Unit #5: Hash functions and the Pigeonhole principle

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Separate Chaining
  - Open Addressing

Learning Goals

- Provide examples of the types of problems that can benefit from a hash data structure.
- Identify the types of search problems that do not benefit from hashing (e.g. range searching) and explain why.
- Evaluate collision resolution policies.
- Compare and contrast open addressing and chaining.
- Describe the conditions under which `find` using a hash table takes $\Omega(n)$ time.
- `Insert`, `delete`, and `find` using various open addressing and chaining schemes.
- Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply.

Reminder: Dictionary ADT

Dictionary operations

- create
- destroy
- insert
- find
- delete

key	value
Multics	MULTiplexed Information and Computing Service
Unics	single-user Multics
Unix	multi-user Unics
GNU	GNU's Not Unix

- insert(Linux, Linus Torvald's Unix)
- find(Unix)

Stores values associated with user-specified keys

Hash Table Goal

We can do:

a[2]= "GNU's Not Unix"

0	
1	
2	GNU's Not Unix
3	
	⋮
$m-1$	

We want to do:

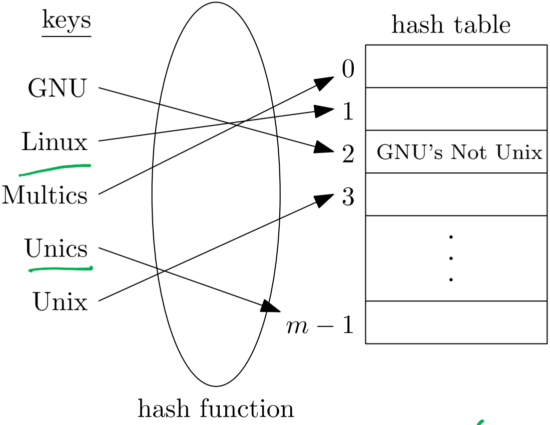
a["GNU"]= "GNU's Not Unix"

Multics	
Linux	
GNU	GNU's Not Unix
Unix	
	⋮
Unics	

associative array

Hash table approach

Choose a **hash function** to map keys to indices.

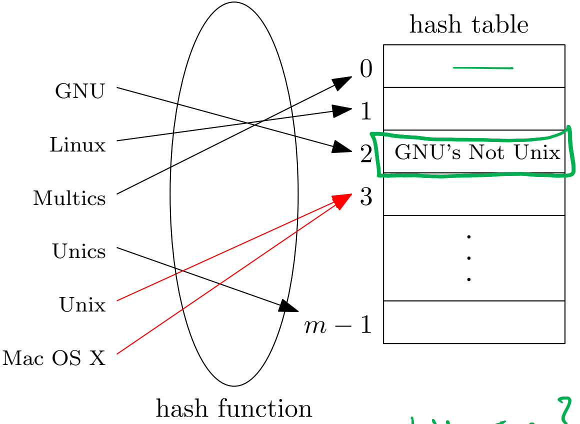


hash("GNU") = 2

– fast
– few/no collisions

Collisions

A **collision** occurs when two different keys x and y map to the same index (i.e. **slot** in table), $\text{hash}(x) = \text{hash}(y)$.



Can we prevent collisions?

increase table size?
choose better hash function?

Simple, naïve hash table code

doesn't handle collisions

```
void insert(const Key & key ) {        def
  int index = hash(key) % m;
  HashTable[index] = key;         HashEntry(key, def);
}

Value & find(const Key & key ) {
  int index = hash(key) % m;
  return HashTable[index];       could be empty
                         .def
}
```

What should the hash function, hash, be?

What should the table size, m, be?

What do we do about collisions?

Good hash function properties

Using knowledge of the kind and number of keys to be stored, we should choose our hash function so that it is:

- fast to compute, and
- causes few collisions (we hope).

Numeric keys We might use hash(x) = x mod m with m larger than the number of keys we expect to store.

