



# CS1113

# Foundations of Computer Science I

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# CS1113

## Foundations of Computer Science II

Lectures:

Monday, 14:00, WGB 107

Wednesday 15:00, WGB G.05

Problem Solving Classes (not starting yet):

Monday 10:00-12:00, WGB G20

Monday 15:00 -17:00, WGB G20

# Formal Module Description

- **Module Objective:** to develop advanced skills in the foundational techniques needed to analyse, design, implement and communicate computational problems and solutions.
- **Module Content:** Predicate logic; representing and solving computational problems with trees and graphs; analysis of simple data structures, algorithms and problem spaces.
- **Learning Outcomes:**
  - Formulate computational problems using predicate logic specifications;
  - Represent and solve computational problems with trees and graphs;
  - Analyse simple data structures and algorithms.

# Informal Module Description

- **Objective:**

- learn how to express complex problems clearly and precisely
- learn some important abstract structures
- learn some important algorithms
- learn how to analyse problems and solutions

- **How are you going to achieve it?**

- keep up with the module as you are going along
- attend all lectures, and participate in them
- in the problem classes, work on the problems
- submit the assignment each week

<http://osullivan.ucc.ie/teaching/cs1113/>

# Assessment

- **Assessment:**
  - Formal Written Examination 80 marks;
  - Continuous Assessment 20 marks (In-Class Tests 20 marks)
  - (there will be 2 in-class tests)
- **Formal Written Examination:**
  - 1 x 1½ hr(s) paper(s) to be taken in Summer 2015.

# Relation to CS1112

- this module assumes you understand and are comfortable with **everything** covered in CS1112:
  - sets, functions, relations, propositional logic, algorithms
- you are allowed to register for CS1113 as long as you remained registered for CS1112 up to the end of the 1<sup>st</sup> term
  - formally, you do not require a pass in CS1112
  - but if you didn't get a pass, you will struggle, and you will have to work hard now to catch up
- provisional marks for CS1112 will be released in early February and final confirmed marks will be released in June

# Formal Module Description

- **Module Objective:** to develop advanced skills in the foundational techniques needed to analyse, design, implement and communicate computational problems and solutions.
- **Module Content:** **Predicate logic**; representing and solving computational problems with trees and graphs; analysis of simple data structures, algorithms and problem spaces.
- **Learning Outcomes:**
  - Formulate computational problems using predicate logic specifications;
  - Represent and solve computational problems with trees and graphs;
  - Analyse simple data structures and algorithms.



# CS1113

## Introduction to Predicate Logic

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Introduction to Predicate Logic

A brief review of propositional logic  
Why we need a more expressive logic  
Predicates

# Brief review of propositional logic

1. write specifications and descriptions of situations precisely and unambiguously
  2. understand when one situation is implied by another (and so we can prove that an argument is valid, or that some initial assumptions lead to a conclusion)
- *propositions* are statements that are either true or false
    - propositional symbols  $\{p, q, r, \dots\}$  represent propositions
  - *connectives* combine propositions to get larger more complicated statements
    - $\neg p$ ,       $p \wedge q$ ,       $p \vee q$ ,       $p \rightarrow q$ ,       $p \leftrightarrow q$   
 $(NOT)$      $(AND)$      $(OR)$      $(IF \dots THEN)$      $(IF \text{ AND ONLY IF})$

# Truth Tables

- We specified how to determine the truth or falsehood of a combined statement by considering the truth value of the propositions and the connective which joins them

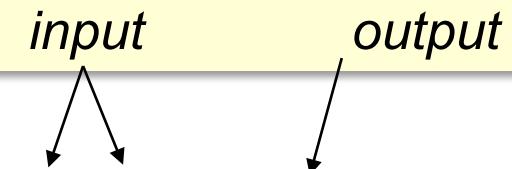
$X$	$\neg X$
T	F
F	T

$X$	$Y$	$X \vee Y$
T	T	T
T	F	T
F	T	T
F	F	F

$X$	$Y$	$X \wedge Y$
T	T	T
T	F	F
F	T	F
F	F	F

$X$	$Y$	$X \rightarrow Y$
T	T	T
T	F	F
F	T	T
F	F	T

Note function representation:

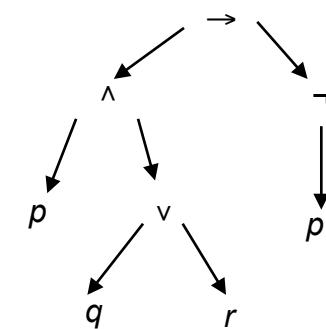


$X$	$Y$	$X \leftrightarrow Y$
T	T	T
T	F	F
F	T	F
F	F	T

# Building truth tables for larger statements

- write down a column for each propositional symbol that appears in the statement
- write a row for each possible combination of truth values for the atomic propositions
- work out the structure of the statement by building a tree
- each connective in the tree gets a new column in the table
- add the columns in order, starting at the bottom of the tree, and only adding a column for a connective when all its branches have been done
- complete the truth values for each column in turn using the basic truth tables for each connective

$$(p \wedge (q \vee r)) \rightarrow \neg p$$



p	q	r	qvr	$p \wedge (q \vee r)$	$\neg p$	$(p \wedge (q \vee r)) \rightarrow \neg p$
T	T	T	T	T	F	F
T	T	F	T	T	F	F
T	F	T	T	F	F	F
T	F	F	F	F	F	T
F	T	T	F	F	T	T
F	T	F	F	F	T	T
F	F	T	F	F	T	T
F	F	F	F	F	T	T

determines exactly those conditions in which the large statement is true, and those in which it is false

# Logical Equivalence

- two statements are logically equivalent if and only if they have the same truth values in all situations

E.g.

$$\neg(X \vee Y) \equiv \neg X \wedge \neg Y$$

$$\neg(X \wedge Y) \equiv \neg X \vee \neg Y$$

$$X \rightarrow Y \equiv \neg X \vee Y$$

X	Y	$\neg X$	$\neg Y$	$\neg X \wedge \neg Y$	$X \vee Y$	$\neg(X \vee Y)$
T	T	F	F	F	T	F
T	F	F	T	F	T	F
F	T	T	F	F	T	F
F	F	T	T	T	F	T

# Valid Arguments

- a valid argument is a set of initial statements and a sequence of new statements, such that each new statement follows logically from the previous ones
  - if the earlier statements are true, then the new statement must also be true
- a valid argument does not state that the conclusion is true; it states that *if* the initial facts are true, *then* the conclusion must be true
- we could show a conclusion is valid by building a large truth table, consider only the rows where all initial statements are true, and show that the conclusion is also true

# Rules of Inference

- We can construct a valid argument using rules of inference, which describe when we can add a new statement

$$x \rightarrow y$$

$$\underline{x}$$

$$\therefore y$$

*Modus Ponens*

$$\underline{x \wedge y}$$

$$\therefore x$$

*Simplification*

$$x \rightarrow y$$

$$\underline{y \rightarrow w}$$

$$\therefore x \rightarrow w$$

*Chaining*

$$x \rightarrow y$$

$$\underline{\neg y}$$

$$\therefore \neg x$$

*Modus Tollens*

$$\underline{x}$$

$$\therefore x \vee y$$

*Addition*

$$x \leftrightarrow y$$

$$\underline{x}$$

$$\therefore y$$

*Elimination*

$$x \vee y$$

$$\underline{\neg x}$$

$$\therefore y$$

*Unit Resolution*

$$x$$

$$\underline{y}$$

$$\therefore x \wedge y$$

*Conjunction*

$$x \vee y$$

$$\underline{\neg y \vee w}$$

$$\therefore x \vee w$$

*Resolution*

# Examining a rule of inference

$$\begin{array}{c} x \vee y \\ \neg x \\ \hline \therefore y \\ \text{Unit Resolution} \end{array}$$

Whenever we have statements that match those above the line, we can add the one below the line, because it is implied: if the statements above the line are true, then the statement below the line must also be true

Proving the rule:

We are only interested in situations where

i.  $x \vee y$  is true

and

ii.  $\neg x$  is true

and in those cases we see that  $y$  must be true

X	Y	$\neg X$	$X \vee Y$
T	T	F	T
T	F	F	T
F	T	T	T
F	F	T	F

# Proving a conclusion is implied by some assumptions

From the premises

1.  $p \rightarrow q$
2.  $\neg q \wedge r$
3.  $\neg p \rightarrow s$
4.  $s \rightarrow t$

derive the conclusion

$t$

1.  $p \rightarrow q$
2.  $\neg q \wedge r$
3.  $\neg p \rightarrow s$
4.  $s \rightarrow t$
5.  $\neg q$     *from 2 by (i)*     $[x=\neg q, y=r]$
6.  $\neg p$     *from 1 & 5 by (ii)*     $[x=p, y=q]$
7.  $s$     *from 3 & 6 by (iii)*     $[x=\neg p, y=s]$
8.  $t$     *from 4 & 7 by (iii)*     $[x=s, y=t]$

(i)	(ii)	(iii)
$\underline{x \wedge y}$	$x \rightarrow y$ $\underline{\neg y}$	$x \rightarrow y$ $\underline{x}$
$\therefore x$	$\therefore \neg x$	$\therefore y$

Give each new statement a number.  
Say what other statements it came  
from, using what rule, and with what  
substitutions.

## So what have we got?

We now have a procedure for proving whether one statement follows on from the truth of a collection of other statements.

- we can work out the consequences of claims we make, or parameters we set, or systems that we design
- we can analyse arguments to see if they are valid

The procedure does **not** depend on the meaning of the basic propositions

- the procedure can be automated

All that remains is to express what we want to say in terms of basic propositions and connectives.

# Example

Given the facts:

1. "if the user does not provide the correct password then the user is given access to the open area"
2. "if the user can read sensitive data then the user must have access to the restricted area"
3. "the user cannot have access to the open area at the same time as the user has access to the restricted area"
4. "the user has not provided the correct password"

show that the conclusion

"the user cannot read the sensitive data"

must follow, by representing the facts and conclusions as propositional statements and constructing a valid argument

$p$  = the user provides the correct password

$o$  = the user has access to the open area

$r$  = the user has access to the restricted area

$s$  = the user can read the sensitive data

1.  $\neg p \rightarrow o$
2.  $s \rightarrow r$
3.  $\neg(o \wedge r)$
4.  $\neg p$
5.  $\neg o \vee \neg r$  *from 3, L.E*
6.  $o$  *from 4&1, M.P.*
7.  $\neg r$  *from 5&6, U.R.*
8.  $\neg s$  *from 2&7, M.T.*

## Is there anything left to do?

Suppose we have determined the following fact:

*the college network is secure if and only if every computer connected to the network has an active copy of FireGuard™ security suite*

We can check the status of *FireGuard™* on any computer.  
There are thousands of computers on the network.

How do we represent this using propositions so that  
(i) we can conclude that the network is secure, or  
(ii) when we detect that an individual computer has  
*FireGuard™* turned off, we conclude that the network is not  
secure?

*the college network is secure if and only if every computer connected to the network has an active copy of FireGuard™ security suite*

p: the college network is secure

q: every computer connected to the network has an active copy of *FireGuard™* security suite

$p \Leftrightarrow q$

I now tell you that computer x123g has an inactive *FireGuard™*

?

*the college network is secure if and only if every computer connected to the network has an active copy of FireGuard™ security suite*

Perhaps rewrite the sentence as

*the college network is secure if and only if computer x1 has an active copy of FireGuard™ security suite and x2 has an active copy of FireGuard™ security suite and x3 has an active copy of FireGuard™ security suite and ... x3926 has an active copy of FireGuard™ security suite*

I now add a new computer x3927 to the network, and it has an inactive *FireGuard™*

?

# Introducing Predicates and Quantifiers

We need to expand our logical language to talk about sets of objects. We will do this using *predicates* and *quantifiers*.

**Predicates** are relations acting on elements of sets. **Quantifiers** say how many elements must satisfy the relation.

Instead of saying

- *The network is secure if and only if computer x<sub>1</sub> has an active copy of FireGuard™ security suite and x<sub>2</sub> has an active copy of FireGuard™ security suite and x<sub>3</sub> has an active copy of FireGuard™ security suite and ... x<sub>3926</sub> has an active copy of FireGuard™ security suite*

we will go back to the original statement, and say

The network is secure if and only if

all computers on the network have an active copy of FireGuard™

quantifier

element in a set

predicate

# Notation for predicates

Each predicate will have a name

- the name may be a word or a single capital letter

Each predicate must be applied to a specified number of objects taken from specified sets (or to variables representing those objects)

We write the objects and variables in the specified order between round brackets after the predicate name. E.g.

has\_firewall(comp23)

is\_connected\_to(comp23,cs\_domain)

less\_than(5,9)

P(brian)

objects

has\_firewall(x)

less\_than(x, y)

Q(x,y)

variables

# Predicates and variables

In " $x > 7$ ",  $x$  is a **variable**, and " $> 7$ " is a predicate, which is applied to a single integer (i.e. an element of  $\mathbb{Z}$ )

If we assign the value 9 to  $x$ , the statement is true ( $9 > 7$ ).

If we assign the value 1 to  $x$ , the statement is false ( $1 > 7$ ).

In " $x > y$ ",  $x$  and  $y$  are variables, and " $>$ " is a predicate, which is applied to two integers (and so " $>$ " is a relation over  $\mathbb{Z} \times \mathbb{Z}$ ).

If we assign 5 to  $x$  and 1 to  $y$ , the statement is true ( $5 > 1$ ).

If we assign 3 to  $x$  and 9 to  $y$ , the statement is false ( $3 > 9$ ).

The assignment of a value to each variable in the statement turns the statement into a proposition.

## Interpreting predicates

- Suppose we let  $P(x)$  stand for the statement " $x > 7$ "

The predicate name is  $P$ , and it applies to a single object in  $Z$

$P(4)$  is the statement " $4 > 7$ " (which has truth value F)

$P(9)$  is the statement " $9 > 7$ " (which has truth value T)

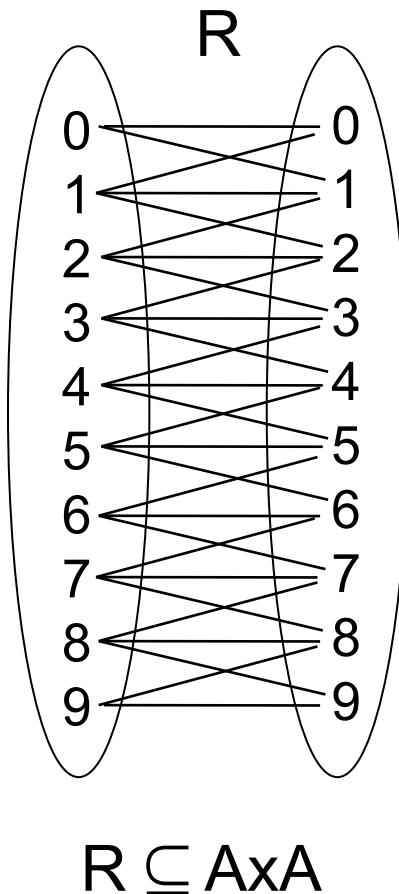
$P(x)$  has no truth value

Example: what are the truth values, if any, of the following?

- (i)  $P(1)$
- (ii)  $P(y)$
- (iii)  $P(12)$

# Predicates and Relations

Let  $R$  be the relation "is no more than 1 different from", applied to two elements of  $A = \{0,1,2,3,4,5,6,7,8,9\}$



In terms of relations, for  $x \in A$  and  $y \in A$ , if  $(x,y) \in R$ , we wrote  $xRy$  or  $R(x,y)$

So we wrote  $R(4,5)$  and  $R(7,7)$ , but we wrote  $\neg R(9,1)$ .

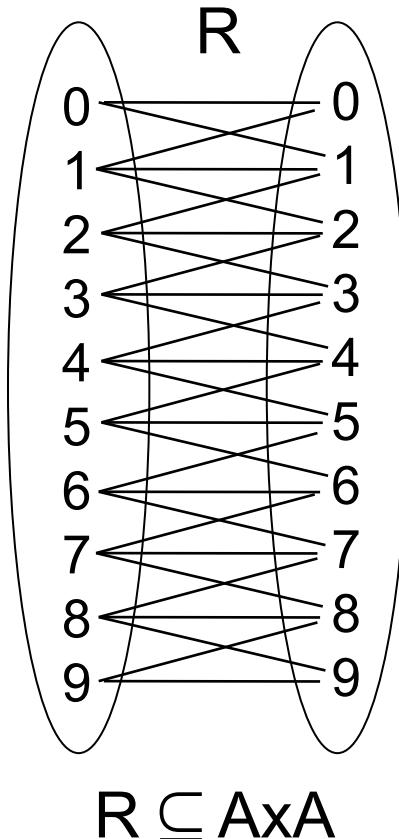
In logic, we say  $R(4,5)$  and  $R(7,7)$  are true, but  $R(9,1)$  is false.

(or we may say  $\neg R(9,1)$  is true).

# Predicates and Functions

We can also think of a predicate as a function mapping from the sets to {T,F}

"is no more than 1 different from"



A	A	{T,F}
0	0	T
0	1	T
0	2	F
0	3	F
...		
1	0	T
1	1	T
1	2	T
1	3	F
...		
2	0	F
2	1	T
2	2	T
2	3	T
2	4	F
...		

# Unary predicates

The truth-function representation makes it easy to think of predicates which act on a single element.

Let  $C$  be a set of computers. The predicate "has\_firewall" acts on this set, and specifies which computer has a firewall.

has_firewall	
$C$	$\{T, F\}$
comp1	T
comp2	T
comp3	F
comp4	T
comp5	F
...	

"has\_firewall" is a **unary** predicate (or unary relation) because it acts on a single set.

## Predicates: Further examples

A predicate is a relation acting on one or more sets. Each predicate is applied to a specified number of objects from specified sets, listed in a specified order.

E.g. if C is a set of national football teams , then we might have a predicate "*inWC2014*", which applies to a single object from the set C, and is true if the team qualified for the World Cup 2014 finals.

*inWC2014(England)* is true  
*inWC2014(Ireland)* is false

$$inWC2014 \subseteq C$$

E.g. we might have a predicate "*sameGroup*", which applies to two objects from C, and is true if the two teams were in the same group in *WorldCup2014*

*sameGroup(Italy, England)* is true  
*sameGroup(England, Germany)* is false  
*sameGroup(England, Scotland)* is false

$$sameGroup \subseteq C \times C$$

# Using predicates with connectives

We can combine predicates using connectives

$inWC2014(\text{Brazil}) \wedge inWC2014(\text{Mexico}) \wedge sameGroup(\text{Brazil}, \text{Mexico})$

(which happens to be a true statement)

Define a predicate *beats*, which applies to two teams from C, and which is true if the first team beats the second, and a predicate *winsGroup*, which applies to one team from C, and which is true if the team wins its group.

$(beats(\text{Brazil}, \text{Mexico}) \wedge beats(\text{Brazil}, \text{Croatia}) \wedge beats(\text{Brazil}, \text{Cameroon})) \rightarrow winsGroup(\text{Brazil})$

(which is also a true statement)

## So what have we done?

Previously, a proposition was simple atomic statement which was either true or false, and had no internal structure.

We can now talk about

- sets of elements
- cross-products of sets of elements
- relations over sets of elements
- whether or not a tuple of elements satisfies a relation and we can use these to create propositions.

We can also use *variables* instead of elements, to get statements which are not (yet) propositions.

Next we will define a way to talk about these statements which contain variables.

# Next lecture

Quantifiers



# CS1113

## Quantifiers in Predicate Logic

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Quantifiers

$\forall$  -- the universal quantifier

$\exists$  -- the existential quantifier

truth values of quantified statements

translating specifications into quantified logic with predicates

$$\neg \forall x P(x) \equiv \exists x \neg P(x)$$

## Previous Lecture

We introduced sets and relations, and allowed our propositions to talk about elements that satisfy a relation (or *predicate*).

`is_connected_to(comp23,cs_domain)`  
`(beats(Brazil,Mexico) ∧ beats(Brazil,Croatia) ∧ beats(Brazil,Cameroon))`  
→  
`winsGroup(Brazil)`

We introduced *variables*, which could stand for any element of a set, but that means the statements are not propositions (because they are neither definitely true nor definitely false)

`has_firewall(x)`  
`less_than(x, y)`  
`sameGroup(x,Australia) → beats(x,Australia)`

## The aim

We now want to be able to talk about these statements with variables, and have a procedure to determine whether what we say is true or false

- Is there a value for the variable that makes the statement true?
- Which values make the statement true?
- How many values make the statement true?
- Is the statement true for all possible values we could give to the variable?

"all computers on the network have an active copy of *FireGuard™* "

Is the statement `hasActiveFireGuard(x)` true no matter which computer on the network we replace `x` with?

From now on, we will assume all predicates are defined over some universal set  $U$  (or over cross products of the universal set  $U \times U \times \dots \times U$ ).

# The universal quantifier

To state that a predicate is true for all possible assignments of values to its variables, we use a special symbol  $\forall$ , called the **universal quantifier**.

If we write  $\text{firewall}(x)$  to represent the statement that  $x$  has an active firewall, then to say all possible objects have an active firewall, we write:

$$\forall x \text{ firewall}(x)$$

Note: think of  $\forall$  as an upside-down A, standing for "for All"

Exercise:

if  $Q(x)$  is the statement  $x < x+1$ , where  $U = \mathbb{Z}$ , what does

$$\forall x Q(x)$$

say? Is it true or false?

# Truth value of the universal quantifier

$\forall x P(x)$

is true if in all possible assignments of a value to  $x$ ,  $P(x)$  is true

$\forall x P(x)$

is false if there is one or more possible assignments of a value to  $x$  that makes  $P(x)$  false

## Examples

$R(x)$  is the statement  $x^2 \geq 0$  where  $U = \mathbb{Z}$

$\forall x R(x)$  is a true statement

$R(5)$  is true, since  $5^2 = 25 \geq 0$

$R(-3)$  is true, since  $(-3)^2 = 9 \geq 0$

$P(x)$  is the statement  $x > 7$  where  $U = \mathbb{Z}$

$\forall x P(x)$  is a false statement

$P(3)$  is false, since  $3 \not> 7$

# The existential quantifier

To state that a predicate is true for at least one assignment of values to its variables, we use the special symbol  $\exists$ , called the **existential quantifier**.

If  $\text{firewall}(x)$  represents the statement that  $x$  has an active firewall, then to say at least one object has an active firewall, we write:

$$\exists x \text{ firewall}(x)$$

Note: think of  $\exists$  as backwards E, standing for "there Exists"

Exercise: if  $Q(x)$  is the statement  $x = x^2$ , and  $U=\mathbb{Z}$ , what does

$$\exists x Q(x)$$

say? Is it true or false?

# Truth value of the existential quantifier

- |                  |  |
|------------------|--|
| $\exists x P(x)$ | is true if there is one or more assignments of a value to $x$ such that $P(x)$ is true |
| $\exists x P(x)$ | is false if there is no possible assignment of a value to $x$ that makes $P(x)$ true   |

## Examples

$R(x)$  is the statement  $x \geq 0$  where  $U = \mathbb{Z}$

$\exists x R(x)$  is a true statement

$R(5)$  is true, since  $5 \geq 0$

$P(x)$  is the statement  $x^2 < 0$  where  $U = \mathbb{Z}$

$\exists x P(x)$  is a false statement

There is no integer whose square is less than 0

# Examples

Consider students currently registered on the UCC database.

Represent the following using quantifiers and predicates:

1. All students have a student ID number.
2. Some students are American

Let  $U$  be the set of UCC students

Let  $\text{hasID}(x)$  state that  $x$  has a student ID number

Let  $\text{american}(x)$  state that  $x$  is american

1.  $\forall x \text{ hasID}(x)$
2.  $\exists x \text{ american}(x)$

# Examples

Consider students currently registered on the UCC database, and degree programmes in the UCC Calendar

Represent the following using quantifiers and predicates:

1. There exists at least one student who studies both Computer Science and Chinese.

Let  $U$  be the set of UCC students and degree programmes

Let  $\text{student}(x)$  state that  $x$  is a student

Let  $\text{studies}(x,y)$  state that student  $x$  studies degree program  $y$

Let CS and CH be elements of  $U$ , representing Computer Science and Chinese.

1.  $\exists x (\text{student}(x) \wedge \text{studies}(x, \text{CS}) \wedge \text{studies}(x, \text{CH}))$

## Example

Let  $P(x)$  be " $x > 0$ " and let  $Q(x)$  be " $2^*x > 0$ ", where  $U=\mathbb{Z}$

Then the wff

$$\forall x (P(x) \rightarrow Q(x))$$

says for all integers, if  $x > 0$ , then  $2^*x > 0$ .

This is a true statement. For any integer  $x$ ,  $P(x) \rightarrow Q(x)$  is true.

(using the truth table for  $\rightarrow$ , we cannot find an integer which makes  $P(x)$  true and  $Q(x)$  false. i.e. we cannot find an integer  $x$  for which  $x > 0$  but  $2^*x \leq 0$ )

## Example

For all students in UCC, if the student is qualified to enter 2<sup>nd</sup> year, then the student must have earned 50 credits.

Express this in logic.

Let  $U$  = all students in the UCC database

Let  $\text{credits}(x)$  mean  $x$  has achieved 50 credits, for  $x \in U$

Let  $\text{qualified}(x)$  mean  $x$  is qualified to enter 2<sup>nd</sup> year, for  $x \in U$

$$\forall x \ (\text{qualified}(x) \rightarrow \text{credits}(x))$$

## Example

For all students in UCC CS 1<sup>st</sup> year, either the student is registered for CS1105, or has a pass for CS1105, or has a pass for MA1015

Express this in logic.

## Class Exercise

If  $U$  the set of all international soccer teams, and *european* is a unary predicate which is true if the team is european, what do the following say?

- 1)  $\forall x (inWC2014(x) \rightarrow \text{european}(x))$
- 2)  $\exists x (inWC2014(x) \wedge \text{sameGroup}(x, \text{Spain}))$

Are these statements true or false?

## Two Equivalences

Let  $P$  be a predicate that acts on a single object from  $U$

$$\neg \exists x P(x) \equiv \forall x \neg P(x)$$

"it is not true that there is an object  $x$  that has property  $P$ " is logically equivalent to "for every possible object  $x$ , the property  $P$  is not held by  $x$

$$\neg \forall x P(x) \equiv \exists x \neg P(x)$$

"it is not true that all objects  $x$  have property  $P$ " is logically equivalent to  
"there is at least one object  $x$  which does not have property  $P$

$$\neg \forall x \text{inWC2014}(x) \equiv \exists x \neg \text{inWC2014}(x)$$

# Next lecture

the language of quantified logic (version 1)

logical equivalences

examples



# CS1113

# Writing Statements in Predicate Logic

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Writing Predicate Logic Statements

Equivalence:  $\neg \forall x P(x) \equiv \exists x \neg P(x)$

Translating and understanding statements

The language of predicate logic (v1)

## Two Equivalences

Let  $P$  be a predicate that acts on a single object from  $U$

$$\neg \exists x P(x) \equiv \forall x \neg P(x)$$

"it is not true that there is an object  $x$  that has property  $P$ " is logically equivalent to "for every possible object  $x$ , the property  $P$  is not held by  $x$

$$\neg \forall x P(x) \equiv \exists x \neg P(x)$$

"it is not true that all objects  $x$  have property  $P$ " is logically equivalent to  
"there is at least one object  $x$  which does not have property  $P$

$$\neg \forall x \text{inWC2014}(x) \equiv \exists x \neg \text{inWC2014}(x)$$

## Example

Let the domain be the set of students in the UCC database.  
Let  $\text{president}(x)$  be the predicate which is true when  $x$  is the President of Ireland.

There is no student who is the President of Ireland.

Rewrite as: there does not exist a student such that the student is the President of Ireland.

In logic:  $\neg \exists x \text{ president}(x)$   
which is equivalent to  $\forall x \neg \text{president}(x)$

which says for each student, the student is not the President of Ireland.

Proof of:  $\neg \exists x P(x) \equiv \forall x \neg P(x)$

$\exists$  is the existential quantifier

$\exists x f_1$ , says there is at least one value we could assign to  $x$  that would make  $f_1$  true

Suppose I claim  $\neg \exists x P(x)$

- Then I am claiming *it is not true there is at least one value I could assign to  $x$  that makes  $P(x)$  true*.
- So I am claiming *there is no value I could assign to  $x$  that makes  $P(x)$  true*.
- So I am claiming *every single value I try to assign to  $x$  fails to make  $P(x)$  true*, no matter what that value is
- But assigning a value to  $x$  in  $P(x)$  turns it into a proposition, which must be either true or false
- So I am claiming *every value I assign to  $x$  makes  $P(x)$  false*
- So I am claiming *every value I assign to  $x$  makes  $\neg P(x)$  true*
- So I am claiming  $\forall x \neg P(x)$

Exercise: do the other direction

## Class Exercise

What do the following say? How could you rewrite them?

$$\neg \forall x (\text{european}(x) \rightarrow \text{inWC2014}(x))$$

$$\neg \exists x (\neg \text{inWC2014}(x) \wedge \text{sameGroup}(x, \text{Spain}))$$

## Example

Let the domain be the set of all computers connected to the campus network. Translate the following into logic, and then rearrange it into an equivalent formula.

"There does not exist a computer which has no virus protection and which is accredited by the Computer Centre."

# Review: the language of propositional logic

- to make things more precise, we defined a language of propositional logic, by specifying what were *well-formed formulae (wffs)*:
  - We use:
    - propositional symbols:  $\{p, q, r, \dots, p', p'', \dots\}$
    - logical connectives:  $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$
    - punctuation:  $\{ (, ) \}$
1. All propositional symbols on their own are wffs
  2. if  $p$  and  $q$  are wffs, then so are  
 $(p)$ ,  $(\neg p)$ ,  $(p \wedge q)$ ,  $(p \vee q)$ ,  $(p \rightarrow q)$  and  $(p \leftrightarrow q)$
  3. nothing else is a wff unless it can be formed by repeatedly applying rules 1 and 2 above.

# The language of predicate logic (version 1)

$U$  is the universal set (or **domain of discourse**) of constants

$V$  is a set of variables, such that  $V \cap U = \emptyset$

$\Pi$  is a set of predicate symbols such that  $\Pi \cap V = \Pi \cap U = \emptyset$

1. if  $P$  is a predicate symbol  $\in \Pi$  and  $t_1, t_2, \dots, t_i \in V \cup U$   
then  $P(t_1, t_2, \dots, t_i)$  is a **well formed formula** (wff)
2. If  $W$  is a wff, and  $x$  is a variable, then  
 $\forall x W$  and  $\exists x W$  are wffs
3. If  $W_1$  and  $W_2$  are wffs, then  
 $\neg W_1$ ,  $W_1 \wedge W_2$ ,  $W_1 \vee W_2$ ,  $W_1 \rightarrow W_2$  and  $W_1 \leftrightarrow W_2$  are wffs
4. If  $W$  is a wff, then  $(W)$  is a wff
5. Nothing else is a wff

## Exercise: What is wrong with these attempted statements?

*sameGroup(England  $\wedge$  Spain)*

is **NOT** a well-formed formula – we cannot have connectives inside the brackets of a predicate.

*beats(Brazil,  $\exists x$ )*

is **NOT** a well-formed formula – we cannot have quantifiers inside the brackets of a predicate.

*$\forall$ European(x)*

is **NOT** a well-formed formula – each quantifier must act directly on a variable.

*$\forall P P(Ireland, x)$*

is **NOT** a well-formed formula – we cannot quantify a predicate (at least, not in CS1113, where we only look at "First Order" Predicate Logic)

## Restricting the domain

Sometimes, we will want to make statements about a subset of the universal set.

Suppose  $U$  is the set of integers  $Z$  ( $= \{\dots, -2, -1, 0, 1, 2, \dots\}$ ).  
How do we say for all  $x$  bigger than 1,  $x^2 > x$ ?

*For all integers  $x$ , if  $x$  is bigger than 1 then  $x^2$  is bigger than  $x$*

$$\forall x ((x > 1) \rightarrow (x^2 > x))$$

To make this easier, we can change the notation, and write

$$\forall x > 1 x^2 > x$$

*(but formally, this should be written as we did above)*

## Translating from English: Example

"All students in UCC have a UCC ID card"

Let the domain,  $U$ , be the set of all people

Let  $uccStudent(x)$  mean  $x$  is a UCC student

Let  $uccCard(x)$  mean  $x$  has a UCC ID card

$$\forall x (uccStudent(x) \rightarrow uccCard(x))$$

Alternatively if  $S$  is the set of all UCC students, we could say

$$\forall x \in S \ uccCard(x)$$

and if  $U = S$

$$\forall x \ uccCard(x)$$

## More Examples

Write predicate logic statements for the following.

- state the universe, and any subsets of interest
  - define the predicates
  - write a logic statement for each example

- Some students in UCC were born in County Kerry
- Some students take both CS1105 and MG1002
- All students registered for CS1105 have access to the CS1105 restricted web pages
- It is not true that all students are registered for both CS1105 and MG1002
- Some students are either not taking CS1105 or not taking MG1002
- All students in 1<sup>st</sup> year CS have a CS email address and a UCC email address
- All students connecting from a UCC IP address or who provide the correct password can access the CS1105 restricted web pages

Exercise

# Next lecture

Examples



# CS1113

## Multiple Quantifiers in Predicate Logic

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# More than one quantifier

Understanding exactly what a quantifier is quantifying

Using multiple quantifiers

*“somewhere in the world, a pedestrian is knocked down by a car every minute”*

So far, all our example predicate logic statements have had a single quantified variable

e.g.  $\forall x (uccStudent(x) \rightarrow uccCard(x))$

But the language definition doesn't restrict us:

2. If  $W$  is a wff, and  $x$  is a variable, then  
 $\forall x W$  and  $\exists x W$  are wfss

We can keep adding new quantifiers and variables onto the front, and we still get a well-formed statement.

What would it mean? How would we define which part of the sentence each quantifier applies to?

# Quantifier Scope

In an expression in predicate logic, the scope of a quantifier is the sub-part of the expression to which it applies – either the predicate immediately following it, or a wff in brackets.

E.g. in  $\forall x (P(x) \rightarrow \exists y Q(y))$

the scope of  $\forall x$  is  $(P(x) \rightarrow \exists y Q(y))$   
the scope of  $\exists y$  is  $Q(y)$ .

In  $\forall x P(x) \rightarrow Q(y)$

the scope of  $\forall x$  is  $P(x)$

In quantified expressions, we can re-use the variable in the quantifiers if the scope of the quantifiers do not overlap.

e.g.  $\forall x P(x) \wedge \exists y Q(y)$

and  $\forall x P(x) \wedge \exists x Q(x)$  are logically equivalent.

## Bindings and free variables

In a wff, a variable  $x$  is **bound** if it is inside the scope of either  $\forall x$  or  $\exists x$  (so it is quantified).

So in the wffs  $\forall x P(x)$ ,  $\exists x P(x)$ ,  $\forall x Q(x,y)$  and  $\exists x Q(x,y)$ ,  $x$  is a bound variable.

In a wff, a variable  $x$  is **free** if it is not bound (it is not quantified).

So in the wffs  $P(x)$  and  $Q(x,y)$ ,  $x$  is a free variable.

Note that in  $\forall x P(y)$ ,  $\exists x P(y)$ ,  $\forall x Q(x,y)$  and  $\exists x Q(x,y)$ ,  $y$  is a free variable – it is not in the scope of  $\forall y$  or  $\exists y$

## More formal definitions of $\forall$ and $\exists$

Let  $W$  be some wff, and let  $x$  be a variable.

$\forall x W$  is true if and only if for each constant  $a \in U$ , when we replace all free occurrences of  $x$  in  $W$  by  $a$ , we get a true statement.

Note: **free in  $W$**   
(not free in  $\forall x W$ )

- if we can find one or more constants  $b \in U$  where this doesn't apply, then  $\forall x W$  is false

$\exists x W$  is true if and only if there is at least one constant  $a \in U$ , such that when we replace all free occurrences of  $x$  in  $W$  by  $a$ , we get a true statement.

- if there is no constant  $a \in U$  for which this applies, then  $\exists x W$  is false

## Consistent assignments in quantifier scope

Let *formula* be a wff in predicate logic in which a free variable  $x$  appears more than once. Then in  $\exists x$  (*formula*) when we interpret it by assigning values to  $x$ , we must assign the same value to each occurrence of  $x$  (and similarly for  $\forall x$  (*formula*))

For example, in  $\exists x$  ( $P(x) \wedge Q(x)$ ), we want to find at least one value  $v$  such that  $(P(v) \wedge Q(v))$  is true.

In  $\forall x$  ( $P(x) \rightarrow Q(x)$ ), if that is true, we must show that  $(P(v_1) \rightarrow Q(v_1))$  is true, and  $(P(v_2) \rightarrow Q(v_2))$  is true, and so on,

but we do not care about e.g.  $(P(v_1) \rightarrow Q(v_2))$ , because it has different values assigned to  $x$ .

## Logically equivalent formulae

We saw the idea of logically equivalent formulae in propositional logic:

"Two formulas in propositional logic are **equivalent** if and only if they have the same truth functions. I.e. no matter what values we assign to the atomic propositions, the two formulas have the same truth value."

We can extend this to handle statements in predicate logic:

Two formulae are **logically equivalent** if and only if they have the same truth value regardless of what interpretation we give to the predicate symbols, or what domains we specify for the variables.

## Example equivalences

$$\exists x (P(x) \vee Q(x)) \equiv (\exists x P(x)) \vee (\exists x Q(x))$$

$$\forall x (P(x) \wedge Q(x)) \equiv (\forall x P(x)) \wedge (\forall x Q(x))$$

We will demonstrate that the first one is true. To do this, we will show

- (i) whenever the left hand side is true, the right hand side is also true, and
- (ii) whenever the right hand side is true, the left hand side is also true.

We will do this without giving any interpretation for P or Q, and without specifying a domain for the variables.

## Example Proof

(i) Suppose  $\exists x (P(x) \vee Q(x))$  is true.

Then there is some value  $v$  in the domain so that  $P(v) \vee Q(v)$  is true.

Therefore, from propositional logic truth tables, we know that either  $P(v)$  is true, or  $Q(v)$  is true (or both).

But then either  $\exists x P(x)$  is true or  $\exists x Q(x)$  is true.

Therefore, from truth tables,  $(\exists x P(x)) \vee (\exists x Q(x))$  is true.

(ii) Suppose  $(\exists x P(x)) \vee (\exists x Q(x))$  is true.

Therefore, either  $\exists x P(x)$  is true or  $\exists x Q(x)$  is true.

So there is some  $v$  such that  $P(v)$  is true or there is some  $v$  such that  $Q(v)$  is true.

So there is some  $v$  such that  $P(v) \vee Q(v)$  is true.

Therefore,  $\exists x (P(x) \vee Q(x))$  is true.

$U$  = all students

$\text{younger}(x,y)$ : student  $x$  is younger than (or same age as) student  $y$

So what does

$\forall x \exists y \text{ younger}(x,y)$

mean?

And what about

$\exists x \forall y \text{ younger}(x,y)$

$\forall x W$  is true if and only if for each constant  $a \in U$ , when we replace all free occurrences of  $x$  in  $W$  by  $a$ , we get a true statement.

$\exists x W$  is true if and only if there is at least one constant  $a \in U$ , such that when we replace all free occurrences of  $x$  in  $W$  by  $a$ , we get a true statement.

## Nested Quantifiers

$\forall x \forall y P(x,y)$  says no matter which constant  $a$  we choose from  $U$ ,  $\forall y P(a,y)$  is true. That means no matter which  $b$  we choose from  $U$ ,  $P(a,b)$  is true.

- So no matter what pair of constants  $(a,b)$  we choose from  $U$ ,  $P(a,b)$  is true (and  $a$  and  $b$  might be same constant)

$\exists x \exists y P(x,y)$  says there exists at least one constant  $a \in U$  such that  $\exists y P(a,y)$  is true. That means there exists at least one constant  $b \in U$  such that  $P(a,b)$  is true.

- So there is at least one pair of constants  $(a,b)$  in  $U$  such that  $P(a,b)$  is true (and  $a$  and  $b$  might be the same constant)

## Nested Quantifiers

$\forall x \exists y P(x,y)$  says no matter which constant  $a$  we choose from  $U$ ,  $\exists y P(a,y)$  is true. That means there exists at least one constant  $b \in U$  such that  $P(a,b)$  is true.

- So no matter what constant  $a$  we choose from  $U$ , we must then be able to find a constant  $b \in U$  such  $P(a,b)$  is true (and  $b$  might be different for each different  $a$ ).

$\exists x \forall y P(x,y)$  says there is at least one constant  $a \in U$  such that  $\forall y P(a,y)$  is true. That means no matter which constant  $b \in U$  we choose,  $P(a,b)$  is true.

- So there is at least one constant  $a \in U$ , such that no matter what constant  $b \in U$  we choose,  $P(a,b)$  is true (so  $a$  is the same for each different  $a$ ).

## Nested Quantifiers: example

Let the domain be  $\mathbb{Z}$  (i.e. the integers =  $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ )

$\forall x \forall y x < y$  is false, since we could set  $x=3, y=2$

$\forall x \exists y x < y$  is true – for any value of  $x$  that you give me, I can find a value of  $y$  such that  $x < y$

$\exists x \forall y x < y$  is false – the statement says there is an integer  $x$  smaller than every other integer

$\exists x \exists y x < y$  is true, since we could set  $x = 2, y = 10$

## Quantifier Order vs Variable order in the predicate

The order of the quantified variables does **not have to match** the order that the variables appear in the predicate or wff that is being quantified.

It is OK to write  $\exists y \forall x P(x,y)$

(it means there is some constant  $a$  such that  $\forall x P(x,a)$  is true)

Often, you will have to write it in this order, to be able to say what you mean to say ...

# Example Specifications

*"Every current student in the UCC database must be registered for some degree program"*

S = current students, D = degree programs, M = modules, U = SUDUM

*registered(x,y): x is registered for y*

$$\forall x \in S \ \exists y \in D \text{ registered}(x,y)$$

*"There is a module that is studied by every CS student"*

*enrolled(x,y): x is enrolled on y*

$$\exists z \in M \ \forall x \in S (\text{registered}(x,CS) \rightarrow \text{enrolled}(x,z))$$

# The order of quantifiers is important!

$\forall x \exists y P(x,y)$

To check this is true, we try the first value for  $x$ , and try to find a value for  $y$  that makes  $P(x,y)$  true; we then select the next value for  $x$ , and try to find a value for  $y$ ; and repeat for each value of  $x$ . Note that the value of  $y$  may be different each time.

$\exists y \forall x P(x,y)$

To check this is true, we try the first value for  $y$ , and check that each value of  $x$  makes  $P(x,y)$  true; if that fails, we try the next value for  $y$ , and check that each value of  $x$  makes  $P(x,y)$  true; and we keep going until we find a value of  $y$  that works. So we are looking for a single value for  $y$  which works for every value of  $x$ .

## Example

Translate the following into logic:

*"a pedestrian is knocked down by a car every minute"*

$P$  = pedestrians,  $M$  = minutes,  $U = P \cup M$

$\text{knockedDown}(x,m)$ :  $x$  is knocked down by a car in minute  $m$ .

Is the answer (a) or (b)?

