# From Center and Spread

For some DP problem, we try to find the best answer within of a one dimension array, we should think from small to big, first try to find the answer on every element or every two adjacent elements, the expand the window to one more or two more by adding the neighbor on the left and on the right, keep on growing the window to the whole array. When we are resolve a bigger window, which is not with two adjacent elements, we can always assume the smaller window within the big one is always resolved.

Sometimes, to resolve a window, we need a cursor move from the left end of the window to the right end of the window.

It is quite interesting that most of such problem has at least second solution, for example a divide and conquer based on backtracking with memorization. The idea is also simple, you divide the whole array into left window and right window, at very beginning you do not know the answer from left or from right, but you can recursively solve them. For a repeated window use the answer from the cache.

## 5. Longest Palindromic Substring

Medium

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.

**Example 1:**

**Input:** "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**Example 2:**

**Input:** "cbbd"

**Output:** "bb"

### Analysis

We have a two dimension matrix dp[i][j], i < j, which indicate from the index i to j, including the substring is palindromic or not.

/// <summary>

/// Leet code #5. Longest Palindromic Substring

/// Given a string S, find the longest palindromic substring in S.

/// You may assume that the maximum length of S is 1000.

///

/// Example 1:

/// Input: "babad"

/// Output : "bab"

/// Note : "aba" is also a valid answer.

///

/// Example 2 :

/// Input : "cbbd"

/// Output : "bb"

/// </summary>

string LeetCode::longestPalindrome(string s)

{

string result;

size\_t n = s.size();

vector<vector<int>> dp(n, vector<int>(n));

for (size\_t k = 0; k < n; k++)

{

for (size\_t i = 0; i < n - k; i++)

{

if (s[i] == s[i + k])

{

if ((k < 2) || (dp[i + 1][i + k - 1] == 1))

{

dp[i][i + k] = 1;

if (k + 1 > result.size())

{

result = s.substr(i, k + 1);

}

}

}

}

}

return result;

}

## 375. Guess Number Higher or Lower II

Medium

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to **n**. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.

However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

**Example:**

n = 10, I pick 8.

First round: You guess 5, I tell you that it's higher. You pay $5.

Second round: You guess 7, I tell you that it's higher. You pay $7.

Third round: You guess 9, I tell you that it's lower. You pay $9.

Game over. 8 is the number I picked.

You end up paying $5 + $7 + $9 = $21.

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

### Analysis

The cost to pick any specific number is the cost of the number itself, plus the maximum of upper window and lower window. Starting from a single number and build the 2D matrix

/// <summary>

/// Leet code #375. Guess Number Higher or Lower II

///

/// We are playing the Guess Game. The game is as follows:

/// I pick a number from 1 to n. You have to guess which number I picked.

/// Every time you guess wrong, I'll tell you whether the number I picked

/// is higher or lower.

/// However, when you guess a particular number x, and you guess wrong,

/// you pay $x.

/// You win the game when you guess the number I picked.

/// Example:

/// n = 10, I pick 8.

/// First round: You guess 5, I tell you that it's higher. You pay $5.

/// Second round: You guess 7, I tell you that it's higher. You pay $7.

/// Third round: You guess 9, I tell you that it's lower. You pay $9.

/// Game over. 8 is the number I picked.

/// You end up paying $5 + $7 + $9 = $21.

/// Given a particular n ≥ 1, find out how much money you need to have to

/// guarantee a win.

/// Hint:

/// 1.The best strategy to play the game is to minimize the maximum loss you

/// could possibly face.

/// Another strategy is to minimize the expected loss. Here, we are interested

/// in the first scenario.

/// 2.Take a small example (n = 3). What do you end up paying in the worst case?

/// 3.Check out this article if you're still stuck.

/// 4.The purely recursive implementation of minimax would be worthless for even a small n.

/// You MUST use dynamic programming.

/// 5.As a follow-up, how would you modify your code to solve the problem of minimizing the

/// expected loss, instead of the worst-case loss?

/// </summary>

int LeetCode::getMoneyAmount(int n)

{

vector<vector<int>> matrix(n, vector<int>(n));

for (int step = 0; step < n; step++)

{

for (int i = 0; i < n; i++)

{

if (step == 0)

{

matrix[i][i + step] = 0;

}

else if ((step == 1) && (i + step < n))

{

matrix[i][i + step] = i + 1;

}

else if (i + step < n)

{

int min\_value = INT\_MAX;

for (int k = i + 1; k < i + step; k++)

{

int left = matrix[i][k - 1];

int right = matrix[k + 1][i + step];

int value = max(left, right) + k + 1;

min\_value = min(min\_value, value);

}

matrix[i][i + step] = min\_value;

}

else

{

break;

}

}

}

return matrix[0][n - 1];

}

## 486. Predict the Winner

Medium

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player. This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

**Example 1:**

**Input:** [1, 5, 2]

**Output:** False

**Explanation:** Initially, player 1 can choose between 1 and 2.   
If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).   
So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.   
Hence, player 1 will never be the winner and you need to return False.

