MATH182 DISCUSSION 5 WORKSHEET SOLUTIONS

1. Knapsacks are pretty small; let's get a wagon instead

Exercise 1. Prove that if $q_1, \ldots, q_n$ is an optimal solution to the unbounded knapsack problem for a given weight limit W, and $q'_1, \ldots, q'_n \in Q$ are such that $q'_i \leq q_i$ for $i = 1, \ldots, n$, then $q'_1, \ldots, q'_n$ is an optimal solution for the weight limit $W' := W - \sum_{i=1}^{n}(q_i - q'_i)w_i$. This shows that UKP has optimal substructure.

Proof. The idea here, as with many other optimal substructure proofs, is that if there were a better solution for the subproblem, then we could replace that portion of the overall solution with the better solution and get a better solution to the overall problem.

Let $\widetilde{q}_1, \ldots, \widetilde{q}_n$ be an optimal solution for the weight limit W', with value $\widetilde{v}$. Then $\hat{q}_i = \widetilde{q}_i + q_i - q'_i$ gives a solution for weight limit W, since

$$\sum_{i=1}^{n} \hat{q}_i w_i \;=\; \sum_{i=1}^{n} \widetilde{q}_i + \sum_{i=1}^{n}(q_i - q'_i)w_i \;\leq\; W' + (W - W') \;=\; W$$

(this construction uses that $Q = \mathbb{N}$ is closed under addition and nonnegative subtraction, which is why it works for the unbounded knapsack problem but not the others). The value of this solution is

$$\sum_{i=1}^{n} \widetilde{q}_i v_i + \sum_{i=1}^{n} q_i v_i - \sum_{i=1}^{n} q'_i v_i \widetilde{v} + v - v',$$

where v and v' are the values of $q_1, \ldots, q_n$ and $q'_1, \ldots, q'_n$, respectively. Since $q_1, \ldots, q_n$ is optimal, this is at most v, so $\widetilde{v} + v - v' \leq v$, i.e. $\widetilde{v} \leq v'$. Since $\widetilde{q}_1, \ldots, \widetilde{q}_n$ was optimal for W', we must have $\widetilde{v} = v'$ and v' is also optimal. □

Exercise 2. Prove that if $q_1, \ldots, q_n$ is an optimal solution to the fractional or 0-1 knapsack problem for a given weight limit W, and $j \leq n$, then $q_1, \ldots, q_j$ is an optimal solution optimal solution for the weights $w_1, \ldots, w_j$ and and values $v_1, \ldots, v_j$, and weight limit $W' := W - \sum_{i=j+1}^{n} q_i w_i$. This shows that these knapsack problems have optimal substructure.

Proof. Let $\widetilde{q}_1, \ldots, \widetilde{q}_j$ be an optimal solution for the subproblem, and define $\hat{q}_i = \begin{cases} \widetilde{q}_i & \text{if } i \leq j \\ q_i; & \text{if } i > j \end{cases}$ for $i = 1, \ldots, n$. Then $\hat{q}_1, \ldots, \hat{q}_n$ has total weight at most W, and has value

$$\sum_{i=1}^{j} \widetilde{q}_i v_i + \sum_{i=j+1}^{n} q_i v_i \;\geq\; \sum_{i=1}^{j} q_i v_i + \sum_{i=j+1}^{n} q_i v_i \;=\; \sum_{i=1}^{n} q_i v_i,$$

where the inequality follows from optimality of the $\widetilde{q}_i$. Since $q_1, \ldots, q_n$ is optimal, the inequality must be an equality, so $q_1, \ldots, q_j$ has the same value as $\widetilde{q}_1, \ldots, \widetilde{q}_j$ and is optimal. □

Exercise 3. Give examples to show that the 0-1 and fractional knapsack problems do not have the optimal substructure property we proved for the unbounded knapsack problem.

