Analysis of Algorithms
CS 375, Fall 2022
Project 2
Due in part by **Monday, Oct. 24**, and
in part by **Thursday, Nov. 3**.
(Please see notes and descriptions below!)

Project 2: Algorithm Design and Explanation—Loop Invariants, Exhaustive Search, and Beyond!

In this assignment, you'll work in teams of four to design multiple algorithms, to use loop invariants to understand and explain algorithms, and to create and deliver a presentation about some of the algorithms you designed. The goals of this project are:

- to give you practice designing algorithms—starting here with exhaustive search algorithms—and improving the efficiency of algorithms;
- to give you practice using loop invariants to explain algorithm correctness;
- 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

This project is a multi-part assignment, with different deadlines for different parts: the deadline for Part 1 is **Oct. 24**, and the deadline for Parts 2–4 is **Nov. 3** (see Section 5 below). As an overview, here are the parts of the project assignment, as presented in class on Oct. 17:

1. **Design Exhaustive Search Algorithms** Your team will collectively design exhaustive search algorithms for a variety of problems, with specifications given below.
2. **Improve Time Efficiency** Your team will pick one of the problems and make your exhaustive search algorithm more efficient.
3. **Reduction** For the same problem chosen for part 2 above—yes, make sure it's the same one—you will *reduce* that problem to one of the other problems given in part 1. (This is a new topic for us—more about it soon!)
4. **Create and Give a Presentation** Your team will present work from all three other parts of the assignment about the problem you chose for parts 2 and 3 above, **using loop invariants** where appropriate to explain the correctness of your algorithms.

There are things to do for each part of the assignment, as described individually below.

1 Design Exhaustive Search Algorithms

For this project, your team will collectively solve eight problems with exhaustive search algorithms. Then, you will submit revised solutions for four of those problems: one on which you will focus heavily, including doing a presentation; and three others, for which your submission will be a simpler write-up. Instructions are below, but as always, please be in touch with any questions!

1.1 Independent Problem Solving

Your team has eight problems to solve, grouped into three Categories of related problems, described in Section 6 below. With four people in your team, each of you should come up with exhaustive search algorithms to solve **two of the eight problems**, under these constraints:

- Each person in your team should solve two problems, but they cannot be from the same Category.
- Your team of four people should collectively solve all eight problems—each person solving two of the eight.
- Once you've met as a team to divide the eight problems among the four of you, **please work fully individually on these exhaustive search algorithms, not discussing them with anyone but your TAs and Prof., and not using any additional resources other than your CLRS textbook.** There is, of course, a reason for this—more about this below!

Each individual on the team will individually submit their two algorithms to their Submitted Work folder. This will be graded as *a Smaller Assignment for the individual submitter*—not part of the project grade—and graded based on effort. Full credit will be given for strong, demonstrated effort regardless of whether or not the solutions are correct!

1.2 Team Problem Solving

Once each team member has submitted their two algorithms, your entire team should meet to discuss the algorithms. Part of the point of this is to give and receive constructive feedback to / from teammates about algorithm design, as you're reading over each other's work *having not seen it before*—if you have worked together on the initial submissions, it will diminish the value of this part of the project.

Then, for your team's final project submission, your team will work together to revise and write up algorithms for *three* of the original eight problems, keeping in mind the following:

- Your three problems must **not** include the problem on which you're doing your presentation.
- Your three problems must include one from each Category in Section 6 below.
- Each of your three algorithms should be presented as pseudocode, with a short English description of how the algorithm works, and a concise, high-level analysis of its time and space complexity. *You do not need to give a detailed correctness argument or use loop invariants for these three algorithms.*

For other parts of this project, your team will also work on an algorithm for a fourth of the eight problems. As described in Sections 2–4, your team will improve on the time efficiency of that algorithm, you will come up with a solution for it using a *reduction*, and you will give a presentation about it accompanied by an informative write-up that includes complexity analyses and the use of loop invariants for explaining correctness.

To Submit for this part of the project

- By **11:59pm on Monday, October 24**, each individual team member should submit a document containing exhaustive search algorithms for their two of the eight problems. Standard file naming conventions apply: Please submit your typewritten answers in a PDF file named `CS375_Proj2Stage0_<userid>.pdf` where `<userid>` is replaced by your full Colby userid, and submit it to your `SubmittedWork` folder.

