Randomized Algorithms
Part Four
Announcements

- Problem Set Three due right now.
  - Due Wednesday using a late day.
- Problem Set Four out, due next Monday, July 29.
  - Play around with randomized algorithms!
  - Approximate \textbf{NP}-hard problems!
  - Explore a recent algorithm and why hashing matters!
Outline for Today

- **Chained Hash Tables**
  - How can you compactly store a small subset of a large set of elements?

- **Universal Hash Functions**
  - Groups of functions that distribute elements nicely.
Associative Structures

• The data structures we've seen so far are linear:
  • Stacks, queues, priority queues, lists, etc.
• In many cases, we want to store data in an unordered fashion.
• Queries like
  • Add element $x$.
  • Remove element $x$.
  • Is element $x$ contained?
Bitvectors

- A **bitvector** is a data structure for storing a set of integers in the range \{0, 1, 2, 3, ..., Z − 1\}.
- Store as an array of Z bits.
- If bit at position \(x\) is 0, \(x\) does not appear in the set.
- If bit at position \(x\) is 1, \(x\) appears in the set.
Analyzing Bitvectors

- What is the runtime for
  - Inserting an element?
  - Removing an element?
  - Checking if an element is present?
- How much space is used if the bitvector contains all $Z$ possible elements?
- How much space is used if the bitvector contains $n$ of the $Z$ possible elements?
Another Idea

• Store elements in an unsorted array.
• To determine whether $x$ is contained, scan over the array elements and return whether $x$ is found.
• To add $x$, check to see $x$ is contained and, if not, append $x$.
• To remove $x$, check to see if $x$ is contained and, if so, remove $x$. 
Analyzing this Approach

- How much space is used if the array contains all $Z$ possible elements?
- How much space is used if the array contains $n$ of the $Z$ possible elements?
- What is the runtime for
  - Inserting an element?
  - Removing an element?
  - Checking if an element is present?
The Tradeoff

- Bitvectors are fast because we know where to look to find each element.
- Bitvectors are space-inefficient because we store one bit per possible element.
- Unsorted arrays are slow because we have to scan every element.
- Unsorted arrays are space-efficient because we only store the elements we use.
- This is a time-space tradeoff: we can improve performance by using more space.
Combining the Approaches

- Bitvectors always use a fixed amount of space and support fast lookups.
  - Good when number of possible elements is low, bad when number of possible elements is large.
- Unsorted arrays use variable space and don't support fast lookups.
  - Good when number of used elements is low, bad when number of used elements is large.
Chained Hash Tables

- Suppose we have a **universe** $U$ consisting of all possible elements that we could want to store.
- Create $m$ **buckets**, numbered $\{0, 1, 2, \ldots, m-1\}$ as an array of length $m$. Each bucket is an unsorted array of elements.
- Find a rule associating each element in $U$ with some bucket.
- To see if $x$ is contained, look in the bucket $x$ is associated with and see if $x$ is there.
- To add $x$, see if $x$ is contained and add it to the appropriate bucket if it's not.
- To remove $x$, see if $x$ is contained and remove it from its bucket if it is.
Association rule: 
\[(\text{length of first name}) \mod 4\]
Bucket 0  Bucket 1  Bucket 2  Bucket 3

Association rule: Party in bucket 1!
Analyzing Runtime

- The three basic operations on a hash table (insert, remove, lookup) all run in time $O(1 + X)$, where $X$ is the total number of elements in the bucket visited.
  - *(Why is there a 1 here?)*
- Runtime depends on how well the elements are distributed.
- If $n$ elements are distributed evenly across all the buckets, runtime is $O(1 + n / m)$.
- If there are $n$ elements distributed all into the same bucket, runtime is $O(n)$. 
Hash Functions

• Chained hash tables only work if we have a mechanism for associating elements of the universe with buckets.

• A **hash function** is a function

\[
h : U \rightarrow \{0, 1, 2, ..., m - 1\}
\]

• In other words, for any \( x \in U \), the value of \( h(x) \) is the bucket that \( x \) belongs to.

• Since \( h \) is a mathematical function, it's defined for all inputs in \( U \) and always produces the same output given the same input.

