

Applications of Modular Arithmetic

Lecture 5 • Discrete Mathematics and Algorithms

The RSA algorithm

Generate 2 large primes, p and q

p	19014104183583901283
q	50579838717232030631

(these are actually very small, should be 1024-2048 bits long)

Find their product, $n = pq$, as well as $\varphi(n) = (p - 1)(q - 1)$

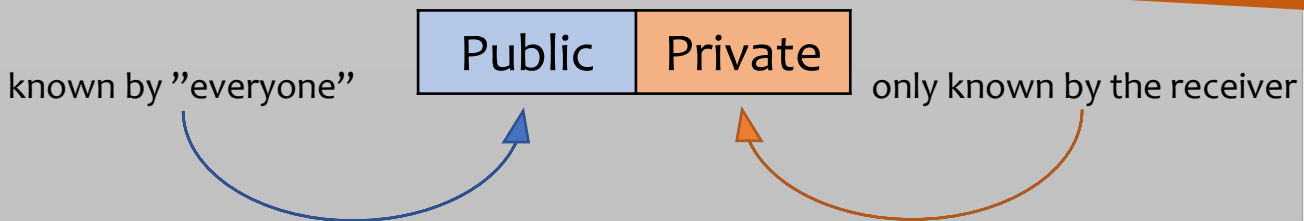
n	961730322958320540523406086367136199573
$\varphi(n)$	961730322958320540453812143466320287660

Choose an integer e , $1 < e < \varphi(n)$, such that $\gcd(e, \varphi(n)) = 1$

e	65537
-----	-------

Calculate d , the multiplicative inverse of $e \bmod \varphi(n)$

d	248367879458467356564700406307390794973
-----	---



Encryption

Write your message as an integer, m , with $1 < m < n$

m	12001907081800220418141204
-----	----------------------------

Calculate the encrypted message $c = m^e \bmod n$

c	483936546861920939820266303617617211066
-----	---

Send c to the receiver

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Decryption

Receive c and calculate $m = c^d \bmod n$

m	12001907081800220418141204
-----	----------------------------

Extract the plaintext message from the integer m

MATHISAWESOME

The RSA algorithm

Generate 2 large primes, p and q

p	19014104183583901283
q	50579838717232030631

(these are actually very small, should be 1024-2048 bits long)

Find their product, $n = pq$, as well as $\varphi(n) = (p - 1)(q - 1)$

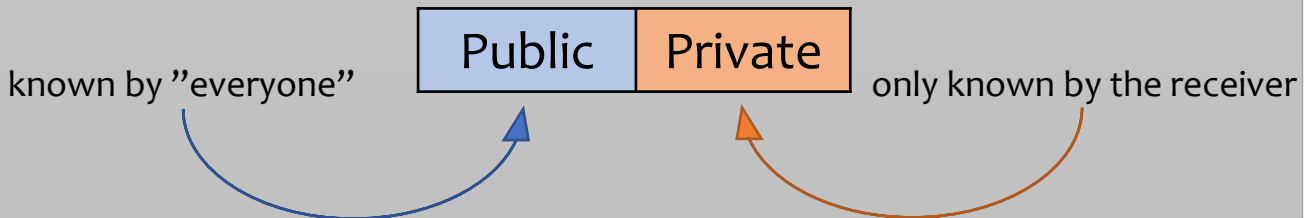
n	961730322958320540523406086367136199573
$\varphi(n)$	961730322958320540453812143466320267660

Choose an integer e , $1 < e < \varphi(n)$, such that $\gcd(e, \varphi(n)) = 1$

e	65537
-----	-------

Calculate d , the multiplicative inverse of $e \bmod \varphi(n)$

d	248367879458467356564700406307390794973
-----	---



The algorithm works because d is kept private.

However, you could prime factorize n to get p and q .
With p and q , you can calculate $\varphi(n)$, and then d .
The algorithm is now broken.

