Math 590 Homework 7 Fall 2018

Due: **Thursday**, November 29

1. **Greedy Ferries?**[1]

   Ranopolis is a sprawling city divided in half by the great Muddy River. The city has a number of ferries that shuttle cars across every day. Each ferry had three lanes for cars, and these lanes are of varying lengths (even within one ferry, the three lanes may not necessarily have the same length). When cars line up single-file at a ferry crossing, their lengths are measured automatically. Each car is then directed into one of the three lanes of the ferry. The cars can be parked bumper-to-bumper, with no space between them. *The objective is to park as many cars as possible on a ferry without overfilling any of the three lanes.* Notice that the cars must be processed in the order in which they arrived at the ferry crossing point and we can't skip one car in order to pack a later car (that violates fairness). **All lane lengths and car lengths are positive integers.** Let n denote the number of cars and let ℓ_1, ℓ_2, and ℓ_3 denote the lengths of the three lanes.

   (a) [2pts] Prof. I Lai was hired by the Ranopolis Ferry Transit Authority to design an algorithm for assigning cars to the ferry lanes. He proposed the following greedy algorithm:

   'Park each car in the lane that has the most remaining space, and break any ties arbitrarily.'

   Show a small counterexample where this greedy algorithm does NOT pack the most cars. You will need to give a sequence of car lengths, the three lane lengths, and then show what Prof. I Lai's algorithm would do and show that there exists a better solution for that same input.

   ANSWER:
   Consider the case where we have all three lanes of length $\ell_1 = \ell_2 = \ell_3 = 4$, with the cars of length $\{2, 2, 2, 3\}$. Then according to this greedy algorithm, we place the first car in lane 1, the second car in lane 2, then the third car in lane 3. This means that there is no room left for the last car. However, if we had put two of the length 2 cars into the same lane, we could have fit the final car.

[1] Adapted from problem sets created by Harvey Mudd College CS Professor Ran Libeskind-Hadas.

(b) [2pts] It turns out that this problem can be efficiently solved using an algorithmic approach we have already discussed in class. Briefly discuss what approach you would use, and how it might be implemented efficiently (you do not need to fully describe the algorithm, but instead indicate how you could approach this problem).

ANSWER:

This problem could be solved with a use-it-or-lose-it approach and dynamic programming. We cannot skip any cars, which means that as we consider each car in order, we have to decide which lane to put that car in. So our three recursive cases would be to put the car into lane 1, lane 2, or lane 3; remove the car from the list, subtract its length from the given lane, and recurse. The base case would be that there are no more cars. We could turn this into a dynamic programming approach by noting that the list of cars and the length of the lanes decreases each iteration. We would have a 4D DP table (the length of each lane and which car we are considering). The recursive use-it or lose-it algorithm:

```
def ferry(L1, L2, L3, cars[1...n]):
    # Base Case: no cars left.
    if len(cars) == 0:
        return 0

    # Set the values for the three options to 0 and only update
    # if the first car fits in the respective lane.
    lane1 = 0, lane2 = 0, lane3 = 0

    # Try lane 1.
    if cars[1] < L1:
        lane1 = 1 + ferry(L1-cars[1], L2, L3, cars[2...n])

    # Try lane 2.
    if cars[1] < L2:
        lane2 = 1 + ferry(L1, L2-cars[1], L3, cars[2...n])

    # Try lane 3.
    if cars[1] < L3:
        lane3 = 1 + ferry(L1, L2, L3-cars[1], cars[2...n])

    # Return the best of the three.
    return max(lane1, lane2, lane3)
```

Note: this could actually be done with only a 3D table tracking the lengths of the three lanes. This is because a new car can only be considered if one of the three lanes has been decreased. This approach is more complicated and would require careful bookkeeping.

2. **Rural Cell Phone Towers**

The cell phone service provider Vorizan has decided to invest in cell towers in rural areas. Here, a rural area can be treated as a long straight road with houses along it. Vorizan's objective is to deploy the least number of cell towers along the road so that every house will be within 3000 or fewer meters of a cell tower.

Your task is to develop a greedy algorithm that takes as input an array of n points on a line, representing the locations of n houses. Each point is the distance in meters from the left starting point of the road (the origin) and those points are in sorted order (increasing distance from the starting point). The algorithm must return a sorted array of positions at which to place the cell towers. That array must use the least number of cell towers possible. (Aside: this is a 1D variation of the set cover problem.)

