Analysis of Algorithms
CS 375, Fall 2022
Project 3
Due **BY 11:59pm** on **Saturday, Nov. 19**

Project 3: Recursion and Logic!

In this assignment, you'll work in teams of four to design algorithms for working with propositional logic expressions, and to use inductive arguments and recurrences to explain the correctness and complexity of recursive algorithms. The goals of this project are:

- to give you practice designing recursive algorithms and working with recursive definitions;
- to give you practice using inductive arguments and recurrences to explain correctness and complexity of recursive algorithms;
- to give you practice thinking about and working with propositional logic;
- to give you practice creating and giving a technical presentation; and
- to give you practice working with other students as a team.

The Project Assignment

Here are the parts of this assignment:

1. **Propositional Logic Evaluation** Programming languages, as part of their work, have algorithms to evaluate boolean expressions. Now it's your turn! Your team will design and explain an algorithm to evaluate propositional logic expressions.
2. **The *Satisfiability* Problem** The *Satisfiability* problem (or *SAT*, for short) is a classic CS problem: Given a boolean expression, is there any way to assign *True* or *False* to its variables to satisfy the expression—that is, to make it evaluate to *True*? Your team will write an exhaustive search algorithm to solve this problem.
3. **Improvements** Your team will improve upon your exhaustive search algorithm for *SAT*.
4. **Create and Give a Presentation** Your team will present your work, including explanations of correctness and complexity of your algorithms.

The project involves working with recursive algorithms, so your correctness and complexity arguments will involve *inductive arguments* and *recurrences*. There are things to do for each part of the assignment, as described individually below.

As usual in CS375, *excessively* inefficient or clunky algorithms may not receive full credit. That doesn't mean all algorithm designs must be optimally efficient, but if your algorithm does substantially more work than a straightforward brute force approach might, it may not earn full credit. Please feel free to ask me clarifying questions about this, if you'd like!

1 Propositional Logic Evaluation

Programming languages need definitions of arithmetic and boolean logic expressions, so that a compiler or interpreter can check expressions for type correctness and evaluate them as part of computation. The structures and definitions underlying this work are typically recursive (e.g., *grammars* that define what a valid arithmetic or boolean expression could be). For this part of your project, you'll be given a recursive definition for propositional logic expressions—expressions that evaluate to *True* or *False*—and you'll create a recursive algorithm to evaluate these expressions!

1.1 Our Definition of Propositional Logic Expressions

Propositional logic expressions—or *PLEs*, for short—are combinations of variables and logical operators such that, if we assign a value of either *True* or *False* to every variable, the expression would evaluate to either *True* or *False*. For examples:

- Any variable that we define as a *propositional variable* (a *boolean*, in many programming languages) is a valid PLE. If we have a propositional variable p and assigned it the value *True* for instance, it will evaluate to *True*.
- An expression such as (p and q) is a valid PLE. For instance, if we assign p the value *True* and q the value *True*, (p and q) should evaluate to *True* because both p and q are true.
- An expression built from smaller PLEs, such as (p and (q or ((not r) and q))), would evaluate to either *True* or *False*, depending on the values assigned to variables p, q, and r. (See below!)

For this project, we'll use four propositional operations to build PLEs from variables: *not*, *and*, *or*, and *implies* (sometimes also called "if-then"). It is common to use *truth tables* to describe what propositional operations do on their input arguments—i.e., the *semantics* of these operations. For this project, those standard descriptions and truth tables are presented as part of a Supplementary Document about Truth Tables available from our course's *Project Assignments* website.

Here's an **extremely important note:**

> **It is essential that you understand the material on truth tables as part of this project.** As always, please see me with any questions! I recommend that you **read and understand that document before continuing with the project**.

1.1.1 Recursive Definition of PLE Syntax

The truth tables describe what our four propositional operators *mean*, in terms of what values they evaluate to, but to specify what your algorithm might take as input, we also need to define the full *syntax* of all possible expressions that might be evaluated by your algorithm. Here's a recursive definition of the syntax for valid propositional logic expressions (*PLEs*):

Base case Given an initial set of *propositional variables* (e.g., $p, q, r, \ldots$), all elements of that set are valid PLEs. **Important note:** In this context, we will use any single *lowercase* letter as a propositional variable.

