Induction and recursion are related concepts.

**Induction** is a proof technique, **recursion** is a related programming concept.

Induction proof: assume it is true for $n$, and show it for $n+1$ using the assumption.

Recursion: assume you know how to solve the problem when the input size is $n$, and design a solution for the case $n+1$ using the solution for $n$.

Note that we could have used numbering as $(n-1)$ and $n$, instead of $n$ and $(n+1)$!!!
Topics

• Mathematical induction
• Well ordered property
• Second principle of induction
• Recursive definition
• Recursive algorithms

It may be good at this point to refresh yourself on predicates, predicate logic a.k.a. propositional functions. To do this go to my Lecture 2, handouts 02 Propositional Logic.pdf and start reading from page 59, (you may wish to skip 60-63), and continue by slide 64. Focus on slide 66.”

Important math task is to discover and characterize regular patterns or sequences.

The main mathematical tool to prove statements about sequences is induction.

Induction is also a very important tool in computer science because most programs are repetition of a sequence of statements.

Example on how induction works:

Imagine climbing an infinitely high ladder. How do you know whether you will be able to reach an arbitrarily high rung?

Suppose you make the following two assertions about your climbing:

1) I can definitely reach the first rung.
2) Once getting to any rung, I can always climb to the next one up.

If both statements are true, then by statement 1 you can get to the first one, and by statement 2, you can get to the second. By statement 2 again, you can get to the third, and fourth, etc.

Therefore, you can climb as high as you wish. Note: both of these assertions are necessary for you to get anywhere on the ladder.

If only statement 1 is true, you have no guarantee of getting beyond the first rung.

If only statement 2 is true, you may never be able to get started.

The rungs of the ladder are numbered with the positive integers (1,2,3,...). Assume a specific property that a number might have now. Instead of “reaching an arbitrarily high rung”, we can talk about an arbitrary positive integer having that property P.

We will use the shorthand P(n) to denote the positive integer n having property P. How can we use the ladder-climbing technique to prove that P(n) is true for all positive n?

The two assertions we need to prove are:

1) P(1) is true
2) for any positive k, if P(k) is true, then P(k+1) is true

Assertion 1 means we must show the property is true for 1; assertion 2 means that if any number has property P then so does the next number. If we can prove both of these statements, then P(n) holds for all positive integers, (which is same as climbing to an arbitrary rung of the ladder).

The Principle of Mathematical Induction can be used as a proof method on statements that have a particular form, and it can be stated as follows:

A proof by mathematical induction that a proposition P(n) is true for every positive integer n consists of two steps:

BASIS CASE: Show that the proposition P(1) is true.

INDUCTIVE STEP: Assume that P(k) is true for an arbitrarily chosen positive integer k, and show that under that assumption, P(k+1) must be true.

From these two steps we conclude (by the principle of mathematical induction) that for all positive integers n, P(n) is true.

Note: we don't prove that P(k) is true (except for k = 1). Instead, we show that if P(k) is true, then P(k+1) must also be true. That's all that is necessary according to the Principle of Mathematical Induction. The assumption that P(k) is true is called the induction hypothesis.

Understand that P(n) and P(k) are not numbers; they are the propositions that are true or false.
4.1 Mathematical Induction

• A powerful, rigorous technique for proving that a predicate \( P(n) \) is true for every natural number \( n \), no matter how large.
• Essentially a “domino effect” principle.
• Based on a predicate-logic inference rule:

\[
\begin{align*}
P(0) & \quad \text{Basis step} \\
\forall k \geq 0 \ (P(k) \rightarrow P(k+1)) & \quad \text{Inductive step} \\
\therefore \forall n \geq 0 \ P(n) & \quad \text{"The First Principle of Mathematical Induction"}
\end{align*}
\]

The “Domino Effect”

• Premise #1: Domino #1 falls.
• Premise #2: For every \( n \in \mathbb{N} \), if domino \#n falls, then so does domino \#n+1.
• Conclusion: All of the dominoes fall down!

Validity of Induction

• Proof that \( \forall k \geq 0 \ P(k) \) is a valid consequent: Given any \( k \geq 0 \), the 2\textsuperscript{nd} antecedent \( \forall n \geq 0 \ (P(n) \rightarrow P(n+1)) \) trivially implies that \( \forall n \geq 0 \ (n \geq k \rightarrow (P(n) \rightarrow P(n+1))) \), i.e., that \( (P(0) \rightarrow P(1)) \land (P(1) \rightarrow P(2)) \land \ldots \land (P(k-1) \rightarrow P(k)) \). Repeatedly applying the hypothetical syllogism rule to adjacent implications in this list \( k-1 \) times then gives us \( P(0) \rightarrow P(k) \); which together with \( P(0) \) (antecedent #1) and \textit{modus ponens} gives us \( P(k) \). Thus \( \forall k \geq 0 \ P(k) \).

\textit{Modus ponens} or \textit{MP} is an abbreviation of \textit{modus ponendo ponens} which is an old Latin saying standing for "the way that affirms by affirming"

It is not a \textit{logical law}, it is, rather one of the accepted mechanisms for the construction of proofs.

