<https://leetcode.com/problems/burst-balloons/>

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] \* nums[i] \* nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return *the maximum coins you can collect by bursting the balloons wisely*.

**Example 1:**

Input: nums = [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

**Example 2:**

Input: nums = [1,5]

Output: 10

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* 0 <= nums[i] <= 100

**Attempt 1: 2023-09-03**

**Solution 1:  DFS + Memoization (360 min)**

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

// Append 1 at both head and tail based on original array

int[] arr = new int[n + 2];

arr[0] = 1;

arr[n + 1] = 1;

for(int i = 1; i <= n; i++) {

arr[i] = nums[i - 1];

}

Integer[][] memo = new Integer[n + 2][n + 2];

return helper(arr, 0, n + 1, memo);

}

private int helper(int[] arr, int left, int right, Integer[][] memo) {

if(left + 1 == right) {

return 0;

}

if(memo[left][right] != null) {

return memo[left][right];

}

int result = 0;

for(int i = left + 1; i < right; i++) {

result = Math.max(result, arr[left] \* arr[i] \* arr[right] + helper(arr, left, i, memo) + helper(arr, i, right, memo));

}

return memo[left][right] = result;

}

}

Time Complexity : O(N^3)

Space Complexity : O(N^2)

**Solution 2:  DP (360 min)**

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

// Append 1 at both head and tail based on original array

int[] arr = new int[n + 2];

arr[0] = 1;

arr[n + 1] = 1;

for(int i = 1; i <= n; i++) {

arr[i] = nums[i - 1];

}

int[][] dp = new int[n + 2][n + 2];

// First for window size from 1 to n

for(int window = 1; window <= n; window++) {

// Then for left pointer from 1 to (n-window+1)

for(int left = 1; left <= n - window + 1; left++) {

int right = left + window - 1;

// Now right is already fixed i.e., (left+window-1),

// so now third loop from left to right. Each time

// update the value in dp for left to right.

for(int i = left; i <= right; i++) {

dp[left][right] = Math.max(dp[left][right], arr[left - 1] \* arr[i] \* arr[right + 1] + dp[left][i - 1] + dp[i + 1][right]);

}

}

}

return dp[1][n];

}

}

Time Complexity : O(N^3)

Space Complexity : O(N^2)

