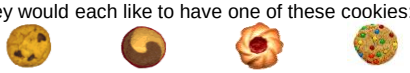
Lecture 14

Recursion

1

Huey, Hayleigh, Patti & Leo Like Cookies (1/2)

- They would each like to have one of these cookies:

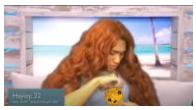


- How many ways can they distribute the cookies amongst themselves?
  - the first character who picks has four choices
  - three choices are left for the second character
  - two choices left for the third character
  - the last character has to take what remains (poor Leo!)

2

Huey, Hayleigh, Patti & Leo Like Cookies (2/2)

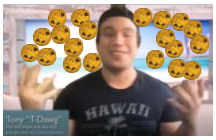
- Thus we have 24 different ways the characters can choose cookies (4! = 4 x 3 x 2 x 1 = 24)
- What if we wanted to solve this problem for all of the contestants on Love Island?



3

Factorial Function

- Model this problem mathematically:
  factorial (n!) calculates the total number of unique *permutations*, or the number of different ways to arrange/order *n* items



- Small examples:
  1! = 1
  2! = 2*1 = 2
  3! = 3*2*1 = 6
  4! = 4*3*2*1 = 24
  5! = 5*4*3*2*1 = 120
- *Iterative* definition: n! = n * (n-1) * (n-2) * … * 1
- *Recursive* definition: n! = n * (n-1)! for n >= 1 and 0! = 1

4

Recursion (1/2)

- Models problems that are self-similar
  - decompose a whole task into smaller, similar sub-tasks
  - each subtask can be solved by applying the same technique
- Whole task solved by combining solutions to sub-tasks
  - special form of divide and conquer at every level

5

Recursion (2/2)

- Task is defined in terms of itself
  - in Java, recursion is modeled by method that calls itself, but each time with simpler case of the problem, hence the recursion will "bottom out" with a **base case** eventually
  - **base case** is a case simple enough to be solved directly, without recursion; otherwise infinite recursion and `StackOverflowError`
  - what is the base case of the factorial problem?
  - Java will bookkeep each invocation of the same method just as it does for nested methods that differ, so there is no confusion
  - usually you combine the results from the separate invocations

6

Factorial Function Recursively (1/2)

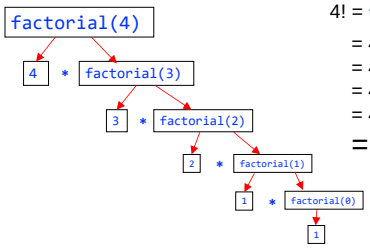
- Recursive factorial algorithm
  - the factorial function is not defined for negative numbers
    - the first conditional checks for this precondition
    - it is good practice to document and test preconditions (see code example)
  - number of times method is called is the depth of recursion (1 for 0!)
    - what is depth of (4!)?

```
public class RecursiveMath{
    //instance variables, other code elided
    public int factorial (int num) {
        if (num < 0){
            System.out.println("Input must be >= 0");
            return -1; // return -1 for invalid input
        }

        int result = 0;
        if (num == 0){ // base case: 0! = 1
            result = 1;
        }
        else{  //general case
            result = num * this.factorial(num - 1);
        }
        return result;
    }
}
```


7/55

7

Factorial Function Recursively (2/2)



4! = factorial(4)

= 4 * 3!
= 4 * 3 * 2!
= 4 * 3 * 2 * 1!
= 4 * 3 * 2 * 1 * 0!
= 24

8/55

8

Lecture Question

Given the following non-practical code:

```
public class RecursiveMath {

    public int recursiveAddition(int n) {
        if (n<=1) {
            return 1;
        } else {
            return this.recursiveAddition(n-1);
        }
    }

}
```

What is the output of `this.recursiveAddition(4)`?

A. 1
B. 9
C. 10
D. `StackOverflowError`

9/55

9

Lecture Question

Given the following code:

```
public class RecursiveMath {

    public int funkyFactorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * this.funkyFactorial(n-2);
        }
    }

}
```

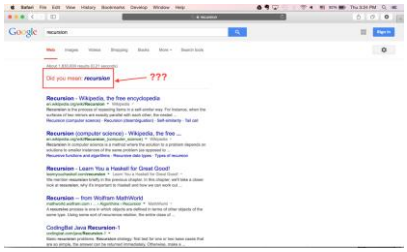
What is the output of `this.funkyFactorial(5)`?

A. 1
B. 5
C. 15
D. `StackOverflowError`

10/55

10

If you want to know more about recursion...