**But prime factorization is an insurmountable problem
for large integers ...**
... hopefully.

P versus NP problem

From Wikipedia, the free encyclopedia

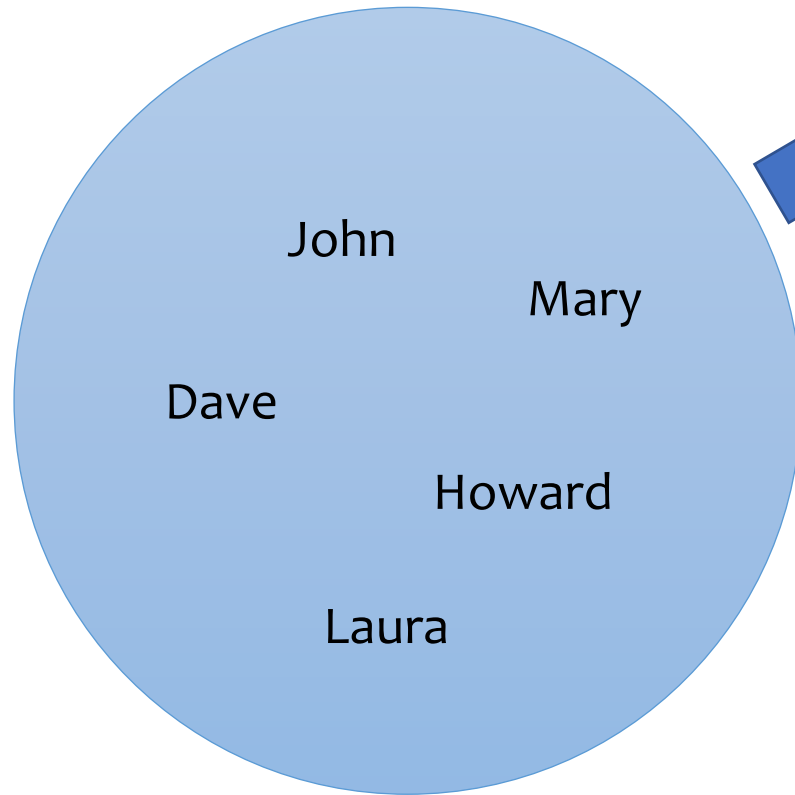
:

The problem has been called the most important open problem in [computer science](#).^[1]
Aside from being an important problem in [computational theory](#), a proof either way would have profound implications for mathematics, [cryptography](#), algorithm research, [artificial intelligence](#), [game theory](#), multimedia processing, [philosophy](#), [economics](#) and many other fields.^[2]

It is one of the seven [Millennium Prize Problems](#) selected by the Clay Mathematics Institute, each of which carries a US\$1,000,000 prize for the first correct solution.

Hashing

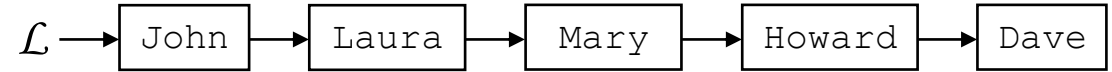
Suppose we have some elements that we wish to keep under control.



Keep under control means that we are able to insert, delete, and search for elements

IDEA #1

A list!



Searching for Howard goes through John, Laura and Mary: slow!

IDEA #2

A hash table!

The elements John, Mary, Dave, Howard and Laura are **keys**. A hash function h turns a **key** into an **index**. Here's a hash function:

$h(\text{key}) = \text{the place in the alphabet of the first letter in the key}$

E.g., $h(\text{"John"}) = 9$

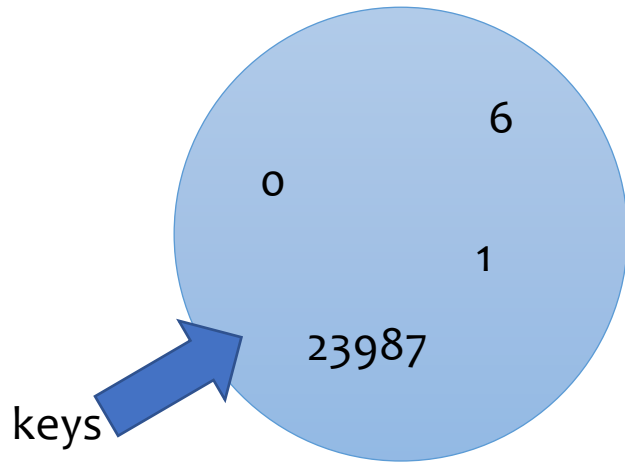
																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					</
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

Searching for Howard goes straight to Howard: fast!

