Example 1: Routing

What is the best way to route a packet from $X$ to $Y$, esp. in high speed, high volume networks

A: Pick the shortest path from $X$ to $Y$

B: Send the packet to a random node $Z$, and let $Z$ route it to $Y$ (possibly using a shortest path from $Z$ to $Y$)
Example 1: Routing

- What is the best way to route a packet from $X$ to $Y$, esp. in high speed, high volume networks
  
  **A:** Pick the shortest path from $X$ to $Y$
  
  **B:** Send the packet to a random node $Z$, and let $Z$ route it to $Y$ (possibly using a shortest path from $Z$ to $Y$)

- Valiant showed in 1981 that surprisingly, B works better!
  
  - Turing award recipient in 2010
Example 2: Transmitting on shared network

- What is the best way for $n$ hosts to share a common network?
  
  A: Give each host a turn to transmit
  
  B: Maintain a queue of hosts that have something to transmit, and use a FIFO algorithm to grant access
  
  C: Let every one try to transmit. If there is contention, use random choice to resolve it.

- Which choice is better?
Topics

1. Intro
2. Probability Basics
   - Discrete Probability
   - Coupon Collection
   - Birthday
   - Balls and Bins
3. Taming distribution
   - Quicksort
4. Probabilistic Algorithms
   - Caching
   - Hashing
   - Universal/Perfect hash
   - Closest pair
   - Bloom filter
   - Rabin-Karp
   - Prime testing
Simplify, Decentralize, Ensure Fairness

Randomization can often:
- Enable the use of a simpler algorithm
- Cut down the amount of book-keeping
- Support decentralized decision-making
- Ensure fairness

Examples:
- Media access protocol: Avoids need for coordination — important here, because coordination needs connectivity!
- Load balancing: Instead of maintaining centralized information about processor loads, dispatch jobs randomly.
- Congestion avoidance: Similar to load balancing
Set Theory and Probability

- A countable *sample space* $S$ is a nonempty countable set.
- An *outcome* $\omega$ is an element of $S$.
- A *probability function* $Pr : S \rightarrow \mathbb{R}$ is a total function such that
  - $Pr[\omega] \geq 0$ for all $\omega \in S$, and
  - $\sum_{\omega \in S} Pr[\omega] = 1$
Set Theory and Probability

- A countable *sample space* $S$ is a nonempty countable set.
- An *outcome* $\omega$ is an element of $S$.
- A *probability function* $Pr : S \rightarrow \mathbb{R}$ is a total function such that
  - $Pr[\omega] \geq 0$ for all $\omega \in S$, and
  - $\sum_{\omega \in S} Pr[\omega] = 1$
- An *event* $E$ is a subset of $S$. Its probability is given by:
  $$Pr[E] = \sum_{\omega \in E} Pr[\omega]$$
Probability Rules from Set Theory

Many probability rules follow from the rules on set cardinality

**Sum Rule:** If $E_0, E_1, \ldots, E_n, \ldots$ are pairwise disjoint events, then

$$Pr[\bigcup_{n \in \mathbb{N}} E_n] = \sum_{n \in \mathbb{N}} Pr[E_n]$$
Probability Rules from Set Theory

Many probability rules follow from the rules on set cardinality

**Sum Rule:** If $E_0, E_1, \ldots, E_n, \ldots$ are pairwise disjoint events, then
\[
Pr[\bigcup_{n \in \mathbb{N}} E_n] = \sum_{n \in \mathbb{N}} Pr[E_n]
\]

**Complement Rule:** $Pr[\overline{A}] = 1 - Pr[A]$
Many probability rules follow from the rules on set cardinality

**Sum Rule:** If $E_0, E_1, \ldots, E_n, \ldots$ are pairwise disjoint events, then

$$Pr[\bigcup_{n \in \mathbb{N}} E_n] = \sum_{n \in \mathbb{N}} Pr[E_n]$$

**Complement Rule:** $Pr[\overline{A}] = 1 - Pr[A]$

**Difference Rule:**

$$Pr[B - A] = Pr[B] - Pr[A \cap B]$$
Probability Rules from Set Theory

Many probability rules follow from the rules on set cardinality

**Sum Rule**: If $E_0, E_1, \ldots, E_n, \ldots$ are pairwise disjoint events, then

$$Pr[\bigcup_{n \in \mathbb{N}} E_n] = \sum_{n \in \mathbb{N}} Pr[E_n]$$

**Complement Rule**: $Pr[\overline{A}] = 1 - Pr[A]$ 

**Difference Rule**: 

$$Pr[B - A] = Pr[B] - Pr[A \cap B]$$

**Inclusion–Exclusion**: 

$$Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$$

**Union Bound**: $Pr[A \cup B] \leq Pr[A] + Pr[B]$
Probability Rules from Set Theory

Many probability rules follow from the rules on set cardinality

**Sum Rule:** If $E_0, E_1, \ldots, E_n, \ldots$ are pairwise disjoint events, then

$$\Pr[\bigcup_{n \in \mathbb{N}} E_n] = \sum_{n \in \mathbb{N}} \Pr[E_n]$$

**Complement Rule:** $\Pr[A] = 1 - \Pr[A]$

**Difference Rule:**

$$\Pr[B - A] = \Pr[B] - \Pr[A \cap B]$$

**Inclusion–Exclusion:**

$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$

**Union Bound:** $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$

**Monotonicity:** $A \subseteq B \rightarrow \Pr[A] \leq \Pr[B]$
Uniform Probability Spaces

A finite probability space $S$ said to be uniform if $\Pr[\omega]$ is the same for all $\omega$. In such spaces:

$$\Pr[E] = \frac{|E|}{|S|}$$

We often this assumption.
Conditional Probability

- Probability of an event under a condition
- The condition limits consideration to a subset of outcomes
  - Consider this subset (rather than whole of \( S \)) as the space of all possible outcomes

\[
Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]}
\]
Extending Probability Rules for Conditional Probability

Product Rule 2: \( Pr[E_1 \cap E_2] = Pr[E_1] \cdot Pr[E_2|E_1] \)
**Extending Probability Rules for Conditional Probability**

**Product Rule 2:** \( \Pr[E_1 \cap E_2] = \Pr[E_1] \cdot \Pr[E_2|E_1] \)

**Product Rule 3:** \( \Pr[E_1 \cap E_2 \cap E_3] = \Pr[E_1] \cdot \Pr[E_2|E_1] \cdot \Pr[E_3|E_1 \cap E_2] \)
Extending Probability Rules for Conditional Probability

Product Rule 2: \( \Pr[E_1 \cap E_2] = \Pr[E_1] \cdot \Pr[E_2|E_1] \)

Product Rule 3: \( \Pr[E_1 \cap E_2 \cap E_3] = \Pr[E_1] \cdot \Pr[E_2|E_1] \cdot \Pr[E_3|E_1 \cap E_2] \)

Bayes’ Rule: \( \Pr[B|A] = \frac{\Pr[A|B] \cdot \Pr[B]}{\Pr[A]} \)
Extending Probability Rules for Conditional Probability

**Product Rule 2:** \( Pr[E_1 \cap E_2] = Pr[E_1] \cdot Pr[E_2|E_1] \)

**Product Rule 3:** \( Pr[E_1 \cap E_2 \cap E_3] = Pr[E_1] \cdot Pr[E_2|E_1] \cdot Pr[E_3|E_1 \cap E_2] \)

**Bayes’ Rule:** \( Pr[B|A] = \frac{Pr[A|B] \cdot Pr[B]}{Pr[A]} \)

**Total Probability Law:** \( Pr[A] = Pr[A|E] \cdot Pr[E] + Pr[A|\bar{E}] \cdot Pr[\bar{E}] \)

**Total Probability Law 2:** If \( E_i \) are mutually disjoint and \( Pr[\bigcup E_i] = 1 \) then
\[
Pr[A] = \sum Pr[A|E_i] \cdot Pr[E_i]
\]

**Inclusion-Exclusion:** \( Pr[A \cup B|C] = Pr[A|C] + Pr[B|C] - Pr[A \cap B|C] \)
Independence

An event $A$ is independent of $B$ iff the following (equivalent) conditions hold:

- $P_r[A|B] = P_r[A]$
- $P_r[A \cap B] = P_r[A] \cdot P_r[B]$
- $B$ is independent of $A$

Often, independence is an assumption.