(a)  $\exists x \in P \ \forall m \in M \ \text{knockedDown}(x,m)$

(b)  $\forall m \in M \ \exists x \in P \ \text{knockedDown}(x,m)$

# Examples

Translate into English:

$$\exists x \forall y (\text{notequal}(x,y) \rightarrow \neg \text{sameTown}(x,y))$$

*notequal(x,y)*:  $x \neq y$

*sameTown(x,y)*:  $x$  and  $y$  live in the same town

$U$  is the set of all students in CS1113.

Translate into logic:

"every student knows the mobile phone number of some other student"

and assume the domain is the set of all students.

# Next Lecture

interpreting statements with multiple quantifiers

formal arguments with quantifiers



# CS1113

# Quantified Logic and Arguments

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Quantified Logic and Arguments

Interpreting statements with multiple quantifiers

The full language of quantified logic

Formal arguments with quantifiers

# Using functions in predicate logic

Sometimes, we will want to make a statement about a function of an object rather than the object itself.

Suppose in the UCC database, we define a function *mentor*, which for each student designates a specific individual as a mentor.

$\text{mentor} : \text{Students} \rightarrow \text{People}$

Let  $\text{professor}(x)$  be the statement that  $x$  is professor, where  $x$  is a person in the UCC database.

*predicate*

We can now make a claim about the database, there is at least one student who has a professor for a mentor:

$\exists x \text{ professor}(\text{mentor}(x))$

# Terms in Predicate Logic Sentences

$U$  is the universal set (or **domain of discourse**) of **constants**

$V$  is a set of variables, such that  $V \cap U = \emptyset$

$\Phi$  is a set of **function** symbols  $\Phi \cap V = \Phi \cap U = \emptyset$

1. Every symbol representing a specific element of  $U$  is a **term**
2. Every variable in  $V$  is a term
3. If  $f$  is a function symbol ( $f \in \Phi$ ), and  $t_1, t_2, \dots, t_n$  are terms, then  $f(t_1, t_2, \dots, t_n)$  is a term
4. Nothing else is a term

# The language of predicate logic (version 2)

$\Pi$  is a set of predicate symbols such that

$$\Pi \cap V = \Pi \cap U = \Pi \cap \Phi = \emptyset$$

1. if  $P$  is a predicate symbol and  $t_1, t_2, \dots, t_i$  are terms  
then  $P(t_1, t_2, \dots, t_i)$  is a **well formed formula** (wff)
2. If  $W$  is a wff, and  $x$  is a variable, then  
 $\forall x W$  and  $\exists x W$  are wffs
3. If  $W_1$  and  $W_2$  are wffs, then  
 $\neg W_1$ ,  $W_1 \wedge W_2$ ,  $W_1 \vee W_2$ ,  $W_1 \rightarrow W_2$  and  $W_1 \leftrightarrow W_2$  are wffs
4. If  $W$  is a wff, then  $(W)$  is a wff
5. Nothing else is a wff

## Use of function in our examples

If  $P(x)$  and  $Q(y)$  are predicates, and  $f(x)$  is a function, then

$\exists x P(f(x))$  **is** a well formed statement in logic  
(and says there is at least one value in the domain such that when you apply function  $f$  to it, and then check predicate  $P$  on the output, it is true)

$\exists x P(Q(x))$  is **not** a well formed statement, since you cannot have a predicate symbol as an argument for a predicate.

This is really confusing, especially if you use capital letters for functions, or lower-case for predicate. We will try not to use it in examples, but if we do, you must make it clear what names are functions.

But you will see examples of this in textbooks and in later years ...

# Arguments in Predicate Logic

We need to create valid arguments in the same way that we did for propositional logic – e.g. to prove that a solution does actually meet a specification, to prove that a network or database is in a legal state, or to persuade someone that they should believe in a given conclusion.

We will use the same rules of inference as before, but we will add six new ones to handle predicates.

# Rules of Inference

$\forall x P(x)$

$P(a)$  for any value  $a$   
in the domain

$P(a)$  for some value  $a$

in the domain

$\exists x P(x)$

$P(c)$  for an arbitrary  $c$   
in the domain

$\forall x P(x)$

$\exists x P(x)$

$P(c)$  for some value  $c$   
in the domain

$\forall x (P(x) \rightarrow Q(x))$

$P(a)$  for some value  $a$   
in the domain

$Q(a)$

$\forall x (P(x) \rightarrow Q(x))$

$\neg Q(a)$  for some value  $a$   
in the domain

$\neg P(a)$

## Example Arguments

All mice like cheese. Mickey is a mouse. Therefore Mickey likes cheese.

Let  $\text{mouse}(x)$  state that  $x$  is a mouse.

Let  $\text{likesCheese}(x)$  state that  $x$  likes cheese.

1.  $\forall x (\text{mouse}(x) \rightarrow \text{likesCheese}(x))$  initial statement
2.  $\text{mouse}(\text{Mickey})$  initial statement
3.  $\text{likesCheese}(\text{Mickey})$

# Example Arguments

Are the following valid arguments?

All students in 2<sup>nd</sup> year CS at UCC passed CS1107. Enda Kenny is a student in 2<sup>nd</sup> year CS at UCC. Therefore Enda Kenny passed CS1107.

All students in 2<sup>nd</sup> year CS at UCC passed CS1107. Enda Kenny passed CS1107. Therefore Enda Kenny is a student in 2<sup>nd</sup> year CS at UCC.

# Next Lecture

Simple Algorithms



# CS1113

## Simple Algorithms

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Simple Algorithms

a review of algorithms  
writing down algorithms  
informal analysis

linear search  
binary search

# Algorithms: a reminder

From  
lecture 02:

## The Fundamental Concept of Computing

- Anything a computer does is based on carrying out a sequence of steps.

An algorithm is an ordered, deterministic, executable, terminating set of instructions

- "ordered" does not mean just a simple sequence
  - we can have branches, loops and function calls, as long as we know which instruction to carry out at each step
- we must carry out the instructions in the specified order
  - we can't jump ahead to work out what needs to be done
- the instructions must be clear, unambiguous, executable, and *complete*
  - they should cover all possible cases

# Writing down algorithms

- For it to be useful, an algorithm must be written in some language that is understood by whoever or whatever is going to read it
  - You are learning how to express algorithms in Python, so that they can be executed by computers
- We are interested in something more fundamental: the essence of the algorithm, independent of the computer or architecture or programming language
  - but we still have to write it in some consistent style so that we can understand what the algorithm says and how efficient it would be if it were executed

# Example: finding the minimum value in a sequence

Problem: given a finite sequence of integers, find the smallest integer in the sequence

Input: a sequence  $x[1], x[2], x[3], x[4], \dots, x[n]$  of integers

Output: an integer min

1.  $\text{min} := x[1]$
2. for each value of  $i$  from 2 to  $n$
3.   if  $x[i] < \text{min}$
4.     then  $\text{min} := x[i]$
5. return  $\text{min}$

## Example: executing the algorithm

Suppose our input is the sequence  $<5,8,3,23,4,-2,1,0>$

$n: 8$

$x[1]=5$

Input: a sequence  $x[1], x[2], x[3], x[4], \dots, x[n]$  of integers

$x[2]=8$

Output: an integer min

$x[3]=3$

1.  $\min := x[1]$

$x[4]=23$

2. for each value of  $i$  from 2 to  $n$

$x[5]=4$

3. if  $x[i] < \min$

$x[6]=-2$

4. then  $\min := x[i]$

$x[7]=1$

5. return  $\min$

$x[8]=0$

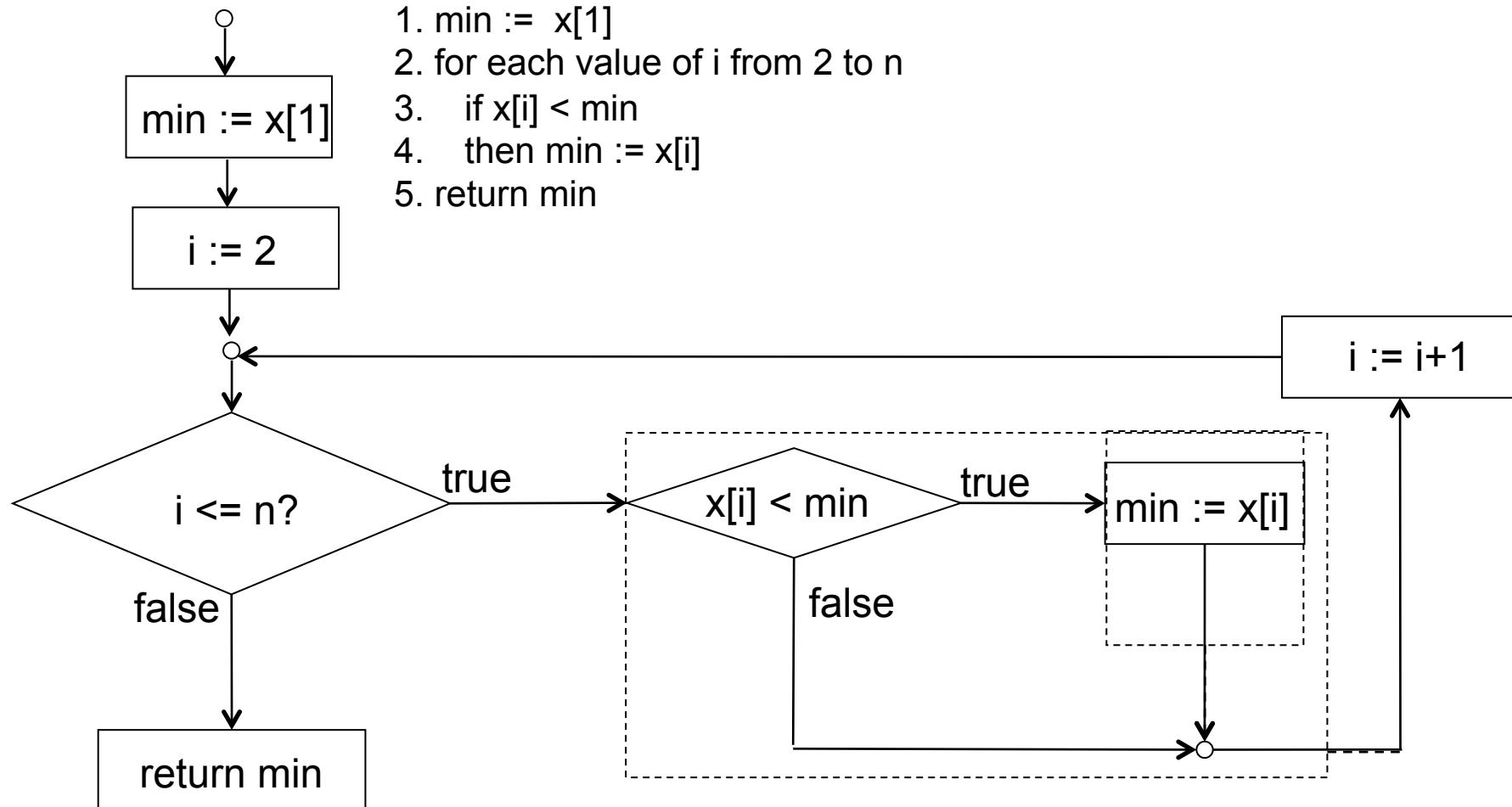
	$i: 2$	$i: 3$	$i: 4$	$i: 5$	$i: 6$	$i: 7$	$i: 8$
$x[1]: 5$	$x[i]: 8$	$x[i]: 3$	$x[i]: 23$	$x[i]: 4$	$x[i]: -2$	$x[i]: 1$	$x[i]: 0$
min: 5		min: 3			min: -2		



# Visualising with a flowchart

Input: a sequence  $x[1], x[2], x[3], x[4], \dots, x[n]$  of integers

Output: an integer min



# Informal language for algorithms

Always state the input and output

Input: a sequence  $x[1], x[2], x[3], x[4], \dots, x[n]$  of integers  
Output: an integer min

use variable names

1.  $\text{min} := x[1]$
2. for each value of  $i \in \{1, 2, \dots, n\}$ 
  - 3. if  $x[i] < \text{min}$
  - 4. then  $\text{min} := x[i]$
5. return  $\text{min}$

for loop – go round the loop this many times

indent three spaces inside  
a branch or loop

use " $:=$ " for assignment – assign the value of  $x[1]$  to min

if-then-else: if test is true, do "then" part; if test is false,  
do "else" part if there is one; then move on to next line

# Example: informal analysis of the algorithm

An algorithm is an ordered, deterministic, executable, terminating set of instructions

- **Correctness** Does it do what we want?
  - at first, min is set to the first element; each time we go round the loop, min is set to the smallest we have seen so far; so at the end, min tells us the smallest element.
- **Termination** Does it terminate?
  - the input is finite, and the *for* loop says increment  $i$  each time we go round the loop until we reach the input length. Inside the loop we don't change  $i$ , and inside each loop we do a finite number of steps, so the *for* loop definitely terminates, and so the algorithm terminates
- **Complexity** How many steps?
  - 1 initial assignment step, and then once round the for loop for each other element of the sequence (so  $n-1$  times round the loop); inside the loop, we do 1 test, and we may do 1 assignment of a value to a variable. So at most, we need  $n$  assignments, and  $n-1$  tests.

## Example: finding the average value in a sequence

Problem: given a finite sequence of integers, find the average value of the sequence

Input: sequence  $x[1], x[2], x[3], \dots, x[n]$  of integers

Output: a decimal number ave

1. sum := 0
2. for each value of i from 1 to n
3.   sum := sum+x[i]
4. ave := sum/n
5. return ave

# Example: informal analysis of the algorithm

- **Correctness** Does it do what we want?
  - at first, sum is set to 0; each time we go round the loop, sum is set to the total of all values we have seen; so at the end of the loop, sum is the total of all values; we then divide by the number of values to get the average.
- **Termination** Does it terminate?
  - the input is finite, and the *for* loop says increment  $i$  each time we go round the loop until we reach the input length. Inside the loop we don't change  $i$ , and inside each loop we do a single addition and a single assignment, so the *for* loop terminates, and so the algorithm terminates
- **Complexity** How many steps?
  - 1 initial assignment step, and then once round the for loop for each element of the sequence (so  $n$  times round the loop); inside the loop, we do 1 addition, and we do 1 assignment of a value to a variable. After the loop, we do one division and 1 assignment. So we need  $n+2$  assignments, and  $n$  additions, and 1 division.

# Search

Often, we will need to find a particular value somewhere in a sequence of values.

- searching for a word in a spell-checker
- searching for a book in a library catalogue
- searching for a name in an address book
- searching for a keyword in a web page
- searching for a DVD on an e-commerce site

If we have to do this type of search often inside a program, then the speed at which we can do it is important.

We will look at some different ways to search, and see how long they take on average.

# Linear Search

The simplest way to search a sequence is simply to consider each element in turn until we find the one we want.

Input: sequence  $x[1], x[2], x[3], \dots, x[n]$  of objects

Input:  $y$ , the object we want to find

Output:  $pos$ , the position of  $y$  in the sequence, or 0 if it isn't there

1.  $pos := 0$
2.  $i := 1$
3.  $\text{while } (pos == 0 \text{ and } i \leq n)$       
  4.     $\text{if } (x[i] == y)$
  5.     $\text{then } pos := i$
  6.     $\text{else } i := i + 1$
7.  $\text{return } pos$

"while" loop – similar to the "for" loop, but we now keep going round as long as the test is true

# Informal analysis of linear search

- does the algorithm do what we want?
  - we consider each object in the sequence in turn, until we find the object we want, exit the loop and report the position, or we reach the end and report 0, so yes.
- does it terminate?
  - the while loop terminates if  $pos \neq 0$  or  $i > n$ .  $i$  starts at the value 1. Inside the loop, each time we either increase the value of  $i$ , or we assign the value of  $i$  to  $pos$ .  $i$  is never 0, so at some stage, we will either assign a non-zero value to  $pos$ , or we will assign a value bigger than  $n$  to  $i$ . So the loop will stop, and so will the algorithm.
- how many steps does it take?
  - at worst, if the object is not in the sequence, we will examine every object, and so will take  $n$  steps.

## Binary search

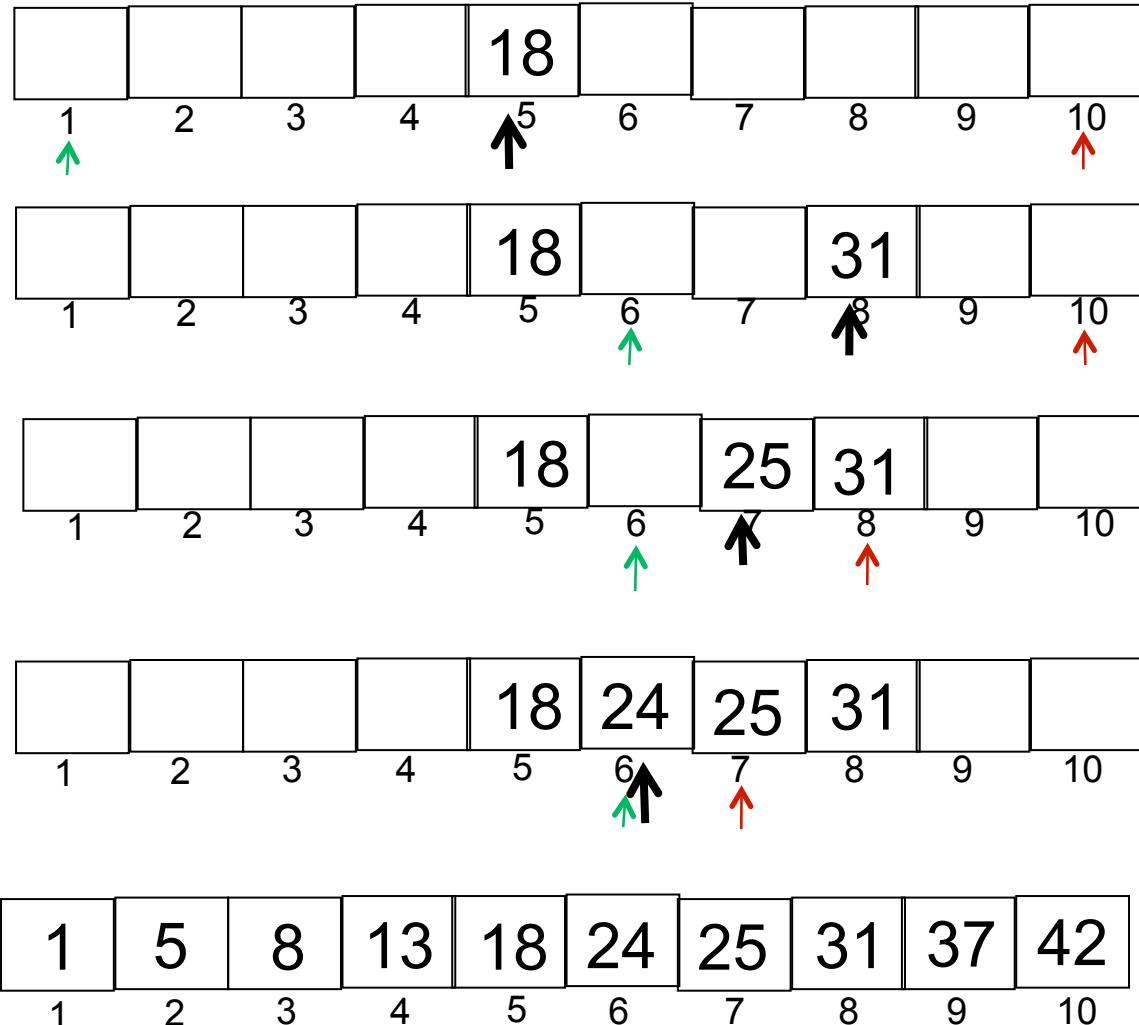
Sometimes, we know that the objects in the input sequence are given in some order. If so, we can do a different search that is better in the worst case.

We start by looking at the middle of the sequence. If that element is ordered before our desired object, then we shrink our view to be from the middle to the end, and keep looking in the same way; otherwise, we shrink our view to be from the start until the middle, and continue in the same way.

Eventually, our view will shrink to a single object. Either it is the one we want, or we know our object is not in the sequence.

# Example binary search

Looking for the number 24 in the following list:



# Binary search

Input: a finite sequence of objects  $x[1], x[2], \dots, x[n]$

Input:  $y$ , the object we are looking for

Output:  $pos$ , position of  $y$  in sequence, 0 if not there

1. earliest := 1
2. latest := n
3. while earliest < latest
4.   i := floor((earliest+latest)/2)
5.   if  $x[i] < y$
6.     then earliest := i+1
7.     else latest := i
8.   if  $x[i] == y$
9.     then pos := i
10.   else pos := 0
11. return pos

# Informal analysis of binary search

- does it terminate?
  - *Earliest* starts less than *latest*. Inside the loop, we set  $i$  to be between them. We then bring *earliest* up to  $i$ , or bring *latest* down to  $i+1$ , so the gap between *earliest* and *latest* is always shrinking. Eventually, they will be the same, and then the loop stops. So the algorithm terminates
- how many steps does it take?
  - at worst, if the object is not in the sequence, each time we halve the list. This means we do  $\log_2 n$  times round the loop.

Next lecture ...

## An Introduction to Graphs



# CS1113

# Graphs in Computer Science

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# An introduction to Graphs

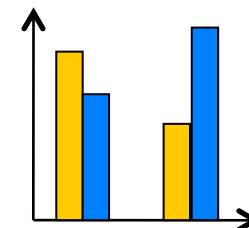
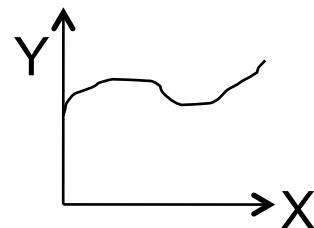
Graphs in computer science

- simple graphs
- multigraphs
- directed graphs

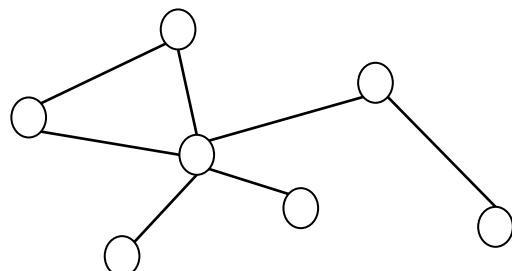
simple graph properties

# What is a graph?

1. a visual representation of the relationship between the values of two or more variables



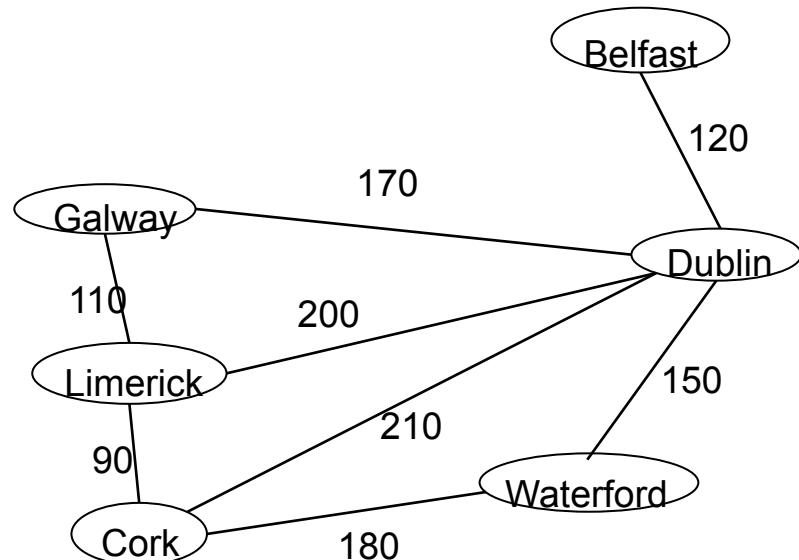
2. a representation of the relationships between multiple entities



the standard  
computer science  
use

# Examples

representing main roads for route planning



augmented with  
travel times

note: numbers attached to links

# Social Networks

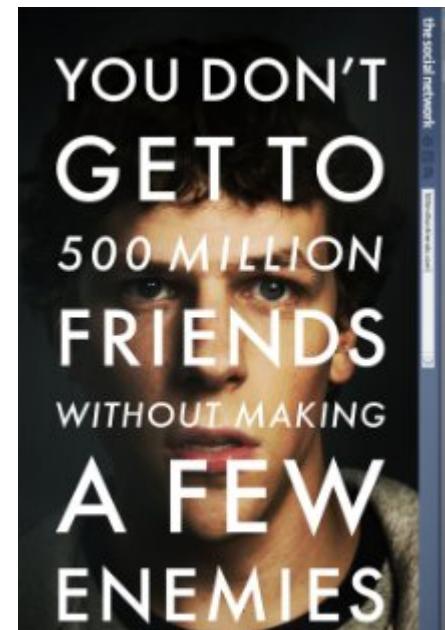
A **social network** is a graph where the objects are people, and the links show a social relationship.

"FOAF (an acronym of Friend of a Friend) is a machine-readable ontology describing persons, their activities and their relations to other people and objects.

FOAF allows groups of people to describe social networks ...

FOAF is an extension to RDF Resource Description Framework and is defined using OWL Web Ontology Language. "

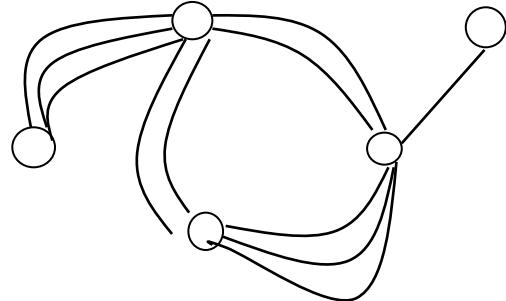
(From wikipedia)





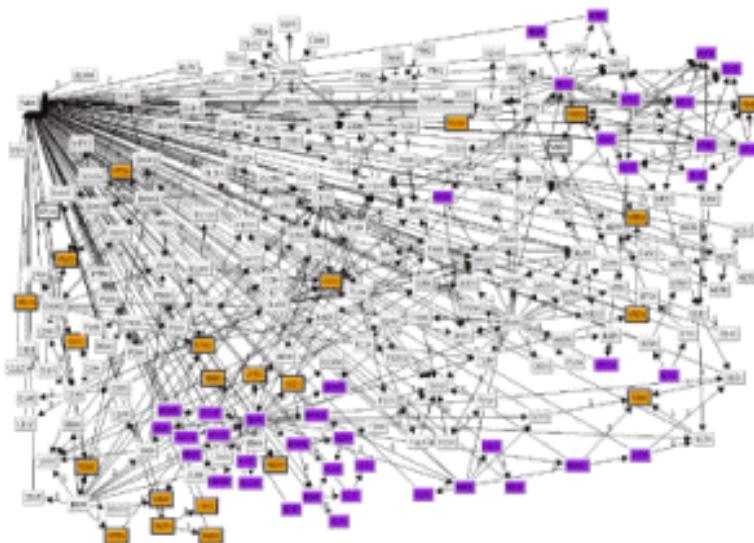
How far apart can two people be? How many links do I  
need to follow to get from one website to another?  
Which websites (or people) are the most connected?

# Call graphs



Each link represents a call between two telephone numbers

Note: multiple links between nodes



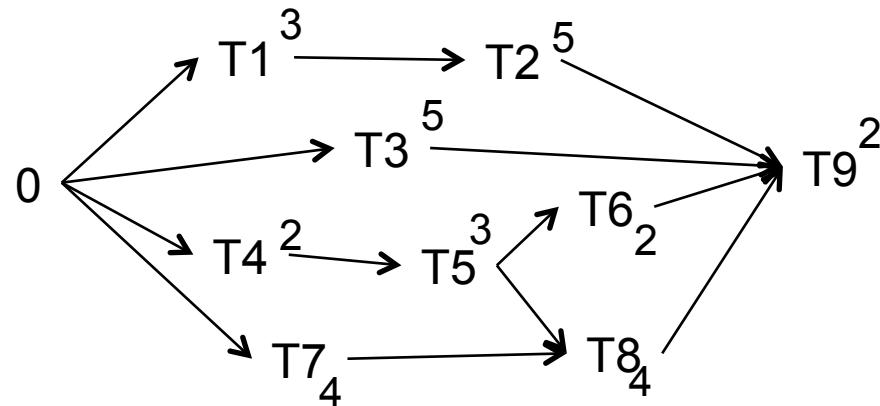
Each link represents a function call between two different modules in a large software system

Recursive functions will require an edge from a node to itself

Image taken from "Release: Reconstruction of Legacy Systems for Evolutionary Change", by Munro Burd and Young, The Centre for Software Maintenance, University of Durham

# Project Planning

A set of tasks to complete. Graph shows precedence order.



Numbers show time  
to complete the task

Note: nodes now  
have a label

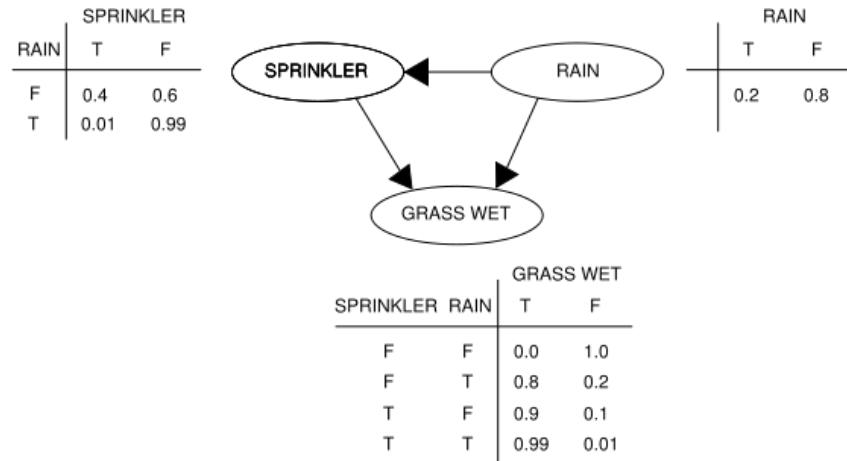
How quickly can I finish all my tasks?  
How many resources would I need?

Note: links  
now have a  
direction

If I have 2 resources, how quickly can I finish?

# Bayesian Networks and Influence Diagrams

representing the presence of an influence between two parameters



Note: complex information now stored with each node

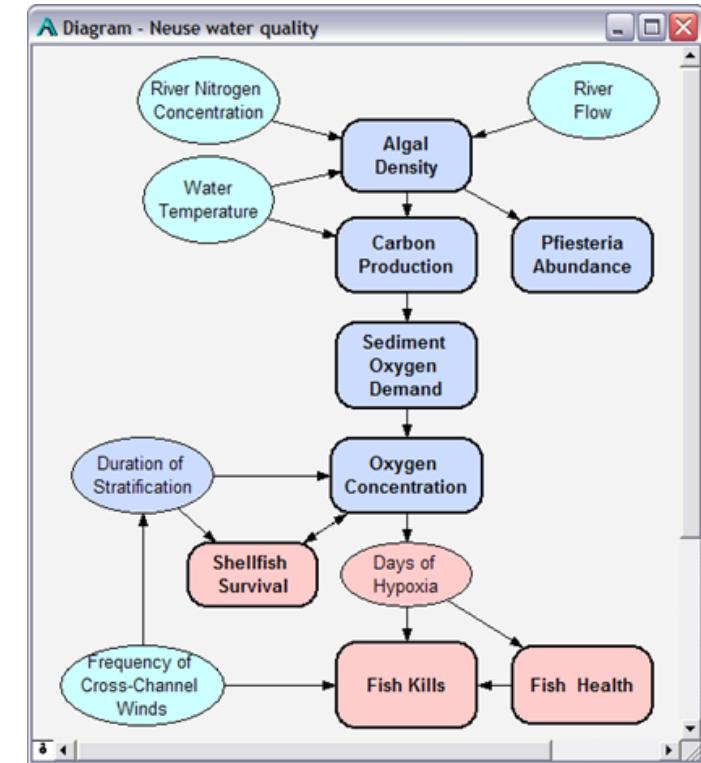
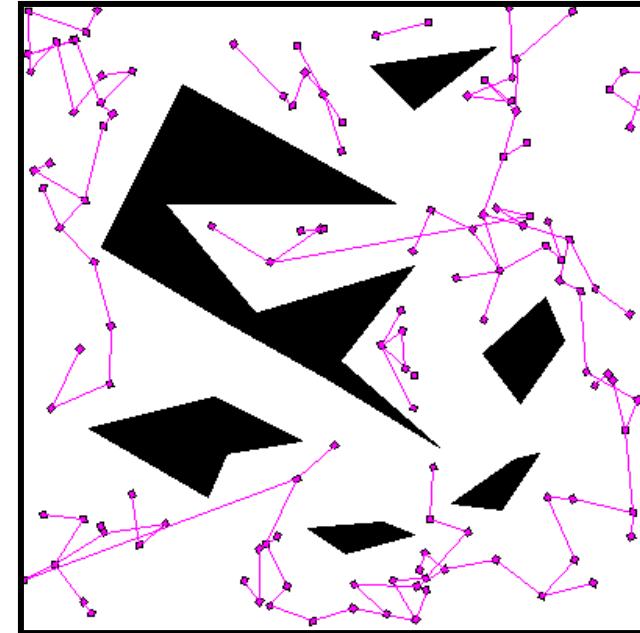


image taken from  
<http://www.lumina.com/casestudies/NeuseEstuary.htm>

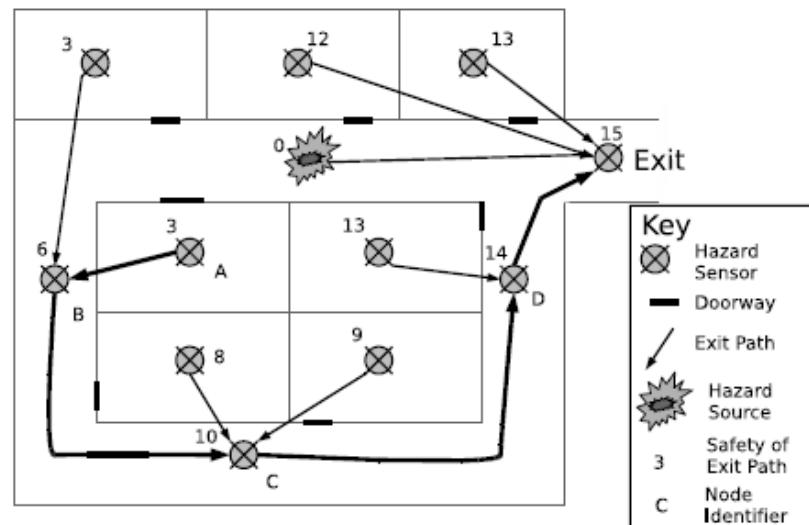
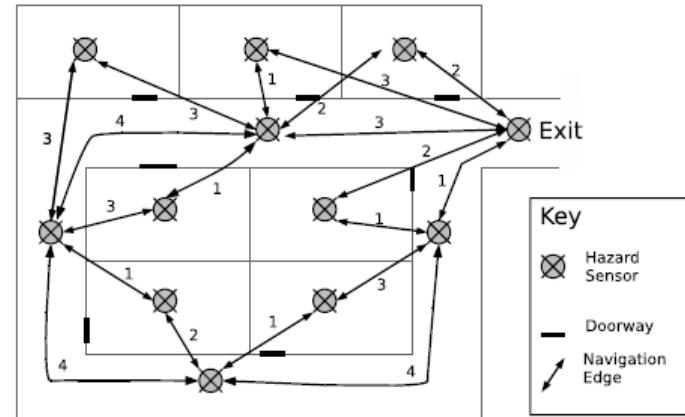
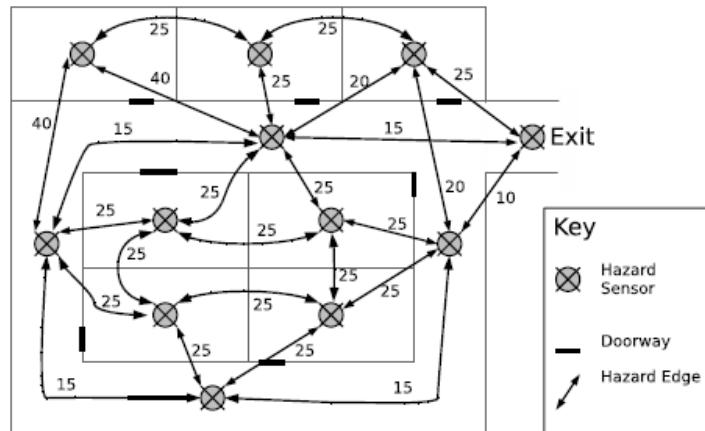
# Path planning in computer games



- intelligent agents must plan fast routes through an environment to achieve tasks or to defeat opponents
- environment is represented internally as a graph (which might be changing)

images taken from "Motion planning in games" talk by Marc Overmars at Intl workshop on Motion Planning in Virtual Environments", 2005  
- see <http://people.cs.uu.nl/markov/>  
- see also thesis on Quake bots [http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren\\_Jean-Paul\\_van/thesis.pdf](http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren_Jean-Paul_van/thesis.pdf)

# Preparing Evacuation Routes



images taken from "Emergency Evacuation using Wireless Sensor Networks", by Matthew Barnes, Hugh Leather and D. K. Arvind

# Graphs in Computer Science

Graphs are everywhere in computer science:

- in the representation of application problems
- in general algorithms and data structures

We will have to work out answers to questions like:

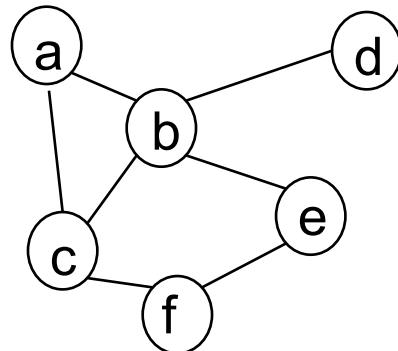
- are all the nodes connected to the rest of the graph?
- what is the shortest path between two particular nodes?
- on average, how many links separate any two nodes?

We need to

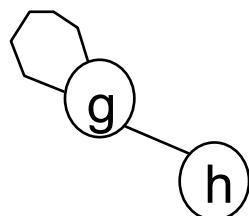
- agree a consistent language for talking about graphs
- work out how to represent them in programs
- understand the main algorithms for computing with graphs

# Simple Graphs

A **simple graph**  $G$  is a pair  $(V, E)$ , where  
 $V$  is a set of **vertices** (representing the objects)  
 $E$  is a set of **edges**, where each edge in  $E$  is a set of 1 or 2 vertices (representing the links between vertices)



$$G = (V, E), \text{ where}$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$



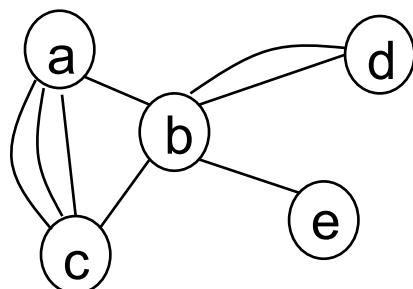
$$G' = (V', E'), \text{ where}$$
$$V' = \{g, h\}$$
$$E' = \{\{g\}, \{g, h\}\}$$

# Multigraphs

A **multigraph**  $G$  is a pair  $(V, E)$ , where

$V$  is a set of **vertices** (representing the objects)

$E$  is a **bag** of **edges**, where each edge in  $E$  is a set of 1 or 2 vertices (representing the links between vertices)



$G = (V, E)$ , where

$V = \{a, b, c, d, e\}$

$E = \{\{a, b\}, \{a, c\}, \{a, c\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, d\}, \{b, e\}\}$

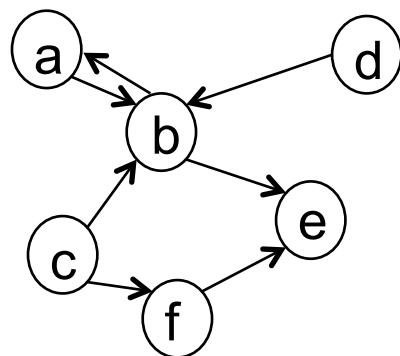
A **bag**, or **multiset**, is a set in which repeated elements are allowed

# Directed Graphs

A **directed graph**  $G$  is a pair  $(V, E)$ , where

$V$  is a set of **vertices** (representing the objects)

$E$  is a set of **edges**, where each edge in  $E$  is an ordered pair of vertices (representing the links between vertices)



$$G = (V, E), \text{ where}$$

$$V = \{a, b, c, d, e, f\}$$

$$E = \{(a, b), (b, a), (b, e), (c, b), (c, f), (d, b), (f, e)\}$$

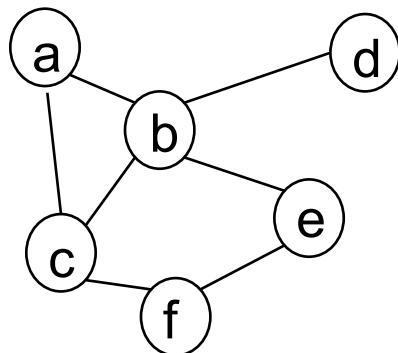
Note: the order in the ordered pair matters.  
 $(a, b)$  and  $(b, a)$  are different edges

## Adjacent vertices

Assume we have a simple graph  $G = (V, E)$ .

Two vertices,  $v_1$  and  $v_2$ , are **adjacent** if there is an edge  $\{v_1, v_2\}$  in  $E$ .

If edge  $x = \{v_1, v_2\}$ , then  $x$  is **incident on**  $v_1$  and  $v_2$ .  $v_1$  and  $v_2$  are the **endpoints** of  $e$ .



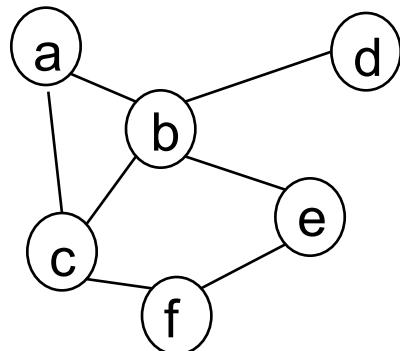
$$G = (V, E), \text{ where}$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

Vertices  $a$  and  $b$  are adjacent.

Vertices  $c$  and  $d$  are not adjacent.

## Vertex degree

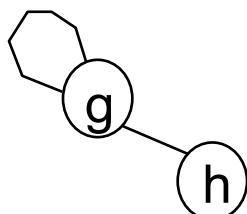
The degree of a vertex,  $v$ , is the number of times edges are incident on  $v$  (so an edge from  $v$  to itself counts twice)



$$G = (V, E), \text{ where}$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

The degree of vertex  $a$  is 2, and degree of vertex  $b$  is 4.

The function  $\text{deg}: V \rightarrow \mathbb{N}$  returns the degree of any vertex.



$$G' = (V', E'), \text{ where}$$
$$V' = \{g, h\}$$
$$E' = \{\{g\}, \{g, h\}\}$$

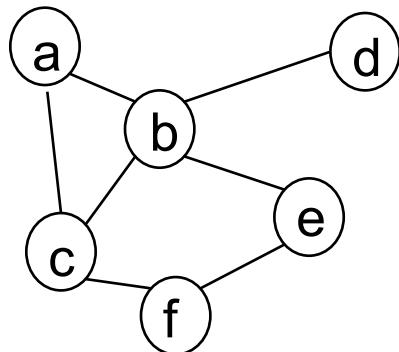
The degree of vertex  $g$  is 3, and degree of vertex  $h$  is 1.

