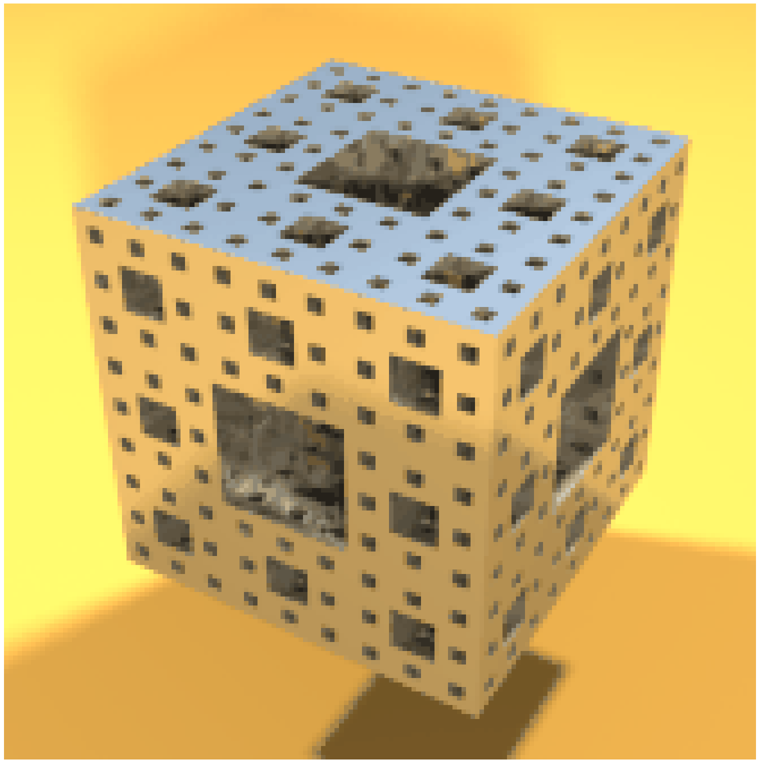
Algorithms and Analysis

Lesson 7: *Recurse!*



Induction, integer power, towers of hanoi, analysis

Outline

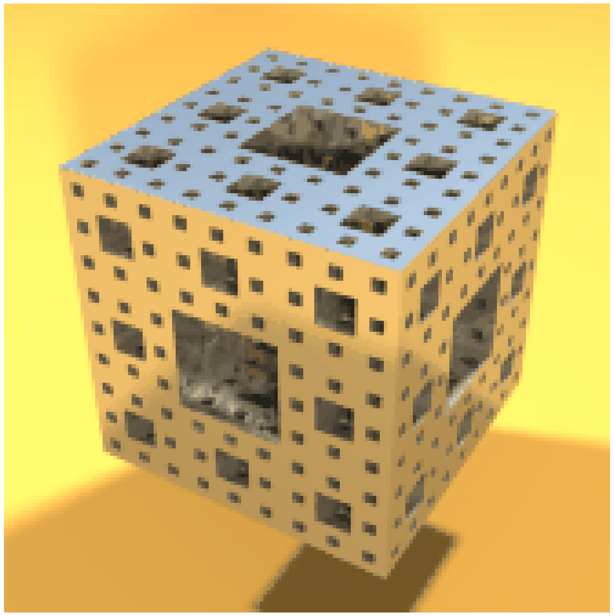
1. **Simple Recursion**

2. Programming Recursively

   - Simple Examples
   - Thinking about Recursion

3. Analysis of Recursion

   - Integer Powers
   - Towers of Hanoi



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z

Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z

Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type

- We repeat this until we reach a trivial case we can solve by some other means

- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.

  ⋆ Definition of factorial:
  $n! = n \times (n-1)!$ with $0! = 1$

  ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z

Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z

Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z

Recursion

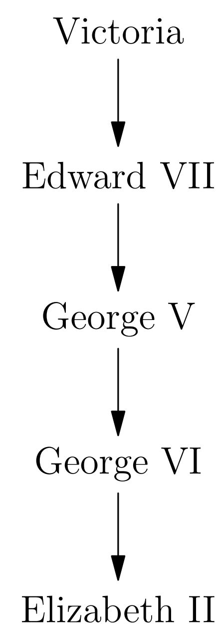
- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type

- We repeat this until we reach a trivial case we can solve by some other means

- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.

