# 40.317 Lecture 2

Dr. Jon Freeman

21 May 2020

Slido Event #A590

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

# Agenda

- Version control via Git
- A review of algorithm complexity

# Version Control: Motivation

Why are we studying this topic?

- Keeping code under version control is a requirement in every well-run company in every industry.
- Version control is only the first, rather basic step on the path to:
  - Application Lifecycle Management (ALM)
  - “DevOps” practices

# What Does a VC System Do?

- A VC system manages changes made by multiple users to a set of repositories, or “repos”.
- Each repo is a collection of files, structured in folders and sub-folders.
- The system maintains the history of every change made to every file.
  - Any version of any file can be recovered.

# What Does a VC System Do?

Users can:

- *check in* or *check out* files
- *tag* (or "*label*") a set of related changes to multiple files
- create *branches* to isolate more ambitious changes
- *merge* their changes, either by file or by branch
  - Merging is tricky and hard – think about why!

# Traditional Version Control (1/2)

- The key component is a single, centralised server.
- The complete history of all changes to all repos resides there, and nowhere else.
- Users need this server to be up, and accessible, in order to do anything.
  - It's a "single point of failure"
  - Working offline, or remotely, is not easy
- Users work in a local "shadow" which they keep synced to the server.

# Traditional Version Control (2/2)

- Users of the same repo need plenty of ongoing face to face communication in order to:
  - propose and discuss possible changes
  - approve, reject, or revise existing changes
  - coordinate and avoid surprises
- Users can “check out and lock” files, which helps with coordination but is crude and inefficient.

# An Introduction to Git

Git is a *distributed* version control (VC) system.

- Written by Linus Torvalds in 2005
  - Solely for Linux kernel development at first
- Free and open source
- Now the most popular tool of its kind
- Not to be confused with GitHub, a popular collection of open source projects, managed via Git

# Quick Installation Instructions

1) Download and install Git Tools from https://git-scm.com/downloads

2) Do basic configurations via Git Bash
   - `git config --global user.name "<your> <name>"`
   - `git config --global user.email your_name@your_email.com`
   - (I also specify Notepad++ as my text editor.)

# What is *distributed* version control?

Imagine you are a developer working as part of a distributed team:

- Your colleagues live and work all over the world, in many time zones.
- You have never met any of them in person (!).
- Ongoing, extensive communication is out of the question.