## The sum of the degrees of a simple graph

The sum of the degrees of a simple graph is equal to twice the number of edges in the graph.

if  $G = (V, E)$ , and  $|E| = k$ , then

$$\sum_{v \in V} \deg(v) = 2k$$



$$G = (V, E), \text{ where}$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

$$\deg(a)=2, \deg(b)=4, \deg(c)=3, \deg(d)=1, \deg(e)=2, \deg(f)=2$$

$$2+4+3+1+2+2=14 = 2*7$$

There are 7 edges

# The Handshaking Lemma

The number of vertices with odd degree is even.

## Proof

The sum of the degrees of all vertices is  $2k$ , which is even.

Partition the vertices into two groups – the vertices with even degree, and the vertices with odd degree.

The sum of the degrees of even-degree vertices must be even, say  $2x$ .

Let the sum of the degrees of odd-degree vertices be  $y$ .

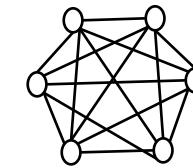
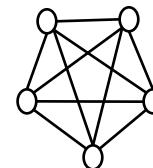
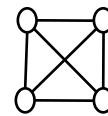
Then  $y+2x=2k$ . So  $y=2k-2x = 2(k-x)$ , which is even.

But all the numbers I added up to get  $y$  were odd numbers. So there must have been an even number of them.

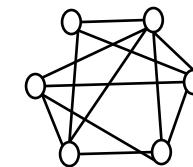
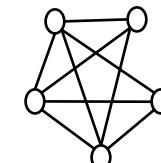
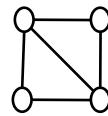
# Complete graphs

A **complete** graph is one where every vertex is adjacent to every other vertex.

Complete graphs:



Not complete graphs:



# Next lecture ...

paths, subgraphs and connectivity



# CS1113 Paths in Graphs

**Lecturer:**

Professor Barry O'Sullivan

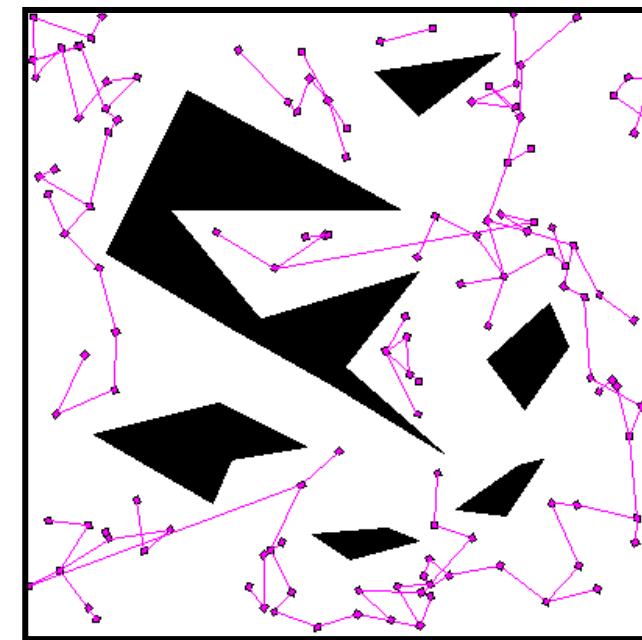
Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Paths

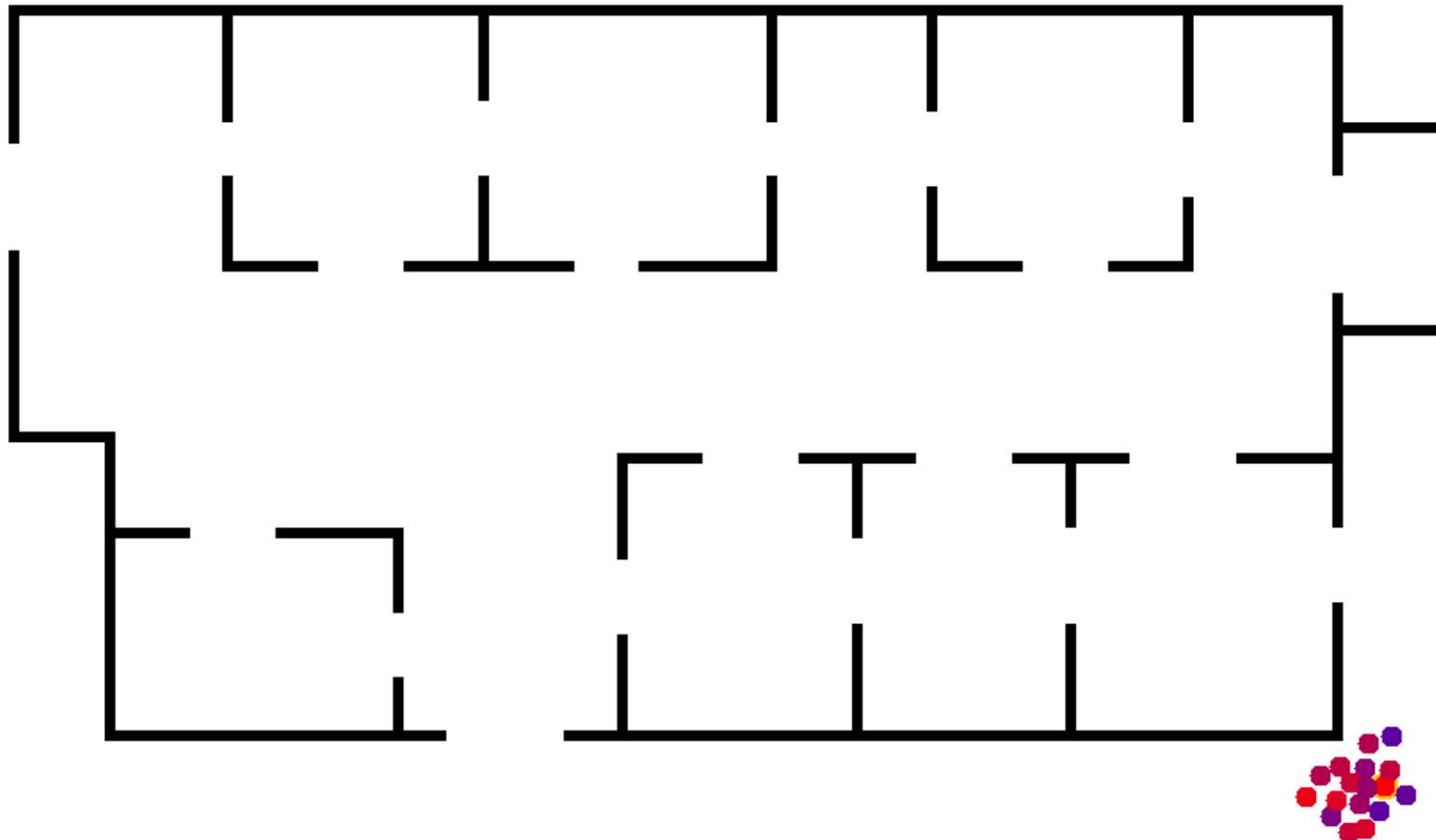
paths  
connected graphs  
shortest paths

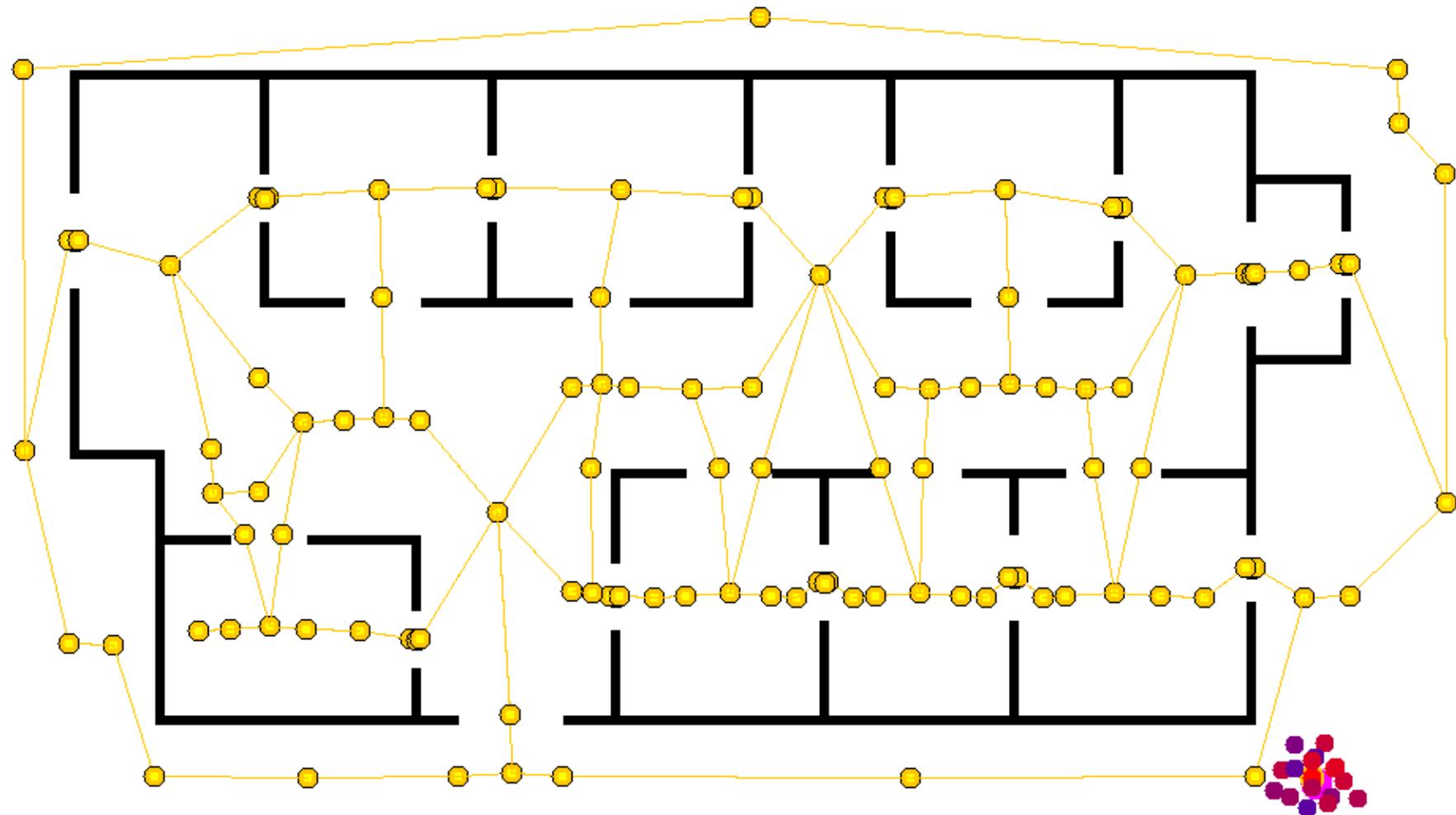


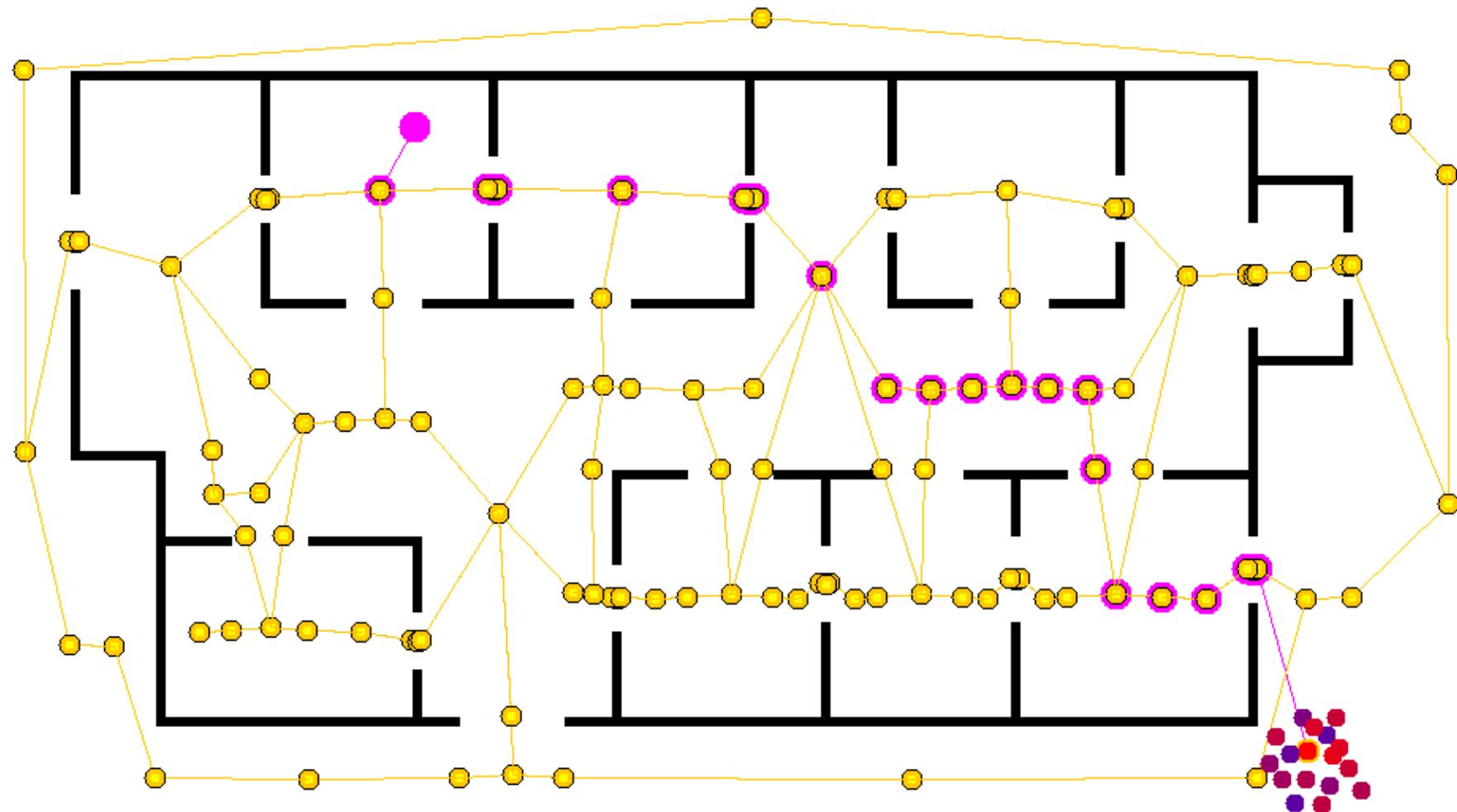
# Reasoning with graphs

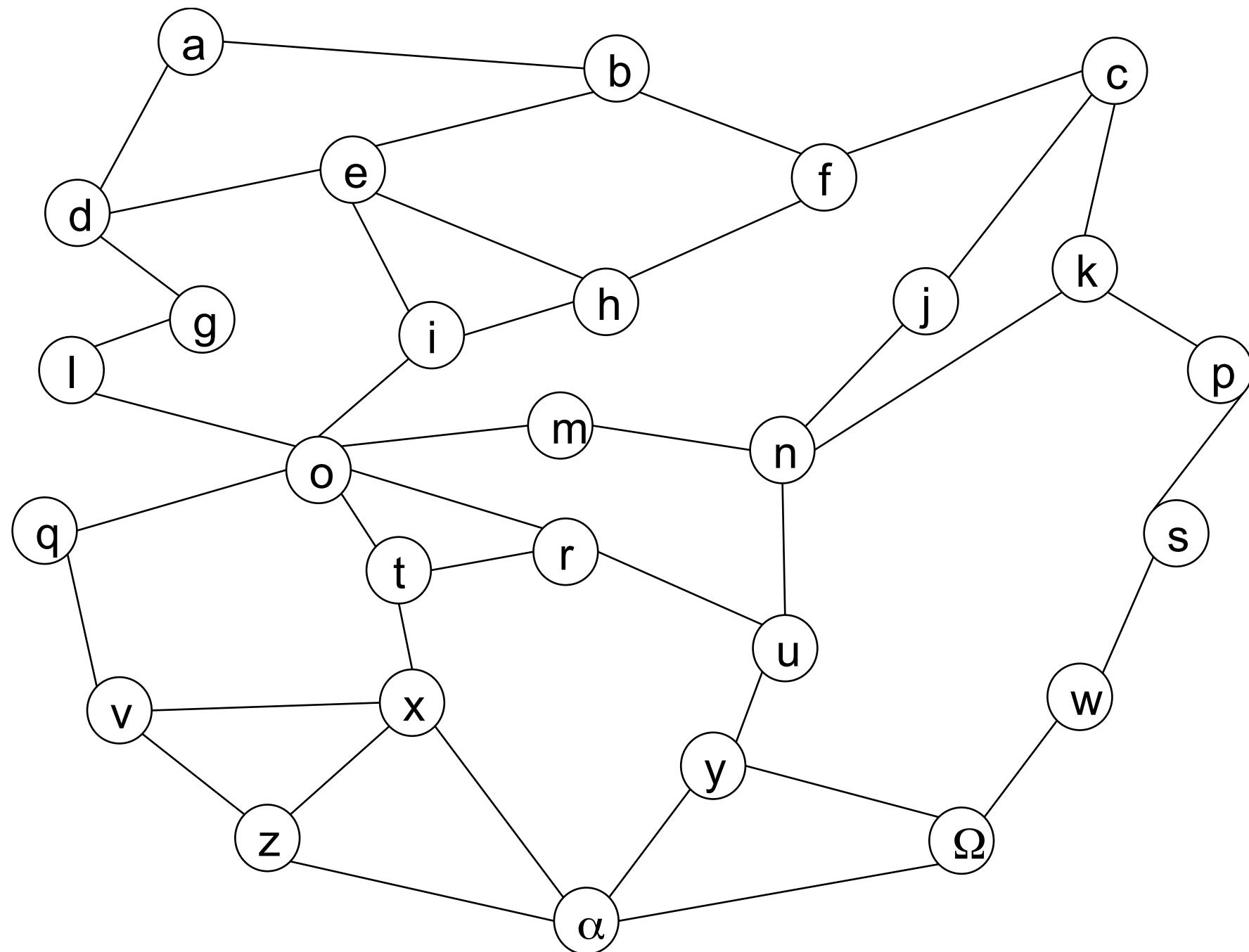
- in telecommunications networks, we will want to identify a route along which we can send data, packets or voice traffic to connect two users
- in route planning, we may want to compute the shortest route from one location to another, by moving across multiple connecting edges
- in social networks, we might want to ask how distant two individuals are (i.e. how many association links are needed to connect one person to another)

In each of these cases, we want to find a sequence of edges, and move along each edge in a particular direction, in order to connect two vertices





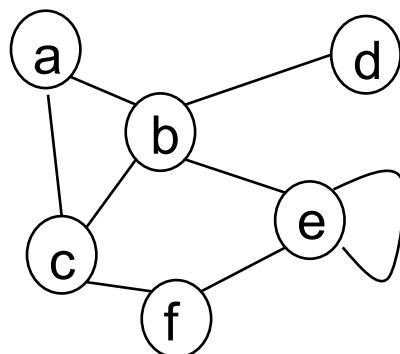




# Graphs (reminder)

A graph is an abstract representation of the relationships between multiple objects.

A simple graph  $G = (V, E)$ , where  $V$  is a set of vertices, and  $E$  is a set of edges, where each edge is a set containing 1 or 2 of the vertices from  $V$ .



$$V = \{a, b, c, d, e, f\}$$
$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, f\}, \{e\}, \{e, f\}\}$$

A simple graph is a symmetric relation on  $V$

A simple graph has at most one edge between any pair of vertices.  
An edge can be from a vertex to itself, when it is called a **loop**.

$G = (V, E)$ , where

$V = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, \alpha, \Omega\}$

$E = \{ \{a, b\}, \{a, d\}, \{b, e\}, \{b, f\}, \{c, f\}, \{c, k\}, \{d, e\}, \{d, g\}, \{e, h\}, \{e, i\}, \{f, h\}, \{g, l\}, \{h, i\}, \{i, o\}, \{j, n\}, \{k, n\}, \{k, p\}, \{l, o\}, \{m, n\}, \{m, o\}, \{n, u\}, \{o, q\}, \{o, r\}, \{o, t\}, \{p, s\}, \{q, v\}, \{r, t\}, \{r, u\}, \{s, w\}, \{t, x\}, \{u, y\}, \{v, x\}, \{v, z\}, \{w, \Omega\}, \{x, z\}, \{x, \alpha\}, \{y, \alpha\}, \{y, \Omega\}, \{z, \alpha\}, \{\alpha, \Omega\} \}$

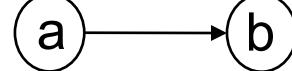
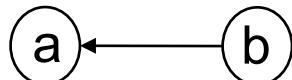
From last lecture:

Two vertices,  $v_1$  and  $v_2$ , are **adjacent** if there is an edge  $\{v_1, v_2\}$  in  $E$ .

## Oriented edges

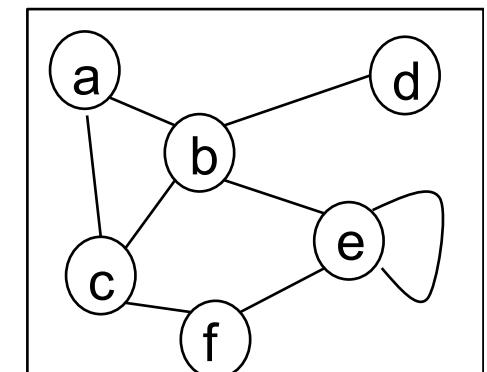
An **orientation** of an edge assigns a direction to that edge, selecting one of the edges as the **startpoint**.

Example: the edge  $\{a,b\}$  has two different orientations:

- $(a,b)$  with  $a$  as the endpoint 
- $(b,a)$  with  $b$  as the endpoint 

Example: the edge  $\{e\}$  has only one orientation:

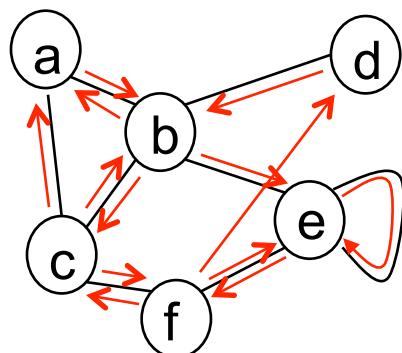
- $(e,e)$  with  $e$  as the endpoint 



# Paths

A **path** in a graph is a sequence of oriented edges, such that the endpoint of one oriented edge is the startpoint of the next oriented edge in the sequence.

Examples:



$\langle(a,b), (b,e),(e,f),(f,c)\rangle$  is a path  
 $\langle(a,b),(b,c),(c,a)\rangle$  is a path  
 $\langle(f,e),(e,e),(e,f),(f,c)\rangle$  is a path

$\langle(d,b),(f,c)\rangle$  is not a path  
 $\langle(c,f),(e,f)\rangle$  is not a path  
 $\langle(d,b),(c,f),(b,c)\rangle$  is not a path  
 $\langle(c,b),(b,a),(b,e)\rangle$  is not a path  
 $\langle(c,f),(f,d)\rangle$  is not a path

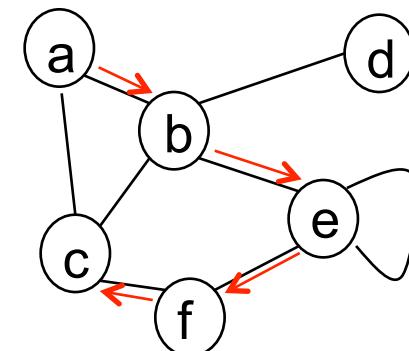
What about (i)  $\langle(b,c),(c,f),(f,e),(e,e),(e,b)\rangle$ ?  
(ii)  $\langle(f,c),(c,b),(a,b),(b,d)\rangle$ ?  
(iii)  $\langle(c,a)\rangle$ ?

For simple graphs, we can describe a path by listing the sequence of vertices it visits, since each pair of vertices is linked by at most one edge.

Example: the path  $\langle(a,b),(b,e),(e,f),(f,c)\rangle$  can be represented by  $\langle a,b,e,f,c\rangle$

For a path represented by a vertex sequence  $\langle v_1, v_2, \dots, v_n \rangle$ ,

- $v_1$  is the **start** of the path
- $v_n$  is the **end** of the path



Example: for the path  $\langle a,b,e,f,c\rangle$ , a is the start, c is the end

The **length** of a path is the number of edges in the sequence (or the number of vertices minus 1)

# Multiple paths in simple graphs

For a given pair of vertices, there may be many different paths which connect them.

Example: to connect  $a$  to  $f$ , we have

$\langle a, b, c, f \rangle$

$\langle a, b, e, f \rangle$

$\langle a, b, e, e, f \rangle$

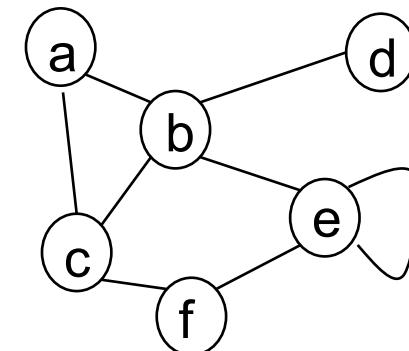
$\langle a, c, b, e, f \rangle$

$\langle a, c, b, e, e, f \rangle$

$\langle a, b, c, a, b, e, f \rangle$

$\langle a, c, f \rangle$

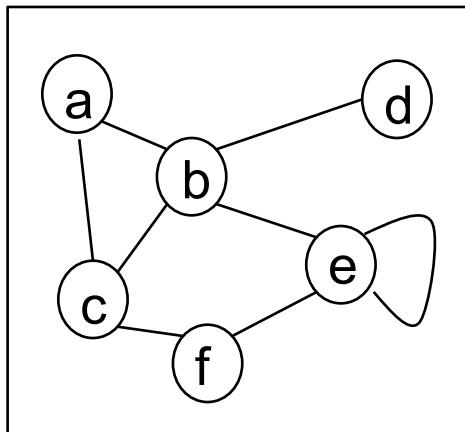
and many more ...



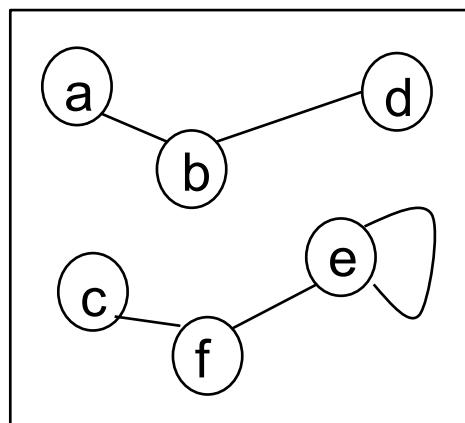
How do we find the shortest path?

## Connected graphs

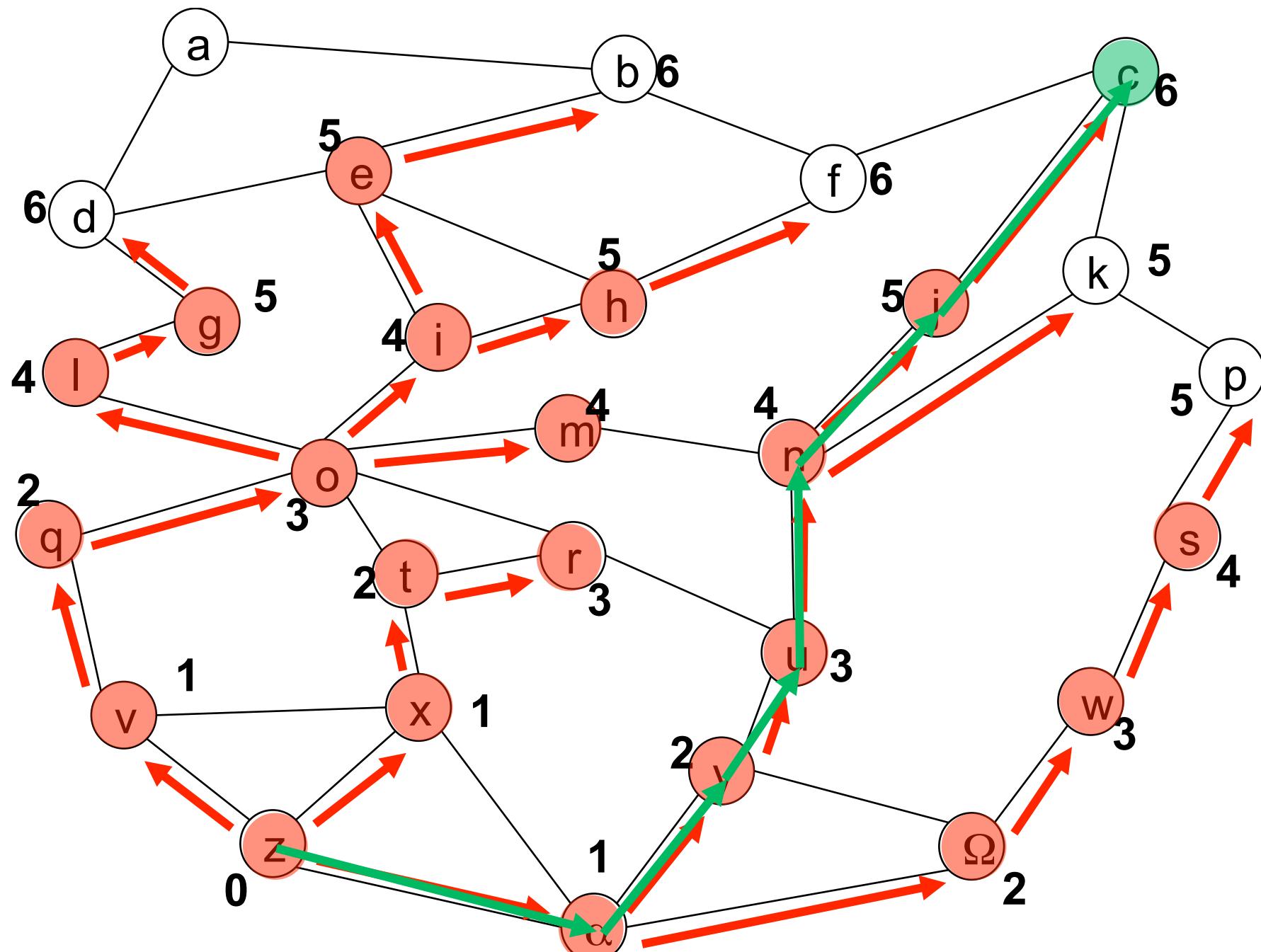
A graph is **connected** if every pair of vertices  $v,w$  can be connected by a path which starts at  $v$  and ends at  $w$ .



is a connected graph



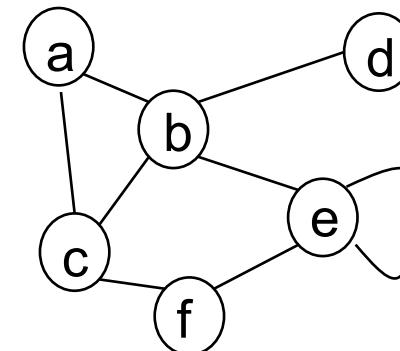
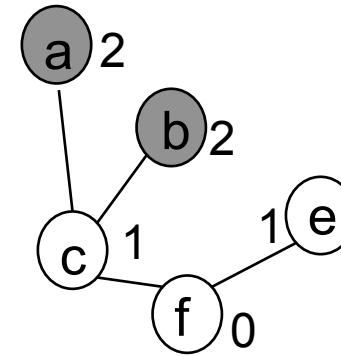
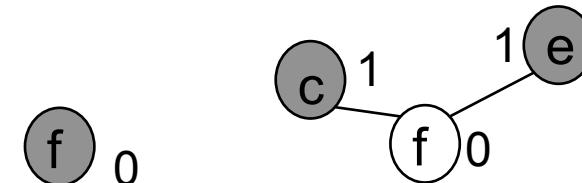
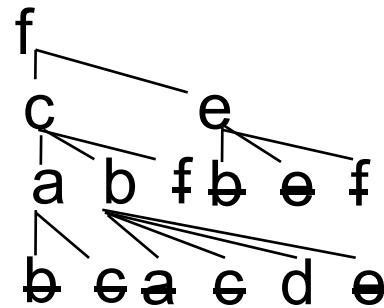
is not a connected graph – there is no path between, for example,  $a$  and  $f$ .



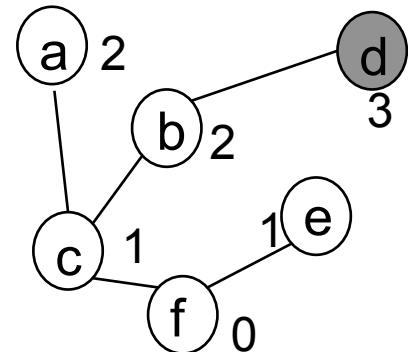
# Finding the shortest path (informal)

We start at the start vertex. We look at all vertices that are adjacent that we haven't seen before. If one of them is the end vertex, we stop; else we mark as being 1 hop away. We then take each 1-hop vertex in turn, and look at all vertices that are adjacent to them (ignoring any we have seen before). If one of them is the end vertex, we stop; else we mark as 2 hops. We then take each 2-hop vertex in turn, ... and so on. Either we find the end vertex via the shortest path, or the start vertex cannot be connected to the end vertex.

Example: find the shortest path from f to d



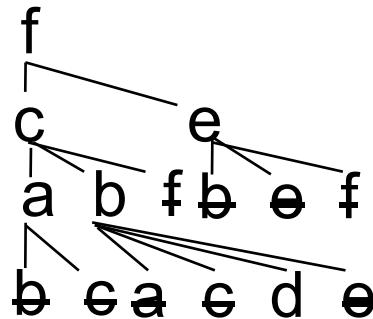
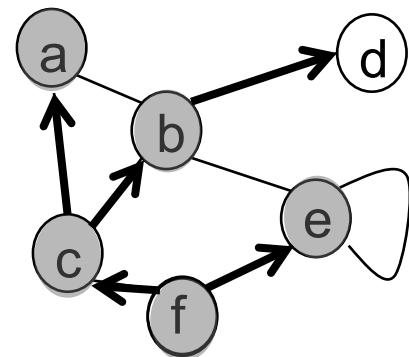
... but  
how do  
we record  
the path?



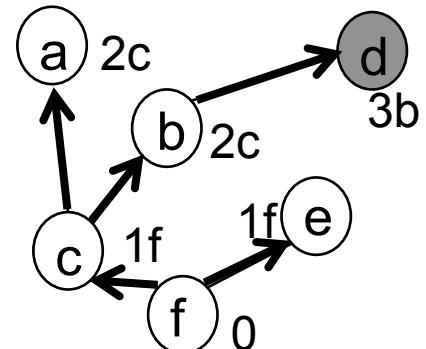
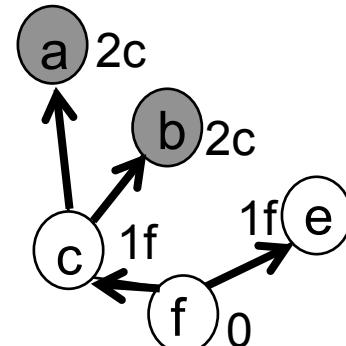
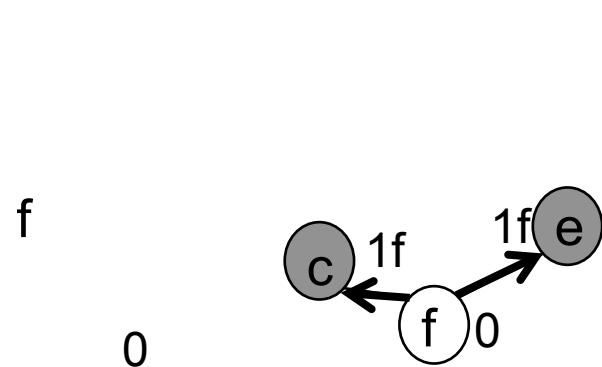
# Recording the shortest path

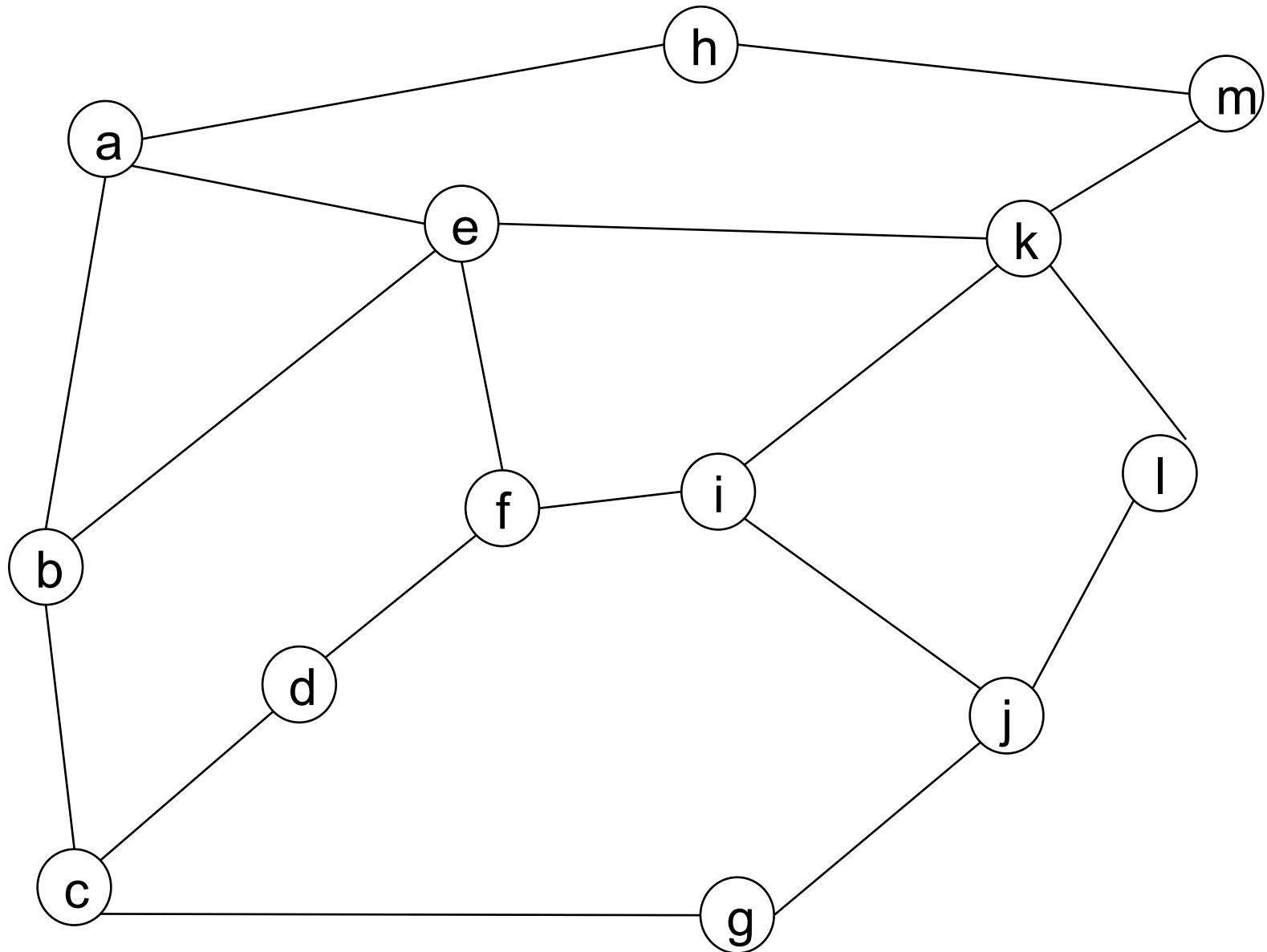
We will maintain a 2D array (i.e. a table), one column for each vertex in  $V$ . The first row says whether the vertex has been expanded; the second says the number of edges to get to it; the third says the vertex that precedes it on the shortest path from the start vertex.

Each time we expand a vertex  $v$  in the search, we look at all its adjacent vertices: if the corresponding cell already has a value, then we have seen it before; else, we assign  $v$  to the cell, to say we reached here from vertex  $v$ .



a	b	c	d	e	f
done?	1	1	1	1	1
edges?	2	2	1	3	1
previous	c	c	f	b	f





# Algorithm for finding shortest path

Algorithm: shortestPath

Input: a simple graph  $G=(V,E)$ , where  $V = \{1, 2, \dots, n\}$

Input: a vertex  $x$  from  $V$ , the start

Input: a vertex  $y$  from  $V$ , the end

Output: an table representation of the path, or null if none

**Exercise:** prove that the shortest path does not visit any vertex twice (you don't need this algorithm)

```
1. L := a table with 3 rows for each of n vertices, all null
2. v := x
3. L[v][2] := 0          //the smallest number of edges to v from x
4. L[v][3] := 0          //the previous vertex in the path to v
5. while v is not null
6.     edges := L[v][2] + 1      //will grow path by one edge
7.     for each vertex j adjacent to v           //expand paths from v
8.         if L[j][2]== null           //found a shortest path to j
9.             then L[j][2] := edges      //update j's edge count
10.            L[j][3] := v           //say how we got here (v -> j)
11.            if j == y, return L    //stop- found shortest path to y
12.            L[v][1] := 1           //mark v as done
13.            v := vertex with smallest L[v][2] and with L[v][1]== null
14.            or null if there isn't one //find shortest path to grow
15.        return null           //we didn't reach the target by any path
```

# Next lecture ...

Paths with travel times

Dijkstra's Algorithms for finding shortest paths



# CS1113

## Shortest Paths in Graphs

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

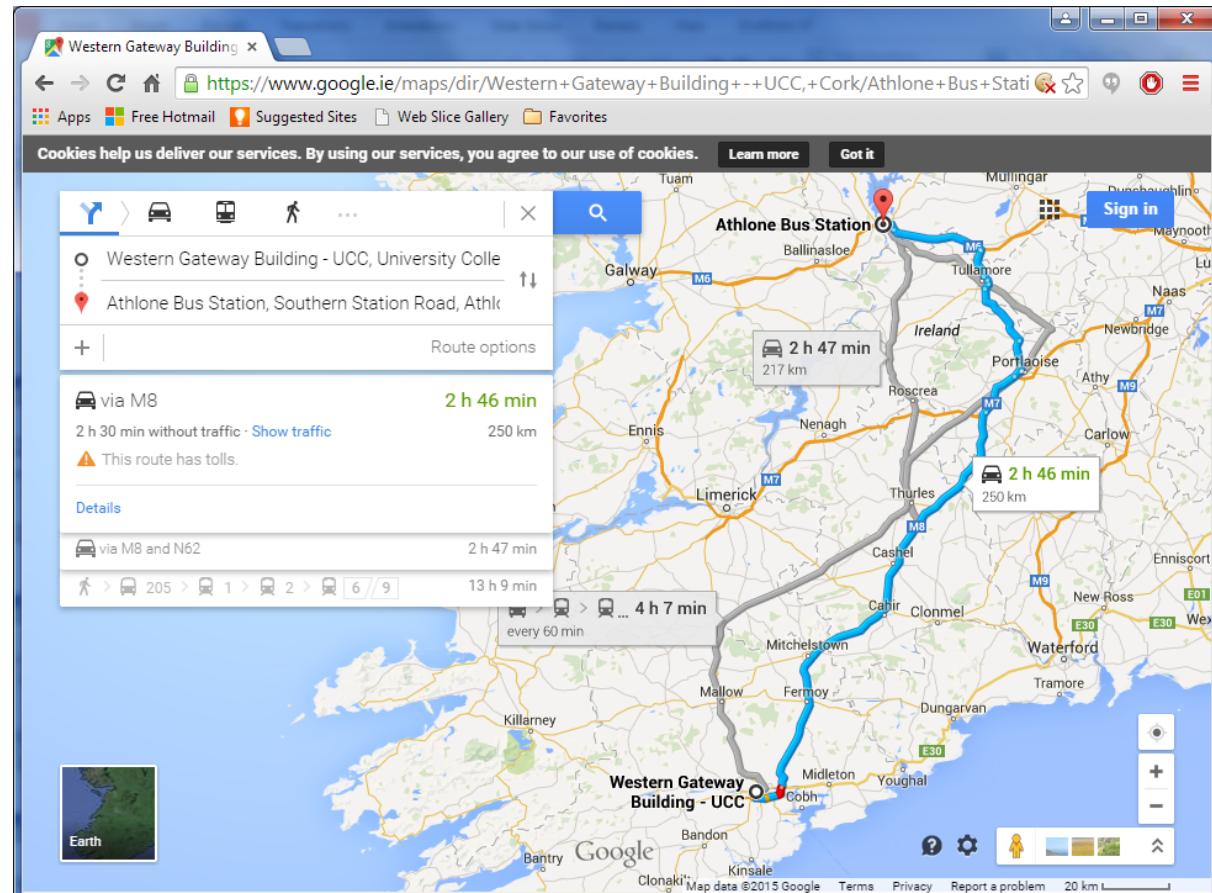
<http://osullivan.ucc.ie/teaching/cs1113/>

# Fastest Paths

Graphs with edge weights

Fastest paths

Dijkstra's Algorithm



**Driving directions**

|

via M8      **2 h 46 min**      250 km

2 h 30 min without traffic · [Show traffic](#)

This route has tolls.

