AMT 223 1.0 Discrete Mathematics

Dr. G.H.J. Lanel

Semester 2 - 2018

Outline

1. Induction and Recursion
   - Recursive Definitions
   - Recursive Algorithms
   - Proof by Mathematical Induction

Recursion

Sometimes it is difficult to define an object explicitly. However, it may be easy to define this object in terms of itself. This process is called recursion. *(Rosen 2012, 7th ed. p. 344)*

Recursive Definitions-ctd

A recursive definition of a function defines values of the functions for some inputs in terms of the values of the same function for other inputs. For example, the factorial function n! is defined by the rules

$0! = 1$. and $(n+1)! = (n+1)n!$.

Recursive Definitions-ctd

A recursive definition of a function defines values of the functions for some inputs in terms of the values of the same function for other inputs. For example, the factorial function n! is defined by the rules

$0! = 1$. and $(n+1)! = (n+1)n!$.

- A recursive (or inductive) definition of an object X consists of two parts.
- The first part describes a few of the pieces of X, usually one, sometimes two or three, occasionally more, and on rare occasions, none. This part is called the base case of the definition.
- The second part of a recursive definition describes how new pieces are determined by other pieces already defined; this part is called the inductive (or recursive) part of the definition.

- A recursive (or inductive) definition of an object X consists of two parts.
- The first part describes a few of the pieces of X, usually one, sometimes two or three, occasionally more, and on rare occasions, none. This part is called the base case of the definition.
- The second part of a recursive definition describes how new pieces are determined by other pieces already defined; this part is called the inductive (or recursive) part of the definition.

- A recursive (or inductive) definition of an object X consists of two parts.
- The first part describes a few of the pieces of X, usually one, sometimes two or three, occasionally more, and on rare occasions, none. This part is called the base case of the definition.
- The second part of a recursive definition describes how new pieces are determined by other pieces already defined; this part is called the inductive (or recursive) part of the definition.

Example

- A sequence of natural numbers that arises again and again in discrete mathematics (and, apparently, in nature as well) was defined (recursively) by an Italian mathematician in the thirteenth century.
- This sequence is usually denoted $f_0, f_1, f_2, \ldots$ and is known as the **Fibonacci sequence.**
- We do not define the sequence by giving an explicit formula for the n^{th} term of the sequence.

  Instead, we define $\{f_n\}$ by stating explicitly what the first two terms in the sequence are, and then giving a formula which shows how each of the remaining terms is determined by terms that appear earlier in the sequence (the recursive part of the definition).

Example

- A sequence of natural numbers that arises again and again in discrete mathematics (and, apparently, in nature as well) was defined (recursively) by an Italian mathematician in the thirteenth century.
- This sequence is usually denoted $f_0, f_1, f_2, \ldots$ and is known as the **Fibonacci sequence**.
- We do not define the sequence by giving an explicit formula for the n^{th} term of the sequence.

  Instead, we define $\{f_n\}$ by stating explicitly what the first two terms in the sequence are, and then giving a formula which shows how each of the remaining terms is determined by terms that appear earlier in the sequence (the recursive part of the definition).

Example

- A sequence of natural numbers that arises again and again in discrete mathematics (and, apparently, in nature as well) was defined (recursively) by an Italian mathematician in the thirteenth century.

- This sequence is usually denoted $f_0, f_1, f_2, \ldots$ and is known as the **Fibonacci sequence**.

- We do not define the sequence by giving an explicit formula for the n^{th} term of the sequence.

  Instead, we define $\{f_n\}$ by stating explicitly what the first two terms in the sequence are, and then giving a formula which shows how each of the remaining terms is determined by terms that appear earlier in the sequence (the recursive part of the definition).

Example

- A sequence of natural numbers that arises again and again in discrete mathematics (and, apparently, in nature as well) was defined (recursively) by an Italian mathematician in the thirteenth century.
- This sequence is usually denoted $f_0, f_1, f_2, \ldots$ and is known as the **Fibonacci sequence**.
- We do not define the sequence by giving an explicit formula for the n^{th} term of the sequence.

  Instead, we define $\{f_n\}$ by stating explicitly what the first two terms in the sequence are, and then giving a formula which shows how each of the remaining terms is determined by terms that appear earlier in the sequence (the recursive part of the definition).

