**1 - Stone Game**

Alex and Lee play a game with piles of stones.  There are an even number of piles **arranged in a row**, and each pile has a positive integer number of stones piles[i].

The objective of the game is to end with the most stones.  The total number of stones is odd, so there are no ties.

Alex and Lee take turns, with Alex starting first.  Each turn, a player takes the entire pile of stones from either the beginning or the end of the row.  This continues until there are no more piles left, at which point the person with the most stones wins.

Assuming Alex and Lee play optimally, return True if and only if Alex wins the game.

**Example 1:**

**Input:** [5,3,4,5]

**Output:** true

**Explanation:**

Alex starts first, and can only take the first 5 or the last 5.

Say he takes the first 5, so that the row becomes [3, 4, 5].

If Lee takes 3, then the board is [4, 5], and Alex takes 5 to win with 10 points.

If Lee takes the last 5, then the board is [3, 4], and Alex takes 4 to win with 9 points.

This demonstrated that taking the first 5 was a winning move for Alex, so we return true.

**Note:**

1. 2 <= piles.length <= 500
2. piles.length is even.
3. 1 <= piles[i] <= 500
4. sum(piles) is odd.

Dynamic programming will be the way to go while solving this problem. The idea behind this is to divide the piles into n sizes, where n is the number of piles. This will be used to store all possible values that the choices will make, so when we come up against a smaller instance of the problem, or in other words, when we have to solve a problem that we have already solved, we can simply access that result and not to extra work. By filling this table, we can then find all instances of the game inside.

Let’s look at this pile: [3 9 1 2]

We can make the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 3,0 | 9,3 | 4,9 | 11,4 |
| 2 |  | 9,0 | 9,1 | 10,2 |
| 3 |  |  | 1,0 | 2,1 |
| 4 |  |  |  | 2,0 |

Were the rows and columns represent each player, and the first number represents the player who chooses first and the second number the player who chooses second. Were each time we need to find an answer that has already been solve we look at the table to access it.

Dynamic Programming always reminds me of the edit distance table, were we solve every instance of the problem and store it in a similar table as the one above. Much like that table, all already solved outcomes are accessed to determine the next value. This was my IDEAL way to solve this. I thought of a similar problem, such as the edit distance, and tried to implement the same idea with the new rules and parameters.

**2 - Minimum Falling Path Sum**

Given a **square** array of integers A, we want the **minimum** sum of a *falling path* through A.

A falling path starts at any element in the first row, and chooses one element from each row.  The next row's choice must be in a column that is different from the previous row's column by at most one.

**Example 1:**

**Input:** [[1,2,3],[4,5,6],[7,8,9]]

**Output:** 12

**Explanation:**

The possible falling paths are:

* [1,4,7], [1,4,8], [1,5,7], [1,5,8], [1,5,9]
* [2,4,7], [2,4,8], [2,5,7], [2,5,8], [2,5,9], [2,6,8], [2,6,9]
* [3,5,7], [3,5,8], [3,5,9], [3,6,8], [3,6,9]

The falling path with the smallest sum is [1,4,7], so the answer is 12.

**Note:**

1. 1 <= A.length == A[0].length <= 100
2. -100 <= A[i][j] <= 100

For this problem, a dynamic programming method, like in all problems will be implemented. The idea behind this, is that a recursively, the algorithm would repeat previously solved instances of the problems over and over, much like the towers of Hanoi implementation. So, to solve this, a simpler solution will be done.

The idea is to implement a sort of shortest path algorithm, where the last row will be intact, and it will try to find the smallest “path” in the array by selecting the smallest number in the row above with a different column by at least one.

Let’s use the example above to demonstrate this idea:

[1 2 3] [1 2 3] [12 14 15]

[4 5 6] [11 12 14] [11 12 14]

[7 8 9] [ 7 8 9] [7 8 9]

To explain, the method will start at the last row because the values will not change. From there it will look for the smallest value one row above and within the same column or an adjacent one. Let’s say we start at 7, from there we can either go to 4 or 5, choosing 4 because it’s the smallest value, hence a new matrix with this new value will be made so we now that the smallest possible way to get to the 4, is from 7. We repeat this idea for all possible paths and finding the smallest path possible. Finally, the first row will have all possible shortest solutions to get to that number. With the smallest number in that row being the Minimum “Rising” path.

The way IDEAL and Duke 7 were implemented were the following:

First, we had to understand the problem, because after my first time reading it, I was a bit confused. When I finally understood this, I defined the problem as a “shortest path” problem. I drew the rows as a matrix and then I started to try to solve this. Seeing how my Data Structures class just covered graphs, I thought of the shortest path implementation to solve this problem. What I learned from this problem is that, like in all problems, trying to see the problem in a simpler way, will make it easier to understand and easier to find a solution.