**JAVA | DP | Divide and Conquer | Sliding Window | Detailed Explanation Using Image**

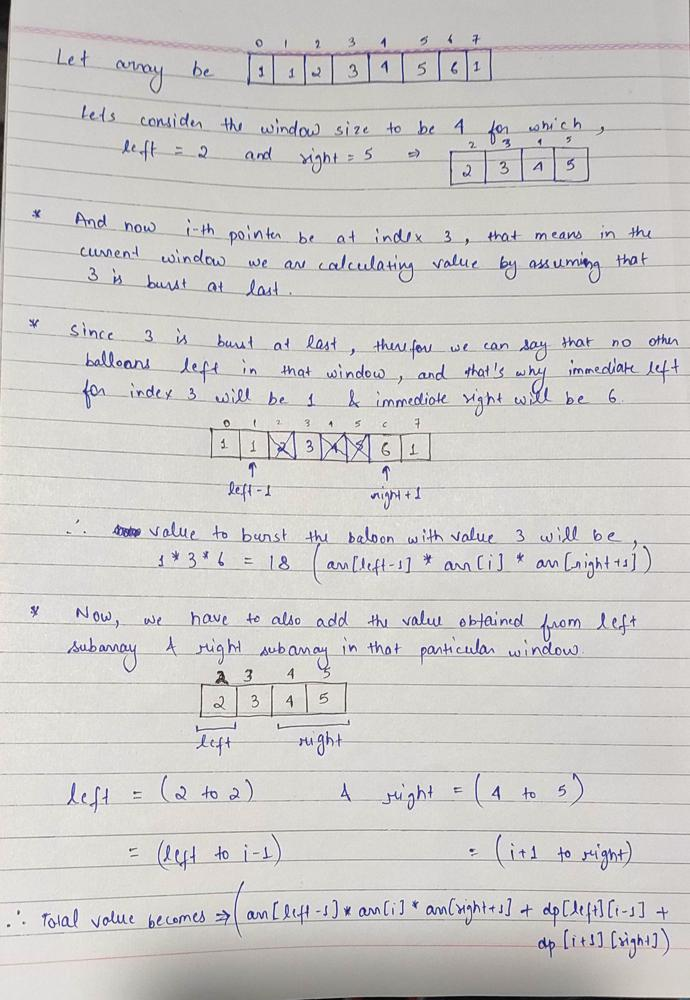
**Refer to**

<https://leetcode.com/problems/burst-balloons/solutions/1659162/java-dp-divide-and-conquer-sliding-window-detailed-explanation-using-image/>

**Intuition:** Suppose you have been given array [1,2,3,4] which u can see as [1,1,2,3,4,1] (padded by 1 on both side). So now if u decide to burst balloon with value 3 at last then that means all other balloons will already burst and so total value to burst the balloons will be = (1 \* 3 \* 1 + left + right) where left will be max cost of bursting balloons left to 3 i.e., [1,2] and right will be max cost of bursting balloons t the right of 3 i.e., [4].So here we can clearly observe that to get the answer for window of size n we need to have the answer for window of size 2(left) and 1(right) that is smaller subproblems. So here we can definitely think of DP.(Divide and Conquer DP)

**Explanation of Approach:**

* Since we need smaller window answer for larger windows, so we have to store the answer for window of each size from 1 to n.
* Then we will have two pointers left and right to point at the two ends of our current window. For example if we have array [1,1,2,3,4,5,1] and we have to get the answer for subarray [2,3,4] then left will point at index 2and right at index 4.
* Now in the current window we have to burst balloons in such sequence that we get the max value. And for this we have to check for each balloon in that window whether it can give the max value if burst at last.
* So for this we have to traverse from left to right in the window and each time calculate the value assuming ith balloon is burst at last.
* So while filling Dp we will be filling values for left to right window , i.e.,
* dp[left][right] = Max(already calculated value, burst this ith balloon last and add left and right subarray points within the window)
* dp[left][right] = max(dp[left][right], arr[left-1] \* arr[i] \* arr[right+1] + dp[left][i-1] + dp[i+1][right])



So now we just need to apply 3 loops:

1. 1st for window size from 1 to n
2. then for left pointer from 1 to (n-window+1)
3. Now right is already fixed i.e., (left+window-1), so now third loop from left to right.
4. Each time update the value in dp for left to right.

Time Complexity : O(N^3)

❌**Approach 0: Recursion (Here we will only understand why to not do using recursion)**

**For that Lets first see the power of DP over recursion:** if we do it by recursion then we have to make N! calls to cover all the permutations of N sized array and then to calculate points for each permutation we need O(N). So by recursion Time Complexity will be O(N!N)So now if N=20,then N! = 2.43 \* 10^18 => N!N = 4.86 \* 10^19and if we assume we can do 10^8 computations in 1sec then the above value of N!N goanna take approx. 3168 years to compute!! LOL!!

And at the same time for DP the Time Complexity is N^3 = 8000 < 1sec

##### **From 3168 years to less than 1sec => I think this is more than enough to understand the power of DP.**

**Approach 1 : Memoization**

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

int arr[] = new int[n+2];

arr[0] = arr[n+1] = 1;

for(int i=1;i<=n;i++){

arr[i] = nums[i-1];

}

int memo[][] = new int[n+2][n+2];

return burst(memo, arr, 0, n + 1);

}

public int burst(int[][] memo, int[] nums, int left, int right) {

if (left + 1 == right) return 0;

if (memo[left][right] > 0) return memo[left][right];

int ans = 0;

for (int i = left + 1; i < right; ++i){

ans = Math.max(ans, nums[left] \* nums[i] \* nums[right]

+ burst(memo, nums, left, i) + burst(memo, nums, i, right));

}

memo[left][right] = ans;

return ans;

