Gavin Browning - A01887359
CS- 5050
9/16/18

Assignment 3: Solve four problems using DP
1) Name of Problem: **Maximum value robber problem**

Problem statement: Given **n** items each with a **weight[i]** and a positive value **v[i]**, a bag to store the objects and maximum weight limit **L** (the heaviest the robber can carry), find the subset of objects that is less than or equal to **L** that is worth the maximum cumulative value.

Derive Recursive Function:
First simplify the problem to determine the maximum value that can be obtained, not the actual objects.

Write: What are the variables that describe all problems? *(hint: think of how the number of objects available and the weight limit will change as the robber puts objects in the bag)* weight[i]: Want this to be small.
V[i]: Want this to be large.
L: Max weight limit. n: number of items
Write: What is the type of the solution?
Array: A list of items that maximizes value, less than or equal to L
Math: Write the function declaration:
int[] maxValue(int n, int L, int[] weights, int[] values)
n

Think: What are the simplest problems? What are their solutions?

Math: define the base cases:
if n == 0: Return []
if L <= 0: Return []

Think: How can a larger problem be broken into smaller problems? (Think simply, but creatively)
Math: Write down the simpler problems in terms of the larger problem description
Given n items, can all n items fit in the bag L?
What subset has the highest value.

Think: How is the solution to the input problem constructed from the solutions to the smaller problems?
Math: Write down the solution construction step

Return the highest value, that can fit in bag L.

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

1) int [] maxValue (int n, int L, int[] weights, int[] values)

2) n == 0: return []
   L <= 0: return []

3) sum (weights)<= L
   for each i in n{
      results[] += max value (without i) }
   return best result

Convert to a Dynamic Program:

Review the existing recursive algorithm definition

Review: What are the arguments to the recursive function? These will be the indexes into the cache. int n, int L, weights, values

Review: What is the return type of the function? This will be the type that is stored in the cache. int[ ]

Think: Are all the arguments integers or are they of mixed type? If mixed, maybe a dictionary or hash table may be best. Will all possible solutions be generated, or only a few specific problems? If all, then an array may be best, if only a few, then maybe a dictionary or hash table should be used.

Write the declaration of the cache data structure. This will be defined once, as the first statement of the new DP function:

cache{ }

Think: How can the base cases in the recursive solution be stored in the cache? If the test checks that all of the arguments are a specific value, then this will become a single assignment into the cache. If the test checks only some of the arguments, or tests for ranges of argument values, then loops will be needed that enumerate all possible input argument values that will pass the test. For each simple problem instance generated, make an assignment into the cache, storing the value returned by the base case.

Code: Write the loops/assignments that fill in the cache with the base cases

cache [[ ]] = [ ]

Review: the recursive calls to understand how the main problem is made simpler and what smaller solutions are needed to compute the main solution.

Think: How do the function arguments change in the recursive calls? Do they always go smaller? What order should the DP loop through the values of the function

parameters, such that the smaller problems are always computed before the larger problems?

Write: the code that scans through the cache in the correct order. Here the loops will iterate over the values to the function arguments from smallest value (excluding the base cases) to the largest values. If the function takes more than one argument, nested loops will be needed.

```
cache.get(int n, int L, int[] weights, int[] values)
for key, value in cache:
    # do something
```

Review the code that calls the smaller problem instances recursively and computes the main solution.

Code: Rewrite this recursive code by replacing subproblem function calls with cache access and returns with cache assignments. This code will be the body of the code inside the nested for loops. Note, that the names of the variables in the recursive code will have to be changed to the iterative variables of the DP loops.

```
if n == 0: return []
if L <= 0: return []
    cache.get(int n, int L, int[] weights, int[] values)
```

Code: add the return statement at the end that returns the solution to the original problem by indexing into the cache

```
for each i in n {
        results[] += maxValue(without i)}
cache[(best result)]
return best result
```

Write the Trace Back routine:
This will traverse through the solution cache identifying the objects that the robber should grab.

Think: What are the details of the problem we need, and how they are used in the code?
Write: Identity the data structure to store the solution details. This could be a list of objects or values.

