Dynamic Programming

Q#1: Coin Change Problem to return the amount with minimum number of coins.
Given an array of coins of size "N" having infinite occurrences of against different currency values. The task is to return the amount using the minimum number of coins. We are taking the assumption that there exists a $1 coin at the 1st index of C[].

```
//Brute force recursive
Coin Change(C[], N, Amt)
{
   if(N == 1)        //Since we need to return the amount in any case so when we are left with 1 coin of
      return Amt  //$1 then number of coins required to return the amount will be equals to "Amt"
   if(Amt == 0)    //When the "Amt" becomes 0 then no more coin will be required to return the "Amt"
      return 0      //so return 0
   if(C[N] > Amt) //if "Amt" is less than the currency value at index "N" then we need to drop the coin
      return Coin Change(C, N-1, Amt)
   else //Now it is our freedom to either pick or drop the coin. So we will check both and will pick min.
      return min(Coin Change(C,N-1,Amt), 1+CoinChange(C, N, Amt-C[N]))
}
//DP bottom-up/iterative
Coin Change(C[], N, Amt)
{
   //declaration
   R[0...N][0...Amt]
   //Initialization
   for(i=0 to N)
      R[i][1] = i
   for(j=0 to Amt)
      R[0][j] = 0
   //Computation
   for(i=1 to N)
   {
      for(j=1 to Amt)
      {
         if(C[i] > j)
            R[i][j] = R[i-1][j]
         else
            R[i][j] = min (R[i-1][j], 1+R[i][j-C[i]])
      }
   }
   //final result
   return R[N][Amt]
}
```

Q#2: Binary (0/1) knapsack problem to maximize the profit.
Given a knapsack of capacity "M" Kg. There are "N" number of items available having different weights and values. The weights and their corresponding values are stored in the arrays W[], and V[] respectively. The task is to maximize profit.

//Brute force recursive

```
KSP(W[], V[], N, M)
{
   if(N == 0)  // if no more item left then profit will be 0.
      return 0
   if(M == 0) // if there is no more capacity in the knapsack then profit will be 0
      return 0
   if(W[N] > M) // if the weight of item is greater than the capacity then, drop the item.
      return KSP(W,V,N-1,M)
   else  // Now it is our freedom to either pick or drop the item so we will check both possibilities.
      return max(KSP(W,V,N-1,M), V[N]+KSP(W,V,N-1,M-W[N]))
}
```

//DP bottom-up/iterative

```
KSP(W[], V[], N, M)
{
   //declaration
   R[0...N][0...M]
   //Initialization
   for(i=0 to N)
      R[i][0] = 0
   for(j=0 to M)
      R[0][j] = 0
   //Computation
   for(i=1 to N)
   {
      for(j=1 to Amt)
      {
         if(W[i] > j)
            R[i][j] = R[i-1][j]
         else
            R[i][j] = max(R[i-1][j], V[i]+R[i][j-W[i]])
      }
   }
   //final result
   return R[N][M]
}
```

Q#3: Longest common sub-sequence problem.
Given two strings "A and B" of length "N and M" respectively. The task is to find the length of longest common sub-sequence of the given strings.
Input: A[] = "satisfactory" and B[] = "station"
Output: 5, Longest common substring will be "stato"

//Brute force recursive
```
LCS(A[], B[], N, M)
{
   if(N == 0)  //if no more characters left in 1st string then longest common sub-sequence will be 0.
      return 0
   if(M == 0) //if no more characters left in 2nd string then longest common sub-sequence will be 0.
      return 0
   if(A[N] == B[M]) //if characters of A and B at index N and M are equal then we found 1 common
      return 1 + LCS(A,B,N-1,M-1)  // character and will check for remaining characters.
   else
      return max(LCS(A,B,N-1,M), LCS(A,B,N,M-1))
}
```
//DP bottom-up/iterative
```
LCS(A[],B[],N,M)
{
   //declaration
   R[0...N][0...M]
   //Initialization
   for(i=0 to N)
      R[i][0] = 0
   for(j=0 to M)
      R[0][j] = 0
   //Computation
   for(i=1 to N)
   {
      for(j=1 to M)
      {
         if(A[i] == B[j])
            R[i][j] = 1 + R[i-1][j-1]
         else
            R[i][j] = max(R[i-1][j], R[i][j-1])
      }
   }
   //final result
   return R[N][M]
}
```

Q#4: Rod Cutting problem to maximize the profit.
There exists a rod of length "N" and an array of price P[] for different segments. The task is to cut the rod in such a ways so that the profit can be maximized.

