CS 2305: Discrete Mathematics for Computing I

Lecture 25

- KP Bhat

Proving a result using Strong Induction

Example: Prove that every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

Solution: Let $P(n)$ be the proposition that postage of n cents can be formed using 4-cent and 5-cent stamps.

BASIS STEP: $P(12)$, $P(13)$, $P(14)$, and $P(15)$ hold.

- $P(12)$ uses three 4-cent stamps.
- $P(13)$ uses two 4-cent stamps and one 5-cent stamp.
- $P(14)$ uses one 4-cent stamp and two 5-cent stamps.
- $P(15)$ uses three 5-cent stamps.

INDUCTIVE STEP: Assume that $P(j)$ is true for $12 \leq j \leq k$, where $k \geq 15$.

- $k - 3 \geq 12$
- So we can form postage of $k - 3$ cents using just 4 cent and 5 cent stamps
- Now if we add a 4-cent stamp to the postage for $k - 3$ cents, we can form postage for $k - 3 + 4 = k + 1$ cents

We have shown that if the inductive hypothesis is true, then $P(k + 1)$ is also true. This completes the inductive step.

Because we have completed the basis step and the inductive step, we conclude that every postage of n cents, where n is at least 12, can be formed using 4-cent and 5-cent stamps.

Proving the same result using Mathematical Induction

Example: Prove that every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

Solution: Let *P*(*n*) be the proposition that postage of *n* cents can be formed using 4-cent and 5-cent stamps.

BASIS STEP: *P(12)* is true since we can form postage of 12 cents using three 4-cent stamps.

INDUCTIVE STEP: Assume P(k) is true for any $k \geq 12$

We will consider two cases.

i. The postage requires at least one 4-cent stamp

- In this case we can replace the 4-cent stamp with a 5-cent stamp, to form postage of *k + 1* cents.

- The postage does not require any 4-cent stamps
  - Since $k \geq 12$, the postage will require at least three 5-cent stamps. In this case we can replace three 5-cent stamps with four 4-cent stamps, to form postage of k + 1 cents.

Because we have completed the basis step and the inductive step, we conclude that every postage of n cents, where n is at least 12, can be formed using 4-cent and 5-cent stamps.

Another Proof by Strong Induction

Conjecture: Given a sequence defined as follows:

$a_0 = 0$, $a_1 = 1$, and $a_n = 3a_{n-1} - 2a_{n-2}$ for $n \geq 2$

Show that $a_n = 2^n - 1$ for all $n \in W$, the set of whole numbers

Whole numbers consist of 0 and the natural numbers

- -ve numbers are excluded

Solution:

BASIS STEP:

P(0) is true since $2^0 - 1 = 0$

P(1) is true since $2^1 - 1 = 1$

INDUCTIVE STEP:

Let $k \geq 2$ and assume that the statement is true for 0, 1, 2, ….k-1

Now $a_k = 3a_{k-1} - 2a_{k-2} = 3*(2^{k-1} -1) - 2*(2^{k-2} -1) = 3*2^{k-1} -3 - 2^{k-1} + 2$

$= 3*2^{k-1} - 2^{k-1} - 1 = (3 -1)*2^{k-1} - 1 = 2*2^{k-1} - 1 = 2^k - 1$

$\therefore$ the result holds for n = k as well

From the Basis Step and the Inductive Step we have proved that $a_n = 2^n - 1$ for all $n \in W$

An Incorrect "Proof" by Strong Induction

THIS IS A FLAWED PROOF FOR STRONG INDUCTION

Inductive Hypothesis: For every natural number n, $2^n = 2$.

BASIS STEP: For $n = 1$, $2^1 = 2$

ATTEMPTED INDUCTIVE STEP: Let us assume that the result holds true for all natural numbers $n \leq k$

$2^{k+1} = 2^k * 2 = 2^k * (2^k/2^{k-1}) = 2 * (2/2)$ [By inductive hypothesis]

$2^{k+1} = 2 * 1 = 2$

$\therefore$ if the result holds for all natural numbers $n \leq k$, it holds for $k+1$ as well

Where is the error?