- By **11:59pm on Thursday, November 3**, the entire team should submit a document containing their polished write-ups of algorithms for three of the eight problems, as described above. Please submit these typewritten answers in a PDF file named `CS375_Proj2_ThreeAlgos_Team_<INITIALS>.pdf`. Additional instructions for submitting this `ThreeAlgos` document, along with the remainder of your work for the project, are given in Section 5 below.

2 Improve Time Efficiency

As mentioned in Section 1, your team will choose one of the eight problems on which to give a presentation and focus for the other parts of this project. In the remainder of this project assignment, I'll use the variable name `P` to refer to the problem on which your team chooses to focus (just so that there's a name for it!).

After your team has worked together to arrive at a good exhaustive search solution for problem `P`, in your work for Part 1 of the project, your next step will be to improve upon the time efficiency of that exhaustive search solution.

In general—not just for this project, but in general for algorithm design—there are a few ways to think of improving upon an algorithm's time efficiency. They include:

1. You can make it somewhat faster with small(-ish) changes that streamline but do not substantially redesign the algorithm or change its asymptotic complexity class.

2. You can make it **much** faster with a substantial redesign—perhaps even giving a new algorithm altogether—which might even improve its asymptotic complexity class.

3. You can focus on *special cases* of the input that can be solved very efficiently. That is, instead of coming up with a more efficient solution that works for *all* possible inputs in the problem's specification, come up with a solution that is much more efficient—i.e., a better asymptotic complexity class—for *some* of the possible inputs.

   For example, imagine that you've solved a problem with this input specification:

   > **Input**: k, a positive integer

But you then come up with a much faster algorithm that will work only when k is an even number. That faster algorithm does not meet the original problem specifications, but it is an improvement in the special case of an even integer input.

NOTE: Smaller input sizes are rarely considered special cases in this sense. If a suggested "improvement" is "It's the same algorithm, but it's super-fast on small inputs!", that likely is not actually a useful improvement. Please see me if your team has a suggested improvement that requires looking at only very small inputs, to confirm that it's worthy of including in your project write-up / presentation!

For this project, your team will come up with improvements upon the efficiency of your original exhaustive search algorithm for problem `P`. You are encouraged to think of improvements in terms of the three ways listed above, and you are especially encouraged not to restrict yourself to only the first or second of them—special-case improvements can be very helpful, even on very simple-seeming special cases!

There are no fixed criteria for this project about exactly how much you must speed up your initial exhaustive search algorithm—or, with the "special cases" approach, how broadly applicable your improvements might be, to cover as many cases as possible. Your team can also propose **up to three** algorithms or modifications that improve the efficiency of your original algorithm for problem `P`, although submitting three improvements is not *necessarily* better than one. Ultimately, 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 time efficiency, apply more broadly to possible inputs, and are more thoroughly and helpfully analyzed and described—but in some cases, one substantial improvement might achieve that better that three small ones. (I hope these criteria make intuitive sense to you. 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 cases of any of the eight problems presented. You might, however, create polynomial time solutions for special cases, which you could choose to include in your write-up / presentation!

To Submit for this part of the project In your presentation for this project, your team should present not only your original exhaustive search algorithm for problem `P` but also your improvements to it. In the write-up that accompanies your presentation, please include the following:

- A full description of your exhaustive search algorithm for problem `P`, including a short English description of it, pseudocode for it, a concise and convincing correctness argument for it **using loop invariants** to establish correctness, and a concise, high-level complexity analysis for it.

- English descriptions of each improvement. Each description should include 1–2 sentences about how you came up with the ideas behind that proposed improvement.

- Pseudocode showing what each improvement does.

- A concise, high-level complexity analysis showing how much each improvement actually improved the time efficiency of the original exhaustive search algorithm. Although you do not need to use formal definitions of asymptotic complexity in your analysis, you might want to use some part of them—in particular, if your improvement doesn't change the asymptotic complexity class, you might describe its improvement in terms of a lower *leading constant* for complexity analysis.

You do not necessarily need to give a separate correctness argument for your improvements, **although if they affect the loop invariant(s) your team previously used** to show correctness of the exhaustive search algorithm, you do need to show that the improvements also solve the problem correctly, which could involve a modified loop invariant.

More details about your presentation and its write-up are in Section 4 below.

3 Reduction

Sometimes, we can incorporate solutions to previously solved problems as subroutines in an algorithm we're designing. For this project, your team will do that in a specific way: You'll *reduce* problem `P` to another problem.

