CS1570 Style Guidelines

Fall 2017

In addition to being graded on correctness of your solutions, each problem you submit (with the exception of programming assignments) will also receive a style score, which is a multiplier between 1.7 and 2.1. Your final grade for a problem can be calculated by points_earned * style_score.

The style score is meant to reflect how effectively you communicated your solution. This includes the clarity of your language, the directness of your approach, and approachablity of your notation. For tips on clear proof writing, please refer to the email which Professor Valiant sent out with the subject line **"[CSCI-1570-S01] communicating effectively and stylishly"**. We have included a copy of our rubric below, as well as an example of each score.

Scores

	What it Means:	Reasons
1.7	This response is below the standards of this course. Grader cannot follow the structure of the submitted proof or algorithm.	Only pseudocode given, response contained no explanations, or explanations in incomplete or unconnected sentences, writing was illegible.
1.8	The grader had to put in substantial effort to follow the structure of a submitted proof or algorithm.	Confusing or unexplained notation, excessive wordiness, missing definitions, "back solving", or other unfit proof structure that results in added complexity
1.9	Grader was able to follow; some common or small issues that did not detract from the overall clarity of the proof.	Complex notation, wordiness, or unneeded proof structure; minor steps are introduced without motivation
2.0	The grader was able to follow all steps clearly and with little effort.	Clean and common notation, with clear direction throughout the proof.
2.1	Rare grade - "better than we thought possible." The grader is amazed by the novelty and clarity of the solution	Exceptionally well worded and thought out.

1.7

Example:

```
foo(n):
    if(!n) then stop
    Else n -> n_1 and n_2
          bar(n_1, n_2) after
            n_1 <- foo(n_1), n_2 <- foo(n_2)

bar(x, y):
   if x(0) < y(0): add x(0) else add y(0)
```

```
remove x(0) or y(0)
add bar(x, y)
```

1.8

Example:

Our algorithm for *mergesort* shall be as follows: begin by splitting L into two pieces, and then call *mergesort* on both L_1 and L_2. After this, call the helper function *merge* on L_1 and L_2. This will result in the array L being sorted. Our helper function *merge* will take in two arrays, and iterate through both of them at the same time, constantly selecting the lesser of the two items at $head(L_1)$ and $head(L_2)$ and then appending to M. If either *merge* or *mergesort* get as input an array of a single element, it will immediately return that array.

1.9

Example:

We construct an algorithm, `mergesort`, which will take as input a list of numbers, L, which can either be in the Reals or Integers (or anything that can be ordered) and output a list M, which contains the exact same elements of L except now sorted in increasing order.

We begin by splitting L into two subarrays of equal length, L_1 and L_2. If there are an odd number of elements in L, then we can arbitrarily choose to put the extra element in either L_1 or L_2, since a single element will not effect the runtime of the algorithm. We will first sort the subarrays L_1 and L_2, and then recombine the sorted results using the `merge` algorithm, which we will also later define, to recreate a sorted L and thus get a sorted version.

In order to sort L_1 and L_2, we recursively call `mergesort`(L_1) and `mergesort`(L_2), since `mergesort` will correctly sort any list, and save the results in M_1 and M_2. Since M_1 and M_2 are both sorted by `mergesort`, we can combine the two lists in sorted order by simply appending the smallest number at the head of either list to M, and then removing that element from its list, and repeating.

2.0

Example:

We construct an algorithm, `mergesort`, which will take as input a list of numbers, L, and output a list M, which contains the elements of L in increasing order.

If L contains a single element, than that list is sorted, so we just terminate and return L immediately.

We begin by splitting L into two subarrays of equal length, L_1 and L_2. We will first sort the subarrays L_1 and L_2, and then recombine the sorted results to create a sorted L.

In order to sort L_1 and L_2, we recursively call `mergesort`(L_1) and `mergesort`(L_2), and save the results in M_1 and M_2. Since M_1 and M_2 are both sorted, we can combine the two lists in sorted order by simply appending the smallest number at the head of either list to M, and then removing that element from its list, and repeating until all items have been added to M. At any point in this procedure, the smallest number in M_1 and in M_2 will be at the front of the list, which allows us to add the smallest number which is not already in M to that list.