Invariants

Objectives

- Writing better code
- Loop invariants
- Correctness

Writing Better Code

Writing Good Code

- Not all code is equal
  - Correct and reliable code is one of our goals
  - Is a new car correct or reliable?
- Other characteristics of good code
  - Affordable
  - Well designed
  - Maintainable
  - Extendable

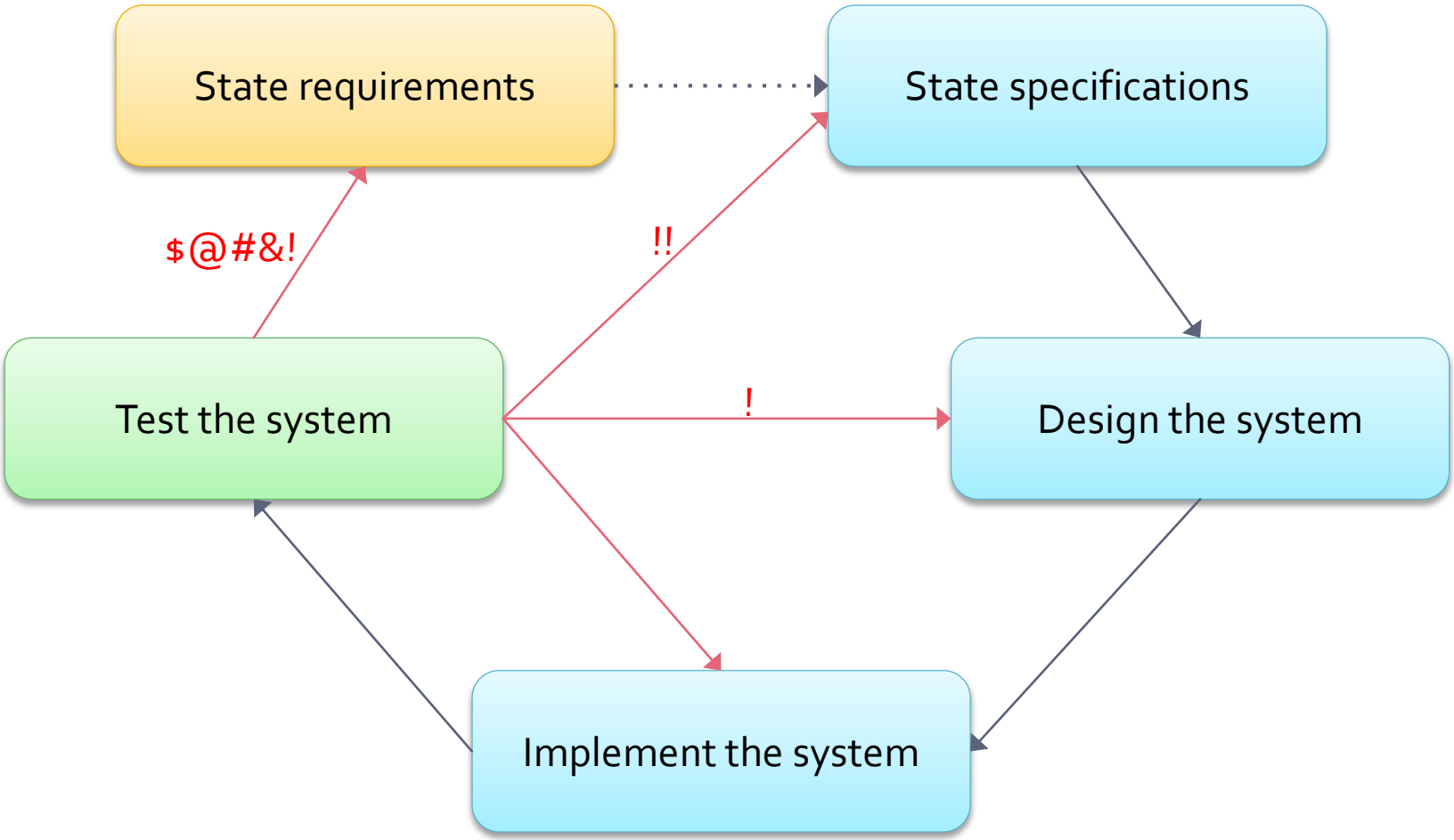
Duality of Code

- Code serves two purposes
- It is the precise expression of an algorithm to the computer
  - Which follows instructions literally
- Code is the expression of an algorithm to another programmer
  - Concerned with the problem the algorithm solves
  - Note that *another programmer* might be you in the future!