Modus ponens is a very common way to make conclusions in classical logic, i.e., it’s a \textit{rule of inference}, and it goes as follows:

\[
\begin{align*}
\text{If } A \text{, then } B. \\
A. \\
\text{Therefore, } B.
\end{align*}
\]
**Example 1:**

Let \( P \) be the proposition that the sum of the first \( n \) odd numbers is \( n^2 \); that is,

\[
P(n): 1 + 3 + 5 + \ldots + (2n - 1) = n^2
\]

**Proof**

(The \( n \)th odd number is \( 2n - 1 \), and the next odd number is \( 2n + 1 \).) Observe that \( P(n) \) is true for \( n = 1 \), that is,

\[
P(1): 1 = 1^2
\]

Assuming \( P(n) \) is true, we add \( 2n + 1 \) to both sides of \( P(n) \), obtaining

\[
1 + 3 + 5 + \ldots + (2n - 1) + (2n + 1) = n^2 + (2n + 1) = (n + 1)^2
\]

which is \( P(n + 1) \). That is, \( P(n + 1) \) is true whenever \( P(n) \) is true. By the principle of mathematical induction, \( P \) is true for all \( n \).

A little more formally, same proof goes as follows

---

**Example 2:**

**Theorem.** \( \forall n > 0, n < 2^n \). Prove it!

**Proof.** Let \( P(n) = (n < 2^n) \)

- **Base case:** \( P(1) = (1 < 2^1) = (1 < 2) \Rightarrow T. \)

- **Inductive step:** For \( n > 0 \), prove \( P(n) \to P(n+1) \).
  - Assuming \( n < 2^n \), prove \( n+1 < 2^{n+1} \).
    - Note \( n + 1 < 2^n + 1 \) (by inductive hypothesis)
      \[
      < 2^n + 2^n = 2 \cdot 2^n
      \]
      \[
      < 2^{n+1}
      \]
    - So \( n + 1 < 2^{n+1} \), and we’re done.
Steps in doing an inductive proof:

1) state the theorem, which is the proposition P(n)
2) prove that P(base case) is true
3) state the inductive hypothesis (substitute k for n)
4) state what must be proven (substitute k+1 for n)
5) state that you are beginning your proof of the inductive step, and proceed to manipulate the inductive hypothesis (which we assume is true) to find a link between the inductive hypothesis and the statement to be proven. Always state explicitly where you are invoking the inductive hypothesis.
6) Always finish your proof with something like: P(k+1) is true when P(k) is true, and therefore P(n) is true for all natural numbers.

Example 3: Prove that for every positive integer n, the sum of the first n positive integers is \( \frac{n(n+1)}{2} \). This is the classic example of an inductive proof.

Theorem. The following proposition is true for all positive integers P(n): 1+2+3+...+n = \( \frac{n(n+1)}{2} \)

BASE CASE: P(1) asserts that 1 = 1(1+1)/2 = 1, which is true.

INDUCTIVE STEP:
Assume for some integer k, P(k): 1+2+3+...+k = \( \frac{k(k+1)}{2} \)
Show: P(k+1): 1+2+3+...+(k+1) = \( \frac{(k+1)(k+2)}{2} \)

Proof of the Inductive Step:
By the induction hypothesis, we already have a formula for the first k integers. So, a formula for the first (k+1) integers may be found by simply adding (k+1) to both sides of the induction hypothesis, and simplifying:

\[
1+2+3+...+k(k+1) = \frac{k(k+1)}{2} + (k+1)
= \frac{k(k+1) + 2(k+1)}{2}
= \frac{(k+1)(k+2)}{2}
= \frac{1}{2}((k+1)(k+2))
\]

Thus P(k+1) is true when P(k) is true, and therefore P(n) is true for all natural numbers.

Generalizing Induction

- Rule can also be used to prove \( \forall n \geq c \ P(n) \) for a given constant \( c \in \mathbb{Z} \), where maybe \( c \neq 0 \).
  - In this circumstance, the base case is to prove P(c) rather than P(0), and the inductive step is to prove \( \forall n \geq c \ (P(n) \rightarrow P(n+1)) \).

- Induction can also be used to prove \( \forall n \geq c \ P(a_n) \) for any arbitrary series \( \{a_n\} \).
  - Can reduce these to the form already shown.

Until now we have been using the so-called weak induction. There is a variation called strong induction. Rather than assuming that P(k) is true to prove that P(k+1) is true, we assume that P(i) is true for all i where (basis of induction) \( \leq i \leq k \).

From this assumption, we prove P(k+1). It’s stronger in the sense that we are allowed to come to the same conclusion while assuming more, but the assumption is a natural one based on our understanding of weak induction. In fact, weak induction and strong induction are logically equivalent. That is, assuming either one is a valid rule of inference, we can show that the other is.

Strong or Complete Induction:

BASE CASE: Prove P(base) is true
INDUCTION: Assume P(base), P(base+1)...P(k) are true, and prove that P(k+1) is true.

More formal statement is on next slide.
Second Principle of Induction

a.k.a. “Strong Induction”

• Characterized by another inference rule:
  \[ P(0) \quad \forall n \geq 0: (\forall 0 \leq i \leq n, P(i)) \to P(n+1) \]
  \[ \therefore \forall n \geq 0: P(n) \]

• The only difference between this and the 1st principle is that:
  – the inductive step here makes use of the stronger hypothesis that \( P(i) \) is true for all smaller numbers \( i < n+1 \), not just for \( i = n \).

