Exercises

Please do the exercises on your own.

1. (2 pt.) Consider the graph $G$ below.

(a) (1 pt.) In what order does Prim’s algorithm add edges to an MST when started from vertex C?

(b) (1 pt.) In what order does Kruskal’s algorithm add edges to an MST?

[We are expecting: For both, just a list of edges. You do not need to draw the MST, and no justification is required.]

SOLUTION:

(a) Prim’s algorithm adds edges in the order:
   
   \{C,F\}, \{F, E\}, \{E, D\}, \{A, D\}, \{A, B\}

(b) Kruskal’s algorithm returns the same tree, and adds edges in the order:
   
   \{A,D\}, \{A, B\}, \{D, E\}, \{F, C\}, \{E,F\}
2. (6 pt.) In this exercise we’ll look at a continuous variant of the knapsack problem that we saw in class. You have a knapsack with a capacity of $Q$ ounces and there are $n$ items; the difference between this exercise and the version that we saw in class is that you can take a fractional amount of each item. For example, perhaps one item is 3.6 ounces of brightly colored sand; you can choose to take 2.5235 ounces of sand for your knapsack if that’s how much you want.

Each item $i$ has a value per ounce $v_i > 0$ (measured in units of dollars per ounce) and a quantity $q_i > 0$ (measured in ounces). There are $q_i$ ounces of item $i$ available to you, and for any real number $x \in [0, q_i]$, the total value that you derive from $x$ ounces of item $i$ is $x \cdot v_i$.

Your goal is to choose an amount $x_i \geq 0$ to take for each item $i$ in order to maximize the value $\sum_i x_i v_i$ that you receive while satisfying:

1. you don’t overfill the knapsack (that is, $\sum_i x_i \leq Q$), and
2. you don’t take more of an item than is available (that is, $0 \leq x_i \leq q_i$ for all $i$).

Assume that $\sum_i q_i \geq Q$, so there always is some way to fill the knapsack.

(a) (0 pt.) Suppose that you already have partially filled your knapsack, and there is some amount of each item left. What item should you take next, and how much?

[We are expecting: Nothing, this part is worth zero points, but it’s a good thing to think about before you go on to the next part.]

(b) (3 pt.) Design a greedy algorithm which takes as input $Q$ along the tuples $(i, v_i, q_i)$ for $i = 0, \ldots, n-1$, and outputs tuples $(i, x_i)$ so that (1) and (2) hold and $\sum_i x_i v_i$ is as large as possible. Your algorithm should take time $O(n \log(n))$.

[We are expecting:
- Pseudocode AND an English explanation of what it is doing.
- A justification of the running time.]

(c) (3 pt.) Fill in the inductive step below to prove that your algorithm is correct.

- **Inductive hypothesis:** After making the $t$'th greedy choice, there is an optimal solution that extends the solution that the algorithm has constructed so far.
- **Base case:** Any optimal solution extends the empty solution, so the inductive hypothesis holds for $t = 0$.
- **Inductive step:** (you fill in)
- **Conclusion:** At the end of the algorithm, the algorithm returns a set $S^*$ of tuples $(i, x_i)$ so that $\sum_i x_i = Q$. Thus, there is no solution extending $S^*$ other than $S^*$ itself. Thus, the inductive hypothesis implies that $S^*$ is optimal.

[We are expecting: A proof of the inductive step: assuming the inductive hypothesis holds for $t - 1$, prove that it holds for $t$.]

**SOLUTION:**

(a) I would always take the item worth the most per ounce, until it is gone!