//Brute force recursive

```
RodCut(P[], N)
{
   if(N == 0)
      return 0
   q = 0
   for(i=1 to N)
      q = max(q, P[i] + RodCut(P, N-i))
   return q
}
```

//DP bottom-up/iterative

```
RodCut(P[], N)
{
   //declaration
   R[0...N]
   //Initialization
   R[0] = 0
   //Computation
   for(j = 1 to N)
   {
      q = 0
      for(i=1 to j)
         q = max(q, P[i] + R[j-i])
      R[j] = q
   }
   return R[N]
}
```

Q#5: Edit Distance problem to convert one string into another using minimum operations. (operations are insert, delete, and replace).

```
//Brute force recursive
Edit Dist(A[],B[],N,M)
{
   if(N == 0)
      return M
   if(M == 0)
      return N
   if(A[N] == B[M])
      return EditDist(A,B,N-1,M-1)
   else
      return 1 + min(EditDist(A,B,N-1,M), EditDist(A,B,N,M-1), EditDist(A,B,N-1,M-1))
}

//DP bottom-up/iterative
EditDist(A[],B[],N,M)
{
   //declaration
   R[0...N][0...M]
   //Initialization
   for(i=0 to N)
      R[i][0] = i
   for(j=0 to M)
      R[0][j] = j
   //Computation
   for(i=1 to N)
   {
      for(j=1 to M)
      {
         if(A[i] == B[j])
            R[i][j] = R[i-1][j-1]
         else
            R[i][j] = 1 + min(R[i-1][j], R[i][j-1], R[i-1][j-1])
      }
   }
   //final result
   return R[N][M]
}
```

Q#6: CoinChange problem to determine the count of all possible ways to return the amount.

```
//Brute force recursive
CoinChange(C[], N, Amt)
{
   if(Amt == 0)
      return 1
   if(N == 0)
      return 0
   if(C[N] > Amt)
      return CoinChange(C, N-1, Amt)
   else
      return CoinChange(C,N-1,Amt) + CoinChange(C,N,Amt-C[N])
}
//DP bottom-up/iterative
CoinChange(C[], N, Amt)
{
   //declaration
   R[0...N][0...Amt]
   //Initialization
   for(i=0 to N)
      R[i][0] = 1
   for(j=0 to Amt)
      R[0][j] = 0
   //Computation
   for(i=1 to N)
   {
      for(j=1 to Amt)
      {
         if(C[i] > j)
            R[i][j] = R[i-1][j]
         else
            R[i][j] = R[i-1][j] + R[i][j-C[i]]
      }
   }
   //final result
   return R[N][Amt]
}
```

Q#7: Subset sum problem to determine whether there exists any subset of array whose sum sum will be equals to the target value "K"

```
//Brute force recursive
SSP(A[], N, K)
{
   if(K == 0)
      return 1
   if(N == 0)
      return 0
   if(A[N] > K)
      return SSP(A, N-1, K)
   else
      return SSP(A,N-1,K) || SSP(A,N-1,K-A[N])
}
//DP bottom-up/iterative
SSP(A[], N, K)
{
   //declaration
   R[0...N][0...K]
   //Initialization
   for(i=0 to N)
      R[i][0] = 1
   for(j=0 to K)
      R[0][j] = 0
   //Computation
   for(i=1 to N)
   {
      for(j=1 to K)
      {
         if(A[i] > j)
            R[i][j] = R[i-1][j]
         else
            R[i][j] = R[i-1][j] || R[i][j-A[i]]
      }
   }
   //Final result
   return R[N][K]
}
```

Maximum subarray sum

Idea: Create an array R[] and fill this array determining max sub-array ending at each index "i" and then determine the maximum of R[] to get the overall max sub-array sum. You can also keep track of overall max sub-array sum in the same loop as given below. Main condition is **R[i] = max (R[i-1]+A[i], A[i]) i.e.,** index "i" of R will be **maximum of (**max so far plus current index value **or** only the current index value**)**

```
Max_SubArr_Sum(A[], N)
{
   m_sum = A[1]  // To keep track of overall maximum sub-array sum of the given input array.
   R[1...N] // To determine maximum sub-array ending at each index.
   R[1] = A[1] //max sub-array ending at index#1 will be A[1]

   for(i=2 to N)
   {
//max sub-array ending at index i will be equals to max(R[i-1]+A[i], A[i])
      R[i] = max(R[i-1]+A[i], A[i]);   //
      m_sum = max(R[i], m_sum);  //update overall maximum
   }
   return m_sum
}
```

Sample Input:

N = 8

Arr[]	-2	-3	4	-1	-2	1	5	-3

===

Dry Run:

Hard Coded Initialization:

m_sum = -2 (to store overall maximum sub-array sum)

R[]	-2							

===

i = 2

R	-2	-3						

m_sum = -2 (previous value retained)

===

i = 3

R	-2	-3	4					

m_sum = 4 (value updated)

===

i = 4

R	-2	-3	4	3				

m_sum = 4 (value updated)

===

i = 5

R	-2	-3	4	3	1			

m_sum = 4 (previous value retained)

===

i = 6

R	-2	-3	4	3	1	2		

m_sum = 4 (previous value retained)

==

i = 7

R	-2	-3	4	3	1	2	7	

m_sum = 7 (value updated)

==

i = 8

R	-2	-3	4	3	1	2	7	4

m_sum = 7 (previous value retained)

Overall max sub-array sum = 7

Time Complexity: **O(N)**

Space Complexity: **O(N)**