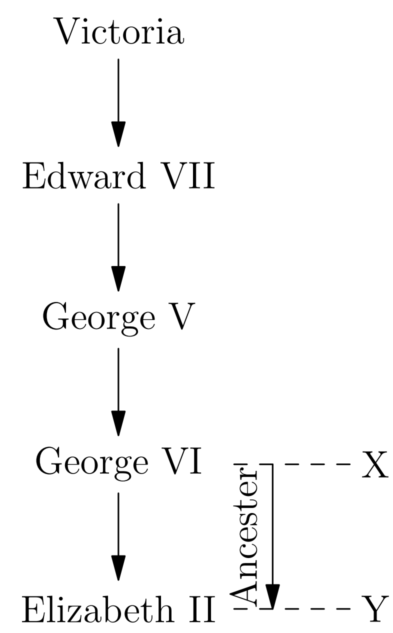
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$

  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Recursion

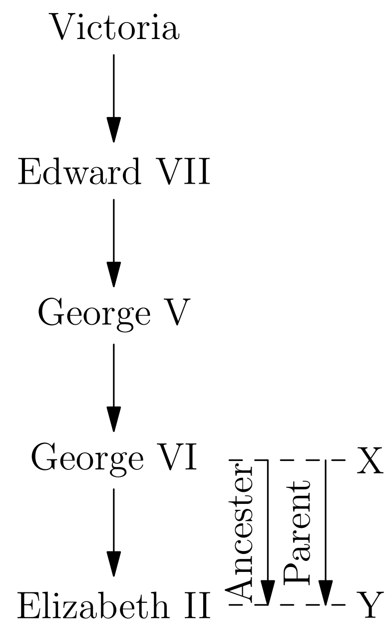
- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type

- We repeat this until we reach a trivial case we can solve by some other means

- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.

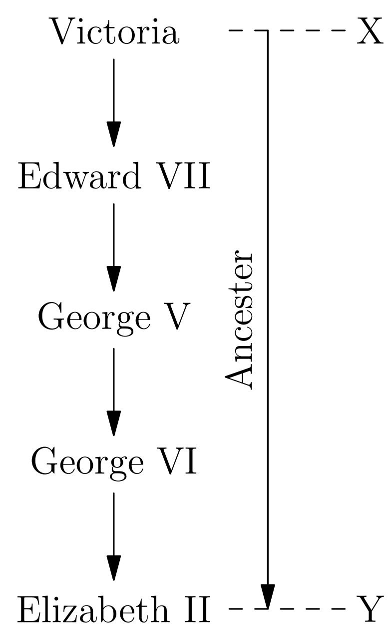
  - ⋆ Definition of factorial: $n! = n \times (n-1)!$ with $0! = 1$

  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Recursion

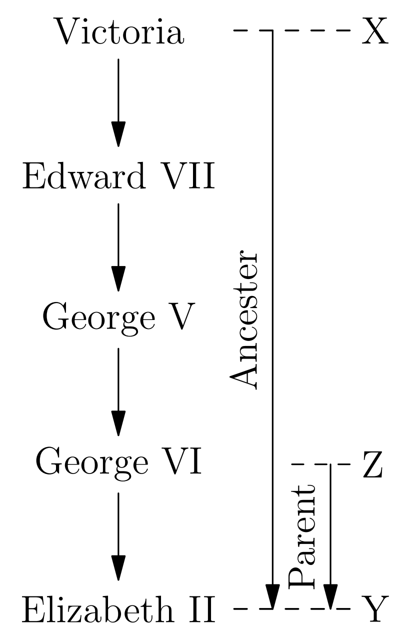
- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type

- We repeat this until we reach a trivial case we can solve by some other means

- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.