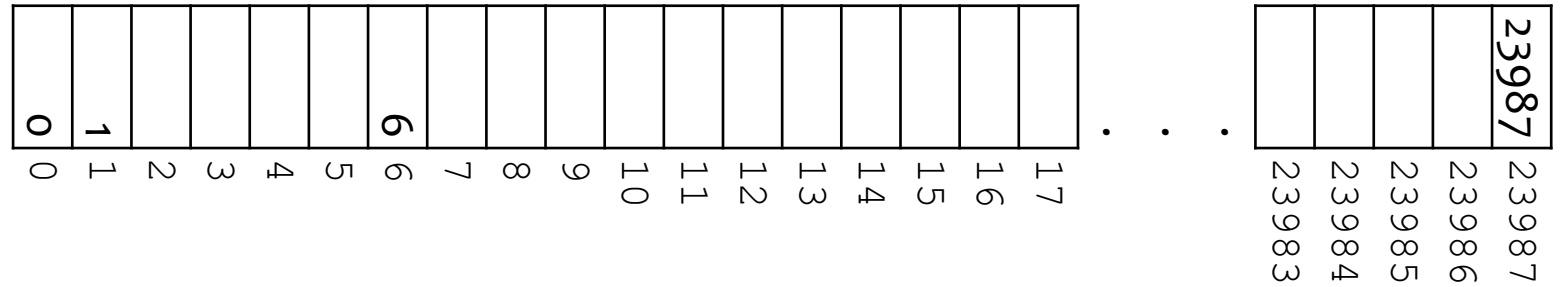
Why do you think the hash function above is a bad choice?

Good hash functions

Let's return to strings in a bit, and start out with a simpler example:



Bad hash function: $h(k) = k$



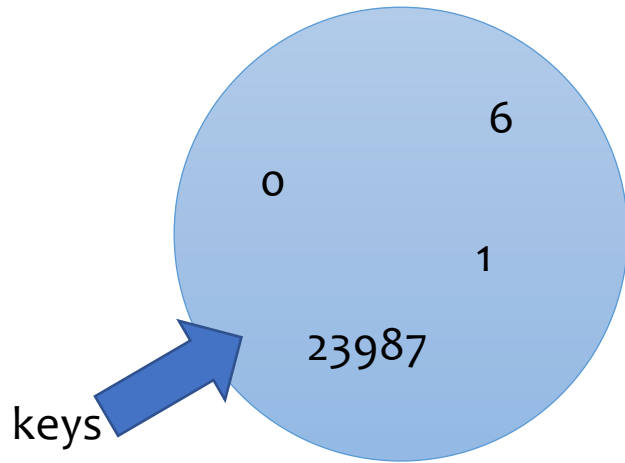
A waste of computer memory!

Instead, let's use an array of length 4.

For this, we need to find some function that turns any key into a number between 0 and 3 ...

Any ideas?

Modulus: Ideal for hashing



Good hash function: $h(k) = k \bmod 4$

$h(0) = 0, h(1) = 1, h(6) = 2, h(23987) = 3$

0	1	6	23987
0	1	2	3

Using the hash-table, we can very quickly find the position of any key!

Another example

maps all integer keys to the range $[0,9]$
 \Rightarrow appropriate for array of length 10



$$h(x) = (x^2 - 6x + 9) \bmod 10$$

$$h(4) = (16 - 24 + 9) \bmod 10 = 1$$

$$h(-7) = (49 + 42 + 9) \bmod 10 = 0$$

$$h(0) = (0 - 0 + 9) \bmod 10 = 9$$

$$h(8) = (64 - 48 + 9) \bmod 10 = 5$$

0	1	2	3	4	5	6	7	8	9

What if the keys are strings?

We need to turn the string into a number!

Maybe just add the ASCII-values?

“DANY” = $68+65+78+89 = 300$

“ANDY” = ...

See the problem?



Letter	ASCII Code	Binary
A	065	01000001
B	066	01000010
C	067	01000011
D	068	01000100
E	069	01000101
F	070	01000110
G	071	01000111
H	072	01001000
I	073	01001001
J	074	01001010
K	075	01001011
L	076	01001100
M	077	01001101
N	078	01001110
O	079	01001111
P	080	01010000
Q	081	01010001
R	082	01010010
S	083	01010011
T	084	01010100
U	085	01010101
V	086	01010110
W	087	01010111
X	088	01011000
Y	089	01011001
Z	090	01011010

What if the keys are strings?

We need to turn the string into a number!

Maybe just add the ASCII-values?

