

Recursion vs Induction

CS3330: Algorithms
The University of Iowa

1

1

Recursion

- Recursion means defining something, such as a function, in terms of itself
 - For example, let $f(x) = x!$
 - We can define $f(x)$ as
$$f(x) = \text{if } x < 2 \text{ then } 1 \text{ else } x * f(x-1)$$
- Recursion is a powerful problem-solving technique that often produces very clean solutions to even complex problems.
- Recursive solutions can be easier to understand and to describe than iterative solutions.

2

2

Recursion example

- Sequences are functions from natural numbers to reals:

$$f(i) = a_i$$

$$a_0, a_1, a_2, a_3, \dots, a_n$$

Example: Find $f(1), f(2), f(3)$, and $f(4)$,

where $f(0) = 1$, and

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

$$f(1) = f(0)^2 + f(0) + 1 = 1^2 + 1 + 1 = 3$$

$$f(2) = f(1)^2 + f(1) + 1 = 3^2 + 3 + 1 = 13$$

$$f(3) = f(2)^2 + f(2) + 1 = 13^2 + 13 + 1 = 183$$

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

3

Iterative vs. Recursive

- Iterative**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 & \text{if } n>0 \end{cases}$$

Function does NOT call itself

- Recursive**

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times \text{factorial}(n-1) & \text{if } n>0 \end{cases}$$

Function calls itself

4

Iterative Algorithm

```
factorial(n) {  
  i = 1  
  factN = 1  
  while (i <= n)  
    factN = factN * i  
    i = i + 1  
  return factN  
}
```

- The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.
- In general, iterative solution is computed from small to big.

5

Recursive Algorithm

```
factorial(n) {  
  if (n = 0)  
    return 1  
  else  
    return n*factorial(n-1)  
  end if  
}
```

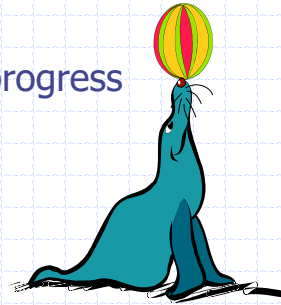
Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version →

We have eliminated the loop and implemented the algorithm with one 'if' statement.

6

Recursion Breakdown

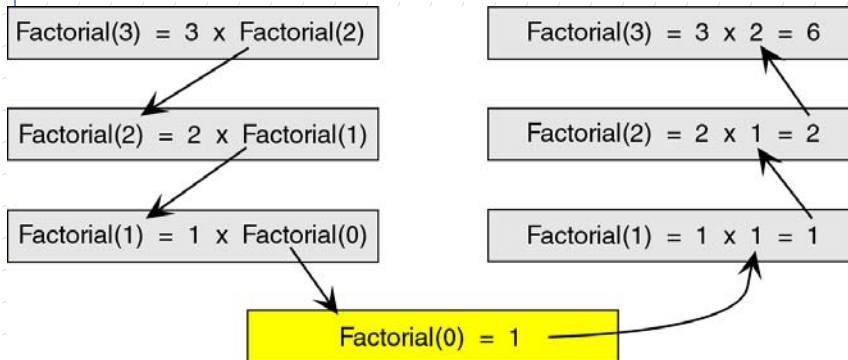
- 1. Base cases:
 - Always have at least one case that can be solved without using recursion.
- 2. Recursive cases:
 - Any recursive call must make progress toward a base case.



7

Recursion Breakdown

- To see how the recursion works, let's break down the factorial function to solve factorial(3)



8

How Recursion Works

- A recursive solution solves a problem by solving a smaller instance of the same problem.
- It solves this new problem by solving an even smaller instance of the same problem.
- Eventually, the new problem will be so small that its solution will be either obvious or known.
- This solution will lead to the solution of the original problem.

9

How Recursion Works

- To truly understand how recursion works we need to first explore how any function call works.
- When a program calls a subroutine (function) the current function must suspend its processing.
- The called function then takes over control of the program.
- When the function is finished, it needs to return to the function that called it.
- The calling function then 'wakes up' and continues processing.