*How can you work together productively?*

# What Makes Git Different (1/2)

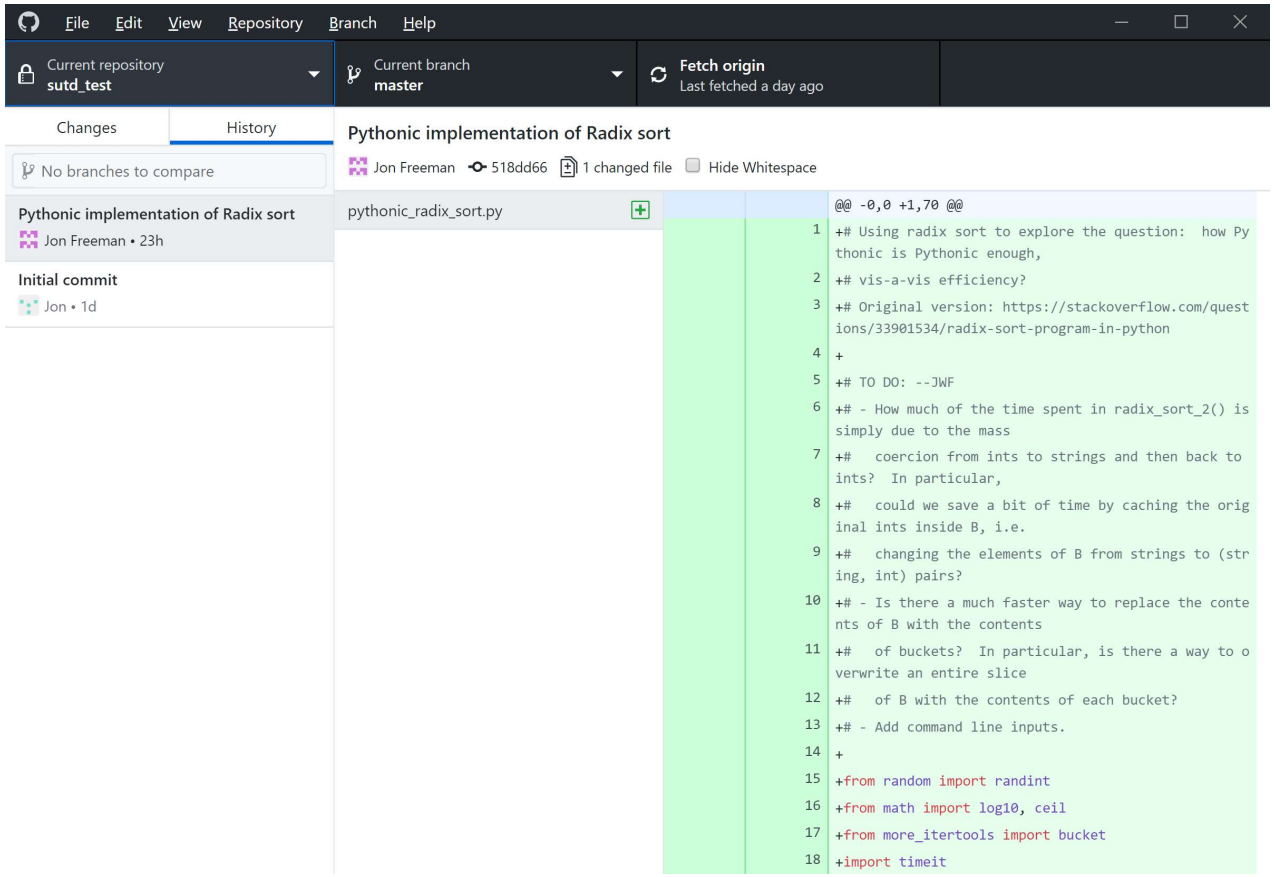
- Everyone's local repo is self-contained: it has the complete history of all changes ever made.
  - Moving a (local) repo to another laptop is fast, easy, and trouble-free
- Users can check in, branch, and merge *locally*, i.e. no server access required.
- Users only push their changes to, or pull other changes from, their server when it is necessary / possible / convenient.

# What Makes Git Different (2/2)

- There is no concept of “checking out and locking a file.”
- Instead, users signal their *intention* to introduce a change by creating a pull request.
  - Each pull request can be approved or rejected by one or more moderators.
  - The pull request itself is an artifact which creates a *platform for discussion*: anyone can use it to contribute feedback and suggestions.

