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. Decentralize
   - Medium Access
   - Coupon Collection
   - Birthday
   - Balls and Bins
3. Taming distribution
4. Probabilistic Algorithms
   - Caching
   - Closest pair
   - Hashing
   - Universal/Perfect hash
   - Bloom filter
   - Rabin-Karp
   - Prime testing
   - Min-cut
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

A Randomized Protocol for Medium Access

- Suppose $n$ hosts want to access a shared medium
  - If multiple hosts try at the same time, there is contention, and the “slot” is wasted.
  - A slot is wasted if no one tries.
  - How can we maximize the likelihood of every slot being utilized?

- Suppose that a randomized protocol is used.
  - Each host transmits with a probability $p$
  - What should be the value of $p$?

- We want the likelihood that one host will attempt access (probability $p$), while others don’t try (probability $(1-p)^{n-1}$)
  - Find $p$ that maximizes $p(1-p)^{n-1}$
  - Using differentiation to find maxima, we get $p = 1/n$

Maximum probability (when $p = 1/n$)

$$\frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1}$$

- Note $(1 - \frac{1}{n})^{n-1}$ converges to $1/e$ for reasonably large $n$
  - About 5% off $e$ at $n = 10$.
  - So, let us simplify the expression to $1/ne$ for future calculations

What is the efficiency of the protocol?

- The probability that some host gets to transmit is $n \cdot 1/ne = 1/e$

Is this protocol a reasonable choice?

- Wasting almost 2/3rd of the slots is rarely acceptable

How long before a host $i$ can expect to transmit successfully?

- The probability it fails the first time is $(1 - 1/ne)$
  - Probability $i$ fails in $k$ attempts: $(1 - 1/ne)^k$
  - This quantity gets to be reasonably small (specifically, $1/e$) when $k = ne$
  - For larger $k$, say $k = ne \cdot c \ln n$, the expression becomes
    $$\left((1 - 1/ne)^{ne}\right)^{c \ln n} = (1/e)^{c \ln n} = (e^{\ln n})^{-c} = n^{-c}$$

- So, a host has a reasonable success chance in $O(n)$ attempts
  - This becomes a virtual certainty in $O(n \ln n)$ attempts
A Randomized Protocol for Medium Access

- What is the expected wait time?
  - “Average” time a host can expect to try before succeeding.

\[ E[X] = \sum_{j=0}^{\infty} j \cdot Pr[X = j] \]

- For our protocol, expected wait time is given by

\[ 1 \cdot p + 2 \cdot (1 - p)p + 3 \cdot (1 - p)^2p \cdots = p \sum_{i=1}^{\infty} i \cdot (1 - p)^{i-1} \]

- How do we sum the series \( \sum ix^{i-1} \)?
  - Note that \( \sum_{i=1}^{\infty} x^i = \frac{1}{(1-x)} \). Now, differentiate both sides:
  \[
  \sum_{i=1}^{\infty} ix^{i-1} = -\frac{1}{(1-x)^2}
  \]

- Expected wait time is

\[ p \sum_{i=1}^{\infty} i \cdot (1 - p)^{i-1} = \frac{p}{p^2} = \frac{1}{p} \]

- We get an intuitive result — a host will need to wait \( 1/p = ne \) slots on the average.

- \textbf{Note:} The derivation is a general one, applies to any event with probability \( p \); it is not particular to this access protocol.

How long will it be before every host would have a high probability of succeeding?

- We are interested in the probability of

\[ S(k) = \bigcup_{i=1}^{n} S(i, k) \]

- Note that failures are not independent, so we cannot say that

\[ Pr[S(k)] = \sum_{i=1}^{n} Pr[S(i, k)] \]

but certainly, the rhs is an upper bound on \( Pr[F(k)] \).

- We use this approximate \textit{union bound} for our asymptotic analysis.

If we use \( k = ne \), then

\[ \sum_{i=1}^{n} Pr[S(i, k)] = \sum_{i=1}^{n} \frac{1}{e} = n/e \]

which suggests that the likelihood some hosts failed within \( ne \) attempts is rather high.

If we use \( k = cn \ln n \) then we get a bound:

\[ \sum_{i=1}^{n} Pr[S(i, k)] = \sum_{i=1}^{n} n^{-c/e} = n^{(e-c)/e} \]

which is relatively small — \( O(n^{-1}) \) for \( c = 2e \).

Thus, it is highly likely that all hosts will have succeeded in \( O(n \ln n) \) attempts.
A Randomized Protocol: Conclusions

- High school probability background is sufficient to analyze simple randomized algorithms
- Carefully work out each step
  - Intuition often fails us on probabilities
- If every host wants to transmit in every slot, this randomized protocol is a bad choice.
  - 63% wasted slots is unacceptable in most cases.
  - Better off with a round-robin or queuing based algorithm.
- How about protocols used in Ethernet or WiFi?
  - Optimistic: whoever needs to transmit will try in the next slot
  - Exponential backoff when collisions occur
  - Each collision halves $p$

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 $j$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:
  \[ E[X] = \sum_{i=0}^{n-1} E[X_i] = \sum_{i=0}^{n-1} n/(n - j) = nH(n) \]
  - 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.
- Abstracts the media access protocol just discussed!

