# 题目

给你一个整数数组 nums ，你可以对它进行一些操作。

每次操作中，选择任意一个 nums[i] ，删除它并获得 nums[i] 的点数。之后，你必须删除 所有 等于 nums[i] - 1 和 nums[i] + 1 的元素。

开始你拥有 0 个点数。返回你能通过这些操作获得的最大点数。

示例 1：

输入：nums = [3,4,2]

输出：6

解释：

删除 4 获得 4 个点数，因此 3 也被删除。

之后，删除 2 获得 2 个点数。总共获得 6 个点数。

示例 2：

输入：nums = [2,2,3,3,3,4]

输出：9

解释：

删除 3 获得 3 个点数，接着要删除两个 2 和 4 。

之后，再次删除 3 获得 3 个点数，再次删除 3 获得 3 个点数。

总共获得 9 个点数。

提示：

1 <= nums.length <= 2 \* 104

1 <= nums[i] <= 104

# 分析

要解决这个问题，我们可以将其转化为类似“打家劫舍”的动态规划问题。核心思路是通过统计每个数字的总点数，再利用动态规划计算不相邻数字的最大点数和。

解题思路

1、统计数字总点数：

- 由于删除一个数字x后，所有x-1和x+1的数字都会被删除，因此选择x就不能选择x-1和x+1。这与“打家劫舍”中“不能抢劫相邻房屋”的约束类似。

- 首先统计每个数字x的总点数（即x出现的次数乘以x），存储在sum数组中，其中sum[x]表示数字x的总点数。

2、动态规划求解：

- 定义dp[i]表示处理到数字i时能获得的最大点数。

- 状态转移方程：

- 若选择数字i，则不能选择i-1，因此dp[i] = dp[i-2] + sum[i]。

- 若不选择数字i，则最大点数等于处理到`i-1`的最大点数，即dp[i] = dp[i-1]。

- 综上，dp[i] = max(dp[i-1], dp[i-2] + sum[i])。

- 初始状态：dp[0] = sum[0]（数字0的总点数），dp[1] = max(sum[0], sum[1])（数字0和1中取最大）。

代码：

class Solution {

public:

int deleteAndEarn(vector<int>& nums) {

if (nums.empty()) return 0;

// 找到数组中的最大值，确定sum数组的大小

int max\_num = \*max\_element(nums.begin(), nums.end());

vector<int> sum(max\_num + 1, 0);

// 统计每个数字的总点数（出现次数 × 数字值）

for (int num : nums) {

sum[num] += num;

}

// 动态规划计算最大点数

if (max\_num == 0) return sum[0]; // 只有数字0的情况

vector<int> dp(max\_num + 1, 0);

dp[0] = sum[0];

dp[1] = max(sum[0], sum[1]);

for (int i = 2; i <= max\_num; ++i) {

dp[i] = max(dp[i-1], dp[i-2] + sum[i]);

}

return dp[max\_num];

}

};

代码解释：

1、统计总点数：

首先找到数组中的最大数字max\_num，创建sum数组记录每个数字的总点数（例如，若nums = [2,2,3,3,3,4]，则sum[2] = 4，sum[3] = 9，sum[4] = 4）。

2、动态规划计算：

- 初始化dp[0]和dp[1]，分别处理只有数字0和数字0、1的情况。

- 对于i >= 2的数字，根据“选择i或不选择i”的两种情况，通过状态转移方程计算dp[i]，最终dp[max\_num]即为最大点数。

复杂度分析

- 时间复杂度：O(n + max\_num)，其中n是nums的长度，max\_num是nums中的最大数字。统计总点数耗时O(n)，动态规划遍历耗时O(max\_num)。

- 空间复杂度：O(max\_num)，用于存储sum数组和dp数组。

该方法通过转化问题模型，高效利用动态规划求解，适用于题目中nums长度和数字范围的约束。