My username for leetcode is benschwartz33 and for hackerrank it is bschwart

The climbing the leaderboard problem looks at multiple scores from an arcade game compared against the current leaderboard scores. The player is tracking their scores as they improve and this algorithm is meant to give the players ranking for each of their scores.

An example case could be a player is playing a game with leaderboard scores of [100, 95, 95, 90, 90, 75, 70, 65]. For equivalent scores such as the two scores of 95 and 90, they share a ranking. So 95 gives you second place and 90 gives you third place. Now let the players scores be [15, 25, 50, 65, 95, 100]. The rankings for this player's scores are [7, 7, 7, 6, 2, 1].

My algorithm for constructing the list of rankings is fairly simple. I simply parse through the leaderboard scores until I find a score that is higher than the players. Once we find that score the player must be ranked one lower. In order to make the coding easier, I converted the list of rankings to a set and back to a list to remove any duplicates since they will share the same ranking. I then sorted the list in ascending order. Additionally, since the list of player's scores is given in ascending order we can skip any part of the ranked list that we have already looked at.

So for the earlier example [100, 95, 95, 90, 90, 75, 70, 65] becomes [65, 70, 75, 90, 95, 100]. The ranking for each index i is 6-i. Now to find the rankings for the scores [15, 25, 50, 65, 95, 100] we parse through the leaderboard list for each of the player's scores. 15, 25, and 50 are all less than 65 so the algorithm will stop after the first index and give each of those a ranking of 7. 65 matches the last spot in the leaderboard so we move on and look at 70 which is larger than 65. So we give the next score a ranking of 6. Now looking at 95, we parse through starting at 70 and stop at 100 because it is the first element that is larger than 95. So that score is ranked one below at second place. Lastly looking at 100, we start at 95 and since there is no element on the leaderboard that is larger than 100 we give that a ranking of 1. This gives us the ranking of [7, 7, 7, 6, 2, 1] that we saw earlier.

In this pseudocode ranked represents the list of leaderboard scores and player represents the list of scores from the player. ranks is the list I use to store the ranks of each of the players scores.

```
def climbingLeaderboard(ranked, player):
  remove the duplicates from ranked and sort in ascending order
  initialize k to 0
  initialize ranks to an empty list

  iterate through player over the variable i:
    iterate through ranked from index k to the end over the variable j:
      if the element of ranked is greater than the element of player:
        append the rank to ranks (the length of ranked-j+1)
        set k to j
```

```
            break from the loop
        if we reach the end of the list append 1 to ranks


    return ranks
```

This algorithm runs in O(n+m) where n is the size of ranked and m is the size of player. For ranked we visit each element at most 2 times because each time through the loop we start at the element we were last at. We visit each element in player exactly one time. So the big O run time is O(n) + O(m) = O(n+m)

Something from the course that helped me come up with this solution was from first learning about recursion. In the textbook they discuss saving previously visited results to avoid repeatedly computing the same results over and over. My first solution to this problem simply iterated through the entire ranked list each time to find the rank of the current score. While I wasn't directly thinking about the example from the textbook, it did occur to me that it would be much more efficient to skip any rank that I had already looked at which is the type of thinking that this course tought me to do.

The next problem is called Goodland Electricity. We are looking at a country called Goodland and trying to get electricity to every city that is spaced evenly along a line. The goal is to find the minimum number of powerplants for a given range of the power plants and a given set of cities, some of which can contain power plants and some of which cannot.

The input is given as an integer k that gives the range of the power plants and a list containing 1s and 0s. A 1 represents a city that can contain a power plant and a 0 represents a city that cannot. A city is in range of the power plant if the distance from the plant is less than k. For example, if we have k=2 and cities [0,0,1,0,0] and we place a power plant at index 2 it can reach the two neighboring cities but cannot reach the cities at index 0 or 4 because they are two spaces away and we need distance<k. For this example, it is not possible to place powerplants such that every city has electricity.

Let's look at a more comprehensive example. Let k=3 and cities = [1,1,0,1,1,0,1,0,1,1,1]. We can place power plants at indexes [0, 3, 6, 9] and every city will have electricity. However, this is not optimal because [1, 4, 8] also gets electricity to every city using one less power plant.

