Lecture #35

Today

- Dynamic programming and memoization.
- Anatomy of Git.

Dynamic Programming

- A puzzle (D. Garcia):
  - Start with a list with an even number of non-negative int
  - Each player in turn takes either the leftmost number rightmost.
  - Idea is to get the largest possible sum.
- Example: starting with (6, 12, 0, 8), you (as first player) sho the 8. Whatever the second player takes, you also get the 1 total of 20.
- Assuming your opponent plays perfectly (i.e., to get as much sible), how can you maximize your sum?
- Can solve this with exhaustive game-tree search.

Obvious Program

- Recursion makes it easy, again:

```
int bestSum(int[] V) {
  int total, i, N = V.length;
  for (i = 0, total = 0; i < N; i += 1) total += V[i];
  return bestSum(V, 0, N-1, total);
}

/** The largest sum obtainable by the first player in the c
 *  game on the list V[LEFT .. RIGHT], assuming that TOTAL
 *  sum of all the elements in V[LEFT .. RIGHT].  */
int bestSum(int[] V, int left, int right, int total) {
  if (left > right)
    return 0;
  else {
    int L = total - bestSum(V, left+1, right, total-V[left]
    int R = total - bestSum(V, left, right-1, total-V[right
    return Math.max(L, R);
  }
}
```

- Time cost is $C(0) = 1,\ C(N) = 2C(N-1)$; so $C(N) \in \Theta(2^N)$

Still Another Idea from CS61A

- The problem is that we are recomputing intermediate resul
times.
- Solution: *memoize* the intermediate results. Here, we pa
$N \times N$ array ($N =$ `V.length`) of memoized results, initialize

```
int bestSum(int[] V, int left, int right, int total, int[][
  if (left > right)
    return 0;
  else if (memo[left][right] == -1) {
    int L = total - bestSum(V, left+1, right, total-V[left]
    int R = total - bestSum(V, left, right-1, total-V[right
    memo[left][right] = Math.max(L, R);
  }
  return memo[left][right];
 }
}
```

- Now the number of recursive calls to `bestSum` must be $O($
$N =$ the length of V, an enormous improvement from $\Theta(2^N)$!

Iterative Version

- I prefer the recursive version, but the usual presentation
  idea—known as *dynamic programming*—is iterative:

```
int bestSum(int[] V) {
  int[][] memo = new int[V.length][V.length];
  int[][] total = new int[V.length][V.length];
  for (int i = 0; i < V.length; i += 1)
    memo[i][i] = total[i][i] = V[i];
  for (int k = 1; k < V.length; k += 1)
    for (int i = 0; i < V.length-k-1; i += 1) {
      total[i][i+k] = V[i] + total[i+1][i+k];
      int L = total[i][i+k] - memo[i+1][i+k];
      int R = total[i][i+k] - memo[i][i+k-1];
      memo[i][i+k] = Math.max(L, R);
    }
  return memo[0][V.length-1];
}
```

- That is, we figure out ahead of time the order in which the
  ized version will fill in `memo`, and write an explicit loop.
- Save the time needed to check whether result exists.
- But I say, why bother unless it's necessary to save space?

Longest Common Subsequence

- **Problem:** Find length of the longest string that is a subsequ each of two other strings.
- **Example:** Longest common subsequence of
  "`sally␣sells␣sea␣shells␣by␣the␣seashore`" and
  "`sarah␣sold␣salt␣sellers␣at␣the␣salt␣mines`"

  is
  "`sa␣sl␣sa␣sells␣␣the␣sae`" (length 23)
- Similarity testing, for example.
- Obvious recursive algorithm:

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  if (k0 == 0 || k1 == 0) return 0;
  if (S0[k0-1] == S1[k1-1]) return 1 + lls(S0, k0-1, S1, k1
  else return Math.max(lls(S0, k0-1, S1, k1), lls(S0, k0, S
}
```

- Exponential, but obviously memoizable.

Memoized Longest Common Subsequence

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  int[][] memo = new int[k0+1][k1+1];
  for (int[] row : memo) Arrays.fill(row, -1);
  return lls(S0, k0, S1, k1, memo);
}

private static int lls(String S0, int k0, String S1, int k1, int[
  if (k0 == 0 || k1 == 0) return 0;
  if (memo[k0][k1] == -1) {
    if (S0[k0-1] == S1[k1-1])
      memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
    else
      memo[k0][k1] = Math.max(lls(S0, k0-1, S1, k1, memo),
                              lls(S0, k0, S1, k1-1, memo));
  }
  return memo[k0][k1];
}
```

Q: How fast will the memoized version be?

