Problem 2473: Minimum Cost to Buy Apples

Problem Information

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

Problem Description

You are given a positive integer

n

representing

n

cities numbered from

1

to

n

. You are also given a

2D

array

roads

, where

roads[i] = [a

i

, b

i

, cost

i

]

indicates that there is a

bidirectional

road between cities

a

i

and

b

i

with a cost of traveling equal to

cost

i

.

You can buy apples in

any

city you want, but some cities have different costs to buy apples. You are given the 1-based array

appleCost

where

appleCost[i]

is the cost of buying one apple from city

i

.

You start at some city, traverse through various roads, and eventually buy

exactly

one apple from

any

city. After you buy that apple, you have to return back to the city you

started

at, but now the cost of all the roads will be

multiplied

by a given factor

k

.

Given the integer

k

, return

a 1-based array

answer

of size

n

where

answer[i]

is the

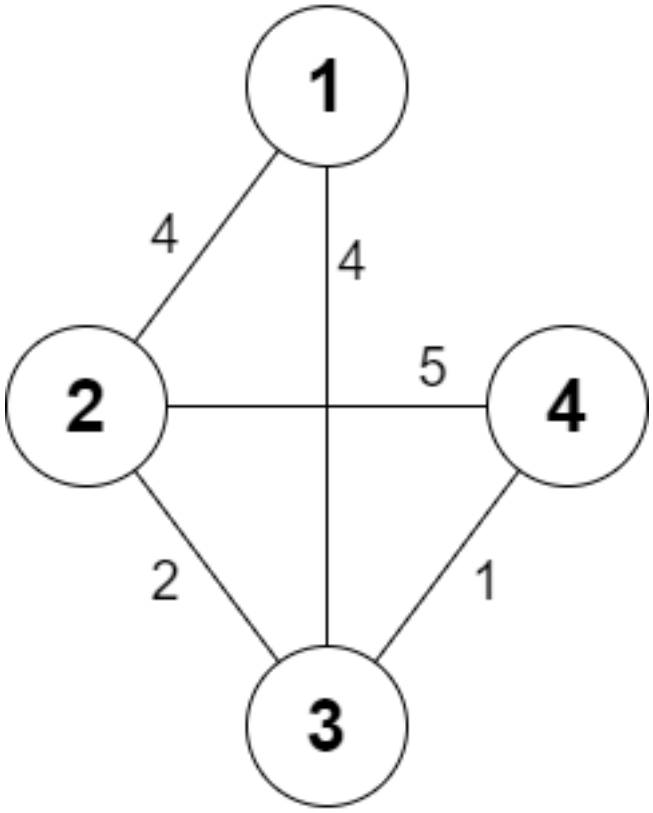
minimum

total cost to buy an apple if you start at city

i

.

Example 1:



Input:

n = 4, roads = [[1,2,4],[2,3,2],[2,4,5],[3,4,1],[1,3,4]], appleCost = [56,42,102,301], k = 2

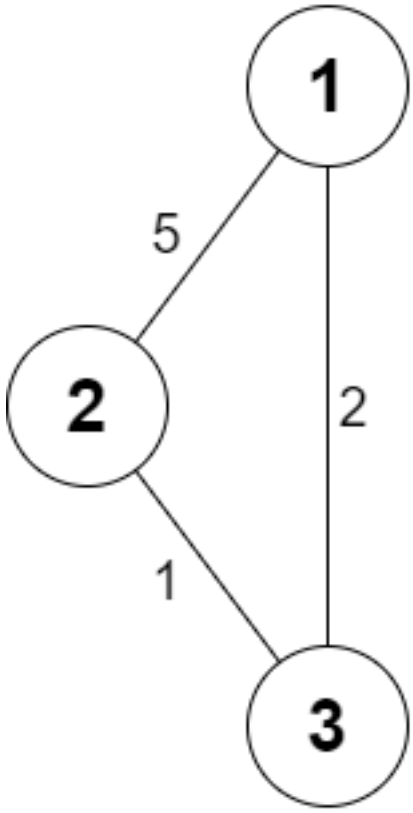
Output:

