Leet Code DP

This paper will cover some LeetCode Dynamic Programming.

# Top Down or Bottom up

In some type of Dynamic Programming problem we can do from top to bottom or from bottom to top. The traditional DP is from bottom to top, with the distance between start node and end node slowly growing.

If we do top to bottom we can consider the problem as divide and conquer, with the recursive function all.

Let’s see the following example:

**1039. Minimum Score Triangulation of Polygon**

Medium

Given N, consider a convex N-sided polygon with vertices labelled A[0], A[i], ..., A[N-1] in clockwise order.

Suppose you triangulate the polygon into N-2 triangles.  For each triangle, the value of that triangle is the **product** of the labels of the vertices, and the *total score* of the triangulation is the sum of these values over all N-2 triangles in the triangulation.

Return the smallest possible total score that you can achieve with some triangulation of the polygon.

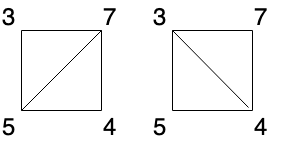
**Example 1:**

**Input:** [1,2,3]

**Output:** 6

**Explanation:** The polygon is already triangulated, and the score of the only triangle is 6.

**Example 2:**



**Input:** [3,7,4,5]

**Output:** 144

**Explanation:** There are two triangulations, with possible scores: 3\*7\*5 + 4\*5\*7 = 245, or 3\*4\*5 + 3\*4\*7 = 144. The minimum score is 144.

**Example 3:**

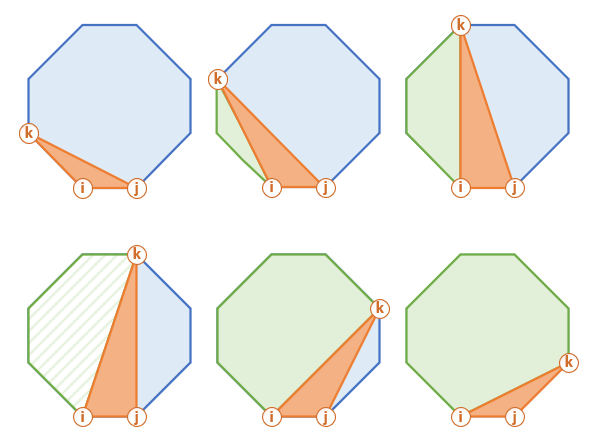
**Input:** [1,3,1,4,1,5]

**Output:** 13

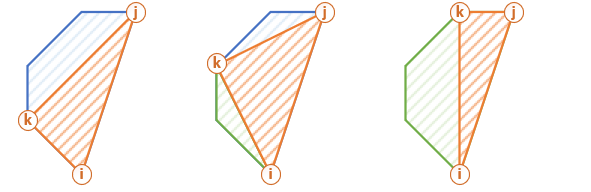
**Explanation:** The minimum score triangulation has score 1\*1\*3 + 1\*1\*4 + 1\*1\*5 + 1\*1\*1 = 13.

**Note:**

1. 3 <= A.length <= 50
2. 1 <= A[i] <= 100



This is how this procedure looks for a sub-polygon (filled with diagonal pattern above).



# Top-Down Solution

• Fix one side of the polygon i, j and move k within (i, j).  
• Calculate score of the i, k, j "orange" triangle.  
• Add the score of the "green" polygon i, k using recursion.  
• Add the score of the "blue" polygon k, j using recursion.  
• Use memoisation to remember minimum scores for each sub-polygons.

int LeetCode::minScoreTriangulation(vector<int>& A, int start, int end, unordered\_map<string, int>& cache)

{

if (end - start < 2) return 0;

string key = to\_string(start) + "," + to\_string(end);

if (cache.count(key) > 0) return cache[key];

int result = INT\_MAX;

for (int middle = start + 1; middle < end; middle++)

{

int sum = minScoreTriangulation(A, start, middle, cache);

sum += A[start] \* A[middle] \* A[end];

sum += minScoreTriangulation(A, middle, end, cache);

result = min(result, sum);

}

cache[key] = result;

return result;

}

int LeetCode::minScoreTriangulation(vector<int>& A)

{

unordered\_map<string, int> cache;

return minScoreTriangulation(A, 0, A.size() - 1, cache);

}

int LeetCode::minScoreTriangulationII(vector<int>& A)

{

vector<vector<int>> dp(A.size(), vector<int>(A.size()));

for (size\_t k = 2; k < A.size(); k++)

