CS 3510 Honors Algorithms
Solutions : Midterm 2

1. Let A be an NP-Complete decision problem whose inputs are from a set S, and let A' be the same decision problem whose inputs are from a set T.

   a) If $S \subseteq T$ then A' is in NP.
   FALSE: If $S \subseteq T$, then you can of course check those instances from S with just a small amount of information. However, since T can contain additional inputs, you do not necessarily know that yes instances to these can also be checked efficiently with a small amount of information. For example, consider the class of problems that given a graph G answers "yes" if G is not 3-colorable. If S is bipartite graphs, then you can always check for a yes instance efficiently. However, if T contains a graph G that is not bipartite, there is no efficient way to check, with only a small amount of information, that G is not 3-colorable.

   b) If $T \subseteq S$ then A' is in NP.
   TRUE: If A is in NP, then for all instances in S there is polynomial information and an efficient procedure that will convince you that you have a yes instance. If T is a subset of these instances, then the same check will work on instances of T.

   c) If $S \subseteq T$ then A' is NP-hard.
   TRUE: If A is NP-hard, then there is a reduction from any NP-complete problem, say B, to A. The same reduction will map instances of B to A' as well, but of course it only maps to instances that are in S. Since $S \subseteq T$, this reduction will also show that A' is NP-hard.

   d) If $T \subseteq S$ then A' is NP-hard.
   FALSE: We know that there is a reduction from any NP-hard problem to A, but these might rely on mapping to instances in $S \setminus T$. In this case the reduction does not go through. If S is all SAT instances and T is just instances of 2-SAT, then A' is not NP-hard even though A is.

2. (a) If you have a single hash function h you use it to map 50 elements

from $\{0, \ldots, 199\}$ to $\{0, \ldots, 9\}$, what is the maximum number of elements that can be hashed to the same value?

Since nothing is specified about the hash function, it is possible that all the elements get mapped to the same value. So 50 is the possible maximum.

(b) If you have a 2-universal family of 25 hash functions from $\{0, \ldots, 199\} \rightarrow \{0, \ldots, 9\}$ and you notice that $h_1(5) = h_1(12)$, then what is the maximum number of hash functions from this family (besides h_1) that can map 5 and 12 to the same point?

This is a 2-universal family. So the probability of collision is atmost 1/10. Since there are 25 hash functions, there can be atmost $\lfloor 25/10 \rfloor = 2$ hash functions which collide for a specific pair of inputs. So at most one more hash function (besides h_1), can map 5 and 12 to the same point.

(c) Suppose $h_u : \{0, \ldots, 99\} \rightarrow \{0, \ldots, 9\}$ by setting $h_u(x) = x \pmod{10}$ and $h_t : \{0, \ldots, 99\} \rightarrow \{0, \ldots, 9\}$ by setting $h_t(x) = (x - h_u(x))/10$. Then $h_u(x)$ is the unit's digit of x and $h_t(x)$ is the ten's digit of x. Is this pair of functions 2-universal? Why or why not?

Consider numbers 26 and 27. $h_t(26) = h_t(27) = 2$. $h_u(26) = 6$ and $h_u(27) = 7$. So the probability of collision for this pair of numbers is $1/2 > 1/10$. So this pair of hash functions is not 2-universal.

3. Show that the following languages are NP-Complete. You can use the fact that the Hamiltonian Cycle and Independent set problems are NP-Complete, or any other NP-Complete problem that you know.

a) Half-Cycle = $\{G | G$ is an undirected graph on $2n$ vertices with a simple cycle on n vertices$\}$.

To show that Half-Cycle is in NP, we just note that if we are given the vertices in the cycle in order, then we can check that this cycle contains exactly half the vertices, that consecutive vertices are connected by an edge in the graph, and that each vertex that appears on the cycle appears only once. Each of these checks can be done in polynomial time.

We will now show that Half-Cycle is NP-hard by reducing from Hamiltonian Cycle to Half-Cycle. Let $G = (V, E)$ be an instance to Hamiltonian Cycle. We form $G' = (V', E')$ by defining V' as the set V with $|V|$ new vertices added and letting $E' = E$ (so we are just adding n independent vertices to the graph G). If G' is a yes instance to Half-Cycle, then there must be a cycle of length n among the first n vertices (those that came from V) since the new vertices are all disconnected. Therefore, this must constitute a Hamiltonian Cycle in G. Conversely, if there is a Hamiltonian Cycle in G, then this cycle has length n in G', a graph with $2n$ vertices. Therefore this cycle is a Half-Cycle in G'. This shows that Half-Cycle is NP-hard. Since it is also in NP, we have shown that Half-Cycle is NP-complete.

b) Indepedent-Set-or-Reverse = $\{(G, k)|G$ has an independent set of size k or the complement of G has an independent set of size k.

To show ISOR is in NP, just notice that if we have a yes instance, then there are k vertices that form an independent set or a clique. We can check that all pairs of vertices in this set are connected, or that they are all disconnected, in polynomial time.

We will show that ISOR is NP-hard by reducing from Independent Set. Let $G = (V, E), k$ be an instance of Independent Set. We form a new graph $G' = (V', E')$ by taking G and adding n new vertices. The edge set $E' = E$, and we add no additional edges. The input to ISOR is $(G', k + n)$. If $(G', k + n)$ is a yes instance of ISOR, then there is an independent set of size $k + n$ or a clique of size $k + n$ – however, the clique must be contained in the copy of G and therefore can only have at most n vertices. Therefore it must be that we have an independent set of size $k + n$. This must contain at least k vertices in G, so this is a yes instance of IS on (G, k). Conversely, if we have an independent set of size k in G, then these vertices together with the n new vertices in G' form an independent set of size $k + n$ in G', so this is also a yes instance of ISOR. Therefore, by this reduction, ISOR is NP-hard and consequently NP-complete.

4. Given an array of integers $a_1, a_2, \ldots, a_n$ positive and negative, such as $-1.3, 2, -7, 4, 2, -2, 3, -1$, you want to find the largest sum of contin-

guous integers. In this case it would be $4 + 2 - 2 + 3 = 7$.

We can accomplish this by dynamic programming. For each $i = 0, \ldots, n$, define **maxsum**$[i]$ to be the value of the largest sum seen so far. We also need **maxsuff**$[i]$ to be the largest sum of a suffix ending at a_i (In the array above, maxsum[4] = 5, maxsuff[4] = 0.)

Fill in the blanks in the dynamic programming algorithm:

(a) Initialize:

$$\text{maxsum}[0] = 0$$

$$\text{maxsuff}[0] = 0$$

(b) Iteration:
for $i = 0, \ldots, n-1$,

$$\text{maxsum}[i+1] = \max(\text{maxsum[i]}, \text{maxsuff[i+1]})$$

for $i = 0, \ldots, n-1$,

$$\text{maxsuff}[i+1] = \max(\text{maxsuff[i]} + \text{a}_{\text{i+1}}, 0)$$

(c) What other data structure do you need in order to recover in the end of the algorithm, the beginning and end of the maximum continguous sum?

We could have two more bit-arrays, indicating which of the two alternatives we picked for computing the maxsuff at each j. So after we find out the maxsum at the end of the algorithm, we could scan back through looking at which of the earliest entries were the maxsum equal to the overall maxsum. That will be the end of the longest contiguous sum. The beginning will be the last point before the end, where the maxsuff started from 0.

(d) What is the running time of the algorithm?

We do constant amount of comparisons at each stage. So the algorithm is linear time. Even the backtracking to find out the beginning and the end of the maxsum can be done in linear time, so the algorithm runs in linear time, $\Theta(n)$.