(b) We greedily take item with the most value per ounce:

```python
# assume that they are ordered so that item 0 has the largest value-per-weight
v_0 = 0
items = []
for i in range(n): # assume that they are ordered so that item 0 has the largest value-per-weight v_0
    if q_i < roomLeft:
        # assume that they are ordered so that item 0 has the largest value-per-weight v_0
        v_0 = v_0
```
items.append((i, q_i))
roomLeft = roomLeft - q_i
else:
    items.append((i, roomLeft))
    break
return items

(c) For the inductive step, assume that we have just made the $t$'th greedy choice we have taken a set $S_t$ of $(i, x_i)$ tuples and there is some optimal solution $S^*$ so that $S_t \subseteq S^*$. Now suppose we choose the next item greedilly: suppose it is $(i, x_i)$. Let $y_i$ be the amount of $i$ that appears in $S^*$ (so $y_i$ may be anything in $[0, q_i]$).

If $y_i \geq x_i$, then we are done, since $S^*$ extends $S_{t+1}$. (That is, we can make $S^*$ by adding stuff to $S_{t+1}$).

On the other hand, suppose that $y_i < x_i$. Then the situation looks like this:

\[
\begin{array}{ccc}
S_{t+1}: & S_t & x_i \text{ oz of } i \\
S^*: & S_t & y_i \text{ oz of } i & \text{Other stuff}
\end{array}
\]

Call this stuff $A$

Let $A$ be $x_i - y_i$ ounces of stuff from $S^*$ that is not item $i$, and also not in $S_t$. (See diagram). Then the total value of the stuff in $A$ is at most $(x_i - y_i)v_i$, since all of the items counted in $A$ are not more valuable (per ounce) than item $i$ – otherwise we would have chosen one of those items instead of item $i$.

Now consider the assignment $S^{**}$ that we get by switching $A$ for $(x_i - y_i)$ ounces of item $i$ in $S^*$. By the above, this cannot decrease the value, so $S^{**}$ is still optimal. But now $S^{**}$ contains $x_i$ ounces of item $i$, and so it extends the choices we have made so far.
3. (6 pt.) [k-well-connected graphs.] Let \( G = (V, E) \) be an undirected, unweighted graph with \( n \) vertices and \( m \) edges. For a subset \( S \subseteq V \), define the subgraph induced by \( S \) to be the graph \( G' = (S, E') \), where \( E' \subseteq E \), and an edge \( \{u, v\} \in E \) is included in \( E' \) if and only if \( u \in S \) and \( v \in S \).

For any \( k < n \), say that a graph \( G \) is \( k \)-well-connected if every vertex has degree at least \( k \).

For example, in the graph \( G \) below, the subgraph \( G' \) induced by \( S = \{a, b, c, d\} \) is shown on the right. \( G' \) is 3-well-connected, since every vertex in \( G' \) has degree at least 3. However, \( G \) is not 3-well-connected since vertex \( E \) has degree 2.

Design a greedy algorithm to find a maximal set \( S \subseteq V \) so that the subgraph \( G' = (S, E') \) induced by \( S \) is \( k \)-well-connected. In the example above, if \( k = 3 \), your algorithm should return \( \{a, b, c, d\} \), and if \( k = 4 \) your algorithm should return the empty set.

You may assume that your representation of a graph supports the following operations:

- \texttt{degree(v)}: return the degree of a vertex in time \( O(1) \)
- \texttt{remove(v)}: remove a vertex and all edges connected to that vertex from the graph, in time \( O(\text{degree}(v)) \).

Your algorithm should run in time \( O(n^2) \).

You do not need to prove that your algorithm works, but you should give an informal (few sentence) justification.

[HINT: Think about greedily removing vertices.]

[We are expecting:

- Pseudocode AND an English description of what your algorithm is doing.
- An informal justification of the running time.
- An informal justification that the algorithm is correct.]
**SOLUTION:** Our algorithm will greedily remove any vertex with degree less than \( k \), along with all the edges attached to it:

```python
myGreedyAlg(G = (V,E), k):
    while True:
        for v in V:
            if G.degree(v) < k:
                G.remove(v)
                break
        else:
            # if we did not break
            break
    return vertices(G)
```

The running time is at most \( O(n^2) \) because the while loop executes at most \( n \) times (since each time it removes a vertex and there are only \( n \) vertices), and within each iteration of the while loop, we loop over all vertices \( v \in V \) for another factor of \( n \). Each time the inner for-loop runs, we additionally remove a single vertex \( v \), which takes time \( O(deg(v)) \). Since the degree of \( v \) is at most \( k \) if it is getting removed, this takes time at most \( O(k) \). Thus, the inner for-loop takes time \( O(n + k) = O(n) \). Altogether the running time is the \( O(n \times n) = O(n^2) \).