Informally, in general, reducing problem A to problem B means creating an algorithm that, if you plugged in a subroutine that solved problem B, would immediately be able to solve problem A—the algorithm reduces the task of solving problem A to the task of solving problem B. We'll call such an algorithm a *reduction* from A to B.

As a concrete, very simple example—which we also went over in our Oct. 19 class meeting—consider problem A with these specifications:

Input: List $L = [c_1, c_2, \ldots c_n]$ of numbers.
Output: *True* if the first element of L, c_1, is 375; *False* otherwise.

And consider problem B with these specifications:

Input: List $M = [d_1, d_2, \ldots d_k]$ of numbers.
Output: *True* if the last element of M, d_k, is 375; *False* otherwise.

For these problems, a reduction from A to B would take some input L—remember, this reduction is an algorithm for some problem A, so it has to take an input intended for A—and create a new list $M = [c_1]$ that contains only the first element of L. Then, the reduction would use M as input to a subroutine that solved problem B. If that subroutine returned *True*, that would mean $c_1 = 375$ (**do you see why?**), which in turn means that the first element of L is 375, so the reduction solving A should return *True*. On the other hand, if that subroutine for B returned `False` on input of list M, that would mean c_1 is not 375, so the reduction solving A should return *False*.

There are other possible reductions that could have worked—for instance, M could instead have been created as $M = [c_n, \ldots c_1]$, built from all elements of L in reverse order, rather than just a list with one element; the rest of the reduction would have been exactly the same. For this project, your team just needs to present one correct reduction. (Please note again that this is a very simple example—your work for this project may not be quite this simple!)

For this project, your team will create a reduction *from* problem `P` *to* any other of the eight problems given in the project. Here is the specific way to think of it for this assignment:

- An all-powerful creature has bestowed upon you a wonderful gift of magic! They've given you seven magic subroutines—one for each problem in Section 6 other than `P`—that will solve each problem in O(1) time! To use one of these magic subroutines, just give it some input that matches the input specifications for the problem, and then it will instantly give you a correct True or False answer for that input to that problem!

- There's a catch, though: You only get to use *one* of these magic subroutines—any one you choose, but only that one—and you only get to use it *once*. After that, all seven subroutines disappear!

- Your task is to write a new algorithm to solve problem `P` that makes use of the magic subroutine of your choice. It shouldn't be an exhaustive search algorithm anymore; the magic subroutine can do the hard work of exhaustive search! In fact, if you are careful in choosing and employing the magic subroutine, you could even come up with a *polynomial time* algorithm for problem `P`!

  (Recall from lecture that a polynomial time algorithm is one that is $O(n^k)$ for some constant k. This is much faster than any exponential- or factorial-time algorithm—even a large polynomial like n^{100} has a rate of growth much slower than 2^n.)

Just to give a name to the problem you choose to reduce `P` to, let's use variable `Q` to refer to that problem—i.e., you'll using the magic subroutine that solves problem `Q` as part of your reduction from `P` to `Q`. Your reduction should thus transform any possible input p for problem `P` into an input q for problem `Q`, such that when you get a True or False answer about input q for `Q`, you can use that to come up with a correct True or False answer about input p for problem `P`.

For this part of the project, in addition to creating the reduction as described above, you will also do all of the following:

- Analyze the time complexity of your reduction, under the assumption that the solution for `Q` comes in $O(1)$ time. **For maximal credit, your reduction should be in polynomial time**—$O(n^k)$ for some k—but don't worry about what constant k you use. Every correct polynomial time reduction will be equally good for this project!

- Include the reduction in your presentation (see Section 4 below). Be sure to include problems `P` and `Q` you're reducing from and to, a short description of your reduction algorithm (pseudocode is not required, though you may include it if you think it helps your presentation!), a short explanation of correctness, and a short complexity analysis.

- Describe the reduction in the write-up document accompanying the presentation. Again, state the problems `P` and `Q` you're reducing from and to, and give a helpfully complete description of the reduction—an English description is required; pseudocode is optional—along with a short explanation of correctness and your complexity analysis of the reduction.

  Your explanation of correctness does not need to use loop invariants—your reduction will probably be straightforward enough that loop invariants aren't required. If your team thinks loop invariants might be a good idea to use, though, feel free to do so, or feel free to ask me about it!

  Important note: A correct reduction must be **exactly consistent** with the Input / Output specifications for both `P` and `Q`. For full credit, your explanation of correctness should explicitly refer to those specifications.

