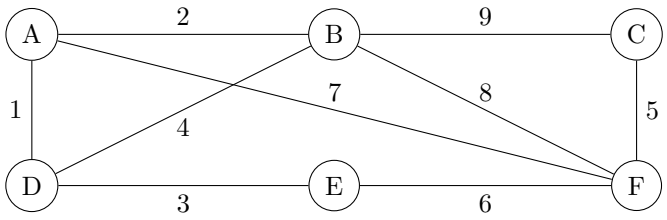
Exercises

Exercises should be completed **on your own.**

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



   (a) **(1 pt.)** What MST does Prim's algorithm return? In what order does Prim's algorithm add edges to the MST when started from vertex C?

   (b) **(1 pt.)** What MST does Kruskal's algorithm return? In what order does Kruskal's algorithm add edges to the MST?

   [We are expecting: For both, just a list of edges. 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. **(4 pt.)** At a Thanksgiving dinner, there are n food items $f_0, \ldots, f_{n-1}$. Each food item has a tastiness $t_i > 0$ (measured in units of deliciousness per ounce) and a quantity $q_i > 0$ (measured in ounces). There are q_i ounces of food f_i available to you, and for any real number $x \in [0, q_i]$, the total deliciousness that you derive from eating x ounces of food f_i is $x \cdot t_i$. (Notice that x here doesn't have to be an integer).

   Unfortunately, you only have capacity for Q ounces of food in your belly, and you would like to maximize deliciousness subject to this constraint. Assume that there is more food than you can possibly eat: $\sum_i q_i \geq Q$.

   (a) **(2 pt.)** Design a greedy algorithm which takes as input the tuples (f_i, t_i, q_i), and outputs tuples (f_i, x_i) so that $0 \leq x_i \leq q_i$, $\sum_i x_i \leq Q$, and $\sum_i x_i t_i$ is as large as possible. Your algorithm should take time $O(n \log(n))$.

   [We are expecting: Pseudocode and a short English explanation.]

   (b) **(2 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 an set S^* of tuples (f_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) We greedily take the most tasty foods first:

```
Sort the foods by tastiness:  assume that f_0 is the most tasty food.
roomLeft = Q
foods = []
for i in range(n):
    if q_i < roomLeft:
        foods.append((f_i, q_i))
        roomLeft = roomLeft - q_i
    else:
        foods.append((f_i,roomLeft))
        break
    return foods
```

We note that this model of Thanksgiving dinner isn't quite right, because of course there is always room for dessert.

(b) For the inductive step, assume that we have just made the t'th greedy choice we have taken a set S_t of (f_i, x_i) tuples and there is some optimal solution S^* so that $S_t \subseteq S^*$. Now suppose we choose the next food greedily: suppose it is f_i, x_i.
Let y_i be the amount of f_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 $S_{t+1} \subseteq S^*$. on the other hand, suppose that $y_i < x_i$. Notice that $x_i \leq roomLeft$ in either case of the if-else statement, so this implies that $y_i < roomLeft$. In that case, there are at least $roomLeft - y_i$ units of foods f_j for $j > i$ (that is, less tasty foods). Consider swapping these $roomLeft - y_i$ units of less tasty foods for $roomLeft - y_i$ units of f_i in S^*, to create S^{**}. This cannot decrease tastiness, so if S^* was optimal then S^{**} is also optimal.
But now S^{**} is an optimal solution so that $S_{t+1} \subseteq S^{**}$ which is what we wanted to show.

Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(6 pt.)** Consider the problem of **making change.** Suppose that coins come in denominations $P = \{p_0, \ldots, p_m\}$ cents (for example, in the US, this would be $P = \{1, 5, 10, 25\}$, corresponding to pennies, nickels, dimes, and quarters). Given n cents (where n is a non-negative integer), you would like to find the way to represent n using the fewest coins possible. For example, in the US system, 55 cents is minimally represented using three coins, two quarters and a nickel.

   (a) **(3 pt.)** Suppose that the denominations are $P = \{1, 10, 25\}$ (aka, the US ran out of nickels). Your friend uses the following greedy strategy for making change:

Algorithm 1: makeChange(n)

Input: n and P
coins = [ ];
while $n > 0$ **do**
    $p^* \leftarrow \max\{p \in P : p \leq n\}$;
    $n = n - p^*$;
    coins.append(p^*) ;
return *coins*

   Your friend acknowledges that this won't work for general P (for example if $P = \{2\}$ then we simply can't make any odd n), but claims that for this particular P it does work. That is, your friend claims that this algorithm will always return a way to make n out of the denominations in P with the fewest coins possible.

   Is your friend correct for $P = \{1, 10, 25\}$?

   [We are expecting: Your answer, and either a proof or a counterexample. If you do a proof by induction, make sure to explicitly state your inductive hypothesis, base case, inductive step, and conclusion.]

   (b) **(3 pt.)** Your friend says that additionally their algorithm should work for any P of the form $P = \{1, 2, 4, 8, \ldots, 2^s\}$.

   Is your friend correct for $P = \{1, 2, 4, \ldots, 2^s\}$?