Definition can be generalized to $3$ (or $n$) events. Events $E_1$, $E_2$ and $E_3$ are mutually independent iff all of the following hold:

- $P_r[E_1 \cap E_2] = P_r[E_1] \cdot P_r[E_2]$
- $P_r[E_2 \cap E_3] = P_r[E_2] \cdot P_r[E_3]$
- $P_r[E_1 \cap E_3] = P_r[E_1] \cdot P_r[E_3]$
- $P_r[E_1 \cap E_2 \cap E_3] = P_r[E_1] \cdot P_r[E_2] \cdot P_r[E_3]$
Coupon Collector Problem

Suppose that your favorite cereal has a coupon inside. There are $n$ types of coupons, but only one of them in each box. How many boxes will you have to buy before you can expect to have all of the $n$ types?

What is your guess?
Coupon Collector Problem

Suppose that your favorite cereal has a coupon inside. There are $n$ types of coupons, but only one of them in each box. How many boxes will you have to buy before you can expect to have all of the $n$ types?

What is your guess?

Let us work out the expectation. Let us say that you have so far $j - 1$ types of coupons, and are now looking to get to the $j$th type. Let $X_j$ denote the number of boxes you need to purchase before you get the $j + 1$th type.
Coupon Collector Problem

- Note $E[X_j] = 1/p_j$, where $p_j$ is the probability of getting the $jth$ coupon.
Coupon Collector Problem

- Note $E[X_j] = 1/p_j$, where $p_j$ is the probability of getting the $j$th coupon.
- Note $p_j = (n - j)/n$, so $E[X_j] = n/(n - j)$
**Coupon Collector Problem**

- Note $E[X_j] = 1/p_j$, where $p_j$ is the probability of getting the $j$th coupon.

- Note $p_j = (n - j)/n$, so $E[X_j] = n/(n - j)$

- We have all $n$ types when we finish the $X_{n-1}$ phase:

$$E[X] = \sum_{i=0}^{n-1} E[X_j] = \sum_{i=0}^{n-1} n/(n - j) = nH(n)$$

- Note $H(n)$ is the harmonic sum, and is bounded by $\ln n$
Coupon Collector Problem

- Note $E[X_j] = 1/p_j$, where $p_j$ is the probability of getting the $j$th coupon.
- Note $p_j = (n - j)/n$, so, $E[X_j] = n/(n - j)$
- We have all $n$ types when we finish the $X_{n-1}$ phase:

$$E[X] = \sum_{i=0}^{n-1} E[X_j] = \sum_{i=0}^{n-1} n/(n - j) = nH(n)$$

- Note $H(n)$ is the harmonic sum, and is bounded by $\ln n$
- Perhaps unintuitively, you need to buy $\ln n$ cereal boxes to obtain one useful coupon.
Birthday Paradox

What is the smallest size group where there are at least two people with the same birthday?

- 365
- 183
- 61
- 25
Birthday Problem

The probability that two students have different birthdays: \( \frac{364}{365} \)

In a class of \( n \), there are \( \binom{n}{2} \) pairs of students to consider.

If we assume that whether one pair shares a birthday is independent of another, we can simply multiply these probabilities

\[
Pr(\text{no two persons with same birthday}) \approx \left( \frac{364}{365} \right)^{\binom{n}{2}} \approx \left( \frac{364}{365} \right)^{n^2/2}
\]

For \( n = 44 \), this formula yields a probability of 7%

\( n = 23 \) is enough to have better than even chance of finding two with the same birthday.
Birthday Problem: More Accurate Approach

What is the probability of finding two people with the same birthday in this class?

There are $365^n$ possible sequences of birthdays for $n$ people

We assume these are all equally likely

Number of sequences without repetition: $365 \cdot 364 \cdots (365 - (n - 1))$

Probability that no two of $n$ persons have same birthday:

$$\frac{365 \cdot 365 - 1}{365} \cdots \frac{365 - (n - 1)}{365} = \left( 1 - \frac{0}{365} \right) \left( 1 - \frac{1}{365} \right) \cdots \left( 1 - \frac{n - 1}{365} \right)$$

Use the approximation $(1 - x) < e^{-x}$ to derive an upper bound:

$$Pr(\text{no two persons with same birthday}) < e^0 \cdot e^{\frac{-1}{365}} \cdot e^{\frac{-n-1}{365}} = e^{\frac{-1}{365} \sum_{i=1}^{n-1} i} = e^{\frac{-n(n-1)}{2 \cdot 365}}$$

For $n = 44$, this evaluates to 7.5%
Birthday Paradox Vs Coupon Collection

- Two sides of the same problem

**Coupon Collection**: What is the minimum number of samples needed to cover every one of $N$ values

**Birthday problem**: What is the maximum number of samples that can avoid covering any value more than once?
Birthday Paradox Vs Coupon Collection

- Two sides of the same problem
  - **Coupon Collection**: What is the minimum number of samples needed to cover every one of $N$ values
  - **Birthday problem**: What is the maximum number of samples that can avoid covering any value more than once?

- So, if we want enough people to ensure that every day of the year is covered as a birthday, we will need $365 \ln 365 \approx 2153$ people!
  - Almost 100 times as many as needed for one duplicate birthday!
Balls and Bins

If $m$ balls are thrown at random into $n$ bins:

- What should $m$ be to have more than one ball in some bin?
Balls and Bins

If $m$ balls are thrown at random into $n$ bins:

- What should $m$ be to have more than one ball in some bin?
- Birthday problem
If $m$ balls are thrown at random into $n$ bins:

- What should $m$ be to have more than one ball in some bin?  
  - Birthday problem

- What should $m$ be to have at least one ball per bin?
Balls and Bins

If $m$ balls are thrown at random into $n$ bins:

- What should $m$ be to have more than one ball in some bin? 
  - Birthday problem

- What should $m$ be to have at least one ball per bin? 
  - Coupon collection
Balls and Bins

If $m$ balls are thrown at random into $n$ bins:

- What should $m$ be to have more than one ball in some bin?
  - Birthday problem

- What should $m$ be to have at least one ball per bin?
  - Coupon collection

- What is the maximum number of balls in any bin?
  - Such problems arise in load-balancing, hashing, etc.
Balls and Bins: Max Occupancy

- Probability \( p_{1,k} \) that the first bin receives at least \( k \) balls:
  - Choose \( k \) balls in \( \binom{m}{k} \) ways
  - These \( k \) balls should fall into the first bin: prob. is \( (1/n)^k \)
  - Other balls may fall anywhere, i.e., probability 1:\(^1\)

\[
\binom{m}{k} \left( \frac{1}{n} \right)^k = \frac{m \cdot (m-1) \cdots (m-k+1)}{k!n^k} \leq \frac{m^k}{k!n^k}
\]

---

\(^1\)This is actually an upper bound, as there can be some double counting.
Balls and Bins: Max Occupancy

- Probability $p_{1,k}$ that the first bin receives at least $k$ balls:
  - Choose $k$ balls in $\binom{m}{k}$ ways
  - These $k$ balls should fall into the first bin: prob. is $(1/n)^k$
  - Other balls may fall anywhere, i.e., probability 1:

  $\binom{m}{k} \left( \frac{1}{n} \right)^k = \frac{m \cdot (m-1) \cdots (m-k+1)}{k! n^k} \leq \frac{m^k}{k! n^k}$

- Let $m = n$, and use Sterling’s approx. $k! \approx \sqrt{2\pi k} (k/e)^k$:

  $P_k = \sum_{i=1}^{n} p_{i,k} \leq n \cdot \frac{1}{k!} \leq n \cdot \left( \frac{e}{k} \right)^k$

\[\text{\footnotesize \textsuperscript{1}This is actually an upper bound, as there can be some double counting.}\]
Balls and Bins: Max Occupancy

