List of Slides

1 **Topic:** Recursion
2 **Section 1:** Introduction
3 What is a recursive definition?
4 Using a recursive definition
5 Could that calculation have gone on forever?
6 Okay, why does it ‘work’ for all positive n?
7 Recursive method
8 Factorial method
9 Recursion versus iteration
10 Coming up
11 **Section 2:** Lecture attendance count
12 Lecture attendance count
13 Lecture attendance count: instructions
14 Would it work?
15 Could we have done it iteratively?
16 **Section 3:** The sum of the agesof my descendents

17 Another human example
18 Sum of ages of descendents
19 Ooops – that’s not quite right!
20 **Section 4:** Factorial
21 Running factorial
22 Factorial code
23 Why does it work?
24 Can factorial be implemented iteratively?
25 **Section 5:** Fibonacci
26 What is Fibonacci?
27 Running Fibonacci
28 Fibonacci code
29 Can Fibonacci be implemented iteratively?
30 **Section 6:** Countdown
31 What is Countdown?
32 Designing a recursive algorithm
33 Running countDown
34 The key to the countDown solution

35 Output from countDown method
36 Countdown code
39 Running countDown, looking at the code
40 Can countDown be implemented iteratively?
41 **Section 7:** Dice combinations
42 Dice combinations
43 The key to the dice combinations solution
44 Dice code
47 Can printDiceCombinations be implemented iteratively?
48 **Section 8:** Vowel movement
49 Vowel movement
50 The key to the vowel movements solution
51 Vowel movement code
55 Can outputVowelMovements be implemented iteratively?
56 **Section 9:** Towers of Hanoi: another puzzle
57 Towers of Hanoi
58 The key to the towers solution
59 Tower code

63 Towers code
68 Running it again
69 Can Towers of Hanoi be implemented iteratively?
70 **Section 10:** Summary
71 Recursion summary

Topic

Recursion

Section 1

Introduction

What is a recursive definition?

- When something is defined in terms of itself, that is a **recursive definition**.
- It is an old mathematical concept.
- For example, a recursive definition of the factorial function.

```
Assuming n is a positive whole number.

fact 1 = 1
fact n = n * fact(n - 1)
^^^^          ^^^^
```

- In order to define `fact` we use `fact`!

Using a recursive definition

- And so:

```
fact 4
= 4 * (fact 3)
= 4 * (3 * (fact 2))
= 4 * (3 * (2 * (fact 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

Could that calculation have gone on forever?

- What if the argument is zero or negative?
- E.g:

```
fact -1
= -1 * (fact -2)
= -1 * (-2 * (fact -3))
= -1 * (-2 * (-3 * (fact -4)))
= -1 * (-2 * (-3 * (-4 * (fact -5))))
...
```

- Ho, hum. It is only defined (i.e., gets an answer) for positive arguments – but we said that upfront!

Okay, why does it 'work' for all positive n?

- `fact` is defined for all positive integer arguments, `n`.
- `n` could be `1` – the result is `1`.
  - `fact 1 = 1`
- This is known as the **base case**.
- All other (positive integer) `n` are bigger than `1`, and so we can subtract `1` from `n` only a finite number of times before reaching `1`.
  - `fact n = n * fact(n - 1)`
- This is known as the **recursive case**.

Recursive method

- A **recursive method** is one which is defined in terms of itself.
- The definition of a method is its body.
- This means the body of a recursive method contains one or more calls to itself.
- To be **well defined** it must:
  - Have a clearly identified range of **argument**s for which it is meant to work.
  - Have at least one **base case**, which does not cause a call to the same method.
  - In all **recursive case**s the arguments passed to a call of the same method, are nearer to a base case, than those given; assuming the given ones are among those the method is meant to work for.

Factorial method

