#1-1: Introductions & Algorithms

CS SCHOLARS – PROGRAMMING

Learning Objectives

Understand the **resources** and **expectations** associated with the course

Define the essential components of computer science, **algorithms** and **abstraction**

Construct **plain-language algorithms** to solve basic tasks

Course Goals

The goal of this course is to develop a solid understanding of **computer programming**.

This involves both understanding **program syntax** and understanding the **soft skills of programming**.

By the end of this course, you should be able to:

- Understand and use the core components of programming
- Build small interactive programs that react to user input

Programming and the World

Programs have a huge influence on the world around us! You can take programming as a tool and apply it to basically any domain.

Some program applications are obvious:

- applications on your computer
- smartphone apps
- social media
- video games

Others are more subtle:

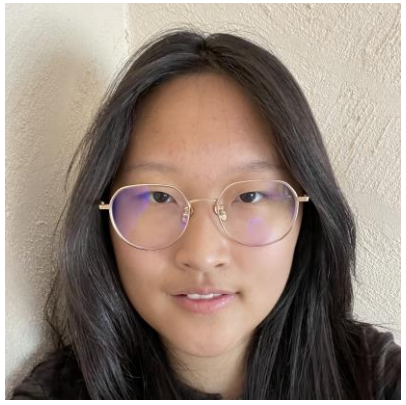
- analysis of sports performance to help players improve
- rendering of realistic animation for movies, games, and more
- reducing waste by redirecting supplies where they're needed
- simulation of climate change to study where changes can be made

Course Intro

Activity: Introductions

Instructor: Prof. Kelly Rivers

TAs: Emily Getty and Dian Zhu



Activity: introduce yourself!

Primary Resources

Course Website: http://www.krivers.net/css-m23/

- Schedule, syllabus, course materials

Gradescope: https://www.gradescope.com/courses/544050

- Assignment submission, feedback

Slack:

- Logistics, fast questions, requests for one-on-one meetings

Course Plan

Core Track:

- Essential elements of programming from basics to interactive programs
- A new topic every day!
- One homework assignment a week

Advanced Track:

- Advanced concepts in programming and computer science
- Two new topics a week (both released Monday)
- Optional bonus problems on main assignment

Core Track

Week 1: Building Blocks – Algorithms, programming basics, and functions

Week 2: Control Flow – Conditionals, testing, loops, and algorithmic thinking

Week 3: Interaction – Strings, lists, and user interaction

Week 4: Scaling Up – Top-down design, libraries, and a class-chosen topic

Advanced Track

Week 1: External Libraries & Data Representation

Week 2: Recursion & Concurrency

Week 3: Data Structures & Efficiency Analysis

Self-taught from slides, but you're welcome to ask questions during work period or on slack.

Make sure you've mastered the core track and finished the core homework before tackling these!

Homework Logistics

Download starter files from course website
- Written portion as fillable PDF, programming portion as .py file

Work on assignments during work periods (usually afternoon sessions) or in the evening
- Collaboration is encouraged, but make sure to write up your own answers – don't copy!

Split into four parts: Review, Core, Spicy, and Bonus
- Review – ensure you're solid on the fundamentals, get extra practice with the ideas
- Core – primary assessment of the material
- Spicy – a little extra challenge, usually in the form of harder problem-solving
- Bonus – practice the advanced material

To fully complete an assignment, complete either Review + Core or Core + Spicy

Homework Submission

Upload your .pdf and .py files to Gradescope on the relevant assignments.

.py files are (mostly) autograded; wait on the page for a few seconds to see your results.
.pdf files are manually graded sometime in the following week.

You can resubmit as many times as you like.

Regular deadline: usually Friday end-of-day.

Revision deadline: usually following Wednesday end-of-day.

We'll demo how to submit on Friday and how to view feedback next Monday.

Homework Grading

We're going to use **mastery grading**. Each problem will be graded as:

- **Mastered** – you fully understand the concept. Scored as 1/1.
- **Sufficient** – you understand the concept well enough to move forward, though you may still make occasional mistakes. Scored as 0.5/1.
- **Insufficient** – you need to keep working on the concept. Scored as 0/1.

You should aim to master or show sufficient knowledge on every problem you attempt.

End-of-Program Evaluation

At the end of the program, you'll receive a personal evaluation.

You will be evaluated primarily based on **participation** (engagement and interaction during lecture periods) and **homework performance** (successful completion of the homework assignments).

You'll also be evaluated based on your performance on a **final evaluation**. This will take place on the last day of class and will cover the core material.

Activity: Set Up the IDE

Thonny: https://thonny.org/ (or another IDE of your choice)

Activity: Go ahead and download it now!

Note: if you're using a Chromebook, you probably won't be able to download applications. Create an account on https://replit.com/ instead.

- repl.it is free, but your code will be publicly viewable. To keep your code private, set up a GitHub student account with your andrewID or high school email (https://education.github.com/pack#offers), then connect it to your repl.it account.

Algorithms

What is Computer Science?

Computer science is the study of **computation**, and **computational devices**. This can be studied through many different lenses, including:

- Computational **theory** – what are the possibilities and limitations of computation?
- Computational **application** – how can we use computation to fulfill a specific need?
- Computational **discovery** – given data, can we find patterns and answer questions through computation?
- Computational **expression** – how can computation change the way we communicate and engage with others?
- **Critical** computing – how does computation affect our lives, and how should it be regulated?

