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,...
```

$f(i) = 2*i$

```
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,  ...
```

?

Method 1: find a function which computes the i^{th} element directly

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

$f:N \rightarrow R$
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,...
```

$f(1) = 2$
$f(i) = f(i-1)+2$, when $i > 1$

```
b) 1,  5,  9,  13,  ...
```

$f(1) = 1$
$f(i) = f(i-1)+4$, 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,  ...
```

$f(1) = 1$
$f(2) = 1$
$f(i) = f(i-1) + f(i-2)$, when $i > 2$

Method 2: a function which computes the i^{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

$f(0) = 1$ $\qquad\qquad$ $0! = 1$

$f(x) = x*f(x\text{-}1)$ when $x > 0$ $\qquad$ $x! = x*(x\text{-}1)!$

```
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

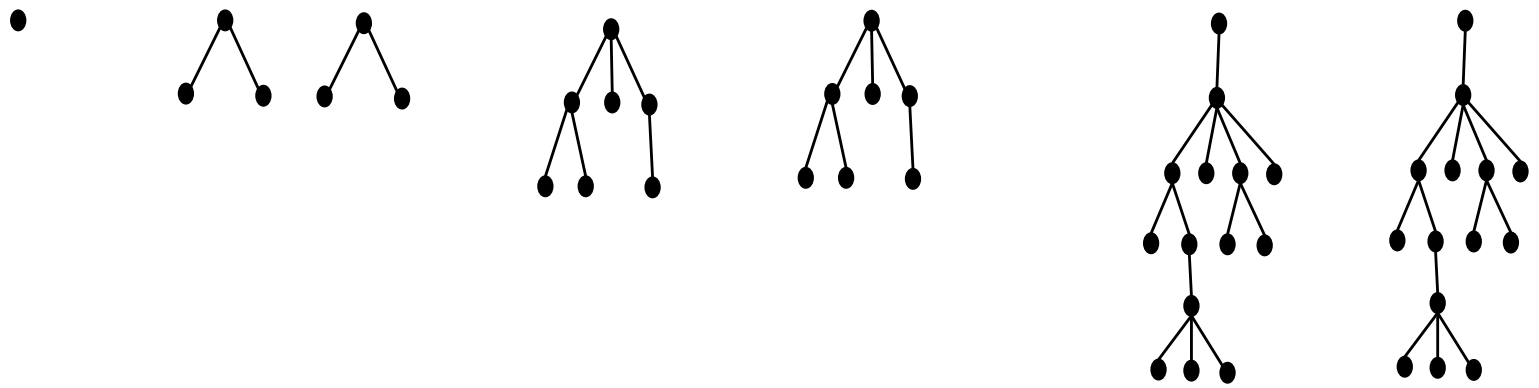
The recursive rule specifies members of the set based on things already known to be members

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

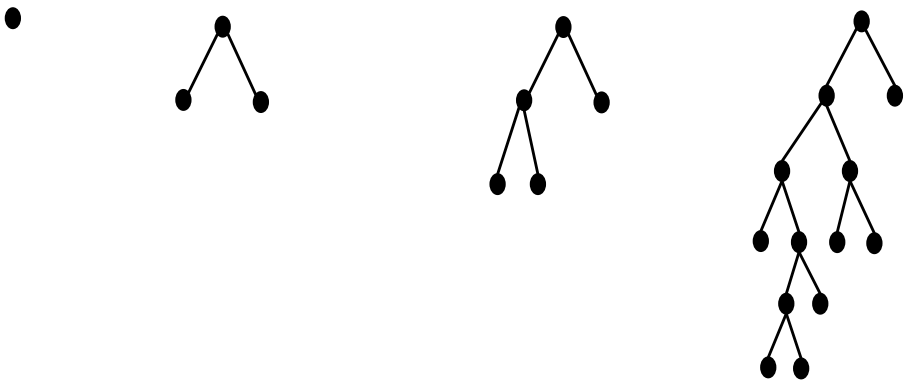
2) if we have different rooted trees t_1, t_2, ..., t_n with roots v_1, v_2, ..., v_n, and a new vertex v, then we can form a new rooted tree with root v by adding separate edges from v to each of the v_i



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 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 >= 0
Output: integer equal to the factorial of n
1. if n == 0 return 1
2. else return n*factorial(n-1)
```

Proving recursive algorithms correct

the standard approach is to use induction

Example: the algorithm *factorial* correctly computes
$n!=n*(n-1)*(n-2)*...*2*1$

Proof

When input $n=0$, the algorithm returns 1, and 0!=1, so true.

Assume true for input $n=k$:

i.e. *factorial(k)* correctly computes $k*(k-1)*(k-2)*...*2*1$

Now consider $n=k+1$.

factorial(k+1) returns the result of (k+1) * factorial(k+1-1) .

But *factorial(k+1-1)* = *factorial(k)* = $k*(k-1)*...*2*1$ (assumption).

So *factorial(k+1)* = $(k+1)*k*(k-1)*...*2*1$.

And so result is true by induction.

Proving run-times of recursive algorithms

Again, we normally do this by induction.

Example: the algorithm *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. *factorial(k)* requires *k* multiplications.
Now consider $n=k+1$. *factorial(k+1)* first computes *factorial(k)* and then multiplies the result by $(k+1)$. But *factorial(k)* requires *k* multiplications, by assumption. Therefore *factorial(k+1)* must require *k+1* multiplications. So result is true for $n=k+1$

Therefore true for all $k>=0$, 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 power(x,p-1)*x
```

Exercise: prove this is correct.

Exercise: prove the algorithm 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*.

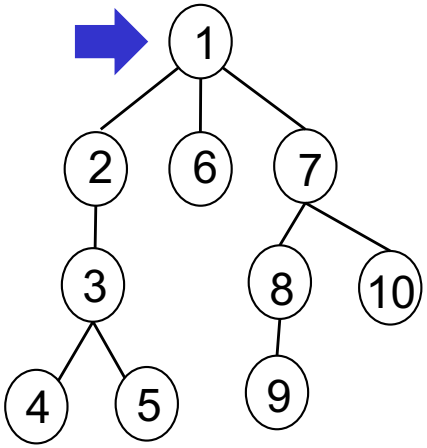
name(v) which returns a name associated with a vertex *v*.

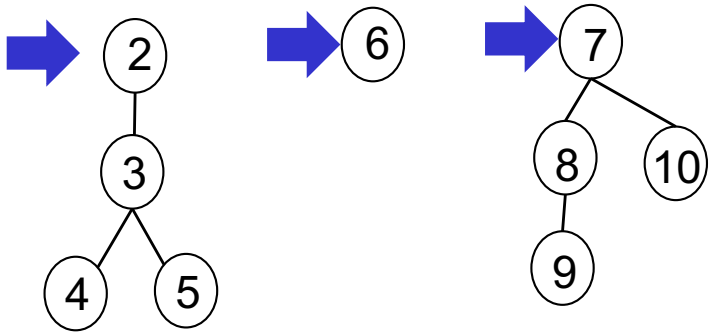
We can write an algorithm which flattens the tree, printing out the name of each vertex in turn.

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(root(t))
2. L := subtrees(t)
3. for each element s of L in turn
4.     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 := 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 >= b >= 0
output: integer greatest common divisor of a and b
1. if b == 0
2.     return a
3. else
4.     return gcd(b, a mod b)
```

base case

recursive rule

gcd(*a*,0) = *a*
gcd(*a*,*b*) = *gcd*(*b*,*a mod b*)

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
}
```