**HASH TABLE**

1. [**https://leetcode.com/problems/find-the-number-of-winning-players/**](https://leetcode.com/problems/find-the-number-of-winning-players/)

**A screenshot of a computer

Description automatically generated**

**#Approach**

First, I count the wins by converting the `pick` list, which contains pairs of player numbers and their respective wins, into tuples and passing them to a `Counter` object. This step tallies the number of wins for each player.

Next, I filtered the winning players using a set comprehension. For each player in the `Counter`, I have checked if the number of wins (`count`) is greater than the player's number. If it is, I have then added that player to the set.

Finally, I returned the length of the set, which represents the number of players who have won more matches than their player number. This set comprehension effectively filters and counts only those players who meet the winning condition.

I used Python's `Counter` and set comprehensions, I made the method concise and efficient for calculating the desired count.

2.<https://leetcode.com/problems/design-neighbor-sum-service/description/>

A screenshot of a computer program

Description automatically generated

The `neighborSum` class is designed to handle a grid of integers and provides methods to compute sums based on the position of a given value. Upon initialization, the class constructs a dictionary mapping each grid value to its coordinates, which allows for efficient lookups.

Specifically, the `\_\_init\_\_` method initializes the grid, computes its size, and fills a dictionary (`value\_to\_position`) with the positions of each value. This setup requires iterating over each cell in the grid, resulting in a time complexity of \(O(n^2)\), where \(n\) is the grid's dimension, and a space complexity of \(O(n^2)\) for storing the position information.

The `adjacentSum` method calculates the sum of values from cells adjacent to the cell containing the specified value. It checks the cells directly above, below, to the left, and to the right of the target cell while ensuring that these cells are within the grid boundaries. This method performs a constant number of operations regardless of grid size, yielding a time complexity of \(O(1)\) and using minimal additional space, so its space complexity is \(O(1)\).

Similarly, the `diagonalSum` method computes the sum of values from cells that are diagonally adjacent to the cell containing the specified value. It checks the cells diagonally positioned at top-left, top-right, bottom-left, and bottom-right, while also ensuring boundary conditions. Like `adjacentSum`, this method operates in constant time, with a time complexity of \(O(1)\) and a space complexity of \(O(1)\), as it also uses a fixed amount of extra space.

3.<https://leetcode.com/problems/distribute-candies/>

A screenshot of a computer

Description automatically generated

To solve the problem of distributing candies optimally, I focused on maximizing the variety of candy types Alice can consume while ensuring she only eats half of the total candies. The approach involves first calculating the maximum number of candies Alice can eat, which is n/2 where n is the total number of candies. Next, I identified the number of distinct candy types using a set, which gives the count of unique types available. Finally, I compared the two values—n/2 and the number of unique candy types—and chose the smaller one. This ensures that Alice eats the maximum variety of candies without exceeding her limit.

4. <https://leetcode.com/problems/two-sum-iv-input-is-a-bst/>

A screenshot of a computer program

Description automatically generated

**#Approach**

To find out if there are two distinct nodes in a binary tree that add up to a given number `k`, I use a hash set to keep track of the values I've seen. I start by doing a breadth-first search (BFS) with a queue. For each node, I calculate what value I need to find to reach `k` (which is `k - node.val`). I then check if this needed value is already in my hash set. If it is, it means I've already seen a node with that value, and together they sum up to `k`, so I return `True`. If not, I add the current node's value to the hash set and continue with the BFS. If I finish going through the tree without finding such a pair, I return `False`. This method is efficient because it uses a hash set to quickly check for the needed values.