“DANY” = $68+65+78+89 = 300$

“ANDY” = ...

See the problem?

We want to use a method that also takes the *order* of the letters into account!



Letter	ASCII Code	Binary
A	065	01000001
B	066	01000010
C	067	01000011
D	068	01000100
E	069	01000101
F	070	01000110
G	071	01000111
H	072	01001000
I	073	01001001
J	074	01001010
K	075	01001011
L	076	01001100
M	077	01001101
N	078	01001110
O	079	01001111
P	080	01010000
Q	081	01010001
R	082	01010010
S	083	01010011
T	084	01010100
U	085	01010101
V	086	01010110
W	087	01010111
X	088	01011000
Y	089	01011001
Z	090	01011010

The java string hashCode

To take account of the *order* of the letters, each value is multiplied by a power of 31.

For ANDY for example, we have

$$A = 65, N = 78, D = 68, Y = 89$$

The string hashCode turns this into the value

$$65 \cdot 31^3 + 78 \cdot 31^2 + 68 \cdot 31^1 + 89 \cdot 31^0 = \text{huge number}$$

So when we want to create a hashtable containing the key ANDY, this is the number that is put into the (modulus) hash function

The java string hashCode

```
public static void print() {  
    String example = "I love Discrete Mathematics and Algorithms";  
    System.out.println( example.hashCode() );  
    System.out.println( "hello world".hashCode() );  
}  
print()
```

1393966407

1794106052

What if two keys have the same index?

It happens that two keys are hashed to the same index. For example for the hash function

$h(\text{"key"}) = \text{"The place in the alphabet of the first letter in the key"}$

$$h(\text{"John"}) = h(\text{"Jim"})$$

which implies that the values corresponding to these keys should be stored in the same place!

This situation is referred to as a **collision**.

What do we do if there is a hash collision?

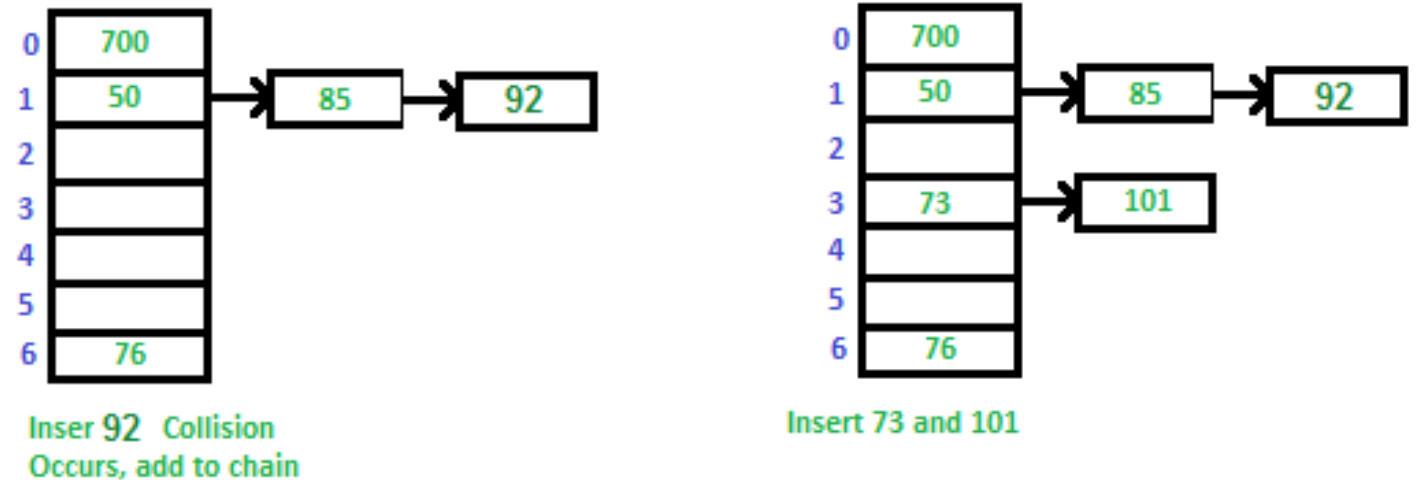
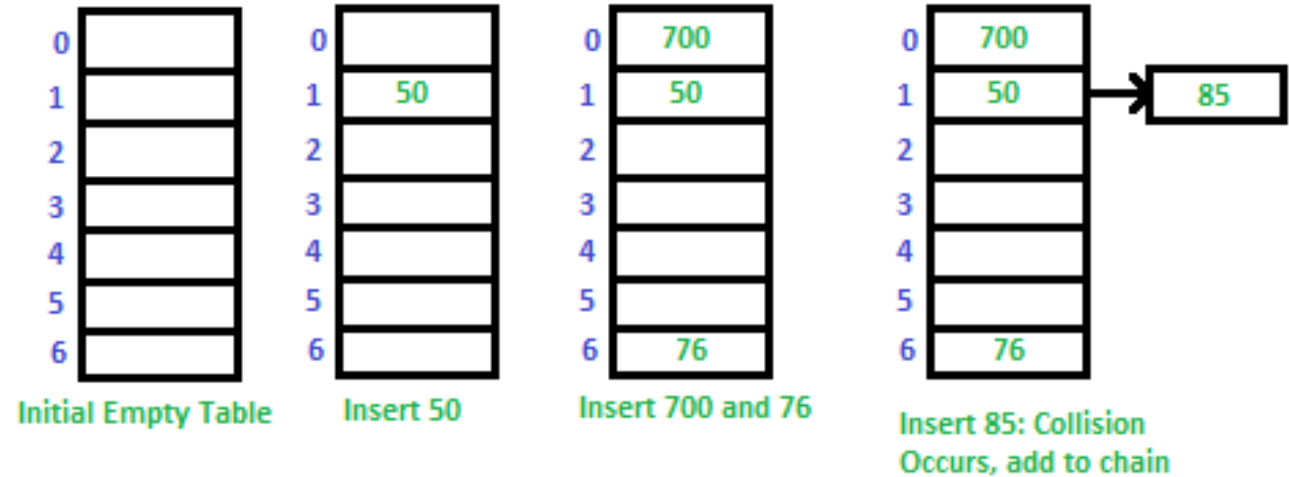
We use one of many hash collision resolution techniques to handle this, the two most popular ones are **separate chaining** and **open addressing**:

Separate chaining deals with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value.

Open addressing deals with hash collisions by finding another place within the hash table for the object to go by offsetting it from the position to which it hashed to.

Separate chaining

Hash function is
 $\text{key} \bmod 7$



Open addressing

General insertion method for open addressing on a table of size N :

```
i = 1
```

```
keyHash = H(k) mod N
```

```
index = keyHash
```

Find the index for the key using the given hash-function




If the index isn't available, then...

```
while table[index] != null:
```

```
    index = (keyHash + P(k, i)) mod N
```

```
    i = i + 1
```

... offset the index with some function $P(k, i)$ which depends on an off-setting-index i



```
insert (k, v) at table[index]
```

The three main open addressing methods

1) Linear probing

When $P(k, i) = P(i)$ is a linear function of i

```
i = 1
```

```
keyHash = H(k) mod N
```

```
index = keyHash
```

```
while table[index] != null:
```

```
    index = (keyHash + P(k, i)) mod N
```

```
    i = i + 1
```

```
insert (k, v) at table[index]
```

2) Quadratic probing

When $P(k, i) = P(i)$ is a quadratic function of i

3) Double hashing

When $P(k, i)$ depends on both k and i .

Linear probing

Example 1: $P(i) = i$

This is the simplest linear probing (and the one we assume if no function is specified). Let's try using it:

Insert the number 10, 11, 12, 17 into a hash table of size 5 using the hash function

$$H(k) = k \bmod 5.$$

$$H(10) = (10 \bmod 5 + P(0)) \bmod 5 = 0$$

$$H(11) = (11 \bmod 5 + P(0)) \bmod 5 = 1$$

$$H(12) = (12 \bmod 5 + P(0)) \bmod 5 = 2$$

$$H(17) = (17 \bmod 5 + P(0)) \bmod 5 = 2$$

Collision! Increase i and try again

$$H(17) = (17 \bmod 5 + P(1)) \bmod 5 = 3$$

This simply corresponds to “if the index is occupied, just use the next available spot”!

