CSC 320 Summer 2017: Assignment #1
Due at beginning of class, Fri. May 19

Read Chapter 1 of the text (reviews math prerequisites).

1. Put your name on your assignment. Questions should be **in order**.

2. Assignments are submitted on paper at the beginning of class on the due date.

3. Show your work unless otherwise stated.

4. Draw BIG boxes for your marks on the top of the first page of your submission. Place a 0 in the corresponding box for any questions you omit. For this assignment:

Question:	1	2	3	4	5	6	7	8	9	10
Marks	0	0	0	0	0	0	0	0	0	0

5. For this assignment, you should also upload two programs to connex: either gen.c or Gen.java for Question 3(a), and either new_ham.c or New_Hamilton.java for Questions 7-10. Please do not tar or zip or otherwise compress these files.

1. [Notation for languages] [10] List the elements of

   (a) $A \times \phi$ (ϕ denotes the empty set),

   (b) $A \times \{e\}$ (e denotes the empty string),

   (c) $A \times B$,

   (d) $A \times A$, and

   (e) $(A \times A) \times B$,

   where A= $\{s, q\}$ and $B = \{a, b, c\}$.

2. Use the following definition of Big Oh:
   Let $f(n)$ and $g(n)$ be functions which map the non-negative integers into the non-negative real numbers. A function $f(n)$ is in the set $O(g(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $f(n) \leq c\, g(n)$.

(a) [4] Prove that $f(n) = 2n^5 + 8n^4 + 3n^3 + 2n^2 + 4$ is in $O(n^5)$.

(b) [3] Suppose that an algorithm A takes at most k^3 steps on a problem of size k. Give a tight upper bound on the number of steps that algorithm A takes on a problem of size $2n^5 + n$. This question is asking for an algebraic expression.

(c) [3] Prove that algorithm A takes $O(n^{15})$ steps on a problem of size $2n^5 + n$.

3. [Review of induction and practice with strings] The aim of this question is to determine a closed formula for the number of strings on a set of s symbols that have length k that satisfy the property that no two consecutive places hold the same symbol. For example, when $s = 4$ and $k = 3$ and the alphabet chosen is {1, 2, 3, 4}, the number of strings is 36 and the strings are:

    121, 123, 124, 131, 132, 134, 141, 142, 143,
    212, 213, 214, 231, 232, 234, 241, 242, 243,
    312, 313, 314, 321, 323, 324, 341, 342, 343,
    412, 413, 414, 421, 423, 424, 431, 432, 434.

(a) [6] Write a recursive method or function that takes as input s and k (plus any other parameters you would like to use) and generates all the strings of length k that satisfy the property that no two consecutive places hold the same symbol. You can assume that s and k are at most 9. Use integers 1, 2, 3, 4, 5, 6, 7, 8, 9 to represent the symbols.

Use your function or method to generate the strings whose number of symbols s ranges from 1 to 7 and whose lengths vary from 1 to 7. Your program should summarize the counts in a table like this (I filled in the count of 36 that corresponds to the answer above, you should fill in all the values):

s / k	1	2	3	4	5	6	7
1	______	______	______	______	______	______	______
2	______	______	______	______	______	______	______
3	______	______	______	______	______	______	______
4	______	______	36	______	______	______	______
5	______	______	______	______	______	______	______
6	______	______	______	______	______	______	______
7	______	______	______	______	______	______	______

This table should be printed just before the program terminates execution. Hand in this table and the code you wrote with your written submission. The version of your program you hand in should print ONLY the chart and not the generated strings.

(b) [4] Conjecture a closed formula for the number of strings on a set of s symbols that have length k that satisfy the property that no two consecutive places hold the same symbol. Your formula should be a function of s and k. It would be wise to add code to your program to check to see if your conjectured formula matches the counts your program gives.

(c) [10] Prove your formula from (b) is correct by induction.