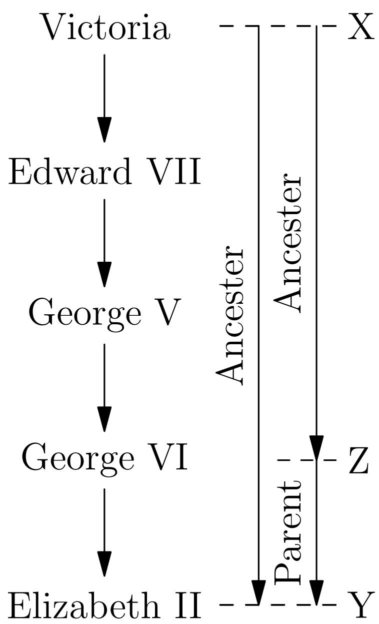
  ⋆ Definition of factorial:
  $n! = n \times (n-1)!$ with $0! = 1$

  ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial:
    $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n)$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n)$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1)$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1}$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1}$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$
$$= 2^{n+1} - 1 + 2^{n+1}$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$

$$= 2^{n+1} - 1 + 2^{n+1} = 2 \times 2^{n+1} - 1$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$

$$= 2^{n+1} - 1 + 2^{n+1} = 2 \times 2^{n+1} - 1 = 2^{n+2} - 1$$

Recap on Proof by Induction

- Let us prove

$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

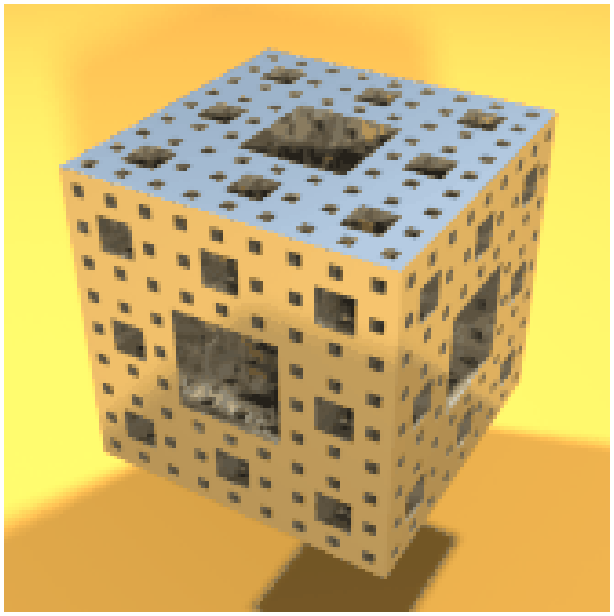
- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$

$$= 2^{n+1} - 1 + 2^{n+1} = 2 \times 2^{n+1} - 1 = 2^{n+2} - 1 \quad ✓$$

Outline

1. Simple Recursion
2. **Programming Recursively**
   - Simple Examples
   - Thinking about Recursion
3. Analysis of Recursion
   - Integer Powers
   - Towers of Hanoi



Programming Recursively

- Most modern programming languages, including C++, allow you to program recursively

- That is they allow functions/methods to be defined in terms of themselves

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    else if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- This will work, is very intuitive, but is certainly not the best way to code a factorial function

Programming Recursively

- Most modern programming languages, including C++, allow you to program recursively

- That is they allow functions/methods to be defined in terms of themselves

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    else if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- This will work, is very intuitive, but is certainly not the best way to code a factorial function

Programming Recursively

- Most modern programming languages, including C++, allow you to program recursively

- That is they allow functions/methods to be defined in terms of themselves

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    else if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- This will work, is very intuitive, but is certainly not the best way to code a factorial function

Integer Powers

- How do we compute 0.95^{25}?

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

0.95^{25}

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24}$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$
$$= 0.95 \times \left(\left(\left((0.95)^3\right)^2\right)^2\right)^2$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$
$$= 0.95 \times \left(\left(\left((0.95)^3\right)^2\right)^2\right)^2 = 0.95 \times \left(\left(\left(0.95 \times (0.95)^2\right)^2\right)^2\right)^2$$

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$
$$= 0.95 \times \left(\left(\left((0.95)^3\right)^2\right)^2\right)^2 = 0.95 \times \left(\left(\left(0.95 \times (0.95)^2\right)^2\right)^2\right)^2$$

six multiplications rather than 24!

