Data Structures: Hashing

Curs 2016
Given a universe $U$, a dynamic set of records, where each record:

- **Array**
- **Linked List** (and variations)
- **Stack** (LIFO): Supports push and pop
- **Queue** (FIFO): Supports enqueue and dequeue
- **Deque**: Supports push, pop, enqueue and dequeue
- **Heaps**: Supports insertions, deletions, find Max and MIN
- **Hashing**
Dynamic Sets.

Given a universe $\mathcal{U}$ and a set of keys $S \subset \mathcal{U}$, for any $k \in S$ we can consider the following operations

- **Search** ($S, k$): decide if $k \in S$
- **Insert** ($S, k$): $S := S \cup \{k\}$
- **Delete** ($S, k$): $S := S \setminus \{k\}$
- **Minimum** ($S$): Returns element of $S$ with smallest $k$
- **Maximum** ($S$): Returns element of $S$ with largest $k$
- **Successor** ($S, k$): Returns element of $S$ with next larger key to $k$
- **Predecessor** ($S, k$): Returns element of $S$ with next smaller key to $k$. 
Recall Dynamic Data Structures

**DICTIONARY**
Data structure for maintaining $S \subseteq U$ together with operations:

- Search $(S, k)$: decide if $k \in S$
- Insert $(S, k)$: $S := S \cup \{k\}$
- Delete $(S, k)$: $S := S \setminus \{k\}$

**PRIORITY QUEUE**
Data structure for maintaining $S \subseteq U$ together with operations:

- Insert $(S, k)$: $S := S \cup \{k\}$
- Maximum $(S)$: Returns element of $S$ with largest $k$
- Extract-Maximum $(S)$: Returns and erase from $S$ the element of $S$ with largest $k$
Priority Queue

Linked Lists:
- \textit{INSERT}: \(O(n)\)
- \textit{EXTRACT-MAX}: \(O(1)\)

Heaps:
- \textit{INSERT}: \(O(\lg n)\)
- \textit{EXTRACT-MAX}: \(O(\lg n)\)

Using a Heap is a good compromise between fast insertion and slow extraction.
Dear Mr. von Neumann:

With the greatest sorrow I have learned of your illness. The news came to me as quite unexpected. Morgenstern already last summer told me of a bout of weakness you once had, but at that time I thought that this was not of any greater significance. As I hear, in the last months you have undergone a radical treatment and I am happy that this treatment was successful as desired, and that you are now doing better. I hope and wish for you that your condition will soon improve even more and that the newest medical discoveries, if possible, will lead to a complete recovery.

Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula $F$ in first order predicate logic and every natural number $n$, allows one to decide if there is a proof of $F$ of length $n$ (length = number of symbols). Let $q(F, n)$ be the number of steps the machine requires for this and let $q(n) = \max_F q(F, n)$. The question is how fast $q(n)$ grows for an optimal machine. One can show that $q(n) \leq k \cdot n$. If there really were a machine with $q(n) - k \cdot n$ (or even $k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number $n$ so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $q(n)$ grows that slowly. Since it seems that $q(n) \leq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $q(n) - k \cdot n$ only means that the number of steps as opposed to trial and error can be reduced from $N$ to $\log N$ or $(\log \log N)^3$. However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

I do not know if you have heard that “Post’s problem”, whether there are degrees of unsolvability among problems of the form $(3 \gamma) \eta(x, \gamma)$, where $\eta$ is recursive, has been solved in the positive sense by a very young man by the name of Richard Friedberg. The solution is very elegant. Unfortunately, Friedberg does not intend to study mathematics, but rather medicine (apparently under the influence of his father). By the way, what do you think of the attempts to build the foundations of analysis on ramified type theory, which have recently gained momentum? You are probably aware that Paul Lorenzen has pushed ahead with this approach to the theory of Lebesgue measure. However, I believe that in important parts of analysis non-eliminable impredicative proof methods do appear.

I would be very happy to hear something from you personally. Please let me know if there is something that I can do for you. With my best greetings and wishes, as well to your wife,

Sincerely yours,
Document similarity

Finding similar documents in the WWW

- Proliferation of almost identical documents
- Approximately 30% of the pages on the web are (near) duplicates.
- Another way to find plagiarism
Hashing functions

Data Structure that supports *dictionary* operations on an universe of *numerical* keys.

Notice the number of possible keys represented as 64-bit integers is $2^{64} = 18446744073709551616$.

Tradeoff *time*/space

Define a **hashing table** $T[0, \ldots, m - 1]

a **hashing function** $h : U \rightarrow T[0, \ldots, m - 1]$

Hans P. Luhn  
(1896-1964)
Simple uniform hashing function.

A good hashing function must have the property that \( \forall k \in U \), \( h(k) \) must have the same probability of ending in any \( T[i] \).

Given a hashing table \( T \) with \( m \) slots, we want to store \( n = |S| \) keys, as maximum.

Important measure: load factor \( \alpha = n/m \), the average number of keys per slot.

