CSCI 261
Computer Science II
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Are Loops the Only Way to Do Repetition?

- Someone's sitting behind me on an airplane wants to know their row
  - Approach 1: Go up to first row, walk back and count up
    - Analogous to a loop
  - Approach 2: Ask the person in front of me for their row and add 1!
Are Loops the Only Way to Do Repetition?

Someone's sitting behind me on an airplane wants to know their row

- Approach 1: Go up to first row, walk back and count up
  - Analogous to a loop
- Approach 2: Ask the person in front of me for their row and add 1!

Demo of Approach 2:

- What's my row number?
  - Ask person in front for their row number
  - Write on yellow slip: The seat-in-front's row + 1
  - Pass yellow slip back to whomever asked you
Thinking Recursively

- Recursion:
  - Applies to problems that can be broken into smaller problems of the same nature (called a subproblem)
  - Example:
    - Just tell the person in front to do exactly what I'm doing!
    - Until they are on the first row (sub-problem is trivial)

```java
public int getRowNumber() {
    if (someoneInFront) {  // Sweet, I get to be lazy (Recursive case)
        int rowInFront = getRowNumber() of person in front
        return rowInfront + 1;
    }
    else {  // Trivial, I'm sitting in first row! (Base case)
        return 1;
    }
}
```
Recursion

- **Base case(s):** Problem becomes trivial!
  - Usually returns a constant value to the caller (doesn't always return something)
  - *Does not* call itself afterwards (terminates recursion!)
Recursion is characterized by:

• **Base case(s):** Problem becomes trivial!
  - Usually returns a constant value to the caller (doesn't always return something)
  - *Does not* call itself afterwards (terminates recursion!)

• **Recursive, or general, case(s):** Divide and conquer:
  - Do the minimal unit of work to reduce problem into a smaller sub-problem of the same exact nature
  - Calls *itself* to solve the smaller sub-problem
Example: MultiPrint

- Suppose I want to print a string \( n \) times:

```
multiPrint("Hello World", 4);
> Hello World
> Hello World
> Hello World
> Hello World
```

```
multiPrint("Hello World", 1);
> Hello World
```

```
multiPrint(0);
```
Example: MultiPrint (Cont.)

- Suppose I want to print a string $n$ times:

  ```java
  public void multiPrint(String str, int n) {
  }
  ```

- Base case?

- Recursive case?
Example: MultiPrint (Cont.)

- Suppose I want to print a string $n$ times:

```java
public void multiPrint(String str, int n) {
    if (n == 0) {
    }
}
```

- Base case?
  - When $n == 0$: don't do anything

- Recursive case?
Example: MultiPrint (Cont.)

- Suppose I want to print a string \( n \) times:

```java
public void multiPrint(String str, int n) {
    if (n == 0) {
    }
    else {
        System.out.println(str);
        multiPrint(str, n-1);
    }
}
```

- Base case?
  - When \( n == 0 \): don't do anything

- Recursive case?
  - Print the String once (that's the bare minimum work to reduce problem)
  - Recursively \texttt{multiPrint()} the remaining \( n-1 \) strings
Example: MultiPrint (Vers. 1.0)

- Rewritten to remove empty if-statement

```java
public void multiPrint(String str, int n) {
    if (n != 0) {
        System.out.println(str);
        multiPrint(str, n-1);
    }
}
```

- Base case?
  - When \( n == 0 \): don't do anything

- Recursive case?
  - Print the String once (that's the bare minimum work to reduce problem)
  - Recursively `multiPrint()` the remaining \( n-1 \) strings
Stack Overflow (Infinite Recursion)

Current implementation:

```java
public void multiPrint(String str, int n) {
    if (n != 0) {
        System.out.println(str);
        multiPrint(str, n-1);
    }
}
```

What if we called `multiPrint("Hello", -1);`?

- Hello
- Hello
- Hello
- Hello
- Hello
- Hello
...  
`java.lang.StackOverFlowError`
Example: MultiPrint (Final)

- Re-written to avoid stack overflow

```java
public void multiPrint(String str, int n) {
    if (n > 0) {
        System.out.println(str);
        multiPrint(str, n-1);
    }
}
```

- Time complexity
  - Base case: $T(0) = 0$ steps
  - Recursive case: $T(n) = ??$
    - First statement: `System.out.println(str)` costs 1 step...
    - But what about the second statement?
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Tracing a Recursive Method Call (Important)

- Track contents of all variables, inputs, outputs, and return values.
  - Keep track of the stack(!) of recursive calls which are still waiting

- Example: What does this mystery method do?
  - Try a trace with \( n = 2 \)

```java
public static void mystery(int n) {
    if (n >= 0) {
        System.out.println(n);
        mystery(n-1);
        System.out.println(n);
    }
}
```

- (Trace demo on board)
Fix This Bug! (Trace It!)