Example: hash(x) = x mod 7
insert(4)
insert(17)
find(12) — No
insert(9)
delete(17)

0	
1	
2	9
3	~~17~~
4	4
5	
6	

$m = 7$

?

Hashing string keys

$$\text{hash}(s_0 \ldots s_{k-1}) = \sum_{i=0}^{k-1} s_i$$

hash("spot") = hash("pots") = hash("tops")

One option

Let string $s = s_0 s_1 s_2 \ldots s_{k-1}$ where each s_i is an 8-bit character.

$$\text{hash}(s) = s_0 + 256 s_1 + 256^2 s_2 + \cdots + 256^{k-1} s_{k-1}$$

Hash function treats string an a base 256 number.

Problems

- hash("really, really big") = well... something really, really big
- hash("anything") mod 256 = hash("anything else") mod 256

Hashing string keys with mod and Horner's Rule

$a + bx + cx^2$

$a + x(b + xc)$

```
int hash( string s ) {
  int h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (256 * h + s[i]) % m;
  }
  return h;
}
```

Compare that to the hash function from yacc:

bitwise AND

& 00111 = 7
10110 = 22
110

```
#define TABLE_SIZE 1024 // must be a power of 2
int hash( char *s ) {
  int h = *s++;
  while( *s ) h = (31 * h + *s++) & (TABLE_SIZE - 1);
  return h;
}
```

relies on null terminated array of chars

What's different?

Hash Function Summary

Goals of a hash function

- Fast to compute
- Cause few collisions

Sample hash functions

- For numeric keys x, $\text{hash}(x) = x \bmod m$
- $\text{hash}(s) =$ string as base 256 number mod m
- Multiplicative hash: $\text{hash}(k) = \lfloor m \cdot \text{frac}(ka) \rfloor$ where $\text{frac}(x)$ is the fractional part of x and $a = 0.6180339887$ (for example).

Fixed hash functions are dangerous

Good hash table performance depends on few collisions.

If a user knows your hash function, she can cause many elements to hash to the same slot. Why would she want to do that?

Denial of Service

Yacc hashes "XY" and "xy" to 769. How can you find many strings that yacc hashes to the same slot?

Protection

- Choose a new hash function at random for every hash table.
- Use a cryptographically secure hash function (such as SHA-2).

slow

Universal hash functions

$x \neq y$

A set $\mathcal{H}$ of hash functions is $\boxed{\textit{universal}}$ if the probability that hash(x) = hash(y) is at most $1/m$ when hash() is chosen at random from $\mathcal{H}$.

Example: Let p be a prime number larger than any key. Choose a at random from $\{1, 2, \ldots, p-1\}$ and choose b at random from $\{0, 1, \ldots, p-1\}$.

$$\text{hash}(x) = ((a \cdot x + b) \bmod p) \bmod m$$

General form of hash functions

1. Map key to a sequence of bytes.
   - Two equal sequences iff two equal keys.
   - Easy. The key probably is a sequence of bytes already.
2. Map sequence of bytes to an integer x.
   - Changing bytes should cause apparently **random** changes to x.
   - Hard. May be expensive. Cryptographic hash.
3. Map x to a table index using $x \bmod m$.

Collisions

Increase table size by how much?

Birthday Paradox

With probability > 1/2, two people, in a room of 23, have the same birthday. (Hash 23 people into $m = 365$ slots. Collision?)

n = # elements hashed

General birthday paradox

$n^2/2$ table size

If we *randomly* hash $\sqrt{2m}$ keys into m slots, we get a collision with probability > 1/2.

Expect $\frac{1}{365}\frac{n^2}{2} \approx 1$

potential collisions

Collision

Unless we know all the keys in advance and design a perfect hash function, we must handle collisions.

What do we do when two keys hash to the same slot?

- separate chaining: store multiple items in each slot
- open addressing: pick a next slot to try

Hashing with Chaining

Separate Chaining

Store multiple items in each slot.

