

Lecture 27

# Software Engineering I

CS61B, Spring 2024 @ UC Berkeley

Industrial Applications of Software Engineering

# Background

---

Lecture 27, CS61B, Spring 2024

## Background

Complexity

Strategic vs Tactical Programming

Resource Configurations and Testing

Commit History Traversal

Summary

I'm the guest lecturer for today, Aniruth Narayanan.

- Senior, majoring in EECS and Business
- Teaching Assistant for 61B
  - First taught 61B in Summer 2021
- From Florida
- Took a gap year to work in industry
  - The subject of this lecture!
  - Later, I'll talk about some of the problems I've worked on

Q & A will be at the very end.



Some important disclaimers:

- Nothing I say should be interpreted as anyone's opinion but my own.
- Nothing contained in these slides is confidential or proprietary information of any company.
- The sections on complexity and strategic/tactical programming are inspired by [Josh Hug's slides](#) (think of these as less detailed versions).

In [61A](#), you were focused on the correctness of a program.

In 61B, you are focused on engineering programs: picking from multiple options by considering tradeoffs.

- You now work on larger scale projects, but still have to abide by many of our requirements and specifications (e.g. functions, runtime, etc.).
- Working on small projects isn't the same as working on large scale, design-oriented programs where the task is defined *by* you.

The best way to learn these differences is through experience; this lecture will have some examples and some light theory as a sampling.

Project 3 allows you to embrace the challenge of large scale yourself.

# Complexity

---

Lecture 27, CS61B, Spring 2024

Background

## **Complexity**

Strategic vs Tactical Programming

Resource Configurations and Testing

Commit History Traversal

Summary

In other disciplines, we're limited by imperfect materials.

Many fields are constrained:

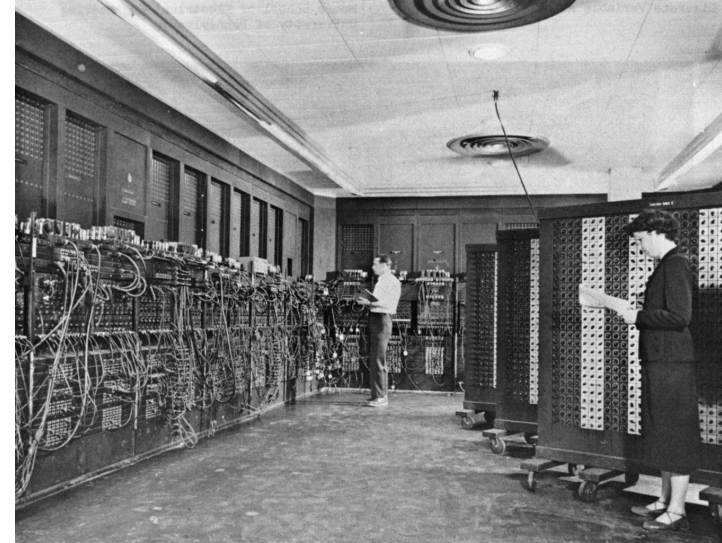
- Chemical engineers have to worry about temperature.
- Material scientists have to worry about how brittle a material is.
- Civil engineers have to worry about the strength of concrete.

In computer science, we've solved most of the underlying material constraints decades ago.

- The sum power of Apollo missions is less than the computing power of your phone.

Early computers were used for limited tasks.

ENIAC: the first general-purpose programmable computer that was 1,000 times faster than anything else at the time.





Rollercoaster Tycoon was an early game (1999) where players would design amusement parks.

99% of the code was written in Assembly (direct machine instructions), with 1% written in C.

This is very hard for programmers to reason about.



The game was designed to run on Intel Pentium CPUs, which were 66 MHz (66 million operations per second).

The Intel i9 14th Gen of today can run at [up to 6GHz](#) (6 billion operations per second). *100x more*

Most video games today require at least 3.5 GHz and 8 GB of RAM.

In 1996, the average computer had only 8 MB or 16 MB of RAM.

The average software product doesn't require the computing power we have available today.

The limitation comes from the creative ways we plan and design what we're building.

1. An individual programmer cannot effectively manage a large software system.
2. Any one programmer should only need to understand a fraction of the codebase.