```
// n must be positive.
public int factorial(int n)
{
  int result;
  if (n == 1)                              // <- Base case.
    result = 1;
  else
  {
    int factNMinus1 = factorial(n - 1); // <- Recursive case.
                                         // <- (n - 1) is nearer to 1.
    result = n * factNMinus1;
  } // else
  return result;
} // factorial
```

Recursion versus iteration

- At first glance, **recursion** can look like another form of **iteration** – execution appears to 'jump back' to the start of the method.
  - Be warned: that is a dangerously wrong view of what is actually happening!
- Indeed many **recursive** methods could (and perhaps should) be easily implemented using a loop instead.
- But many cannot easily be implemented iteratively, as we shall see.
- Recursion is a more powerful, more general tool, than iteration.
- Iteration is in fact merely an optimised implementation of *simple* uses of recursion, known as **tail recursion**.

Coming up

- Lecture attendance count.
- Sum of ages of descendents.
- Factorial. Recursion demo:  factorial-6
- Fibonacci. Recursion demo:  fib-8
- Countdown. Recursion demo:  countdown-1-2-3-4-5-1
- Dice combinations. Combining 6 dice 
- Vowel movement. Input word is ‘El*z*b*th’ 
- Towers of Hanoi. Towers 5 blocks big 

Section 2

Lecture attendance count

Lecture attendance count

- An example of a **recursive** procedure, for humans to follow.
- Suppose, for some arbitrary reason, I wished to know how many students are in this lecture theatre.
- I would, of course, assume that all those present are cooperative, and can follow simple instructions.
- I could pass the following instructions to the left-most person in the front row, and wait for the answer.

Lecture attendance count: instructions

```
Stand up.
Count the number of people in your row (including you), and remember it.
If there is a non-empty row behind you then:
  Pass these instructions to the left-most person in that row, and wait.
  Take the result from that person behind you, add it to your row count,
    -- that is your answer.
else
  Your row count is your answer.
end-if
Pass your answer to whomever asked you to follow these instructions.
```

Would it work?

- Would it work? Maybe we should try it?
- Is it **recursive**?
- Key points about this algorithm:
  - Each person following the instructions has his/her own *separate* notion of:
    - Who 'called' him/her.
    - His/her row count.
    - Who he/she calls, if any.
    - The result got back from whomever he/she called.
  - The result returned by any person is always the number of people in his/her row, plus all those in rows behind.
  - So the result returned by the first person is the count from all the rows.

Could we have done it iteratively?

- Could we have used **iteration** instead of **recursion**?
- Yes, of course!
- One person could have walked up the rows, one by one.
- It was only (really) **tail recursion**, and so is easy to implement using iteration.
- On the other hand, how could we *simply* change our instructions to exploit the parallelism in the room, to get the answer more quickly?
- Well anyway, **concurrent programming** is not a topic of CS1092! ;-)

Section 3

The sum of the ages of my descendents

Another human example

- There is an isolated, if a little crowded, island where people live for thousands of years, because they eat so well and look after themselves.
- One day, a very old, very rich woman wants to give a present to all her descendents, to mark her retirement from running her gold mine for the past 2000 years.
- She wants to give to each descendent, a piece of gold for each year of his or her age.
- She does not even know how many descendents she has, let alone their ages. She does, of course, know her own children.
- In order to figure out how many pieces of gold to get out of her safe, she asks her children for the sum of the ages of their descendents. . . .
- (Oh, by the way, there is never any incest on this island.)

Sum of ages of descendents

- She simply asks herself to follow these instructions!
- ```
  result = 0
  for each child you have (if any):
    Ask him/her to follow these instructions, wait for the result.
    result += result from child.
  end-for
  result += your age
  Pass your result to whomever asked you.
  ```
- Would this work? (What if there was some incest, perish the thought?)
- Could this be easily expressed using **iteration**?
- It is not a simple case of **tail recursion**.
- In fact it uses **multiple recursion**.
  ```

Ooops – that's not quite right!