The performance of hashing depends on how well \( h \) distributes the keys on the \( m \) slots: \( h \) is simple uniform if it hash any key with equal probability into any slot, independently of where other keys go.
How to choose $h$?

Advice: For an exhaustive treaty on Hashing: D. Knuth, Vol. 3 of *The Art of computing programming*

$h$ depends on the type of key:

- If $k \in \mathbb{R}, 0 \leq k \leq 1$ we can use $h(k) = \lfloor mk \rfloor$.

- If $k \in \mathbb{R}, s \leq k \leq t$ scale by $1/(t - s)$, and use the previous method: $h(k/(t - s)) = \lfloor mk/(t - s) \rfloor$. 
The division method

Choose $m$ prime and as far as possible from a power,

$$h(k) = k \mod m.$$  

Fast ($\Theta(1)$) to compute in most languages ($k \% m$)!

Be aware: if $m = 2^r$ the hash does not depend on all the bits of $K$

If $r = 6$ with $k = \overline{1011000111011010}$

$$h(k) = (45530 \mod 64 = 858 \mod 64)$$
In some applications, the keys may be very large, for instance with alphanumeric keys, which must be converted to ascii:

Example: *averylongkey* is converted via ascii:

\[
\begin{align*}
97 \cdot 128^{11} + 118 \cdot 128^{10} +
101 \cdot 128^9 + 114 \cdot 128^8 \\
+ 121 \cdot 128^7 + 108 \cdot 126^6 \\
+ 111 \cdot 128^5 + 110 \cdot 128^4 \\
+ 103 \cdot 128^3 + 107 \cdot 128^2 \\
+ 101 \cdot 128^1 + 121 \cdot 128^0 = n
\end{align*}
\]

which has 84-bits!
Recall mod arithmetic: for \( a, b, m \in \mathbb{Z} \),

\[
(a + b) \mod m = (a \mod m + b \mod m) \mod m
\]

\[
(a \cdot b) \mod m = ((a \mod m) \cdot (b \mod m)) \mod m
\]

If \( a, b \in \mathbb{Z}_m \) and \( b > a \),

\[
(a - b) \mod m = (m - (a - b)) \mod m
\]

\[
(a \mod m) \mod m = a \mod m
\]

Horner’s rule: Given a specific value \( x_0 \) and a polynomial \( A(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 + \cdots + a_n \) to evaluate \( A(x_0) \) in \( \Theta(n) \) steps:

\[
A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)))
\]
How to deal with large $n$

For large $n$, to compute $h = n \mod m$, we can use mod arithmetic + Horner’s method:

$$( (((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121)$$
$$\cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128 + 107)$$
$$\cdot 128 + 101) \cdot 128 + 121 \mod m$$

$= (((((((97 \cdot 128 + 118 \mod m) \cdot 128) \mod m + 101) \cdot \ldots)))))$$
Collision resolution: Separate chaining

For each table address, construct a linked list of the items whose keys hash to that address.

- Every key goes to the same slot
- Time to explore the list = length of the list
Cost of average analysis of chaining

The cost of the dictionary operations using hashing:

- Insertion of a new key: $\Theta(1)$.
- Search of a key: $O(\text{length of the list})$.
- Deletion of a key: $O(\text{length of the list})$.

Under the hypothesis that $h$ is *simply uniform hashing*, each key $x$ is equally likely to be hashed to any slot of $T$, independently of where other keys are hashed.

Therefore, the expected number of keys falling into $T[i]$ is $\alpha = n/m$. 
Cost of search

For an **unsuccessful** search ($x$ is not in $T$) therefore we have to explore the all list at $h(x) \rightarrow T[i]$ with an the expected time to search the list at $T[i]$ is $O(1 + \alpha)$. ($\alpha$ of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)

For an **successful** search search, we can obtain the same bound, (most of the cases we would have to search a fraction of the list until finding the $x$ element.)

Therefore we have the following result: **Under the assumption of simple uniform hashing, in a hash table with chaining, an unsuccessful and successful search takes time $\Theta(1 + \frac{n}{m})$ on the average.**

Notice that if $n = \theta(m)$ then $\alpha = O(1)$ and search time is $\Theta(1)$. 
Universal hashing: Motivation

For every deterministic hash function, there is a set of bad instances.

An adversary can arrange the keys so your function hashes most of them to the same slot.

Create a set $\mathcal{H}$ of hash functions on $\mathcal{U}$ and choose a hashing function at random and independently of the keys.