Solution. Take $w_1 = 1$, $v_1 = 1$, and $W = 3$. The optimal solution to the 0-1 and fractional knapsack problems is to take item 1, i.e. $q_1 = 1$. Taking $q'_1 = 0$, the unique optimal solution to the subproblem with $W' = W - (q_1 - q'_1)w_1 = 2$ is also $q_1 = 1 \neq q'_1$. □

Exercise 4. Show that the fractional knapsack problem has the greedy-choice property. That is, show that if $i \in \{1, \ldots, n\}$ is such that $\frac{v_i}{w_i}$ is maximal, and $q \in [0, 1]$ is such that $qw_i \leq W$, then there is an optimal solution to the fractional knapsack problem with $q_i \geq q$.

Proof. Let $\widetilde{q}_1, \ldots, \widetilde{q}_n$ be an optimal solution. Since $\frac{v_i}{w_i}$ is maximal, for any j, if $u = \min(1 - w_i\widetilde{q}_i, w_j\widetilde{q}_j)$, then we may replace $\widetilde{q}_j$ with $\widetilde{q}_j - \frac{u}{w_j}$ and $\widetilde{q}_i$ with $\widetilde{q}_i + \frac{u}{w_i}$ without increasing the total weight, nor decreasing the total value. Moreover, after this replacement, either $\widetilde{q}_i = 1$ or $\widetilde{q}_j = 0$.

For each $j \neq i$, we perform this replacement with the largest possible u. Since we started with an optimal solution and did not decrease the value, this results in an optimal solution. Moreover, we either have $\widetilde{q}_i = 1$, or $\widetilde{q}_j = 0$ for all $j \neq i$. In the first case, we are done, since $q \leq 1$. In the second case, $\widetilde{q}_i \leq \frac{W}{w_i}$, and by increasing $\widetilde{q}_i$ to $\min(1, \frac{W}{w_i}) \geq q$ we stay under the weight limit and do not decrease the total value. This is then an optimal solution of the required form. □

Exercise 5. Write a greedy algorithm for the fractional knapsack problem based on the above strategy. What is its running time?

Solution. The greedy-choice property shows that it is optimal to take the largest possible amount of the item with the highest value to weight ratio. The following algorithm uses this strategy:

```
FKP(w, v, W)
 1  n = w.length
 2  let r[1..n] be a new array
 3  let S[1..n] be a new array filled with zeroes
 4  for i = 1 to n
 5       r[i] = (v[i]/w[i], i)
 6  sort r by first elements
 7  i = n
 8  while i ≥ 1 and w[r[i][2]] ≤ W
 9       S[r[i][2]] = 1
10       i = i − 1
11  if i > 0
12       S[r[i][2]] = W/w[r[i][2]]
13  return S
```

We used a trick here to sort the items by value-to-weight ratio while remembering their initial positions; this was the purpose of the second elements of the entries of r.

This algorithm has two loops of length at most n, which together have running time $\Theta(n)$. Sorting the array r takes time $\Theta(n \log n)$, so the overall time complexity is $\Theta(n \log n)$. If the items came pre-sorted, then this is reduced to $\Theta(n)$. □

Exercise 6. Construct an instance of the 0-1 or unbounded knapsack problem for which the optimal solution does not include the item with the highest value to weight ratio.

Solution. Take $w_1 = 4$, $w_2 = 5$, $w_3 = 7$, $v_1 = 3$, $v_2 = 4$, $v_3 = 6$, and $W = 10$. The third item has the highest value per unit weight, but if we take the third item then we cannot take either of the other items, and the optimal choice is to take items 1 and 2. □

Exercise 7. Write a dynamic programming algorithm for the unbounded knapsack problem with running time $O(nW)$.

Solution. Our optimal substructure property shows that we can take one item away from an optimal solution and it will remain an optimal solution. Therefore, the optimal solution is, for some j, the optimal solution for $W - w_j$, plus item j. We can find it by trying all possible j. Since we need to solve smaller problems, we store all of the solutions in an array, which we fill in iteratively.

```
UKP(w, v, W)
 1  n = w.length
 2  create a new array S[0..W]
 3  S[0] = 0
 4  for i = 1 to W
 5       best = 0
 6       for j = 1 to n
 7            if w[j] ≤ W and S[W − w[j]] + v[j] > best
 8                 best = S[W − w[j]] + v[j]
 9       S[j] = best
10  return S[W]
```

This contains a **for** loop of length W, which contains a **while** loop of length at most n, and every line of the algorithm takes constant time. It therefore has running time $O(nW)$.