Example 4 - Second Principle

• Show that every \( n > 1 \) can be written as a product \( \prod p_i = p_1 p_2 \ldots p_s \) of some series of \( s \) prime numbers.
  – Let \( P(n) \) mean “\( n \) has that property”

• Base case: \( n=2 \), let \( s=1 \), \( p_1=2 \).
• Inductive step: Let \( n \geq 2 \). Assume \( \forall 2 \leq k \leq n: P(k) \).
  Consider \( n+1 \). If it's prime, let \( s=1 \), \( p_1=n+1 \). Else \( n+1=ab \), where \( 1 < a \leq n \) and \( 1 < b \leq n \).
  Then \( a=p_1 p_2 \ldots p_t \) and \( b=q_1 q_2 \ldots q_u \). Then we have that \( n+1 = p_1 p_2 \ldots p_t q_1 q_2 \ldots q_u \), a product of \( s=t+u \) primes.

Thus, if \( k + 1 \) is composite, it can be written as the product of primes, namely, those primes in the factorization of \( a \) and those in the factorization of \( b \).

Example 5 - Second Principle

A jigsaw puzzle consists of a number of pieces. Two or more pieces with matched boundaries can be put together to form a “big” piece. To be more precise, we use the term block to refer to either a single piece or a number of pieces with matched boundaries that are put together to form a “big” piece. Thus, we can simply say that blocks with matched boundaries can be put together to form another block. Finally, when all pieces are put together as one single block, the jigsaw puzzle is solved. Putting 2 blocks together with matched boundaries is called one move. We shall prove (using strong induction) that for a jigsaw puzzle of \( n \) pieces, it will always take \( n-1 \) moves to solve the puzzle.

BASE CASE: \( P(1) \) is true--for a puzzle with 1 piece, it does not take any moves to solve it.

INDUCTIVE STEP:
Assume \( P(i) \) where \( 1 \leq i \leq k \): for a puzzle with \( i \) pieces, it takes \( i-1 \) moves to solve the puzzle.
Show that for a puzzle with \( k+1 \) pieces, it takes \( k \) moves to solve the puzzle.
Proof of the Inductive Step:
Consider the puzzle with \( k+1 \) pieces. For the last move that produces the solution to the puzzle, we have two blocks: one with \( n_1 \) pieces and the other with \( n_2 \) pieces, where \( n_1 + n_2 = k + 1 \). These two blocks will then be put together to solve the puzzle. According to the induction hypothesis, it took \( n_1 - 1 \) moves to put together the one block, and \( n_2 - 1 \) moves to put together the other block. Including the last move to unite the two blocks, the total number of moves is equal to
\[
[(n_1 - 1) + (n_2 - 1)] + 1 = (k + 1) - 1 = k
\]
P\((k+1)\) is true when P\((i)\) is true, where \( i \leq k \), and therefore P\((n)\) is true for any puzzle size.

One more example: A Comparison

Strong induction vs. non-strong induction

• Determine which amounts of postage can be written with 5 and 6 cent stamps

  – Prove using both versions of induction

• Answer: any postage \( \geq 20 \)

This one can also be phrased closer to previous pages format:

Prove that any postage \( \geq 20 \) can be written with 5 & 6 cent stamps

Proof by weak induction

• Show base case: P\((20)\):
  – \( 20 = 5 + 5 + 5 + 5 \)

• Inductive hypothesis: Assume P\((k)\) is true

• Inductive step: Show that P\((k+1)\) is true
  – If P\((k)\) uses a 5 cent stamp, replace one stamp with a 6 cent stamp
  – If P\((k)\) does not use a 5 cent stamp, it must use only 6 cent stamps
    • Since \( k \geq 18 \), there must be four 6 cent stamps
    • Replace these with five 5 cent stamps to obtain \( k+1 \)

Proof by strong induction

• Show base cases: P\((20)\), P\((21)\), P\((22)\), P\((23)\), and P\((24)\)
  – \( 20 = 5 + 5 + 5 + 5 \)
  – \( 21 = 5 + 5 + 5 + 6 \)
  – \( 22 = 5 + 5 + 6 + 6 \)
  – \( 23 = 5 + 6 + 6 + 6 \)
  – \( 24 = 6 + 6 + 6 + 6 \)

• Inductive hypothesis: Assume P\((20)\), P\((21)\), …, P\((k)\) are all true

• Inductive step: Show that P\((k+1)\) is true
  – We will obtain P\((k+1)\) by adding a 5 cent stamp to P\((k+1-5)\)
  – Since we know P\((k+1-5) = P(k-4)\) is true, our proof is complete
The validity of mathematical induction follows from the Well-Ordering Property (WOP), which is a fundamental axiom of number theory. WOP states that every nonempty set of non-negative integers has a least element. This axiom can be used directly in proofs of theorems relating to sets of integers.

