# Recursion: Introduction and Correctness

<table>
<thead>
<tr>
<th>Lecture B</th>
<th>Jones</th>
<th>MWF 9-9:50am</th>
<th>PCNYN 109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture D</td>
<td>Russell</td>
<td>MWF 4-4:50am</td>
<td>Center 101</td>
</tr>
</tbody>
</table>

http://cseweb.ucsd.edu/classes/sp16/cse21-bd/

April 11, 2016
Today's plan

1. What's recursion?
2. Correctness of recursive algorithms
3. Recurrence relations

_in the textbook:_ Chapter 5 on Induction and Recursion and Sections 8.1-8.3 on Recurrence Equations
What's recursion?

When the function calls itself.

When something relies on previous elements.

A relation that depends on itself.

[smaller inputs]

[previous elements]
What's recursion?

Solving a problem by successively reducing it to the same problem with smaller inputs.

Rosen p. 360
Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

$$b_1 \ b_2 \ b_3 \ \ldots \ b_n$$

A substring of length $k$ of that string is a string of the form

$$b_i \ b_{i+1} \ b_{i+2} \ \ldots \ b_{i+k-1}$$

The substring 010 can be found in several places 0100101000.
A string is a finite sequence of symbols such as 0s and 1s. Which we write as

\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

A substring of length \( k \) of that string is a string of the form

\[ b_i \ b_{i+1} \ b_{i+2} \ \ldots \ b_{i+k-1} \]

The substring 010 can be found in several places 0100101000.
Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

\[ b_1 b_2 b_3 \ldots b_n \]

A substring of length \( k \) of that string is a string of the form

\[ b_i b_{i+1} b_{i+2} \ldots b_{i+k-1} \]

The substring 010 can be found in several places 0100101000.
Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

\[ b_1 \ b_2 \ b_3 \ldots \ b_n \]

A substring of length \( k \) of that string is a string of the form

\[ b_i \ b_{i+1} \ b_{i+2} \ldots \ b_{i+k-1} \]

The substring 010 can be found in several places 0100101000.
Example – Counting a pattern: WHAT

Count how many times the substring 00 occurs in the string 0100101000.

A. 0
B. 1
C. 2
D. 3
E. 4
Problem: Given a string (finite sequence) of 0s and 1s $b_1 \ b_2 \ b_3 \ ... \ \ b_n$, count how many times the substring $00$ occurs in the string.

```
num = 0
for i from 1 to n-1
  if $b_i = 0$ and $b_{i+1} = 0$ then num++
return num.
```

**Design an algorithm to solve this problem**
Example – Counting a pattern: HOW

An Iterative Algorithm
Step through each position and see if pattern starts there.

procedure countDoubleIter(b₁, . . . , bₙ : each 0 or 1)
  count := 0
  if n < 2 then return 0
  for i := 1 to n − 1
    if (bᵢ = 0 and bᵢ₊₁ = 0) then
      count := count + 1
  return count
Example – Counting a pattern: HOW

A Recursive Algorithm
Does pattern occur at the head? Then solve for the rest.

procedure countDoubleRec(b₁, …, bₙ : each 0 or 1)
  if n < 2 then return 0
  if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, …, bₙ)
  return countDoubleRec(b₂, …, bₙ)
This example shows that essentially the same algorithm can be described as iterative or recursive.

But describing an algorithm recursively can give us new insights and sometimes lead to more efficient algorithms.

It also makes correctness proofs more intuitive.
Template for proving correctness of recursive alg.

Overall Structure: Prove that algorithm is correct on inputs of size $n$ by induction on $n$.

Base Case: The base cases of recursion will be the base cases of induction. For each one, say what the algorithm does and say why it is the correct answer.
Template for proving correctness of recursive alg.

(Strong) Inductive Hypothesis: The algorithm is correct on all inputs of size (up to) $k$.

Goal (Inductive Step): Show that the algorithm is correct on any input of size $k + 1$.

Note: The induction hypothesis allows us to conclude that the algorithm is correct on all recursive calls for such an input.
Inside the inductive step

1. Express what the algorithm does in terms of the answers to the recursive calls to smaller inputs.

2. Replace the answers for recursive calls with the correct answers according to the problem (inductive hypothesis.)

3. Show that the result is the correct answer for the actual input. (k+1)
Example – Counting a pattern

procedure countDoubleRec(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

Goal: Prove that for any string b₁, b₂, b₃, ... bₙ,
countDoubleRec(b₁, b₂, b₃, ... bₙ) = the number of places the substring 00 occurs.

