Recursion

Recursion

Recursion

Recursion

Recursion

Recursion

Recursion

Recursion

no iClickers in this set

Function/method calls

- A function or method call is an interruption or aside in the execution flow of a program:

```
...
int a, b, c, d;
a = 3;
b = 6;
c = foo(a, b);
d = 9;
...
```

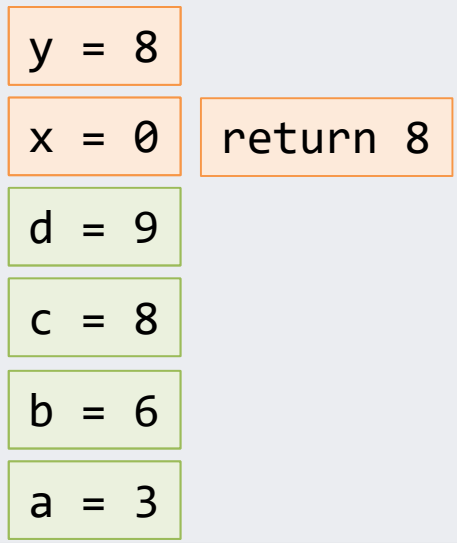
```
int foo(int x, int y) {
  while (x > 0) {
    y++;
    x >>= 1; // bitwise right shift by 1
  }
  return y;
}
```

Activation records on a computer

- A computer handles function/method calls in the same way

```
...
int a, b, c, d;
a = 3;
b = 6;
c = foo(a, b);
d = 9;
...
```

```
int foo(int x, int y) {
  while (x > 0) {
    y++;
    x >>= 1; // bitwise right shift by 1
  }
  return y;
}
```



Recursion works the same way

- The n^{th} number in the Fibonacci series, *fib*(*n*), is:
  - 0 if $n = 0$, and 1 if $n = 1$
  - $fib(n - 1) + fib(n - 2)$ for any $n > 1$
- e.g. what is *fib*(23)
  - Easy if we only knew *fib*(22) and *fib*(21)
  - The answer is *fib*(22) + *fib*(21)
  - What happens if we actually write a function to calculate Fibonacci numbers like this?

