Keenan
CSCI 261 HW4

1. LongestKindOfIncreasingSubsequence
    a. DP Parts:
        i. **What**: Each possible subsequence contains at least three numbers (as opposed to the standard longest-increasing-subsequence problem in which each subsequence contains at least two numbers). Because of this, `OPT[i][j]` represents the longest possible increasing subsequence within the input that ends with indices i and j (j being the last value).
        ii. **How**: `OPT[i][j]` can be increased to `OPT[k][i]+1` whenever a new k is found such that the average of `input[k]` and `input[i]` is less than `input[j]` and `OPT[i][j]` is less than or equal to `OPT[k][i]`.
        iii. **Where**: The solution to the original problem can be found by iterating over the `OPT` 2-d array in O(n^2) time and selecting the maximum value.
    b. Pseudocode

```
init OPT[inputLength][inputLength]
init result = 0
for i from 0 to inputLength
    for j from i+1 to inputLength
        OPT[i][j] = 2
        for k from 0 to i
            avg = (numbers[k] + numbers[i])/2
            if avg < numbers[j] and OPT[i][j] <= OPT[k][i]
                OPT[i][j] = OPT[k][i] + 1
        if OPT[i][j] > result
            result = OPT[i][j]
return result
```

    c. Time complexity: O(n^3)
    d. This algorithm has complexity n^3 because it is a simple triple-nested for loop where each loop iterates at most n times.
2. KnapSack
    a. DP Parts:
        i. **What**: `OPT[j][v]` represents the highest possible value knapsack that can be made with the first j items and has weight exactly equal to v. Additionally, and probably not necessarily, `WeightOPT[j][v]` represents the weight of the knapsack that is represented by `OPT[j][v]`.
        ii. **How**: `OPT[j][v]` is assigned the maximum of the total cost of `OPT[j-1][v-(weight of j-th item)] + the cost of the j-th item` and the cost of `OPT[j-1][v]`, and only if the total weight of the knapsack is equal to v (otherwise it gets assigned 0).
        iii. **Where**: The return value for the original problem is within the range of weights in the OPT array between the minimum and maximum weights with all of the items included. Loop through and take the maximum cost.

To determine the items that belong in the solution, step through the OPT array from the optimal solution by first stepping through the items-included axis until the cost decreases. When it does, include that item and decrease the weight-axis by the weight of the included item. Once you have stepped through each item this way, return the list of included items.

b. Pseudocode

```
// OPT array (keeps track of optimal costs)
init OPT[n+1][maxWeight+1] each value at 0

// WeightOPT (keeps track of the weights for each cost in OPT)
init WeightOPT[n+1][maxWeight+1] each value at 0

for j from 0 to n
    for v from 1 to maxWeight
        // if we're only looking at one item
        if j == 0
            if weights[j] == v
                OPT[j][v] = costs[j]
                WeightOPT[j][v] = weights[j]
        else
            // if this item doesn't fit by itself
            if weights[j] > v
                OPT[j][v] = OPT[j-1][v]
                WeightOPT[j][v] = WeightOPT[j-1][v]
                continue;
            // calculate next OPT entry(s)
            temp = -1
            if WeightOPT[j-1][v - weights[j]] + weights[j] == v
                temp = OPT[j-1][v-weights[j] + costs[j];
            if temp >= OPT[j-1][v]
                OPT[j][v] = temp
                WeightOPT[j][v] = WeightOPT[j-1][v-weights[j] + weights[j]
            else
                OPT[j][v] = OPT[j-1][v]
                WeightOPT[j][v] = WeightOPT[j-1][v]

// find the best legal result (keep track of its indices)
for i from 0 to OPT.length
    for j from 0 to OPT[i].length
        if OPT[i][j] > result and j >= minWeight
            result = OPT[i][j]
            resultI = i
            resultJ = j
print result
```

```
// based off of the found indices (above), track down the items in the result
init ids[]
init idsIterator = 0
while resultI > 0
        if OPT[resultI-1][resultJ] != OPT[resultI][resultJ]
                ids[idsIterator++] = resultI + 1
                resultJ -= weights[resultI]
        resultI--
print ids (from last index to first)
```

c. Time complexity: O(nW_2) where W_2 is maxWeight
d. This algorithm consists of two double-nested for loops (each outer loop iterating n times, and each inner loop iterating at most maxWeight (W_2) times) which each have complexity O(nW_2) and a single while loop that iterates at most n times (and is therefore negligible).