A hash of objects with their related values

Write: the traceback function definition with arguments the same as the recursive definition and the return type as the list of solution components.

int, int[] def maxValue (n, L, weights, values)

Study: the base cases of the recursive algorithm
Write: the same base cases for the traceback routine, this time returning the solution component for each case (often empty).

```
n == 0: return []
L <= 0: return []
```

Study: the problem decomposition + solution construction code of the DP algorithm and identify where the sub-solutions are located in the array, relative to the current problem.
Code: copy over the problem decomposition + solution construction code of the DP into the traceback routine. Modify the code so that the subsolution that was used to create the solution is recorded. Let this subsolution be S0.

```
while n>=0    numpy.perm(weights)
{for combo in numpy.perm(weights)
    if combo Not in Cache: ~~add combo~~.
        Cache.add(combo)
    if cache at combo < value of current combo
        replace the value.
```

Code: that makes up the list of components/solution elements to return. This will often be S0 appended to a recursive call to the traceback routine function. The arguments of the recursive call will be the smaller problem that created S0.

Combine the code into one recursive function and write it below:

```
def maxValue ( L, weights, values):
    Cache={ }
    Cache [[ ]] =∅
    if len(weights) == ∅:
        return ∅
    if L <= ∅:
        return ∅

    for Combo in numpy.Perm (weights):
        Value = getValue(Combo)
        if Combo not in Cache:
            Cache.add( Combo, Value)
        elif Cache[combo] < Value
            Cache [Combo] = Value
    max_value = 0, best_combo = 0
    for Combo, Value in Cache:
        if Value > max_value:
            max_value = Value
            best_combo = Combo

    return max_value
    return best - Combo
```

2) Name of Problem: **Best way to cut up a felled tree**

Problem statement: Given (a) a tree that has been felled and its side branches cut off so that only the trunk remains. The length of this trunk is **L** meters. (b) A table of dollar values for various cut lengths of the trunk, where the cut lengths are integers ranging from **1** to **L** (for example, a cut piece of length 1 meter is worth $2, a cut piece of length 2 is worth $10, etc.). Find the cuts that should be made to maximize the value of the trunk.

Derive Recursive Function:
First simplify the problem so that only the maximum value possible is computed (not the actual cuts).

Write: What are the variables that describe all problems? *(hint: think of how the length of the trunk will change as cuts are made)*
table of dollar values
Cut lengths, L
Write: What is the type of the solution?
Max value of the trunk
int
Math: Write the function declaration:

```
def cutTree(values, L)
```

Think: What are the simplest problems? What are their solutions?

Math: define the base cases:

```
if (L <= 0):
    return 0
```

Think: How can a larger problem be broken into smaller problems? (Think simply, but creatively)
Math: Write down the simpler problems in terms of the larger problem description
Cut the tree into different pieces
Compare the different cuts & values.

Think: How is the solution to the input problem constructed from the solutions to the smaller problems?
Math: Write down the solution construction step
return max value of possible cuts.

Finish Code: Put it all together
1) Function declaration:
2) Base cases:
3) Combine problem decomposition with solution construction:

```
1) def cutTree(values, L)
2)     if (L<=0): return 0
3)     for i in range(0, L)
           maxValue = max(maxValue, value[i] + cutTree(value, L-i-1
       return maxValue
```

Convert to a Dynamic Program:
Review the existing recursive algorithm definition
Review: What are the arguments to the recursive function? These will be the indexes into the cache.
Review: What is the return type of the function? This will be the type that is stored in the cache.

```
int
```

Think: Are all the arguments integers or are they of mixed type? If mixed, maybe a dictionary or hash table may be best. Will all possible solutions be generated, or only a few specific problems? If all, then an array may be best, if only a few, then maybe a dictionary or hash table should be used.
Write the declaration of the cache data structure. This will be defined once, as the first statement of the new DP function:

```
cache[]
```

