Design and Analysis of Algorithms
Randomized Algorithm

Finite Probability Space

Probability space $(\Omega, \mathcal{E}, P)$ is a mathematical construct that provides a formal model of a random process or "experiment"

- A sample space Ω: the set of all possible outcomes.
- An event space $\mathcal{E}$: a set of outcomes in the sample space.
  - each individual element in Ω is called a basic event
- A probability function: which assigns each event in the event space a probability, which is a real number between 0 and 1.

Random variable X is a measurable function $P : \Omega \to S$ from sample space Ω to a measurable space S, i.e., $[0, 1]$.

$\Omega = \{1, 2, 3, 4, 5, 6\}$

Basic event E_i: the die lands on i

$P(E_i) = 1/6$ for $i \in [6]$

Composition of Events

$\bar{E}$: the complement of event E, i.e., E does not occur.

- Definition $\Rightarrow \Pr[\bar{E}] = 1 - \Pr[E]$

$E_1 \wedge E_2$ denotes their conjunction: both E_1 and E_2 occur

- Definition $\Rightarrow \Pr[E_1 \wedge E_2] \leq \min\{\Pr[E_1], \Pr[E_2]\}$
- E_1 and E_2 are independent $\Rightarrow \Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2]$

$E_1 \vee E_2$ denotes their disjunction: either E_1 or E_2 occurs

- Definition $\Rightarrow \Pr[E_1 \vee E_2] \geq \max\{\Pr[E_1], \Pr[E_2]\}$
- E_1 and E_2 are independent $\Rightarrow \Pr[E_1 \vee E_2] = \Pr[E_1] + \Pr[E_2]$

Example. Let E be the event that a random dice throw is even. Clearly, $E = E_1 \vee E_2 \vee E_3$, and $\Pr[E] = 1/6 + 1/6 + 1/6 = 1/2$.

Union Bound. $\Pr[\bigvee_{i=1}^{k} E_i] \leq \sum_{i=1}^{k} \Pr[E_i]$

Expectation

Let X be a random variable. Its expectation (or, expected value) $\mathbb{E}(X)$ is the probability-weighted average of $x \in \Omega$:

$$\mathbb{E}(X) = \sum_{x \in \Omega} x \cdot \Pr[X = x] = \sum_{x \in \Omega} x \cdot P(x)$$

Two useful properties of expectation

When $\Omega = \{0, 1\}$, we refer to X as a 0-1 variable.

$$\mathbb{E}(X) = \sum_{x \in \Omega} x \cdot P(x) = \Pr[X = 1]$$

Linearity. For variables X and Y with arbitrary dependencies:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y), a, b \in \mathbb{N}$$

$$\mathbb{E}(\sum_{i=1}^{k} X_i) = \sum_{i=1}^{k} \mathbb{E}(X_i)$$

expectation of the sum is the sum of the expectation

Applications of Linearity

Benefit. Decouples a complex calculation into simpler pieces.

We then provide two examples of how to use linearity of expectation.

Applications of Linearity

Benefit. Decouples a complex calculation into simpler pieces.

We then provide two examples of how to use linearity of expectation.

Lemma. Let x be a event that happens with probability p. Then on average the experiment needs to be carried out $1/p$ times before it happens.

Applications of Linearity

Benefit. Decouples a complex calculation into simpler pieces.

We then provide two examples of how to use linearity of expectation.

Lemma. Let x be a event that happens with probability p. Then on average the experiment needs to be carried out $1/p$ times before it happens.

Proof 1. Using total probability theorem:

$$N = 1 \cdot p + \cdots + i \cdot (1-p)^{i-1}p + \cdots \Rightarrow N = 1/p$$

Applications of Linearity

Benefit. Decouples a complex calculation into simpler pieces.

We then provide two examples of how to use linearity of expectation.

Lemma. Let x be a event that happens with probability p. Then on average the experiment needs to be carried out $1/p$ times before it happens.

Proof 1. Using total probability theorem:

$$N = 1 \cdot p + \cdots + i \cdot (1-p)^{i-1}p + \cdots \Rightarrow N = 1/p$$

Proof 2. Let N be the expected number of times before it happens. We certainly need at least one shot, and if it happens, we're done. If not (which occurs with probability $1-p$), we need to repeat. Hence: $N = 1 + [p \cdot 0 + (1-p) \cdot N] \Rightarrow N = 1/p$

Guessing Cards (without memory) (1/2)

Game. Shuffle a deck of n cards; turn them over one at a time; try to guess each card.

Memoryless guessing. No psychic abilities; can't even remember what's been turned over already. Guess a card from full deck uniformly at random.

Guessing Cards (without memory) (2/2)

Claim. The expected number of correct guesses is 1.

Proof. [surprisingly effortless using linearity of expectation]

- Let $X_i = 1$ if i-th prediction is correct and 0 otherwise.
- Let $X =$ number of correct guesses $= X_1 + \cdots + X_n$.
- $\mathbb{E}[X_i] = \Pr[X_i = 1] = 1/n$.
- linearity $\Rightarrow$
  $\mathbb{E}[X] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n] = 1/n + \cdots + 1/n = 1$.

