Lab 10 solutions

1. In the slides, Greedy Strategy #2 for solving the Knapsack Problem was the following:

   Greedy Strategy #2. Try arranging S in decreasing order of *value per weight*. For each i, let $b_i = v_i/w_i$. Scan the new arrangement S′ of S and put in items as long as the weight restriction permits; skip over items that will cause the weight to exceed W.

   Give an example of a Knapsack problem for which this strategy does *not* give an optimal solution.

   Solution: Try {s0, s1, s2} where w[] = {14, 10, 10}, v[] = {30, 20, 20}, W = 20. Then the sequence of benefits in decreasing order is 30/14, 20/10, 20/10. The greedy strategy will pick {s0} and have to stop. Value obtained in that case is 30. But an optimal solution is {s1, s2}, with value 40.

2. Below, the BinarySearch and Recursive Fibonacci algorithms are shown. In each case, what are the subproblems? Why do we say that the subproblems of BinarySearch *do not overlap* and the subproblems of Recursive Fibonacci *overlap?* Explain.

```
Algorithm binSearch(A, x, lower, upper)
    Input: Already sorted array A of size n, value x to be
           searched for in array section A[lower]..A[upper]
    Output: true or false

 if lower > upper then return false
 mid ← (upper + lower)/2
 if x = A[mid] then return true
 if x < A[mid] then
      return binSearch(A, x, lower, mid – 1)
 else
      return binSearch(A, x, mid + 1, upper)
```

```
Algorithm fib(n)
    Input: a natural number n
    Output: F(n)

 if (n = 0 || n = 1) then return n

 return fib(n-1) + fib(n-2)
```

   Solution: Subproblems for binSearch involve checking middle value in smaller and smaller sections of the input array. Subproblems in fib are computations of fib(k) for inputs k < n. In binSearch, each self call examines a middle value that is either to the left or to the right of the middle value examined in the previous call, so there is no overlap. In fib, calls fib(n-1) and fib(n-2) will both need to call all of the following: fib(n-3), fib(n-4), . . ., fib(1), fib(0), so there is substantial overlap of the subproblems.

3. Consider the following SubsetSum problem: S = {4, 2, 5, 3}, k = 5. Fill in *the first row* of the table for the bottom-up dynamic programming solution for this problem. (Locate the formula for this in the slides.)

   Solution:

	0	**1**	**2**	**3**	**4**	**5**
0	**{}**	**NULL**	**NULL**	**NULL**	**{4}**	**NULL**

4. Consider the following SubsetSum problem: S = {4, 3, 5, 6}, k = 8. Part of the table A[i,j] for the bottom-up dynamic programming solution is provided. Use the recursive formula given in the slides to compute the values of A[1,7] and A[2,7].

A[i,j]	0	1	2	3	4	5	6	7	8
0	{}	NULL	NULL	NULL	{4}	NULL	NULL	NULL	NULL
1	{}	NULL	NULL	{3}	{4}	NULL	NULL	**{4,3}**	
2								**{4,3}**	
3									

5. [Optional] Consider the following Knapsack problem: S = {s0, s1, s2, s3}, w[] = {3, 1, 3, 5}, v[] = {4, 2, 3, 2}, W = 7. Part of the table A[i,j] for the bottom-up dynamic programming solution is provided. Use the recursive formula given in the slides to compute the values of A[2,7] and A[3,7].

A[i,j]	0	1	2	3	4	5	6	7
0	{}	{}	{}	{s0}	{s0}	{s0}	{s0}	{s0}
1	{}	{s1}	{s1}	{s0}	{s0, s1}	{s0, s1}	{s0,s1}	{s0,s1}
2	{}	{s1}	{s1}	{s0}	{s0, s1}	{s0, s1}	{s0,s2}	**{s0,s1,s2}**
3								**{s0,s1,s2}**

6. Use the Knapsack problem you created in Problem 1 as a starting point, but now find an optimal solution for the *fractional* knapsack problem based on the same input data.

Solution. We again use this data for the fractional knapsack problem:
items: {s0, s1, s2}
weights: w[] = {14, 10, 10}
values: v[] = {30, 20, 20}
max weight: W = 20.
Then the sequence of benefits in decreasing order is 30/14, 20/10, 20/10.
Using fractionalKnapsack, we define x0, x1, x2 to be 1.0, 0.6, 0.0. Then
total weight = 1.0 * 14 + 0.6 * 10 + 0.0 * 10 = 20
total value = 1.0 * 30 + 0.6 * 20 = 42

7. Devise a dynamic programming solution for the following problem:
   Given two strings, find the length of longest subsequence that they share in common.

   Different between substring and subsequence:
   Substring: the characters in a substring of S must occur contiguously in S.
   Subsequence: the characters can be interspersed with gaps.
   For example: Given two Strings - “regular” and “ruler”, you algorithm should output 4.

   Recursive Brute Force Solution:

   define the prefixes $S_i = a_1 .... a_i$ and $T_j = b_1 ... b_j$

   Algorithm **LCS(S_i, T_j)**
   Input **String S_i and T_j with length i and j, respectively**
   Output **Length of the LCS of S_i and T_j**

```
if i = 0 || j = 0 then
    return 0
else if S[i] = T[j] then
    return LCS(S_{i-1}, T_{j-1}) + 1
else
    return max { LCS(S_{i-1}, T_j), LCS(S_i, T_{j-1}) }
```

Dynamic Programming Solution:

Let $L_{i,j}$ be the length of the LCS for S_i and T_j, $L_{i,j}$ = LCS(S_i , T_j)

If (S[i]=T[j])

$$L_{i,j} = L_{i-1,j-1} + 1$$

else

$$L_{i,j} = \max(L_{i-1,j}, L_{i,j-1})$$

Algorithm ***LCS***(X, Y):

Input: Strings ***X*** and ***Y*** with ***m*** and ***n*** elements, respectively

Output: L is an (***m*** + 1)x(***n*** +1) array such that L[*i*, *j*] contains the length of the LCS of X[1..i] and Y[1..j]

```
m ← X.length
n ← Y.length
for i ← 0 to m do
    L[i, 0] ← 0
for j ← 0 to n do
    L[0, j] ← 0
for i ← 1 to m do
    for j ← 1 to n do
      if X[ i ] = Y[ j ]  then
          L[i, j] ← L[i -1, j -1] + 1
      else
          L[i, j] ← max { L[i -1, j], L[i, j -1] }
 return L
```

8. *(Optional Interview Question)* Devise a dynamic programming solution for the following problem:

   Given a positive integer n, find the least number of perfect square numbers which sum to n. (Perfect square numbers are 1, 4, 9, 16, 25, 36, 49, …)
   For example, given n = 12, return 3; (12 = 4 + 4 + 4)
   Given n = 13, return 2; (13 = 4 + 9)
   Given n = 67 return 3; (67 = 49 + 9 + 9)

Solution - main idea:

dp[n] = min{dp[n-i*i]} + 1

where n-i*i >=0 and i >=1