Your presentation and accompanying write-up document will contain all the work you need to submit for this part of the project. Please make sure your reduction is clearly and concisely described in the presentation itself, and all helpful details for understanding the reduction are included in the write-up!

Hint: See Section 6.2 for a hint that might (or might not, depending on your approach!) be useful for reductions involving problems in the Social Networks Category.

4 Create and Give a Presentation

At this point, your team has done a lot of work on problem `P`. Let's hear about it!

Your team will give a technical presentation about the algorithms you've created for problem `P`. For your presentation, create slides (in PowerPoint, Google Slides, or some other application of your choice) and take 15–20 minutes to present all of the material needed. Presentations that are too long or too short may not receive full credit (too short often indicates that some important material was not well presented; too long often indicates that additional preparation would have resulted in a more effective talk), so it is recommended that you target a 16–18 minute presentation. If you think your talk will be much longer or shorter than that, please discuss that with me—I will be happy to help you find a good balance for your presentation.

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!**

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 minutes at a time, followed by a different teammate—so, for example, in a roughly 16 minute presentation, each person on a four-person team would take two non-consecutive shifts of presenting for roughly 2 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!

Here are some things you should include in your presentation (not necessarily in this order!):

Your exhaustive search algorithm For your exhaustive search algorithm, please include:

- An accessible description of the problem `P` you solved.
- A high-level summary of your algorithm and how it works.
- A short example that you step through, to give your audience a sense of what problem `P` is and how your algorithm works. It may be appropriate to only step through a part of an example instead of an entire one, but you should do enough to fully illuminate how your algorithm works for your audience.

- Pseudocode of the algorithm, along with a correctness argument *using a loop invariant.*
- A complexity argument of the algorithm, including what the worst-case and best-case complexities are, and how much space is needed beyond the original input.

Your improvements to your exhaustive search algorithm For each of the improvements you're presenting, please include:

- A high-level description of the improved algorithm.
- A short example that you step through, to give your audience a sense how the improvement differs from the original exhaustive search algorithm. Once again, you may not need to go through a full example, but you should do enough to illuminate the differences in your improved algorithm.
- Pseudocode of the improved algorithm, along with a correctness argument. As noted above, you need not use loop invariants for this (though you could if you thought it was necessary), but you do need to give a concise and convincing correctness argument. You can refer to your original exhaustive search algorithm and its correctness without re-explaining them.
- A complexity argument of the algorithm, including what the worst-case and best-case complexities are, and how much space is needed beyond the original input.
- A comparison of the complexity of your improved algorithm with that of your exhaustive search algorithm.

Your reduction algorithm For your reduction, please include:

- An accessible description of the problem `Q` you're reducing to.
- A high-level description of the reduction algorithm that solves `P`, including how it uses the subroutine for `Q` in that solution.
- A short example, to show your audience what the reduction does—transforming input to `P` into input to `Q`, and using output from the subroutine for `Q` to get a correct answer on the input to `P`.
- Pseudocode of the reduction, along with a correctness argument. As noted in Section 3 above, this will involve referring to the specifications of `P` and `Q`. You need not use loop invariants for this; just give a concise and convincing correctness argument that the reduction meets specifications and solves `P` correctly (assuming the subroutine solves `Q` correctly). You can refer to your original exhaustive search algorithm and its correctness without re-explaining them.
- Worst-case time complexity and space complexity arguments for the reduction, assuming the subroutine for `Q` has $O(1)$ time and space complexity. (Magic!)

You should assume that your audience is at the level of CS students who are familiar with asymptotic complexity and loop invariants but are not yet experts with them. For example, assume that your audience knows a set of size n has 2^n subsets, a list of length n has $n!$ permutations, and all about the relative growth rates of functions used in asymptotic complexity (including knowing what "polynomial time" means), but would need to be walked through details involving leading constants and n_0 thresholds in definitions of asymptotic complexity.

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.

To help prepare for your presentation, please look through the documents linked from CS375's Project Assignments page:

- Some general *advice on how to give good technical presentations*—Dale Skrien shared this with his classes, and I am passing it along to you!
- A tutorial on screen recording with Zoom, from Colby Academic ITS.
- Advice on setting up a good environment for a web conference using Zoom from Colby Academic ITS. (I'm not sure how useful this will be, but I'm including it just in case.)

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 understand the highlights of your presentation even if they do not see your talk. This document must be typed (submitted in PDF) and contain some important details that you may not have time to include in your talk itself. (For example, some small but important details of complexity arguments might not fit in the 15–20 minutes of your talk, but they can be included in the write-up.) For full credit, your write-up must be polished, well formatted for a professional technical presentation, easy to read, and free of grammatical errors.