MINECRAFT		
Minimum		
Optimum		
OS	Windows 10 or later, macOS 10.15 or later, Linux 64-bit	Windows 10 or later, macOS 10.15 or later, Linux 64-bit
Architecture	ARM, x64, x86	ARM, x64, x86
Memory	2 GB	4 GB
Motion Controller	Not specified	Not specified
Headset	Not specified	Not specified
Processor	Intel Core i3-3210 3.2 GHz   AMD A8-7600 APU 3.1 GHz   Apple M1 or equivalent	Intel Core i5-4690 3.5 GHz   AMD A10-7800 APU 3.5 GHz   Apple M1 or equivalent
Graphics	Intel HD Graphics 4000   AMD Radeon R5	NVIDIA GeForce 700 series or AMD Radeon Rx 200 series (excluding integrated chipsets) with OpenGL 4.45

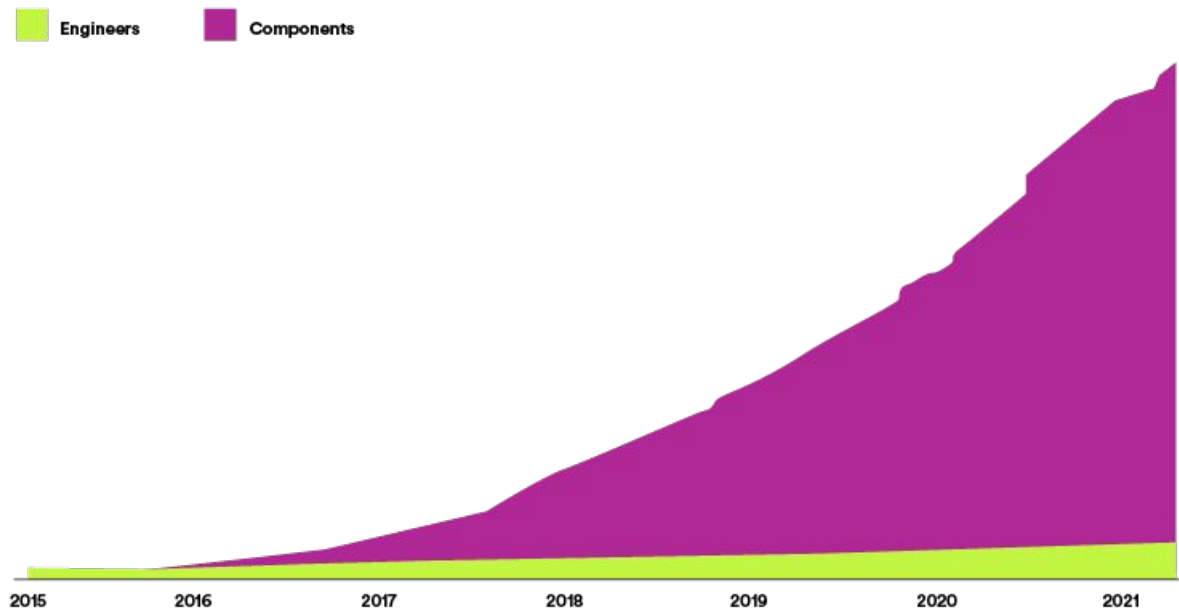
*“Anything related to the structure of a software system that makes it hard to understand and modify it”* - John Ousterhout, “A Philosophy of Software Design”

As programs become more feature-rich, their complexity increases. Our goal is to keep software simple.

Why? Complex systems require a lot of effort to make small improvements.

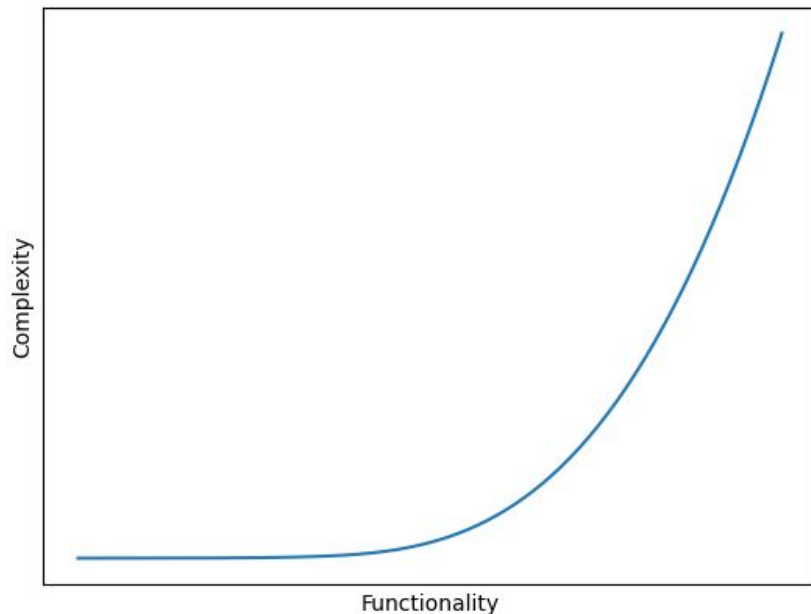
- It takes longer to understand how code works
- It is more difficult to fix bugs
- It is harder to modify functionality
  - Unknown unknowns: it's not clear what you need to know to make modifications
  - Common in large codebases

Here's an example from [Spotify's Engineering Blog](#) of their approach to software. They have >1 billion lines of code, 60 million used in production, from thousands of components.