10

How Recursion Works

- To do this we use a stack.
- Before a function is called, all relevant data is stored in a ***stackframe***.
- This stackframe is then pushed onto the system stack.
- After the called function is finished, it simply pops the stackframe off the stack to return to the original state.
- By using a stack, we allow functions call other functions (including themselves) which can call other functions, etc.

11

Limitations of Recursion

- The main disadvantage of programming recursively is that, while it makes it is easier to write simple and elegant programs, it also makes it easier to write inefficient ones.
- When we use recursion to solve problems, we are interested exclusively with correctness, and not at all with efficiency. Consequently, our simple, elegant recursive algorithms may be inherently inefficient.

12

Fibonacci sequence

- fibonacci(0) = 0
 - fibonacci(1) = 1
 - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) for n>1
- This definition is a little different than the previous recursive definitions because it has two base cases, not just one; in fact, you can have as many as you like.
 - In the recursive case, there are two recursive calls, not just one. There can be as many as you like.

13

Fibonacci sequence

- Definition of the Fibonacci sequence
 - Non-recursive:
$$F(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{\sqrt{5} \cdot 2^n}$$
 - Recursive:
$$F(n) = F(n-1) + F(n-2)$$

or:
$$F(n+2) = F(n+1) + F(n)$$
- Must always specify base case(s)!
 - $F(0) = 0, F(1) = 1$
 - Note that some will use $F(1) = 1, F(2) = 1$

14

14

Fibonacci sequence in Java

```

long Fibonacci (int n) {
    if ( n == 0 ) return 0;
    else if ( n == 1 ) return 1;
    else return Fibonacci (n-2) + Fibonacci (n-1);
}

long Fibonacci2(int n) {
    return (long) ((Math.pow((1.0+Math.sqrt(5.0)), n) -
        Math.pow((1.0-Math.sqrt(5.0)), n)) /
        (Math.sqrt(5) * Math.pow(2, n)));
}

```

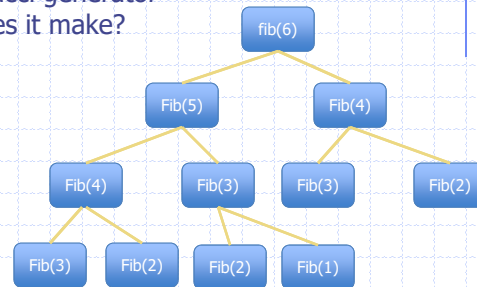
15

15

Exponential Time Algorithms

- Consider the recursive Fibonacci generator
- How many recursive calls does it make?

- $F(1)$: 1
- $F(2)$: 1
- $F(3)$: 3
- $F(4)$: 5
- $F(5)$: 9
- $F(10)$: 109
- $F(20)$: 13,529
- $F(30)$: 1,664,079
- $F(40)$: 204,668,309
- $F(50)$: 25,172,538,049
- $F(100)$: 708,449,696,358,523,830,149 $\approx 7 * 10^{20}$
 - ♦ At 1 billion recursive calls per second (generous), this would take over 22,000 years



16

16

How many additions used by Fibonacci(n)?

```
long Fibonacci(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return Fibonacci(n-2) + Fibonacci(n-1);
}
```

Solve it by recursion: Let $a(n)$ be the number of additions used by Fibonacci(n).

```
a(0) = 0
a(1) = 0
a(n) = a(n-2) + a(n-1) + 1.
```

Exercise C-4.3: $a(n) = \text{Fibonacci}(n+1) - 1$.

17

17

Exponential Space Algorithm!!!