---

**Western Gateway Building - UCC**  
University College, Western Road, Cork

---

Get on N40 in Cork from N71  
 4 min (2.5 km)

- Head west on Western Rd/N22 toward R846  
 Continue to follow N22  
280 m
- Slight left onto Victoria Cross/N71  
 Continue to follow N71  
1.2 km
- At the roundabout, take the 2nd exit onto Sarsfield Rd/N71/R849  
650 m
- Slight left onto the ramp to Carrigaline/Ringaskiddy/Rochestown/Rosslare  
400 m

---

Take M8 to R445 in Laois. Take exit 18 from M7  
 1 h 30 min (168 km)

---

Get on M6 in Westmeath from N80 and N52  
 40 min (47.2 km)

Sign in

Driving directions

Map data ©2015 Google Terms Privacy Report a problem 20 km

Print

X

Earth

Google

Paths and circuits

## The cost of a path

The algorithm in the previous lecture finds a path with the least number of edges.

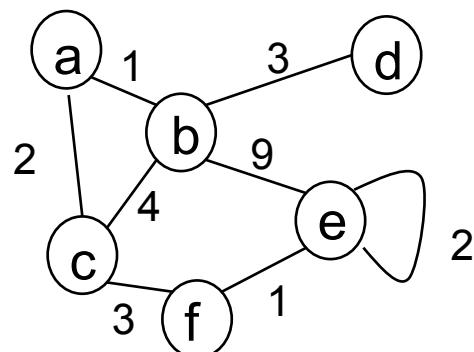
But what if each edge has different properties?

- e.g. distances, travel times, monetary cost, number of enemies who will attack you, energy use, ...

The cost of a path will be the sum of the costs of all the individual edges in the path.

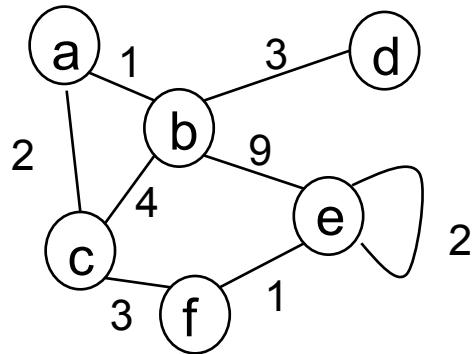
# Weighted Graphs

A **weighted simple graph**  $G$  is a pair  $(V, E)$ , where  
 $V$  is a set of **vertices** (representing the objects)  
 $E$  is a set of **weighted edges**, where each weighted edge in  
 $E$  is an ordered pair, consisting of an edge (a set of 1 or  
2 vertices) followed by a numerical **weight**.



$$\begin{aligned} G &= (V, E), \text{ where} \\ V &= \{a, b, c, d, e, f\} \\ E &= \{\{\{a, b\}, 1\}, \{\{a, c\}, 2\}, \{\{b, c\}, 4\}, \\ &\quad \{\{b, d\}, 3\}, \{\{b, e\}, 9\}, \{\{c, f\}, 3\}, \\ &\quad \{\{e, e\}, 2\}, \{\{e, f\}, 1\}\} \end{aligned}$$

## Path costs

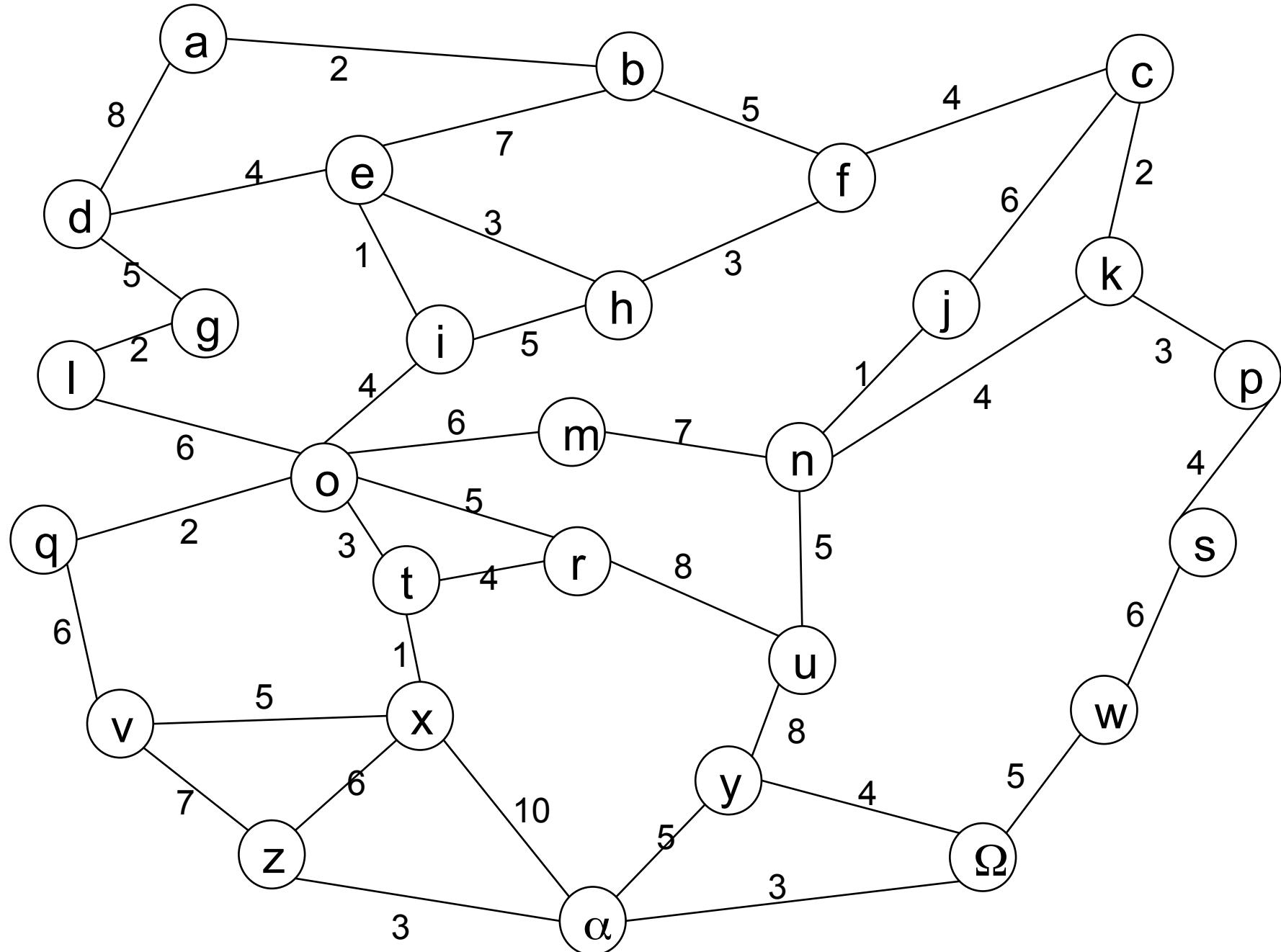


$G = (V, E)$ , where  
 $V = \{a, b, c, d, e, f\}$   
 $E = \{(\{a, b\}, 1), (\{a, c\}, 2), (\{b, c\}, 4), (\{b, d\}, 3), (\{b, e\}, 9), (\{c, f\}, 3), (\{e, e\}, 2), (\{e, f\}, 1)\}$

Path $\langle a, b, c, f \rangle$ :	$\langle (a, b), (b, c), (c, f) \rangle$	<u>Total cost</u>
Edge costs:	1      4      3	8

What is the cost of  $\langle f, e, e, b \rangle$ ?

Can we write an algorithm that computes the cheapest path between any pair of vertices?



$G = (V, E)$ , where

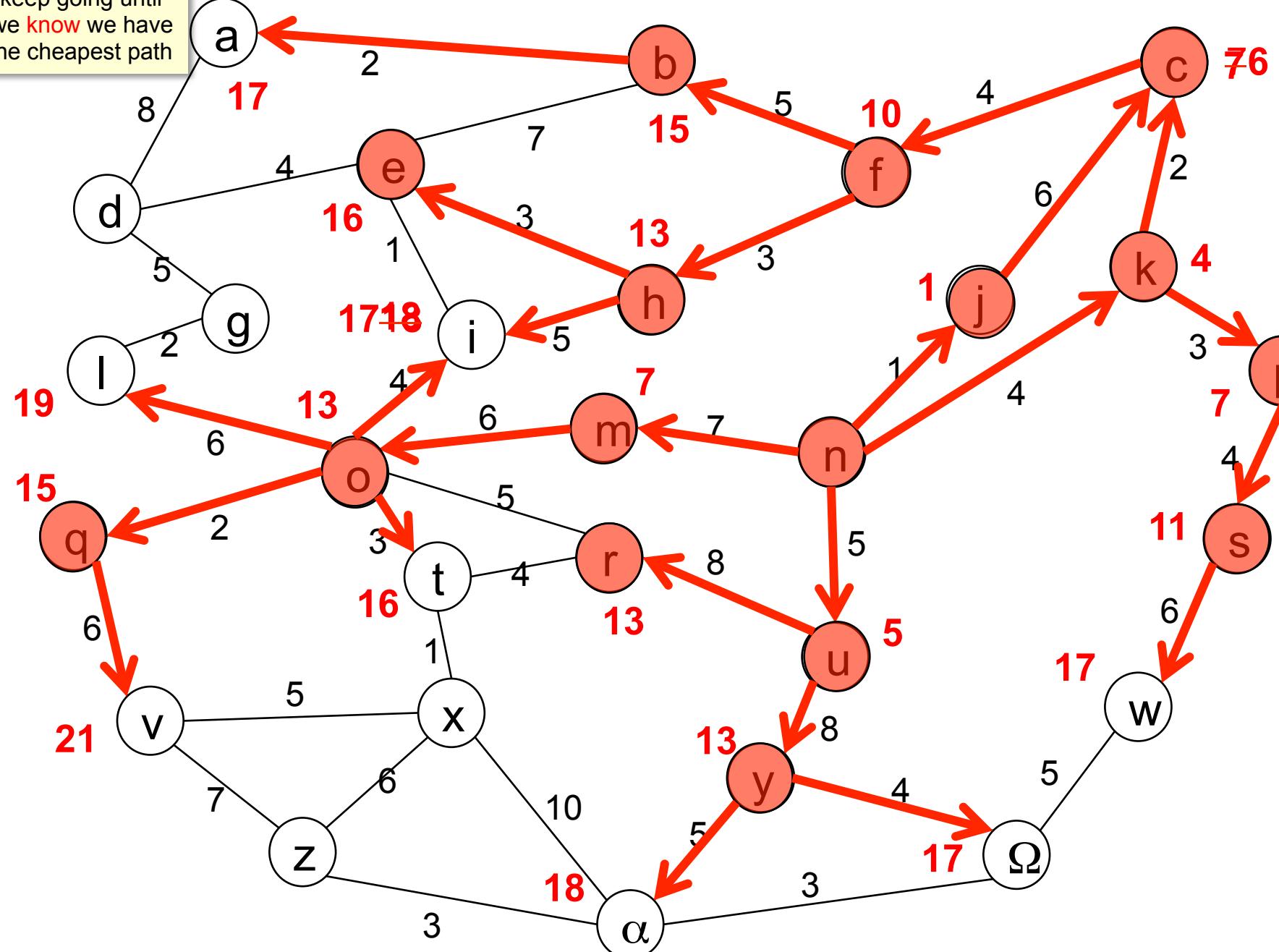
$V = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, \alpha, \Omega\}$

$E = \{ (\{a,b\}, 2), (\{a,d\}, 8), (\{b,e\}, 7), (\{b,f\}, 5), (\{c,f\}, 5), (\{c,k\}, 5), (\{d,e\}, 4), (\{d,g\}, 5), (\{e,h\}, 3), (\{e,i\}, 1), (\{f,h\}, 3), (\{g,l\}, 2), (\{h,i\}, 5), (\{i,o\}, 4), (\{j,n\}, 1), (\{k,n\}, 4), (\{k,p\}, 3), (\{l,o\}, 6), (\{m,n\}, 7), (\{m,o\}, 6), (\{n,u\}, 5), (\{o,q\}, 2), (\{o,r\}, 5), (\{o,t\}, 3), (\{p,s\}, 4), (\{q,v\}, 6), (\{r,t\}, 4), (\{r,u\}, 8), (\{s,w\}, 6), (\{t,x\}, 1), (\{u,y\}, 8), (\{v,x\}, 5), (\{v,z\}, 7), (\{w,\Omega\}, 5), (\{x,z\}, 6), (\{x,\alpha\}, 10), (\{y,\alpha\}, 5), (\{y,\Omega\}, 4), (\{z,\alpha\}, 3), (\{\alpha,\Omega\}, 3) \}$

Basic idea: grow all paths out from the start one edge at a time, always expanding from the vertex with the lowest cost so far.

When two paths join up at the same vertex, only keep the cheaper one. When we try to expand from the target vertex, we know we must have found the cheapest path.

keep going until  
we **know** we have  
the cheapest path



# Finding the cheapest path (informal)

Start at the start vertex

Record that it is the start and costs 0 to get there

For each adjacent vertex, record the cost of getting there from the current vertex, and record how we got there

Mark the current vertex as being "done"

From all the vertices that are not done but that have some cost recorded, find the one with lowest cost, and make that the current vertex

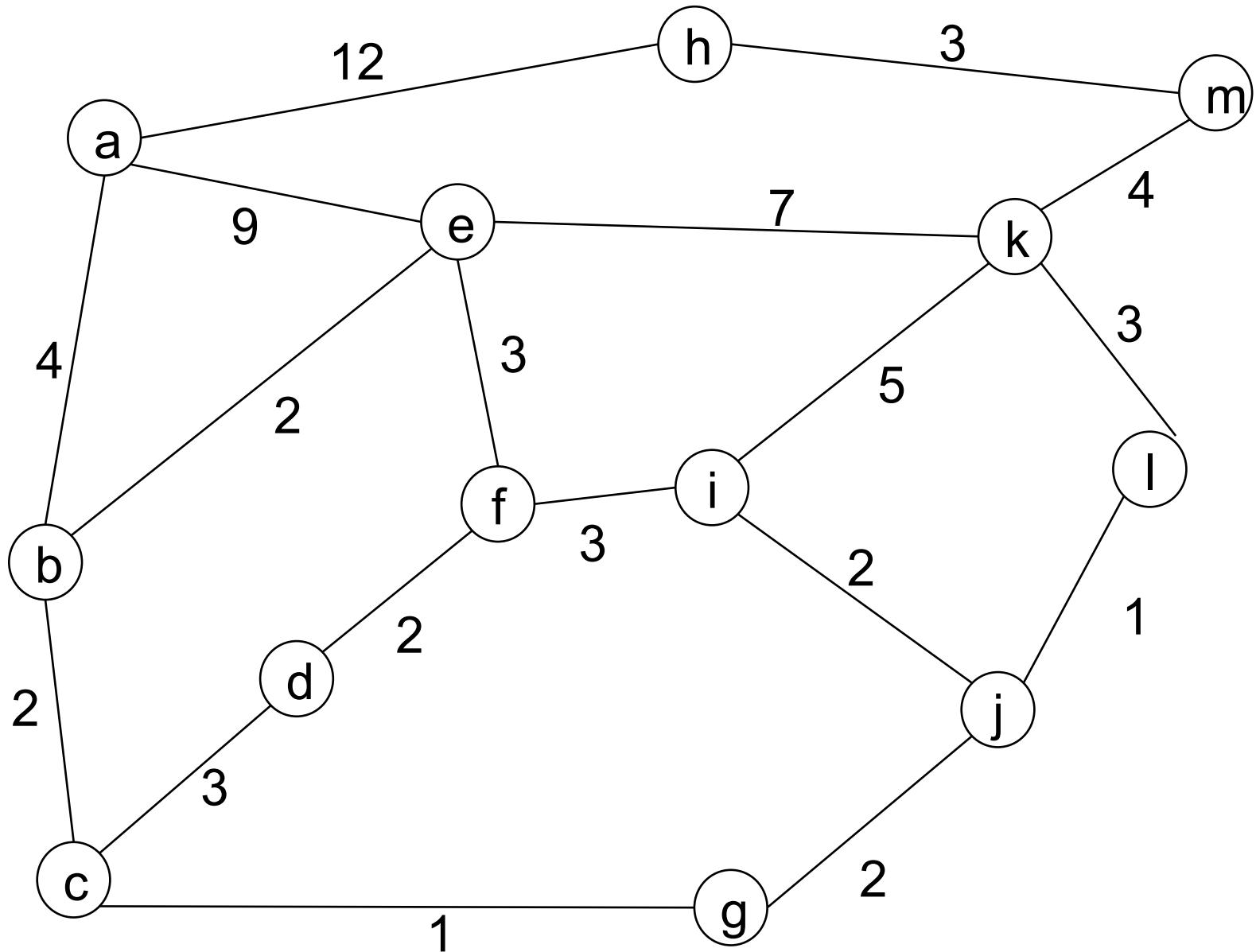
If it is the target, we are finished and can report success; if not, we will try expanding from that vertex

For each adjacent vertex to the current one, if it is not "done", compute the cost of extending the path, and if it is a better cost or a new cost, update our records

Mark the current vertex as done

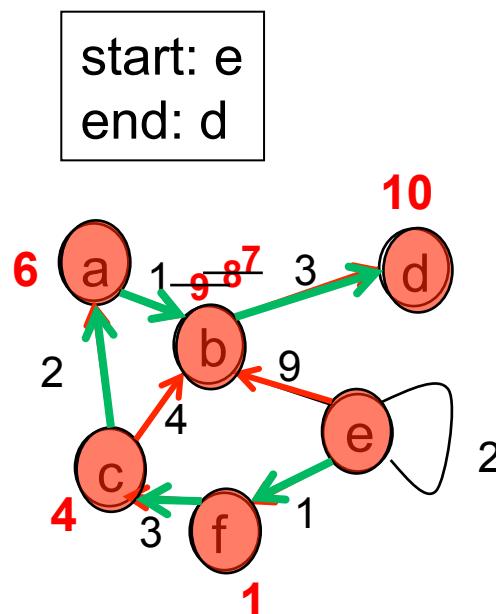
And repeat ...

If we run out of vertices, stop and report failure



# Recording the cost and the path

We use an array with 3 cells for each vertex in the graph.  
The first cell says whether we have checked that vertex.  
The second cell contains our current cost of getting there.  
The third cells will tell us the previous vertex on that path.



vertex:	a	b	c	d	e	f	
done?	1	1	1		1	1	
cost:	6	9	8	7	4	10	
previous:	c	e	c-a	f	b	0	e

# Dijkstra's algorithm for finding the cheapest path

Algorithm: cheapestpath1 (from Dijkstra)

Input: a weighted graph  $G=(V,E)$ , where  $V = \{1, 2, \dots, n\}$

Input: a vertex  $x$  from  $V$ , the start

Input: a vertex  $y$  from  $V$ , the end

Output: an table representation of the path, or null if none

```
1. L := a table with 3 rows for each of n vertices, all null
2. v := x
3. L[v][2] := 0                                //the cost of getting to v
4. L[v][3] := 0                                //the previous vertex in the path to v
5. while v is not null
6.     for each vertex j adjacent to v          //expand paths from v
7.         if L[j][1] != 1                        //if j not done
8.             then cost := L[v][2] + weight of edge {v,j} //compute cost
9.             if L[j][2]== null OR cost < L[j][2]        //if better
10.                then L[j][2] := cost                 //update j's cost
11.                L[j][3] := v                      //say how we got here
12.            L[v][1] := 1                        //mark v as done
13.            v := vertex with smallest L[v][2] and with L[v][1]== null
               or null if there isn't one           //find cheapest vertex
14.            if v == y, return L                //stop - we've found the cheapest path
15.        return null                         //we didn't reach the target by any path
```

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	4				9			12					
b	4		2		2								
c		2		3			1						
d			3			2							
e	9	2				3				7			
f				2	3				3				
g			1							2			
h	12											3	
i					3				2	5			
j						2		2			1		
k					7			5			3	4	
l									1	3			
m							3		4				

	a	b	c	d	e	f	g	h	i	j	k	l	m
?													
cost													
path													

Next lecture ...

Representing Graphs  
Connected Graphs  
Circuits



# CS1113

## Circuits in Graphs

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

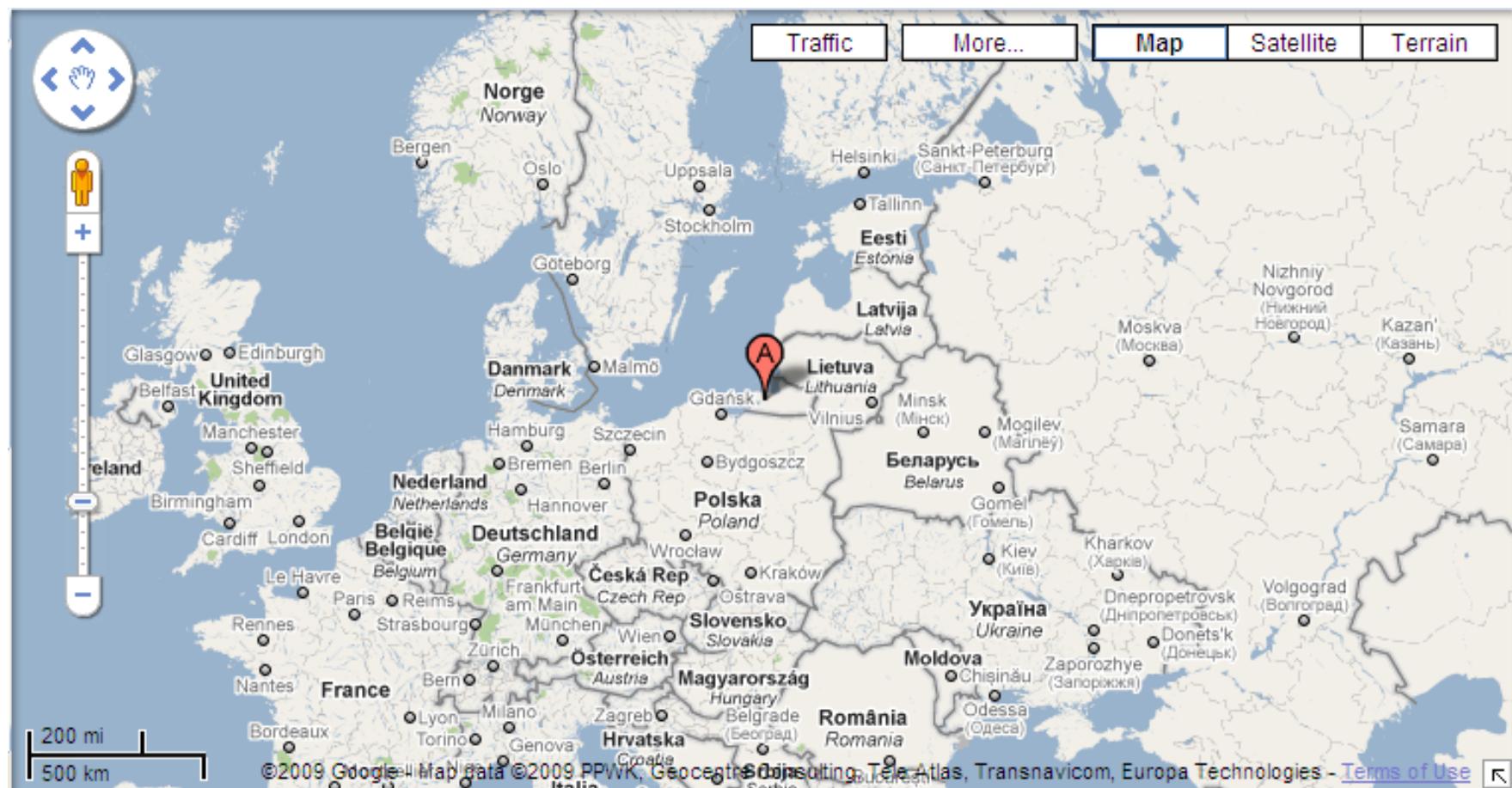
# Circuits

A walk round the bridges of Königsberg

...and...

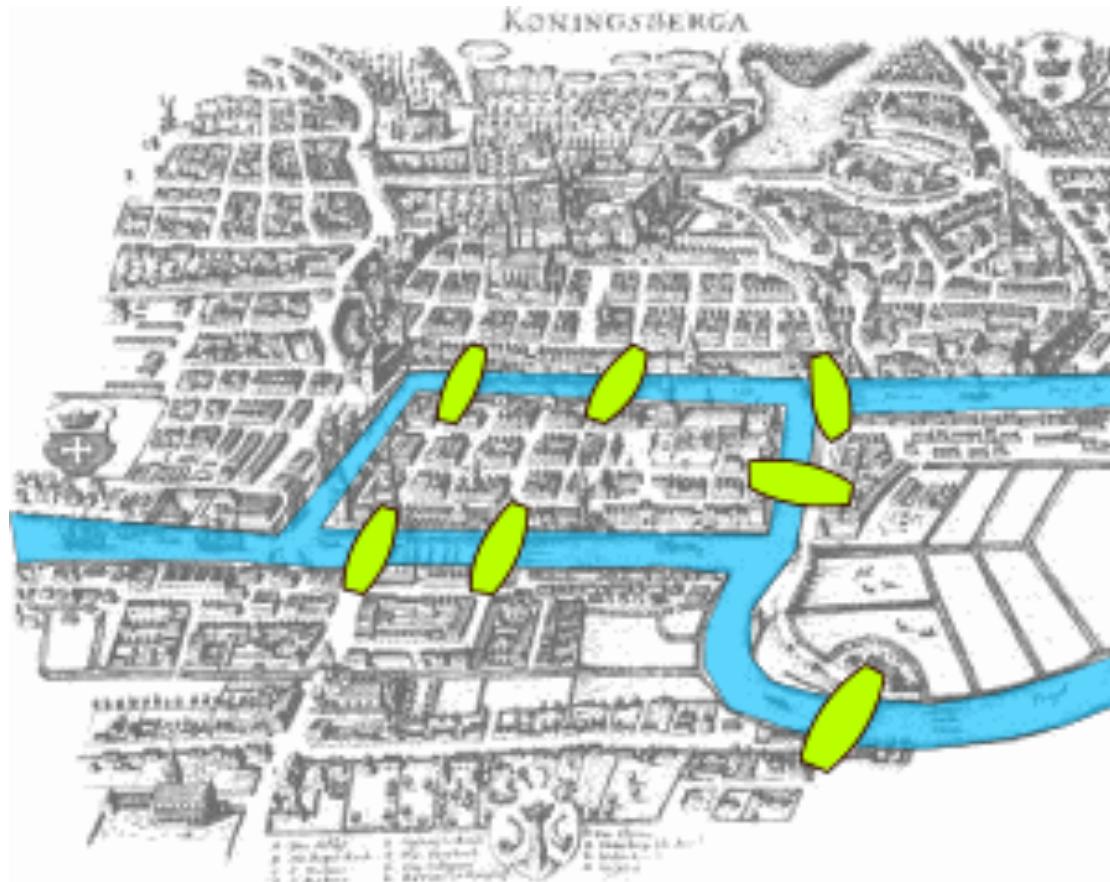
- circuits
- Euler circuits
- Hamiltonian circuits
- representing graphs
- subgraphs
- connected graphs





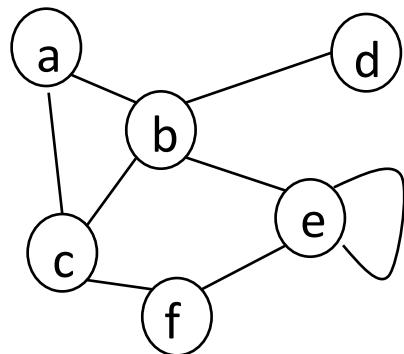
# The Bridges of Königsberg

The residents of Königsberg liked to take a walk on Sunday afternoons. Was it possible for them to cross every bridge exactly once, and return to their starting point?



# Circuits

A **circuit** is a path that starts and ends at the same vertex.



$\langle(a,b),(b,c),(c,a)\rangle$   
 $\langle(f,e),(e,e),(e,f),(f,c),(c,b),(b,a),(a,c),(c,f)\rangle$

$\langle(a,b),(b,a)\rangle$

$\langle(e,e)\rangle$

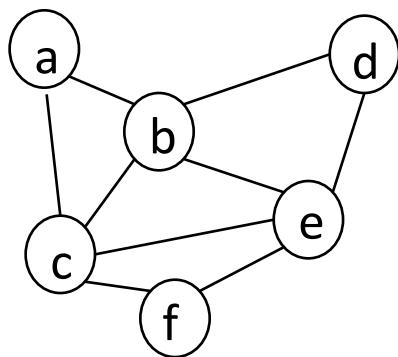
are all circuits

$\langle(d,b),(f,c)\rangle$   
 $\langle(d,b),(b,c),(c,f)\rangle$   
 $\langle(c,b),(b,a),(b,c),(c,b)\rangle$

are not circuits

# Euler Circuits

An **Euler circuit** is a circuit that contains every edge in the graph exactly once.



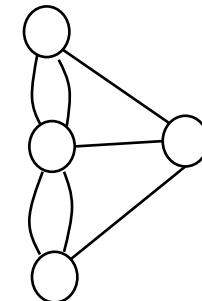
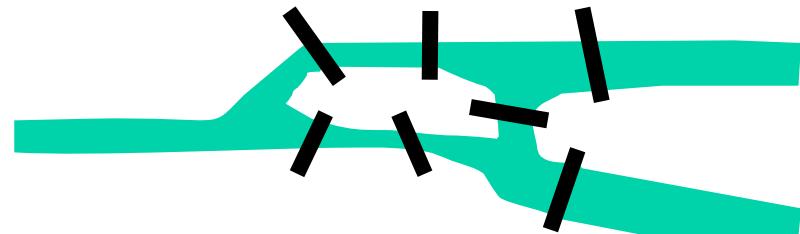
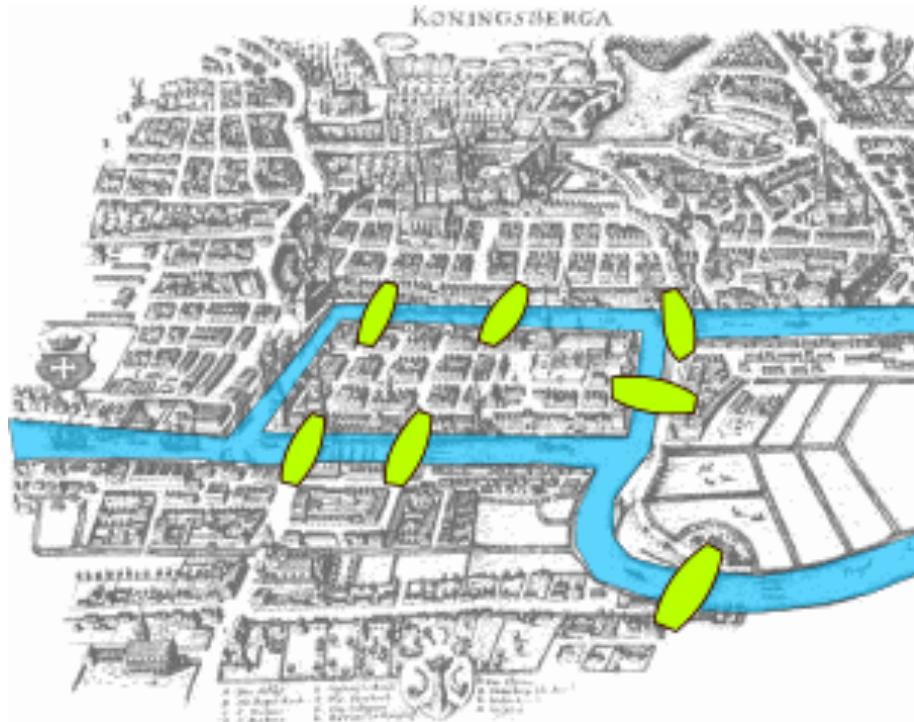
$\langle (c,b),(b,a),(a,c),(c,f),(f,e),(e,d),(d,b),(b,e),(e,c) \rangle$   
 $\langle (b,e),(e,d),(d,b),(b,c),(c,e),(e,f),(f,c),(c,a),(a,b) \rangle$   
are both Euler circuits

$\langle (a,c),(c,f),(f,e),(e,d),(d,b),(b,a) \rangle$   
is not an Euler circuit

An **Euler path** is a path that contains every edge in the graph exactly once.

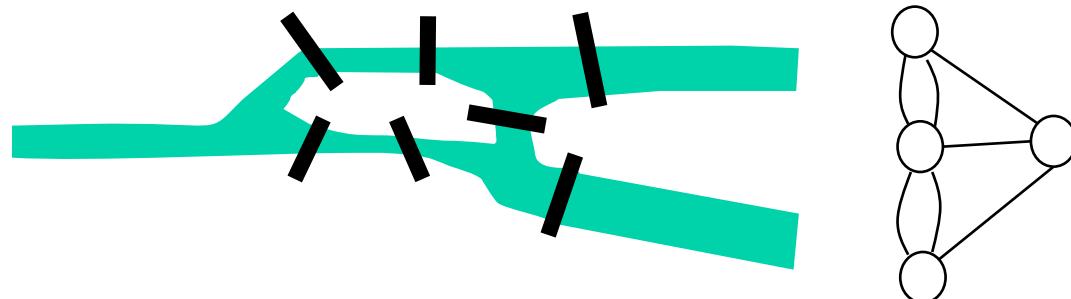
# The Bridges of Königsberg

The residents of Königsberg like to take a walk on Sunday afternoons. Is it possible for them to cross every bridge exactly once, and return to their starting point?



Is there an Euler circuit for the above multigraph?

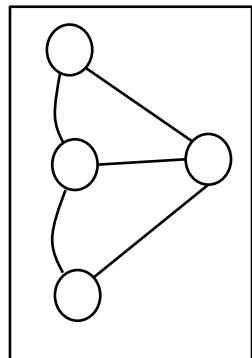
# Euler Circuits



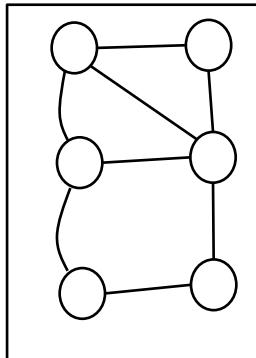
An Euler circuit visits every edge exactly once, and starts and finishes at the same vertex

Is there an Euler circuit for the above multigraph? No

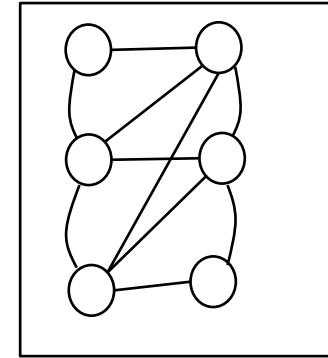
How about these?



No



No



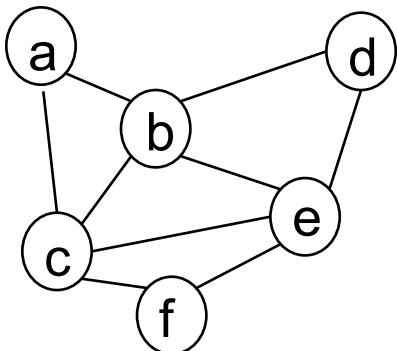
yes

A graph has an Euler circuit if and only if every vertex in the graph has even degree.

# Hamiltonian Circuits

A **Hamiltonian circuit** is a circuit that contains every vertex in the graph exactly once.

$\langle(a,c),(c,f),(f,e),(e,d),(d,b),(b,a)\rangle$   
is a Hamiltonian circuit



$\langle(c,b),(b,a),(a,c),(c,f),(f,e),(e,d),(d,b),(b,e),(e,c)\rangle$   
is not a Hamiltonian circuit (it visits c, b and e twice)

$\langle(d,e),(e,f),(f,c),(c,b),(b,a)\rangle$   
is not a Hamiltonian circuit (it starts and finishes at different vertices)

A **Hamiltonian path** is a path that contains every vertex in the graph exactly once.

William Hamilton



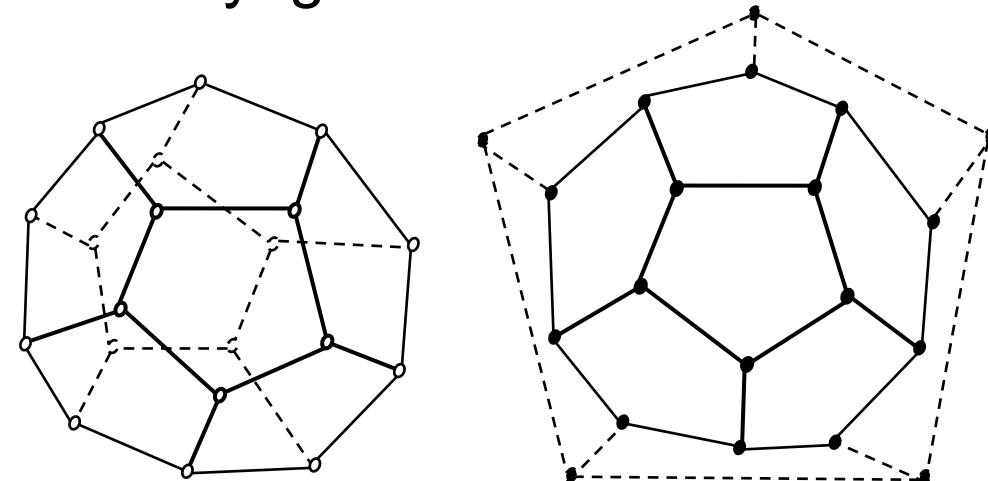
William Rowan Hamilton

Born	August 4, 1805 Dublin, Ireland
Died	September 2, 1865 (aged 60) Dublin, Ireland
Residence	Ireland
Nationality	Irish
Field	Mathematician, physicist, and astronomer
Institutions	Trinity College Dublin
Alma mater	Trinity College Dublin

from wikipedia

There is no simple method for deciding whether or not a graph has a Hamiltonian circuit – we have to search for one.

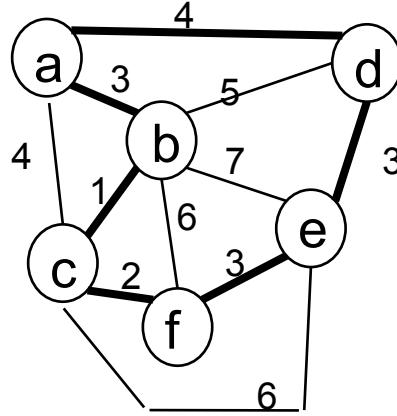
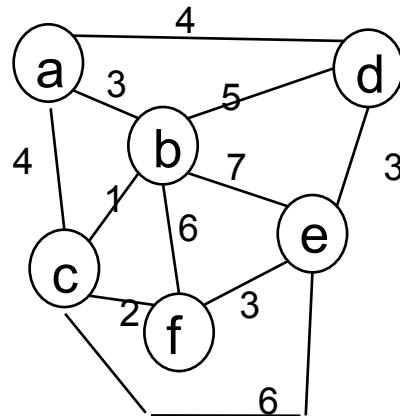
"A voyage around the world"



Exercise: find a circuit that visits every vertex exactly once.

# Hamiltonian Circuits and TSP

The idea of the Hamiltonian circuit is the basis of the **travelling salesman problem**



There is no known efficient algorithm for solving the TSP

Find the cheapest circuit that visits every city, but doesn't visit any city twice.

We will see the travelling salesman problem in later lectures.

# Representing Graphs

If graphs are ubiquitous in computer science and applications, then we need a way of representing them in programs

We defined graphs in terms of sets of vertices, and sets of edges, where the edges might themselves be sets: this is all we needed for understanding their properties and algorithms

But for programming, we need a representation that makes it easy to find out which edges connect which vertices

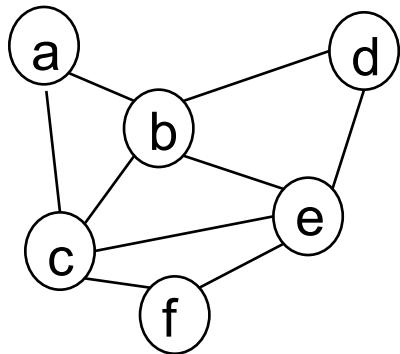
- we want it to allow fast lookup and response
- we want also want to minimise the amount of memory

There are two main methods:

- adjacency lists
- 2-dimensional arrays

# Adjacency list representation

With each vertex, we associate a list containing all other vertices to which it is linked



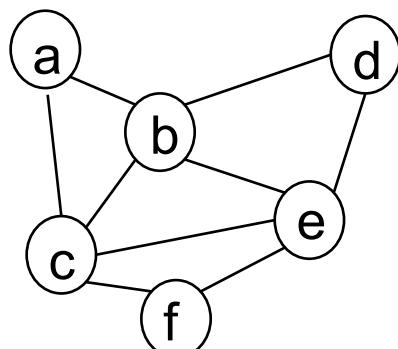
a: <b,c>	e: <b,c,d,f>
b: <a,c,d,e>	f: <c,e>
c: <a,b,e,f>	
d: <b,e>	

- easy to use for directed graphs
- easy to use for edge or vertex-weighted graphs
- can also be used for multigraphs
- only requires space for the edges that exist

but can be slow to process, since we have to search the adjacency list to find out whether or not two vertices are connected.

