**Overview**

This problem is an excellent example of why it is important to read the problem description carefully. We can make a substantial optimization by noticing that the order of the elements in nums is irrelevant.

At any operation, we pick a number, let's say x, and delete **all** occurrences of x - 1 and x + 1. That means if x occurs multiple times in nums and we take one, we may as well take all of them because after deleting all x - 1 and x + 1, we can only stand to gain more points by taking additional x.

Before we start, we can simplify nums by collecting all duplicate numbers together. As an example, nums can be represented as a hash map with numbers as keys which map to the number of times the key occurs in nums. [2, 2, 3, 3, 3, 4] would be converted to {2: 2, 3: 3, 4: 1}. Furthermore, since we are only really concerned about how many points each number can give us, we can multiply the keys and values to represent how many points taking the key will give us. In this case, we would have {2: 4, 3: 9, 4: 4}.

**In the first approach of this article, we will talk about an algorithm that solves this problem in detail. In the second approach, we will implement the same algorithm in a slightly more efficient way.**

Then, we will look at some approaches that build on the idea from the first two approaches, but each one will have differences in their time and space complexities.

**Approach 1: Top-Down Dynamic Programming**

**Intuition**

There are two characteristics of this problem that hint towards the use of dynamic programming (DP). The first is that the problem is asking us to find the **maximum** of something. The second is that we need to make decisions on which numbers to take, and each decision may influence future decisions. For example, if we wanted to take all the fives, then we can no longer take the fours or sixes.

As described in the [Dynamic Programming Explore Card](https://leetcode.com/explore/learn/card/dynamic-programming/), these two characteristics suggest that we should consider a DP solution. If you're new to DP, then reading this card is highly recommended.

The difficult part about this problem is figuring out how to always make the optimal decisions on which elements to take. Let's look at an example to illustrate the problem: nums = [5, 4, 5, 4, 3, 5, 3]. As we mentioned above, we can convert this into a format that's easier to use. We have two 3, two 4, and three 5. That means:

* If we take all the 3, we get 3 \* 2 = 6 points.
* If we take all the 4, we get 4 \* 2 = 8 points.
* If we take all the 5, we get 5 \* 3 = 15 points.

If we tried moving from smaller keys to larger ones in a greedy manner, our first decision would be to take all 3 or all 4. We can only choose one because picking one deletes the other one. Well, we get 8 points if we pick 4 and 6 points if we pick 3, so perhaps we should greedily pick 4. Unfortunately, this prevents us from picking 5, which gives us way more points.

Perhaps a different greedy strategy would work. How about sorting the keys by how many points they give, and then greedily trying to pick the most points? That way, we would take the 15 points here immediately. Well, what if we had an example nums = [5, 5, 5, 6, 6, 6, 6, 7, 7]? We would have:

* 5 --> 15 points.
* 6 --> 24 points.
* 7 --> 14 points.

Again, this greedy strategy won't work. Those 24 points are tempting, but by taking them, we delete 29 points. In these small examples, it's easy to see what the correct decisions are. But for larger test cases, the question can get out of hand quickly. We need to ensure that we always make the correct decision.

This is where DP comes in. We'll skip over the underlying details of DP, so the article stays focused on the problem - but if DP is relatively new to you, then you will likely find the DP explore card to be a very helpful resource.

We can formulate DP algorithms in 3 easy steps. First, we need some sort of memory that stores the answer to our question. Because we're doing top-down, we'll use a hash table for memory and a recursive function. Let's declare a function maxPoints. We want maxPoints(num) to return the maximum points that we can gain if we only consider all the elements in nums with values between 0 and num.

The second thing we need is a recurrence relation, a way to move between states. Let's say that we are currently at some arbitrary number x, where x is in nums one or more times. How can we find maxPoints(x)? When it comes to x, we have to make a choice: **take, or don't take.**

* If we take x, then we gain points equal to x times the number of times x occurs in nums - we can pre-compute these values. For now, let's call this value gain. However, because of the deletion, by taking x, we are no longer allowed to take x - 1. The largest number that we can still consider is x - 2. Therefore, if we choose to take x, then the most points that we can have here is gain + maxPoints(x - 2), where gain is how many points we gain from taking x and maxPoints(x - 2) is the maximum number of points we can obtain from the numbers between x - 2 and 0.
* If we choose not to take x, then we don't gain any points here, but we still may have accumulated some points from numbers smaller than x. Because we didn't take x, we did not close the door to x - 1. In this case, the most points we can have here is maxPoints(x - 1).

This forms our recurrence relation: for an arbitrary x, maxPoints(x) = max(maxPoints(x - 1), maxPoints(x - 2) + gain), where gain is the number of points we can gain from taking x.

The problem is, even though we figured out how to find maxPoints(x), how do we find maxPoints(x - 1) and maxPoints(x - 2)? That would involve finding maxPoints(x - 3) and maxPoints(x - 4) and so on.

The third component of a dynamic programming solution is base cases. Typically, we can find base cases with a little bit of logical thinking. First, maxPoints(0) will always be equal to 0. Second, when considering maxPoints(1), we only care about the elements 0 and 1. We do not care about 2 because of how we defined maxPoints(x). Looking at the recurrence relation, we know that if we arrived at 1, it means that we must not have taken 2, and because 1 times any quantity will be greater than or equal to the number of points we can get from taking 0, we should always take 1 (if there are any).

With these base cases, we can find maxPoints(2). With maxPoints(2) calculated, we can find maxPoints(3), all the way up to maxPoints(max(nums)). Remember, we defined maxPoints(x) as the maximum points we can gain when we consider the numbers from 0 to x, so maxPoints(max(nums)) covers the entire input, and stores the answer to the original problem.

Because each call to maxPoints will create 2 extra calls (the recurrence relation looks for maxPoints(num - 1) and maxPoints(num - 2)), we would end up with an exponential amount of function calls. To avoid this, we will **memoize** our function. When we find the answer for a certain number for the first time, we will store this answer. Then in the future, we can refer to this value instead of repeating computation.