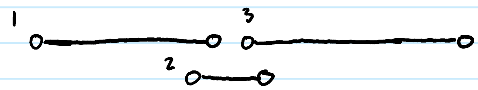
(1) **(Counterexamples to greedy procedures)** (30 points) Prove that the following greedy procedures for the Activity Selection Problem are *not* correct. Each procedure considers the activities in a particular order, and selects an activity if it is compatible with those already chosen.

(a) The activities are considered in order of increasing *duration*.
(b) The activities are considered in order of increasing *start-time*.
(c) The activities are considered in order of increasing *number of overlaps* with the remaining compatible activities. (This is a dynamically-determined order.)

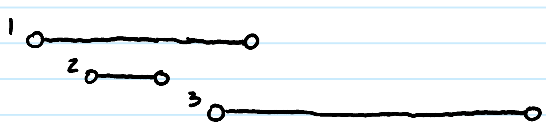
(Note: To prove an optimization procedure is not correct, it suffices to demonstrate a *counterexample*: namely, an instance of the problem that has a feasible solution that is better than the one the procedure outputs.)

a) Lets say we have a list of activities as such:



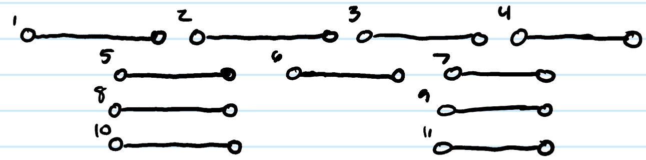
where the number is simply the number of the activity and the first point of each line is its start time and the last point its end time. If we sort by duration we get a list of [2, 1, 3]. Our greedy algorithm will then pick the first in the list (activity 2) and add it to S then continue down the list until it found the next compatible activity with the set of activities S. No other activities are compatible with activity 2 so sorting by duration will only choose activity 2 to add to its selection. In sorting by end time we get an optimal solution of a set with activities 1 and 3 which is clearly more optimal in cardinality than the solution obtained by sorting by duration. Therefore sorting by increasing duration does not give an optimal solution to a greedy algorithm.

b) With a list of activities like this:



Sorting by start time would give you [1, 2, 3] and a greedy algorithm will add the first activity in the list to the set of selected activities (S). So activity 1 will be added to S. Then, we continue down the list until we find the next activity that is compatible with S. In this case there are no other activities compatible with activity 1 so S will only contain activity 1. If we sort by end time we get a list of [2, 1, 3], greedily select 2 as our first activity, skip 1 because its not compatible, then select 3. Sorting by end times S = {2, 3} while sorting by start time S' = {1}. S is clearly a more optimal solution than S' so sorting be increasing start time using a greedy algorithm does not give an optimal solution.

c) With a list of activities like this:



By choosing the activity with the least number of overlaps first we would choose activity 6 since it only overlaps 2 other activities. Then, the only other compatible activities will be 1, 4, 5, 7, 8, 9, 10, 11. From here we would only be able to add two more activities, one from [1, 5, 8, 10] and one from [4, 7, 9, 11] so the cardinality of this solution will be 3. However, sorting using the end times will give a cardinality of 4 with a solution of [1, 2, 3, 4]. Therefore, sorting by increasing number of overlaps and using a greedy solution does not give us an optimal solution.

(2) **(Trip refueling)** (30 points) Suppose you want to travel from city A to city B by car, following a fixed route. Your car can travel m miles on a full tank of gas, and you have a map of the n gas stations on the route between A and B that gives the number of miles between each station.

Design a greedy algorithm to find a way to get from A to B without running out of gas that minimizes the total number of refueling stops, in $O(n)$ time. Prove that your algorithm finds an optimal sequence of stops.

First you will set your current position to A. Next find the furthest gas station that is within m miles from your current position. Stop at that gas station, fill up your tank and set your current position to that gas station. Continue to find the next furthest gas station from your current position within m miles until B is within m miles of the gas station you are currently at.

```
function  find Route (A, B, n, m) begin
    \\ assume n is in order of increasing distance from A
    cur_pos = A
    while (cur_pos != B) begin
        if  B isn't within m miles of cur_pos do
            find closest gas station  \\ add distances between closest gas stations
                                         until you go over m
            add that gas station to output
            fill up gas
            set cur_pos to current gas station
        else do
            cur_pos = B
    end
    return output
end
```

Correctness

Lemma – Suppose a partial solution S is contained in an optimal solution. Let S' be the augmentation of S found by the greedy algorithm. Then, S' is also contained in an optimal solution.

Proof of lemma

Let S^* be the optimal solution that contains partial solution S

– S is a prefix of S^*

– Let b be the next gas station selected that is between our current position and B and within m miles of us.

– Let a be the next gas station the greedy algorithm adds

If $a=b$, S^* also contains the augmentation S' then the lemma holds because our augmentation is the same as our partial solution.

If $a \neq b$ then b is further (but still within m miles) then a. Then, S^* doesn't contain S', then modify S^* to get another complete solution $\tilde{S}$ that does contain S'

Then, if $\tilde{S}$ contains S', $\tilde{S}$ contains b and S' is a partial solution and $\tilde{S}$ is an optimal solution because it will always contain the furthest gas station within m miles.

This proves the lemma.

Theorem – The greedy algorithm for finding the number of stops finds an optimal solution

Proof -

If there are no stops and $|A-B| < m$ the solution is {}
so before selecting any stops our partial solution is {} which is
a subset of the optimal solution.