Must be careful once we choose one particular hashing function for a given key, we always use the same function to deal with the key.
Universal hashing

Let \( \mathcal{U} \) be the universe of keys and let \( \mathcal{H} \) be a collection of hashing functions with hashing table \( T[0, \ldots, m - 1] \), \( \mathcal{H} \) is universal if \( \forall x, y \in \mathcal{U}, x \neq y \), then

\[
|\{ h \in \mathcal{H} | h(x) = h(y) \}| \leq \frac{|\mathcal{H}|}{m}.
\]

In an equivalent way, \( \mathcal{H} \) is universal if \( \forall x, y \in \mathcal{U}, x \neq y \), and for any \( h \) chosen uniformly from \( \mathcal{H} \), we have

\[
\Pr[h(x) = h(y)] \leq \frac{1}{m}.
\]
Universality gives good average-case behaviour

**Theorem**

If we pick a u.a.r. $h$ from a universal $H$ and build a table using and hash $n$ keys to $T$ with size $m$, for any given key $x$ let $Z_x$ be a random variable counting the number of collisions with others keys $y$ in $T$.

$$E[\#\text{collisions}] \leq n/m.$$ 

**Proof** We want to compute the expected list at $T[i]$.

For each key $x$, define indicator rv to count how many others keys hash to the same slot.

$$Z_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Then $E[Z_{xy}] \leq 1/m$ and $Z_x = \sum_{y \in T - \{x\}} Z_{xy}$
Proof

\[ E[Z_{xy}] = E \left[ \sum_{y \in T - \{x\}} Z_{xy} \right] \]
\[ = \sum_{y \in T - \{x\}} E[Z_{xy}] \]
\[ = \sum_{y \in T - \{x\}} 1/m = \frac{n - 1}{m} \quad \square \]

Therefore, universal hash functions dismount adversarial strategy
Construction of a universal family: $\mathcal{H}$

To construct a family $\mathcal{H}$ for $N = \max\{\mathcal{U}\}$ and $T[0, \ldots, m − 1]$:

- $\mathcal{H} = \emptyset$.
- Choose a prime $p$, $N \leq p \leq 2N$. Then $\mathcal{U} \subset \mathbb{Z}_p = \{0, 1, \ldots, p − 1\}$.
- Choose independently and u.a.r. $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$. Given a key $x$ define $h_{a,b}(x) = ( (ax + b) \mod p ) \mod m$.
- $\mathcal{H} = \{ h_{a,b} | a, b \in \mathbb{Z}_p, a \neq 0 \}$.

Example: $p = 17, m = 6$ we have $\mathcal{H}_{17,6} = \{ h_{a,b} : a \in \mathbb{Z}_p^+, b \in \mathbb{Z}_p \}$ if $x = 8, a = 3, b = 4$ then $h_{3,4}(8) = ((3 \cdot 8 + 4) \mod 17) \mod 6 = 5$
Properties of $\mathcal{H}$

1. $h_{ab} : \mathbb{Z}_p \rightarrow \mathbb{Z}_m$.
2. $|\mathcal{H}| = p(p - 1)$. (We can select $a$ in $p - 1$ ways and $b$ in $p$ ways)
3. Specifying an $h \in \mathcal{H}$ requires $O(\lg p) = O(\lg N)$ bits.
4. To choose $h \in \mathcal{H}$ select $a, b$ independently and u.a.r. from $\mathbb{Z}_p^+$ and $\mathbb{Z}_p$.
5. Evaluating $h(x)$ is fast.
Theorem

The family $\mathcal{H}$ is universal.

For the proof:
Chapter 11 of Cormen, Leiserson, Rivest, Stein: *An introduction to Algorithms*
Cuckoo Hashing

Pagh, Rodler: *Cuckoo Hashing*. ESA-2001

We have two hash tables $T_1, T_2$ with size $m$ each and two hash functions $h_1$ for $T_1$ and $h_2$ for $T_2$.

Can use for instance $h_1(x) = x \mod m$ and $h_2(x) = \lceil x/m \rceil \mod m$

Searching $\Theta(1)$ in worst case, rather than in expectation.

- **Search an element $x$**: constant worst case complexity ($x$ only can be in the 2 positions $h_1(x)$ or in $h_2(x)$)
- **Delete an element**: constant worst case complexity (look at the 2 positions and erase the element)
- **Inserte an element**: expected constant complexity.
To insert one key $x$:

- Start by inserting it in $T_1$.
- If $h_1$ hits an empty slot, insert $x$ there.
- If there is $y$ in $h_1(x)$ kick $y$ to $h_2(y)$ if possible, and place $x$ in the empty slot in $T_1$.
- Repeat this process, bouncing between $T_1$ and $T_2$, until arrive to stability.
Cuckoo: Insertion

To insert one key $x$:

- Start by inserting it in $T_1$.
- If $h_1$ hits an empty slot, insert $x$ there.
- If there is $y$ in $h_1(x)$ kick $y$ to $h_2(y)$ if possible, and place $x$ in the empty slot in $T_1$.
- Repeat this process, bouncing between $T_1$ and $T_2$, until arrive to stability.
Cuckoo Hashing: Long cycles of insertion

One complication is that the cuckoo may loop for ever. The probability of such an event is small. In such a case choose an upper bound in the number of slot exchanges, and if it exceeds, do a rehash: choose new functions and start.