If we want the actual solution, not just the optimal value, we can modify the algorithm slightly to keep track of which choice of item was optimal for each weight:

```
UKPsol(w, v, W)
 1  n = w.length
 2  create a new array S[0..W]
 3  S[0] = (0, 0)
 4  for i = 1 to W
 5       best = 0
 6       bestitem = 0
 7       for j = 1 to n
 8            if w[j] ≤ W and S[W − w[j]] + v[j] > best
 9                 best = S[W − w[j]][2] + v[j]
10                 bestitem = j
11       S[j] = (bestitem, best)
12  create a new array sol[1..n] filled with zeroes
13  u = W
14  while S[u] ≠ (0, 0)
15       sol[S[u][1]] = sol[S[u][1]] + 1
16       u = u − w[S[u][1]]
17  return sol
```

This algorithm also has running time $O(nW)$, and returns an array containing the optimal quantities of each item. □

Exercise 8. Write a dynamic programming algorithm for the 0-1 knapsack problem with running time $O(nW)$.

Solution. Our optimal substructure property shows that an optimal solution for a 0-1 knapsack problem is either an optimal solution for the first $n-1$ items and the same weight limit W, or the last item together with an optimal solution for the first $n-1$ items and weight limit $W - w_n$. This gives the following dynamic programming algorithm:

```
ZOKP(w, v, W)
 1  n = w.length
 2  create a new array S[0..n, 0..W]
 3  for i = 0 to W
 4       S[0, i] = 0
 5  for j = 1 to n
 6       for i = 0 to W
 7            if i ≥ w[j]
 8                 S[j, i] = max(S[j − 1, i], S[j − 1, i − w[j]] + v[j])
 9            else S[j, i] = S[j − 1, i]
10  return S[n, W]
```

This algorithm takes constant time to fill in each entry of an $(n+1) \times (W+1)$ array, so has running time $\Theta(nW)$.

As above, we can obtain the actual solution by modifying our algorithm to record the optimal choice:

```
ZOKPSOL(w, v, W)
 1  n = w.length
 2  create a new array S[0..n, 0..W]
 3  initialize an empty list sol
 4  for i = 0 to W
 5       S[0, i] = 0
 6  for j = 1 to n
 7       for i = 0 to W
 8            if i ≥ w[j] and S[j − 1, i − w[j]] + v[j] ≥ S[j − 1, i]
 9                 S[j, i] = (S[j − 1, i − w[j]] + v[j], 1)
10            else S[j, i] = (S[j − 1, i], 0)
11  let sol be a new, empty list
12  for j = n downto 1
13       if S[j, W][2] == 1
14            append j to sol
15            W = W − w[j]
16  return sol
```

This algorithm likewise has running time $O(nW)$, and returns a list containing the indices of the optimal items to choose. □

2. No late homework accepted

You have n homework assignments to complete, with due dates $d_1, \ldots, d_n$. The ith homework is worth a_i points if completed by the due date, and zero otherwise. Each assignment will take you one day to complete. You wish to find an order in which to do your homework so as to maximize your total points.

Exercise 9. Show that this problem has optimal substructure with respect to choosing the first $k \leq n$ assignments. That is, if a given ordering is optimal for all n assignments, then the ordering of the first k is optimal for those k assignments.

