Algorithm Engineering

Exam

- there will be a written exam
- content will be everything from lecture and exercises
- admission barrier: none
- it is **YOUR OWN** responsibility to withdraw from the course within the first 10 weeks of the semester, if you do not want to take the exam and finish the course
- date: 12.02.2018, EAP 2, R3325, 10-12 am

What is Algorithm Engineering

Algorithm engineering refers to the process required to transform a pencil-and-paper algorithm into a ***robust****,* ***efficient****,* ***well tested****, and* ***easily usable*** *implementation. Thus it encompasses a number of topics, from modeling cache behavior to the principles of good software engineering; its main focus, however, is experimentation.*

Bader, Moret, Sanders - 2002

Why Algorithm Engineering

Gap between theoretical analysis and experience in practice

- theory simplifies, abstracts from: memory hierarchy, NUMA architectures, advanced CPU instruction
- hides these parameters in constants within Big-O notation → Asymptotically optimal algorithms can be impractical
- Worst case analysis: the worst case may never occur on actual data (e.g. Quicksort)

We can bridge the gap by addressing the (impractical) assumptions by means of experimentation.

Experiment with Algorithms

You may not want to improve your implemented algorithm but you want to verify and know about its properties in real applications.

Key Properties

- Usability
- Correctness
- Efficiency

Summary

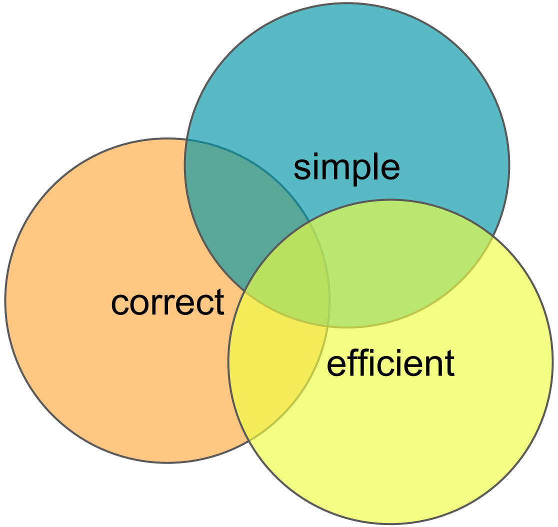
Engineering an Algorithm

1. Implement easy to understand, usable, and tested algorithm
2. Wring (desired) efficiency

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

Donald E. Knuth - 1974

Paths to Glory



Engineering Aspects

- Readability/Documentation → usability
- Testing → correctness
- Debugging → correctness
- Profiling & Measuring → efficiency

Literature

- Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison-Wesley Longman, 1999. ISBN: 0-201-61586-X
- Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. 2nd. USA: Addison-Wesley, 2010. ISBN: 0136108040, 9780136108047

Robert Pike Rules

Robert Pike Rules

1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
3. Fancy algorithms are slow when ***n*** is small, and ***n*** is usually small. Fancy algorithms have big constants. Until you know that ***n*** is frequently going to be big, don't get fancy. (Even if ***n*** does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.

Robert Pike Rules (cont.)

4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures. The following data structures are a complete list for almost all practical programs: array, linked list, hash table, binary tree. Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.
5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be selfevident. Data structures, not algorithms, are central to programming.

Robert Pike Rules (cont.)

6. There is no Rule 6.

Unix Philosophy

- Modularity
  Write simple parts connected by clean interfaces.
- Clarity
  Clarity is better than cleverness.
- Composition
  Design Programs to be connected to other programs.
- Separation
  Separate policy from mechanism; separate interfaces from engines.
- Simplicity
  Design for Simplicity; add complexity only where you must.

Unix Philosophy (cont.)

- Transparency
  Design for visibility, to make inspection and debugging easier.
- Representation
  Fold knowledge into data, so program logic can be stupid and robust.
- Least Suprise
  When designing an interface, always do the least suprising thing.
- Silence
  When a program has nothing suprising to say, it should say nothing.
- Repair
  When you must fail, fail early and loudly.