Implementing Integer Power

- Integer power looks rather intimidating to code
- However, the recursive definition is easy
- We can easily code this function recursively

```
double power(double x, long n)    // (Overflow is possible)
{
    return  n <  0  ? 1 / power(x,-n)           // Negative power
          : n == 0  ? 1                          // Special case
          : n == 1  ? x                          // Base case
          : n%2 == 0 ? (x = power(x, n/2)) * x  // Even power
          :             x * power(x, n-1);      // Odd power
}
```

Implementing Integer Power

- Integer power looks rather intimidating to code
- However, the recursive definition is easy
- We can easily code this function recursively

```
double power(double x, long n)    // (Overflow is possible)
{
    return  n <  0  ? 1 / power(x,-n)             // Negative power
        :  n == 0   ? 1                           // Special case
        :  n == 1   ? x                           // Base case
        :  n%2 == 0 ? (x = power(x, n/2)) * x     // Even power
        :              x * power(x, n-1);         // Odd power
}
```

Implementing Integer Power

- Integer power looks rather intimidating to code
- However, the recursive definition is easy
- We can easily code this function recursively

```
double power(double x, long n)   // (Overflow is possible)
{
    return  n <  0  ? 1 / power(x,-n)           // Negative power
        :  n == 0   ? 1                         // Special case
        :  n == 1   ? x                         // Base case
        :  n%2 == 0 ? (x = power(x, n/2)) * x   // Even power
        :              x * power(x, n-1);       // Odd power
}
```

Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { //  (Overflow is possible)
    return n <  0  ? power_recurse(1.0/x,-n)  //  Negative power
        :  n == 0   ? 1                        //  Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
   return  n == 1  ? x                                //  Base case
       : n%2 == 0 ? (x = power_recurse(x, n/2)) * x  //  Even powe
       :              x * power_recurse(x, n-1);     //  Odd power
}
```

Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { // (Overflow is possible)
    return n < 0 ? power_recurse(1.0/x,-n)   // Negative power
        : n == 0  ? 1                         // Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
    return n == 1  ? x                                    // Base case
        : n%2 == 0 ? (x = power_recurse(x, n/2)) * x     // Even power
        :             x * power_recurse(x, n-1);          // Odd power
}
```

Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { // (Overflow is possible)
    return n < 0 ? power_recurse(1.0/x,-n)  // Negative power
        : n == 0  ? 1                       // Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
    return n == 1  ? x                                  // Base case
        : n%2 == 0 ? (x = power_recurse(x, n/2)) * x    // Even powe
        :            x * power_recurse(x, n-1);         // Odd power
}
```

Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { // (Overflow is possible)
    return n < 0 ? power_recurse(1.0/x,-n)  // Negative power
        : n == 0 ? 1                         // Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
    return n == 1 ? x                                  // Base case
        : n%2 == 0 ? (x = power_recurse(x, n/2)) * x  // Even powe
        :            x * power_recurse(x, n-1);        // Odd power
}
```

Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse

- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case

- That is,

  - ⋆ Ensure that you use a 'smaller problem'
  - ⋆ Assume that you can solve the 'smaller problem'

Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse
- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case
- That is,
  - ⋆ Ensure that you use a 'smaller problem'
  - ⋆ Assume that you can solve the 'smaller problem'

Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse
- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case
- That is,
  - ⋆ Ensure that you use a 'smaller problem'
  - ⋆ Assume that you can solve the 'smaller problem'

Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse

- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case

- That is,

  ⋆ Ensure that you use a 'smaller problem'
  ⋆ Assume that you can solve the 'smaller problem'

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - ⋆ returns a value or
  - ⋆ change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

Unrolling Recursion

- Recursion can frequently be replaced
- E.g. we can easily write a factorial function

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    long res = 1;
    for (int i=2; i<=n; i++)
        res *= i;
    return res;
}
```

with no function calls this will run much faster than the recursive version

Unrolling Recursion

- Recursion can frequently be replaced
- E.g. we can easily write a factorial function

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    long res = 1;
    for (int i=2; i<=n; i++)
        res *= i;
    return res;
}
```

with no function calls this will run much faster than the recursive version

Unrolling Recursion

- Recursion can frequently be replaced
- E.g. we can easily write a factorial function

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    long res = 1;
    for (int i=2; i<=n; i++)
        res *= i;
    return res;
}
```

with no function calls this will run much faster than the recursive version

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator

- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B

- E.g. the greatest common denominator of 70 and 25 is 5

- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$

- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator

- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B

- E.g. the greatest common denominator of 70 and 25 is 5

- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$

- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator

- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B

- E.g. the greatest common denominator of 70 and 25 is 5

- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$

- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\mathrm{gcd}(A, B) = \mathrm{gcd}(B, A \mod B)$
  - ⋆ $\mathrm{gcd}(A, 0) = A$
- Thus $\mathrm{gcd}(70, 25) = \mathrm{gcd}(25, 20) = \mathrm{gcd}(20, 5) = \mathrm{gcd}(5, 0) = 5$

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - ⋆ $\gcd(A, B) = \gcd(B, A \mod B)$
  - ⋆ $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

    if (b==0) {
        return a;
    }
    return gcd(b, a%b);
}
```