# Friendly Interfaces to Git (1/2)

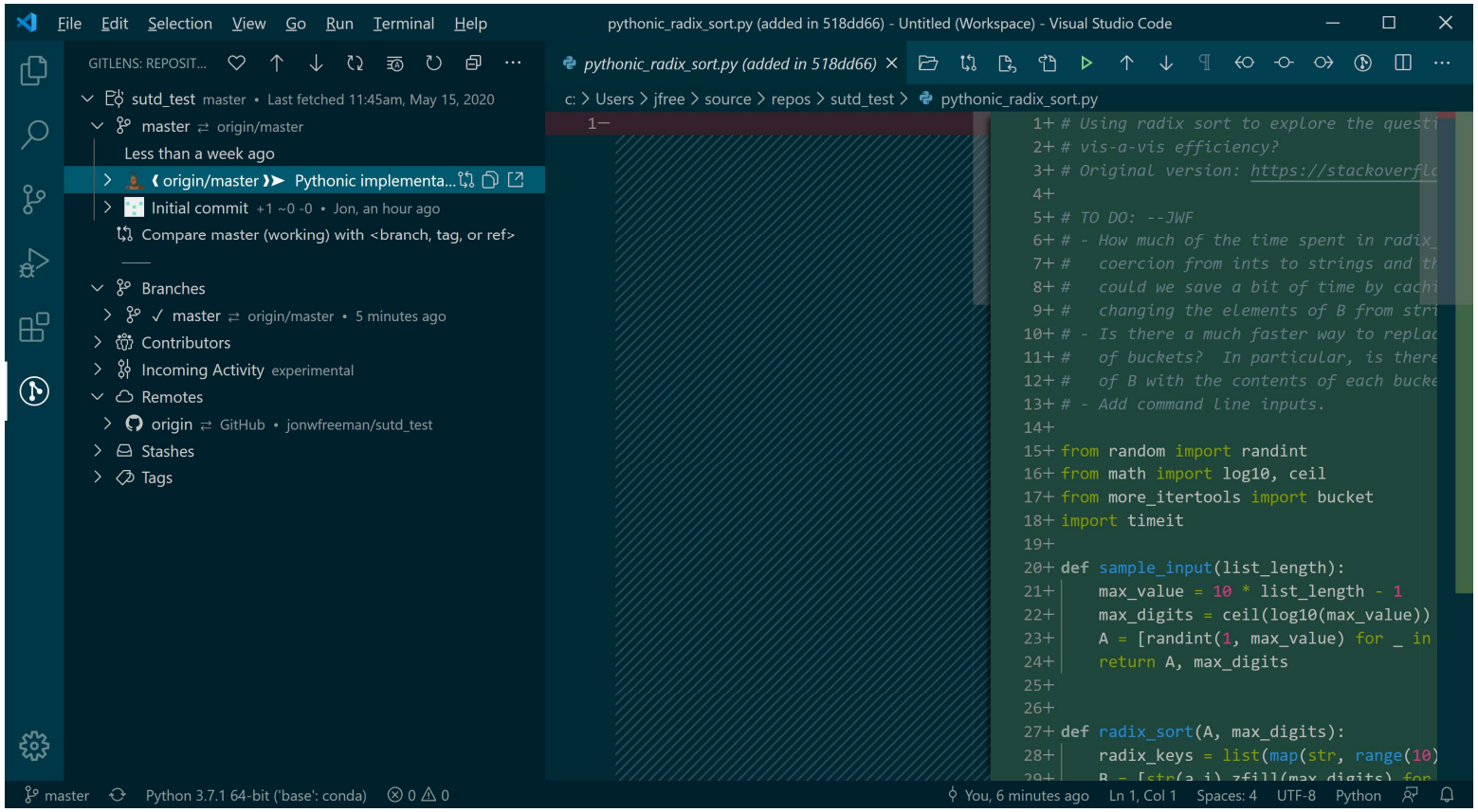
For repos stored on GitHub, GitHub Desktop is the obvious choice.




**A BETTER WORLD BY DESIGN.**

# Friendly Interfaces to Git (2/2)

For repos stored elsewhere, I currently recommend VS Code + GitLens extension.



# Working with a Git Server

Working with a Git server means

- cloning one of its remote repos to a local repo on your laptop;
- pulling (and hence merging) other people's changes from it;
- pushing your own changes to it.
  - The entities you push and pull are commits.
  - You must initiate every push and pull; a Git client will never push or pull for you.

Tip: On a multi-user repo, pull frequently!

# Working with a Git Server, cont'd.

Regarding Git servers, GitHub made a welcome announcement on 14 April 2020:

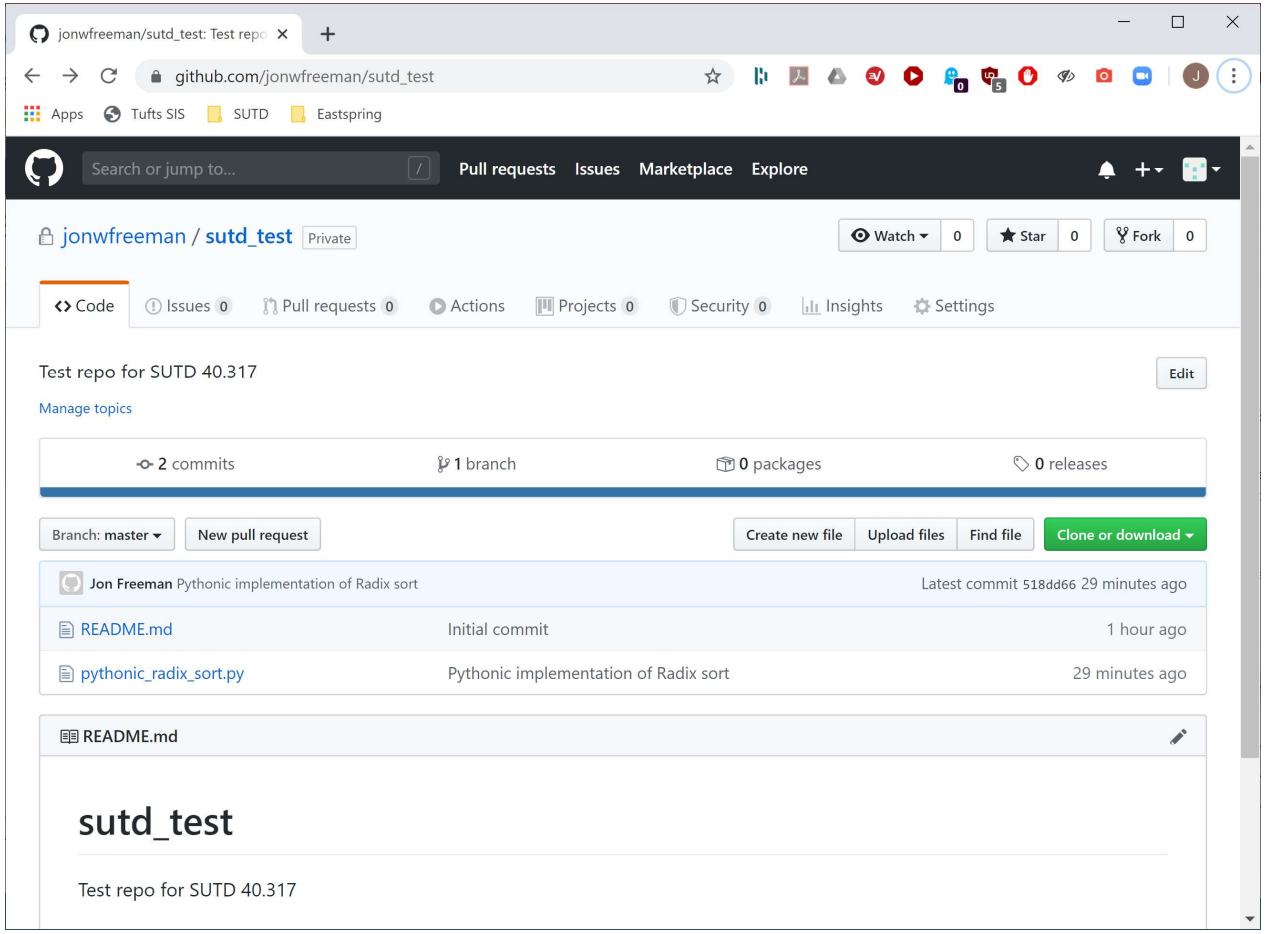
All core GitHub features are now free for all users, e.g.,

- unlimited private (or public) repos
- unlimited number of collaborators

This is not a limited-time promotion.

# Git Demonstration

Based around a private repo on GitHub.