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
```

There is very little overhead in calling this function, as it has only one word of local memory, for the parameter n . However, when we try to compute $\text{factorial}(20)$, there will end up being 21 words of memory allocated - one for each invocation of the function.

The input takes $O(\log n)$ space, but the code takes $O(n)$, exponential in terms of $\log n$.

18

Limitations of Recursion

- Multiple recursive calls may involve extensive overhead because they use calls.
- When a call is made, it takes time to build a stackframe and push it onto the system stack.
- Conversely, when a return is executed, the stackframe must be popped from the stack and the local variables reset to their previous values – this also takes time.

19

The overheads that may be associated with a function call are:

- **Space:** Every invocation of a function call may require space for parameters and local variables, and for an indication of where to return when the function is finished. Typically this space (allocation record) is allocated on the stack and is released automatically when the function returns. Thus, a recursive algorithm may need space proportional to the number of nested calls to the same function.
- **Time:** The operations involved in calling a function - allocating, and later releasing, local memory, copying values into the local memory for the parameters, branching to/returning from the function - all contribute to the time overhead.

20

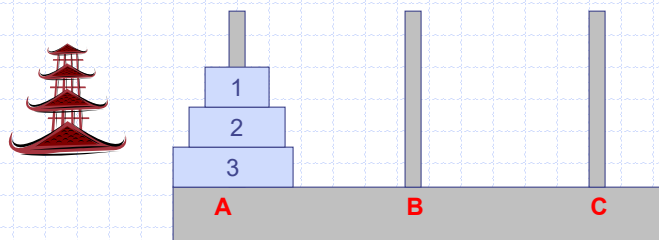
Summary

- Recursion is based upon calling the same function over and over, whereas iteration simply 'jumps back' to the beginning of the loop. A function call is often more expensive than a jump.
- In general, there is no reason to incur the overhead of recursion when its use does not gain anything.
- Recursion is truly valuable when a problem has no simple iterative solution.

21

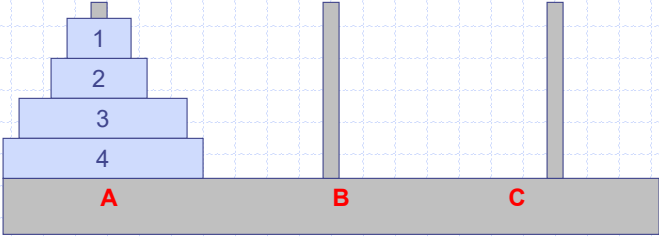
Hanoi Tower - *Instructions*

- 1.** Transfer all the disks from pole A to pole B.
- 2.** You may move only ONE disk at a time.
- 3.** A large disk may not rest on top of a smaller one at anytime.



22

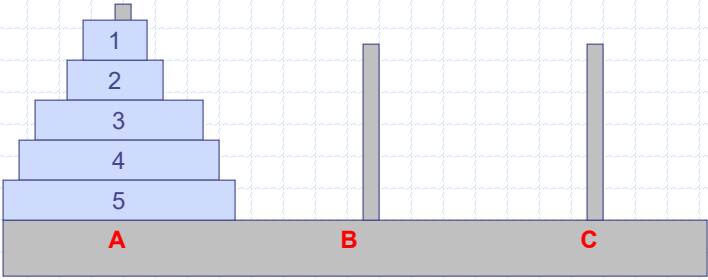
Try this one!



Shortest number of moves??


23

And this one

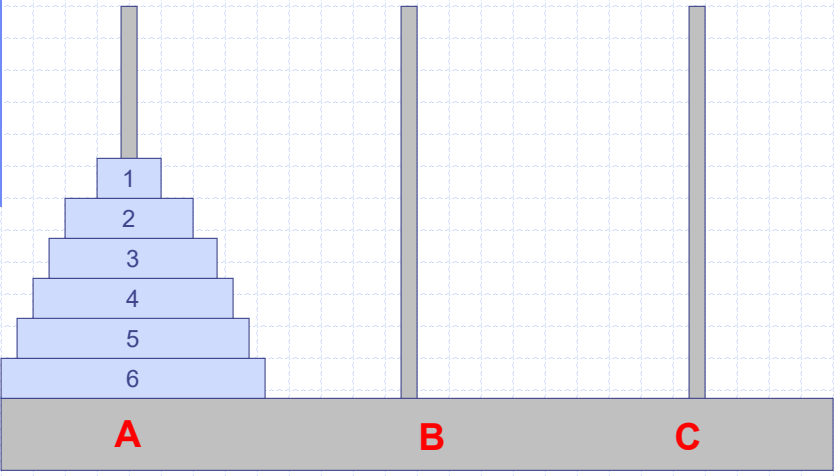


Shortest number of moves??

24



Now try this one!



Shortest number of moves??

25

How to solve Tower of Hanoi of n disks?

- If $n = 1$, "move disk 1 from A to B", done.
- If $n > 1$,
 1. Solve the Tower of Hanoi of $n-1$ disks, from A to C;
 2. "move disk n from A to B"
 3. Solve the Tower of Hanoi of $n-1$ disks, from C to B.