Unix Philosophy (cont.)

- Generation
  Avoid hand-hacking. Write programs that write programs, when you can.
- Optimization
  Prototype before polishing. Get it working before you optimize it.
- Extensibility
  Design for the future; it will be here sooner than you think.

Summary

- Semantically equivalent programs may not have equal performance
- Performance matters in practical applications beyond theoretical analysis.
- Adopt good engineering habits.
- Measure, measure, measure, …
- Squeezing the hardware

Version Control with Git

Distributed Version Control with Git

What is Git?

- Distributed version control system
- Developed by Linus Torvalds
- Runs almost everywhere
- Used by Linux kernel, Samba, X.Org, Qt, GNOME, Android, ...

Features

- Very flexible work flows
- Fast and scalable
- Cryptographic secure history

How Git stores its Data

Everything is an object with a SHA1

All git objects have a type, content, and size (of the content). For a given object its (object) name is a 40-digit hash (SHA1) of its content.

Object Types

- Blob Object → data
- Tree Object
  list of Blob and Tree names with its type and file name

How Git stores its Data (cont.)

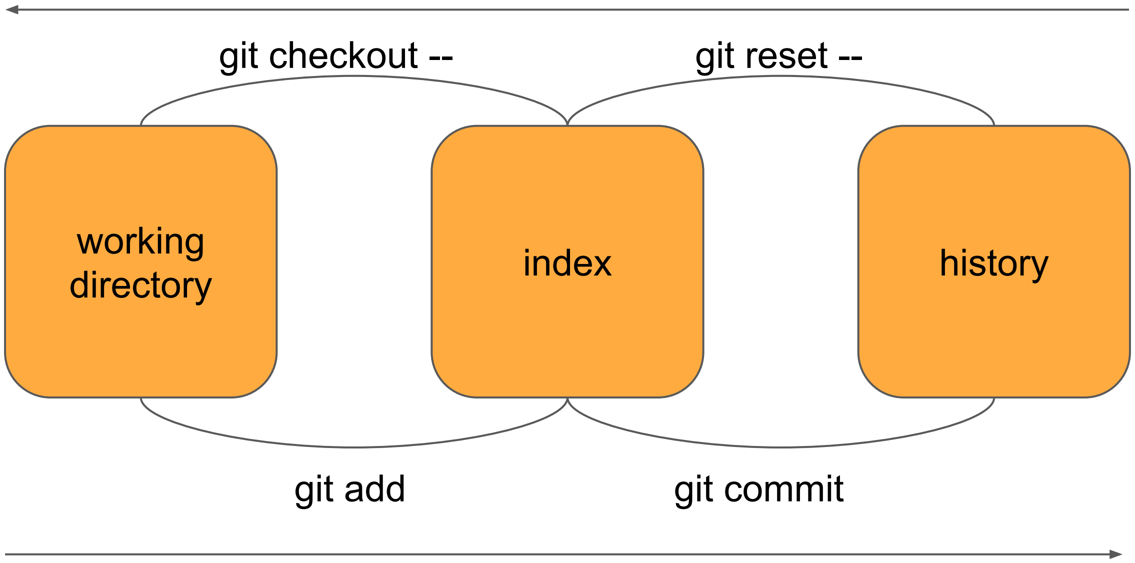
Everything is an object with a SHA1

- Commit Object
  Content: 1 tree name, 0+ parent commit name(s), author (with date), committer (with date), and a commit message
- Tag Object
  Content: type, name, tagger, tag message

The commit history of a project forms a directed acyclic graph.

Assuming SHA1 is safe: Given a commit and its SHA1 the whole history of the project is secured. I.e. a change in the history can be noticed.

Interacting with Git



Creating a Repository

- Make the current directory a repository

  `$ git init`

- Create a repository in the directory `myrepository/`

  `$ git init myrepository`

- Create a bare repository accessible by all

  `$ git init --bare --shared=all /git/myrepository.git`