- Probability $p_{1,k}$ that the first bin receives at least $k$ balls:
  - Choose $k$ balls in $\binom{m}{k}$ ways
  - These $k$ balls should fall into the first bin: prob. is $(1/n)^k$
  - Other balls may fall anywhere, i.e., probability 1:
    \[
    \binom{m}{k} \left(\frac{1}{n}\right)^k = \frac{m \cdot (m-1) \cdots (m-k+1)}{k! n^k} \leq \frac{m^k}{k! n^k}
    \]

  - Let $m = n$, and use Sterling’s approx. $k! \approx \sqrt{2\pi k} (k/e)^k$:
    \[
    P_k = \sum_{i=1}^{n} p_{i,k} \leq n \cdot \frac{1}{k!} \leq n \cdot \left(\frac{e}{k}\right)^k
    \]

  - Some arithmetic simplification will show that $P_k < 1/n$ when
    \[
    k = \frac{3 \ln n}{\ln \ln n}
    \]

---

1This is actually an upper bound, as there can be some double counting.
Balls and Bins: Summary of Results

$m$ balls are thrown at random into $n$ bins:

- Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
Balls and Bins: Summary of Results

$m$ balls are thrown at random into $n$ bins:

- Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
- No bin expected to be empty: $m = n \ln n$
Balls and Bins: Summary of Results

$m$ balls are thrown at random into $n$ bins:

- Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
- No bin expected to be empty: $m = n \ln n$
- Expected number of empty bins: $ne^{-m/n}$
Balls and Bins: Summary of Results

$m$ balls are thrown at random into $n$ bins:

- Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
- No bin expected to be empty: $m = n \ln n$
- Expected number of empty bins: $ne^{-m/n}$
- Max. balls in any bin when $m = n$:
  \[ \Theta(\ln n / \ln \ln n) \]
  
  - This is a probabilistic bound: chance of finding any bin with higher occupancy is $1/n$ or less.
  
  - Note that the absolute maximum is $n$. 
Randomized Quicksort

- Picks a pivot at random. What is its complexity?
Randomized Quicksort

- Picks a pivot at random. What is its complexity?

- If pivot index is picked uniformly at random over the interval \([l, h]\), then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, expected complexity of randomized quicksort is given by:

\[
T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n-i))
\]
Randomized Quicksort

• Picks a pivot at random. What is its complexity?

• If pivot index is picked uniformly at random over the interval \([l, h]\), then:
  • every array element is equally likely to be selected as the pivot
  • every partition is equally likely
  • thus, expected complexity of randomized quicksort is given by:

\[
T(n) = n + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n - i))
\]

Summary: Input need not be random

• Expected \(O(n \log n)\) performance comes from externally forced randomness in picking the pivot
Cache or Page Eviction

- Caching algorithms have to evict entries when there is a miss
  - As do virtual memory systems on a page fault
Cache or Page Eviction

- Caching algorithms have to evict entries when there is a miss
  - As do virtual memory systems on a page fault

- Optimally, we should evict the “farthest in future” entry
  - But we can’t predict the future!
Cache or Page Eviction

- Caching algorithms have to evict entries when there is a miss
  - As do virtual memory systems on a page fault

- Optimally, we should evict the “farthest in future” entry
  - But we can’t predict the future!

- Result: many candidates for eviction. How can be avoid making bad (worst-case) choices repeatedly, even if input behaves badly?
Cache or Page Eviction

- Caching algorithms have to evict entries when there is a miss
  - As do virtual memory systems on a page fault

- Optimally, we should evict the “farthest in future” entry
  - But we can’t predict the future!

- Result: many candidates for eviction. How can we avoid making bad (worst-case) choices repeatedly, even if input behaves badly?

- Approach: pick one of the candidates at random!
Hash Tables

A data structure for implementing:

**Dictionaries:** Fast look up of a record based on a key.

**Sets:** Fast membership check.
Hash Tables

- A data structure for implementing:
  - **Dictionaries**: Fast look up of a record based on a key.
  - **Sets**: Fast membership check.

- Support expected $O(1)$ time *lookup*, *insert*, and *delete*
Hash Tables

A data structure for implementing:

- **Dictionaries**: Fast look up of a record based on a key.
- **Sets**: Fast membership check.

Support expected $O(1)$ time *lookup, insert, and delete*

Hash table entries may be:

- **fat**: store a pair $(key, object)$
- **lean**: store pointer to object containing key
Hash Tables

- A data structure for implementing:
  - **Dictionaries**: Fast look up of a record based on a key.
  - **Sets**: Fast membership check.

- Support expected $O(1)$ time *lookup*, *insert*, and *delete*

- Hash table entries may be:
  - **fat**: store a pair $(key, object)$
  - **lean**: store pointer to object containing key

- Two main questions:
  - *How to avoid $O(n)$ worst case behavior?*
  - How to ensure *average case performance* can be realized *for arbitrary distribution of keys?*
Direct access: A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.
Hash Table Implementation

**Direct access:** A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.

**Index based on hash:** Given a hash function $h$ (fixed for the entire table) and a key $x$, use $h(x)$ to index into an array $A$. 

Hash Table Implementation

**Direct access**: A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.

**Index based on hash**: Given a hash function $h$ (fixed for the entire table) and a key $x$, use $h(x)$ to index into an array $A$.
- Use $A[h(x) \mod s]$, where $s$ is the size of array
- Sometimes, we fold the mod operation into $h$. 
Hash Table Implementation

Direct access: A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.

Index based on hash: Given a hash function $h$ (fixed for the entire table) and a key $x$, use $h(x)$ to index into an array $A$.

- Use $A[h(x) \mod s]$, where $s$ is the size of array
  - Sometimes, we fold the mod operation into $h$.
- Array elements typically called buckets
Direct access: A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.

Index based on hash: Given a hash function $h$ (fixed for the entire table) and a key $x$, use $h(x)$ to index into an array $A$.

- Use $A[h(x) \mod s]$, where $s$ is the size of array
  - Sometimes, we fold the mod operation into $h$.
- Array elements typically called *buckets*

*Collisions bound to occur* since $s \ll |\mathcal{U}|$

- Either $h(x) = h(y)$, or
- $h(x) \neq h(y)$ but $h(x) \equiv h(y) \pmod{s}$
Collisions in Hash tables

- **Load factor** \(\alpha\): Ratio of number of keys to number of buckets

If keys were random:
- What is the max \(\alpha\) if we want \(\leq 1\) collisions in the table?
- If \(\alpha = 1\), what is the maximum number of collisions to expect?

Both questions can be answered from balls-and-bins results: \(1/\sqrt{n}\), and \(O(\ln n / \ln \ln n)\). Real world keys are not random. Your hash table implementation needs to achieve its performance goals independent of this distribution.
Collisions in Hash tables

- **Load factor** $\alpha$: Ratio of number of keys to number of buckets

- *If* keys were random:
  - What is the max $\alpha$ if we want $\leq 1$ collisions in the table?
  - If $\alpha = 1$, what is the maximum number of collisions to expect?

Both questions can be answered from balls-and-bins results: $1/\sqrt{n}$, and $O((\ln n)/\ln \ln n)$. Real world keys are not random. Your hash table implementation needs to achieve its performance goals independent of this distribution.
Collisions in Hash tables

- **Load factor** $\alpha$: Ratio of number of keys to number of buckets

- *If keys were random:*
  - What is the max $\alpha$ if we want $\leq 1$ collisions in the table?
  - If $\alpha = 1$, what is the maximum number of collisions to expect?