- She has included herself in the sum of all the ages!
- Maybe these instructions would have been better?

```
result = 0
for each child you have (if any):
  Ask him/her to follow these instructions, wait for the result.
  result += result from child.
  result += age of that child.
end-for
Pass your result to whomever asked you.
```

Section 4

Factorial

Running factorial

- Let us start with a simple example of a **recursive method** – factorial.
- Recursion demo:  factorial-6

Factorial code

- Here is the code.

```
public int factorial(int n)
{
  int result;
  if (n == 1)
    result = 1;
  else
  {
    int factNMinus1 = factorial(n - 1);
    result = n * factNMinus1;
  } // else
  return result;
} // factorial
```

- Recursion demo:  factorial-6-code

Why does it work?

- Each call to `factorial()` has its own *separate* notions of:
  - The **parameter** `n`.
  - The **local variable** `result`.
  - The local variable `factNMinus1`.
  - Where it was called from (e.g: which instantiation of the method).
  - Where it is up to.
- So, for example if `n` is `6`, there will be six local variables called `result`, etc..

Can factorial be implemented iteratively?

- Our `factorial()` method uses simple **tail recursion**, so it is obvious how to implement it iteratively.
- No-one would really implement it using **recursion**, would they?
- On the other hand, the **recursive** version is more 'obviously correct' with respect to the mathematical definition of factorial. Especially when expressed as follows (removing the fairy steps).

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

- Write an iterative version of factorial. Compare it with the mathematical definition – is it 'obviously correct'?

Section 5

Fibonacci

What is Fibonacci?

- This is another mathematical function, which can be used to model breeding patterns.

```
Assuming n is a positive integer.

fib 1 = 1
fib 2 = 1
fib n = fib(n - 1) + fib(n - 2)
```

- For example, `n` might be the number of generations, and the result might be the number of rabbits you will have after that many generations!

Running Fibonacci

- Let us see Fibonacci running.
- Recursion demo:  fib-6
- Recursion demo:  fib-8
- Recursion demo:  fib-10

Fibonacci code

- Here is the code.

```
public int fibonacci(int n)
{
  int result;
  if (n == 1 || n == 2)
    result = 1;
  else
  {
    int fibNMinus1 = fibonacci(n - 1);
    int fibNMinus2 = fibonacci(n - 2);
    result = fibNMinus1 + fibNMinus2;
  } // else
  return result;
} // fibonacci
```

- Recursion demo:  fib-6-code

Can Fibonacci be implemented iteratively?

- Our `fibonacci()` method does not use simple **tail recursion**, so it is not obvious how to implement it iteratively.
- However, after some thought you should be able to find a wholely different way of implementing it, which has **linear time complexity**, i.e. the time taken to run is proportional to `n`.
- When this is expressed **recursive**ly, it does only use tail recursion, and so can easily be implemented in a loop.
- Find that solution! Is it ‘obviously correct’ with respect to the mathematical definition of the function?
- Do you want a clue? Recursion demo:  fastfib-6

Section 6

Countdown

What is Countdown?

- Countdown is a TV show!
- One puzzle involves solving numerical sums. Here we will keep it simple.
  - You have five positive whole numbers, separated by question marks.
  - There is a sixth number, which is the target.
  - The question marks can either be `+` or `/` operators.
  - Ignore operator precedence, and there are no brackets – just work left to right.
  - Find an operator for each question mark, so that the five numbers total the target, if possible.
- E.g: Solve the following puzzles.

```
 1 ? 2 ?  3 ? 4 ?  5 == 1
24 ? 4 ? 59 ? 5 ? 87 == 100
```

Designing a recursive algorithm

- The key to designing a **recursive** solution to a problem is to identify the **base case** and **recursive case**.
- The base case corresponds to input values for which the solution is ‘easy’.
- The recursive case involves us finding an instance of the *same* problem, which is ‘smaller’, and such that the solution to the smaller problem helps us solve the given one.