(a) [2pts] Describe the simplest greedy algorithm that you can for this problem.

ANSWER:

Place the first tower as far from the origin as possible while still covering the first house. Remove that house, and any covered house, from our list. Repeat.

More specifically, if the houses are at locations h_i, then:

- Place first tower at location $h_1 + 3000$.
- Remove any house such that $h_i < h_1 + 6000$.
- Repeat starting at the new 'first house'.

(b) [2pts] What is the worst-case asymptotic runtime of your algorithm? Give the best bound that you can and explain in a sentence or two.

ANSWER:

The worst case of our algorithm is $O(n)$ because we will have to look at each house only one time. When we place a tower, we scan through the list until we reach the next uncovered house, then place the new tower. This only requires scanning through the list once.

(c) [2pts] Prove the correctness of your algorithm.

ANSWER:

Assume we have an optimal solution that placed towers at locations σ_1, σ_2, etc, while our greedy algorithm placed towers at locations g_1, g_2, etc. If they are different solutions, then consider the first location where they differ, call it σ_i and g_i, and let's say that the house our greedy algorithm was covering in this iteration was h_j. We now make three observations:

- Since we were covering house h_j with our greedy algorithm, it must be that $g_i = h_j + 3000$ by the design of our algorithm.
- Note that since the two algorithms were the same until this choice, it must be that all previous houses were covered, i.e., all house $h_i < h_j$ have been covered.
- Our algorithm made the greedy choice, meaning that $g_i > \sigma_i$ because the optimal solution also had to cover house h_j, and could only do so with a tower at most 3000 m away.

Because $\sigma_i < g_i$, we know that there is no house further from the original covered by tower σ_i that was not also covered by g_i. And by the second observation, we know that there were no houses closer to the origin that were covered by σ_i that weren't already covered (since h_j was the first uncovered house). Therefore the optimal choice did not cover any more houses than our greedy choice. We could now use induction to prove that the greedy choice was an optimal at each iteration.

3. **Greedy Party Planning!**[1]

You've accepted a job as senior algorithm designer at the company Millisoft. One day, your boss Gill Bates comes to you with the following problem:

'I'm throwing a company party! As you know, Millisoft has a hierarchical structure. You can think of it as a tree. The president, that's me, is at the root of the tree. Below the root are supervisors, below them are managers, below them are team leaders, etc., etc., until you get down to the leaves - the summer interns. The tree is not necessarily binary: some non-leaf nodes may have one "child", others two, and others even more. To make the party fun, I thought it would be best that we don't invite an employee along with their immediate boss (their parent in the tree). So how can I choose which employees to invite to guarantee the largest possible party?'

In other words, your task is to take as input a tree representing the company hierarchy and compute the largest number of employees (nodes) that can be selected such that no two adjacent nodes (i.e., a node and its child) are chosen.

(a) [2pts] Describe a *greedy algorithm* for this problem (in either pseudo-code or clear English). A greedy algorithm, in this case, is one that visits vertices one-by-one (in some order of your choosing) and makes binding decisions on whether or not to invite that vertex before moving on to the next vertex.

ANSWER:

Search through the tree with DFS until we locate a vertex that has only leaf-children. Invite the leaves and remove their parent from the tree (without inviting). Repeat this process until every vertex has been invited or removed from the tree.

(b) [2pts] What is the worst-case asymptotic runtime of your algorithm?

ANSWER:

Recall that the runtime of DFS was $O(|V| + |E|)$. Since we are dealing with a tree, we know that it must be the case that $|E| = |V| - 1$. Therefore the runtime of each DFS is at worst $O(n)$ when there are n employees. Since in the worst case, we only remove 2 nodes from our tree each time, we could have to run DFS $n/2$ times. This gives an overall runtime of $O(n^2)$.

Note: we can actually do better than the algorithm described above. We can do this by noticing a very important fact about DFS: it will completely explore a subtree before considering any other subtrees. This means that we can modify DFS to allow for a single traversal of the tree that will find the optimal set of employees to invite.

The idea here is to allow DFS to revisit vertices, noticing that it will revisit a vertex one last time after that vertex's entire subtree has been explored. If we require that a vertex is invited or uninvited by the time DFS returns to it for the last time, then we will guarantee that all vertices are invited with a single pass of DFS.

While we explore the tree, we will number each vertex we visit, starting at 1 and incrementing by 1 for each **new** (unnumbered) vertex. This gives us an interesting property: when we revisit a vertex, if the difference between its assigned number and the current counter is equal to its number of children, then all of its children were leaves! We have discovered a way to check if a vertex is the parent of only leaf-children.