**Example 2:**

**Input:** [1, 5, 233, 7]

**Output:** True

**Explanation:** Player 1 first chooses 1. Then player 2 have to choose between 5 and 7. No matter which number player 2 choose, player 1 can choose 233.  
Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.

**Note:**

1. 1 <= length of the array <= 20.
2. Any scores in the given array are non-negative integers and will not exceed 10,000,000.
3. If the scores of both players are equal, then player 1 is still the winner.

### Analysis

For any one stone you know the winner, then how about 2, 3,.., n-1, n? At every window you will have a maximum stones for the winner and the remaining for the loser, with one more stone, the winner and loser can change.

/// <summary>

/// Leet code #486. Predict the Winner

///

/// Given an array of scores that are non-negative integers. Player 1 picks

/// one of the numbers from either end of the array followed by the player 2

/// and then player 1 and so on. Each time a player picks a number, that

/// number will not be available for the next player. This continues until

/// all the scores have been chosen. The player with the maximum score wins.

///

/// Given an array of scores, predict whether player 1 is the winner. You

/// can assume each player plays to maximize his score.

///

/// Example 1:

/// Input: [1, 5, 2]

/// Output: False

/// Explanation: Initially, player 1 can choose between 1 and 2.

/// If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5.

/// If player 2 chooses 5, then player 1 will be left with 1 (or 2).

/// So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.

/// Hence, player 1 will never be the winner and you need to return False.

///

/// Example 2:

/// Input: [1, 5, 233, 7]

/// Output: True

/// Explanation: Player 1 first chooses 1. Then player 2 have to choose

/// between 5 and 7. No matter which number player 2 choose, player 1 can

/// choose 233.

/// Finally, player 1 has more score (234) than player 2 (12), so you need

/// to return True representing player1 can win.

///

/// Note:

/// 1.1 <= length of the array <= 20.

/// 2.Any scores in the given array are non-negative integers and will not

/// exceed 10,000,000.

/// 3.If the scores of both players are equal, then player 1 is still the winner.

/// </summary>

bool LeetCode::predictTheWinner(vector<int>& nums)

{

if (nums.empty()) return true;

vector<vector<pair<int, int>>> sum(nums.size(), vector<pair<int, int>>(nums.size()));

for (size\_t step = 0; step < nums.size(); step++)

{

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

{

int first; // winner stone

int second; // loser stone

if (step == 0)

{

first = nums[i];

second = 0;

sum[i][i + step] = make\_pair(first, second);

}

else if (i + step < nums.size())

{

first = max(nums[i] + sum[i + 1][i + step].second, nums[i + step] + sum[i][i + step - 1].second);

second = nums[i] + sum[i + 1][i + step].first + sum[i + 1][i + step].second - first;

sum[i][i + step] = make\_pair(first, second);

}

}

}

return (sum[0][nums.size() - 1].first >= sum[0][nums.size() - 1].second);

}

## 516. Longest Palindromic Subsequence

Medium

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

**Example 1:**  
Input:

"bbbab"

Output:

4

One possible longest palindromic subsequence is "bbbb".

**Example 2:**  
Input:

"cbbd"

Output:

2

One possible longest palindromic subsequence is "bb".

### Analysis

Starting from length of 1, then 2, 3…n, expanding the window. On any window, if left edge is equal to the right edge, it is 2 + f(inner string), or it is max(f(left part), f(right part)

/// <summary>

/// Leet code #516. Longest Palindromic Subsequence

///

/// Given a string s, find the longest palindromic subsequence's length in s. You may

/// assume that the maximum length of s is 1000.

///

/// Example 1:

/// Input:

/// "bbbab"

/// Output: 4

/// One possible longest palindromic subsequence is "bbbb".

///

/// Example 2:

/// Input:

/// "cbbd"

/// Output: 2

/// One possible longest palindromic subsequence is "bb".

