You are given an integer array values where values[i] represents the value of the ith sightseeing spot. Two sightseeing spots i and j have a **distance** j - i between them.

The score of a pair (i < j) of sightseeing spots is values[i] + values[j] + i - j: the sum of the values of the sightseeing spots, minus the distance between them.

Return *the maximum score of a pair of sightseeing spots*.

**Example 1:**

**Input:** values = [8,1,5,2,6]

**Output:** 11

**Explanation:** i = 0, j = 2, values[i] + values[j] + i - j = 8 + 5 + 0 - 2 = 11

**Example 2:**

**Input:** values = [1,2]

**Output:** 2

Let denote the maximal score among the values . In the simplest case, is the maximal score over the last two elements , and so we must have .

For , we claim the recursion

This recursion may be explained as follows: represents the maximal score among . When we prepend the term onto this list, we must ask whether the maximal pair has changed. If it has, then the new pair is for some , and the new maximal score is . If it has not, then the maximal pair still resides within and we have .

The maximal score among all the values is . The above idea may be coded naively as

class Solution:

    def maxScoreSightseeingPair(self, values: List[int]) -> int:

        n = len(values)

        @cache

        def s(i): # best score in values[i],...,values[n-1]

            if i == n-2:

                return values[n-2] + values[n-1] - 1

            return max(s(i+1), max(values[i] + values[j] + i - j for j in range(i+1,n)))

        return s(0)

However, this algorithm has time complexity because of the maximum over .

We can resolve this with a trick. Write

If we define , then we have the pair of recursions

with the boundary conditions

With these considerations in mind, the improved code reads

class Solution:

    def maxScoreSightseeingPair(self, values: List[int]) -> int:

        n = len(values)

        @cache

        def s(i):

            if i == n-2:

                return values[n-2] + values[n-1] - 1

            return max(s(i+1), values[i] + i + M(i+1))

        @cache

        def M(i):

            if i == n-1:

                return values[n-1] - (n-1)

            return max(M(i+1), values[i] - i)

        return s(0)

This will pass Leetcode’s time and memory criteria, but we can improve further by using for-loops in place of recursions. First, we’d need to resolve the inconsistent boundary indexes. We can write

or

I will choose the latter because it seems more intuitive. The third version of the code reads

class Solution:

    def maxScoreSightseeingPair(self, values: List[int]) -> int:

        n = len(values)

        s = [None] \* (n-1)

        M = [None] \* (n-1)

        s[n-2] = values[n-2] + values[n-1] - 1

        M[n-2] = max(values[n-2] - (n-2), values[n-1] - (n-1))

        for i in reversed(range(n-2)):

            s[i] = max(s[i+1], values[i] + i + M[i+1])

            M[i] = max(M[i+1], values[i] - i)

        return s[0]

This code runs in time and takes memory, but we can reduce the memory to by over-writing the variables and instead of storing them as arrays. In so doing, we obtain the fourth and final version of our code, which runs in time and takes memory

class Solution:

    def maxScoreSightseeingPair(self, values: List[int]) -> int:

        n = len(values)

        s = values[n-2] + values[n-1] - 1

        M = max(values[n-2] - (n-2), values[n-1] - (n-1))

        for i in reversed(range(n-2)):

            s = max(s, values[i] + i + M)

            M = max(M, values[i] - i)

        return s