# Table representation

- two dimensional array (or matrix, or table)
- the rows and columns are indexed by the vertices (so we have an  $n \times n$  table for a graph with  $n$  vertices)
- each cell contains a "T" or a "1" if there is an edge between the corresponding vertices



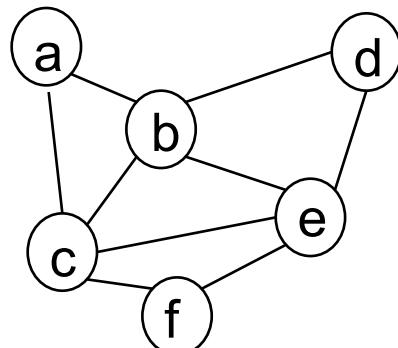
	a	b	c	d	e	f
a		1	1			
b	1		1	1	1	
c	1	1			1	1
d		1			1	
e		1	1	1		1
f			1		1	

## Properties of table representation

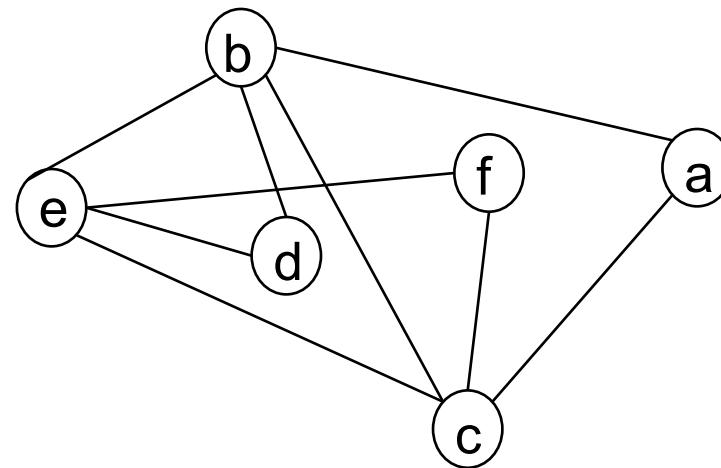
- in a simple graph, the table is symmetric
- in a directed graph, the table might not be symmetric
- for graphs with weights on the edges, we can represent the weight in the cell instead of "T" or "1"
- not good for representing multigraphs
- if the graph is sparse (i.e. not many edges), this wastes space
- but it is efficient to search – looking to see if there is an edge between two vertices  $v$  and  $w$  just involves looking at the cell  $[v][w]$ .

## The graph is not the sketch

- the way we draw the graph on the page does not change the underlying graph



and

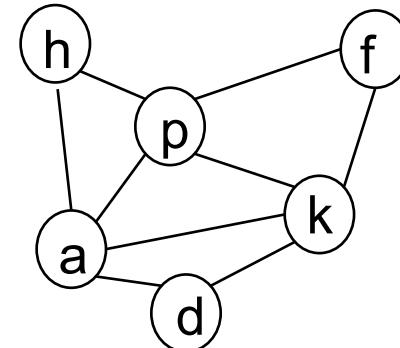
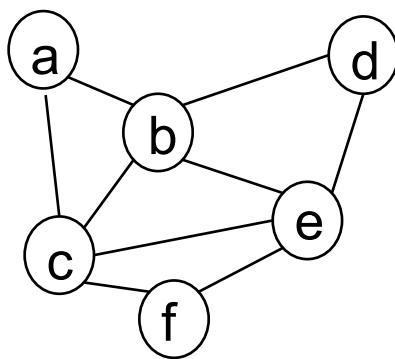


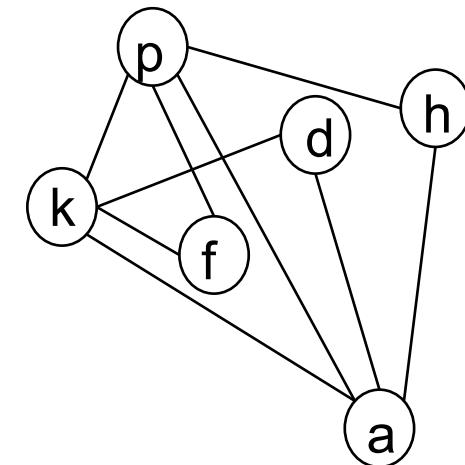
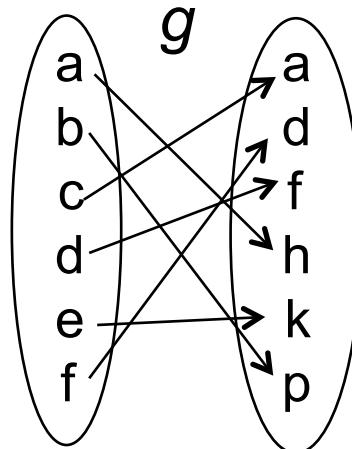
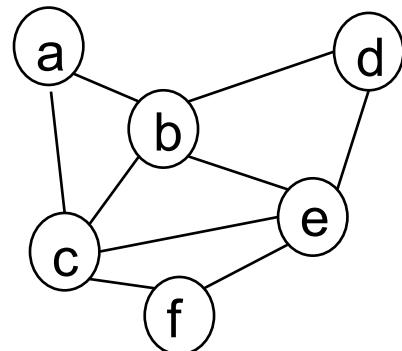
are the **same graph**. They will have an identical table representation.

## Isomorphic graphs

Suppose we have two graphs  $G_1=(V_1,E_1)$  and  $G_2=(V_2,E_2)$

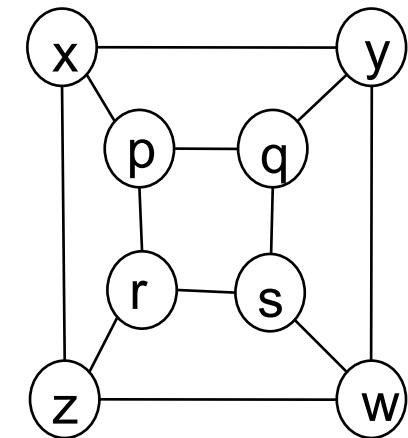
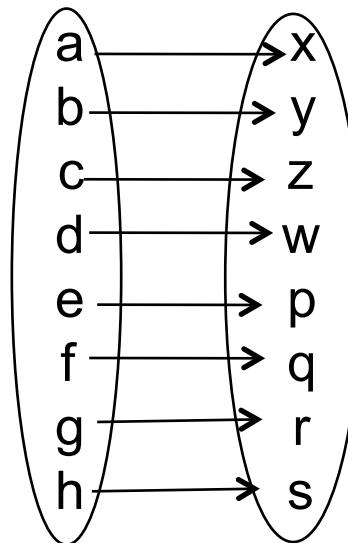
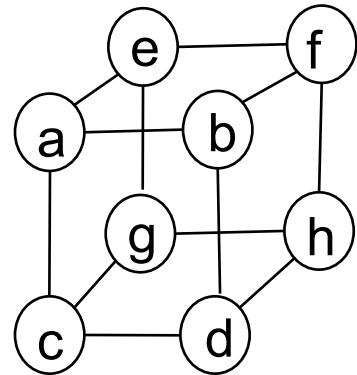
Then  $G_1$  and  $G_2$  are **isomorphic** if and only if there is a bijective function  $g: V_1 \rightarrow V_2$  such that there is an edge in  $E_1$  between  $v_i$  and  $v_j$  if and only if there is an edge in  $E_2$  between  $g(v_i)$  and  $g(v_j)$   
i.e. if we can rename the vertices in  $G_1$  so that  $G_1$  and  $G_2$  become the same graph





	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>
<b>a</b>		1	1			
<b>b</b>	1		1	1	1	
<b>c</b>	1	1			1	1
<b>d</b>		1			1	
<b>e</b>		1	1	1		1
<b>f</b>			1		1	

	<b>h</b>	<b>p</b>	<b>a</b>	<b>f</b>	<b>k</b>	<b>d</b>
<b>h</b>		1	1			
<b>p</b>	1			1	1	
<b>a</b>	1	1			1	1
<b>f</b>		1			1	
<b>k</b>		1	1	1		1
<b>d</b>			1		1	

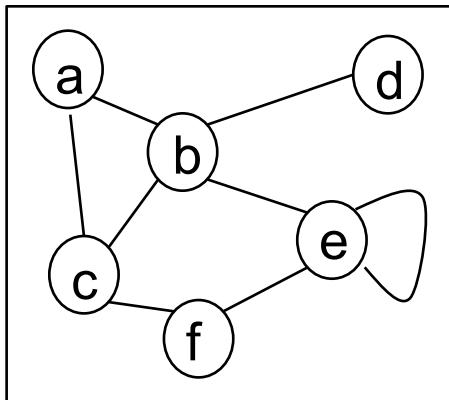


	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>	<b>h</b>	
<b>a</b>	1	1		1					
<b>b</b>	1			1		1			
<b>c</b>	1			1			1		
<b>d</b>		1	1					1	
<b>e</b>	1				1	1			
<b>f</b>		1			1			1	
<b>g</b>			1	1				1	
<b>h</b>				1	1	1			

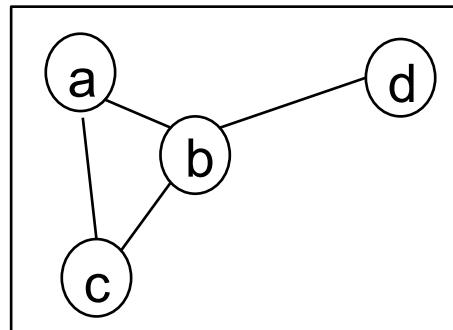
	<b>x</b>	<b>y</b>	<b>z</b>	<b>w</b>	<b>p</b>	<b>q</b>	<b>r</b>	<b>s</b>	
<b>x</b>		1	1		1				
<b>y</b>	1			1		1			
<b>z</b>	1			1			1		
<b>w</b>		1	1					1	
<b>p</b>	1					1	1		
<b>q</b>		1			1			1	
<b>r</b>			1		1		1		
<b>s</b>				1		1	1	1	

# Subgraphs

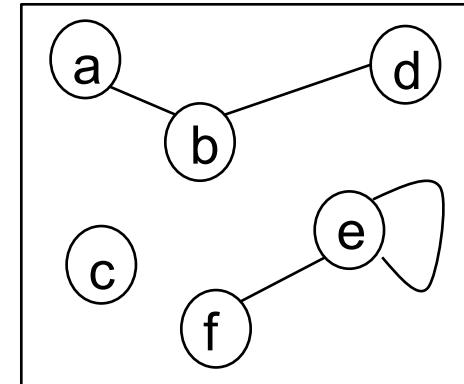
A **subgraph**  $H$  of a graph  $G=(V,E)$  is a graph  $H=(W,F)$ , such that  $W \subseteq V$ , and  $F \subseteq E$ . Note that  $H$  is a graph, so every edge in  $F$  links vertices in  $W$ .



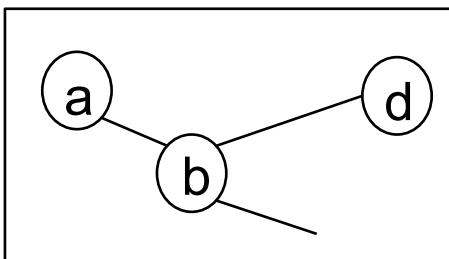
$G$



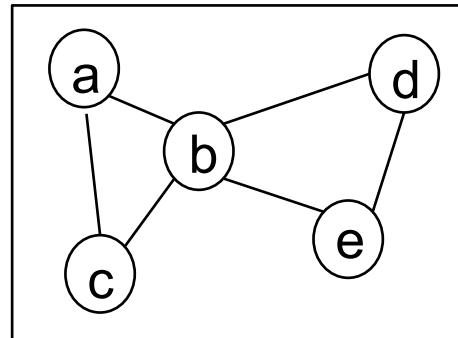
subgraph



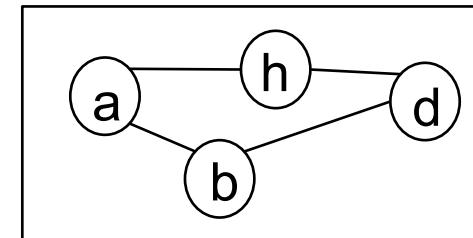
subgraph



NOT a subgraph



NOT a subgraph

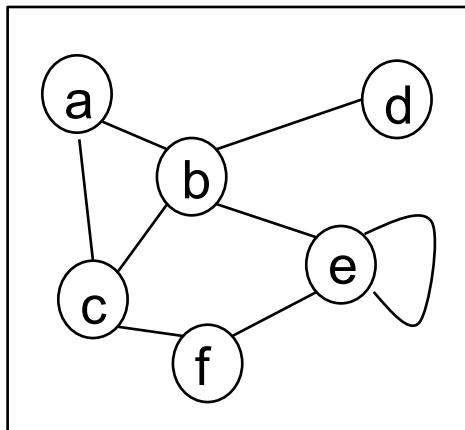


NOT a subgraph

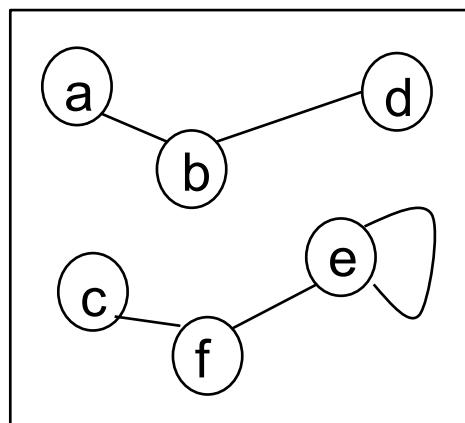
# Connected graphs

reminder

A graph is **connected** if every pair of vertices  $v,w$  can be connected by a path which starts at  $v$  and ends at  $w$ .



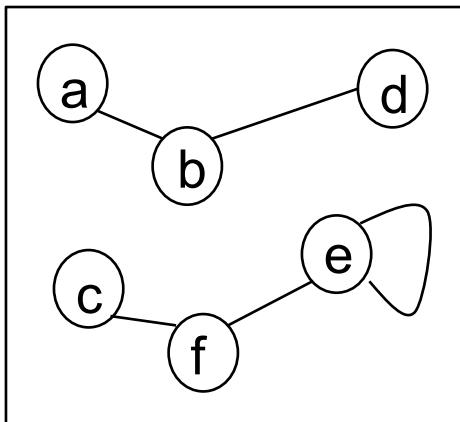
is a connected graph



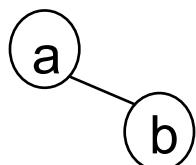
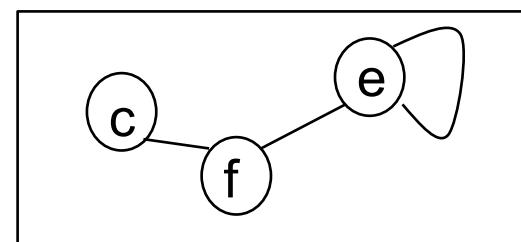
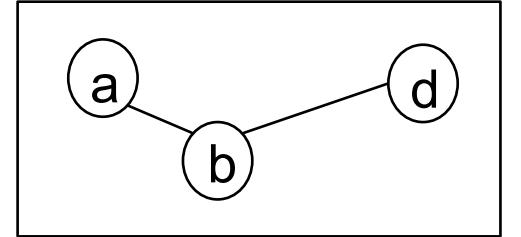
is not a connected graph – there is no path between, for example,  $a$  and  $f$ .

# Connected graphs

A **connected component**,  $H$ , of a graph  $G$  is a connected subgraph of  $G$  that is not a proper subgraph of any other connected subgraph of  $G$ .  
(we could say  $H$  is a **maximal** connected subgraph of  $G$ )



has two connected components:



is not a connected component (it is a connected subgraph, but not a maximal connected subgraph)

Next lecture ...

Trees

spanning trees

minimum spanning trees



# CS1113

## Trees

**Lecturer:**

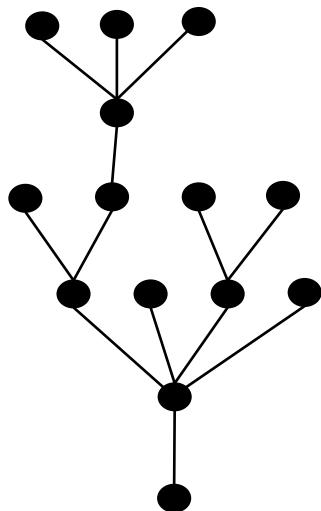
Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Trees



applications of trees

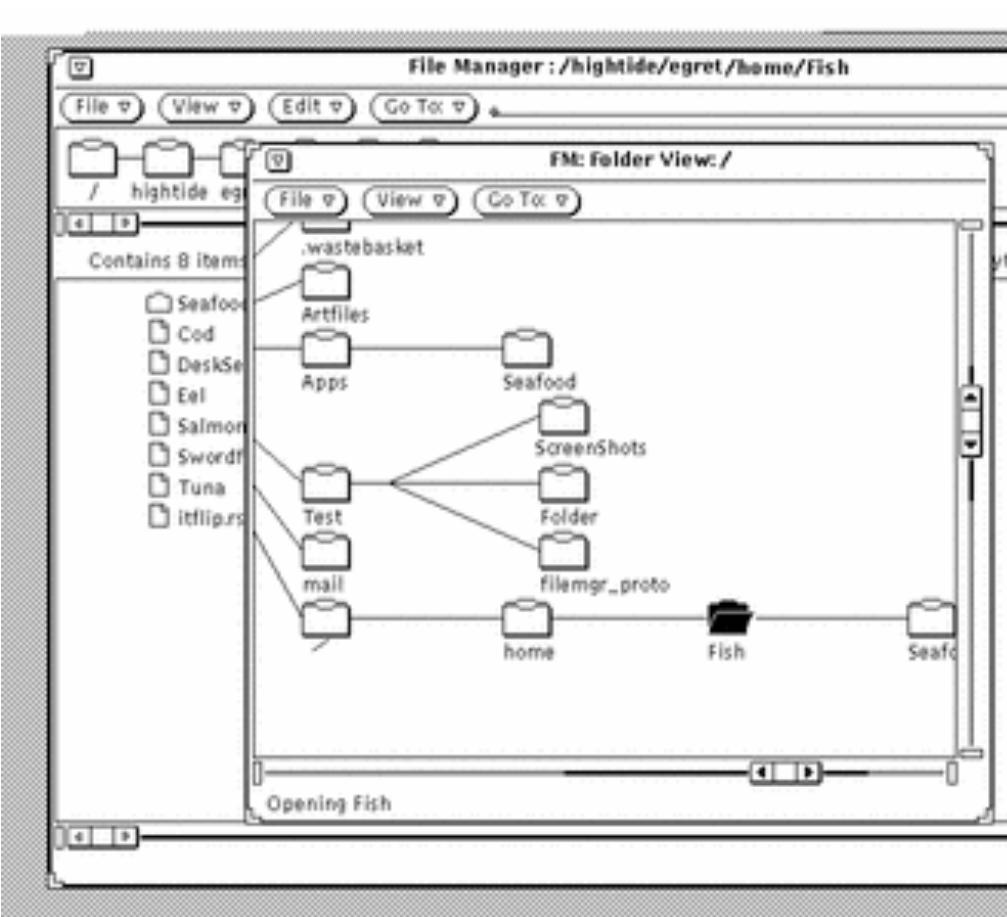
cycles  
trees  
rooted trees

spanning trees  
Prim's algorithm



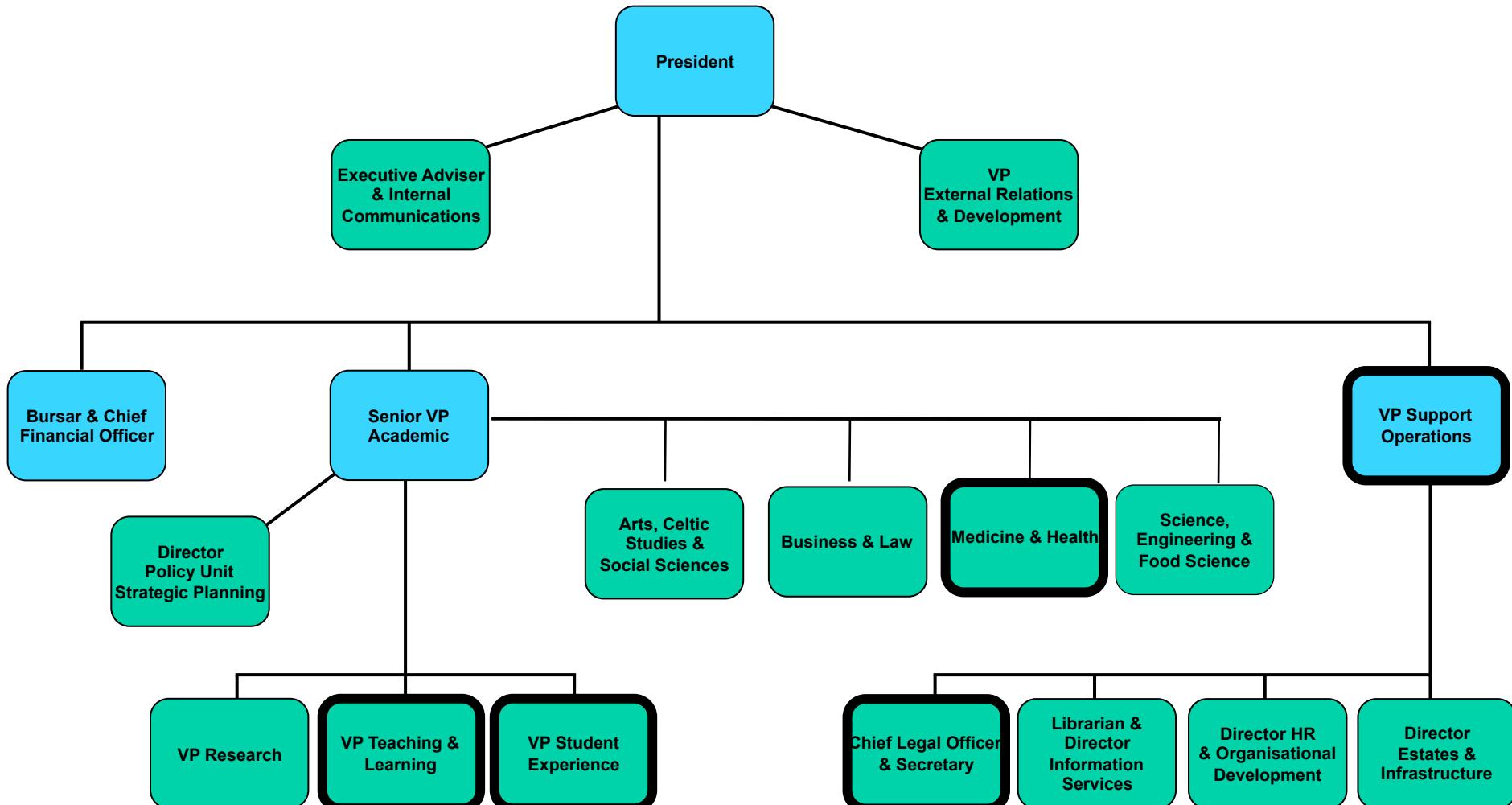
# Uses of Trees

File systems:



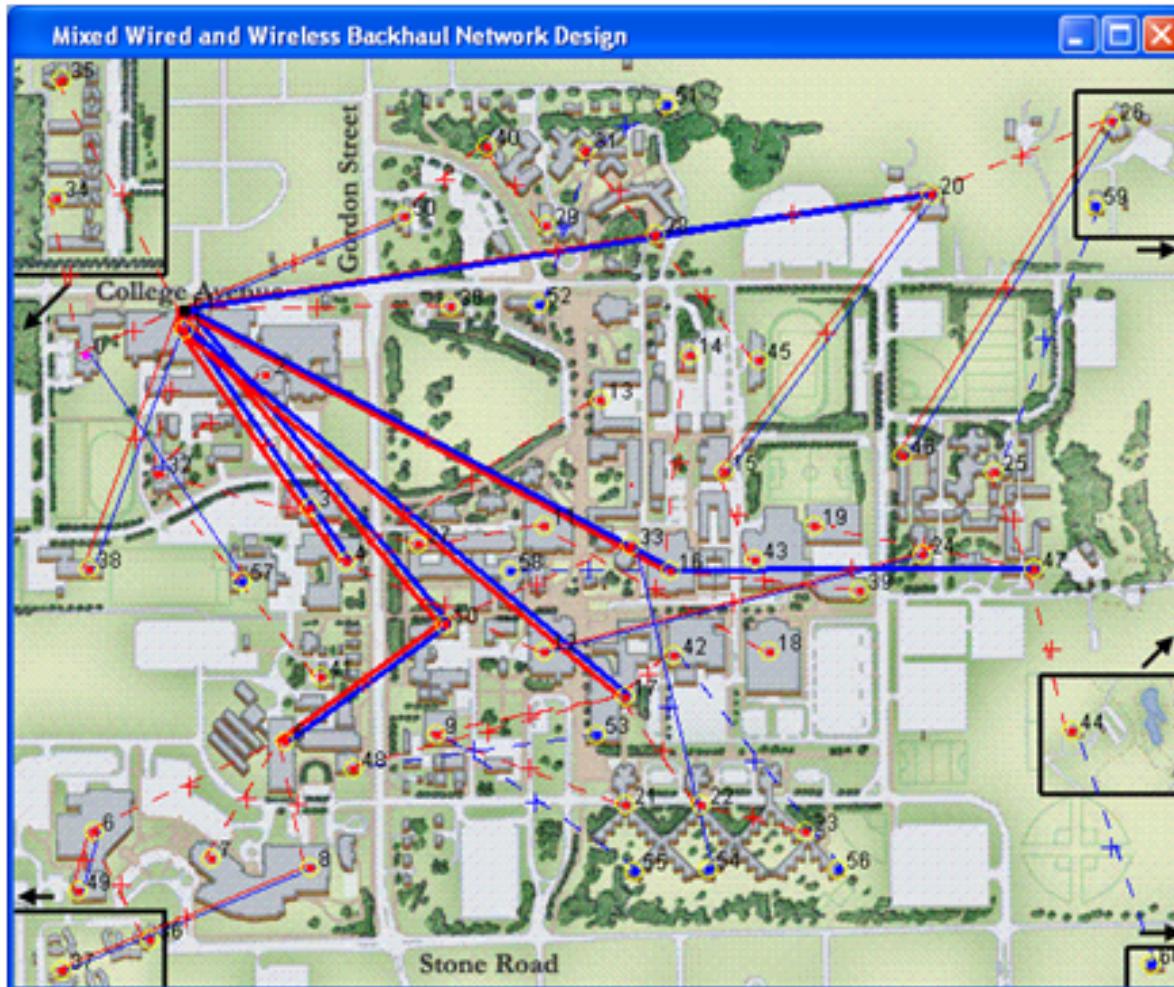
from [www.sun.com](http://www.sun.com)

# organisational structure:



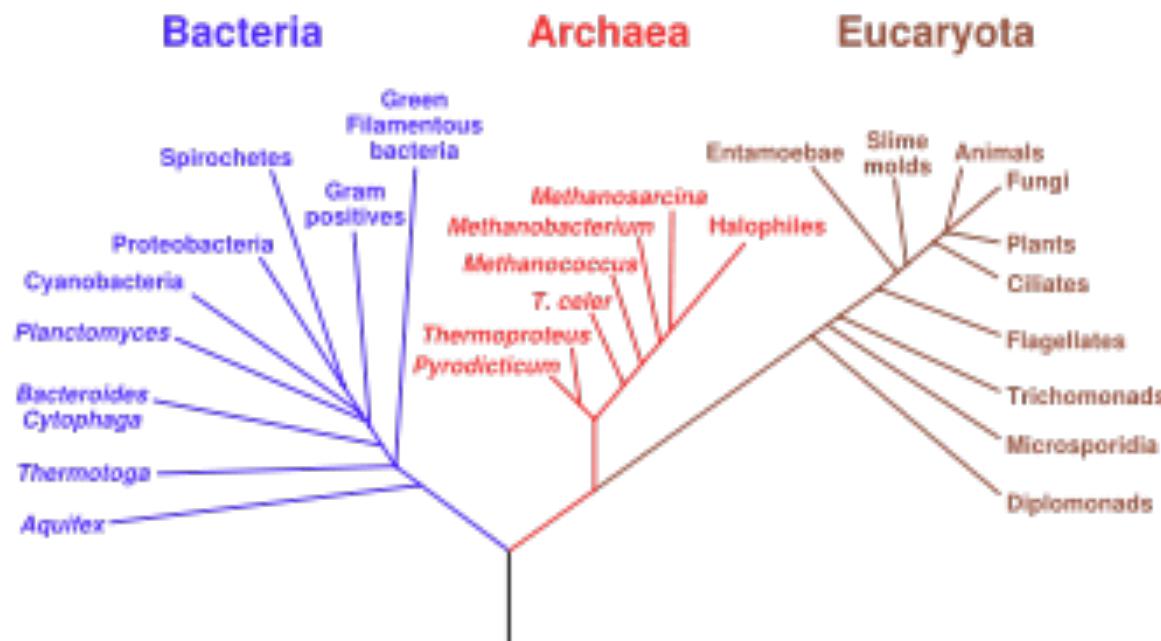
from UCC president's address (modified)

## mobile phone network backhaul:



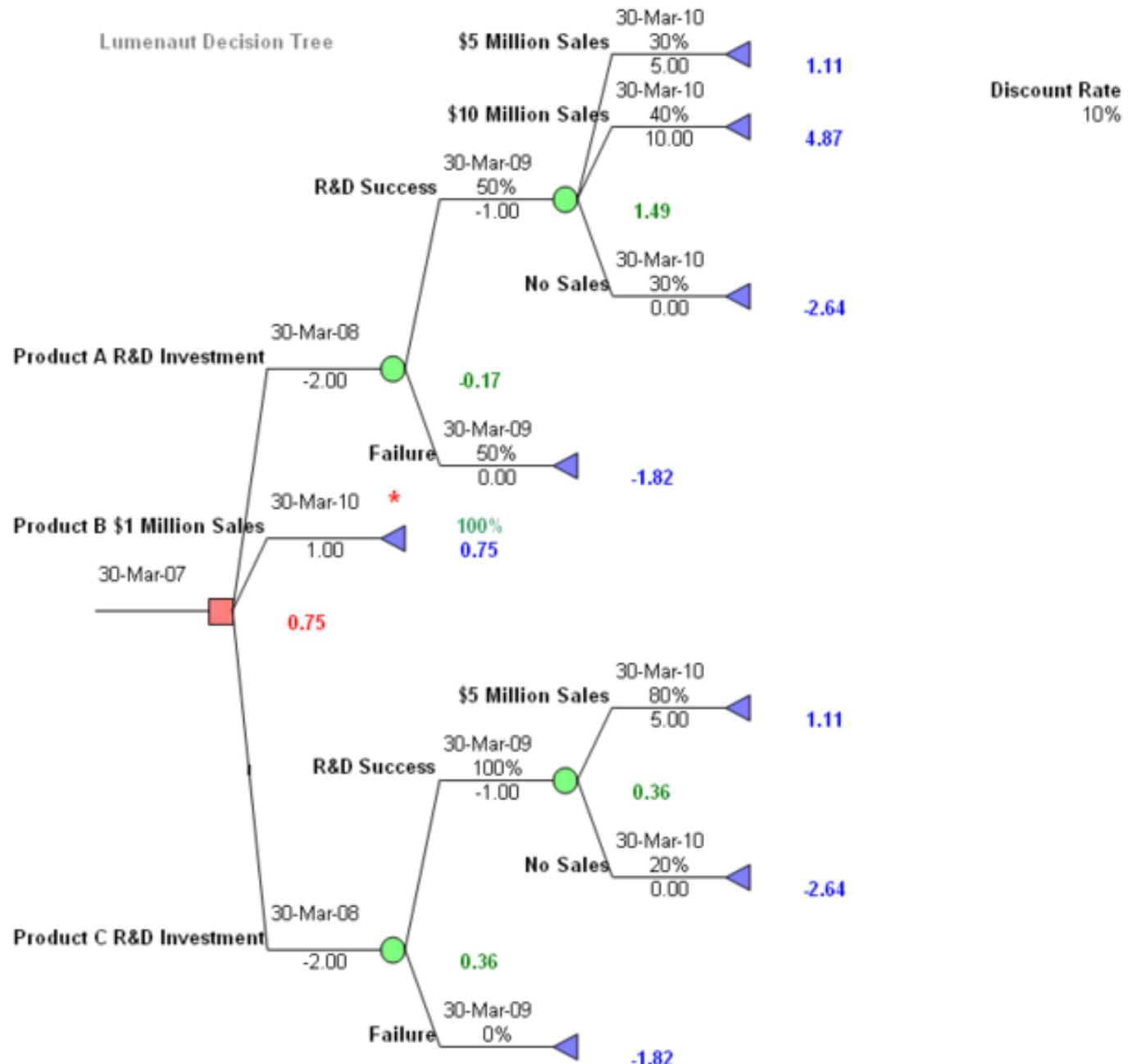
phylogenetic tree:

## Phylogenetic Tree of Life



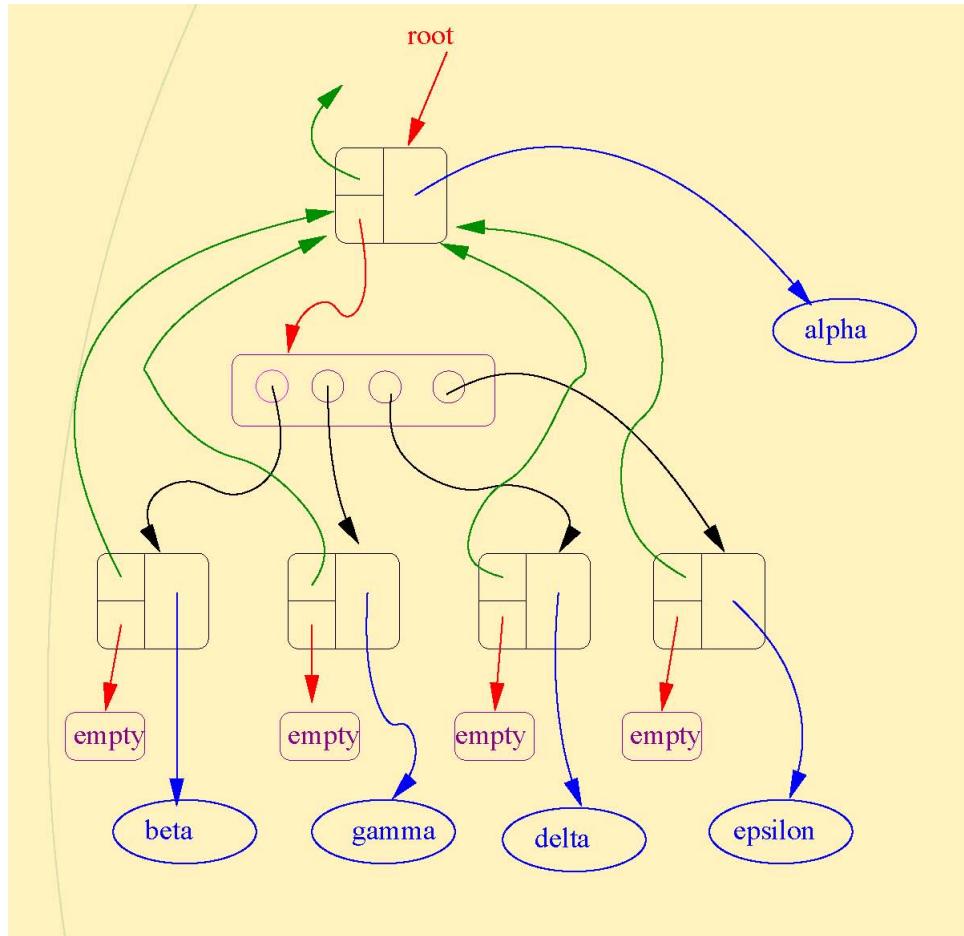
from wikipedia

# Decision trees:



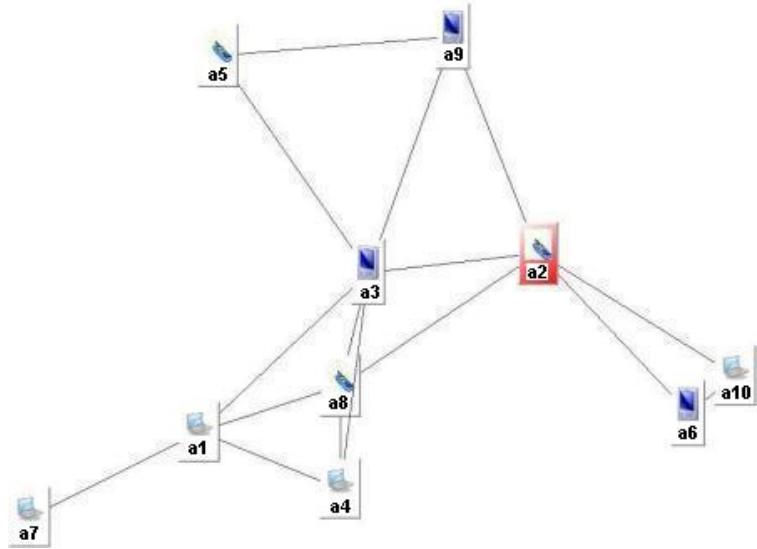
from wikipedia

# Computer science algorithms and data structures:

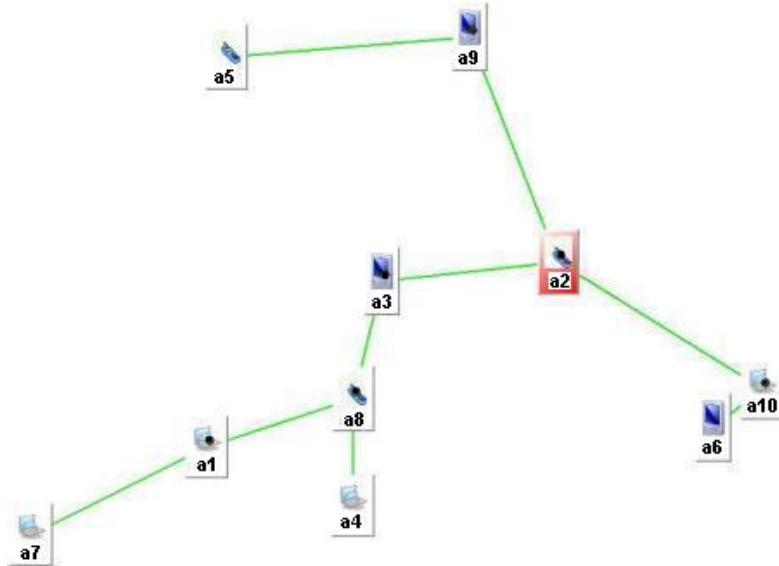


from CS2201 Data Structures, by Kieran Herley

## mobile ad hoc networks broadcast trees:



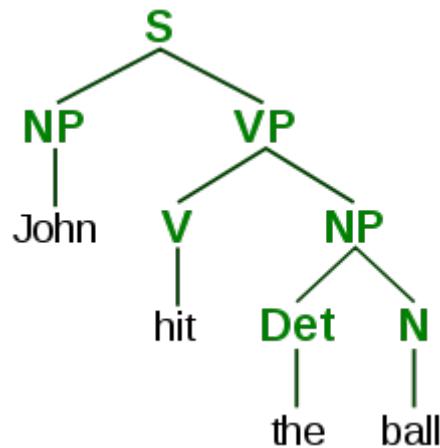
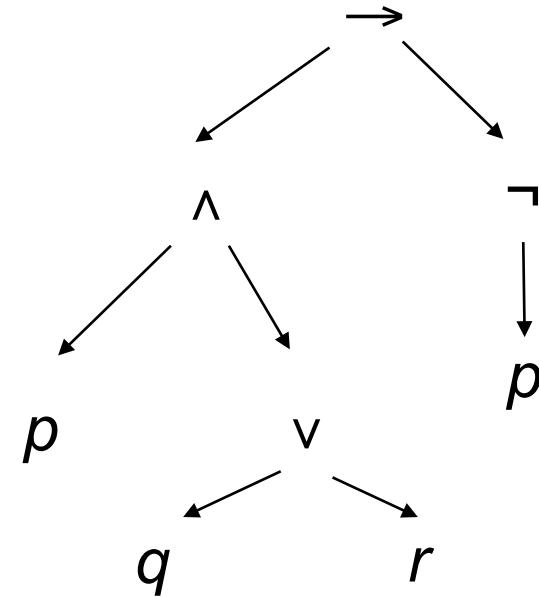
mobile radios, links  
show ability to  
communicate  
directly



rooted tree of active links,  
with minimum total  
broadcast power  
requirements

## propositional logic syntax trees

$$(p \wedge (q \vee r)) \rightarrow \neg p$$

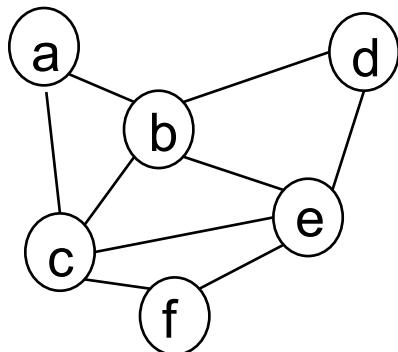


analysing structure of natural language sentences

# Cycles

A **cycle** is a circuit with the following properties

- it contains at least one edge
- no edge is visited twice
- no vertex is visited twice except the start/finish vertex

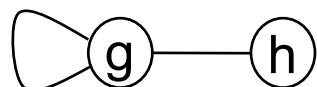


$\langle(b,e),(e,f),(f,c),(c,b)\rangle$  is a cycle

$\langle(a,c),(c,f),(f,e),(e,d),(d,b),(b,a)\rangle$  is a cycle

$\langle(b,c),(c,e),(e,f),(f,c),(c,a),(a,b)\rangle$  is not a cycle (c twice)

$\langle(d,e),(e,d)\rangle$  is not a cycle (edge d--e twice)

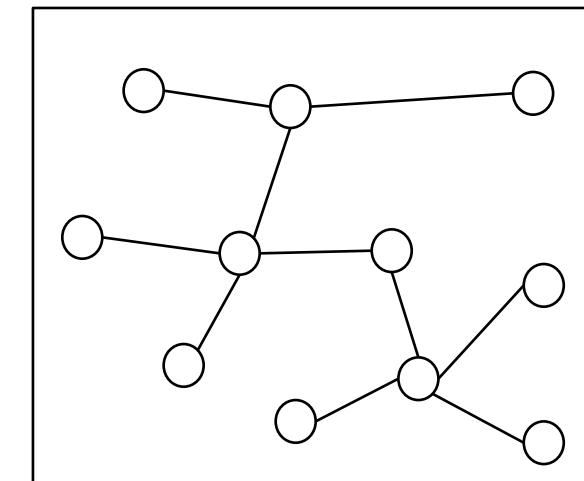
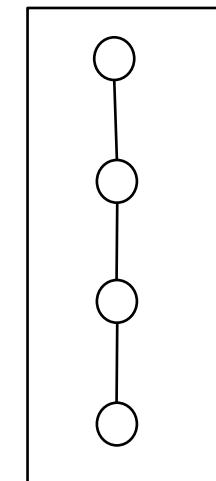
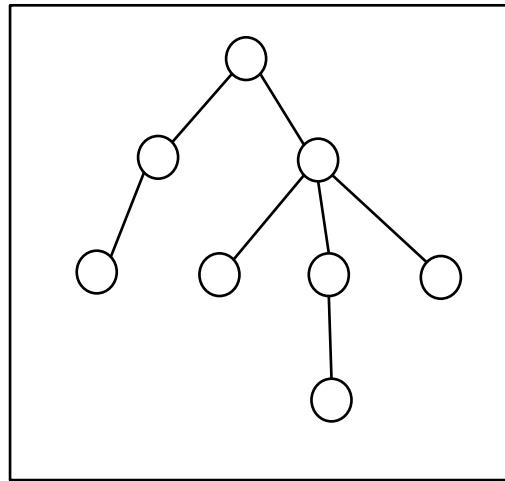


$\langle(g,g)\rangle$  is a cycle

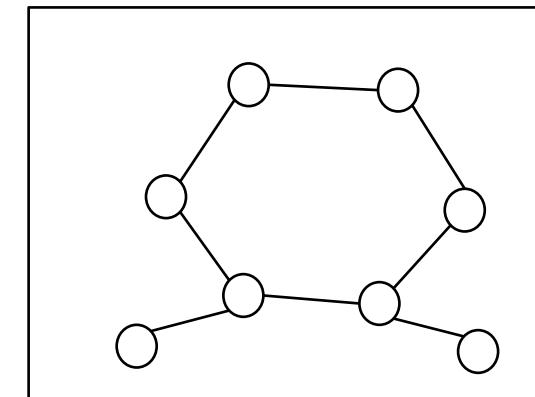
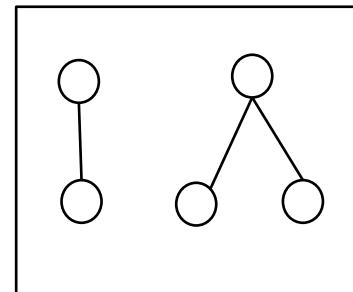
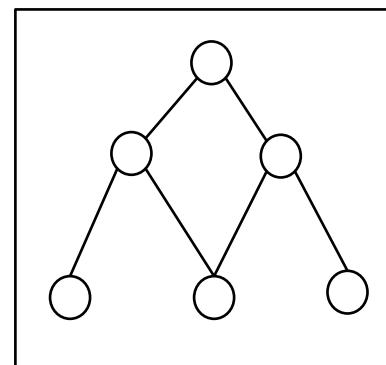
# Trees

A **tree** is a connected undirected simple graph with no cycles

trees:



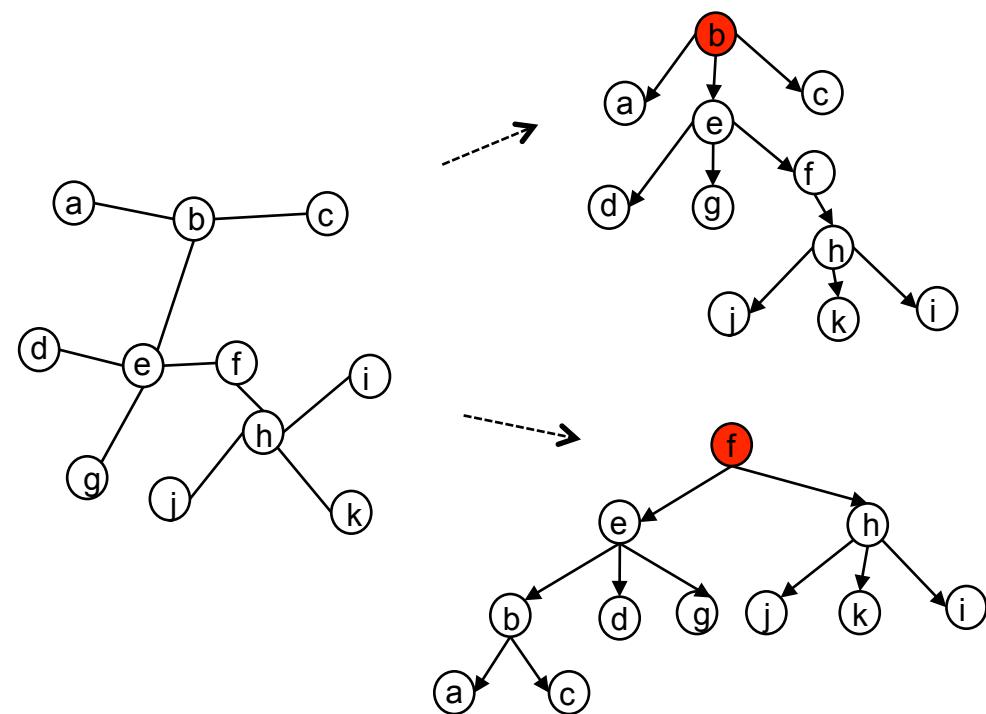
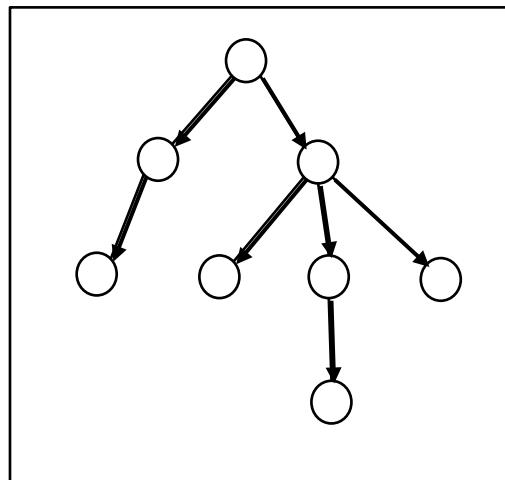
not  
trees:



# Rooted Trees

Usually, when we think of trees, we assume there is a root.