3. AllWhiteSquare
    a. DP Parts:
        i. **What**: `OPT[i][j]` represents the height (or width) of the largest white square in the given matrix that the `[i][j]` input is the bottom-right corner of. (0 if it is black)
        ii. **How**: `OPT[i][j]` is assigned the minimum value of the item above it `(OPT[i-1][j])`, the item to the left of it `(OPT[i][j-1])`, and the item in the top-left corner from it `(OPT[i-1][j-1])`.
        iii. **Where**: The solution to the original problem will have to be found either by looping through the OPT matrix after filling it in or by keeping track of the maximum value while filling it in. It does not have a guaranteed location. (in my algorithm it is kept track of while filling it in)
    b. Pseudocode

```
init OPT[][] = input // OPT[i][j] is 1 for 'w' and 0 for 'b'

for i from 0 to OPT.length
        for j from 0 to OPT.length
                if OPT[i][j] != 0 and i > 0 and j > 0
                        OPT[i][j] = min(OPT[i-1][j-1], OPT[i-1][j], OPT[i][j-1])+1)
                        if OPT[i][j] > result
                                result = OPT[i][j]
return result;
```

    c. This algorithm runs with O(n^2) time complexity.
    d. This algorithm is a simple double-nested for loop where each loop iterates at most n times.

4. MatrixChain
    a. DP Parts

i. **What**: `OPT[i][j]` represents the best possible computational cost to multiply all matrices from the i-th matrix to the j-th matrix.

ii. **How**: For each possible split point between matrix i and j, `OPT[i][j]` is assigned the sum of calculations for the left side, the right side, and the multiplication of the two sides together, but only if that sum is less than the current value of `OPT[i][j]`.

iii. **Where**: The solution to the original problem will always be in the `OPT[1][n]` location because we are looking for the solution that involves multiplying all matrices from 1 to n. The actual solution including parenthesized matrices can be determined through divide and conquer by dividing the original list of matrices down to its base cases and building upwards using the split-points kept track of in the `traceBack` array. (The `traceBack[i][j]` array is created and maintained while filling in the actual OPT array and contains the best split-points for each set of two matrices, i and j)

b. Pseudocode

```
init OPT[input.length][input.length]
init traceBack[input.length][input.length]

for d from 0 to OPT.length
   for i from 1 to OPT.length
      j = i + d
      if i == j
         OPT[i][j] = 0
      else
         OPT[i][j] = MAX_VALUE
         for k from i to j - 1
            int temp = OPT[i][k] + OPT[k+1][j] + input[i-1]*input[k]*input[input[j]
            if OPT[i][j] > temp
               OPT[i][j] = temp

               // used later in solution
               // (k is the split-point between the given matrices)
               traceBack[i][j] = k

// best computational cost of multiplying all the given matrices
print( OPT[1][OPT.length - 1] )

// print out parenthesized matrix multiplication for best result
writeSolution(1, input.length)

String writeSolution(int l, int r) {
   if l == r
      return "A" + l
   if l + 1 == r
```

```
        return "( A" + l + " x A" + r + " )"

    return "( " + writeSolution(l, traceBack[l][r]) + " x " +
                  writeSolution(traceBack[l][r] + 1, r) + " )"
}
```

c. This algorithm runs with O(n^3) time complexity.
d. This algorithm has a simple triple-nested for loop where each loop iterates at most n times. Following (for printing the solution) is a recursive function that concatenates strings in O(n) time (because it uses divide and conquer and has a recurrence of T(n) = 2T(n/2) + c, when n > 2; using the simplified master theorem, case 1 is true, equating T(n) to O(n).). Overall, the recursive function is negligible leaving the time complexity at O(n^3).