```

static Hanoi (int n, char A, char B, char C) {
    if (n==1) println("move disk 1 from " + A + " to " + B);
    else {
        Hanoi(n-1, A, C, B);
        println("move disk " + n + " from " + A + " to " + B);
        Hanoi(n-1, C, B, A);
    }
}

```

Counting the moves:
Let $f(n)$ be the number of moves for n disks.

$f(1) = 1;$
 $f(n) = 2f(n-1) + 1.$

26

26

Let $f(n)$ be the least number of moves for n disks.

$$f(1) = 1;$$

$$f(n) = 2f(n-1) + 1.$$

Number of Disks	Number of Moves
1	$f(1) = 1$
2	$f(2) = 2*1 + 1 = 3$
3	$f(3) = 2*3 + 1 = 7$
4	$f(4) = 2*7 + 1 = 15$
5	$f(5) = 2*15 + 1 = 31$
6	$f(6) = 2*31 + 1 = 63$

27

Let $f(n)$ be the number of moves for n disks.

$$f(1) = 1;$$

$$f(n) = 2f(n-1) + 1.$$

Prove: $f(n) = 2^n - 1$.

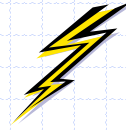
By induction.

- Base step: $n = 1$.
 - Left = $f(1) = 1$;
 - Right = $2^1 - 1 = 1$
- Induction hypothesis: $f(n-1) = 2^{n-1} - 1$.
- Inductive step:
 - Left = $f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$.
 - Right = $2^n - 1$.

28

28

Exponential Growth



So the formula for finding the number of steps it takes to transfer n disks from post A to post C is:

$$2^n - 1$$

- If $n = 64$, the number of moves of single disks is 2 to the 64th minus 1, or 18,446,744,073,709,551,615 moves! If one worked day and night, making one move every second it would take slightly more than 580 billion years to accomplish the job! - far, far longer than some scientists estimate the solar system will last.

29

Main Benefits of Recursive Algorithms

- Invariably recursive functions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.
- The program directly reflects the abstract solution strategy (algorithm).
- From a practical software engineering point of view these are important benefits, greatly enhancing the cost of maintaining the software.

30

Consider the following program for computing the Fibonacci function.

```

int s1, s2 ;
int fib (int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else {
        s1 = fib(n-1);
        s2 = fib(n-2);
        return s1 + s2;
    }
}

```

fib(2) = ?
fib(3) = ?
fib(4) = ?

31

The main thing to note here is that the variables that will hold the intermediate results, S1 and S2, have been declared as global variables.

- This is a mistake. Although the function looks just fine, its correctness crucially depends on having local variables for storing all the intermediate results. As shown, it will not correctly compute the Fibonacci function for n=4 or larger. However, if we move the declaration of s1 and s2 inside the function, it works perfectly.
- This sort of bug is very hard to find, and bugs like this are almost certain to arise whenever you use global variables to store intermediate results of a recursive function.

32

From Recursion to Iteration

- Most recursive algorithms can be translated, by a fairly mechanical procedure, into iterative algorithms. Sometimes this is very straightforward - for example, most compilers detect a special form of recursion, called **tail recursion**, and automatically translate into iteration without your knowing. Sometimes, the translation is more involved: for example, it might require introducing an explicit stack with which to "fake" the effect of recursive calls.

33

What is Tail Recursion?

- Recursive methods are either
 - Tail recursive
 - Nontail recursive
- Tail recursive method has the recursive call as the last operation in the method.
- Recursive methods that are not tail recursive are called non-tail recursive.

34

Is Factorial Tail Recursive?

- Is the method `facto` a tail recursive method?

```
int facto(int x){
    if (x==0)
        return 1;
    else
        return x*facto(x-1);
}
```

- When returning back from a recursive call, there is still one pending operation, multiplication.
- Therefore, `facto` is a non-tail recursive method.

35

Another Example

- Is this method tail recursive?

