Data Structures and Algorithms II
Fall 2021 Section 2
Final Exam

Name:

Email:

Write all your answers directly within this test booklet. Use the backs of pages if necessary. This is an open-book, open-notes test. You should not use any electronic devices. You may use whatever scrap paper you like, but do not hand it in.

Please also write your name on the top of each page.

(1) If a problem, P_A, can be reduced in polynomial time to a problem, P_B, we can express this as follows: $P_A \leq_P P_B$.

Assume you have already proven that the 10 problems P_1 ... P_{10} are in complexity class NP. Further assume you know that the following reductions, involving the 10 problems P_1 ... P_{10}, are all possible:

$P_1 \leq_P P_5$
$P_2 \leq_P P_3$
$P_2 \leq_P P_8$
$P_3 \leq_P P_6$
$P_4 \leq_P P_3$
$P_5 \leq_P P_1$
$P_6 \leq_P P_4$
$P_6 \leq_P P_9$
$P_7 \leq_P P_2$
$P_{10} \leq_P P_7$

Note that these are not necessarily the only possible reductions involving these problems. In general, do not make any unstated assumptions.

(a) Assume that you discover a polynomial time reduction from 3SAT to P_2. (We discussed 3SAT in class; it is known to be NP-complete.) Which of the 10 NP problems P_1 ... P_{10}, if any, are definitely NP-complete (list all that apply)?

(b) Given the reduction from 3SAT to P_2, and also assuming that $P \neq NP$, which of the 10 NP problems P_1 ... P_{10}, if any, might be in P (list all that apply)?

(c) Which of the 10 NP problems P_1 ... P_{10}, if any, are definitely reducible in polynomial time to 3SAT (list all that apply)?

(2) Consider the bottom-up approach discussed in class for solving the longest common subsequence problem. For your convenience, I am repeating the pseudo-code here:

```
LongestCommonSubsequence ( String X, String Y )
  n ← length(X)
  m ← length(Y)
  loop i from 0 to m
    M[0,i] ← 0
  Loop i from 0 to n
    M[i,0] ← 0
  Loop i from 1 to n
    Loop j from 1 to m
      if Xi == Yj
        M[i, j] ← M[i-1, j-1] + 1
      else if M[i-1, j] ≥ M[i, j-1]
        M[i, j] ← M[i-1, j]
      else
        M[i, j] ← M[i, j-1]
```

(a) Assume that this approach is applied to find the longest common subsequence of "PETER" and "COOPER". Draw the matrix that would result from the application of the algorithm to this data.

Now consider the algorithm that uses the matrix, already filled in, to retrieve the longest common subsequence. For your convenience, I am repeating the pseudo-code here:

```
LongestCommonSubsequence ( String X, String Y, Matrix M )
  i ← length(X)
  j ← length(Y)
  subsequence ← empty string
  while i > 0 and j > 0
    if Xi == Yj
      subsequence ← Xi + subsequence
      i ← i - 1
      j ← j - 1
    else if M[i-1, j] ≥ M[i, j-1]
      i ← i - 1
    else
      j ← j - 1
```

(b) Starting at the bottom-right of the matrix that you drew in part (a), draw a jagged line (moving left, up, or diagonally as appropriate) showing the path taken through the matrix to retrieve the longest common subsequence for "PETER" and "COOPER". (I am not asking you to rewrite the matrix; draw your answer over the matrix that you drew for part (a).)

(c) What is the longest common subsequence of "PETER" and "COOPER"?

(d) When the first algorithm (to fill in the matrix) is applied to two strings of lengths N and M, what is the worst-case complexity of the algorithm, expressed using big-Theta notation?

(e) When the second algorithm (to retrieve the longest common subsequence) is applied to two strings of lengths N and M, what is the worst-case complexity of the algorithm, expressed using big-Theta notation?

(3) Briefly answer the following questions related to algorithm strategies.

(a) With a program that generates many pseudo-random numbers, how often should the pseudo-random number generator be seeded?

(b) What is the difference between an exhaustive search and a backtracking search?

