HW 9

- #3 – don't mix up the two versions of the problem
  - the original problem is to find an ordering of the print jobs so that no job misses its deadline, if possible
  - an optimization version is to find an ordering of the print jobs which minimizes the amount by which the most late job misses its deadline
  - the hint says –
    - develop a greedy algorithm for the optimization version
    - then explain how you can use a schedule with min max lateness to get a schedule where no job misses its deadline or determine that isn't possible
  - tackling the feasible schedule version directly requires showing that if the algorithm picks a job which misses its deadline, there is no other ordering that will work
    - this can be tricky because there may be another ordering which gets that job done on time but some other job misses...

HW 9

- give the steps of the greedy process from class, not just a statement of the algorithm or a narrative account of your reasoning process
  - establish the problem
    - specifications
    - examples
  - identify avenues of attack
    - targets
    - paradigms and patterns
    - greedy choices and counterexamples
  - define the algorithm
    - main steps
    - exit condition
    - setup
    - wrapup
    - special cases
    - algorithm
  - show termination and correctness
    - termination: measure of progress, making progress, the end is reached
    - correctness: loop invariant, establish the loop invariant, maintain the loop invariant, final answer
  - determine efficiency
    - implementation
    - time and space
    - room for improvement
- for process issues that occurred in more than one of the problems, the issue was generally only pointed out once
  - the same comments were not necessarily repeated for all of #1-3

HW 9

- patterns and main steps should make it clear what you are iterating through and what the choice is

```
for each thing
  make a choice about it
```

  - patterns identify what the choice is about
  - main steps address specifically how the choice is made (the greedy choice)

HW 9

task	iterative pattern	choice
subset	process input	include element in the solution or not
	produce output	which element to include in the solution next
ordering	process input	where to add the element into the solution-so-far
	produce output	which element to append to the solution-so-far
labeling	process input	the label to give the element

- typically only consider a process input pattern for labeling tasks
  - for each input item, determine the label
- produce output would be to produce the next (element,label) pair – the output is the labeling, which is a collection of (element,label) pairs
  - ...which could be generated by going through each element and deciding on its label – but that's just process input
  - ...or by going through each label and determining which element(s) have that label – but that leads to nested loops when more than one element can have the same label
    - nested loops are more complex to argue termination and correctness for – need address for each loop

HW 9

Greedy choices –

- aim for a simple algorithm – does how the choice is made even matter?
  - e.g. the captioning problem has two sets of choices – what program to caption next and which employee to assign to that program
  - prove that the choice matters with a counterexample – is there a wrong choice to pick?
  - if you aren't using some aspect of your algorithm's choice in the correctness proof (establishing and maintaining the invariant), either your proof is incorrect or the choice doesn't matter
- if several alternatives could satisfy the greedy choice, make sure that picking any of them is OK
  - if there's a wrong choice, it must be excluded

HW 9

Greedy choices –

- for identifying avenues of attack, identify what all you have to base the greedy choice on
  - don't just jump immediately to what you think is the right choice – you might be wrong...
  - focusing prematurely on one choice often means you don't try very hard (or at all) to find counterexamples – and it's hard to prove correct what isn't

HW 9

For all algorithms (not just greedy ones), describe the algorithm with an appropriate level of detail.

- too much detail is worse than not helpful – it obscures understanding

- consider the purpose of the description
  - to address correctness, you only need to know the result of an action, not how it is carried out
    - e.g. for each person in alphabetical order …
    - e.g. 2-coloring of a graph whose vertices represent regions and edges connect adjacent region
    - e.g. assign a currently-available employee
  - to convey the algorithm to another person, you only need to give more specific steps if they don't know how to do something
    - well-known problems – your audience would generally know an algorithm (or how to locate an algorithm) e.g. traverse a graph, shortest path, MST
    - obvious brute force

HW 9

For all algorithms (not just greedy ones), describe the algorithm with an appropriate level of detail.