• For simplicity, we'll assume hash functions can be computed in \( O(1) \) time.
Choosing Good Hash Functions

• The efficiency of a hash table depends on the choice of hash function.

• In the upcoming analysis, we will assume $|U| \gg m$ (that is, there are vastly more elements in the universe than there are buckets in the hash table.)
  
  • Assume at least $|U| > mn$, but probably more.
A Problem

**Theorem:** For any hash function $h$, there is a series of $n$ values that, if stored in the table, all hash to the same bucket.

**Proof:** Because there are $m$ buckets, under the assumption that $|U| > mn$, by the pigeonhole principle there must be at least $n + 1$ elements that hash to the same bucket. Inserting any $n$ of those elements into the hash table places all those elements into the same bucket. ■
A Problem

- No matter how clever we are with our choice of hash function, there will always be an input that will degenerate operations to worst-case $\Omega(n)$ time.
- Theoretically, limits the worst-case effectiveness of chained hashing.
- Practically, leads to denial-of-service attacks.
Randomness to the Rescue

- For any fixed hash function, there is a degenerate series of inputs.
- The hash function itself cannot involve randomness.
  - *(Why?)*
- However, what if we choose which hash function to use at random?
A (Very Strong) Assumption

- Let's suppose that when we create our hash table, we choose a `totally random function` \( h : U \rightarrow \{0, 1, 2, ..., m - 1\} \) as our hash function.
  - This has some issues; more on that later.
- Under this assumption, what would the expected cost of the three major hash table operations be?
Some Notation

• As before, let $n$ be the number of elements in a hash table.
• Let those elements be $x_1, x_2, \ldots, x_n$.
• Suppose that the element that we're looking up is the element $z$.
  • Perhaps $z$ is in the list; perhaps it's not.
Analyzing Efficiency

- Suppose we perform an operation (insert, lookup, delete) on element $z$.
- The runtime is proportional to the number of elements in the same bucket as $z$.
- For any $x_k$, let $C_k$ be an indicator variable that is 1 if $x_k$ and $z$ hash to the same bucket (i.e. $h(x_k) = h(z)$) and is 0 otherwise.
- Let random variable $X$ be equal to the number of elements in the same bucket as $z$. Then

$$X = \sum_{x_i \neq z} C_i$$
Analyzing Efficiency

\[ E[X] = E\left[ \sum_{x_i \neq z} C_i \right] \]

\[ = \sum_{x_i \neq z} E[C_i] \]

\[ = \sum_{x_i \neq z} P(h(x_i) = h(z)) \]

\[ = \sum_{x_i \neq z} \frac{1}{m} \]

\[ \leq \frac{n}{m} \]

So the expected cost of an operation is \( O(1 + E[X]) = O(1 + n / m) \)
Analyzing Efficiency

• Assuming we choose a function uniformly at random from all functions, the expected cost of a hash table operation is $O(1 + n / m)$.

• What's the space usage?
  • $O(m)$ space for buckets.
  • $O(n)$ space for elements.
  • Some unknown amount of space to store the hash function.
A Problem

- We assume $h$ is chosen uniformly at random from all functions from $U$ to $\{0, 1, \ldots, m - 1\}$.
- There are $m^{|U|}$ possible functions from $U$ to $\{0, 1, \ldots, m - 1\}$. (Why?)
- How much memory does it take to store $h$?
- If we assign $k$ bits to store $h$, there are $2^k$ possible combinations of those bits.
- We need at least $|U| \log_2 m$ bits to store $h$.
- **Question:** How can we get this performance without the huge space penalty?
Analyzing Efficiency

$$E[X] = E\left[ \sum_{x_i \neq z} C_i \right]$$

$$= \sum_{x_i \neq z} E[C_i]$$

$$= \sum_{x_i \neq z} P(h(x_i) = h(z))$$

$$= \sum_{x_i \neq z} \frac{1}{m}$$

$$\leq \frac{n}{m}$$

So the expected cost of an operation is $O(1 + E[X]) = O(1 + n / m)$
Universal Hash Functions