Recursive / Inductive cases Let's say that the uppercase letters P and Q stand for PLEs. That is, P and Q each stand for some syntactic expression (e.g., (not (q or p)), unlike the lowercase letters that are variables *within* PLEs.

Then, the following are also PLEs:

1. (not P)
2. (P and Q)
3. (P or Q)
4. (P implies Q)

Note that the parentheses are part of what's needed for valid syntax. So, for example, (p implies (q and (not (r or p)))) is a syntactically correct PLE. That would be determined by following the recursive definition, with these steps:

1. r and p are both valid *PLEs*, because they are propositional variables, and the base case of our definition says they are *PLEs*.

2. Because r and p are both *PLEs*, recursive case 3 of our definition above says that (r or p) is a *PLE*.

3. Because (r or p) is a *PLE*, recursive case 1 says that (not (r or p)) is a *PLE*.

4. Because q is a *PLE* (because it's a propositional variable, thus a *PLE* by the base case) and (not (r or p)) is a *PLE*, recursive case 2 says that (q and (not (r or p))) is a *PLE*.

5. Because p is a *PLE* (shown above) and (q and (not (r or p))) is a *PLE*, recursive case 4 says that (p implies (q and (not (r or p)))) is a *PLE*.

This is the kind of thing that compilers / interpreters for programming languages do! Please see me if you'd like to go over that chain of reasoning.

1.2 An Algorithm for Evaluating Propositional Logic Expressions

For this part of the project, you will create an algorithm for evaluating PLEs, using the syntax and semantics given in the sections above. Here are the input and output specifications that your algorithm must meet:

> **Input**: A PLE P; and a dictionary-like data structure M that assigns a *truth value*—i.e., either *True* or *False*—to every propositional variable that appears in P. Each variable is assigned exactly one value.
> **Output**: If P is a propositional variable, return the truth value assigned to it in M. Otherwise, return the truth value to which P evaluates, in accord with the propositional operation truth tables and treating every variable appearing in P as evaluating to the value associated with it in M.

As a very small example, if M assigns $[p = \mathit{True}, q = \mathit{True}]$, then variables p and q evaluate to *True*, and PLE (p and q) evaluates to *True*. For more examples of evaluating a PLE in accord with our truth tables and an assignment of truth values to variables, please see the **Some example exercises** section at the end of our Supplementary Document about Truth Tables and semantics, and see Section 2 below in this document. And as always, please feel free to see me with any questions about evaluating PLEs!

Your algorithm must be recursive, following the two "Zen Principles" in lecture. The base case of your algorithm should correspond to the base case of the definition of the syntax of PLEs; your recursive cases should correspond to the recursive cases of the definition of PLEs. If your team would like to employ any other design structure for your recursive algorithm, please discuss it with me very early in your process—it might not earn full credit on this project.

You may assume that you have the following functions to use in creating your algorithm:

- The function *var_Val(v,M)* that returns the truth value assigned to propositional variable v in M.
- The function *is_Var(P)* that takes a single *PLE* P as input and returns *True* if P is a propositional variable, *False* otherwise.
- The functions *is_Not(P)*, *is_And(P)*, *is_Or(P)*, and *is_Implies(P)*, each of which takes a single PLE P as its only input argument and returns *True* if *PLE* P is formed by applying the appropriate operation (*not*, *and*, *or*, *implies*, respectively) to its arguments; these functions return *False* otherwise. **Examples:**
  1. *is_Var((p and (q or r)))* returns *False*, because (p and (q or r)) is not a propositional variable.
  2. *is_and((p and (q or r)))* returns *True* because *PLE* (p and (q or r)) is formed as the result of the *and* operation on two arguments: the first is p, and the second is (q or r).
  3. *is_or((p and (q or r)))* returns *False* because (p and (q or r)) is the result of the *and* operation on two arguments. (The *or* operation in that expression is nested within one of the arguments to the *and* operation, so (p and (q or r)) is not formed by *or* of two arguments.)