/// </summary>

int LeetCode::longestPalindromeSubseq(string s)

{

int n = s.size();

vector<vector<int>> matrix(n, vector<int>(n));

for (int step = 0; step < n; step++)

{

for (int i = 0; i < n; i++)

{

if (step == 0)

{

matrix[i][i + step] = 1;

}

else if ((step == 1) && (i + step < n))

{

matrix[i][i + step] = (s[i] == s[i + step]) ? 2 : 1;

}

else if (i + step < n)

{

int max\_value = INT\_MIN;

if (s[i] == s[i + step])

{

max\_value = max(max\_value, 2 + matrix[i + 1][i + step - 1]);

}

max\_value = max(max\_value, matrix[i + 1][i + step]);

max\_value = max(max\_value, matrix[i][i + step - 1]);

matrix[i][i + step] = max\_value;

}

else

{

break;

}

}

}

return matrix[0][n - 1];

}

### Another Solution

If you do sequence match the string to the reverse string, it also works.

## 647. Palindromic Substrings

Medium

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

**Example 1:**

**Input:** "abc"

**Output:** 3

**Explanation:** Three palindromic strings: "a", "b", "c".

**Example 2:**

**Input:** "aaa"

**Output:** 6

**Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

**Note:**

1. The input string length won't exceed 1000.

### Analysis

We can start the window with one letter, then 2 letters then 3, check the palindromic substring and count.

/// <summary>

/// Leetcode #647. Palindromic Substrings

///

/// Given a string, your task is to count how many palindromic substrings

/// in this string.

/// The substrings with different start indexes or end indexes are counted

/// as different substrings even they consist of same characters.

/// Example 1:

/// Input: "abc"

/// Output: 3

/// Explanation: Three palindromic strings: "a", "b", "c".

///

/// Example 2:

/// Input: "aaa"

/// Output: 6

/// Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

///

/// Note:

/// The input string length won't exceed 1000.

/// </summary>

int LeetCode::countSubstrings(string s)

{

int n = s.size();

int result = 0;

vector<vector<bool>> dp(n, vector<bool>(n, false));

for (size\_t k = 0; k < s.size(); k++)

{

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

{

if (i + k >= s.size()) break;

if (k == 0)

{

dp[i][i] = true;

result++;

}

else if (s[i] == s[i + k])

{

if ((k == 1) || (dp[i + 1][i + k - 1]))

{

dp[i][i + k] = true;

result++;

}

}

}

}

return result;

}

### Another Solution

You can also consider sequence match with the reverse string, and count the whole match, until its original position.

## 664. Strange Printer

Hard

There is a strange printer with the following two special requirements:

1. The printer can only print a sequence of the same character each time.
2. At each turn, the printer can print new characters starting from and ending at any places, and will cover the original existing characters.

Given a string consists of lower English letters only, your job is to count the minimum number of turns the printer needed in order to print it.

**Example 1:**

**Input:** "aaabbb"

**Output:** 2

**Explanation:** Print "aaa" first and then print "bbb".

**Example 2:**

**Input:** "aba"

**Output:** 2

**Explanation:** Print "aaa" first and then print "b" from the second place of the string, which will cover the existing character 'a'.

**Hint**: Length of the given string will not exceed 100.

### Analysis

Split a string at any place, the whole cost will be the left part plus the right part, end of first part is same as end of last part, we can deduct count by 1.

int LeetCode::strangePrinter(string s)

{

if (s.empty()) return 0;

vector<vector<int>> dp(s.size(), vector<int>(s.size(), INT\_MAX));

for (size\_t len = 0; len < s.size(); len++)

{

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

{

if (i + len >= s.size()) break;

if (len == 0)

{

dp[i][i + len] = 1;

continue;

}

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

{

int count = dp[i][j] + dp[j + 1][i + len];

if (s[j] == s[i + len]) count--;

dp[i][i + len] = min(dp[i][i + len], count);

}

}

}

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

}

## 312. Burst Balloons

Hard

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

**Note:**

* You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.
* 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

**Example:**

**Input:** [3,1,5,8]

**Output:** 167

**Explanation:** nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

  coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

### Analysis

index: 0 1 2 3

+---------------+

nums: 1 | 3 | 1 | 5 | 8 | 1

+---------------+

// It may be easier to figure out the recurrence relation by looking at the last step.

// At the last step, you only have one ballon (3, 1, 5 or 8) to burst, right?