A greedy algorithm can be used to determine the least number of power plants possible. First, we pick the furthest city to the right that can have a powerplant in the first k-1 cities. From there we continue to pick the furthest city to the right within the 2k cities following the last powerplant we placed. That way we are always choosing the farthest city that still keeps every previous city in range. If there is ever a range without any suitable cities we know that there is no solution so we immediately quit and return -1.

Looking at the previous example k=3 cities=[1,1,0,1,1,0,1,0,1,1,1] we first pick the furthest city within the first three which gives us 1. Then we pick the city within the next 6 which are [0,1,1,0,1,0]. So we place a power plant at city 6. Finally, there are 4 cities left and city 10 is suitable so we put a power plant there. Now the first power plant covers cities 0-3, the second power plant covers cities 4-8 and the last power plant covers cities 8-10. So the algorithm returns that 3 is the minimum number of power plants required to get electricity to every city.

```
def pylons(k, arr):
  initialize position to 0
  initialize count to 0

  while position is less than the length of arr:
    initialize i to position-k+1(set to 0 if negative)
    initialize j to position+k
    while i is less than j and the length of arr:
      set position to i if the ith city is a 1
      increment i

    return -1 if a suitable city was not fount

    increment count

    add k to position
  return count
```

The algorithm does not look at any elements that come before a powerplant that has already been placed. It is possible to reference a following city a second time because it checks every city within the range. But each city can be checked at most two times because the algorithm will either find a city that is after it or it will fail and quit. If there was a suitable city before it already would have been chosen by the algorithm. So, since we are visiting each city at most 2 times we get a run time of O(n).

This problem is very similar to the problem we did in class to find the optimal placement for telephone poles on a street. Doing this problem, along with all the other practice coming up with greedy algorithms helped my immensely in solving this problem.

The next algorithm find the longest palindrome substring of a given string. A palindrome is a string in which the characters in the first half of the string directly mirror the characters in the second half. So if you were to reverse the string it would be exactly the same.

For example, the string “aabbaa” is a palindrome. The string “ababb” is not a palindrome but the substring “aba” is.

In order to check if a string is a palindrome, we can move from the center outward checking each pair of characters to see if they are equal. If we reach the end then it is a palindrome and if there is ever a pair of characters that is not equal we can stop. In order to find the palindromic substrings, we can do this for each character in the string. For each character, we step outwards until we reach a pair of characters that don’t match. If the generated substring is longer than the previous longest we save it and the length. Then once we reach the last character in the string we can return the current saved palindrome.

For “ababb” we start at the first character ‘a’ and move outwards. b is not equal to null so a is saved as the longest palindrome and we move to ‘b’. From b we move outward one space where ‘a’=’a’ then one more space where it fails and since the length of “aba” is more than “a” we save “aba” as the longest palindrome. Moving out from the next character ‘a’ we end up with the palindrome “bab” which is the same length as the previous longest “aba”. Since the algorithm only replaces if the new palindrome is longer we keep “aba” and move to the next character. The next ‘b’ only produces a 1 character string because its neighbors are a and b. Finally the last ‘b’ also gives a 1 character string. So at the end we have “aba” saved and the algorithm returns it as the longest palindromic substring.

```
class Solution(object):
  def longestPalindrome(self, s):
    initialize longest to an empty string
    initialize length to 0

    for each character in s:
      continuously step 1 step left and 1 step right from the character until non-matching character are reached
      if the length of the produced palindrome is longer than length set longest to the new string and length to its legnth

                  repeat the above steps starting from the current character on the left and the following character on the right to take into account strings in which the longest palindrome has even length

    return longest
```

This algorithm has a worst case run time of O(n^2) where n is the size of the string. We are iterating through each character in the string and for each character there are up to n/2 possible comparisons. So we get O(n*(n/2)) = O(n^2)

My original thought for this problem was to use dynamic programming. I realized that I would need to check over and over whether or not something was a palindrome. The most efficient way I knew to do that was to go from the outside characters and work in toward the center continuously checking if the characters were equal. I also realized that it would work just as well to go from the middle outward. It seemed much easier to iterate through the list saving the largest palindrome as we went, a technique we have used before. Additionally, this problem reminds me of the divide and conquer algorithm to find the maximum sum subarray where you have to check the subarray between the two lists by moving outward from the center and saving the maximum array as you go.