Birthday Paradox

- What is the smallest size group where there are at least two people with the same birthday?
  - 365
  - 183
  - 61
  - 25
Birthday Paradox

- The probability that the $i^{th}$ person's birthday is distinct from previous $i$ is approx. \[ p_i = \frac{N - i}{N} \]

- Let $X_i$ be the number of duplicate birthdays added by $i$: \[ E[X_i] = 0 \cdot p_i + 1 \cdot (1 - p_i) = 1 - p_i = \frac{i}{N} \]

- Sum up $E_i$'s to find the # of distinct birthdays among $n$: \[ E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{i}{N} = \frac{n(n - 1)}{2N} \]

Thus, when $n \approx 27$, we have one duplicate birthday.

---

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?
  - Birthday problem

- What should $m$ be to have at least one ball per bin?
  - Coupon collection, media access protocol example

- What is the maximum number of balls in any bin?
  - Such problems arise in load-balancing, hashing, etc.

---

Balls and Bins: Max Occupancy

- Probability $p_{i,k}$ that the first bin receives at least $k$ balls:
  - Choose $k$ balls in $\binom{n}{k}$ ways
  - These balls should fall into the first bin: prob. is $(1/n)^k$
  - Other balls may fall anywhere, i.e., probability 1:
    \[ \binom{m}{k} \left( \frac{1}{n} \right)^k = \frac{m \cdot (m-1) \cdot \ldots \cdot (m-k+1)}{k! n^k} \leq \frac{m^k}{k! n^k} \]
  - 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
    \[ k = \frac{3 \ln n}{\ln n} \]

---

\[ ^1 \text{We are assuming that } i-1 \text{ birthdays are distinct: reasonable if } n \ll N \]

\[ ^2 \text{More accurate calculation will yield } n = 24.6 \]

\[ ^3 \text{This is actually an upper bound, as there can be some double counting.} \]
**Balls and Bins: Summary of Results**

- $m$ balls are thrown at random into $n$ bins:
  - Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
  - 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?
- 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
  - 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?
  - Approach: pick one of the candidates at random!

**Closest pair**

- We studied a deterministic divide-and-conquer algorithm for this problem.
  - Quite complex, required multiple sort operations at each stage.
  - Even then, the number of cross-division pairs to be considered seemed significant
  - Result: deterministic algorithm difficult to implement, and likely slow in practice.
  - Can a randomized algorithm be simpler and faster?
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
  - **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
      $(2x/\delta, 2y/\delta)$ 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

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

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

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*
  - *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.

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 \( i = 1, 2, \ldots \)
  - Binary probing: Try \( h(x) \oplus i \), where \( \oplus \) stands for exor.
  - Quadratic probing: For \( i \)th probe, use \( h(x) + ci + c2i^2 \)

- 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 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

- 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 \( n \)

- 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 \( U \) 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 \( U \to \{1, \ldots, n\} \) must map \(|U|/n\) inputs to same value!

  
  Note: \(|U|\) 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 \( a \).

Universality of prime multiplicative hashing

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

- \( h(x) = h(y) \) 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:

  \[
  rx \mod p = kn + ry \mod p, \quad \text{where } k \leq |p/n|
  \]

  \[
  r x \equiv kn + ry \pmod p
  \]

  \[
  r(x - y) \equiv kn \pmod p
  \]

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

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

- In other words, \( x \) and \( y \) collide for \( p/n \) out of \( p \) possible values of \( r \), i.e., collision probability is \( 1/n \)

Universal Hashing Using Multiplication

**Observation (Multiplication Modulo Prime)**

If \( p \) is a prime and \( 0 < a < p \)

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

- \( \forall a \equiv b \pmod p \)

Prime multiplicative hashing

Let the key \( x \in U \), \( p > |U| \) be prime, and \( 0 < r < p \) be random. Then

\[
 h(x) = (rx \mod p) \mod n
\]

is universal.

Prove: \( \Pr[h(x) = h(y)] = \frac{1}{n}, \text{ for } x \neq y \)

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:

  \( (a*x) >> (\text{WORDSIZE}-\text{HASHBITS}) \)

- 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.

- Strings can also be handled like vectors, or alternatively, as a polynomial evaluated at a random point \( a \), with \( p \) a prime:
\[
h(x) = \sum_{i=0}^{l} x_i a^i \pmod{p}
\]

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

 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., \( \Pr[h(x) = h(y)] = 1/p \)

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.
  - 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.
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
- 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: False positives

- Prob. that a bit is *not* set by $h_i$ 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
  $$(1 - e^{-kr/m})^k = (1 - p)^k$$
Bloom Filters

- Consider 
  \[ (1 - e^{-kr/m})^k \]
- Note that the table can potentially store very large number of entries with very low false positives
  - For instance, with \( k = 20 \), \( m = 10^9 \) bits (12M bytes), and a false positive rate of \( 2^{-10} = 10^{-3} \), can store 60M keys of arbitrary size!
- Exercise: What is the optimal value of \( k \) to minimize false positive rate for a given \( m \) and \( r \)?
- But large \( k \) values introduce high overheads
- 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

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

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 = \( (m + 1)/q \)

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!