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1,$
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1$,
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1,$
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
    - Base case: $f_0 = 1, f_1 = 1,$
    - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
    - The base case tells us what f_0 and f_1 are.
    - What about f_2?
        - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
        - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1$,
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1,$
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1$,
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- Specifically, we set
  - Base case: $f_0 = 1, f_1 = 1,$
  - Recursive part: $f_n = f_{n-1} + f_{n-2}$, for all $n \geq 2$
- Note how this definition in fact determines the entire sequence in a definite and unambiguous way.
  - The base case tells us what f_0 and f_1 are.
  - What about f_2?
    - The inductive part of the definition tells us that $f_2 = f_1 + f_0$
    - Since we already know f_1 and f_0, we can compute f_2, namely $f_2 = 1 + 1 = 2$.

- What about f_3?
  - By definition, $f_3 = f_2 + f_1$; but since we know f_2 because of the calculation we just finished, and since we know f_1 from the base case, we can compute that $f_3 = 2 + 1 = 3$.
- Obviously, we can continue in this way as long as we wish. finding successively that

$f_4 = 3 + 2 = 5$, $f_5 = 5 + 3 = 8$, $f_6 = 13$, $f_7 = 21$, $f_8 = 34$, and so on.

- What about f_3?
  - By definition, $f_3 = f_2 + f_1$; but since we know f_2 because of the calculation we just finished, and since we know f_1 from the base case, we can compute that $f_3 = 2 + 1 = 3$.
- Obviously, we can continue in this way as long as we wish. finding successively that

$f_4 = 3 + 2 = 5$, $f_5 = 5 + 3 = 8$, $f_6 = 13$, $f_7 = 21$, $f_8 = 34$, and so on.

- What about f_3?
  - By definition, $f_3 = f_2 + f_1$; but since we know f_2 because of the calculation we just finished, and since we know f_1 from the base case, we can compute that $f_3 = 2 + 1 = 3$.
- Obviously, we can continue in this way as long as we wish. finding successively that

$$f_4 = 3 + 2 = 5, f_5 = 5 + 3 = 8, f_6 = 13, f_7 = 21, f_8 = 34, \text{ and so on.}$$

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Algorithm: Generating the Fibonacci sequence.

procedure *iterative_fibonacci*(n : natural number)

This algorithm computes and stores as $f_o, f_i, .., f_n$ the first n + 1 terms of the Fibonacci sequence

$f_0 \leftarrow 1$

$f_1 \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$f_i \leftarrow f_{i-1} + f_{i-2}$

return(f_n)

Well-Formed Formulas

- An important use of recursive definitions in discrete mathematics and computer science is for defining sets of strings.
- In a programming language, for example, certain strings of symbols (letters, digits, punctuation marks, etc.) are valid variable names, expressions, statements, or programs, while others are not.
- Rules of syntax determine which strings are allowed.

Well-Formed Formulas

- An important use of recursive definitions in discrete mathematics and computer science is for defining sets of strings.
- In a programming language, for example, certain strings of symbols (letters, digits, punctuation marks, etc.) are valid variable names, expressions, statements, or programs, while others are not.
- Rules of syntax determine which strings are allowed.

Well-Formed Formulas

- An important use of recursive definitions in discrete mathematics and computer science is for defining sets of strings.
- In a programming language, for example, certain strings of symbols (letters, digits, punctuation marks, etc.) are valid variable names, expressions, statements, or programs, while others are not.
- Rules of syntax determine which strings are allowed.

Well-Formed Formulas

- An important use of recursive definitions in discrete mathematics and computer science is for defining sets of strings.
- In a programming language, for example, certain strings of symbols (letters, digits, punctuation marks, etc.) are valid variable names, expressions, statements, or programs, while others are not.
- Rules of syntax determine which strings are allowed.

- In most cases, rules of syntax can be described recursively.
- It is satisfying conceptually to give recursive definitions in this context.
- Furthermore, such definitions make it easier to write compilers to recognize strings that are valid programs in high-level programming languages and translate them into machine-language programs.

- In most cases, rules of syntax can be described recursively.
- It is satisfying conceptually to give recursive definitions in this context.
- Furthermore, such definitions make it easier to write compilers to recognize strings that are valid programs in high-level programming languages and translate them into machine-language programs.

