Introduction to Algorithms

Lecture 5

Prof. Manolis Kellis
<table>
<thead>
<tr>
<th>Unit</th>
<th>Pset</th>
<th>Week</th>
<th>Date</th>
<th>Lecture (Tuesdays and Thursdays)</th>
<th>Recitation (Wed and Fri)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro</td>
<td>PS1</td>
<td>1</td>
<td>Tue Feb 01</td>
<td>1. Introduction and Document Distance</td>
<td>1. Python and Asymptotic Complexity</td>
</tr>
<tr>
<td>Binary Search Trees</td>
<td>Out: 2/1</td>
<td>2</td>
<td>Tue Feb 08</td>
<td>3. Scheduling and Binary Search Trees</td>
<td>2. Peak Finding correctness &amp; analysis</td>
</tr>
<tr>
<td></td>
<td>HW lab: Sun 2/13</td>
<td>2</td>
<td>Thu Feb 15</td>
<td>5. Hashing I: Chaining, Hash Functions</td>
<td>4. Rotations and AVL tree deletions</td>
</tr>
<tr>
<td></td>
<td>PS2 Out: 2/15</td>
<td>3</td>
<td>Thu Feb 17</td>
<td>6. Hashing II: Table Doubling, Rolling Hash</td>
<td>5. Hash recipes, collisions, Python dicts</td>
</tr>
<tr>
<td></td>
<td>Due: Mon 2/28</td>
<td>3</td>
<td>Thu Feb 22</td>
<td>7. President’s Day - Monday Schedule - No Class</td>
<td>6. Probability review, Pattern matching</td>
</tr>
<tr>
<td></td>
<td>Due: Mon 3/7</td>
<td>5</td>
<td>Thu Mar 03</td>
<td>9. Sorting II: Heaps</td>
<td>9. Heap Operations</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wed Mar 09</td>
<td>Q1. Quiz 1 in class at 7:30pm. Covers L1-R10. Review Session on Tue 3/8 at 7:30pm.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Due: Mon 4/11</td>
<td>8</td>
<td>Thu Apr 05</td>
<td>16. Shortest Paths III: Dijkstra</td>
<td>16. Speeding up Dijkstra’s algorithm</td>
</tr>
<tr>
<td></td>
<td>HW lab: Sun 4/10</td>
<td>8</td>
<td>Thu Apr 07</td>
<td>17. Graph applications, Genome Assembly</td>
<td>17. Euler Tours</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>PS6</td>
<td>10</td>
<td>Tue Apr 12</td>
<td>18. DP I: Memoization, Fibonacci, Crazy Eights</td>
<td>18. Limits of dynamic programming</td>
</tr>
<tr>
<td></td>
<td>Out: Tue 4/12</td>
<td>10</td>
<td>Wed Apr 13</td>
<td>Q2. Quiz 2 in class at 7:30pm. Covers L11-R17. Review Session on Tue 4/13 at 7:30pm.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Due: Fri 4/29</td>
<td>10</td>
<td>Thu Apr 14</td>
<td>19. DP II: Shortest Paths, Genome sequence alignment</td>
<td>19. Edit Distance, LCS, cost functions</td>
</tr>
<tr>
<td>Numbers Pictures</td>
<td>PS7 out Thu 4/28</td>
<td>12</td>
<td>Tue Apr 26</td>
<td>22. Numerics I - Computing on large numbers</td>
<td>22. Models of computation return!</td>
</tr>
<tr>
<td>(NP)</td>
<td>Due: Fri 5/6</td>
<td>12</td>
<td>Thu Apr 28</td>
<td>23. Numerics II - Iterative algorithms, Newton’s method</td>
<td></td>
</tr>
<tr>
<td>Beyond</td>
<td></td>
<td></td>
<td>Thu May 5</td>
<td>25. Complexity classes, and reductions</td>
<td>25. Undecidability of Life</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Thu May 10</td>
<td>26. Research Directions (15 mins each) + related classes</td>
<td></td>
</tr>
</tbody>
</table>
Our plan ahead

• **Today: Genomes, Dictionaries, and Hashing**
  – Intro, basic operations, collisions and chaining
  – Simple uniform hashing assumption
  – Hash functions, python implementation

• **Thursday: Speeding up hash tables**
  – Faster comparison: Signatures
  – Faster hashing: Rolling Hash

• **Next week: Space issues**
  – Dynamic resizing and amortized analysis
  – Open addressing, deletions, and probing
Our plan for today: Hashing I

• Today: Genomes, Dictionaries, and Hashing
  ➢ Matching genome segments
  ❑ Introduction to dictionaries
  ❑ Hash function: definition
  ❑ Resolving collisions with chaining
  ❑ Simple uniform hashing assumption
  ❑ Hash functions in practice: mod / mult
  ❑ Python implementation

• Thursday: Speeding up hash tables

• Next week: Space issues
Comparing two genomes bit by bit

Mouse chromosomes 1-19, X

Human

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 X
DNA matching: All about strings