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 \( c \) is relatively prime to \( N \), \( ab \not\equiv ab' \) unless \( b \equiv b' \).
- Thus, at least half of the numbers \( x < N \) that are relatively prime to \( N \) will fail Fermat’s test.
Primality Testing

- 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.

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 \( N+1, N+2, \text{ etc.} \) 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 \( a^r, a^{2u} \text{ etc.} \)
  - Let \( a^{r u} \) be the first term that is equivalent to 1.
  - If \( r > 0 \) and \( a^{r u} \neq -1 \) then \( N \) is composite
- This combined test detects non-primes with a probability of at least 0.75 for all numbers.
Global Min-cut in Undirected Graphs

- Compute the minimum number of edges that need to be severed to disconnect a graph
- Yields the edge-connectivity of the graph

A multigraph whose minimum cut has three edges.

Deterministic Global Min-cut

- Replace each undirected edge by two (opposing) directed edges
- Pick a vertex \( s \)
- for each \( t \) in \( V \) compute the minimum \( s-t \) cut
- The smallest among these is the global min-cut
- Repeating min-cut \( O(|V|) \) times, so it is expensive and complex.

Randomized global min-cut

- Relies on repeated “collapsing” of edges, illustrated below
  - Pick a random edge \((u,v)\), and delete it
  - Replace \( u \) and \( v \) by a single vertex \( uv \)
  - Replace each edge \((x,u)\) by \((x,uv)\)
  - Replace each edge \((x,v)\) by \((x,uv)\)
- Note: edges maintain their identity during this process

\[
\begin{align*}
\text{GuessMinCut}(V,E) & \\
\text{if } |V| = 2 & \text{ then return the only cut remaining} \\
\text{Pick an edge at random and collapse it to get } V', E' & \\
\text{return } \text{GuessMinCut}(V', E')
\end{align*}
\]

- Does this algorithm make sense? Why should it work?
- Basic idea: Only a small fraction of edges belong to the min-cut, reducing the likelihood of them being collapsed
- Still, when almost every edge is being collapsed, how likely is it that min-cut edges will remain?

A graph \( G \) and two collapsed graphs \( G/(b,e) \) and \( G/(c,d) \).
**Guess MinCut Correctness Probability**

- If min-cut has \( k \) edges, then every node has min degree \( k \)
- So, there are \( nk/2 \) edges
- The likelihood of collapsing them in the first step is \( 2/n \)
- The likelihood of preserving min-cut edges is \( (n-2)/n \)
- We thus have the following recurrence for likelihood of preserving min-cut edges in the final solution:
  
  \[
P(n) \geq \frac{n-2}{n} \cdot P(n-1) \geq \frac{n-2}{n} \cdot \frac{n-4}{n-1} \cdot \frac{n-6}{n-2} \cdot \ldots \cdot \frac{2}{n-2} \cdot \frac{1}{n-1} = \frac{2}{n(n-1)}
  \]

So, the probability of being wrong is high
- by repeating it \( O(n^2 \ln n) \) times, we reduce it to \( 1/n^c \).

Overall runtime is \( O(n^4 \ln n) \), which is hardly impressive.

**Power of Two Random Choices for Min-cut**

- Divide random collapses into two phases
  - An initial "safe" phase that shrinks the graph to \( 1 + n/\sqrt{2} \) nodes
  - Probability of preserving min-cut is
    \[
    \frac{(n/\sqrt{2})(n/\sqrt{2}+1)}{p(n-1)} \geq \frac{1}{2}
    \]
  - A second "unsafe" phase that is run twice, and the smaller min-cut is picked

- A single run of unsafe phase is simply a recursive call
  - A kind-of-divide and conquer with power-of-two
    - Since input size decreases with each level of recursion, total time is reduced in spite of exponential increase in number of iterations

- We get the following recurrence for correctness probability:
  \[
P(n) \geq 1 - \left( 1 - \frac{1}{2} P\left( \frac{n}{\sqrt{2}} + 1 \right) \right)^2
  \]
  which yields a result of \( \Omega(1/\log n) \)
- Need \( O(\log^2 n) \) repetitions to obtain low error rate
- For runtime, we have the recurrence
  \[
  T(n) = O(n^2) + 2T\left( \frac{n}{\sqrt{2}} + 1 \right) = O(n^2 \log n)
  \]
  - Incl. \( \log^2 n \) iterations, total runtime is \( O(n^4 \log^3 n)! \)