Guessing Cards (with memory) (1/2)

Game. Shuffle a deck of n cards; turn them over one at a time; try to guess each card.

Guessing with memory. Guess a card uniformly at random from cards not yet seen.

Guessing Cards (with memory) (2/2)

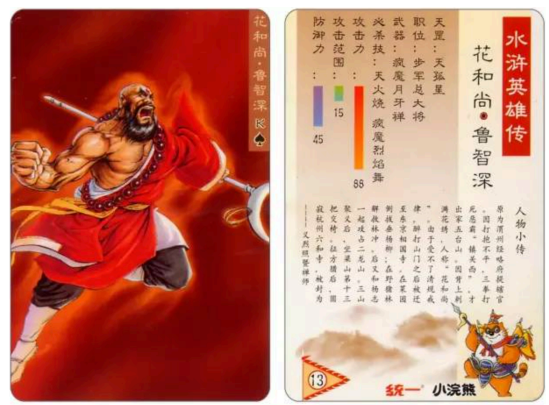
Claim. The expected number of correct guesses is $\Theta(\log n)$.

Proof.

- Let $X_i = 1$ if i-th prediction is correct and 0 otherwise.
- Let $X =$ number of correct guesses $= X_1 + \cdots + X_n$.
- $\mathbb{E}[X_i] = \Pr[X_i = 1] = 1/(n - (i-1))$.
- linearity $\Rightarrow \mathbb{E}[X] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n] = 1/n + \cdots + 1/2 + 1/1 = \mathsf{H}(n) = \Theta(\log n)$.

Coupon Collector (1/3)

Coupon collector. Each box of cereal contains a coupon. There are n different types of coupons. Assuming all boxes are equally likely to contain each coupon, how many boxes before you have ≥ 1 coupon of each type?



Coupon Collector (2/3)

Claim. The expected number of steps is $\Theta(n \log n)$.

Proof.

- Let Phase j = time between j and $j+1$ distinct coupons.
- Let X_j = number of steps you spend in phase j.
- Let X = number of steps in total = $X_0 + X_1 + \cdots + X_{n-1}$.

$$\mathbb{E}(X) = \sum_{j=1}^{n-1} \mathbb{E}(X_j) = \sum_{j=0}^{n-1} \frac{n}{n-j} = n \sum_{i=1}^{n} \frac{1}{i} = n\mathsf{H}(n)$$

probability of success $= (n-j)/n \Rightarrow$ expected steps $= n/(n-j)$

Coupon Collector (3/3)

If the distribution of X_i is not uniform, $\mathbb{E}(X)$ will be higher.

Coupon Collector (3/3)

If the distribution of X_i is not uniform, $\mathbb{E}(X)$ will be higher.

Coupon Collector (3/3)

If the distribution of X_i is not uniform, $\mathbb{E}(X)$ will be higher.

长大之后我才知道，水浒卡是一代人共同的记忆，可能有数千万人卷入其中．据说，小浣熊公司挣到几个亿，用利润建起一栋办公大楼．

—某知乎网友

Markov's Inequality

Markov's inequality. Let X be a non-negative random variable and $v > 0$. Then $\Pr[X \geq v] \leq \mathbb{E}(X)/v$.

Proof. Say X takes values in Ω. We have:

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{x \in \Omega} x \cdot \Pr[X = x] \\
&\geq \sum_{x \in \Omega, x < v} 0 \cdot \Pr[X = s] + \sum_{x \in \Omega, x \geq v} v \cdot \Pr[X = s] \\
&= v \cdot \Pr[X \geq v]
\end{aligned}$$

Markov's inequality is useful when little is known about X.

Chernoff Bounds (above mean) (1/2)

Theorem. Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \cdots + X_n$. Then for any $\mu \geq \mathbb{E}[X]$ and for any $\delta > 0$, we have:

$$\Pr[X > (1+\delta)\mu] \leq \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu}$$

sum of independent 0-1 random variables is tightly centered on the mean

Proof. We apply a number of simple transformations. $\forall t > 0$,

$$\Pr[X > (1+\delta)\mu] = \Pr[e^{tX} > e^{t(1+\delta)\mu}] \leq e^{-t(1+\delta)\mu} \cdot \mathbb{E}(e^{tX})$$

e^{tx} is monotone in x

Markov's inequality: $\Pr[X > a] \leq \mathbb{E}(X)/a$

$$\mathbb{E}(e^{tX}) = \mathbb{E}(e^{t\sum_{i=1} X_i}) = \prod_i \mathbb{E}(e^{tX_i})$$

definition of X

independence among X_i

Chernoff Bounds (above mean) (2/2)

Let $p_i = \Pr[X_i = 1]$. By the fact that $\forall \alpha \geq 0$, $1 + \alpha \leq e^{\alpha}$, we have:

$$\mathbb{E}(e^{tX_i}) = p_i e^t + (1 - p_i)e^0 = 1 + p_i(e^t - 1) \leq e^{p_i(e^t - 1)}$$

Combining everything:

$$\begin{aligned}
\Pr[X > (1+\delta)\mu] &\leq e^{-t(1+\delta)\mu} \prod_i \mathbb{E}(e^{tX_i}) && \text{//previous slide} \\
&\leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t - 1)} && \text{//inequality above} \\
&\leq e^{-t(1+\delta)\mu} e^{\mu(e^t - 1)} && \text{// } \sum_i p_i = \mathbb{E}(X) \leq \mu
\end{aligned}$$

Finally, choose $t = \ln(1 + \delta)$.

