

DSC 40B

Theoretical Foundations II

Hashing

Hashing

- ▶ One of the most important ideas in CS.
- ▶ Tons of uses:
 - ▶ Verifying message integrity.
 - ▶ Fast queries on a large data set.
 - ▶ Identify if file has changed in version control.

Hash Function

- ▶ A **hash function** takes a (large) object and returns a (smaller) “fingerprint” of that object.

How?

- ▶ Looking at certain bits, combining them in ways that look random.

Hash Function Properties

- ▶ Hashing same thing twice returns the same hash.
- ▶ Unlikely that different things have same fingerprint.
 - ▶ But not impossible!

Example

- ▶ MD5 is a **cryptographic** hash function.
 - ▶ Hard to “reverse engineer” input from hash.
- ▶ Returns a *really large* number in hex.

a741d8524a853cf83ca21eabf8cea190

- ▶ Used to “fingerprint” whole files.

Example

```
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin!" | md5  
f11eed2391bbdoa5a2355397c089fafd
```

Example

```
> md5 slides.pdf  
e3fd4370fda30ceb978390004e07b9df
```

Why?

- ▶ I release a piece of software.
- ▶ I host it on Google Drive.
- ▶ Someone (Google, US Gov., etc.) decides to insert extra code into software to spy on users.
- ▶ You have no way of knowing.

Why?

- ▶ I release a piece of software & **publish the hash**.
- ▶ I host it on Google Drive.
- ▶ Someone inserts extra code.
- ▶ You download the software and hash it. If hash is different, you know the file has been changed!

Another Use

- ▶ Want to place images into 100 bins.
- ▶ How do we decide which bin an image goes into?
- ▶ Hash function!
 - ▶ Takes in an image.
 - ▶ Outputs a number in $\{1, 2, \dots, 100\}$.

Hashing for Data Scientists

- ▶ Don't need to know much about *how* hash function works.
- ▶ But should know how they are used.

DSC 40B

Theoretical Foundations II

Hash Tables

Membership Queries

- ▶ **Given:** a collection of n numbers and a target t .
- ▶ **Find:** determine if t is in the collection.

Goal

- ▶ We want a strategy that supports fast queries, insertions.

Approach #1

- ▶ Store data in a linked list.
- ▶ Initial cost: $\Theta(n)$.
- ▶ Query: linear search, $\Theta(n)$.
- ▶ Insertion: $\Theta(1)$.

Approach #2

- ▶ Store data in a **sorted** linked list.
- ▶ Initial cost: $\Theta(n \log n)$.
- ▶ Query: binary search, $\Theta(\log n)$.
- ▶ Insertion: $\Theta(?)$

Approach #3: Direct Address Tables

- ▶ Example: zip codes. E.g., 92124
- ▶ Query: is a given zip code in the data set?
- ▶ Observation: zip codes are between 0 and 99,999

Direct Address Tables

- ▶ Idea: keep an **array** of 100,000 entries.
- ▶ If we see zip code x , mark $\text{arr}[x]$ as one.

```
# loading the table
table = np.zeros(100_000)

for zip_code in data:
    table[zip_code] = 1
```

```
# query: is 43201 in data?  
if table[43201] == 1:  
    print('Yes')  
else:  
    print('No')
```

```
# insertion: insert 43201
table[43201] = 1:
```

Exercise

What is the time complexity of querying a DAT?
What about insertion?

Analysis

- ▶ Query: $\Theta(1)$.
- ▶ Insertion: $\Theta(1)$
 - ▶ As long as it is “in bounds”.

Exercise

What is the biggest problem with a Direct Address Table?

Memory

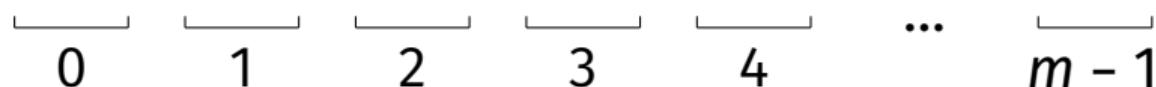
- ▶ Lots of wasted space.
- ▶ Example: a DAT for phone numbers requires 9,999,999,999 entries.

Approach #4: Hash Tables

- ▶ Pick a table size m .
 - ▶ Usually $m \approx$ number of things you'll be storing.
- ▶ Create hash function to turn input into a number in $\{0, 1, \dots, m - 1\}$.
- ▶ Create DAT with m bins.

