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)*

L = maximum weight limit n = no of objects v[i] = value of object at i
w[i] = weight of the object at i

Write: What is the type of the solution?

int

Math: Write the function declaration:

```
int maxR(int L, int n)
```

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

Math: define the base cases:

```
if L=0 return 0;
if n=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

```
max (v[n] + maxR(L-w[n], n-1), maxR(L, n-1));
// max of when a value is picked vs. when it is not picked
```

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 (L-w[n]>=0)
    return max (v[n] + maxR(L-w[n], n-1), maxR(L, n-1));
else
    return maxR(L, n-1);
// if no not enough space left in the bag, dont pick an object
```

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

```
int maxR(int L, int n)
{   if L=0
      return 0;
    if n=0
      return 0;
    if (L- w[n] >= 0)
      return max(v[n]+maxR(L-w[n], n-1), maxR(L, n-1));
    else return maxR(L, n-1);
```

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:

```
int cache[L+1][n+1]
```

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(i=0; i<=L; i++)
  for(j=0; j<=n; j++)
    if(i==0 OR j==0)
      cache[i][j]=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=1; i<=L; i++)
{
    for(j=1; j<=n; 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.

```
if(i-w[j]>=0)
        cache[i][j] = max(v[j] + cache[i-w[j]][j-1], cache[i][j-1]);
else
        cache[i][j] = cache[i][j-1];
```

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

```
for(i=1; i<=L; i++)
{
    for(j=1; j<=n; j++)
    {
        if(i-w[j]>=0)
            cache[i][j] = max(v[j] + cache[i-w[j]][j-1],
                                                cache[i][j-1]
        else
            cache[i][j] = cache[i][j-1];
    }
}
return cache[L][n];
```

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.

list

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

```
list traceback (int L, int 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 (L==0 OR n==0)
    return []; //Empty list
```

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(j=1; j<=n; j++) //checking the different items for the max value
{   if (L-w[j]>=0)
    {  if ((v[j]+cache[L-w[j]][j-1])>= sol)
       {        sol = v[j] + cache[L-w[j]][j-1];
                key[k] = w[j]}}} //storing the weight of the subsolution
```

in list key

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.

```
return key[k] + traceback(L-w[j], n-1);
```

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

```
list Traceback (int L, int n)
{
    sol = cache[L][n-1];
    if (L==0 OR n==0)
        return [];
    for(j=1;j<=n;j++)
    {
        if((L-w[j])>=0)
        {
            if((v[j]+cache[L-w[j]][j-1])>=sol)
            {
                sol = v[j]+cache[L-w[j]][j-1];
                key[k] = w[j]; // finding the subsolution
                               and storing it into key.
            }
        }
    }
    k++;
    return key[k] + Traceback(L-w[j], n-1);
}
```

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)*

l = length of the tree v[i] = value of the cut at i w[i] = length of the cut at i

Write: What is the type of the solution?

```
int
```

Math: Write the function declaration:

```
int maxR(int 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

```
for(i=0; i<n; i++)
{   if (l-w[i]>=0)
    {
        best = max(v[i] + maxR(l-w[i], best);
    }
}
```

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

```
for(i=0; i<n; i++)
{   if(l-w[i]>=0)
    {
        best = max(v[i] + maxR(l-w[i], best);
    }
}
return best;
}
```

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

```
int maxR(int l)
{   if(l<0) return 0;
    for(i=0; i<n; i++)
    {   if((l-w[i])>=0)
            best= max(v[i] + maxR(l-w[i]), best);
    }
    return best;
}
```

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:

```
int maxR(int l)
```

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=1; i<=l; i++)
  {   for(j=1; j<=n; 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.

```
if (i-w[j]>=0)
        sol = max(v[j] + cache[i-w[j]], sol);
```

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

```
for(i=1; i<=l; i++)
{   sol = 0;
    for(j=1; j<=n; j++)
    {   if(i-w[j]>=0)
        {   sol = max(v[j] + cache[i-w[j]], sol);
        }
    }
    cache[i] = sol;
}
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.

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

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

```
list traceback(int 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 []; // empty list
```

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(j=1; j<=n; j++) // checking the different cuts for max value
{   if(L-w[j]>=0)
    {   if((v[j]+cache[L-w[j]]>sol)
        {       sol=v[j]+cache[L-w[j]];
                key[k]=w[j]; ??? // finding and storing the weight
                                 of the subsolution in
                                 list key
```