Complexity scales exponentially with respect to functionality.

Each new piece of functionality has to interact with all the existing functionality in various ways, for all possible combinations.



There are two kinds of complexity:

1. Unavoidable (Essential) Complexity
  - a. Inherent, inescapable complexity caused by the underlying functionality
2. Avoidable Complexity
  - a. Complexity that we can address with our choices

In response to avoidable complexity, we can:

1. Make code simpler and more obvious
  - a. Using sentinel nodes in Project 1
2. Modules
  - a. Abstraction: the ability to use a piece without understanding how it works based on some specification
  - b. Interfaces - HashMap, BSTMap both are Maps

Tackling code complexity from software scale is common in a wide variety of industries:

- Software Engineering
- Data Science
- Machine Learning
- Human-Computer Interaction
- Analyst
- And more!

All of these roles have to manage functionality and interactions with programs.

You can utilize these concepts in whatever field you go into.

# Strategic vs Tactical Programming

---

Lecture 27, CS61B, Spring 2024

Background

Complexity

**Strategic vs Tactical Programming**

Resource Configurations and Testing

Commit History Traversal

Summary



The focus is on getting something working quickly, leveraging workarounds.

Ex: Code that contains many nested if statements to handle many separate cases that is hard to explain.

Prototypes, proof-of-concepts leverage tactical programming. The goal is to show that something could theoretically work.

However:

- There's minimal time spent on overall design
- Things are complicated with workarounds to get things working
- Refactoring takes time and potentially means restarting
  - Consider Project 2 runtime requirements and planning the constructor
- Often, the prototype ends up deployed in the real world due to a lack of time

A different form of programming that emphasizes long term strategy over quick fixes. The objective is to write code that works *elegantly* - at the cost of planning time.

Code should be:

- Maintainable
- Simple
- Future-proof
  - 61B projects have deadlines; afterwards, you can throw it away

If the strategy is insufficient, go back to the drawing board before continuing work.

This is your design document for Project 2B/2C.

# Resource Configurations and Testing

---

Lecture 27, CS61B, Spring 2024

Background

Complexity

Strategic vs Tactical Programming

**Resource Configurations and Testing**

Commit History Traversal

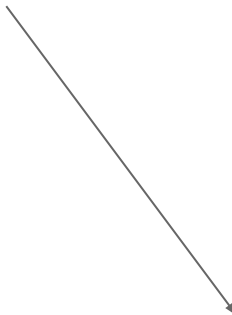
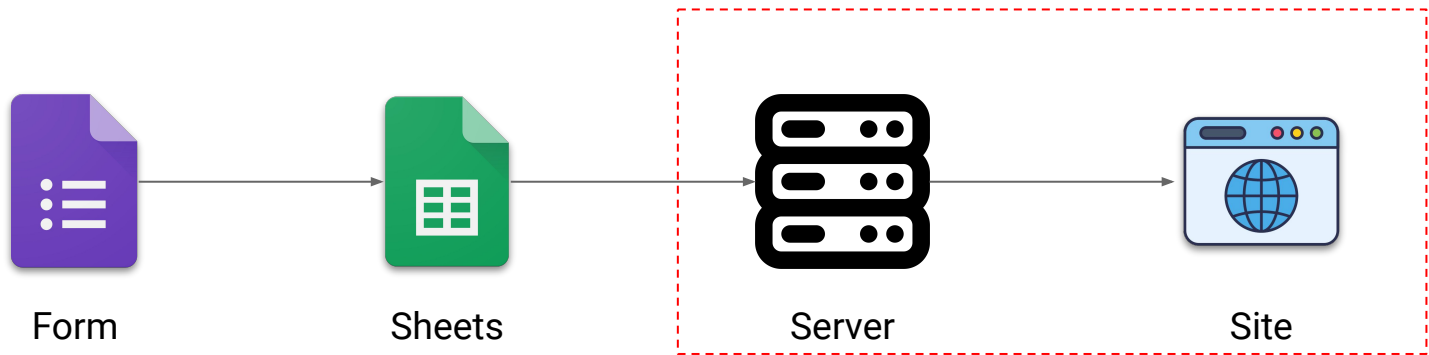
Summary

Retool is a company that helps developers build internal tools. I worked as a software engineer intern on the Connect team in Fall 2022.

