Array[Double], pairs2: List[List[String]], rates2: Array[Double]): Double = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_amount(initial_currency :: String.t, pairs1 :: [[String.t]], rates1
:: [float], pairs2 :: [[String.t]], rates2 :: [float]) :: float
def max_amount(initial_currency, pairs1, rates1, pairs2, rates2) do

end
end
```

Erlang:

```
-spec max_amount(InitialCurrency :: unicode:unicode_binary(), Pairs1 ::
[[unicode:unicode_binary()]], Rates1 :: [float()], Pairs2 ::
[[unicode:unicode_binary()]], Rates2 :: [float()]) -> float().
max_amount(InitialCurrency, Pairs1, Rates1, Pairs2, Rates2) ->
```

```
.
```

Racket:

```
(define/contract (max-amount initialCurrency pairs1 rates1 pairs2 rates2)
(-> string? (listof (listof string?)) (listof flonum?) (listof (listof
string?)) (listof flonum?) flonum?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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 Solution {
public:
double maxAmount(string initialCurrency, vector<vector<string>>& pairs1,
vector<double>& rates1, vector<vector<string>>& pairs2, vector<double>&
rates2) {

}
};
```

Java Solution:

```
/**
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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 Solution {
public double maxAmount(String initialCurrency, List<List<String>> pairs1,
double[] rates1, List<List<String>> pairs2, double[] rates2) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximize Amount After Two Days of Conversions
Difficulty: Medium
Tags: array, string, graph, search

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 Solution:
def maxAmount(self, initialCurrency: str, pairs1: List[List[str]], rates1:
List[float], pairs2: List[List[str]], rates2: List[float]) -> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxAmount(self, initialCurrency, pairs1, rates1, pairs2, rates2):
"""
:type initialCurrency: str
:type pairs1: List[List[str]]
:type rates1: List[float]
:type pairs2: List[List[str]]
:type rates2: List[float]
:rtype: float
"""
```

JavaScript Solution:

```
/**
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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
*/

/**
* @param {string} initialCurrency
* @param {string[][]} pairs1
* @param {number[]} rates1
* @param {string[][]} pairs2
* @param {number[]} rates2
* @return {number}
*/
var maxAmount = function(initialCurrency, pairs1, rates1, pairs2, rates2) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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
*/

function maxAmount(initialCurrency: string, pairs1: string[][], rates1:
number[], pairs2: string[][], rates2: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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 Solution {
public double MaxAmount(string initialCurrency, IList<IList<string>> pairs1,
double[] rates1, IList<IList<string>> pairs2, double[] rates2) {

}
}
```

C Solution:

```
/*
* Problem: Maximize Amount After Two Days of Conversions
* Difficulty: Medium
* Tags: array, string, graph, search
*
* 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
*/

double maxAmount(char* initialCurrency, char*** pairs1, int pairs1Size, int*
pairs1ColSize, double* rates1, int rates1Size, char*** pairs2, int
pairs2Size, int* pairs2ColSize, double* rates2, int rates2Size) {

}
```

Go Solution:

```
// Problem: Maximize Amount After Two Days of Conversions
// Difficulty: Medium
// Tags: array, string, graph, search
//
// 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

func maxAmount(initialCurrency string, pairs1 [][]string, rates1 []float64,
pairs2 [][]string, rates2 []float64) float64 {

}
```

Kotlin Solution:

```
class Solution {
fun maxAmount(initialCurrency: String, pairs1: List<List<String>>, rates1:
DoubleArray, pairs2: List<List<String>>, rates2: DoubleArray): Double {

}
}
```

Swift Solution:

```
class Solution {
func maxAmount(_ initialCurrency: String, _ pairs1: [[String]], _ rates1:
[Double], _ pairs2: [[String]], _ rates2: [Double]) -> Double {

}
}
```

Rust Solution:

```
// Problem: Maximize Amount After Two Days of Conversions
// Difficulty: Medium
// Tags: array, string, graph, search
//
// 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

impl Solution {
pub fn max_amount(initial_currency: String, pairs1: Vec<Vec<String>>, rates1:
Vec<f64>, pairs2: Vec<Vec<String>>, rates2: Vec<f64>) -> f64 {

}
```

```
}
```

Ruby Solution:

```
# @param {String} initial_currency
# @param {String[][]} pairs1
# @param {Float[]} rates1
# @param {String[][]} pairs2
# @param {Float[]} rates2
# @return {Float}
def max_amount(initial_currency, pairs1, rates1, pairs2, rates2)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $initialCurrency
* @param String[][] $pairs1
* @param Float[] $rates1
* @param String[][] $pairs2
* @param Float[] $rates2
* @return Float
*/
function maxAmount($initialCurrency, $pairs1, $rates1, $pairs2, $rates2) {

}
}
```

Dart Solution:

```
class Solution {
double maxAmount(String initialCurrency, List<List<String>> pairs1,
List<double> rates1, List<List<String>> pairs2, List<double> rates2) {

}
}
```

Scala Solution:

```
object Solution {
def maxAmount(initialCurrency: String, pairs1: List[List[String]], rates1:
Array[Double], pairs2: List[List[String]], rates2: Array[Double]): Double = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_amount(initial_currency :: String.t, pairs1 :: [[String.t]], rates1
:: [float], pairs2 :: [[String.t]], rates2 :: [float]) :: float
def max_amount(initial_currency, pairs1, rates1, pairs2, rates2) do

end
end
```

Erlang Solution:

```
-spec max_amount(InitialCurrency :: unicode:unicode_binary(), Pairs1 ::
[[unicode:unicode_binary()]], Rates1 :: [float()], Pairs2 ::
[[unicode:unicode_binary()]], Rates2 :: [float()]) -> float().
max_amount(InitialCurrency, Pairs1, Rates1, Pairs2, Rates2) ->
.
```

Racket Solution:

```
(define/contract (max-amount initialCurrency pairs1 rates1 pairs2 rates2)
(-> string? (listof (listof string?)) (listof flonum?) (listof (listof
string?)) (listof flonum?) flonum?)
)
```