**Example:** We have \( \{y, x, w, z, u\} \)

\[
\begin{align*}
  h_1(x) &= i; & h_1(y) &= i; & h_1(w) &= j; & h_1(z) &= j; \\
  h_2(x) &= l; & h_2(y) &= k; & h_2(w) &= l; & h_2(z) &= k
\end{align*}
\]

Next we hash \( u \): \( h_1(u) = j \) and \( h_2(u) = k \)
Cuckoo Hashing: An example

We wish to hash the set of keys: (20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6) using the functions $h_1(k) = k \mod 11$ and $h_2(k) = \left\lfloor \frac{k}{11} \right\rfloor \mod 11$.
Cuckoo Hashing: An example

We wish to hash the set of keys:
\((20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6)\)
using the functions \(h_1(k) = k \mod 11\) and \(h_2(k) = \left\lfloor \frac{k}{11} \right\rfloor \mod 11\).

<table>
<thead>
<tr>
<th>(k)</th>
<th>(h_1)</th>
<th>(h_2)</th>
<th>(h_1)</th>
<th>(h_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>105</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo Hashing: An example

We wish to hash the set of keys: 
(20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6)
using the functions $h_1(k) = k \mod 11$ and $h_2(k) = \lfloor \frac{k}{11} \rfloor \mod 11$. 

<table>
<thead>
<tr>
<th></th>
<th>$h_1$</th>
<th>$h_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>105</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

0  | 67 |
1  |    |
2  |    |
3  |    |
4  |    |
5  | 105|
6  |    |
7  | 105|
8  |    |
9  |  53|

0  | 20 |
1  |    |
2  |    |
3  |    |
4  |    |
5  |  50|
6  |    |
7  |  75|
8  |    |
9  | 100|
Cuckoo Hashing: An example

We wish to hash the set of keys: (20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6) using the functions $h_1(k) = k \mod 11$ and $h_2(k) = \left\lfloor \frac{k}{11} \right\rfloor \mod 11$.
Cuckoo Hashing: An example

Hash functions: $h_1(k) = k \mod 11$ and $h_2(k) = \lfloor \frac{k}{11} \rfloor \mod 11$.

<table>
<thead>
<tr>
<th></th>
<th>$h_1$</th>
<th>$h_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>75</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>105</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

With 6 we have to rehash!!!
Complexity

Cuckoo hashing has a complexity:

- **Search an element** $x$: constant worst case complexity ($x$ only can be in the 2 positions $h_1(x)$ or in $h_2(x)$)
- **Delete an element**: constant worst case complexity (look at the 2 positions and erase the element)
- **Insert an element**: expected constant complexity.

*It is a simple alternative to perfect matching, to implement a dictionary with reasonable space and constant searching time.*

Other models, for example $d$-hashing tables.
Bloom filter

Given a set of elements $S$, we want a Data structure for supporting insertions and querying about membership in $S$.

In particular we wish a DS s.t.

- *minimizes* the use of memory,
- *can check membership* as fast as possible.


A hash data structure where each register in the table is one bit
Query on a list of e-mails

We have a set $S$ of $10^9$ e-mail addresses, where the typical e-mail address is 20 bites. Therefore it does not seem reasonable to store $S$ in main memory. We can spare 1 Gigabyte of memory, which is approximately $10^9$ bytes or $8 \times 10^9$ bites. How can we put $S$ in main memory to query it?
Definition Bloom filter

Create a one bit hash table $T[0, \ldots, m-1]$, and a hash function $h$. Initially all $m$ bits are set to 0.

Giving a set $S = \{x_1, \ldots, x_n\}$ define a hashing function $h : S \rightarrow T$. For every $x_i \in S$, $h(x_i) \rightarrow T[j]$ and $T[j] := 1$. Given a set $S$ a function $h()$ and a table $T[m]$: 

Insert $(x)$
$h(x) \rightarrow i$

if $T[i] == 0$ then
$T[i] = 1$
end if

inS(y)
$h(x) \rightarrow i$

if $T[i] == 1$ then
return Yes
else
return No
end if

Notice: once we have hashed $S$ into $T$ we can erase $S$. 
Bloom filter needs $O(m)$ space and answers membership queries in $\Theta(1)$.

Inconvenience: Do not support removal and may have false positive.

In a query $y \in S?$, a Bloom filter always will report correctly if indeed $y \in S$ ($h(y) \rightarrow T[i]$ with $T[i] = 1$), but if $y \notin S$ it may be the case that $h(y) \rightarrow T[i]$ with $T[i] = 1$, which is called a False positive.

How large is the error of having a false positive?
Probability of having a false positives

Let $|S| = n$, we constructed a BF $(h, T[m])$ with all elements in $S$. If we query about $y \in S?$, with $y \not\in S$, and $h(y) \rightarrow T[i]$, what is the probability that $T[i] = 1$?

After all the elements of $S$ are hashed into the Bloom filter, the probability that a specific $T[i] = 0$ is $(1 - \frac{1}{m})^n \sim e^{-n/m}$.

