7MR10020: Scientific Programming
Assignment 2

Test	Method	Size	Time (List)	Time (NumPy)
1	Brute Force	5x5	~0s	~0s
2	Brute Force	10x10	1.012s	1.880s
3	Brute Force	13x13	48.162s	90.636s
4	Brute Force	20x20	Too long	Too long
5	Dynamic	20x20	N/A	~0s
6	Dynamic	200x200	N/A	0.310s
7	Dynamic	400x400	N/A	1.330s
8	Dynamic	2000x2000	N/A	~31s

Table 1: Runtime for the two methods for different image sizes and input types

Brute Force (with Recursion):

Using the brute force approach, we generate all the possible paths and then evaluate them to retrieve the best one. As can be seen in table 1, this is not a very good approach as it begins to struggle shortly after the array's/list's size becomes greater than 10x10. This is because the total number of paths being evaluated is increasing exponentially.

Comparing the two data types:

- A Python list uses a data buffer that contains pointers to its objects in memory. Therefore, while iterating over it, the python list just needs to look up that pointer and fetch the object.
- On the other hand, a NumPy array stores elements in a data buffer as bytes. To fetch them, it finds those bytes and then wraps them in a NumPy object. This means that it needs to recreate the number instead of just fetching it

For the solution of this problem, initially we're only interested in generating a list of all possible paths using the dimensions of our data structure and not actually looking at the values in it. Then using the newly created list (or generator) of paths, we evaluate the paths by retrieving the values at each node in the image and adding them up to get the best score/path. Therefore, since no actual arithmetic is performed on the image data structure, the Python list is faster than NumPy for this case.

Dynamic Programming:

Dynamic programming (DP) is often used to optimise recursive problems. This is done by storing the results of subproblems so that we do not have to re-evaluate them if they are needed again later and so we reduce the time complexity from exponential to polynomial. Perhaps we could have used a greedy algorithm, but since it won't always reach the best solution and DP is more than fast enough for our problem, DP is the better solution here.

Our problem has two features that make dynamic programming work:

1. An optimal substructure
2. Overlapping subproblems

As seen from the results in table 1, this approach was indeed an improvement over the previous one. The 20x20 matrix which would take probably days for the brute-force solution (tried up to 10 hours) needs ~0 seconds using DP. What's even more impressive is that DP at 2000x2000 only takes ~31 seconds to complete. The effectiveness of not having to re-evaluate subproblems that were encountered before is clearly shown by these results.

Furthermore, we can see the effect of NumPy arrays as well. Since we perform arithmetic operations on the list to compute the best path, it is a better option than a Python list.