Lecture 5: Hashing

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 25, 2021

Overview

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

Computational Model

Before we talk about hash functions, we need to state our model of computation:

Definition (Word RAM model)

In the word RAM[a] model:

- all elements are integers that fit in a machine word of w bits
- Basic operations (comparison, arithmetic, bitwise) on such words take $\Theta(1)$ time
- We can also access *any* position in the array in $\Theta(1)$ time

[a] RAM stands for Random Access Model

What is hashing?

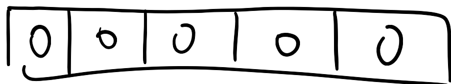
We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

$m < 2^w$ $\quad\therefore$ can store each element of U in one word

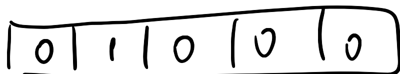
What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.



insert(1)



What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.

- Insertion: $O(1)$, Deletion: $O(1)$, Search: $O(1)$
- Memory: $O(m \log(m))$ *(this is very bad!)*

if assume $\log(m) < w$

then $O(m)$ space

What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.

- Insertion: $O(1)$, Deletion: $O(1)$, Search: $O(1)$
- Memory: $O(m\log(m))$ *(this is very bad!)*

Want to also achieve optimal memory $O(n\log(m))$. For this we will use a technique called *hashing*.

or be as close as possible

- A *hash function* is a function $h : U \to [0, n-1]$, where $|U| = m >> n$.
- A *hash table* is a data structure that consists of:
  - a table T with n cells $[0, n-1]$, each cell storing $O(\log(m))$ bits
  - a hash function $h : U \to [0, n-1]$

From now on, we will define memory as *# of cells*.

Why is hashing useful?

- Designing efficient data structures (dictionaries) for searching
- Data streaming algorithms
- Derandomization

  When randomness needed only involves that pairs, or triples, or *small number of elements* "*look independent*."
- Cryptography

  Construct functions that look random to adversaries, but are easy for us to compute.
- Complexity Theory
- many more

Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \rightarrow [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

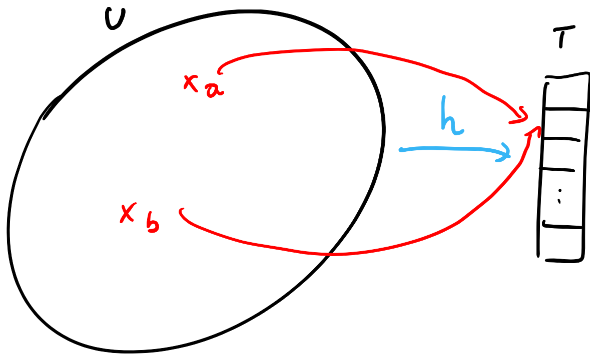
Ideally, want hash function to map *different keys* into *different locations*.

Definition (Collision)

A *collision* happens for hash function h with inputs $x, y \in U$ if

$$x \neq y \quad \text{and} \quad h(x) = h(y).$$

Pigeonhole principle $\Rightarrow$ impossible without knowing keys in advance.



$|U| > |T|$

Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

Ideally, want hash function to map *different keys* into *different locations*.

Definition (Collision)

A *collision* happens for hash function h with inputs $x, y \in U$ if

$$x \neq y \quad \text{and} \quad h(x) = h(y).$$

Pigeonhole principle $\Rightarrow$ impossible without knowing keys in advance.

Will settle for: # collisions *small with high probability*.

Definition (perfect hashing): given a set $S \subset U$ and hash function $h : U \to [0, .., n-1]$

h is <u>perfect</u> for S if $\forall\, x, y \in S$, $x \neq y$

$$h(x) \neq h(y)$$

i.e. there are no collisions.

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Simplest version to keep in mind:

$$\Pr_{h \in_R \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{\text{poly}(n)} \qquad \forall x \neq y \in U$$

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Simplest version to keep in mind:

$$\Pr_{h \in_R \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{\text{poly}(n)} \qquad \forall x \neq y \in U$$

Assumptions:

- keys are independent from hash function we choose.
- we **do not** know keys in advance (even if we did, nontrivial problem!)

Question

Still could have collisions. How do we handle them?

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!

$$h : [0, m-1] \to [0, n-1]$$

balls — bins

taking n labelled balls from U ⇔ randomly throwing (unlabeled) n balls into bins (+ h)

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1
- maximum number of collisions (max load) in one particular location: $O(\log n / \log \log n)$ keys

Random Hash Functions?