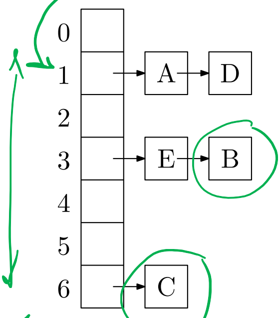
#hashed = n

How?

- Common choice is an unordered linked list (a chain).
- Could use any dictionary ADT implementation.

find(D)

m



Result

+
- Can hash more than m items into a table of size m.

−
- Performance depends on the length of the chains.

−
- Memory is allocated on each insertion.

hash(A) = hash(D) = 1 ✓
hash(E) = hash(B) = 3

Access time for Chaining

Load Factor

$$\alpha = \frac{\#\text{ hashed items}}{\text{table size}} = \frac{n}{m}$$

Assume we have a uniform hash function (every item hashes to a uniformly distributed slot).

Search cost

On average,

- an unsuccessful search examines α items.
- a successful search examines $1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$ items.

comparison that finds item

We want the load factor to be small.

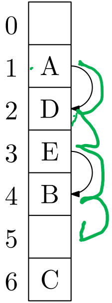
$\frac{n-1}{m}$ other items in this slot
about 1/2 come before

Open Addressing

Allow only one item in each slot. The hash function specifies a *sequence* of slots to try.

hash(A) = hash(D) = 1

hash(X) = 1



Insert If the first slot is occupied, try the next, then the next, ... until an empty slot is found.

Find If the first slot doesn't match, try the next, then the next, ... until a match (found) or an empty slot (not found).

Result

- Cannot hash more than m items into a table of size m. [Pigeonhole Principle]
- \+ Hash table memory allocated once.
- Performance depends on number of trys.

Probe Sequence

The sequence of slots we examine when inserting (and finding) a key.

key #tries

A probe sequence is a function, $h(k, i)$, that maps a key k and an integer i to a table index.
Given key k:

- We first examine slot $h(k, 0)$.
- If it's full, we examine slot $h(k, 1)$.
- If it's full, we examine slot $h(k, 2)$.
- And so on...

If all the slots in the probe sequence are full, we fail to insert the key.
The time to insert is the number of slots we must examine before finding an empty slot.

Linear probing: $h(k, i) = (\text{hash}(k) + i) \bmod m$

no deletes allowed

```
Entry *find( const Key & k ) {
  int p = hash(k) % size;
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + 1) % size;
  }
  return NULL;
}
```

Linear probing example

$h(k,i) = (k+i) \% 7$

mod

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5	insert(10) 10%7 = 3	insert(55) 55%7 = 6
0				47 ○	47	47 ●
1						55 ○
2		93 ○	93	93	93	93
3					10 ○	10
4						
5			40 ○	40 ●	40	40
6	76 ○	76	76	76 ●	76	76 ●

Access time for linear probing

If $\alpha < 1$, linear probing will find an empty slot. — good

Linear probing suffers from **primary clustering**: creation of long consecutive sequences of filled slots. (They tend to get longer and merge.)

Performance quickly degrades for $\alpha > 1/2$.

Quadratic probing: $h(k, i) = (\text{hash}(k) + i^2) \bmod m$

$P = (\text{hash}(k) + (i-1)^2)_{\bmod m}$

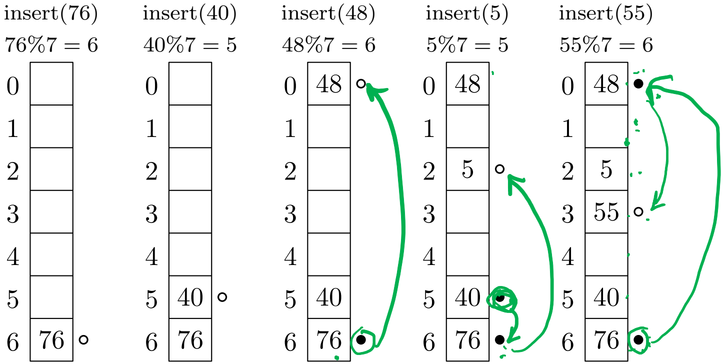
```
Entry *find( const Key & k ) {
  int p = hash(k) % size;
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + 2*i - 1) % size;
  }
  return NULL;
}
```