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

     if (b==0) {
        return a;
     }
     long c = a%b;
     a = b;
     b = c;
     return gcd(a, b);
}
```

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{
  while(true) {
     if (b==0) {
        return a;
     }
     long c = a%b;
     a = b;
     b = c;
   }
}
```

- Example of tail recursion

When Definitely not to Recurse

- A classic recursively defined sequence is the Fibonacci series
  - ⋆ $f_n = f_{n-1} + f_{n-2}$
  - ⋆ $f_1 = f_2 = 1$

- We might be tempted to write a recursive function to define the series

```
long fibonacci(long n)
{
  if (n<=2)
     return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Why shouldn't you want to do this?

When Definitely not to Recurse

- A classic recursively defined sequence is the Fibonacci series
  - ⋆ $f_n = f_{n-1} + f_{n-2}$
  - ⋆ $f_1 = f_2 = 1$

- We might be tempted to write a recursive function to define the series

```
long fibonacci(long n)
{
  if (n<=2)
     return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Why shouldn't you want to do this?

When Definitely not to Recurse

- A classic recursively defined sequence is the Fibonacci series
  - ⋆ $f_n = f_{n-1} + f_{n-2}$
  - ⋆ $f_1 = f_2 = 1$
- We might be tempted to write a recursive function to define the series

```
long fibonacci(long n)
{
  if (n<=2)
      return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Why shouldn't you want to do this?

Fibonacci

```
long fib(long n)
{
  if (n<=2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

Fibonacci

```
long fib(long n)
{
  if (n<=2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

```
fib(6)
```

Fibonacci

```
long fib(long n)
{
  if (n<=2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

$$\begin{tikzcd} \texttt{fib(6)} \arrow[d] \\ \texttt{fib(5)} \end{tikzcd}$$

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Outline

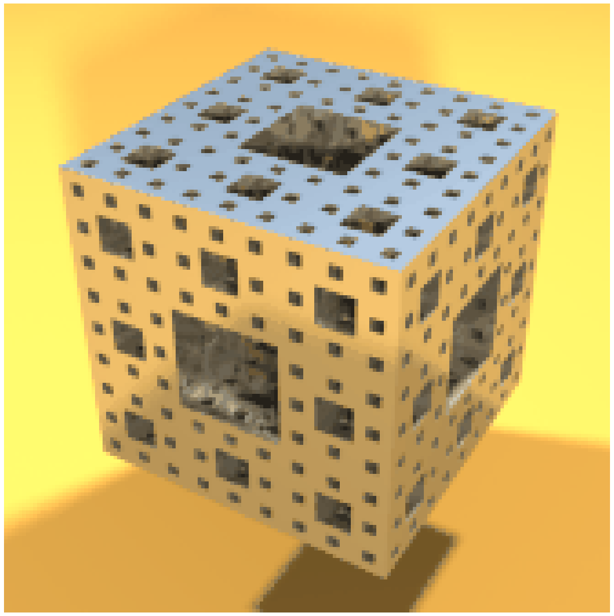
1. Simple Recursion

2. Programming Recursively

   - Simple Examples
   - Thinking about Recursion

3. **Analysis of Recursion**

   - Integer Powers
   - Towers of Hanoi



Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ ‘divides’ we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \le T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \le 2(m-1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - ⋆ $b_i \in \{0, 1\}$
  - ⋆ $b_m = 1$
  - ⋆ m is the number of digits in the binary representation of n
  - ⋆ $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - ⋆ After $m - 1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m - 1)$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \le n < 2^{m+1}$$

$$m \le \log_2(n) < m + 1$$

- But $T(n) \le 2(m-1) \le 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \le n < 2^{m+1}$$
$$m \le \log_2(n) < m + 1$$

- But $T(n) \le 2(m-1) \le 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$

$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m - 1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

Towers of Hanoi

In an ancient city, so the legend goes, monks in a temple had to move a pile of 64 sacred disks from one location to another. The disks were fragile; only one could be carried at a time. A disk could not be placed on top of a smaller, less valuable disk. In addition, there was only one other location in the temple (besides the original and destination locations) sacred enough for a pile of disks to be placed there.