For context on my work, here's what the company is useful for:

- Imagine you make surveys with lots of responses (Google Sheets).
- You want to build a dashboard to show common responses. This involves:
  - A website (frontend) to display the results
  - A server (backend) to get data from the resources (Google Sheets)
  - Maintenance and development of both!
- Instead, Retool allows writing queries linked to drag and drop components.
- Much of this power comes from being able to *connect* to nearly any API or database.
  - API: An interface for working with data; we don't care about how it works underneath the hood, just that we can use the specifications.

# Use Case Workflow



NewsNebula Embed

Components Modules

Commonly used

Table

Text

Button

Text input

Number input

Select

Container

Form

Tabbed Container

Modal

Chart

Key value

Image

Navigation

Text inputs

Editable text

Editable text area

Email

JSON editor

Password

Rich text editor

Text area

Text input

Content

Search

Submit new

Name	Status	Submit by	Publish date	Views (k)
[working title] Wants and Needs of an America...	Draft	10/22/2023	TBD	
Riding the World's Tallest Elevator	Draft	10/18/2023	TBD	
The Sweet Science	Draft	10/12/2023	TBD	
Hula Hoop Summer	Draft	10/09/2023	10/22/2023	
Runners of the Deep: A Profile of Scuba Divers...	Published	09/24/2023	10/05/2023	66.14
The 10 Surprising Benefits of Sunlight	Published	09/23/2023	05/05/2023	76.98
You Already Know the Best Way to Dance	Published	09/21/2023	02/05/2023	34.35

Showing 12 of 168

< 1 of 14 >

Share Commit

Staging v1.0.0


All queries completed

1 0 Console Help

Disclaimer: the method to implement resources was decided before I joined, but I needed to both understand and upgrade functionality.

The goal is to design a system for users to specify configuration information that allows Retool to connect to and retrieve information from a resource on the user's behalf. In our previous analogy, this would be a Google Sheets URL.

## Configure Databricks

 [Databricks docs](#)

\* Name

The name for this resource when creating queries in the Retool editor.

\* Folder 

Select a folder to store your resource.

### General

 [Import from connection string](#)

\* Host 

\* Port 

HTTP path 

Default Catalog 

Default Schema 

### Import from connection string

Enter a database connection string and Retool will auto-populate the resource configuration.

[Import](#)

The naive implementation is to build a new page for each resource in its entirety.

Problems with this solution:

- Making a modification (such as wording) would require updating every page
- Breaking changes from updated requirements requires updating every page
  - What is the List interface changed from `size()` to `.length`?
- Adding a new resource is complicated (and requires a lot of copy-paste)



The ultimate solution was to build a ResourceConfig - a way to describe properties of a resource with a few parameters using common patterns to generate inputs.

- Sounds a lot like an interface!
- This leverages *modularity* and *obviousness* when designing a config.

Most databases have similar requirements:

- a host
- a port
- a name
- connection options



For non-database resources without patterns, we can customize elements with properties to accept.

- *Overriding* the default database functionality!



Not all resources were valid.

- If a user did not submit the necessary information for a resource, they wouldn't be allowed to make the resource.
  - Java prevents you from writing knowingly bad code with compiler errors.
- This was specified in a *validation function* included in the resource config.
  - Validation for a resource comes included!
  - Without knowing what a resource is, we can call on the validation function for any resource - guaranteed to exist from the interface.
    - Any List can use the size method, independent of what kind of List.

When users put in their credentials, there's an option to Test Connection with a sample request. If the validation function failed, this was grayed out.

My objective was to add a way to track missing fields and incorporate some validation for fields.

This way, a user can be informed what needs to be changed when making a resource fails.

A strategy question: How can we expand functionality to include this? What do we need to change?

The solution was to rethink validation as passing the missing fields test.

- Parameters would be cycled through, and if one was missing, the accompanying “missing field” is added to the list.
- If the list of missing fields is empty, then the resource can be tested/created!

The fields can be upgraded to integrate and generalize validation.

- For example, every port number for a database should be a number.
- Upgrading the one field component updates *all* usages.
  - Analogy: A superclass modification affects all subclasses that inherit that functionality.

Testing in the real world is very important! There is *no autograder* to leverage to gain confidence in a solution.

Would the right missing fields be displayed?