Natural to consider following approach:

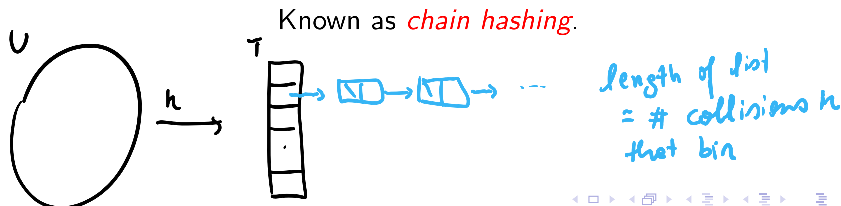
From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1
- maximum number of collisions (max load) in one particular location: $O(\log n / \log \log n)$ keys

Solving collisions: store all keys hashed into location i by a linked list.

Known as *chain hashing*.



Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1
- maximum number of collisions (max load) in one particular location: $O(\log n / \log \log n)$ keys

Solving collisions: store all keys hashed into location i by a linked list.

Known as *chain hashing*.

Could also pick *two* random hash functions and use *power of two choices*.
Collision bound becomes $O(\log \log n)$.

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(n)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

$h(1) = 2 \quad h(3) = 1 \quad h(17) = 4 \ldots$

17 comes again need to check whether I already defined it

($O(n)$ time)

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(n)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

Remark

Thus, for random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(n)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

Remark

Thus, for random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

How do we cope with the computational problem that arose with randomness?

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (size of our input).

" word size "
$O(1)$

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (size of our input).

Question

How many hash functions can we have with the property above?

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (size of our input).

Question

How many hash functions can we have with the property above?

poly(m) functions, as each function takes at most $O(\log m)$ bits to describe. Thus these are *succinct functions* (easy to describe and compute) which have *random-like* properties!

$$2^{O(\log m)} = 2^{c \cdot \log m} = m^c = \text{poly}(m)$$

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (size of our input).

Question

How many hash functions can we have with the property above?

poly(m) functions, as each function takes at most $O(\log m)$ bits to describe. Thus these are *succinct functions* (easy to describe and compute) which have *random-like* properties!

Part of *derandomization/pseudorandomness*: huge subfield in TCS!

k-wise independence

Weaker notion of independence.

"sacrifice some randomness to get better computation"

k-wise independence

Weaker notion of independence.

Definition (Full Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be (fully) independent if they satisfy

$$\Pr\left[\bigcap_{i=1}^{n} X_i = a_i\right] = \prod_{i=1}^{n} \Pr[X_i = a_i]$$

$\forall a_i$

↳ in range of X_i

k-wise independence

Weaker notion of independence.

Definition (Full Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be (fully) independent if they satisfy

$$\Pr\left[\bigcap_{i=1}^{n} X_i = a_i\right] = \prod_{i=1}^{n} \Pr[X_i = a_i]$$

Definition (k-wise Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be k-wise independent if for any set $J \subset [n]$ such that $|J| \leq k$ they satisfy

→ small subsets of our random variables

$\forall a_i$ in range X_i

$$\Pr\left[\bigcap_{i \in J} X_i = a_i\right] = \prod_{i \in J} \Pr[X_i = a_i]$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly distributed, independent random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent, uniform random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly distributed, independent random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent, uniform random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly distributed, independent random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent, uniform random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they uniformly random?

$S = \{1, 2, 3\}$

$X_S = Y_1 \oplus Y_2 \oplus Y_3 \rightarrow \begin{cases} 0 & \text{w.p. } 1/2 \\ 1 & \text{w.p. } 1/2 \end{cases}$

$Y_1 \oplus Y_2 = a$, $a \in \{0, 1\}$; Y_3: $1/2 \to a$, $1/2 \to 1-a$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly distributed, independent random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent, uniform random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they uniformly random?
- Why are they pairwise independent?

$$\Pr[X_{S_1} = a_1 \text{ and } X_{S_2} = a_2] = \Pr[X_{S_1} = a_1] \cdot \Pr[X_{S_2} = a_2]$$

$S_1 \neq S_2 \quad \exists\, j \in S_2 \setminus S_1$

even if we set all variables in S_1

$X_{S_2} = Y_j \oplus [\quad]$ (not set; $\frac{1}{2} \to 0$, $\frac{1}{2} \to 1$)

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly distributed, independent random bits $Y_1, \dots, Y_b$, we can generate $2^b - 1$ pairwise independent, uniform random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they uniformly random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

$X_{\{1\}}$ $\quad X_{\{2\}}$ $\quad X_{\{1,2\}}$

0 $\quad 0$ $\quad \Rightarrow 0$ w.p. 1

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ (integers mod p)

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they uniformly random?

