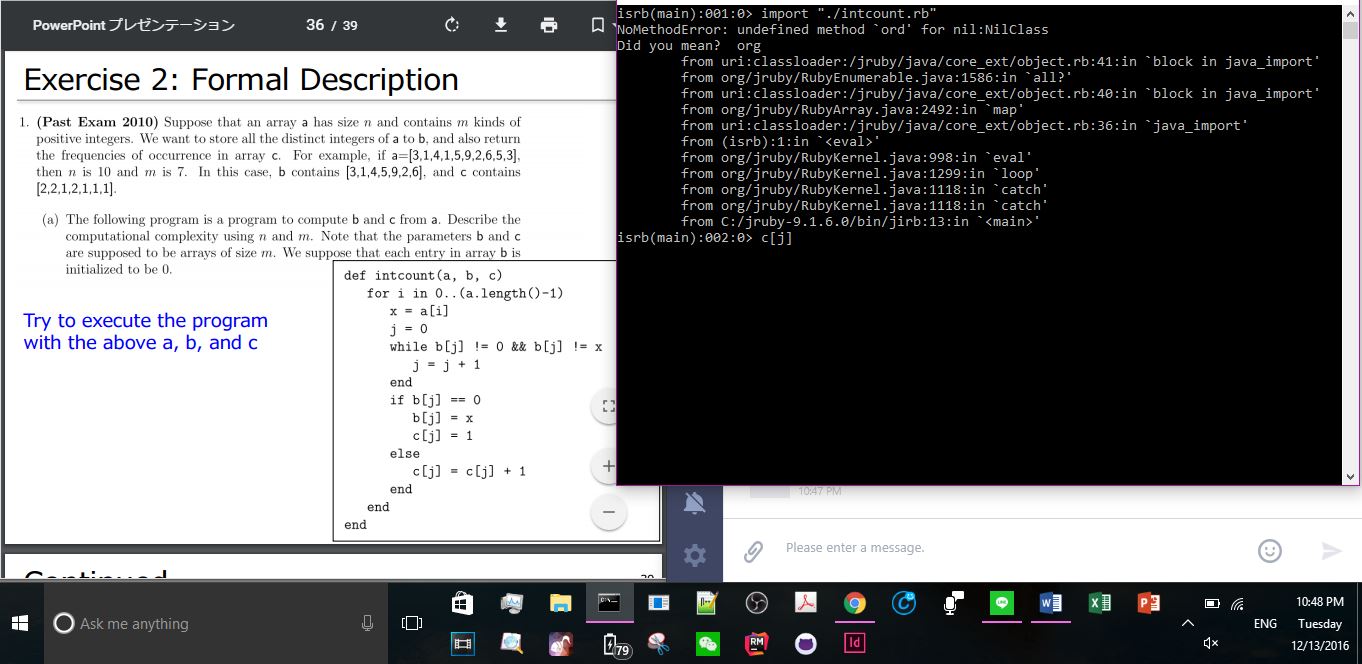
#580012-F  
#Week 9: Complexity

#Exercises Questions  
  
#Exercise 1: Software Processing Time

The easiest way to solve this question involves taking the limit towards infinity of either   
N^2 / (N\*log2 (N)) OR (N\*log2 (N)) / N^2 and seeing which one will evaluate towards infinity. The first case of (N\*log2 (N)) / N^2 will evaluate towards infinity, meaning that “B” will evaluate quicker with greater numbers of records. Hence, while A may be faster with a very small number of records, B is faster with a greater number.

#Exercise 2: Counting Data

For some reason, my computer will not run the intercount program, so I could not attempt to directly solve by example. Probably something to do with a broken part of the Java Library that JRuby or Oracle did not correctly code.



That said, the code itself is still analyzable. We first create the variable I, which will be a recursive storage variable for running through the program for the number of cases (numbers) that make up a part of the array “a”. Variable “x” is defined to be some value at the given “I”-Th position of a, while j is a constant 0, which we soon learn means the “0th position” (humanly, 1st number of the array). This brings us to a logical while statement, which analyses whether the j-Th position of array b is not the number 0 or the same value as x. In these special cases, we are told to cycle into the next position of b, as the sole purpose of this beginning part of the program is to weed out non-positive 0 and non-comparative b[j] = a[i] AND only solve the cases that are indeed possible. Again, remember that the problem defines two separate and given arrays, “a” or” b”, of either same or different lengths, and then an array c that orders a concatenation of a + b and then gives the integer frequency of occurrence of each positive number in order. OK, so back to reality. If we find that b[j] = 0, for some random position j in the array, then we call that b[j] = a[i] (in which we add in that value into a new re-writing of array a, which means that in the recursive check, it is a unique number. This means that we can call c[j] = 1. We then else this with a case where b[j] is not a value of 0, and we allow for b[j] to be occurring multiple times. This can be so if between array a and b, there is a value that occurs many times in both arrays, such as “3” in this array. Here, we cycle c[j] multiple times through, so that we can count in increasing amounts how many times that number occurs in the array “c”.

1. Knowing this, we know that increasing a marginal n will increase the number of calculations required by the total of all calculations times more, so every marginal amount is proportional to the O (n2) some constant to the power of n (Cn), because for every new number in n for array a and value x, we must calculate values for b, and c, and since c depends on b, it is like saying we are working with an array of b \* c for n^2. For “m”, we must marginally increase calculations to all of the arrays related to c, meaning we will get the O (n) typed complexity, according to similar examples given in the slides, because we are dealing with a single type array.
2. This part of the problem involves changing the program so that we are dealing in O (n) time. I think the last part with the c is OK, as it is a sum\_loop format, but we must de-interlace (separate) the b and c parts of this program for it to run in O (n) time. Literally, we will have to chunk this program into more byte-sized pieces.

def intcount\_loop(a,b,c)

for i in 0..(a.length()-1)

x = a[i]

j = 0

for j in 0…(b.length()-1)

if b[j] == 0

b[j] = x

else

b[j] = b[j] +1

end

if b[j] != 0 || b[j] != x

c[j] = 1

else

c[j] = c[j] + 1

end

end

end

end