**CS 1290**

**Algorithmic Thinking in Problem Solving**

**Exam 2**

**Due: November 27, 2019, 11:59pm**

**Instructions**

Solve **5** of the following problems using dynamic programming (you **must** use dynamic programming to earn points. Other solutions will not be accepted).

For each problem, you must do the following:

1. Define the problem/solution recursively (this is the most important step – if you don’t do this correctly, you will automatically **get a 0** for the problem regardless of the code you write).
2. **Briefly** talk about how you plan to store solutions to sub-problems and combining them to solve the global problem (talk about the data structure/variables you’ll use to solve the problem).
3. Talk about how you used **IDEAL** and **Duke 7** to tackle the problem
4. Code your solution.

**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.

1. This problem should only return true if Alex wins, because of this, only one counter is needed to keep track of the difference between Alex’s points and Lee’s points, if by the end of the game the counter is positive then Alex wins and it returns true, otherwise it returns false. The recursive property of this game lie in the way the state changes; every time a pile of stones gets picked the current state containing the possible options change, each time a pile of rocks gets picked, the current optimal play should be calculated.
2. There is no need to have more data structures for this problem, it can be solved with just the input array. During Alex’s turn we obtain the maximum and add it to the score, and during Lee’s turn we obtain the maximum and subtract it from the score. This strategy is not actually optimal since it’s just a greedy pick instead of an adversarial search like minimax.
3. **Identify**. First I had to break down this problem into multiple subproblems, what is the return condition, how do you get points, how do you lose points, how does each turn play, and what does each action do to change the state of the game.  
   **Define**. The only case where the return statement is true is when Alex wins, any other case returns false.  
   **Explore**. There are many approaches to this problem, we could have a counter for each player that contains their points and compare them at the end; it could be a single counter that indicates who is winning.  
   **Duke’s seven steps:   
   Work an example.** I worked the example [5,3,4,5].  
   **Write down what you did.** Alex takes 5, the piles change to [3,4,5] and the counter is at 5. Lee takes 5, the piles change to [3,4] and the counter is at 0. Alex takes 4, the piles change to [3] and the counter is at 4. Lee takes 3 and the counter is at 1.  
   **Generalize.** There is only one counter that keeps track of the difference between Alex and Lee, both players take the max they can.

**3 - Palindromic Substrings**

1. The recursive property of this problem comes from all the substrings that can be made from a string, since a substring can be made from any number of sequential characters in the string then there is a need to check every substring individually; this would require to check the substrings that have already been checked before.
2. One way this can be done is by comparing the substring it to its inverse by comparing each character and adding one to the palindrome counter. This would be done in a nxn Boolean matrix.
3. **Identify**. What are the challenges posed by the substrings, how many substrings can be obtained from a string, how do you use the data from the previous iteration in the current iteration, what data structures should be used?   
   **Define**. Check the correct number of substrings. How to know if a string is a palindrome, return the correct number of palindromes in a string.   
   **Explore**. It could use a stack to compare each substring to itself. It could use the string as an array and compare the first half to the second half in opposite order.  
   **Duke’s seven steps:   
   Work an example.** I worked the example “aaa”.  
   **Write down what you did.** there are 6 substrings: a, a, a, aa, aa, aaa. All of them are palindromes  **Generalize.** The number of substrings in a string is based on the length of the string, the number of palindromes cannot exceed n\*(n+1)/2.

**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.

1. For any pair there can be n number of pairs that can chain to it, this means that it is necessary to check through every pair in the array before chaining one to find the optimal pair. Once that is done that process has to be repeated for the next chain.
2. This solution could use an array to store the length of the longest chain from each starting point, once it is done checking all the possibilities it should return the max of the values stored in the array.
3. **Identify**. How is a chain made, what needs to be stored and where?   
   **Define**. Return the longest chain possible.  
   **Explore**. This could be solved by first sorting the array based on the first value, and then using a counter to check the length of the chain for the indexes with the lowest value.  
   **Duke’s seven steps:   
   Work an example.** I worked the example [[1,2], [2,3], [3,4]]  
   **Write down what you did.** The longest chain is of size 2, the chain is [1,2], [3,4]. [2,3] cannot connect because 2 is not less than 2, the same can be said for the 3 in [2,3] with [3,4] **Generalize.** The longest possible chain is of size n while the shortest possible chain is of size 1 if n>0.

**7 - Integer Break**

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

1. The parts of n can be broken into smaller pieces, and those could also have factors, each of those has a maximum product too so each of those multiplies all the other maximum products to obtain the total maximum product.
2. This could be stored using an array of size n+1, the index represents the factor, and the stored value represents the maximum product for that factor. After storing each solution to the factors then the maximum product of n can be calculated by using the stored values.
3. **Identify**. How to obtain the values that add to that number. How to obtain the maximum product of a number using those values. How to use the stored values to calculate the next values.   
   **Define**. Return the maximum possible value obtained from multiplying the factors of a number n.  
   **Explore**. Instead of using an array it could use a HashTable with a number being the key and the value being the maximum possible product of that number.  
   **Duke’s seven steps:   
   Work an example.** I worked the example n=8.  
   **Write down what you did.**  {0,1,1,2,4,6,9,12, 16} 2 breaks into 1+1, 1\*1 = 1. 3 breaks into 2+1, 2\*1 = 2, 4 into 2+2, 2\*2 = 4, 5 into 3+2, 3\*2 = 6, 6 into 3+3, 3\*3 = 9, 8 into 4+4, 4\*4 = 16.   
   **Generalize.** It is only necessary to break the number n/2 times to find the maximum possible product because of the commutative property of addition.

**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*.

1. N could be composed of multiple perfect squares, after subtracting a value from n the resulting value is also composed of perfect squares, this happens for every number until the result is equal to 0.
2. This could be stored using an array of size n+1, the index represents the value and the stored value represents the minimum number of perfect squares required to sum to that index.
3. **Identify**. How to obtain the minimum product of a number using previously calculated values.   
   **Define**. Return the minimum possible number of perfect squares required to sum to a number n.  
   **Explore**. Instead of using an array it could use a HashTable with a number being the key and the value being the minimum number of perfect squares required to add to that number.  
   **Duke’s seven steps:   
   Work an example.** I worked the example n=13.  
   **Write down what you did.**  {0, 1, 4, 9} 9+4= 13 so the result is 2.