$\cancel{(i+1)^2 = i^2 + 2i + 1}$

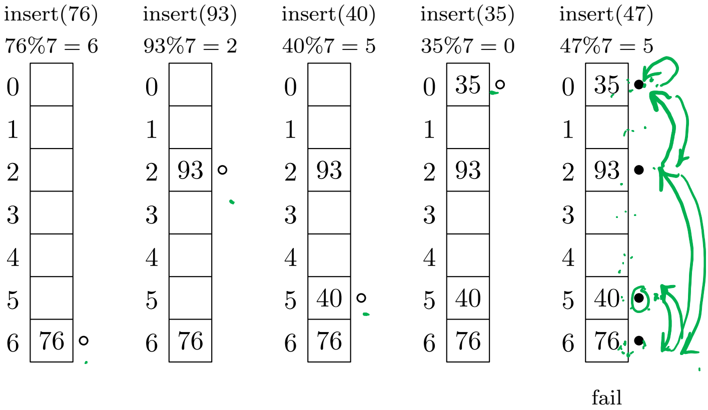
$i^2 = (i-1)^2 + 2i - 1$

Quadratic probing example

$h(k, i) = (k + i^2) \bmod 7$



Quadratic probing example



Quadratic probing: First $\lceil m/2 \rceil$ probes are distinct

good news

Claim: If m is prime, the first $\lceil m/2 \rceil$ probes are distinct.
Proof: (by contradiction) Suppose for some $0 \le i < j \le \lfloor m/2 \rfloor$,

$$
\begin{aligned}
& (\mathrm{hash}(k) + i^2) \bmod m = (\mathrm{hash}(k) + j^2) \bmod m \\
\Leftrightarrow \quad & i^2 \bmod m = j^2 \bmod m \\
\Leftrightarrow \quad & (i^2 - j^2) \bmod m = 0 \\
\Leftrightarrow \quad & (i - j)(i + j) \bmod m = 0
\end{aligned}
$$

Since m is prime, one of $(i - j)$ and $(i + j)$ must be divisible by m.
But $0 < i + j < m$ and $-\lfloor m/2 \rfloor \le i - j < 0$ because
$0 \le i < j \le \lfloor m/2 \rfloor$.

Result

If table size m is prime and there are $< \lceil m/2 \rceil$ full slots (i.e., $\alpha < 1/2$), then quadratic probing will find an empty slot.

Quadratic probing: Only $\lceil m/2 \rceil$ probes are distinct

Claim: For any $j \in \{\lceil m/2 \rceil, \lceil m/2 \rceil + 1, \ldots, m-1\}$, there is an $i \in \{1, 2, \ldots, \lfloor m/2 \rfloor\}$ such that $i^2 \bmod m = j^2 \bmod m$.

bad news

Proof: Let $i = m - j$.

$$\underline{i^2} = \underline{(m-j)^2} = \underline{m^2 - 2mj} + j^2 = j^2 \mod m.$$

For example: $m = 7$

$$\begin{aligned}
\text{hash}(k) + 0^2 &= \text{hash}(k) + 0 \mod 7 \\
\text{hash}(k) + 1^2 &= \text{hash}(k) + 1 \mod 7 \\
\text{hash}(k) + 2^2 &= \text{hash}(k) + 4 \mod 7 \\
\text{hash}(k) + 3^2 &= \text{hash}(k) + 2 \mod 7 \\
\text{hash}(k) + 4^2 &= \text{hash}(k) + 2 \mod 7 \\
\text{hash}(k) + 5^2 &= \text{hash}(k) + 4 \mod 7 \\
\text{hash}(k) + 6^2 &= \text{hash}(k) + 1 \mod 7
\end{aligned}$$