Using the intermediate location, the monks began to move disks back and forth from the original pile to the pile at the new location, always keeping the piles in order (largest on the bottom, smallest on the top). According to the legend, before the monks could make the final move to complete the new pile in the new location, the temple would turn to dust and the world would end.

Towers of Hanoi

In an ancient city, so the legend goes, monks in a temple had to move a pile of 64 sacred disks from one location to another. The disks were fragile; only one could be carried at a time. A disk could not be placed on top of a smaller, less valuable disk. In addition, there was only one other location in the temple (besides the original and destination locations) sacred enough for a pile of disks to be placed there.

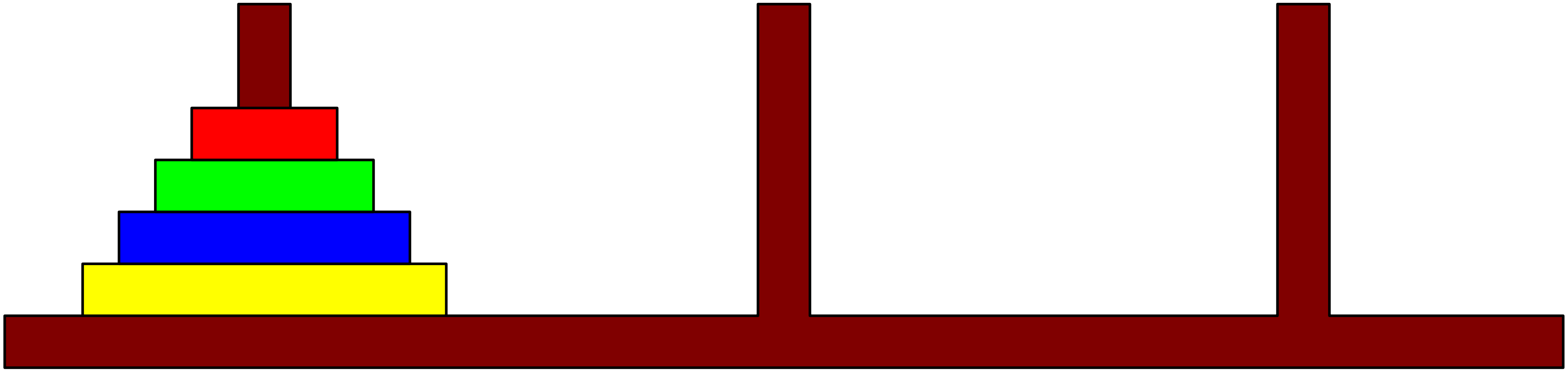
Using the intermediate location, the monks began to move disks back and forth from the original pile to the pile at the new location, always keeping the piles in order (largest on the bottom, smallest on the top). According to the legend, before the monks could make the final move to complete the new pile in the new location, the temple would turn to dust and the world would end.

A Smaller Tower of Hanoi

- Here is a smaller problem of just four disks

A Smaller Tower of Hanoi

- Here is a smaller problem of just four disks



Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Algorithms in the Real World

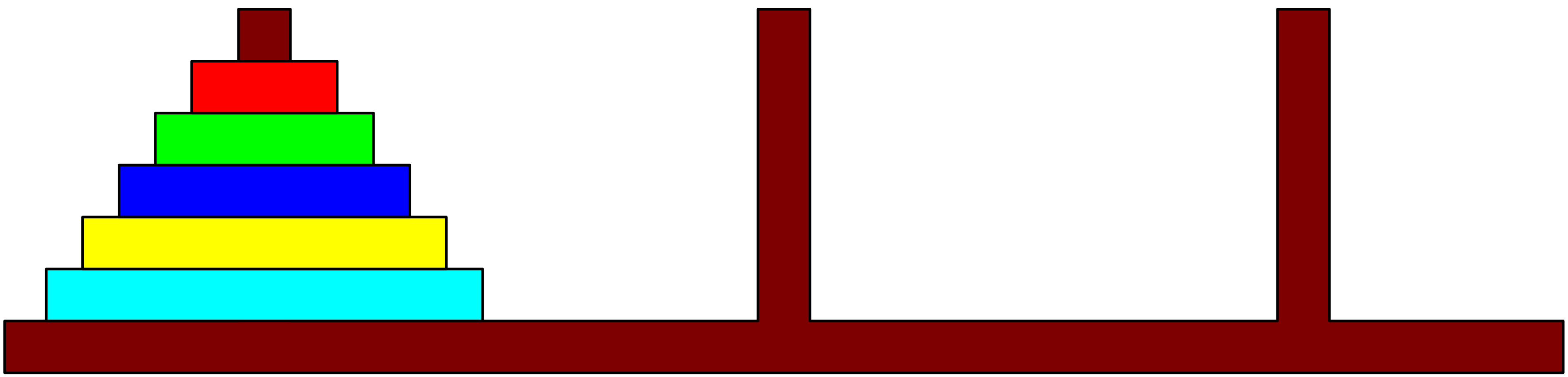
- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```