[54,42,48,51]

Explanation:

The minimum cost for each starting city is the following: - Starting at city 1: You take the path 1 -> 2, buy an apple at city 2, and finally take the path 2 -> 1. The total cost is 4 + 42 + 4 * 2 = 54. - Starting at city 2: You directly buy an apple at city 2. The total cost is 42. - Starting at city 3: You take the path 3 -> 2, buy an apple at city 2, and finally take the path 2 -> 3. The total cost is 2 + 42 + 2 * 2 = 48. - Starting at city 4: You take the path 4 -> 3 -> 2 then you buy at city 2, and finally take the path 2 -> 3 -> 4. The total cost is 1 + 2 + 42 + 1 * 2 + 2 * 2 = 51.

Example 2:



Input:

n = 3, roads = [[1,2,5],[2,3,1],[3,1,2]], appleCost = [2,3,1], k = 3

Output:

[2,3,1]

Explanation:

It is always optimal to buy the apple in the starting city.

Constraints:

2 <= n <= 1000

1 <= roads.length <= 2000

1 <= a

i

, b

i

<= n

a

i

!= b

i

1 <= cost

i

<= 10

5

appleCost.length == n

1 <= appleCost[i] <= 10

5

1 <= k <= 100

There are no repeated edges.

Code Snippets

C++:

```
class Solution {
public:
vector<long long> minCost(int n, vector<vector<int>>& roads, vector<int>&
appleCost, int k) {

}
};
```

Java:

```
class Solution {
public long[] minCost(int n, int[][] roads, int[] appleCost, int k) {

}
}
```

Python3:

```
class Solution:
def minCost(self, n: int, roads: List[List[int]], appleCost: List[int], k:
int) -> List[int]:
```

Python:

```
class Solution(object):
def minCost(self, n, roads, appleCost, k):
"""
:type n: int
:type roads: List[List[int]]
:type appleCost: List[int]
:type k: int
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[][]} roads
* @param {number[]} appleCost
* @param {number} k
* @return {number[]}
*/
```

```
var minCost = function(n, roads, appleCost, k) {

};
```

TypeScript:

```
function minCost(n: number, roads: number[][], appleCost: number[], k:
number): number[] {

};
```

C#:

```
public class Solution {
public long[] MinCost(int n, int[][] roads, int[] appleCost, int k) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
long long* minCost(int n, int** roads, int roadsSize, int* roadsColSize, int*
appleCost, int appleCostSize, int k, int* returnSize) {

}
```

Go:

```
func minCost(n int, roads [][]int, appleCost []int, k int) []int64 {

}
```

Kotlin:

```
class Solution {
fun minCost(n: Int, roads: Array<IntArray>, appleCost: IntArray, k: Int):
LongArray {

}
```

```
}
```

Swift:

```
class Solution {
func minCost(_ n: Int, _ roads: [[Int]], _ appleCost: [Int], _ k: Int) ->
[Int] {

}
}
```

Rust:

```
impl Solution {
pub fn min_cost(n: i32, roads: Vec<Vec<i32>>, apple_cost: Vec<i32>, k: i32)
-> Vec<i64> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[][]} roads
# @param {Integer[]} apple_cost
# @param {Integer} k
# @return {Integer[]}
def min_cost(n, roads, apple_cost, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $roads
* @param Integer[] $appleCost
* @param Integer $k
* @return Integer[]
*/
```

```
function minCost($n, $roads, $appleCost, $k) {

}
}
```

Dart:

```
class Solution {
List<int> minCost(int n, List<List<int>> roads, List<int> appleCost, int k) {

}
}
```

