Randomization

Algorithmic design patterns.
* Greedy.
* Divide-and-conquer.
* Dynamic programming.
* Network flow.
* Randomization.

Randomization. Allow fair coin flip in unit time.

Why randomize? Can lead to simplest, fastest, or only known algorithm for a particular problem.

Ex. Symmetry breaking protocols, graph algorithms, quicksort, hashing, load balancing, Monte Carlo integration, cryptography.

Max-3-SAT approximation algorithm

Expectation: two properties

Useful property. If $X$ is a 0/1 random variable, $E[X] = \Pr[X = 1]$.

Pf. $E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \sum_{j=0}^{\infty} j (1-p)^j p = \frac{p}{1-p} \cdot \sum_{j=0}^{\infty} (1-p)^j = \frac{p}{1-p} \cdot \frac{1}{p} = \frac{1}{p}$

Linear independence. Given two random variables $X$ and $Y$ defined over the same probability space, $E[X + Y] = E[X] + E[Y]$.

Benefit. Decouples a complex calculation into simpler pieces.
Guessing cards

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

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

**Claim.** The expected number of correct guesses is 1.

**Pf.** (surprisingly effortless using linearity of expectation)

- Let \( X_i = 1 \) if \( i \)th prediction is correct and 0 otherwise.
- Let \( X = X_1 + \ldots + X_n \).
- \( E[X_i] = P[X_i = 1] = 1/n \).
- \( E[X] = E[X_1] + \ldots + E[X_n] = 1/n + \ldots + 1/n = 1. \) □

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

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

**Pf.**

- Let \( X_i = 1 \) if \( i \)th prediction is correct and 0 otherwise.
- Let \( X = X_1 + \ldots + X_n \).
- \( E[X_i] = 1/(n-j-i) \).
- \( E[X] = E[X_1] + \ldots + E[X_n] = 1/n + \ldots + 1/2 + 1/1 = H(n) \).

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

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

**Pf.**

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

\[
E[X_i] = \sum_{j=1}^{n} \frac{n}{n-j} = n \sum_{j=1}^{n} \frac{1}{n-j} = nH(n) \\
\text{prob of success} = \frac{n-j}{n} \\
\text{expected waiting time} = \frac{n}{n-j} \\
\]

**Maximum 3-satisfiability.** Given a 3-SAT formula, find a truth assignment that satisfies as many clauses as possible.

**Remark.** NP-hard search problem.

**Simple idea.** Flip a coin, and set each variable true with probability \( \frac{1}{2} \), independently for each variable.

**Claim.** Given a 3-SAT formula with \( k \) clauses, the expected number of clauses satisfied by a random assignment is \( 7k/8 \).

**Pf.** Consider random variable \( Z_j = \begin{cases} 1 & \text{if clause } C_j \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases} \)

- Let \( Z = \) weight of clauses satisfied by assignment \( Z \).
- \( E[Z] = \sum_{j=1}^{n} \frac{1}{k} E[Z_j] \).

**The Probabilistic Method**

**Corollary.** For any instance of 3-SAT, there exists a truth assignment that satisfies at least a \( 7/8 \) fraction of all clauses.

**Pf.** Random variable is at least its expectation some of the time. □

**Probabilistic method.** [Paul Erdös] Prove the existence of a non-obvious property by showing that a random construction produces it with positive probability!
Maximum 3-satisfiability: analysis

Q. Can we turn this idea into a 7/8-approximation algorithm?
A. Yes (but a random variable can almost always be below its mean).

Lemma. The probability that a random assignment satisfies \( \geq \frac{7}{8} k \) clauses is at least \( \frac{1}{8^k} \).

**Pf.** Let \( p_j \) be probability that exactly \( j \) clauses are satisfied; let \( p \) be probability that \( \geq \frac{7}{8} k \) clauses are satisfied.

\[
\begin{align*}
\frac{7}{8} k &= E[X] \\
&= \sum_{j=0}^{\infty} j p_j \\
&= \sum_{j=0}^{\infty} j p_j + \sum_{j=1}^{k} j p_j - \sum_{j=1}^{k} j p_j \\
&< \left( \frac{7}{8} - \frac{1}{k} \right) k p + \frac{k}{8} \\
&= \left( \frac{k}{8} \right) - 1 + k p
\end{align*}
\]

Rearranging terms yields \( p \geq \frac{1}{8^k} \).

\[\blacksquare\]

Johnson's algorithm. Repeatedly generate random truth assignments until one of them satisfies \( \geq \frac{7}{8} k \) clauses.

Theorem. Johnson's algorithm is a 7/8-approximation algorithm.

**Pf.** By previous lemma, each iteration succeeds with probability \( \geq \frac{1}{8^k} \). By the waiting-time bound, the expected number of trials to find the satisfying assignment is at most \( 8^k \).

Dictionary data type

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

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

**Challenge.** Universe \( U \) can be extremely large so defining an array of size \(|U|\) is infeasible.

**Applications.** File systems, databases, Google, compilers, checksums, P2P networks, associative arrays, cryptography, web caching, etc.

