Leetcode 1014 – Best Sightseeing Pair

You are given an integer array values where values[i] represents the value of the i^{th} 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 s_i denote the maximal score among the values $v_i, \dots, v_{n-1}$. In the simplest case, s_{n-2} is the maximal score over the last two elements v_{n-2}, v_{n-1}, and so we must have $s_{n-2} = v_{n-2} + v_{n-1} - 1$.

For $i < n-2$, we claim the recursion

$$s_i = \max\left\{s_{i+1}, \max_{j>i}\left(v_i + v_j + i - j\right)\right\}$$

This recursion may be explained as follows: s_{i+1} represents the maximal score among $v_{i+1}, \dots, v_{n-1}$. When we prepend the term v_i onto this list, we must ask whether the maximal pair has changed. If it has, then the new pair is (v_i, v_j) for some $v_j \in \{v_{i+1}, \dots, v_{n-1}\}$, and the new maximal score is $s_i = v_i + v_j + i - j$. If it has not, then the maximal pair still resides within $\{v_{i+1}, \dots, v_{n-1}\}$ and we have $s_i = s_{i+1}$.

The maximal score among all the values $v_0, \dots, v_{n-1}$ is s_0. 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 $O(n^2)$ time complexity because of the maximum over j.

Leetcode 1014 – Best Sightseeing Pair

We can resolve this with a trick. Write

$$\max_{j>i}(v_i + v_j + i - j) = v_i + i + \max_{j>i}(v_j - j)$$

If we define $M_i = \max_{j>i}(v_j - j)$, then we have the pair of recursions

$$\begin{aligned} s_i &= \max\{s_{i+1}, v_i + i + M_i\} \\ M_i &= \max\{M_{i+1}, v_i - i\} \end{aligned}$$

with the boundary conditions

$$\begin{aligned} s_{n-2} &= v_{n-2} + v_{n-1} - 1 \\ M_{n-1} &= v_{n-1} - (n-1) \end{aligned}$$

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

$$\begin{aligned} s_{n-1} &= -\infty \\ M_{n-1} &= v_{n-1} - (n-1) \end{aligned}$$

or

$$\begin{aligned} s_{n-2} &= v_{n-2} + v_{n-1} - 1 \\ M_{n-2} &= \max\{v_{n-2} - (n-2), v_{n-1} - (n-1)\} \end{aligned}$$

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

```
class Solution:
```

Leetcode 1014 – Best Sightseeing Pair

```
    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 $O(n)$ time and takes $O(n)$ memory, but we can reduce the memory to $O(1)$ by over-writing the variables s and M instead of storing them as arrays. In so doing, we obtain the fourth and final version of our code, which runs in $O(n)$ time and takes $O(1)$ 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
```