• How to find ‘corresponding’ pieces of DNA
• Given two DNA sequences
  – Strings over 4-letter alphabet
• Find longest substring that appears in both
  – Algorithm vs. Arithmetic
  – Algorithm vs. Arithmetic
  – L19: Subsequence - much harder (e.g. Algorithm)
• Other applications:
  – Plagiarism detection
  – Word autocorrect
  – Jeopardy!
Naïve Algorithm

• Say strings $S$ and $T$ of length $n$
• For $L = n$ downto 1
  ➢ for all length $L$ substrings $X_1$ of $S$
    ➢ for all length $L$ substrings $X_2$ of $T$
      ➢ if $X_1 = X_2$, return $L$
• Runtime analysis
  – $n$ candidate lengths
  – $n$ strings of that length in $X_1$
  – $n$ strings of that length in $X_2$
  – $L$ time to compare the strings
  – Total runtime: $\Omega(n^4)$
Improvement 1: Binary Search on L

• Start with $L = n/2$
• for all length $L$ substrings $X_1$ of $S$
• for all length $L$ substrings $X_2$ of $T$
  • if $X_1 = X_2$, success, try larger $L$
  • if failed, try smaller $L$

• Runtime analysis
  \[ \Omega(n^4) \rightarrow \Omega(n^3 \log n) \]
Improvement 2: Python Dictionaries

- For every possible length $L = n, \ldots, 1$
  - Insert all length $L$ substrings of $S$ into a dictionary
  - For each length $L$ substring of $T$, check if it exists in dictionary

- Possible lengths for outer loop: $n$
- For each length:
  - at most $n$ substrings of $S$ inserted into dictionary, each insertion takes time $O(1) \times L$ ($L$ is paid because we have to read string to insert it)
  - at most $n$ substrings of $T$ checked for existence inside dictionary, each check takes time $O(1) \times L$
  - Overall time spent to deal with a particular length $L$ is $O(Ln)$
- Hence overall $O(n^3)$
- With binary search on length, total is $O(n^2 \log n)$
- “Rolling hash” dictionaries improve to $O(n \log n)$ (next time)
Our plan for today: Hashing I

• Today: Genomes, Dictionaries, and Hashing
  ✓ Matching genome segments
  ➢ Introduction to dictionaries
  ❏ Hash function: definition
  ❏ Resolving collisions with chaining
  ❏ Simple uniform hashing assumption
  ❏ Hash functions in practice: mod / mult
  ❏ Python implementation

• Thursday: Speeding up hash tables
• Next week: Space issues
Dictionaries: Formal Definition

• It is a set containing items; each item has a key
• what keys and items are is quite flexible
• Supported Operations:
  – Insert(key, item): add item to set, indexed by key
  – Delete(key): delete item indexed by key
  – Search(key): return the item corresponding to the given key, if such an item exists
  – Random_key(): return a random key in dictionary

• Assumption: every item has its own key (or that inserting new item clobbers old
• Application (and origin of name): Dictionaries
  – Key is word in English, item is word in French
Dictionaries are everywhere

- Spelling correction
  - Key is misspelled word, item is correct spelling

- Python Interpreter
  - Executing program, see a variable name (key)
  - Need to look up its current assignment (item)

- Web server
  - Thousands of network connections open
  - When a packet arrives, must give to right process
  - Key is source IP address of packet, item is handler
Implementation

• use BSTs!
  • can keep keys in a BST, keeping a pointer from each key to its value
  • $O(\log n)$ time per operation

• Often not fast enough for these applications!

• Can we beat BSTs?
  
  _if only we could do all operations in $O(1)$..._
Dictionaries: Attempt #1

- Forget about BSTs..
- Use table, indexed by keys!

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key1</td>
<td>item1</td>
<td></td>
</tr>
<tr>
<td>key2</td>
<td>item2</td>
<td></td>
</tr>
<tr>
<td>key3</td>
<td>item3</td>
<td></td>
</tr>
</tbody>
</table>
Problems...

- What if keys aren’t numbers?

  How can I then index a table?

  “Everything is a number”
  -- Pythagoras
Interpreting words as numbers

• What if keys aren’t numbers?
  – Anything in the computer is a sequence of bits
  – So we can pretend it’s a number

• Example: English words
  – 26 letters in alphabet
    ⇒ can represent each with 5 bits
  – Antidisestablishmentarianism has 28 letters
  – 28*5 = 140 bits
  – So, store in array of size $2^{140}$ ….oops

• Isn’t this too much space for 100,000 words?
Our plan for today: Hashing I

- Today: Genomes, Dictionaries, and Hashing
  - Matching genome segments
  - Introduction to dictionaries
  - Hash function: definition
  - Resolving collisions with chaining
  - Simple uniform hashing assumption
  - Hash functions in practice: mod / mult
  - Python implementation

- Thursday: Speeding up hash tables

- Next week: Space issues
Hash Functions

• Exploit sparsity
  – Huge universe $U$ of possible keys
  – But only $n$ keys actually present
  – Want to store in table (array) of size $m\sim n$

• Define hash function $h:U \rightarrow \{1..m\}$
  – Filter key $k$ through $h(\ )$ to find table position
  – Table entries are called buckets

• Time to insert/find key is
  – Time to compute $h$ (generally length of key)
  – Plus one time step to look in array
The ‘magic’ of hash functions

PHENOMENAL
COSMIC
POWERS!!

itty bitty living space

With apologies to Disney
Hashing exploits sparsity of space

\[ \mathcal{U} \text{ : universe of all possible keys; huge set} \]
\[ K \text{ : actual keys; small set but not known in advance} \]
All keys map to small space...