}

}

**Approach2 : DP**

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

int arr[] = new int[n+2];

arr[0] = arr[n+1] = 1; // Giving padding of 1 to the corner elements

for(int i=1;i<=n;i++){

arr[i] = nums[i-1]; //final padded array

}

int dp[][] = new int[n+2][n+2];

for(int window = 1;window<=n;window++){ // window size

for(int left = 1;left<=n-window+1;left++){ // left pointer

int right = left+window-1; // right pointer

for(int i=left;i<=right;i++){ // iterate from left to right

dp[left][right] = Math.max(dp[left][right], (arr[left-1]\*arr[i]\*arr[right+1]) + dp[left][i-1] + dp[i+1][right]);

}

}

}

return dp[1][n];

}

}

**Refer to**

<https://grandyang.com/leetcode/312/>

这道题提出了一种打气球的游戏，每个气球都对应着一个数字，每次打爆一个气球，得到的金币数是被打爆的气球的数字和其两边的气球上的数字相乘，如果旁边没有气球了，则按1算，以此类推，求能得到的最多金币数。参见题目中给的例子，题意并不难理解。那么大家拿到题后，总是会习惯的先去想一下暴力破解法吧，这道题的暴力搜索将相当的复杂，因为每打爆一个气球，断开的地方又重新挨上，所有剩下的气球又要重新遍历，这使得分治法不能 work，整个的时间复杂度会相当的高，不要指望可以通过 OJ。而对于像这种求极值问题，一般都要考虑用动态规划 Dynamic Programming 来做，维护一个二维动态数组 dp，其中 dp[i][j] 表示打爆区间 [i,j] 中的所有气球能得到的最多金币。题目中说明了边界情况，当气球周围没有气球的时候，旁边的数字按1算，这样可以在原数组两边各填充一个1，方便于计算。这道题的最难点就是找状态转移方程，还是从定义式来看，假如区间只有一个数，比如 dp[i][i]，那么计算起来就很简单，直接乘以周围两个数字即可更新。如果区间里有两个数字，就要算两次了，先打破第一个再打破了第二个，或者先打破第二个再打破第一个，比较两种情况，其中较大值就是该区间的 dp 值。假如区间有三个数呢，比如 dp[1][3]，怎么更新呢？如果先打破第一个，剩下两个怎么办呢，难道还要分别再遍历算一下吗？这样跟暴力搜索的方法有啥区别呢，还要 dp 数组有啥意思。所谓的状态转移，就是假设已知了其他状态，来推导现在的状态，现在是想知道 dp[1][3] 的值，那么如果先打破了气球1，剩下了气球2和3，若之前已经计算了 dp[2][3] 的话，就可以使用其来更新 dp[1][3] 了，就是打破气球1的得分加上 dp[2][3]。那假如先打破气球2呢，只要之前计算了 dp[1][1] 和 dp[3][3]，那么三者加起来就可以更新 dp[1][3]。同理，先打破气球3，就用其得分加上 dp[1][2] 来更新 dp[1][3]。说到这里，是不是感觉豁然开朗了 ^.^

那么对于有很多数的区间 [i, j]，如何来更新呢？现在是想知道 dp[i][j] 的值，这个区间可能比较大，但是如果知道了所有的小区间的 dp 值，然后聚沙成塔，逐步的就能推出大区间的 dp 值了。还是要遍历这个区间内的每个气球，就用k来遍历吧，k在区间 [i, j] 中，假如第k个气球最后被打爆，那么此时区间 [i, j] 被分成了三部分，[i, k-1]，[k]，和 [k+1, j]，只要之前更新过了 [i, k-1] 和 [k+1, j] 这两个子区间的 dp 值，可以直接用 dp[i][k-1] 和 dp[k+1][j]，那么最后被打爆的第k个气球的得分该怎么算呢，你可能会下意识的说，就乘以周围两个气球被 nums[k-1] \* nums[k] \* nums[k+1]，但其实这样是错误的，为啥呢？dp[i][k-1] 的意义是什么呢，是打爆区间 [i, k-1] 内所有的气球后的最大得分，此时第 k-1 个气球已经不能用了，同理，第 k+1 个气球也不能用了，相当于区间 [i, j] 中除了第k个气球，其他的已经爆了，那么周围的气球只能是第 i-1 个，和第 j+1 个了，所以得分应为 nums[i-1] \* nums[k] \* nums[j+1]，分析到这里，状态转移方程应该已经跃然纸上了吧，如下所示：

