

# CS 106B, Lecture 8

## Recursion

# Plan for Today

- Learn a powerful algorithmic technique called *recursion*
  - Exploit self-similarity in problems
- We will spend several days on recursion – don't worry if it doesn't make sense today
  - Goal: do as many examples as we can
  - You should **practice**: [CodeStepByStep](#), section problems, or examples from the textbook
  - Highly encourage the reading for this week!

# Recursion

- **recursion:** Function that calls itself
  - Solving a problem using recursion depends on solving smaller (simpler) occurrences of the same problem until the problem is simple enough that you can solve it directly
  - Key question: *"How is this problem self-similar?"* – what are the smaller subproblems that make up the bigger problem?
- Occurs in many places in code and in real world:
  - Looking up a word in dictionary may involve looking up other words
  - Nested structures (trees, file folders, collections) can be self-similar

# Recursive Programming

- **recursive programming**: Writing functions that call themselves to solve problems that are recursive in nature.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems
  - Leads to **elegant**, simplistic, short code (when used well)
  - A key component of many of our assignments in this course

# Non-recursive factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.  
// Assumes n >= 1.  
int factorial(int n) {  
    int total = 1;  
    for (int i = 1; i <= n; i++) {  
        total *= i;  
    }  
    return total;  
}
```

- Important observations:

$$0! = 1! = 1$$

$$4! = \underline{4 * 3 * 2 * 1}$$

$$5! = 5 * \underline{4 * 3 * 2 * 1}$$
$$= 5 * 4!$$

# Recursive factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.  
// Assumes n >= 0.  
int factorial(int n) {  
    if (n <= 1) { // base case  
        return 1;  
    } else {  
        return n * factorial(n - 1); // recursive case  
    }  
}
```

- The recursive code handles a small part of the overall task (multiplying by  $n$ ), then makes a recursive call to handle the rest.
  - The recursive version is written without using any loops.
    - Recursion *replaces* the loop
  - We separate the code into a *base case* (a simple case that does not make any recursive calls), and a *recursive case*.

# Recursive stack trace

```
int factorial(int n) { // 4
    if (n <= 1) {           // base case
        return 1;
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

```
int factorial(int n) { // 3
    if (n <= 1) {           // base case
        return 1;
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

```
int factorial(int n) { // 2
    if (n <= 1) {           // base case
        return 1;
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

```
int factorial(int n) { // 1
    if (n <= 1) {           // base case
        return 1;
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

# Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case:** A simple occurrence that can be answered directly
  - **recursive case:** A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem
  - *Key idea:* In a recursive piece of code, you handle a small part of the overall task yourself (usually the work involves modifying the results of the smaller problems), then make a recursive call to handle the rest.
  - Ask yourself, "How is this task **self-similar**?"
    - "How can I describe this algorithm in terms of a smaller or simpler version of itself?"



# Three Rules of Recursion

- Every (valid) input must have a case (either recursive or base)
- There **must** be a base case that makes no recursive calls
- The recursive case must make the problem simpler and make forward progress to the base case

# Recursive tracing

- Consider the following recursive function:

```
int mystery(int n) {  
    if (n < 10) {  
        return n;  
    } else {  
        int a = n / 10;  
        int b = n % 10;  
        return mystery(a + b);  
    }  
}
```

**Q:** What is the result of: `mystery(648)` ?

**A.** 8      **B.** 9      **C.** 54      **D.** 72      **E.** 648

# Recursive stack trace

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
        int mystery(int n) { // n = 9
            if (n < 10) {
                return n; // return 9
            } else {
                int a = n / 10;
                int b = n % 10;
                return mystery(a + b);
            }
        }
    }
}
```

# isPalindrome exercise

- Write a recursive function `isPalindrome` that accepts a string and returns `true` if it reads the same forwards as backwards.

<code>isPalindrome("madam")</code>	→ <code>true</code>
<code>isPalindrome("racecar")</code>	→ <code>true</code>
<code>isPalindrome("step on no pets")</code>	→ <code>true</code>
<code>isPalindrome("able was I ere I saw elba")</code>	→ <code>true</code>
<code>isPalindrome("Q")</code>	→ <code>true</code>
<code>isPalindrome("Java")</code>	→ <code>false</code>
<code>isPalindrome("rotater")</code>	→ <code>false</code>
<code>isPalindrome("byebye")</code>	→ <code>false</code>
<code>isPalindrome("notion")</code>	→ <code>false</code>

- What is a good **base case**?

# isPalindrome

- What is our stopping point (*base case*)?
- How is this problem *self-similar*?
- What is the minimum *amount of work*?
- How can we make the problem *simpler* by doing the least amount of work?

# isPalindrome

- What is our stopping point (*base case*)?
  - Empty string or string of length 1
- How is this problem *self-similar*?
  - Palindromes can be written as:  $x[\text{SMALLER\_PALINDROME}]x$ , where  $x$  stands for some letter
- What is the minimum *amount of work*?
  - Testing the equality of outside characters
- How can we make the problem *simpler* by doing the least amount of work?
  - Peel off the outside characters and test if the middle is a palindrome

# isPalindrome solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(string s) {
    if (s.length() < 2) {    // base case
        return true;
    } else {                // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

# Announcements

- Homework 2 due on Wednesday at **5PM**
- Homework 1 grades will be released by your section leader on or before Wednesday
- Alternate midterms are being scheduled this week. Keep an eye out for an email from Kate



# Multiple calls tracing