Hashing

**Hash function.** \( h : U \rightarrow \{0, 1, \ldots, n-1\} \).

**Hashing.** Create an array \( H \) of size \( n \). When processing element \( u \), access array element \( H[h(u)] \).

**Collision.** When \( H[u] = H[v] \) but \( u \neq v \). (birthday paradox)

- A collision is expected after \( O(n^2) \) random insertions.
- Separate chaining: \( H[l] \) stores linked list of elements \( u \) with \( H[u] = l \).

**Ad-hoc hash function**

**Ad hoc hash function.**

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

**Deterministic hashing.** If \( |U| \geq n^2 \), then for any fixed hash function \( h \), there is a subset \( S \subseteq U \) of \( n \) elements that all hash to same slot. Thus, \( \Theta(n) \) time per search in worst-case.

Q. But isn't ad-hoc hash function good enough in practice?
Algorithmic complexity attacks

When can’t we live with ad hoc hash functions?

• Obvious situations: aircraft control, nuclear reactors.
• Surprising situations: denial-of-service attacks.

malicious adversary learns your ad hoc hash function in q, by reading Java APIs and causes a big pile-up in a single slot that grinds performance to a halt

Real world exploits. [Crosby-Wallach 2003]

• Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem
• Perf 5.8.0: insert carefully chosen strings into associative array.
• Linux 2.4.20 kernel: save files with carefully chosen names.

Surprising situations: denial of service attacks.

Obvious situations: aircraft control, nuclear reactors.

Designing a universal family of hash functions

Choose a prime number $p = n$. ...no need for randomness here

Integer encoding. Identify each element $u \in U$ with a base-$p$ integer of $r$ digits: $x = (x_1, x_2, ..., x_r).

Hash function. Let $A = \text{set of all} \ r \text{-digit, base-} p \text{ integers. For each} \ a = (a_0, a_1, ..., a_r) \text{ where } 0 \leq a < p,$ define

$h_a(x) = \left( \sum_{i=0}^{r} a_i \cdot x_i \right) \mod p$ 

Hash function family. $H = \{ h_a : a \in A \}$.

Hashing performance

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

• Running time depends on length of chains.
• Average length of chain $\approx m/n$.
• Choose $n = m$ so on average O(1) per insert, lookup, or delete.

Challenge. Achieve idealized randomized guarantees, but with a hash function where you can easily find items where you put them.

Approach. Use randomization in the choice of $h$.

adversary knows the randomized algorithm you’re using, but doesn’t know random choices that the algorithm makes.

Universal hashing

Universal family of hash functions. [Carter-Wegman 1980s]

• For any pair of elements $u, v \in U$,
• Can select random $h$ efficiently.
• Can compute $h(u)$ efficiently.

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


Universal hashing: analysis

Proposition. Let $H$ be a universal family of hash functions; let $h \in H$ be chosen uniformly at random from $H$ and let $u \in U$. For any subset $S \subseteq U$ of size at most $n$, the expected number of items in $S$ that collide with $u$ is at most 1.

Pf. For any element $x \in S$, define indicator random variable $X_u = 1$ if $h(x) = h(u)$ and 0 otherwise. Let $X$ be a random variable counting the total number of collisions with $u$.

$E[X] \leq |S| \leq 1$ 

Q. OK, but how do we design a universal class of hash functions?

Designing a universal family of hash functions

Theorem. [Chebyshev 1850] There exists a prime between $n$ and $2n$.

Modulus. Choose a prime number $p = n$. ...no need for randomness here

Integer encoding. Identify each element $u \in U$ with a base-$p$ integer of $r$ digits: $x = (x_1, x_2, ..., x_r).

Hash function. Let $A = \text{set of all} \ r \text{-digit, base-} p \text{ integers. For each} \ a = (a_0, a_1, ..., a_r) \text{ where } 0 \leq a < p,$ define

$h_a(x) = \left( \sum_{i=0}^{r} a_i \cdot x_i \right) \mod p$ 

Hash function family. $H = \{ h_a : a \in A \}$.
Load balancing

Chernoff Bounds (above mean)

**Theorem.** Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \ldots + X_n$. Then for any $\delta$ and for any $t > 0$, we have

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

**Proof.** We apply a number of simple transformations.

* For any $t > 0$,
  $$\Pr[X > (1 + \delta)E[X]] = \Pr\left[ e^{tX} > e^{t(1 + \delta)E[X]} \right] = e^{-t(1 + \delta)E[X]} \cdot E[e^{tX}]$$
  Fix $x = e^{tX}$ as function in $x$.
  Markov's inequality: $\Pr[X > a] \leq E[X]/a$

  * Now $E[e^{tX}] = E[e^{t\sum X_i}] = \prod_i E[e^{tX_i}]$ (definition of $X$)

* Combining everything:
  $$\Pr[X > (1 + \delta)E[X]] = e^{-t(1 + \delta)E[X]} \cdot \prod_i E[e^{tX_i}]$$

  Previous slide: Inequality above

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

Chernoff Bounds (below mean)

**Theorem.** Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \ldots + X_n$. Then for any $1 \leq E[X]$ and for any $0 < \delta < 1$, we have

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

**Proof idea.** Similar.

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

Load balancing

**Load balancing.** System in which $m$ jobs arrive in a stream and need to be processed immediately on $n$ identical processors. Find an assignment that balances the workload across processors.