What do we mean by 'computation'? We can reduce this to two core themes: **algorithms** and **abstraction**.

Read more: https://cacm.acm.org/blogs/blog-cacm/249787-what-liberal-arts-and-sciences-students-need-to-know-about-computing/fulltext

Algorithms and Abstraction

Algorithms are procedures that specify how to do a needed task or solve a problem. They are used to standardize processes and communicate them between different people.

Algorithms can be incredibly powerful, but they're still designed by humans, which means they're vulnerable to human flaws.

Algorithms are like recipes, tax codes, and sewing patterns. When you give someone directions to a location, you're communicating an algorithm.

Abstraction is a technique used to make complex systems manageable by changing the amount of detail used to represent or interact with the system.

This can be done by identifying the most important features of a system and generalizing away unessential features.

Abstraction shows up in many interactions – for example, you can pay for groceries through many modalities (cash, debit, credit, an app), and each is **implemented** slightly differently, but all are just different representations of money.

Activity: Make a PB & J Algorithm

You do: work with a group to write a list of instructions (an **algorithm**) on how to make a peanut butter and jelly sandwich.

Before you begin, consider what level of **abstraction** to use. Assume the user knows the ingredients and how to do basic actions but has no cooking experience.

We'll test your instructions afterwards...

We assume that the user can identify the ingredients and tools, and knows basic actions, but does not know complex actions.

An Algorithm with Moderate Abstraction

1. Before starting: make sure you have a bag of bread, a jar of peanut butter, a jar of jelly, a plate, and a knife
2. Open bag of bread
3. Reach hand in and take out 2 slices of bread
4. Place each slice on a plate
5. Open jar of peanut butter
6. Pick up knife and stick sharp side of knife into open jar
7. Use knife to scoop out peanut butter
8. Wipe and spread peanut butter on one slice of bread
9. Repeat 5, 6, 7 until slice of bread is covered in peanut butter. Then close jar
10. Open jar of jelly
11. Pick up knife and stick sharp side of knife into open jar
12. Use knife to scoop out jelly
13. Wipe and spread jelly on non-PB slice of bread
14. Repeat 10, 11, 12 until the slice of bread is covered in jelly. Then close jar.
15. Put the peanut butter side of one slice of bread on the jelly side of the other.
16. Result: you now have a peanut butter and jelly sandwich on a plate

An Algorithm with Heavy Abstraction

1. Before starting: make sure you have bread, peanut butter, and jelly
2. Get two slices of bread
3. Spread peanut butter on one slice
4. Spread jelly on the other slice
5. Combine slices into a sandwich
6. Result: you now have a peanut butter and jelly sandwich

If we've already taught someone the basics of sandwich-making, teaching them to make a PB & J sandwich is a lot simpler!

Note that we don't define *how* to spread the peanut butter or jelly. Maybe the user will have a different approach to ours.

If someone doesn't even know the basic assumptions (a toddler, or a robot), we'll need to define every item used and how to execute even the simplest steps. And we're still making assumptions here!

An Algorithm with Low Abstraction

1. Before starting: make sure you have [specific quantity and type of bread in plastic bag with tab], hand, plate...
2. Define bread as a grain-based substance that has been divided into 1 inch wide parts (slices). Bread is in a plastic container (bag)
3. Open bread bag by gently pulling a plastic tab away from the plastic wrap.
4. Define hand as the appendage at the end of your arm. Define fingers as the smaller appendages at the end of your hand
5. Define plate as a hard, flat, usually-circular surface
6. Move hand into the opening in the bread bad. Move fingers to close position around the top bread slice
7. Lift hand until it is outside of bread bag.
8. Move hand over the plate, then down so that it is touching plate. Open fingers around the bread slice.
9. Repeat steps 5-7 so that a second bread slice is on the plate.
10. ...

Designing Good Algorithms

Designing algorithms at the right level of abstraction is a large part of computer science. When we represent an algorithm as **program code**, we communicate with a computer to tell it how to do a specific task.

What are the core parts of an algorithm?

- It should specify what is needed at the beginning (**input**)
- It should specify what is produced at the end (**output**)
- It should specify how to get from the beginning to the end (**steps**)

Algorithms can be Measured Many Ways

It's not always good enough to make an algorithm that works. There are many ways to solve any given problem, and these different approaches can be compared to determine which is 'best'.

Here are some metrics we might use to assess an algorithm:

- It should produce the right output for all given inputs (**correctness**)
- It shouldn't take too long to finish (**efficiency**)
- Others should be able to understand and modify it as needed (**clarity**)
- It shouldn't be broken by unexpected behavior (**robustness**)
- And more!

[if time] Activity: Ideate on Course Goals

You do: what are your goals for this course? What do you want to be able to do four weeks from now?

- Take three minutes to reflect on your goals. Feel free to write or type up your thoughts.
- Then we'll move into small groups where you can discuss your ideas.
- In the last five minutes of class, fill out your thoughts here: http://bit.ly/css23-goals

There are no right or wrong answers – I just want to know what all of you are interested in.

Learning Objectives

Understand the **resources** and **expectations** associated with the course

Define the essential components of computer science, **algorithms** and **abstraction**

Construct **plain-language algorithms** to solve basic tasks