


Recursion

Part 3

1




Program Structure

How they work

2

Program Structure

- When writing a program, you must be aware how it works "behinds the scenes"
- In particular, you must understand memory and how it is used.




Spring 2023 Sacramento State - CS&E - CS&E 130 3

3

Program Structure

- There are possible issues that can arise that can negatively impact your programs
- ... and possibly make them unresponsive



Spring 2023 Sacramento State - CS&E - CS&E 130 4

4


Some Terminology

- When you call a function, you can specify pieces of data called *arguments*
- These match the format of the function – which is specified in its *parameters*
- Basically
 - arguments are *passed* to the parameters
 - they match, in order, on a one-to-one basis
 - arguments → parameters

Spring 2023 Sacramento State - CS&E - CS&E 130 5

5

Scope




- Scope* refers how a variable/function is bound (i.e. visible to the rest of your program)
- Data is often stored differently, based on its scope

Spring 2023 Sacramento State - CS&E - CS&E 130 6

6

Global Variables



- You can declare variables outside functions
- These are visible to all functions in the class (or module)
- These are known as *global variables*

Spring 2023 Sacramento State - Oak - CS130 7

7

Global Variables

```

int total;

void printTotal()
{
    System.out.println(total);
}

int main()
{
    total = 1000;
    ...
}


```

Visible to all functions!

Spring 2023 Sacramento State - Oak - CS130 8

8

Global Variables



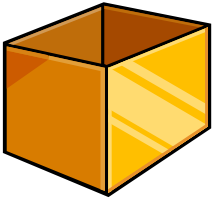
- They can be useful for sharing data between functions
- However, it can be problematic
 - variables can be modified in ways that cause side effects in your program
 - it is better to use local variables and pass them to other functions

Spring 2023 Sacramento State - Oak - CS130 9

9

Local Variables

- When you create functions, each can have *local* variables
- These are only "visible" to the function in which they are declared
- So, other functions cannot access them

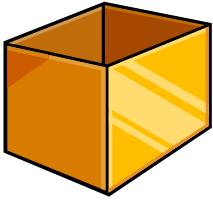


Spring 2023 Sacramento State - Oak - CS130 10

10

Variable Scope

- Different functions can have local variables with the same name
- Why?
 - they can't "see" each other
 - they are different variables, anyway
 - ... so, there is no problem



Spring 2023 Sacramento State - Oak - CS130 11

11

Variable Scopes

```

int hello()
{
    int x;
}

int main()
{
    int x;
}

```

Not the same variable

Spring 2023 Sacramento State - Oak - CS130 12

12

Variable Scopes

```

int hello()
{
    double x;
}

int main()
{
    int x;
}

```

Don't have to be the same type
 (they are different variables)

13

Example: Average Function

```

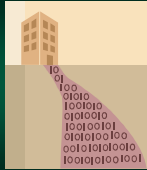
double average(double a, double b)
{
    double avg;

    avg = (a + b) / 2;
    return avg;
}

```

Parameters are also local variables

14



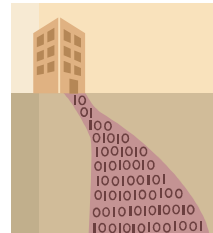
The System Stack & Heap

Making the Functions Function & Data Delightful

15

The System Stack & Heap

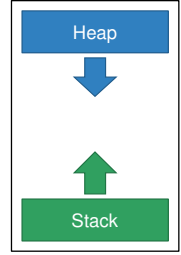
- Computers maintain two types of memory for running programs: *The Stack* and *The Heap*
- Each has a specific purpose, and, in tandem, they make modern programs possible



16

The System Stack & Heap

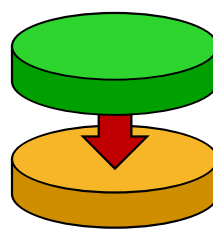
- Each is stored in your computer's main memory
- They grow "towards" each other (and, hopefully, will never meet)



17

The System Stack

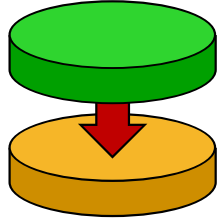
- The System Stack is used to store local variables and allow your program to support functions
- So, anytime you call a function or declare a local variable, a stack is used



18

The System Stack

- Each time a function calls another function an *Activation Record* is placed on the *stack*
- It contains all the information that the instance of a function requires



Spring 2023

Seacramento State - CS&I - CS&I 130

19

19

Contents of the Activation Record

- The Activation Record contains:
 - parameters
 - local variables
 - return address (used by the processor)
- Data in an activation record is temporary to that "instance" of a function
- In other words, data does not persist after the function ends

Spring 2023

Seacramento State - CS&I - CS&I 130

20

20

The Power of Stacks

- Because the stack is a First-In-Last-Out structure, it allows function nesting
- And even a more powerful concept – *recursion*
- Examples
 - web browser "back button"
 - undo sequence in a text editor

Spring 2023

Seacramento State - CS&I - CS&I 130

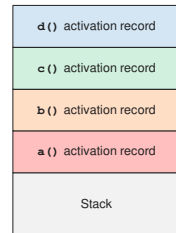
21

21

21

Nesting Activation Records

- For example:
 - `main()` calls `a()`
 - `a()` calls `b()`
 - `b()` calls `c()`
 - `c()` calls `d()`
- Each activation record is pushed onto the stack



Spring 2023

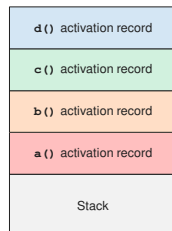
Seacramento State - CS&I - CS&I 130

22

22

Nesting Activation Records

- When a function "returns", its activation record is pop'd and discarded
- The local variables cease to exist
- Only the return value is passed to the caller



Spring 2023

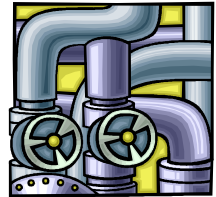
Seacramento State - CS&I - CS&I 130

23

23

The Heap

- Nothing on the system stack persists forever – it is quite temporary
- So, how do we make data last indefinitely? ...or, as long as our program is active



Spring 2023

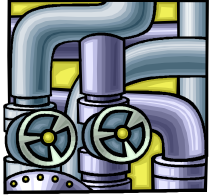
Seacramento State - CS&I - CS&I 130

24

24

The Heap

- *The Heap* is used to store dynamic allocation
- It is allocated *as needed*
- ... not to be confused with the Heap Data Structure (which we will cover later)



Spring 2023 Sacramento State - CS&E - CS&E 130 25


25

The Heap

- Anytime you create objects using *"new"*...
 - the heap is used to allocate storage
 - system performs garbage collection after the memory is no longer needed
- Unlike the stack, data persists regardless of function calls

Spring 2023 Sacramento State - CS&E - CS&E 130 26

26




Reference Types

The objective of Object Oriented Programs

27

Reference Types




- Most languages are based on largely based on building abstract data types called *reference types*
- They are links to nebulous *objects* – whose contents & implementation are unknown

Spring 2023 Sacramento State - CS&E - CS&E 130 28

28

Reference Types



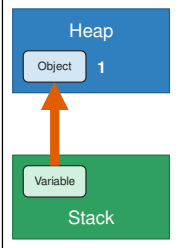
- This is known as *object-oriented programming*
- ... and is the basis of all modern programming languages

Spring 2023 Sacramento State - CS&E - CS&E 130 29

29

Reference Types

- So, local variables exist on the stack
- But... they reference (*contain the address of*) objects stored on the system heap

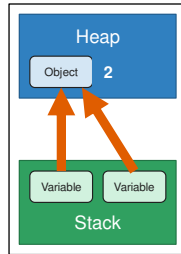


Spring 2023 Sacramento State - CS&E - CS&E 130 30

30

Reference Types

- This allows multiple variables to point to the same object
- This is called *aliasing*
- The system keeps track of how many references each object has



31

Garbage Collection

- Programming languages use *garbage collection* to reclaim unused data from the heap
- Policy is to reclaim the memory used by objects that *can no longer be accessed* (i.e. no references)



32

Garbage Collection

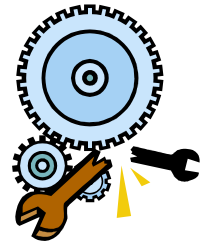
- So, languages maintain a counter on each object
 - if you add a reference, it increments
 - if a reference is removed, it decrements
- When it reaches zero, the object can be removed



33

Loitering

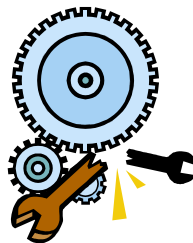
- It is possible to "remove" an item from the ADT, but accidentally keep a reference (link) to it
- The item is effectively an *orphan* - it will be never be accessed again by the ADT



34

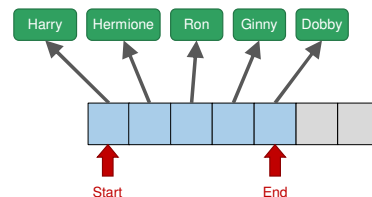
Loitering

- The garbage collector has no way to know unless it's overwritten
- So, under this condition, the object is said to *loiter* – stay in memory with no purpose
- This can negatively affect performance

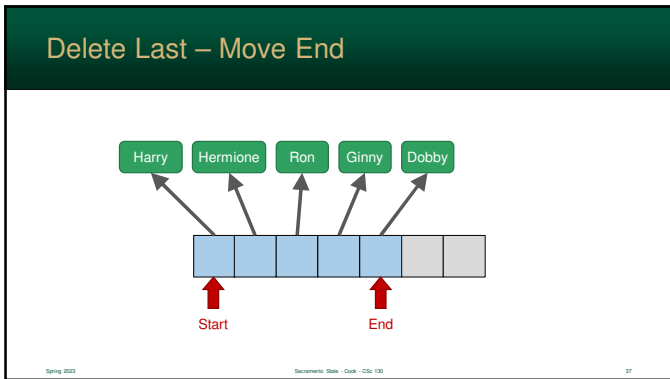


35

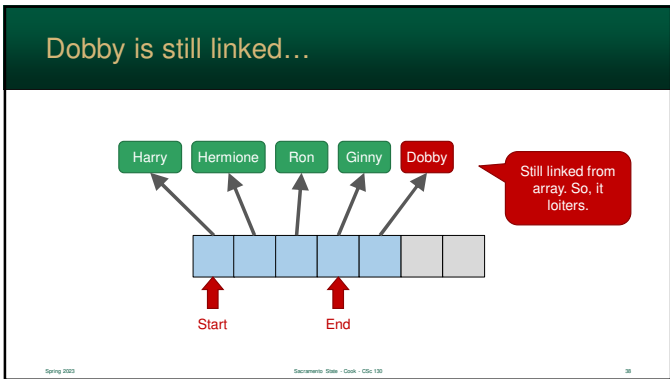
Array Storing a List (partially filled)



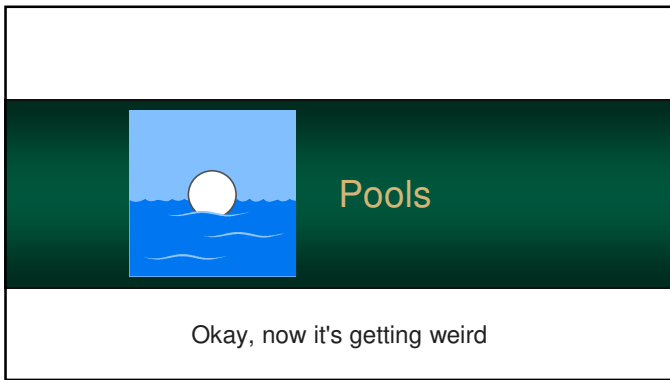
36



37



38



39

Pools

- Creating and destroying objects is expensive on the heap
- So, we want to minimize the constant creation and deletion of new nodes

Spring 2023 Sacramento State - Oak - CS 130 40

40

Why?

- Arrays can be wasteful ...
 - in space – when there are partially
 - in time – created and destroyed frequently
- Linked lists can be wasteful...
 - require memory to be allocated each time a node is created
 - puts a lot of work on the heap

Spring 2023 Sacramento State - Oak - CS 130 41

41

Jump in the Pool

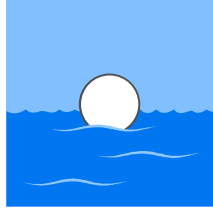
- One solution is to maintain a *pool*
- This is a collection of nodes that are allocated early and are used as, kind of, a recycling bin

Spring 2023 Sacramento State - Oak - CS 130 42

42

Jump in the Pool

- If a node is needed, one is removed from the pool
- If a node is removed, and the array has room, it is placed back in the array (after the data field is set to null, of course)



Spring 2023

Sacramento State - CS&E - CS130

43

43

Even more approaches

- You can also use a "pool" for linked lists
- So, your Linked List class
 - would have a linked list of valid nodes
 - and another list of unused nodes
 - the danger here is that you don't limit the size of the pool – and it grows **forever**
 - so, if you use two linked lists, keep a pool member count

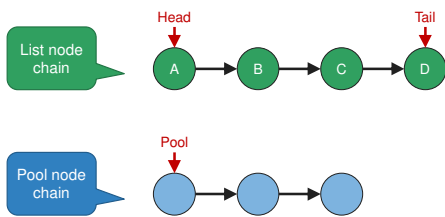
Spring 2023

Sacramento State - CS&E - CS130

44

44

Linked List with a List Pool



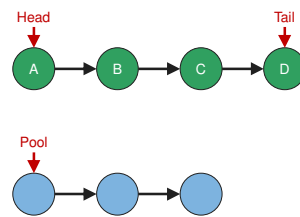
Spring 2023

Sacramento State - CS&E - CS130

45

45

Delete Head: Remove Node (1 of 3)



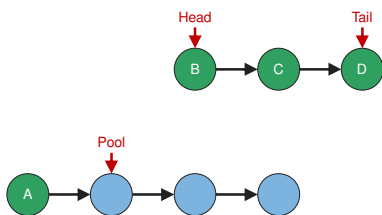
Spring 2023

Sacramento State - CS&E - CS130

46

46

Delete Head: Link (2 of 3)



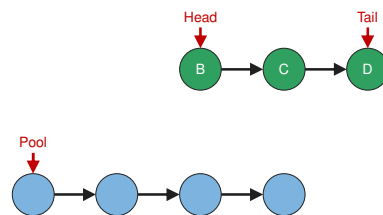
Spring 2023

Sacramento State - CS&E - CS130

47

47

Delete Head: Clear Value (3 of 3)

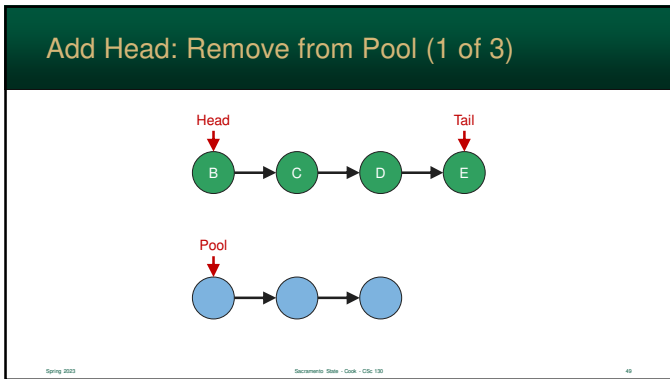


Spring 2023

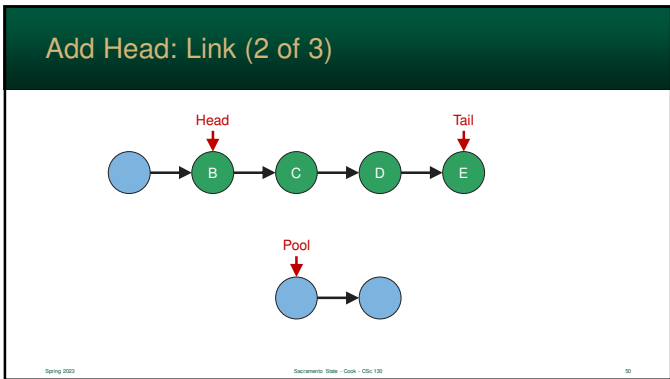
Sacramento State - CS&E - CS130

48

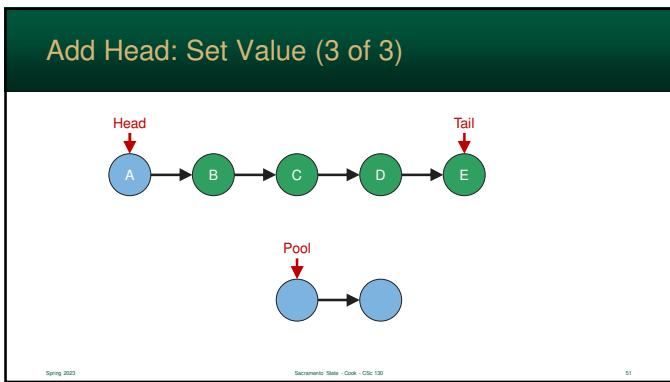
48



49



50



51

Recursion

The best way to learn recursion...
is to, first, learn recursion!

52

Recursion

- *Recursion* occurs when a function directly or indirectly calls *itself*
- This results in a loop
- However, it doesn't use iterative structures such as For or While loops

53

Recursion

- This can greatly simplify programming tasks
- Commonly used to traverse a graph, tree, or run complex calculations
- While powerful, it is costly on computer resources
- ...and can also create pitfalls

54

Breaking a Problem Down

- Recursion allows a problem to be broken down into smaller instances of themselves
- Each call will represent a smaller, simpler, version of the **same** problem
- Eventually, it will reach a "base case" which will **not** require any more recursive calls

Spring 2023

Sacramento State - Oak - CS130

55

55

Where Recursion Shines

- When the program can be broken into smaller pieces, recursion is a great solution
- Examples:
 - graph traversal – searching, etc....
 - state machines
 - sorting
 - many math problems

Spring 2023

Sacramento State - Oak - CS130

56

56

Danger: Never Ending

- If you break down a task into smaller parts... at some point, it should become a single value
- If not, the function will never end and will recurse **forever** – *at least until the computer runs out of resources*



Spring 2023

Sacramento State - Oak - CS130

57

57

Danger: Accidental Recursion

- Accidental recursion is a common mistake by beginner programmers
- Recursion can be done directly or indirectly
 - for example: A calls B, B calls C, C calls A
 - organize your code carefully!



Spring 2023

Sacramento State - Oak - CS130

58

58

Results of These Dangers...

- Runaway recursion
 - function will recurse **forever**
 - eventually all memory is exhausted
- You will see either...
 - "stack overflow" error
 - "heap exhaustion" error



Spring 2023

Sacramento State - Oak - CS130

59

59

To infinity... but not beyond

```
void toInfinity()  
{  
    System.out.println("To infinity!");  
    toInfinity();  
    System.out.println("and beyond!");  
}
```

We never get here!

Spring 2023

Sacramento State - Oak - CS130

60

60

Designing a Recursive Function

- Does the problem lend itself to recursion?
 - can the problem be broken down into smaller instances of itself?
 - is there a iterative version that is better
- Is there a base case?
 - is there a case where recursion will stop?
 - remember: **ALWAYS** have a stopping point!

Spring 2023


Sacramento State - CS&E - CS130

61

61



62



Examples of Recursion

Examples defined as examples defined as examples...

63

Example: Factorials

- Factorials are classic mathematical problem that lends itself easily to recursion
- If you don't remember, a factorial of n is defined as the value of n multiplied by all lesser integers ≥ 1
- For example: $5! \rightarrow 5 \times 4 \times 3 \times 2 \times 1 \rightarrow 120$

Spring 2023

Sacramento State - CS&E - CS130

64

64

Example: Factorials

- It should be easy to observe that $n!$ can be defined as $n \times (n - 1)!$
- So, $n!$ can be computed by multiplying n by the factorial of one less than it
- $4! \rightarrow 4 \times 3! \rightarrow 4 \times 3 \times 2! \rightarrow 4 \times 3 \times 2 \times 1$

Spring 2023

Sacramento State - CS&E - CS130

65

65

Example: Factorials

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

Base case

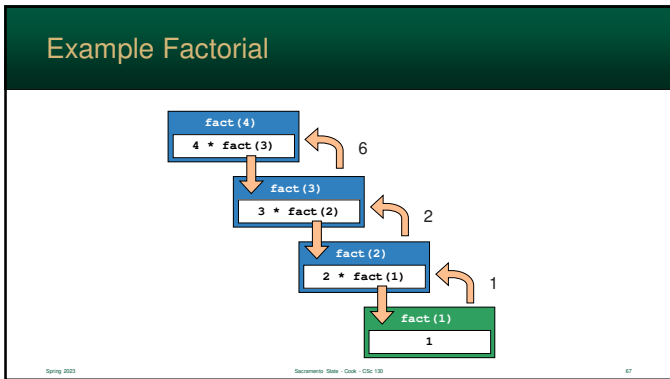
Recursion

Spring 2023

Sacramento State - CS&E - CS130

66

66

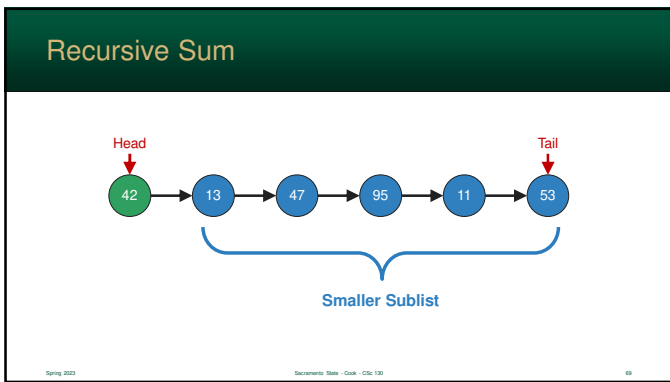


67

Example: Linked Lists

- Linked Lists can also be recursively defined
- Every list can be seen as collection of smaller lists
- So, recursion (while not recommended) is possible

68

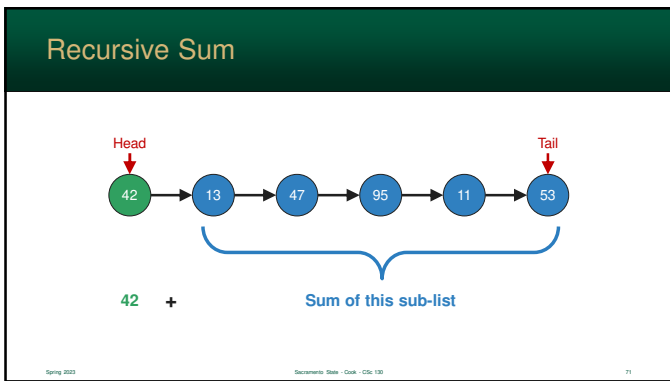


69

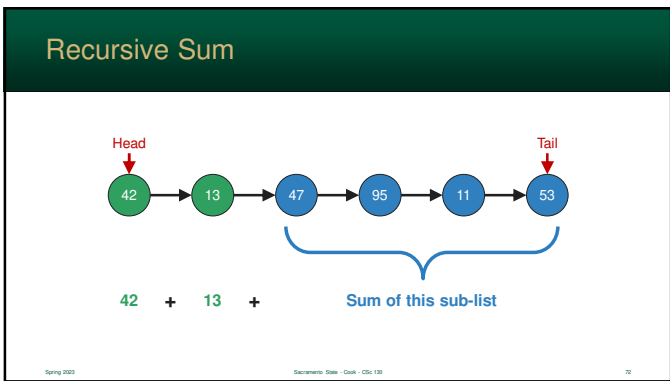
Recursion Example: Sum

- Recursion usually happens in the recursively defined structure itself
- In other words, for linked lists, the *recursion will happen in the node*

70



71



72

Sum Example (in the node class)

```
double sum()
{
    if (this.next == null)
    {
        return this.value;
    }
    else
    {
        return this.value + this.next.sum();
    }
}
```

Base case (end of the list)

Recursion on smaller list

73

Iteration vs. Recursion

- **Any** program that can be expressed using recursion, can be done through iteration
- The recursive solution will often be far simpler – **more "eloquent" to read**
- ... but is never more efficient due to the overhead of calling functions

74

Historical Perspective

Some really cool solutions!

75

Some Well-known Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing



76

Some Well-known Problems

- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination



77


Fibonacci Numbers



- Rabbits tend to reproduce like... well... rabbits
- Mathematician *Fibonacci* analyzed this situation and created a mathematical system to predict this phenomena
- It is used today in finance, simulation, and several computer science algorithms

78

Fibonacci Numbers



- The problem:
 - start with a pair of rabbits
 - at month #2, the rabbits begin to reproduce
 - the female gives birth to a new pair of rabbits: one male and one female
 - babies mature at the same rate and will have more babies
- Fibonacci number sequences predict the total pairs after n months

Spring 2023
Sacramento State - CS&E - CS&E 130
79

79

Fibonacci Numbers

- After two months, the female gives birth creating a new pair... then they get pregnant again!
- This continues forever.....
- Sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```

if n == 1 then Fib(n) = 1
if n == 2 then Fib(n) = 1
if n > 2 then Fib(n) = Fib(n-2) + Fib(n-1)

```

Spring 2023
Sacramento State - CS&E - CS&E 130
80

80

Example 2: Fibonacci Numbers

```

f(3) = f(2) + f(1) = 1 + 1 = 2
f(4) = f(3) + f(2) = 2 + 1 = 3
f(5) = f(4) + f(3) = 3 + 2 = 5
f(6) = f(5) + f(4) = 5 + 3 = 8

```

Spring 2023
Sacramento State - CS&E - CS&E 130
81

81

Example: Fibonacci Numbers

```

int fib(int n)
{
  if (n == 1 || n == 2)
  {
    return 1;
  }
  else
  {
    return fib(n-2) + fib(n-1);
  }
}

```

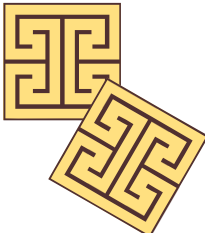
Recursion

Spring 2023
Sacramento State - CS&E - CS&E 130
82

82

Greatest Common Divisor

- A common problem in computer science is finding the greatest common divisor of two integers
- For example: the GCD of 64 and 40 is 8

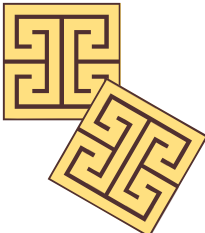


Spring 2023
Sacramento State - CS&E - CS&E 130
83

83

Greatest Common Denominator

- *Euclid* created an ingenious algorithm for finding the greatest common divisor
- This is known example of recursion – first solved using geometry using the metaphor of a tile floor



Spring 2023
Sacramento State - CS&E - CS&E 130
84

84

Euclid's Algorithm

- Euclid's algorithm is recursive
- You reapply the expression below until the second value of $\text{gcd}(n, m)$ is zero.
- In this case, n will be the GCD

$$\text{gcd}(n, m) \rightarrow \text{gcd}(m, n \bmod m)$$

Spring 2023

Sacramento State - Gosh - CS 130

85

85

Euclid's Algorithm Examples

- 60 and 24
 - $\text{gcd}(60, 24) \rightarrow \text{gcd}(24, 12) \rightarrow \text{gcd}(12, 0)$
 - the result is 12
- 84 and 20
 - $\text{gcd}(84, 20) \rightarrow \text{gcd}(20, 4) \rightarrow \text{gcd}(4, 0)$
 - result is 4
- These might seem trivial, but it can find HUGE numbers quite easily

Spring 2023

Sacramento State - Gosh - CS 130

86

86