The Well-Ordering Property

- **Well-ordering property Axiom** says that:
  - Every non-empty set of non-negative integers has a minimal (smallest, least) element.
    \[ \forall \emptyset \subseteq \mathbb{N} : \exists m \in S : \forall n \in S : m \leq n \]
  - and thus, WOP proves that the Induction is valid because
  - This implies that \{n|\neg P(n)\} (if non-empty) has a min. element \(m\), but then the assumption that \(P(m-1) \rightarrow P(m-1)+1\) would be contradicted.
    (check also page 278 in 6th edition and 314 in 7th one of the book)

A standard way to use Well Ordering to prove that some property, \(P(n)\) holds for every nonnegative integer \(n\)

To prove that \"\(P(n)\) is true for all \(n \in \mathbb{N}\)\" using the Well Ordering Principle:

- Define the set, \(C\), of counterexamples to \(P\) being true. Namely, define
  \[ C := \{ n \in \mathbb{N} | P(n) \text{ is false} \} . \]
- Assume for proof by contradiction that \(C\) is nonempty.
- By the Well Ordering Principle, there will be a smallest element, \(n\), in \(C\).
- Reach a contradiction (somehow) —often by showing how to use \(n\) to find another member of \(C\) that is smaller than \(n\). (This is the open-ended part of the proof task.)
- Conclude that \(C\) must be empty, that is, no counterexamples exist. QED

The notation \(\{n | P(n)\}\) means \"the set of all elements \(n\), for which \(P(n)\) is true.\"

or, in another way

- Why is mathematical induction a valid proof technique?
- The reason comes from the well ordering property for the set of positive integers.
- Suppose we know that \(P(1)\) is true and that the proposition \(P(k) \rightarrow P(k+1)\) is true for all positive integers \(k\). To show that \(P(n)\) must be true for all positive integers \(n\), assume that there is at least one positive integer for which \(P(n)\) is false. Then the set \(S\) of positive integers for which \(P(n)\) is false is nonempty.
- Thus, by the well-ordering property, \(S\) has a least element, which will be denoted by \(m\). We know that \(m\) cannot be 1, because \(P(1)\) is true. Because \(m\) is positive and greater than 1, \(m - 1\) is a positive integer. Furthermore, because \(m - 1\) is less than \(m\), it is not in \(S\), so \(P(m - 1)\) must be true. Because the conditional statement \(P(m - 1) \rightarrow P(m)\) is also true, it must be the case that \(P(m)\) is true. This contradicts the choice of \(m\). Hence, \(P(n)\) must be true for every positive integer \(n\).
and, in the plainest English the last statements go as:

You assume that
• the set of integers S for which P(n) is false is nonempty. By WOP, there would be a smallest positive integer k for which P(k) is false.
• You then obtain a contradiction, showing that S must be empty.
• The contradiction is derived from the fact that for positive integer j with j < k, P(j) must be true due to the way k was chosen.

Example 6  Proof by WOP now

• Theorem: Every natural number n can be written as a product of primes.
• Proof: Let S be the set of natural numbers that cannot be written as a product of primes. Then by the WOP, S has a smallest element, which we will call n. n must not be a prime, because if it was, it could be written as a product of one prime, itself.
• Thus n = rs for some numbers such that 1 < r, s < n. Since both r and s are smaller than n, both can be written as products of primes. But that means that n is the product of primes, which is a contradiction. Thus the set S must be empty.
• In other words, there is no set of natural numbers that cannot be written as product of primes
4.3 & 4.4: Recursive Definitions

- In induction, we prove all members of an infinite set satisfy some predicate \( P \) by:
  - proving the truth of the predicate for larger members in terms of that of smaller members.

  Induction proves some CLOSED FORM EXPRESSION.

- In recursive definitions, we similarly define a function, a predicate, a set, or a more complex structure over an infinite domain (universe of discourse) by:
  - defining the function, predicate value, set membership, or structure of larger elements in terms of those of smaller ones.

- In structural induction, we inductively prove properties of recursively-defined objects in a way that parallels the objects’ own recursive definitions.

Recursion

- **Recursion** is the general term for the practice of defining an object in terms of itself
  - or of part of itself

- An inductive proof establishes the truth of \( P(n+1) \) recursively in terms of \( P(n) \).
- There are also recursive algorithms, definitions, functions, sequences, sets, and other structures.

What is a meaning of defining in terms of itself?

For example, let \( f(x) = x! \)

We can define \( f(x) \) as

\[
f(x) = x \cdot f(x-1), \text{ or one can also use } f(x+1) = (x+1) \cdot f(x)
\]

… more recursion examples

- Find \( f(1), f(2), f(3), \) and \( f(4) \), where \( f(0) = 1 \)
  a) Let \( f(n+1) = f(n) + 2 \)
     - \( f(1) = f(0) + 2 = 1 + 2 = 3 \)
     - \( f(2) = f(1) + 2 = 3 + 2 = 5 \)
     - \( f(3) = f(2) + 2 = 5 + 2 = 7 \)
     - \( f(4) = f(3) + 2 = 7 + 2 = 9 \)

  And, the closed form for this series is \( a_n = 2n + 1, n = 0, 1, 2, \ldots \)

  b) Let \( f(n+1) = 3f(n) \)
     - \( f(1) = 3 \cdot f(0) = 3 \cdot 1 = 3 \)
     - \( f(2) = 3 \cdot f(1) = 3 \cdot 3 = 9 \)
     - \( f(3) = 3 \cdot f(2) = 3 \cdot 9 = 27 \)
     - \( f(4) = 3 \cdot f(3) = 3 \cdot 27 = 81 \)

  And, the closed form for this series is \( a_n = 3^n, n = 0, 1, 2, \ldots \)
... more recursion examples