Therefore, for a $y \not\in S$, the probability $h(y)$ goes to a bit set already to 1 is $1 - (1 - \frac{1}{m})^n \sim 1 - e^{-n/m}$.

To minimize the probability of false positive, we must make $e^{-n/m}$ small, i.e, $m$ must be very large.
Alternative: Amplify

Take $k$ different functions $\{h_1, h_2, \ldots, h_k\}$ in the same 2-universal set of functions.

Ex. Bloom filter with 3 hash functions: $h_1$, $h_2$, $h_3$.

When making a query about if $y \in S$, compute $h_1(y), \ldots, h_t(y)$, if one of them is 0 we certainty $y \notin S$, else (if all the $k$ hashing go to bits with value 1) $y \in S$ with some probability.

After hashing the $n$ element $k$ times to $T$:

$$\Pr[T[i] = 1] = (1 - e^{-kn/m})^k.$$ 

The probability that all $h_1(y), \ldots, h_k(y)$ go to bits already to 1 is:

$$p = (1 - e^{-kn/m})^k.$$
Optimal number of hash functions

For given $n$ and $m$ the number of hash functions $k$, which minimizes the false positive is:

$$k = \frac{n}{m} \ln 2 \sim 0.69314718056 \frac{n}{m}$$

To obtain the size $m$ of $T$ substitute the value of $k$ in $p = (1 - e^{-kn/m})^k$ to get

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Therefore, to maintain a fixed false positive probability, the length of the Bloom table must grow linearly with $n$. 
Practical issues

For password checking:
If $D$ has 100000 common words, each of 7 characters \(\Rightarrow\) we need 700000 bytes
Use 5 tables of 160000 bits each \(\Rightarrow\) need a total of 800000 bits = 100000 bytes.
The probability of error is 0.02

On the other hand although the results shown before are asymptotic, there also work for practical values of $n$. Figure in the side table give the probability of false positive wrt to $n$
Applications Bloom filters

*Blum filters are useful when a set of keys is used and space is important.*

- Summarizing content to aid collaborations in overlay and peer-to-peer networks.
- Packet routing: Bloom filters provide a means to speed up or simplify packet routing protocols.
- Useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

A. Broder, M. Mitzenmacher: *Network applications of Bloom filters: A survey.* Internet Mathematics, 1,4: 485-509, 2005
String matching

The string matching problem: given a text $TX[1 \ldots n]$ and a pattern $P[1 \ldots \ell]$, where elements of $TX$ and $P$ are drawn from the same alphabet $\Sigma$, we wish to find all the occurrences of $P$ in $TX$, together with the position they start to occur.

$TX$: a b c a b a a b c a b a b a a c b a a b a b
$P$: a b a a

$TX$: a b c a b a a b c a b a b a a c b a a b a b

Given a string $x$ and $y$:

$|x|$ its length

$xy$ its concatenation with length $|x| + |y|$
Naive algorithm

Search \((TX,P)\)
for \(i = 1\) to \(n - \ell\) do
    if \(PT[1,\ldots,\ell] = TX[i,\ldots,i+\ell-1]\) then
        print \(P\) occurs at \(i\)
    end if
end for

This algorithm has complexity \(\Theta((n - \ell + 1)\ell)\), worst case \(O(n^2)\)
Hashing


Given $TX (|TX| = n)$ and pattern $P (|P| = \ell)$, want to indicate define a hash function $h$ a table $T[0, \ldots, m - 1]$.

Notice each symbol in $TX$ is a key. Wlog consider alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. 
Karp-Rabin’s hashing algorithm

Idea: Break TX into overlapping substring of length $= \ell$, $S_0, S_1, \ldots S_i, \ldots$ and compute the decimal value of each substring $S_i$ and of $P$. 

\begin{align*}
\text{TX} & \quad \text{P} \\
0 & \quad 0 \\
8 & \quad 1 \\
6 & \quad 7 \\
1 & \quad 9 \\
7 & \quad 3 \\
9 & \quad 5 \\
3 & \quad 7 \\
5 & \quad 3 \\
7 & \quad 4 \\
8 & \quad 2 \\
6 & \quad 1 \\
7 & \quad 9 \\
9 & \quad 3 \\
3 & \quad 5 \\
S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_6 \\
\text{T} & \quad \text{h} \\
17935 & \quad 86179 \\
S_0 & \quad 61793 \\
S_1 & \quad 17935 \\
S_2 & \quad 17935
\end{align*}
First Hashing algorithm

Let $s_i$ denote the decimal value of $S_i$ and $p$ the decimal value of $P$. Use Horner’s rule to compute $p$ in time $\Theta(\ell)$:

$$p = P[\ell - 1] + 10(P[\ell - 2] + \cdots + (10P[0])) \cdots$$

In the same way, use Horner’s rule to compute for $0 \leq i < n$:

$$t_i = S_i[i]10^{\ell - 1} + S_i[i + 1]10^{\ell - 1} + \cdots + S_i[i + \ell - 1]10^0$$