$X_1 = Y_1 + Y_2 \mod p$

$a \in \mathbb{F}_p \quad (x, a-x)$

have p pairs (Y_1, Y_2) s.t. $X_1(Y_1, Y_2) = a$

$\Pr[X_1 = a] = \frac{p}{p^2} = \frac{1}{p}$

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they uniformly random?
- Why are they pairwise independent?

$\Pr[X_i = a_i \wedge X_j = a_j] = \frac{1}{p^2}$

$Y_1 + i Y_2 = a_i \mod p$

$Y_1 + j Y_2 = a_j \mod p$

} system of 2 equations (linearly independent) in 2 variables

$(j-i) Y_2 = (a_j - a_i)$, $j - i \neq 0$

$\therefore$ we have exactly one solution

$\therefore \Pr[X_i = a_i \wedge X_j = a_j] = \frac{1}{p^2}$

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they uniformly random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

NO.

$X_0 = 0$
$X_1 = 0$
$X_2 = 1$

Prob. 0 of happening

(inconsistent system of equations)

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they uniformly random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

Can think of these random variables as picking a random line.

- If we only know one point of the line, the second point is still uniformly random. *pairwise independence*
- Two points determine the line. *not 3-wise independent*

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Definition (Universal Hash Functions)

Let U be a universe with $|U| \geq n$. A family of hash functions $\mathcal{H} = \{h : U \to [0, n-1]\}$ is *k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$, we have

$$\Pr_{h \in_R \mathcal{H}} [h(u_1) = h(u_2) = \ldots = h(u_k)] \leq 1/n^{k-1}$$

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Definition (Universal Hash Functions)

Let U be a universe with $|U| \geq n$. A family of hash functions $\mathcal{H} = \{h : U \to [0, n-1]\}$ is *k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$, we have

$$\Pr_{h \in_R \mathcal{H}} [h(u_1) = h(u_2) = \ldots = h(u_k)] \leq 1/n^{k-1}$$

Definition (Strongly Universal Hash Functions)

$\mathcal{H} = \{h : U \to [0, n-1]\}$ is *strongly k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$ and *for any* values $y_1, \ldots, y_k \in [0, n-1]$, we have

$$\Pr_{h \in_R \mathcal{H}} [h(u_1) = y_1, \ldots, h(u_k) = y_k] \leq 1/n^k$$

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Family $\mathcal{H}$ is *strongly k-universal* if the random variables $h(0), \ldots, h(|U| - 1)$ are *k-wise independent*.

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Family $\mathcal{H}$ is *strongly k-universal* if the random variables $h(0), \ldots, h(|U| - 1)$ are *k-wise independent*.

Can use random variables to construct universal hash functions!

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]$.

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := a \cdot x + b \mod p \quad | \quad a, b \in [0, p-1]\}$$

is strongly 2-universal.

$h_{a,b}(i) = a \cdot i + b$ $h_{a,b}(j) = a \cdot j + b$

Same proof we did before.

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]$.

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := a \cdot x + b \mod p \quad | \quad a, b \in [0, p-1]\}$$

is strongly 2-universal.

How do we make the domain U much larger than image of the maps? (as usually in hashing size of universe much larger than size of table)

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]$.

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := a \cdot x + b \mod p \mid a, b \in [0, p-1]\}$$

is strongly 2-universal.

How do we make the domain U much larger than image of the maps? (as usually in hashing size of universe much larger than size of table)

Proposition

Let $U = [0, p^k - 1] \equiv [0, p-1]^k$ *and* $\mathrm{a} = (a_0, \ldots, a_{k-1}) \in \mathbb{F}_p^k$

$\mathbb{F}_{p^k} \equiv \mathbb{F}_p^k$

$$\mathcal{H} = \{h_{\mathrm{a},b}(\mathrm{x}) := \langle \mathrm{a}, \mathrm{x} \rangle + b \mod p \mid \mathrm{a} \in U, b \in [0, p-1]\}$$

$\hookrightarrow a_0 \cdot x_0 + a_1 x_1 + \cdots + a_{k-1} \cdot x_{k-1}$

is strongly 2-universal.

Strongly 2-universal families of hash functions

Proposition

Let $U = [0, p^k - 1] \equiv [0, p-1]^k$ *and* $\mathrm{a} = (a_0, \ldots, a_{k-1})$

$$\mathcal{H} = \{h_{\mathrm{a},b}(\mathrm{x}) := \langle \mathrm{a}, \mathrm{x} \rangle + b \mod p \quad | \quad \mathrm{a} \in U, b \in [0, p-1]\}$$

is strongly 2*-universal.*

Proof similar to first proof (subsets)

(there $p=2$)

2-universal families of hash functions

$\mathbb{F}_p \to \mathbb{F}_p \to \mathbb{Z}/n\mathbb{Z}$ simple hash function

$\mathbb{F}_p = U$

What if my hash table size is not a prime?