Access time for quadratic probing

- Only the first $\lceil m/2 \rceil$ slots in a quadratic probe sequence are distinct — the rest are duplicates.
+ Quadratic probing doesn't suffer from primary clustering.
- Quadratic probing suffers from **secondary clustering**: all items that initially hash to the same slot follow that same probe sequence.

How could we avoid that?

Double hashing: $h(k, i) = (\text{hash}(k) + i\ \text{hash}_2(k)) \bmod m$

calculate inc once

```
Entry *find( const Key & k ) {
  int p = hash(k) % size, inc = hash2(k);
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + inc) % size;
  }
  return NULL;
}
```

inc instead of 1

Choosing $\mathrm{hash}_2(k)$

$\mathrm{hash}_2(k)$ should:

- be quick to evaluate ✓
- differ from $\mathrm{hash}(k)$
- never be 0 $(\mathrm{mod}\ m)$

We'll use:

$$\mathrm{hash}_2(k) = r - (k \bmod r)$$

for a prime number $r < m$.

Double hashing example

$h(k,i) = [k + i \cdot (5 - (k \bmod 5))] \bmod 7$

insert(76): $76\%7 = 6$

insert(93): $93\%7 = 2$

insert(40): $40\%7 = 5$

insert(47): $47\%7 = 5$, $5 - (47\%5) = 3$

insert(10): $10\%7 = 3$

insert(55): $55\%7 = 6$, $5 - (55\%5) = 5$

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76

Access time for double hashing

$\alpha = \frac{n}{m}$ ← # things in table ← table size

+ For $\alpha < 1$, double hashing will find an empty slot (assuming *m* and hash_2 are well-chosen).

+ No primary or secondary clustering.

− One extra hash calculation.

assume probe sequence is random sequence of slots in table

Probability of success (probe empty slot) is $1-\alpha$

Expected # probes until success $= \frac{1}{1-\alpha}$

Deletion in Open Addressing

Example: $\text{hash}(k) = k \bmod 7$.

Insert(9) insert (9)

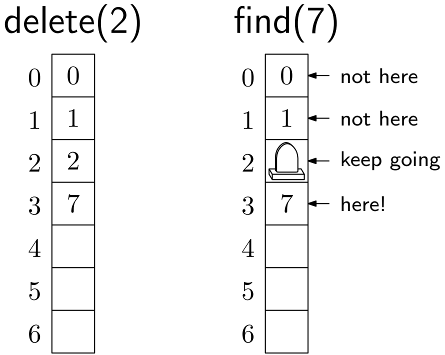
	delete(2)		~~find(7)~~	
0	0	0	0	← not here
1	1	1	1	← not here
2	2	2	9	← end of search?!
3	7	3	7	
4	9	4	9	
5		5		
6		6		

Put a tombstone in the slot.

Find Treat tombstone as an occupied slot.
Insert Treat tombstone as an empty slot.
However, you may need to Find before Insert if you want to avoid duplicate keys (which you do).

Deletion in Open Addressing

Example: hash(k) $= k \bmod 7$.



Put a tombstone in the slot.

Find Treat tombstone as an occupied slot.

Insert Treat tombstone as an empty slot.

However, you may need to Find before Insert if you want to avoid duplicate keys (which you do).

Resizable hash tables

An insert using open addressing cannot succeed with a load factor of 1 or more. [Pigeonhole Principle]

An insert using open addressing with quadratic probing may not succeed with a load factor $> 1/2$.

Whether you use chaining or open addressing, large load factors lead to poor performance!

Hint: Think resizable arrays!