- At the beginning all registers to 0.
- Hash $P \rightarrow T$ $h(p) = p \mod m$, if $h(P) = i$ then $T[i] := 1$
- Run through TX, hashing each set of $\ell$ consecutive characters into $T$
- If one of them goes to a $T[i]$ ($T[i] = 1$), double check that the $\ell$ $S_k$ match $P$ (i.e. $s_k - p = 0$)

Complexity: $O(n\ell)$
Rolling Hash-1

Instead of looking to $O(n)$ substrings independently, we may take advantage the substrings have a lot of overlap:

$s_i = 79357 \rightarrow s_{i+1} = 93573 \rightarrow s_{i+2} = 35734$

$$s_{i+1} = S_{i+1}[i+1]10^{\ell-1} + S_{i+1}[i+2]10^{\ell-1} + \cdots + S_{i+1}[i+\ell-1]10^1$$

$$(S_i \setminus \{S_i[i]\}) \ast 10$$

$$+ S_{i+1}[i+\ell]10^0$$

Knowing $s_i$ to get $s_{i+1}$ with we only have to deal with the element leaving ($S_i[i]$) and the element incorporating ($S_{i+\ell}$):

$$s_{i+1} = (s_i - (S_i[i] \ast 10^\ell)) \ast 10 + S_{i+1}[i+\ell]$$
Rolling Hash-2

Recall mod magic: for $a, b, m \in \mathbb{Z}$,
\[(a + b) \mod m = (a \mod m + b \mod m) \mod m\]
\[(a \cdot b) \mod m = ((a \mod m) \cdot (b \mod m)) \mod m\]
If $a, b \in \mathbb{Z}_m$ and $b > a$, \[(a - b) \mod m = (m - (a - b)) \mod m\]
\[(a \mod m) \mod m = a \mod m\]

Using the hash function $h(a) = a \mod m$, for any $a \in \mathbb{N}$
\[h(s_{i+1}) = ((s_i - (S_i[i] \times 10^\ell)) \times 10 + S_{i+1}[i + \ell]) \mod m\]

Therefore given $h(s_i)$ we can compute $h(s_{i+1})$ in $\Theta(1)$ steps.
Example

$\text{TX}=861793$, $m = 73$,
Preprocess: $h(86179) = 39$ and $10^4 \mod 73 = 72$.

$h(61793) = ((86179 - 8 \cdot 10^4) \cdot 10 + 3) \mod 73$

$= (((h(86179) - (8 \cdot 10^4) \mod 73) \cdot 10 + 3) \mod 73 + 3) \mod 73$

$= ((47 \cdot 10) \mod 73 + 3) \mod 73 = 35$
Karp-Rabin Algorithm

Given a text $|TX| = n$ pattern $P = \ell$, hash table $|T| = m$, hash function $h = \ast \mod m$:

Karp-Rabin $(TX, P, T)$
$p = 0; t_0 = 0; q = 10^{\ell-1} \mod m$
for $j = 0$ to $\ell - 1$ do
  $h(p) = (10p + P[j]) \mod m$
  $h(t_0) = (10t_0 + TX[j]) \mod m$
end for
for $i = 0$ to $n - \ell$ do
  if $h(p) == h(t_i)$ then
    if $P[0 \ldots \ell - 1] == TX[i \ldots i + \ell - 1]$ then
      return Match at $i$
    end if
  else
    $h(t_{i+1}) = (10(t_i - T[i + 1]q) + T[i + \ell + 1]) \mod m$
  end if
end for
Complexity

- To use any other radix $d \neq 10$ it behaves the same as radix-10. We has to substitute 10 by $d$.
- Using rolling has we could speed the computation of the hash function of each $\ell$-string to $\Theta(1)$, once we compute the first one in $O(\ell)$
- The total complexity depends of the number of comparisons. Each comparison takes $\Theta(\ell)$.
- If TX and $P$ are such that the algorithm must make $\Theta(n)$ comparisons, the total complexity is $\Theta(n\ell)$
- In most practical applications (genomics, text searching, etc.), string searching using Karp-Rabin takes $O(n + \ell) = O(n)$. 
Complexity

- Regarding collisions from hashing different substrings, we must choose \( m \) a large prime integer, which fits into a computer word and make sure it keeps basic operations constants. For instance, if \( m = O(n) \) then the expected number of collisions is \( \Theta(1) \) collision in each slot, if \( m = O(n^2) \) we expect \( O(1/n) \) number of collisions per cell, which is nice, but at expenses of having a very large \( T \).

- There is a fast algorithm for string matching knuth-Morris-Pratt \( \Theta(n^2) \). But the simplicity of Karp-Rabin and the easiness to generalize to non-textual applications, makes K-R a good choice, widely used in practice.
Theorem 2 shows that regular digraphs, like undirected graphs, reach absorption in expected polynomial time. In next Section we show that the same does not hold for general digraphs. In particular, we construct an infinite family \( (G_N) \) of strongly connected digraphs indexed by a positive integer \( N \).