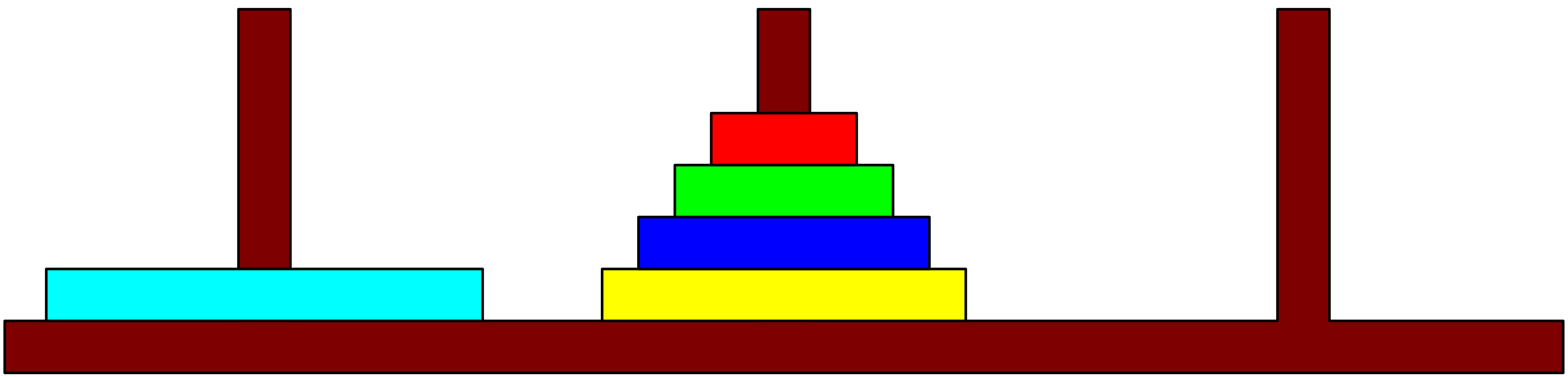
Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



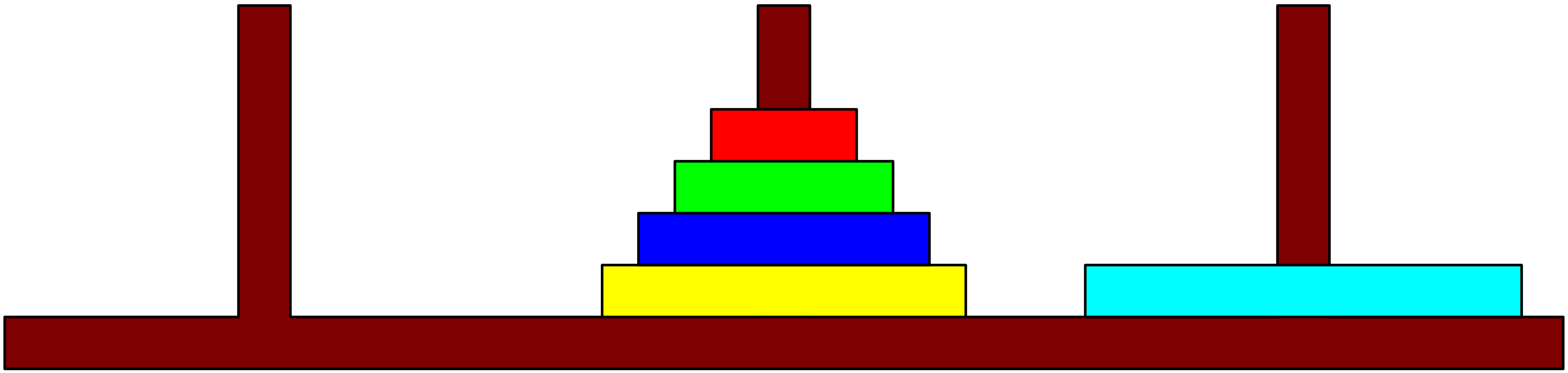
Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