Think: How can the base cases in the recursive solution be stored in the cache? If the test checks that all of the arguments are a specific value, then this will become a single assignment into the cache. If the test checks only some of the arguments, or tests for ranges of argument values, then loops will be needed that enumerate all possible input argument values that will pass the test. For each simple problem instance generated, make an assignment into the cache, storing the value returned by the base case.
Code: Write the loops/assignments that fill in the cache with the base cases

```
cache[0] = 0
```

Review: the recursive calls to understand how the main problem is made simpler and what smaller solutions are needed to compute the main solution.
Think: How do the function arguments change in the recursive calls? Do they always go smaller? What order should the DP loop through the values of the function

parameters, such that the smaller problems are always computed before the larger problems?

Write: the code that scans through the cache in the correct order. Here the loops will iterate over the values to the function arguments from smallest value (excluding the base cases) to the largest values. If the function takes more than one argument, nested loops will be needed.

```
for i in range(1, L+1):
    maxValue = INT_MIN
    for j in range(i):
        maxValue = max(maxValue, value[j] + cache[i-j-1])
    cache[i] = MaxValue
return cache[L]
```

Review the code that calls the smaller problem instances recursively and computes the main solution.

Code: Rewrite this recursive code by replacing subproblem function calls with cache access and returns with cache assignments. This code will be the body of the code inside the nested for loops. Note, that the names of the variables in the recursive code will have to be changed to the iterative variables of the DP loops.

```
for i ...
    maxValue = INT_MIN
    for j ...
        maxValue = max(maxValue, value[j] + cache[i-j-1])
    cache[i] = maxValue
```

Code: add the return statement at the end that returns the solution to the original problem by indexing into the cache

```
return cache[L]
```

Write the Trace Back routine:
This will traverse through the solution cache identifying the best list of choices the player can make. Let left be represented as True and right represented as False. (list of cut sizes)

Think: What are the details of the problem we need, and how they are used in the code?
Write: Identity the data structure to store the solution details. This could be a list of objects or values.

```
list of values
```

Write: the traceback function definition with arguments the same as the recursive definition and the return type as the list of solution components.

```
int[] def cutTree(values, L)
```

Study: the base cases of the recursive algorithm
Write: the same base cases for the traceback routine, this time returning the solution component for each case (often empty).

```
if L <= 0
    return 0
```

Study: the problem decomposition + solution construction code of the DP algorithm and identify where the sub-solutions are located in the array, relative to the current problem.
Code: copy over the problem decomposition + solution construction code of the DP into the traceback routine. Modify the code so that the subsolution that was used to create the solution is recorded. Let this subsolution be S0.

```
maxValue = max(maxValue, value[j] + cache[i-j-1]
cache[i] = maxValue
```

Code: that makes up the list of components/solution elements to return. This will often be S0 appended to a recursive call to the traceback routine function. The arguments of the recursive call will be the smaller problem that created S0.

```
for j in range(i):
```

Combine the code into one recursive function and write it below:

```
int[] cutTree(values, L)
cache[]
cache[] = ∅
if (L <= 0):
    return ∅
for i in range(1, L+1):
    maxValue = ∅
    for j in range(i):
        maxValue = max(maxValue, value[j] + cache[i-j-1])
    cache[i] = maxValue
return cache[L]
```

3) Name of Problem: **Pick the best coins game**

Problem statement: Given a two-person game described as follows:
There is a single row of **n** coins laid out between the two players, each coin has a positive integer value (**n** is even) **v[i]**. The player whose turn it is to move can pick one of the coins from either side of the row. The goal is to have a pile of coins with the most total value. Write a function that will return the list of actions the player should take (pick left or pick right).

