Hashing

COMPSCI 230 — Discrete Math

April 18, 2017
Outline

1. Number of Trials to First Success (from last time)

2. Case Study: Analysis of Hashing
   Associative Storage and Retrieval
   Hash Tables
Number of Trials to First Success

- Define a random variable on a Bernoulli trial:
  \[ N(C) = n \text{ iff the first } H \text{ is in trial } n \in \mathbb{N} \]
- \[ N = n \text{ iff } n - 1 T's \text{ are followed by one } H \]
- [and what happens after that does not matter]
- \[ N : S^\infty \rightarrow \mathbb{R} \] (as a r.v. should!)
- Example: \( n = 4 \). \((T, T, T, H, \ldots)\)
- Because of independence, \( P(N = n) = q^{n-1}p \)
- **Geometric distribution** (no upper bound on \( n \))
- \( \mathbb{E}[N] \): expected number of trials until the first success
- How many times do you need to flip a coin on average until it comes up \( H \)?
A Summation Lemma

• For $0 < x < 1$

\[
\sum_{n=0}^{\infty} nx^n = \frac{x}{(1 - x)^2}
\]

• Proof: We know that for $0 < x < 1$

\[
\sum_{n=0}^{\infty} x^n = \frac{1}{1 - x}
\]

\[
\sum_{n=0}^{\infty} nx^n = x \sum_{n=0}^{\infty} nx^{n-1} = x \sum_{n=0}^{\infty} \frac{d}{dx} x^n = x \frac{d}{dx} \sum_{n=0}^{\infty} x^n
\]

\[
= x \frac{d}{dx} \frac{1}{1 - x} = x \frac{1}{(1 - x)^2} = \frac{x}{(1 - x)^2}
\]

\[\square\]
Expected Number of Trials to First Success

- The expected number of trials to first success in a Bernoulli experiment with success probability $p$ is

$$\mathbb{E}[N] = \frac{1}{p}$$

- Proof: Let $N(C) = n$ iff the first $H$ is in trial $n \in \mathbb{N}$

$$\mathbb{E}[N] = \sum_{C \in \mathbb{S}^\infty} N(C) P(C) = \sum_{n=1}^{\infty} n P(N = n)$$

$$= \sum_{n=1}^{\infty} n q^{n-1} p = \frac{p}{q} \sum_{n=0}^{\infty} n q^n$$

(from lemma) $$= \frac{p}{q} \frac{q}{(1-q)^2} = \frac{p}{q} \frac{q}{p^2} = \frac{1}{p}$$

- Application case study: analysis of hashing
Associative Storage and Retrieval

- Store and lookup data items by key. Examples:
  - Purchase orders by customer
  - Student record by last name
  - Catalog by model number

- Associate a **key** to each **value**. In Python:
  ```python
  >>> grades = dict()
  >>> grades['Smith'] = [87, 80, 93]
  >>> grades['Brown'] = [82, 80, 78]
  >>> grades['Brown']
  [82, 80, 78]
  >>> grades['Jones']
  KeyError: 'Jones'
  ```
Implementation 0

• Insert \((key, value)\) pairs in a list as they come in
• Lookup: Search the list sequentially until the \texttt{key} is found (return \texttt{value}) or the end of the list is reached (return \texttt{None})

```python
T0 = []

def insert0(key, value, table=T0):
    table.append((key, value))

def lookup0(key, table=T0):
    for item in table:
        if item[0] == key:
            return item[1]
    return None
```

• [Also need \texttt{makeTable, reset, delete, ...}]

COMPSCI 230 — Discrete Math  Hashing  April 18, 2017  7 / 15
Implementation 0 $\rightarrow$ Implementation 2

- **Functionality issue**: if you insert two values with the same key, you only get the first one back
- **Desired behavior?** Overwrite? Raise error? Ignore second value? Store and return both?
- **Efficiency**:
  - Insertion is straightforward and efficient
  - Lookup takes on average $n/2$ steps if there are $n$ items in the table [but what does this really mean?]
  - Lookup bogs down when $n$ is large
- **Improvement 1**: If key already exists, overwrite the value (no garbage accumulation)
- **Improvement 2**: Insert `(key, value)` pairs so that keys are sorted (insertion slower, lookup faster)
Implementation 1: Overwrite Duplicates