Make It Easy to Read

- Comments in C
  - `/* block comments */`
    - Block comments for: pre- / post-conditions, expected behaviours, revision documentation
  - `// inline comments`
    - Inline comments for assertions, and / or a high-level description of algorithm, perhaps at a pseudocode level
- Variable naming
  - Choose names to aid the understanding of code
  - Naming conventions vary between codeshops
- Whitespace
  - Indentation, blank lines
  - Expression formatting

Design Approach



Testing + Debugging Go Hand in Hand

- Test bounds and extreme cases individually as well as "typical" cases
- Debug by
  - Probing variables
  - Hand-simulation
  - Debugger (profiler)

Reasoning About Code

- Preconditions (before)
  - Conditions that must be met in order for the function to operate correctly
- Assertions (during)
  - Conditions that must be met during execution of the function
- Postconditions (after)
  - Conditions that will be met by the function upon termination of the function
- Error handling (return codes, not exceptions)

Building Larger Projects

- Decompose problem into:
  - Pseudocode
  - Functions
  - Data types
  - Multiple files
- Build and test incrementally
  - Write 500 lines and then attempt to debug? *or*
  - Write 25 lines and then attempt to debug?

Six Stages of Debugging

1. That can't happen
2. That doesn't happen on my machine
3. Please don't let that happen
4. Why does that happen?
    1. The other guy's code is buggy
    2. The compiler is buggy
5. Oh, I see
6. How did that ever work?

Loop Invariants

Function Calls

- A puzzle
  - Write a program that outputs the first n cubes
    - Without using multiplication – only addition and subtraction
- Why?
  - CPUs are historically slow at multiplication compared to addition or subtraction
  - The speed differences vary

```
n = 10

0
1
8
27
64
125
216
345
512
729
```

Cubes Algorithm

- Calculating cubes only using addition
  - For each i from 0 to n-1
    - Compute the i^{th} square by adding i to itself i times
    - Compute the i^{th} cuibe by adding the ith square to itself i times
    - Output the i^{th} cube

Cubes Program

```
int main () {
  int n = 10;
  for (int i = 0; i < n; i++) {
     // Compute square == i*i
     int square = 0;
        for (int j = 0; j < i; j++) {
         // Assertion:
         square += i;
     }

     // Compute cube = i*i*i
     int cube = 0;
     for (int j = 0; j < i; j++) {
         cube += square;
     }
     printf("%d\n", cube);
  }
}
```

Do you believe that the value of *square* will equal *i*i* at the end of the loop?

What is a good assertion?

Assertions, also called loop invariants are usually related to the *post-condition*

Post-condition for square loop: *square* = *i* * *i*

When the loop terminates *j* = *i*

Assertion: *square* = *j* * *i*

What is a Good Loop Invariant?

- A loop invariant is a statement that is true for every iteration of the loop
  - Usually asserted at the beginning of the loop
  - Usually parametrized by the loop index
    - *j* in the case of the square calculation
- A good loop invariant should capture the progress of the algorithm
  - The invariant should carry all state information
  - The invariant should imply the post condition at the end of the loop

Mathematical Reasoning

- Use reasoning to capture the behavior of an algorithm
  - State invariants at various checkpoints
  - Show that the invariant is correct
    - At the first checkpoint
    - During execution between checkpoints
  - Conclude that the post-condition holds
    - i.e. that the invariant is correct at and after the last checkpoint

Is it true for the first step?

