Assignment 35

Activity Selection

```
select_activities(activities):
  selected_activities = []
  sort(activities, key=activity.finish_time, order=ascending)
  booked_time_range = [0, 0)
  for activity in activities:
    activity_interval = [activity.start_time, activity.finish_time)
    if no_overlap(activity_interval, booked_time_range):
      selected_activities.add(activity)
      booked_time_range = union(booked_time_range, activity_interval)
```

1. In class, we saw an algorithm (copied above) which sorts activities by their finishing time in ascending order and always makes the greedy choice by selecting the activity with the earliest ending time that does not conflict with any previously selected activities. Below, two different greedy strategies are outlined. For each one, state whether or not the strategy always produces a maximal set of non-conflicting activities, and justify your answer.
    2. Sort activities by start times in descending order and continuously make the greedy choice by selecting the activity with the latest start time that does not conflict with any previously selected activities.
       A: this algorithm is equivalent to the proven-optimal algorithm above, just "backwards": it operates last-to-first rather than the first-to-last pattern of the original algorithm.
    3. Sort activities by runtime in ascending order and continuously make the greedy choice by selecting the activity with the shortest runtime that does not conflict with any previously selected activities.
       A: This may be proven invalid by a counterexample. Say that on the interval from 10am to 12pm, there are activities 10am-11am, 11am-12pm, and 10:59am-11:01am. This algorithm would prioritize the short, conflicting event in the middle of the interval and choose a solution with one event in the period 10am to 12pm, rather than the optimal two events.
2. The algorithm above keeps track of the union of intervals of all of the selected activities. However, keeping track of unions of intervals is actually keeping track of more data than necessary, and is not necessarily a trivial operation. Additionally, checking if a single interval overlaps with a union of intervals may also not be trivial. Instead of a union of intervals, what much simpler piece of data could be kept track of, and instead of seeing if one interval overlaps with a union of intervals, what could be checked?

A: only the last chosen event needs to be kept at any given time. since the events are sorted already, the next event will not overlap if it doesn't overlap with the current latest-running event, a.k.a the last one picked

3. Recall the definition of a partition P of a set S: P consists of subsets of S such that no subset is empty, they are all pairwise disjoint, and their union is equal to S. The cardinality of the partition is the number of subsets it contains. For example, $\{\{1,2\},\{3\}\}$ is a partition of the set $\{1,2,3\}$ with cardinality 2.

   Imagine that if instead of all sharing one resource, activities with overlapping times could simultaneously occur by using different resources (such as different rooms), and one wants to schedule all of the activities while using as few resources as possible. Devise an algorithm which partitions the activities such that each subset of the partition has no overlapping activities and that the cardinality of the partition produced is minimal. State why your algorithm produces an optimal solution.

   Your algorithm should always be working towards an optimal partition, not haphazardly producing potentially viable partitions and testing them (i.e. do not brute-force all possible partitions). **Hint:** What is the smallest possible size of the partition, and does your algorithm produce it?

   A: my algorithm would consist of repeated passes of the original greedy algorithm for events. the algorithm would start by finding the maximal solution of events, then it would remove those events and put them into partition #1. the maximal solution for the remainder of events would then be found and put into partition #2, and the process would continue until no events are left.

```
greedy_partition(activities, partitions):
    while(activities.length > 0):
        maximal = select_activities(activities)
        partitions.addNew(maximal)
        activities = activities - maximal
     return partitions
```

For extra credit, program and test your algorithm.

A: See Attached "greedy-partitioning.rkt" (requires expect/rackunit)