University of Virginia Computer Science
CS216: Program and Data Representation, Spring 2006 01 March 2006

Exam 1 Comments

Order Notation

For each of the questions below, fill in the missing symbol with one of these choices:

- $=$ — the sets are equal
- $\subset$ — the left set is a strict subset (cannot be equal) of the right set
- $\supset$ — the left set is a strict superset (cannot be equal) of the right set
- $\subseteq$ — the left set is a subset (can be equal) of the right set
- $\supseteq$ — the left set is a superset (can be equal) of the right set
- $\neq$ — the sets are not equal, but there is no subset or superset relationship

You should select the strongest possible choice (for example, if two sets are equal and you select subset, that is a correct statement, but not worth full credit).

For each answer, provide a short justification of your answer. Your justification should follow from the definitions of the order notations.

1. (Average 4.49 out of 5 points) $O(n) \neq \Theta(n^2)$

The sets are unequal (in fact they are disjoint, and have no members in common). The set $\Theta(n^2)$ is all functions that grow as fast as n^2; the set $O(n)$ is the set of all functions that grow no faster than n. Since n^2 grows faster than n, there is no overlap between the sets.

2. (4.11 / 5) $\Theta(n) \subset O(2n)$

First, note that the factor has no effect on the growth rate, so $O(2n)$ is equivalent to $O(n)$. The set $\Theta(n)$ is the set of all functions that grow as fast as n; this is a proper subset of $O(n)$ which is the set of all functions that grow no faster than n. There are functions in $O(n)$ that are not in $\Theta(n)$ such as constant functions or $\log n$.

3. (4.14 / 5) $\varnothing = O(n!) \cap \Omega(n^n)$

The intersection of $O(n!)$ and $\Omega(n^n)$ is empty, since n^n grows faster than $n!$ (see Problem Set 1).

4. (4.82 / 5) $O(1) \supset \Theta(1)$

This one is tricky, and we accepted $=$, $\supseteq$ or $\supset$ as a full credit answer (with a good explanation). Certainly we known every element of $\Theta(1)$ is also in $O(1)$ from the definitions of Θ and O. The tougher question is figuring out if there is any function in $O(1)$ that is not in $\Theta(1)$. From the informal definitions, this would mean is there any function that produces a positive value as output that grows slower than a constant. Intuitively, it would seem that functions like $1/n$ have this property — as n increases, the value of the function decreases asymptotically. To answer it more convincingly, we need to consider the definitions of O and Θ.

If f is in $O(1)$, then we know there are constants $c > 0$ and $n_0 \geq 0$ such that $f(n) < c$ for all $n \geq n_0$. (Since $g(n) = 1$, the $g(n)$ term goes away.) For $O(1)$ to be equal to $\Theta(1)$, we would need to show that this also implies f is in $\Omega(1)$. This requires that we can find constants $c > 0$ and $n_0 \geq 0$ such that $f(n) > c$ for all $n \geq n_0$. That is, f is in $\Omega(1)$ if and only if 1 is in $O(f)$. Consider $f(n) =$

$1/n$. We cannot have 1 is in $O(1/n)$ since for some value of n we know eventually $1 > c * 1 / n$ for some value of c (we can choose $n > c$ for any choice of c). Hence, we know $O(1)$ is a strict superset of $\Theta(1)$.

Lists

5. (6.98 / 10) Complete the definition of the `ListNode` method `reverse`, that is called by the `LinkedList` method `reverse` to produce a reversed self list as its output (the same elements as in self, but in reverse order). For example,

```
l = LinkedList.LinkedList ().append(1).append(2).append(3)
r = l.reverse ()
```

should make `r` the list `[3, 2, 1]` and leave `l` as the list `[1, 2, 3]`.

The rest of the code is taken from the `LinkedList.py` implementation of an immutable list datatype from Problem Set 2.

The easiest way to think of reverse is simply adding the first element to the end of the list resulting from reversing the rest of the elements:

```
def reverse (self):
    head = ListNode (self.__info)
    if self.__next == None:
        return head
    else:
        return self.__next.reverse().append (self.__info)
```

There are more complex iterative ways (that are more efficient) to do this by switching the `__next` pointers down the list, but they are much tougher than the simple recursive solution.