Proof. If there were a better ordering of those k, we could use that ordering, and the same ordering of the rest of the assignments, and obtain a better overall solution. More precisely, let $\sigma = \sigma_1 \sigma_2 \ldots \sigma_n$

be a permutation of $1, \ldots, n$ which gives an optimal order in which to do the assignments, and let $\tau = \tau_1 \ldots \tau_k$ be a permutation of $\sigma_1 \ldots \sigma_k$ which is optimal among all orderings of $\sigma_1 \ldots \sigma_k$. Then the difference between the point values of $\tau_1 \ldots \tau_k \sigma_{k+1} \ldots \sigma_n$ and $\sigma_1 \ldots \sigma_n$ is equal to the difference between $\tau_1 \ldots \tau_k$ and $\sigma_1 \ldots \sigma_k$. Since τ was optimal, this difference is nonnegative; since σ was optimal, the difference is nonpositive. Thus they are equal and $\sigma_1 \ldots \sigma_k$ is optimal. □

Exercise 10. Write a greedy algorithm to solve this problem, and determine its running time.

Solution. Our strategy will be to start with the highest-value assignment, and schedule it to be done as late as possible before its due date. We choose the latest possible time because an earlier time might crowd out assignments with earlier due dates: given a schedule which places it earlier, we can swap it with the assignment done at the later time. This swap cannot decrease our total points, and might increase them. Having scheduled the highest-value assignment, we repeat with the second-highest value, third-highest, and so on.

Let's give a formal proof that this produces an optimal solution. For simplicity of notation, we assume that the highest-value assignment is the first one, with due date d_1. We first need to show that there is an optimal solution with assignment 1 done on day d_1. This is straightforward: given an optimal solution, swap assignment 1 and whichever assignment j is scheduled for day d_1. If assignment 1 was scheduled later than d_1, then the new schedule gains a_1 points from having assignment 1 be on time, while losing at most $a_j \leq a_1$ from potentially making assignment j late. If assignment 1 was scheduled earlier, then we lose no points, and potentially gain a_j points from moving assignment j earlier. In either case, the new schedule is at least as good.

Now we show that we can recursively apply this strategy. This uses a somewhat more complicated optimal substructure property than the one we proved above. By fixing a homework assignment to do an a given day, we effectively remove that day from consideration for further scheduling. Any assignments due that day may as well be due the previous day, and we can shift later days forward to close the gap. Therefore, after having scheduled assignment 1 for day d_1, we define a new set of due dates $d'_2, \ldots, d'_n$, by $d'_i = \begin{cases} d_i & \text{if } d_i < d_1 \\ d_i - 1 & \text{if } d_i \geq d_1 \end{cases}$. Given a solution $\sigma'_2 \ldots \sigma'_n$ to this scheduling problem, we set $\sigma_1 = d_1$ and $\sigma_i = \begin{cases} \sigma'_i & \text{if } \sigma_i < d_i \\ \sigma'_i + 1 & \text{if } \sigma_i \geq d_i \end{cases}$. Then $\sigma_1 \ldots \sigma_n$ is a solution to the original scheduling problem, with point value a_1 greater than that of $\sigma'_2 \ldots \sigma'_n$. This implies that an optimal choice of $\sigma'_2 \ldots \sigma'_n$ will give an optimal solution $\sigma_1 \ldots \sigma_n$.

We now need to implement this strategy in pseudocode. Instead of changing due dates and adjusting schedules for each recursive step, we track which days are still available with markers in an array. This is both easier to implement and conceptually simpler; the other method was only needed for the formalism of optimal substructure. (We could also have expanded the problem by allowing certain dates to be unavailable for homework, and proved the corresponding optimal substructure property for that problem.) The following pseudocode does this:

```
DeadlineSchedule(d, a)
 1  sort a, and rearrange d to match
 2  n = a.length
 3  let S[1..n] be a new array, filled with zeroes
 4  Let Q be a new queue
 5  for i = n downto 1
 6        j = d[i]
 7        while j ≥ 1 and S[j] ≠ 0
 8              j = j − 1
 9        if j > 0
10              S[j] = i
11        else Enqueue(Q, i)
12  for k = 1 to n
13        if S[k] == 0
14              S[k] = Dequeue(Q)
15  return S
```

This has running time $O(n^2)$. It can be made faster by using a data structure which allows faster scanning for empty slots. The returned schedule uses the sorted list of assignments, but the algorithm can be modified to return indices for the original list, with the same trick we used for the knapsack problem. □