CSCI 432/532, Spring 2026

Homework 2

Due Monday, February 2 at 9am

Submission Requirements

- Type or clearly hand-write your solutions into a PDF format so that they are legible and professional. Submit your PDF on Gradescope.
- Do not submit your first draft. Type or clearly re-write your solutions for your final submission. If your submission is not legible, we will ask you to resubmit.
- Use Gradescope to assign problems to the correct page(s) in your solution. If you do not do this correctly, we will ask you to resubmit.

Collaboration and Resources

The *best* resources for completing the homework are (in no particular order):

- Your own notes from lectures and problem sessions, and/or lecture recordings.
- The lecture notes linked from the course website for the lecture day.
- The questions channel on Discord.
- Office hours.
- Discussions with classmates.

You may also access almost any resource in preparing your solution to the homework. The exception is that you may not seek answers to the problems on the homework directly, such as by pasting them into a GenAI tool, searching for solutions on a search engine, looking for them on Chegg or similar websites, or asking another person for the solution.

However, you ***must***

- Write your solutions in your own words—this means that you should never be *copying* anything of substance from any source, and
- credit every resource you use, including GenAI tools, by providing links or full chat transcripts. If you use the provided LaTeX template, you can use the `Sources` environment for this. Ask if you need help!

Remember, if you choose to use GenAI tools, not only must you provide a full transcript of your conversation (or a link to one), you must also use the following prompt (or some variation of it), and provide a full transcript of the conversation.

Prompt:

You are acting as a teaching assistant for an advanced undergraduate/graduate algorithms and computer science theory course.
Your role is not to provide full solutions or final answers.
Instead, you should act like we are having a conversation. Ask me a single question and let me respond. Avoid asking me many questions at once or giving me a lot of information at one time. Guide me using hints, leading questions, partial insights, and sanity checks. Help me debug my own proofs, algorithms, or reductions. Do not give a complete solution or full proofs. Be patient with me. Don’t give me details so that I can move on to the next part of a problem. Make sure that I understood before moving on.
The course uses Jeff Erickson’s Models of Computation lecture notes, so you should guide me to answer problems using the definitions, notation, and style from those notes.
Assume I am a serious student trying to learn, not to shortcut the assignment. I am attaching the assignment, so you should refuse to provide a complete solution to the problems listed. Of course, you can walk me through the "Solved Problems" listed at the end if I ask.

1. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, give an equivalent regular expression, and briefly explain why your regular expression is correct. Note that there are infinitely many correct answers for each language.

   (a) (3 points) All strings that begin with the prefix `001`, end with the suffix `100`, and contain an odd number of `1`'s.

   (b) (3 points) All strings that contain both `0011` and `1100` as substrings.

   (c) (4 points) All strings that contain the substring `01` an odd number of times.

Regular expression rubric. 10 points =

- + 2 for a syntactically correct regular expression.
- + 4 for a brief English explanation of your regular expression. This is how you argue that your regular expression is correct.
  - 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.
  - Yes, you need to write it down. Yes, even if it's "obvious". Remember that the goal of the homework is to communicate with people who aren't as clever as you.
- + 4 for correctness.
  - −4 for incorrectly answering $\emptyset$ or Σ^*.
  - −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.
  - −4 for incorrectly including/excluding an infinite number of strings.
- Regular expressions that are more complex than necessary may be penalized. Regular expressions that are significantly too complex may get no credit at all. On the other hand, minimal regular expressions are not required for full credit.

2. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, describe a DFA that accepts the language, and briefly describe the purpose of each state. You can describe your DFA using a drawing or using formal mathematical notation.

   (a) All strings in 1^*01^*.

   (b) All strings containing the substring `01010010`.

   (c) All strings that represent a multiple of 5 in base 3. For example, this language contains the string `10100`, because $10100_3 = 90_{10}$ is a multiple of 5. (In general, numbers in base 3 could contain the symbol `2`, but those are excluded from this language.)

NFA/DFA rubric. 10 points =

- + 2 for an unambiguous description of a DFA or NFA, including the states set Q, the start state s, the accepting states A, and the transition function δ.
  - Drawings:
    - Use an arrow from nowhere to indicate the start state s.

