COMP 3711H – Fall 2016
Tutorial 8
Solution Sketch

1. Let $G = (V, E)$ be a connected undirected graph in which all edges have weight either 1 or 2. Give an $O(|V| + |E|)$ algorithm to compute a minimum spanning tree of G. Justify the running time of your algorithm. (*Note:* You may either present a new algorithm or just show how to modify an algorithm taught in class.)

   Solution

   You could run a Prim's algorithm, just implementing your Priority Queue differently. Every priority queue operation except for the initialization (which will take $O(V)$ time) will now take $O(1)$ time so the algorithm will run in $O(E + V)$ time.

   To do this, note that since every value in the priority queue is either ∞, 0 or 1 you can implement the priority queue by maintaining 3 doubly linked lists. List 0 contains all items with key value ∞, List 1, those with key values 1 and List 2 those with key values 2.

   When you start put everyone is in List 0 because they all have key value ∞. You can create this priority-queue in $O(V)$ time.

   To implement Extract-Min: If List 1 contains a value, just pull off the first one. Otherwise, if List 2 is not empty, pull the first value off of List 2. Otherwise, pull the first item off List 0. Extract Min takes $O(1)$ time.

   To implement Decrease Key: (a) take the item out of its current list which can be done in $O(1)$ time (because the list is doubly linked). (b) Insert the item into the front of the appropriate new list. This can be done in $O(1)$ time because there are only two such possible lists.

2. Give an $O(n^2)$ time dynamic programming algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers (i.e, each successive number in the subsequence is greater than or equal to its predecessor). For example, if the input sequence is $\langle 5, 24, 8, 17, 12, 45\rangle$, the output should be either $\langle 5, 8, 12, 45\rangle$ or $\langle 5, 8, 17, 45\rangle$.

   Hint: Let $d[i]$ be the length of the longest increasing subsequnce whose last item is item i.

   Solution: Algorithm: We first give an algorithm which finds the length of the longest increasing subsequence; later, we will modify it to report a subsequence with this length.

   Let $X_i =< x_1, \ldots, x_i >$ denote the prefix of X consisting of the first i items. Define $c[i]$ to be the length of the longest increasing subsequence that ends with x_i. It is clear that the length of the longest increasing subsequence in X is given by $\max_{1 \le i \le n} c[i]$.

   The longest increasing subsequence that ends with x_i has the form $< Z, x_i >$ where Z is the longest increasing subsequence that ends with x_r for some $r < i$ and $x_r \le x_i$. Thus, we have the following recurrence relation:

$$c[i] = \begin{cases} 1 & \text{if } i = 1 \\ 1 & \text{if } x_r > x_i \text{ for } 1 \le r < i \\ \max\limits_{\substack{1 \le r < i \\ x_r \le x_i}} c[r] + 1 & \text{if } i > 1 \end{cases}$$

   The basis follows from the fact the longest increasing subsequence in a sequence consisting of one number is the number itself. The recurrence relation says that if all the numbers

to the left of i are greater than x_i then the length of the longest increasing subsequence ending in x_i is 1. Otherwise, the length of the longest increasing subsequence ending in x_i is 1 more than the length of the longest increasing subsequence ending at a number x_r to the left of x_i such that x_r is no greater than the x_i.

We store the $c[i]$'s in an array whose entries are computed in order of increasing i. After computing the c array we run through all the entries to find the maximum value. This is the length of the longest increasing subsequence in X.

In order to report the optimal subsequence we need to store for each i, not only $c[i]$ but also the value of r which achieves the maximum in the recurrence relation. Denote this by $r[i]$. Then we can trace the solution as follows. Let $c[k] = \max_{1 \le i \le n} c[i]$. Then x_k is the last number in the optimal subsequence. The second to last number is $x_{r[k]}$, the third to last number is $x_{r[r[k]]}$ and so on until we have found the all the numbers of the optimal subsequence.

Running Time: *Since it takes $O(i)$ time to compute the i-th entry of the c array, the total time to compute the c array is $O(\sum i) = O(n^2)$. It takes $O(n)$ time to find the maximum in the c array. Finally, the time to trace the solution is $O(n)$. Thus, the running time is dominated by the time it takes to compute the c array, which is $O(n^2)$.*

3. The subset sum problem is: Given a set of n positive integers, $S = \{x_1, x_2, \ldots, x_n\}$ and an integer W determine whether there is a subset $S' \subseteq S$, such that the sum of the elements in S' is equal to W. For example, if $S = \{4, 2, 8, 9\}$ and $W = 11$, then the answer is "yes" because there is a subset $S' = \{2, 9\}$ whose elements sum to 11. Give a dynamic programming solution to the subset sum problem that runs in $O(nW)$ time. Justify the correctness and running time of your algorithm.

   Solution

   The solution is to construct a boolean array $A[i, j]$, $0 \le i \le n$ and $0 \le j \le W$, defined as follows: $A[i, j] = true$ if there is a subset of $\{x_1, x_2, \ldots, x_i\}$ that sums to j, else $A[i, j] = false$. We start with some observations.