As usual, you will give an English description and pseudocode of your algorithm, along with correctness and worst-case complexity arguments.

- For correctness, because this is a recursive algorithm, *be sure to give an inductive argument* that concisely and convincingly explains that your algorithm is correct (including termination on all inputs that meet the input specifications).
- For complexity, because this is a recursive algorithm, *be sure to set up and solve a recurrence* that correctly expresses the worst case complexity of your algorithm. You will have multiple choices about what n stands for in your complexity argument, and you will want to choose one that allows you to set up a recurrence that is relatively easy to solve—you may well not want it to stand for the length of the input as a number of characters, but instead stand for some other quantity that helpfully represents the size of the input! You should then explain, for a full-credit complexity argument, why

the asymptotic complexity you get using that n is the same that you'd get if you used the entire size of the input (**Hint**: Explain that the differences will be covered by the leading constant in the asymptotic definition).

Choosing the quantity to represent by n may require some thought. If you have some options in mind but aren't sure which to choose, feel free to discuss them with me!

HINT: For students who want to code this up in an actual programming language ... Go for it! But you might want to start by creating your own new recursive data type that corresponds to the recursive, syntactic definition of PLEs above. For people who aren't comfortable with recursion, it might be tempting to just treat a PLE as a string, but please do not do so here—this project is focused on building your toolkits for thinking about recursive definitions and algorithms. (Besides which, it might be harder to do this if you treat PLEs as strings. In general, for programming languages, compilers have components that do *lexing* and *parsing* to translate from the original string of text of a program, into the definitions that underlie the programming language; we won't write those components here!)

2 The *Satisfiability* Problem

The *Satisfiability* problem (or *SAT*, for short) is a classic CS problem: Given a propositional logic expression, is there any way to assign *True* or *False* to its variables to satisfy the expression—that is, to make it evaluate to *True*? This problem has a wide range of applications—it can be applied for playing games or solving puzzles such as Sudokus, and it has a role in the theoretical foundations of algorithm complexity, which we'll get to later in the semester—and as part of this project, you'll write an exhaustive search algorithm to solve this problem!

As some useful terminology, for any PLE P, we'll say that a *satisfying assignment* is a way of assigning truth values (*True* or *False*) to every propositional variable in P such that P then evaluates to *True*. If it helps, you can think of it as a dictionary-like data structure like the one called M in the input specifications given in Section 1.2 above, under the further restriction that P **must** evaluate to *True* for that M to be a *satisfying assignment* for P. For examples:

- For PLE $P = (p \text{ and } q)$, the assignment $[p = \mathit{True}, q = \mathit{True}]$ *is* a satisfying assignment. Moreover, no other assignment of truth values to variables would be a satisfying assignment for that P. (Do you see why? Please ask me questions if it isn't clear to you!)

- For PLE $P = (p \text{ implies } (q \text{ and } (\text{not } (r \text{ or } p))))$, the assignment $[p = \mathit{True}, q = \mathit{True}, r = \mathit{False}]$ is not a satisfying assignment for P, because as described in our Supplementary Document about Truth Tables, P evaluates to *False* under that assignment.

- For the same PLE $P = (p \text{ implies } (q \text{ and } (\text{not } (r \text{ or } p))))$, the assignment $[p = \mathit{False}, q = \mathit{True}, r = \mathit{False}]$ *is* a satisfying assignment: $(r \text{ or } p)$ evaluates to *False*, so $(\text{not } (r \text{ or } p))$ evaluates to *True*, so $(q \text{ and } (\text{not } (r \text{ or } p)))$ evaluates to *True*, and finally $(p \text{ implies } (q \text{ and } (\text{not } (r \text{ or } p))))$ evaluates to *True*.

I didn't go through the steps in the individual evaluations of PLEs above—they're the same idea as in our Supplementary Document about Truth Tables, but please see me if there are any questions about them!

Here are the input and output specifications that your algorithm for the *SAT* problem must meet:

> **Input**: A PLE P.
> **Output**: **True** if there exists some way of assigning truth values to all propositional variables in P such that P evaluates to *True* (following the evaluation described by our truth tables); **False** otherwise.