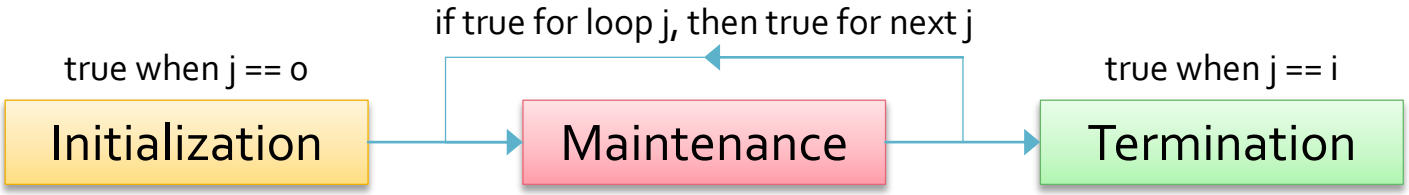
If its true for some arbitrary value of the loop control index is it true for the next?

If so we can conclude that it is true for all values of the loop control variable

This is mathematical induction

Proving Correctness

```
// …
// POST: square == i*i
int square = 0;
for (int j = 0; j < i; j++) {
    // Assertion: square == j * i
    square += i;
}
```



Proof

```
// POST: square == i*i
int square = 0;
for (int j = 0; j < i; j++) {
   // Assertion: square == j * i
   square += i;
}
```

- Initialization
  - Is the invariant true on the first loop?
    - When *j* == 0, *square* has been initialized to 0
    - Which satisfies square = *j***i*
- Maintenance
  - If the invariant holds at the start of loop *j*, does it hold at the start of loop *j*+1?
    - At the start of loop *j*, *square* == *j***i*
    - After the loop iteration *square* == *j***i* + *i* == (*j*+1) * *i*, the invariant for the next loop
- Termination
  - Since the invariant holds for all *j*, it holds after the last iteration
    - Therefore, when *j* == 1, *square* == *i* * *i*

So What?

- Why are we teaching you to prove correctness, ot to do proofs in general
  - Learn to do proofs to get better at reasoning about code
- Getting practice at thinking about invariants will
  - Make your code better
  - Make it easier to work out what other people's code is intended to do
- Computers cannot verify programs
  - In general this is an impossible problem

What's This?

```
int main () {
  int n = 10;
  int a = 6;
  int b = 1;
  int c = 0;

  for (int i = 0; i < n; i++) {
     printf("%d\n", c);
     c += b;
     b += a;
     a += 6;
  }
}
```

.. not this ...

What does it do?

1 – simulate it on paper

2 – type and run the program

Why is the program significant?

Hint: O Notation running time

Invariants

Note that $(n + 1)^3 - n^3 = 3(n + 1)n + 1$

```
int main () {
  int n = 10;
  int a = 6;
  int b = 1;
  int c = 0;

  for (int i = 0; i < n; i++) {
      // Assertion: a = 6(i+1)
      // Assertion: b = 3i*(i+1)+1
      // Assertion: c = i*i*i
      printf("%d\n", c);
      c += b;
      b += a;
      a += 6;
  }
}
```