- We want this method to print 0, 1, 2, ..., n-1, n
  - Trace it work n = 3

```java
public static void ascendingPrint(int n) {
    int i = 0;
    if (i <= n) {
        System.out.println(i);
        i++;
        ascendingPrint(n-1);
    }
}
```
Fix This Bug! (Trace It!)

- We want this method to print 0, 1, 2, ..., n-1, n

```java
public static void ascendingPrint(int n) {
    int i = 0;
    if (i <= n) {
        System.out.println(i);
        i++;
        ascendingPrint(n-1);
    }
}
```

- Fixed version below:

```java
/* Prints 0, 1, 2, ..., n
 * @param n Number to print up to
 * @param i Always pass in 0
 */
public static void ascendingPrint(int n, int i) {
    if (i <= n) {
        System.out.println(i);
        i++;
        ascendingPrint(n, i);
    }
}
```
Rewritten:

- Why is this better than before?

```java
/* Prints 0, 1, 2, ..., n
 * @param n Number to print up to
 */
public static void ascendingPrint(int n) {
    ascendingPrintHelper(n, 0);
}
private static void ascendingPrintHelper(int n, int i) {
    if (i <= n) {
        System.out.println(i);
        i++;
        ascendingPrintHelper(n, i);
    }
}
```
Fix Another Bug!

- Want to return the number of odd numbers in the given list.
- Idea (this is sound):
  - Remove first element from the list
  - If it's odd, then add 1 to the number of odds in the remaining list
  - If it's even, then add 0 to the number of odds in the remaining list

```java
public static int numOdd(List<Integer> list) {
}
```
Fix Another Bug!

- Want to return the number of odd numbers in the given list.

- Idea (this is sound):
  - Remove first element from the list
  - If it's odd, then add 1 to the number of odds in the remaining list
  - If it's even, then add 0 to the number of odds in the remaining list

```
public static int numOdd(List<Integer> list) {
    int val = list.remove(0);
    if (val % 2 == 1) {
        return 1 + numOdd(list);
    }
    else {
        return numOdd(list);
    }
}
```

- Trace with [1, 2, 3], and edge cases (e.g., list with 1 item, empty list)
Fix Another Bug! (Final)

- Want to return the sum of the odd numbers in the given list.
  - Trace with [1, 2, 3], and edge cases (e.g., list with 1 item, empty list)

```java
public static int numOdd(List<Integer> list) {
    if (list.size() == 0) {
        return 0; // There was no base case!
    }
    int val = list.remove(0);
    if (val % 2 == 1) {
        return 1 + numOdd(list);
    } else {
        return numOdd(list);
    }
}
```

*(Challenge: list becomes empty after you call this method. What if that's not ok?)*
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion

Sierpinski's Carpet
### Factorial

- The factorial \( n! \) is the product of \( n \) and all integers \( < n \)

- For example:
  - \( 7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040 \)
  - \( 3! = 3 \times 2 \times 1 \)
  - \( 1! = 1 \)
  - \( 0! = ? \)

- Example: How many different ways can you order a list of 3 numbers?
  - Answer: \( 3! = 3 \times 2 \times 1 = 6 \)

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Factorial

- What we just saw was the iterative definition of a factorial:

\[
\begin{align*}
n! &= \begin{cases} 
  1, & \text{if } n = 0 \\
  n \times (n - 1) \times \ldots \times 2 \times 1, & \text{if } n > 0 
\end{cases}
\end{align*}
\]

- How to express recursively?
  - Look for a way express the problem as:
    - A small unit of work done that can be done now, and
    - Remaining amount work is of the same exact nature
  - \(7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1\) could be rewritten recursively as \(7! = 7 \times 6!\)
Factorial (Cont.)