Please see individual sections above for more information about details to include in your presentation write-up about the exhaustive search, improvements, and reduction algorithms.

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.

Dress Rehearsal As part of this project, please schedule a dress rehearsal with me. This should be a live, in-person presentation, rather than on Zoom—the intent is to be as effective as possible in giving feedback on the organization and content of your talk, rather than on using Zoom technology. (As in our class meetings, masks will be required for our dress rehearsal meeting. If that will be problematic for any of your teammates, please let me know!) Plan on 30–40 minutes for the dress rehearsal. Come to the dress rehearsal already having practiced your talk, with your slides ready and your write-up ready for me to look at while you're presenting—the rehearsal is a *dress* rehearsal, not a *draft* rehearsal.

So that the dress rehearsal time can be used as effectively as possible, you are strongly encouraged to record a draft rehearsal / practice run of your talk beforehand and do a self-evaluation of how it went, identifying areas of strength and room for improvement. Time permitting, I will be happy to give feedback on that recording during our appointment time!

Please note there will be significant deductions to your grade if your eventual project submission includes a poor presentation—including things like poor organization, poor clarity of speaking, or poor knowledge of the material—so please, use your draft rehearsal(s) and our dress rehearsal time wisely to polish your work.

I expect to schedule all dress rehearsal appointments for the afternoons of **Friday, Oct. 28** and **Saturday, Oct. 29**. Please email me to set up an appointment, and please be as flexible as possible with your availability for scheduling—those will be very busy days!

Some suggestions for getting audiences engaged in a presentation

Note from your Prof.: Dale Skrien gave these suggestions to his students for his presentation assignments. I'm not sure that they all fully apply to this presentation, but in the interest of giving you good advice about technical presentations in general, I'm passing them along to you.

- Get the audience to care about the subject. For example, get the presentation started by asking a question whose answer the audience cares about.
- Keep examples simple and focused. Don't make the audience think about irrelevant things.
- Use conversational tones in presentations. Use "I", "me", and "we" so that the listener's brain thinks it's in a conversation.
- Garr Reynolds, the author of *Presentation Zen*, says, "the principles and techniques for creating a presentation for a conference or a keynote address have more in common with the principles and techniques behind the creation of a good documentary film or a good comic book than the creation of a conventional static business document with bullet points."
- Something to think about regarding your presentation (also from Garr): "If the audience could remember only one thing (and you'll be lucky if they do), what do you want it to be?"

Please feel free to ask me questions about them, if you'd like!

5 Submission Instructions

Deadline: 11:59pm, Oct. 24 For the individual work in Section 1, as described in that section, every individual should submit typewritten answers in a PDF file named `CS375_Proj2Stage0_<userid>.pdf` where `<userid>` is replaced by your full Colby userid, and submit it to your `SubmittedWork` folder.

Deadline: 11:59pm, Nov. 3 For all of the group work in this project, a "designated submitter" from each team should submit **four** items, one to their Google Drive `SubmittedWork` folder, and three by emailing them to me. The file to submit to the `SubmittedWork` folder of the designated submitter:

- A document containing their polished write-ups of algorithms for the three problems solved for Section 1, as described above. Please submit these typewritten answers in a PDF file named `CS375_Proj2_ThreeAlgos_Team_<INITIALS>.pdf`, where `<INITIALS>` is 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, the file from that team would be called `CS375_Proj2_ThreeAlgos_Team_EA_ST.pdf`.

The items to email 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_Proj2_Slides_Team_<INITIALS>.pdf`.

- The write-up document that accompanies your presentation, which should be a PDF file called `CS375_Proj2_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_Proj2_Presentation_Team_<INITIALS>.mp4`

  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. 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!

6 The Eight Problems to be Solved

Below are the eight problems to be solved, divided into three Categories. Note that each of the problems is a *decision problem*—it asks for a True / False answer to be given.

For each of the problems below, in all three Categories, your exhaustive search algorithm will need to look through either all subsets of a set or all permutations of a list. For your work, please assume that you can use algorithms to create the relevant lists for your exhaustive search, meeting the specifications given here (the same as those in our lecture notes) and with the time and space complexities given here:

- **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, meeting these specifications:

  Input: $S = \{s_0, s_1, s_2, \ldots, s_{n-1}\}$, a set of n elements
  Output: L, a list of all subsets of S

- **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, meeting these specifications:

  Input: $L = [s_0, s_1, s_2, \ldots, s_{n-1}]$, a list of n elements
  Output: PSL, a list of all permutations of L

Note that you are not told how these algorithms work, and they may not be identical to the ones we derived in class—you should just assume that they exist for your use and meet these specifications.