4. [10] Let $S = \{w : w$ is a string over $\Sigma = \{1, 2, 3, 4\}$ with no two consecutive places holding the same symbol $\}$. Is this set countable or uncountable? Prove your claim.

5. [10] Let $S = \{w : w$ is a (possibly infinite) sequence over $\Sigma = \{1, 2, 3, 4\}$ with no two consecutive places holding the same symbol $\}$. Is this set countable or uncountable? Prove your claim.

6. [10] Let $S = \{w : w$ is a (possibly infinite) sequence over $\Sigma = \{1, 2\}$ with no two consecutive places holding the same symbol $\}$. Is this set countable or uncountable? Prove your claim.

[Review of graph theory, solving problems using a black box approach]

Basic definitions: A path that contains every vertex of a graph is a *Hamiltonian path*. A cycle that contains every vertex of a graph is a *Hamiltonian cycle.*
HAMILTON PATH PROBLEM: Does graph G have a Hamiltonian path?
HAMILTON CYCLE PROBLEM: Does graph G have a Hamiltonian cycle?

I've written some code for the Hamilton Path Problem and the Hamilton Cycle Problem in C and in Java. You may select either language to do this question. Or you can use a C++ compiler with the C code (I wrote it so that it should compile as C or C++).

If you choose C, the files you require are:

1. The test driver for the program: main.c
2. The code for the functions **hamilton_path** and **hamilton_cycle**: ham.c
3. Templates for your new functions: new_ham.c
4. A sample input file: in.txt

Before you get started, copy these files, compile with :
```
gcc main.c ham.c new_ham.c
```
and run on the sample input file:
```
a.out < in.txt > out.txt
```
[on a unix machine, ask me if you do not know how to do this on your computer].

If you choose Java, the files you require are:

1. The test driver for the program (main) and the Graph class: Graph.java
2. Code to aid in the input (you do not need to know any details of this code): Keyboard.java

3. The code for the methods **hamilton_path** and **hamilton_cycle**: Hamilton.java
4. Templates for your new methods: New_Hamilton.java
5. A sample input file: in.txt

Before you get started, copy these files, compile with:
javac Graph.java Keyboard.java Hamilton.java New_Hamilton.java
and run on the sample input file:
java Graph < in.txt > out.txt
[on a unix machine, ask me if you do not know how to do this on your computer].
The output (out.txt) is identical for both the C and java programs.

The point of these questions is to prove that if either problem can be solved in polynomial time, so can the other. You must test your programs on the graphs in file *in.txt*. For programming credit for Questions 7 and 9, upload the file new_ham.c or New_Hamilton.java to connex. Do not modify any of the other files. In order to check your answers to Questions 8 and 10, please also print your routines from Questions 7 and 9 and hand them in with your written submissions.

7. [10] Write code for a new Hamilton Path routine **new_hamilton_path** which returns 1 if the graph it is called with has a Hamilton path and 0 otherwise. You may use my Hamilton Cycle routine **hamilton_cycle** as a black box. Explain in your comments why your algorithm always works. For full marks, your program must run in polynomial time given a **hamilton_cycle** routine that runs in polynomial time.

8. [5] Suppose that my **hamilton_cycle** routine takes time which is in $O(n^6 + m^4)$ where n is the number of vertices and m is the number of edges of the input graph. Express the time complexity of your code as a function of n and m .

9. [10] Write code for a new Hamilton Cycle routine **new_hamilton_cycle** which returns 1 if the graph it is called with has a Hamilton cycle and 0 otherwise. You may use my Hamilton Path routine **hamilton_path** as a black box. Explain in your comments why your algorithm always works. For full marks, your program must run in polynomial time given a **hamilton_path** routine that runs in polynomial time.

10. [5] Suppose that my routine **hamilton_path** takes time which is in $O(n^3 + m^2)$ where n is the number of vertices and m is the number of edges of the input graph. Express the time complexity of your code as a function of n and m.