- Both questions can be answered from balls-and-bins results: $1/\sqrt{n}$, and $O(\ln n / \ln \ln n)$

- **Real world keys are not random.** Your hash table implementation needs to achieve its performance goals independent of this distribution.
Chained Hash Table

- Each bucket is a linked list.
- Any key that hashes to a bucket is inserted into that bucket.
- What is the average search time, as a function of $\alpha$?
Chained Hash Table

- Each bucket is a linked list.
- Any key that hashes to a bucket is inserted into that bucket.
- What is the *average* search time, as a function of $\alpha$?
  - It is $1 + \alpha$ if:
    - you assume that the distribution of lookups is independent of the table entries, OR,
    - the chains are not too long (i.e., $\alpha$ is small)
Open addressing

- If there is a collision, probe other empty slots
  
  **Linear probing:** If $h(x)$ is occupied, try $h(x) + i$ for $i = 1, 2, ...$
  
  **Binary probing:** Try $h(x) \oplus i$, where $\oplus$ stands for exor.
  
  **Quadratic probing:** For $i$th probe, use $h(x) + c_1 i + c_2 i^2$
Open addressing

• If there is a collision, probe other empty slots
  
  **Linear probing:** If $h(x)$ is occupied, try $h(x) + i$ for $i = 1, 2, ...$
  
  **Binary probing:** Try $h(x) \oplus i$, where $\oplus$ stands for exor.
  
  **Quadratic probing:** For $i$th probe, use $h(x) + c_1 i + c_2 i^2$
  
• Criteria for secondary probes
  
  **Completeness:** Should cycle through all possible slots in table
  
  **Clustering:** Probe sequences shouldn’t coalesce to long chains
  
  **Locality:** Preserve locality; typically conflicts with clustering.
Open addressing

- If there is a collision, probe other empty slots
  
  **Linear probing:** If $h(x)$ is occupied, try $h(x) + i$ for $i = 1, 2, ...$
  
  **Binary probing:** Try $h(x) \oplus i$, where $\oplus$ stands for exor.
  
  **Quadratic probing:** For $i$th probe, use $h(x) + c_1 i + c_2 i^2$

- Criteria for secondary probes
  
  **Completeness:** Should cycle through all possible slots in table
  
  **Clustering:** Probe sequences shouldn’t coalesce to long chains
  
  **Locality:** Preserve locality; typically conflicts with clustering.

- Average search time can be $O(1/(1 - \alpha)^2)$ for linear probing, and $O(1/(1 - \alpha))$ for quadratic probing.
Chaining Vs Open Addressing

- Chaining leads to fewer collisions
  - Clustering causes more collisions w/ open addressing for same $\alpha$
  - However, for lean tables, open addressing uses half the space of chaining, so you can use a much lower $\alpha$ for same space usage.
Chaining Vs Open Addressing

- Chaining leads to fewer collisions
  - Clustering causes more collisions w/ open addressing for same $\alpha$
  - However, for lean tables, open addressing uses half the space of chaining, so you can use a much lower $\alpha$ for same space usage.

- Chaining is more tolerant of “lumpy” hash functions
  - For instance, if $h(x)$ and $h(x + 1)$ are often very close, open hashing can experience longer chains when inputs are closely spaced.
  - Hash functions for open-hashing having to be selected very carefully
Chaining Vs Open Addressing

- Chaining leads to fewer collisions
  - Clustering causes more collisions w/ open addressing for same $\alpha$
  - However, for lean tables, open addressing uses half the space of chaining, so you can use a much lower $\alpha$ for same space usage.

- Chaining is more tolerant of “lumpy” hash functions
  - For instance, if $h(x)$ and $h(x + 1)$ are often very close, open hashing can experience longer chains when inputs are closely spaced.
  - Hash functions for open-hashing having to be selected very carefully

- Linked lists are not cache-friendly
  - Can be mitigated w/ arrays for buckets instead of linked lists
Chaining Vs Open Addressing

- Chaining leads to fewer collisions
  - Clustering causes more collisions w/ open addressing for same $\alpha$
  - However, for lean tables, open addressing uses half the space of chaining, so you can use a much lower $\alpha$ for same space usage.

- Chaining is more tolerant of “lumpy” hash functions
  - For instance, if $h(x)$ and $h(x + 1)$ are often very close, open hashing can experience longer chains when inputs are closely spaced.
  - Hash functions for open-hashing having to be selected very carefully

- Linked lists are not cache-friendly
  - Can be mitigated w/ arrays for buckets instead of linked lists

- Not all quadratic probes cover all slots (but some can)
Resizing

- Hard to predict the right size for hash table in advance
  - Ideally, $0.5 \leq \alpha \leq 1$, so we need an accurate estimate
Resizing

- Hard to predict the right size for hash table in advance
  - Ideally, $0.5 \leq \alpha \leq 1$, so we need an accurate estimate

*It is stupid to ask programmers to guess the size*

- Without a good basis, only terrible guesses are possible
Resizing

- Hard to predict the right size for hash table in advance
  - Ideally, $0.5 \leq \alpha \leq 1$, so we need an accurate estimate

- *It is stupid to ask programmers to guess the size*
  - Without a good basis, only terrible guesses are possible

- **Right solution:** Resize tables automatically.
  - When $\alpha$ becomes too large (or small), rehash into a bigger (or smaller) table
Hard to predict the right size for hash table in advance
- Ideally, \(0.5 \leq \alpha \leq 1\), so we need an accurate estimate

*It is stupid to ask programmers to guess the size*
- Without a good basis, only terrible guesses are possible

**Right solution:** Resize tables automatically.
- When \(\alpha\) becomes too large (or small), rehash into a bigger (or smaller) table
- Rehashing is \(O(n)\), but if you increase size by a factor, then amortized cost is still \(O(1)\)
Resizing

- Hard to predict the right size for hash table in advance
  - Ideally, $0.5 \leq \alpha \leq 1$, so we need an accurate estimate

- *It is stupid to ask programmers to guess the size*
  - Without a good basis, only terrible guesses are possible

- **Right solution:** Resize tables automatically.
  - When $\alpha$ becomes too large (or small), rehash into a bigger (or smaller) table
  - Rehashing is $O(n)$, but if you increase size by a factor, then amortized cost is still $O(1)$
  - Exercise: How to ensure amortized $O(1)$ cost when you resize up as well as down?
Average Vs Worst Case

- Worst case search time is $O(n)$ for a table of size $n$
Average Vs Worst Case

- Worst case search time is \( O(n) \) for a table of size \( n \)
- With hash tables, it is all about avoiding the worst case, and achieving the average case
Average Vs Worst Case

- Worst case search time is $O(n)$ for a table of size $n$

- *With hash tables, it is all about avoiding the worst case, and achieving the average case*

Two main challenges:

- *Input is not random*, e.g., names or IP addresses.
- Even when input is random, $h$ may cause “lumping,” or non-uniform dispersal of $\mathcal{U}$ to the set $\{1, \ldots, n\}$
Average Vs Worst Case

- Worst case search time is $O(n)$ for a table of size $n$
- *With hash tables, it is all about avoiding the worst case, and achieving the average case*

Two main challenges:
- *Input is not random*, e.g., names or IP addresses.
- Even when input is random, $h$ may cause “lumping,” or non-uniform dispersal of $\mathcal{U}$ to the set $\{1, \ldots, n\}$

Two main techniques
- Universal hashing
- Perfect hashing
Universal Hashing

No single hash function can be good on all inputs

Any function $\mathcal{U} \rightarrow \{1, \ldots, n\}$ must map $|\mathcal{U}|/n$ inputs to same value!

*Note:* $|\mathcal{U}|$ can be much, much larger than $n$. 
Universal Hashing

- No single hash function can be good on all inputs
  - Any function $\mathcal{U} \to \{1, \ldots, n\}$ must map $|\mathcal{U}|/n$ inputs to same value!

  *Note: $|\mathcal{U}|$ can be much, much larger than $n$.*

**Definition**

A family of hash functions $\mathcal{H}$ is universal if

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{n} \quad \text{for all } x \neq y$$
Universal Hashing

- No single hash function can be good on all inputs
  - Any function $\mathcal{U} \to \{1, \ldots, n\}$ must map $|\mathcal{U}|/n$ inputs to same value!

  Note: $|\mathcal{U}|$ can be much, much larger than $n$.

**Definition**

A family of hash functions $\mathcal{H}$ is universal if

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{n} \quad \text{for all } x \neq y$$

**Meaning:** If we pick $h$ at random from the family $\mathcal{H}$, then, probability of collisions is the same for any two elements.

**Contrast with non-universal hash functions** such as

$$h(x) = ax \mod n, \quad (a \text{ is chosen at random})$$

Note $y$ and $y + kn$ collide with a probability of $1$ for every $a$. 