10	11	12	17	
0	1	2	3	4

Linear probing

Example 2: $P(i) = 3i$

Let's try this with a table of size 9 and $H(k) = k \bmod 9$.
We will try inserting the value 10 in the table to the right

$$H(10) = (10 \bmod 9 + P(0)) \bmod 9 = 1$$

Collision! Increase i and try again:

$$H(10) = (10 \bmod 9 + P(1)) \bmod 9 = 4$$

Collision! Increase i and try again:

$$H(10) = (10 \bmod 9 + P(2)) \bmod 9 = 7$$

Collision! Increase i and try again:

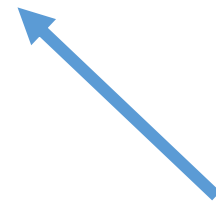
$$H(10) = (10 \bmod 9 + P(3)) \bmod 9 = 1$$

Collision! Increase i and try again:

$$H(10) = (10 \bmod 9 + P(4)) \bmod 9 = 4$$

Collision!

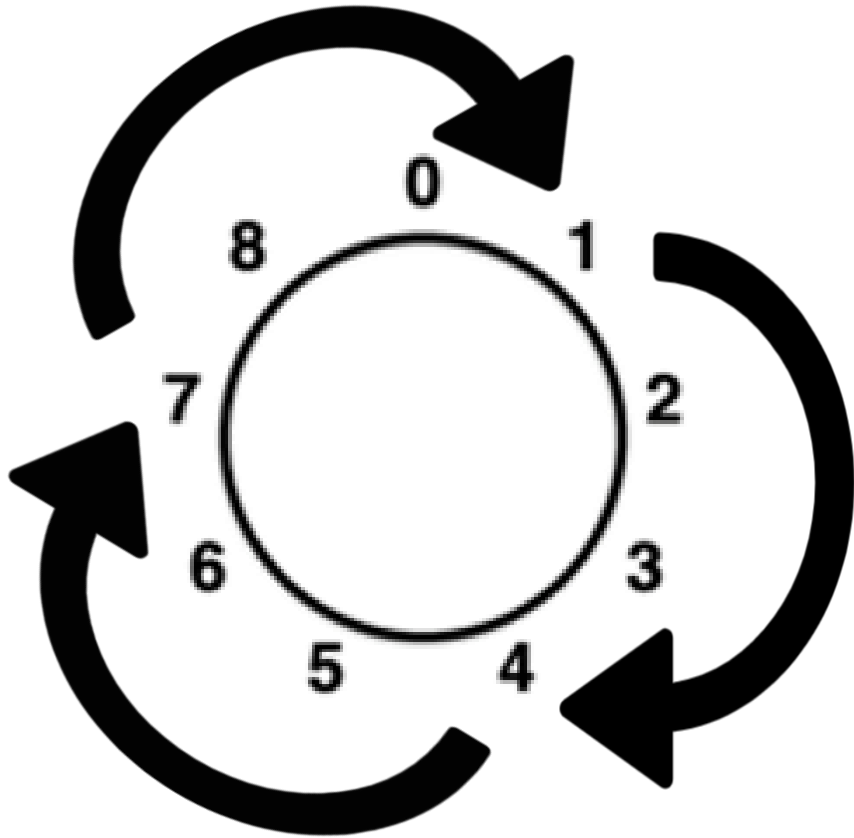
0	
1	1
2	
3	
4	19
5	
6	
7	28
8	



We end up looping
through the indices 1, 4,
and 7 infinitely!

Cycles in modular arithmetic

No matter how many times 3 is added to an index modulus 9, it can only reach two of the other indices!



How can we avoid these cycles?

... by using relatively prime numbers

If we want to be able to reach all indices in a table of length N using linear probing $P(i) = a \cdot i$, choose a to be relatively prime to N .

Example: In the table to the right, we could e.g. choose the linear probing function

$$P(i) = 5 \cdot i,$$

since $\gcd(5,9) = 1$.

0	
1	1
2	
3	
4	19
5	
6	
7	28
8	

Quadratic probing

Example: $P(i) = i^2$

Insert the number 76, 40, 48 and 5 into a hash table of size 7 using the hash function $H(k) = k \bmod 7$. In case of collisions, use quadratic probing with $P(i) = i^2$

$$H(76) = (76 \bmod 7 + P(0)) \bmod 7 = 6$$

$$H(40) = (40 \bmod 7 + P(0)) \bmod 7 = 5$$

$$H(48) = (48 \bmod 7 + P(0)) \bmod 7 = 6$$

Collision! Increase i and try again:

$$H(48) = (48 \bmod 7 + P(1)) \bmod 7 = 0$$

$$H(5) = (5 \bmod 7 + P(0)) \bmod 7 = 5$$

Collision! Increase i and try again:

$$H(5) = (5 \bmod 7 + P(1)) \bmod 7 = 6$$

Collision! Increase i and try again:

$$H(5) = (5 \bmod 7 + P(2)) \bmod 7 = 2$$

48		5			40	76
0	1	2	3	4	5	6

Double hashing

Example: $P(k, i) = i \cdot [7 - (k \bmod 3)]$

Insert the numbers 18 and 38 into a hash table of size 10 using the hash function $H(k) = k \bmod 10$.