In addition, here are some reminders about *sets* and *graphs* that might be useful:

- The specifications for some of the problems below involve sets. Please recall that by the definition of a set, no two values in a set can be equal to each other. All of the specifications were written to be consistent with this definition.

- The specifications for some of the problems below involve graphs. Every graph G is defined as a combination of a set V of vertices in the graph and a set E of edges that connect some (or all, or none) of the vertices in the graph; for short, we say $G = (V, E)$. See CLRS, Appendix B.4 (pg. 1168) for more about graphs—and, as always, please feel free to ask me any questions about definitions regarding graphs!

The following sections give the three Categories containing the eight problems to be solved.

6.1 Category: Taking Stuff

The problems to be solved in this Category are:

1. **Fair Share** You and a friend are in a room with n valuable items—with values $c_1, c_2 \ldots c_n$—and you want to take all of them! But only if you each take *exactly* the same value with you. Is that possible with the items in front of you?

The `Fair-Share` problem:

> **Input**: A set $C = \{c_1 \dots c_n\}$ of n positive integer values (the values of the n items).
> **Output**: *True* if there exists a subset S of C for which the sum of the values in S is *exactly* the sum of the values not in S; *False* otherwise.

For example: If $C = \{3, 6, 9, 12\}$, a correct algorithm for `Fair-Share` would return *True*, because there's a set $S = \{6, 9\}$ where the sum of the values is 15, and the sum of values not in S is $3 + 12 = 15$. On the other hand, if we consider set $C = \{1, 5, 10, 25, 50, 100\}$, there is no set S of values from C that could equal the values not in S (try it—no subset of C works for these values!), so a correct algorithm would return *False*.

2. **Price is *Exactly* Right** You're in a store with n items, with costs $c_1, c_2 \dots c_n$, and you have an amount V to spend on these items. Can you spend *exactly* V on some (or all) of the items from this store?

The `Price-Exactly-Right` problem:

> **Input**: A set $C = \{c_1 \dots c_n\}$ of n positive integer values (the values of the n items); and a positive integer V.
> **Output**: *True* if there exists a subset S of C for which the sum of the values in S is *exactly* equal to V; *False* otherwise.

For example: If $C = \{1, 2, 3, 9\}$ and $V = 12$, a correct algorithm should return *True*, because there exists subset $C = \{1, 2, 9\}$ for which the values add up to 12. (That's not the only subset with values that add to 12, but one is enough for the algorithm to return *True*.) On the other hand, for the same $C = \{1, 2, 3, 9\}$, if $V = 8$, there is no subset S of C for which the values of S could add up to *exactly* 8.

3. **Book Bag** You're at a used book sale where there's a deal available: If you pay a flat fee—let's call the fee amount K—they give you a bag with a capacity of C and you can take as many books as you want, as long as they all fit in that bag. (The numbers K and C can be in whatever units you like—as long as we're consistent throughout the problem, it doesn't matter which they are.) You can choose books to take from a set $B = \{b_1 \dots b_n\}$ of n books, and each book b_i has a size $s(b_i)$ and a value $v(b_i)$. Is it possible to find some subset S of the books such that all the books in S could fit in the bag they give you, and the total value of the books in S add up to more than the amount K that you'd pay for the deal?

The `Book-Bag` problem:

> **Input**: Set $B = b_1 \dots b_n$ so that each b_i has a positive integer size s_i and a positive integer value v_i; positive integer capacity C; positive integer fee K.
> **Output**: *True* if there exists a subset $S = \{a_1 \dots a_m\}$ of B for which the sum of the sizes $\Sigma_{i=1}^{m} s(a_i)$ is less than or equal to C *and* the sum of the values $\Sigma_{i=1}^{m} v(a_i)$ is greater than or equal to K; *False* otherwise.