Chernoff Bounds (below mean)

Theorem. Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \cdots + X_n$. Then for any $\mu \leq \mathbb{E}(X)$ and for any $0 < \delta < 1$, we have:

$$\Pr[X < (1-\delta)\mu] < e^{\delta^2 \mu/2}$$

Proof idea is similar.

Remark. Not quite symmetric since only makes sense to consider $\delta < 1$.

Motivation: Dictionary Data Type

Dictionary. Given a universe U of possible elements, maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S is efficient.

- $S \subseteq U$ is what we actually care about, which may be static or dynamic. Typically $|S| \ll |U|$, e.g., S is the set of names of students in this class $\ll 128^{80}$.

Dictionary interface.

- create(): initialize a dictionary with $S = \emptyset$.
- insert(u): add element $u \in U$ to S.
- delete(u): delete u from S (if s is currently in S).
- lookup(u): is u in S?

Applications. File systems, databases, compilers, checksums P2P networks, associative arrays, cryptography, web caching, AI search algorithm etc.

Easy Case: Static Dictionary

For static dictionary, we could use a sorted array with binary search for lookups. $|S| \ll |U|$ and will be fixed once for all.

- create $\rightsquigarrow O(|S| \log |S|)$
- lookup $\rightsquigarrow O(\log |S|)$
- no delete and insert

How about dynamic dictionary?

Challenges for Dynamic Dictionary

Universe U can be extremely large, S could be grow increasingly

1. Attempt 1: Use sorted array as static dictionary, complexity of lookup is $O(\log |S|)$, complexity of insert and delete is high $\rightsquigarrow$ $O(|S|)$
2. Attempt 2: Define an array A of size $|U|$, set $A[u] = 1$ if and only if $u \in S$.
   - U may not a subset of $\mathbb{N} \rightsquigarrow$ elements can not be directly used as index
   - U could be so large $\rightsquigarrow$ A will be very large and complexity now depends on $|U|$, not $|S|$

Right data structure for dynamic dictionary

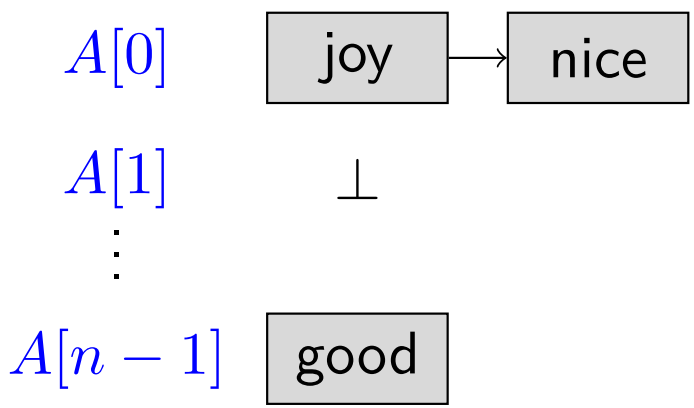
- Balanced search tree.
- Hashing gives an alternative approach: often the fastest and most convenient way

Hashing

Hash function. $h : U \to \{0, 1, \ldots, n-1\}$.

Idea of hashing approach: array + separate chaining

- Create an array $A[n]$, each $A[i]$ stores a linked list of elements u with $h(u) = i$.
- To insert an element u, place u at the top of the list at $A[h(u)]$.
- To perform a lookup of element u, simply compute the index $i = h(u)$, and then walk down the list at $A[i]$ until you find it (or walk off the list).
- To delete, one simply has to perform a delete operation on the associated linked list.



Thinking

The role of hash is two-fold

- shrink a large universe U (could be sparse) to a small universe of size n, define $m = |S|$
- the small universe is a subset of $\mathbb{N} \rightsquigarrow$ can be used as index

One great property of hashing is that all the dictionary operations are incredibly easy to implement.

The question we now turn to is:

What do we need for a hashing scheme to achieve good performance?

Desirable Properties of Good Hash Function

Efficiently computable. h is fast to compute.

- Relatively easy to attain: we will view the time to compute $h(x)$ as a constant.
- Remember in our heads: h shouldn't be too complicated, because that affects the overall runtime.

Few collisions. collision $\leftrightarrow h(u) = h(v)$ but $u \neq v$

- Collisions affect the time to perform lookups and deletes.
- Collision is inevitable after $\Theta(\sqrt{n})$ (birthday paradox) random insertions, but we hope that collisions are few so that keys are nicely spread out.
- Note that insert take time $O(1)$ no matter what the length of the chain, while lookup and delete takes time linear to the length of the chain. Given nicely spread property, the time to lookup an item x is $O(1)$ when $m = O(n)$.

