1039. Minimum Score Triangulation of Polygon

```
Medium
                   Dynamic Programming
          <u>Array</u>
```

Problem Description

this polygon into triangles, in a way that minimizes the total score of the triangulation. The total score is defined as the sum of the scores of each triangle, where the score of a triangle is calculated as the product of the values of its three vertices. The challenge lies in determining the best way to triangulate the polygon (i.e., how exactly to make the cuts to form triangles), such that the total score is as small as possible. Intuition

You are given a convex, n-sided polygon where the value of each vertex is provided as an integer array values. The goal is to split

allows us to break down the problem into smaller subproblems, remember (or cache) those solutions, and use them to build up a solution to the larger problem. To apply dynamic programming, we look for a recurrence relation that expresses the solution to a problem in terms of the solutions to its subproblems. Here, we define dfs(i, j) as the minimum score possible for triangulating the sub-polygon from

incredibly inefficient because it would involve an exponential number of combinations to check. Instead, dynamic programming

The intuition behind solving this problem involves dynamic programming. Trying out each possible triangulation would be

vertex i to vertex j. To find dfs(i, j), we consider placing a triangle with one edge being the line segment from i to j, and the third vertex being k, where i < k < j. For each possible value of k, dfs(i, j) can then be recursively defined as the minimum of dfs(i, k) + dfs(k, j) + values[i] * values[k] * values[j].The base case for this recursive function is when the sub-polygon has only two vertices (i + 1 == j); in this case, no triangle

can be formed, and the score is 0.

To improve the efficiency of this algorithm, we use memoization, which is built into Python with the @cache decorator. This stores the results of dfs computations so that we don't have to recompute them when the same subproblems arise. Thus, the solution to our problem is the value of dfs(0, len(values) - 1), which represents the minimum score possible for

Solution Approach

The implementation of the solution leverages a design pattern known as <u>dynamic programming</u>. This programming paradigm solves complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid

The implementation starts by defining a function dfs(i, j) which represents the minimum triangulation score that can be achieved within the vertices from i to j of the polygon. This function uses the memoization technique with the @cache decorator

redundant computations.

triangulating the entire polygon.

to remember the results of previous computations, which is crucial for improving the performance of the solution. In the dfs function, there is a base case check:

• When i + 1 == j, it means there are only two vertices left, which cannot form a triangle. In this case, the score is 0. For the recursive step: We loop through all possible third vertices k for the triangle, where k ranges from i+1 to j−1.

• We add this score to the recursion of the remaining sub-polygon divided by this triangle, which results in two subproblems: dfs(i, k) and

and efficient solution to the problem.

dfs(k, j) representing the minimum score from i to k and from k to j, respectively. • The min function is applied to choose the smallest possible score amongst all combinations of k.

Thus, the function dfs computes the desired triangulation score in a bottom-up manner, using previously stored results to calculate larger subproblems. The time complexity of this approach is 0(n^3), where n is the number of vertices in the polygon, as

there are at most n choices for i and j, and within each recursive call, we iterate over k in a range of size at most n.

• For each k, we calculate the score of the triangle formed by vertices i, k, and j, which is values[i] * values[k] * values[j].

Finally, the minimum score for the entire polygon can be obtained by calling dfs(0, len(values) - 1), which represents the minimum triangulation score from the first to the last vertex of the polygon. The implementation of this algorithm uses no additional data structures apart from the given input and the internal caching

mechanism provided by @cache. The combination of recursion, memoization, and minimization at each step makes for an elegant

Example Walkthrough Let's consider a small example where our polygon has 4 vertices with the values array given as values = [1, 3, 1, 4].

score for each triangle is the product of its vertex values. Starting with the function dfs(i, j): We need to triangulate the polygon from vertex 0 to vertex 3, hence we call dfs(0, 3).

○ We then add this score to the score for the remaining sub-polygon formed by the vertices 0 to 1, and 1 to 3. Here, the sub-polygon 0-1 is

According to the problem, we aim to triangulate the polygon in such a way that the total score is as small as possible, where the

• We have two possibilities for vertex k as the third vertex for the triangle since k can be either 1 or 2. For k = 1:

 Now, for dfs(1, 3), we have only two points so again it does not form a triangle, resulting in score 0. \circ The total for k = 1 is 12 + 0 + 0 = 12. For k = 2:

Therefore, by recursively applying this approach using memoization to prevent redundant calculations, we determine that the

minimum score for triangulating the polygon with vertices 1, 3, 1, 4 is 7. The triangulation that yields this score is by slicing the

∘ We again add this score to the score for the sub-polygon formed by vertices 0 to 2, and 2 to 3. The sub-polygon 2–3 is just a line, so dfs(2, 3) is 0. • We need to calculate dfs(0, 2). This is a triangle [0, 1, 2] with a score 1 * 3 * 1 = 3.