The next problem is searching a rotated sorted array. A rotated array is one in which the elements where rotated around a certain index. So we start with a sorted array e.g. [1,2,3,4,5,6] then let's rotate it around index 2. Every element after index 2 is moved in front to give us [4,5,6,1,2,3].

The algorithm I used to search the array is a binary search using the divide and conquer strategy. First, the algorithm finds the midpoint of the array. If the desired element is larger than the midpoint we search the second half of the array. If the desired element is smaller than the midpoint we search the first half of the array. If the midpoint is equal to the desired element then we have found it and we can return that index. For the rotated array we need to split the array into two halves at the rotation index so we have two separate sorted arrays. Then the binary search is done on both arrays.

So if we are looking for the number 3 from this array [4,5,6,1,2,3] we do a binary search on [4,5,6] and [1,2,3]. In the first array we start at 5 which is larger than 3 so we move to the left and check 4. We have nowhere left to check so we know that 3 is not in this list. We then check the second list. The midpoint is 2 which is smaller than 3 so we check the right half. The next element we look at is 3 which is the element we are looking for. The binary search would return 2 as the index so we add the length of the first array to get the index in the original array. So the algorithm will find that 3 is at index 5 in the original array.

```
class Solution(object):
  def search(self, nums, target):


    for each element in nums:
      if this element is smaller than the previous one break the loop and call this index i

    initialize left as nums indexed from 0 to i
    initialize right as nums indexed from i to the end

    def binarySearch(arr,x):
      find the midpoint of arr and call it mid
```

if the element at mid is x return mid
if x is larger than the element at mid search the right half of arr
if x is smaller than the element at mid search the left half of arr

if we get to the end and x was not found return -1

conduct a binary search on left for target
conduct a binary search on right for targer

if target was found in right add i to the index

return the index found by one of the binary searches or -1 if target was not found

The binary search function is cutting the input size in half each time through so it is O(logn). The loop to find the rotation point could run through the entire array so it is O(n). So this algorithm is O(n) + 2O(logn) = O(n).

It's pretty easy to see what idea from the course helped me solve this problem. The binary search is something that have worked out a few times in practicing divide and conquer algorithms so it came very naturally to use it here. When I saw how the array was sorted I immediately thought to split it into two sorted arrays so I could use the binary search.

The next problem is to create all the letter combinations possible from a given string of digits 2-9. On a telephone the numbers are mapped to certain letters. 2 is mapped to abc 3 is mapped to def and so on. This algorithm prints out every possible combinations of letters for a given number.

Take the number 56. 5 is mapped to jkl and 6 is mapped to mno. So every possible combination of letters is [jm, jn, jo, km, kn, ko, lm, ln, lo].

The algorithm to produce this result uses recursive backtracking. It works very similar to how the problem is solved by hand. In the above example I put j with each possible letter from 6, then k, then l. So the algorithm takes a digit, and starting with the first mapped letter it moves one recursive call deeper and computes every possible combinations of letters following. Then it moves to the next character in the current digit. So it will continue to recurse until it reaches the last digit, add each character, move up to the previous digit, go to the second character for that digit, and recurse back to the last digit. This traverses through the possibilities the same way a depth first search traverses through a graph. It goes to the farthest possible point, backs up, and continues.

Let's walk through the algorithm on the example 456. We start from 4 and the first letter is g so add that to the current string. Then move to 5 where the first character is j. Then move to 6 where the first

character is m. We have reached the end of the characters so we add the string “gjm” to a list. Now we add the rest of the characters of 6 to “gj” so we get “gjn” and “gjo”. Now our list is [“gjm”, “gjn”, “gjo”] and we have exhausted the characters mapped to 6. So we backtrack to 5 and go to the next character. We are still using g but now the second character is k. Then we go back to 6 and add each character to the string. So our list is [“gjm”, “gjn”, “gjo”, “gkm”, “gkn”, “gko”]. Then we move to the last character in 5 and do the same thing giving [“gjm”, “gjn”, “gjo”, “gkm”, “gkn”, “gko”, “glm”, “gln”, “glo”]. Now that every character in 5 has been exhausted we backtrack to 4 and move to the next character. We repeat this process until the last character in 4 is complete. And that gives us every possible letter combination.

```
class Solution(object):
  def letterCombinations(self, digits):
    if digits is an empty string return an empty list

    create a list that stores the letters mapped to each number and call it mapping
    rinitialize result to an empty list

    def recurse(result, digits, string, index, mapping):
      if index has reached the end of digits append string to result and return

      initialize aMap as the letters mapped to digit at index

      for each character in aMap:
        add the character to string and recurse to index+1


    call recurse with result, digit, an empty string, index=0, and mapping

    return result
```