- For the “patterned” database resources, I wrote a utility that tested every possible combination of inputs and outputs.
  - If there were mock resources, this could be expanded further.
- For the other resources, select test cases were chosen and manually written.

This pattern is powerful - updating the utility updates the tests for *all* resources.

Resources (and libraries/dependencies) update all the time, sometimes with breaking changes. This is future-proof code.

# Commit History Traversal

---

Lecture 27, CS61B, Spring 2024

Background

Complexity

Strategic vs Tactical Programming

Resource Configurations and Testing

**Commit History Traversal**

Summary

## The Problem

I want to find which versions of my project contain a certain change, to compare some metrics like adoption. Some of these versions are *special* - they were deployed into the real world.

Would you want to have 5 or 6 updates to your phone every day? Apple and Google release new versions in a cycle, with minor changes in between.

Gradescope only takes in a number of your submissions in a given time interval, not every one.

They don't release every single time a new commit is pushed internally.



For simplicity:

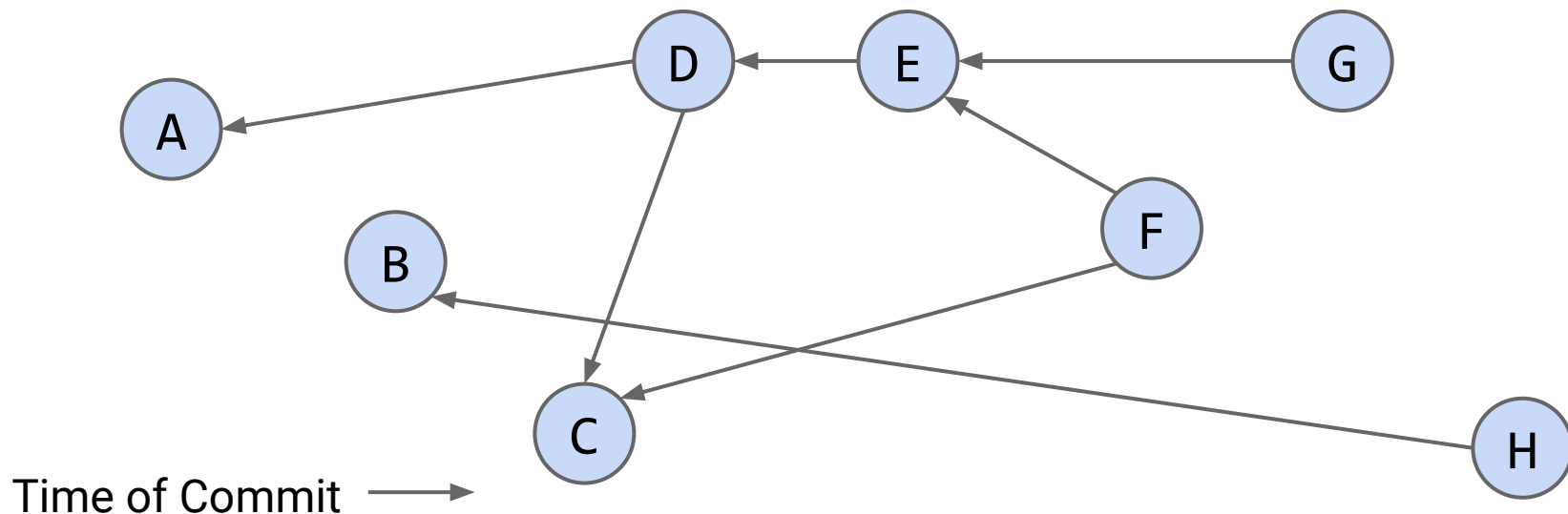
- Each change is a git commit - the same git commits generated when you use `git commit -m "submitting lab1"` and pushed using `git push origin main`.
  - Versions refer to the most recent git commit on a given branch.
- There is easy access to all of the git commits that can be loaded on any computer in a reasonable amount of time.
- There are *a lot* of commits - caching the results (i.e. storing which commits are in every single deployment) is infeasible.
- Once a change is made, we don't undo or modify that change (in practice, we can use `git blame`).

Manually splitting the commits for inclusion by hand is *slow*, when there's a lot (~millions or billions) of them.