Low-complexity Deterministic Hash Function

We refer to low-complexity and deterministic hash function as ad-hoc hash function.

```
int hash(String s, int n) {
   int hash = 0;
   for (int i = 0; i < s.length(); i++)
      hash = (31 * hash) + s[i];
   return hash % n;
}
```

hash function à la Java string library

Lot of ad-hoc functions that work well in practice for typical sets S, but fails to provide good worst-case guarantee

Bad News about Ad-hoc Hashing

Ad-hoc hash cannot offer any guarantee in adversarial setting if $|U| \geq mn$, where $|S| = m$.

- Pigeon-hole principle + Deterministic $\Rightarrow$ there are at least one bin have m elements
- Low-complexity $\Rightarrow$ easy to invert: the adversary may choose S to be precisely such m elements (the preimage of a bin)
- Consequently, all data keys land in the same bin, making hashing useless $\rightsquigarrow$ lookup, delete complexity is $O(|S|)$

When can't we live with ad-hoc hash function?

Obvious situations: aircraft control, nuclear reactor, pace maker

Surprising situations: denial-of-service attacks.

- malicious adversary learns your ad-hoc hash function (e.g., by reading Java API) and causes a big pile-up in a single slot that grinds performance to a halt

Hashing Performance

Ideal hash function. Maps arbitrary m elements uniformly at random to n hash slots.

- Such requirement guarantees a low number of collisions in expectation, even if the data is chosen by an adversary.

- Running times of all operations depend on length of chains
- Average length of chain $= \alpha = m/n$
- Choose $n \approx m \Rightarrow$ expect $O(1)$ per insert, lookup, or delete

---

Attempt. Pick a random function from all hash function that maps U to $\mathbb{Z}_n$ (truly random). The adversary knows the set of all hash functions, but doesn't know the random choice you make.

Challenge. The set of all functions is too large $\rightsquigarrow$ even not be efficiently sampleable

Universal Hashing (Carter-Wegman 1980s)

Truly random is overkill. Weaker property might be suffice and admits small set of hash functions.

A universal family of hash functions is a set of hash functions H mapping a universe U to the set $\{0, 1, \ldots, n-1\}$ such that:

- For any pair of elements $u \neq v$:

$$\Pr_{h \xleftarrow{R} H}[h(u) = h(v)] \leq 1/n$$

- Can sample random h efficiently.
- Can compute $h(u)$ efficiently.

Intuition. Any two keys of the universe collide with probability at most $1/n$ when $h \xleftarrow{R} H$.

Why not other Definitions?

Alternative definition???

$$\forall u \in U, \forall y \in \mathbb{Z}_n, \Pr_{h \xleftarrow{R} H}[h(u) = y] \leq 1/n$$

Why not other Definitions?

Alternative definition???

$$\forall u \in U, \forall y \in \mathbb{Z}_n, \Pr_{h \xleftarrow{R} H}[h(u) = y] \leq 1/n$$

Issues

- Fail to capture uniform requirement
- Pathological example: easy to build H meeting the above definition, but each $h \in H$ map all inputs to the same value $\rightsquigarrow$ totally useless

Why not other Definitions?

Alternative definition???

$$\forall u \in U, \forall y \in \mathbb{Z}_n, \Pr_{h \xleftarrow{R} H}[h(u) = y] \leq 1/n$$

Issues

- Fail to capture uniform requirement
- Pathological example: easy to build H meeting the above definition, but each $h \in H$ map all inputs to the same value $\rightsquigarrow$ totally useless

---

Why universal hash give us uniform guarantee?

$$(h, h(u)) \approx_s (h, y \xleftarrow{R} \mathbb{Z}_n), \text{where } h \xleftarrow{R} H, u \xleftarrow{R} S$$

- This is exactly what leftover hash lemma tells us: when the input distributes randomly over S (of high min-entropy), output distributes randomly over $\mathbb{Z}_n$.

Example of (Non-)Universal Hashing

$$U = \{a, b, c, d, e, f\},\ n = 2$$

Table: Non-Universal Hashing

H	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

$$H = \{h_1, h_2\}, h \xleftarrow{R} H$$

$$\Pr[h(a) = h(b)] = 1/2$$

$$\Pr[h(a) = h(c)] = 1$$

$$\Pr[h(a) = h(d)] = 0$$

$$\dots$$

Table: Universal Hashing

H	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

$$H = \{h_1, h_2, h_3, h_4\}, h \xleftarrow{R} H$$

$$\Pr[h(a) = h(b)] = 1/2$$

$$\Pr[h(a) = h(c)] = 1/2$$

$$\Pr[h(a) = h(d)] = 1/2$$

$$\Pr[h(a) = h(e)] = 1/2$$

$$\Pr[h(a) = h(f)] = 0$$

$$\dots$$

Key Lemma

Theorem. Let H be a universal family of hash functions mapping a universe U to the set $\mathbb{Z}_n$, let $h \xleftarrow{R} H$; then for any set $S \subseteq U$ of size m, for any $u \in U$ (e.g., that we might want to lookup), the expected number of collisions between u and other elements in S is at most $|S|/n$.