Memoized Longest Common Subsequence

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  int[][] memo = new int[k0+1][k1+1];
  for (int[] row : memo) Arrays.fill(row, -1);
  return lls(S0, k0, S1, k1, memo);
}

private static int lls(String S0, int k0, String S1, int k1, int[
  if (k0 == 0 || k1 == 0) return 0;
  if (memo[k0][k1] == -1) {
    if (S0[k0-1] == S1[k1-1])
      memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
    else
      memo[k0][k1] = Math.max(lls(S0, k0-1, S1, k1, memo),
                              lls(S0, k0, S1, k1-1, memo));
  }
  return memo[k0][k1];
}
```

Q: How fast will the memoized version be? $\Theta(k_0 \cdot k_1)$

Git: A Case Study in System and Data-Struct Design

- Git is a distributed version-control system, apparently the m ular of these currently.
- Conceptually, it stores snapshots (*versions*) of the files an tory structure of a project, keeping track of their relati authors, dates, and log messages.
- It is *distributed,* in that there can be many copies of a give itory, each supporting indepenent development, with mach transmit and reconcile versions between repositories.
- Its operation is extremely fast (as these things go).

A Little History

- Developed by Linus Torvalds and others in the Linux communi
  the developer of their previous, propietary VCS (Bitkeepe
  drew the free version.
- Initial implementation effort seems to have taken about 2–3
  in time for the 2.6.12 Linux kernel release in June, 2005.
- As for the name, according to Wikipedia,

  > Torvalds has quipped about the name Git, which is Br
  > English slang meaning "unpleasant person". Torvalds said:
  > an egotistical bastard, and I name all my projects after my
  > First 'Linux', now 'git'." The man page describes Git as
  > stupid content tracker."

- Initially, was a collection of basic primitives (now called "plu
  that could be scripted to provide desired functionality.
- Then, higher-level commands ("porcelain") built on top of t
  provide a convenient user interface.

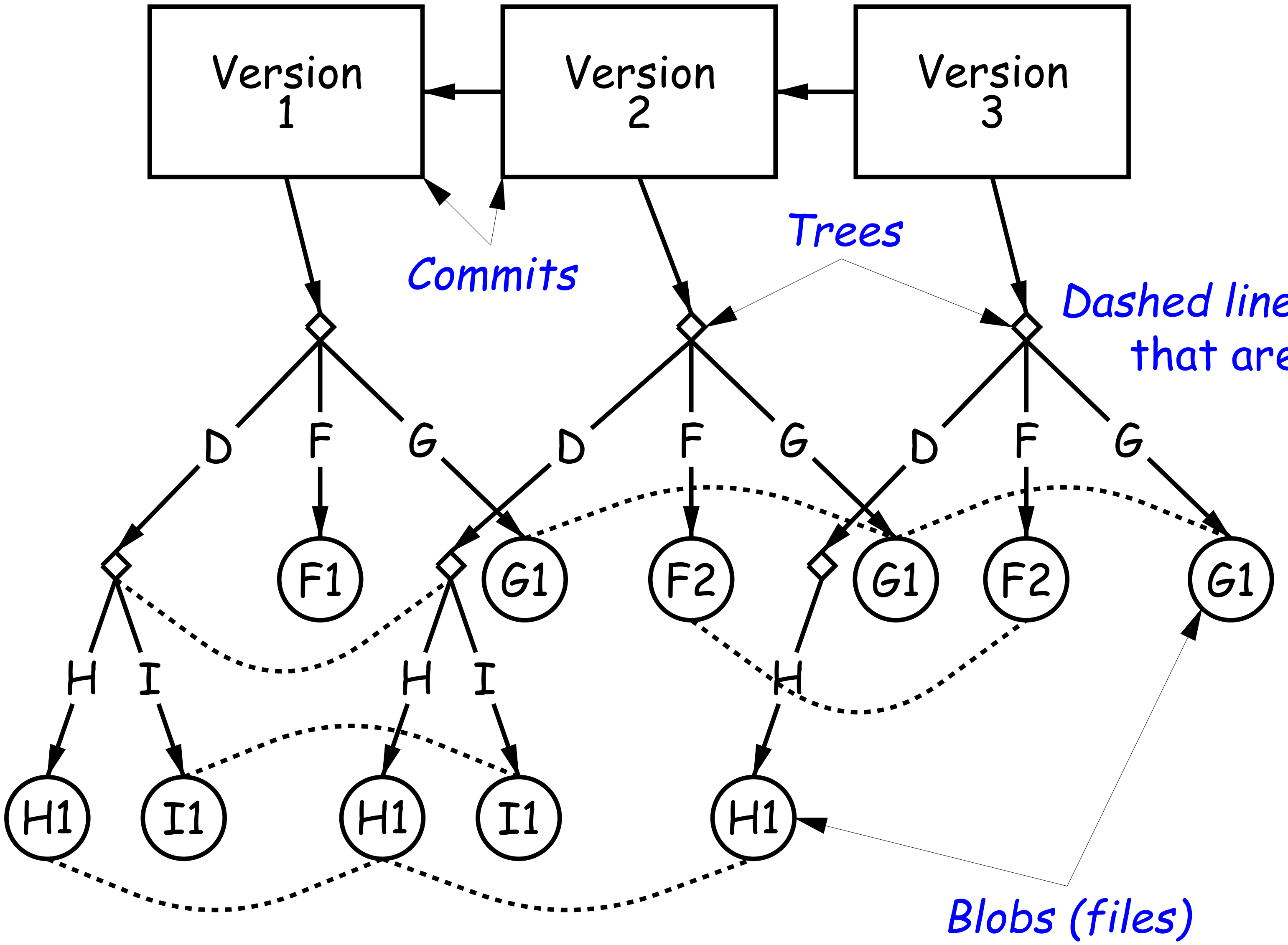