- **The recursive method:**

```java
public static long factorial(long n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n-1);
}
```

- You often see factorial defined (alternatively) as follows:

\[
\begin{align*}
n! &= \begin{cases} 
1, & \text{if } n = 0 \\
n \times (n - 1)!, & \text{if } n > 0 
\end{cases}
\end{align*}
\]
Do Some Traces!

- Let's see a trace for `factorial(3)`
  - Remember that callers of recursive methods must wait until callee returns
  - Remember that each call to `factorial` produces its own set of:
    - Parameters (when applicable)
    - Local variables (when applicable)

- Never forget to check *edge cases* too! Trace these edge cases:
  - `factorial(1)`
  - `factorial(0)`
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Example: Recursive List Summation

- **Base case:**
  - The sum is trivially 0, if the list is empty.

- **Recursive case:**
  - Remove the number at the head of the list
  - Add it to the sum of the remaining list

```java
public int recursiveSum(List<Integer> list) {
    if (list.size() == 0) { // base case (trivial)!
        return 0;
    }
    return list.remove(0) + recursiveSum(list); // recursive case (be lazy)
}
```

- Potential issue: What's inside list after the call?
Example: Recursive List Summation

- Your Turn!
  - Find a partner or 2, finish writing the method

- Base case?

- Recursive case?

```java
public int recursiveSum(List<Integer> list) {
    // Base case
    if (list.isEmpty()) {
        return 0;
    }
    // Recursive case
    return list.get(0) + recursiveSum(list.subList(1, list.size()));
}
```
Example: Recursive List Summation

```java
public int recursiveSum(List<Integer> list) {
    if (list.size() == 0) { // base case (trivial)
        return 0;
    }
    return list.remove(0) + recursiveSum(list); // recursive case (be lazy)
}
```

### Complexity:

\[ T(n) = ? \]
- Depends... What kind of list?
- If LinkedList: \( O(n) \)
- If ArrayList: \( O(n^2) \)
Problem the current algorithm destroys the list due to remove()
  • Write a recursiveSum algorithm that does not
  • Hint: may require more input parameters

Old code:

```java
public int recursiveSum(List<Integer> list) {
    if (list.size() == 0) { // base case (trivial)!
        return 0;
    }
    return list.remove(0) + recursiveSum(list);
}
```
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Fibonacci Numbers (Recursive)

- **Fibonacci Numbers** are described as:
  - Given: 0 is the 0th Fib number and 1 is the 1st Fib number
    - All other numbers are the sum of the previous two Fib numbers

- Recursively find the nth Fibonacci number

```java
public static long fib(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fib(n-2) + fib(n-1);
    }
}
```

- **Complexity?**

  \[ T(n) = ? \]

  \[ T(n) = O(2^n) \]

  Recursive algorithm has exponential time complexity!!!
Fibonacci Numbers (Iterative)

- Observation: Why compute $\text{fib}(n-1)$ and $\text{fib}(n-2)$ again and again?
  - Seems totally redundant
  - Save (cache) their results in auxiliary storage: called **memoization**
  - Yay, now: $T(n) = O(n)$

```java
public static long fib(int n) {
    int[] fibo = new int[n+1];
    fibo[0] = 0;
    fibo[1] = 1;
    for (int i = 2; i <= n; i++) {
        fibo[i] = fibo[i-2] + fibo[i-1]; // memoize
    }
    return fibo[n];
}
```
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Recursive Palindrome Checker

- A palindrome is a sequence of number or letters that is read the same forwards and backwards
A palindrome is a sequence of number or letters that is read the same forwards and backwards.
A palindrome is a sequence of numbers or letters that is read the same forwards and backwards.
Recursive Palindrome Checker

- A palindrome is a sequence of number or letters that is read the same forwards and backwards

```
4 5 1 1 5 4
```

Done! (true)
Recursive Palindrome Checker

- A non-palindrome

```
4 9 0 1 5 4
```
Recursive Palindrome Checker

- A non-palindrome

```
4 9 0 1 5 4
```

(start)  (end)
Recursive Palindrome Checker

- A non-palindrome

Done! (false)
A recursive palindrome checker:

- The start and end indices point to the "ends"
  - If start > end, return true
  - Otherwise, a list is a palindrome if items at the "ends" match and if the remaining sublist is a palindrome

```java
public static boolean isPalindrome(int[] list, int start, int end) {
    if (start > end) {
        return true;
    }
    return list[start] == list[end] && isPalindrome(list, start + 1, end - 1);
}
```
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
    - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Recursive Binary Search