Universal Hashing Using Multiplication

Observation (Multiplication Modulo Prime)

If $p$ is a prime and $0 < a < p$

- $\{1a, 2a, 3a, \ldots, (p - 1)a\} = \{1, 2, \ldots, p - 1\} \pmod{p}$

- $\forall a \exists b \ ab \equiv 1 \pmod{p}$
Observation (Multiplication Modulo Prime)

If $p$ is a prime and $0 < a < p$

- $\{1a, 2a, 3a, \ldots, (p - 1)a\} = \{1, 2, \ldots, p - 1\} \pmod{p}$
- $\forall a \exists b \ ab \equiv 1 \pmod{p}$

Prime multiplicative hashing

Let the key $x \in \mathcal{U}$, $p > |\mathcal{U}|$ be prime, and $0 < r < p$ be random. Then

$$h(x) = (rx \mod p) \mod n$$

is universal.

Prove: $Pr[h(x) = h(y)] = \frac{1}{n}$, for $x \neq y$
Universality of prime multiplicative hashing

Need to show $Pr[h(x) = h(y)] = \frac{1}{n}$, for $x \neq y$

$h(x) = h(y)$ means $(rx \mod p) \mod n = (ry \mod p) \mod n$

Note $a \mod n = b \mod n$ means $a = b + kn$ for some integer $k$. Using this, we eliminate $\mod n$ from above equation to get:

$$rx \mod p = kn + ry \mod p, \text{ where } k \leq \lfloor p/n \rfloor$$

$$rx \equiv kn + ry \pmod{p}$$

$$r(x - y) \equiv kn \pmod{p}$$

$$r \equiv kn(x - y)^{-1} \pmod{p}$$

So, $x, y$ collide if $r = n(x - y)^{-1}, 2n(x - y)^{-1}, \ldots, \lfloor p/n \rfloor n(x - y)^{-1}$

In other words, $x$ and $y$ collide for $p/n$ out of $p$ possible values of $r$, i.e., collision probability is $1/n$
Binary multiplicative hashing

- Faster: avoids need for computing modulo prime
Binary multiplicative hashing

- Faster: avoids need for computing modulo prime
- When $|\mathcal{U}| < 2^w$, $n = 2^l$ and $a$ an odd random number

$$h(x) = \left\lfloor \frac{ax \mod 2^w}{2^{w-l}} \right\rfloor$$
Binary multiplicative hashing

- Faster: avoids need for computing modulo prime
- When $|\mathcal{U}| < 2^w$, $n = 2^l$ and $a$ an odd random number
  
  $$h(x) = \left\lfloor \frac{ax \mod 2^w}{2^{w-l}} \right\rfloor$$

- Can be implemented efficiently if $w$ is the wordsize:
  
  $$(a \cdot x) \gg (\text{WORDSIZE-HASHBITS})$$
Binary multiplicative hashing

- Faster: avoids need for computing modulo prime
- When $|U| < 2^w$, $n = 2^l$ and $a$ an odd random number
  \[
  h(x) = \left\lfloor \frac{ax \mod 2^w}{2^{w-l}} \right\rfloor
  \]
- Can be implemented efficiently if $w$ is the wordsize:
  \[
  (a \cdot x) \gg (\text{WORDSIZE-HASHBITS})
  \]
- Scheme is near-universal: collision probability is $O(1)/2^l$
Prime Multiplicative Hash for Vectors

Let \( p \) be a prime number, and the key \( x \) be a vector \([x_1, \ldots, x_k]\) where \( 0 \leq x_i < p \). Let

\[
h(x) = \sum_{i=1}^{k} r_i x_i \pmod{p}
\]

If \( 0 < r_i < p \) are chosen at random, then \( h \) is universal.
Prime Multiplicative Hash for Vectors

Let \( p \) be a prime number, and the key \( x \) be a vector \([x_1, \ldots, x_k]\) where \( 0 \leq x_i < p \). Let

\[
h(x) = \sum_{i=1}^{k} r_i x_i \pmod{p}
\]

If \( 0 < r_i < p \) are chosen at random, then \( h \) is universal.

- Strings can also be handled like vectors, or alternatively, as a polynomial evaluated at a random point \( a \), with \( p \) a prime:

\[
h(x) = \sum_{i=0}^{l} x_i a^i \pmod{p}
\]
Universality of multiplicative hashing for vectors

Since $x \neq y$, there exists an $i$ such that $x_i \neq y_i$
Universality of multiplicative hashing for vectors

- Since $x \neq y$, there exists an $i$ such that $x_i \neq y_i$
- When collision occurs, $\sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p}$
Universality of multiplicative hashing for vectors

- Since $x \neq y$, there exists an $i$ such that $x_i \neq y_i$
- When collision occurs, $\sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p}$
- Rearranging, $\sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \pmod{p}$
Universality of multiplicative hashing for vectors

- Since $x \neq y$, there exists an $i$ such that $x_i \neq y_i$

- When collision occurs, $\sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p}$

- Rearranging, $\sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \pmod{p}$

- The lhs evaluates to some $c$, and we need to estimate the probability that rhs evaluates to this $c$
Universality of multiplicative hashing for vectors

- Since \( x \neq y \), there exists an \( i \) such that \( x_i \neq y_i \)
- When collision occurs, \( \sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p} \)
- Rearranging, \( \sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \pmod{p} \)
- The lhs evaluates to some \( c \), and we need to estimate the probability that rhs evaluates to this \( c \)
- Using multiplicative inverse property, we see that \( r_i = c(y_i - x_i)^{-1} \pmod{p} \).
Universality of multiplicative hashing for vectors

- Since \( x \neq y \), there exists an \( i \) such that \( x_i \neq y_i \)

- When collision occurs, \( \sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \ (mod \ p) \)

- Rearranging, \( \sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \ (mod \ p) \)

- The lhs evaluates to some \( c \), and we need to estimate the probability that rhs evaluates to this \( c \)

- Using multiplicative inverse property, we see that \( r_i = c(y_i - x_i)^{-1} \ (mod \ p) \).

- Since \( y_i, x_i < p \), it is easy to see from this equation that the collision-causing value of \( r_i \) is distinct for distinct \( y_i \).
Universality of multiplicative hashing for vectors

- Since $x \neq y$, there exists an $i$ such that $x_i \neq y_i$

- When collision occurs, $\sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p}$

- Rearranging, $\sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \pmod{p}$

- The lhs evaluates to some $c$, and we need to estimate the probability that rhs evaluates to this $c$

- Using multiplicative inverse property, we see that $r_i = c (y_i - x_i)^{-1} \pmod{p}$.

- Since $y_i, x_i < p$, it is easy to see from this equation that the collision-causing value of $r_i$ is distinct for distinct $y_i$.

- Viewed another way, exactly one of $p$ choices of $r_i$ would cause a collision between $x_i$ and $y_i$, i.e., $Pr[h(x) = h(y)] = 1/p$
Perfect hashing

Static: Pick a hash function (or set of functions) that avoids collisions for a given set of keys
Perfect hashing

**Static:** Pick a hash function (or set of functions) that avoids collisions for a given set of keys

**Dynamic:** Keys need not be static.

**Approach 1:** Use $O(n^2)$ storage. Expected collision on $n$ items is 0. But too wasteful of storage.

Don’t forget: more memory usually means less performance due to cache effects.
Perfect hashing

**Static:** Pick a hash function (or set of functions) that avoids collisions for a given set of keys

**Dynamic:** Keys need not be static.

**Approach 1:** Use $O(n^2)$ storage. Expected collision on $n$ items is 0. But too wasteful of storage.

Don’t forget: more memory usually means less performance due to cache effects.

**Approach 2:** Use a secondary hash table for each bucket of size $n_i^2$, where $n_i$ is the number of elements in the bucket.

Uses only $O(n)$ storage, *if* $h$ is universal
Hashing Summary

- Excellent average case performance
  - Pointer chasing is expensive on modern hardware, so improvement from $O(\log n)$ of binary trees to expected $O(1)$ for hash tables is significant.

Universal/Perfect hash

Closest pair
Excellent average case performance
- Pointer chasing is expensive on modern hardware, so improvement from $O(\log n)$ of binary trees to expected $O(1)$ for hash tables is significant.