Overall Structure: We are proving this claim by induction on n.
Proof of Base Case

procedure countDoubleRec(b_1, ..., b_n : each 0 or 1)
    if n < 2 then return 0  Base Case
    if (b_1 = 0 and b_2 = 0) then return 1 + countDoubleRec(b_2, ..., b_n)
    return countDoubleRec(b_2, ..., b_n)

Base Case: n < 2 i.e. n = 0, n = 1.

n = 0: The only input is the empty string which has no substrings. The algorithm returns 0 which is correct.

n = 1: The input is a single bit and so has no 2-bit substrings. The algorithm returns 0 which is correct.
Proof: Inductive hypothesis

```
procedure countDoubleRec(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)
```

Inductive hypothesis: Assume that for any input string of length \( k \), \( countDoubleRec(b₁, b₂, b₃, ... bₖ) \) = the number of places the substring 00 occurs.

Inductive Step: We want to show that \( countDoubleRec(b₁, b₂, b₃, ... bₖ₊₁) \) = the number of places the substring 00 occurs for any input of length \( k + 1 \).
Proof: Inductive step

procedure countDoubleRec(b₁, …, bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, …, bₙ)
    return countDoubleRec(b₂, …, bₙ)

Case 1: b₁ = 0 and b₂ = 0: countDoubleRec(b₁, b₂, b₃, … bₖ₊₁) = 1 + countDoubleRec(b₂, b₃, … bₖ₊₁) = 1 + the number of occurrences of 00 in b₂, b₃, … bₖ₊₁ = one occurrence of 00 in first two positions + number of occurrences in later appearances.

Case 2: otherwise:
    countDoubleRec(b₁, b₂, b₃, … bₖ₊₁) = countDoubleRec(b₂, b₃, … bₖ₊₁) = the number of occurrences of 00 in b₂, b₃, … bₖ₊₁ = the number of occurrences starting at the second position = the total number of occurrences since the first two are not an occurrence.
Proof: Conclusion

procedure countDoubleRec(b₁, . . . , bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, . . . , bₙ)
    return countDoubleRec(b₂, . . . , bₙ)

We showed the algorithm was correct for inputs of length 0 and 1. And we showed that if it is correct for inputs of length k > 0, then it is correct for inputs of length k + 1.

Therefore, by induction on the input length, the algorithm is correct for all inputs of any length. 😊
Time analysis for counting patterns.

procedure countDoubleRec(b₁, ..., bₙ : each 0 or 1)
    if \( n < 2 \) then return 0
    if \( b₁ = 0 \) and \( b₂ = 0 \) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

How long does this algorithm take?

It’s hard to give a direct answer because it seems we need to know how long the algorithm takes to know how long the algorithm takes.

Solution: We really need to know how long the algorithm takes on smaller instances to know how long it takes for larger lengths.
A recurrence relation
(also called a recurrence or recursive formula)
expresses $f(n)$
in terms of previous values, such as
$f(n-1)$, $f(n-2)$, $f(n-3)$….

Example:

$f(n) = 3*f(n-1) + 7$ tells us how to find $f(n)$ from $f(n-1)$

$f(1) = 2$ also need a base case to tell us where to start
Recurrence relation for time analysis

procedure countDoubleRec(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

Let T(n) represent the time it takes for this algorithm on an input of length n.

Then T(n) = T(n-1) + c for some constant c.
(The recursive call is of length n-1 and so it takes time T(n – 1). The rest of the algorithm is constant time.)
Solving the Recurrence

\[ T(0) = T(1) = c_0 \]
\[ T(n) = T(n-1) + c \]

To find a closed form of \( T(n) \), we can unravel this recurrence.

\[
T(n) = T(n-1) + c = (T(n-2) + c) + c = ((T(n-3) + c) + c) + c = \cdots = T(1) + c + c + \cdots + c
\]
\[
= T(1) + (n-1)c = cn + c_0 - c \in \Theta(n)
\]
Two ways to solve recurrences

1. Guess and Check

Start with small values of n and look for a pattern. Confirm your guess with a proof by induction.

2. Unravel

Start with the general recurrence and keep replacing n with smaller input values. Keep unraveling until you reach the base case.
Subsequences

Given a string (finite sequence) of symbols

\[ b_1 \ b_2 \ b_3 \ldots \ b_n \]

A subsequence of length \( k \) of that string is a string of the form

\[ b_{i_1}, b_{i_2}, \ldots, b_{i_k} \]

where \( 1 \leq i_1 < i_2 < \ldots < i_k \leq n \). The subsequence 010 can be found in a whole bunch of places in 0100101000.
Given two strings (finite sequences) of characters*:

\[ a_1 \ a_2 \ a_3 \ \ldots \ a_n \]
\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

what's the length of the longest string which is a subsequence in both strings?

What should be the output for the strings AGGACAT and ATTACGAT?