Intuitively, the algorithm works because we are only removing vertices that are “safe” to remove. That is, no optimal solution could have a vertex of degree less than \( k \) in it, so by removing a vertex of degree less than \( k \), we haven’t ruled out success.

4. (10 pt.) **[Fish Stops.]** Plucky the Pedantic Penguin is walking \( t \) miles across Antarctica. He needs to eat along the way, but he can only eat when there’s a fishing hole for him to catch fish. He can walk at most \( m \) miles between meals, and he knows how \( n \) fishing holes are laid out along his route.

Plucky is given an array \( F \) so that \( F[i] \) gives the distance from the start of his journey to the \( i \)'th fishing hole. There are \( n \) fishing holes along the way, including at the beginning and the end: \( F[0] = 0, F[n - 1] = t \). For example, the array \( F = [0, 3, 4, 6, 10, 12] \), with \( t = 12 \) corresponds to the setup below:

```
         >   >   >   >   >   >
          | | | | | | |
          0 3 4 6 10 12
```

Plucky wants to stop as few times as possible, given that he can walk at most \( m \) miles without eating. (It is okay if he walks exactly \( m \) miles between meals). He starts out hungry, so he will always fish at 0 miles; he will also always fish at his destination (at \( t \) miles), whether or not he’s hungry.

In the example above, if \( m = 4 \), then Plucky should stop 5 times (including his stops at the beginning and the end), for example at 0, 4, 6, 10, 12 miles.

(a) (4 pt.) Design a greedy algorithm for Plucky to use. The algorithm should have the following properties:

- Your algorithm should take as input the array \( F \), as well as the parameters \( m \) and \( t \). You may assume that \( F \) is sorted.
• Your algorithm should output a list `fishStops` which contains a shortest list of places that Plucky could stop for fish. In the example above, the algorithm could output `[0, 4, 6, 10, 12]`. If Plucky cannot make it to his destination `t` miles away, then your algorithm should return `Stay Home`.

• Your algorithm should run in time $O(n)$.

[We are expecting: Pseudocode AND an English description of what it is doing. You do not need to justify the running time.]

(b) (6 pt.) Prove by induction that your algorithm is correct. You may assume that there is a way for Plucky to make it `t` miles (aka, the algorithm won’t return `Stay Home`) if it’s easier.

[We are expecting: A formal proof by induction. Be sure to clearly state your inductive hypothesis, base case, inductive step, and conclusion.]

SOLUTION:

(a) The basic idea is as follows. Suppose that Plucky has planned his first $j$ stops, and the last one is $x$. Then the algorithm will return the stop $y$ ∈ $F$ so that $y - x \leq m$. That is, $y$ is the farthest that Plucky can make it from $x$.

Here is the pseudocode:

```python
scheduleFishStops( F, m, t ):
    n = length(F); assert F[0] = 0 and F[n-1] = t
    fishStops = [ F[0] ]
    lastMeal = F[0]
    for i = 1,...,n-2:
        if F[i] - lastMeal <= m and F[i+1] - lastMeal > m:
            fishStops.append( F[i] )
            lastMeal = F[i]
    if t - lastMeal > m:
        return "Stay home"
    else:
        fishStops.append(t)
    return fishStops
```

(b) Say that an array $S$ of length $r$ is a feasible schedule if $S$ is a sorted array containing $r$ elements of $F$ so that $S[0] = 0$, $S[r-1] = t$, and for all $i \in \{1, \ldots, r-1\}$, $S[i] - S[i-1] \leq m$.