Since the assertion $c = i^3$ holds on every loop, the algorithm is correct

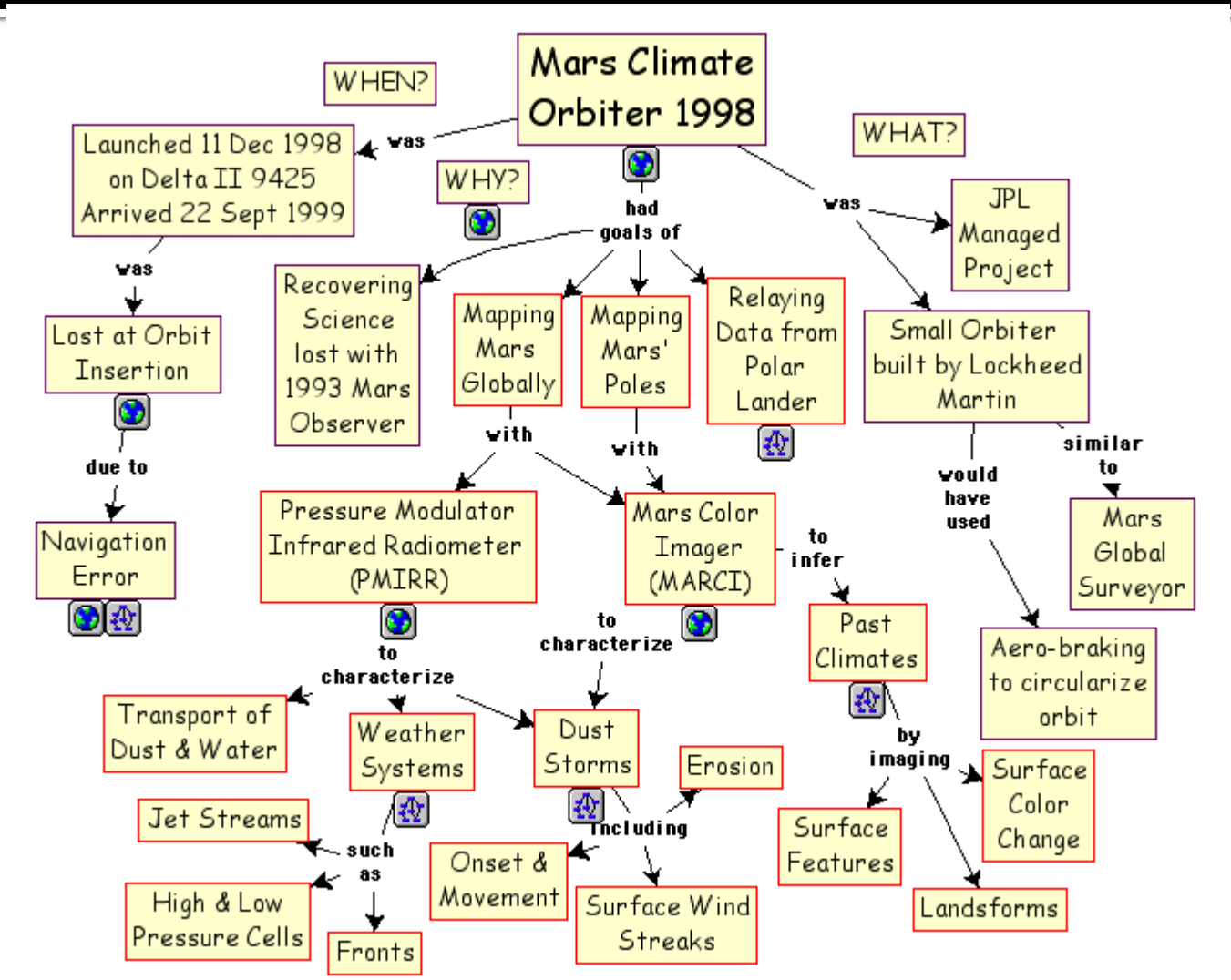
- Initialization: When i=0:
  - $a = 6(0+1) = 6$
  - $b = 3*0*(0+1)+1 = 1$
  - $c = 0^3 = 0$
- Maintenance: At the start of loop i:
  - $a = 6(i + 1)$
  - $b = 3i(i + 1) + 1$
  - $c = i^3$
- After c += b, c changes to:
  - $c = i^3 + 3i(i + 1) + 1$
  - $c = i^3 + 3i^2 + 3i + 1$
  - $c = (i + 1)^3$
- After b += a, b changes to:
  - $b = 3i(i + 1) + 1 + 6(i + 1)$
  - $b = (i + 1)(3i + 6) + 1$
  - $b = 3(i + 1)(i + 2) + 1$
- After a += 6, a changes to:
  - $a = 6(i + 1) + 6$
  - $a = 6(i + 2)$

Correctness

Ariane5 Rocket

- June 4, 1996 – launch of the Ariane5
  - https://www.youtube.com/watch?v=PK_yguLapgA
- A function in the inertial reference system tried to convert a floating point number to a signed 16 bit integer
  - The conversion resulted in a value that was out of range
  - The run time error was detected in both the active and backup computers which both shut down
  - Resulting in a loss of altitude control
  - This, in turn, resulted in the rocket turning uncontrollably and breaking apart
  - The breakup of the rocket was detected by an on-board monitor
  - Explosive charges were then detonated to destroy the rocket in the air