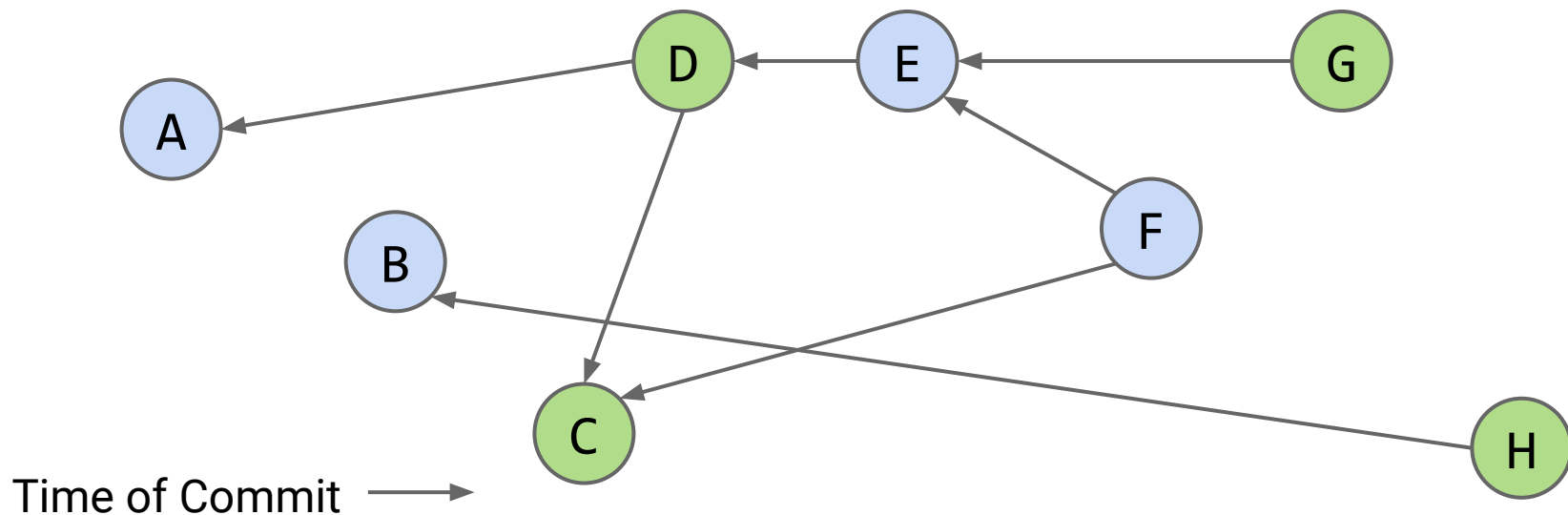
Each commit has a “parent” (potentially two if it is a merge commit).

- Vertices are commits.
- Edges are directed; from commit to its parent.
  - Commits store their parents, since upon creation commits only know parent(s), not children).



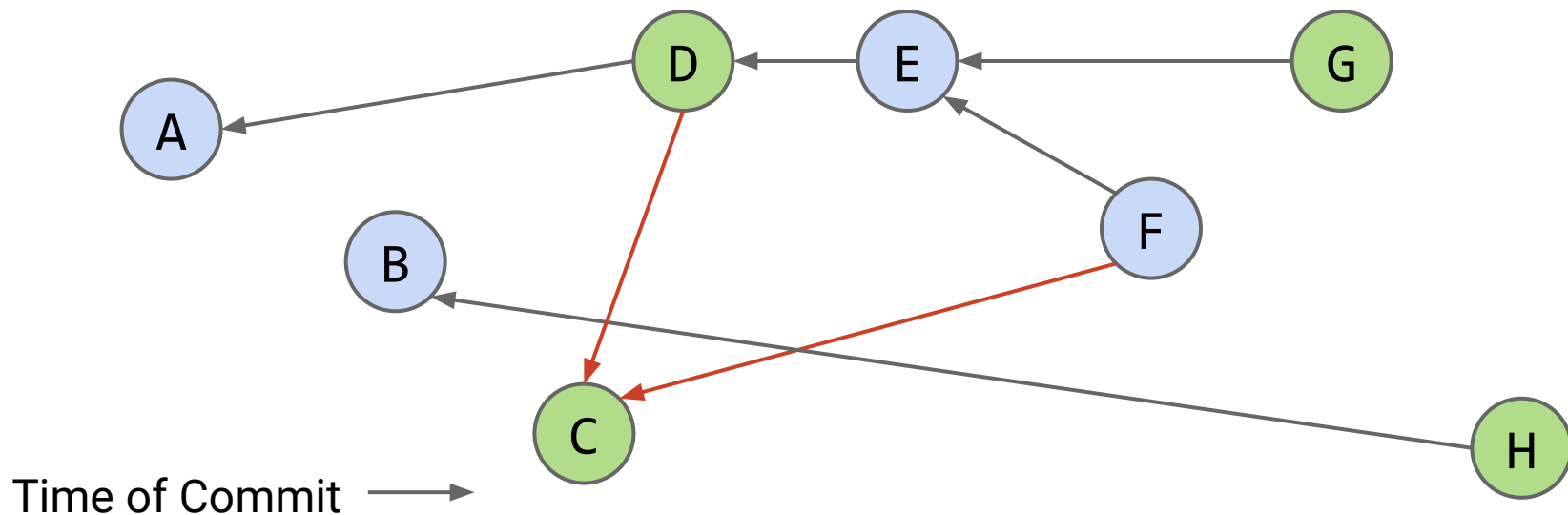
Consider the following graph of commits, where arrows are drawn to a commit's parents. Some of these are the deployed commits, marked in green.