Major User-Level Features (I)

- Abstraction is of a graph of versions or snapshots (called *c*
  of a complete project.
- The graph structure reflects ancestory: which versions car
  which.
- Each commit contains
  - A directory tree of files (like a Unix directory).
  - Information about who committed and when.
  - Log message.
  - Pointers to commit (or commits, if there was a merge) fro
    the commit was derived.

Conceptual Structure

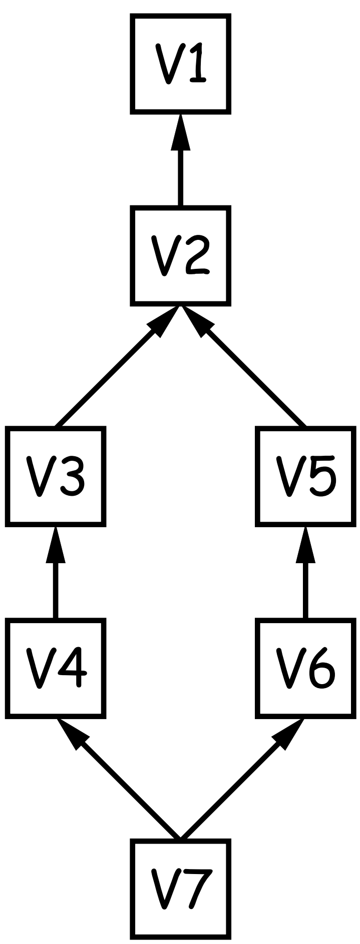
- Main internal components consist of four types of *object:*
  - *Blobs:* basically hold contents of files.
  - *Trees:* directory structures of files.
  - *Commits:* Contain references to trees and additional info
    (committer, date, log message).
  - *Tags:* References to commits or other objects, with ac
    information, intended to identify releases, other importa
    sions, or various useful information. (Won't mention furt
    day).

Commits, Trees, Files

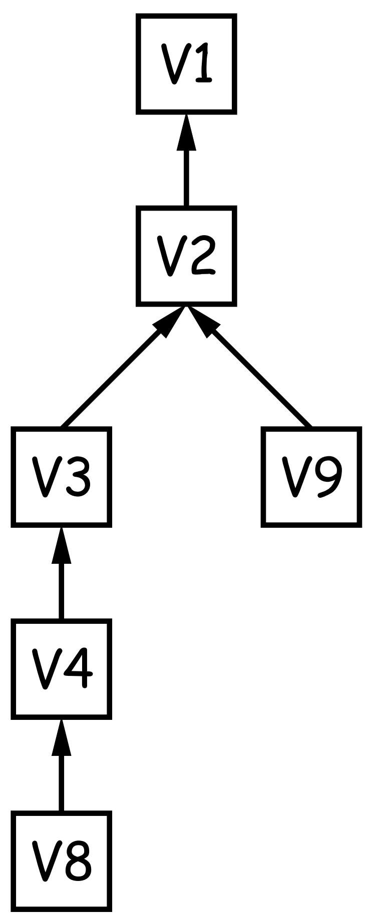


Version Histories in Two Repositories

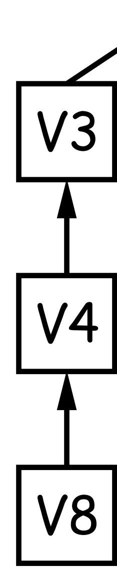
Repository 1



Repository 2



R
after



Major User-Level Features (II)

- Each commit has a name that uniquely identifies it to all ver
- Repositories can transmit collections of versions to each oth
- Transmitting a commit from repository A to repository B r only the transmission of those objects (files or directory that B does not yet have (allowing speedy updating of repos
- Repositories maintain named *branches*, which are simply ide of particular commits that are updated to keep track of t recent commits in various lines of development.
- Likewise, *tags* are essentially named pointers to particular c Differ from branches in that they are not usually changed.

Internals

- Each Git repository is contained in a directory.
- Repository may either be *bare* (just a collection of obje metadata), or may be included as part of a working director
- The data of the repository is stored in various *objects* corr ing to files (or other "leaf" content), trees, and commits.
- To save space, data in files is *compressed*.