- **Recursive Case:**
  - Is the middle element what I'm looking for? (Tiny amount of work)
    - If so, we're done (found it!)
    - If not, ask again on right half of the list. (recursive call)

- **Base Case:**
  - There are no more elements in the list to consider
    - Are the left and right pointers crossed?
    - If so, we're done (didn't find it)
Recursive Binary Search

```java
public static int binarySearch(int[] list, int target, int left, int right) {
    if (left > right) {  //base case
        return -1;
    }

    int mid = (left + right) / 2;
    if (target == list[mid]) {
        return mid;
    }
    else if (target > list[mid]) {
        return binarySearch(list, target, mid+1, right);
    }
    else {
        return binarySearch(list, target, left, mid-1);
    }
}
```

- Let `int[] A = {10, 20, 23, 40, 50, 65, 90}`

- Trace for `binarySearch(A, 90, 0, A.length-1);`
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion
Some data structures can also be recursive

- What does it mean?
  - Has another version of itself in class definition

Already seen one: Linked Lists

- Base case:
  - A trivial linked list has no nodes (i.e., is null)
- Recursive case:
  - A general linked list consists of a head node that points to a linked list
Recursive Data Structures

- Already seen one: Linked Lists
  - **Base case:**
    - A trivial linked list has no nodes (i.e., is null)
  - **General case:**
    - A general linked list consists of a head node that points to a linked list
Recursive Data Structures

- Already seen one: Linked Lists
  - Base case:
    - A trivial linked list has no nodes (i.e., is null)
  - General case:
    - A general linked list consists of a head node that points to a linked list

```
head
  next  data
      2

head
  next  data
      33

head
  next  data
      9

head
  next  data
      2

null
```
Recursive Data Structures

- Already seen one: Linked Lists
  - Base case:
    - A trivial linked list has no nodes (i.e., is null)
  - General case:
    - A general linked list consists of a head node that points to a linked list

```
  data
next
2

33
next

9
next

2
next
null

head

remaining sublist
```
Recursive Data Structures

- Already seen one: Linked Lists
  - Base case:
    - A trivial linked list has no nodes (i.e., is null)
  - General case:
    - A general linked list consists of a head node that points to a linked list

```
head
  next
  data
  head
  next
  data
  head
  next
  data
  head
  next
  data
  null
```
Recursive Data Structures

- Already seen one: Linked Lists
  - Base case:
    - A trivial linked list has no nodes (i.e., is null)
  - General case:
    - A general linked list consists of a head node that points to a linked list

```
2 33 9
```

```
head
\---
next
\---
data
head
\---
next
\---
data
head
\---
next
\---
data
null
remaining sublist
```
Recursive Data Structures

- Already seen one: Linked Lists
  - **Base case:**
    - A trivial linked list has no nodes (i.e., is null)
  - **General case:**
    - A general linked list consists of a head node that points to a linked list
Recursive List Operations

- Since Linked Lists are recursive, we can easily define (or redefine) their operations to be recursive too!

- Let's (re)define these recursively *(RecLinkedList code)*
  
  - **String toString()**
    - Returns the string representation of the list
  
  - **void replaceAll(E old_item, E new_item)**
    - Replaces all occurrences of *old_item* with *new_item*
Outline

- Recursion
  - Definition and Setup
  - Tracing
- Examples
  - Factorial
  - List Sum
  - Fibonacci Numbers
  - Checking Palindrome
  - Binary Search Revisited
- Recursive Data Structures
- Conclusion

Sierpinski's Gasket
A recursive method is one that calls itself to solve a problem

- One of the key concepts in CS and math
- Many problems are naturally recursive: factorial, Fibonacci, etc.
  - *Recursion is for programming-simplicity*

Base case is extremely important

- Recursion won't terminate without it!
  - If the base-case is never met, running the code will result in *a stack overflow*

General case

- Does a few trivial steps to reduce the problem size (toward the base case), and makes a recursive call on the smaller subproblem
Administrivia 3/19

- Homework 4 due Monday, 3/26

- New stuff posted:
  - Stack implementation, Stack applications, Recursion Examples (today)
  - Review guide 2 posted
    - Exam II on 4/3

- Solutions posted:
  - Lab 7: Airport Simulator (multi-queue)

- Reading:
  - Chap 5
  - Lecture 6 Recursion posted
Talk

• Richard Weiss