A **rooted tree** is a tree in which one of the vertices is designated the **root**, and all edges are then directed away from that root.



# Describing vertices in rooted trees

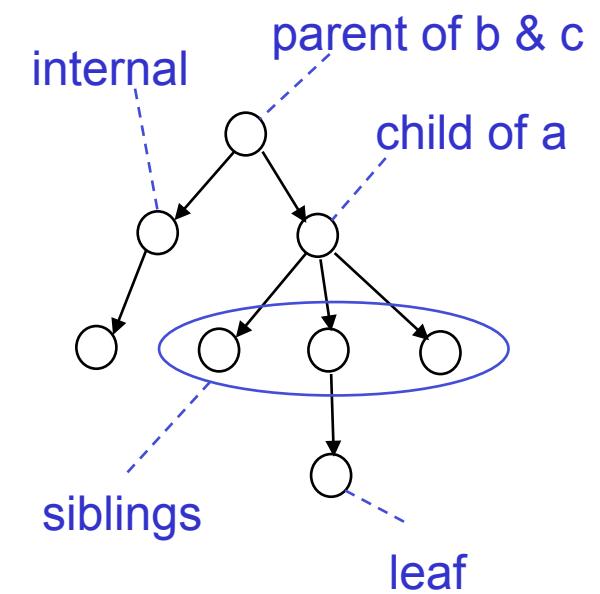
If there is a directed edge from  $x$  to  $y$ , then  $x$  is the **parent** of  $y$ , and  $y$  is a **child** of  $x$ .

If two vertices  $y$  and  $w$  have the same parent, then they are **siblings** of each other.

A vertex with no children is a **leaf**. A vertex with children is an **internal** vertex.

The **ancestors** of a vertex  $v$  are all the vertices in the path from  $v$  to the root (except for  $v$  itself).

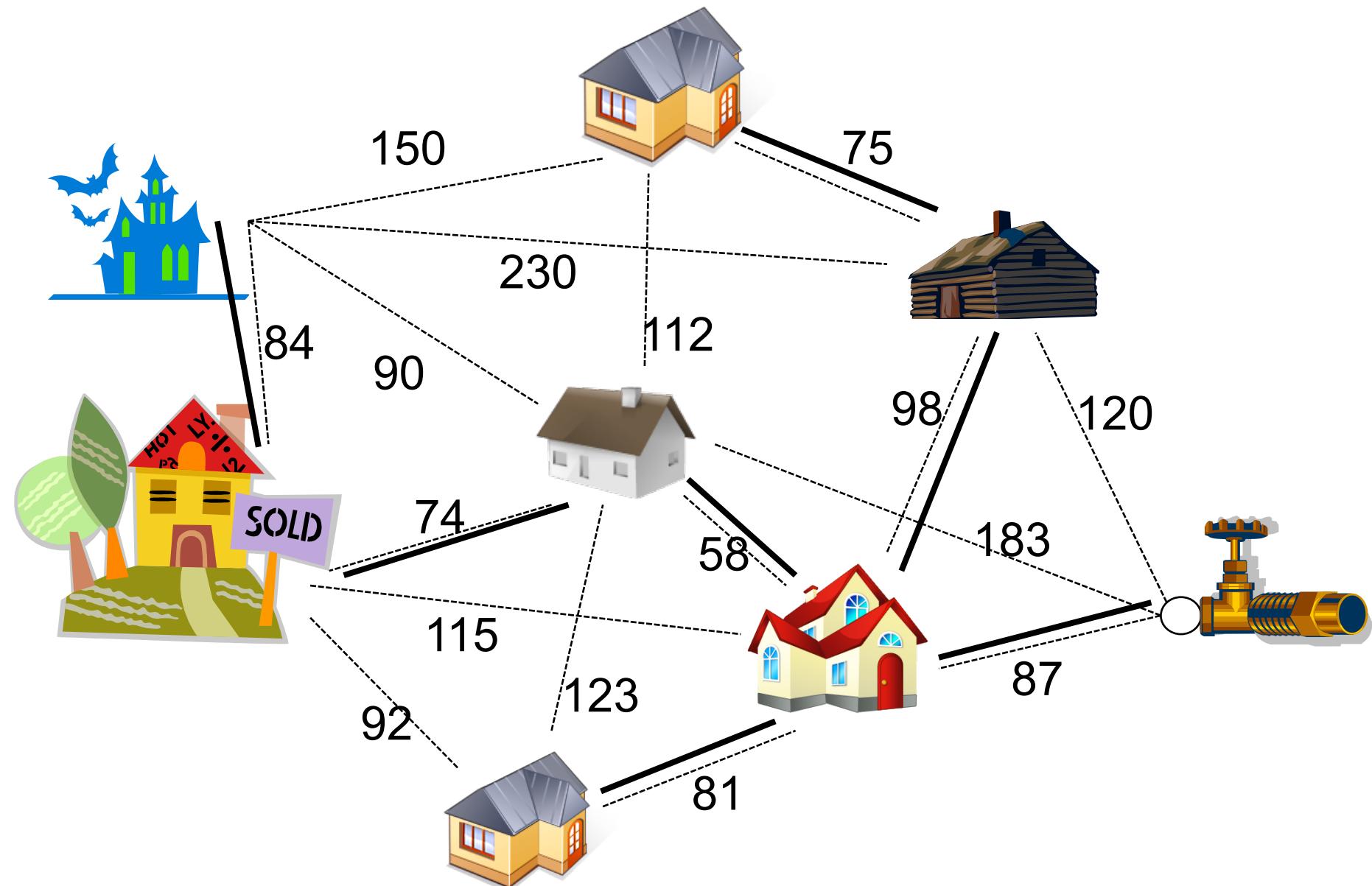
The **descendants** of a vertex  $v$  are all the vertices that have  $v$  as an ancestor.



## Properties of trees

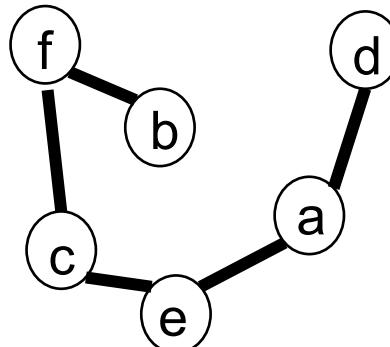
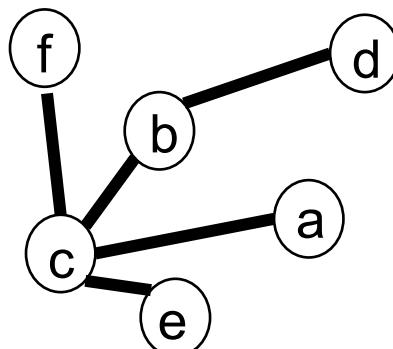
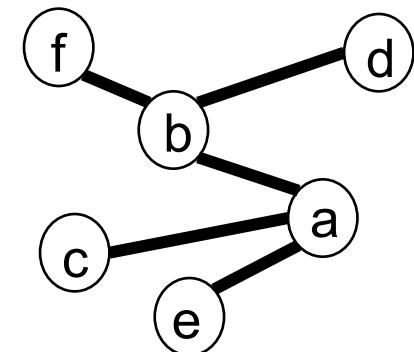
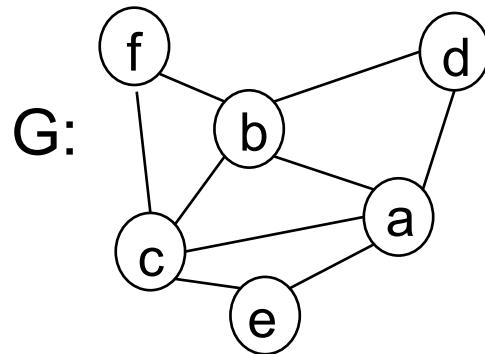
- between any pair of vertices, there is exactly one simple path
- adding a new edge to a tree creates a cycle
- removing any edge from a tree makes it a disconnected graph
- every rooted tree has at least one leaf
- in any tree with 2 or more vertices, there is at least one vertex with degree=1
- any tree with  $n$  vertices has exactly  $(n-1)$  edges

All of these statements can be proved easily ...



# Spanning trees

Let  $G=(V,E)$  be a simple graph. A **spanning tree** of  $G$  is a subgraph of  $G$  that contains every vertex in  $V$ .



Find 3 different spanning trees of  $G$

A simple graph is connected if and only if it has a spanning tree

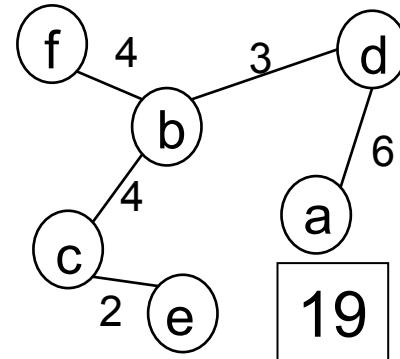
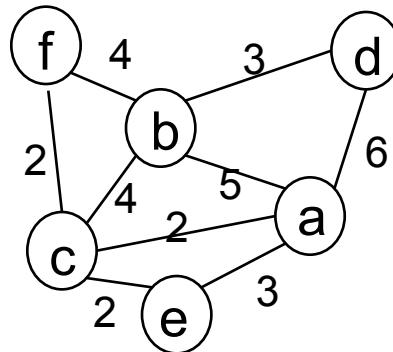
### Proof

First, suppose  $G=(V,E)$  is simple graph which has a spanning tree. The tree includes all vertices in  $V$ , and by definition, all the vertices are connected, so  $G$  must be connected.

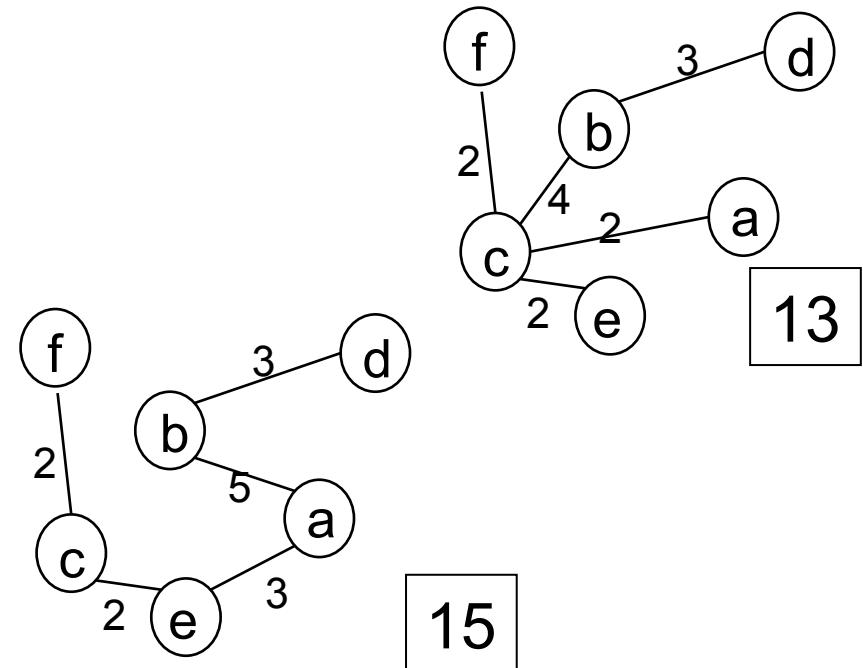
Second, suppose  $G=(V,E)$  is a connected simple graph. If  $G$  is already a tree, then  $G$  is its own spanning tree. Suppose now  $G$  is not a tree, and so it must have a cycle. We can remove any one of the edges in the cycle, and the resulting graph must still be connected and still contains all vertices. We keep repeating this process until the resulting graph has no cycles. At this point, we have a simple connected graph with no cycles, containing all vertices of  $G$  – i.e. a spanning tree of  $G$ .

# Minimal spanning trees

Consider a simple connected graph with weights on its edges. Different spanning trees will use different edges, and so the costs will be different.



19



15

13

How do we find the spanning tree with the minimum sum of weights?

## Informal algorithm

Pick any vertex. Pick the cheapest edge that links to a vertex we haven't picked yet. Now pick the cheapest edge that connects one of our chosen vertices to a vertex we haven't picked yet. Keep doing the same step until we have selected every vertex. The vertices and the edges that we have chosen give us a spanning tree.

At each stage, we are picking the cheapest way to connect a new vertex to the tree we are creating. This style of algorithm is called **greedy** – at each step, we greedily pick what seems to be the best option.

Note: we are NOT growing the tree out in a path from the root to the leaves – at each stage we add a new vertex and connect it anywhere in the previous tree that is cheapest

# Prim's Algorithm

## Algorithm: Prim's

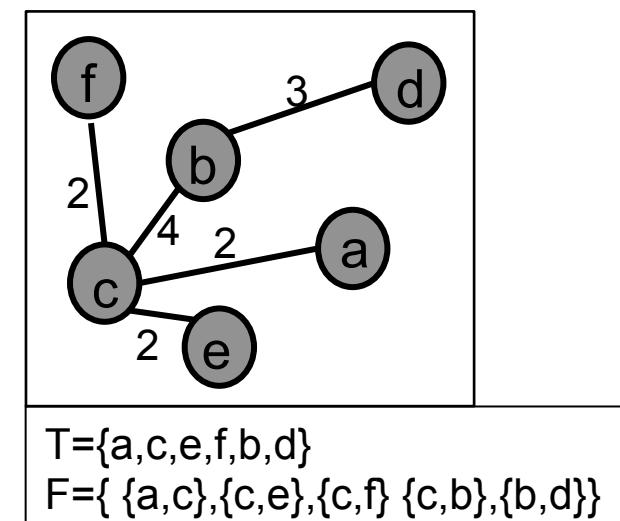
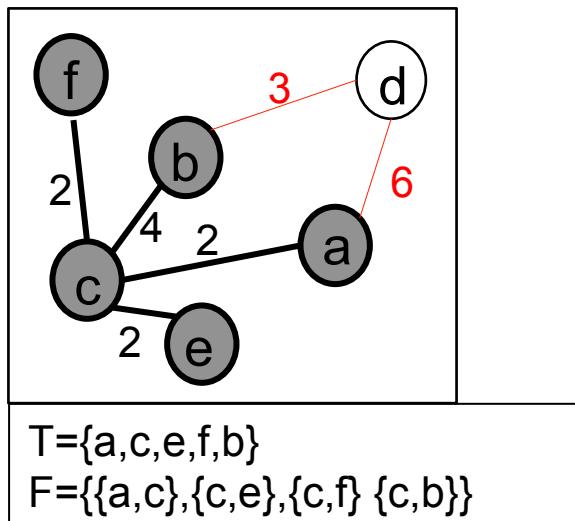
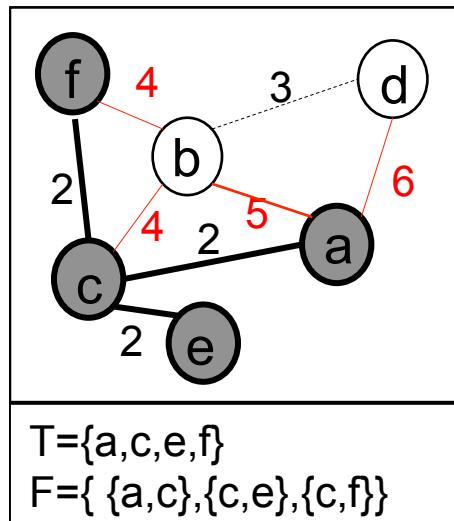
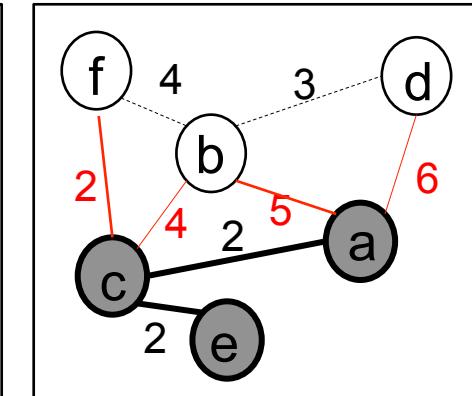
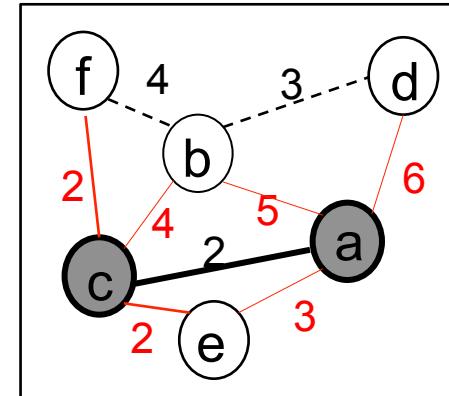
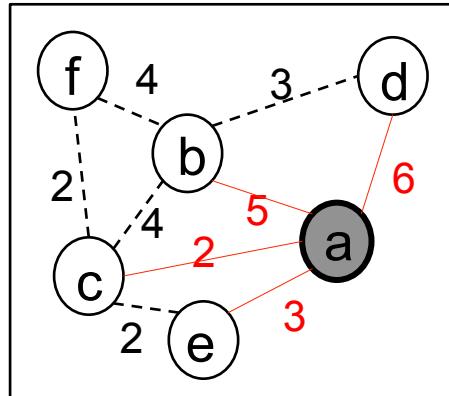
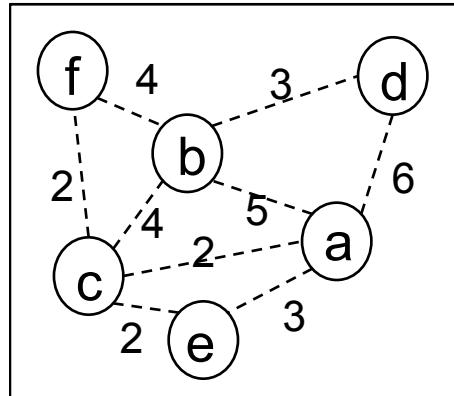
Input: connected undirected graph  $G = (V, E)$  with edge weights and  $n$  vertices

Output: a spanning tree  $T = (V, F)$  for  $G$

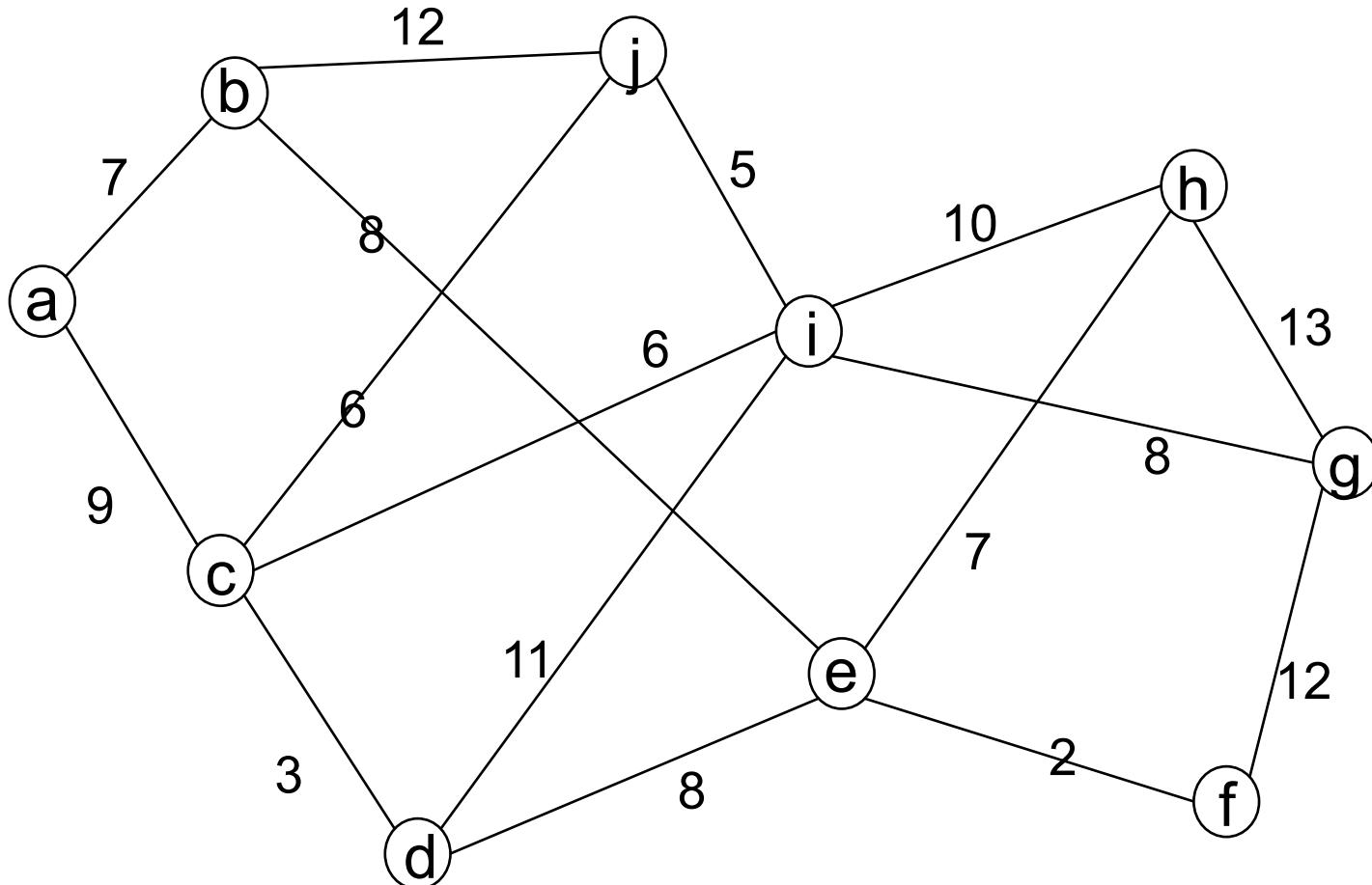
1.  $T := \{v\}$ , where  $v$  is any vertex in  $V$
  2.  $F := \{ \}$
  3. for each  $i$  from 2 to  $n$
  4.      $e := \{w, y\}$ , an edge with minimum weight in  $E$  such that  $w$  is in  $T$  and  $y$  is not in  $T$
  5.      $F := F \cup \{e\}$
  6.      $T := T \cup \{y\}$
  7. return  $(T, F)$
- step 4: find all edges that connect to vertices in  $T$ , but which would not create a cycle in the tree, and choose the one with lowest weight

It is possible to prove that Prim's algorithm does find the minimum spanning tree.

# Prim's algorithm: example



Exercise: find the minimum spanning tree using Prim's algorithm (start with vertex a)



Next lecture ...

Counting, and analysing algorithms



# CS1113

# Algorithm Analysis & Counting Rules

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Introduction to analysing algorithms and data structures

making one choice from multiple sets (the sum rule)

making multiple choices from multiple sets (the product rule)

# The story so far

- fundamental concept of computing: carrying out sequences of instructions
- 2<sup>nd</sup> most important concept: representing and manipulating information
- main challenge has been stating what you want, precisely and unambiguously

# Review

We have looked at:

- the basic idea of an algorithm – an ordered deterministic executable terminating set of instructions
- how to describe collections of objects (sets)
- how to represent transformations of collections (functions)
- how to describe relationships between objects in the collections (relations)
- how to specify requirements (logic)
- how to prove solutions meet requirements, and that arguments are correct (logic)
- how to represent and use structured collections of objects (graphs)

and we have looked at examples in databases and specifications for computer systems and in path planning.

# Assessing algorithms

In other modules, you are learning how to write algorithms in particular languages, how to represent information, and how to understand computer systems.

To judge how good our solutions are, we need to assess them:

- are our instructions correct?
- how long will it take to carry out the instructions?
- how much space do I need to represent this information?
- how many cases do I have to consider to solve a problem?
- what are good algorithms for different types of relations?
- how do I prove any claims I make?

We will start by learning how to count steps, choices and combinations.

## Example: making a single choice

Suppose I want to pick a single student representative for all full-time registered undergraduate computer science students at UCC. Let's say there are 94 students registered for 1st year, 98 students registered for 2nd year, 76 students registered for third year, and 60 students registered for 4th year. Nobody is registered for more than one year (and everybody is registered in at least one of the four years).

How many different possible choices are there?

We simply add together the total number in each category.

$$94 + 98 + 76 + 60 = 328.$$

## The Sum Rule

In general, if I have to select an object from  $n$  sets  $S_1, S_2, \dots, S_n$ , where none of the sets have any elements in common, then there are  $|S_1| + |S_2| + \dots + |S_n|$  possible selections.

This is called the *sum rule*.

Expressing it in terms of set operations, we have:

For sets  $S_1, S_2, \dots, S_n$  such that  $\forall i \forall j \neq i S_i \cap S_j = \{ \}$ , then  
 $|S_1 \cup S_2 \cup \dots \cup S_n| = |S_1| + |S_2| + \dots + |S_n|$

# Sum Rule and Computational Steps

Suppose I must complete all the actions from one set  $S_1$ , then all actions from another set  $S_2$ , and so on up to  $S_n$ , where all the actions are different. If each action takes 1 second, then the total computation will take  $|S_1| + |S_2| + \dots + |S_n|$  seconds.

Example: initialising a university records database  
for each student  $X_1$  to  $X_n$

- add the student to the database
- for each lecturer  $L_1$  to  $L_m$ 
  - add the lecturer to the database
- for each employee  $E_1$  to  $E_t$ 
  - add the employee to the database

How many database additions do I need to make?

## Example: making multiple choices

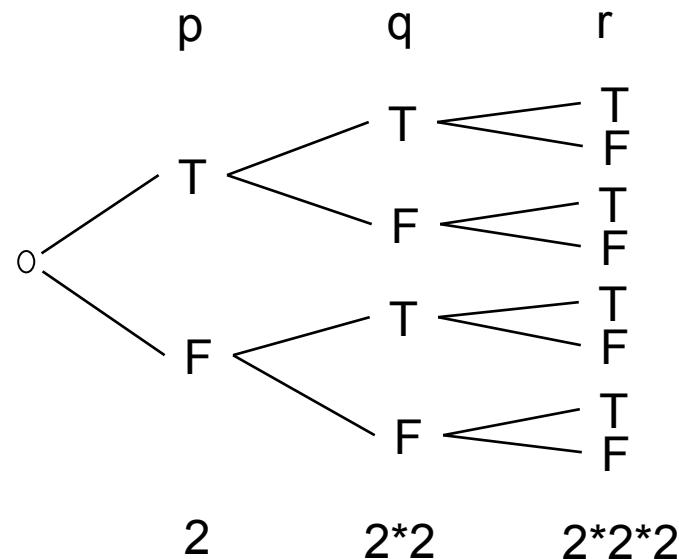
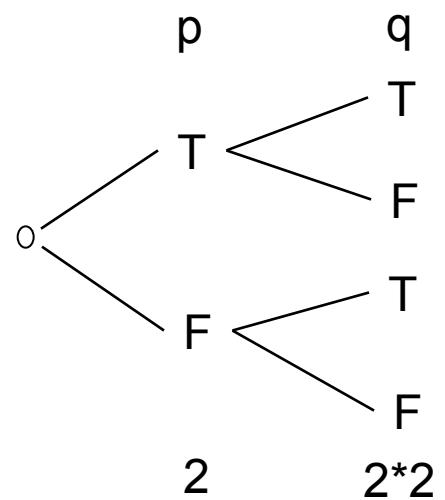
When building a truth table in propositional logic, how many rows do we need?

We have to consider all possible assignments of values to the variables. For each variable, there are only two possible values (T or F).

If the statement is  $p \wedge q$ , then there are two variables. There are two choices for the first variable, and for each of those, there are two choices for the second. Therefore there are  $2 * 2 = 4$  choices in total.

For  $p \wedge q \wedge r$ , there are three variables. So 2 choices for  $p$ , and for each of them, 2 choices for  $q$ , and for each of them, 2 choices for  $r$ . This gives  $2 * 2 * 2 = 8$  choices in total.

We can visualise this by drawing a tree of choices



p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	r	$p \wedge q$	$(p \wedge q) \vee r$
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

## Example

Suppose the department is considering a new system for assigning usernames to students. Each student will be given a two-letter name, consisting of lower-case characters only. How many usernames will this system allow?

There are 26 lower-case letters. Therefore, there are 26 ways to assign the first letter, and for each of these, there are 26 ways to assign the second letter. That means there are  $26 \times 26 = 676$  possible usernames.

## The Product Rule

In general, if I have to select one object from set A and one object from set B, where A has  $n$  elements and B has  $m$  elements, then there are  $n*m$  possible combined selections.

This is called the *product rule*:  $|A \times B| = |A| * |B|$

In general, if I have to select one object from each of the sets  $S_1, S_2, \dots, S_t$ , where set  $S_i$  has  $n_i$  elements, then there are  $n_1 * n_2 * \dots * n_t$  possible combined selections

$$|S_1 \times S_2 \times \dots \times S_t| = |S_1| \times |S_2| \times \dots \times |S_t|$$

We have seen this before: the size of the cartesian product

## Example

The standard system architecture for PCs uses bytes consisting of 8 bits. That is, a byte is a sequence of 8 binary digits (i.e. 0s or 1s). How many different bytes are possible?

There are 2 choices for the first bit, and for each of those, 2 choices for the second, etc.,  
so there are  $2 \times 2 = 2^8 = 256$ .

In 16-bit arithmetic, for representing integers, we use the first bit for the sign, which leaves 15 bits. That means we can represent  $2^{15}$  positive integers. Since the first integer is 0, that means the largest positive integer we can represent is  $2^{15}-1$ .

So "maxint" = 32767

## Example 1

Suppose the computer system requires you to choose a password which consists of a lowercase letter, a digit, a digit, and a lowercase letter, in that order. How many possible passwords are there?

## Example 2

Suppose we have two sets – A has 6 elements, and B has 5 elements. How many different functions can we specify from A to B?

## Example 3

Suppose we have a set of 5 elements. How many possible subsets are there? In general, for any given set, how many possible subsets are there?

## Example: counting actions in a program

```
for each student with id ranging from 1 to n
    obtain student with id=i from the database
    for each project with id ranging from 1 to m
        obtain project with id=j from the database
        if student_i worked on project j
            output (student_i, project_j) to the screen
```

How many times do I read details from the database?

If reading details from a database is a time-consuming operation, is there a better way of writing this algorithm?

# Next lecture ...

## More on counting combinations



# CS1113

## More Counting Rules

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# More principles for counting actions or choices

combining sum and product rules  
inclusion and exclusion  
factorials  
permutations

pigeonhole principle

## Example: combining sum and product

Module codes across University College Skibbereen must start with a department code of either 1 or 2 uppercase letters, and then have 3 digits, where the first digit must be either a 1, 2, 3 or 4. How many different module codes can be represented?

## Example (sum rule revision)

"Module codes across University College Skibbereen must start with a department code of either 1 or 2 uppercase letters, and ...."

How many different department codes can be represented?

Sum rule:

$$\forall i \forall j \text{ s.t. } j \neq i \text{ and } S_i \cap S_j = \{\},$$
$$|S_1 \cup S_2 \cup \dots \cup S_n| = |S_1| + |S_2| + \dots + |S_n|$$

There are 26 single letters, and  $26 \times 26$  pairs of letters (using the product rule), so there are  $26 + 676 = 702$  possible letter codes

## Example (product rule revision)

" ... and then have 3 digits, where the first digit must be either a 1, 2, 3 or 4. "

How many different module codes can be represented for a department?

Product rule:  $|S_1 \times S_2 \times \dots \times S_t| = |S_1| \times |S_2| \times \dots \times |S_t|$

There are 4 choices for the 1<sup>st</sup> digit. There are 10 choices for the 2<sup>nd</sup> digit, and 10 choices for the 3<sup>rd</sup> digit.

Therefore, there are  $4 \times 10 \times 10 = 400$  possible module codes.

## Example: combining sum and product

Module codes across University College Skibbereen must start with a department code of either 1 or 2 uppercase letters, and then have 3 digits, where the first digit must be either a 1, 2, 3 or 4. How many different module codes can be represented?

We have already seen by the sum rule that there are 702 possible department codes.

We have already seen by the product rule that there are 400 possible numerical codes for each dept.

So, by the product rule, there are  $702 \times 400 = 280800$  possible module codes in total.

## Example

How many UCC students have a 1st year computer science lecture on a Wednesday?

There are lectures for CS1111 and CS1113 on Wednesday, but for no other module. We have to count the number of students across 2 sets, so it looks like the sum rule.

BUT: there are 93 students registered for CS1111, 89 students registered for CS1113, and 85 students are registered for both modules.

So 93 students have a CS1111 lecture, and 89 students have a CS1113 lecture, giving  $93+89= 182$  entries on the sign-up sheets. But 85 of the students appear in both, so the total number of individual students is  $93+89-85 = 97$ .

# Principle of Inclusion and Exclusion

If I have to select one object from two sets  $A$  and  $B$ , then there are  $|A| + |B| - |A \cap B|$  possible selections.

$$\text{For sets } A \text{ and } B, |A \cup B| = |A| + |B| - |A \cap B|$$

If I have to select one object from 3 sets  $A$ ,  $B$  and  $C$ , then there are  $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$  possible selections.

For sets  $A, B$  and  $C$ ,

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

There is a rule for arbitrarily many sets, but we will leave that until later.

# Factorial

As our analysis of choices and algorithms gets more complex, we will need to introduce abbreviations for commonly occurring patterns.

The **factorial** of a number  $n$  is the multiplication of all the integers from  $n$  down to 1 and is written  $n!$   
(and so  $n! = n*(n-1)*(n-2)*...*2*1$ )

$$3! = 3*2*1 = 6$$

$$4! = 4*3*2*1 = 4*(3!) = 24$$

$$5! = 5*4*3*2*1 = 5*4! = 120$$

Exercise: work out the value of the following:

- (i) 2!
- (ii) 6!

# Example

Suppose we have three towns we want to visit: Tralee, Limerick and Waterford. The order in which we do the visits matters (travel time, etc). We will travel by the main roads connecting the towns, but we want to find the most efficient sequence. How many different routes do we need to consider before we are sure we have considered them all?

There are 6 possible routes:

- <Tralee, Limerick, Waterford>
- <Tralee, Waterford, Limerick>
- <Limerick, Tralee, Waterford>
- <Limerick, Waterford, Tralee>
- <Waterford, Tralee, Limerick>
- <Waterford, Limerick, Tralee>



## Calculating the number of routes

For the previous example, we could have calculated the number of routes easily, without writing them all out, using the product rule.

There are three possible choices for the first town. Once we have fixed the first, there are two remaining choices for the second. Once we have fixed the second, there is only one town remaining. So by the product rule, there are  $3 \times 2 \times 1$  possible sequences.

Using the factorial notation, there are  $3!$  possible sequences.

# Permutations

If we have a set of  $n$  elements, then a sequence of those  $n$  elements (where each element appears exactly once) is a **permutation** of the set.

For a set with  $n$  elements, there are  $n!$  different possible permutations.

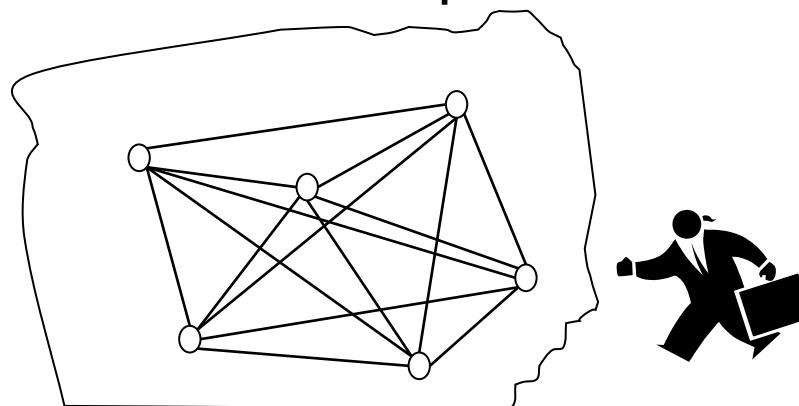
There are  $n$  choices for the first element,  $n-1$  for the second,  $n-2$  for the third, and so on, down to 1 choice for the last.

By the product rule, there are  $n*(n-1)*(n-2)*...*1 = n!$  possible permutations.