Answer: Here the base case is n = 1. However, we need the induction hypothesis for both n = k and n = k − 1, and in the case k = 1 the latter value is n = 1 − 1 = 0, which is out of range of the induction hypothesis.

Recursive Algorithms

Introduction

Section 5.4.1

Recursive Algorithms

Definition: An algorithm is called *recursive* if it solves a problem by reducing it to an instance of the same problem with smaller input.

For the algorithm to terminate, the instance of the problem must eventually be reduced to some initial case for which the solution is known.

A recursive function consists of two parts:

i. A base case that is processed without recursion; and

ii. A recursive case that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case

Each successive invocation of the recursive function, other than the base case, is a slightly simpler or smaller version of the previous invocation

Recursive Factorial Algorithm

Example: Give a recursive algorithm for computing $n!$, where n is a nonnegative integer.

Solution: Use the recursive definition of the factorial function.

procedure *factorial*(n: nonnegative integer)

if $n = 0$ **then return** 1

else return $n \cdot factorial\ (n - 1)$

{output is $n!$}

Fact(5) = 5 x 4 x 3 x 2 x 1
= 5 x Fact(4)
= 5 x 4 x Fact(3)
= 5 x 4 x 3x Fact(2)
= 5 x 4 x 3x 2 x Fact(1)
= 5 x 4 x 3x 2 x 1 x Fact(0)
= 5 x 4 x 3x 2 x 1 (Fact(0) =1)

Recursive Exponentiation Algorithm

Example: Give a recursive algorithm for computing a^n, where a is a nonzero real number and n is a nonnegative integer.

Solution: Use the recursive definition of a^n.

procedure *power*(a: nonzero real number, n: nonnegative integer)

if $n = 0$ **then return** 1

else return $a \cdot power(a, n - 1)$

{output is a^n}

Background: Fibonacci Series

- Fibonacci series are the numbers in the following sequence:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
  - The first two numbers are 0 and 1 and each subsequent number in the series is equal to the sum of the previous two numbers
- Note:- Some mathematicians consider the first two numbers of the series to be 1, 1, since that is how it was formulated by Fibonacci, when he was trying to model the population growth of rabbits

Recursive Fibonacci Algorithm

procedure *fibonacci*(*n*: nonnegative integer)

if $n \leq 1$ **then return** *n*

else return *fibonacci(n-1) + fibonacci(n-2)*

{output is n^{th} Fibonacci number}

Recursive Binary Search Algorithm

Example: Construct a recursive version of a binary search algorithm.

Solution: Assume we have $a_1, a_2, \ldots, a_n$, an increasing sequence of integers. Initially i is 1 and j is n. We are searching for x.

procedure *binary search*($i, j, x :$ integers, $1 \le i \le j \le n$)
$m := \lfloor (i + j)/2 \rfloor$
if $x = a_m$ **then**
    return m
else if ($x < a_m$ and $i < m$) **then**
    return *binary search*($i, m-1, x$)
else if ($x > a_m$ and $j > m$) **then**
    return *binary search*($m+1$, j, x)
else return 0
{output is location of x in $a_1, a_2, \ldots, a_n$ if it appears, otherwise 0}

Merge Sort 1

Merge Sort works by iteratively splitting a list into two sublists until each sublist has one element.

Each sublist is represented by a binary tree.

At each step a pair of sublists is successively merged into a list with the elements in increasing order. The process ends when all the sublists have been merged.