dp[i][j] = max(dp[i][j], nums[i - 1] \* nums[k] \* nums[j + 1] + dp[i][k - 1] + dp[k + 1][j])                 ( i ≤ k ≤ j )

有了状态转移方程了，就可以写代码，下面就遇到本题的第二大难点了，区间的遍历顺序。一般来说，遍历所有子区间的顺序都是i从0到n，然后j从i到n，然后得到的 [i, j] 就是子区间。但是这道题用这种遍历顺序就不对，在前面的分析中已经说了，这里需要先更新完所有的小区间，然后才能去更新大区间，而用这种一般的遍历子区间的顺序，会在更新完所有小区间之前就更新了大区间，从而不一定能算出正确的dp值，比如拿题目中的那个例子 [3, 1, 5, 8] 来说，一般的遍历顺序是：

[3] -> [3, 1] -> [3, 1, 5] -> [3, 1, 5, 8] -> [1] -> [1, 5] -> [1, 5, 8] -> [5] -> [5, 8] -> [8]

显然不是我们需要的遍历顺序，正确的顺序应该是先遍历完所有长度为1的区间，再是长度为2的区间，再依次累加长度，直到最后才遍历整个区间：

[3] -> [1] -> [5] -> [8] -> [3, 1] -> [1, 5] -> [5, 8] -> [3, 1, 5] -> [1, 5, 8] -> [3, 1, 5, 8]

这里其实只是更新了 dp 数组的右上三角区域，最终要返回的值存在 dp[1][n] 中，其中n是两端添加1之前数组 nums 的个数。参见代码如下：

解法一：

class Solution {

public:

int maxCoins(vector<int>& nums) {

int n = nums.size();

nums.insert(nums.begin(), 1);

nums.push\_back(1);

vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));

for (int len = 1; len <= n; ++len) {

for (int i = 1; i <= n - len + 1; ++i) {

int j = i + len - 1;

for (int k = i; k <= j; ++k) {

dp[i][j] = max(dp[i][j], nums[i - 1] \* nums[k] \* nums[j + 1] + dp[i][k - 1] + dp[k + 1][j]);

}

}

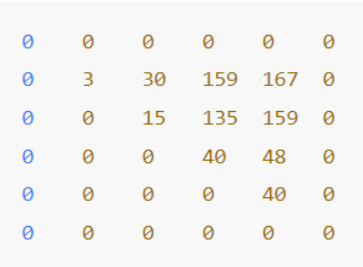
}

return dp[1][n];

}

};

对于题目中的例子[3, 1, 5, 8]，得到的dp数组如下：



0 0 0 0 0 0

0 3 30 159 167 0

0 0 15 135 159 0

0 0 0 40 48 0

0 0 0 0 40 0

0 0 0 0 0 0

这题还有递归解法，思路都一样，就是写法略有不同，参见代码如下：

解法二：

class Solution {

public:

int maxCoins(vector<int>& nums) {

int n = nums.size();

nums.insert(nums.begin(), 1);

nums.push\_back(1);

vector<vector<int>> dp(n + 2, vector<int>(n + 2, 0));

return burst(nums, dp, 1 , n);

}

int burst(vector<int>& nums, vector<vector<int>>& dp, int i, int j) {

if (i > j) return 0;

if (dp[i][j] > 0) return dp[i][j];

int res = 0;

for (int k = i; k <= j; ++k) {

res = max(res, nums[i - 1] \* nums[k] \* nums[j + 1] + burst(nums, dp, i, k - 1) + burst(nums, dp, k + 1, j));

}

dp[i][j] = res;

return res;

}

};