For example: Let $B = \{b_1, b_2, b_3\}$ where b_1 has size 1 and value 5, b_2 has size 2 and value 12, and b_3 has size 3 and value 8. Then, if $C = 5$ and $K = 18$, a correct

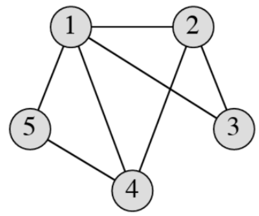

Figure 1: An example graph, referred to in examples accompanying problem statements. (Figure taken from CLRS.)

algorithm should return *True* on inputs B, C, K, because the set of books $\{b_2, b_3\}$ has their total size equal to 5, less than or equal to C, and their total value equal to 20, greater than or equal to K. For the same B and K, however, but $C = 4$, a correct algorithm should return *False*; one way to see this is that the greatest value possible for a set of books from B with total size less than or equal to 4 would be 17—from the subset $\{b_1, b_2\}$—and that's not greater than or equal to K.

6.2 Category: Social Networks

In all of the problems in this Category, we'll be using graphs to represent social networks! Every vertex in a graph will represent a person, and every edge between two people will represent that the people know each other. Please assume graphs for problems in this Category are undirected.

The problems to be solved in this Category are:

1. **Clique** A *clique* is defined to be a collection $C = \{c_1 \ldots c_j\}$ of people such that every pair of people in C know each other. Because we're using a graph $G = (V, E)$ to represent the social network, a clique is a subset C of the vertices of the graph for which every pair of vertices in C has an edge between them. (C could, in principle, be equal to V, which would be a *complete* graph.) **The question**: Given a number K, is there a clique of size K in the social network we're studying?

   The `Clique` problem:

   > **Input**: Graph $G = (V, E)$, positive integer $K \leq |V|$.
   > **Output**: *True* if there is a clique C of size K in G; *False* otherwise.

   For example: In Figure 1, the graph contains multiple cliques of size 3, such as the set $\{1, 2, 4\}$. The set $\{1, 2, 3, 4\}$ is not a clique, because 3 is not connected to 4.

2. **Strangers** In the social network $G = (V, E)$, we will define a group of people $S = \{s_1 \ldots s_j\}$ to be *strangers to each other* when for every pair of people in S, they do not know each other. Because we're using graph $G = (V, E)$ to represent the social network, a *set of strangers* is a subset S of the vertices of the graph for which no edge in E exists between any two people in S. (S could, in principle, be equal to V, which would be a maximally *sparse* graph.) **The question**: Given a number K, is there a set of strangers S of size K in the social network we're studying?

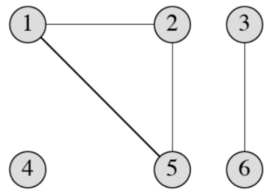

Figure 2: An example graph, referred to in examples regarding the *complement* of a graph. (Figure taken from CLRS.)

The `Strangers` problem:

> **Input**: Graph $G = (V, E)$, positive integer $K \leq |V|$.
> **Output**: *True* if there is a set of strangers S where S has size K in G; *False* otherwise.

For example: In Figure 1, the set $\{2, 5\}$ is a set of strangers of size 2 (there are also others), but there is no set of strangers of size 3—for every three people in the graph, there's a connection between some two of them. (Try it!)

3. **Network Cover** In the social network $G = (V, E)$, we will define a group of people $P = \{p_1 \dots p_j\}$ to be a *network cover* if, across all the people in P, every social connection in the network involves at least one of the people in P. Because we're using graph G to represent the social network, a network cover is a subset P of the vertices in G such that every edge in E involves at least one person in P. **The question**: Given a number K, is there a network cover P of size K in the social network we're studying?

   The `Network-Cover` problem:

   > **Input**: Graph $G = (V, E)$, positive integer $K \leq |V|$.
   > **Output**: *True* if there is a network cover P where P has size K in G; *False* otherwise.

   For example: In Figure 1, the set $\{1, 2, 4\}$ is a network cover of size 3 (there are also others), but the set $\{1, 4\}$ is not a network cover, because the connection between 2 and 3 is not covered by $\{1, 4\}$.

Hint: This hint only applies to *reductions* (Section 3), not other parts of the Project. When thinking about designing reductions involving the problems in the Social Networks Category (Section 6.2), you might want to consider the *complement* of a graph as part of your reduction. By definition, for a graph $G = (V, E)$, the *complement* of G is a graph $G' = (V, E')$, where the vertices are the same as in G but the edges are *all edges not in E*. More precisely, considering every edge in a graph as a pair of vertices, $E' = \{(u, v) \mid u, v \in V, \text{ but } (u, v) \notin E\}$. As a concrete example, let G be the graph in Figure 2; then, there would be edges from vertex 4 to every other vertex in the complement G', because none of those edges are in G, but edge $(1, 2)$ would not be in G' because there is an edge between vertices 1 and 2 in G.