- too much detail is worse than not helpful – it obscures understanding

- consider the purpose of the description
  - to assess running time, implementation details are needed – but only when there is a choice or the specifics aren't well known
    - for data structures, reference high-level ADTs where possible
      - e.g. priority queue instead of heap

HW 9

- special cases
  - for greedy algorithms, be alert to any situations where the greedy choice is not a unique alternative
    - e.g. duplicate values could lead to a tie in the greedy choice
    - if there's a wrong choice about how to resolve a tie, the greedy choice needs to be revised to ensure no ties
  - avoid overly special cases
    - e.g. all *n* programs have identical start/end times vs two programs have same start or same end – every element is the same is a very specific version of duplicate elements exist

HW 9

Loop invariants for greedy algorithms –

- the invariant needs to address both legality and optimality
  - legality is what makes a legal solution e.g. no employee is assigned to caption two shows at the same time
  - addressing optimality is what allows you to conclude that the solution produced is the best such legal solution
- for optimality, use a staying ahead argument – the algorithm's solution so far is at least as good as a comparable portion of an optional solution
  - an outright claim that the partial solution so far is the best can be too much to prove with too few concrete things to reason about
    - can lead to *the invariant holds because the algorithm made the right choice* as "justification" – but that the algorithm made the right choice is what we are trying to show
  - "comparable portion" requires an apples-to-apples comparison
    - e.g. for subset tasks, compare the first *k* items the algorithm picks to the first *k* items in the optimal solution – when considered in the same order as the algorithm's picks

HW 9

Loop invariants for greedy algorithms –

- the staying ahead quantity can't be the optimization quantity if the optimization quantity is the number of loop iterations
  - e.g. Boston to Seattle, repeatedly pick the next gas station to stop at
    - one stop is picked with each iteration, so minimizing the number of stops means minimizing the number of loop iterations
    - the algorithm's partial solution is the stops after *k* iterations

HW 9

Maintaining the invariant – assuming that the invariant is true after *k* iterations, show that it is still true after *k*+1 iterations.

Critical key point –

- we want to show that what our algorithm does in this iteration means that if the invariant was true at the beginning of this iteration, it is still true afterwards
- but – the argument is *not:* the algorithm made the right choice, therefore the invariant holds
  - that the algorithm's choice is the right one is what we are trying to show, not something we can assume
- the argument *is:* the algorithm's choice could not have resulted in breaking the invariant, therefore the invariant holds
  - ...and the algorithm's choice is the right one because those choices resulted in the invariant still being true after the last iteration and the invariant being true for a complete solution means we have a correct solution

HW 9

Maintaining the invariant –

- use proof by contradiction to show that the algorithm's choice could not have resulted in breaking the invariant
  - assume that the algorithm's choice broke the invariant
  - think about what that means and what you can deduce about the situation
  - find a contradiction
  - ...and thus the algorithm's choice can't have broken the invariant
- compare only the algorithm's partial solution and a comparable part of an optimal solution, not process
  - *the optimal solution must select at least one program per iteration* – we have no idea how the optimal solution was generated (iterative algorithm, recursive algorithm, an oracle, ...)

HW 9

Maintaining the invariant –

- be careful of stealth the-algorithm-made-the-right-choice arguments
  - e.g. the algorithm only assigned a new employee to caption a program if there weren't any available, therefore the minimum number of employees have been used
    - but this conclusion requires establishing that there aren't any other ways those same employees could have been assigned to the programs that would have allowed an existing employee to caption the current program
  -

HW 9

- the "final answer" step of showing correctness bridges the gap between the loop invariant and a correct solution
  - if the loop invariant is *the algorithm's partial solution is at least as good as the comparable part of the optimal solution in terms of the quantity being optimized*, then the step is direct –
    - the exit condition says that the partial solution is complete
    - combining the exit condition and the loop invariant yields: the algorithm's complete solution is at least as good as the complete optimal solution in terms of the quantity being optimized
    - and since it is impossible for the algorithm to have a better solution than the optimal, its solution must be optimal

HW 9

- the "final answer" step of showing correctness bridges the gap between the loop invariant and a correct solution
  - if the optimization goal is the number of iterations, we need to show that the algorithm had the right number of iterations –
    - there are three possible outcomes
      - the algorithm had more iterations / produced a bigger solution than the optimal ($|A| > |O|$)
      - the algorithm had fewer iterations / produced a smaller solution than the optimal ($|A| < |O|$)
      - the algorithm had the same number of iterations / produced the same size solution as the optimal ($|A| = |O|$)
    - show that $|A| = |O|$ (that the algorithm's solution is optimal) by showing that the other cases are impossible
      - one is impossible because the algorithm can't do better than the optimal (nothing can)
      - for the other –
        - combining the exit condition and the loop invariant to get: *the algorithm's complete solution is at least as good as the complete optimal solution in terms of some quantity*
        - explain why that statement means the algorithm couldn't have resulted in a shorter / longer solution than the optimal