**COT 6410 Exam I Key**

1) (a) is the Clique Problem, and it is known to be NP–Complete.

(b) is MaxClique, and it is not known to be in the set NP (let alone, NP–Complete) because we know no way of checking a Yes answer in deterministic polynomial time.

MaxClique is NP–Hard: because it is possible to provide a Turing Reduction from Clique. Let AMaxClique and AClique be the best algorithms for MaxClique and Clique, respectively.

1) Accept an arbitrary instance of Clique (G, k).

2) i 🡸 k

2) While i ≤ n and AMaxClique (G, i) = False, set i 🡨 i+1

3) If i = n+1 then return (False)

else return (True)

**Proof:** Suppose the given Clique instance is true. Then MaxClique must be true for some i such that k ≤ i ≤ n. Thus, in Step 3, i < n+1 and the result is True.

Suppose the instance of Clique is false. Then, for every i such that k ≤ i ≤ n, then AMaxClique will return False, and results in i = n+1 for step 3.

MaxClique is in P if and only if P = NP (i.e., in NP–Easy) because we can also design a Turing reduction from MaxClique to Clique.

1) Accept an arbitrary instance of MaxClique (G, k)

2) if AClique (G, k) = True and AClique (G, k+1) = False then return (True)

else return (False)

**Proof:** If an instance of MaxClique is True, there is a clique of size k, but none of size k+1. Therefore, on this graph, AClique will be true for k, but not for k+1.

If the instance of MaxClique is False, then either the largest clique is less than k, or greater than k. In the first case AClique(G, k) will be False and in the second AClique(G, k+1) will be true. In both cases, False will be returned.

2) P is the set of decision problems that have deterministic polynomial algorithms.

NP is the set of decision problems that have nondeterministic polynomial algorithms.

NP–Complete is the set of "hardest" Problems in NP. That is, problem X is in NP–Complete if and only if for every problem Y in NP there is a polynomial transformation reducing Y to X (Y 🡺p X).

(Note: it is not enough to say X is NP–Complete if X is in NP and there is a Y in NP–Complete such that Y 🡺p X. This calls for a first problem in NP–Complete. That is, a recursive definition must have a defined base case.)

3) There exists constants c and N such that f(n) ≤ cg(n) for all n ≥ N.

4) (a) (call it ProcAssignment) is in NP. For any Yes instance and an oracle provided solution of the instance:

For i = 1 to m

Add the tj's assigned to Pi

Compare this value with Ti (they will be equal, but you still must check it.)

This requires no more than O(mn) time and is clearly polynomial.

(b) The ProcAssignment problem is in NP–Complete.

Partition 🡺p ProcAssignment (others may also work – SubsetSum)

accept an arbitrary instance of Part (S, n)

B 🡨 sum of the si's.

ti 🡸 si for 1 ≤ i ≤ n

Create two processors P1 and P2 with T1 = T2 = B.

return(AProcAssingment(P, 2, T, n))

**Proof:** If the instance of Part is Yes, then that same partition into processor assigments will result in the instance of ProcAssignment being Yes. If there can be no partition of the instance of Part, then no partition to processors can exist.

5) X is in NP if and only if Co-X is in Co–NP. (Note that the Yes instances of Co-X are the No instance of X.) Yes instances of X can be verified in deterministic polynomial time. But, for many problems in NP – particularly those in NP–Complete – the No instances do not seem to be able to be verified by a polynomial deterministic algorithm. Thus, it "seems" the Co-NP and NP may not coincide.

On the other hand, for problems X in P there exists a deterministic polynomial algorithm which solves all instances – Yes and No ones. Therefore, both Yes and No instance can be verified in polynomial time. Hence, X and Co-X are both in P (and also in Co–P) thus, P = Co–P. (Note: to be able to verify X and Co–X in deterministic polynomial time does not place X in P. That is a conjecture believed to be true, but it is yet unproven.)

6) A 🡺p B.

A and B must both be decision problems (note, there is no requirement that they be in the same complexity class).

1) Accept an arbitrary instance of A, IA.

2) Transform IA into some instance, f(IA) of problem B.

so that IA is Yes in A if and only if f(IA) is Yes in B

3) The time, except for solving f(IA) with B's best algorithm is deterministic polynomial in terms of the length of IA.

7) The basic NDTM is a DTM with the "transition function" replaced by a "transition mapping" which allows the NDTM to be in multiple states at one time. From this, we can obtain the equivalent view of bounded parallelism. From this, we see that a Yes instance is simply a single execution path which can be directed by "oracles" stationed at each state with a nondeterministic transition – to only take the correct path leading to the accept state. This evolves into the "super oracle" with simply writes the correct sequence of states to visit on the input tape. We see from this, to compute the time, the correct answer to the instance will also provide the path timing information. Then, we have the "verifying" machine which accepts the Yes instance and answer and verifies its validity.

8) (a) A is "no harder than" B means we are able to show (prove) that instances of A can be solved by using an algorithm for B in such a way that the total time is no more that a polynomial multiple of the complexity of B. This process shows that

if B is polynomial then A is also polynomial. Equivalently, if A is proven to be exponential, then B is also exponential.

(b) One method for demonstrating this is via Turing Reductions, shown below (another is by the simpler "polynomial reduction – actually a special case of Turning Reductions).

1) Accept an arbitrary instance of problem A

2) create zero or more instances of problem B

3) solve the created instances by using B's best algorithm.

4) Compute the answer for the given instance of A.

The overall time must be polynomial except for the time taken by B's algorithm.