Proof. Let C_u be a random variable counting the total number of collisions with u, induced by $h \xleftarrow{R} H$. For any $s \in S$, define random variable $C_{us} = 1$ if $h(s) = h(u)$ and 0 otherwise.

$$\begin{aligned}
\mathbb{E}[C_u] &= \mathbb{E}[\textstyle\sum_{s\in S} C_{us}] && \text{//definition of } C_u, C_{us} \\
&= \textstyle\sum_{s\in S} \mathbb{E}[C_{us}] && \text{//linearity of expectation} \\
&= \textstyle\sum_{s\in S} \Pr[C_{us} = 1] && \text{//} C_{us} \text{ is 0-1 variable} \\
&\leq \textstyle\sum_{s\in S} \frac{1}{n} = \frac{|S|}{n} && \text{//universal}
\end{aligned}$$

Application to Dictionary Construction

We now immediately get the following corollary.

Corollary. If H is universal then for any element u insert, lookup, and delete operations in which there are at most m elements in the system at any one time, the expected total complexity of operation for a random $h \overset{R}{\leftarrow} H$ is only $O(m/n)$ (viewing the time to compute h as constant).

Proof. According to the above theorem, for any given u, the expected number of collisions is bounded by m/n, so is the average length of chain $\Rightarrow$ the expected complexity of all operations is bounded by $O(m/n)$.

Question: can we actually construct a universal H?

The answer is affirmative.

Many universal families are known (for hashing integers, vectors, strings), their evaluation is often very efficient, and their security is unconditional (do not rely on any assumptions).

Universal hashing has numerous uses in computer science,

- implementations of hash tables
- randomized algorithms
- cryptography

Designing a universal family of hash functions

Modulus. We will use a prime number p for the size of the hash table.

Input encoding. Assume $U \subseteq [0, p^\ell - 1]$ for some integer ℓ. We can identify each element $u \in U$ with a base$-p$ integer of ℓ digits: $u = \mathbf{x} = (x_1, x_2, \ldots, x_\ell)$. Alternatively, we view each element u as a ℓ-dimension vector over $\mathbb{Z}_p$

Hash function family. Let $\mathbf{a} = (a_1, a_2, \ldots, a_\ell)$ be function index or key, let $A = \mathbb{Z}_p^\ell$. Define $H = \{h_\mathbf{a} : \mathbf{a} \in A\}$, where $h_\mathbf{a}$ is defined as:

$$h_\mathbf{a}(\mathbf{x}) = \langle \mathbf{a}, \mathbf{x} \rangle \bmod p = \left(\sum_{i=1}^{\ell} a_i x_i \right) \bmod p$$

maps universe U to set $\{0, 1, \ldots, p-1\}$

Proof of Construction

Theorem. $H = \{h_{\mathbf{a}} : \mathbf{a} \in A\}$ is a family of UHF.

Proof. Let $\mathbf{x} = (x_1, x_2, \ldots, x_\ell)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_\ell)$ be two distinct elements of U. Show that $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] \leq 1/p$, where $\mathbf{a} \xleftarrow{R} \mathbb{Z}_p^\ell$.

Since $\mathbf{x} \neq \mathbf{y}$, there exists at least one index j such that $x_j \neq y_j$.

We have $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$ iff

$$a_j \underbrace{(y_j - x_j)}_{z} \equiv \underbrace{\sum_{i \neq j} a_i (x_i - y_i)}_{m} \pmod{p}$$

- Assume $\mathbf{a}$ was chosen uniformly at random by first selecting all a_i where $i \neq j$, then selecting a_j at random.
- Since p is prime, $a_j z \equiv m \bmod p$ has exactly one solution among p possibilities $\Leftarrow z \neq 0$ is invertible in group $\mathbb{Z}_p^*$, $\mathbb{Z}_p$ does not have zero divisor.
- Thus $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] \leq 1/p$.

Universal hashing: Summary

Goal. Given a universe U, maintain a subset $S \subseteq U$ so that insert, delete and lookup are efficient.

Use universal hashing to build hash map, choose p so that $m \leq p \leq 2m$, where $m = |S|$

- Fact: there exists at least one prime between m and $2m$, can be found by another randomized algorithm.

Consequence

- Space used: $\Theta(m)$.
- Expected number of collisions per operation is $\leq 1 \Rightarrow O(1)$ time per insert, delete, or loop up

Discussion

Q. Why not directly using cryptographic hash functions?

- not randomized in nature $\rightsquigarrow$ any form of uniformity may rely on assumptions
- cryptographic hash functions are heavy to use
- output length is typically at least 128 bits $\rightsquigarrow$ might be too large for applications; truncating output may compromise collision resistance

Q. Why not using pseudorandom functions?

- full-fledged pseudorandomness is overkill for uniform hashing purpose
- too strong $\rightsquigarrow$ heavy to use

Summary

UHF does not capture any form of independence, pairwise independent hash does, capturing mutual independence.

