CS5112: Algorithms and Data Structures for Applications

Lecture 3: Hashing

Ramin Zabih

Some figures from Wikipedia/Google image search
Administrivia

• Web site is: https://github.com/cornelltech/CS5112-F18
  – As usual, this is pretty much all you need to know
• Quiz 1 out today, due Friday 11:59PM
  – Very high tech!
  – Coverage through Greg’s lecture on Tuesday
• TA’s and consultants coming shortly
• We have a slack channel
Today

• Clinic this evening (here), Greg on hashing
• Associative arrays
• Efficiency: Asymptotic analysis, effects of locality
• Hashing
• Additional requirements for cryptographic hashing
• Fun applications of hashing!
  – Lots of billion-dollar ideas
Associative arrays

- Fundamental data structure in CS
- Holds (key, value) pairs, a given key appears at most once
- API for associative arrays (very informal!)
  - Insert(k, v, A) -> A’, where A’ has the new pair (k, v) along with A
  - Lookup(k, A) -> v, where v is from the pair (k, v) in A
- Lots of details we will ignore today
  - Avoiding duplication, raising exceptions, etc.
- “Key” question: how to do this fast
How computer scientists think about efficiency

• Two views: asymptotic and ‘practical’
• Generally give the same result, but math vs engineering
• Asymptotic analysis, a.k.a. “big O”
  – Mathematical treatment of algorithms
  – Worst case performance
  – Consider the limit as input gets larger and larger
Big O notation: main ideas

• 2 big ideas:
  – [#1] Think about the worst case (don’t assume luck)
  – [#2] Think about all hardware (don’t assume Intel/AMD)

• Example: find duplicates in array of $n$ numbers, dumbly
  – For each element, scan the rest of the array
  – We scan $n - 1$, $n - 2$, ... elements [#2]
  – So we examine $\sum_{i=1}^{n-1} i$ elements, which is $\frac{(n-1)(n-2)}{2}$
  – Which is ugly... but what happens as $n \to \infty$? [#1]

• Write this as $O(n^2)$
Some canonical complexity classes

- **Constant time algorithms, $O(1)$**
  - Running time doesn’t depend on input
  - Example: find the first element in an array

- **Linear time algorithms, $O(n)$**
  - Constant work per input item, in the worst case
  - Example: find a particular item in the array

- **Quadratic time algorithms, $O(n^2)$**
  - Linear work per input item, such as find duplicates

- **Clever variants of quadratic time algorithms, $O(n \log n)$**
  - A few will be discussed in the clinic tonight
Big O notation and its limitations

• Overall, practical performance correlates very strongly with asymptotic complexity (= big O)
  – The exceptions to this are actually famous
• Warning: this does not mean that on a specific input an $O(1)$ algorithm will be faster than an $O(n^2)$ one!
Linked lists
Linked lists as memory arrays

- We’ll implement linked lists using a memory array
  - This is very close to what the hardware does
    
    \[
    \begin{array}{cccccccccc}
    1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
    \end{array}
    \]

- A linked list contains “cells”
- A value, and where the next cell is
  - We will represent cells by a pair of adjacent array entries
Example

8 → 4 → 1 → 3

1 2 3 4 5 6 7 8 9

8 5 1 7 4 3 3 0 0
Locality and efficiency

• Locality is important for computation due to physics
  – The amount of information you can pack into a given area
• The hardware is faster when your memory references are local in terms of time and space
• Time locality: access the same data repeatedly, then move on
• Space locality: access nearby data (in memory)
Memory hierarchy in a modern CPU

**Typical Memory Hierarchy (Intel Core i7)**

- **L0**: CPU registers (optimized by compiler)
- **L1**: On-chip L1 cache (SRAM) 8-way associative in Intel Core i7
- **L2**: Off-chip L2 cache (SRAM) 8-way associative in Intel Core i7
- **L3**: Off-chip cache L3 shared by multiple cores (SRAM) 16-way associative in Intel Core i7
- **L4**: Main memory (DRAM)
- **L5**: Local secondary storage (local disks)
- **L6**: Remote secondary storage (distributed file systems, web servers)
Complexity of associative array algorithms

<table>
<thead>
<tr>
<th>DATA STRUCTURE</th>
<th>INSERT</th>
<th>LOOKUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Binary search tree (BST)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced BST</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

So, why use anything other than a hash table?
Hashing in one diagram

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value_1</td>
</tr>
<tr>
<td>1</td>
<td>value_2</td>
</tr>
<tr>
<td>2</td>
<td>value_3</td>
</tr>
<tr>
<td>3</td>
<td>value_4</td>
</tr>
</tbody>
</table>
What makes a good hash function?