6. (8.46 / 10) What is the asymptotic running time of your `reverse` implementation? Explain your answer convincingly, and be sure to define any variables you use and state any assumptions you make clearly.

Our reverse implementation has running time in $\Theta(n^2)$ where n is the number of elements in the input list. (Note, it is possible to implement reverse with running time in $\Theta(n)$ by adjusting the `__next` pointers directly.)

There will be n calls to `reverse`, once for each element in the list (actually, there are n - 1 because the base case stops the recursion for the list of length 1). But, each call to reverse involves a call to our `append` method. The provided `append` implementation has running time in $\Theta(m)$ where m is the size of the input to `append` (see Problem Set 2). The average length of the input list to append is $n / 2$, so the average running time of the append call is in $\Theta(n / 2) = \Theta(n)$.

We are making n calls to reverse, each of which involves running time $\Theta(n)$ from the call to append, so the total running time will be in $\Theta(n^2)$.

Matching

7. (9.2 / 10) Prove the greedy partnering algorithm shown is not optimal by showing an input for which it would not produce the correct result.

We just need to find an input list where the best match for the first student does not produce the optimal global match. Here's a simple example:

- students = { "Alice", "Bob", "Colleen" }

- The goodness scores of (Alice, Bob) = 10, (Alice, Colleen) = 20, and (Bob, Colleen) = 110. (Bob and Colleen are in the same section and different years; Alice and Bob are in different years and different sections; Alice and Colleen are in different majors, the same year, and different sections.)
- The greedy algorithm will first find the best partner for Alice, which is Colleen. Then, there is no partner left fo Bob, so the total goodness score is 20. If we matched Bob with Colleen, and left Alice unpaired, the goodness score is 110.

8. (4.3 / 5) What is the asymptotic running time of `assignPartners`? Be sure to define any variables you use in your answer and state your assumptions about Python operations clearly.

$\Theta(n^2)$ where n is the number of students. We have two nested for loops, each of which iterates through the students. So, there are $\Theta(n^2)$ evaluations of the inner for loop (there are not exactly n^2 iterations, since we skip the inner loop if the student already has a partner; in the worst case, however, no students will be assigned partners if the best match for each student is with None).

The body of the inner for loop evaluates `goodnessScore` twice. This involves indexing into `records` to find each students record, and doing a dictionary lookup on the field we are comparing, as well as the `find` call in the notpartners list. We need to assume all these operations have running times in $O(1)$ for the overall running time to be in $\Theta(n^2)$. The one we are most concerned about is `find`, which searches the notpartners string for each student to see if it contains the other student. This is likely to have running time in $O(s)$ where s is the length of the notpartners string. If we assume those lengths are small and fixed (which is in fact the case for CS216 students), then this is still constant time as n grows. If we assume people have a fixed fraction of the rest of the world they wouldn't want to partner with, then we expect the lengths of the partner lists to grow as a fraction of n. This would make the overall running time of `assignPartners` in $O(n^3)$.

9. (6.37 / 10) Define the `allPossiblePartnerAssignments` procedure Zulma needs.

The easiest way to do this was to realize that we can find all possible partner assignments by arranging the students in all possible orders, and then just having them partner with the adjacent student. The subtlety (which no one got quite correctly) with this approach is dealing with the possibility that someones best match is with None (recall that the `goodnessScore` returns `-1` if either partner is `None`, but can return a more negative score if the one partner is on the others notpartners list). So, we need to consider all possible orderings of the students with the list extended with enough `None` values so each student could potentially be partnered with `None`. We do this by appending enough `None`s to the list before calling `allLists`. We use the `allLists` procedure from Section 3 to produce all possible lists.

```
def allPossiblePartnerAssignments (students):
   s = students[:] # we use a copy to avoid modifying input list
   for n in range (len (students)):
      s.append (None)
   for ordering in allLists (s):
      assignments = { }
      for i in range (len (ordering)):
         if i + 1 == len(ordering)
            assignments[ordering[i]] = None
         else:
            if not ordering[i] == None: # avoid adding partners for None
               assignments[ordering[i]] = ordering[i + 1]
            if not ordering[i + 1] == None:
               assignments[ordering[i + 1]] = ordering[i]
      yield assignments
```