Derive Recursive Function:
First simplify the problem to just compute the maximum value of coins that can be obtained (not the actual coins)

Write: What are the variables that describe all problems? *(hint: think of how the row of coins will change as players pick up coins from either end of the row)*

Write: What is the type of the solution?
Array: list of actions a player should take; left or right

Math: Write the function declaration:
String[] bestCoins(int* row, int n)

Think: What are the simplest problems? What are their solutions?
let i be left coin, j be right coin

Math: define the base cases:
if i == j
return left (doesn't matter)
if i < j
return right
or if i > j
return left

Think: How can a larger problem be broken into smaller problems? (Think simply, but creatively)
Math: Write down the simpler problems in terms of the larger problem description
given a row of n=2 coins, Pick the max coin.
Which coin is larger? Pick it

Think: How is the solution to the input problem constructed from the solutions to the smaller problems?
Math: Write down the solution construction step

return left if i > j
return Right if i < j

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

even
v

```
1) String[] bestCoins(int* row, int n)
2) if i == j: return left
   if i > j: return left
   if i < j: return right
3) for k in range(i, j) // keep track of moves too
       max(valueI + min(bestCoins(i+2, j), bestCoins(i+1, j-1)),
           valueJ + min(bestCoins(i+1, j-1), bestCoins(i, j-2)))
Return moves
```

Convert to a Dynamic Program:

Review the existing recursive algorithm definition

Review: What are the arguments to the recursive function? These will be the indexes into the cache.

Review: What is the return type of the function? This will be the type that is stored in the cache.

Think: Are all the arguments integers or are they of mixed type? If mixed, maybe a dictionary or hash table may be best. Will all possible solutions be generated, or only a few specific problems? If all, then an array may be best, if only a few, then maybe a dictionary or hash table should be used.

Write the declaration of the cache data structure. This will be defined once, as the first statement of the new DP function:

```
cache[n][n]
```

Think: How can the base cases in the recursive solution be stored in the cache? If the test checks that all of the arguments are a specific value, then this will become a single assignment into the cache. If the test checks only some of the arguments, or tests for ranges of argument values, then loops will be needed that enumerate all possible input argument values that will pass the test. For each simple problem instance generated, make an assignment into the cache, storing the value returned by the base case.

Code: Write the loops/assignments that fill in the cache with the base cases

```
for (k = 0; k < n; ++k) {
    for (i = 0, j = k; j < n; ++i, ++j)
    :
    }
cache[i][j] = max(row[i] + min(a, b), row[j] + min(b, c));

a = (((i+2) <= j) ? table[i+2][j] : 0);
b = (((i+1) <= (j-1)) ? table[i+1][j-1] : 0);
c = ((i <= (j-2)) ? table[i][j-2] : 0);
```

Review: the recursive calls to understand how the main problem is made simpler and what smaller solutions are needed to compute the main solution.

Think: How do the function arguments change in the recursive calls? Do they always go smaller? What order should the DP loop through the values of the function

parameters, such that the smaller problems are always computed before the larger problems?

Write: the code that scans through the cache in the correct order. Here the loops will iterate over the values to the function arguments from smallest value (excluding the base cases) to the largest values. If the function takes more than one argument, nested loops will be needed.

```
for (k=0; k<n; ++k)
    for (i=0, j=k; j<n; ++i, ++j)
        :

cache[i][j] = max(row[i]+min(a,b), row[j]+min(b,c));
```

Review the code that calls the smaller problem instances recursively and computes the main solution.

Code: Rewrite this recursive code by replacing subproblem function calls with cache access and returns with cache assignments. This code will be the body of the code inside the nested for loops. Note, that the names of the variables in the recursive code will have to be changed to the iterative variables of the DP loops.

```
for (i=0, j=k; j<n; ++i, ++j)
    a = ((i+2)<=j) ? table[i+2][j] : ∅;
    b = ((i+1)<=(j-1)) ? table[i+1][j-1] : ∅;
    c = (i<=(j-2)) ? table[i][j-2] : ∅;
    cache[i][j] = max(row[i]+min(a,b), row[j]+min(b,c));
```

Code: add the return statement at the end that returns the solution to the original problem by indexing into the cache

```
return cache[0][n-1];
```

Write the Trace Back routine:
This will traverse through the solution cache identifying the list of best choices the player can make. Let left be represented as True and right represented as False.

Think: What are the details of the problem we need, and how they are used in the code?
Write: Identity the data structure to store the solution details. This could be a list of objects or values.