- In most cases, rules of syntax can be described recursively.
- It is satisfying conceptually to give recursive definitions in this context.
- Furthermore, such definitions make it easier to write compilers to recognize strings that are valid programs in high-level programming languages and translate them into machine-language programs.

This is an example that occur in most high-level programming languages.

Example.

Suppose that a variable name is allowed to be any string of one or more characters, each of which is either a letter or a digit, the first of which must be a letter. We can describe the set V of all variable names as follows.

Base case:
If x is a letter, then x is a variable name.

Recursive part:
If α is a variable name and x is a letter or a digit, then αx is also a variable name.

This is an example that occur in most high-level programming languages.

Example.

Suppose that a variable name is allowed to be any string of one or more characters, each of which is either a letter or a digit, the first of which must be a letter. We can describe the set V of all variable names as follows.

Base case:
If x is a letter, then x is a variable name.

Recursive part:
If α is a variable name and x is a letter or a digit, then αx is also a variable name.

This is an example that occur in most high-level programming languages.

Example.

Suppose that a variable name is allowed to be any string of one or more characters, each of which is either a letter or a digit, the first of which must be a letter. We can describe the set V of all variable names as follows.

Base case:
If x is a letter, then x is a variable name.

Recursive part:
If α is a variable name and x is a letter or a digit, then αx is also a variable name.

This is an example that occur in most high-level programming languages.

Example.

Suppose that a variable name is allowed to be any string of one or more characters, each of which is either a letter or a digit, the first of which must be a letter. We can describe the set V of all variable names as follows.

Base case:
If x is a letter, then x is a variable name.

Recursive part:
If α is a variable name and x is a letter or a digit, then αx is also a variable name.

This is an example that occur in most high-level programming languages.

Example.

Suppose that a variable name is allowed to be any string of one or more characters, each of which is either a letter or a digit, the first of which must be a letter. We can describe the set V of all variable names as follows.

Base case:
If x is a letter, then x is a variable name.

Recursive part:
If α is a variable name and x is a letter or a digit, then αx is also a variable name.

- The first statement is the base case, and from this we get the valid variable names of length 1, such as W or M.
- The second statement is the inductive part of the definition.
- It tells us how to construct valid variable names from other valid variable names.
- Specifically, it tells us that we can take any valid variable name and concatenate onto the end of it any letter or digit.

- The first statement is the base case, and from this we get the valid variable names of length 1, such as W or M.
- The second statement is the inductive part of the definition.
- It tells us how to construct valid variable names from other valid variable names.
- Specifically, it tells us that we can take any valid variable name and concatenate onto the end of it any letter or digit.

- The first statement is the base case, and from this we get the valid variable names of length 1, such as W or M.
- The second statement is the inductive part of the definition.
- It tells us how to construct valid variable names from other valid variable names.
- Specifically, it tells us that we can take any valid variable name and concatenate onto the end of it any letter or digit.

- The first statement is the base case, and from this we get the valid variable names of length 1, such as W or M.
- The second statement is the inductive part of the definition.
- It tells us how to construct valid variable names from other valid variable names.
- Specifically, it tells us that we can take any valid variable name and concatenate onto the end of it any letter or digit.

- For example, since W is a valid variable name, so are $W8$ and WE.
- Then since $W8$ is a valid variable name, so is $W8R$.
- Our recursive definition tells us not only that certain elements are in the set we are defining, but also that *the only* elements in the set are the ones that are forced to be there by the definition, in other words, the objects that can be built up according to the rules given in the definition.

- For example, since W is a valid variable name, so are $W8$ and WE.
- Then since $W8$ is a valid variable name, so is $W8R$.
- Our recursive definition tells us not only that certain elements are in the set we are defining, but also that *the only* elements in the set are the ones that are forced to be there by the definition, in other words, the objects that can be built up according to the rules given in the definition.

- For example, since W is a valid variable name, so are $W8$ and WE.
- Then since $W8$ is a valid variable name, so is $W8R$.
- Our recursive definition tells us not only that certain elements are in the set we are defining, but also that *the only* elements in the set are the ones that are forced to be there by the definition, in other words, the objects that can be built up according to the rules given in the definition.