- Git can *garbage-collect* the objects from time to time to sa tional space.

The Pointer Problem

- Objects in Git are files. How should we represent pointers b
  them?
- Want to be able to *transmit* objects from one repository to
  with different contents. How do you transmit the pointers?
- Only want to transfer those objects that are missing in the
  repository. How do we know which those are?
- Could use a counter in each repository to give each object
  unique name. But how can that work consistently for two i
  dent repositories?

Content-Addressable File System

- Could use some way of naming objects that is universal.
- We use the names, then, as pointers.
- Solves the "Which objects don't you have?" problem in an way.
- Conceptually, what is invariant about an object, regardless o itory, is its *contents*.
- But can't use the contents as the name for obvious reasons.
- Idea: Use a *hash of the contents* as the address.
- Problem: That doesn't work!
- Brilliant Idea: Use it anyway!!

How A Broken Idea Can Work

- The idea is to use a hash function that is so unlikely to have sion that we can ignore that possibility.
- *Cryptographic Hash Functions* have relevant property.
- Such a function, f, is designed to withstand cryptoanalytic In particular, should have
  - *Pre-image resistance:* given $h = f(m)$, should be comput infeasible to find such a message m.
  - *Second pre-image resistance:* given message m_1, should b sible to find $m_2 \neq m_1$ such that $f(m_1) = f(m_2)$.
  - *Collision resistance:* should be difficult to find *any* two m $m_1 \neq m_2$ such that $f(m_1) = f(m_2)$.
- With these properties, scheme of using hash of contents as extremely unlikely to fail, even when system is used maliciou

SHA1

- Git uses *SHA1* (Secure Hash Function 1).
- Can play around with this using the `hashlib` module in Pytho
- All object names in Git are therefore 160-bit hash codes tents, in hex.
- E.g. a recent commit in the shared CS61B repository could be (if needed) with

```
git checkout e5984920195676621 8a3ad6ee1c3aab37d
```