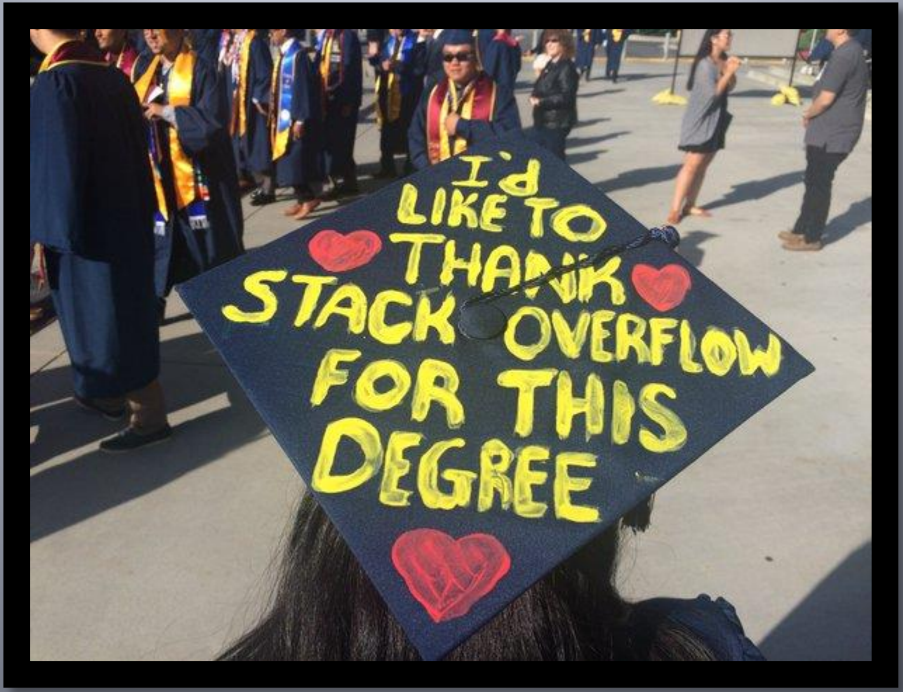
Part 2

Recursion

Objectives

- Drawbacks
- Recursion and induction
- More recursive functions

Drawbacks of Recursion



Inefficent Recursive Functions

- Some recursive algorithms are inherently inefficient
  - e.g. the recursive Fibonacci algorithm which repeats the same calculation again and again
  - Look at the number of times `fib(2)` is called
- Such algorithms should be *implemented* iteratively
  - Even if the solution was *determined* using recursion

Recursion Overhead

- Recursive algorithms have more overhead than similar iterative algorithms
  - Because of the repeated function calls
  - Each function call has to be inserted on the stack
- It is often useful to derive a solution using recursion and implement it iteratively
  - Sometimes this can be quite challenging!

Stack Overflow

- Some recursive functions result in the call stack becoming full
  - Which usually results in an error and the termination of the program
- This happens when function calls are repeatedly pushed onto the stack
  - And not removed from the stack until the process is complete

Recursive Sum

```
int sum(int x){
    if (x == 0 || x == 1){
        return x;
    } else {
        return x + sum(x – 1);
    }
}
```

- This function works
  - But try running it to sum the numbers from 1 to 8,000
  - It will probably result in a stack overflow
    - Since 8,000 function calls have to be pushed onto the stack!

Recursive Factorial

```
int factorial (int x){
    if (x == 0 || x == 1){
        return 1;
    } else {
        return x * factorial(x - 1);
    }
}
```

- This function won't result in a stack overflow
  - Why not?
    - Hint – think how fast factorials increase

Recursive Factorial Trace

- What happens when we run factorial(5)?
  - Each function call returns a value to a calculation in a previous call
- Like this
  - factorial(5)
    - factorial(4)
      - fact(3)
        - factorial(2)
          - factorial(1)
          - 1 returned to factorial(2)
        - 2 returned to factorial(3)
      - 6 returned to factorial(4)
    - 24 returned to factorial(5)
  - 120 returned from factorial(5)
- The final answer is only computed in the final return statement

```
int factorial (int x){
	if (x == 0 || x == 1){
		return 1;
	} else {
		return x * factorial(x – 1);
	}
}
```

A calculation is performed after returning to previous calls

Tail Recursion

```
int factorialTail (int x, int result){
    if (x <= 1){
        return result;
    } else {
        return factorialTail(x-1, result * x);
    }
}
```

- Another recursive factorial function
  - The recursive call is the last statement in the algorithm and
    - The final result of the recursive call is the final result of the function
  - The function has a second parameter that contains the result

Tail Recursion Trace

- Here is the trace of factorialTail(5, 1)
  - factorialTail(5, 1)
    - factorialTail(4, 5)
      - factorialTail(3, 20)
        - factorialTail(2, 60)
          - factorialTail(1, 120)
          - 120 returned to factorialTail(2)
        - 120 returned to factorialTail(3)
      - 120 returned to factorialTail(4)
    - 120 returned to factorialTail(5)
  - 120 returned from factorialTail(5)
- The final answer is returned through each of the recursive calls, and is calculated at the bottom of the recursion tree

```
int factorialTail (int x, int result){
	if (x <= 1){
		return result;
	} else {
		return factorialTail(x-1, result * x);
	}
}
```