In case of collisions, use double hashing with $P(k, i) = i \cdot [7 - (k \bmod 3)]$

sometimes called h_1

$$H(18) = (18 \bmod 10 + P(18, 0)) \bmod 10 = 8$$

sometimes called h_2

$$H(38) = (38 \bmod 10 + P(38, 0)) \bmod 10 = 8$$

Collision! Increase i and try again

$$H(38) = (38 \bmod 10 + P(38, 1)) \bmod 10 = 3$$

			38					18	
0	1	2	3	4	5	6	7	8	9

Collisions happen more often if the table is too small

Therefore, there are cases where the right thing to do is to start over using a bigger hash table!

To quantify when this is necessary, we calculate the **load factor**

Load factor

$$\text{Load factor } \lambda = \frac{\text{number of elements in hash table}}{\text{length of hash table}}$$

Example: What is the load factor of this hash table?

	44	41			18							
0	1	2	3	4	5	6	7	8	9	10	11	12

$$\lambda = \frac{3}{13} = 0.23$$

In general we want the load factor to be (significantly) less than one to avoid collisions!

Re-hashing

If λ gets close to 1, it might be necessary to create a new hash table and a new hash function.

It is typically appropriate to choose a table that is approximate twice as large. It is an advantage to use hash tables with prime-length, as they are prone to fewer collisions.

So if the old hash table had a length of N , choose the prime closest to $2N$ for the new hash table.

Combining RSA and hashing: Signing

Generate 2 large primes, p and q

p	19014104183583901283
q	50579838717232030631

(these are actually very small, should be 1024-2048 bits long)

Find their product, $n = pq$, as well as $\varphi(n) = (p - 1)(q - 1)$

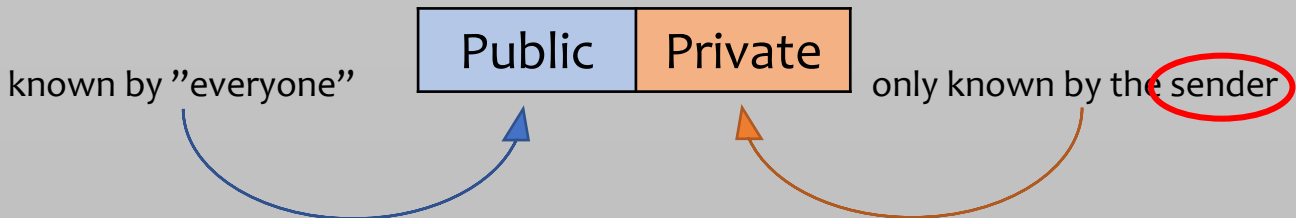
n	961730322958320540523406086367136199573
$\varphi(n)$	961730322958320540453812143466320267660

Choose an integer e , $1 < e < \varphi(n)$, such that $\gcd(e, \varphi(n)) = 1$

e	65537
-----	-------

Calculate d , the multiplicative inverse of $e \bmod \varphi(n)$

d	248367879458467356564700406307390794973
-----	---



Signing

Write your message as an integer, m , with $1 < m < n$

m	12001907081800220418141204
-----	----------------------------

Hash m using a hash function H , e.g. $h_1 = H(m) = m^2 \bmod n$

h_1	774906771682379435606426725514825375521
-------	---

Use the private key to compute a signature $s = h_1^d \bmod n$

s	320135403981232904661317443892865071211
-----	---

Send s and m to the receiver

Verification

Calculate $h_1 = s^e \bmod n$

h_1	774906771682379435606426725514825375521
-------	---

Calculate $h_2 = H(m) = m^2 \bmod n$

h_2	774906771682379435606426725514825375521
-------	---

Verify that $h_1 = h_2$