• Almost nobody writes their own hash function
  – Like a random number generator, very easy to get this wrong!
• Deterministic
• Uniform
  – With respect to your input!
  – Technical term for this is entropy
• (Sometimes) invariant to certain changes
  – Example: punctuation, capitalization, spaces
Examples of good and bad hash functions

• Suppose we want to build a hash table for CS5112 students
• Is area code a good hash function?
• How about zip code?
• Social security numbers?
  – https://www.ssa.gov/history/ssn/geocard.html
• What is the best and worst hash function you can think of?
Cryptographic hashing

• Sample application: bragging that you’ve solved HW1
  – How to show this without sharing your solution?
• Given the output, **hard** to compute an input with that output
  – Given \( m = \text{hash}(s) \) hard to find \( s': m = \text{hash}(s') \)
  – Sometimes called a 1-way function
• Given the input, hard to find a matching input
  – Given \( s \) hard to find \( s': \text{hash}(s) = \text{hash}(s') \)
• Hard to find two inputs with same output: \( \text{hash}(s) = \text{hash}(s') \)
What does “hard to find” mean?

- Major topic, center of computational complexity
- Loosely speaking, we can’t absolutely prove this
- But we can show that if we could solve one problem, we could solve another problem that is widely believed to be hard
  - Because lots of people have tried to solve it and failed!
- This proves that one problem is at least as hard as another
  - “Problem reduction”
Handling collisions

• More common than you think!
  – Birthday paradox
  – Example: 1M buckets and 2,450 keys uniformly distributed
  – 95% chance of a collision

• Easiest solution is chaining
  – E.g. with linked lists
Now for the fun part...

What cool stuff can we do with hashing?
Rabin-Karp string search

• Find one string ("pattern") in another
  – Naively we repeatedly shift the pattern
  – Example: To find "greg" in "richardandgreg" we compare greg against "rich", "icha", "char", etc. ('shingles' at the word level)

• Instead let's use a hash function $h$

• We first compare $h(\text{"greg"})$ with $h(\text{"rich"})$, then $h(\text{"icha"})$, etc.

• Only if the hash values are equal do we look at the string
  – Because $x = y \Rightarrow h(x) = h(y)$ (but not $\Leftarrow$ of course!)
Rolling hash functions

• To make this computationally efficient we need a special kind of hash function $h$
• As we go through “richardandgreg” looking for “greg” we will be computing $h$ on consecutive strings of the same length
• There are clever ways to do this, but to get the flavor of them here is a naïve way that mostly works
  – Take the ASCII values of all the characters and multiply them
  – Reduce this modulo something reasonable
Large backups

- How do we backup all the world’s information?
- Tape robots!
- VERY SLOW access
Bloom filters

- Suppose you are processing items, most of them are cheap but a few of them are very expensive.
  - Can we quickly figure out if an item is expensive?
  - Could store the expensive items in an associative array
  - Or use a binary valued hash table?
    - Efficient way to find out if an item might be expensive
- We will query set membership but allow false positives
  - I.e. the answer to $s \in S$ is either ‘possibly’ or ‘definitely not’
- Use a few hash functions $h_i$ and bit array $A$
  - To insert $s$ we set $A[h_i(s)] = 1 \forall i$
Bloom filter example

- Example has 3 hash functions and 18 bit array
- \( \{x, y, z\} \) are in the set, \( w \) is not

Figure by David Eppstein, https://commons.wikimedia.org/w/index.php?curid=2609777
Application: web caching

- CDN’s, like Akamai, make the web work (~70% of traffic)
- About 75% of URL’s are ‘one hit wonders’
  – Never looked at again by anyone
  – Let’s not do the work to put these in the disk cache!
    - Cache on second hit
- Use a Bloom filter to record URL’s that have been accessed
- A one hit wonder will not be in the Bloom filter

Bloom filters really work!

Cool facts about Bloom filters

• You don’t need to build different hash functions, you can use a single one and divide its output into fields (usually)
• Can calculate probability of false positives and keep it low
• Time to add an element to the filter, or check if an element is in the filter, is independent of the size of the element (!)
• You can estimate the size of the union of two sets from the bitwise OR of their Bloom filters
MinHash

• Suppose you want to figure out how similar two sets are
  – Jacard similarity measure is \( J(A, B) = \frac{|A \cap B|}{|A \cup B|} \)
  – This is 0 when disjoint and 1 when identical
• Define \( h_{min}(S) \) to be the element of \( S \) with the smallest value of the hash function \( h \), i.e. \( h_{min}(S) = \arg \min_{s \in S} h(s) \)
  – This uses hashing to compute a set’s “signature”
• Probability that \( h_{min}(A) = h_{min}(B) \) is \( J(A, B) \)
• Do this with a bunch of different hash functions
MinHash applications

• Plagiarism detection in articles
• Collaborative filtering!
  – Amazon, NetFlix, etc.
Distributed hash tables (DHT)

- BitTorrent, etc.
- Given a file name and its data, store/retrieve it in a network
- Compute the hash of the file name
- This maps to a particular processor, which holds the file