Scala:

```
object Solution {
def minCost(n: Int, roads: Array[Array[Int]], appleCost: Array[Int], k: Int):
Array[Long] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_cost(n :: integer, roads :: [[integer]], apple_cost :: [integer], k
:: integer) :: [integer]
def min_cost(n, roads, apple_cost, k) do

end
end
```

Erlang:

```
-spec min_cost(N :: integer(), Roads :: [[integer()]], AppleCost ::
[integer()], K :: integer()) -> [integer()].
min_cost(N, Roads, AppleCost, K) ->
.
```

Racket:

```
(define/contract (min-cost n roads appleCost k)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer? (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
* 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:
vector<long long> minCost(int n, vector<vector<int>>& roads, vector<int>&
appleCost, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
* 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 long[] minCost(int n, int[][] roads, int[] appleCost, int k) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost to Buy Apples
Difficulty: Medium
Tags: array, graph, queue, heap

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 minCost(self, n: int, roads: List[List[int]], appleCost: List[int], k:
int) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minCost(self, n, roads, appleCost, k):
"""
:type n: int
:type roads: List[List[int]]
:type appleCost: List[int]
:type k: int
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
```

```
* 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 {number} n
* @param {number[][]} roads
* @param {number[]} appleCost
* @param {number} k
* @return {number[]}
*/
var minCost = function(n, roads, appleCost, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
* 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 minCost(n: number, roads: number[][], appleCost: number[], k:
number): number[] {

};
```

C# Solution:

```
/*
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
* 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 long[] MinCost(int n, int[][] roads, int[] appleCost, int k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost to Buy Apples
* Difficulty: Medium
* Tags: array, graph, queue, heap
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
long long* minCost(int n, int** roads, int roadsSize, int* roadsColSize, int*
appleCost, int appleCostSize, int k, int* returnSize) {

}
```

Go Solution:

```
// Problem: Minimum Cost to Buy Apples
// Difficulty: Medium
// Tags: array, graph, queue, heap
//
// 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 minCost(n int, roads [][]int, appleCost []int, k int) []int64 {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun minCost(n: Int, roads: Array<IntArray>, appleCost: IntArray, k: Int):
LongArray {

}
}
```

Swift Solution:

```
class Solution {
func minCost(_ n: Int, _ roads: [[Int]], _ appleCost: [Int], _ k: Int) ->
[Int] {

}
}
```

Rust Solution:

```
// Problem: Minimum Cost to Buy Apples
// Difficulty: Medium
// Tags: array, graph, queue, heap
//
// 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 min_cost(n: i32, roads: Vec<Vec<i32>>, apple_cost: Vec<i32>, k: i32)
-> Vec<i64> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[][]} roads
# @param {Integer[]} apple_cost
# @param {Integer} k
# @return {Integer[]}
def min_cost(n, roads, apple_cost, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[][] $roads
* @param Integer[] $appleCost
* @param Integer $k
* @return Integer[]
*/
function minCost($n, $roads, $appleCost, $k) {

}
}
```

Dart Solution:

```
class Solution {
List<int> minCost(int n, List<List<int>> roads, List<int> appleCost, int k) {

}
}
```

Scala Solution:

```
object Solution {
def minCost(n: Int, roads: Array[Array[Int]], appleCost: Array[Int], k: Int):
Array[Long] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_cost(n :: integer, roads :: [[integer]], apple_cost :: [integer], k
:: integer) :: [integer]
def min_cost(n, roads, apple_cost, k) do

end
end
```

Erlang Solution:

```
-spec min_cost(N :: integer(), Roads :: [[integer()]], AppleCost ::
[integer()], K :: integer()) -> [integer()].
min_cost(N, Roads, AppleCost, K) ->
.
```

Racket Solution:

```
(define/contract (min-cost n roads appleCost k)
(-> exact-integer? (listof (listof exact-integer?)) (listof exact-integer?)
exact-integer? (listof exact-integer?))
)
```