    - ⋆ Use doubled circles to indicate accepting states A.
    - ⋆ If $A = \emptyset$, say so explicitly.
    - ⋆ If your drawing omits a junk/trash/reject/hell state, say so explicitly.
    - ⋆ 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, using mathematical notation, or using an algorithm.
    - ⋆ You must explicitly specify $\delta(q, a)$ for every state q and every symbol a.
    - ⋆ If you are describing an NFA with ε-transitions, you must explicitly specify $\delta(q, \varepsilon)$ for every state q.
    - ⋆ If you are describing a DFA, then every value $\delta(q, a)$ must be a single state.
    - ⋆ If you are describing an NFA, then every value $\delta(q, a)$ must be a set of states.
    - ⋆ In addition, if you are describing an NFA with ε-transitions, then every value $\delta(q, \varepsilon)$ must be a set of states.
  - – Product constructions: You must give a complete description of each of the DFAs you are combining (as either drawings, text, or recursive products), together with the accepting states of the product DFA. In particular, we will not assume that product constructions compute intersections by default.

+ 4 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, explaining the states in the factor DFAs is both necessary and sufficient.
  - – Yes, we mean it. A perfectly correct drawing of a perfectly correct DFA with no state explanation is worth at most 6 points.

+ 4 for correctness.
  - – −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.)
  - – −4 for incorrectly accepting every string, or incorrectly rejecting every string.
  - – −2 for incorrectly accepting/rejecting more than one but a finite number of strings.
  - – −4 for incorrectly accepting/rejecting an infinite number of strings.

Solved problems

4. ***C comments*** are the set of strings over alphabet $\Sigma = \{$`⋆`, `/`, `A`, `⋄`, `↲`$\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here `↲` represents the newline character, `⋄` represents any other whitespace character (like the space and tab characters), and `A` represents any non-whitespace character other than `⋆` or `/`.[1] There are two types of C comments:

   - Line comments: Strings of the form `//`$\cdots$`↲`
   - Block comments: Strings of the form `/⋆`$\cdots$`⋆/`

   Following the C99 standard, we explicitly disallow ***nesting*** comments of the same type. A line comment starts with `//` and ends at the first `↲` after the opening `//`. A block comment starts with `/⋆` and ends at the the first `⋆/` completely after the opening `/⋆`; in particular, every block comment has at least two `⋆`s. For example, each of the following strings is a valid C comment:

   `/⋆⋆⋆/`   `//⋄//⋄↲`   `/⋆///⋄⋆⋄↲⋆⋆/`   `/⋆⋄//⋄↲⋄⋆/`

   On the other hand, *none* of the following strings is a valid C comment:

   `/⋆/`   `//⋄//⋄↲⋄↲`   `/⋆⋄/⋆⋄⋆/⋄⋆/`

   (Questions about C comments start on the next page.)

[1]The actual C commenting syntax is considerably more complex than described here, because of character and string literals.
- The opening `/⋆` or `//` of a comment must not be inside a string literal (`"`$\cdots$`"`) or a (multi-)character literal (`'`$\cdots$`'`).
- The opening double-quote of a string literal must not be inside a character literal (`'"'`) or a comment.
- The closing double-quote of a string literal must not be escaped (`\"`)
- The opening single-quote of a character literal must not be inside a string literal (`"`$\cdots$`'`$\cdots$`"`) or a comment.
- The closing single-quote of a character literal must not be escaped (`\'`)
- A backslash escapes the next symbol if and only if it is not itself escaped (`\\`) or inside a comment.

For example, the string `"/⋆\\\"⋆/"/⋆"/⋆\"/⋆"⋆/` is a valid string literal (representing the 5-character string `/⋆\"\⋆/`, which is itself a valid block comment!) followed immediately by a valid block comment. ***For this homework question, just pretend that the characters*** `'`***,*** `"`***, and*** `\` ***don't exist.***

Commenting in C++ is even more complicated, thanks to the addition of *raw* string literals. Don't ask.

Some C and C++ compilers do support nested block comments, in violation of the language specification. A few other languages, like OCaml, explicitly allow nesting block comments.

(a) Describe a regular expression for the set of all C comments.