- Find $f(1)$, $f(2)$, $f(3)$, and $f(4)$, where $f(0) = 1$

c) Let $f(n+1) = 2^{f(n)}$

  - $f(1) = 2^{f(0)} = 2^1 = 2$
  - $f(2) = 2^{f(1)} = 2^2 = 4$
  - $f(3) = 2^{f(2)} = 2^4 = 16$
  - $f(4) = 2^{f(3)} = 2^{16} = 65536$

d) Let $f(n+1) = f(n)^2 + f(n) + 1$

  - $f(1) = f(0)^2 + f(0) + 1 = 1^2 + 1 + 1 = 3$
  - $f(2) = f(1)^2 + f(1) + 1 = 3^2 + 3 + 1 = 13$
  - $f(3) = f(2)^2 + f(2) + 1 = 13^2 + 13 + 1 = 183$
  - $f(4) = f(3)^2 + f(3) + 1 = 183^2 + 183 + 1 = 33673$

Recursively Defined Functions

- Simplest case:
  One way to define a function $f : \mathbb{N} \rightarrow S$ (for any set $S$) or series $a_n = f(n)$ is to:
    - Define $f(0)$ ($f(1), f(2), \ldots, f(n-1)$).
    - For $n > 0$, define $f(n)$ in terms of $f(0), \ldots, f(n-1)$.
- Ex. Define the series $a_n := 2^n$ recursively:
  - Let $a_0 := 1$.
  - For $n > 0$, let $a_n := 2a_{n-1}$.

Another Example

- Suppose we define $f(n)$ for all $n \in \mathbb{N}$ recursively by:
  - Let $f(0) = 3$
  - For all $n \in \mathbb{N}$, let $f(n+1) = 2f(n) + 3$

  - What are the values of the following $f$-s?
    - $f(1) = 9$  $f(2) = 21$  $f(3) = 45$  $f(4) = 93$

Recursive Definition of Factorial

- Give an inductive (recursive) definition of the factorial function,
  $F(n) := n! := \prod_{1 \leq i \leq n} i = 1 \cdot 2 \ldots \cdot n$.
  - Base case: $F(0) := 1$
  - Recursive part: $F(n) := n \cdot F(n-1)$.
    - $F(1) = 1$
    - $F(2) = 2$
    - $F(3) = 6$
    - $F(4) = 24$ …
More Easy Examples

• Write down recursive definitions for:
  – $a \cdot n$ (a real, $n$ natural) using only addition
  – $a^n$ (a real, $n$ natural) using only multiplication
  – $\sum_{0 \leq i \leq n} a_i$ (for an arbitrary series of numbers $\{a_i\}$)
  – $\prod_{0 \leq i \leq n} a_i$ (for an arbitrary series of numbers $\{a_i\}$)

Fibonacci sequence is a good old, and often useful today, series

• Definition of the Fibonacci sequence
  – Non-recursive, or closed form:
  
    $F(n) = \left(\frac{1 + \sqrt{5}}{2}\right)^n - \left(\frac{1 - \sqrt{5}}{2}\right)^n$

  – Recursive: $F(n) = F(n-1) + F(n-2)$
    or: $F(n+1) = F(n) + F(n-1)$

• Must always specify base case(s)!
  – $F(1) = 0$, $F(2) = 1$
  – Note that some will use $F(0) = 0$, $F(1) = 1$

The Fibonacci Series

• The Fibonacci series $f_{n \geq 0}$ is a famous series defined by:
  $f_0 \equiv 0, \ f_1 \equiv 1, \ f_{n \geq 2} \equiv f_{n-1} + f_{n-2}$

  $0 \rightarrow + \rightarrow 1 \rightarrow + \rightarrow 1 \rightarrow + \rightarrow 2 \rightarrow + \rightarrow 3 \rightarrow + \rightarrow 5 \rightarrow + \rightarrow 8 \rightarrow + \rightarrow 13 \rightarrow + \rightarrow 21 \rightarrow + \rightarrow$ Leonardo Fibonacci
  1170-1250

  $0+1=1, 1+1=2, 2+2=3, 3+3=5, 5+8=13$ or simply, 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ..., infinity.

Inductive Proof about Fib. Series – Upper Bound

• Theorem. $f_n < 2^n$.

  Proof. By induction. Implicitly for all $n \in \mathbb{N}$
  Base cases: $f_0 = 0 < 2^0 = 1 \quad f_1 = 1 < 2^1 = 2$

  Note use of base cases of recursive def'n.

  • Inductive step: Use 2nd principle of induction (strong induction). Assume $\forall k < n, \ f_k < 2^k$.
  • Then $f_n = f_{n-1} + f_{n-2}$ and obviously $f_n < 2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2^n$. ■
A lower bound on Fibonacci series

- **Theorem.** For all integers \( n \geq 3 \), \( f_n > \alpha^{n-2} \), where \( \alpha = (1+\sqrt{5})/2 \approx 1.61803 \).