We select the minimum score for our two k options, which is 7, so dfs(0, 3) returns 7.

just a line and does not form a triangle, so dfs(0, 1) is 0, and we need to calculate dfs(1, 3).

Since i + 1 does not equal j, we proceed to find the minimum triangulation score.

• We calculate the score of the triangle [0, 1, 3] which is 1 * 3 * 4 = 12.

• We calculate the score of the triangle [0, 2, 3] which is 1 * 1 * 4 = 4.

 \circ So, the total for k = 2 is 4 + 3 + 0 = 7.

polygon into triangles using vertices [0, 2, 3] and [0, 1, 2].

from functools import lru_cache # lru_cache is used since @cache is not defined

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

Compute the minimum score of a triangulation of a polygon with 'values' vertices. :param values: An array of scores associated with each vertex of the polygon. :return: The minimum score of a triangulation. @lru_cache(maxsize=None) # Adds memoization to reduce time complexity def min_score(i: int, j: int) -> int:

Compute the minimum score by triangulating the polygon from vertex i to vertex j.

:param i: Start vertex index of the polygon part being considered.

:return: Minimum triangulation score for the vertices between i and j.

between vertices 'i' and 'j', by choosing an intermediate vertex 'k'

triangulation scores for the polygons (i, ..., k) and (k, ..., j).

and adding the score formed by the triangle (i, k, j) plus the minimum

min_score(i, k) + min_score(k, j) + values[i] * values[k] * values[j]

for k in range(i + 1, j) # Iterate over all possible k between i and j

:param j: End vertex index of the polygon part being considered.

if i + 1 == j: # Base case: no polygon to triangulate if only two vertices return 0 # Compute the minimum score by considering all possible triangulations

return min(

int minScore = Integer.MAX_VALUE;

for (int k = start + 1; k < end; ++k) {</pre>

// Cache the calculated minimum score

memoization[start][end] = minScore;

return minScore;

minScore = Math.min(minScore, score);

// Iterate through all possible partitions of the polygon

// Find the minimum score from all possible partitions

// Recursively find the minimum score by considering the partition of the polygon

int score = dfs(start, k) + dfs(k, end) + vertexValues[start] * vertexValues[k] * vertexValues[end];

// into a triangle and two smaller polygons, and take the sum of their scores

Solution Implementation

Python

class Solution:

```
# Initiate the recursion with the full polygon from the first to the last vertex
       return min_score(0, len(values) - 1)
# Note: List and Tuple type hints should be imported from the typing module for this to work
from typing import List
Java
class Solution {
    private int numVertices;
    private int[] vertexValues;
    private Integer[][] memoization;
    // Calculate the minimum score of triangulation for a polygon using given vertex values
    public int minScoreTriangulation(int[] values) {
       numVertices = values.length;
       vertexValues = values;
       memoization = new Integer[numVertices][numVertices];
       return dfs(0, numVertices - 1);
    // Depth-First Search helper method to recursively find the minimum score
    private int dfs(int start, int end) {
       // Base case: If the polygon has no area (two adjacent vertices), return 0
       if (start + 1 == end) {
            return 0;
       // Check if the score has already been computed and cached
       if (memoization[start][end] != null) {
            return memoization[start][end];
       // Initialize answer to a large number so any minimum can replace it
```

```
C++
class Solution {
public:
    int minScoreTriangulation(vector<int>& values) {
       int n = values.size();
       // Dynamic programming matrix to store the minimum scores
       vector<vector<int>> dp(n, vector<int>(n, 0));
       // A lambda function that performs a depth-first search to find the minimum score
        function<int(int, int)> dfs = [&](int left, int right) -> int {
           // Base case: if only two vertices, no triangle can be formed
            if (left + 1 == right) {
                return 0;
           // If we have already computed this subproblem, return the stored value
            if (dp[left][right]) {
                return dp[left][right];
            // Initialize the answer to a high value. (1 << 30) represents a large number.
            int ans = 1 << 30;
           // Consider all possible points 'k' for triangulating the polygon
            for (int k = left + 1; k < right; ++k) {</pre>
                // Calculate the minimum score by considering the current triangle formed by vertices left, k, right
                // and adding the minimum scores of the two remaining sub-polygons
                ans = min(ans, dfs(left, k) + dfs(k, right) + values[left] * values[k] * values[right]);
           // Store the result in the dynamic programming matrix and return it
            dp[left][right] = ans;
            return ans;
       // Call the dfs function for the entire range of the given vertices
       return dfs(0, n - 1);
};
TypeScript
function minScoreTriangulation(values: number[]): number {
    const size = values.length;
    // Create a 2D array 'dp' for dynamic programming, initialized with zeros
    const dp: number[][] = Array.from({ length: size }, () => Array.from({ length: size }, () => 0));
   // Iterate over possible triangulations lengths
    for (let length = 3; length <= size; ++length) {</pre>
```

