Data Structures and Algorithms II
Fall 2021 Section 1
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 P_1 and P_2 are in complexity class NP. There is another problem, P_3, that may or may not be in NP. Finally, as discussed in class, we already know that the 3SAT problem is NP-complete.

For each of the below GIVEN statements, state whether the following OTHER statement is definitely true, definitely false, or possible (i.e., it could be true or false). Answer each question independently. Do not make any unstated assumptions. You do not have to explain your answers.

(a) GIVEN: $3SAT \leq_P P_1$
OTHER: P_1 is in NP-complete

(b) GIVEN: $P_1 \leq_P 3SAT$
OTHER: P_1 is in NP-complete

(c) GIVEN: $3SAT \leq_P P_1$ and $P_1 \leq_P P_2$
OTHER: P_2 is in NP-complete

(d) GIVEN: $3SAT \leq_P P_1$ and $P_1 \leq_P P_3$
OTHER: P_3 is in NP-complete

(e) GIVEN: $3SAT \leq_P P_1$ and $P_1 \leq_P P_2$
OTHER: P_2 is in NP-hard

(f) GIVEN: $3SAT \leq_P P_1$ and $P_1 \leq_P P_3$
OTHER: P_3 is in NP-hard

(g) GIVEN: $3SAT \leq_P P_1$ and $P \neq NP$
OTHER: P_1 is in P

(h) GIVEN: $P_1 \leq_P P_2$ and $P_2 \leq_P 3SAT$ and $P \neq NP$
OTHER: P_1 is in P

(i) GIVEN: P_1 is in P and $P \neq NP$
OTHER: $3SAT \leq_P P_1$

(j) GIVEN: P_1 is in P and $P \neq NP$
OTHER: $P_1 \leq_P 3SAT$

(2) Consider the following four versions of pseudo-code for computing the N^{th} Fibonacci number. As discussed in class, actual implementations would really be computing the right-most 32 bits of the N^{th} Fibonacci number. For Version #4 below, assume the existence of a global array called "known" that has been appropriately initialized.

Version #1:

```
Fibonacci( integer N )
  if N ≤ 2
    return 1

  Fib[1] ← 1
  Fib[2] ← 1
  Loop i from 3 to N
    Fib[i] ← Fib[i-1] + Fib[i-2]

  return Fib[n]
```

Version #2:

```
Fibonacci( integer N )

  if N ≤ 2
    v ← 1
  else
    v ← Fibonacci(N-1)+Fibonacci(N-2)

  return v
```

Version #3:

```
Fibonacci( integer N )
  prev1 ← 1
  prev2 ← 1

  if N ≤ 2
    return 1
  else
    Loop x from 3 to N
      cur ← prev2 + prev1
      prev2 ← prev1
      prev1 ← cur
  return cur
```

Version #4:

```
Fibonacci( integer N )
  if known[N] != unknown
    return known[N]

  if N ≤ 2
    v ← 1
  else
    v ← Fibonacci(N-1)+Fibonacci(N-2)

  known[N] ← v
  return v
```

For each of the four versions above, answer the following questions (there is additional space to answer these questions on the next page):

i. Is this an example of a divide-and-conquer algorithm, top-down dynamic programming, bottom-up dynamic programming, or none of these?

ii. Is the worst-case time complexity of the algorithm with respect to N linear, polynomial (but not linear), or exponential?

iii. If the function is called with an argument of 7 (i.e., 7 is passed to N), exactly how many total function calls will take place (including the original function call)?

(Extra space for question 2…)

(3) One of the NP-complete problems discussed in class is the Bin Packing Problem, which can be defined as follows. Suppose you have N items, and each item has a positive integer size. You also have a set of bins to place items in, and each bin has the same integer capacity, C. (We are not considering shapes of items; if an item has a size less than or equal to the remaining capacity of a bin, it can be placed in the bin.) The decision version of the problem asks if it is possible to place the N items in B or fewer bins. This problem is known to be NP-complete. The optimization version of the problem asks you to determine the minimum number of bins that can hold the N items.

For this problem, you are going to consider two algorithms (both described casually) that might be applied to the optimization version of this problem. We will assume that all items have a size that is less than or equal to C (otherwise, it is impossible to place all the items into bins).

Algorithm I:

Start with the first bin. Place the largest possible item that fits in it into the first bin. Then put in the next largest item that fits into the remainder of the first bin. Continue this process until no remaining item fits into the first bin. Then, move on to the second bin, and place the largest remaining item that fits in it into the second bin. Then put in the next largest item that fits into the remainder of the second bin. Continue this process until no remaining item fits into the second bin. Then, repeat the process with the third bin, the fourth bin, etc. Stop when all items have been placed into bins. At the end of the algorithm, return the set of bins.

Algorithm II:

Consider every possible permutation of the N items; there will be N! combinations. For each of the N! permutations, loop through the items in the order determined by the permutation. Put as many items that fit into the first bin, then put as many of the remaining items that fit into the second bin, then put as many of the remaining items that fit into the third bin, etc. When all items have been placed into bins, check if this is the smallest number of bins that has been used for any permutation so far; if so, remember this number and set of bins. At the end of the algorithm, return the final number and set of bins that was remembered.

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

i. Assuming that all items have a size that is less than or equal to C, is the algorithm guaranteed to eventually place all items into bins?
ii. Assuming that all items are eventually placed into bins, will the algorithm determine the optimal solution (i.e., the solution that uses the fewest possible bins)?
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 3…)

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

(a) Assuming P ≠ NP, can a regular Turing machine solve the Hamiltonian Cycle Problem, which is known to be NP-complete? Briefly explain your answer.

(b) We have learned that the complexity class called NP-complete is defined in terms of decision problems, which ask yes/no questions. Most of these problems have harder versions that are optimization problems, typically asking us to find the best (usually maximum or minimum) example of something. Consider such a minimization problem (the optimization version of some NP-complete problem). Assuming P ≠ NP, is it possible a that polynomial-time algorithm exists for some such problems that is guaranteed to find an answer at most 50% larger than the optimal answer? Briefly explain your answer.

(c) What is represented by the input to a universal Turing machine?

(d) If it turns out that P = NP (defying the expectations of most computer scientists), would it then be possible for a deterministic Turing machine to solve the halting problem? Would it be possible for a non-deterministic Turing machine to solve the halting problem? (I'm not asking for explanations.)

(e) Can the Boolean Satisfiability Problem (SAT) be reduced to every NP-hard question? Briefly explain your answer.

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

(a) Related to Huffman coding, we have learned that the optimal trie (representing an optimal prefix code) will always be a full binary tree, meaning that every node will either be a leaf or have two children (i.e., every node has 0 or 2 children, never exactly 1 child). Why?

(b) Suppose a Tower of Hanoi puzzle had 10 disks. Exactly how many moves would it take to solve the puzzle with the fewest possible moves? If the first move is move 1, the second move is move 2, etc., what would be the number of the move that moves the largest disk?

(c) Suppose a function called RAND returns a nonnegative random integer from 0 to MAX inclusive, where MAX is some large positive integer. Show how to use RAND to computer a random integer in the range of 100 to 200, inclusive.

(d) Consider a typical use of top-down dynamic programming, including memorization to remember already-computed sub-solutions to a problem in a matrix. Once a sub-solution gets recorded in the matrix, does it ever have to get updated? Briefly explain your answer.

(e) For an NP-complete problem such as the knapsack problem, can an exhaustive search theoretically be used to determine the answer?