```
void tail(int i) {
    if (i>0) {
        system.out.print(i+"")
        tail(i-1)
    }
}
```

It is tail recursive!

36

Third Example

- Is the following program tail recursive?

```
void prog(int i) {
    if (i>0) {
        prog(i-1);
        System.out.print(i+"");
        prog(i-1);
    }
}
```

- No, because there is an earlier recursive call, which is not the last operation.
- In tail recursion, the recursive call should be the last operation, and there should be no earlier recursive calls whether direct or indirect.

37

Advantage of Tail Recursive Method

- Tail Recursive methods are easy to convert to iterative.

```
void tail(int i){
    if (i>0) {
        system.out.println(i+"");
        tail(i-1)
    }
}
```

```
void iterative(int i){
    for (;i>0;i--)
        System.out.println(i+"");
}
```

- Smart compilers can detect tail recursion and convert it to iterative to optimize code
- Used to implement **for** loops in languages that do not support loop structures explicitly (e.g. prolog)

38

Converting Non-tail to Tail Recursive

- A non-tail recursive method can be converted to a tail-recursive method by means of an "auxiliary" parameter used to form the result.
- The technique is usually used in conjunction with an "auxiliary" function. This is simply to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

```
int fact_aux(int n, int result) {
    if (n == 0) return result;
    return fact_aux(n - 1, n * result)
}
```

```
int fact(n) {
    return fact_aux(n, 1);
}
```

```
int fact(int n){
    if (n==0) return 1;
    else return n*fact(n-1);
}
```

39

Converting Non-tail to Tail Recursive

- A tail-recursive Fibonacci method can be implemented by using two auxiliary parameters for accumulating results.

```

                                auxiliary parameters!
int fib_aux ( int i , int nextResult, int result)
{
    if (i == 0)
        return result;
    return fib_aux(i - 1, nextResult + result, nextResult);
}
```

To calculate fib(n) , call fib_aux(n,1,0)

40

Patterns for Converting Tail Recursive to Iterative

```
F(x) {
  if (P(x)) return G(x);
  S(x);
  return F(H(x));
}
```

- If P(x) is true, the value of F(x) is the value of some other function G(x). Otherwise, the value of F(x) is the value of the function F on some other value, H(x)

```
F(x) {
  while (!P(x)) {
    S(x);
    x = H(x);
  }
  return G(x);
}
```

41

Converting Tail Recursive to Iterative

```
int fact_aux(int n, int result) {
  if (n == 1) return result;
  return fact_aux(n - 1, n * result);
}
```

```
F(x) {
  if (P(x)) return G(x);
  S(x);
  return F(H(x));
}
```

the function F is fact_aux

$x \Leftrightarrow (n, \text{result})$, tuple of the two parameters

$P(n, \text{result}) \Leftrightarrow (n == 1)$

$G(n, \text{result}) \Leftrightarrow \text{result}$

$H(n, \text{result}) \Leftrightarrow (n - 1, n * \text{result})$

$S(n, \text{result}) \Leftrightarrow \text{nothing}$

```
int fact_iter(int n, int result) {
  while (n != 1) {
    (n, result) = (n - 1, n * result);
  }
  return result;
}
```

```
F(x) {
  while (!P(x)) {
    S(x);
    x = H(x);
  }
  return G(x);
}
```

42

Converting Tail Recursive to Iterative

```

int fib_aux ( int n, int nRes, int res){
    if (n == 0) return res;
    return fib_aux(n - 1, nRes + res, nRes);
}

F(x) {
    if (P(x)) return G(x);
    S(x);
    return F(H(x));
}

```

the function F is fib_aux
 $x \Leftrightarrow (n, nRes, res)$, tuple of the three parameters
 $P(n, nRes, res) \Leftrightarrow (n == 0)$
 $G(n, nRes, res) \Leftrightarrow res$
 $H(n, nRes, res) \Leftrightarrow (n - 1, nRes + res, nRes)$
 $S(n, result) \Leftrightarrow \text{nothing}$

```

int fib_iter(int n, int nRes, int res) {
    while (n != 0) {
        (n, nRes, res) = (n - 1, nRes + res, nRes);
    }
    return res;
}