   [We are expecting: Your answer, and either a proof or a counterexample. If you do a proof by induction, make sure to explicitly state your inductive hypothesis, base case, inductive step, and conclusion.]

   SOLUTION:

   (a) This is false. For example, to make $n = 30$ this algorithm would return $\{25, 1, 1, 1, 1, 1\}$, while the best way is $\{10, 10, 10\}$.

   (b) This is true. To see this, we do a proof by induction.

- **Inductive hypothesis.** After choosing t coins, there is a minimal solution that extends the current solution.
- **Base case.** When $t = 0$, there is clearly a minimal solution that extends the current (empty) solution.
- **Inductive Step.** Suppose that we have chosen $t - 1$ coins and are about to choose the t'th. Let S be the multiset of coins so far, and let $x = \sum_{s \in S} s$ be the sum we have so far. By the inductive hypothesis there is a minimal solution extending S, suppose that it's $S \sqcup T$ (here, the $\sqcup$ means "multiset disjoint union" or concatenation), so that $\sum_{t \in T} t = n - x$. Suppose toward a contradiction that T does not contain 2^r, where $r \leq s$ is the largest r so that $2^r \leq n - x$. (That is, r is the choice that the greedy algorithm makes). We claim that there must be some $\ell < r$ so that T contains two copies of 2^ℓ. To see this, notice that otherwise we'd have

$$\sum_{t \in T} t \leq \sum_{\ell < r} 2^\ell = 2^r - 1 < n - x,$$

  which contradicts the fact that $\sum_{t \in T} = n - x$. But then consider T' which is formed from T by replacing the two copies of 2^ℓ with one copy of $2^{\ell+1}$. Then $S \sqcup T'$ is a way to make change of n, but it involves fewer coins than $S \sqcup T$ did. This contradicts the definition of $S \sqcup T$ as an optimal solution.
- **Conclusion.** After considering the smallest coin, we have found a way to make change of n. (This is true for many reasons, but an easy one is because $1 \in P$, so the algorithm will always be able to choose some p^* until n is zero). By invoking the inductive hypothesis at the end of the algorithm, we conclude that there is a minimal solution extending the current solution. Since the only solution extending the current solution is the current solution (adding any more coins would cause it not to be a solution anymore), the current solution must be minimal.

2. **(7 pt.)** On Thanksgiving day, you arrive on an island with n turkeys. You've already had thanksgiving dinner (and maybe you prefer tofurkey anyway), so you don't want to eat the turkeys; but you do want to wish them all a Happy Thanksgiving. However, the turkeys each have very different sleep schedules. Turkey i is awake only in a single closed interval $[a_i, b_i]$. Your plan is to stand in the center of the island and say loudly "Happy Thanksgiving!" at certain times $t_1, \ldots, t_m$. Any turkey who is awake at one of the times t_j will hear the message. It's okay if a turkey hears the message more than once, but you want to be sure that every turkey hears the message at least once.

   (a) **(3 pt.)** Design a greedy algorithm which takes as input the list of intervals $[a_i, b_i]$ and outputs a list of times $t_1, \ldots, t_m$ so that m is as small as possible and so that every turkey hears the message at least once. Your algorithm should run in time $O(n \log(n))$.
   [We are expecting: Pseudocode and an English description of the main idea of your algorithm, as well as a short justification of the running time.]

   SOLUTION: The high-level idea of the algorithm is as follows: First, the algorithm sorts the intervals by end time. Then, it goes through the intervals in order: if b_i is the next end time, and Turkey i has not heard the message, then we broadcast happy thanksgiving at time t_i.
   Naively, the idea above would take time $O(n^2)$, since for each endpoint processed we'd have to look through the list of times t_j output so far and see if the current turkey has heard the message. However, we can make it run in time $O(n \log(n))$ (dominated by the time to sort the intervals) by not actually looking through the list, but just keeping track of them as we go. The pseudocode is as follows.

```
def tellTurkeys(awakeTimes):
    # awakeTimes is a list [ [a_0,b_0], ..., [a_{n-1}, b_{n-1}] ]
    sort awakeTimes based on the b_i's, with the smallest first.
    tellTurkeyTimes = []
    while len(awakeTimes) > 0:
        wake, sleep = awakeTimes.pop(0)
        tellTurkeyTimes.append(sleep)
        while len(awakeTimes) > 0 and awakeTimes[0][0] <= sleep:
            awakeTimes.pop(0)
    return tellTurkeyTimes
```

   (b) **(4 pt.)** Prove that your algorithm is correct.
   [We are expecting: A proof by induction]

   SOLUTION: To prove that the algorithm is correct, formally we proceed by induction.