Running countDown

- Let us see countDown running.
- Recursion demo:  countdown-1-2-3-4-5-1
- Recursion demo:  countdown-24-4-59-5-87-100

The key to the countDown solution

- The **base case** is when we have just one number, instead of five numbers, and that number does or does not equal the desired target.
- The **recursive case**s are based on trying to solve a smaller problem, which has one less number than the given one.
  - i.e. to solve

    ```
    countDown(x1, x2, x3, x4, x5, target)
    ```

    we try one or both of the following smaller problems:

    ```
    countDown(0, x1, x2, x3, x4, target - x5)
    countDown(0, x1, x2, x3, x4, target * x5)
    ```

Output from countDown method

- `countDown(1, 2, 3, 4, 5, 1)`

  produces

```
1
 + 2 = 3
 / 3 = 1
 + 4 = 5
 / 5 = 1
```

Countdown code

- Here is the code.

```
// The five numbers are passed in as parameters, x1, x2, x3, x4 and x5.
// The desired result is the sixth parameter.
// For any x, the value zero represents 'no value'.
// This permits us to have less than five numbers on recursive calls
// by having the first 1, 2, 3 or 4 numbers all being zero.
// If x5 is zero, something has gone wrong!
// If x4 is 0, then x5 is the only number.

public boolean countDown(int x1, int x2, int x3, int x4, int x5,
                         int desiredResult)
{
  if (x5 == 0) // No numbers! Should not happen.
    return false;
```

Countdown code

```
else if (x4 == 0) // Only 1 number.
  if (desiredResult == x5) // Success!
  {
    System.out.println(x5);
    return true;
  } // if
  else // Failure.
    return false;
```

Countdown code

```
    else // Try + between x4 and x5, and if that fails, try / instead.
    {
      if (countDown(0, x1, x2, x3, x4, desiredResult - x5))
      { // Success with + between x4 and x5.
        System.out.println(" + " + x5 + " = " + desiredResult);
        return true;
      } // if
      else if (countDown(0, x1, x2, x3, x4, desiredResult * x5))
      { // Success with / between x4 and x5.
        System.out.println(" / " + x5 + " = " + desiredResult);
        return true;
      } // if
      else // Failure
        return false;
    } // else
  } // countDown
```

Running countDown, looking at the code

- Now let us watch the code running.
- Recursion demo:  countdown-24-4-59-5-87-100-code
- Recursion demo:  countdown-1-2-3-4-5-1-code

Can countDown be implemented iteratively?

- Our `countDown()` method does not use simple **tail recursion**, so it is not obvious how to implement it iteratively.
- Have a go at finding an iterative solution!

Section 7

Dice combinations

Dice combinations

- Recall the example of generating all combinations of three dice.
- It used three nested loops, each iterating from 1 to 6.
- What if we wanted four dice? Ten?
- Any number?

The key to the dice combinations solution

- We work through all the dice, from first to last.
- The **base case** is when we have reached the end of the list of dice.
- The **recursive case**s are based on printing out the combinations of all the dice to the right of the point we are at.
- So, in order to print out all the combinations, we make the first die go through all its possible numbers, and for *each* value, **recursive**ly find all the combinations of the dice to the right of it.

Dice code

- Here is the code.

```
public class Dice
{
  public static void main(String [] args)
  {
    int noOfDice = Integer.parseInt(args[0]);
    diceValues = new int[noOfDice];
    printDiceCombinations(0);
  } // main
```

Dice code

```
private static int[] diceValues;

private static void printDiceCombinations(int currentDieNumber)
{
  if (currentDieNumber == diceValues.length)
  {
    int sumOfDice = 0;
    for (int dieNumber = 0; dieNumber < diceValues.length; dieNumber++)
      sumOfDice += diceValues[dieNumber];
    System.out.print(sumOfDice + " from");
    for (int dieNumber = 0; dieNumber < diceValues.length; dieNumber++)
      System.out.print(" " + diceValues[dieNumber]);
    System.out.println();
  } // if
```

Dice code

```
    else
      for (diceValues[currentDieNumber] = 1;
           diceValues[currentDieNumber] <= 6;
           diceValues[currentDieNumber]++)
        printDiceCombinations(currentDieNumber + 1);
  } // printDiceCombinations
} // class Dice
```

- Trying it. Combining 2 dice 
- Trying it. Combining 6 dice 

Can printDiceCombinations be implemented iteratively?