T1 = []

def insert1(key, value, table=T1):
    for i in range(len(table)):
        if table[i][0] == key:
            table[i] = (key, value)
    return

    table.append((key, value))

    return lookup1(key, table=T1):
        return lookup0(key, table)
Sketch of Implementation 2

- Main idea: keys are kept in sorted order
- There must be an ordering for keys
- To insert (recursive definition):
  - If table is empty, insert \((\text{key}, \text{value})\)
  - Otherwise
    - Look up key in the middle of the table
    - If same as \text{key}, overwrite \text{value}
    - Otherwise:
      - If \text{key} belongs to the left half, insert it there
      - Otherwise, insert it in the right half

- To look up, do the same, but return \text{value} instead of inserting or overwriting
- If table is empty on lookup, return \text{None}
- \text{Binary search}
Implementation 2: Sorted Keys

T2 = []

def insert2(key, value, start=0, finish=None, table=T2):
    if finish is None: finish = len(table)
    if start == finish: table.insert(start, (key, value))
    else:
        middle = (start + finish) // 2
        if table[middle][0] == key:
            table[middle] = (key, value)
        elif table[middle][0] < key:
            insert2(key, value, middle+1, finish)
        else: insert2(key, value, start, middle)

def lookup2(key, start=0, finish=None, table=T2):
    if finish is None: finish = len(table)
    if start == finish: return None
    else:
        middle = (start + finish) // 2
        if table[middle][0] == key: return table[middle][1]
        elif table[middle][0] < key:
            return lookup2(key, middle+1, finish)
        else: return lookup lookup2(key, start, middle)
Implementation 3: Python `dict`

- It takes time proportional to $\log(\text{len(table)})$ to insert/lookup an item
- Lookup and insertion are faster
- We can do better in practice

```python
T3 = dict()

def insert3(key, value, table=T3):
    table[key] = value

def lookup3(key, table=T3):
    try:
        return table[key]
    except KeyError:
        return None
```

- In practice, this is faster than binary search
- Yes, but how does Python implement a `dict`?
Hashing

- Let $\text{index} = h(\text{name})$ be the last two digits of a student’s unique ID
- $h$ is an example of a hash function
- Make table an array of $b = 100$ lists, each initially empty
- Each list is called a bucket
- Insert: Append $(\text{key}, \text{value})$ to bucket $h(\text{key})$
- Lookup: Search bucket $h(\text{key})$ sequentially for the key
- Just as in implementation 1, but $b$ lists instead of one, and a hash function
Case Study: Analysis of Hashing

Hash Tables

Technicallity: Hash Keys

- \textbf{index} = h(name) = last two digits of SUID
- Requires access to the Duke directory (slow)
- Simple alternatives for a $b$-bucket hash function $h$:
  - Add ASCII codes for name characters, then modulo $b$
    (best if $b$ is prime)
  - Exclusive-OR of ASCII codes for name characters
    (no modulo needed if 256 buckets are OK)

- \textbf{Good hash function generates all bucket indices with equal likelihood} so bucket lists are short
- It is not too hard to design an OK hash function
- It is hard to design an optimal hash function
- Use builtin Python function \texttt{hash}
Implementation 4: Hand-Made Hashing

• Just `insert1` or `lookup1` in bucket $h(key)$:

```python
buckets = 10
T4 = [[] for b in range(buckets)]

def h(value, table=T4):
    return hash(value) % len(table)

def insert4(key, value, table=T4):
    insert1(key, value, table[h(key, table)])

def lookup4(key, table=T4):
    return lookup1(key, table[h(key, table)])
```

• Why not use `insert2` and `lookup2`?
• Main idea in hashing is to minimize collisions
• ... so searching a bucket sequentially is OK
  (may even be faster with low collision probability!)