Which deployed commits contain the change in commit C?



Consider the following graph of commits, where arrows are drawn to a commit's parents. Some of these are the deployed commits, marked in green.

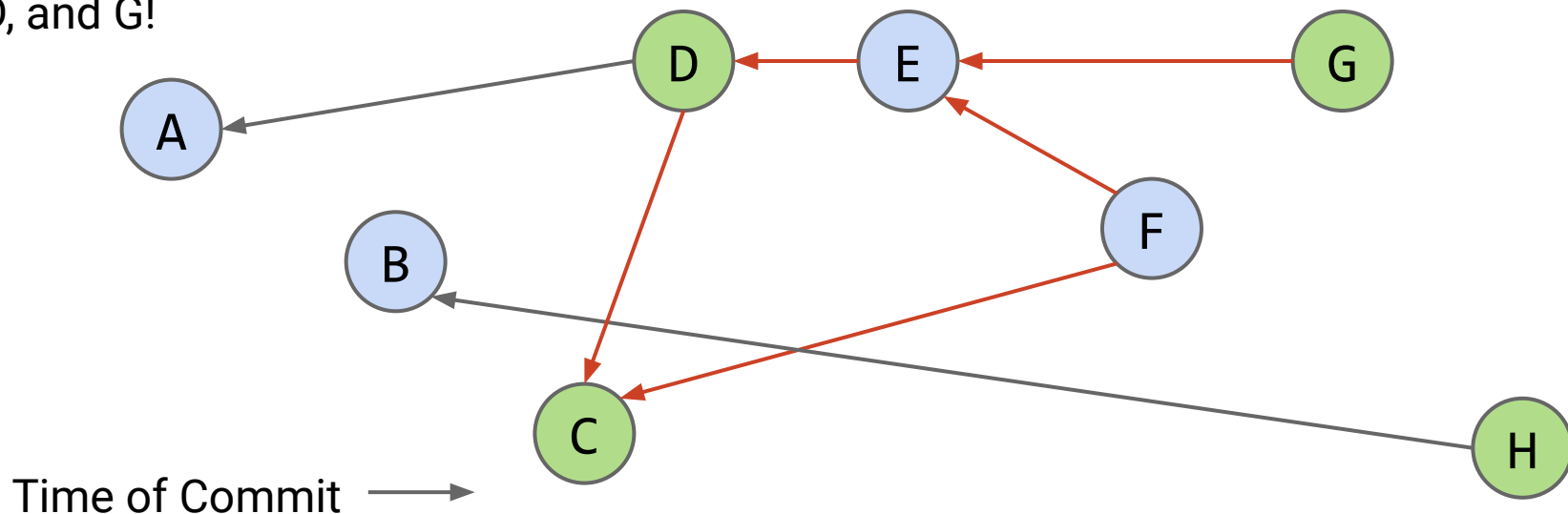
Which deployed commits contain the change in commit C?



Consider the following graph of commits, where arrows are drawn to a commit's parents. Some of these are the deployed commits, marked in green.

Which deployed commits contain the change in commit C?

C, D, and G!