You don't need to use the complement of a graph in your reduction, but you might want to in some cases. As always, please feel free to talk with me about these concepts!

6.3 Category: Maps and Touring

In the problems in this Category, we'll be using graphs to represent maps! Every vertex in a graph will represent a location, and every edge between two locations will represent that we can travel between those locations in either direction. Please assume graphs for problems in this Category are undirected.

For these problems, we'll define a *tour* on a map $G = (V, E)$: A tour is a path that starts in some initial city c_1 in V and then, following edges in E, passes through every other city in V exactly once before returning to c_1. For example, in the map represented by the graph in Figure 3, one possible tour is represented by the shaded edges: the one from u to w to v to x and then back to starting city u. Note that there are many possible tours through all cities on this map, this is just one possibility.

Also, for the problems in this section, note that in a tour, it doesn't really matter which city we indicate as the starting city—since the tour goes through all of them exactly once before looping back to where it started, its starting point could equivalently be anywhere for these problems. The direction also doesn't matter for the problems in this section, because the graphs are undirected. For example, the tour indicated by the shaded edges in Figure 3 could be viewed as starting at city x and going x to v to w to u to x just as well as u to w to v to x to u.

The problems to be solved in this Category are:

1. **Traveling Salesman** In this classic CS problem, we start with a *complete* graph $G = (V, E)$, in which every pair of cities v_i, v_j in V is connected by an edge in E. In addition to the graph, there is a distance function d that gives a distance $d(v_i, v_j)$ between every pair of cities; assume that the distance is the same in either direction, so $d(v_i, v_j) = d(v_j, v_i)$ for every pair of cities.

   It is our traveling salesperson's job to make a tour—to start from their home city, then follow edges in the graph to visit every other city *exactly once* before returning to their home city. **The question**: Given a number K, is it possible to make a tour while covering total distance K or less?

   The `Traveling-Salesman` problem:

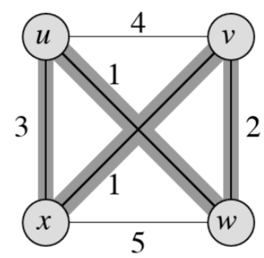

Figure 3: An example *tour* on a graph, referred to in examples accompanying problem statements. Here, there are many possible tours through all cities on this map, including one from u to w to v to x and then back to starting city u (shown by shaded edges in the graph). (Figure taken from CLRS.)

> **Input**: Graph $G = (V, E)$, distance function d as described above, and positive integer K.
> **Output**: *True* if there is a tour of V having distance K or less; *False* otherwise.

For example: In Figure 3, the distance function d giving distances between cities is indicated by the number over each edge—e.g., $d(v, x) = 1$ and $d(x, w) = 5$. Consider the map in that Figure, that distance function d, and number $K = 5$. A correct algorithm would return *False*, because there is no tour of all those cities (remember, it has to end up back where it started!) with distance 5 or less. On the other hand, with that map, that distance function, and $K = 9$, a correct algorithm would return *True*—indeed, the shaded edges are a tour of distance 7, which is less than 9.

2. **Hamiltonian Tour** Unlike the Traveling Salesman example, the touring company of *Hamilton* does *not* start with a complete graph—instead, they start with some map $G = (V, E)$ that may or may not have edges between any two cities. There is also no distance function to be considered here; all that matters is whether or not they can get from one city to the next.

   Their goal, however, is to make a tour, in the same technical sense of *tour* used in the Traveling Salesman problem—to visit every city exactly once before returning home. **The question**: Given a map $G = (V, E)$, is there a tour in the map, visiting each city exactly once before returning to the city from which it started?

   The `Hamiltonian-Tour` problem:

   > **Input**: Graph $G = (V, E)$.
   > **Output**: *True* if there is a tour in G; *False* otherwise.

   For example: In Figure 1, there are many paths that are not tours in this sense. For example, if a possible tour started at city 5, and then went to 1, and then 2 and then 3, it could not get to city 4 without going back to 1 along the way, and that's not a tour in our sense of the word—it would visit city 1 more than once before returning to where it started. Nonetheless, there is a tour through that graph—for example, it could go from 1 to 3 to 2 to 4 to 5 before going back to 1—so a correct algorithm would return *True* on that graph.