Because you are creating an exhaustive search algorithm, as in your previous project (Project 2—**Loop Invariants, Exhaustive Search, and Beyond!**), please assume once again that you can use algorithms to create the relevant lists of all subsets or all permutations to use in your exhaustive search, with the given time and space complexities:

- **Generate-All-Subsets** You may use a `Generate-All-Subsets(S)` algorithm that has *time complexity* $\Theta(n \cdot 2^n)$ and *space complexity* $\Theta(n \cdot 2^n)$ on input S of size n.
- **Generate-All-Permutations** You may use a `Generate-All-Permutations(L)` algorithm that has *time complexity* $\Theta(n \cdot n!)$ and *space complexity* $\Theta(n \cdot n!)$ on input L of size n.

Please see Project 2 for the input / output specifications of these algorithms.

As usual, you will give an English description and pseudocode of your algorithm for *SAT*, along with correctness and worst-case complexity arguments.

- For correctness: Use the appropriate technique to explain correctness, where needed. If all or part of your exhaustive search algorithm is recursive, give an inductive argument; if all or part of your algorithm is iterative, you do not need to specify and formally use a loop invariant, but your correctness argument should nonetheless be based on what we know to be true each time through the main loop of your algorithm.
- For complexity: Again, use the appropriate technique to explain correctness, where needed. If all or part of your exhaustive search algorithm is recursive, you do not necessarily need to *solve* a recurrence, but you should give a concise and convincing complexity argument that is based on a recurrence that correctly expresses the worst case complexity; if all or part of your algorithm is iterative, give a concise and convincing correctness argument based on its structure, as usual.

Your algorithm and explanation should have enough details to make it clear that your team fully understands all work needed for your solution. That may include writing and explaining algorithms for the following subroutines, which would be part of your *SAT* algorithm:

- A subroutine that returns a list of all propositional variables in a PLE P.
- A subroutine that creates an assignment of truth values to all variables in P such that some specified ones are assigned the value *True*, and the rest are assigned *False*.

These are *not* primitive functions, so if you want to use that functionality, be sure to write algorithms (and input / output specifications) for functions that give you that functionality!

IMPORTANT NOTE: If your team decides *not* to create and use these two subroutines, but instead to try a different approach for a *SAT* algorithm, **please discuss that with me as soon as possible.**

You may, however, assume that you are given functions for use in your algorithm that do standard operations on lists and dictionary-like data structures—e.g., creating and adding pairs to dictionaries, adding elements to lists, appending / extending lists (as in Python). You may also assume that those functions have reasonable time and space complexities, but in your complexity arguments, be clear to say what the complexities are of all functions relevant to the complexity of your algorithm for *SAT*.

HINT: Be sure that you understand what your algorithm is doing an exhaustive search over. The input / output specifications are helpful that way: You are *given* a PLE P, so you're not looking for a PLE P that is true; instead, you're looking for a satisfying assignment for the PLE P that you're given.

3 Improvements

As you did in Project 2, after your team has worked together to arrive at a good exhaustive search algorithm in your work for Part 2 of this project, your next step will be to improve upon that exhaustive search solution.

Please see Project 2, Section 2 for suggestions about how to think of improving algorithm efficiency. Once again, your team is especially encouraged not to restrict yourself to only the first or second of the three ways listed there—special-case improvements can be very helpful, even on very simple-seeming special cases!

Your team should propose **one or two** algorithms or modifications that improve upon your exhaustive search *SAT* algorithm. There are no fixed criteria for this project about exactly how much your improvements must improve upon your initial exhaustive search algorithm; your team will earn more credit on this part of the project for improvements that show more depth of thought about the problem and its solutions, achieve greater efficiency, have greater impact for important special cases (e.g., make it very efficient when used to solve Sudokus), apply more broadly to possible inputs, and are more thoroughly and helpfully analyzed and described. As with Project 2, I hope these criteria make intuitive sense to you, and as always, please feel free to ask me questions!

Hint: You are advised **not** to try to come up with polynomial-time solutions for the general case of *SAT*.