Array: list of actions a player should take

Write: the traceback function definition with arguments the same as the recursive definition and the return type as the list of solution components.

```
String [] bestCoins( row, n)
```

Study: the base cases of the recursive algorithm
Write: the same base cases for the traceback routine, this time returning the solution component for each case (often empty).

```
if j == i
    return Vi
if i<j                    if i>j
    return right              return left
```

Study: the problem decomposition + solution construction code of the DP algorithm and identify where the sub-solutions are located in the array, relative to the current problem.
Code: copy over the problem decomposition + solution construction code of the DP into the traceback routine. Modify the code so that the subsolution that was used to create the solution is recorded. Let this subsolution be S0.

```
(cache[i][j] = max(row[i] + min(a,b), row[j] + min(b,c));
}
return table[0][n-1];
```

Code: that makes up the list of components/solution elements to return. This will often be S0 appended to a recursive call to the traceback routine function. The arguments of the recursive call will be the smaller problem that created S0.

```
for (k=0; k<n; ++k){
    for(i=0, j=k; j<n; ++i, ++j)
        :
```

// i is left j is right

Combine the code into one recursive function and write it below:

```
String[] bestCoins(row, n)
                   Cache[[]]
if j==i
    return Vi → add to cache[]
if i<j
    return right Value → add to cache[]
if j>i
    return left Value → add to cache[]

for(K=0; K<n; ++K) {
   for(i=0, j=K, j<n; ++i, ++j)
   a=((i+2)<=j ? cache[(i+2][j] : 0;
   b=((i+1)<=(j-1))? table[i+1][j-1] : 0;
   C=(i<=(j-2)? table[i][j-2] : 0;

   Cache[i][j] = max[row[i]+min(a,b), row[j]+min(b,C)
   Add string[] of moves to cache[[]]
}

}
maxValue= Cach[0][n-1];
return maxValue string[] of moves
```

4) Name of Problem: **Split an input string into a list of keywords**

Problem statement: Given an input string of characters and a dictionary of keywords, determine if the input string can be exactly split into a sequence of keywords. If so, return the list of keywords. For example, if the input string is "whatthegibb" and the dictionary is {"egibb", "bin", "jim", "hello", "what", "att", "he"} the answer would be no (or False). If the input string was "atthebin" the answer would be True, with the words "att", "he" and "bin").

Derive Recursive Function:
First simplify the problem to just compute whether there exists a solution first.

Write: What are the variables that describe all problems? *(hint: think of how the input string can be reduced in size)*
size: size of string
dictionary: list of keywords.
Write: What is the type of the solution?
Bool: T/F

Math: Write the function declaration:
Bool StringSplit (string)

Think: What are the simplest problems? What are their solutions?

Math: define the base cases:
if size == 0: Return true

Think: How can a larger problem be broken into smaller problems? (Think simply, but creatively)
Math: Write down the simpler problems in terms of the larger problem description
Take a string and split it up into sub strings.
Compare sub string to dictionary of key words.

Think: How is the solution to the input problem constructed from the solutions to the smaller problems?
Math: Write down the solution construction step
If the substrings are in the dictionary,
return true.

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

```
1) Bool StringSplit (string input){
2)     if size == 0: return true
3)     for (int i=1; i<= input.size(); i++){
           if (input.Substr(0 i) is in dictonary &&
               StringSplit (input.Substr(i, input.size(-i)))
               return true

       }
```

Convert to a Dynamic Program:

Review the existing recursive algorithm definition

Review: What are the arguments to the recursive function? These will be the indexes into the cache.

Review: What is the return type of the function? This will be the type that is stored in the cache.

Bool

Think: Are all the arguments integers or are they of mixed type? If mixed, maybe a dictionary or hash table may be best. Will all possible solutions be generated, or only a few specific problems? If all, then an array may be best, if only a few, then maybe a dictionary or hash table should be used.