{

for (size\_t i = 0; i < A.size(); i++)

{

size\_t j = i + k;

if (j >= A.size()) break;

for (size\_t m = i + 1; m < j; m++)

{

if (dp[i][j] == 0)

{

dp[i][j] = dp[i][m] + A[i] \* A[m] \* A[j] + dp[m][j];

}

else

{

dp[i][j] = min(dp[i][j], dp[i][m] + A[i] \* A[m] \* A[j] + dp[m][j]);

}

}

}

}

return dp[0][A.size() - 1];

}

# Backpack Problem

The tradional backpack problem is to assume the backpack size as integer and iterate the back size from 0 to full by try each possible item. However, there is another solution which is to build up all possible pack size with maximum value. Based on my experience, this is easier to understand.

**474. Ones and Zeroes**

Medium

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of **m** 0s and **n** 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given **m** 0s and **n** 1s. Each 0 and 1 can be used at most **once**.

**Note:**

1. The given numbers of 0s and 1s will both not exceed 100
2. The size of given string array won't exceed 600.

**Example 1:**

**Input:** Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3

**Output:** 4

**Explanation:** This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

**Example 2:**

**Input:** Array = {"10", "0", "1"}, m = 1, n = 1

**Output:** 2

**Explanation:** You could form "10", but then you'd have nothing left. Better form "0" and "1".

int LeetCode::findMaxOneZeroForm(vector<string>& strs, int m, int n)

{

if (strs.size() == 0) return 0;

vector<pair<int, int>> str\_count\_list;

for (size\_t i = 0; i < strs.size(); i++)

{

pair<int, int> str\_count = make\_pair(0, 0);

for (size\_t j = 0; j < strs[i].size(); j++)

{

if (strs[i][j] == '0') str\_count.first++;

else if (strs[i][j] == '1') str\_count.second++;

}

str\_count\_list.push\_back(str\_count);

}

vector<vector<int>> dp\_map(vector<vector<int>>(m + 1, vector<int>(n + 1)));

for (size\_t i = 0; i < str\_count\_list.size(); i++)

{

for (int zero = m; zero >= 0; zero--)

{

for (int one = n; one >= 0; one--)

{

if ((zero >= str\_count\_list[i].first) && (one >= str\_count\_list[i].second))

{

dp\_map[zero][one] = max(1 + dp\_map[zero - str\_count\_list[i].first][one - str\_count\_list[i].second], dp\_map[zero][one]);

}

}

}

}

return dp\_map[m][n];

}

int LeetCode::findMaxOneZeroFormII(vector<string>& strs, int m, int n)

{

unordered\_map<int, unordered\_map<int, int>> dp;

int result = 0;

for (size\_t i = 0; i < strs.size(); i++)

{

string str = strs[i];

int zero = std::count(str.begin(), str.end(), '0');

int one = std::count(str.begin(), str.end(), '1');

if (zero > m || one > n) continue;

unordered\_map<int, unordered\_map<int, int>> next = dp;

next[zero][one] = max(dp[zero][one], 1);

result = max(result, next[zero][one]);

for (auto zero\_itr : dp)

{

int add\_zero = zero + zero\_itr.first;

if (add\_zero > m) continue;

for (auto one\_itr : zero\_itr.second)

{

int add\_one = one + one\_itr.first;

if (add\_one > n) continue;

next[add\_zero][add\_one] = max(next[add\_zero][add\_one], one\_itr.second + 1);

result = max(result, next[add\_zero][add\_one]);

}

}

dp = next;

}

return result;

}

**1049. Last Stone Weight II**

Medium

860FavoriteShare

We have a collection of rocks, each rock has a positive integer weight.

Each turn, we choose **any two rocks** and smash them together.  Suppose the stones have weights x and y with x <= y.  The result of this smash is:

* If x == y, both stones are totally destroyed;
* If x != y, the stone of weight x is totally destroyed, and the stone of weight y has new weight y-x.

At the end, there is at most 1 stone left.  Return the **smallest possible** weight of this stone (the weight is 0 if there are no stones left.)

**Example 1:**

**Input:** [2,7,4,1,8,1]

**Output:** 1

**Explanation:**

We can combine 2 and 4 to get 2 so the array converts to [2,7,1,8,1] then,

we can combine 7 and 8 to get 1 so the array converts to [2,1,1,1] then,

we can combine 2 and 1 to get 1 so the array converts to [1,1,1] then,

we can combine 1 and 1 to get 0 so the array converts to [1] then that's the optimal value.