Proof. (Using strong induction.)
- Let \( P(n) = (f_n > \alpha^{n-2}) \).
- **Base cases:**
  - For \( n = 3 \), \( \alpha^{3-2} = \alpha \approx 1.61803 < 2 = f_3 \).
  - For \( n = 4 \), \( \alpha^{4-2} = \alpha^2 = (1+2\sqrt{5})/2 \approx 2.61803 < 3 = f_4 \).
- **Inductive step:**
  - For \( k \geq 4 \), assume \( P(j) \) for \( 3 \leq j \leq k \), prove \( P(k+1) \).
  - Note \( \alpha^2 = \alpha + 1 \).
  - \( \alpha^{k+1} - 2 = \alpha^{k-1} - \alpha^2 = \alpha^{k-1} - (\alpha + 1) = \alpha^{k-2} + \alpha^{k-3} \).
  - By inductive hypothesis, \( f_{k-1} > \alpha^{k-3} \) and \( f_k > \alpha^{k-2} \). So, \( \alpha^{k+1} - 2 = \alpha^{k-2} + \alpha^{k-3} < f_k + f_{k-1} = f_{k+1} \).
  - Thus \( P(k+1) \). ■

Lamé’s Theorem

- **Thm.** \( \forall a, b \in \mathbb{Z}^+, a \geq b \), the number of steps in Euclid’s algorithm to find \( \gcd(a, b) \) is \( \leq 5k \), where \( k = \lceil \log_{10} b \rceil + 1 \) is the number of decimal digits in \( b \).
  - Thus, Euclid’s algorithm is linear-time in the number of digits in \( b \).
- **Proof:**
  - Uses the Fibonacci sequence!
  - See next 2 slides.

Proof of Lamé’s Theorem

- Consider the sequence of division-algorithm equations used in Euclid’s algorithm:
  - \( r_0 = r_1q_1 + r_2 \) with \( 0 \leq r_2 < r_1 \)
  - \( r_1 = r_2q_2 + r_3 \) with \( 0 \leq r_3 < r_2 \)
  - …
  - \( r_{n-2} = r_{n-1}q_{n-1} + r_n \) with \( 0 \leq r_n < r_{n-1} \)
  - \( r_{n-1} = r_nq_n + r_{n+1} \) with \( r_{n+1} = 0 \) (terminate)
- The number of divisions (iterations) is \( n \).

Lamé Proof, continued

- Since \( r_0 \geq r_1 > r_2 > \ldots > r_n \),
  each quotient \( q_i = \lceil r_{i-1}/r_i \rceil \geq 1 \), \( i = 1, \ldots, n-1 \).
- Since \( r_{n-1} = r_nq_n \) and \( r_{n-1} > r_n, q_n \geq 2 \).
- So we have the following relations between \( r \) and \( f \):
  - \( r_n \geq 1 = f_2 \)
  - \( r_{n-1} \geq 2r_n \geq 2 = f_3 \)
  - \( r_{n-2} \geq r_{n-1} + r_{n-2} \geq f_2 + f_3 = f_4 \)
  - …
  - \( r_2 \geq r_3 + r_4 + f_1 + f_2 = f_5 \)
  - \( b = r_0 \geq r_2 + r_3 + \ldots + r_{n-1} = f_{n+1} \).

- Thus, if \( n+2 \) divisions are used, then \( b \geq f_{n+1} > \alpha^{n-1} \).
  - Thus, \( \log_{10} b \geq \log_{10}(\alpha^{n-1}) = (n-1)\log_{10} \alpha = (n-1)0.208 > (n-1)/5 \).
  - If \( b \) has \( k \) decimal digits, then \( \log_{10} b < k \), so \( n-1 < 5k \), so \( n \leq 5k \).
Recursively Defined Sets

• An infinite set $S$ may be defined recursively, by giving:
  i) A small finite set of base elements of $S$.
  ii) A rule for constructing new elements of $S$ from previously-established elements.
  iii) Implicitly, $S$ has no other elements but these.

• Example. Let $3 \in S$, and let $x+y \in S$ if $x, y \in S$.
  What is $S$?
  What about this? $3+3=6, 3+6=9, 6+6=12, 3+9=12, \ldots$

The Set of All Strings

• Def. Given an alphabet $\Sigma$, the set $\Sigma^*$ of all strings over $\Sigma$ can be recursively defined by:
  
  $\lambda \in \Sigma^*$ ($\lambda := \text{“the empty string”}$)
  
  $w \in \Sigma^* \land x \in \Sigma \rightarrow wx \in \Sigma^*$

  Note that $wx$ is a string not a product
  
• Exercise: Prove that this definition is equivalent to our old one:
  
  $\Sigma^* \equiv \bigcup_{n \in \mathbb{N}} \Sigma^n$

Other Easy String Examples

• Give recursive definitions for:
  • The concatenation of strings $w_1 \cdot w_2$.
  • The length $\ell(w)$ of a string $w$.
  • Well-formed formulae of propositional logic involving $T, F$, propositional variables, and operators in $\{\neg, \land, \lor, \rightarrow, \leftrightarrow\}$.
  • Well-formed arithmetic formulae involving variables, numerals, and operations in $\{+, -, \ast, \uparrow\}$.

Rooted Trees

• Trees will be covered in more depth in chapter 9.
  • Briefly, a tree is a graph in which there is exactly one undirected path between each pair of nodes.
  • An undirected graph can be represented as a set of unordered pairs (called edges) of objects called nodes.
  • Definition of the set of rooted trees:
    i) Any single node $r$ is a rooted tree.
    ii) If $T_1, \ldots, T_n$ are disjoint rooted trees with respective roots $r_1, \ldots, r_n$, and $r$ is a node not in any of the $T_i$'s, then another rooted tree is $\{(r, r_1), (r, r_2), \ldots, (r, r_n)\} \cup T_1 \cup \ldots \cup T_n$.
    iii) That is all.
Illustrating Rooted Tree Def’n.