Recursive Algorithms

- A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.
- More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.
- For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

Recursive Algorithms

- A recursive algorithm is an algorithm which calls itself with ”smaller (or simpler)” input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.
- More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.
- For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

Recursive Algorithms

- A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.
- More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.
- For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

Recursive Algorithms-ctd

Example 1: Algorithm for finding the k-th even natural number Note here that this can be solved very easily by simply outputting $2 * (k - 1)$ for a given k.

The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Even (positive integer k)

Input: k , a positive integer

Output: k-th even natural number (the first even being 0)

Algorithm: if $k = 1$, then return 0; else return Even $(k - 1) * 2$.

The Towers of Hanoi Puzzle

- Now we consider an old puzzle called the towers of Hanoi. Stripped of its mystical problem is as follows.
- You (the person working the puzzle) are presented with three tall pegs sticking up from a solid base.
- On one of the pegs stands a tower of n solid disks with holes in their centers, all of different diameters.
- No disk sits on a disk of smaller diameter, so the stack of disks on the peg looks like a cone, wide at the bottom and narrow at the top.

The Towers of Hanoi Puzzle

- Now we consider an old puzzle called the towers of Hanoi. Stripped of its mystical problem is as follows.
- You (the person working the puzzle) are presented with three tall pegs sticking up from a solid base.
- On one of the pegs stands a tower of n solid disks with holes in their centers, all of different diameters.
- No disk sits on a disk of smaller diameter, so the stack of disks on the peg looks like a cone, wide at the bottom and narrow at the top.

The Towers of Hanoi Puzzle

- Now we consider an old puzzle called the towers of Hanoi. Stripped of its mystical problem is as follows.
- You (the person working the puzzle) are presented with three tall pegs sticking up from a solid base.
- On one of the pegs stands a tower of n solid disks with holes in their centers, all of different diameters.
- No disk sits on a disk of smaller diameter, so the stack of disks on the peg looks like a cone, wide at the bottom and narrow at the top.

The Towers of Hanoi Puzzle

- Now we consider an old puzzle called the towers of Hanoi. Stripped of its mystical problem is as follows.
- You (the person working the puzzle) are presented with three tall pegs sticking up from a solid base.
- On one of the pegs stands a tower of n solid disks with holes in their centers, all of different diameters.
- No disk sits on a disk of smaller diameter, so the stack of disks on the peg looks like a cone, wide at the bottom and narrow at the top.

Following figure shows this initial position when $n = 5$.

We label the pegs A, B, and C, as shown, and we label the disks 1 to n, from smallest to largest.

Following figure shows this initial position when $n = 5$.

We label the pegs A, B, and C, as shown, and we label the disks 1 to n, from smallest to largest.

- Your task is to move the disks so that the whole stack, which began on peg A, ends up on peg B.
- Two rules must be followed.
  - First, you can move only one at a time, removing it from the top of the stack on its current peg and placing it on top of the stack on some other peg.
  - Second, a disk may never be placed on top of a smaller disk.

- Your task is to move the disks so that the whole stack, which began on peg A, ends up on peg B.
- Two rules must be followed.
  - First, you can move only one at a time, removing it from the top of the stack on its current peg and placing it on top of the stack on some other peg.
  - Second, a disk may never be placed on top of a smaller disk.

- Your task is to move the disks so that the whole stack, which began on peg A, ends up on peg B.
- Two rules must be followed.
  - First, you can move only one at a time, removing it from the top of the stack on its current peg and placing it on top of the stack on some other peg.
  - Second, a disk may never be placed on top of a smaller disk.

- Your task is to move the disks so that the whole stack, which began on peg A, ends up on peg B.
- Two rules must be followed.
  - First, you can move only one at a time, removing it from the top of the stack on its current peg and placing it on top of the stack on some other peg.
  - Second, a disk may never be placed on top of a smaller disk.

- Your task is to move the disks so that the whole stack, which began on peg A, ends up on peg B.
- Two rules must be followed.
  - First, you can move only one at a time, removing it from the top of the stack on its current peg and placing it on top of the stack on some other peg.
  - Second, a disk may never be placed on top of a smaller disk.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

The problem is to solve the puzzle:

Find a sequence of moves that will result in the entire stack of n disks standing on peg B. As a secondary problem, we might ask how many moves the most efficient algorithm will take to perform this task.