The underlying structure of the graph \( G_{N,N} \) is a large undirected clique on \( N \) vertices and a long directed path. Each vertex of the clique sends an edge to the first vertex of the path, and each vertex of the clique receives an edge from the path’s last vertex. We refer to the first \( N \) vertices of path as \( P \) and the remainder as \( Q \). Each vertex of \( P \) has out-degree 1 but receives 4 \( |P| \) edges from \( Q \).

Suppose that \( N \) is sufficiently large with respect to \( r \) and consider the Moran process on \( G_{N,N} \). Given the relative sizes of the clique and the path, there is a reasonable probability (about \( \frac{1}{N^2} \)) that the initial mutant is in the clique. The edges to and from the path have a negligible effect so it is reasonably likely (probability at least \( 1 - \frac{1}{N} \)) that we will then reach the state where half the clique vertices are mutants. To reach absorption from this state, one of two things must happen.

For the process to reach extinction, the mutants already in the clique must die out. Because the interaction between the clique and path is small, the number of mutants in the clique is very close to a random walk on \( \{0, \ldots, N\} \) with upward drift \( r \), and the expected time before such a walk reaches zero from \( N/2 \) is exponential in \( N \).

\[
S = \{ v \in V(G), r_{td} < r_{ut} \} \leq \bar{Y}[t]
\]

and note that, for \( v \in V \setminus S, r_{td} = r_{tu} \). For \( v \in V \), let \( t_v \) be a random variable drawn from \( \text{Exp}(r_{tu}) \) and, for \( v \in S \), let \( t_v \) be \( \text{Exp}(r_{ut} - r_{tu}) \). From the definition of the exponential distribution, it is easy to see that, for each \( v \in S, \min(t_v, t_u) \sim \text{Exp}(r_{ut}) \).

If some \( t_v \) is minimal among \( \{ t_v, \forall v \in V \setminus \bar{Y}[t] \} \), then choose an out-neighbour \( w \) of \( v \) u.a.r. and set \( \bar{Y}[t+1] = Y[\bar{Y}[t] \cup \{w\}] \). If \( t_v \) is minimal among \( \{ t_v, \forall v \in \bar{Y}[t] \} \), then choose an out-neighbour \( w \) of \( v \) u.a.r. and set \( \bar{Y}[t+1] = Y[\bar{Y}[t] \cup \{w\}] \).

In both cases, the continuous-time Moran process has been faithfully simulated up to time \( t + \tau \), where \( \tau = t_v \) in the first case and \( \tau = r_{ut} \) in the second case, and the memorylessness of the exponential distribution allows the coupling to continue from \( Y[t+\tau] \) and \( \bar{Y}[t+\tau] \).

Our main technical tool is stochastic domination. Intuitively, one expects that the Moran process has a higher probability of reaching fixation when the set of mutants is \( S \) than when it is some subset of \( S \), and that it is likely to do so in fewer steps. It also seems obvious that modifying the process by continuing to allow all transitions that create new mutants but forbidding some transitions that remove mutants should make fixation faster and more probable. Such intuitions have been used in proofs in the literature; it turns out that they are essentially correct, but for rather subtle reasons.

The Moran process can be described as a Markov chain \( (Y_t)_{t \geq 0} \), where \( Y_0 \) is the set \( S \subseteq V(G) \) of mutants at the 0th step. The normal method to make the above intuitions formal would be to demonstrate a stochastic domination by coupling the Moran process \( (Y_t)_{t \geq 0} \) with another copy \( (Y_t')_{t \geq 0} \) of the process where \( Y_0 \subseteq Y_0' \). The coupling was designed so that \( Y_1 \subseteq Y_1' \) would ensure that \( Y_t \subseteq Y_t' \) for all \( t \geq 1 \). However, a simple example shows that such a coupling does not always exist for the Moran process. Let \( G \) be the undirected path with two edges: \( V(G) = \{1, 2, 3\} \) and \( E(G) = \{(1, 2), (2, 3), (3, 2)\} \). Let \( (Y_t)_{t \geq 0} \) and \( (Y_t')_{t \geq 0} \) be Moran processes on \( G \) with \( Y_0 = \{1, 2\} \) and \( Y_0' = \{2, 3\} \). With probability \( \frac{1}{2} \), we have \( Y_1 = \{1, 2\} \). The only possible value for \( Y_2 \) that contains \( Y_2 = \{1, 2\} \) but occurs with probability only \( \frac{1}{2(2r-1)} \). Therefore, any coupling between the two processes fails because

\[
Pr(Y_2 \not\subseteq Y_2') \geq \frac{r(r-1)}{2(2r+2)(2r+1)},
\]

which is strictly positive for any \( r > 1 \). The problem is that, when vertex 2 becomes a mutant, it becomes more likely to be the next vertex to reproduce and, correspondingly, every other vertex becomes less likely. This can be seen as the new mutant “slow down” all the other vertices in the graph.