# Travelling Salesman problem

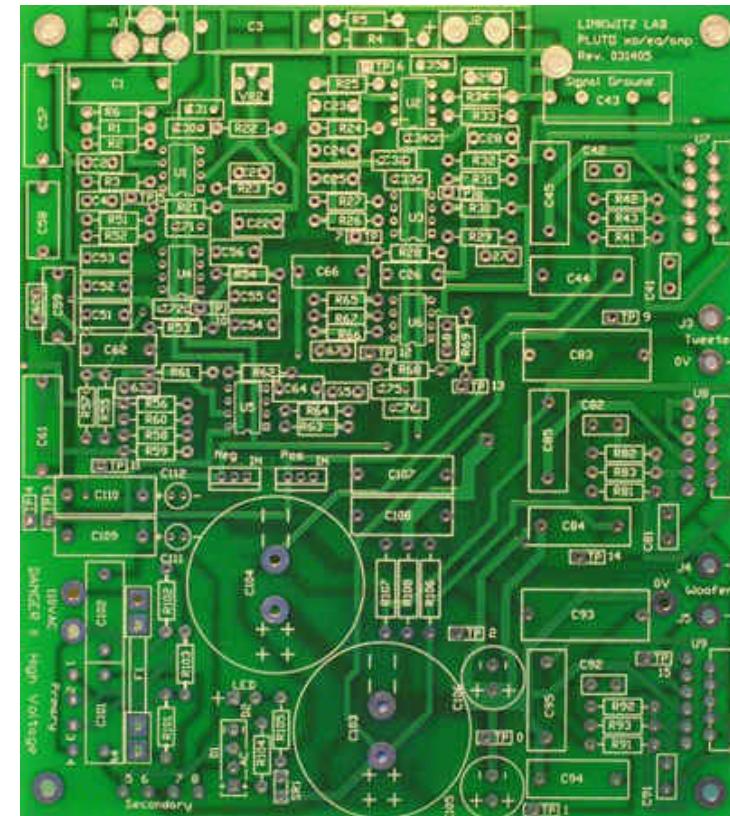
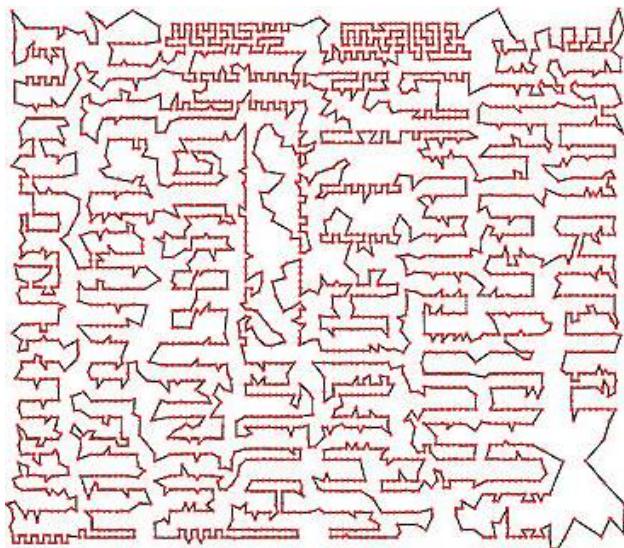
The example of finding routes is part of an important problem that reappears through computing, known as the **travelling salesman problem**.

Given a set of cities, and a table showing the distance between any pair of cities, find the shortest route around the cities which does not visit any city twice (except the starting point). In other words, find the permutation which gives the shortest distance (or find the cheapest Hamiltonian Circuit).



# Travelling Salesman Problem: Application

In the manufacture of printed circuit boards, each hole has to be drilled or stamped into the board, and each component fixed in place. What is the shortest path for the drill to move around the board?



# How hard can it be to solve the TSP?

The obvious way to solve it is to generate each possible permutation, and see how expensive each one is.

But the number of permutations grows very quickly as the number of cities in the problem increases

Suppose we have 20 cities in the problem.

There are then  $20!$  permutations.

$20! \sim 10^{18}$ .

There have only been  $\sim 10^{17}$  seconds since the Big Bang.

Understanding how long an algorithm will take to run is an important part of software development.

# TSP

The travelling salesman problem (TSP) has no known efficient algorithm – that is, as the number of cities increases, the time taken to find the best permutation increases exponentially, even for the best known algorithm. What is more, almost all computer scientists believe no efficient algorithm is possible.

There are many other problems that we can prove are similar to the TSP, and we can prove that if we find an efficient algorithm for one of them, we can do it for all of them.

Understanding the limits of computational power is important.

You will study these issues in 2<sup>nd</sup> year and 3<sup>rd</sup> year.

## Example: sub-permutations

Suppose a web site wants to conduct a user survey. Each customer is asked to select the top four photographs from the site. There are 30 photographs in total. How many different responses could there be?

Each user has 30 options for the top photograph. For each one of those, there are 29 choices for the 2<sup>nd</sup> photo, then 28 choices for the 3<sup>rd</sup>, and 27 choices for the 4<sup>th</sup>.

So there are  $30 \times 29 \times 28 \times 27 = 657720$  possible responses.

## r-Permutations

An **r-permutation** is an ordered sequence of  $r$  different elements from a set with  $n$  elements, where  $n \geq r$ .

For a set with  $n$  elements, there are  $n*(n-1)*\dots*(n-r+1)$  possible r-permutations.

Sometimes, this will be written  ${}^n P_r$

Exercise: what is the value of the following:

- (i)  ${}^4 P_2$
- (ii)  ${}^{10} P_7$

Exercise: in a race with 7 runners, how many possible finishing orders are there for 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup>?

# Computing r-permutations

It is normal to express the number of r-permutations using factorials:

$${}^n P_r = n! / (n-r)!$$

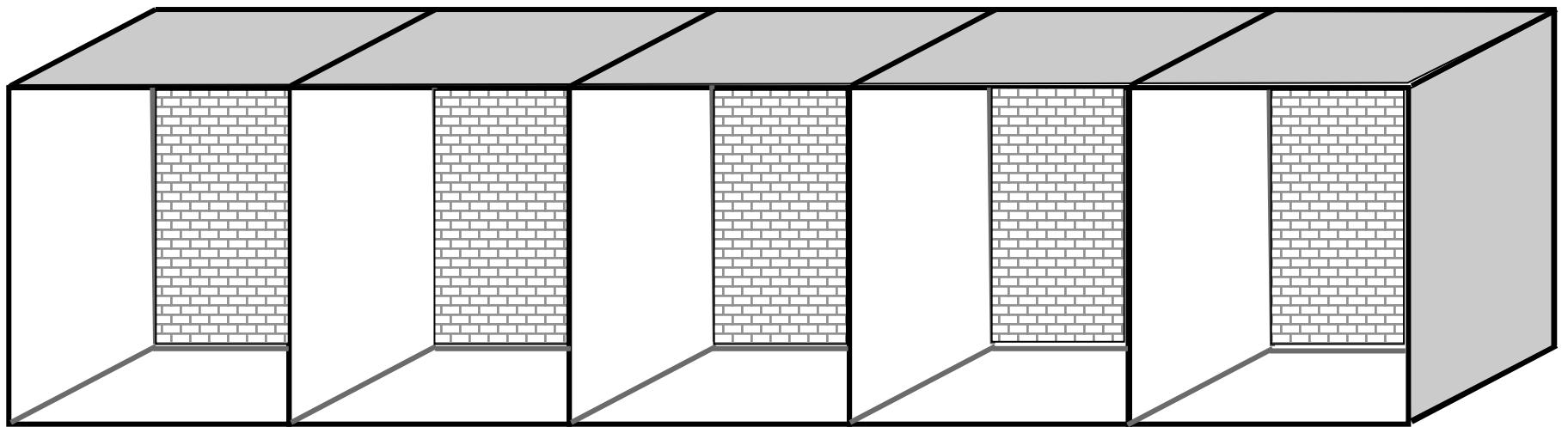
We can do this because

$$n! = n * (n-1) * (n-2) * \dots * (n-r+1) * (n-r) * (n-r-1) * \dots * 2 * 1$$

and so

$$n! / (n-r)! = n * (n-1) * (n-2) * \dots * (n-r+1) = {}^n P_r$$

$$\begin{aligned} \text{E.g. } 9! / 4! &= 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 / 4 * 3 * 2 * 1 \\ &= 9 * 8 * 7 * 6 * 5 \\ &= 9 * (9-1) * \dots * (9-4+1) \end{aligned}$$



# The pigeonhole principle

Suppose I have 6 pigeons. Each pigeon sleeps in a box in my pigeonloft. I have only 5 boxes. Therefore, at least one box contains 2 or more pigeons.

This reasoning gives us the **pigeonhole principle** (which is surprisingly useful in analysing algorithms):

If I want to place  $k+1$  objects into  $k$  boxes, and  $k$  is a positive integer, then at least 1 box must have 2 or more objects.

If  $N$  objects are placed into  $k$  boxes, then at least 1 box has at least  $\text{ceil}(N/k)$  objects (i.e. the first integer bigger than or equal to  $N/k$ )

## Example: assigning jobs to processors

One of the main tasks in managing computational resources is load balancing – ensuring that the workload is spread as evenly as possible over the processors for maximum efficiency.

Suppose we have 6 processors, and 20 jobs. Suppose the minimum running time for any job is 5 seconds. I need to return the results of all jobs within 15 seconds. Can I do it?

By the pigeonhole principle, at least one processor must have  $\lceil 20/6 \rceil$  jobs scheduled on it. So at least one processor must have at least 4 jobs scheduled. The best we can hope for is that each of those 4 jobs takes the minimum time to run of 5 seconds. So that processor will take at least  $4*5=20$  seconds to return all results. Therefore we cannot complete the task.

## Next lecture ...

combinations where the order doesn't matter



# CS1113

## Counting Combinations

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Counting Combinations

counting the number of choices we have in different situations  
(and what you should know when you play the lottery)

## Example: picking project teams

Suppose we have a group of five software engineers, {alice, bob, carol, darragh, eileen}, and we want to pick a team of three of them to work on a project.

How many different possible teams are there?

By the permutation rule, there are  ${}^5P_3 = 5!/(5-3)! = 5!/2! = (5*4*3*2*1)/(2*1) = 5*4*3 = 60$  different *permutations* of 3 people from a group of 5.

But many of these teams will be the same: e.g.  
<alice,bob,carol> is the same team as <carol,alice,bob>

So how many teams have we double counted?

## Example (continued)

By the permutation rule, each team of three people can be represented as  $3! = 6$  different permutations.

So out of our total of 60, for every permutation we keep, we need to throw away 5 duplicates.

So there are  $60/6 = 10$  unique teams.

{alice,bob,carol}	{alice,darragh,eileen}
{alice,bob,darragh}	{bob,carol,darragh}
{alice,bob,eileen}	{bob,carol,eileen}
{alice,carol,darragh}	{bob,darragh,eileen}
{alice,carol,eileen}	{carol,darragh,eileen}

## Unordered Combinations: "n choose r"

Often, when choosing some elements from a set, the order in which we list them does not matter.

To choose  $r$  elements from a set of size  $n$ , there are  ${}^n P_r = n!/(n-r)!$  permutations.

But if we ignore the order in which we list the elements, then each permutation is one of a set of  $r!$  duplicates. We only need to keep 1 out of every set of  $r!$ , and so we need to divide our total by  $r!$  to get the number of unordered sets.

There are  $n!/r!(n-r)!$  ways of choosing  $r$  elements from  $n$ .

We write  $n!/r!(n-r)!$  as either  ${}^n C_r$  or  $\binom{n}{r}$  and say " $n$  choose  $r$ ".

## Example: Poker hands

How many different poker hands are possible?



There are 52 cards in a pack. Each card is different. A standard poker hand has 5 cards. The order in which the cards are placed in the hand does not matter.

How many different ways can we choose 5 items from 52?

$$\binom{52}{5} = \frac{52!}{5!(52-5)!} = \frac{52!}{5!*47!} = \frac{52 * 51 * 50 * 49 * 48 * 47 * 46 * ... * 1}{(5 * 4 * 3 * 2 * 1) * (47 * 46 * ... * 1)} = \frac{52 * 51 * 50 * 49 * 48}{5 * 4 * 3 * 2 * 1} = 2598960$$

## Example: Robot route planning

In the simple grid below, a robot can move from one square into an adjacent square (but not diagonally). How many possible paths are there from A to B, if the robot never moves to a square that is further away from B than its current square?

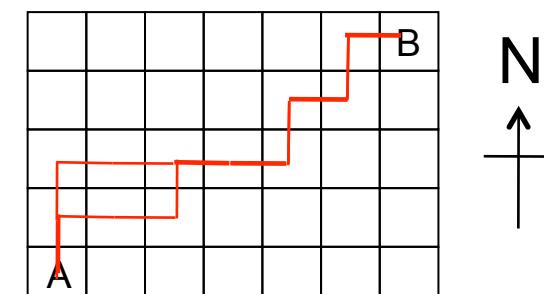
Each move is either N or E (and never S or W). There must be exactly 4 N steps and 6 E steps, and so 10 steps in total. So we can represent the route as an array of 10 directions. E.g.

route 1: NNEEEENENE

1234567890

route 2: NEENEENENE

1234567890



## robot route planning (continued)

route 1: NNEEEENENE  
1234567890

route 2: NEENEENENE  
1234567890

We can represent the routes by the positions of the four Ns:

route 1: 1-2-7-9                          route 2: 1-4-7-9

So two routes are different if the set of four positions for the Ns are different. Therefore, choosing a route is essentially choosing four numbers from  $\{0,1,2,\dots,9\}$ , so the number of ways of doing this is:

$$\binom{10}{4} = \frac{10!}{4!(10-4)!} = \frac{10!}{4!6!} = \frac{10 * 9 * 8 * 7}{4 * 3 * 2 * 1} = 210$$

# Computing choice combinations

Sometimes, you will need to compute combinations inside programs. Be careful! As we saw before, factorials can grow very quickly, producing numbers that are too large to be represented accurately.

For example,  ${}^{20}C_3 = 20!/3!*17!$

If you try to compute 20! first, you will need to store a value larger than  $10^{18}$ .

Instead, do the cancellation first:

$$\frac{20!}{3!*17!} = \frac{20 * 19 * 18 * 17 * 16 * 15 * ... * 1}{(3 * 2 * 1) * (17 * 16 * 15 * ... * 1)} = \frac{20 * 19 * 18}{3 * 2 * 1}$$

# Combinations and products

The rules for combinations, products and sums can be used together (but you need to think carefully about what you are doing).

Suppose we want to enter a mixed soccer team into a 5-a-side tournament. The tournament rules say we must have 2 women and 3 men. There are 5 women to choose from, and 10 men. How many possible teams are there?

We choose 2 women from 5, and for each of these sets, we can then choose 3 men from 10. Therefore, there are:

$$\binom{5}{2} * \binom{10}{3} = \frac{5!}{2!3!} * \frac{10!}{3!7!} = \frac{5 * 4}{2 * 1} * \frac{10 * 9 * 8}{3 * 2} = \frac{20}{2} * \frac{720}{6} = 10 * 120 = 1200$$

## Example: Poker hands (cont)

How many different *full house* poker hands are possible?

A full house consists of 3 cards of one rank, and 2 cards of another rank.



We must choose the rank of the 3-of-a-kind, then choose the 3 suits, then choose the rank of the pair, and choose the 2 suits. There are 13 possible ranks, and 4 possible suits.

$$\binom{13}{1} * \binom{4}{3} * \binom{12}{1} * \binom{4}{2} = \frac{13!}{1!(13-1)!} * \frac{4!}{3!(4-3)!} * \frac{12!}{1!(12-1)!} * \frac{4!}{2!(4-2)!} = 13 * 4 * 12 * \frac{4!}{2!*2!} == 3744$$

# Examples



In the national lottery (Lotto), each panel asks you to choose 6 numbers from 1 to 45. How many different 6-number entries can you make? What are your chances of winning a share in the jackpot?

Matching 3 numbers wins €5. How many different results (i.e. set of balls drawn in the lottery) would give you €5? What are your chances of winning €5?

# Equivalence

$$\binom{n}{r} = \binom{n}{n-r}$$

Proof:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n!}{(n-r)!r!} = \frac{n!}{(n-r)!(n-n+r)!} = \frac{n!}{(n-r)!(n-(n-r))!} = \binom{n}{n-r}$$

## Examples

You must select 6 staff for a project team. Regulations state that there must be more experienced staff on the team than junior staff. You have 8 experienced staff members, and 12 junior staff. How many ways could you form the team?

Exercise: how many bit strings of length 8 have at least two 1s and at least two 0s?

## more on assigning jobs to processors

Suppose now we have 3 processors, and 4 identical jobs. How many ways can we distribute the jobs?

Let the processors be called A, B and C.

If we put 1 job on A, 2 on B and 1 on C, we write ABBC, etc.

The full set of choices is:

AAAA	AABB	ABBB	ACCC	BBCC
AAAB	AABC	ABBC	BBBB	BCCC
AAAC	AACC	ABCC	BBCB	CCCC

which gives 15 choices in total.

## re-expressing the choices

We can adapt the way we write the choices to give us a general rule.

We can rewrite ABBC as  $*|**|*$  where we interpret the number of stars before the 1<sup>st</sup> "|" as being the number of jobs on processor A, then the number of stars between the 1<sup>st</sup> and 2<sup>nd</sup> "|" as being the jobs on B, etc..

We know there are exactly 4 jobs, so we have 4 stars, and 2 "|".  
Each different sequence gives a different distribution of jobs.  
e.g.  $***||^*$  means 3 jobs on A, 0 jobs on B, and 1 job on C.

We can now rewrite this as the sequence of positions in which a star appears.

So 1236 gives  $***||^*$  which is AAAC, while 1346 gives  $*|**|*$ , giving ABBC

So the number of different sets of star positions is the number of different ways we can distribute the jobs over the processors.

# number of ways to distribute 4 jobs over 3 processors

We have 4 jobs and 3 processors, and we represent one assignment by stating where the 4 stars appear in a sequence of 6 symbols.

So the number of ways of choosing 4 positions from 6 is the number of different assignments:

$$\binom{6}{4} = \frac{6!}{4!*2!} = \frac{6*5}{2*1} = 15 \text{ ways of distributing 4 jobs on 3 processors}$$

How did we get the values '6' and '4'?

There were 4 jobs (which gives 4 stars), and 3 processors needed to be separated by 2 "|". This gives  $4+2=6$  positions.

So we chose 4 from 6.

## Choosing $r$ items with repeats from $n$ types

If we must choose a set of  $r$  items where we have  $n$  types to choose from, and we can repeat some of the items,

we have  $\binom{r+n-1}{r}$  different ways of doing it.

Note: if we must choose a *permutation* of  $r$  items where we have  $n$  types to choose from, and we can repeat items, then we have  $n^r$  ways of doing it.

## Example

We all have to eat 5 portions of fruit or vegetables each day. Suppose we only eat 4 types: apples, oranges, bananas and carrots. How many different menus can we create?

# Combinations and permutations: summary

	repetitions?	Formula	Notation
#r-permutations	no	$\frac{n!}{(n-r)!}$	${}^n P_r$
	yes	$n^r$	
#r-combinations	no	$\frac{n!}{(n-r)!r!}$	${}^n C_r$ or $\binom{n}{r}$
	yes	$\frac{(r+n-1)!}{r!(n-1)!} = \binom{r+n-1}{r}$	

# Binomial Expansion

$$(x+y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \binom{n}{2}x^{n-2}y^2 + \binom{n}{3}x^{n-3}y^3 + \cdots + \binom{n}{n-2}x^{n-(n-2)}y^{n-2} + \binom{n}{n-1}x^{n-(n-1)}y^{n-1} + \binom{n}{n}y^n$$

$$(x+y)^2 = \binom{2}{0}x^2 + \binom{2}{1}xy + \binom{2}{2}y^2 = \frac{2!}{0!2!}x^2 + \frac{2!}{1!1!}xy + \frac{2!}{2!0!}y^2 = x^2 + 2xy + y^2$$

$$\begin{aligned}(x+y)^3 &= \binom{3}{0}x^3 + \binom{3}{1}x^2y + \binom{3}{2}xy^2 + \binom{3}{3}y^3 \\&= \frac{3!}{0!3!}x^3 + \frac{3!}{1!2!}x^2y + \frac{3!}{2!1!}xy^2 + \frac{3!}{3!0!}xy^3 \\&= x^3 + 3x^2y + 3xy^2 + y^3\end{aligned}$$

Next lecture ...

classifying algorithm runtime

# CS1113 Algorithm Complexity

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Classifying Algorithm Run-time

comparing linear and binary search

growth of functions

big-oh notation

proof techniques

# Linear search vs Binary search

Previously, we looked at two algorithms for searching a sequence of data values: linear search and binary search.

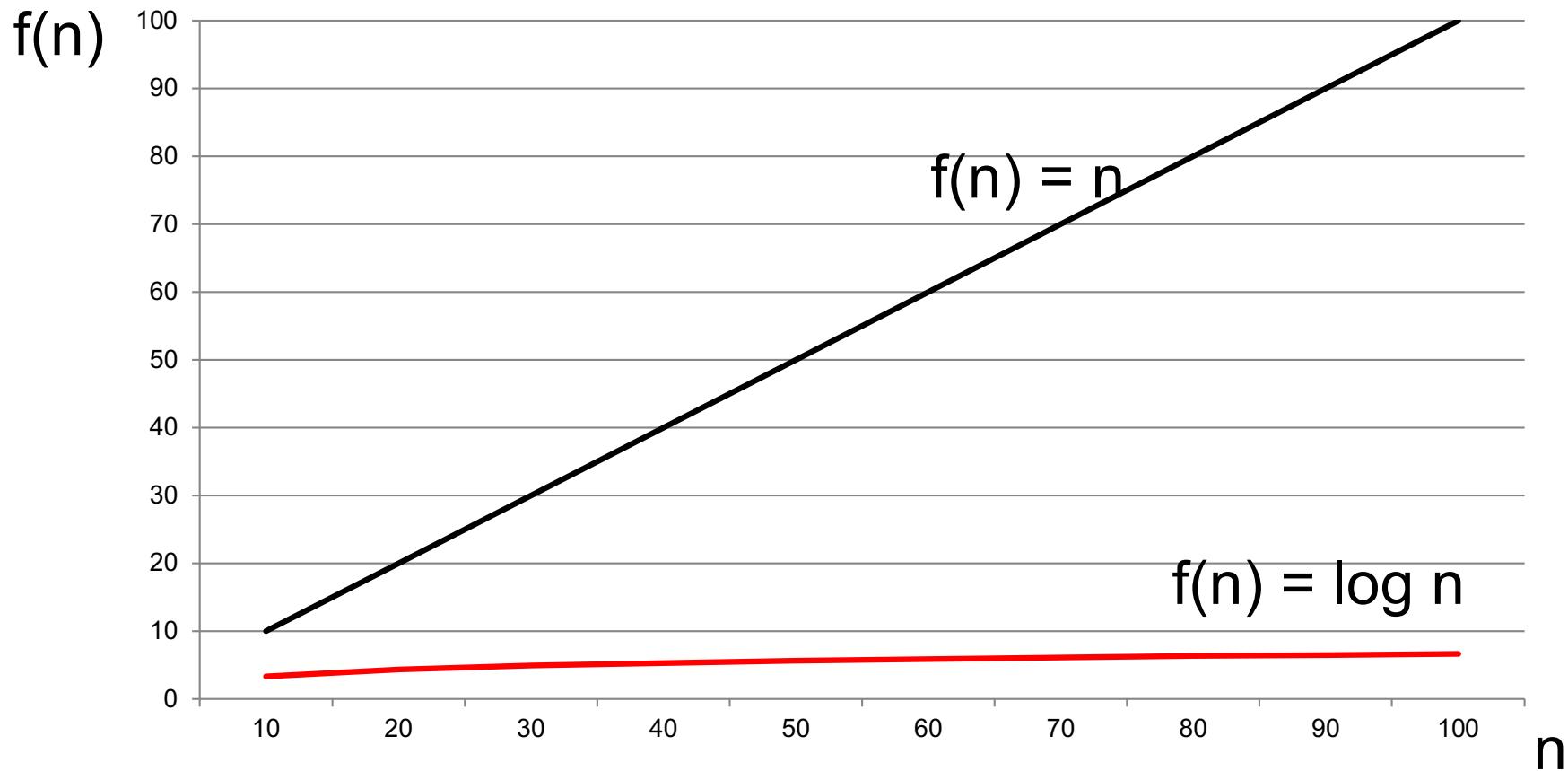
For each one, we worked out how much work it would have to do in the worst case. For a sequence of length  $n$ :

- linear search goes round its loop  $n$  times
- binary search goes round its loop  $\log_2 n$  times

and they each do a similar amount of work inside their loop.

But what does this mean in practice? Is the difference significant?

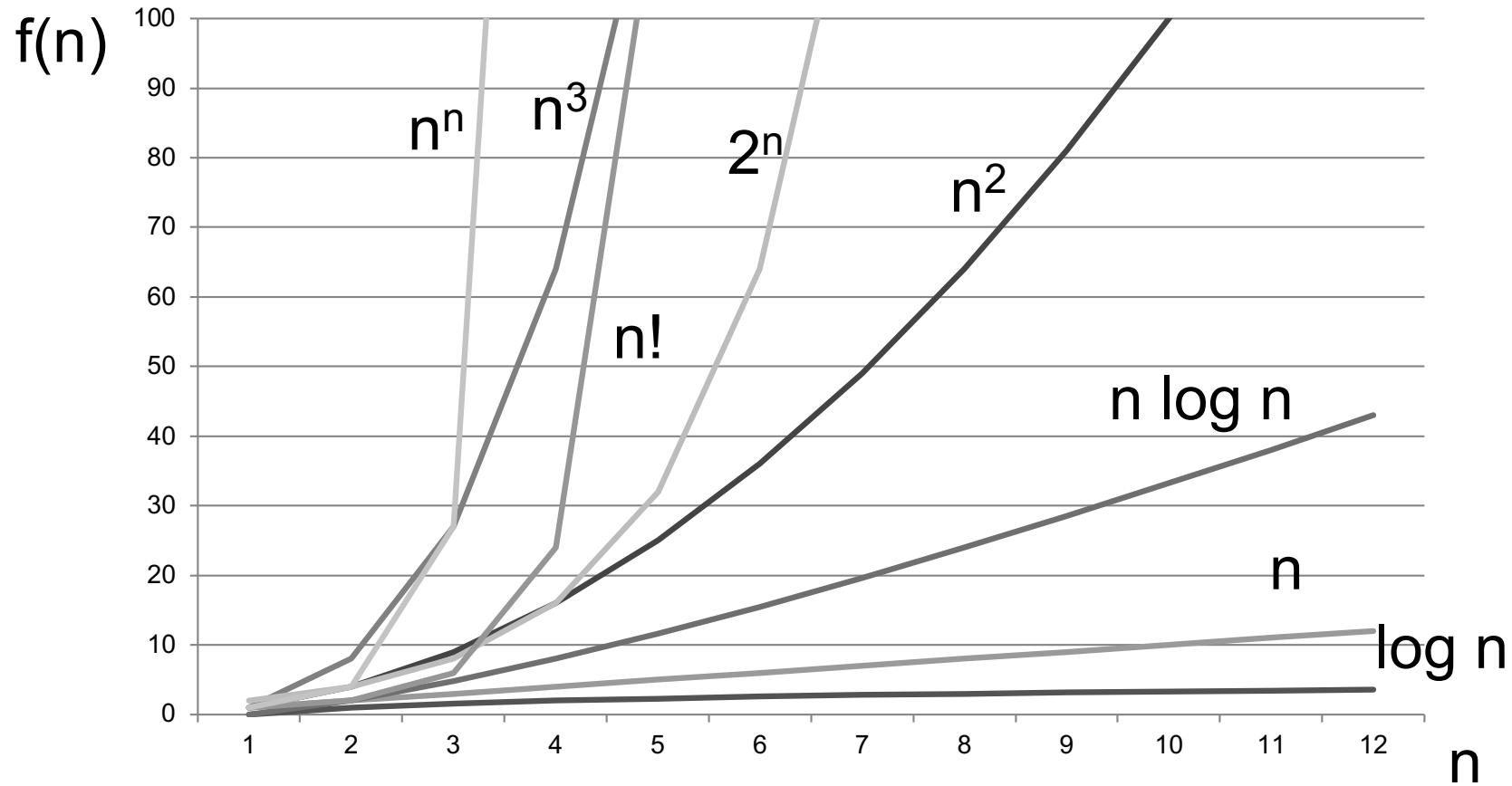
# Comparing $n$ and $\log n$



If the sequence has 100 elements, linear search may be 15 times slower.

If the sequence has 1000 elements, linear search may be 100 times slower.

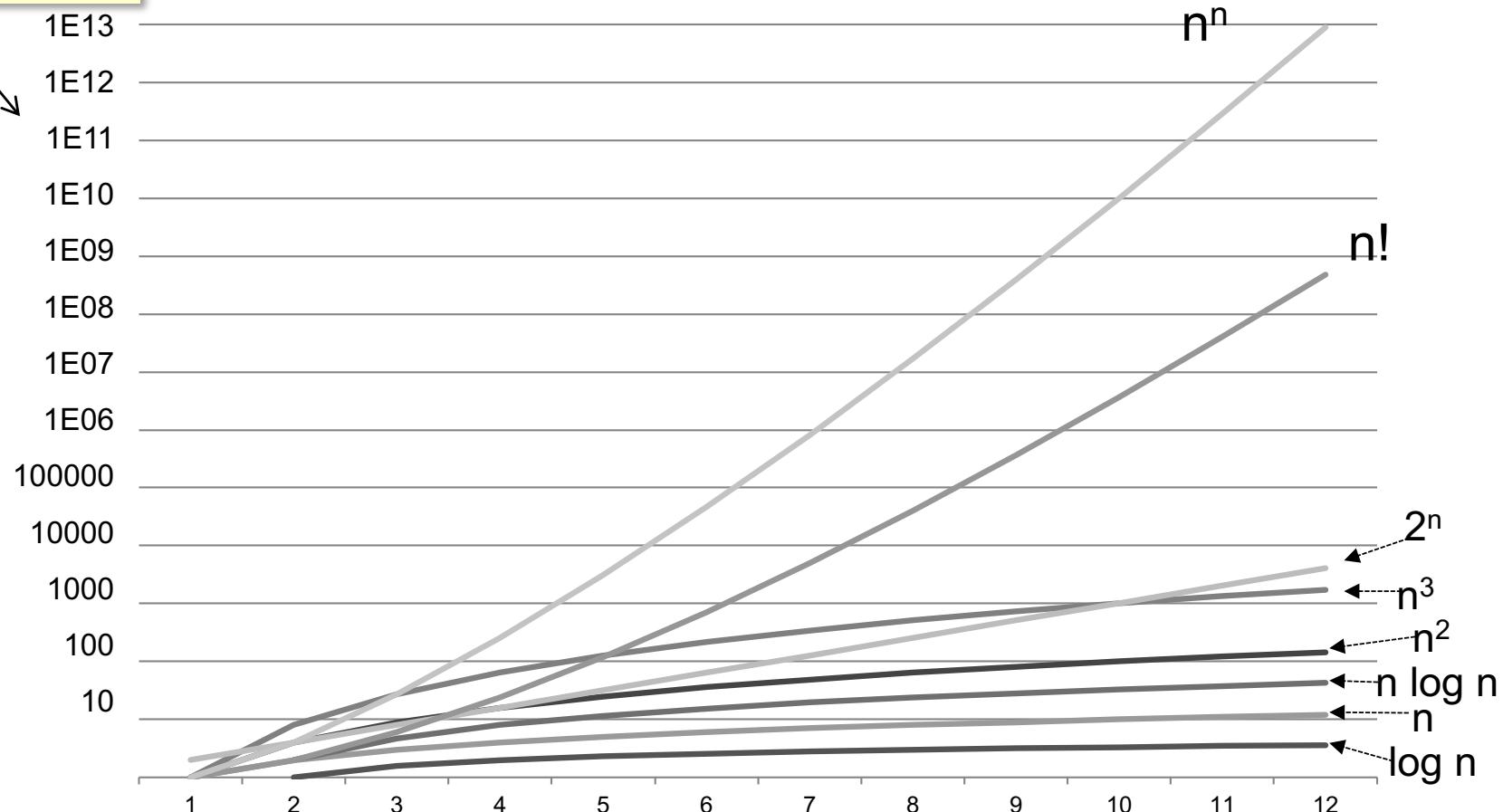
# Comparing other functions



Note:  $n^n$ ,  $n^3$ ,  $n!$  off the scale by  $n=5$ .

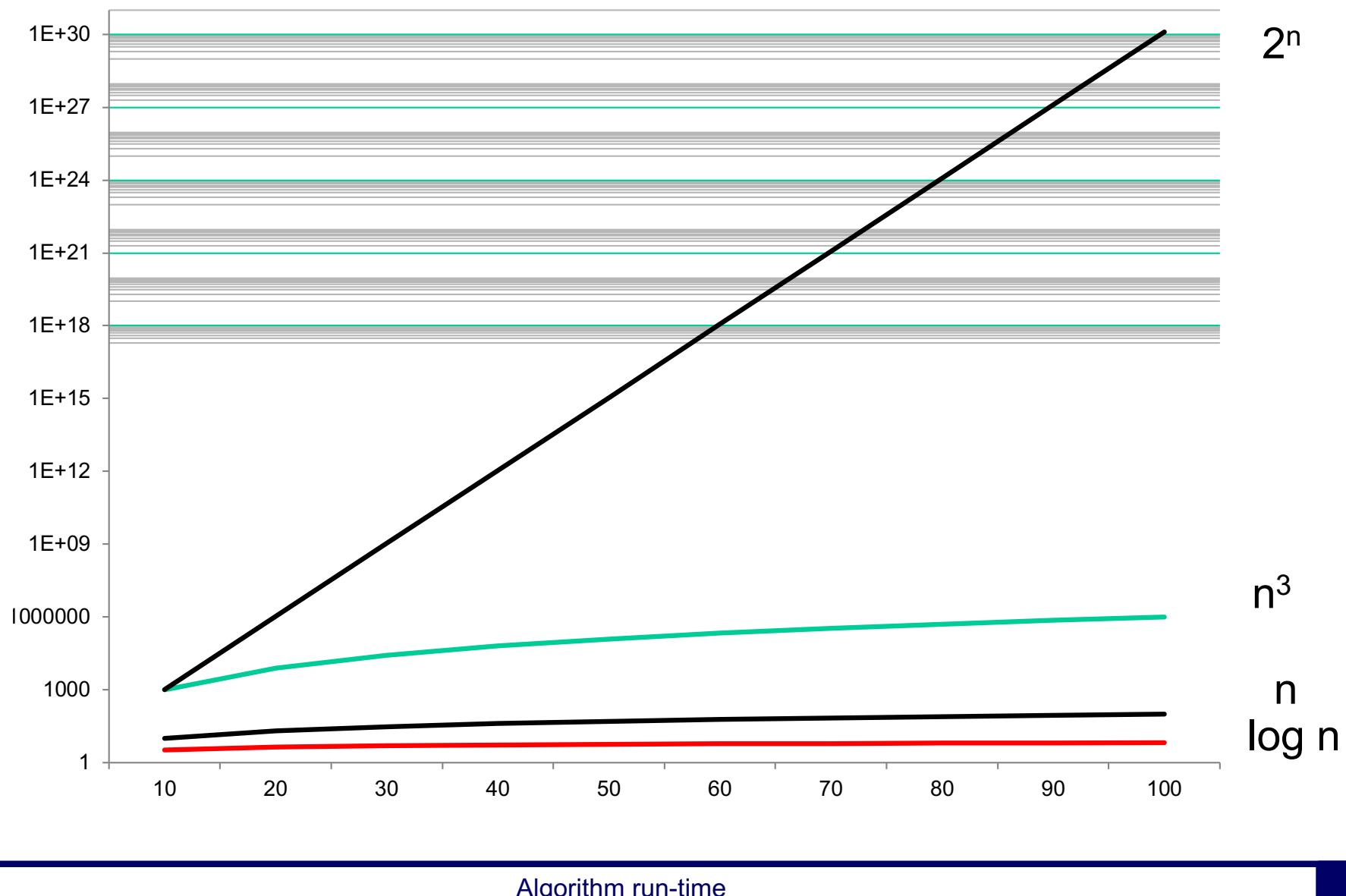
# comparing functions

Note the scale



Note: when we reach  $n=12$ ,  $n^3$  is now less than  $2^n$  and  $n!$

# Comparing functions for larger input



# Comparing functions (continued)

- As we extended out the graph, the difference between the functions became clearer and more consistent
- For larger values of  $n$ , the differences are significant
  - choosing an algorithm with a high run-time will make your program useless for large data sets
- Also note that we were only comparing simple functions of  $n$ 
  - we didn't look at, for example,  $n^2+n+3$ ,  $4n+1$ , or  $2n^2+5n$
- We only need a general picture of how quickly a function grows, and we use the simple functions as a reference
  - can we have any confidence in this comparison?

# Big-oh notation

Consider two functions  $f$  and  $g$ , mapping positive integers to positive integers ( so  $f : \mathbb{N} \rightarrow \mathbb{N}$  and  $g : \mathbb{N} \rightarrow \mathbb{N}$ )

We will say  $f(x)$  is  $O(g(x))$  if there are two constant values  $k$  and  $C$  so that whenever  $x$  is bigger than  $k$ ,  $f(x) \leq C * g(x)$ .

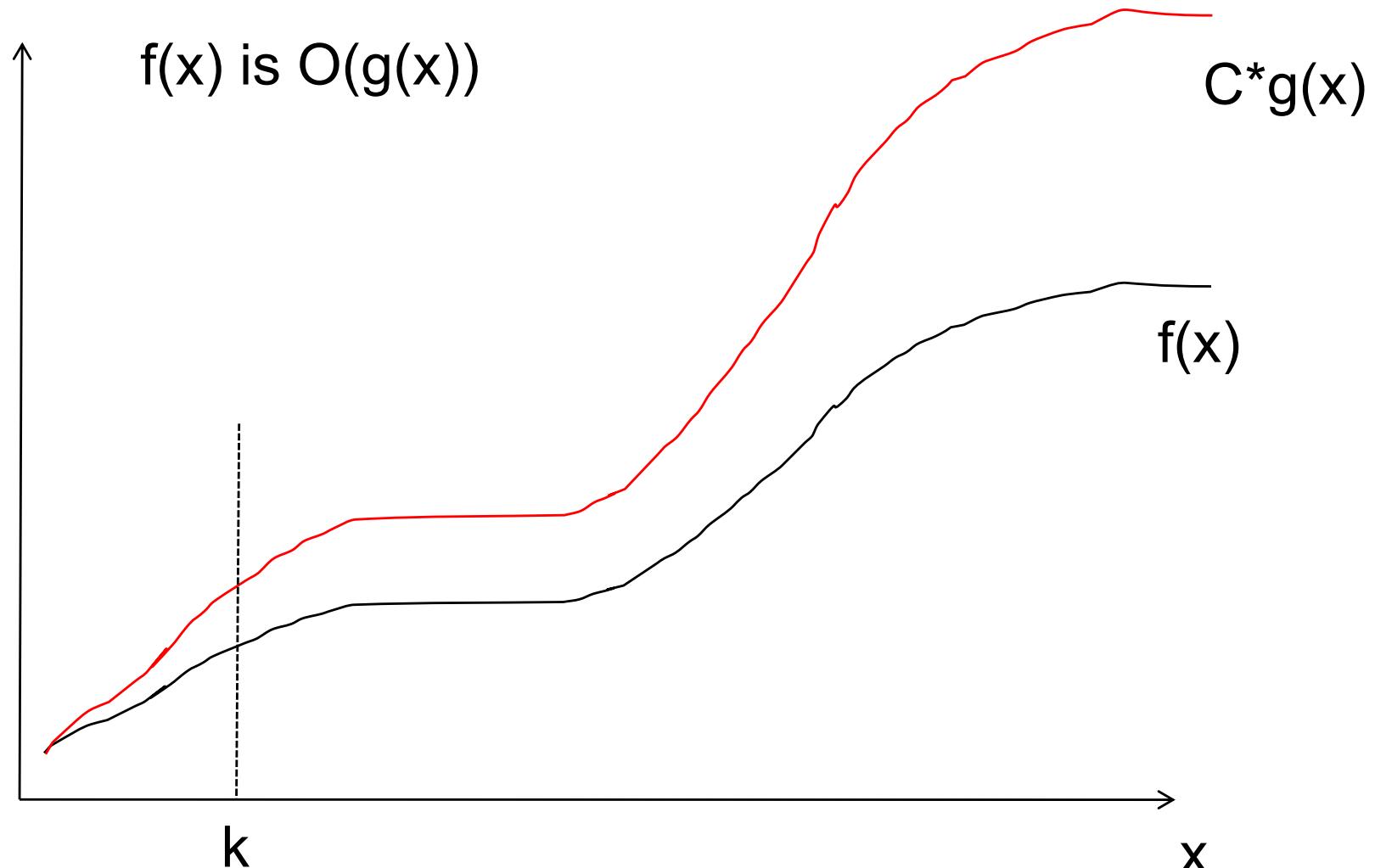
This means that when  $x$  is big enough,  $f(x)$  is never more than some constant multiple of  $g(x)$ , and so  $f(x)$  will not become drastically worse than  $g(x)$ .

We read this as " $f(x)$  is big-oh of  $g(x)$ "

Formally,

$f(x)$  is  $O(g(x))$  if and only if  $\exists k \exists C \forall x > k f(x) \leq C * g(x)$

# What does big-oh mean for function growth?



For all values of  $x > k$ ,  $f(x)$  will be below the red line.

## Big-oh Example

$x^2+1$  is  $O(x^2)$

Proof [From the definition of  $O(\cdot)$ , this claims that for some  $k$ , there is a constant  $C$  so that whenever  $x>k$ , then  $x^2+1 \leq C*x^2$ . That is what we want to prove – i.e. find  $k$  and  $C$ .]

When  $x>0$ ,  $x^2+1 \leq x^2+x^2 = 2x^2$  ( $x \in \mathbb{N}$ , so  $x \geq 1$ , and  $x^2 \geq 1$ )

So if  $k=0$  and  $C=2$ , we have: whenever  $x>k$ ,  $x^2+1 \leq C*x^2$ .

Therefore,  $x^2+1$  is  $O(x^2)$ .

This says that  $x^2+1$  does not grow significantly faster than  $x^2$ . So any algorithm taking  $x^2+1$  steps will be roughly similar to an algorithm taking  $x^2$  steps.

# Proof techniques: direct proof

- On the previous slide, we proved that  $x^2+1$  is  $O(x^2)$ .
- By **proof**, we mean a convincing argument that the statement is correct.
- In this case, we gave a **direct proof**
  - we started with what we knew, and applied reasoning to turn those known facts into the statement we wanted
- in logic, we used direct proof to prove that conclusions followed from initial facts
  - the logic proofs were formal – we listed all facts, showed all steps, and used only valid rules of inference
  - normally, our proofs will be more informal – we may skip steps, and rely on human understanding

## Example 2

$4x^2+5x-3$  is  $O(x^2)$

Proof

So  $4x^2+5x-3$  does not grow too fast compared to  $x^2$

## Example 3

$x^3+2x^2+2$  is not  $O(x^2)$

Proof [First, we will assume  $x^3+2x^2+2$  is  $O(x^2)$ , and then we will show that this leads to a contradiction.]

Suppose  $x^3+2x^2+2$  **is**  $O(x^2)$ . Then, by the definition of  $O(.$ ), there is some pair of constants  $k$  and  $C$  such that for every  $x>k$ ,  $x^3+2x^2+2 \leq C*x^2$ .

Consider any value of  $x$  bigger than  $k$ .

So we have  $x^3+2x^2+2 \leq Cx^2$

Then we must have  $x+2+2/x^2 \leq C$

Therefore  $x+2 \leq C$  (*since  $2/x^2 > 0$ , and we are no longer adding it*)

But if we pick any value of  $x$  s.t.  $x>C$ , we must have  $x+2 > C$ .

These last two statements contradict each other, so something is wrong in our reasoning. The only thing we could have got wrong is the assumption that  $x^3+2x^2+2$  is  $O(x^2)$ . Therefore,  $x^3+2x^2+2$  is not  $O(x^2)$ .