   - **Inductive Hypothesis.** When we add the t'th broadcast time, there is an optimal solution that extends the current solution.
   - **Base Case.** After we've added no broadcast times, there is an optimal solution which extends the empty solution.
   - **Inductive step.** Suppose that we've added t broadcast times $t_0, \ldots, t_{t-1}$, and assume by induction that this extends to an optimal solution. We would like to show that there is an optimal solution extending $t_0, \ldots, t_t$, where t_t is the time we'll choose next.
     Let $T^* = [t_0, t_1, \ldots, t_{t-1}, s_t, s_{t+1}, \ldots, s_m]$ be the optimal solution extending $t_0, \ldots, t_{t-1}$.
     First, we claim that $s_t \leq t_t$. To see this, suppose toward a contradiction that $s_t > t_t$. But then the turkey whose naptime was t_t would never hear the message (since we chose t_t to be the end time of a turkey who hasn't yet heard the message), so T^* wouldn't have been a valid solution. Thus, $s_t \leq t_t$ as claimed.
     Next we argue that t_t is just a good a choice as s_t, in the sense that the schedule

$$T^{**} = [t_0, \ldots, t_{t-1}, t_t, s_{t+1}, \ldots, s_m]$$

is still an optimal solution. Since T^* and T^{**} have the same length, we only need to show that T^{**} is a legitimate solution, meaning that all turkeys hear the message. Suppose that there's some turkey that does not hear the message in T^{**}. But then this turkey must have (a) have not already heard the message during $t_0, \ldots, t_{t-1}$, and (b) heard the message at s_t and (c) not heard the message at t_t. In particular, this turkey's interval (a_i, b_i) satisfies $a_i \leq s_t \leq b_i < t_t$. However, this contradicts the choice of t_t, which was the smallest naptime of an un-notified turkey; if $s_t \leq b_i < t_t$, then we should have chosen b_i instead of t_t in the algorithm above. Thus, every turkey hears the message in T^{**}, so T^{**} is indeed an optimal turkey-telling-timetable.
This implies that there is an optimal schedule, T^{**}, which extends $t_0, \ldots, t_t$, which is what we wanted to show.

- **Conclusion.** At the end of the algorithm, there is an optimal solution which extends the current one. Since the current one notifies all turkeys, any solution that strictly extends it (adding more broadcast points) would be sub-optimal, so the current solution must be optimal.

3. **(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) **(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. Let (u, v) be the heaviest edge in T. Then deleting (u, v) from T will disconnect it into two trees, A and B. Now use A and B and T' to come up with a cheaper MST than T (and hence a contradiction).]

[We are expecting: A proof.]

(b) **(2 pt.)** Show that the converse to part (a) is not true. That is, a minimum-maximum spanning tree is not necessarily a minimum spanning tree.

[We are expecting: A counter-example, with an explanation of why it is a counter-example.]

(c) **(1 bonus pt.)** Give a deterministic $O(m)$ algorithm to find a minimum-maximum spanning tree in a connected weighted undirected graph G with n vertices and m edges.

[We are expecting: Pseudocode, along with an English description of the idea of the algorithm, and an informal justification of correctness and running time.]

SOLUTION:

(a) Suppose that T' is a MST. We have to show that T' minimizes Y as well. Suppose toward a contradiction that this is not true, and there is some other spanning tree T that has a smaller Y-value than T'. Suppose that the edge with the maximum weight in T' is (x, y). Then $(x, y) \notin T$ (because all of the edge weights in T are smaller than that of (x, y).

Now, (x, y) separates T' into two subtrees: if we remove (x, y) from T', we are left with two subtrees, T'_x which contains x and T'_y which contains y. Let V_x be the vertices in T'_x, and V_y in T'_y. Notice that V_x and V_y are disjoint, and $V_x \cup V_y = V$, the whole vertex set.

Now consider T, the tree with smaller Y-value. It's a spanning tree, so it must connect V_x and V_y. In particular, there is some edge $(u, v) \in T$ so that $u \in T_x$ and $v \in T_y$. Notice that $w(x, y) > w(u, v)$, because every edge in T has weight $< w(x, y)$.

Now consider the tree T'' which is obtained form T' by removing (x, y) and adding (u, v). That is, $T'' = T'_x \cup T'_y \cup (u, v)$. We claim that this is still a spanning tree:

- It still spans. To see this, notice that we haven't changed the vertices that are touched by the tree at all.
- It has no cycles. To see this, suppose that there were a cycle C in T''. Then it must involve the new edge (u, v), since there were no cycles in T'. But then consider the cut (V_x, V_y) that we found above. The edge (u, v) crosses this cut, so there must be some other edge in C, and hence in T'', that crosses this cut. But by the definition of $T'' = T'_x \cup T'_y \cup (u, v)$, there is no other edge that crosses this cut, so there can't have been a cycle to begin with.

Now that we've established that T'' is still a spanning tree, notice that it has cost strictly less than T' did. This is because we swapped out a heavier edge, (x, y), for a lighter edge, (u, v). But that's a contradiction, since T' was an MST.

(b) 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 tree with minimal Y-value but not minimal X-value.

(c) See here: `https://en.wikipedia.org/wiki/Minimum_bottleneck_spanning_tree#Camerini.27s_algorithm_for_undirected_graphs`

Note: a link to the Wikipedia page is not sufficient to get the bonus point on this problem (but it is sufficient for the solutions because the course staff aren't after the bonus point :)).