// So let's have something like opt(END) to denote max coins we can get at the end,

// we know this opt(END) must come from one of the above 4 cases (by bursting 3, 1, 5 or 8)

// So it seems like a good idea to use opt(i, j) to represent the max coins we can get from bursting balloons[i, j],

// let's change opt(END) to opt(i, j) accordingly

opt(0, 3) = max(

// If 3 is the last ballon, this means balloons[1, 5, 8] have been burst already,

// since we don't know the max coins we got by bursting[1, 5, 8], we'll

// use opt(1, 3) to denote max coins we can get by bursing balloons from 1 to 3, which is [1, 5, 8]

1 \* 3 \* 1 + opt(1, 3),

// Similarily, we have

1 \* 1 \* 1 + opt(0, 0) + opt(2, 3),

1 \* 5 \* 1 + opt(0, 1) + opt(3, 3),

1 \* 8 \* 1 + opt(0, 2)

)

// You may have figured out the recurrence relation from the above equation, if not,

// let's look at how we'll calculate opt(1, 3)

// Similar to the above equation,

opt(1, 3) = max(

3 \* 1 \* 1 + opt(2, 3),

3 \* 5 \* 1 + opt(1, 1) + opt(3, 3),

3 \* 8 \* 1 + opt(1, 2)

)

// Thus, we have

>> state:

opt[i][j] denotes the max coins we can get by bursting balloons[i, j]

>> recurrence relationship:

for k from i to j

max(

// Edge cases are not considered in the below equation!

nums[i - 1] \* nums[k] \* nums[j + 1] + opt[i][k - 1] + opt[k + 1][j]

)

/// <summary>

/// Leet code #312. Burst Balloons

/// Given n balloons, indexed from 0 to n-1. Each balloon is painted with a

/// number on it represented by array nums.

/// You are asked to burst all the balloons. If the you burst balloon i you

/// will get nums[left] \* nums[i] \* nums[right] coins.

/// Here left and right are adjacent indices of i. After the burst, the left

/// and right then becomes adjacent.

/// Find the maximum coins you can collect by bursting the balloons wisely.

/// Note:

/// (1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore

/// you can not burst them.

/// (2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

/// Example:

/// Given [3, 1, 5, 8]

/// Return 167

/// nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

/// coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

/// </summary>

int LeetCode::maxBurstBalloonCoins(vector<int>& nums)

{

vector<int> balloons;

balloons.push\_back(1);

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

{

balloons.push\_back(nums[i]);

}

balloons.push\_back(1);

vector<vector<int>> coins(balloons.size(), vector<int>(balloons.size(), 0));

for (size\_t gap = 2; gap < balloons.size(); gap++)

{

for (size\_t i = 0; i < balloons.size() - gap; i++)

{

size\_t first = i;

size\_t last = i + gap;

if (gap == 2)

{

coins[first][last] = balloons[first] \* balloons[first + 1] \* balloons[last];

}

else

{

int max\_coins = 0;

for (size\_t j = first + 1; j < last; j++)

{

int sum\_coins =

coins[first][j] + coins[j][last] +

balloons[first] \* balloons[j] \* balloons[last];

max\_coins = max(max\_coins, sum\_coins);

}

coins[first][last] = max\_coins;

}

}

}

return coins[0][balloons.size() - 1];

}

## 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.

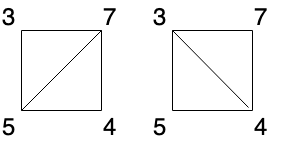
**Example 1:**

**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

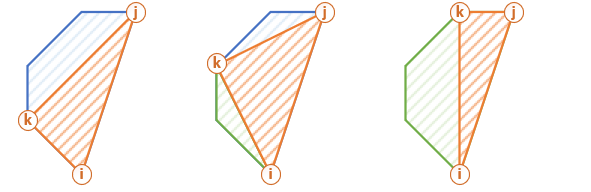
**Explanation:** The minimum score triangulation has score 1\*1\*3 + 1\*1\*4 + 1\*1\*5 + 1\*1\*1 = 13.

**Note:**

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

### Analysis

We can always start with adjacent 3 points, calculate the score, then expand to adjacent 4 points, and move the cursor between the second point and the third point and calculate the triangles with different scenarios.



/// <summary>

/// Leet code #1039. Minimum Score Triangulation of Polygon

///

/// 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

/// </summary>

int LeetCode::minScoreTriangulation(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];

}

### Another Solution

Please notice the DP with from center and spread pattern can also be resolve by backtracking, with memorization.