Example

```
hash('hello') == 3  
hash('data') == 0  
hash('science') == 4
```



Collisions

- ▶ The **universe** is the set of all possible inputs.
- ▶ This is usually much larger than m (even infinite).
- ▶ Not possible to assign each input to a unique bin.
- ▶ If $\text{hash}(a) == \text{hash}(b)$, there is a **collision**.

Chaining

- ▶ Collisions stored in same bin, in linked list.
- ▶ **Query:** Hash to find bin, then linear search.



The Idea

- ▶ A good hash function will utilize all bins evenly.
 - ▶ Looks like uniform random distribution.
- ▶ If $m \approx n$, then only a few elements in each bin.
- ▶ As we add more elements, we need to add bins.

Average Case

- ▶ n elements in bin.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins¹.
- ▶ Expected bin size: n/m .

¹Of course, they are placed deterministically.

Analysis

- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n/m)$ for linear search.
 - ▶ Total: $\Theta(1 + n/m)$.
 - ▶ We usually guarantee $m = O(n)$, $\implies \Theta(1)$.
- ▶ Insertion: $\Theta(1)$.

Worst Case

- ▶ Everything hashed to same bin.
 - ▶ Really unlikely!
 - ▶ Adversarial attack?
- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n)$ for linear search.
 - ▶ Total: $\Theta(n)$.

Worst Case Insertion

- ▶ We need to ensure that $m \leq c \cdot n$.
 - ▶ Otherwise, too many collisions.
- ▶ If we add a bunch of elements, we'll need to increase m .
- ▶ Increasing m means allocating a new array, $\Theta(m) = \Theta(n)$ time.

Main Idea

Hash tables support constant (expected) time insertion and membership queries.

Dictionaries

- ▶ Hash tables can also be used to store (key, value) pairs.
- ▶ Often called **dictionaries** or **associative arrays**.

Hashing in Python

- ▶ `dict` and `set` implement hash tables.
- ▶ Querying is done using `in`:

```
>>> # make a set
>>> L = {3, 6, -2, 1, 7, 12}
>>> 1 in L # Theta(1)
False
>>> 7 in L # Theta(1)
True
```

DSC 40B

Theoretical Foundations II

Fast Algorithms with Hash Tables

Faster Algorithms

- ▶ Hashing is a super common trick.
- ▶ The “best” solution to interview problems often involves hashing.

Example 1: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose durations sum to **exactly** D .

Recall: Previous Solutions

- ▶ Brute force: $\Theta(n^2)$.
- ▶ Sort, use sorted structure: $\Theta(n \log n) + \Theta(n)$.
- ▶ Theoretical lower bound: $\Omega(n)$?
- ▶ Can we speed this up with hash tables?

Idea

- ▶ To use hash tables, we want to frame problem as a **membership query**.

Example

- ▶ Suppose flight is 360 minutes long.
- ▶ Suppose first movie is fixed: 120 minutes.
- ▶ Is there a movie lasting $(360 - 120) = 140$ minutes?

```
def optimize_entertainment_hash(times, D):
    hash_table = dict()
    for i, time in enumerate(times):
        hash_table[time] = i

    for i, time in enumerate(times):
        target = D - time
        if target in hash_table:
            return i, hash_table[target]
```

Example 2: Anagrams

Definition

Two strings w_1 and w_2 are **anagrams** if the letters of w_1 one can be permuted to make w_2 .

Examples

- ▶ abcd / dbca
- ▶ listen / silent
- ▶ sandiego / doginsea

Problem

- ▶ Given a collection of n strings, determine if any two of them are anagrams.

Exercise

Design an efficient algorithm for solving this problem. What is its time complexity?

Solution

- ▶ We need to turn this into a **membership query**.
- ▶ **Trick:** two strings are anagrams iff

```
sorted(w_1) == sorted(w_2)
```

```
def any_anagrams(words):
    seen = set()
    for word in words:
        w = sorted(word)
        if w in seen
            return True
        else:
            seen.add(w)
```

Hashing **Downsides**

- ▶ Problem must involve **membership query**.

Example: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose added durations is **closest** to D .

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ There is no way to quickly query entry closest to given input.

Example: Number of Elements

- ▶ Given a collection of n numbers and two endpoints, a and b , determine how many of the numbers are contained in $[a, b]$.
- ▶ Not a membership query.
- ▶ Idea: **sort** and use modified binary search.

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ But we often want similar items at the same time.
- ▶ Results in many **cache misses, slow.**

Hashing **Downsides**

- ▶ Memory overhead.