```
int mystery(int n) {  
    if (n < 10) {  
        return (10 * n) + n;  
    } else {  
        int a = mystery(n / 10);  
        int b = mystery(n % 10);  
        return (100 * a) + b;  
    }  
}
```

**Q:** What is the result of: `mystery(348)` ?

- A.** 3828      **B.** 348348      **C.** 334488      **D.** 80403      **E.** none

# Multiple calls tracing

```
// call 1: 348
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

```
// call 2a: 34
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

```
// call 2b: 8
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

```
// call 3a: 3
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

```
// call 3b: 4
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

# Recursive Big O

- Below is the "pseudocode" for finding Big O of a function
  - Note that this is not real code; this is to show the recursive nature of finding Big O
  - Self-similarity: find Big O of smaller code blocks and combine them
  - This Big O pseudocode doesn't cover function calls and some other cases (for pedagogical purposes)

```
findBigO(codeSnippet):  
  if codeSnippet is a single statement:  
    return O(1)  
  if codeSnippet is loop:  
    return number of times loop runs * findBigO(loop inside)  
  for codeBlock in codeSnippet:  
    return the sum of findBigO(codeBlock)
```

# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

```
cout << "Have a nice Life!" << endl;
```

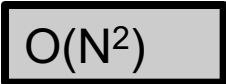
# Finding Big O Recursively

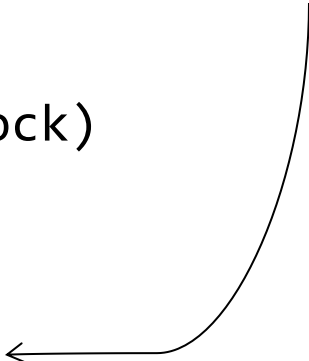
```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

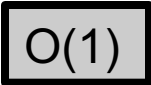
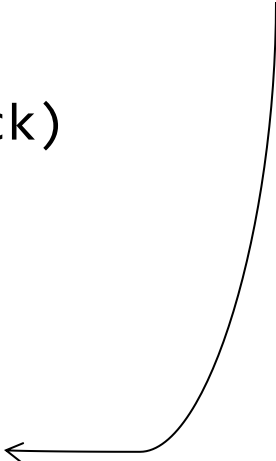
```
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:   
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)  
  
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}  
  
cout << "Have a nice Life!" << endl;
```



# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:   
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)  
  
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;   
    }  
}  
  
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

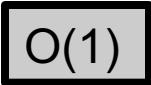
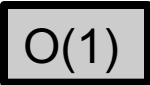
```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

```
cout << "Have a nice Life!" << endl;
```




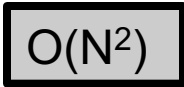
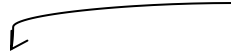
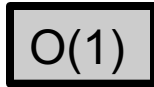
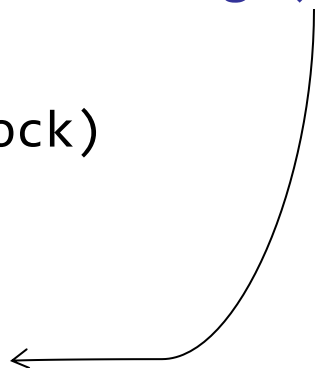
# Finding Big O Recursively

```
findBigO(codeSnippet):  
  if codeSnippet is a single statement:  
    return O(1)  
  if codeSnippet is loop:    
    return number of times loop runs * findBigO(loop inside)  
  for codeBlock in codeSnippet:  
    return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
  for (int j = 3; j <= 219; j++) {  
    cout << "sum: " << i + j << endl;  
  }  
}
```

```
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

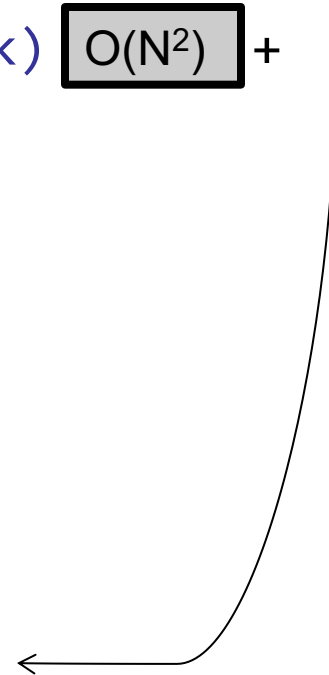
```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:      
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)  
  