But all benefits will be reversed if collisions occur too often
- Universal hashing is a way to ensure expected average case even when input is not random.

Perfect hashing can provide efficient performance even in the worst case, but the benefits are likely small in practice.
Finding closest pair of points

*Problem*: Given a set of $n$ points in a $d$-dimensional space, identify the two that have the smallest Euclidean distance between them.

*Applications*: A central problem in graphics, vision, air-traffic control, navigation, molecular modeling, and so on.
Randomized Closest Pair: Key Ideas

- Divide the plane into small squares, hash points into them
- Pairwise comparisons can be limited to points within the squares very closeby
Randomized Closest Pair: Key Ideas

- Divide the plane into small squares, hash points into them
  - Pairwise comparisons can be limited to points within the squares very closeby

- Process the points in some random order
  - Maintain min. distance $\delta$ among points processed so far.
  - Update $\delta$ as more points are processed
Randomized Closest Pair: Key Ideas

- Divide the plane into small squares, hash points into them
  - Pairwise comparisons can be limited to points within the squares very closeby

- Process the points in some random order
  - Maintain min. distance $\delta$ among points processed so far.
  - Update $\delta$ as more points are processed

- At any point, the “small squares” have a size of $\delta/2$
  - At most one point per square (or else points are closer than $\delta$)
  - Points closer than $\delta$ will at most be two squares from each other
    - Only constant number of points to consider
  - Requires rehashing all processed points when $\delta$ is updated.
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns
  - **Storage:** # of squares is $1/\delta^2$, which can be very large
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns
  - **Storage:** # of squares is $1/\delta^2$, which can be very large
    - Use a dictionary (hash table) that stores up to $n$ points, and maps $(2x_i/\delta, 2y_i/\delta)$ to $\{1, \ldots, n\}$
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns

  **Storage:** # of squares is $1/\delta^2$, which can be very large
  - Use a dictionary (hash table) that stores up to $n$ points, and maps $(2x_i/\delta, 2y_i/\delta)$ to $\{1, \ldots, n\}$
  - To process a point $(x_j, y_j)$
    - look up the dictionary at $(x_j/\delta \pm 2, y_j/\delta \pm 2)$
    - insert if it is not closer than $\delta$
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns

  **Storage:** # of squares is $1/\delta^2$, which can be very large
  
  - Use a dictionary (hash table) that stores up to $n$ points, and maps $(2x_i/\delta, 2y_i/\delta)$ to
    
    $\{1, ..., n\}$
  
  - To process a point $(x_j, y_j)$
    
    - look up the dictionary at $(x_j/\delta \pm 2, y_j/\delta \pm 2)$
    
    - insert if it is not closer than $\delta$

  **Rehashing points:** If closer than $\delta$ — very expensive.
Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance

- Two main concerns
  
  **Storage:** # of squares is $1/\delta^2$, which can be very large
  
  - Use a dictionary (hash table) that stores up to $n$ points, and maps $(2x_i/\delta, 2y_i/\delta)$ to $$\{1, \ldots, n\}$$
  
  - To process a point $(x_j, y_j)$
    
    - look up the dictionary at $(x_j/\delta \pm 2, y_j/\delta \pm 2)$
    
    - insert if it is not closer than $\delta$

  **Rehashing points:** If closer than $\delta$ — very expensive.

- Total runtime can all be “charged” to insert operations,
  
  - incl. those performed during rehashing

  so we will focus on estimating inserts.
Randomized Closest Pair: # of Inserts

**Theorem**

*If random variable $X_i$ denotes the likelihood of needing to rehash after processing $k$ points, then*

$$X_i \leq \frac{2}{i}$$
Randomized Closest Pair: # of Inserts

Theorem

If random variable $X_i$ denotes the likelihood of needing to rehash after processing $k$ points, then

$$X_i \leq \frac{2}{i}$$

- Let $p_1, p_2, \ldots, p_i$ be the points processed so far, and $p$ and $q$ be the closest among these
Randomized Closest Pair: # of Inserts

**Theorem**

If random variable $X_i$ denotes the likelihood of needing to rehash after processing $k$ points, then

$$X_i \leq \frac{2}{i}$$

- Let $p_1, p_2, \ldots, p_i$ be the points processed so far, and $p$ and $q$ be the closest among these.
- Rehashing is needed while processing $p_i$ if $p_i = p$ or $p_i = q$. 
Randomized Closest Pair: # of Inserts

Theorem

If random variable $X_i$ denotes the likelihood of needing to rehash after processing $k$ points, then

$$X_i \leq \frac{2}{i}$$

- Let $p_1, p_2, \ldots, p_i$ be the points processed so far, and $p$ and $q$ be the closest among these.
- Rehashing is needed while processing $p_i$ if $p_i = p$ or $p_i = q$.
- Since points are processed in random order, there is a $2/i$ probability that $p_i$ is one of $p$ or $q$. 

Theorem

The expected number of inserts is $3n$. 

Processing of $p_i$ involves $i$ inserts if rehashing takes place, and 1 insert otherwise. So, expected inserts for processing $p_i$ is $i \cdot X_i + 1 \cdot (1 - X_i) = 1 + (i - 1) \cdot X_i = 1 + 2(i - 1) = 3i - 1 \leq 3n$. Upper bound on expected inserts is thus $3n$. 

Look Ma! I have a linear-time randomized closest pair algorithm—and it is not even probabilistic!
Randomized Closest Pair: # of Inserts

Theorem

The expected number of inserts is $3n$.

- Processing of $p_i$ involves
  - $i$ inserts if rehashing takes place, and 1 insert otherwise
Randomized Closest Pair: # of Inserts

Theorem

The expected number of inserts is $3n$.

- Processing of $p_i$ involves
  - $i$ inserts if rehashing takes place, and 1 insert otherwise

So, expected inserts for processing $p_i$ is

$$i \cdot X_i + 1 \cdot (1 - X_i) = 1 + (i - 1) \cdot X_i = 1 + \frac{2(i - 1)}{i} \leq 3$$

Upper bound on expected inserts is thus $3n$. Look Ma! I have a linear-time randomized closest pair algorithm—and it is not even probabilistic!
Randomized Closest Pair: # of Inserts

Theorem

*The expected number of inserts is 3n.*

- Processing of $p_i$ involves
  - $i$ inserts if rehashing takes place, and 1 insert otherwise

- So, expected inserts for processing $p_i$ is
  \[
i \cdot X_i + 1 \cdot (1 - X_i) = 1 + (i - 1) \cdot X_i = 1 + \frac{2(i - 1)}{i} \leq 3
  \]

- Upper bound on expected inserts is thus $3n$
Theorem

The expected number of inserts is $3n$.

- Processing of $p_i$ involves
  - $i$ inserts if rehashing takes place, and 1 insert otherwise

- So, expected inserts for processing $p_i$ is
  \[
  i \cdot X_i + 1 \cdot (1 - X_i) = 1 + (i - 1) \cdot X_i = 1 + \frac{2(i - 1)}{i} \leq 3
  \]

- Upper bound on expected inserts is thus $3n$

Look Ma! I have a linear-time randomized closest pair algorithm—And it is not even probabilistic!
Probabilistic Algorithms

- Algorithms that produce the correct answer with some probability
- By re-running the algorithm many times, we can increase the probability to be arbitrarily close to 1.0.
Bloom Filters

- To resolve collisions, hash tables have to store keys: $O(mw)$ bits, where $w$ is the number of bits in the key
Bloom Filters

- To resolve collisions, hash tables have to store keys: $O(mw)$ bits, where $w$ is the number of bits in the key
- What if you want to store very large keys?
Bloom Filters

- To resolve collisions, hash tables have to store keys: $O(mw)$ bits, where $w$ is the number of bits in the key
- What if you want to store very large keys?
- **Radical idea:** Don’t store the key in the table!
  - Potentially $w$-fold space reduction
Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
- Hash table is simply a bitvector $B[1..m]$
Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
- Hash table is simply a bitvector $B[1..m]$
- To insert key $x$, set $B[h_1(x)], B[h_2(x)], \ldots, B[h_k(x)]$