• How rooted trees can be combined to form a new rooted tree…

Extended Binary Trees

• A special case of rooted trees.

  • Def. Extended binary trees (EBT):
    i) The empty set $\emptyset$ is an EBT.
    ii) If $T_1, T_2$ are disjoint EBTs, then $e_1 \cup e_2 \cup T_1 \cup T_2$ is an EBT, where
        - $e_1 = \emptyset$ if $T_1 = \emptyset$, and
        - $e_1 = \{(r, r_1)\}$ if $T_1 \neq \emptyset$ and has root $r_1$, and
        - similarly for $e_2$.
    iii) That is all.

Full Binary Trees

• A special case of extended binary trees.

  • Def. Recursive definition of full binary trees (FBT):
    – i) A single node $r$ is a FBT.
      • Note this is different from the EBT base case.
    – ii) If $T_1, T_2$ are disjoint FBTs, then $e_1 \cup e_2 \cup T_1 \cup T_2$, where
        - $e_1 = \emptyset$ if $T_1 = \emptyset$, and
        - $e_1 = \{(r, r_1)\}$ if $T_1 \neq \emptyset$ and has root $r_1$, and
        - similarly for $e_2$.
      • Note this is the same as the EBT recursive case!
        – Can simplify it to “If $T_1, T_2$ are disjoint FBTs with roots $r_1$ and $r_2$, then $\{(r, r_1), (r, r_2)\} \cup T_1 \cup T_2$ is an FBT.”
    – i) That is all.

Structural Induction

• Proving something about a recursively defined object using an inductive proof whose structure mirrors the object’s definition.
Example

• **Thm.** Let $3 \in S$, and let $x + y \in S$ if $x, y \in S$, and that is all. Let $A = \{n \in \mathbb{Z}^+ | (3|n)\}$. Than $A = S$.

**Proof.** We show that $A \subseteq S$ and $S \subseteq A$.
- To show $A \subseteq S$, show $[n \in \mathbb{Z}^+ \land (3|n)] \rightarrow n \in S$.
  - **Inductive proof.** Let $P(n) \equiv n \in S$. Induction over positive multiples of 3. Base case, $n = 3$, thus $3 \in S$ by def'n. of $S$.
  Inductive step. Given $P(n)$, prove $P(n+3)$. By inductive hyp., $n \in S$, and $3 \in S$, so by def'n of $S$, $n + 3 \in S$.
- To show $S \subseteq A$: let $n \in S$, show $n \in A$.
  - **Structural inductive proof.** Let $P(n) :\equiv n \in A$.
    Base case. $3 \in S$. Since $3|3$, $3 \in A$.
    Recursive step. $x, y \in S$, $n = x + y \in S$ and $x, y \in A$. Since $3|x$ and $3|y$, we have $3|(x + y)$, thus $x + y = n \in A$.

---

**Recursive Algorithms (§4.4)**

- Recursive definitions can be used to describe algorithms as well as functions and sets.
- **Ex.** A procedure to compute $a^n$.
  - procedure power($a \neq 0$: real, $n \in \mathbb{Z}$)
    - if $n = 0$ then return 1
    - else return $a \cdot$ power($a$, $n-1$)

---

**Efficiency of Recursive Algorithms**

- The time complexity of a recursive algorithm may depend critically on the number of recursive calls it makes.
- **Ex.** Modular exponentiation to a power $n$ can take log($n$) time if done right, but linear time if done slightly differently.
  - Task: Compute $b^n \mod m$, where $m \geq 2$, $n \geq 0$, and $1 \leq b < m$. 

Modular Exponentiation Alg. #1

- Uses the fact that \( b^n = b \cdot b^{n-1} \) and that \( x \cdot y \mod m = x \cdot (y \mod m) \mod m \).
  (Prove the latter theorem at home.)

- procedure \( mpower(b \geq 1, n \geq 0, m > b \in \mathbb{N}) \)
  
  - \{Returns \( b^n \mod m \).\}
  
  if \( n = 0 \)
    then return 1
  else return \((b \cdot mpower(b, n-1, m)) \mod m\)

- Note this algorithm takes \( \Theta(n) \) steps!

Modular Exponentiation Alg. #2

- Uses the fact that \( b^{2k} = b^k \cdot b^k = (b^k)^2 \).

- procedure \( mpower(b, n, m) \) \{same signature\}
  
  - if \( n = 0 \) then return 1
  - else if \( 2 | n \)
    then return \( mpower(b, n/2, m)^2 \mod m \)
  - else return \( (mpower(b, n-1, m) \cdot b) \mod m \)

- What is its time complexity? \( \Theta(\log n) \) steps

A Slight Variation

- Nearly identical but takes \( \Theta(n) \) time instead!

- procedure \( mpower(b, n, m) \) \{same signature\}
  
  - if \( n = 0 \) then return 1
  - else if \( 2 | n \)
    then return \((mpower(b, n/2, m) \cdot \)
      
      \( mpower(b, n/2, m)) \mod m \)
  - else return \( (mpower(b, n-1, m) \cdot b) \mod m \)

  The number of recursive calls made is critical!