- This problem is ideal for illustrating the recursive approach.
- All we have to do is reduce the problem to a simpler problem.
- Let $hanoi(X, Y, Z, n)$ be the algorithm which, we hope, solve the problem just stated.

- We will assume that a solution to the problem consisting of printing a sequence of statements of the form ”Move disk i from peg p to peg q.”
- If, for example, we call $hanoi(A, B, C, 2)$, then we hope to see the output:

  Move disk 1 from peg A to peg C.

  Move disk 2 from peg A to peg B.

  Move disk 1 from peg C to peg B.

- We will assume that a solution to the problem consisting of printing a sequence of statements of the form ”Move disk i from peg p to peg q.”
- If, for example, we call $hanoi(A, B, C, 2)$, then we hope to see the output:

  Move disk 1 from peg A to peg C.

  Move disk 2 from peg A to peg B.

  Move disk 1 from peg C to peg B.

- We will assume that a solution to the problem consisting of printing a sequence of statements of the form "Move disk i from peg p to peg q."
- If, for example, we call $hanoi(A, B, C, 2)$, then we hope to see the output:

  Move disk 1 from peg A to peg C.

  Move disk 2 from peg A to peg B.

  Move disk 1 from peg C to peg B.

- We will assume that a solution to the problem consisting of printing a sequence of statements of the form ”Move disk i from peg p to peg q.”
- If, for example, we call $hanoi(A, B, C, 2)$, then we hope to see the output:

  Move disk 1 from peg A to peg C.

  Move disk 2 from peg A to peg B.

  Move disk 1 from peg C to peg B.

- We will assume that a solution to the problem consisting of printing a sequence of statements of the form ”Move disk i from peg p to peg q.”
- If, for example, we call $hanoi(A, B, C, 2)$, then we hope to see the output:

  Move disk 1 from peg A to peg C.

  Move disk 2 from peg A to peg B.

  Move disk 1 from peg C to peg B.

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*$(X, Y, Z$: peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*$(X, Z, Y, n - 1)$
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*$(Z, Y, X, n - 1)$
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n-1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n-1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*$(X, Z, Y, n-1)$
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*$(Z, Y, X, n-1)$
end
return

Algorithm:
Recursive solution to the towers of Hanoi puzzle

procedure *hanoi*(X, Y, Z : peg names, n : positive integer)

{this procedure prints out in order the moves needed to transfer n disks from peg X to peg Y, following the rules of the towers of Hanoi puzzle; the peg names X, Y, and Z must be A, B, and C, in some order}

if $n = 1$ **then print**("Move disk 1 from peg" X "to peg" Y".")
else
begin
call *hanoi*($X, Z, Y, n - 1$)
print("Move disk" n "from peg" X "to peg" Y".")
call *hanoi*($Z, Y, X, n - 1$)
end
return

Recursion Versus Iteration

- We discussed recursive definitions of functions (or sequences, which are really just functions) and gave many examples.
- It is usually straightforward to take a recursive definition of a function and turn it into a recursive procedure for computing the function.
- Recall the recursive definition of the Fibonacci sequence, which we write in functional notation to suit our needs in this example:

$f(0) = f(1) = 1$ and $f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

Recursion Versus Iteration

- We discussed recursive definitions of functions (or sequences, which are really just functions) and gave many examples.
- It is usually straightforward to take a recursive definition of a function and turn it into a recursive procedure for computing the function.
- Recall the recursive definition of the Fibonacci sequence, which we write in functional notation to suit our needs in this example:

  $f(0) = f(1) = 1$ and $f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

Recursion Versus Iteration

- We discussed recursive definitions of functions (or sequences, which are really just functions) and gave many examples.
- It is usually straightforward to take a recursive definition of a function and turn it into a recursive procedure for computing the function.
- Recall the recursive definition of the Fibonacci sequence, which we write in functional notation to suit our needs in this example:

$f(0) = f(1) = 1$ and $f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

Recursion Versus Iteration

- We discussed recursive definitions of functions (or sequences, which are really just functions) and gave many examples.
- It is usually straightforward to take a recursive definition of a function and turn it into a recursive procedure for computing the function.
- Recall the recursive definition of the Fibonacci sequence, which we write in functional notation to suit our needs in this example:

  $f(0) = f(1) = 1$ and $f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

Recursion Versus Iteration

- We discussed recursive definitions of functions (or sequences, which are really just functions) and gave many examples.
- It is usually straightforward to take a recursive definition of a function and turn it into a recursive procedure for computing the function.
- Recall the recursive definition of the Fibonacci sequence, which we write in functional notation to suit our needs in this example:

  $f(0) = f(1) = 1$ and $f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

The **recursive procedure** for computing f is short and simple.

procedure $f(n$: natural number)
{this procedure computes the value of $f(n)$ in the Fibonacci sequence, recursively from the definition}
if $n < 2$ **then return** (1) **else return**$(f(n-1) + f(n-2))$

The **recursive procedure** for computing f is short and simple.

procedure $f(n :$ natural number$)$
$\{$this procedure computes the value of $f(n)$ in the Fibonacci sequence, recursively from the definition$\}$
if $n < 2$ **then return** (1) **else return**$(f(n-1) + f(n-2))$

Iterative computation of the Fibonacci sequence.

procedure iterative $f(n:$ natural number)
{this procedure computes the value of $f(n)$ in the Fibonacci sequence, iteratively, using $O(1)$ space}
if $n < 2$ **then return** (1)
else
begin
$y \leftarrow 1$ {the last number in the sequence}
$x \leftarrow 1$ {the number in the sequence before y}
for $i \leftarrow 2$ **to** n **do**
begin
$z \leftarrow x + y$ {the next number in the sequence}
$x \leftarrow y$
$y \leftarrow z$
end
return(z)
end

Iterative computation of the Fibonacci sequence.

procedure iterative $f(n$: natural number)
{this procedure computes the value of $f(n)$ in the Fibonacci sequence, iteratively, using $O(1)$ space}
if $n < 2$ **then return** (1)
else
begin
$y \leftarrow 1$ {the last number in the sequence}
$x \leftarrow 1$ {the number in the sequence before y}
for $i \leftarrow 2$ **to** n **do**
begin
$z \leftarrow x + y$ {the next number in the sequence}
$x \leftarrow y$
$y \leftarrow z$
end
return(z)
end

Proof by Mathematical Induction

- Induction is the primary way we prove universal truths about entities of unbounded size (like natural numbers).
- If the size is bounded, then we can do proof by cases.
- Induction is also the way we define things about entities of unbounded size.

Proof by Mathematical Induction

- Induction is the primary way we prove universal truths about entities of unbounded size (like natural numbers).
- If the size is bounded, then we can do proof by cases.
- Induction is also the way we define things about entities of unbounded size.

Proof by Mathematical Induction

- Induction is the primary way we prove universal truths about entities of unbounded size (like natural numbers).
- If the size is bounded, then we can do proof by cases.
- Induction is also the way we define things about entities of unbounded size.

Principle of Mathematical Induction

Let $P(n)$ be an infinite collection of statements with $n \in N$. Suppose that

- $P(1)$ is true, and
- $P(k) \Longrightarrow P(k+1), \forall k \in N$.

Then, $P(n)$ is true $\forall n \in N$.

When constructing the proof by induction, you need to present the statement $P(n)$ and then follow three simple steps.

- **INDUCTION BASE:**
  check if $P(1)$ is true, i.e. the statement holds for $n = 1$,
- **INDUCTION HYPOTHESIS:**
  assume $P(k)$ is true, i.e. the statement holds for $n = k$,
- **INDUCTION STEP:**
  show that if $P(k)$ holds, then $P(k+1)$ also does.

When constructing the proof by induction, you need to present the statement $P(n)$ and then follow three simple steps.

- **INDUCTION BASE:**
  check if $P(1)$ is true, i.e. the statement holds for $n = 1$,
- **INDUCTION HYPOTHESIS:**
  assume $P(k)$ is true, i.e. the statement holds for $n = k$,
- **INDUCTION STEP:**
  show that if $P(k)$ holds, then $P(k + 1)$ also does.

When constructing the proof by induction, you need to present the statement $P(n)$ and then follow three simple steps.

- **INDUCTION BASE:**
  check if $P(1)$ is true, i.e. the statement holds for $n = 1$,
- **INDUCTION HYPOTHESIS:**
  assume $P(k)$ is true, i.e. the statement holds for $n = k$,
- **INDUCTION STEP:**
  show that if $P(k)$ holds, then $P(k+1)$ also does.