- Our `printDiceCombinations()` method does not use simple **tail recursion**, so it is not obvious how to implement it iteratively.
- Have a go at finding an **iterative** solution – not just for a fixed number of dice, like the nested loops approach. You can do it, if you approach the problem in a wholely different way. (Hint: base N counting, where N is the number of dice?)
- Is the iterative solution (significantly) more efficient? Is it shorter or longer code? Is it easier or harder to see that it is correct?

Section 8

Vowel movement

Vowel movement

- We have an input string, a word, but with some/all of the vowels replaced by asterisks.
- Output is all possible ‘words’ where each asterisk is replaced by every vowel in turn.
- So, if we have two asterisks in the input, there are 25 ‘words’ in the output.
- Try running it. Input word is ‘J*hn’ 
- Try running it. Input word is ‘El*z*b*th’ 

The key to the vowel movements solution

- It is actually very similar to dice combinations.
- We work through all the characters, from first to last.
- The **base case** is when we have reached the end of the word – we print it out.
- The **recursive case**s are based on printing out the combinations of all the characters to the right of the point we are at, each prepended with what we have so far in the word.
- So, in order to print out all the ‘words’, if the character at the current position is `'*'` we set it in turn to every vowel, and for *each* vowel, **recursive**ly find all the ‘words’ to the right of it.

Vowel movement code

```
public class VowelMovements
{
  public static void main(String [] args)
  {
    inputStringBuffer = new StringBuffer(args[0]);
    outputVowelMovements(0);
  } // main
```

Find out about `StringBuffer` – it is like `String`, but we can change the contents of an **instance** of it.

Vowel movement code

```
private static void outputVowelMovements(int scanPosition)
{
  // scanPosition is where we are up to in our scan from
  // left to right. If we have reached the end,
  // we can print the string and return.

  if (scanPosition >= inputStringBuffer.length())
    System.out.println(inputStringBuffer);

  // If we have not found '*' then move on to the next.
  else if (inputStringBuffer.charAt(scanPosition) != '*')
    outputVowelMovements(scanPosition + 1);
```

Vowel movement code

```
// Otherwise change '*' to 'a', 'e', 'i', 'o', 'u'
// and for each move on.
else
{
  inputStringBuffer.setCharAt(scanPosition, 'a');
  outputVowelMovements(scanPosition + 1);
  inputStringBuffer.setCharAt(scanPosition, 'e');
  outputVowelMovements(scanPosition + 1);
  inputStringBuffer.setCharAt(scanPosition, 'i');
  outputVowelMovements(scanPosition + 1);
  inputStringBuffer.setCharAt(scanPosition, 'o');
  outputVowelMovements(scanPosition + 1);
  inputStringBuffer.setCharAt(scanPosition, 'u');
  outputVowelMovements(scanPosition + 1);
```

Vowel movement code

```
      // Put the asterisk back to restore the value,
      // and also needed for later recursions past this point.
      inputStringBuffer.setCharAt(scanPosition, '*');
    } // if
  } // outputVowelMovements

} // class VowelMovements
```

Can outputVowelMovements be implemented iteratively?

- Our `outputVowelMovements()` method does not use simple **tail recursion**, so it is not obvious how to implement it iteratively.
- Have a go at finding an iterative solution! You can do it, if you approach the problem in a wholely different way – similar to what you did for the dice combinations.
- Is the iterative solution (significantly) more efficient? Is it shorter or longer code? Is it easier or harder to see that it is correct?