// Find minimum score triangulation for each subarray of 'values' of the current length

for (let start = 0; start + length - 1 < size; ++start) {</pre>

// Update the score of the triangulation

dp[start][end] = Number.MAX_SAFE_INTEGER;

for (let k = start + 1; k < end; ++k) {</pre>

// Initialize a high value to find a minimum later on

// Consider all points as possible third vertices to form triangles

const end = start + length - 1;

```
// Return the minimum score of triangulating the whole polygon
      return dp[0][size - 1];
from functools import lru_cache # lru_cache is used since @cache is not defined
class Solution:
   def min_score_triangulation(self, values: List[int]) -> int:
        Compute the minimum score of a triangulation of a polygon with 'values' vertices.
        :param values: An array of scores associated with each vertex of the polygon.
        :return: The minimum score of a triangulation.
       @lru_cache(maxsize=None) # Adds memoization to reduce time complexity
       def min_score(i: int, j: int) -> int:
            Compute the minimum score by triangulating the polygon from vertex i to vertex j.
            :param i: Start vertex index of the polygon part being considered.
            :param j: End vertex index of the polygon part being considered.
            :return: Minimum triangulation score for the vertices between i and j.
            if i + 1 == j: # Base case: no polygon to triangulate if only two vertices
               return 0
            # Compute the minimum score by considering all possible triangulations
            # between vertices 'i' and 'j', by choosing an intermediate vertex 'k'
            # and adding the score formed by the triangle (i, k, j) plus the minimum
            # triangulation scores for the polygons (i, ..., k) and (k, ..., j).
            return min(
               min_score(i, k) + min_score(k, j) + values[i] * values[k] * values[j]
                for k in range(i + 1, j) # Iterate over all possible k between i and j
       # Initiate the recursion with the full polygon from the first to the last vertex
        return min_score(0, len(values) - 1)
# Note: List and Tuple type hints should be imported from the typing module for this to work
from typing import List
```

dp[start][end] = Math.min(dp[start][end], dp[start][k] + dp[k][end] + values[start] * values[k] * values[end]);

Time and Space Complexity

Time Complexity: The time complexity of the algorithm can be determined by the number of unique subproblems and the time it takes to solve each subproblem. The function dfs is called with different arguments i and j, which represent the indices in the values array, effectively standing for vertices of the polygon.

The provided code defines a method minScoreTriangulation for finding the minimum score triangulation of a convex polygon

with the vertices labeled by integer values. The implementation uses a top-down dynamic programming approach with

and for every i, there are at most n choices for j. This gives us n^2 unique ranges. Inside each call to dfs, there is a for loop that iterates up to n times, which suggests that each subproblem is solved in O(n) time.

However, due to the memoization (@cache), these subproblems are solved only once, and the subsequent calls retrieve the results

Considering that i and j represent a range in the values array, there are at most n (where n is the length of values) choices for i,

in 0(1) time. Hence, the overall time complexity is $0(n^3)$, since we have n^2 subproblems, and each takes 0(n) time to compute initially.

Space Complexity:

memoization.

- The space complexity is determined by the space required for memoization and the depth of the recursion stack. • The memoization table or cache will store the results for each unique pair of i and j, resulting in 0(n^2) space.
- The maximum depth of the recursion tree can be n in case of a linear sequence of recursive calls, therefore adding O(n) space complexity. Together, the space complexity of the algorithm is $O(n^2)$ due to memoization being the dominant term.