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {   
        cout << "sum: " << i + j << endl;  
    }  
}  
  
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock) O(N2) +
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

```
cout << "Have a nice Life!" << endl;
```



# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

```
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

```
findBigO(codeSnippet):  
  if codeSnippet is a single statement:  
    return O(1)  
  if codeSnippet is loop:  
    return number of times loop runs * findBigO(loop inside)  
  for codeBlock in codeSnippet:  
    return the sum of findBigO(codeBlock)  $O(N^2)$  +  $O(1)$ 
```

```
for (int i = 0; i < N * N; i += 3) {  
  for (int j = 3; j <= 219; j++) {  
    cout << "sum: " << i + j << endl;  
  }  
}
```

```
cout << "Have a nice Life!" << endl;
```

# Finding Big O Recursively

```
findBigO(codeSnippet):  
    if codeSnippet is a single statement:  
        return O(1)  
    if codeSnippet is loop:  
        return number of times loop runs * findBigO(loop inside)  
    for codeBlock in codeSnippet:  
        return the sum of findBigO(codeBlock)
```

```
for (int i = 0; i < N * N; i += 3) {  
    for (int j = 3; j <= 219; j++) {  
        cout << "sum: " << i + j << endl;  
    }  
}
```

final result:  $O(N^2)$

```
cout << "Have a nice Life!" << endl;
```

# power exercise

- Write a function **power** that accepts integer parameters for a base and exponent and computes  $\text{base}^{\text{exponent}}$ .
  - Write a recursive version of this function (one that calls itself).
  - Solve the problem without using any loops.
  - What is our stopping point (*base case*)?
  - How is this problem *self-similar*?
  - What is the minimum *amount of work*?
  - How can we make the problem *simpler* by doing the least amount of work?

# Initial solution

```
// Returns base ^ exp.  
// Assumes exp >= 1.  
int power(int base, int exp) {  
    if (exp == 1) {  
        return base;  
    } else {  
        return base * power(base, exp - 1);  
    }  
}
```



# The call stack

- Each previous call waits for the next call to finish.

– cout << **power(5, 3)** << endl;

```
// first call: 5      3
int power(int base, int exp) {
    if (exp == 1) {
        // second call: 5      2
        int power(int base, int exp) {
            if (exp == 1) {
                // third call: 5      1
                int power(int base, int exp) {
                    if (exp == 1) {
                        return base; // 5
                    } else {
                        return base * power(base, exp - 1);
                    }
                }
            }
        }
    }
}
```

# "Recursion Zen"

- The real, even simpler, base case is an exp of 0, not 1:

```
int power(int base, int exp) {  
    if (exp == 0) {  
        // base case; base^0 = 1  
        return 1;  
    } else {  
        // recursive case: x^y = x * x^(y-1)  
        return base * power(base, exp - 1);  
    }  
}
```

- **Recursion Zen:** The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

Opposite is **arms-length recursion**

*(our informal term)*

# Preconditions

- **precondition:** Something your code *assumes is true* when called.
  - Often documented as a comment on the function's header:

```
// Returns base ^ exp.  
// Precondition: exp >= 0  
int power(int base, int exp) {
```

- Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.
- What if the caller doesn't listen and passes a negative power anyway?  
What if we want to actually *enforce* the precondition?

# Throwing exceptions

`error(expression);`

- In Stanford C++ lib's "error.h"
- Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.
- alternative: throw *something*
  - *something* can be an int, a string, etc.
- Why would anyone ever *want* a program to crash?

# power solution 2

```
// Returns base ^ exp.  
// Precondition: exp >= 0  
int power(int base, int exp) {  
    if (exp < 0) {  
        throw "illegal negative exponent";  
    } else ...  
        ...  
}
```