Today:
- Quiz 2
- Some more operations on sets
- How a computer might manipulate sets: dictionaries and disjoint-sets
  (INSERT/IN/DELETE; UNION/FIND/MAKESET)

Various laws
Prove them by tracing through the definitions

De Morgan's laws:

1. 

2. 

Proof (of first claim): 

Be careful!!

Cartesian Product (= Cross product)

\[ A \times B = \{(a,b): A \in A, B \in B\} \]

\[ \mathbb{R}^2 \] points in the plane
An array of chessmen might be represented by BYTES\textsuperscript{64}
**Unordered Product**

\[ A \& B = \{(a,b): A \in A, B \in B\} \quad \text{// when I learned graph theory -- never saw it since!} \]

**Power Set**

\[ \mathcal{P} - \text{Power set operator, unary operator (takes 1 input). } \mathcal{P}(x) \text{ is the “set of}\]

all subsets of \( x \)"

\[ \mathcal{P}(X) = \{A: A \subseteq X\} \]

Example: \( X = \{a, b, c\} \)

Example:

Variant notation: \( \mathcal{P}(X) = 2^X \)

Notation is suggestive of size –

For \( X \) finite, \( |\mathcal{P}(X)| = 2^{|X|} \)

**Dictionary ADT**

and its realization with a list and with a hash table

Want to be able to **Insert** items into a dictionary and to **Lookup** if an item is already in the dictionary. (Sometimes want to be able to **Delete** an item, too.) For concreteness, think of the items we are inserting as strings.

Example: discover how many distinct words are in a book.

Implementation

1) A **list** of words, each one appearing at most once.

2) A **hash table**.

Explain how each works.

Show how to modify the hash table to do a frequency count.

**Representing a collection of sets in a computers**

A different game – we are going to maintain a collection of **disjoint sets**. We want to be able to figure out if two things are in the same set, or in different sets. For example, each point in the set might represent a person and when we learn that person one and person two know one another – maybe one calls or emails the other – then we combine them. Each set then represents people that know one another through **some path** of knowing.
More interesting applications will come later, when we do graph theory.
You want to realize

- **find(x)** return a *canonical name* for the unique set containing x.
  x and y are in the same set iff find(x)=find(y)
- **union(x,y)** *merge the sets containing x and y.*
- **makeset** (x) create a set containing the element x. Return a canonical name for it

Naïve implementation: list of elements

Smarter – “union/find data structure”
Union by rank
Collapsing find.
Any sequence of \( n \) operations takes \( n \alpha(n) \) time, for an incredibly slows growing function \( \alpha(n) \). [Omit big-O because not yet introduced]

Tarjan (1975)

```plaintext
function MakeSet(x)
    x.parent := x
    x.rank := 0
function Union(x, y)
    xRoot := Find(x)
    yRoot := Find(y)
    if xRoot == yRoot
        return
    if xRoot.rank < yRoot.rank
        xRoot.parent := yRoot
    else if xRoot.rank > yRoot.rank
        yRoot.parent := xRoot
    else
        yRoot.parent := xRoot
```

// x and y are not already in same set. Merge them.
if xRoot.rank < yRoot.rank
    xRoot.parent := yRoot
else if xRoot.rank > yRoot.rank
    yRoot.parent := xRoot
else
    yRoot.parent := xRoot
```
xRoot.rank := xRoot.rank + 1

The second improvement, called path compression, is a way of flattening the structure of the tree whenever Find is used on it. The idea is that each

function Find(x)
    if x.parent != x
        x.parent := Find(x.parent)
    return x.parent