Mars Climate Orbiter - 1999



Mars Climate Orbiter - 1999

- 1999 MCO was lost just as it entered Mars orbit
  - It entered Mars orbit too low and disintegrated when it hit the upper atmosphere
  - Cost of orbiter - $330 million
- What went wrong
  - Failed translation of Imperial units into Metric units in a segment of navigation-related mission software
- Root cause
  - Systems engineering, project management, and communication problems

Therac-25

- Therac-25 was a computerized radiation therapy machine
  - Manufactured in 1982 by Atomic Energy of Canada Limited

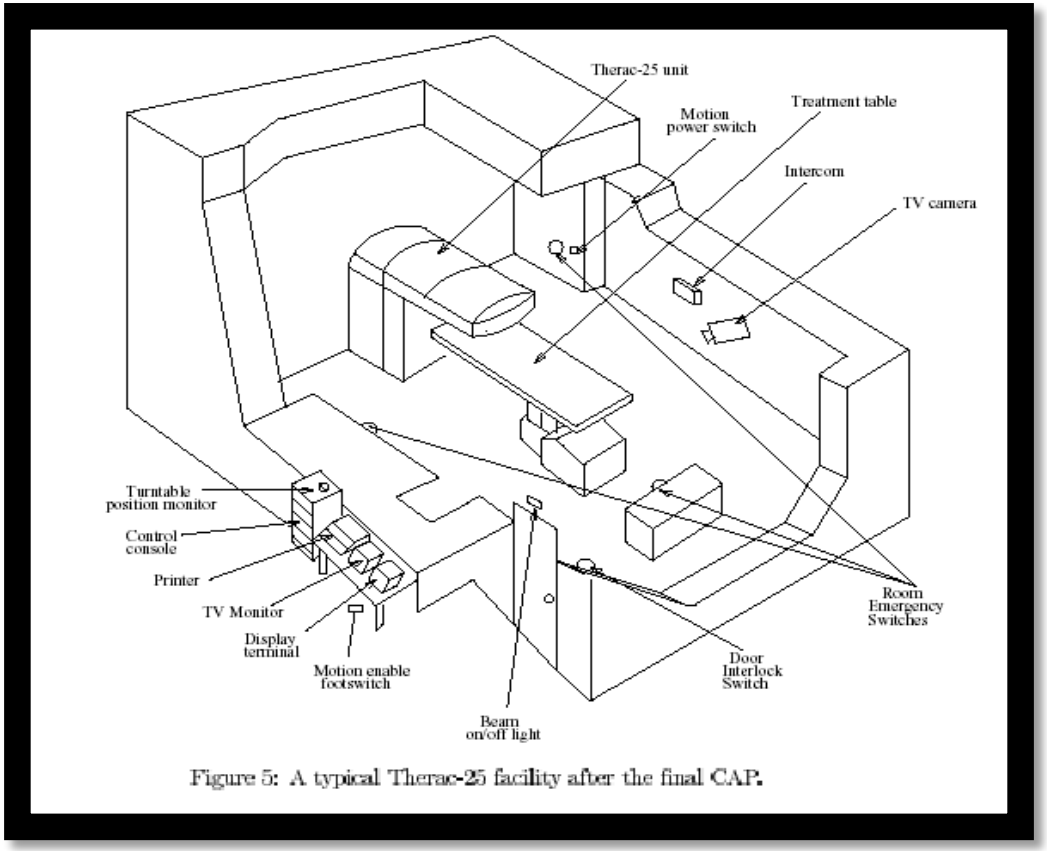


Figure 5: A typical Therac-25 facility after the final CAP.