Write the declaration of the cache data structure. This will be defined once, as the first statement of the new DP function:

```
Cache[]
```

Think: How can the base cases in the recursive solution be stored in the cache? If the test checks that all of the arguments are a specific value, then this will become a single assignment into the cache. If the test checks only some of the arguments, or tests for ranges of argument values, then loops will be needed that enumerate all possible input argument values that will pass the test. For each simple problem instance generated, make an assignment into the cache, storing the value returned by the base case.

Code: Write the loops/assignments that fill in the cache with the base cases

```
bool Cache[input.size() + 1]
memset (cache, 0, sizeof (cache))
```

Review: the recursive calls to understand how the main problem is made simpler and what smaller solutions are needed to compute the main solution.
Think: How do the function arguments change in the recursive calls? Do they always go smaller? What order should the DP loop through the values of the function parameters, such that the smaller problems are always computed before the larger problems?
Write: the code that scans through the cache in the correct order. Here the loops will iterate over the values to the function arguments from smallest value (excluding the base cases) to the largest values. If the function takes more than one argument, nested loops will be needed.

```
for ( i = 1 ;  i <= input.sizer) ; i++ )
        :
    for ( int  j = i + 1 ; j <= input.size() ; j ++ )
```

Review the code that calls the smaller problem instances recursively and computes the main solution.
Code: Rewrite this recursive code by replacing subproblem function calls with cache access and returns with cache assignments. This code will be the body of the code inside the nested for loops. Note, that the names of the variables in the recursive code will have to be changed to the iterative variables of the DP loops.

```
for ( i ... ) {
    if (cache[i] == false && input.substr(0, i) is in dictionary)
        cache[i] = true;
    if(cache[i] == true){
        if (i == input.size())
```

Code: add the return statement at the end that returns the solution to the original problem by indexing into the cache

```
            return true.
        for (int j ...        {
            if( cach[j] == false && input.substr(i, j-i) is in dictionary)
                cache[j] = true
            if (j == input.sizer) && cache[j] == true)
                return true
    }
}
}
```

Write the Trace Back routine:
This will traverse through the solution cache identifying the list of keywords found.

Think: What are the details of the problem we need, and how they are used in the code?
Write: Identity the data structure to store the solution details. This could be a list of objects or values.

List of bool

Write: the traceback function definition with arguments the same as the recursive definition and the return type as the list of solution components.

```
bool[] stringSplit (input)
```

Study: the base cases of the recursive algorithm
Write: the same base cases for the traceback routine, this time returning the solution component for each case (often empty).

```
if size == 0
    return true
```

Study: the problem decomposition + solution construction code of the DP algorithm and identify where the sub-solutions are located in the array, relative to the current problem.
Code: copy over the problem decomposition + solution construction code of the DP into the traceback routine. Modify the code so that the subsolution that was used to create the solution is recorded. Let this subsolution be S0.

```
for (i ...)
    if (cache[i] == false && input.Substr(0, i) is in dictionary)
        cache[i] = true
    if (cache[i] == true) {
        if (i == input.size())
            return true
        for (int j ...)
            if (cache[j] == false && input.Substr(i, j-i) is in dictionary)
                cache[j] = true
            if (j == input.size() && cache[j] == true); return true
```

Code: that makes up the list of components/solution elements to return. This will often be S0 appended to a recursive call to the traceback routine function. The arguments of the recursive call will be the smaller problem that created S0.

```
for (i=1;  i<= input.size(); i++)

    for(j= i+1 ; j<= input. size(); j++)
```

Combine the code into one recursive function and write it below:

```
Bool [ ]   string Split (input ) {
    Cache [ ]
    Cache [input.size() + 1 ]
    memset (cache, 0 , sizeof(cache))
    if  size == 0
       return true
    for (i=1;  i<= input.size(); i++) {
       if (cache [i] == false && input. substr(0,i ) is in dictionary)
            cache [i ] =true ;
       if (cache [ i] == true){
          if (i == input. size())
             return true
          for( j = i +1; j<= input.size(); j++ ) {
             if ( cache [ j ] == false&& input.substr(i ,j - i ) isin dictionary
                  cache [ j ] = true
             if ( j == input. size() && cache [ j ] == true)
                  return true
          }
       }
    }
```