"CS/ECE 374 A": Algorithms & Models of Computation, Spring 2025

Final Exam Solutions — May 15, 2025

Name:	
NetID:	

- Please *clearly PRINT* your name and your NetID in the boxes above.
- This is a closed-book but you are allowed a 2 pages (4 sides) hand written cheat sheet that you have to submit along with your exam. If you brought anything except your writing implements, put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear. The exam has 7 problems, each worth 10 points.
- **You have 180 minutes (3 hours) for the exam.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.**
- **Write everything inside the box around each page.** Anything written outside the box may be cut off by the scanner.
- **Proofs are required only if we specifically ask for them.** You may state and use (without proof or justification) any results proved in class or in the problem sets unless we explicitly ask you for one.
- You can do hard things!
- **Do not cheat.** You know the student code and all that jazz. Grades do matter, but not as much as you may think, and your values are more important.

---

1 Short Answer and True/False

(2.5 pts) Give asymptotically tight bounds for the following sum and recurrence:

(a) $\sum_{i=1}^{n} (\log i)^2$

Solution: $\sum_{i=1}^{n} = \Theta\left(n \log^2(n)\right)$. At least $n/2$ terms are at least $\log^2(n/2)$ ■

(b) $T(n) = 2T(n-2) + 1$ for $n > 2$; $T(n) = 1$ for $n \leq 2$.

Solution: $T(n) = \Theta(2^{n/2})$. $T(n) = 1 + 2 + 4 + \ldots + 2^{n/2-1} = 2^{n/2} - 1$ ■

Rubric: 1.25 points each. No work is required, so the points are all-or-nothing.

(7.5 pts) True/False: fill in the box corresponding to your answer.

(c) Every language in NP is decidable.

Solution: True. You can check every possible witness in exponential time. ■

(d) There is a polynomial-time reduction from 10-Color to 3-Color.

Solution: True. 10-Color is in NP and 3-Color is NP-complete. ■

(e) If L is regular, L^* is in P.

Solution: True. L^* is regular, and thus in P. ■

(f) Let G be a DAG such that for every pair of vertices u, v, either u can reach v or v can reach u. Then G has a Hamiltonian Path.

Solution: True. Adjacent vertices in a topological order of G must have an edge between them, so a topological order gives us a Hamiltonian path. ■

(g) Let $G = (V, E)$ be a connected graph with distinct edge weights. Then for every vertex v, the MST of G contains the lightest edge that is adjacent to v.

Solution: True. A special case of the lemma from lecture, where we take $S = \{v\}$. ■

(g) The MST of a graph G is unique if and only if its edge weights are distinct.

Solution: False. It is possible for the MST to be unique even with repeated edge weights—for example, consider a tree where every edge has weight 1. ■

(i) Let G be a graph with non-negative edge weights. Then the *maximum* weight spanning tree G can be computed in polynomial time.

Solution: True. Negate all the edges and compute Min-ST, which is the Max-ST for the original weights. ■

(j) Given a graph G where every edge has length 100 or 374, the shortest s-t path can be computed in linear time.

Solution: True. Replace each edge as many vertices as its weight, and run BFS. ■

(k) If L and its complement $\bar{L}$ are recursively enumerable then L is decidable.

Solution: True. Run M_L and $M_{\bar{L}}$ in parallel; one must halt. ■

(l) If L is recursively enumerable then L^* is recursively enumerable.

Solution: True. Try each possible split of the string in parallel. ■

Rubric: .75 points each. No justification is required, so the points are all-or-nothing.

2 Classification

(10 pts) For each of the problems below choose the *best* running time/complexity from the list below, based on what you know from the course. Fill in a **single circle** corresponding to your choice.

- L There is an algorithm that runs in linear time
- Q There is an algorithm that runs in quadratic time
- C There is an algorithm that runs in cubic time
- P There is a polynomial-time algorithm
- NP The problem is in NP
- D The problem is decidable
- U The problem is undecidable

(a) Given a directed graph G, check whether G is a DAG.

Solution: L. Run DFS and reject if a cycle exists. ■

(b) $L = \{\langle M \rangle, w \mid M \text{ rejects } w.\}$

Solution: U. Can reduce the Halting problem to this. ■

(c) Checking whether a given array A of n numbers is sorted.