The succession of merged lists is represented by a binary tree.

https://www.youtube.com/watch?v=4VqmGXwpLqc

https://www.youtube.com/watch?v=es2T6KY45cA

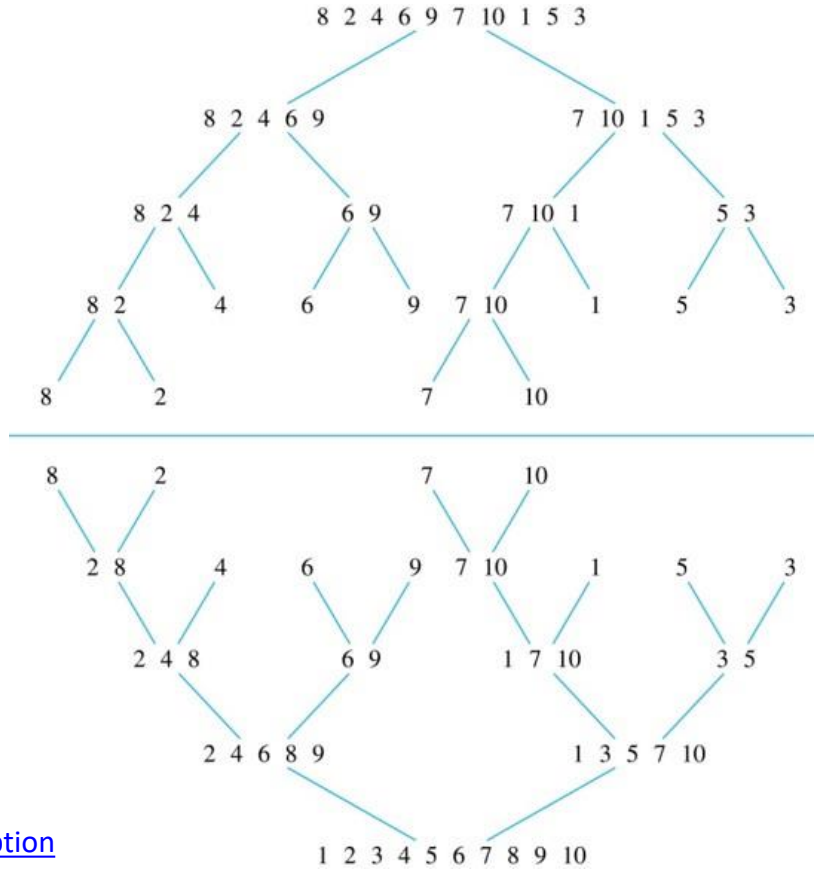
Merge Sort 2

Example: Use merge sort to put the list

8,2,4,6,9,7,10, 1, 5, 3

into increasing order.

Solution:



Jump to long description

Recursive Merge Sort 1

Example: Construct a recursive merge sort algorithm.

Solution: Begin with the list of n elements L.

procedure *mergesort*($L = a_1, a_2,\ldots,a_n$)

if $n > 1$ **then**

$m := \lfloor n/2 \rfloor$

$L_1 := a_1, a_2,\ldots,a_m$

$L_2 := a_{m+1}, a_{m+2},\ldots,a_n$

L := *merge*(*mergesort*(L_1), *mergesort*(L_2))

{L is now sorted into elements in increasing order}

Recursive Merge Sort 2

Subroutine *merge*, which merges two sorted lists.

procedure *merge*(L_1, L_2 :sorted lists)

L := empty list

while L_1 and L_2 are both nonempty

    remove smaller of first elements of L_1 and L_2 from its list;

        put at the right end of L

    if this removal makes one list empty

        then remove all elements from the other list and append them to L

return L {L is the merged list with the elements in increasing order}

Complexity of Merge Sort: It can be proved (beyond the scope of this course) that the complexity of Merge Sort is *O(n log n)*. In more advanced classes you will learn that the fastest comparison based sorting algorithms have *O(n log n)* time complexity. So Merge Sort achieves the best possible big-O estimate of time complexity for a comparison based sorting algorithm.