Images from Wikipedia Commons
Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
- Hash table is simply a bitvector $B[1..m]$
- To insert key $x$, set $B[h_1(x)], B[h_2(x)], \ldots, B[h_k(x)]$

Membership check for $y$: all $B[h_i(y)]$ should be set
- No false negatives, but false positives possible
Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
- Hash table is simply a bitvector $B[1..m]$
- To insert key $x$, set $B[h_1(x)], B[h_2(x)], \ldots, B[h_k(x)]$

Membership check for $y$: all $B[h_i(y)]$ should be set
- No false negatives, but false positives possible
- No deletions possible in the current algorithm.
Bloom Filters: False positives

- Prob. that a bit is \textit{not} set by \( h_1 \) on inserting a key is \((1 - 1/m)\)
Bloom Filters: False positives

- Prob. that a bit is *not* set by $h_1$ on inserting a key is $(1 - 1/m)$
- The probability it is not set by any $h_i$ is $(1 - 1/m)^k$
Bloom Filters: False positives

- Prob. that a bit is not set by $h_1$ on inserting a key is $(1 - 1/m)$
- The probability it is not set by any $h_i$ is $(1 - 1/m)^k$
- The probability it is not set after $r$ key inserts is $(1 - 1/m)^{kr} \approx e^{-kr/m}$
Bloom Filters: False positives

- Prob. that a bit is *not* set by $h_1$ on inserting a key is $(1 - 1/m)$
- The probability it is not set by any $h_i$ is $(1 - 1/m)^k$
- The probability it is not set after $r$ key inserts is $(1 - 1/m)^{kr} \approx e^{-kr/m}$

- Complementing, the prob. $p$ that a certain bit is set is $1 - e^{-kr/m}$
Bloom Filters: False positives

- Prob. that a bit is not set by $h_1$ on inserting a key is $(1 - 1/m)$
  - The probability it is not set by any $h_i$ is $(1 - 1/m)^k$
  - The probability it is not set after $r$ key inserts is $(1 - 1/m)^{kr} \approx e^{-kr/m}$

- Complementing, the prob. $p$ that a certain bit is set is $1 - e^{-kr/m}$

- For a false positive on a key $y$, all the bits that it hashes to should be a 1. This happens with probability

$$\left(1 - e^{-kr/m}\right)^k = (1 - p)^k$$
Bloom Filters

- Note: $n = m/r$ is the storage (in bits) used per key.

- So, we can rewrite the FP equation as:

$$\left(1 - e^{-kr/m}\right)^k = \left(1 - e^{-k/n}\right)^k$$
Bloom Filters

- Note: \( n = m/r \) is the storage (in bits) used per key.

- So, we can rewrite the FP equation as:
  \[
  \left(1 - e^{-kr/m}\right)^k = \left(1 - e^{-k/n}\right)^k
  \]

- Optimal value of \( k \) can be shown to be \( n \ln 2 \).
  - The FP rate simplifies to \( 0.5^{n\ln 2} = 0.619^n \)

- A Bloom filter that uses just 8 bits per key to store an *arbitrary sized key* will have an FP rate of 2%
Bloom Filters

- Note: \( n = m/r \) is the storage (in bits) used per key.

- So, we can rewrite the FP equation as:

\[
\left(1 - e^{-kr/m}\right)^k = \left(1 - e^{-k/n}\right)^k
\]

- Optimal value of \( k \) can be shown to be \( n \ln 2 \).
  - The FP rate simplifies to \( 0.5^{n \ln 2} = 0.619^n \)

- A Bloom filter that uses just 8 bits per key to store an *arbitrary sized key* will have an FP rate of 2%

- **Important:** Bloom filters can be used as a prefilter, e.g., if actual keys are in secondary storage (e.g., files or internet repositories)
Using arithmetic for substring matching

**Problem:** Given strings $T[1..n]$ and $P[1..m]$, find occurrences of $P$ in $T$ in $O(n + m)$ time.

**Idea:** To simplify presentation, assume $P$, $T$ range over $[0-9]$

- Interpret $P[1..m]$ as digits of a number
  \[ p = 10^{m-1}P[1] + 10^{m-2}P[2] + \cdots + 10^{m-m}P[m] \]
- Similarly, interpret $T[i..(i + m - 1)]$ as the number $t_i$
- Note: $P$ is a substring of $T$ at $i$ iff $p = t_i$
- To get $t_{i+1}$, shift $T[i]$ out of $t_i$, and shift in $T[i + m]$:
  \[ t_{i+1} = (t_i - 10^{m-1}T[i]) \cdot 10 + T[i + m] \]

We have an $O(n + m)$ algorithm. Almost: we still need to figure out how to operate on $m$-digit numbers in constant time!
Rabin-Karp Fingerprinting

Key Idea

- Instead of working with $m$-digit numbers,
- perform all arithmetic modulo a *random* prime number $q$,
- where $q > m^2$ fits within wordsize

All observations made on previous slide still hold

- Except that $p = t_i$ does not guarantee a match
- Typically, we expect matches to be infrequent, so we can use $O(m)$ exact-matching algorithm to confirm probable matches.
Carter-Wegman-Rabin-Karp Algorithm

**Difficulty with Rabin-Karp:** Need to generate random primes, which is not an efficient task.

**New Idea:** Make the radix random, as opposed to the modulus

- We still compute modulo a prime $q$, but it is not random.

**Alternative interpretation:** We treat $P$ as a polynomial

$$p(x) = \sum_{i=1}^{m} P[m - i] \cdot x^i$$

and evaluate this polynomial at a randomly chosen value of $x$

**Like any probabilistic algorithm** we can increase correctness probability by repeating the algorithm with different randoms.

- Different prime numbers for Rabin-Karp
- Different values of $x$ for CWRK
Carter-Wegman-Rabin-Karp Algorithm

\[ p(x) = \sum_{i=1}^{m} P[m - i] \cdot x^i \]

Random choice does not imply high probability of being right.

- You need to explicitly establish correctness probability.
Carter-Wegman-Rabin-Karp Algorithm

\[ p(x) = \sum_{i=1}^{m} P[m - i] \cdot x^i \]

*Random choice does not imply high probability of being right.*

- You need to explicitly establish correctness probability.

So, what is the likelihood of false matches?

- A false match occurs if \( p_1(x) = p_2(x) \), i.e., \( p_1(x) - p_2(x) = p_3(x) = 0 \).
Carter-Wegman-Rabin-Karp Algorithm

\[ p(x) = \sum_{i=1}^{m} P[m - i] \cdot x^i \]

*Random choice does not imply high probability of being right.*

- You need to explicitly establish correctness probability.

So, what is the likelihood of false matches?

- A false match occurs if \( p_1(x) = p_2(x) \), i.e., \( p_1(x) - p_2(x) = p_3(x) = 0 \).

- Arithmetic modulo prime defines a *field*, so an \( m \)th degree polynomial has \( m + 1 \) roots.
Carter-Wegman-Rabin-Karp Algorithm

\[ p(x) = \sum_{i=1}^{m} P[m - i] \cdot x^i \]

*Random choice does not imply high probability of being right.*

- You need to explicitly establish correctness probability.

So, what is the likelihood of false matches?

- A false match occurs if \( p_1(x) = p_2(x) \), i.e., \( p_1(x) - p_2(x) = p_3(x) = 0 \).

- Arithmetic modulo prime defines a *field*, so an \( m \)th degree polynomial has \( m + 1 \) roots.

