Program set 3: solution

This set of questions focus on dynamic programming.

1. Knapsack without repetitions. Consider the following knapsack problem:

   The total weight limit $W = 10$ and

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

   Solve this problem using the dynamic programming algorithm presented in class. Please show the two dimensional table $L(w, j)$ for $w = 0, 1, ..., W$ and $j = 1, 2, 3, 4$.

Solution : Subproblems definition

Let $V(n, W)$ be the maximum value obtainable with item 1,2,…,n and weight W.

Recursive Formulation

To get $V(n, W)$ we consider

- Whether we can accommodate the item in the current weight limit
  $w_i \leq W$
- Whether accommodating the current item will lead to a greater value for that weight.
  $V(i, W) = V(i-1, W-w_i) + v_i$

So our decision is reduced to whether or not to include item n or not. Our objective here is to maximize the value, hence we take the maximum of these values over all i and w_i.

$$V(n, W) = \max \begin{cases} V(n-1, W-w_n) + v_n & \text{Use item n} \\ V(n-1, W) & \text{Discard item n} \end{cases}$$

Base cases:

$$V(i, 0) = 0, for\ i = 0,1, \dots, n$$
$$V(0, j) = 0, for\ j = 0,1, \dots, W$$

Pseudo-code

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

Complexity

- The value table has $(n+1)\times(W+1)$ entries, each requiring $O(1)$ time to compute.
- Overall complexity is $O(nW)$.

Output

L	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Sample code:

http://rerun.me/2014/05/27/the-knapsack-problem/

2. Give a dynamic programming algorithm for the following task.

Input: A list of n positive integers $a_1, a_2, \ldots, a_n$ and a number t.
Question: Does some subset of the a_i's add up to t? (You can use each a_i at most once.)

Solution: Subproblem definition and Recursive formulation

The running time should be $O(nt)$.

Consider the subproblem $L(i, s)$, which returns the answer of "Does a subset of $a_1, ..., a_i$ sum up to s?". Below are some substeps that will help you develop your algorithm.

a What are the two options we have regarding item i toward answering the subproblem $L(i, s)$?
The two options are using a_i or not using a_i.

b For each of the option, how would it change the subproblem? More specifically, what happens to the target sum and what happens to the set of available integars?
If we use a_i, the target sum will become $s - a_i$, and the available integers that can be used to achieve this target sum will become $a_1, ..., a_{i-1}$, which is captured by subproblem $L(i-1, s-a_i)$.
If we do not use a_i, the target sum will remain to be s and the available integers that can be used to achieve this target sum will become $a_1, ..., a_{i-1}$, which is captured by subproblem $L(i-1, s)$.

c Based on the answers to the previous two questions, write a recursive formula that expresses $L(i, s)$ using the solutions to smaller subproblems.
The recursive formula is $L(i, s) =$ $L(i-1, s)$ OR $L(i-1, s-a_i)$. Here we assume that the subproblem returns either 0 or 1, where 0 means the answer is no, and 1 means that the answer is yes.

Pseudo-code

d Provide pseudocode for the dynamic programming algorithm that builds the solution table $L(i, s)$ and returns the correct answer to the final problem.

$L(i, 0) = True$ for $i = 1, \cdots, n$
$L(0, s) = False$ for $s = 1, \cdots, t$
for $i = 1, \cdots, n$
    for $s = 1, \cdots, t$
        if $s < a_i$: $L(i, s) = L(i-1, s)$
        else: $L(i, s) = L(i-1, s)$ OR $L(i-1, s-a_i)$
    end for
end for
return $L(n, t)$

e Modify the pseudocode such that it will not only return the correct "yes", "no" answer, but also return the exact subset if the answer is "yes".

$L(i, 0) = True$ for $i = 1, \cdots, n$
$S(i, 0) = \emptyset$ for $i = 1, \cdots, n$
$L(0, s) = False$ for $s = 1, \cdots, t$
$S(0, s) = \emptyset$ for $i = 1, \cdots, n$
for $i = 1, \cdots, n$
    for $s = 1, \cdots, t$
        if $s < a_i$: $L(i, s) = L(i-1, s)$; $S(i, s) = S(i-1, s)$
        else if $L(i-1, s)$: $L(i, s) = True$; $S(i, s) = S(i-1, s)$
        else if $L(i-1, s-a_i)$: $L(i, s) = True$; $S(i, s) = S(i-1, s-a_i) \cup i$
        else: $L(i, s) = False$; $S(i, s) = \emptyset$
    end for
end for
return $L(n, t)$ and $S(n, t)$

Complexity

- Table L has $(n+1) \times (t+1)$ entries
- Overall complexity is $O(nt)$

Sample code

http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/

3.

6.2. You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \cdots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Solution: Suproblems definition
Let $OPT(i)$ be the minimum total penalty to get to hotel i.

Recursive formulation
To get $OPT(i)$, we consider all possible hotels j we can stay at the night before reaching hotel i. For each of these possibilities, the minimum penalty to reach i is the sum of:

- the minimum penalty $OPT(j)$ to reach j,
- and the cost $(200 - (a_j - a_i))^2$ of a one-day trip from j to i.

Because we are interested in the minimum penalty to reach i, we take the minimum of these values over all the j:

$$OPT(i) = \min_{0 \le j < i} \{OPT(j) + (200 - (a_j - a_i))^2\}$$

And the base case is $OPT(0) = 0$.

Pseudo-code

```
// base case
OPT[0] = 0
// main loop
for i = 1...n:
  OPT[i] = min([OPT[j] +(200 − (a_j − a_i))^2 for j=0...i-1])
// final result
return OPT[n]
```

Complexity
- We have n subproblems,
- each subproblems i takes time O(i)

The overall complexity is

$$\sum_{i=1}^{n} O(i) = O(\sum_{i=1}^{n}) i = O\frac{n(n-1)}{2} = O(n^2)$$

Sample code
http://www.solvemyproblem.net/Webed/forum/thread.asp?tid=1122

4.

6.3. Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- **At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.**
- **Any two restaurants should be at least k miles apart, where k is a positive integer.**

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

Solution: Subproblems definition

Let $D(i)$ be the maximum profit Yuckdonald's can obtain from locations 1 to i.

Recursive formulation

We have two choices for location i

- Not to open a restaurant at location i, in which case the optimal profit can be obtained from location 1,2,…,$(i-1)$ by $D(i-1)$.
- Open a restaurant at location i. In this case there is an expected profit of p_i. After building at location i, the nearest location to the left where we can build is

  $$i^* = \max\{j \leq i : m_j \leq m_i - k\}$$

  The profit of this can be denoted by $D(i^*)$.

Since we want to maximize our profit, we are interested in the maximum of these values over all i.

$$D(i) = \max\{D(i-1), p_i + D(i^*)\}$$

Base case: $D(0) = 0$.

Pseudo-code

Assuming i^* are known

```
D(0) = 0                    //base case
for i = 1 to n
        noRestaurant = D(i − 1)
        Restaurant = p_i + D(i*)        //storing both estimated profits in variables
        if (noRestaurant > Restaurant)  //deciding if profit is maximum with or without i
                D(i) = noRestautant
        else
                D(i) = Restaurant
return D(n)
```

Complexity

- We have n subproblems
- Each subproblem requires finding an i^* which can be done in time $O(logn)$ by using binary search on the ordered list of location
- And computing the maximum of two values which can be done in constant time
- Overall complexity is $O(nlogn)$

Sample code:

http://stackoverflow.com/questions/35673228/restaurant-maximum-profit-using-dynamic-programming

****Improving the complexity****

i^* can be computed in $O(n)$ time by using the fact that it is an ordered list i.e.
$0 = 1^* \leq 2^* \leq \cdots \leq n^* < n$.

If we start looking for i^* at $j = i - 1$ and increment j untill we find the correct value of i^*, the overall complexity can be reduced to $O(n)$.

5.

6.8. Given two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$, we wish to find the length of their *longest common substring*, that is, the largest k for which there are indices i and j with $x_ix_{i+1} \cdots x_{i+k-1} = y_jy_{j+1} \cdots y_{j+k-1}$. Show how to do this in time $O(mn)$.

Solution: Subproblem Definition

For $1 \leq i \leq n$ *and* $1 \leq j \leq m$, we define a subproblem $L(i,j)$ to be the length of the longest common substring of x and y terminating at x_i *and* y_j.

Recursion Formulation

- If $x_i \neq y_j$, the character is not common in the two strings and $L(i,j) = 0$.
- If $x_i = y_j$, the character is common and the length of the common substring is $1 + L(i-1, j-1)$.

Overall recursive formulation:

$$L(i,j) = \begin{cases} L(i-1,j-1)+1 & \text{if equal}(x_i, y_j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Base case:
For all $1 \leq i \leq n$ *and* $1 \leq j \leq m$,
$L(0,0) = 0$
$L(i,0) = 0$
$L(0,j) = 0$

We are interested in the maximum value of $L(i,j)$ over all $1 \leq i \leq n$ *and* $1 \leq j \leq m$.

Pseudo-code

```
for i = 0 to n
        L(i,0) = 0
for j = 0 to m
        L(0,j) = 0          //base cases

for i = 1 to n
        for j = 1 to m
                if ( x_i = y_j)
                        L(i,j) = 1 + L(i-1,j-1)
                else
                        L(i,j) = 0
return max { L(i,j)}
```

Complexity

- We have $m * n$ subproblems and each takes constant time to evaluate through the recursion
- Overall complexity $O(mn)$.

Sample code
http://algorithms.tutorialhorizon.com/dynamic-programming-longest-common-substring/