## The implications

By the previous example,  $x^3+2x^2+2$  will grow much faster than  $x^2$  when  $x$  is large.

So if we have two algorithms, A and B, for handling the same task, where the input is of size  $n$ , and

- the number of steps required by A is  $O(n^2)$
- the number of steps required by B is  $n^3+2n^2+2$ ,

then if the input might be large, we should prefer algorithm A.

For large input, algorithm B will require more time than A, and as the input gets larger, the gap in performance will continue to get bigger.

# Proof Technique: proof by contradiction

The proof of the previous example used **proof by contradiction**.

We wanted to prove some statement.

We first assume the negation of the statement. We then apply reasoning to show that that assumption leads to a contradiction. If all our reasoning is correct, then the only thing that could be wrong is the assumption. Therefore, the statement must be correct.

# Polynomial growth

If  $f(x) = a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_2x^2 + a_1x + a_0$   
where each  $a_i$  is a constant, then  $f(x)$  is  $O(x^n)$

Proof      Let  $x > 1$

$$\begin{aligned} f(x) &= a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_2x^2 + a_1x + a_0 \\ &\leq |a_n|x^n + |a_{n-1}|x^{n-1} + |a_{n-2}|x^{n-2} + \dots + |a_2|x^2 + |a_1|x + |a_0| \\ &\leq |a_n|x^n + |a_{n-1}|x^n + |a_{n-2}|x^n + \dots + |a_2|x^n + |a_1|x^n + |a_0|x^n \\ &= (|a_n| + |a_{n-1}| + |a_{n-2}| + \dots + |a_2| + |a_1| + |a_0|)x^n \\ &= Cx^n, \text{ for } C = |a_n| + |a_{n-1}| + |a_{n-2}| + \dots + |a_2| + |a_1| + |a_0| \end{aligned}$$

so we can take  $k=0$ ,  $C = |a_n| + |a_{n-1}| + |a_{n-2}| + \dots + |a_2| + |a_1| + |a_0|$   
and so  $f(x)$  is  $O(x^n)$

If the running time of an algorithm is a polynomial function of the input size  $x$ , we only need to worry about the highest power of  $x$

# Summary

- we are interested in how many steps our algorithms might need in the worst case
  - we call this the (worst case) **run-time** of the algorithm
- we state the run-time as a function of the size of the input
  - if the function grows too quickly, then programs that implement the algorithms become inefficient
- we use the big-oh notation to classify different functions
  - if  $f(x)$  is not  $O(g(x))$ , then  $f(x)$  will soon become significantly larger than  $g(x)$
  - if the run-time function is a polynomial, we only care about the highest power
- we have a simple hierarchy of run-times based on big-oh:

$$\log n < n < n \log n < n^2 < n^3 < \dots < 2^n < n! < n^n$$

# Next lecture ...

algorithms for sorting data

# CS1113 Sorting

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Algorithms for sorting data

Bubble sort

Summation

# Sorting a sequence of items

The task of re-arranging some sequence of items into a specific order is one of the most common sub-tasks in many computer applications. E.g.:

- books listed in order of total sales on Amazon
- web pages listed in order of page rank on Google
- items listed in order of recommendation score in recommender systems
- teams in a sports league listed in order of points
- names in an address book being ordered alphabetically

Re-arranging a sequence into some order is called **sorting**

There are two main issues:

(i) how to do it, and (ii) how to do it efficiently.

## Example

Sort the sequence 6,2,4,9,1,3 in increasing order.

initial sequence:

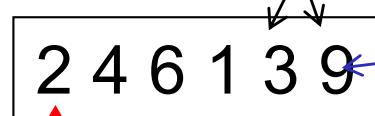
6 2 4 9 1 3

target sequence:

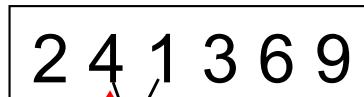
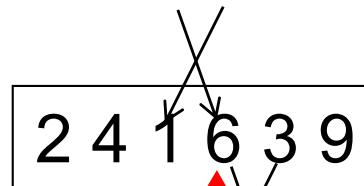
1 2 3 4 6 9

### First method

We will pass through the sequence from front to back swapping each pair that are in the wrong order. This will push the biggest number to the end. We then repeat until all numbers are in the right place.



1<sup>st</sup> pass complete  
Biggest element  
now in last place



2<sup>nd</sup> pass complete  
2<sup>nd</sup> biggest element  
now in 2<sup>nd</sup> last place

3<sup>rd</sup> pass complete  
3<sup>rd</sup> biggest element  
now in 3<sup>rd</sup> last place

4<sup>th</sup> pass complete  
4<sup>th</sup> biggest element  
now in 4<sup>th</sup> last place

5<sup>th</sup> pass complete  
5<sup>th</sup> biggest element  
now in 5<sup>th</sup> last place

# Algorithm: Bubble Sort

Input: a sequence  $x_1, x_2, \dots, x_n$  (where  $n > 1$ )

Output: a sequence  $y_1, y_2, \dots, y_n$ , which is an ordering of the  $x_i$

1. for each  $i$  from 1 to  $n-1$
2.     for each  $j$  from 1 to  $n-i$
3.         if  $x_j > x_{j+1}$
4.             then swap the positions of  $x_j$  and  $x_{j+1}$

1. each time round outer loop, we will push  $i^{\text{th}}$  biggest element into  $i^{\text{th}}$  last place
2. so must consider each of the first  $(n-i)$  positions in turn
3. if the current element is bigger than the one that comes after it
4. swap the current element with the one that comes after it

# Bubble Sort run-time is $O(n^2)$

## Proof

The outer loop considers each value of  $i$  from 1 to  $n-1$ , and calls the inner loop each time.

Each time the inner loop is called, the value of  $i$  is fixed, and we then make  $(n-i)$  comparisons (and at most  $(n-i)$  swaps).

The number of comparisons is then

$$\begin{aligned} & (n-1) + (n-2) + (n-3) + \dots + (n-(n-2)) + (n-(n-1)) \\ &= (n-1) + (n-2) + (n-3) + \dots + (2) + (1) \\ &= (n-1)*n/2 \quad [\text{proof of this on next slides}] \\ &= (n^2-n)/2 \\ &= n^2/2 - n/2 \end{aligned}$$

which is  $O(n^2)$  [by the result proved in previous lecture, since we have a polynomial with  $n^2$  as the highest power of  $n$ ]

# Summation

Often when we are analysing algorithms, or analysing memory requirements (and even specifying problems), we will want to write down a series of numbers or variables which have to be added together. It is tedious to write out the series each time.

Instead, if there is an easy pattern to the series, we will use a shorthand based on the Greek capital letter sigma:

$$\sum_{i=low}^{high} pattern$$

means add together all the values that match the pattern, where  $i$  ranges from  $low$  to  $high$ .  
 $i$  is called the **index** of the summation

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + (n-1) + n$$

$$\sum_{i=1}^n x_i = x_1 + x_2 + x_3 + \dots + x_{n-1} + x_n$$

# Examples

Expand the following:

$$\sum_{k=1}^5 \frac{1}{k}$$

$$\sum_{j=0}^8 y_j$$

$$\sum_{i=0}^n a_i x^i$$

$$\sum_{i=0}^n \binom{n}{i} x^{n-i} y^i$$

What are the values of the following?

$$\sum_{k=1}^8 k$$

$$\sum_{i=1}^4 i^2$$

$$\sum_{i=1}^n i = \frac{1}{2} * n * (n+1)$$

## Proof

Expand the left hand side to get  $1+2+3+\dots+(n-1)+n$

Add that expression to itself, but write it out backwards underneath the first version, aligning the values, and then add each aligned pair:

$$\begin{array}{ccccccccc} 1 & +2 & +3 & + \dots & + (n-1) & + & n \\ + n & +(n-1) & +(n-2) & + \dots & + 2 & + & 1 \\ \hline (n+1) & +(n+1) & +(n+1) & + \dots & + (n+1) & + & (n+1) \end{array}$$

There are  $n$  of these terms, so this sum gives  $n*(n+1)$ .

But we had to double our original series to get this, so our original expression must be  $n*(n+1)/2$

# Next lecture ...

insertion sort

induction

# CS1113

## Selection Sort, Insertion Sort, and Proof by Induction

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Selection Sort, Insertion Sort, and Proof by Induction

Selection Sort

Insertion sort

Principle of mathematical induction

Example proofs of statements from algorithm analysis

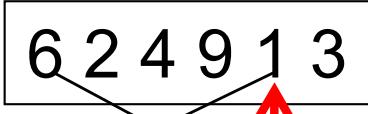
# Selection Sort

Pass through the list ( $n-1$ ) times.

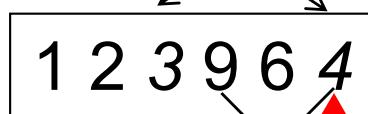
On the first pass, we find the smallest item and swap it with the item in 1<sup>st</sup> place.

On the second pass, we find the 2<sup>nd</sup> smallest item, and swap it with the item in 2<sup>nd</sup> place.

...



1<sup>st</sup> pass complete  
Smallest element  
now in first place



2<sup>nd</sup> pass complete  
2<sup>nd</sup> smallest element  
now in 2<sup>nd</sup> place



3<sup>rd</sup> pass complete  
3<sup>rd</sup> smallest element  
now in 3<sup>rd</sup> place



4<sup>th</sup> pass complete  
4<sup>th</sup> smallest element  
now in 4<sup>th</sup> place

5<sup>th</sup> pass complete  
5<sup>th</sup> smallest element  
now in 5<sup>th</sup> place

# Algorithm: Selection Sort

Input: a sequence  $x_1, x_2, \dots, x_n$  (where  $n > 1$ )

Output: a sequence  $y_1, y_2, \dots, y_n$ , which is an ordering of the  $x_i$

- ```

1. for each i from 1 to n-1       $x_i$  is the new position
2.     min := i                  we are trying to establish
3.     for each j from i+1 to n   consider all values after  $x_i$ 
4.         if  $x_j < x_i$           if  $x_j$  is smallest so far
5.             min := j           remember j
6.             temp :=  $x_i$ 
7.              $x_i := x_j$           swap  $x_i$  with smallest remaining
8.              $x_j := temp$ 
9. return x sequence

```

# Selection sort is $O(n^2)$

## Proof

We go round the outer loop  $n-1$  times.

Each time, we check the remaining elements, so  
 $n-1, n-2, \dots, 1$  checks

So we have

$$(n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i = (n(n-1)/2) = n^2 / 2 - n / 2$$

comparisons, which is  $O(n^2)$

## Insertion sort

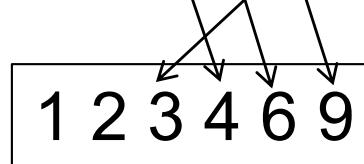
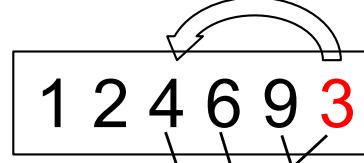
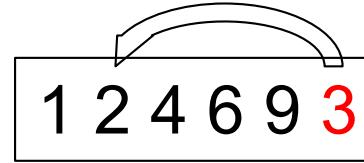
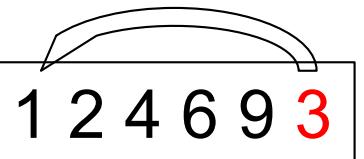
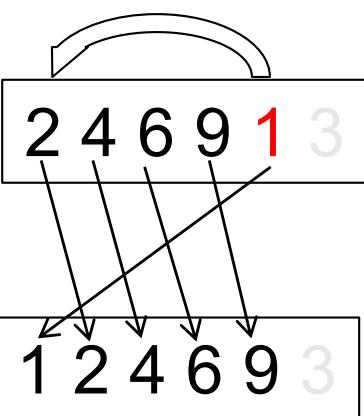
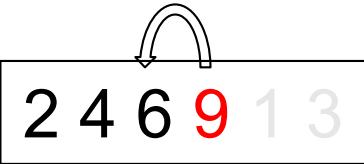
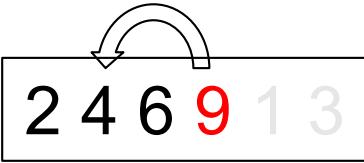
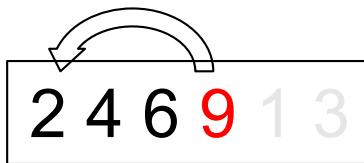
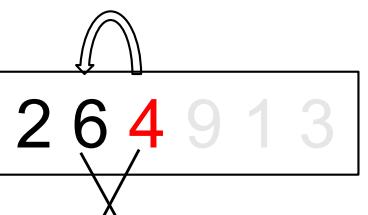
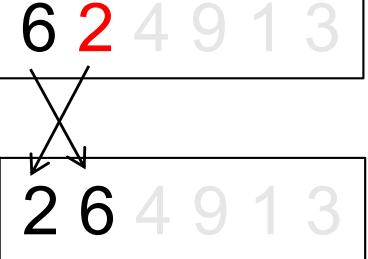
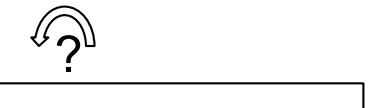
We now consider a different algorithm for sorting a sequence.

First we consider the sub-sequence of just the first element. It is obviously in the right position in that sub-sequence.

Now we add the current 2<sup>nd</sup> element to that subsequence. Either it goes after the 1<sup>st</sup> (and so we do nothing), or it goes before, so we move the old 1<sup>st</sup> element to 2<sup>nd</sup> place.

Then we add the 3<sup>rd</sup> element. Either it goes in 3<sup>rd</sup> place (so we do nothing), or it goes into 2<sup>nd</sup> place, so we shove the element that was there into 3<sup>rd</sup>, or it goes 1<sup>st</sup>, and we shove the other two elements back one place.

We repeat until we have added the last element.



# Algorithm: Insertion Sort

Input: a sequence  $x_1, x_2, \dots, x_n$  (where  $n > 1$ )

Output: a sequence  $y_1, y_2, \dots, y_n$ , which is an ordering of the  $x_i$

1. for each  $i$  from 2 to  $n$
2.      $j := 1$
3.     while  $x_j < x_i$
4.          $j := j + 1$
5.     temp :=  $x_i$
6.     for each  $k$  from  $i$  down to  $j + 1$
7.          $x_k := x_{k-1}$
8.      $x_j := \text{temp}$

$x_i$  is the new element  
we are trying to insert

keep going until we find the right place

we will insert  $x_i$  in front of  $x_j$  but store  $x_i$  for now

working forwards, copy each  
element into the next position

which leaves  $x_j$ 's old position free, and we now  
insert the value we stored into that position

# Insertion sort is $O(n^2)$

## Proof

In the worst case, all the elements are already in the correct order, and each time we extend the subsequence by one more element, we have to compare it against all other elements in the subsequence (and then against itself to stop the while loop).

The length of the subsequence ranges from 2 to  $n$ , and so we need to compare against 1, 2, 3, ...,  $n-1$  elements. So we have

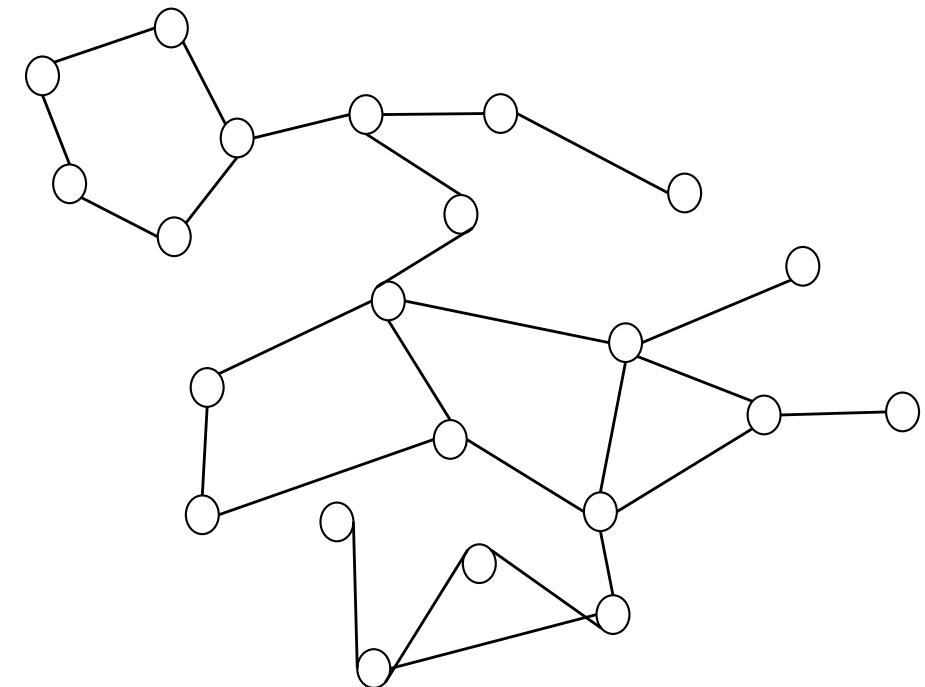
$$1 + 2 + 3 + \dots + n - 1 = \sum_{i=1}^{n-1} i = (n(n-1)/2) = n^2 / 2 - n / 2$$

comparisons, which is  $O(n^2)$

## Example

A network is a collection of devices, with links between pairs of devices, and where there is a finite chain of links between any pair.

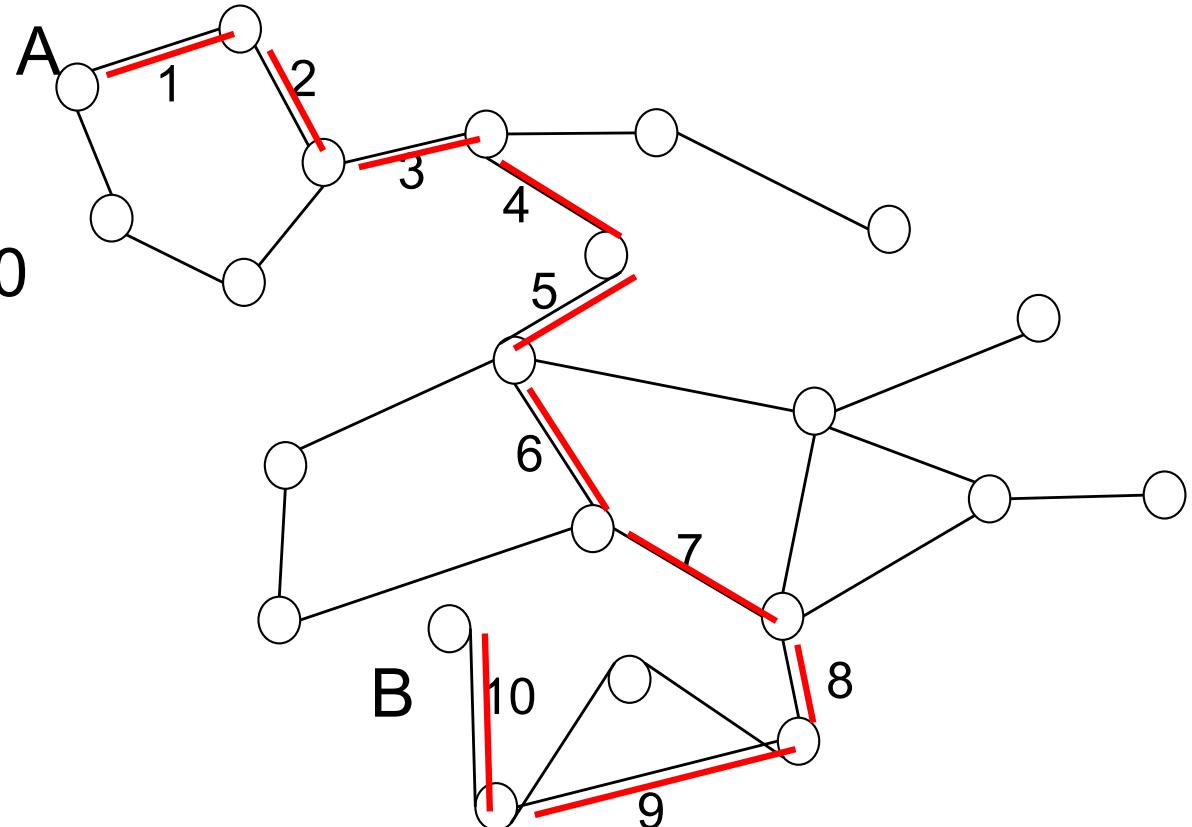
I can send a packet from one device along a link to its neighbour. Any device that receives a packet can send it on in turn to one of its neighbours.



Can I send a packet from any node to any other node?

Choose A and B.  
There is a chain that connects them with 10 links.

I can send a packet from A along link 1.  
It can then be sent along link 2. Then 3, then 4, 5, 6, 7, 8, 9 and 10, when it reaches B.



But that argument was for a specific chain of known length.  
Can we prove it for all possible pairs (or all possible chains)?

# Proof

Every pair of devices has a finite chain of links connecting them.

If the chain between a pair is of length 1, then there is a single link between them, so I can send a packet along it.

Now suppose I can prove that I can send a packet along any chain of length  $k$ . (\*\*)

Consider a chain of length  $k+1$  connecting nodes A and B. The chain must consist of a chain of length  $k$  from A to some device C, and then a single link from C to B. By (\*\*), I can send a packet from A to C. It can then be sent on from C to B over the single link. So I can send a packet along any chain of length  $k+1$ .

So I have proved that if I can send a packet along a chain of length  $k$ , then I can also send it along a chain of length  $k+1$ . But I have also proved I can send it along a chain of length 1. Therefore also of length 2, length 3, and so on for all finite lengths, and so I have proved I can send a packet between any pair of devices in the network.

# Mathematical Induction

The proof on the previous slide used the principle of mathematical induction.

base case

We prove a claim for the simplest case, typically the value 1.

We then show that if we can prove it for some value  $k$ , then we can also prove it for  $k+1$ .

inductive step

We can then prove it for all values of  $k$ .

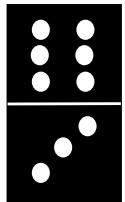
In logic terms, for some predicate  $P$ ,

we have  $P(1)$

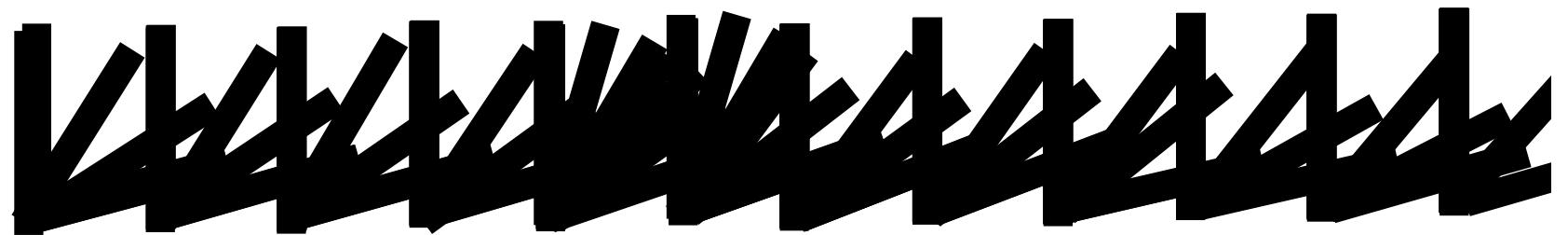
and we also have  $\forall x (P(x) \rightarrow P(x+1))$

so we can deduce  $P(2), P(3), P(4)$ , etc.

# Induction as toppling dominoes



prove the dominoes are set up so that if any one domino falls, the one after it will also fall



prove that the first one falls

and so all dominoes must fall

# Example Proof by Induction

Prove  $n^2+n$  is even, for all integers  $n>0$ .

## Proof

Note that if a number  $x$  is even, then there is another number  $y$  such that  $x=2y$ , and if two numbers  $w$  and  $z$  are even, then  $w+z$  is even.

When  $n=1$ ,  $n^2+n = 1^2+1 = 1+1 = 2$ , which is even.

Now suppose result true for  $n=k$ , for some  $k>0$ . That is,  $k^2+k$  is even.

$$\begin{aligned} \text{When } n=k+1, n^2+n &= (k+1)^2+k+1 = k^2+2k+1+k+1 \\ &= k^2+k+k+1+k+1 \\ &= k^2+k + 2(k+1) \end{aligned}$$

But we have assumed  $k^2+k$  is even, and  $2(k+1)$  is even, so their sum is also even. Therefore, if the result is true for  $n=k$ , it is also true for  $n=k+1$ . But we showed the result was true for  $n=1$ . Therefore, by induction, result is true for all  $n>0$ .

**Example:**  $\sum_{i=1}^n i = n * (n + 1) / 2$

Proof

When  $n=1$ ,  $\sum_{i=1}^1 i = 1 = \frac{1}{2} * 1 * 2$  and so result is true for  $n=1$ .

Now suppose true for  $n=k$ . i.e.  $\sum_{i=1}^k i = \frac{1}{2} * k * (k + 1)$  [★]

Consider  $n=k+1$ :

$$\begin{aligned}\sum_{i=1}^{k+1} i &= \left( \sum_{i=1}^k i \right) + (k + 1) = \frac{1}{2} * k * (k + 1) + (k + 1) \quad (\text{using } \star) \\ &= \left( \frac{1}{2} * k + 1 \right) * (k + 1) \\ &= \left( \frac{1}{2} (k + 2) \right) * (k + 1) \\ &= (k + 1) * ((k + 1) + 1) / 2\end{aligned}$$

So if result is true for  $n=k$ , it is true for  $n=k+1$ . But result is true for  $n=1$ , and so by induction, is true for all  $n>1$ .

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

This result is needed in the analysis of the runtime of various algorithms.

## Proof

*Will be done on the blackboard*

# Next lecture ...

other proof methods

recursion

# CS1113 Recursion

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Recursion and Proof

Recursive Definitions

Recursive Algorithms

# What is the next element?

a) 2, 4, 6, 8, ...

b) 1, 5, 9, 13, ...

c) 1, 4, 9, 16, ...

d) 1, 1, 2, 3, 5, 8, 13, 21, ...

# How do we describe the sequence precisely?

1    2    3    4

- a) 2, 4, 6, 8, ...

$$f(i) = 2*i$$

Method 1: find a function  
which computes the  $i^{\text{th}}$   
element directly

- b) 1, 5, 9, 13, ...

$$f(i) = (4*i)-3$$

- c) 1, 4, 9, 16, ...

$$f(i) = i^2$$

- d) 1, 1, 2, 3, 5, 8, 13, 21, ...

?

# How do we describe the sequence precisely?

1    2    3    4

- a) 2, 4, 6, 8, ...

$$f(i) = 2*i$$

- b) 1, 5, 9, 13, ...

$$f(i) = (4*i)-3$$

- c) 1, 4, 9, 16, ...

$$f(i) = i^2$$

- d) 1, 1, 2, 3, 5, 8, 13, 21, ...

?

Note:

The mathematical definition of a sequence is a function

$f: \mathbb{N} \rightarrow \mathbb{R}$

That is, a function which specifies an output value for each integer, corresponding to each place in the sequence

# How do we describe the sequence precisely?

1    2    3    4

- a) 2, 4, 6, 8, ...

$$f(1) = 2$$

$$f(i) = f(i-1) + 2, \text{ when } i > 1$$

- b) 1, 5, 9, 13, ...

$$f(1) = 1$$

$$f(i) = f(i-1) + 4, \text{ when } i > 1$$

- c) 1, 4, 9, 16, ...

$$f(1) = 1$$

$$f(i) = f(i-1) + 2*i - 1$$

- d) 1, 1, 2, 3, 5, 8, 13, 21, ...

$$f(1) = 1$$

$$f(2) = 1$$

$$f(i) = f(i-1) + f(i-2), \text{ when } i > 2$$

Method 2: a function which computes the  $i^{\text{th}}$  element from previous ones

# Recursive Definitions

a **recursive definition** of a function is one which

- specifies the output directly for one or more **base cases**
- specifies a **recursive** rule for other input based on the function value for other (usually smaller) inputs

Example: factorial

$$f(0) = 1 \quad 0! = 1$$

$$f(x) = x * f(x-1) \text{ when } x > 0 \quad x! = x * (x-1)!$$

```
def factorial(n):  
    if n < 2:  
        return 1  
    else:  
        return n*factorial(n-1)
```

base case

recursive rule

# Recursive Structures

We can use essentially the same idea to define complex structures. We have seen this already ...

We defined well-formed formulae (wff) of propositional logic as follows:

1. All propositional symbols on their own are wffs
2. if  $p$  and  $q$  are wffs, then so are  $(p)$ ,  $(\neg p)$ ,  $(p \wedge q)$ ,  $(p \vee q)$ ,  $(p \rightarrow q)$  and  $(p \leftrightarrow q)$
3. nothing else is a wff unless it can be formed by repeatedly applying rules 1 and 2 above.

base case

recursive rule

The recursive rule specifies members of the set based on things already known to be members

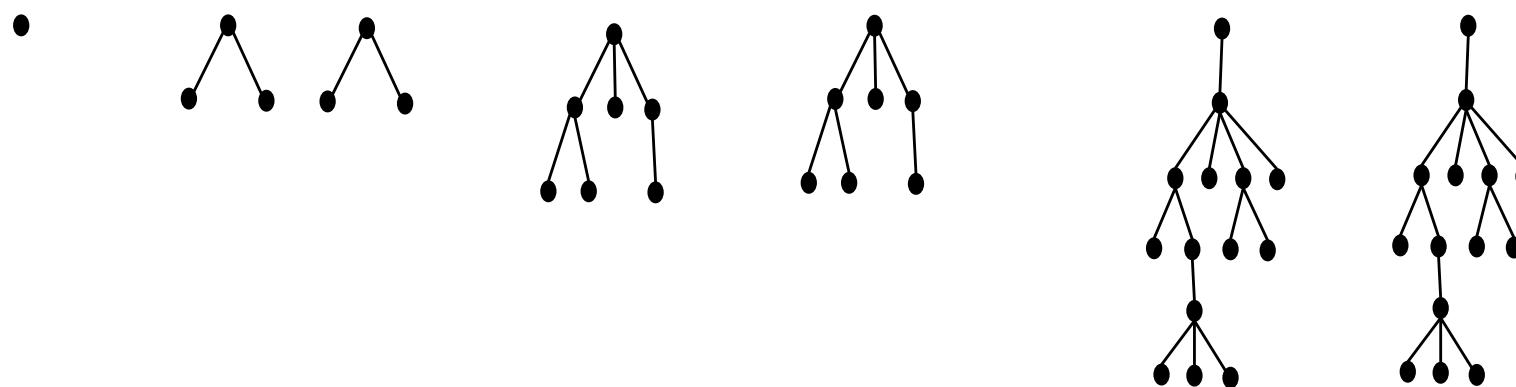
# General definition of rooted trees

A **vertex** is some entity.

An **edge** is a link between two vertices.

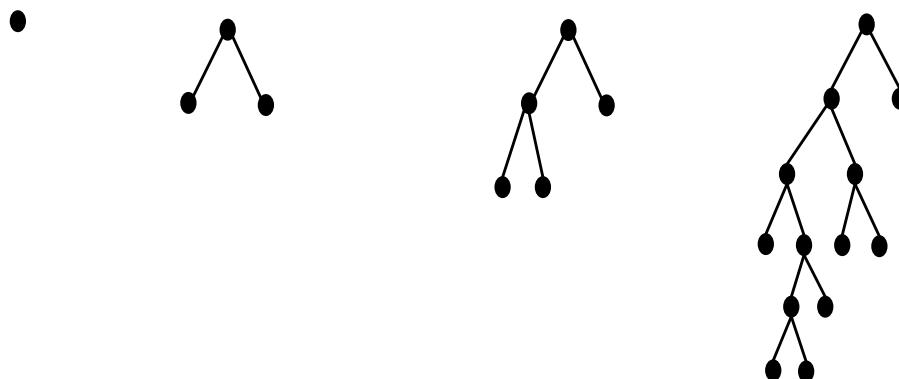
1) a single vertex is a **rooted tree**, and the vertex is the **root**

2) if we have different rooted trees  $t_1, t_2, \dots, t_n$  with roots  $v_1, v_2, \dots, v_n$ , and a new vertex  $v$ , then we can form a new rooted tree with root  $v$  by adding separate edges from  $v$  to each of the  $v_i$



# General definition of full binary trees

- 1) a single vertex is a **full binary tree**, and the vertex is the **root**
- 2) if we have two different full binary trees  $t_1$ , and  $t_2$  with roots  $v_1$  and  $v_2$ , and a new vertex  $v$ , then we can form a new full binary tree with root  $v$  by adding two separate edges from  $v$  to  $v_1$  and  $v_2$



# Proving statements about recursive structures

We can use *structural* induction, in which the base case of the proof proves the statement for the base case of the definition, and the inductive step prove the result for the recursive rule.

Example: every vertex in a full binary tree has either 0 or 2 vertices connected below it in the tree

## Proof

For single vertex trees, every vertex has 0 vertices below it. For the recursive step, assume result is true for the two trees  $t_1$  and  $t_2$ . Now consider the new tree formed. Nothing has been added below any of the vertices in  $t_1$  or  $t_2$ , so we only need to look at the new vertex  $v$ .  $v$  has been connected to exactly two nodes (roots of  $t_1$  and  $t_2$ ). Therefore result is true for all full binary trees.

# Recursive Algorithms

A recursive algorithm is one which uses a call to itself, on different input. Normally, the input is smaller.

## Example

Algorithm: factorial

Input: integer  $n \geq 0$

Output: integer equal to the factorial of  $n$

1. if  $n == 0$  return 1
2. else return  $n * \text{factorial}(n-1)$

# Proving recursive algorithms correct

the standard approach is to use induction

Example: the algorithm *factorial* correctly computes

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$

## Proof

When input  $n=0$ , the algorithm returns 1, and  $0!=1$ , so true.

Assume true for input  $n=k$ :

i.e.  $\text{factorial}(k)$  correctly computes  $k * (k-1) * (k-2) * \dots * 2 * 1$

Now consider  $n=k+1$ .

$\text{factorial}(k+1)$  returns the result of  $(k+1) * \text{factorial}(k+1-1)$ .

But  $\text{factorial}(k+1-1) = \text{factorial}(k) = k * (k-1) * \dots * 2 * 1$  (assumption).

So  $\text{factorial}(k+1) = (k+1) * k * (k-1) * \dots * 2 * 1$ .

And so result is true by induction.

# Proving run-times of recursive algorithms

Again, we normally do this by induction.

Example: the algorithm  $\text{factorial}(n)$  requires  $n$  multiplications.

## Proof

When  $n=0$ , the result is returned with 0 multiplications, so true  
Assume result is true for  $n=k$  for some  $k>0$ . i.e.  $\text{factorial}(k)$  requires  $k$  multiplications.

Now consider  $n=k+1$ .  $\text{factorial}(k+1)$  first computes  $\text{factorial}(k)$  and then multiplies the result by  $(k+1)$ . But  $\text{factorial}(k)$  requires  $k$  multiplications, by assumption. Therefore  $\text{factorial}(k+1)$  must require  $k+1$  multiplications. So result is true for  $n=k+1$

Therefore true for all  $k\geq 0$ , by induction.

## Example: power

### Algorithm: power

Input: integer  $x > 0$ , integer  $p > 0$

Output: integer equal to  $x$  to the power of  $p$

1. if  $p == 1$  return  $x$
2. else return  $\text{power}(x, p-1) * x$

Exercise: prove this is correct.

Exercise: prove the algorithm  $\text{power}(x,n)$  requires  
 $(n-1)$  multiplications, for  $n \geq 1$

# Algorithms over recursive structures

Suppose we now have a representation of a tree in our programming language, with functions:

*root( $t$ )*, which returns the vertex that is the root of the tree  $t$

*subtrees( $t$ )* which returns a list of subtrees whose roots are directly linked to the root of  $t$ .

*name( $v$ )* which returns a name associated with a vertex  $v$ .

We can write an algorithm which flattens the tree, printing out the name of each vertex in turn.

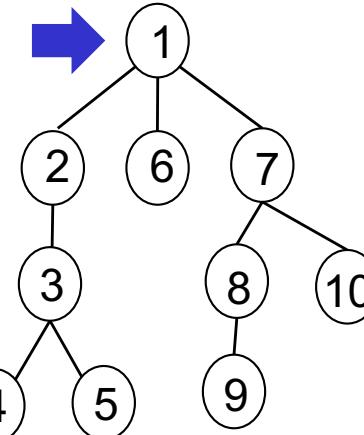
# Algorithm to flatten a tree

Algorithm: flatten

Input: a tree  $t$

output: no return value, but the name of each vertex of  
 $t$  is printed to the screen

1. print name( $\text{root}(t)$ )
2.  $L := \text{subtrees}(t)$
3. for each element  $s$  of  $L$  in turn
4.      $\text{flatten}(s)$

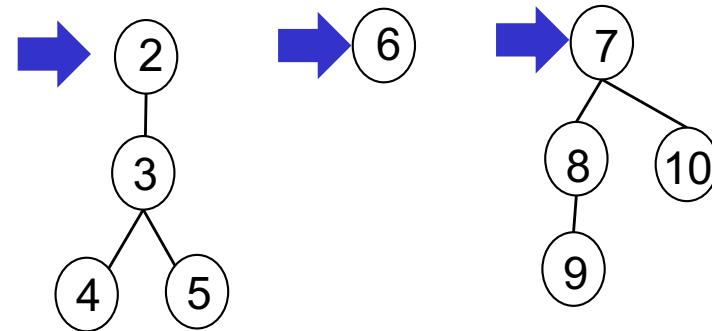


Algorithm: flatten

Input: a tree  $t$

output: no return value, but the name  
of each vertex of  $t$  is printed  
to the screen

1. print name(root( $t$ ))
2.  $L := \text{subtrees}(t)$
3. for each element  $s$  of  $L$  in turn
4.     flatten( $s$ )



1 2 3 4 5 6 7 8 9 10



# Euclid's Algorithm

given two positive integers  $n$  and  $m$ , find the greatest common divisor (i.e. the largest integer that divides into both of them with no remainder)

input: two integers  $a, b$  with  $a \geq b \geq 0$

output: integer greatest common divisor of  $a$  and  $b$

1. if  $b == 0$
2.     return  $a$
3. else
4.     return  $\text{gcd}(b, a \bmod b)$

**base case**

**recursive rule**

$\text{gcd}(a, 0) = a$   
 $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$

```
def gcd(a, b):  
    if b == 0:  
        return a  
    else:  
        return gcd(b, a % b)  
    }
```

# CS1113 Proof

**Lecturer:**

Professor Barry O'Sullivan

Office: 2.65, Western Gateway Building

email: *b.osullivan@cs.ucc.ie*

<http://osullivan.ucc.ie/teaching/cs1113/>

# Proof

Proof methods:  
direct proof  
proof by induction  
proof by contradiction  
proof by cases

Example proofs of statements from Algorithm Analysis

# What is a ‘Proof’?

- A proof is a valid argument that shows that some statement is true
- Proofs can be formal
  - E.g. Proofs in propositional logic applying inference rules and logical equivalences
- Or informal - but even an informal proof must be convincing, and must cover all loopholes
  - no gaps, no handwaving, no wishful thinking
  - must be precise and unambiguous
  - use mathematical and logical notation
  - explain and justify every step

# Uses of proof in computing

- Showing that an algorithm (or program) does what it is supposed to do
- Showing that one algorithm has, in the worst case, a lower runtime than another algorithm
- Showing that an operating system is secure
- Showing that a protocol for computing across a network is safe and will not enter deadlock
- Showing that a system specification is consistent
- Checking that a decision is justified in an intelligent program

# We have already seen ...

- Direct proof
  - E.g. Proof of the Handshaking Lemma in Lecture 9
  - E.g. Proof that  $x^2+1$  is  $O(x^2)$  in Lecture 18
- Proof by contradiction
  - E.g. Proof that  $x^3+2x^2+2$  is not  $O(x^2)$  in Lecture 18
- Proof by induction
  - Prove a statement is true for a simple base case
  - Prove that **if** statement is true for an intermediate case (e.g. of size  $k$ ) **then** it must be true for the next case (e.g. of size  $k+1$ )
  - E.g. Proof that  $n^2+n$  is even in Lecture 20

$$\forall n \geq 4 \quad 2^n < n!$$

Proof

$n!$  is not  $O(2^n)$

## Proof

# Proof by contrapositive

Revision: For a statement  $p \rightarrow q$ , its **contrapositive** is  $\neg q \rightarrow \neg p$   
A conditional is true if and only if its contrapositive is true

Sometimes, it is easier to prove the contrapositive.

Example: if  $n^2$  is even, then  $n$  is even

## Direct proof attempt

Suppose  $n^2$  is even. Then  $n^2 = 2k$  for some  $k$ . ... *but now what?*

## Proof (by contrapositive)

We will show that if  $n$  is not even, then  $n^2$  is not even

Suppose  $n$  is not even. Then  $n=2k+1$ , for some  $k$ .

Then  $n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2(2k^2+2) + 1$ , which must be odd.

Therefore, by the contrapositive, if  $n^2$  is even,  $n$  is even

# Proof by counter example

Sometimes, we will want to prove that a statement is false.

Example claim: for any integer  $x$ , we can find two integers  $y$  and  $z$  so that  $y^2+z^2 = x$ .

We will prove this false by finding an integer  $x$  which does not obey this pattern.

Consider  $x = 3$ .  $y^2$  and  $z^2$  are both positive.

If  $|y| \geq 2$ , then  $y^2 \geq 4$ , and so  $y^2+z^2 \geq 4$ .

So  $|y|$  must be either 0 or 1. If  $|y| = 0$ , then  $y^2=0$ , so  $z^2$  must be 3. But there is no integer  $z$  such that  $z^2 = 3$ . Therefore  $y$  cannot be 0.

Suppose  $y = 1$ . Then  $y^2 = 1$ , so  $z^2 = 2$ . But there is no integer  $z$  such that  $z^2=2$ . So  $y$  cannot be 1. But those were the only possible values for  $y$ .

Therefore, there are no integers  $y$  and  $z$  such that  $y^2+z^2=3$ , and so the statement is false.

## Proof by cases

Sometimes, we will need to break a statement down into a number of different cases, and show each one is true.

Example: for two integers  $x$  and  $y$ , if  $x=y^2$ , then  $x$  is of the form  $4k$  or  $4k+1$ , for some other integer  $k$ .

### Proof

Case(i):  $y$  is even. So there is an integer  $p$  with  $y=2p$ .

Then  $x = y^2 = (2p)^2 = 4p^2 = 4k$ , if we set  $k=p^2$ .

Case(ii):  $y$  is odd. So there is an integer  $q$  with  $y=2q+1$ .

Then  $x = y^2 = (2q+1)^2 = 4q^2 + 4q + 1 = 4(q^2+q)+1$

and so  $x = 4k+1$ , if we set  $k=q^2+q$

These are all possible cases for  $y$ , so the statement is true.

# What can go wrong?

- showing something works for one or two examples, instead of for all possible values
- assuming the result you want to prove
- not covering all cases
- making jumps in the logic that are not true
- not presenting it as a convincing argument
- leaving large gaps in the argument and assuming it is clear what is happening

# THE END

of the new material ...

# Next lecture ...

## Revision