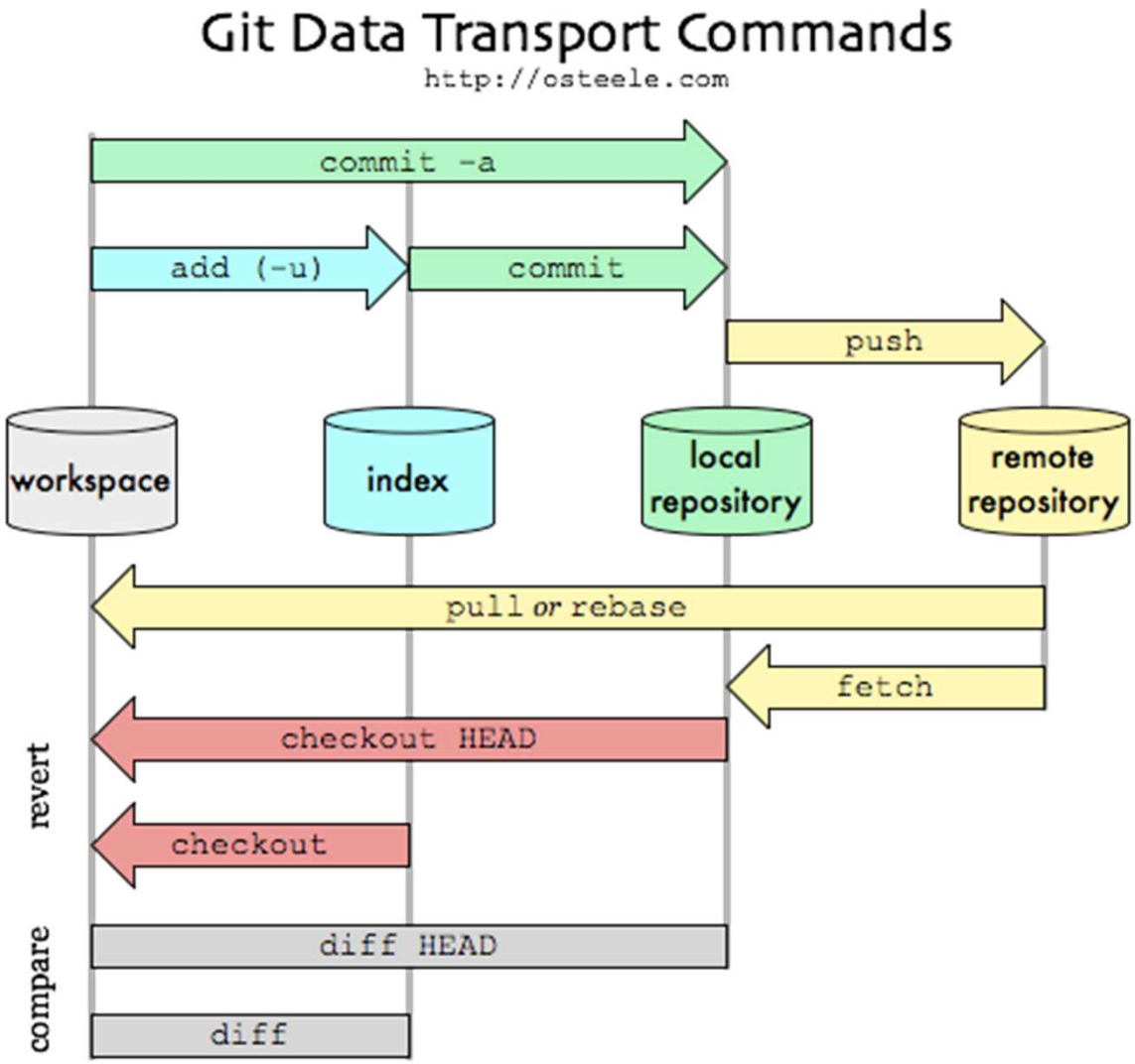


# Discussion Question

A Git repo on your laptop does not have to be pushed to any server.

Why might a purely local repo be useful to you?

# Further Details are Out of Scope




**A BETTER WORLD BY DESIGN.**

# Branching in Git

- Branching in Git is quick and easy.
- Git's recommended practice is to create a branch for *every* change you make, even small changes.
  - Easier said than done for busy people!

# Branching Strategies

There are five common strategies, for the five most common use cases:

1) No Branches. For small or medium-sized teams which don't require isolation for teams, features, or releases.
2) Branch for Release. Create a branch before release time, then merge changes from the release branch back into the "trunk" after release.

# Branching Strategies

3) Branch for Maintenance. Create a branch to maintain an old build which you might or might not merge back into the trunk later.
4) Branch for Feature. Create one branch for each new specific feature, merging them back into the trunk once the respective work is completed.
5) Branch for Team. Branch to isolate sub-teams so they can work in parallel towards different milestones.

# Branching Strategies

Most often you will want to adopt the first strategy, i.e. no branches.

The first strategy has become the most popular overall, due to the proliferation of websites and online services with frequent deployments to production.

# Branching in Git: a Clarification

These branching strategies apply to branches *as they appear on the server*.

With Git, “no branches” does not mean “never create a branch.” It means “never *push* one of your branches *to the server*.”

Git recommended practice is to create a *local* branch for every commit. Then “no branches” means *merge before pushing*.

# Algorithm Complexity: Motivation

Why are we discussing this topic?

- To think about efficient vs. inefficient algorithms more precisely.
- To have a common vocabulary for describing an algorithm's efficiency.
- To improve your ability to solve difficult coding problems.

# “Big O”, “Big Omega”, and “Big Theta”

- These are asymptotic bounds a function can have with respect to some other, typically simpler, function.
- We have two functions $f(n)$ and $g(n)$ and we use one, typically the simpler one, to *bound* the other.
- Suppose we want to use $g(n)$ to bound $f(n)$ and $c$, $c_1$, $c_2$ are constants.