To get around this problem, we consider a continuous-time version of the process, \( \bar{Y}[t] \) (\( t \geq 0 \)). Given the set of mutants \( Y[t] \) at time \( t \), each vertex waits an amount of time before reproducing. For each vertex, this period of time is chosen according to the exponential distribution with parameter equal to the vertex’s fitness, independently of the other vertices. (Thus, the parameter is \( r \) if the vertex is a mutant, and 1 otherwise.) If the first vertex to reproduce is \( v \) at time \( t + \tau \), then, as in the standard, discrete-time version of the process, one of its out-neighbours \( w \) is chosen uniformly at random, the individual at \( w \) is replaced by a copy of the one at \( v \). But the time at which \( w \) will next reproduce is exponentially distributed with parameter given by its new fitness. The discrete-time process is recovered by taking the sequence of configurations each time a vertex reproduces.

In continuous time, each member of the population reproduces at a rate given by its fitness, independently of the rest of the population whereas, in discrete time, the population has to co-ordinate the decide who will reproduce next. This is still true in continuous time that vertex \( w \) becoming a mutant makes it less likely that each vertex \( v \neq w \) will be the next to reproduce. However, the vertices are not slowed down as they are in discrete time: they continue to reproduce at rates determined by their fitness. This distinction allows us to establish the following coupling lemma, which formalises the intuitions discussed above.

Theorem 3 shows that regular digraphs, like undirected graphs, reach absorption in expected polynomial time. In next Section, we show that the same does not hold for general digraphs. In particular, we construct an infinite family \( (G_{N,N}) \) of strongly connected digraphs indexed by a positive integer \( N \).

When \( k \) is a positive integer, \( [k] \) denotes \( \{1, \ldots, k\} \). We consider the evolution of the Moran process on a strongly connected directed graph (digraph). Consider such a digraph
Common substring problem

Given two texts \( T_{x_1} \) and \( T_{x_2} \), with \(|T_{x_1}| = |T_{x_2}| = n\) discover if they share a common substring of length \( \ell \). Define \( h \) and \( T[0 \cdots m - 1] \) and use rolling hash (notice blancs should be considered as an extra symbol):

1. Hash the first substring of length \( \ell \) in \( T_{x_1} \) to \( T \). \((O(\ell))\)
2. Use rolling hash to compute the subsequent \( n - 1 \) substring in \( T_{x_1} \), hashing each one to \( T \). \((O(n))\)
3. Hash the first substring of length \( \ell \) in \( T_{x_2} \) to \( T \). \((O(\ell))\)
4. Use rolling hash to compute the subsequent \( n - 1 \) substring in \( T_{x_2} \), hashing each one to \( T \). For each substring, check if there are collisions with substrings from \( T_{x_1} \). \((O(n))\)
5. If a substring of \( T_1 \) collide with a substring of \( T_2 \) do a string comparison on those substrings. \((O(\ell))\)

If the number of collisions should be small the complexity is \( O(n) \). But for large number of collisions it could be \( O(n^2) \).
Applications: Cryptographic hash functions

Cryptographic hash functions: One way hash functions

Variable length original data  Fixed length “digest” of data

Hola

Cryptographic hash function

Setze jutges d’un jutjat mengen el fetge d’un penjat si el penjat es despenges es menjaria els setze fetges dels setze jutges que l’han jutjat

DFCD3454BBEA788A 751A6 96C 2 4D9700 9 CA992D17

46042841 935C7F80 9158585AB94AE241 26EB3CEA
Cryptographic hash

Cryptographic hash functions have to string of any length and output a fixed-length hash value, in general in hexadecimal.

Hexadecimal = Radix 16

\[(4CF5)_{16} = (4 \times 16^3 + 12 \times 16^2 + 15 \times 16^1 + 5 \times 16^0) = 19701\]

For security reasons, modern crypto-hash implementations give yield very large integers, for instance MD5 gives a 128 bits integer, SHA512 yields a 512 bits integer.

To make those output integers more compact it is customary to represent them in Radix 16.
Properties of cryptographic hash functions

Due to the applications, malicious adversaries try to hack crypto-hash functions. Crypto-hash functions should behave as random while being deterministic and efficiently computable.

Main properties of a good crypto-hash function $h_c$

- For any message $M$, $h_c(M)$ is easy to compute
- Pre-image resistance it is not feasible to recover $M$ from $k = h_c(M)$
- Collision resistance it is not feasible to find $M_1 = M_2$ s.t. $h_c(M_1) = h_c(M_2)$
- It is not feasible to modify $M$ without changing $h_c$
Applications of crypto-hash functions: Password verification

Reduce security breach for passwords storing.
Store the hash digest on a table with users names.

There are Cryptography Hash functions to get a cryptography\textit{h}\textit{e} integer from the fingerprint.

- **"hello"**
- **"world"**
- **password store**
- **Access Denied**
- **Access Granted**