**Note:**

1. 1 <= stones.length <= 30
2. 1 <= stones[i] <= 100

If we think that we do not know the backpack size we can do this way:

int LeetCode::lastStoneWeightII(vector<int>& stones)

{

int sum = 0;

for (size\_t i = 0; i < stones.size(); i++) sum += stones[i];

unordered\_set<int> dp;

for (size\_t i = 0; i < stones.size(); i++)

{

if (dp.empty())

{

if (stones[i] \* 2 <= sum) dp.insert(stones[i]);

}

else

{

unordered\_set<int> next;

if (stones[i] \* 2 <= sum) next.insert(stones[i]);

for (auto x : dp)

{

if ((x + stones[i]) \* 2 <= sum) next.insert(x + stones[i]);

}

dp.insert(next.begin(), next.end());

}

}

int result = INT\_MAX;

for (auto x : dp)

{

result = min(result, sum - 2\*x);

}

return result;

}

But in this problem actually we can know the maximum backpack size is 1500, (half of total weight of maximum stones as negative. You can use the following formula.

int LeetCode::lastStoneWeightIIA(vector<int> &A)

{

bitset<1501> dp = { 1 };

int sum = 0, res = 100;

for (int a : A)

{

sum += a;

for (int i = 1500; i >= a; --i)

{

dp[i] = dp[i] + dp[i - a];

}

}

for (int i = sum / 2; i > 0; --i)

{

if (dp[i]) return sum - i - i;

}

return 0;

}

# Longest Common Subsequence

**1035. Uncrossed Lines**

Medium

1325FavoriteShare

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

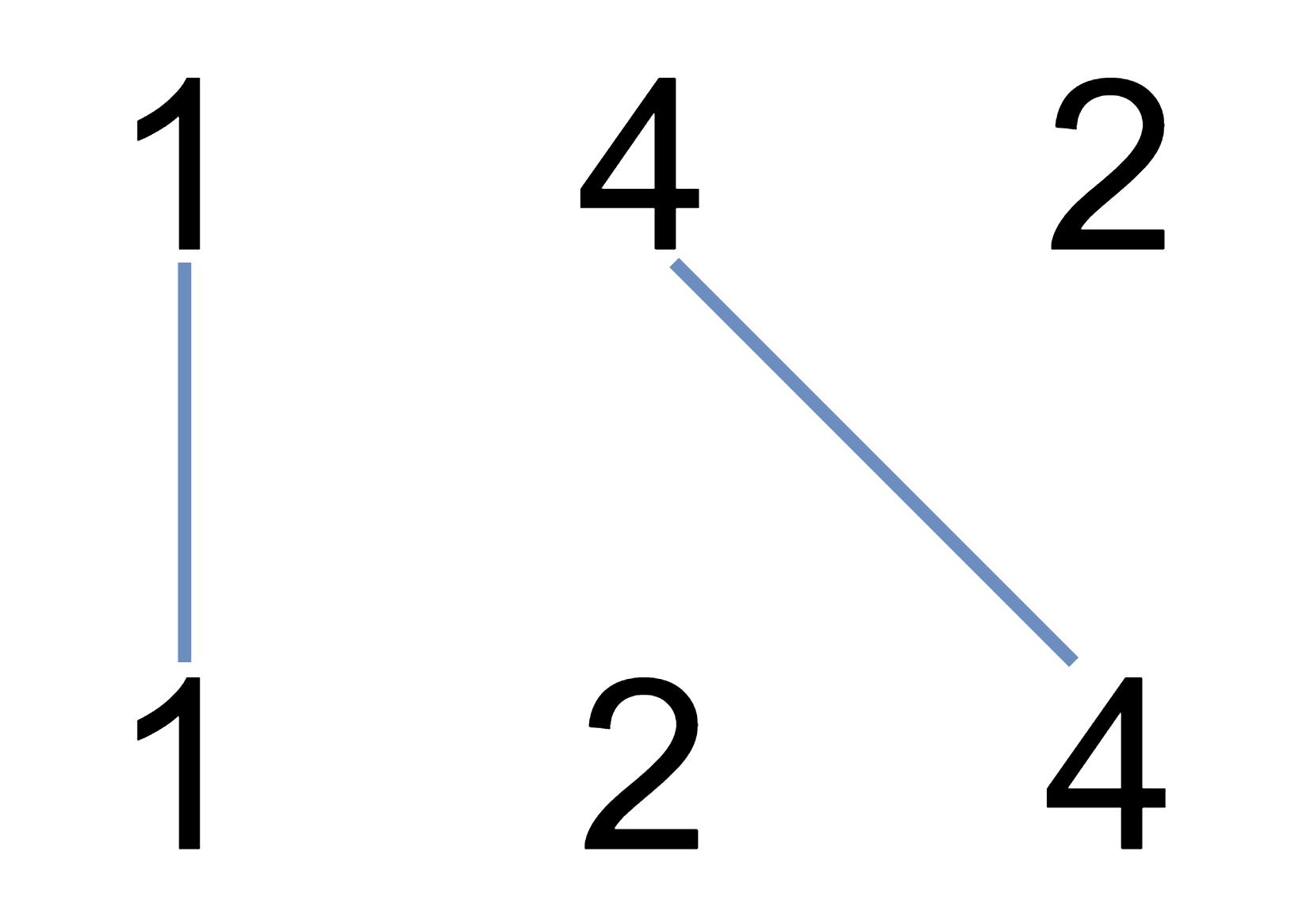
Now, we may draw *connecting lines*: a straight line connecting two numbers A[i] and B[j] such that:

* A[i] == B[j];
* The line we draw does not intersect any other connecting (non-horizontal) line.

Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

**Example 1:**



**Input:** A = [1,4,2], B = [1,2,4]

**Output:** 2

**Explanation:** We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

**Example 2:**

**Input:** A = [2,5,1,2,5], B = [10,5,2,1,5,2]

**Output:** 3

**Example 3:**

**Input:** A = [1,3,7,1,7,5], B = [1,9,2,5,1]

**Output:** 2

**Note:**

1. 1 <= A.length <= 500
2. 1 <= B.length <= 500
3. 1 <= A[i], B[i] <= 2000

This problem can be regarded as a longest common sequence, so map to the string comparison algorithm in DP.

We compare two string with comparing each character based on their sequence. If two characters are same we take the left-up corner number and add one, otherwise we choose the maximum of upper side or left side.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 5 | 2 | 1 | 5 | 2 |
| 2 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 2 | 2 |
| 1 | 0 | 1 | 1 | 2 | 2 | 2 |
| 2 | 0 | 1 | 2 | 2 | 2 | 3 |
| 5 | 0 | 1 | 2 | 2 | 3 | 3 |

int LeetCode::maxUncrossedLines(vector<int>& A, vector<int>& B)

{

vector<vector<int>> dp(A.size() + 1, vector<int>(B.size() + 1));

dp[0][0] = 0;

for (size\_t i = 0; i < A.size(); i++)

{

for (size\_t j = 0; j < B.size(); j++)

{

dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j]);

if (A[i] == B[j]) dp[i + 1][j + 1] = max(dp[i + 1][j+1], dp[i][j] + 1);

}

}

return dp[A.size()][B.size()];

}

# DP with many choices

When we think DP in a continus array, normally we should think from the point with any index, and only look the subarray before this point, and see how may option we should collect the data and record them.

For example, in the following problem on any index in the array, we can see the partition ending with current index as 1, 2, 3, … K

**1043. Partition Array for Maximum Sum**

Medium

11710FavoriteShare

Given an integer array A, you partition the array into (contiguous) subarrays of length at most K.  After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return the largest sum of the given array after partitioning.

**Example 1:**

**Input:** A = [1,15,7,9,2,5,10], K = 3

**Output:** 84

**Explanation**: A becomes [15,15,15,9,10,10,10]

**Note:**

1. 1 <= K <= A.length <= 500
2. 0 <= A[i] <= 10^6

int LeetCode::maxSumAfterPartitioning(vector<int>& A, int K)

{

vector<int> result(A.size());

vector<int> dp(A.size());

for (int i = 0; i < (int)A.size(); i++)

{

dp[i] = A[i];

for (int j = 0; j < K; j++)

{

int first = i - j;

if (first < 0) break;

dp[i] = max(dp[i], A[first]);

int prev = first - 1;

if (prev < 0)

{

result[i] = max(result[i], dp[i] \* (j + 1));

}

else

{

result[i] = max(result[i], result[prev] + dp[i] \* (j + 1));

}

}

}

return result[result.size() - 1];

}