Therac-25

- The model was software-controlled by a PDP-11 computer
  - Previous versions were hardware- controlled
    - With mechanical interlocks to prevent overdose and used software merely for convenience
- In case of software error, cryptic codes were given back to the operator, such as
  - "MALFUNCTION xx", where 1 < xx < 64

Therac-25

- Operators were rendered insensitive to the errors
  - They happened often, and they were told it was impossible to overdose a patient
- However, from 1985-1987, six people received massive overdoses of radiation
  - At least three of them died from the overdose

Therac-25: Main Cause

- Race condition often happened when operator entered data quickly
  - Then hit the UP arrow key to correct but values weren't reset properly
- AECL never noticed quick data-entry
  - Since their testers didn't perform the tasks on a daily basis
- The problem existed in previous units
  - But they had a hardware interlock mechanism to prevent it

Computers Do Not Make Mistakes

- Algorithms on a computer are written in a formal and unambiguous programming language
  - Which cannot be misinterpreted by the computer
- Modern hardware is essentially bug-free
  - Therefore "computer errors" are either
    - Programmer errors or
    - User errors

PEBKAC

RTFM

Preventing Bugs

- Compilers
  - Find syntax errors
  - Warn of common bugs and suggest syntax corrections
- Beyond syntax, the compiler cannot help you fix your program
  - It does not know what you are trying to achieve
- Code must be thoroughly tested to remove bugs

Testing and Debugging

- The more you test your program the more likely you are to find bugs, particularly
  - Run-time errors
  - Logic errors
  - Infinite loops
- Your code is only as good as your tests
  - Some bugs may never be discovered
  - Is testing better than the proof of a loop invariant
- Types of testing
  - White box
  - Black box

Proving Correctness

- We can use mathematical proofs to reason about algorithms
  - Such as assertions and loop invariants
- Can this process be automated?
  - Is there an algorithm that takes a problem description, *P*, and an algorithm, *A*, and determines if *A* solves *P*?
  - In general – no!

Loop Invariant Review

- A loop invariant is a statement that is true in every iteration of a loop
  - Usually asserted at the beginning of the loop and
  - Usually parametrized by the loop index
- A good loop invariant should indicate the progress of the algorithm
  - The invariant should carry all state information from loop to loop
  - The invariant should imply the post-condition at the end of the last iteration of the loop

Loop Invariant Review

- Use mathematical reasoning to capture the behavior of an algorithm
  - State invariants at various checkpoints
  - Show that the invariant holds
    - At the first checkpoint
    - During execution between checkpoints
  - Conclude that the post-condition holds
    - The invariant holds at the last checkpoint
- But what if the algorithm has no loops?
  - Invariants are very powerful for recursive algorithms

Initialization

Maintenance

Termination

Invariants and Recursion

```
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long factorial(int x){
        if(x <= 1){
                return 1;
        }
        return x * factorial(x-1);
}
```

Factorial Definition

0! = 1

1! = 1

n! = n × (n-1)! for all n ≥ 2

Assume that the invariant holds for any smaller case

Recursive sub-call

Assume that *factorial*(*x*-1) correctly returns (*x*-1)!

Another Example

```
// POST: returns base**exp
int power(int base, unsigned int exp){
        if(exp == 0){
                return 1;
        }

        return base * power(base, exp-1);
}
```

Definition of b^e

$b^0 = 1$

$b^e = b \times b^{(e-1)}$ for all $b > 0$

What does the running time depend on?

Assume that the recursive call to *power*(*base*, *exp*-1) correctly returns the correct value

Let n be the value of *exp*

$T(n) = O(1) + T(n-1)$ when $n > 0$

$T(0) = O(1)$

This is a *recurrence relation*

Better Solution

- Is there a solution to the power function with a faster O Notation running time?
  - Use divide and conquer
- Key observation
  - Can you compute result quickly if the exponent is even?
  - Call *power*(*base*, *exp*/2) and square the result
- Remember that any smaller case is correct
  - It does not have to be incrementally smaller

Another Example

```
// POST: returns base**exp
int power(int base, unsigned int exp){
        if(exp == 0){
                return 1;
        }
        int x = power(base, exp/2);
        if(exp % 2 == 1){ //odd
                return x * x * base;
        }else{
                return x * x;
        }
}
```

What's the running time?

$T(n) = O(1) + T(n/2)$ when $n > 0$

$T(0) = O(1)$

$T(n) = O(\log n)$