4 Presentation

For Project 2, you were part of a team that gave a presentation 15–20 minutes in length. For this project, you will instead give a presentation that is **10–12 minutes** in length. As many of us who frequently give presentations believe, making a 10-minute presentation is different from making a 20-minute presentation, and it's important to be able to do both!

Your team will give a technical presentation about the algorithms you've created for this project. As you did for Project 2, create slides (in PowerPoint, Google Slides, or some

other application of your choice) for your presentation; hints and suggestions for technical presentations from Project 2 also apply here. With only 10–12 minutes available for this presentation, however, you may need to be selective about how many slides you create and how much material they contain. For an effective presentation, your audience should have time to read and process every slide presented to them, so spending too little time on a slide may well result in an ineffective presentation. For full credit, your team must make an effective presentation lasting between 10–12 minutes; anything outside that range may not receive full credit. If you think your talk will be longer or shorter than that, please discuss that with me—I will be happy to help you find a good balance for your presentation.

You should assume that your audience is at the level of CS students who are familiar with recursive design, the general form of inductive arguments, and recurrences, but who are not yet experts with them. For example, assume that your audience knows what an inductive argument is and how it relates to recursion, and what a recurrence is an how it relates to time complexity of recursive algorithms, but would need to be walked through details involving setting up and solving a particular recurrence, or how an inductive argument applies to a particular algorithm. (To the extent that you are explaining iterative algorithms, please make the same assumptions about your audience's familiarity with those concepts as you did for Project 2; see the Project 2 assignment for details, if needed.) You should also assume your audience has no previous knowledge of your algorithms or any problems involved, and they may not quickly grasp any subtleties.

Your presentation should describe at least the following:

- Our definition of PLEs.
- Your algorithm for evaluating PLEs and how it works. Consider including at least part of a short example, with enough context and content to fully illuminate how your algorithm works for your audience. (See note below about including examples.)
- Your algorithm for *SAT* and how it works, including anything you think is particularly interesting about your approach to the problem. Consider including at least part of a short example, with enough context and content to fully illuminate how your algorithm works for your audience. (See note below about including examples.)
- At least one proposed improvement from Section 3 above. Include a high-level description of the improvement, and a comparison that shows how much this improves upon the exhaustive search algorithm. Consider including enough of a short example to illuminate how it differs from your original exhaustive search algorithm. (See note below about including examples.)
- Complexity and correctness results for your algorithms. There may not be time in your presentation for full correctness and complexity arguments for every algorithm, but to ensure that your presentation is authoritative for your audience, include **at least** the following:

  1. One correctness argument showing command of inductive arguments.
  2. One complexity argument showing command of recurrences and solving them.
  3. A correctness and complexity *result* (not necessarily a full argument) accompanied by a very brief high-level explanation (if not a longer explanation) for each algorithm presented.

Don't worry about space complexity for your presentation unless you feel it is important to understand interesting elements of your algorithms.

In a 10-minute presentation, there will likely not be time to go into deep details on anything; all explanations will need to be high-level, focusing on the essential details for enabling your audience to understand your work. (Your write-up document should contain additional details, enabling your team to show full command of your work and describe important components.)

NOTE about including examples in this presentation: It is expected that your presentation will have one or more illustrative examples to enable your audience to understand your algorithms. You may need to be judicious in choosing your examples due to time constraints, however, perhaps using one example to illustrate multiple components of your work. In 10–12 minutes, there may not be time to include all the examples you might like!

As with the previous Project, this is a team presentation and a class assignment, so ideally, the entire team would learn about all parts of the topic being presented, and not only would the workload be balanced among team members, but it would also *appear* balanced to viewers. For that reason, your presentation should consist of each person presenting for roughly 2–3 minutes at a time, followed by a different teammate—so, for example, in a 10–12 minute presentation, each person on a four-person team would take one shift of presenting for roughly 2–3 minutes each. This structure might require a conceptual topic to be split among multiple individuals in the presentation, due to the impositions of time limits, but that's part of the pedagogical benefit of this—it encourages more people to engage with more different parts of the topic being presented. **Important note:** *Presentations not following this structure will not receive full credit* for this assignment. If there are questions about what's expected in terms of the division among teammates of time spent presenting, please let me know!