Solution:

$$\mathsf{//}(\mathsf{/} + \star + \mathsf{A} + \diamond)^* \hookleftarrow \quad + \quad \mathsf{/}\star\big(\mathsf{/} + \mathsf{A} + \diamond + \hookleftarrow + \star\star^*(\mathsf{A} + \diamond + \hookleftarrow)\big)^*\ \star^*\star\mathsf{/}$$

The first subexpression matches all line comments, and the second subexpression matches all block comments. Within a block comment, we can freely use any symbol other than $\star$, but any run of $\star$s must be followed by a character in $(\mathsf{A} + \diamond + \hookleftarrow)$ or by the closing slash of the comment. ■

Rubric: Standard regular expression rubric. This is not the only correct solution.

(b) Describe a regular expression for the set of all strings composed entirely of blanks ($\diamond$), newlines ($\hookleftarrow$), and C comments.

Solution:

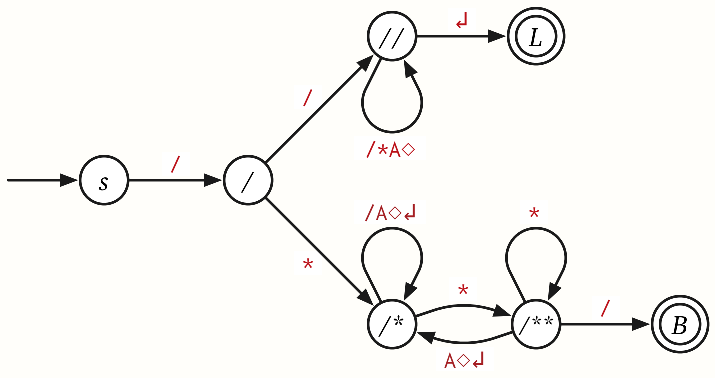
$$\big(\diamond + \hookleftarrow \ + \ \mathsf{//}(\mathsf{/} + \star + \mathsf{A} + \diamond)^*\hookleftarrow + \mathsf{/}\star(\mathsf{/} + \mathsf{A} + \diamond + \hookleftarrow + \star\star^*(\mathsf{A} + \diamond + \hookleftarrow))^*\star\star^*\mathsf{/}\big)^*$$

This regular expression has the form $(\langle\mathsf{whitespace}\rangle + \langle\mathsf{comment}\rangle)^*$, where $\langle\mathsf{whitespace}\rangle$ is the regular expression $\diamond + \hookleftarrow$ and $\langle\mathsf{comment}\rangle$ is the regular expression from part (a). ■

Rubric: Standard regular expression rubric. This is not the only correct solution.

(c) Describe a DFA that accepts the set of all C comments.

Solution: The following eight-state DFA recognizes the language of C comments. All missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

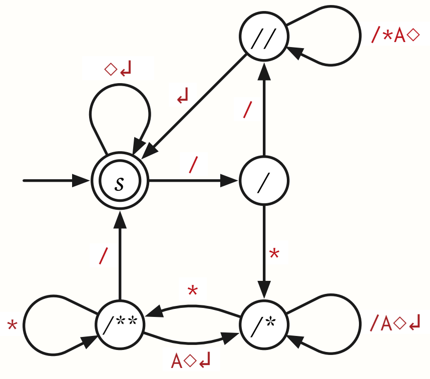
- *s* — We have not read anything.
- / — We just read the initial /.
- // — We are reading a line comment.
- *L* — We have just read a complete line comment.
- /* — We are reading a block comment, and we did not just read a ⋆ after the opening /⋆.
- /** — We are reading a block comment, and we just read a ⋆ after the opening /⋆.
- *B* — We have just read a complete block comment.

■

Rubric: Standard DFA design rubric. This is not the only correct solution, or even the simplest correct solution. (We don't need two distinct accepting states.)

(d) Describe a DFA that accepts the set of all strings composed entirely of blanks (◇), newlines (↲), and C comments.

Solution: By merging the accepting states of the previous DFA with the start state and adding white-space transitions at the start state, we obtain the following six-state DFA. Again, all missing transitions lead to a hidden reject state.



The states are labeled mnemonically as follows:

- *s* — We are between comments.
- / — We just read the initial / of a comment.
- // — We are reading a line comment.
- /* — We are reading a block comment, and we did not just read a ⋆ after the opening /⋆.
- /** — We are reading a block comment, and we just read a ⋆ after the opening /⋆.

■

Rubric: Standard DFA design rubric. This is not the only correct solution, but it is the simplest correct solution.