Recursive Euclid’s Algorithm

- procedure \( gcd(a, b \in \mathbb{N}) \)
  
  if \( a = 0 \)
    then return \( b \)
  else return \( gcd(b \mod a, a) \)

- Note recursive algorithms are often simpler to code than iterative ones...

- However, they can consume more stack space, if your compiler is not smart enough.
Recursive Linear Search

• Note there is no real advantage to using recursion here, rather than just looping for \( \text{loc} := i \) to \( j \)...

• procedure \( \text{search(a: series; i, j: integer; x: item to be found)} \)
  
  {Find \( x \) in series \( a \) at a location \( \geq i \) and \( < j \)}
  
  if \( a_i = x \) then return \( i \) (At the right item? Return it!)
  if \( i = j \) then return 0  (No locations in range? Failure!)
  return \( \text{search(i+1, j, x)} \) (Try rest of range)

Recursive Binary Search

• procedure \( \text{binarySearch(a, x, i, j)} \) {same sig}
  
  {Find location of \( x \) in \( a \), \( \geq i \) and \( < j \)}
  
  \( m := \lfloor (i + j)/2 \rfloor \)  (Go to halfway point.)
  if \( x = a_m \) then return \( m \)  (Did we luck out?)
  if \( x < a_m \land i < m \)  (If it’s to the left,)
    then return \( \text{binarySearch(a, x, i, m-1)} \) {Check that \( \frac{1}{2} \)}
  else if \( a_m < x \land m < j \)  (If it’s to right,)
    then return \( \text{binarySearch(a, x, m+1, j)} \) {Check that \( \frac{1}{2} \)}
  else
    return 0  (No more items, failure.)

Recursive Fibonacci Algorithm

• procedure \( \text{fibonacci(n \in N)} \)
  
  if \( n = 0 \) then return 0
  if \( n = 1 \) then return 1
  return \( \text{fibonacci(n-1)} + \text{fibonacci(n-2)} \)

• Is this an efficient algorithm?
  – Is it polynomial-time in \( n \)?
• How many additions are performed?

Analysis of Fibonacci Procedure

• Thm. The preceding procedure for \( \text{fibonacci(n)} \) performs \( f_{n+1} - 1 \) addition operations.

Proof. By strong structural induction over \( n \), based on the procedure’s own recursive definition.

• Base cases: \( \text{fibonacci(0)} \) performs 0 additions, and \( f_0 + 1 = 1 \) = 1 = 0. Likewise, \( \text{fibonacci(1)} \) performs 0 additions, and \( f_1 + 1 = f_1 + 1 = 1 \) = 0.
• Inductive step: For \( n \geq 1 \), by strong inductive hypothesis, \( \text{fibonacci(n-1)} \) and \( \text{fibonacci(n-2)} \) do \( f_{n-1} \) and \( f_{n-1} \) additions respectively, and \( \text{fibonacci(n)} \) adds 1 more, for a total of \( f_{n-1} + f_{n-1} + 1 = f_n + f_{n-1} + 1 = f_{n+1} \).
A more efficient algorithm

- **procedure** `findFib(a, b, m, n)`  
  (Given \(a = f_{m-1}, b = f_m,\) and \(m \leq n\), return \(f_n\))  
  if \(m = n\)  
    then return \(b\)  
  return `findFib(b, a+b, m+1, n)`

- **procedure** `fastFib(n)`  
  (Find \(f_n\) in \(\Theta(n)\) steps.)  
  if \(n = 0\)  
    then return 0  
  return `findFib(0, 1, 1, n)`

Recursive Merge Sort

- **procedure** `sort(L = \ell_1, ..., \ell_n)`  
  if \(n > 1\) then  
    \(m := \lfloor n/2 \rfloor\)  
    \(L := \text{merge} (\text{sort} (\ell_1, ..., \ell_m), \text{sort} (\ell_{m+1}, ..., \ell_n))\)  
  return \(L\)

  • The merge (next slide) takes \(\Theta(n)\) steps, and merge-sort takes \(\Theta(n \log n)\).

Recursive Merge Method

- **procedure** `merge(A, B: sorted lists)`  
  (Given two sorted lists \(A = (a_1, ..., a_{\mid A\mid})\), \(B = (b_1, ..., b_{\mid B\mid})\), return a sorted list of all.)  
  if \(A = ()\) then return \(B\)  
  if \(B = ()\) then return \(A\)  
  if \(a_1 < b_1\) then return \((a_1, \text{merge}((a_2, ..., a_{\mid A\mid}), B))\)  
  else return \((b_1, \text{merge}(A, (b_2, ..., b_{\mid B\mid})))\)

Merge Routine

- **procedure** `merge(A, B: sorted lists)`  
  \(L = \text{empty list}\)  
  \(i := 0, j := 0, k := 0\)  
  while \(i < \mid A\mid \land j < \mid B\mid\)  
    \(L_k := B_j;\)  
    \(j := j + 1\)  
  \(L_k := A_i;\)  
  \(i := i + 1\)  
  if \(A_i < B_j\) then  
    \(L_k := A_i;\)  
    \(i := i + 1\)
  else if \(A_i > B_j\) then  
    \(L_k := B_j;\)  
    \(j := j + 1\)
  \(k := k + 1\)  
  return \(L\)

Takes \(\Theta(\mid A\mid + \mid B\mid)\) time
References

- Rosen
  Discrete Mathematics and its Applications, 6e