F(x) {
    while (!P(x)) {
        S(x);
        x = H(x);
    }
    return G(x);
}

```

43

Recursive Selection Sort

```

void ssort_rec(int n, int A[]) {
    if (n <= 1) return;
    int i = arg_max(n, A); // find index of max in A
    swap(A, i, n-1); // swap elements at i and n-1.
    ssort_rec(n - 1, A);
}

```

Quiz Questions:

ssort_rec is tail-recursion!!!
 What is the result of removing this tail-recursion?

Suppose $\text{arg_max}(i, A[])$ takes $O(i)$ time.
 What is the time complexity of $\text{ssort_rec}(n, A)$?

44

Recursion

- Recursion is more than just a programming technique. It has two other uses in computer science and software engineering, namely:
 - As a way of describing, defining, or specifying things.
 - As a way of designing solutions to problems (divide and conquer, dynamic programming, etc).

45

Defining sets via recursion

- Same as mathematical induction:
 - Base case (or basis step)
 - Recursive step
- Example: the set of positive integers
 - Basis step: $1 \in S$
 - Recursive step: if $x \in S$, then $x+1 \in S$

46

46

Defining sets via recursion

- Give recursive definitions for:
 - a) The set of odd positive integers
 - $1 \in S$
 - If $x \in S$, then $x+2 \in S$
 - b) The set of positive integer powers of 3
 - $3 \in S$
 - If $x \in S$, then $3*x \in S$
 - c) The set of polynomials with integer coefficients
 - $0 \in S$
 - If $p(x) \in S$, then $p(x) + cx^n \in S$
 - $c \in \mathbf{Z}$, $n \in \mathbf{Z}$ and $n \geq 0$

47

47

Defining strings via recursion

- Terminology
 - ε is the empty string: ""
 - Σ is the set of all letters: $\{ a, b, c, \dots, z \}$
 - ◆ The set of letters can change depending on the problem
- We can define a set of strings Σ^* as follows
 - Base step: $\varepsilon \in \Sigma^*$
 - If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$
 - Thus, Σ^* is the set of all the possible strings that can be generated with the alphabet

48

48

Defining strings via recursion

- Let $\Sigma = \{ 0, 1 \}$
- Thus, Σ^* is the set of all binary strings
 - Or all possible computer files

49

49

String length via recursion

- How to define string length recursively?
 - Basis step: $len(\epsilon) = 0$
 - Recursive step: $len(wx) = len(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$
- Example: $len(\text{"aaa"})$
 - $len(\text{"aaa"}) = len(\text{"aa"}) + 1$
 - $len(\text{"aa"}) = len(\text{"a"}) + 1$
 - $len(\text{"a"}) = len(\text{""}) + 1$
 - $len(\text{""}) = 0$
 - Result: 3

50

50

Strings via recursion example

- Given a string $x = a_1a_2\dots a_n$, x^R stands for its reversal: $x^R = a_n a_{n-1} \dots a_1$. Eg. $x = abc$, $x^R = cba$.
- A string x is a palindrome if $x = x^R$. Eg. $x = aba$.
- Give a recursive definition for P , the set of string that are palindromes
 - We will define set P , which is the set of all palindromes
- **Basis step:** $\varepsilon \in P$
- **Second basis step:** $x \in P$ when $x \in \Sigma$
- **Recursive step:** $xpx \in P$ if $x \in \Sigma$ and $p \in P$

51

51

How many binary strings of length n that do not contain the pattern 11?

- $n = 0$: 1 string (ε , the empty string)
- $n = 1$: 2 strings (0 and 1)
- $n = 2$: 3 strings (00, 01, 10)
- $n = 3$: 5 strings (000, 001, 010, 100, 101)