UHF only captures a weak form of collision resistance

- The adversary must choose the challenge before seeing h. This is in sharp contrast to CRHF in which the adversary chooses u and v after seeing h.
- Bazinga: we can compile UHF to CRHF using lossy functions
- Such weak form of collision resistance implies weak uniformity — define over any set (say, S) fixed before seeing h, but suffices for application of dictionary

1. Preliminaries on Probability Theory
2. Randomized Data Structure
   - Dictionary
3. Randomized Algorithm
   - Randomized Quicksort and Quickselect
   - Probabilistic Primality Testing
   - Schwartz-Zippel Lemma and Applications
     - Polynomial Identity Test
     - Matrix Identity Test
4. Summary

Randomization

Algorithmic design patterns

- Greedy
- Divide-and-conquer
- Dynamic programming
- Randomization

Randomization. Allow fair coin tossing

- In practice, access to a PRG

The benefit of randomize: Can lead to simplest, fast, or only known algorithm for a particular problem.

Examples. Graph algorithms, quicksort, quickselect, hashing, cryptography

Two-sided error.

- $\mathcal{BPP}$. [Monte Carlo] problems solvable with two-sided error in poly-time. (name after the famous gambling resort)

One-sided error.

- $\mathcal{RP}$. If the correct answer is yes, return yes with probability $\geq 2/3$. If the correct answer is no, always return no.
- co-$\mathcal{RP}$. If the correct answer is no, return no with probability $\geq 2/3$. If the correct answer is yes, always return yes.

can decrease error probability to negligible

Zero-sided error.

- $\mathcal{ZPP}$. [Las Vegas] problems solvable in expected poly-time.

Relation to Other Complexity Classes

Theorem. $\mathcal{P} \subseteq \mathcal{ZPP} = \mathcal{RP} \cap \text{co-}\mathcal{RP} \subseteq \mathcal{BPP} \subseteq \mathcal{NP}$

Central open questions

- Does $\mathcal{P} = \mathcal{BPP}$? This question concerns to what extent does randomization help.
  - Many complexity theorists acturally believe that $\mathcal{P} = \mathcal{BPP}$.
  - In other words, there is a way to transform *every* probabilistic algorithm to a deterministic algorithm (one does not toss any coins) while incurring only a polynomial slowdown.

Role of randomness extends far beyond a study of randomized algorithms and classes such as $\mathcal{BPP}$.

- entire area of cryptography, including encryption, signature, interactive and probabilistically checkable proofs rely on randomness in an essential way, sometimes to prove results seemingly did not involve randomness in any way

Randomized Quicksort

Standard Quicksort. Always pick the first element as pivot. The consequence is that the worst-case complexity (when the input array is already sorted) is $O(n^2)$

$$W(n) = W(1) + W(n-1) + O(n) \Rightarrow W(n) = O(n^2)$$

Randomized Quicksort. Each time randomly select an element as pivot. Clearly, the probability of selecting the i-th element is $1/n$

$$\begin{aligned} T(n) &= \frac{1}{n}\left(\sum_{i=1}^{n}(T(i)+T(n-i))\right) + (n-1) \\ &= \frac{2}{n}\sum_{i=1}^{n} T(i) + (n-1) \\ &= \Theta(n \ln n) \end{aligned}$$

Randomized Quickselect (1/2)

Standard Quickselect. First divide all the elements into grpups of size 5, selects their medians, and then selects median m^* from these medians, using m^* as pivot to split the array.

$$W(n) = W(7n/10) + W(n/5) + O(n) \Rightarrow W(n) = \Theta(n)$$

Randomized Quickselect. Randomly select an element as the pivot.

- Best-case: keep picking the median, yield ideal situation being $|S_L|, |S_R| \approx |S|/2 \rightsquigarrow T(n) = T(n/2) + O(n)$.
- Worst-case: we keep picking m^* to be the largest (or the smallest) element, and thereby shrink the array by only one element each time $\rightsquigarrow T(n) = \Theta(n^2)$.

Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the average running time lie? Fortunately, it lies very close to the best-case time.

Randomized Quickselect (2/2)

The running time depends on the random choices of m^*.

- To distinguish between lucky and unlucky choices of m^*, we will call m^* good if it lies within 25-th to 75-th percentile of the array. We like these choices of m^* because they ensure that the sublists S_L and S_R have size of at most $3/4$ that of S, so that the array shrinks substantially.
- Good m^*'s are abundant: half the elements of any list must fall between the 25th to 75th percentile.

Randomly pick and check: on average two try we will obtain a good m^*:

$$T(n) \leq T(3n/4) + O(n) \Rightarrow T(n) = O(n)$$

- Expected polynomial time. On any input, the randomized algorithm returns the correct answer after a linear number of steps, on the average.

Primality Testing

Primality testing. Given an integer N and wish to determine whether or not it is prime.

Algorithms for primality testing were sought after even before the advent of computers, as mathematician needed them to test various conjectures.

- Ideally, we want efficient algorithms that run in time polynomial in the size of N' representation, i.e., poly$(\log N)$.
- For centuries mathematicians knew of no such efficient algorithm for this problem.
- In 1970's efficient probabilistic algorithms for primality testing were discovered, give one of the first demonstration of the power of probabilistic algorithms.
- At 2004, Agrawal, Kayal, and Saxena gave a deterministic polynomial-time algorithm for primality testing (breakthrough)

Miller-Rabin Primality Test

The key to the Miller-Rabin algorithm is to find a property that distinguishes primes and composites.