Recursion and Iteration (1)

Recursion

1. Function calls itself, until the base case is executed, and the recursion terminates
2. More overhead; slower
   - Context switching for each recursive call
3. Higher space complexity
   - A new stack frame for each invocation of the recursive call
4. Generally a recursive function is very simple
   - e.g. the recursive algorithm for the Towers of Hanoi problem is very simple and intuitive

Iteration

1. Set of instructions is executed repeatedly, while a given condition is satisfied
2. Less overhead; faster
3. Lower space complexity
4. Sometimes it is hard to follow the change in execution context from iteration to iteration
   - e.g. the iterative algorithm for the Towers of Hanoi problem is very complex

Recursion and Iteration (2)

- In theory any problem that can be solved by a recursive algorithm can be solved by an iterative algorithm, and vice-versa
- Some problems can be better understood and resolved using recursion
  - At a later time an iterative method may be sought, if performance is an issue
    - In reality some problems lend themselves naturally to easier solution using one approach as compared to the other
- The easiest recursive algorithms to convert to iterative are those involving *tail recursion* i.e. recursive algorithms in which no statements are executed after the return from the recursive call

```
procedure factorial(n)
begin
   if n = 0
   then
         return 1
   else
         return n·factorial(n – 1)
end
```

Tail Recursion

Iterative Algorithm for Computing Factorial

```
procedure iterative_factorial(n: nonnegative integer)
int y :=1
for x := 1 to n
        y = y * x
return y
{output is n!}
```

Iterative Algorithm for Computing Fibonacci Numbers

```
procedure iterative_fibonacci(n: nonnegative integer)
if  n = 0 then return 0
else
    x := 0
    y := 1
    for i := 1 to n − 1
        z := x + y
        x := y
        y := z
    return y
    {output is the nth Fibonacci number}
```

The Basics of Counting

Section 6.1

Overview

- Combinatorics, the study of arrangements of objects, is an important part of discrete mathematics
- This subject had its origin in the systematic study of gambling games
- Combinatorics deals with ordered or unordered arrangements of the objects of a set with or without repetitions. These arrangements, called permutations and combinations
- Combinatorics has many practical applications

Basic Counting Principles: The Product Rule[1]

The Product Rule: Suppose that a procedure can be broken down into a sequence of two tasks. If there are n_1 ways to do the first task and for each of these ways of doing the first task, there are n_2 ways to do the second task, then there are $n_1 n_2$ ways to do the procedure.

Generalized Form

If a procedure consists of k steps and

the first step can be performed in n_1 ways,

the second step can be performed in n_2 ways

…

the kth step can be performed in n_k ways

then the entire procedure can be performed in $n_1 n_2 \cdots n_k$ ways

Product Rule: An Illustrative Example

- Assume that a procedure consists of three steps s_1, s_2, s_3
- There are 5 ways to perform s_1: designated as A, B, C, D, E
- There are 4 ways to perform s_2: designated as 1, 2, 3, 4
- There are 3 ways to perform s_3: designated as α, β, γ
- By the Product Rule there are 5 X 4 X 3 = 60 ways to perform the procedure, which are enumerated below

A-1-α, A-1-β, A-1-γ, A-2-α, A-2-β, A-2-γ, A-3-α, A-3-β, A-3-γ, A-4-α, A-4-β, A-4-γ, B-1-α, B-1-β, B-1-γ, B-2-α, B-2-β, B-2-γ, B-3-α, B-3-β, B-3-γ, B-4-α, B-4-β, B-4-γ, C-1-α, C-1-β, C-1-γ, C-2-α, C-2-β, C-2-γ, C-3-α, C-3-β, C-3-γ, C-4-α, C-4-β, C-4-γ, D-1-α, D-1-β, D-1-γ, D-2-α, D-2-β, D-2-γ, D-3-α, D-3-β, D-3-γ, D-4-α, D-4-β, D-4-γ, E-1-α, E-1-β, E-1-γ, E-2-α, E-2-β, E-2-γ, E-3-α, E-3-β, E-3-γ, E-4-α, E-4-β, E-4-γ