When constructing the proof by induction, you need to present the statement $P(n)$ and then follow three simple steps.

- **INDUCTION BASE:**
  check if $P(1)$ is true, i.e. the statement holds for $n = 1$,
- **INDUCTION HYPOTHESIS:**
  assume $P(k)$ is true, i.e. the statement holds for $n = k$,
- **INDUCTION STEP:**
  show that if $P(k)$ holds, then $P(k+1)$ also does.

Dominoes Effect

- Induction is often compared to dominoes toppling.
- When we push the first domino, all consecutive ones will also fall (provided each domino is close enough to its neighbour).
- Similarly with *P(1)* being true, it can be shown by induction that also $P(2)$, $P(3)$, $P(4)$, ... and so on, will be true.
- Hence we prove P(n) for infinite n.

Dominoes Effect

- Induction is often compared to dominoes toppling.
- When we push the first domino, all consecutive ones will also fall (provided each domino is close enough to its neighbour).
- Similarly with *P(1)* being true, it can be shown by induction that also $P(2)$, $P(3)$, $P(4)$, ... and so on, will be true.
- Hence we prove P(n) for infinite n.

Dominoes Effect

- Induction is often compared to dominoes toppling.
- When we push the first domino, all consecutive ones will also fall (provided each domino is close enough to its neighbour).
- Similarly with *P(1)* being true, it can be shown by induction that also $P(2), P(3), P(4), \ldots$ and so on, will be true.
- Hence we prove P(n) for infinite n.

Dominoes Effect

- Induction is often compared to dominoes toppling.
- When we push the first domino, all consecutive ones will also fall (provided each domino is close enough to its neighbour).
- Similarly with *P(1)* being true, it can be shown by induction that also $P(2), P(3), P(4), \ldots$ and so on, will be true.
- Hence we prove P(n) for infinite n.

Versions of induction

Principle of Strong Mathematical Induction

Let $P(n)$ be an infinite collection of statements with $n, r, k \in N$ and $r \leq k$. Suppose that

- $P(r)$ is true, and
- $P(j) \Longrightarrow P(k+1), \forall r \leq j \leq k$.

Then, $P(n)$ is true $\forall n \in N, n \geq r$

Examples

Show that $2^{3n+1} + 5$ is always a multiple of 7.

Solution:
The statement P(n) : 23n+1 + 5 is always a multiple of 7

BASE (n=1):
$2^{3\times1+1} + 5 = 2^4 + 5 = 16 + 5 = 21 = 7 \times 3$. Then $P(1)$ holds.

INDUCTION HYPOTHESIS:
Assume that $P(k)$ is true, so $2^{3k+1} + 5$ is always a multiple of 7, $k \in N$.

INDUCTION STEP:
Now, we want to show that $P(k) \implies P(k+1)$, where
$P(k+1) : 2^{3(k+1)} + 1 + 5 = 2^{3k+4} + 5$ is a multiple of 7.

Examples

Show that $2^{3n+1} + 5$ is always a multiple of 7.

Solution:
The statement P(n) : 23n+1 + 5 is always a multiple of 7

BASE (n=1):
$2^{3\times1+1} + 5 = 2^4 + 5 = 16 + 5 = 21 = 7 \times 3$. Then $P(1)$ holds.

INDUCTION HYPOTHESIS:
Assume that $P(k)$ is true, so $2^{3k+1} + 5$ is always a multiple of 7, $k \in N$.

INDUCTION STEP:
Now, we want to show that $P(k) \implies P(k+1)$, where
$P(k+1) : 2^{3(k+1)} + 1 + 5 = 2^{3k+4} + 5$ is a multiple of 7.

Examples

Show that $2^{3n+1} + 5$ is always a multiple of 7.

Solution:
The statement P(n) : 23n+1 + 5 is always a multiple of 7

BASE (n=1):
$2^{3\times1+1} + 5 = 2^4 + 5 = 16 + 5 = 21 = 7 \times 3$. Then $P(1)$ holds.

INDUCTION HYPOTHESIS:
Assume that $P(k)$ is true, so $2^{3k+1} + 5$ is always a multiple of 7, $k \in N$.