- $n = 4$: 8 strings (0000, 0001, 0010, 0100, 1000, 0101, 1001, 1010)

- Any pattern?
- A Fibonacci sequence!

52

52

How many binary strings of length n that do not contain the pattern 11?

- $n = 2$: 3 strings (00, 01, 10)
- $n = 3$: 5 strings (000, 001, 010, 100, 101)
- $n = 4$: 8 strings (0000, 0001, 0010, 0100, 1000, 0101, 1001, 1010)
- The strings of $n=4$ can be divided into two classes:
 - $X = \{ 0000, 0001, 0010, 0100, 0101 \}$ and
 - $Y = \{ 1000, 1001, 1010 \}$
 - X can be obtained from $n = 3$: adding a leading 0
 - Y can be obtained from $n = 2$: adding leading 10.

53

53

How many binary strings of length n that do not contain the pattern 11?

- Let S_n be the set of binary strings of length n that do not contain the pattern 11.
- For any string x in S_{n-1} , $y = 0x$ is a string of S_n .
- For any string x in S_{n-2} , $z = 10x$ is a string of S_n .
- Any string of S_n is either y or z above.
- Hence $S_n = 0S_{n-1} \cup 10S_{n-2}$, or $|S_n| = |S_{n-1}| + |S_{n-2}|$
- From $|S_0| = 1$, $|S_1| = 2$, we can compute any $|S_n|$.

54

54

Recursive Functions on Natural Numbers

- The set of natural numbers
 - Basis step: $0 \in \mathcal{N}$
 - Recursive step: if $x \in \mathcal{N}$, then $x+1 \in \mathcal{N}$
- Define plus on \mathcal{N} :
 - Basis step: $\text{plus}(0, y) = y$ for all $y \in \mathcal{N}$
 - Recursive step: $\text{plus}(x+1, y) = \text{plus}(x, y)+1$ for all $x, y \in \mathcal{N}$
- Prove that $\text{plus}(x, y) = x+y$.
- Induction on x :
 - Basis step $x=0$: $\text{plus}(0, y) = 0+y$ for all $y \in \mathcal{N}$
 - Induction hypothesis: $\text{plus}(x, y) = x+y$
 - Recursive step $x = u+1$: $\text{plus}(u+1, y) = (u+1)+y$ for all $u, y \in \mathcal{N}$
 - Left = $\text{plus}(u+1, y) = \text{plus}(u, y)+1 = u+y+1$
 - Right = $(u+1)+y = u+y+1$
- This induction proof is also called "structural induction".

55

55

Recursive Functions on Natural Numbers

- Define plus on \mathcal{N} :
 - Basis step: $\text{plus}(0, y) = y$ for all $y \in \mathcal{N}$
 - Recursive step: $\text{plus}(x+1, y) = \text{plus}(x, y)+1$ for all $x, y \in \mathcal{N}$
- Define mult on \mathcal{N} :
 - Basis step: $\text{mult}(0, y) = 0$ for all $y \in \mathcal{N}$
 - Recursive step: $\text{mult}(x+1, y) = \text{plus}(\text{mult}(x, y), y)$ for all $x, y \in \mathcal{N}$
- Prove that $\text{mult}(x, y) = x*y$.
- Induction on x :
 - Basis step $x=0$: $\text{mult}(0, y) = 0*y$ for all $y \in \mathcal{N}$
 - Induction hypothesis: $\text{mult}(x, y) = x*y$
 - Recursive step $x = u+1$: $\text{mult}(u+1, y) = (u+1)*y$ for all $u, y \in \mathcal{N}$
 - Left = $\text{mult}(u+1, y) = \text{plus}(\text{mult}(u, y), y) = u*y+y$
 - Right = $(u+1)*y = u*y+y$

56

56

Recursion Functions on Strings

- Given a string $x = a_1a_2\dots a_n$, x^R stands for its reversal:
 $x^R = a_n a_{n-1} \dots a_1$. Eg. $x = abc$, $x^R = cba$.
- How to define x^R recursively?
 - Base step: $\varepsilon \in \Sigma^*$
 - Recursive step: If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$
- Basis step: $\varepsilon^R = \varepsilon$
- Recursive step: $(wx)^R = x(w)^R$ if $x \in \Sigma$ and $w \in \Sigma^*$
- Theorem: $(aw)^R = (w)^R a$ for all $a \in \Sigma$, $w \in \Sigma^*$.

57

57