Section 9

Towers of Hanoi: another puzzle

Towers of Hanoi

- There are three tall pegs. The leftmost peg holds a tower made from rings of wood, with the largest ring at the bottom, a slightly smaller one on top of that, another even smaller one above that, and so on. The rings have a circular hole in them, just a bit bigger than the pegs. Initially the left peg pokes through all the rings of the tower.
- You have to move the whole tower from the left peg to the right peg, ring by ring, using the middle peg as a temporary storage space.
- You can only move one ring at a time, and you can *never* place a big ring on top of a smaller one!
- Towers 3 blocks big 
- Towers 5 blocks big 
- Towers 7 blocks big 
- Towers 9 blocks big 

The key to the towers solution

- The **base case** is when we have a tower of size zero to move – the job is done!
- For the **recursive case**, we identify a smaller problem that can be done **recursive**ly, to help with the whole tower. That smaller problem is the whole tower except its bottom ring.
- That is, in order to move a tower of N rings from the left peg to the right one, we first move the tower made from the top $N-1$ rings to the *middle* peg. Then we move the bottom ring to the right. Then we move the tower from the middle peg to the right peg, but this time using the left peg as a temporary space.

Tower code

First the basic stuff: a **class** to represent a tower.

```
public class Tower
{
  private final int maximumTowerSize;
  private final int[] blocks;
  private int towerSize;

  public Tower(int requiredSize)
  {
    maximumTowerSize = requiredSize;
    blocks = new int[maximumTowerSize];
    towerSize = 0;
  } // Tower
```

Tower code

```
public void build()
{
  for (int block = maximumTowerSize; block >= 1; block--)
    addOne(block);
} // build

public int getSize()
{
  return towerSize;
} // getSize
```

Tower code

```
public int removeOne()
{
  towerSize--;
  int result = blocks[towerSize];
  blocks[towerSize] = 0; // To assist toString().
  return result;
} // removeOne

public void addOne(int block)
{
  blocks[towerSize] = block;
  towerSize++;
} // addOne
```

Tower code

```
  public String toString()
  {
    ... // Code to represent a tower, as a bit of ‘ascii art’.
    ... // 21 lines of code.
  } // toString

} // class Tower
```

Towers code

Then the main program.

```
public class Towers
{
  private static Tower left, middle, right;
  private static int moveCount = 0;
  private static int recursionLevel = 0;

  private static void showState()
  {
    ... // Code to show the state of the towers
    ... // and the recursion level.
    ... // and have a delay, shorter as recursion level increases.
    ... // 14 lines of code.
  } // showState
```

Towers code

```
private static void moveOne(Tower from, Tower to)
{
  int inHand = from.removeOne();
  to.addOne(inHand);
  moveCount++;
  showState();
} // moveOne

private static void move(Tower from, Tower to)
{
  Tower spare = left;
  if (from == spare || to == spare) spare = middle;
  if (from == spare || to == spare) spare = right;
  move(from.getSize(), from, to, spare);
} // move
```

Towers code

```
public static void main(String [] args)
{
  int towerSize = Integer.parseInt(args[0]);
  left   = new Tower(towerSize);
  middle = new Tower(towerSize);
  right  = new Tower(towerSize);

  left.build();
  showState();
  move(left, right);
} // main
```

Towers code

And finally, the **recursive** bit...

Towers code

```
  private static void move(int noOfBlocksToMove,
                           Tower from, Tower to, Tower spare)
  {
    recursionLevel++; showState();
    if (noOfBlocksToMove > 0)
    {
      move(noOfBlocksToMove - 1, from, spare, to);
      moveOne(from, to);
      move(noOfBlocksToMove - 1, spare, to, from);
    } // if
    recursionLevel--; showState();
  } // move

} // class Towers
```

Running it again

- Towers 3 blocks big 
- Towers 5 blocks big 
- Towers 7 blocks big 
- Towers 9 blocks big 

Can Towers of Hanoi be implemented iteratively?

- Our `move()` method does not use simple **tail recursion**, so it is not obvious how to implement it **iterative**ly.
- But of course it *can* be – even if you have to use additional data structures.
- Have a go at finding an iterative solution – if you dare!

Section 10

Summary

Recursion summary

- Recursion is a powerful tool.
- Some people shy away from it because it can seem a bit tricky.
- You must strive to get comfortable with it.
- Then you will have the ability to choose whether to use **recursion** for the right reasons.
- Factorial is really best done using **iteration**. Fibonacci is best done with the more efficient algorithm which is easily implemented using iteration.
- However, there are many cases where recursion is the best tool for the job, such as the other program examples here.