11/55

11

Turtles in Recursion – from Wikipedia

The following anecdote is told of William James. After a lecture on cosmology and the structure of the solar system, James was accosted by a little old lady.
"Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.
"And what is that, madam?" inquired James politely.
"That we live on a crust of earth which is on the back of a giant turtle."
Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.
"If your theory is correct, madam," he asked, "what does this turtle stand on?"
"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, "but I have an answer to it. And it's this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him."
"But what does this second turtle stand on?" persisted James patiently.
To this, the little old lady crowed triumphantly,
"It's no use, Mr. James — it's turtles all the way down."
— J. R. Ross, Constraints on Variables in Syntax 1967

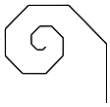

William James (January 11, 1842 – August 26, 1910)
Earliest psychologist

12/55

12

Call Out the Turtles

- Benoit Mandelbrot developed Fractals, a mathematical branch whose principle characteristic is self-similarity at any scale, one of the characteristics of recursions. Fractals are common in nature (botany, lungs, blood vessels, kidneys…), cosmology, antennas, $Z_{n+1} = Z_n^2 + C$ in the complex plane (x, i) where i = sqrt(-1)…
  - check out:
    - https://www.youtube.com/watch?v=2JUAojvFpCohttp://matek.hu/xaos/doku.php
    - https://www.youtube.com/watch?v=aSq2Db3jF_4
    - http://bl.ocks.org/syntagmatic/3736720
    - https://www.youtube.com/watch?v=4LQvjSf6SSw
- Some simpler, non-fractal, but still self-similar shapes composed of smaller, simpler copies of some pattern are simple spirals, trees, and snowflakes
- We can draw these using Turtle graphics
  - iteratively: Start at a particular point, facing in a chosen direction (here up). Draw successively shorter lines, each line at a given angle to the previous one
  - recursively: Start at a particular point, in a given direction. Draw a line of passed-in length, turn the passed-in angle, decrement length and call spiral recursively



13

Designing `Spiral` Class (1/2)

- `Spiral` class defines single draw method
  - turtle functions as pen to draw spiral, so class needs reference to turtle instance
- Constructor's parameters to control its properties:
  - position at which spiral starts is turtle's position
  - length of spiral's starting side
  - angle between successive line segments
  - amount to change length of spiral's side at each step
  - Note: this info is passed to each invocation of recursive method, so next method call depends on previous one

14

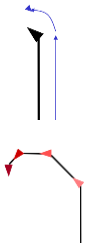
Designing `Spiral` Class (2/2)

```
public class Spiral {
    private Turtle _turtle;
    private double _angle;
    private int _lengthDecrement;
     // passing in parameters to set the properties of the spiral
    public Spiral(Turtle myTurtle, double myAngle, int myLengthDecrement) {
        _turtle = myTurtle;
        _angle = myAngle;
        _lengthDecrement = 1; // default, handles bad parameters
        if (myLengthDecrement > 0){
            _lengthDecrement = myLengthDecrement;
        }
        // draw method defined soon...
    }
}
```

15

Drawing `Spiral`

- First Step: Move turtle forward to draw line and turn some degrees. What's next?
- Draw smaller line and turn! Then another, and another…



16

Sending Recursive Messages (1/2)

- `draw` method uses turtle to trace spiral
- How does `draw` method divide up work?
  - draw first side of spiral
  - then draw smaller spiral (this is where we implement recursion)

```
public void draw(int sideLen){

    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```


17

Sending Recursive Messages (2/2)

- What is the base case?
  - when spiral is too small to see, conditional statement stops method so no more recursive calls are made
  - since side length must approach zero to reach the **base case** of the recursion, the `draw` method is called recursively, passing in a smaller side length each time

```
public void draw(int sideLen){
    // base case: spiral too small to see
    if (sideLen <= 3) {
        return;
    }