- Once the greedy algorithm selects the first stop, by the lemma,
we know that stop will be contained within the optimal solution.
By induction we can show any subsequent gas station we stop at will
also be contained in the optimal solution by the lemma.
Once we reach B all stops in our partial solution will also be
in the optimal solution so our partial solution is not properly
contained by the optimal solution so our partial solution is the
optimal solution.

Time complexity

```
function findRoute (A, B, n, m) begin
    \\ assume n is in order of increasing distance from A
    cur_pos = A
    while (cur_pos != B) begin
        if B isn't within m miles of cur_pos do
            find closest gas station  \\ add distances between closest gas stations
                                         until you go over m
            add that gas station to output   }
            fill up gas                      } Θ(1)
            set cur_pos to current gas station }
        else do
            cur_pos = B   ~ Θ(1)
    end
    return output
end
```

Iterating over each gas station once so we will eventually inspect aer element of n → Θ(n)

$$T(n) = \Theta(n)$$

(3) **(Minimizing average completion-time)** (40 points) Suppose you are given a collection of n tasks that need to be scheduled. With each task, you are given its duration. Specifically, task i takes t_i units of time to execute, and can be started at any time. At any moment, only one task can be scheduled.

The problem is to determine how to schedule the tasks so as to minimize their *average completion-time*. More precisely, if c_i is the time at which task i completes in a particular schedule, the average completion-time for the schedule is $\frac{1}{n}\sum_{1\leq i\leq n} c_i$.

(a) (40 points) Design a *greedy* algorithm that, given the task durations $t_1, t_2, \ldots, t_n$, finds a schedule that minimizes the average completion-time. You may assume that once a task is started, it is run to completion. Your algorithm should take $O(n \log n)$ time.

Analyze the running time of your algorithm, and prove that it is correct using a lemma of the required form.

In order to get the minimal average completion-time we want to sort the collection of tasks by the time it takes to execute that task (t_i). From there we simply select the task with the lowest execution time, then the next lowest, then the next, and so on until we've selected all the tasks.

```
function minimizeAverage(tasks, duration) begin
        sort tasks by execution time
        return tasks
end
```

Time complexity - the only part to consider for the time complexity is sorting the tasks by execution time. We know sorting can be done in nlogn time so

$$T(n) = \Theta(n \log n)$$

Correctness

Lemma - Suppose a partial solution S is contained in an optimal solution. Let S' be the augmentation of S found by the greedy algorithm. Then, S' is also contained in an optimal solution.

Let S^* be the optimal solution that contains the partial solution S where S is a prefix to S^*

Let b be the next task with the lowest execution time to be selected

- We know we have to sort by lowest execution time because the next task will then have a total completion time that includes all the completion times of the tasks before it so all task completed after the first will utilize the first task's execution time.

ex. tasks = [1, 2, 3]
duration = [5, 10, 1]

in order 1, 2, 3 completion time $= \frac{1}{3}(5+15+16) = \frac{36}{3}$

in order 1, 3, 2 " " $= \frac{1}{3}(5+6+16) = \frac{27}{3}$

in order 2, 1, 3 " " $= \frac{1}{3}(10+15+16) = \frac{41}{3}$

in order 2, 3, 1 " " $= 1\ (10+11+16) = 37$

in order 2, 1, 3 " " $= \frac{1}{3}(10+15+16) = \frac{41}{3}$

In order 2, 3, 1 " " $= \frac{1}{3}(10+11+16) = \frac{37}{3}$

in order 3, 1, 2 " " $= \frac{1}{3}(1+6+16) = \frac{23}{3}$ ✓

in order 3, 2, 1 " " $= \frac{1}{3}(1+11+16) = \frac{28}{3}$

the order of 3, 1, 2 for tasks and 1, 5, 10 for duration gave the smallest average.

In this case, since the duration of 1 is selected first all following completion times also contain this execution time

$$\frac{1}{3}((\underline{1})+(\underline{1}+5)+(\underline{1}+5+10))$$

all completion times after the second contain the second lowest execution time.

In order to get the lowest completion time we sort in increasing execution time.

- Let a be the next task picked by the greedy algorithm

- If $a = b$, then S^* also contains the augmentation S' then the lemma holds because the augmentation is the same as our partial solution

- If $a \neq b$, then b has a lower execution time so S^* doesn't contain S'. Then, we modify S^* to get another complete solution $\tilde{S}$ that contains S'

  If $\tilde{S}$ contains S' then it also contains b and will in turn be an optimal solution because it always contains be which we said would give us an optimal solution.

  This proves the lemma.

Theorem - the greedy algorithm for minimizing the average completion-time finds an optimal solution

Before we make a choice for the first task, the set of chosen tasks is empty → {} is a subset of the optimal solution.
Once the greedy algorithm selects the first task, by the lemma we know that task will minimize the average completion time.
Inductively we can see that for every task selected beyond the first and up to and including the last will be part of the optimal solution by the lemma.

(b) **(bonus)** (20 points) Suppose with each task we also have a *release time* r_i, and that a task may not be started before its release time. Furthermore, tasks may be *preempted*, in that a scheduled task can be interrupted and later resumed, and this can happen repeatedly.

Design an algorithm that finds a schedule that minimizes the average completion-time in this new situation. Analyze its running time and prove that it is correct.

For this part we would sort the tasks in the same order (in increasing execution time) except when we come across a task for processing and it has a later release time we skip over it, but remember the release time for any tasks we skip over. Once we reach the release time of a skipped over task we preempt the current task and begin working on the skipped over task with a lower execution time.

Time analysis - Since we have to continuously be checking if any tasks with later release times are ready for processing we may end up going over every task again to see which ones are ready. So, in addition to going over ever task to run it, all tasks may have a release time so we will go over them again which will make the runtime $\Theta(n^2)$