   Basis: *$A[i, 0] = true$, $0 \le i \le n$, because given 0 or more items, you can always form the sum 0 by picking no item. Also, $A[0, j] = false$, $1 \le j \le W$, because if there are no items to pick from, then we cannot form any sum > 0.*

   Last weight too large: *$A[i, j] = A[i-1, j]$ if $i > 0$ and $x_i > j$. The solution cannot contain x_i if x_i exceeds j, the sum to be formed. Therefore the sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_i\}$ if and only if it can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.*

   Last weight not too large: *$A[i, j] = (A[i-1, j - x_i]$ OR $A[i-1, j])$, if $i > 0$ and $j \ge x_i$. This follows from the following observations. If sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$, then either this subset includes item x_i or it does not. If it includes item x_i then it should be possible to form the sum $j - x_i$ using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$; otherwise if it does not include item x_i then it should be possible to form the sum j using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.*

   Combining these observations we have the following recurrence relation:

$$A[i, j] = \begin{cases} true & \text{if } 0 \le i \le n \text{ and } j = 0 \\ false & \text{if } i = 0 \text{ and } 1 \le j \le W \\ A[i-1, j] & \text{if } i > 0 \text{ and } x_i > j \\ A[i-1, j - x_i] \text{ OR } A[i-1, j]) & \text{if } i > 0 \text{ and } j \ge x_i \end{cases}$$

The algorithm takes as inputs the sum to be formed W, the number of items n, and the sequence $x = x_1, x_2, \ldots, x_n$. It stores the $A[i, j]$ values in a table $A[0 \ldots n, 0 \ldots W]$ whose values are computed in order of increasing i (note that for any given i it does not matter in which order we compute the $A[i, j]$'s). Following this order ensures that the table entries used to compute $A[i, j]$ have all been computed before *the algorithm evaluates $A[i, j]$. At the end of the computation, $A[n, W]$ is true, if there is a subset that sums to W, otherwise it is false.*

Dynamic-SubsetSum(x, n, W)
    $A[0, 0] = true$
    for $j = 1$ *to* W *do*
        $A[0, j] = false$
    for $i = 1$ *to* n *do*
        $A[i, 0] = true$
        for $j = 1$ *to* W *do*
            if $x_i > j$ *then*
                $A[i, j] = A[i - 1, j]$
            else $A[i, j] = A[i - 1, j - x_i]$ *OR* $A[i - 1, j]$

Running Time: *Since the table has $O(nW)$ entries and it takes constant time to compute any one entry, the total time to build the table is $O(nW)$. The total running time is $O(nW)$.*

4. Give an $O(nW)$ dynamic programming algorithm for the 0-1 knapsack problem where n is the number of items and W is the max weight that can fit into the knapsack. Recall that the input is i items with given weights $w_1, w_2, \ldots, w_n$ and associated values v_1, $v_2, \ldots, v_n$ and the objective is to choose a set of items with weight $\leq W$ with maximum value.

   Now suppose that you are given *two* knapsacks with the same max weight. Give an $O(nW^2)$ dynamic programming algorithm for finding the maximum value of items that can be carried by the two knapsacks.

   Solution: The implicit assumption in this problem is that W and the w_i are all integers. This was not needed for the fractional knapsack case (and its greedy solution) but is required for the 0-1 *knapsack problems.*

   We first solve the one knapsack case.

   The algorithm is based on defining a table

$$V(i, w), \quad 0 \leq i \leq n, \quad 0 \leq w \leq W$$

   in which $V(i, w)$ is the maximum value of objects from the set of the first i objects *that can be placed in a knapsack that has maximum weight w. The optimal solution to the problem is $V(n, W)$.*

   The algorithm is based on the following recurrence relation:

$$V(i, w) = \max\Big(V(i-1, w),\, V(i-1, w-w_i) + v_i\Big)$$

   The initial conditions are $\forall i$, $V(i, w) = -\infty$ if $w < 0$ and $\forall w \geq 0$, $V(0, w) = 0$. Note that a value of $-\infty$ is essentially being used as a flag for something being impossible.

   The basic idea behind the equation is that there are two possible cases for the optimal knapsack of size w using the first i items. Either the i'th item is not included or the i'th item is included.

   If the i'th item is not included, then the optimal solution is the optimal solution using the first $i-1$ items, which has value $V(i-1, w)$.

   If the i'th item is included, then it adds value v_i. After including it, the knapsack still has weight capacity of $w - w_i$ and this needs to optimally filled by the first $i-1$ items. The best way of doing this has value $V(i-1, w-w_i)$. Adding the two pieces together gives $V(i-1, w-w_i) + v_i$. Note that if $w_i > w$ the i'th item can't fit into the knapsack so this option is not possible. This is flagged in the recurrence by the fact that $V(i-1, w-w_i) = -\infty$. An alternative option is to write the recurrence as

$$V(i, w) = \begin{cases} \max\Big(V(i-1, w),\, V(i-1, w-w_i) + v_i\Big) & \text{if } w_i \leq w \\ V(i-1, w) & \text{if } w_i > w \end{cases}$$

   Given the recurrence we can fill in the recurrence table by, for each fixed $i = 1, 2, \ldots, n$ (in increasing order), calculating $V(i, w)$ from the recurrence for every $w = 1, 2, 3 \ldots, W$. In this order, when it's time for $V(i, w)$ to be calculated, both of $V(i-1, w)$ and $V(i-1, w-w_i)$ are already known.