Rehashing

m

When the load factor gets "too large" ($\alpha >$ some constant threshold), rehash all the elements into a new, larger table:

- takes $\Theta(n)$ time, but amortized $O(1)$ as long as we double table size on the resize
- spreads keys back out, may drastically improve performance
- gives us a chance to change the hash function
- avoids failure for open addressing techniques
- allows arbitrarily large tables starting from a small table
- clears out tombstones

The Pigeonhole Principle

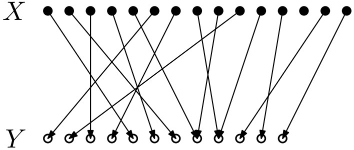
If more than m pigeons fly into m pigeonholes then some pigeonhole contains at least two pigeons.

Corollary

If we hash $n > m$ keys into m slots, two keys will collide.

The Pigeonhole Principle

Let X and Y be finite sets where $|X| > |Y|$.
If $f : X \to Y$, then $f(x_1) = f(x_2)$ for some $x_1 \neq x_2$.



The Pigeonhole Principle: Example #1

Suppose we have 5 colours of Halloween candy, and that there's lots of candy in a bag. How many pieces of candy do we have to pull out of the bag if we want to be sure to get 2 of the same colour?

pigeons = halloween candy selected

holes = colours

a. 2

b. 4

c. 6

d. 8

e. None of these

The Pigeonhole Principle: Example #2

Compression

Any lossless compression algorithm (such as `zip`, `bzip2`, Huffman coding, Sequitur, etc.) will fail to compress some file.

How many files containing n bits are there? 2^n

How many files containing fewer than n bits are there? $\sum_{i=0}^{n-1} 2^i = 2^n - 1$

What are the pigeons? pigeonholes?

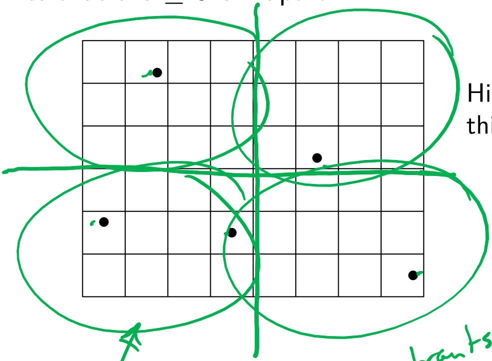
n-bit files

<n-bit files (compressed files)

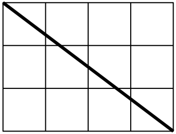
The Pigeonhole Principle: Example #3

Claim

If 5 points are placed in a 6cm x 8cm rectangle, there are two points that are $\leq$ 5 cm apart.



Hint: How long is this diagonal? 5 cm



pigeons = points

holes = quadrants.

5 points 4 quads $\Rightarrow$ 2 points in same quad.

The Pigeonhole Principle: Example #4

$x_1 \quad x_2 \quad \ldots \quad x_{n+1}$

Consider $n+1$ distinct positive integers, each $\leq 2n$. Show that one of them must divide one of the others.

odd #'s $\leq 2n$

For example, if $n = 4$, consider the following sets:

$4+1$

$$\underbrace{\{1, 2, 3, 7, 8\}}_{\text{ex 1}} \quad \underbrace{\{2, 3, 4, 7, 8\}}_{\text{ex 2}} \quad \{2, 3, 5, 7, 8\}$$

Hint: Any integer can be written as $2^k \cdot q$ where k is an integer ≥ 0 and q is odd. E.g., $129 = 2^0 \cdot 129$; $60 = 2^2 \cdot 15$.