(c) In class, we discussed a proof that Huffman encoding produces the optimal prefix code. What is a prefix code? Why is it useful to use a prefix code when encoding a file?

(d) Huffman encoding is an example of a greedy algorithm. Another example of a greedy algorithm covered earlier in our course was Dijkstra's algorithm. Why is Dijkstra's algorithm considered a greedy algorithm?

(e) State whether each of the following statements is true of bottom-up dynamic programming, top-down dynamic programming, both, or neither:

i. It relies on recursion.
ii. It can be much faster than a straight-forward divide-and-conquer algorithm because it avoids the need to recompute solutions to subproblems multiple times.
iii. It fills in the slots of a table or matrix in a specific order (e.g., one row at a time).
iv. For certain inputs, it is possible that not every slot in the table or matrix will get filled in on the way to the solution.

(4) One of the NP-complete problems discussed in class is the Independent Set Problem. A set of vertices I (a subset of vertices, V, in a graph G) is said to be independent if for all (i, j) such that i, j ε I, the edge (i, j) does not exist. The Independent Set Problem asks: Given an undirected graph, G, and a number, k, determine if there is any independent set of nodes, I, with at least k nodes. As explained in class, this is an NP-complete problem.

As defined here, the Independent Set Problem is a decision problem. The optimization version of the problem would ask, given an undirected graph, G, find the largest independent set, I, in the graph (ties could be broken arbitrarily).

For this problem, you are going to consider two algorithms (both described casually) that might be applied to the optimization version of this problem.

Algorithm I:

Start with an empty set of nodes, I. Choose the vertex, v, from the undirected graph, G, that touches the fewest incident edges, and add v to I. Then remove v, all of the vertices adjacent to v, and all of the edges that are incident to those vertices from G. Repeat this process until there are no vertices remaining in G. At the end of the algorithm, return I.

Algorithm II:

Start with an empty set of nodes, I. One at a time, try every possible vertex, v, from the undirected graph, G, as the first vertex to add to I. For each vertex, try adding every possible second vertex, w, to I, but if w is adjacent to v, remove w from I and try another vertex. For each pair of vertices, try adding every possible third vertex, x, to I, but if x is adjacent to v or w, remove x from I and try another vertex. Etc. Each time a new vertex is successfully added to I (without being adjacent to a vertex already in I), check if the current size of I is the largest you have seen so far, and if so, remember this version of I. At the end of the algorithm, return the final memorized version of I.

For each of the two algorithms above, answer the following questions:

i. Will the returned set of vertices necessarily be a valid independent set (not necessarily the largest)?
ii. Will the returned set of vertices necessarily be the optimal independent set (meaning the largest, or tied for the largest, that exists in the graph)?
iii. Is the running time of this algorithm polynomial or exponential?
iv. Which of the following categories best describes this type of algorithm strategy: greedy algorithm, divide-and-conquer algorithm, dynamic programming, randomized algorithm, exhaustive search, or backtracking search?

There is additional space to answer the four questions for each of the two algorithms on the next page.

(Extra space for question 4…)

(5) Briefly answer the following questions related to theoretical computer science.

(a) Assuming that $P \neq NP$, would it be possible to simulate everything that any nondeterministic Turing machine could compute with a regular Turing machine? Briefly explain your answer.

(b) Assume that somebody discovers a polynomial-time algorithm to solve the Hamiltonian Cycle Problem (which we know from class is NP-complete). Would this have any bearing, one way or the other, to the P=NP? question? Briefly explain your answer.

(c) Assume that somebody proves the Hamiltonian Cycle Problem (which we know from class is NP-complete) cannot be solved in polynomial time by any algorithm. Would this have any bearing, one way or the other, to the P=NP? question? Briefly explain your answer.

(d) Suppose that two problems, P1 and P2, are both known to be NP-complete. Is it necessarily the case that each one can be reduced to the other? Briefly explain your answer.

(e) Suppose that a system of mathematics, with enough expressivity to state all desired facts about arithmetic, is consistent (i.e., it has no contradictions). Is it necessarily the case that there exists some true statement, expressible within the system, that cannot be proven using the system? Briefly explain your answer.