INDUCTION STEP:
Now, we want to show that $P(k) \implies P(k+1)$, where
$P(k+1) : 2^{3(k+1)} + 1 + 5 = 2^{3k+4} + 5$ is a multiple of 7.

Examples

Show that $2^{3n+1} + 5$ is always a multiple of 7.

Solution:
The statement P(n) : 23n+1 + 5 is always a multiple of 7

BASE (n=1):
$2^{3\times1+1} + 5 = 2^4 + 5 = 16 + 5 = 21 = 7 \times 3$. Then $P(1)$ holds.

INDUCTION HYPOTHESIS:
Assume that $P(k)$ is true, so $2^{3k+1} + 5$ is always a multiple of 7, $k \in N$.

INDUCTION STEP:
Now, we want to show that $P(k) \implies P(k+1)$, where
$P(k+1) : 2^{3(k+1)} + 1 + 5 = 2^{3k+4} + 5$ is a multiple of 7.

Examples

Show that $2^{3n+1} + 5$ is always a multiple of 7.

Solution:
The statement P(n) : 23n+1 + 5 is always a multiple of 7

BASE (n=1):
$2^{3\times1+1} + 5 = 2^4 + 5 = 16 + 5 = 21 = 7 \times 3$. Then $P(1)$ holds.

INDUCTION HYPOTHESIS:
Assume that $P(k)$ is true, so $2^{3k+1} + 5$ is always a multiple of 7, $k \in N$.

INDUCTION STEP:
Now, we want to show that $P(k) \Longrightarrow P(k+1)$, where
$P(k+1)$: $2^{3(k+1)} + 1 + 5 = 2^{3k+4} + 5$ is a multiple of 7.

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\implies (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\implies 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\implies 2^{3k+4} + 5 = 56x - 35$$

$$\implies 2^{3k+4} + 5 = 7(8x - 5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1}+5$ is always a multiple of 7, so we can write,

$$2^{3k+1}+5 = 7\times x$$

for some $x \in Z$

$$\implies (2^{3k+1}+5)\times 2^3 = 7\times x \times 2^3$$

$$\implies 2^{3k+4}+40 = 7\times x\times 8$$

$$\implies 2^{3k+4}+5 = 56x-35$$

$$\implies 2^{3k+4}+5 = 7(8x-5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\Longrightarrow (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\Longrightarrow 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\Longrightarrow 2^{3k+4} + 5 = 56x - 35$$

$$\Longrightarrow 2^{3k+4} + 5 = 7(8x - 5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\implies (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\implies 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\implies 2^{3k+4} + 5 = 56x - 35$$

$$\implies 2^{3k+4} + 5 = 7(8x - 5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\Longrightarrow (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\Longrightarrow 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\Longrightarrow 2^{3k+4} + 5 = 56x - 35$$

$$\Longrightarrow 2^{3k+4} + 5 = 7(8x - 5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\implies (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\implies 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\implies 2^{3k+4} + 5 = 56x - 35$$

$$\implies 2^{3k+4} + 5 = 7(8x - 5)$$

Example Ctd.

We know from induction hypothesis that $2^{3k+1} + 5$ is always a multiple of 7, so we can write,

$$2^{3k+1} + 5 = 7 \times x$$

for some $x \in Z$

$$\implies (2^{3k+1} + 5) \times 2^3 = 7 \times x \times 2^3$$

$$\implies 2^{3k+4} + 40 = 7 \times x \times 8$$

$$\implies 2^{3k+4} + 5 = 56x - 35$$

$$\implies 2^{3k+4} + 5 = 7(8x - 5)$$

So $2^{3k+4} + 5$ is a multiple by 7 ($P(k+1)$ holds), provided that $P(k)$ is true.

We have shown that $P(1)$ holds and if $P(k)$, then $P(k+1)$ is also true. Hence by the Principle of Mathematical Induction, it follows that $P(n)$ holds for all natural n.

So $2^{3k+4} + 5$ is a multiple by 7 ($P(k+1)$ holds), provided that $P(k)$ is true.

We have shown that $P(1)$ holds and if $P(k)$, then $P(k+1)$ is also true. Hence by the Principle of Mathematical Induction, it follows that $P(n)$ holds for all natural n.