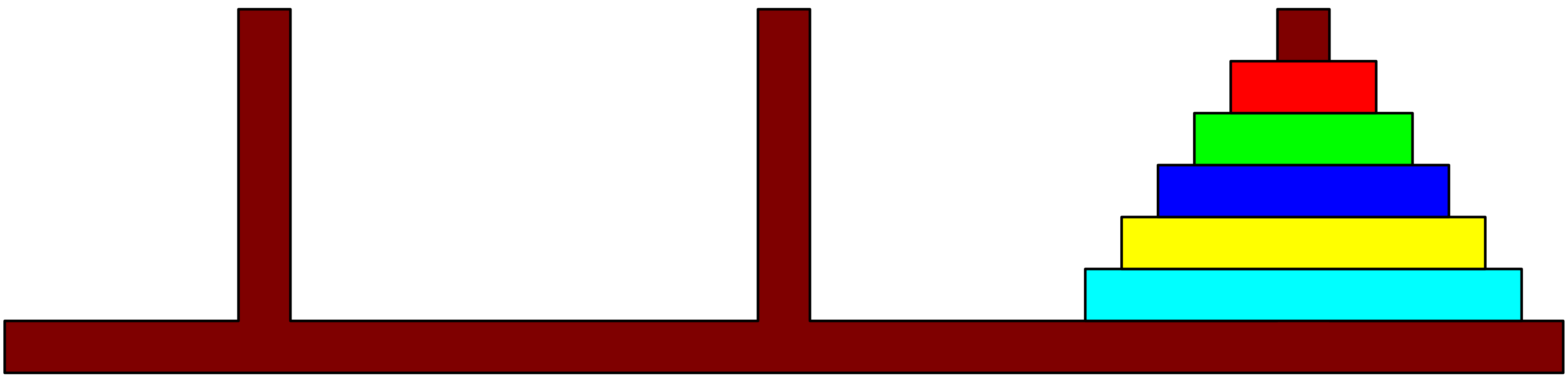
Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n - 1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

How Long Does It Take?

- How many moves does it take to transfer n disks?

How Long Does It Take?

- How many moves does it take to transfer n disks?

- We can use the same procedure as before

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```

How Long Does It Take?

- How many moves does it take to transfer n disks?
- We can use the same procedure as before

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```

  ⋆ $T(n) = 2\,T(n-1) + 1$
  ⋆ $T(0) = 0$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$
- $T(3) = 2 \times 3 + 1 = 6 + 1 = 7$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$
- $T(3) = 2 \times 3 + 1 = 6 + 1 = 7$
- $T(4) = 2 \times 7 + 1 = 14 + 1 = 15$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$
- $T(3) = 2 \times 3 + 1 = 6 + 1 = 7$
- $T(4) = 2 \times 7 + 1 = 14 + 1 = 15$
- Looks like $T(n) = 2^n - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case:

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$
$$T(n) = 2 \times (2^{n-1} - 1) + 1$$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$
$$T(n) = 2 \times (2^{n-1} - 1) + 1$$
$$T(n) = 2^n - 2 + 1$$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$
\begin{aligned}
T(n) &= 2\,T(n-1) + 1 \\
T(n) &= 2 \times (2^{n-1} - 1) + 1 \\
T(n) &= 2^n - 2 + 1 = 2^n - 1
\end{aligned}
$$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$
$$T(n) = 2 \times (2^{n-1} - 1) + 1$$
$$T(n) = 2^n - 2 + 1 = 2^n - 1 \checkmark$$

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)
- In case your interested, if it takes 1 second to move a disk it will take almost $585\,000\,000\,000$ years to move 64 disks

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!