- A set \( \mathcal{H} \) of hash functions from \( U \) to \( \{0, 1, \ldots, m - 1\} \) is called a **universal family of hash functions** iff
  
  For any \( x, y \in U \) where \( x \neq y \), if \( h \) is drawn uniformly at random from \( \mathcal{H} \), then
  
  \[
  P(h(x) = h(y)) \leq 1 / m
  \]

- In other words, the probability of a collision between two elements is at most \( 1 / m \) as long as we choose \( h \) from \( \mathcal{H} \) uniformly at random.
Universal Hashing

\[ E[X] = E \left[ \sum_{x_i \neq z} C_i \right] \]

\[ = \sum_{x_i \neq z} E[C_i] \]

\[ = \sum_{x_i \neq z} P(h(x_i) = h(z)) \]

\[ \leq \sum_{x_i \neq z} \frac{1}{m} \]

\[ \leq \frac{n}{m} \]

So the expected cost of an operation is

\[ O(1 + E[X]) = O(1 + n / m) \]
Universal Hash Functions

- The set of all possible functions from $U$ to $\{0, 1, ..., m - 1\}$ is a universal family of hash functions.
  - However, requires $\Omega(|U| \log m)$ space.
- For certain types of elements, can find families of universal hash functions we can evaluate in $O(1)$ time and store in $O(1)$ space.
- **The Good News:** The intuitions behind these functions are quite nice.
- **The Bad News:** Formally proving that they're universal requires number theory and/or field theory, which is beyond the scope of this class.
Simple Universal Hash Functions

- We'll start with a simplifying assumption and generalize from there.
- Assume $U = \{0, 1, 2, \ldots, m - 1\}$ and that $m$ is prime. (We'll relax this later.)
- Let $\mathcal{H}$ be the set of all functions of the form
  
  $$h(x) = ax + b \pmod{m}$$
  
  - Where $a, b \in \{0, 1, 2, \ldots, m - 1\}$
  
  **Claim:** $\mathcal{H}$ is universal.
Showing Universality

• We'll show $\mathcal{H}$ is universal by showing it obeys a stronger property called **2-independence**:

For any $x_1, x_2 \in U$ where $x_1 \neq x_2$, if $h$ is chosen uniformly at random from $\mathcal{H}$, then for any $y_1$ and $y_2$ we have

$$P(h(x_1) = y_1 \land h(x_2) = y_2) = \frac{1}{m^2}.$$  

• (The probability that you can guess where any two distinct elements will be hashed is $1 / m^2$).

• **Claim**: Any 2-independent family of hash functions is universal.
\[ h(x) = ax + b \]
Showing Universality

- If $h(x) = ax + b \pmod{m}$, knowing two points on the line determines the entire line.

- Can only guess the output at two points by guessing the coefficients: probability is $1 / m^2$!

- Need to use some more advanced math to formalize why this works; revolves around the fact that $\mathbb{F}_m$ is a finite field.
Generalizing the Result

• This hash function only works if $m$ is prime and $|U| = m$.

• Suppose we can break apart any $x \in U$ into $k$ integer “blocks” $x_1, x_2, \ldots, x_k$, where each block is between 0 and $m - 1$.

• Then the set $\mathcal{H}$ of all hash functions of the form

$$h(x) = a_1x_1 + a_2x_2 + \ldots + a_kx_k + b \pmod{m}$$

is universal.

• Intuitively, after evaluating $k - 1$ of the products, you're left with a linear function in one remaining block and the same argument applies.
A Quick Aside

- Most programming languages associate “a” hash code with each object:
  - Java: `Object.hashCode`
  - Python: `__hash__`
  - C++: `std::hash`
- Unless special care is taken, there always exists the possibility of extensive hash collisions!
Looking Forward

• This is not the only type of hash table; others exist as well:
  
  • **Dynamic perfect hash tables** have worst-case $O(1)$ lookup times and $O(n)$ total storage space, but use a bit more memory.
  
  • Open addressing hash tables avoid chaining and have better locality, but require stronger guarantees on the hash function.
  
• Hash functions have *lots* of applications beyond hash tables; you'll see one in the problem set.
Next Time

- Greedy Algorithms
- Interval Scheduling