How to prove $(aw)^R = (w)^R a$?

- Base step: $\varepsilon \in \Sigma^*$
- Recursive step: If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$
- Basis step: $\varepsilon^R = \varepsilon$
- Recursive step: $(wx)^R = x(w)^R$ if $x \in \Sigma$ and $w \in \Sigma^*$
- Structural Induction Proof:
 - Basis case $w = \varepsilon$: $(a\varepsilon)^R = (\varepsilon)^R a$ (easy)
 - Induction hypothesis: $(aw)^R = (w)^R a$
 - Inductive case: $(a(wx))^R = (wx)^R a$ if $x \in \Sigma$ and $w \in \Sigma^*$
 - Left = $(a(wx))^R = ((aw)x)^R = x(aw)^R = x(w)^R a$.
 - Right = $(wx)^R a = (x(w)^R) a = x(w)^R a$.

58

58

How to prove $((w)^R)^R = w$?

- Base step: $\varepsilon \in \Sigma^*$
- Recursive step: If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$
- Basis step: $\varepsilon^R = \varepsilon$
- Recursive step: $(wx)^R = x(w)^R$ if $x \in \Sigma$ and $w \in \Sigma^*$
- Structural Induction Proof:
 - Basis case: $((\varepsilon)^R)^R = \varepsilon$ (easy)
 - Induction hypothesis: $((w)^R)^R = w$
 - Inductive case: $((wx)^R)^R = wx$ if $x \in \Sigma$ and $w \in \Sigma^*$
 - Lemma: $(aw)^R = (w)^R a$
 - Left = $((wx)^R)^R = (x(w)^R)^R = ((w)^R)^R x = wx.$
 - Right = $wx.$

59

59

How to prove $\text{len}((w)^R) = \text{len}(w)$?

- Basis step: $\varepsilon^R = \varepsilon$
- Recursive step: $(wx)^R = x(w)^R$ if $x \in \Sigma$ and $w \in \Sigma^*$
- Basis step: $\text{len}(\varepsilon) = 0$
- Recursive step: $\text{len}(wx) = \text{len}(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$
- Structural Induction Proof:
 - Basis case: $\text{len}((\varepsilon)^R) = \text{len}(\varepsilon)$ (easy)
 - Induction hypothesis: $\text{len}((w)^R) = \text{len}(w)$
 - Inductive case: $\text{len}((wx)^R) = \text{len}(wx)$ if $x \in \Sigma$ and $w \in \Sigma^*$
 - Lemma: $\text{len}(aw) = 1 + \text{len}(w)$ if $a \in \Sigma$ and $w \in \Sigma^*$
 - Left = $\text{len}((wx)^R) = \text{len}(x(w)^R) = 1 + \text{len}((w)^R) = 1 + \text{len}(w).$
 - Right = $\text{len}(wx) = \text{len}(w) + 1.$

60

60

The append function on strings

- Basis step: $len(\varepsilon) = 0$
- Recursive step: $len(wx) = len(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$
- The function $app(x, y)$ "appends" x and y together.
- $app("abc", "xyz") = "abcxyz"$.
- Basis step: $app(v, \varepsilon) = v$ if $v \in \Sigma^*$
- Recursive step: $app(v, wx) = app(v, w)x$ if $v, w \in \Sigma^*$ and $x \in \Sigma$.
- How to prove $len(app(v, w)) = len(v) + len(w)$?

61

61

How to prove $len(app(v, w)) = len(v) + len(w)$?

- Basis step: $len(\varepsilon) = 0$
- Recursive step: $len(wx) = len(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$
- Basis step: $app(v, \varepsilon) = v$ if $v \in \Sigma^*$
- Recursive step: $app(v, wx) = app(v, w)x$ if $v, w \in \Sigma^*$ and $x \in \Sigma$.
- Structural Induction Proof:
 - Basis case: $len(app(v, \varepsilon)) = len(v) + len(\varepsilon)$ (easy)
 - Induction hypothesis: $len(app(v, w)) = len(v) + len(w)$
 - Inductive case: $len(app(v, wx)) = len(v) + len(wx)$
 - Lemma: $len(ax) = 1 + len(x)$ if $a \in \Sigma$ and $x \in \Sigma^*$
 - Left = $len(app(v, wx)) = len(app(v, w)x) = len(app(v, w)) + 1 = len(v) + len(w) + 1$
 - Right = $len(v) + len(wx) = len(v) + len(w) + 1$.

62

62