# Formal Definitions for Integer Functions

- $f(n) \in \mathrm{O}(g(n))$ if $\exists\ c > 0$ and an integer $n_0 > 0$ s.t. $f(n) \leq c \cdot g(n)\ \forall\ n > n_0$.

- $f(n) \in \Omega(g(n))$ if $\exists\ c > 0$ and an integer $n_0 > 0$ s.t. $f(n) \geq c \cdot g(n)\ \forall\ n > n_0$.

- $f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0$ and an integer $n_0 > 0$ s.t. $c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n)\ \forall\ n > n_0$.

# About Those Constant Factors

Apparently $c$, $c_1$, and $c_2$ in these formulas can be anything? Really? Do their values matter or not?

Yes and no:

- They allow us to ignore CPU models, clock speeds, etc., which is highly useful.
- Even so, there will always be library functions and critical systems where they need to be as small as possible.

# “Big O” Notation Interests us the Most

- We will focus on $O(\cdot)$ bounds.
- We will use $O(\cdot)$ to denote upper bounds on the *performance of an algorithm*.
- $O(\cdot)$ can be in terms of either time or space. Usually we focus on time, hence number of *operations performed*.
- To say an algorithm “has $O(\cdot)$ complexity” is to say how many (fast, simple) operations it will perform (at most), given an input of a particular size.

# Common Time Complexities

The following time complexities tend to occur often in practice:

- Constant time
- Logarithmic time
- Linear time
- N log(N) time
- Quadratic time

# “Constant Time” Complexity

- $O(1)$ complexity means our algorithm takes constant time to finish (in the worst case), i.e. *independent of input size*.

- This is the lowest possible complexity of course.

- Examples:
  - Reading one element of an array
  - Basic arithmetic on a pair of numbers
  - Comparing two numbers

# “Logarithmic Time” Complexity

- For inputs of size $N$, the number of basic operations will be proportional to $\log N$.

- When the base of the logarithm is not stated, it is assumed to be 2.
  - Why can we make this assumption, i.e. why can the base be unspecified?

- Example: locating one element in a sorted array via binary search.

# “Linear Time” Complexity

- For inputs of size $N$, the number of basic operations will be proportional to $N$.

- Examples:
  - Locating one element in an *unsorted* array, i.e. naïve search
  - Finding the largest or smallest element in an array
  - Reading the entire contents of a file

# “N log(N) Time” Complexity

- For inputs of size $N$, the number of basic operations will be proportional to $N \log N$.

- The most popular sorting algorithms, namely Quicksort, merge sort, and heapsort, all have this time complexity.
  - In most programming languages the built-in sort function is Quicksort, due to its small constant factors.

# The “Schwartzian Transform”

Knowing that (comparison-based) sorting takes more than linear time, we can understand and appreciate a programming idiom known as the Schwartzian transform.

# “Quadratic Time” Complexity

- For inputs of size $N$, the number of basic operations will be proportional to $N^2$.

- Examples:
  - Simpler sorting algorithms such as bubble sort and insertion sort
  - Reading the entire contents of an $N$x$N$ array

# Beyond Quadratic Complexity

- There are many important problems whose best known algorithms take more than quadratic time, sometimes much more (e.g. exponential time).
  - Let's be grateful public-private key cryptography takes lots of time to break!

- In such cases people will opt for a faster algorithm yielding *approximate* solutions, provided such solutions are useful.