When this happens, we invite the children and uninvite the parent. Then we leave that subtree and never return. But, we technically have to remove the subtree from the overall tree, noting that now all of our numbering is out of order: we've incremented our counter for vertices that are no longer in the tree. So, when we remove the subtree, we decrease our counter back to the value it had when we first entered the subtree. This number is the number assigned to the parent we rejected (since it was the first vertex we looked at in the subtree)!

So, this algorithm is:

- Set `counter = 1`.
- Set the root's assigned number to equal the current counter.
- Explore the tree with DFS, revisiting vertices (i.e., push **all** neighbors into the stack regardless of whether they were already visited). However, we will require that the parent is the first neighbor pushed onto the stack (so we will visit all children before returning to the parent).
- Each time we visit a vertex that does not yet have an assigned number, we increment the counter and assign that number to the new vertex.
- If the vertex we visit has already been assigned a number, we check to see whether the difference between the current counter and the assigned number is equal to the number of children. If it is, we invite the children, reject the parent, remove them all from the tree, and decrease the counter down to the parent's assigned number.

Since we only use one pass of DFS, the runtime of this algorithm is $O(n)$.

(c) [2pts] Prove the correctness of your algorithm using strong mathematical induction.

ANSWER:

We will use induction on the number of remaining vertices in the tree.

Base Case: $n = 1$

In this case, the optimal solution is to invite the single employee, which our greedy algorithms does because it was a root-leaf.

Induction Hypothesis:

For all tress with k or fewer nodes in them, our greedy algorithm provides the optimal solution.

Inductive Step:

Consider a tree with $k + 1$ nodes in it. Our greedy algorithm finds a node with only leaf-children, invites its children and does not invite it.

Now, this must decrease the tree by at least 2 nodes (if there was only a single leaf-child, it is invited and its parent is removed). Since our greedy algorithm only invited the leaves and never the parent, we are not restricting the options made on the rest of the nodes. So by the inductive hypothesis, our algorithm will make an optimal selection for the rest of the tree (since it will have size at most $k - 1$).

So, we just need to show that our single choice was still optimal. Note: at this single step, we only ever choose to uninvite a single parent node. But, we have to invite at least one leaf-child (and more if there were multiple leaf-children). So our choice is never worse than the alternative of inviting the parent and uninviting the leaf-children.

4. **Two Stacks can make a Queue**

A well implemented stack has two operations that it can support in $O(1)$ time: `Push` and `Pop`. It turns out that it is possible to implement an efficient queue using only two stacks. A queue has two supported operations:

- `Enqueue`: where a value is pushed into the rear of the queue.
- `Dequeue`: where a value is popped from the front of the queue.

(a) [2pts] Describe how a queue can be implemented using two stacks. (Hint: a more standard queue implementation typically tracks **two** values: front and rear. You have **two** stacks to use.)

ANSWER:
First, let's name our two stacks. We will have a Rear stack and a Front stack.

When we have an Enqueue operation, we will Push the value onto the Rear stack. Now, clearly that means that the stack will store the values with the first-in element at the bottom of the Rear stack.

So, when we have a Dequeue operation, we will need to access the element from the bottom of the Rear stack. We will Pop each element off the Rear stack and Push them one-by-one onto the Front stack. This reverses their order, meaning that the top of the Front stack is the first-in element of the queue. We can now Pop from the Front stack for each Dequeue operation to access the next most recently Enqueued element. Whenever the Front stack is empty, we will refill it by popping the elements from the Rear stack and Pushing them onto the Front stack.

(b) [2pts] Using the accounting method from class, prove that your stack-based queue has the property that any sequence of n operations (selected from `Enqueue` and `Dequeue`) takes a total of $O(n)$ time resulting in amortized $O(1)$ time for each of these operations!

We first note that at most, each element will get pushed onto the Rear stack once when it is Enqueued, popped from the Rear stack and pushed onto the Front stack once when the Front stack is empty (which must happen before this element can be Dequeued), and popped from the Front stack once when Dequeued. So, assign each element 4 rubles. Spend one ruble for the Push onto the Rear stack, two rubles for the Pop from the Rear stack and the Push onto the Front stack, then the final one ruble for the Pop from the Front stack. This covers all the possible operations, and so a series of n operations will cost at most $4n$ rubles. So we have $O(1)$ amortized time.