Fact 1: If N is prime, then $\forall a \in \{1, \ldots, N-1\}$ we have $a^{N-1} = 1 \bmod N$

This suggests testing whether N is prime by choosing a uniform element a and checking whether $a^{N-1} \stackrel{?}{=} 1 \bmod N$.

- If $a^{N-1} \neq 1 \bmod N$, then N cannot be prime. We refer to any such a a witness that N is composite.
- Conversely, we might hope that if N is not prime then there is a reasonable chance that a randomly chosen a is a witness for composite, and so by repeating this test many times we can determine whether N is prime or not with high confidence.

The Distribution of Witness

Recall that exponentiation modulo N and choosing a uniform element of $\{1, \ldots, N-1\}$ can also be done in polynomial time. It seems that the problem has been solved.

If N is a composite, how many witnesses in $\{1, \ldots, N-1\}$.

- If $a \notin \mathbb{Z}_N^*$, then a is a witness. But, witnesses outside $\mathbb{Z}_N^*$ are few. Actually, if you find one $a \notin \mathbb{Z}_N^*$, you can factor N.
- We then investigate witness inside $\mathbb{Z}_N^*$.

By two simple group-theoretic lemmas, we conclude that:

Fact 2. Fix N. Say there exists one witness that N is composite. Then at least half the elements of $\mathbb{Z}_N^*$ are witnesses that N is composite.

However, the above does not give a complete solution since there are infinitely many composite numbers N that do not have any witnesses. Such N are known as *Carmichael numbers*.

Strong Witness: A Refinement of the above Test

Let $N-1 = 2^r u$, where u is odd and $r > 1$. The algorithm shown previously tests only whether $a^{N-1} = a^{2^r u} = 1 \bmod N$.

A more refined algorithm looks at the sequence of $r+1$ values:

$$a^u, a^{2u}, \ldots, a^{2^r u} \text{ all modulo } N$$

Each term in this sequence is the square of the preceding term. If some value is equal to ± 1 then all subsequent values will be equal to 1.

Strong witness. $a \in \mathbb{Z}_N^*$ is a strong witness if

1. $a^u \neq \pm 1 \bmod N$
2. $a^{2^i u} \neq -1 \bmod N$ for all $i \in \{1, \ldots, r-1\}$

Wrapping all it Together

Theorem. Let N be an odd number that is the not a prime power. Then at least half the elements of $\mathbb{Z}_N^*$ are strong witnesses that N is composite.

Algorithm 1: Miller-Rabin primality test

Input: $N > 2$

1: **if** N *is even* **then return** "composite";
2: **if** N *is a perfect power* **then return** "composite";
3: compute $r > 1$ and odd u such that $N - 1 = 2^r u$;
4: **for** $j = 1$ *to* t **do**
5: $\quad a \xleftarrow{R} \{1, \ldots, N-1\}$;
6: $\quad$ **if** a *is strong witness* **then return** "composite";
7: **end**
8: **return** "prime"

$$N \in \text{PRIME} \Rightarrow \Pr[M(N) = 1] = 1$$
$$N \notin \text{PRIME} \Rightarrow \Pr[M(N) = 0] \geq 1 - 2^{-t}$$

Schwartz-Zippel Lemma

DeMillo-Lipton-Schwartz-Zippel Lemma. Let $P \in \mathbb{F}[x_1, \ldots, x_n]$ be a non-zero polynomial of total degree $d \geq 0$ over a field $\mathbb{F}$ and let $\alpha_1, \ldots, \alpha_n$ be selected independently at random from $S \subset \mathbb{F}$. Then,

$$\Pr[P(\alpha_1, \ldots, \alpha_n) = 0] \leq \frac{d}{|S|}$$

In the single variable case, it follows directly from the fact that a polynomial of degree d can have no more than d roots.

Application of Schwartz-Zippel lemma

- probabilistic polynomial identity testing
- construction of universal hash
- primality testing

Comparison of Two Polynomials

Polynomial identity test: Given two polynomials $C(x)$ and $D(x)$ over a finite field $\mathbb{F}$. decide if they are equal.

Applications: branching programs, probabilistic checkable proofs

Why this problem is not trivial?

It seems that we can check them one by one $\Rightarrow$ complexity is $O(n)$, where $n = \max\{\deg(C), \deg(D)\}$.

Refinement of Problem

The above check only works when the coefficents are given in plain. We need to consider the following two cases.

Case 1: the representation of polynomial is complicated, the unrolling could be way too expensive

- Example 1. $D(x)$ is of the form $A(x) \cdot B(x)$, to figure out the coefficients of $D(x)$ needs polynomial multiplication
  - Naive algorithm: computing coefficients via convolution $O(n^2)$
  - FFT: $O(n \log n)$
- Example 2. $D(x) = \Pi_{i=1}^{n}(x+i)$, complexity of naive unrolling is $O(n^2)$

Case 2: the representation of polynomial is hidden (probably for privacy concerns), the algorithm is only given access to an evaluation oracle

Can we solve the problem without naive comparing method?