Thus, we want to show that `scheduleFishStops(F,m,t)` returns a shortest feasible schedule. As the problem says we can, we assume there exists a feasible schedule. We prove this by induction.

• **Inductive Hypothesis:** Before the iteration-$i$ loop begins, there is a shortest feasible schedule $S^*$ that extends `fishStops`.

• **Base Case:** Before the $i = 1$ loop begins, `fishStops = [F[0]] = [0]`. Since by definition any feasible schedule starts with 0, any optimal feasible schedule $S^*$ must extend `fishStops`, which establishes the inductive hypothesis for $i = 1$.

• **Inductive step:** Suppose that the inductive hypothesis holds for $i = j$, and we want to show it holds for $i = j + 1$.

Let $J$ be the version of `fishStops` before iteration $j$. Let $S^*$ be an optimal feasible schedule extending $J$, which exists by our inductive hypothesis.

Now, two things can happen in iteration $j$. Either we add $F[j]$, or else we don’t.

- If we don’t add $F[j]$, then the inductive hypothesis is satisfied for $i = j + 1$ because at the beginning of iteration $j + 1$ we still have `fishStops = J`, which is extended by $S^*$.
On the other hand, suppose we do add $F[j]$ in iteration $j$, so before iteration $j+1$, \texttt{fishStops} looks like $J' = J + [F[j]]$.
Let $\ell$ be the largest index so that $S^*[\ell] < F[j]$. Notice that $S^*[\ell]$ does not appear in $J$. This is because $F[j]$ was chosen to be the last fish stop that Plucky could reach, so since the schedule $S^*$ stops at the last stop in $J$ and does not stop at $F[j]$, it must stop somewhere in between the last stop of $J$ and $F[j]$.
Now consider $S^{**}$ which is obtained from $S^*$ by replacing the fish stop $S^*[\ell]$ with $F[j]$. The picture looks like this:

\[
\begin{array}{c}
J \quad \quad \quad \quad \quad \quad \quad F[j] \\
S^* \quad \quad \quad \quad \quad \quad \quad S^*[\ell] \quad \quad S^*[\ell+1] \\
S^{**} \quad \quad \quad \quad \quad \quad \quad S^*[\ell] \quad \quad S^*[\ell+1]
\end{array}
\]

$S^{**}$ is still a feasible schedule, because Plucky can get from the last stop in $J$ to $F[j]$ (since the greedy algorithm chose $F[j]$ so that this was possible), and Plucky can get from $F[j]$ to the next stop in $S^{**}$ (which is $S^*[\ell+1]$), because Plucky could get from $S^*[\ell]$ to $S^*[\ell+1]$ and $F[j]$ is by construction in between $S^*[\ell]$ and $S^*[\ell+1]$.
Finally, $S^{**}$ is not any longer than $S^*$ (since we added $F[j]$ and removed $S^*[\ell]$), so it must also be a shortest feasible schedule. This proves the inductive step in the second case.

\textbf{Conclusion:} We conclude that the inductive step holds for all iterations. In particular, at the end of the algorithm, \texttt{fishStops} is an array so that $\texttt{fishStops[-1]}$ is $t$, and so that there is some optimal feasible schedule $S^*$ that extends \texttt{fishStops}. But since $S^*$ also has to end in $t$, this means that \texttt{fishStops} is equal to $S^*$ and thus is optimal.
5. [Minimum-maximum spanning trees.] (6 pt.) Let $G$ be a connected weighted undirected graph. In class, we defined a minimum spanning tree of $G$ as a spanning tree $T$ of $G$ which minimizes the quantity

$$X = \sum_{e \in T} w_e,$$

where the sum is over all the edges in $T$, and $w_e$ is the weight of edge $e$. Define a “minimum-maximum spanning tree” to be a spanning tree that minimizes the quantity

$$Y = \max_{e \in T} w_e.$$

That is, a minimum-maximum spanning tree has the smallest maximum edge weight out of all possible spanning trees.

