Problem 1425: Constrained Subsequence Sum

Problem Information

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

Problem Description

Given an integer array

nums

and an integer

k

, return the maximum sum of a

non-empty

subsequence of that array such that for every two

consecutive

integers in the subsequence,

nums[i]

and

nums[j]

, where

i < j

, the condition

j - i <= k

is satisfied.

A

subsequence

of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

Example 1:

Input:

nums = [10,2,-10,5,20], k = 2

Output:

37

Explanation:

The subsequence is [10, 2, 5, 20].

Example 2:

Input:

nums = [-1,-2,-3], k = 1

Output:

-1

Explanation:

The subsequence must be non-empty, so we choose the largest number.

Example 3:

Input:

nums = [10,-2,-10,-5,20], k = 2

Output:

23

Explanation:

The subsequence is [10, -2, -5, 20].

Constraints:

1 <= k <= nums.length <= 10

5

-10

4

<= nums[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
```

```
int constrainedSubsetSum(vector<int>& nums, int k) {

}
};
```

Java:

```
class Solution {
public int constrainedSubsetSum(int[] nums, int k) {

}
}
```

Python3:

```
class Solution:
def constrainedSubsetSum(self, nums: List[int], k: int) -> int:
```

Python:

```
class Solution(object):
def constrainedSubsetSum(self, nums, k):
"""
:type nums: List[int]
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number} k
* @return {number}
*/
var constrainedSubsetSum = function(nums, k) {

};
```

TypeScript:

```
function constrainedSubsetSum(nums: number[], k: number): number {

};
```

C#:

```
public class Solution {
public int ConstrainedSubsetSum(int[] nums, int k) {

}
}
```

C:

```
int constrainedSubsetSum(int* nums, int numsSize, int k) {

}
```

Go:

```
func constrainedSubsetSum(nums []int, k int) int {

}
```

Kotlin:

```
class Solution {
fun constrainedSubsetSum(nums: IntArray, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func constrainedSubsetSum(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn constrained_subset_sum(nums: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def constrained_subset_sum(nums, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function constrainedSubsetSum($nums, $k) {

}
}
```

Dart:

```
class Solution {
int constrainedSubsetSum(List<int> nums, int k) {

}
}
```

Scala:

```
object Solution {
def constrainedSubsetSum(nums: Array[Int], k: Int): Int = {

}
```

```
}
```

Elixir:

```
defmodule Solution do
@spec constrained_subset_sum(nums :: [integer], k :: integer) :: integer
def constrained_subset_sum(nums, k) do

end
end
```

Erlang:

```
-spec constrained_subset_sum(Nums :: [integer()], K :: integer()) ->
integer().
constrained_subset_sum(Nums, K) ->
.
```

Racket:

```
(define/contract (constrained-subset-sum nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
```

```
int constrainedSubsetSum(vector<int>& nums, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int constrainedSubsetSum(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Constrained Subsequence Sum
Difficulty: Hard
Tags: array, dp, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def constrainedSubsetSum(self, nums: List[int], k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def constrainedSubsetSum(self, nums, k):
"""
:type nums: List[int]
:type k: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} nums
* @param {number} k
* @return {number}
*/
var constrainedSubsetSum = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function constrainedSubsetSum(nums: number[], k: number): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int ConstrainedSubsetSum(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Constrained Subsequence Sum
* Difficulty: Hard
* Tags: array, dp, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int constrainedSubsetSum(int* nums, int numsSize, int k) {

}
```

Go Solution:

```
// Problem: Constrained Subsequence Sum
// Difficulty: Hard
```

```
// Tags: array, dp, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func constrainedSubsetSum(nums []int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun constrainedSubsetSum(nums: IntArray, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func constrainedSubsetSum(_ nums: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Constrained Subsequence Sum
// Difficulty: Hard
// Tags: array, dp, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn constrained_subset_sum(nums: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def constrained_subset_sum(nums, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer
*/
function constrainedSubsetSum($nums, $k) {

}
}
```

Dart Solution:

```
class Solution {
int constrainedSubsetSum(List<int> nums, int k) {

}
}
```

Scala Solution:

```
object Solution {
def constrainedSubsetSum(nums: Array[Int], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec constrained_subset_sum(nums :: [integer], k :: integer) :: integer
def constrained_subset_sum(nums, k) do

end
end
```

Erlang Solution:

```
-spec constrained_subset_sum(Nums :: [integer()], K :: integer()) ->
integer().
constrained_subset_sum(Nums, K) ->
.
```

Racket Solution:

```
(define/contract (constrained-subset-sum nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```