\( \mathcal{U} \): universe of all possible keys

(i) insert item1, with key \( k_1 \)

(ii) insert item2, with key \( k_2 \)

(iii) insert item3, with key \( k_3 \)

(iv) suppose we now try to insert item4, with key \( k_4 \) and \( h(k_4)=h(k_2) \)…
… leading to collisions

\[ \text{\textit{U}} : \text{universe of all possible keys} \]

(i) insert \textbf{item1}, with key \textbf{k1}

(ii) insert \textbf{item2}, with key \textbf{k2}

(iii) insert \textbf{item3}, with key \textbf{k3}

(iv) suppose we now try to insert \textbf{item4}, with key \textbf{k4} and \( h(k4) = h(k2) \)…
Our plan for today: Hashing I

- Today: Genomes, Dictionaries, and Hashing
  - Matching genome segments
  - Introduction to dictionaries
  - Hash function: definition
  - Resolving collisions with chaining
- Simple uniform hashing assumption
- Hash functions in practice: mod / mult
- Python implementation

- Thursday: Speeding up hash tables
- Next week: Space issues
Collisions

• What went/can go wrong?
  – Distinct keys x and y
  – But h(x) = h(y)
  – Called a collision

• This is unavoidable: if table smaller than range, some keys must collide…
  – Pigeonhole principle

• What do you put in the bucket?
Coping with collisions

- **Idea1**: Change to a new “uncolliding” hash function and re-hash all elements in the table
  - Hard to find, and can take a long time if m=O(n)
- **Idea2**: Chaining
  - Linked list of hashed items for each bucket (today)
- **Idea3**: Open addressing
  - Find a different, empty bucket for y (next lecture)
- **Idea4**: Perfect hashing (not covered in 6.006)
  - Create a 2nd-level hash table of size k^2 for each k-element bin, and try several 2nd-level hash functions until no collisions are found (see 6.046)
Chaining

- Each bucket, linked list of contained items
- Space used is space of table plus one unit per item (size of key and item)

\( U \): universe of all possible keys
\( K \): actual keys, not known in advance
Problem Solved?

• To find key, must scan whole list in key’s bucket
• Length L list costs L key comparisons
• If all keys hash to same bucket, lookup cost $\Theta(n)$

Solution: optimism
Our plan for today: Hashing I

• Today: Genomes, Dictionaries, and Hashing
  ✓ Matching genome segments
  ✓ Introduction to dictionaries
  ✓ Hash function: definition
  ✓ Resolving collisions with chaining
    ➢ Simple uniform hashing assumption
  ❏ Hash functions in practice: mod / mult
  ❏ Python implementation

• Thursday: Speeding up hash tables
• Next week: Space issues
Simple uniform hashing assumption

- Definition:
  - Each key $k \in K$ of keys is equally likely to be hashed to any slot of table $T$, independent of where other keys are hashed.

Let $n$ be the number of keys in the table, and let $m$ be the number of slots.

Define the load factor of $T$ to be

\[ \alpha = \frac{n}{m} \]

= average number of keys per slot.
Chaining Analysis under SUHA

Average case analysis:

• n items in table of m buckets
• Average number of items/bucket is $\alpha = n/m$
• So expected time to find some key $x$ is $(1+\alpha)$
• $O(1)$ if $\alpha = O(1)$, i.e. $m = \Omega(n)$

apply hash function and access slot
search the list
Summary (rehash)

• Matching big genomes is a hard problem
  – And you will tackle it in your problem set!
• Dictionaries are pervasive
• Hash tables implement them efficiently
  – Under an optimistic assumption of random keys
  – Can be “made true” by heuristic hash functions
• Key idea for beating BSTs: Indexing
  – Sacrificed operations: previous, successor
• Chaining strategy for collision resolution
• Next two lectures: speed & space improvements
Unit #2: Genomes, Hashing, Dictionaries

- **Today: Genomes, Dictionaries, and Hashing**
  - Intro, basic operations, collisions and chaining
  - Simple uniform hashing assumption
  - Hash functions, Python implementation

- **Thursday: Speeding up hash tables**
  - Faster comparison: Signatures
  - Faster hashing: Rolling Hash

- **Next week: Space issues**
  - Dynamic resizing and amortized analysis
  - Open addressing, deletions, and probing