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.

```
return key[k]+traceback(L-w[j])
```

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

```
int traceback(int L)
{
    sol=0;
    if(L==0)
        return[];
    for(j=1;j<=n;j++)
    {
        if((L-w[j])>=0)
        {
            if((v[j]+cache[L-w[j]]>sol)
            {
                sol=v[j]+cache[L-w[j]];
                key[k]=w[j]; // finding the
                best subsolution
                and storing it
                into list key.
            }
        }
    }
    k++;
    return key[k] + traceback(L-w[j]);
}
```

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)*

n = number of coins v[i] → value of coin at i

Write: What is the type of the solution?

int

Math: Write the function declaration:

int maxC(int l, int r) // l → index of the left most coin
r → index of the right most coin

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

Math: define the base cases: if l > r return 0 // no coins left

if l+1 == r return max(v[l], v[r]) // only 2 coins 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

max(v[l]+min(maxC(l-1,r-1), maxC(l-2,r)), // if pick left coin
v[r]+min(maxC(l-1,r-1), maxC(l,r-2))); // if pick right coin

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(v[l]+min(maxC(l-1,r-1), maxC(l-2,r),
v[r]+min(maxC(l-1,r-1), maxC(l,r-2)));

// max of the two options (picking left coin or right coin), while my opponent can pick either left or right coin (so taking min of his value from his choice)

Finish Code: Put it all together

1) Function declaration:

2) Base cases:

3) Combine problem decomposition with solution construction:

```
int maxC(int l, int r)
{   if (l > r) return 0;
    if (l+1)==r return max(v[l], v[r]);
    return(max(v[l] + min(maxC(l-1,r-1), maxC(l-2,r)),
               v[r] + min(maxC(l-1,r-1), maxC(l,r-2))));
}
```

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:

```
int cache[l+1][r+1]
```

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

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.

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.

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

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.

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

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).

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.

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:

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)*

int l = last index of the string

Write: What is the type of the solution?

Boolean

Math: Write the function declaration:

boolScheck (int l)

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

Math: define the base cases:

```
if(l<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

```
if(is a dictionary(substring) && Scheck(i-1))
    return true;
```

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(l<0) return true;
for(i=l; i>=0; i--)
{   substring = substr of string from index(i, l+1);
    if(is in dictionary(substring) && Scheck(i-1))
        return true;
}
return false;
```

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

```
bool Scheck(int l)          // checks the string backwards for matching keywords.
    if(l<0)                 // If it finds any, checks if rest of the string can be
        return true;        // perfectly split into keywords
    for(i=l; i>=0; i--)
    {   substring = subset of string from index(i, l+1);
        if(isindictionary'(substring) && Scheck(i-1))
            return true;    // return true when the substring and
                            the rest of the string returns true
    }
    return false;
```

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.

Boolean

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:

```
bool cache[l+1]
```

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] = true;
```

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(j=1; j>=0;  )                    // loop being reduced by the length
{ for(i=j; i>=0; i--)                // the substring found
  { if{ ____
        ____
        j=i-1;
  }
}
```

finding out substrings from i->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.

```
substring = s(i, j+1)
if (isindictionary(substring) && cache[i-1]
{       cache[j]=true;
        j=i-1;
}
```

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

```
for(j=1; j>=0;  )
{ for(i=j; i>=0; i--)
  {     substring = s(i, j+1)
        if(isindictionary(substring) && cache[i-1]
        {       cache[j]=true;
                j=i-1;
                ct++;   // counter to check if any
        }               // keywords match or not
  }                     // intial value:
if(ct==0) cache[j]=false;   // ct=0
}return cache[1];
```

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 type string

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

```
list traceback(int l) // l is the last index of the string
```

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 []
```

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=l-1; i>=0; i--) // starting from last index-1, since last index
{   if(cache[i] == true) // will always be true in my DP algo
    {
        string subset = s(i+1, l);
```

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

```
return subset + traceback(i+1);
```

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

```
let traceback (int i)
{     if(i==0)
        return [];
    for(i=i-1; i>=0; i--)
    {     if(cache[i]==true)
          {     string subset = s.(i+1, i);
                return subset + traceback(i+1);
          }
    }
    return [];
}
```

// passing index i+1, since the loop starts with i-1