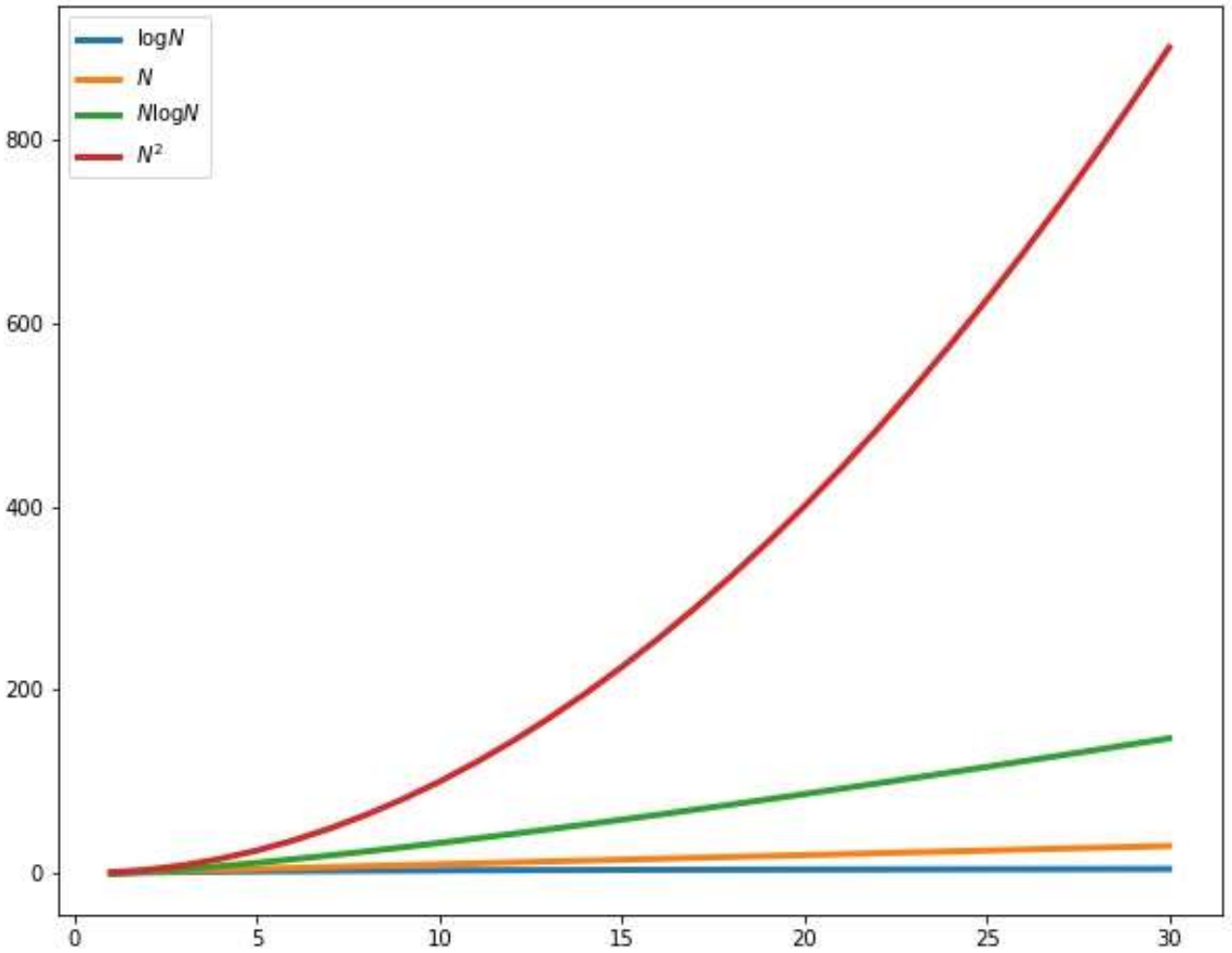
# Multiple Input Size Parameters

The $O(\cdot)$ bound of algorithms which operate on a graph is often expressed in terms of *two* variables: the number of vertices $V$, and the number of edges $E$.

When several graph algorithms with different $O(\cdot)$ bounds are available for the same purpose, your choice will depend on the relative sizes of $V$ and $E$ in your typical use cases.

# Comparing the Common Complexities

Visually:



# Comparing the Common Complexities

Via “back of the envelope” calculations:

Let $N$ be 50000 and assume one basic operation takes 5 milliseconds. Then:

- $\mathrm{O}(\log N)$ ≈ 0.08 seconds
- $\mathrm{O}(N)$ ≈ 4.2 minutes
- $\mathrm{O}(N \log N)$ ≈ 1.08 hours
- $\mathrm{O}(N^2)$ ≈ 145 days (!!)

# Which Complexity is “Fast Enough”?

It’s not a simple question:  counting everyone in Asia “only” takes linear time!

To answer this question you must know:

- the size of the largest possible input; and
- the max. allowed time to return a result.

Also, the code for a slower algorithm might be much more readable, hence preferable if your input sizes will always be fairly small.

# How Complexity Analysis Helps You

Proficiency with this topic helps you:

- Understand your current algorithm's performance
- Propose meaningful speedups, if needed
- Predict the performance of alternative algorithms

# Thank you

**A BETTER WORLD BY DESIGN.**

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN