Meta-rubric.

- These standard rubrics serve multiple purposes. They are a description for students of what we expect in their solutions; they are scaffolding to provide opportunities for partial credit; they are detailed instructions for graders, to help keep grading consistent. We reserve the right to modify these standard rubrics during the semester, but changes will never be applied retroactively.
- Each standard rubric describes partial credit for a problem worth 10 points. Partial credit for subproblems worth less than 10 points should be scaled appropriately. We'll typically round scaled partial credit to the nearest half-integer.
- Most rubrics can be summarized roughly as follows:
  + 2 points if the solution has the correct *syntax*.
  + 4 points if the solution has the correct *meaning*.
  + 4 points for a clear English explanation or proof.
- Many problems in this class have multiple correct solutions, sometimes using very different methods. (For example, most problems that can be solved by dynamic programming can also be solved by a graph reduction.) **Every submission is graded according to the rubric that best matches that submission**, even if that is different from the rubric that matches the official solution.
- Rubrics for individual problems sometimes vary from these standards. For example, if one part of a problem is significantly more difficult or "interesting" than in other problems of the same type, we may give that part more weight than indicated in the standard rubric.
- **The graders cannot read your mind, and they have been instructed not to try.** Submissions that are unclear for any reason — for example, sloppy handwriting, out-of-focus scanning, poor spelling or grammar, inconsistent variable names, not enough detail, too much detail — may be penalized or rejected entirely. The graders have *complete* discretion here; if a grader thinks your submission is unclear, then it is unclear. Similarly, if something in your solution is ambiguous, the graders have been explicitly instructed to interpret it the wrong way.
- Long and painful experience suggests that dismissive words like "clearly", "trivial", "simply", and "just" indicate likely errors, because the author trusted their intuition instead of working through the details. The graders will treat any submissions that include these words with extra scrutiny.

Standard induction rubric. 10 points =

- \+ 1 for explicitly considering an *arbitrary* object.
- \+ 2 for an explicit valid ***strong*** induction hypothesis
  - \+ 1 for an explicit valid *weak* induction hypothesis. Weak induction should die in a dumpster fire.
  - – Yes, we want you to write it down. Yes, even if it's "obvious". Remember that the point of writing any proof is to communicate with people who aren't as clever as you.
- \+ 2 for explicit exhaustive case analysis
  - – No credit here if the case analysis omits an infinite number of objects. (For example: all odd-length palindromes.)
  - – −1 if the case analysis omits a finite number of objects. (For example: the empty string.)
  - – −1 for making the reader infer the case conditions. Spell them out!
  - – No penalty if the cases overlap, or if the solution uses more cases than the reference solution.
- \+ 1 for cases that do not invoke the inductive hypothesis ("base cases")
  - – No credit here if one or more direct cases are missing.
- \+ 2 for correctly applying the ***stated*** inductive hypothesis
  - – No credit here for applying a ***different*** inductive hypothesis, even if that different inductive hypothesis would be valid.
- \+ 2 for other details in cases that invoke the inductive hypothesis ("inductive cases")
  - – No credit here if one or more "inductive cases" are missing.
- – ½ for arguing $(\forall k \leq n : P(k)) \Rightarrow P(n+1)$ (or worse, $P(k) \Rightarrow P(k+1)$) instead of arguing $(\forall k < n : P(k)) \Rightarrow P(n)$. Yes, we know it's mathematically correct, but it's bad style. Write your inductive proofs *exactly* the way you write your recursive algorithms, so that you can use the same neurons for both.

Standard regular expression rubric. 10 points =

- 2 points for a *syntactically* valid regular expression.
- 4 points for a correct regular expression. (8 points on exams, with all penalties doubled)
  - – 1 for a single mistake: one typo, excluding exactly one string in the target language, or including exactly one string not in the target language. (The incorrectly handled string is almost always the empty string ε.)
  - – 2 for incorrectly including/excluding more than one but a finite number of strings. (This rubric item is very rarely applied.)
  - – No credit (by default) for incorrectly including/excluding an infinite number of strings.
  - – No credit for answering with an incorrect expression equivalent to $\emptyset$ or Σ^*.
  - + Problem-specific partial credit for expressions with small structural mistakes. For example, if the construction involves two distinct cases, then a regular expression that handles one case correctly but not the other could receive 2 points out of 4.
  - + Every correct regular expression should receive full credit for correctness, not just the expression(s) in the official solutions. (But see the comment below about complexity.)
- **Homework only:** 4 points for a *brief* English explanation of your regular expression. This is how you argue that your regular expression is correct.
  - – No more than the partial credit for correctness. For example, if your solution received 3 points for correctness, then a perfectly clear English explanation is worth 3 more points.
  - – For longer expressions, you should explain each of the major components of your expression, and separately explain how those components fit together.
  - – We do not want a *transcription*; don't just translate the regular-expression *notation* into English. Your explanation should help convince the reader that you expression is correct.
- *Minimal* regular expressions are not required for full credit. However, expressions that are more complex than necessary may be penalized, and expressions that are *significantly* more complex than necessary may be marked "unclear" and not graded at all.

Standard DFA/NFA design rubric. 10 points =

- 2 points for an unambiguous, syntactically valid description of a DFA or NFA, including the states Q, the start state s, the accepting states A, and the transition function δ. Valid descriptions can take several different forms:
  - **Drawings:**
    - Use an arrow from nowhere to indicate the start state s.
    - Use doubled circles to indicate accepting states A. If $A = \emptyset$, write that explicitly.
    - If your drawing omits a junk/trash/dump/reject state, write that explicitly. Otherwise, there must be a visible transition from every state, for every symbol in Σ.
    - **Draw neatly!** If we can't read your solution, we can't give you credit for it.
  - **Text descriptions:** You can describe the transition function either using a 2d array (indexed by states and input symbols), using mathematical notation, or using an algorithm.
    - If your DFA description omits transitions to a junk/trash/dump/reject state, write that explicitly. Otherwise you must explicitly specify a single state $\delta(q, a)$ for *every* state q and *every* symbol a.
    - If your NFA omits transitions to the empty set, write that explicitly. Otherwise, you must explicitly specify a set of states $\delta(q, a)$ for *every* state q and *every* symbol a, and if you are describing an NFA with ε-transitions, you must explicitly specify a set of states $\delta(q, \varepsilon)$ for *every* state q.
  - **Product constructions:** Give a complete description of each of the DFAs you are combining (as either drawings, text, or recursive products), and explicitly describe the accepting states of the product DFA. In particular, we will *not* assume that product constructions compute intersections.
- 4 points for correctness. (8 points on exams, with all penalties doubled)
  - 1 for a single mistake: a single misdirected transition, a single missing or extra accepting state, rejecting exactly one string that should be accepted, or accepting exactly one string that should be accepted. (The incorrectly accepted/rejected string is almost always the empty string ε.)
  - 2 for incorrectly accepting/rejecting more than one but a finite number of strings.
  - No credit (by default) for incorrectly accepting/rejecting an infinite number of strings.
  - No credit for an incorrect DFA or NFA that either accepts every string or rejects every string
  + Problem-specific partial credit for automata with small structural errors. For example, if the construction involves two distinct cases, then a DFA that handles one case correctly but not the other may receive 2 points out of 4.
  + Every readable and correct automaton should receive full credit for correctness, not just the automata in the official solutions. (But see the comment below about complexity.)
- **Homework only:** 4 points for *briefly* explaining the purpose of each state *in English*. This is how you argue that your DFA or NFA is correct.
  - In particular, each state must have a mnemonic name.
  - For product constructions, explain the purposed of each state in each of the factor DFAs.
  - No more than the partial credit for correctness. For example, if your DFA received 3 points for correctness, then a perfectly clear English explanation of its states is worth 3 points.
- *Minimal* DFAs or NFAs are not required for full credit. However, automata that are more complex than necessary may be penalized, and automata that are *significantly* more complex than necessary may be marked "unclear" and not graded at all.

- Half credit for describing an NFA when the problem explicltly asks for a DFA.

Standard fooling set rubric. 10 points =

- 4 points for the fooling set:
  - + 2 for proposing an explicit infinite set F.
  - + 2 if the proposed set F is actually a fooling set for the target language.
  - — No credit for the proof if the proposed set is not a fooling set.
  - — No credit for the *problem* if the proposed set is finite.
- 6 points for the proof:
  - The proof must correctly consider *arbitrary* pairs of distinct strings $x, y \in F$.
    - — No credit for the proof unless both x and y are *always* in F.
    - — No credit for the proof unless x and y can be *any* pair of distinct strings in F.
  - + 2 for explicitly describing a suffix z that distinguishes x and y.
  - + 2 for proving either $xz \in L$ or $yz \in L$.
  - + 2 for proving either $yz \notin L$ or $xz \notin L$, respectively.

Alternate fooling set rubric. 10 points =

- 4 points for the fooling set:
  - + 2 for proposing an infinite fooling set $X = \{x_1, x_2, x_3, \ldots\}$, by explicitly defining a string x_i for each positive integer i.
  - + 2 if the proposed set X is actually a fooling set for the target language.
  - — No credit for the proof if X is not a fooling set.
  - — No credit for the *problem* if strings in X depend on more than one parameter (for example: 0^*1^*).
- 6 points for the proof:
  - The proof must correctly consider *arbitrary* indices $i < j$.
    - — No credit for the proof unless i and j can be *any* pair of distinct positive integers.
  - + 2 for explicitly describing a suffix z_{ij} that distinguishes x_i and x_j.
  - + 2 for proving either $x_i z_{ij} \in L$ or $x_j z_{ij} \in L$.
  - + 2 for proving either $x_j z_{ij} \notin L$ or $x_i z_{ij} \notin L$, respectively.

Standard language transformation rubric. 10 points =

+ 2 points for a formal, complete, and unambiguous description of the output automaton M', including the states, the start state(s), the accepting states, and the transition function, as functions of an *arbitrary* given DFA M. The description must state whether the output automaton is a DFA or an NFA, and if it is an NFA, whether it uses multiple start states and/or ε-transitions.
  - No points for the rest of the problem if this is missing.
  - Descriptions with minor syntax errors or omissions should receive appropriate partial credit, but syntactically incorrect or omitted components are automatically assumed to be incorrect.
+ 2 points for a *brief* English explanation of the output automaton. We explicitly do *not* want a formal proof of correctness, or an English *transcription*, but a few sentences explaining how your machine works and justifying its correctness. What is the overall idea? What do the states represent? What is the transition function doing? Why these accepting states?
+ 6 points for correctness
  + 1 for correct states — Almost always a product of the states Q of the given DFA with other side information; does the side information make sense? Could you build a transformation using *only* this side information?
  + 1 for correct start state(s)
  + 1 for correct accepting states
  + 3 for correct transition function
    — 1 for a single minor mistake
  - Double-check correctness when the input language is $\emptyset$, or $\{\varepsilon\}$, or 1^*, or Σ^*.
  - Partial credit should be awarded relative to the *most similar correct solution*. For example, if a given incorrect solution can be fixed either by changing the accepting states or by changing the transition function, it should get partial credit for a good transition function.

General instructions for algorithms. Full credit for ***every*** algorithm in the class requires a clear, complete, unambiguous description, at a sufficient level of detail that a strong student in CS 225 could implement it in their favorite programming language (which you don't know), using a software library containing every algorithm and data structure we've seen in CS 124, 173, 225, and 374. In particular:

- A complete description **always** includes the algorithm's running time, even if the problem statement does not include the word "analyze". (On the other hand, unless explicitly stated otherwise, you do not need to analyze the algorithm's space usage.)
- Watch for hand-waving, pronouns without clear antecedents (especially "this"), overconfidence flags like "simply" or "just", meaningless filler words (like "go through the array and"), and the Deadly Sins "repeat this process" or "Do this for all n."
- Don't regurgitate algorithms we've already seen. We've read the textbook. We assume that you've also read the textbook, and that you know that we've read the textbook. Moreover, we assume that you know that we know that you've read the textbook. (Most LLMs have not read the textbook.)
- Every algorithm must be clearly ***specified*** in English, unless the specification is already given *precisely* in the problem statement. A description of the algorithm is not enough; we want a description of *what* your algorithm computes, not just an explanation of *how* your algorithm works.

  In particular, if the algorithm solves a more general problem than requested, you must explicitly specify the more general problem. Similarly, if the algorithm assumes any conditions that are not explicit in the given problem statement, you must state those assumptions explicitly.
- The meaning of every variable must be either clear from context (like n for input size, or i and j for loop indices) or specified explicitly.
- **Target time bounds.** Every algorithm design question has a target time bound. Faster algorithms are worth more points, and slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n in either direction. Partial credit is ***SCALED*** up or down to the new maximum score.

  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students submit incorrect algorithms with the target running time (earning 0/10) instead of correct algorithms that are slower than the target (earning 7/10).

  Yes, it is possible to score more than full credit, even on exams. Past students have earned homework averages and/or exam scores over 100% by submitting faster-than-expected algorithms; a very small number have even earned overall *course* averages over 100%.

  On the other hand, every completely specified and correct algorithm is worth at least 3/10, regardless of its running time. Yes, even if the running time is triply exponential. But watch for hand-waving like "for every subset" or "try every path"; if you want to exhaustively enumerate subsets or paths or some other structure, you have to tell us how.
- Partial credit for incomplete solutions should be based on the running time of the ***best possible*** completion (up to the target running time).

Standard recursive algorithm rubric. 10 points =

- 3 for a clear and correct English description of the problem that your recursive algorithm solves.
  - This is unnecessary if (and only if) a precise description is already given to you. But if your algorithm solves a more general problem, you need to describe that more general problem, Similarly, if your algorithm requires assumptions or conditions that were unstated in the given problem statement, you need to describe those assumptions or conditions.
  - An English explanation of the *algorithm* is not enough. We want a description of *what* your algorithm computes, not (here) an explanation of *how* it works.
- 1 for base case(s)
- 2 for correctly identifying/computing recursive subproblems (“divide”)
- 2 for correctly using the results of recursive calls (“conquer”)
- 2 for time analysis = 1 for correct recurrence + 1 for closed-form solution
- This rubric applies to both divide-and-conquer and backtracking algorithms.

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.)
  - The description must satisfy the following conditions to receive **any** credit:
    - It must have explicit input parameters, and it must explicitly describe how the function value depends on those parameters.
    - It must be consistent with the submitted recurrence. (If the submitted solution does not include a correct recurrence, the description must be consistent with at least one correct recurrence.)
    - It must not refer to internal states of the eventual iterative dynamic programming algorithm, like "the current index" or "the best score so far". The described function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - + 1 for clearly specifying the portion of the top-level input considered by each recursive subproblem (for example: a prefix, a suffix and one other array value, an interval, a pair of prefixes, or a subtree), even if the rest of the description is incomplete or incorrect.
  - – 1 for naming the function any single letter, "OPT", "dp", "memo", or anything similarly undescriptive.
  - An English explanation of the *recurrence* or the *algorithm* is not enough. We want a description of *what* your function returns, not (here) an explanation of *how* that value is computed.
- 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.
  - — 1½ 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 need 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.

- 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***, or at most 5 points out of 10 if you name your memoization array any single letter, "OPT", "dp", "memo", or anything similarly undescriptive".

Standard graph-reduction rubric. 10 points =

- + 3 for constructing the correct graph.
  - + 1 for correct vertices
  - + 1 for correct edges
  - — ½ for forgetting "directed" if the graph is directed
  - + 1 for correct weights, costs, labels, or other annotations, if any
  - ◦ The vertices, edges, and so on must be described as explicit functions of the input data.
  - ◦ For most problems, the graph can be constructed in linear time by brute force; in this common case, no explicit description of the construction algorithm is required. If achieving the target running time requires a more complex algorithm, that algorithm will graded out of 5 points using the appropriate standard rubric, and all other points are cut in half.
- + 3 for explicitly relating the given problem to a specific **problem** involving the constructed graph. For example: "The minimum number of moves is equal to the length of the shortest path in G from the start vertex $(0, 0, 0)$ any target vertex of the form $(k, \cdot, \cdot)$ or $(\cdot, k, \cdot)$ or $(\cdot, \cdot, k)$." or "There is a French-flag walk from s to t in G if and only if $(s, 0)$ can reach $(t, 0)$ in H."
  - + 1 for describing the correct high-level problem (for example, "shortest path" or "reachability")
  - + 2 for all other details. For example: Shortest path in which graph, from which vertex to which other vertex? How does that shortest path relate to the original problem?
  - – No points here for only naming the algorithm, not describing the problem. "Breadth-first search" is not a problem!
- + 2 for correctly applying the correct **algorithm** to solve the stated problem. (For example, "Perform a single breadth-first search in H from $(0, 0, 0)$ and then examine every target vertex." or "Whatever-first search in H.")
  - — 1 for using a slower algorithm than necessary, for example, Dijkstra's algorithm instead of breadth-first search.
  - — ½ for using a more specific algorithm than necessary, for example, breadth- or depth-first search instead of whatever-first search.
  - — 1 for explaining an algorithm from lecture, the textbook, or a prerequisite class instead of just invoking it as a black box.
- + 2 for time analysis in terms of the *input* parameters (not just the number of vertices and edges of the constructed graph).
- ★ An extremely common mistake for this type of problem is to attempt to modify a standard algorithm and apply that modification to the input data, instead of modifying the input data and invoking a standard algorithm as a black box. This strategy can work in principle, but it is much harder to do it correctly, and it is terrible software engineering practice. ***Clearly correct*** solutions using this strategy will be given full credit, but partial credit will be given only sparingly. Really, just do it the other way.
- ★ Another common mistake is to "try all possible paths using (for example) depth-first search and return the best one". That is a reasonable description of a *problem*, but not a good description of an *algorithm*. First, every variant of whatever-first search finds ***exactly one*** path from the start vertex to each reachable vertex. Second, in most graphs, the number of *simple* paths (which don't repeat vertices) is exponential in the number of vertices, and the number of *non-simple* paths is infinite! So even if you could explore all possible paths, you really don't want to!

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 a corresponding instance of Y (the problem we're trying to prove NP-hard),
  - transform the output 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. (Every good instance of X is transformed into a good instance of Y.)
+ 2 points for the "only if" proof of correctness. (Every bad instance of X is transformed into a bad instance of Y.)
+ 1 point for writing "polynomial time" (but only if the reduction is correct)

- 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)

Standard rubrics for undecidability proofs.

- **Diagonalization:**
  - \+ 4 for correct wrapper Turing machine
  - \+ 6 for self-contradiction proof (= 3 for $\Leftarrow$ + 3 for $\Rightarrow$)
- **Reduction:**
  - \+ 4 for correct reduction
  - \+ 3 for "if" proof
  - \+ 3 for "only if" proof
- **Rice's Theorem:**
  - \+ 4 for positive Turing machine
  - \+ 4 for negative Turing machine
  - \+ 2 for other details (including using the correct variant of Rice's Theorem)