**3 - 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:**

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

To solve this, backtracking will be used. The idea is to read the string from the first letter to the last and determining if adding one word at a time will make a palindrome. Also, if the string is of length 3, just check if the first and last characters are the same. If you think about this, all you need to do is to check if the current substring is a palindrome, if it is, add 1 to the counter, and so on until we reach the end of the string and our pointer is out of bounds.

Let’s look at the example “abacc”:

What the algorithm will do is start at the first “a”. Then since a single letter is a palindrome, add one to the counter, then it will go into “ab”, since this is not a palindrome, continue. Then “aba” since this is a palindrome, add 1 to the counter. It will do this for every character in the string, and returning the counter, which in this case will be 8.

The way IDEAL and Duke 7 were implemented were the following:

This problem reminded me of a lab assignment I had previously done where, given a word, you would need to return all accepted anagrams, meaning they were actual words. To do this, a recursive function was made were all anagrams of a word were generated and then it would check if they were in a dictionary of accepted words, returning only the words that were in the dictionary. This problem is in some ways similar, were the difference is the efficiency, where the original problem added extra work that backtracking omits. Using this, I came up with my solution.

**6 - Maximum Length of Pair Chain**

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

**Example 1:**

**Input:** [[1,2], [2,3], [3,4]]

**Output:** 2

**Explanation:** The longest chain is [1,2] -> [3,4]

**Note:**

1. The number of given pairs will be in the range [1, 1000].

One way to approach this is to first sort all the pairs in the array, this is an easier way for the algorithm to work because some chains may be skipped if not sorted. Then, we add the first pair to the chain and just compare the first element of the next pair with the second element of the current pair, if the second is greater, add the next pair to the chain. Then, if the second pair follows the same rule, add the third pair, otherwise, replace the second pair with the third pair if the rule applies. Finally, when all pairs have been visited and the chain is done, return the length of the chain to get the maximum chain possible.

Let’s look at the array = [{12, 14}, {23, 29}, {18, 41}, {30,34}]

First we sort the array to obtain : [{12, 14}, {18, 41}, {23, 29}, {30,34}]

Now we start at the first pair and add it to the chain:

C = (12, 14)

Next we check to see if the rule applies: 14 < 18? Yes, so we add the pair to the chain.

Now we check the next pair: 41 < 23? No, so we replace (18, 41) with (23, 29).

Finally, we check the last pair: 29 < 30? Yes, so we add it to the chain.

C = (12, 24), (23, 29), (30, 34)

Return the length of C which is 3 and the longest chain possible.

The way IDEAL and Duke 7 were implemented were the following:

To identify the problem, I looked at it in a simpler way, which was by first sorting the array. This made it easier for the solution to be identified, as a sorted array would give out the maximum chain possible. Personally, I can’t say if this is a DP or another type of algorithm. But there is a simple idea behind it that should be easy to understand. What I learned from this problem is that sometimes they are easier than they may seem if we just think them through as a computer scientist.

**9 - Perfect Squares**

Given a positive integer *n*, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to *n*.

**Example 1:**

**Input:** *n* = 12

**Output:** 3

**Explanation:** 12 = 4 + 4 + 4.

**Example 2:**

**Input:** *n* = 13

**Output:** 2

**Explanation:** 13 = 4 + 9.

A simple but not efficient way to do this is with a simple recursive function. The idea is to find the minimum number of squares whose sum adds up to the given number. The base case would be if n is greater than 3 because 1-3 would output the same input. We would also set the maximum number of squares to be the number itself since this can be accomplished by just adding 1 n times to reach our goal.

Then we would just go with a loop that determines if the current index squared is greater than the input, if it is, ignore it. Otherwise, if a given index squared is less than our goal, subtract the result from the goal and increase the count by 1. Do this until your goal is reached.

Let’s use the example n = 13.

Our algorithm would execute since it’s greater than 3. Then it would enter the loop starting at 1 and determine if 1^2 is smaller than 13 and subtract it. It will do this recursively and return the minimum of all the possible results. Which in this case would be 9 + 4 = 13, so our output would be 2.

The way IDEAL and Duke 7 were implemented were the following:

I had originally done this problem for another class. I remember that I was a beginner for recursion and had a lot of trouble with this one. So, I tried to analyze the problem and break it into simpler things. First, I had to understand my goal, to find the minimum number of squares. Then I had to understand that I could only do this by using squared numbers, such as 1,4,9,16 and so forth. I then realized that I could find the first squared number that was smaller than the goal and subtract it and keep doing that until I reached 0. Then all I had to add was the min() function to automatically find the smallest number and return that.