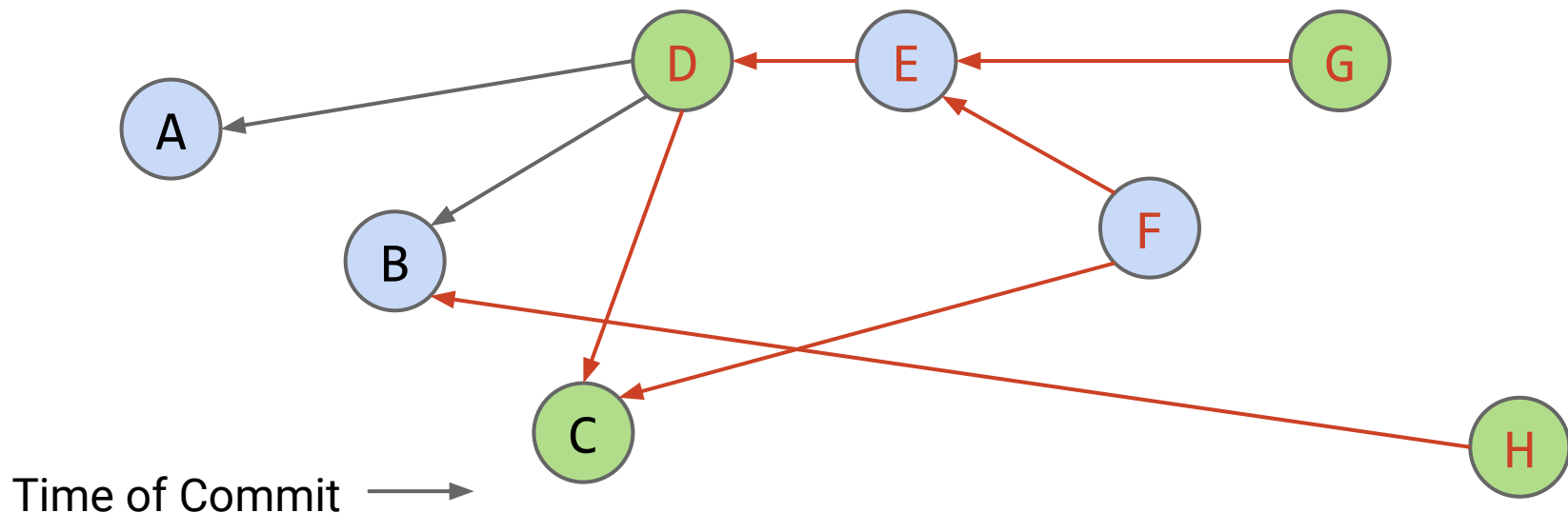
Consider two approaches to do this programmatically:

1. Work backwards in the reverse direction of the commits storing parents
  - a. git is designed to tell us parents of commits (such as through git log)
  - b. This is what we did previously!
2. Work in the direction of the commits storing parents
  - a. We have to grab every single “ending commit” and then work through all the parents until we hit the change we’re checking for in commit C.
  - b. If we don’t hit the change, we can only terminate once we work backwards enough to find commits from all parents that have timestamps *before* commit C.

## Approach 2

This approach requires us to work backwards from all commits after the time of the given commit.

It's also a bit more challenging to implement: we need timestamps and there might be a lot of branches (each with its own ending commit).



The solution comes in the way that we store data!

We normally think about Git in a commit/parent relationship. For example:

```
commit 33f639487be34f7e6f28e1648834f5cf5c1d5475 4d8c496dcd54e254e0442236b0d17bc5df79a48d
Author: Aniruth Narayanan <aniruth.n@berkeley.edu>
Date:   Fri Sep 15 11:10:50 2023 -0700

    projlc my solution
```

What if the direction of the edges was reversed? That way, a BFS from the commit can be done *forward* in time to find deployments that contain it.

This could be implemented using an adjacency list as opposed to storing just one (or two) parent(s) since a commit can only have one parent (two if it's a merge commit), but it could have infinite children.

We can incorporate more time-based optimizations to restrict the search, but here's the general idea:

1. Build a git commit graph in reverse order, storing mappings from commits to a list of children.
2. Generate a set of the deployed commits (separate from the graph).
  - a. Why might I want to use a set instead of a list?
  - b. Runtime! A HashSet gives me amortized constant runtime as opposed to a LinkedList or an ArrayList to check *contains*.
3. Do a BFS from the input commit and check if commits traversed are in the deployed commits.
  - a. Potentially multiple input commits here and compute only step 3.

This is much faster! No more manual search! Obvious *and* modular!

This problem itself is a subproblem of a larger project which is simpler to solve.



# Summary

---

Lecture 27, CS61B, Spring 2024

Background

Complexity

Strategic vs Tactical Programming

Resource Configurations and Testing

Commit History Traversal

## Summary

## Summary and Takeaways

---

- 61B concepts and knowledge is applicable in real world scenarios.
- Good code is more than just working code.
- Code complexity scales with functionality and poses challenges to maintain.
- Practice good design principles in your classes.
- Testing is necessary to have confidence in your code.
- The real world will give you ambiguous problems with loosely defined constraints; it is your duty to make sense of these and then leverage your skills to select the best solutions given constraints while considering tradeoffs.

If this kind of content sounded interesting to you - check out engineering blogs! I also [write about similar topics](#).