Since there are two numbers that are mapped to 4 letters the upper bound for the number of iterations through the loop is 4. And since the input for the recursive call increases by 1 until it reaches the length of digit we get

T(n) = 4T(n-1) + 1
= 1 + 4(1+ 4(T(n-2)))
= 1 + 4 + 4(1 + 4(T(n-3)))
= 1 + 4 + 4 + … 4(1+ 4(T(n-n))) = 4^n

So this algorithm is O(4^n) where n is the length of digits.

There are a few ideas from the course that go into this solution. Obviously recursion is the main technique here. Although, I think the main idea that I was thinking about was the depth first search. I had been practicing graph algorithms for exam 2. While I was working a letter combination solution by hand I noticed that what I was doing was going character by character for each digit until I reached the end and then backing up by 1 digit each time I exhausted one. So it made a lot of sense to implement it that way and using this recursive algorithm was the way to do it.

The last problem is the kingdom division problem. It states that a king is trying to divide his kingdom between his two children. One child will always attack the other if any of their cities are isolated, meaning that city is not connected to one owned by the same child. The question is hwo many ways can the kingdom be divided between the two children such that neither one will attack?

Lets say there are three cities. Since no city can be isolated all three must be given to the same child. So there are two possible solutions. Now lets say we have 4 cities. Let city 1 connect to cities 2 and 3 and city 3 connects to city 4. In this case you can give all 4 cities to one child. Or they can be split so one child gets cities 1 and 2 and the other child gets 3 and 4. In both cases there are 2 possible ways to do it so there are 4 total possible solutions.

The algorithm to solve this uses dynamic programming. First we put the cities in a graph where each node corresponds to a city and each edge is a road between cities. Let the color of the node represent what child the city is given to. At each node there are two possible cases. The first case is that the node is the same color as it's parent. In this case, the children of this node can belong to either child so the number of ways to divide the kingdom at this node is the sum of the number of ways to divide the kingdom for each combination of colors of the children. That is
f(node, same) = sum(f(child1, s(1))*f(child2,s(2))*...*f(childn,s(n))) where each s(i) represents whether that child has the same color as node. The next case is when the node has a different color from its parent. In this case there must be at least one child that has the same color as node so
f(node, different) = f(node, same) -(f(child1, different)*f(child2, different)*…*f(childn, different))
The base case is once we reach a node with no children. In that case it works if it is the same color as its parent, and fails if it is different. So if node has no children f(node, same) = 1 and f(node, different) = 0. Root node has no parent so we calculate it as if it were different than it's parent, since it must have at least one child of the same color. The algorithm then does a depth first search on the graph to calculate each node. And for each possible way to divide the country there are two ways to split it between the two children so the solutions is 2*f(root)

Lets look at the 4 city example from earlier. City 2 has no children so f(2, same) = 1 f(2, different) = 0. City 4 also has no children so f(4, same) = 1 f(4, different) = 0. f(3, same) = f(4,same) + f(4,different) = 1. f(3,different) = 1 - f(4,different) = 1
f(1) = f(2,s)*f(3,s) + f(2,s)*f(3,d) + f(2,d)*f(3,s) + f(2,d)*f(3,d) = 1 + 1 + 0 + 0 = 2
solution = 2f(1) = 4 which matches the intuitive solution.

I was unable to complete an actual implementation of this algorithm so unfortunately I will need to omit the pseudo code.

The depth first search of the tree will reach every node 1 time. At each node we are calculating every possible combination of same and different colored children. So there will be O(2^m) operations where m is the number of children on that node. So the average run time of the algorithm should be O(n*2^m) where n is the number of cities and m is the average number of children that each node has. The upper bound for the run time will be O(n*2^n).

The ideas from this course that go into the algorithm are dynamic programming and graph algorithms. The dynamic programming in this problem definitely feels different from ones we have done in the past, but the idea is the same of coming up with a recurrence relation to solve smaller problems first and using those solution to solve the larger one. Then since the data can be represented as a tree it makes sense to use a depth first search to visit every node.