The default expectation is that you will record your presentation as a screen recording in Zoom. If you believe another option would be better for your team, **please see me about it as soon as possible!**

Your Accompanying Write-Up In addition to the presentation itself, your team will create an accompanying write-up document, which should enable your audience to fully understand the work done for this project if they do not see your talk. This document must be typed (submitted in PDF) and contain all important details, especially those that you did not have time to include in your talk itself. It should contain at least the minimum needed for a presentation of algorithms:

- Pseudocode and English descriptions of every algorithm, including separate pseudocode / English descriptions for each improvement. For each improvement, include 1–2 sentences about how your team came up with the ideas behind that proposed improvement.

- Concise and convincing high-level correctness arguments and complexity analyses. Please use inductive arguments and recurrences when analyzing recursive algorithms. For your team's proposed improvements, you may not need to give a separate correctness argument for each improvement, but if an improvement significantly affects the inductive arguments previously used to show correctness of the exhaustive search algorithm, you do need to show that the improvement also solves the problem correctly, which could involve a modified inductive argument.

As mentioned in Section 2 above, to explain complexity for recursive algorithms for this project, you may not necessarily need to *solve* a recurrence, but you should give a concise and convincing complexity argument that is based on a recurrence that correctly expresses the worst case complexity. For explaining complexity of iterative components of algorithms, give a concise and convincing explanation based on its structure, as usual.

Overall, please include everything needed to concisely demonstrate your command of relevant analytical techniques and other concepts, including recursive design, inductive arguments, and recurrences. Including relevant examples can also be helpful in communicating your work to your audience; your write-up may allow you to include components in examples that time constraints prohibited you from including in your presentation.

As would be expected, your grade will depend on the clarity, readability, and completeness of your write-up, enabling readers to understand your work and demonstrating your command of key concepts. Depending on the margins / font size / etc. of your document, your write-up should probably be 7–10 pages in length. Please keep it as concise as it can be while still containing all relevant information. If your write-up is running longer or shorter than that range, please see me to check whether it contains unneeded material, or too little material; write-ups that are much too long or too short are not maximally effective and may not receive full credit.

Overall, for both the presentation and the write-up, more credit will (as expected) be given to submissions that demonstrate greater scope of work completed, greater depth of insight in the work completed, and more correct and effective presentations of the work. Although no dress rehearsal is required for this project, you are welcome to consult with me about a draft of your presentation or write-up—I will be happy to give you feedback about the level of detail in the draft.

5 Submission Instructions

Deadline: 11:59pm, Nov. 19 A "designated submitter" from each team should submit **three** items by emailing them to me (eaaron@colby.edu):

- A PDF file with all of the slides used for the presentation. Please put two slides per page (as is done for CS375 course lecture notes) and name the file `CS375_Proj3_Slides_Team_<INITIALS>.pdf`.
- The write-up document that accompanies your presentation, which should be a PDF file called `CS375_Proj3_WriteUp_Team_<INITIALS>.pdf`.
- A video file (or link to it) of your presentation. *Please put it in your Google Drive space* if it's too large to simply include in an email. Please name the file `CS375_Proj3_Presentation_Team_<INITIALS>.mp4`

  As usual, `<INITIALS>` is to e replaced by the initials of the team members in the group in the team assignments. E.g., if Eric Aaron and Stephanie Taylor were the teammates, a file from that team would be called `CS375_Proj3_Slides_Team_EA_ST.pdf`.

  Note the preferred `mp4` format. If for any reason you cannot submit an `mp4` video, please let me know as soon as possible!

Lateness policy: To keep pace with the project assignments in CS375, it is important that this assignment be turned in promptly—and it is more important than usual, with this project being due near the end of the semester! For this project, there will be a deduction of 1.5% for each day late—i.e., 1.5% deduction for submitting up to 24 hours late; 3.0% deduction for submitting more than 24 hours late, up to 48 hours; etc—up to a 10% deduction for submitting up to 7 days (168 hours) late. After 7 days, late submissions will receive a 40% deduction. Please submit your work promptly!

As always, extenuating circumstances will be considered—please contact me as soon as possible if any extenuating circumstances are impeding your work on this project!