Homework 9

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

23 September 2016

Problem 10:

(a)

In order to develop a recursive algorithm for this problem, we first need to know two pieces of information about the interval. First we need to know the sum of the length of the collection of intervals, and second we need to know the right endpoint of this collection. By knowing both of these, we are able to verify that we are only picking the most optimal intervals for each collection up to the maximum length. They are the most optimal because if two collections have the same sum, then we will choose the collection that ends with the smallest right endpoint. By choosing the collection with the smallest right endpoint we are able to have more options as to which interval we want to add to the end of this collection.

The recursive algorithm is as follows:
k = the number of intervals in the collection
s = the sum of the collection
$maxCollection(k, s)$ = the smallest right endpoint with k number of intervals and sum s

```
Function: maxCollection

Input: n intervals over the real line
Initialization:
for k ← 0 to n do
    for s ← 1 to n do
        maxCollection(k, s) = ∞
for k ← 0 to n do
    maxCollection(k, 0) = −∞
End Initialization
for k ← 0 to n do
    for s ← 1 to n do
        if x_k > maxCollection(k − 1, s − 1) then
            min(maxCollection(k−1, s), maxCollection(k−1, s−1)+x_k)
        else
            return maxCollection(k − 1, s)
```

This can then be further developed in an iterative, array-based solution with a polynomial runtime.

```
Function: MaxCollection

Input: n intervals over the real line
Initialization:
for k ← 0 to n do
    for s ← 1 to n do
        maxCollection[k][s] = ∞
for k ← 0 to n do
    maxCollection[k][0] = −∞
End Initialization
for k ← 1 to n do
    for s ← n down to 1 do
        if maxCollection[k][s] is defined then
            if x_k > maxCollection[k − 1][s − 1] then
                maxCollection[k][s] = min(maxCollection[k − 1][s], x_k)
            else
                maxCollection[k][s] = maxCollection[k − 1][s]
return max(maxCollection[n][n])
```

(b)

In order to develop an algorithm for this problem, it is first necessary to set the following pruning rules.

1. If two nodes have the same sum and are at the same depth, prune the subtree with the larger right endpoint.

2. If two nodes have the same right endpoint and are at the same depth, prune the subtree with the smaller sum.

These rules lead to the following iterative algorithm:
k = subset of intervals in the collection
s = sum of the lengths of the collection
$A[k][s]$ = the smallest right endpoint of with k number of intervals and sum s

```
for k ← 1 to n do
    for s ← 0 to n do
        if A[k][s] is defined then
            A[k + 1][s] = min(A[k + 1][s], A[k][s])
            A[k + 1][s + 1] = min(A[k + 1][s + 1], x_k)
return A[n][n]
```

Problem 11:

Part A:

Given the final array, backtracking to identify which items were added to the knapsack is fairly straight-forward. The value $V[n, S]$ is the maximum of the values $Value[n-1, S]$ and $Value[n-1, S-w(n)] + v(n)$. To determine which of those two terms was selected as the final value, look at those two cells in the table and determine which is larger: $Value[n-1, S]$ or $Value[n-1, S-w(n)] + v(n)$.

If the former is larger, then the nth item added was not added to the knapsack. Repeat this process starting at the cell $Value[n-1, S]$ until you terminate at the beginning of the array.

If the latter is larger, then the nth item was added to the knapsack. Repeat this process starting at the cell $Value[n-1, S-w(n)] + v(n)$ until you terminate at the beginning of the array.

Part B:

To solve the Knapsack Problem using only $\mathcal{O}(L)$ space and in time $\mathcal{O}(nL)$, create an array A of length L. The i^{th} element in A represents the maximum value available with a weight constraint i. Thus, the final element represents the solution. Populate the i^{th} element in A by taking the larger of the $A[i-1]$ and $A[i-w(i)] + v(i)$. The first of those two values is the maximum value when you do not add the i^{th} element to the array (it is, of course, the maximum value of the previous weight). The second value is the maximum

value available to the knapsack at weight i if you definitely add the i^{th} element to the knapsack.

Problem 17:

To develop a dynamic programming algorithm to determine if the set of gems $\mathcal{G}$ can be partitioned such that the two partitions $\mathcal{P}$ and $\mathcal{Q}$ have the same value and the same number of rubies and emeralds all possible partitions can be initially considered. Let the total value of all the gems be $\mathcal{L}$ and the total number of gems be n.

In considering every possible partition, a new gem is added to either $\mathcal{P}$ or $\mathcal{Q}$. This gem, labeled g_i, will have a value v_i. Additionally, g_i will be either an *emerald* or a *ruby*. Since the value between the two partitions has to be equal and the quantity of the gem type also has to be the same, these values need to be tracked.

Tracking these values as the tree of all possible partitions is generated gives the following pruing rules.

1. If the value of either partition becomes greater than $\mathcal{L}/2$ the tree can be pruned. This is because if a partition has a value greater than half of the max value, the other partition can never catch up.

2. If the number of emeralds or rubies in either partition becomes greater than n, the tree can be pruned. This is because the other partition will never be able to get enough gems of the necessary type to restablish balance.

3. If two branches have the same value, same number of rubies, and the same number of emeralds in both partitions, one branch can be arbitrarily pruned.

Using these pruning rules gives the following polynomial time algorithm.

Function: *Gem Partition*
Input: $\mathcal{G}$
Globals: $A[\ ][\ ][\ ][\ ][\ ][\ ][\ ][\ ][\ ][\ ]$

```
/* iterate through possible gems                                    */
for l = 1 to n do
    /* iterate through values for P                                 */
    for h = 1 to L/2 do
        /* iterate through rubies for P                             */
        for j = 1 to n/2 do
            /* iterate through emeralds for P                       */
            for k to n/2 do
                /* iterate through values for Q                     */
                for x = 1 to L/2 do
                    /* iterate through rubies for Q                 */
                    for y = 1 to n/2 do
                        /* iterate through emeralds for Q           */
                        for z to n/2 do
                            if A[l,h,j,k,x,y,z] is defined then
                                if g_l is a ruby then
                                    A[l+1, h+v_l, j+1, k, x, y, z] = 1
                                    A[l+1, h, j, k, x+v_l, y+1, z] = 1
                                else
                                    A[l+1, h+v_l, j, k+1, x, y, z] = 1
                                    A[l+1, h, j, k, x+v_l, y, z+1] = 1

/* check if partition is possible                                   */
for i = 1 to L/2 do
    for j = 1 to n/2 do
        for k = 1 to n/2 do
            if A[n,i,j,k,i,j,k] is defined then
                if A[n,i,j,k,i,j,k] == 1 then
                    Return: 1
Return: 0
```

The answer to if a partition is possible will be found by searching the space at the n^{th} level where the values for partitions $\mathcal{P}$ and $\mathcal{Q}$ are the same while also having the same number of rubies and emeralds.

This algorithm runs in $\mathcal{O}\left(n^5\mathcal{L}^2\right)$ time. This is a $poly(n + \mathcal{L})$ if $poly(x) = x^7$.