A. Yes. Randomness can help!

Polynomial Identity Test

Polynomial identity test [surprisingly simple]

- Randomly picks $\alpha \xleftarrow{R} \mathbb{F}$, output "yes" if $C(\alpha) = A(\alpha) \cdot B(\alpha)$ and "no" otherwise.

Analysis

- Complexity: the cost of polynomial evaluation is $3 \cdot O(n)$, the cost of comparison is $O(1) \rightsquigarrow$ total complexity is $O(n)$
- When $C(x) = A(x) \cdot B(x)$: algorithm's output is always "yes"
- When $C(x) \neq A(x) \cdot B(x)$: algorithm's output is "no" with probability at least $1 - n/|\mathbb{F}| \Leftarrow$ Schwartz-Zippel lemma

Matrix Identity Test

Problem. Given three $n \times n$ matrices A, B and C over $\mathbb{F}_p$, where $p > n^2$ is a prime number. Check if $A \times B = C$.

Naive solution. Compute $A \times B$ then compare the result with C. The complexity is reduced to matrix multiplication, that is, $O(n^{2.372})$.

Idea. Use randomized test. The main tool is universal hash. Intuitively, it can compress a big object to small fingerprint.

But how? Let us first revisit the construction of universal hash.

Universal Hash, Revisited

Our goal: build a family of universal hash from $\mathbb{F}_p^n \to \mathbb{F}_p$.

Previous construction:

$$h_{\mathbf{a}}(\mathbf{x}) = a_0 x_0 + \cdots + a_{n-1} x_{n-1}, \mathbf{a}, \mathbf{x} \in \mathbb{F}_p^n$$

- By interpreting $\mathbf{x}$ as coefficient vector and $\mathbf{a}$ as the variable vector, Schwartz-Zippel lemma (multi-variable setting) $\Rightarrow$ collision probability is smaller than $n/|\mathbb{F}_p|$; the lower bound is $1/|\mathbb{F}_p|$.

A simpler construction (this is exactly the Reed-Solomon code)

$$h_a(x) = a^1 x_0 + \cdots + a^n x_{n-1}, a \in \mathbb{F}_p, \mathbf{x} \in \mathbb{F}_p^n$$

- By interpreting $\mathbf{x}$ as coefficient vector and a the single variable, Schwartz-Zippel lemma (single-variable setting) $\Rightarrow$ collision probability is smaller than $n/|\mathbb{F}_p|$.

Freivalds' Algorithm

[Freivalds' Algorithm 1977] Randomized algorithm in time $O(n^2)$

1. Pick $r \xleftarrow{R} \mathbb{F}_p$, set $\mathbf{x} = (r, r^2, \ldots, r^n)$
2. Compute $\mathbf{y} = C\mathbf{x}$ and $\mathbf{z} = A \cdot B\mathbf{x}$
3. Output "true" if $\mathbf{y} = \mathbf{z}$ and "false" otherwise.

Complexity analysis: totally $O(n^2)$

- generating vector $\mathbf{x}$: $O(n)$
- matrix-vector product: $3 \times O(n^2)$

Correctness analysis

- When $C = A \cdot B$: algorithm's output is always "yes"
- When $C \neq A \cdot B$: algorithm's output is "no" with probability at least $1 - n/|\mathbb{F}_p| \Leftarrow$ Schwartz-Zippel lemma
  - at least exists one (i, j) such that $C_{ij} \neq D_{ij}$, then consider the difference of row vector as coefficient vector

High-Level View

The polynomial identity test and matrix product test can be generalized to verifiable computation via probabilistic checkable technique

The power of randomness

Verify the correctness of computation via randomized checking rather than re-executing

High-level view: view the randomized algorithm as an interactive proof system

$$A(x) \cdot B(x) = C(x)$$

$$P \qquad \xleftarrow{\alpha \overset{\mathsf{R}}{\leftarrow} \mathbb{F}} \qquad V$$

$$\xrightarrow{A(\alpha), B(\alpha), C(\alpha)} \quad \text{check } A(\alpha)B(\alpha) \overset{?}{=} C(\alpha)$$

1. Preliminaries on Probability Theory
2. Randomized Data Structure
   - Dictionary
3. Randomized Algorithm
   - Randomized Quicksort and Quickselect
   - Probabilistic Primality Testing
   - Schwartz-Zippel Lemma and Applications
     - Polynomial Identity Test
     - Matrix Identity Test
4. Summary

Monte Carlo vs. Las Vegas algorithms

Basic probability theory

Randomized data structure from universal hash

Randomized algorithms

1. Las Vegas $\leftrightarrow \mathcal{ZPP}$: Guaranteed to find correct answer, likely to run in poly-time (acturally expected poly-time).
   - Examples: randomized Quicksort and Quickselect
2. Monte Carlo $\leftrightarrow \mathcal{BPP}$: Guaranteed to run in poly-time, likely to find correct answer.
   - Examples: polynomial identity test, matrix identity test, primality test

Remark. Can always convert a Las Vegas algorithm into Monte Carlo, but no known method (in general) to convert the other way.