- Thus, \( (m + 1)/q \) of the \( q \) (recall \( q \) is the prime number used for performing modulo arithmetic) possible choices of \( x \) will result in a false match, i.e., probability of false positive = \( (m + 1)/q \)
Fermat’s Theorem

\[ a^{p-1} \equiv 1 \pmod{p} \]
**Primality Testing**

**Fermat’s Theorem**

\[ a^{p-1} \equiv 1 \pmod{p} \]

- Recall \( \{1a, 2a, 3a, \ldots, (p - 1)a\} \equiv \{1, 2, \ldots, p - 1\} \pmod{p} \)
Primality Testing

Fermat’s Theorem

\[ a^{p-1} \equiv 1 \pmod{p} \]

- Recall \( \{1a, 2a, 3a, \ldots, (p - 1)a\} \equiv \{1, 2, \ldots, p - 1\} \pmod{p} \)

- Multiply all elements of both sides:

\[ (p - 1)!a^{p-1} \equiv (p - 1)! \pmod{p} \]
Primality Testing

Fermat’s Theorem

\[ a^{p-1} \equiv 1 \pmod{p} \]

- Recall \( \{1a, 2a, 3a, \ldots, (p - 1)a\} \equiv \{1, 2, \ldots, p - 1\} \pmod{p} \)

- Multiply all elements of both sides:
  
  \[
  (p - 1)!a^{p-1} \equiv (p - 1)! \pmod{p}
  \]

- Canceling out \( (p - 1)! \) from both sides, we have the theorem!
Primality Testing

- Given a number $N$, we can use Fermat’s theorem as a probabilistic test to see if it is prime:
  - if $a^{N-1} \not\equiv 1 \pmod{N}$ then $N$ is not prime
  - Repeat with different values of $a$ to gain more confidence
Given a number $N$, we can use Fermat’s theorem as a probabilistic test to see if it is prime:
- if $a^{N-1} \not\equiv 1 \pmod{N}$ then $N$ is not prime
- Repeat with different values of $a$ to gain more confidence

**Question:** If $N$ is *not* prime, what is the probability that the above procedure will fail?
Primality Testing

Given a number $N$, we can use Fermat’s theorem as a probabilistic test to see if it is prime:

- if $a^{N-1} \not\equiv 1 \pmod{N}$ then $N$ is not prime
- Repeat with different values of $a$ to gain more confidence

**Question:** If $N$ is not prime, what is the probability that the above procedure will fail?

- For Carmichael’s numbers, the probability is 1 — but ignore this for now, since these numbers are very rare.
Primality Testing

- Given a number $N$, we can use Fermat’s theorem as a probabilistic test to see if it is prime:
  - if $a^{N-1} \not\equiv 1 \pmod{N}$ then $N$ is not prime
  - Repeat with different values of $a$ to gain more confidence

**Question:** If $N$ is *not* prime, what is the probability that the above procedure will fail?

- For Carmichael’s numbers, the probability is 1 — but ignore this for now, since these numbers are very rare.
- For other numbers, we can show that the above procedure works with probability 0.5
Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for a relatively prime to $N$, then it holds for at least half the choices of $a < N$. 

Primality Testing

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for a relatively prime to $N$, then it holds for at least half the choices of $a < N$.

- If there is no $b$ such that $b^{N-1} \equiv 1 \pmod{N}$, then we have nothing to prove.
Primality Testing

Lemma

If \(a^{N-1} \not\equiv 1 \pmod{N}\) for a relatively prime to \(N\), then it holds for at least half the choices of \(a < N\).

- If there is no \(b\) such that \(b^{N-1} \equiv 1 \pmod{N}\), then we have nothing to prove.
- Otherwise, pick one such \(b\), and consider \(c \equiv ab\).
Primality Testing

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for a relatively prime to $N$, then it holds for at least half the choices of $a < N$.

- If there is no $b$ such that $b^{N-1} \equiv 1 \pmod{N}$, then we have nothing to prove.
- Otherwise, pick one such $b$, and consider $c \equiv ab$.
- Note $c^{N-1} \equiv a^{N-1}b^{N-1} \equiv a^{N-1} \not\equiv 1$
**Lemma**

If \( a^{N-1} \not\equiv 1 \pmod{N} \) for a relatively prime to \( N \), then it holds for at least half the choices of \( a < N \).

- If there is no \( b \) such that \( b^{N-1} \equiv 1 \pmod{N} \), then we have nothing to prove.
- Otherwise, pick one such \( b \), and consider \( c \equiv ab \).
- Note \( c^{N-1} \equiv a^{N-1}b^{N-1} \equiv a^{N-1} \not\equiv 1 \)
- Thus, for every \( b \) for which Fermat’s test is satisfied, there exists a \( c \) that does not satisfy it.
- Moreover, since \( a \) is relatively prime to \( N \), \( ab \not\equiv ab' \) unless \( b \equiv b' \).
Primality Testing

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for a relatively prime to $N$, then it holds for at least half the choices of $a < N$.

- If there is no $b$ such that $b^{N-1} \equiv 1 \pmod{N}$, then we have nothing to prove.
- Otherwise, pick one such $b$, and consider $c \equiv ab$.
- Note $c^{N-1} \equiv a^{N-1}b^{N-1} \equiv a^{N-1} \not\equiv 1$
- Thus, for every $b$ for which Fermat’s test is satisfied, there exists a $c$ that does not satisfy it.
  - Moreover, since $a$ is relatively prime to $N$, $ab \not\equiv ab'$ unless $b \equiv b'$.
- Thus, at least half of the numbers $x < N$ relatively prime to $N$ will fail the test.
Primality Testing

Pick some $a$

Is $a^{N-1} \equiv 1 \mod N$?

Fermat’s test

Pass → “prime”

Fail → “composite”

- When Fermat’s test returns “prime” $Pr[N \text{ is not prime}] < 0.5$

- If Fermat’s test is repeated for $k$ choices of $a$, and returns “prime” in each case, $Pr[N \text{ is not prime}] < 0.5^k$

- In fact, 0.5 is an upper bound. Empirically, the probability has been much smaller.
Primality Testing

Empirically, on numbers less than 25 billion, the probability of Fermat’s test failing to detect non-primes (with $a = 2$) is more like 0.00002

This probability decreases even more for larger numbers.
Prime number generation

Lagrange’s Prime Number Theorem

For large $N$, primes occur approx. once every $\log N$ numbers.
Prime number generation

Lagrange’s Prime Number Theorem
For large $N$, primes occur approx. once every $\log N$ numbers.

Generating Primes
- Generate a random number
- Probabilistically test it is prime, and if so output it
- Otherwise, repeat the whole process

What is the complexity of this procedure?
$O(\log \log N)$ multiplications on $\log N$ bit numbers

If $N$ is not prime, should we try $N + 1$, $N + 2$, ... instead of generating a new random number?
No, it is not easy to decide when to give up.
Prime number generation

Lagrange’s Prime Number Theorem

For large $N$, primes occur approx. once every $\log N$ numbers.

Generating Primes

- Generate a random number
- Probabilistically test it is prime, and if so output it
- Otherwise, repeat the whole process

What is the complexity of this procedure?
Prime number generation

Lagrange’s Prime Number Theorem
For large $N$, primes occur approx. once every $\log N$ numbers.

Generating Primes
- Generate a random number
- Probabilistically test it is prime, and if so output it
- Otherwise, repeat the whole process

What is the complexity of this procedure?
- $O(\log^2 N)$ multiplications on $\log N$ bit numbers
- If $N$ is not prime, should we try $N + 1, N + 2, \ldots$ instead of generating a new random number?
Prime number generation

Lagrange’s Prime Number Theorem
For large $N$, primes occur approx. once every $\log N$ numbers.

Generating Primes
- Generate a random number
- Probabilistically test it is prime, and if so output it
- Otherwise, repeat the whole process

What is the complexity of this procedure?
- $O(\log^2 N)$ multiplications on $\log N$ bit numbers

If $N$ is not prime, should we try $N + 1, N + 2, \ldots$ instead of generating a new random number?
- No, it is not easy to decide when to give up.
Rabin-Miller Test

- Works on Carmichael’s numbers
- For prime number test, we consider only odd $N$, so $N - 1 = 2^t u$ for some odd $u$
- Compute
  \[ a^u, a^{2u}, a^{4u}, \ldots, a^{2^t u} = a^{N-1} \]
- If $a^{N-1}$ is not 1 then we know $N$ is composite.
- Otherwise, we do a follow-up test on $a^u, a^{2u}$ etc.
  - Let $a^{2^r u}$ be the first term that is equivalent to 1.
  - If $r > 0$ and $a^{2^r - 1 u} \neq -1$ then $N$ is composite
- This combined test detects non-primes with a probability of at least 0.75 for all numbers.