    // general case: move sideLen, turn
    // angle and draw smaller spiral
    _turtle.forward(sideLen);
    _turtle.left(_angle);
    this.draw(sideLen - _lengthDecrement);
}
```

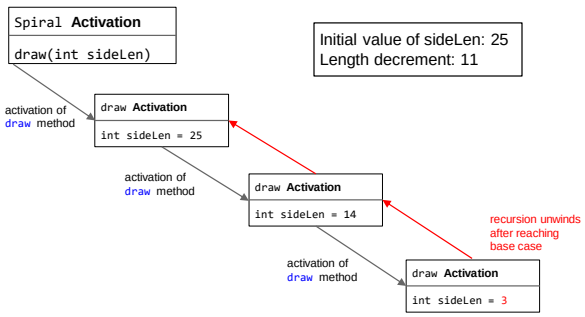

18

Recursive Methods

- We are used to seeing a method call other methods, but now we see a method calling itself
- Method must handle successively smaller versions of original task

Method's Variable(s)

- As with separate methods, each invocation of the method has its own copy of parameters and local variables, and shares access to instance variables
- Parameters let method invocations (i.e., successive recursive calls) "communicate" with, or pass info between, each other
- Java's record of current place in code and current values of parameters and local variables is called the *activation record*
  - with recursion, multiple activations of a method may exist at once
  - at base case, as many exist as depth of recursion
  - each activation of a method is stored on the activation stack (you'll learn about stacks soon)



Lecture Question

Given the following code for the Collatz conjecture:

```
public class RecursiveMath{
    private int _count;
    //constructor elided. _count gets 0, it records
    //number of calls on collatzCounter
    public int collatzCounter(int n) {
        _count += 1;
        if (n == 1) { //base case
            return 1;
        } else {
            if (n % 2 == 0) { //if n is even
                return collatzCounter(n / 2);
            } else {
                return collatzCounter(3 * n + 1);
            }
        }
    }
}
```

What is the value of `_count` after calling `collatzCounter(5)`?

A. 4
B. 5
C. 6
D. `StackOverflowError`

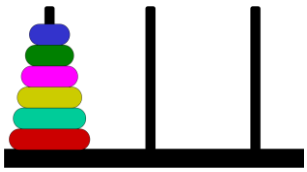
"The **Collatz conjecture** is a conjecture in mathematics named after Lothar Collatz. It concerns a sequence defined as follows: start with any positive integer *n*. Then each term is obtained from the previous term as follows: if the previous term is even, the next term is one half the previous term. Otherwise, the next term is 3 times the previous term plus 1. The conjecture is that no matter what value of *n*, the sequence will always reach 1." (From Wikipedia)

22/55

22

Towers of Hanoi (1/4)

- Game invented by French mathematician Edouard Lucas in 1883
- **Goal**: move tower of n disks, each of a different size (in order, with smallest at top), from left-most peg to right-most peg
- **Rule 1:** no disk can be placed on top of a smaller disk to win
- **Rule 2:** only one disk can be moved at a time

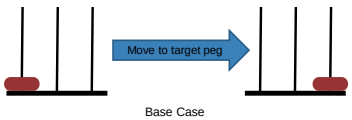


Note: Towers of Hanoi links!
Play it here: https://www.mathsisfun.com/games/towerofhanoi.html
Watch a solution here: https://www.youtube.com/watch?v=l4w1b9mmeFE

23/55

23

One Disk Solution

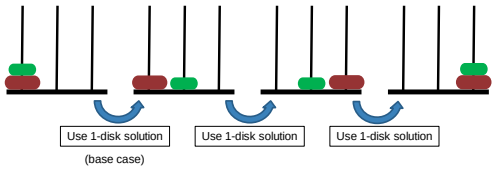


Base Case

24/55

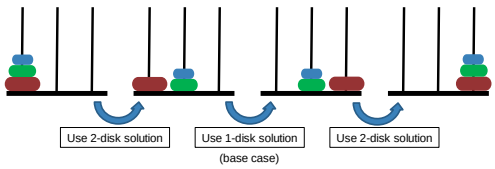
24

Two Disk Solution



25

Three Disk Solution



26

Pseudocode for Towers of Hanoi (1/2)

- Try solving for 5 non-recursively…
- One disk:
  - move disk to final pole
- Two disks:
  - use one disk solution to move top disk to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use one disk solution to move top disk to final pole
- Three disks:
  - use two disk solution to move top disks to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use two disk solution to move top disks to final pole

27

Pseudocode for Towers of Hanoi (2/2)

- In general (for *n* disks)
  - use *n-1* disk solution to move top disks to intermediate pole
  - use one disk solution to move bottom disk to final pole
  - use *n-1* disk solution to move top disks to final pole
- Note: can have multiple recursive calls in a method

28

Lower level pseudocode

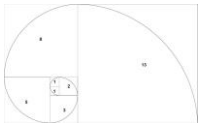
```
//n is number of disks, src is starting pole,
//dst is finishing pole
public void hanoi(int n, Pole src, Pole dst, Pole other){
    if (n==1) {
        this.move(src, dst);
    }
    else {
        this.hanoi(n-1, src, other, dst);
        this.move(src, dst);
        this.hanoi(n-1, other, dst, src);
    }
}

public void move(Pole src, Pole dst){
    //take the top disk on the pole src and make
    //it the top disk on the pole dst
}
```

- That's it! `otherPole` and `move` are fairly simple methods, so this is not much code.
- But try hand simulating this when n is greater than 4. Whoo boy, it is tedious (but not hard!)
- Iterative solution far more complex, and much harder to understand

29

Fibonacci Sequence (1/2)

- 1, 1, 2, 3, 5, 8, 13, 21…
- Each number is calculated by adding the two previous numbers
  - $F_n = F_{n-1} + F_{n-2}$



See a fun application of Fibonacci sequence here

30

Fibonacci Sequence (2/2)

- What is the base case?
  - there are two: n=0 and n=1
- Otherwise, add two previous values of sequence together
  - this is also two recursive calls!

```
// returns nth value of Fibonacci sequence
public int fib(int n){
    if (n < 0) {
        System.out.println("input must be >= 0");
        return -1;
    }
    // base cases: n is 0 or 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // general case: add previous two values
    // using two recursive calls
    return fib(n-1) + fib(n-2);
}
```


31

Lecture Question

Given the following code:

```
public int fib(int n){
    //error check
    if (n < 0) {
        return -1;
    }
    //base case
    if (n == 0 || n == 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

What number would be returned if you **excluded** `n == 1` from the base case and called `fib(2)`?

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

32

Loops vs. Recursion (1/2)

- `Spiral` uses simple form of recursion
  - each sub-task only calls on one other sub-task
  - this form can be used for the same computational tasks as iteration
  - loops (iteration) and simple recursion are computationally equivalent in the sense of producing the same result, if suitably coded (not necessarily the same performance, though -- looping is more efficient)

33

Loops vs. Recursion (2/2)

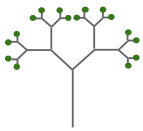
- Iteration is often more efficient in Java because recursion takes more method calls (each activation record takes up some of the computer's memory)
- Recursion is more concise and more elegant for tasks that are "naturally" self-similar (Towers of Hanoi is very difficult to solve iteratively!)

```
public void drawIteratively(int sideLen){

    while(sideLen > 3){
        _turtle.forward(sideLen);
        _turtle.left(_angle);
        sideLen -= _lengthDecrement;

    }
}
```


Indirect Recursion

- Two or more methods act recursively instead of just one
- For example, `methodA` calls `methodB` which calls `methodA` again
- Methods may be implemented in same or different classes
- Can be implemented with more than two methods too

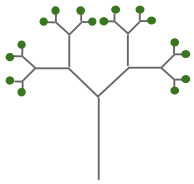
Recursive Binary Tree (1/2)



- The tree is composed of a trunk that splits into two smaller branches that sprout in opposite directions at the same angle
- Each branch then splits as the trunk did until sub-branch is deemed too small to be seen. Then it is drawn as a leaf
- The user can specify the length of a tree's main trunk, the angle at which branches sprout, and the amount by which to decrement each branch

Recursive Binary Tree (2/2)

- Compare each left branch to its corresponding right branch
  - right branch is simply rotated copy
- Branches are themselves smaller trees!
  - branches are themselves smaller trees!
    - branches are themselves smaller trees!
      - ...
- Our tree is self-similar and can be programmed recursively!
  - base case is leaf



Designing the `Tree` Class

- `Tree` has properties that user can set:
  - start position (`myTurtle`'s built in position)
  - angle between branches (`myBranchAngle`)
  - amount to change branch length (`myTrunkDecrement`)
- `Tree` class will define a single `draw` method
  - like `Spiral`, also uses a `Turtle` to draw

```
public class Tree{
private Turtle _turtle;
private double _branchAngle;
private int _trunkDecrement;
    public Tree(Turtle myTurtle, double myBranchAngle,
      int myTrunkDecrement){

        _turtle = myTurtle;

        if(myTrunkDecrement > 0){
            _trunkDecrement = myTrunkDecrement;
        } else {
            _trunkDecrement = 1;
        }

        if(myBranchAngle > 0){
            _branchAngle = myBranchAngle;
        } else {
            _branchAngle = 45;
        }
    }
      // draw method coming up…
}
```


`Tree`'s `draw` Method

- **Base case**: if branch size too small, add a leaf
- **General case**:
  - move `_turtle` forward
  - orient `_turtle` left
  - recursively draw left branch
  - orient `_turtle` right
  - recursively draw right branch
  - reset `_turtle` to starting orientation
  - back up to prepare for next branch

```
private void draw(int trunkLen){
    if (trunkLen <= 0) {
        this.addLeaf();
    } else {
        _turtle.forward(trunkLen);
        _turtle.left(_branchAngle);
        this.draw(trunkLen - _trunkDecrement);
        _turtle.right(2 * _branchAngle);
        this.draw(trunkLen - _trunkDecrement);
        _turtle.left(_branchAngle);
        _turtle.back(trunkLen);
    }
}
```

Overall Program View

```
/* Class that creates a Tree and utilizes its recursive methods in order to draw it.
*/
public class BuildTree {
    Tree _myTree;

    public BuildTree() {
        Turtle turtle = new Turtle()
        double branchAngle = 30;
        int trunkDecrement = 1;
        int trunkLen = 6; //Remember that draw() in Tree class took in a trunkLen
        _myTree = new Tree(turtle, branchAngle, trunkDecrement);
        this.createTree(trunkLen);
     }
    //Method that is going to call draw recursively to draw our tree!
    public void createTree(int currTrunkLen){
        _myTree.draw(currTrunkLen);
    }
}
```


Lecture Question

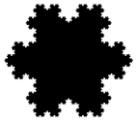
Given the following code:

```
private void draw(int trunkLen) {

   if (trunkLen <= 0) {
      this.addLeaf(); // creates a leaf
                      // at the current location
   } else {
      _turtle.forward(trunkLen);
      _turtle.left(_branchAngle);
      this.draw(trunkLen - _trunkDecrement);
      _turtle.right(2*_branchAngle);
      this.draw(trunkLen - _trunkDecrement);
      _turtle.left(_branchAngle);
      _turtle.back(trunkLen);
   }
}
```

What would happen if you got rid of the first call `this.draw(trunkLen - _trunkDecrement)`?

A. We will only draw the right half of the tree
B. We will draw a spiral that terminates in a leaf
C. Stack Overflow!
D. None of the above

Recursive Snowflake



- Invented by Swedish mathematician, Helge von Koch, in 1904; also known as *Koch Island*
- Snowflake is created by taking an equilateral triangle and partitioning each side into three equal parts. Each side's middle part is then replaced by another equilateral triangle (with no base) whose sides are one third as long as the original.
  - this process is repeated for each remaining line segment
  - the user can specify the length of the initial equilateral triangle's side
  - "mathematical monster": infinite length with a bounded area

43

Snowflake's `draw` Method

- Can draw equilateral triangle iteratively
- `drawSnowFlake` draws the snowflake by drawing smaller, rotated triangles on each side of the triangle (compare to iterative `drawTriangle`)
- `for` loop iterates 3 times
- Each time, calls the `drawSide` helper method (defined in the next slide) and reorients `_turtle` to be ready for the next side

```
public void drawTriangle(int sideLen) {
    for (int i = 0; i < 3; i++) {
        _turtle.forward(sideLen);
        _turtle.right(120.0);
    }
}

public void drawSnowFlake(int sideLen){
    for(int i = 0; i < 3; i++){
        this.drawSide(sideLen);
         _turtle.right(120.0);
    }
}
```


44

Snowflake's `drawSide` method

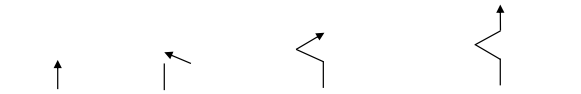
- `drawSide` draws single side of a recursive snowflake by drawing four recursive sides
- **Base case**: simply draw a straight side
- MIN_SIDE is a constant we set indicating the smallest desired side length
- **General case**: draw complete recursive side, then reorient for next recursive side

```
private void drawSide(int sideLen){
    if (sideLen < MIN_SIDE){
        _turtle.forward(sideLen);
    }
    else{
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.right(120.0);
        this.drawSide(Math.round(sideLen / 3));
        _turtle.left(60.0);
        this.drawSide(Math.round(sideLen / 3));
     }
}
```


45

Hand Simulation

```
MIN_SIDE: 20
sideLen:  90
```



1) Call `draw(90)`, which calls `drawSide(90)`, which calls `drawSide(30)`, which calls `drawSide(10)`. Base case reached because 10 < `MIN_SIDE`

2) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn left 60 degrees and then calls `drawSide(10)` again.

3) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn right 120 and then calls `drawSide(10)` for a third time.

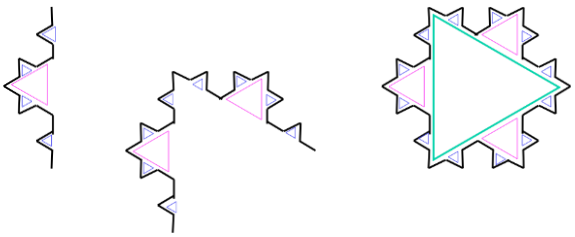
4) `drawSide(10)` returns to `drawSide(30)`, which tells `_turtle` to turn left 60 degrees and then calls `drawSide(10)` for a fourth time.

After this call, `drawSide(30)` returns to `drawSide(90)`, which reorients `_turtle` and calls `drawSide(30)` again.

46

Again: Koch Snowflake Progression

colored triangles added for emphasis only



47

Summary

- Recursion models problems that are self-similar, decomposing a task into smaller, similar sub-tasks.
- Whole task solved by combining solutions to sub-tasks (divide and conquer)
- Since every task related to recursion is defined in terms of itself, method will continue calling itself until it reaches its **base case**, which is simple enough to be solved directly

48

Announcements

- DoodleJump is under way, help slides are out!
  - Early: Monday 3/15
  - On time: Wednesday 3/17
  - Late: Friday 3/19
- SRC Extra credit assignment is out
  - Due Friday 3/12
- Please complete the mid-semester feedback form if you did not do it in section!

IT in the News

ft. Socially Responsible Computing!



Climate Impact of Data Centers

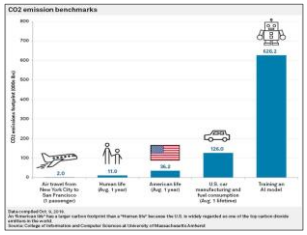
Data centers **consume 3% of the global electricity supply** and **contribute 2% of global greenhouse gas emissions**
- Globally, data centers consume more power than the UK!

Low energy computation = hot topic in computer engineering
- Includes everything from CPU/GPU (hardware) to algorithms (software)

Professor Iris Bahar's research focuses on **energy efficient computing**, from computer architecture to applications


Climate Impact of AI

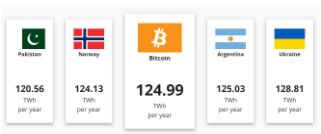
- Researchers at UMass Amherst discovered that **training a single AI model for NLP can emit as much carbon as five cars in their lifetimes**
  - a typical AI research paper requires training ~5,000 individual models (=lifetime emissions of **25,000 cars**)
  - environmental costs grow proportionally to model size, and explode when fine tuning the model's accuracy



52

Climate Impact of Cryptocurrency

- The computer network that maintains the Bitcoin system uses a huge amount of energy
- Most Bitcoin is mined in China by computer servers running on coal-fired power plants
  - In Xinjiang province, Uighur Muslim minorities have been inflicted with human rights abuses mining the coal used to create Bitcoin
- February 2021 Cambridge study shows that annually Bitcoin **consumes 124 Terawatt-hours of electricity** (about 0.5% of the entire world's total) and **produces 37 million tons of carbon dioxide**
  - The energy used by a single Bitcoin transaction could power the average U.S. household for eight days!
  - This will continue to increase in the future, as additional Bitcoin consume exponentially increasing energy



53

What actions are companies taking toward sustainability?

Google has developed AI that teaches itself to minimize the use of energy to cool Google's data centers, reducing its energy requirements by 35%.

Last year, Microsoft reduced its carbon emissions by 6% by purchasing the removal of 1.3 million tons of carbon around the world.

Amazon has 127 solar and wind projects, switched to recyclable packaging in 2019, and will have 10,000 EVs on the road by 2022

- **Who is holding these companies accountable?**
- **Do these small-scale actions make up for large-scale negative environmental impacts?**
  - Ex: Amazon has changed the way we buy and consume products; massive energy + emissions required to produce, transport, and consume third party products

54

How can AI increase sustainability across industries?

- **Agriculture**: AI can transform production by better monitoring and managing environmental conditions and crop yields
- **Energy**: AI can use smart grid systems to manage the demand and supply of renewable energy
- **Transportation**: AI can help reduce traffic congestion, improve the transport of cargo (supply chain logistics), and enable more and more autonomous driving capability
- **Manufacturing**: AI can help reduce waste and energy use in production facilities
- **Facilities management**: AI can help recycle heat within buildings and maximize the efficiency of heating and cooling

55/55

55