The calculation is performed before making the recursive call

Nothing is achieved while returning through the call stack

Tail Recursion and Iteration

- Tail recursive functions can be easily converted into iterative versions
  - This is done automatically by some compilers

```
// Calling Function
int factorial (int x){
    return factorialTail(x, 1);
}
```

```
int factorialTail (int x, int result){
    if (x <= 1){
        return result;
    } else {
        return factorialTail(x-1, result * x);
    }
}
```

```
int factorialIter (int x){
    int result = 1;
    while (x > 1){
        result = result * x;
        x = x-1;
    }
    return result;
}
```

Analyzing Recursive Functions

- It is useful to trace through the sequence of recursive calls
  - This can be done using a *recursion tree*
- Recursion trees can be used to determine the running time of algorithms
  - Annotate the tree to indicate how much work is performed at each level of the tree
  - And then determine how many levels of the tree there are

Recursion and Induction

Mathematical Induction

- Mathematical induction is a method for performing mathematical proofs
- An inductive proof consists of two steps
  - A *base case* that proves the formula hold for some small value (usually 1 or 0)
  - An *inductive step* that proves if the formula holds for some value n, it also holds for n+1
- The inductive step starts with an inductive hypothesis
  - An assumption that the formula holds for n

Recursion and Induction

- Recursion is similar to induction
- Recursion *solves* a problem by
  - Specifying a solution for the base case and
  - Using a recursive case to derive solutions of any size from solutions to smaller problems
- Induction *proves* a property by
  - Proving it is true for a base case and
  - Proving that it is true for some number, *n*, if it is true for all numbers less than *n*

Recursive Factorial

```
int factorial (int x){
    if (x == 0){
        return 1;
    } else {
        return x * factorial(x – 1);
    }
}
```

- Prove, using induction, that the algorithm returns the values
  - *factorial*(0) = 0! = 1
  - *factorial*(n) = n! = $n * (n - 1) * \ldots * 1$ if $n > 0$

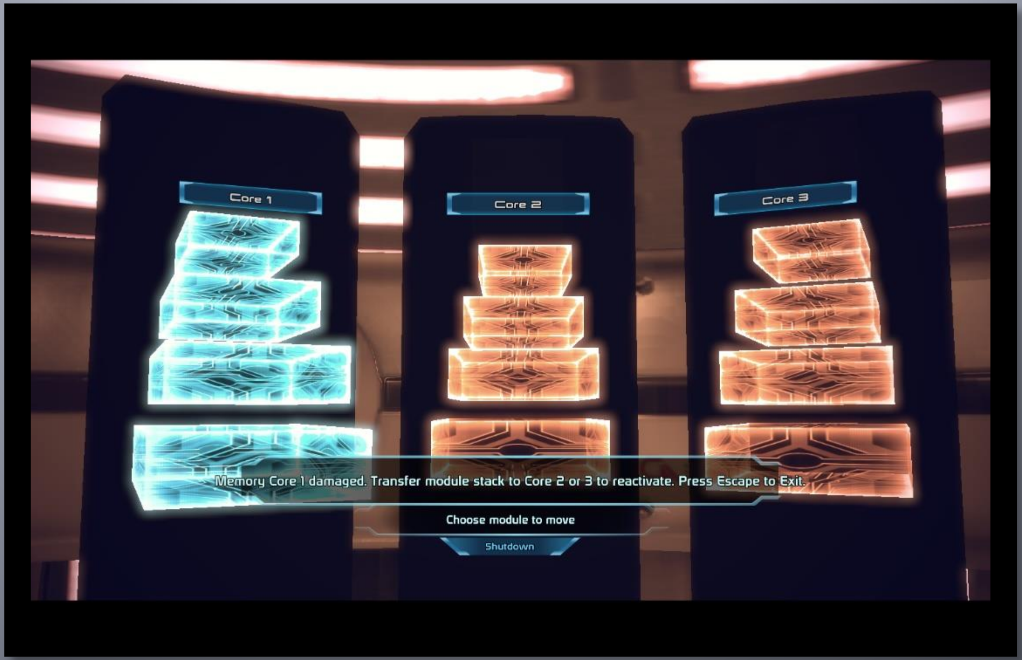
Proof by Induction

- **Base case**: Show that the property is true for $n = 0$, i.e. that *factorial*(0) returns 1
  - This is true by definition as *factorial*(0) is the base case of the algorithm and returns 1
- Establish that the property is true for an arbitrary k implies that it is also true for $k + 1$
- **Inductive hypothesis**: Assume that the property is true for $n = k$, that is assume that
  - $factorial(k) = k * (k - 1) * (k - 2) * \ldots * 2 * 1$

Proof by Induction

- **Inductive conclusion**: Show that the property is true for $n = k + 1$, i.e., that *factorial*($k + 1$) returns
  - $(k + 1) * k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- By definition of the function: *factorial*($k + 1$) returns
  - $(k + 1)$ * *factorial*(k) – the recursive case
- And by the inductive hypothesis: *factorial*(k) returns
  - $k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- Therefore *factorial*($k + 1$) *must* return
  - $(k + 1) * k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- Which completes the inductive proof

More Recursive Functions



More Recursive Algorithms

- Towers of Hanoi
- Eight Queens problem
- Sorting
  - Mergesort
  - Quicksort

Recursive Data Structures

- Linked Lists are recursive data structures
  - They are defined in terms of themselves
- There are recursive solutions to many list methods
  - List traversal can be performed recursively
  - Recursion allows elegant solutions of problems that are hard to implement iteratively
    - Such as printing a list backwards