**Centralized controller.** Assign jobs in round-robin manner. Each processor receives at most $\lceil m/n \rceil$ jobs.

**Decentralized controller.** Assign jobs to processors uniformly at random. How likely is it that some processor is assigned “too many” jobs?
Load balancing: many jobs

**Theorem.** Suppose the number of jobs $m = 16 n \ln n$. Then on average, each of the $n$ processors handles $\mu = 16 \ln n$ jobs. With high probability, every processor will have between half and twice the average load.

**Proof.**

1. Let $X_i, Y_{ij}$ be as before.
2. Applying Chernoff bounds with $\delta = 1$ yields

$$P[X_i > 2\mu] < \left(\frac{e}{2}\right)^{2\mu} = \left(\frac{e}{2}\right)^{16 n \ln n} = \frac{1}{n^4}$$

$$P[X_i < \frac{\mu}{2}] < e^{-\left(\frac{1}{2}\right) \ln 2} = \frac{1}{2}$$

3. Union bound: every processor has load between half and twice the average with probability $\geq 1 - 2/n$. 

---

**Course summary and review**

---

**Algorithmic design paradigms**

- Greedy (see: matroids)
- Divide and Conquer
- Dynamic Programming
- Flows, cuts and matchings
- Linear Programming

more sophisticated/general as go down list

---

**Fundamental algorithms**

- Graph traversals in $O(n + m)$ time
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

- applications:
  - BSF yields shortest paths in undirected graph
  - DFS used for topological sort and strongly connected components

---

**Algorithmic design paradigms**

- Many problems are NP-complete
- Unlikely to have solutions in P
- Coping with intractibility
  - special cases
  - fixed-parameter algorithms/analysis
  - approximation algorithms

---

**Fundamental algorithms**

- Single source shortest paths
  - with non-negative edge weights
    - Dijkstra's algorithm $O(n + m \log n)$
  - with negative edge weights
    - Bellman-Ford $O(nm)$ to detect negative cycles and find shortest paths if no negative cycles
Fundamental algorithms

• All-pairs shortest paths
  – Floyd-Warshall $O(n^3)$

• Minimum cost spanning tree
  – Kruskal $O(m \log m)$
  – Prim $O(m + n \log n)$

• compression via variable length coding
  – Huffman codes $O(n \log n)$

June 3, 2014
CS38 Lecture 19

Data structures

• binary min-heap
  – INSERT $O(\log n)$
  – EXTRACT-MIN $O(\log n)$
  – DECREASE-KEY $O(\log n)$

• Fibonacci heap (amortized analysis)
  – INSERT $O(1)$
  – EXTRACT-MIN $O(\log n)$
  – DECREASE-KEY $O(1)$

June 3, 2014
CS38 Lecture 19

Data structures

• Union-Find data structure
  – path compression
  – union-by-rank

  – amortized analysis:
    m find and n union operations in $O(m \log^* n)$

June 3, 2014
CS38 Lecture 19

Fundamental algorithms

• Sorting in $O(n \log n)$ time
  – heapsort
  – mergesort
  – quicksort with random pivot (expected time)
  – lower bound for comparison-based

• selection in $O(n)$ time
  – randomized and deterministic

June 3, 2014
CS38 Lecture 19

Fundamental algorithms

• closest pair of points in plane $O(n \log n)$

• integer multiplication $O(n^{\log_2 3})$

• matrix multiplication $O(n^{\log_2 2})$

• FFT $O(n \log n)$
  – polynomial multiplication and division with remainder $O(n \log n)$

June 3, 2014
CS38 Lecture 19

Fundamental algorithms

• two strings of length n, m:
  – edit distance $O(nm)$
  – longest-common-subsequence $O(nm)$

June 3, 2014
CS38 Lecture 19
Fundamental algorithms

• max-flow in a network (= min-cut)
  – Ford-Fulkerson method $O(m \ nC)$
  – capacity-scaling $O(m^2 \ \log C)$
  – shortest augmenting path $O(m^2 \ n)$
  – blocking-flow implementation $O(mn^3)$
  – use last for bipartite matching in same time
  – min/max weight perfect matching $O(n^3)$

Fundamental algorithms

• Linear programming
  – primal/dual and strong duality theorem
  – simplex algorithm (worst case exponential)
  – ellipsoid algorithm (in P)
  – in practice: interior points methods

Fundamental algorithms

• Coping with intractibility
  – e.g.: hard graph problems easy on trees
  – e.g.: fixed parameter algorithms for VC

• approximation algorithms
  – knapsack $(1 + \epsilon)$
  – VC and weighted VC $2$ (via LP relaxation)
  – set cover in $m + 1$
  – TSP $1.5$
  – center selection $2$

Fundamental algorithms

• randomized algorithm for global min-cut
• $8/7$ approximation for max-3-sat

• other applications:
  – contention resolution
  – hashing
  – load-balancing
  – …