Solution: L. Scan the array once, checking that each element is larger than the last. ■

(d) Computing the minimum sized vertex cover in a given tree T.

Solution: L. Perform a DP on the tree; each node will be visited once ■

(e) $L = \{\langle M \rangle, w \mid M \text{ is a TM that accepts } w \text{ in } |w|^4 \text{ steps.}\}$.

Solution: P. Let $k = |w|$. Simulate M for k^4 steps, which can be done in $O(k^{4+c})$ time for some constant c. (In fact, with care it can be done in $O(k^4 \log k)$ time.) ■

(f) Given undirected graph G with n nodes checking if G has an independent set of size at least $n - 10$.

Solution: **P.** There are $\binom{n}{10}$ ways to remove 10 nodes, so we can check each possibility in $O(n^{10+c})$ time for some constant c. ■

(g) Computing the longest increasing subsequence in a sequence of n numbers.

Solution: **Q.** Standard DP. ■

(h) Given three DFAs M_1, M_2, and M_3, constructing the DFA M that accepts $L(M_1) \cap L(M_2) \cap L(M_3)$ via the product construction.

Solution: **C.** There are $|Q_1| \cdot |Q_2| \cdot |Q_3|$ possible states for M, which takes cubic time to enumerate. ■

(i) Checking if G has a spanning tree with at most 2 leaves.

Solution: **NP.** A spanning tree with exactly 2 leaves is a Hamiltonian path. ■

(j) Finding the longest s-t path in a undirected graph.

Solution: **NP.** This is NP-hard by reduction from Hamiltonian path.
Technically, this problem isn't in NP since it is a search rather than a decision problem, so we will also accept **D** for this part. ■

Rubric: 1 point each. No justification is required, so the points are all-or-nothing.

3 Shortest Paths

Let $G = (V, E)$ be a directed graph with edge lengths $\ell(e), e \in E$. These lengths could be postive, zero, or negative. Let s, t be two distinct nodes in the graph. Describe an efficient algorithm that outputs the length of the shortest s-t walk in G that contains at most one negative length edge; note that if the walk contains a negative length edge then it can be used only once.

Solution: Since we can only take a single negative edge, it suffices to determine the path with only nonnegative edges and determine which negative edge gives us the shortest distance walk. We can only take a negative edge once, so negative cycles do not need to be handled. We perform the following:

- Remove all negative edges from G.
- Using Dijkstra's, compute the shortest path distances in G from s, stored in an array S.
- Using Dijkstra's, compute the shortest path distances *in the reverse of* G from t, stored in an array T.

Assume that if a node is unreachable, its distance is ∞. We run the following loop:

```
dist ← S[t]
for each negative edge (u, v):
    new_dist ← S[u] + d(u, v) + T[v]
    if ( new_dist < dist):
        dist ← new_dist
return dist
```

Removing all negative edges, iterating through them, and reversing G are all $O(|V| + |E|)$ time. Thus, the runtime is dominated by Dijkstra's, giving us $O(|E| \log |V|)$ time. ■

Rubric:

- 10 points for a fully correct $O(E \log V)$ solution.
- 7 points for a fully correct solution running in time $O(VE)$ (eg running Bellman-Ford *on a layered graph* or modifying the Bellman-Ford DP to enforce only one negative edge) or $O(VE \log V)$ (eg running Dijkstra's V times in the third step).
- 5 points for splitting G into 2 layers, with non-negative edges within each layer and negative edges from the first to the second. (Or otherwise splitting the walk into parts "before the negative edge" and "after the negative edge".)
- No credit for just running Bellman-Ford *on the unmodified input graph*, since that does not enforce the "only one negative edge" condition.

4 NP-Completeness

Given a graph $G = (V, E)$ a cycle cover is a set of cycles $\{C_1, C_2, \ldots, C_h\}$ for some $h \geq 1$ such that each vertex v is contained in at least one of the cycles. Note that the cycles are not required to be vertex or edge disjoint. Prove the problem of deciding whether a given graph G has a cycle cover with at most 3 cycles is NP-Hard.

Solution: We perform the following reduction from HamCycle:
Given a graph G, we construct a new graph H, which is 3 disjoint copies of G.
Claim 1. G has a Hamiltonian cycle $\iff$ H has a cycle cover with at most 3 cycles.

$\Longrightarrow$ If G has a Hamiltonian Cycle, then H has a cycle cover with at most 3 cycles:

Let the Hamiltonian cycle that covers G be C. Then, this cycle covers an entire copy of G in H. There are 3 copies of G in H, thus the mapping of C can be applied to all 3 copies, generating a valid cycle cover in H of at most 3 cycles.

$\Longleftarrow$ If H has a cycle over with at most 3 cycles, then G has a Hamiltonian Cycle:

H has at least 3 disjoint components. To fully cover H, at least 1 cycle is required in each component, and each cycle must cover every vertex in its component. Thus H has exactly 3 disjoint components, and G is connected. A cycle is then a Hamiltonian cycle in its component, and can be mapped as a Hamiltonian cycle in G.

Our reduction takes polynomial time, thus 3-CycleCover is NP-Hard. ■

Standard NP-hardness rubric. 10 points =

+ 1 point for choosing a reasonable NP-hard problem X to reduce from.
  - The Cook-Levin theorem implies that *in principle* one can prove NP-hardness by reduction from *any* NP-complete problem. What we're looking for here is a problem where a simple and direct NP-hardness proof seems likely.
  - You can use any of the NP-hard problems listed in the lecture notes (except the one you are trying to prove NP-hard, of course).
+ 2 points for a *structurally sound* polynomial-time reduction. Specifically, the reduction must:
  - take an *arbitrary* instance of the declared problem X ***and nothing else*** as input,
  - transform that input into one (or more) corresponding instance(s) of Y (the problem we're trying to prove NP-hard),
  - transform the output(s) of the oracle for Y into a reasonable output for X, and
  - run in polynomial time.

  (The output transformation is usually trivial.) This is strictly about the structure of the reduction algorithm, not about its correctness. No credit for the rest of the problem if this is wrong.
+ 2 points for a *correct* polynomial-time reduction. That is, assuming a black-box algorithm that solves Y in polynomial time, the proposed reduction actually solves problem X in polynomial time.
+ 2 points for the "if" proof of correctness. (If our input is in X, the reduction will return True.)
+ 2 points for the "only if" proof of correctness. (If our input is not in X, the reduction will return False)

+ 1 point for writing “polynomial time”

- An incorrect but structurally sound polynomial-time reduction that still satisfies half of the correctness proof is worth at most 5/10 (= 1 for reasonable reduction source + 2 for structural soundness + 2 for half of the proof)
- A reduction in the wrong direction is worth at most 1/10 (for choosing a reasonable problem)

5 Undecidability

Prove that $L = \{\langle M \rangle \mid M \text{ accepts } \textit{at most } 5 \text{ binary strings}\}$ is undecidable. You are not allowed to use Rice's theorem for this question.

Solution: We reduce from DecideHalt:

```
DecideHalt(⟨M, w⟩):
  Construct ⟨M'⟩:
      M'(x):
        run M on w
        return True

  if DecideL(⟨M'⟩) = True:
      return False
  else:
      return True
```

We prove our reduction is correct as follows:

$\Longrightarrow$ Suppose M halts on w.
Then M' accepts all strings.
So DecideL returns False
So our algorithm returns True

$\Longleftarrow$ Suppose M does not halt on w.
Then M' accepts no strings.
So DecideL returns True.
So our algorithm returns False.

In both cases our algorithm for DecideHalt returns the correct answer, however DecideHalt cannot exist, thus L must be undecidable. ■

Standard Undecidability reduction rubric. 10 points =

+ 2 points for a *structurally sound* reduction. Specifically, the reduction must:
  - declare what undecidable problem X you are reducing from,
  - take an *arbitrary* instance of the declared problem X ***and nothing else*** as input,
  - transform that input into one (or more) corresponding instance(s) of Y (the problem we're trying to prove undecidable),
  - transform the output(s) of the oracle for Y into a reasonable output for X, and
  - run in finite time.

  (The output transformation is usually trivial.) This is strictly about the structure of the reduction algorithm, not about its correctness. No credit for the rest of the problem if this is wrong.
+ 2 points for a *correct* reduction. That is, assuming a black-box algorithm that solves Y, the proposed reduction actually solves problem X.

+ 3 points for the "if" proof of correctness. (If our input is in X, the reduction will return True.)
+ 3 points for the "only if" proof of correctness. (If our input is not in X, the reduction will return False)

6 Work and Wait

You are given a sequence of n tasks. If you do the ith task, you receive a reward of Value$[i]$. However, this will also cause you to be too busy to do the next Wait$[i]$ tasks—that is, you will not be able to perform tasks $i+1$ through $i+\text{Wait}[i]$. Additionally, you can do at most k tasks total before you become too tired to do any more. Describe an algorithm that, on input Value$[1..n]$, Wait$[1..n]$, and k, computes the maximum total reward you can get. You can assume all the numbers are non-negative.

Solution: Let $MaxReward(i, j)$ be the maximum reward obtainable if we can only use tasks $i..n$ and have j remaining tasks in our budget. We want to compute $MaxReward(1, k)$. *MaxReward* obeys the following recurrence:

$$MaxReward(i,j) = \begin{cases} 0 & i > n \text{ or } j = 0 \\ \max(MaxReward(i+1,j), & \text{otherwise} \\ \qquad MaxReward(i+\text{Wait}[i]+1, j-1)+\text{Value}[i]) & \end{cases}$$

We can use a $n \times k$ table to memoize *MaxReward*. We will fill it in *decreasing* order with respect to i, and *increasing* order with respect to j. Each subproblem takes $O(1)$ time, thus to compute $MaxReward(1, k)$ takes $O(nk)$ time. ■

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - No credit if the description is inconsistent with the recurrence.
  - No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
  - − 2 for greedy optimizations without proof, even if they are correct.
  - **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**

+ 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)

+ 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***.
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points (= 70% of 3, rounded).

7 Regularity/DFA/NFAs

Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, A_1)$, $M_2 = (Q_2, \Sigma, \delta_2, s_2, A_2)$, and $M_3 = (Q_3, \Sigma, \delta_3, s_3, A_3)$ be DFAs and let L_1, L_2, L_3 be the languages accepted by them.

(a) (5 pts) Describe a DFA M that accepts the language $(L_1 - L_2) \cap L_3$.

Solution: We define the following DFA $M = (Q, \Sigma, \delta, s, A)$:

$$
\begin{aligned}
Q &= Q_1 \times Q_2 \times Q_3 \\
\Sigma &= \Sigma \\
\delta &:= \delta((q_1, q_2, q_3), a) = (\delta_1(q_1, a), \delta_2(q_2, a)\delta_3(q_3, a)) \quad \text{for } (q_1, q_2, q_3) \in Q, a \in \Sigma \\
s &= (s_1, s_2, s_3) \\
A &= \{(q_1, q_2, q_3) \mid q_1 \in A_1, q_2 \notin A_2, q_3 \in A_3\}
\end{aligned}
$$

■

Rubric:

- 2.5 points for using the product construction (either saying "product construction" or writing it out as above suffices). Partial credit if some of the parts are incorrect.
- 2.5 points for correctly defining the accepting states.

(b) (5 pts) Describe an algorithm that given M_1, M_2, M_3 checks whether $(L_1 - L_2) \cap L_3 = \emptyset$.

Solution: Construct M as in part (a) and convert it into its corresponding graph representation. Run BFS from the vertex that represents s, and determine if any state in A is reachable. If so, return false. Otherwise, return True.

The graph we construct will have $|V| = |Q|$ and and $|E| = |Q| \cdot |\Sigma|$, so the total run time of this algorithm is $O(|Q| \cdot |\Sigma|) = O(|Q_1| \cdot |Q_2| \cdot |Q_3| \cdot |\Sigma|)$. ■

Rubric:

- 2 points for constructing the graph corresponding to the DFA from part (a).
- 2 points for checking if any state in A is reachable. (Using a slower algorithm than WFS will cause the total points to be scaled for the slower overall run time.)
- 1 point for finding the runtime *in terms of the input parameters*. (Leaving it in terms of $|Q|$ is fine.)