CS 161	Problem Set 8
Winter 2021	**Due:** Wed, Mar 10 at 11:59 pm PST

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.

What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises

We suggest you do these on your own. As with any homework problem, though, you may ask the course staff for help.

---

1. DP for recurrences

Consider the recurrence relation defined by

$$T(n) = 2T(n-1) + T(n-2) + 1,$$

with $T(0) = T(1) = 0$.

(a) **(4 pt.)** Write a **bottom-up** dynamic programming algorithm that computes $T(n)$. Your algorithm should run in time $O(n)$.

[**We are expecting:** Pseudocode. No explanation is required.]

(b) **(4 pt.)** Write a **top-down** dynamic programming algorithm that computes $T(n)$. Your algorithm should run in time $O(n)$.

[**We are expecting:** Pseudocode. No explanation is required.]

2. (10 pt.) Another greedy algorithm for bipartiteness?

Recall that in the BFS/DFS lecture we learned how to detect if a graph is bipartite[1] using BFS ($O(n + m)$ run time). Your friend comes up with the following greedy algorithm with the same running time:

> Pick an arbitrary vertex v and remove it from the graph. Recurse on the remaining vertices to obtain two independent sets. If possible, add v to one of the independent sets (can be either one); otherwise return that the graph is not bipartite. As a base case, an empty graph is partitioned into two empty sets, and is considered bipartite.

Determine whether or not your friend's algorithm works.

[**We are expecting:** If your friend's algorithm works, an informal proof of correctness. If it doesn't work, a counter example and a short explanation]

[1] Recall that a graph is *bipartite* if the vertices can be partitioned into two *independent sets*, i.e. there are no edges within each subset (but edges may cross from one set to the other).

Problems

You may talk with your fellow CS 161-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.

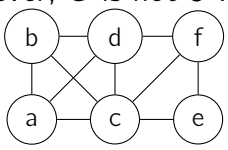
---

3. k**-well-connected graphs**

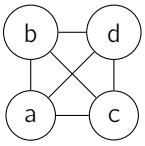
Let $G = (V, E)$ be an undirected, unweighted graph with n vertices and m edges. For a subset $S \subseteq V$, define the **subgraph induced by** S to be the graph $G' = (S, E')$, where $E' \subseteq E$, and an edge $\{u, v\} \in E$ is included in E' if and only if $u \in S$ and $v \in S$.

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.



$G = (V, E)$



$G' = (S, E')$, for $S = \{a, b, c, d\}$

(a) **(5 pt.)** Design a **greedy** algorithm to find a set $S \subseteq V$ of maximum size 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:

- `degree(v)`: return the degree of a vertex in time $O(1)$
- `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 $O(n^2)$.

[**We are expecting:** Pseudocode **AND** an English description of what your algorithm is doing.]

(b) **(5 pt.)** Prove by induction that your algorithm is correct.

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

4. Thanksgiving Turkeys

On Thanksgiving day, you arrive on an island with n turkeys. You've already had Thanksgiving dinner so you don't want to eat the turkeys (and maybe you prefer tofurkey anyway), 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) **(5 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.]

(b) **(5 pt.)** Prove by induction that your algorithm is correct.

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

5. [Optional] Continuous Knapsack

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) 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.

]

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

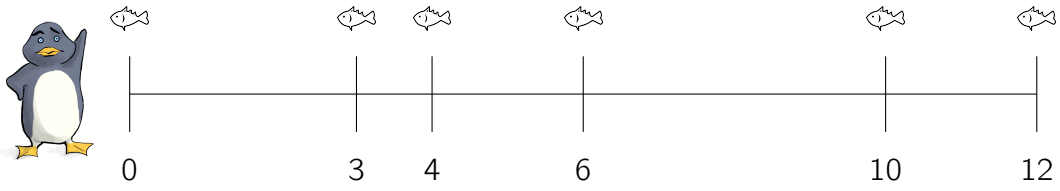
- **Inductive hypothesis:** After making the t'th greedy choice, there is an optimal solution that extends the solution that the algorithm has constructed so far.
- **Base case:** Any optimal solution extends the empty solution, so the inductive hypothesis holds for $t = 0$.
- **Inductive step:** *(you 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.]

6. [Optional] 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:



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) 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) 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.]