$x_i = 2^{k_i} \cdot q_i$

there are $n+1$ q_i's

$q_1 \; q_2 \ldots q_{n+1}$

by PHP, $\exists\, i \neq j$ $q_i = q_j \Rightarrow x_i \mid x_j$ or $x_j \mid x_i$

there are $\underline{n \text{ values}}$ for the q_i's

General Pigeonhole Principle

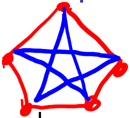
hash $2m+1$ keys into table of size m

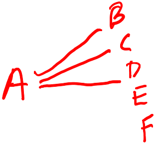
Let X and Y be finite sets with $|X| = n$, $|Y| = m$, and $k = \lceil n/m \rceil$. If $f : X \to Y$ then there exist k distinct values $x_1, x_2, \ldots, x_k \in X$ such that $f(x_1) = f(x_2) = \cdots = f(x_k)$.

Informally: If n pigeons fly into m holes, at least one hole contains at least $k = \lceil n/m \rceil$ pigeons.

Proof: Assume there's no such hole. Then there are at most $(\lceil n/m \rceil - 1)\, m < (n/m)m = n$ pigeons.

Pigeonhole Principle: Example #5





Ramsey's theorem

In any group of 6 people, where each two people are either friends or enemies (i.e., they can't be "neutral"), there must be either 3 pairwise friends or 3 pairwise enemies.

Proof: Let A be one of the 6 people. A has at least 3 friends or at least 3 enemies by the general pigeonhole principle because $\lceil 5/2 \rceil = 3$. (5 people into 2 holes (friend/enemy).)
Suppose A has ≥ 3 friends (the enemies case is similar) and call three of them B, C, and D.
If (B, C) or (C, D) or (B, D) are friends then we're done because those two friends with A forms a triple of friends.
Otherwise (B, C) and (C, D) and (B, D) are enemies and BCD forms a triple of enemies.

$R(5,5)$?? $R(4,4)=18$ $R(3,3)=6$

Pigeonhole Principle: Example #6

While on a 28-day vacation, Martina plays at least one set of tennis each day, but no more that 40 sets over all 28 days. Prove that there is a span of consecutive days in which she plays exactly 15 sets.

Proof: Let x_i be the total number of sets played up to and including day i (for $i = 1, 2, \ldots, 28$). Let $x_0 = 0$.
We need to show that there exist $0 \leq i < j < 28$ such that $x_j = x_i + 15$.
Consider $x_1, x_2, \ldots, x_{28}, x_0 + 15, x_1 + 15, \ldots, x_{27} + 15$. These are 56 integers (pigeons) in the range $[1, 39 + 15]$ (54 holes). Two of these integers are equal by the pigeonhole principle. Since $x_i < x_j$ for $i < j$ (because Martina plays ≥ 1 set per day), the two that are equal must be $x_j = 15 + x_i$. So from day $i + 1$ to day j, Martina plays 15 sets.

Pigeonhole Principle: Example #7

$r=5$
$s=5$

Erdös-Szekeres theorem

Any sequence $x_1, x_2, \ldots, x_n$ of $n \geq (r-1)(s-1)+1$ distinct numbers contains an increasing subsequence of length r or a decreasing subsequence of length s.

a_i	1	2	3	1	4	4	1	3	4	2	5	5	1	6	5	5	6
	4,	7,	12,	3,	62,	14,	2,	8,	11,	5,	20,	17,	1,	22,	15,	13,	18
b_i	1	1	1	2	1	2	3	3	3	4	2	3	5	2	4	5	3

Proof: Label x_i with the pair (a_i, b_i) where a_i is the length of the longest increasing subsequence ending with x_i and b_i is the length of the longest decreasing subsequence ending with x_i. No two numbers receive the same label since (for $i < j$) if $x_i < x_j$ then $a_i < a_j$ and if $x_i > x_j$ then $b_i < b_j$. If for all i, $a_i < r$ and $b_i < s$, then there are only $(r-1)(s-1)$ labels, so by pigeonhole, two numbers receive the same label. Contradiction.

x_i's are pigeons (n of them)

labels are pigeonholes ($(r-1)(s-1)$ if no inc. of r or dec. of s)