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \quad | \quad a, b \in [0, p-1], a \neq 0\}$$

$[0, p-1] = \mathbb{F}_p = U$

is 2-universal (but not strongly 2-universal).

Practice problem: prove the proposition above.

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k - 1$

$$h_{a,b}(x) = ax + b$$

$$h_{a,b,c}(x) = ax^2 + bx + c$$ 3-wise independent

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k - 1$
- Two points determine a line. Similarly, k points determine a univariate polynomial of degree $k - 1$

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k - 1$
- Two points determine a line. Similarly, k points determine a univariate polynomial of degree $k - 1$
- Random degree $k - 1$ polynomials are k-wise independent!
- Practice problem: prove this!

Efficiency

How did pairwise independence improve over random functions?

Efficiency

How did pairwise independence improve over random functions?

Remark

For random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!) to compute. And it takes $O(n \log m)$ storage!

Efficiency

How did pairwise independence improve over random functions?

Remark

For random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!) to compute. And it takes $O(n \log m)$ storage!

Remark

- In XOR example, our function takes $O(b)$ storage space, and $O(b)$ time to compute.[a]

  Store set S $O(b \log b)$ get b bits evaluate $\oplus$ Y_i ies

- In $\mathbb{F}_p$ examples, our function takes $O(1)$ storage space and $O(1)$ time to compute![b]

  Store a, b defining $h_{a,b}$ compute $x \mapsto ax+b$

---

[a] Reminder that we assume that $b < w$.

[b] We assume that $p < 2^w$.

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$
- Only need $0 \neq a$ and $b \in [0, p-1]$ to store a function from $\mathcal{H}$.

$O(\log m)$ storage space

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$
- Only need $0 \neq a$ and $b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$
- Only need $0 \neq a$ and $b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ search time)

No.

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$
- Only need $0 \neq a$ and $b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ search time)

Do not have same expected search time as chain hashing.

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod n \mid a, b \in [0, p-1],\ a \neq 0\}$
- Only need $0 \neq a$ and $b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ search time) ↳ using two random hash functions

  Do not have same expected search time as chain hashing.

Lemma (Maximum number of collisions)

The expected number of collisions [pairs] *when hashing* ℓ *elements using a 2-universal hash family is*

$$\leq \ell^2/2n$$

Hashing with 2-universal families

Lemma (Maximum number of collisions)

The expected number of collisions when hashing ℓ elements using a 2-universal hash family is

$$\ell^2/2n$$

Hashing with 2-universal families

Lemma (Maximum number of collisions)

The expected number of collisions when hashing ℓ elements using a 2-universal hash family is

$$\ell^2/2n$$

Thus, by Markov's inequality, we have $Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$

Lemma (Maximum load of entry of hash table)

With probability $\geq 1/2$ the ~~number of collisions~~ max load when hashing ℓ elements using a 2-universal hash family is

$$\leq \sqrt{\frac{2\ell^2}{n}}.$$

When $\ell \approx n$ (as is usually assumed in hashing), we expect $\sqrt{2n}$.

Can prove max load is $\sqrt{\text{max \# collisions}}$

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

Perfect Hashing

- **Setup:** we are given *in advance* a static set $S \subset U$ of size n
- How to build a hash table with $O(1)$ search time and $O(n)$ memory?
- Can we still do it with a 2-universal family of hash functions?

Perfect Hashing

- **Setup:** we are given *in advance* a static set $S \subset U$ of size n
- How to build a hash table with $O(1)$ search time and $O(n)$ memory?
- Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof: There is no collision with probability $\geq 1/2$.

Perfect Hashing

- **Setup:** we are given *in advance* a static set $S \subset U$ of size n
- How to build a hash table with $O(1)$ search time and $O(n)$ memory?
- Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof: There is no collision with probability $\geq 1/2$.

New idea: build a *two-level* hash table!

Perfect Hashing

- **Setup:** we are given *in advance* a static set $S \subset U$ of size n
- How to build a hash table with $O(1)$ search time and $O(n)$ memory?
- Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof: There is no collision with probability $\geq 1/2$.

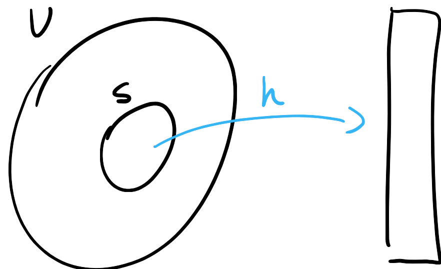
New idea: build a *two-level* hash table!

