

Lecture #35

Today

- Dynamic programming and memoization.
- Anatomy of Git.

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 1

Dynamic Programming

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

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 2

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 choosing
 * game on the list V[LEFT .. RIGHT], assuming that TOTAL is the
 * 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 $C(N) \in \Theta(2^N)$

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 3

Still Another Idea from CS61A

- The problem is that we are recomputing intermediate results many times.
- Solution: *memoize* the intermediate results. Here, we pass in an $N \times N$ array ($N = V.length$) of memoized results, initialized to -1.

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

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

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 4

Iterative Version

- I prefer the recursive version, but the usual presentation of this 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 memoized 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?

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 5

Longest Common Subsequence

- **Problem:** Find length of the longest string that is a subsequence of 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_s_l_s_a_s_e_l_l_s_u_t_h_e_s_a_e" (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, k1-1);
    else return Math.max(lls(S0, k0-1, S1, k1), lls(S0, k0, S1, k1-1));
}
```

- Exponential, but obviously memoizable.

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 6

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[][] memo) {
    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?

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 7

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[][] memo) {
    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? $\Theta(k_0 \cdot k_1)$

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 8

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

- Git is a distributed version-control system, apparently the most popular of these currently.
- Conceptually, it stores snapshots (*versions*) of the files and directory structure of a project, keeping track of their relationships, authors, dates, and log messages.
- It is *distributed*, in that there can be many copies of a given repository, each supporting independent development, with machinery to transmit and reconcile versions between repositories.
- Its operation is extremely fast (as these things go).

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 9

A Little History

- Developed by Linus Torvalds and others in the Linux community when the developer of their previous, proprietary VCS (Bitkeeper) withdrew the free version.
- Initial implementation effort seems to have taken about 2-3 months, 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 British English slang meaning "unpleasant person". Torvalds said: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." The man page describes Git as "the stupid content tracker."
- Initially, was a collection of basic primitives (now called "plumbing") that could be scripted to provide desired functionality.
- Then, higher-level commands ("porcelain") built on top of these to provide a convenient user interface.

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 10

Major User-Level Features (I)

- Abstraction is of a graph of versions or snapshots (called *commits*) of a complete project.
- The graph structure reflects ancestry: which versions came from 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) from which the commit was derived.

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 11

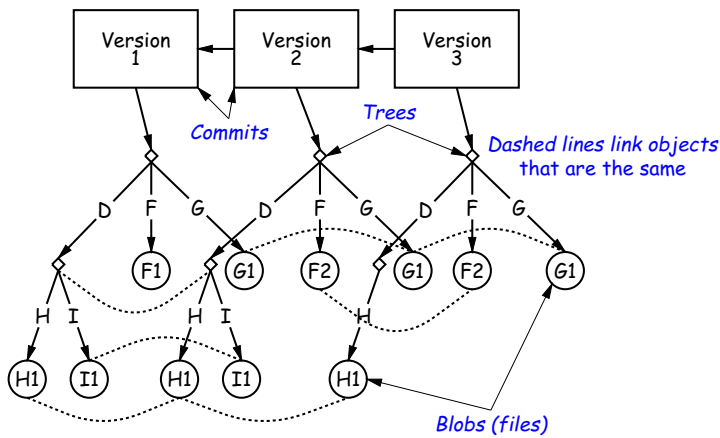
Conceptual Structure

- 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 information (committer, date, log message).
 - *Tags*: References to commits or other objects, with additional information, intended to identify releases, other important versions, or various useful information. (Won't mention further to-day).

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 12

Commits, Trees, Files

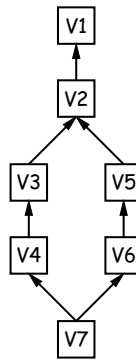


Last modified: Wed Nov 14 11:52:57 2018

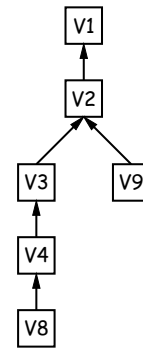
CS61B: Lecture #35 13

Version Histories in Two Repositories

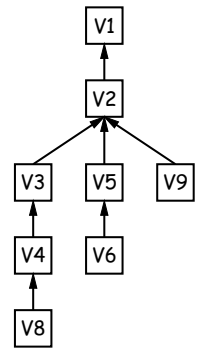
Repository 1



Repository 2



Repository 2
after pushing V6 to it



Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 14

Major User-Level Features (II)

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

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 15

Internals

- Each Git repository is contained in a directory.
- Repository may either be *bare* (just a collection of objects and metadata), or may be included as part of a working directory.
- The data of the repository is stored in various *objects* corresponding 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 save additional space.

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 16

The Pointer Problem

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

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 17

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 obvious way.
- Conceptually, what is invariant about an object, regardless of repository, 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!!

Last modified: Wed Nov 14 11:52:57 2018

CS61B: Lecture #35 18

How A Broken Idea Can Work

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

SHA1

- Git uses **SHA1** (Secure Hash Function 1).
- Can play around with this using the `hashlib` module in Python3.
- All object names in Git are therefore 160-bit hash codes of contents, in hex.
- E.g. a recent commit in the shared CS61B repository could be fetched (if needed) with

```
git checkout e59849201956766218a3ad6ee1c3aab37dfec3fe
```