   There are $O(nW)$ table entries and each requires only $O(1)$ time to evaluate so the entire algorithm uses only $O(nW)$ time.

The above calculates the best Value. *To find the set of items that achieves that value you will need to keep an auxiliary matrix* $Included(i, w)$ *which is set to be* false *or* true, *depending upon whether the max occurs at* $V(i-1, w)$ *or* $V(i-1, w-w_i)+v_i$. *Using our standard approach we can reconstruct the optimal set from this matrix by working backwards from* $Included(n, W)$.

We now discuss the case of two knapsacks. The algorithm for this case is a simple generalization of the previous one and is based on defining a table

$$V(i, w^1, w^2), \quad 0 \leq i \leq n,\ 0 \leq w^1 \leq W,\ 0 \leq w^2 \leq W$$

in which $V(i, w^1, w^2)$ *is the maximum value of objects* from the set of the first i objects *that can be placed in two knapsacks, the first one having weight capacity* w^1, *and the second having weight capacity* w^2. *The optimal solution to the problem is* $V(n, W, W)$.

The algorithm is based on the following recurrence relation:

$$V(i, w^1, w^2) = \max\big(V(i-1,\ w^1,\ w^2),\ V(i-1,\ w^1-w_i,\ w^2)+v_i,\ V(i-1,\ w^1,\ w^2-w_i)+v_i,\big)$$

(with initial conditions $\forall i,\ V(i,\ w^1,\ w^2) = -\infty$ *if* $w^1 < 0$ *or* $w^2 < 0$ *and* $\forall w^1,\ w^2 \geq 0$, $V(0, w^1, w^2) = 0$.

The basic idea behind the equation is is that the three terms on the right hand side correspond to the three cases in which the optimal solution for $V(i, w^1, w^2)$ *(i) does not use item i at all, (ii) puts item i in the first knapsack and (iii) puts item i in the second knapsack. We do not go into more details because this is very similar to the derivation in the previous case.*

Notice that, if all of the items on the right hand side were already known, then the left hand side could be calculated in $O(1)$ *time. It's not hard to find an ordering that satisfies this (which?) so the algorithm runs in* $O(nW^2)$.

As before, this algorithm only finds the best *value. To find the actual items in the two knapsacks you will need to keep an auxiliary matrix that associates with each entry in the* $V(...)$ *matrix how the optimal value for that entry was achieved.*

5. Suppose you want to make change for n (HK) dollars using the fewest number of coins. Assume that each coin's value is an integer.

   Give an $O(nk)$-time dynamic programming algorithm that makes change for any set of k different coin denominations, assuming the set always contains a 1-dollar coin (so a solution always exists).

   Let the coin denominations be $d_1, d_2, ..., d_k$.

   Solution:

   This problem has the optimal substructure property. If we knew that an optimal solution for j *dollars used* **a** *coin of denomination* d_i, *it would have to be true that, in that solution, the change for the remaining* $j - d_i$ *dollars would have to be an optimal (minimum) set of coins for that subproblem, i.e.,* $c[j] = 1 + c[j-d_i]$ *As base cases, we have that* $c[j] = 0$ *for all* $j \leq 0$.
   To develop a recursive formulation, we have to check all denominations, giving

$$c[j] \quad = \quad \begin{cases} 0 & \text{if } j \leq 0, \\ 1 + \min_{1 \leq i \leq k}\{c[j-d_i]\} & \text{if } j > 1. \end{cases}$$

We can compute the $c[j]$ values in order of increasing j by using a table. The following procedure does so, producing a table $c[1..n]$

Note: The code avoids explicitly examining $c[j]$ for $j \leq 0$ by checking $j \geq d_i$ before looking up $c[j - d_i]$.

The procedure also produces a table $denom[1..n]$, where $denom[j]$ is the denomination of a coin used in an optimal solution to the problem of making change for j dollars.

COMPUTE-CHANGE(n, d, k)

```
for j ← 1 to n
    c[j] ← ∞
    for i ← 1 to k
        if j ≥ d_i and 1 + c[j − d_i] < c[j]
            c[j] ← 1 + c[j − d_i]
            denom[j] ← d_i
return c and denom
```

This procedure obviously runs in $O(nk)$ time

We use the following procedure to output the coins used in the optimal solution computed by COMPUTE-CHANGE:

GIVE-CHANGE$(j, denom)$

```
if j > 0
    give one coin of denomination denom[j]
    GIVE-CHANGE(j − denom[j], denom)
```

The initial call is GIVE-CHANGE$(n, denom)$. Since the value of the first parameter decreases in each recursive call, this procedure runs in $O(n)$ time.