Theorem

The two-level approach gives perfect hashing scheme.

Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.

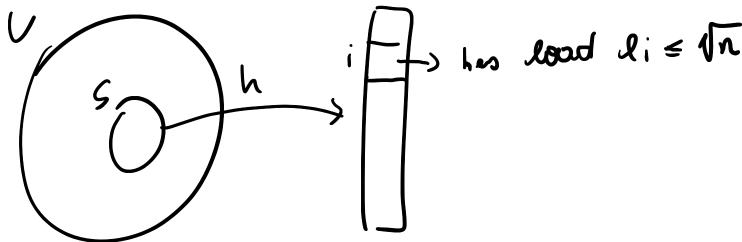


Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)

Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)
- Assume max number of collisions (h, S) is $\leq \sqrt{n}$. Let ℓ_i be the load at i^{th} cell of hash table given by h



Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)
- Assume max number of collisions (h, S) is $\leq \sqrt{n}$. Let ℓ_i be the load at i^{th} cell of hash table given by h
- Thus $\ell_i \leq \sqrt{n}$ and $\sum_{i=1}^{n} \ell_i = |S| = n$

Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)
- Assume max number of collisions (h, S) is $\leq \sqrt{n}$. Let ℓ_i be the load at i^{th} cell of hash table given by h
- Thus $\ell_i \leq \sqrt{n}$ and $\sum_{i=1}^{n} \ell_i = |S| = n$
- By our lemma, if take $h_i : S \to [\ell_i^2]$ from our 2-universal hash family, h_i is perfect with high probability

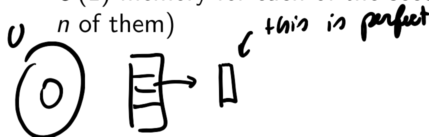
for S_i

set elements mapped to i

$(|S_i| = \ell_i \leq \sqrt{n})$

Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)
- Assume max number of collisions (h, S) is $\leq \sqrt{n}$. Let ℓ_i be the load at i^{th} cell of hash table given by h
- Thus $\ell_i \leq \sqrt{n}$ and $\sum_{i=1}^{n} \ell_i = |S| = n$
- By our lemma, if take $h_i : S \to [\ell_i^2]$ from our 2-universal hash family, h_i is perfect with high probability
- *Memory needed:* to store all hash functions, need $O(n)$ memory
  - $O(1)$ memory for the top level hash function h
  - $O(1)$ memory for each of the second-layer hash functions (and we have n of them)



Proof (sketch) of Theorem

- Pick first layer hash function h uniformly at random from our 2-universal family $\mathcal{H}$. Test h on our set S.
- With probability $\geq 1/2$, max number of collisions in one bin is $\leq \sqrt{n}$. Thus, we will find good hash function for first layer with constant many tries. (with high probability)
- Assume max number of collisions (h, S) is $\leq \sqrt{n}$. Let ℓ_i be the load at i^{th} cell of hash table given by h
- Thus $\ell_i \leq \sqrt{n}$ and $\sum_{i=1}^{n} \ell_i = |S| = n$
- By our lemma, if take $h_i : S \to [\ell_i^2]$ from our 2-universal hash family, h_i is perfect with high probability
- *Memory needed:* to store all hash functions, need $O(n)$ memory
  - $O(1)$ memory for the top level hash function h
  - $O(1)$ memory for each of the second-layer hash functions (and we have n of them)
- *Time to hash*: $O(1)$ time to evaluate each hash function, and we only have two layers. So total time $O(1)$

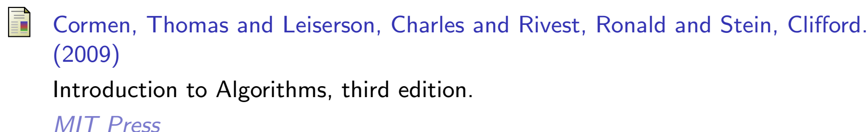
Acknowledgement

- Lecture based largely on Lap Chi's notes and on [CLRS 2009, Chapter 11].
- See Lap Chi's notes at https://cs.uwaterloo.ca/~lapchi/cs466/notes/L05.pdf

References I

Motwani, Rajeev and Raghavan, Prabhakar (2007)
Randomized Algorithms

Mitzenmacher, Michael, and Eli Upfal (2017)
Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.
Cambridge university press, 2017.


Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)
Introduction to Algorithms, third edition.
MIT Press