#### **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.

/// <summary>

/// Leet code #1039. Minimum Score Triangulation of Polygon

///

/// 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

/// </summary>

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

{

unordered\_map<string, int> cache;

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

}

/// <summary>

/// Leet code #1039. Minimum Score Triangulation of Polygon

/// </summary>

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;

}

## 1130. Minimum Cost Tree From Leaf Values

Medium

Given an array arr of positive integers, consider all binary trees such that:

* Each node has either 0 or 2 children;
* The values of arr correspond to the values of each **leaf** in an in-order traversal of the tree.  *(Recall that a node is a leaf if and only if it has 0 children.)*
* The value of each non-leaf node is equal to the product of the largest leaf value in its left and right subtree respectively.

Among all possible binary trees considered, return the smallest possible sum of the values of each non-leaf node.  It is guaranteed this sum fits into a 32-bit integer.

**Example 1:**

**Input:** arr = [6,2,4]

**Output:** 32

**Explanation:**

There are two possible trees. The first has non-leaf node sum 36, and the second has non-leaf node sum 32.

24 24

/ \ / \

12 4 6 8

/ \ / \

6 2 2 4

**Constraints:**

* 2 <= arr.length <= 40
* 1 <= arr[i] <= 15
* It is guaranteed that the answer fits into a 32-bit signed integer (ie. it is less than 2^31).

### Analysis

We can always split a window in the array into two parts as left and right, and think them as left sub tree and right sub tree. We just need to record the sum of non-leaf and the maximum leaf. This problem is O(N^3).

/// <summary>

/// Leet code #1130. Minimum Cost Tree From Leaf Values

///

/// Given an array arr of positive integers, consider all binary trees such

/// that:

///

/// Each node has either 0 or 2 children;

/// The values of arr correspond to the values of each leaf in an in-order

/// traversal of the tree. (Recall that a node is a leaf if and only if it

/// has 0 children.)

/// The value of each non-leaf node is equal to the product of the largest

/// leaf value in its left and right subtree respectively.

/// Among all possible binary trees considered, return the smallest possible

/// sum of the values of each non-leaf node. It is guaranteed this sum fits

/// into a 32-bit integer.

///

///

/// Example 1:

/// Input: arr = [6,2,4]

/// Output: 32

/// Explanation:

/// There are two possible trees. The first has non-leaf node sum 36, and

/// the second has non-leaf node sum 32.

///

/// 24 24

/// / \ / \

/// 12 4 6 8

/// / \ / \

/// 6 2 2 4

///

/// Constraints:

/// 1. 2 <= arr.length <= 40

/// 2. 1 <= arr[i] <= 15

/// 3. It is guaranteed that the answer fits into a 32-bit signed integer

/// (ie. it is less than 2^31).

/// </summary>

int LeetCode::mctFromLeafValues(vector<int>& arr)

{

size\_t n = arr.size();

vector<vector<pair<int, int>>> dp(n, vector<pair<int, int>>(n));

for (size\_t step = 0; step < n; step++)

{

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

{

if (step == 0)

{

dp[i][i] = make\_pair(arr[i], 0);

}

else if (i + step >= n)

{

break;

}

else

{

size\_t left = i;

size\_t right = i + step;

int max\_leaf = 0;

int min\_sum = INT\_MAX;

for (size\_t k = 0; k < n; k++)

{

if (left + k + 1 > right) break;

max\_leaf = max(max\_leaf, dp[left][left + k].first);

max\_leaf = max(max\_leaf, dp[left + k + 1][right].first);

int sum = dp[left][left + k].first \* dp[left + k + 1][right].first;

sum += dp[left][left + k].second + dp[left + k + 1][right].second;

min\_sum = min(min\_sum, sum);

}

dp[left][right].first = max\_leaf;

dp[left][right].second = min\_sum;

}

}

}

return dp[0][n - 1].second;

}

### Another Solution

There is an time O(N) solution, for any number which is smaller than its left and its right, we can choose the minimum product to build the bottom level sub tree, after the sub tree is build we delete the value and only keep its neighbors. We keep the process until with only one item left.

/// <summary>

/// Leet code #1130. Minimum Cost Tree From Leaf Values

/// </summary>

int LeetCode::mctFromLeafValuesII(vector<int>& arr)

{

size\_t n = arr.size();

vector<int> dp;

int result = 0;

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

{

while ((dp.size() > 1) &&(dp[dp.size() - 1] <= dp[dp.size() - 2]) &&

(dp[dp.size() - 1] <= arr[i]))

{

result += dp[dp.size() - 1] \* min(dp[dp.size() - 2], arr[i]);

dp.pop\_back();

}

dp.push\_back(arr[i]);

}

for (int i = 0; i < (int)dp.size() - 1; i++)

{

result += dp[i] \* dp[i + 1];

}

return result;

}