(a) (2 pt.) Given an example of a graph $G$ which has a minimum-maximum spanning tree $T$ so that $T$ is not a minimum spanning tree.

[We are expecting: An example, with an informal explanation of why it is an example.]

(b) (4 pt.) Prove that a minimum spanning tree in a connected weighted undirected graph $G$ is always a minimum-maximum spanning tree for $G$.

[HINT: Suppose toward a contradiction that $T$ is an MST but not a minimum-maximum spanning tree, and say that $T'$ is a minimum-maximum spanning tree. How can you use $T'$ to modify $T$, to come up with a cheaper MST than $T$ (and hence a contradiction)? (Sub-hint: consider the heaviest edge in $T$).]

[We are expecting: A formal proof.]

SOLUTION:

(a) Consider a triangle on vertices $A, B, C$ so that $(A, B) = 1$, and $(A, C), (B, C) = 2$. Then $(A, C), (B, C)$ is a spanning tree with minimal $Y$-value but not minimal $X$-value.

(b) Suppose that $T$ is an MST that is not an MMST. Following the hint, let $T'$ be an MMST. Let $(u, v)$ be the heaviest edge in $T$. Consider removing $(u, v)$ from $T$, to obtain two trees $T_u$ and $T_v$. Now consider the cut between $V(T_u)$ and $V(T_v)$, where $V(T_v)$ means the vertices in $T_v$. There must be some edge in $T'$ that crosses this cut, call it $(x, y)$. Notice that $w(x, y) < w(u, v)$. This is because $(x, y)$ is in an MMST, and so if $w(u, v) \leq w(x, y)$, $T$ would also have been an MMST since $(u, v)$ was its heaviest edge. Now consider the tree $T''$ obtained by connecting $T_u$ and $T_v$ with the edge $(x, y)$. This is still a tree, since $T_u$ and $T_v$ were not connected to each other so adding $(x, y)$ can’t have made a cycle. It spans, because $T_u$ and $T_v$ already touched all the vertices. Finally, the cost of $T''$ is smaller than the cost of $T$, since we switched out a lighter edge for a heavier edge. This is a contradiction of the assumption that $T$ was an MST.
6. (NOT REQUIRED, WORTH ONE BONUS pt.) [Another activity selection algorithm?]

In class, we considered an alternative greedy algorithm for activity selection. The idea was that at each step, we greedily add a valid activity with the fewest conflicts with other valid activities. (An activity is valid if it doesn’t conflict with an already selected activity).

For example, if the activities looked like:

```
\begin{array}{cccccccc}
a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\
0 & 3 & 3 & 3 & 3 & 1 & 1 & 2
\end{array}
```

then the number of conflicts to begin with are:

The algorithm (breaking ties arbitrarily) could choose $a_1$, then $a_6$, then $a_7$, then $a_2$.

Is this algorithm correct?

**We are expecting:** To get the bonus point, give either a counterexample or a formal proof of correctness.

- If you give a proof by induction, make sure to clearly state your inductive hypothesis, base case, inductive step and conclusion. (Note, in this case you should show that the algorithm is correct no matter how it breaks ties).
- If you give a counterexample, it should be a drawing like the one above; you can either draw it by hand or use your favorite software. You should also explain what this algorithm does on your counter-example and why it is not optimal. (Note, in this case it is okay to give an example where there is some way of breaking ties so that the algorithm messes up).

**SOLUTION:** It doesn’t work. Here’s a counter-example:

```
\begin{array}{cccc}
a_1 & a_2 & a_3 & a_4 \\
a_5 & a_6 & a_7 & a_8 \\
a_9 & a_10 & a_11
\end{array}
```

The greedy approach would choose $a_1$ and then it could choose $a_{10}$ and $a_{11}$. (It could also choose $a_1$ and then $a_6$ and $a_9$, or many other options – but it would always choose a solution of size three). However, the optimal solution is $a_2, a_3, a_4, a_5$ which has size four.