A. 1
B. 2
C. 3
D. 4
E. 5

* Could be 0s and 1s, or ACTG in DNA
Example – Longest Common Subsequence: WHAT

Given two strings (finite sequences) of characters*

\[ a_1 \ a_2 \ a_3 \ \ldots \ \ a_n \]
\[ b_1 \ b_2 \ b_3 \ \ldots \ \ b_n \]

what's the length of the longest string which is a subsequence in both strings?

What should be the output for the strings AGGACAT and ATTACGAT?

A. 1
B. 2
C. 3
D. 4
E. 5

* Could be 0s and 1s, or ACTG in DNA
Example – Longest Common Subsequence: HOW

Given two strings (finite sequences) of characters*

\[
\begin{align*}
   a_1 & \quad a_2 & \quad a_3 & \quad \ldots & \quad a_n \\
   b_1 & \quad b_2 & \quad b_3 & \quad \ldots & \quad b_n
\end{align*}
\]

what's the length of the longest string which is a subsequence in both strings?

\[
\text{LCS}(a_1 \ldots a_n, b_1 \ldots b_n) \\
\text{if } n=0 \text{ then return 0}
\]

* Design a recursive algorithm to solve this problem

* Could be 0s and 1s, or ACTG in DNA
Example – Longest Common Subsequence: HOW

A Recursive Algorithm
Do the strings agree at the head? Then solve for the rest.

```
procedure lcsRec(a_1, ..., a_m; b_1, ..., b_n)
    if (m = 0 or n = 0) then return 0
    if a_1 = b_1 then return 1 + lcsRec(a_2, ..., a_m; b_2, ..., b_n)
    return max(lcsRec(a_1, ..., a_m; b_2, ..., b_n), lcsRec(a_2, ..., a_m; b_1, ..., b_n))
```
Example – Longest Common Subsequence: HOW

A Recursive Algorithm
Do the strings agree at the head? Then solve for the rest.

```
procedure lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)
    \text{if } (m = 0 \text{ or } n = 0) \text{ then return } 0
    \text{if } a_1 = b_1 \text{ then return } 1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)
    \text{return } \max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))
```

What would an iterative algorithm look like?
### Example – Binary strings avoiding 00

How many binary strings of length $n$ are there which do not have two consecutive 0s?

<table>
<thead>
<tr>
<th>$n$</th>
<th>OK</th>
<th>NOT OK</th>
<th>How many OK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0, 1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>00, 001, 100</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>01, 10, 11</td>
<td>00, 001, 100</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>010, 011, 101, 110, 111</td>
<td>00, 001, 100</td>
<td>5</td>
</tr>
</tbody>
</table>
Example – Binary strings avoiding 00

How many binary strings of length n are there which do not have two consecutive 0s?

**Recurrence??** \[ B(n) = \text{the number of OK strings of length } n \]

Any (long) "OK" binary string must look like

\[
\begin{align*}
\text{1} & \underline{\quad} \quad \text{or} \quad \text{01} \underline{\quad} \\
\end{align*}
\]

"OK" binary string of length n-1

"OK" binary string of length n-2
Example – Binary strings avoiding 00

How many binary strings of length n are there which do not have two consecutive 0s?

**Recurrence??**  
\[ B(n) = B(n-1) + B(n-2) \]  
\[ B(0) = 1, \ B(1) = 2 \]

Any (long) "OK" binary string must look like

1\_________ or 01\_________
Example – Binary strings avoiding 00

\[ B(n) = B(n-1) + B(n-2) \quad B(0) = 1, B(1) = 2 \]

<table>
<thead>
<tr>
<th>n</th>
<th>B(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>n</td>
<td>??</td>
</tr>
</tbody>
</table>

Fibonacci numbers
Spot the Recursion: Tromino Puzzle

We have a $2^n$ by $2^n$ board missing one square (can be anywhere on the board). We want to cover all the squares on the board (except the missing one) by L-shaped tiles each formed by three adjacent squares. Trominoes cannot overlap.

What's $n$ for this example?
A. 0  
B. 1  
C. 2  
D. 3  
E. 4
We have a $2^n$ by $2^n$ board missing one square (can be anywhere on the board). We want to cover all the squares on the board (except the missing one) by L-shaped tiles each formed by three adjacent squares. Trominoes cannot overlap.

*Where’s the recursion?*
We have a $2^n$ by $2^n$ board missing one square (can be anywhere on the board). We want to cover all the squares on the board (except the missing one) by L-shaped tiles each formed by three adjacent squares. Trominoes cannot overlap.

**Where's the recursion?**

Reduced to 4 similar problems, each on a board of size $2^{n-1}$ by $2^{n-1}$. 
Next Time…

- Time analysis of recursive algorithms
- Solving recurrence relations