Calculating the Fibonacci series

- Let's write a function just like the formula
  - *fib*(*n*) = 0 if *n* = 0, 1 if *n* = 1,
  - otherwise *fib*(*n*) = *fib*(*n* – 1) + *fib*(*n* – 2)

```
int fib(int n)
{
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

The function calls itself

Recursive functions

- The Fibonacci function is *recursive*
  - A recursive function calls itself
  - Each call to a recursive method results in a *separate* call to the method, with its own input
- Recursive functions are just like other functions
  - The invocation (e.g. parameters, etc.) is pushed onto the call stack
  - And removed from the call stack when the end of a method or a return statement is reached
  - Execution returns to the previous method call

Recursive function anatomy

- Recursive functions do not use loops to repeat instructions
  - But use recursive calls, in if statements

- Recursive functions consist of two or more cases, there must be at least one
  - Base case, and
  - Recursive case

(base case size and answer)
→ smallest problem size(s) for which the solution requires no recursion

→ ask recursion to give us a solution for a smaller problem

- use smaller solution to construct solution to our larger problem

"Trust the natural recursion."

Recursion cases

- The base case is a smaller problem with a known solution
  - This problem's solution must *not* be recursive
    - Otherwise the function may never terminate
- There can be more than one base case
  - And base cases may be implicit

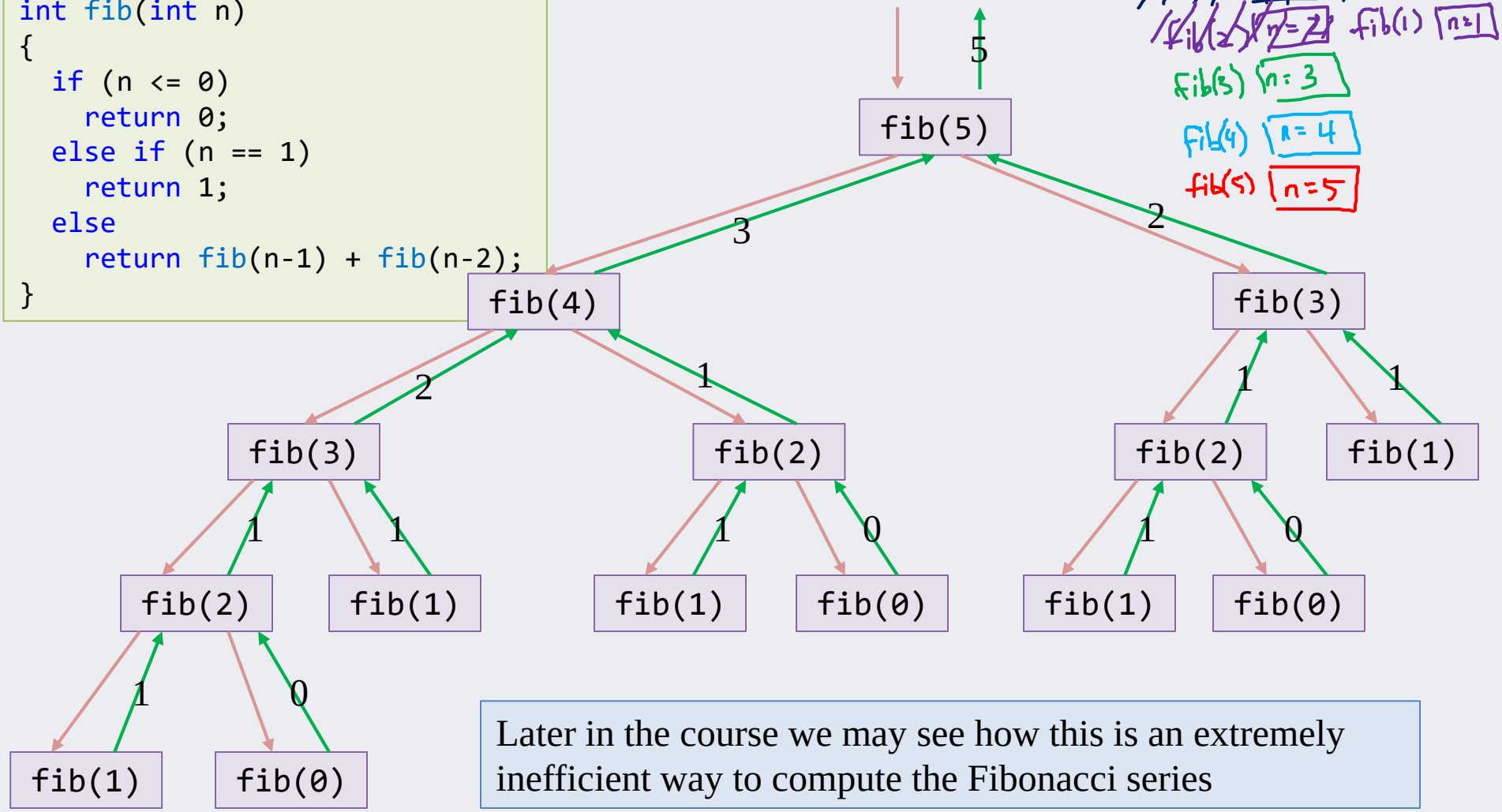
- The recursive case is the same problem with smaller input
  - The recursive case must include a recursive function call
  - There can be more than one recursive case

```
if the problem is small enough to be solved directly
        solve it
otherwise
        (1) recursively apply the algorithm to one or more smaller instances
        (2) use the solution(s) from smaller instances to solve the problem
```

Analysis of `fib(5)`

Recursion tree

```
int fib(int n)
{
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

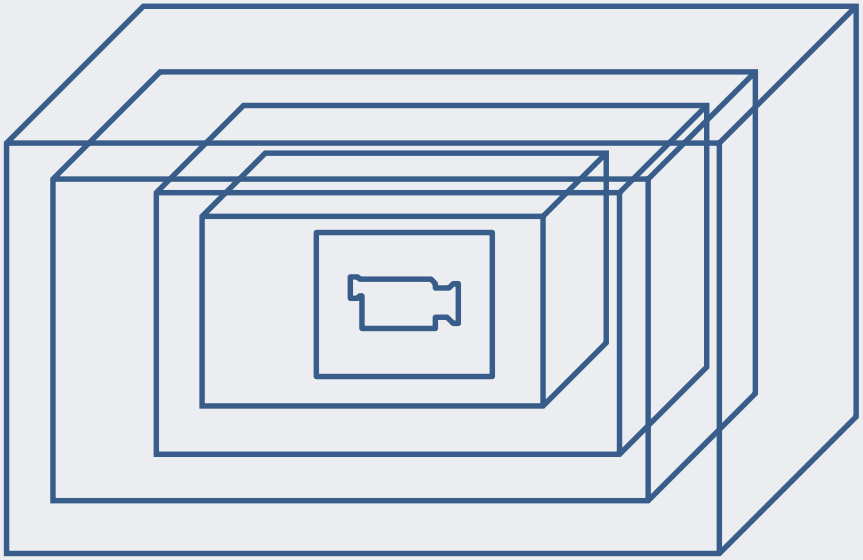


Later in the course we may see how this is an extremely inefficient way to compute the Fibonacci series

Thinking recursively

Opening a present

- Your friends have given you a camera as a birthday present. To prolong the suspense when opening the present, they have bundled it into several nested wrapped packages. How do you open the present?

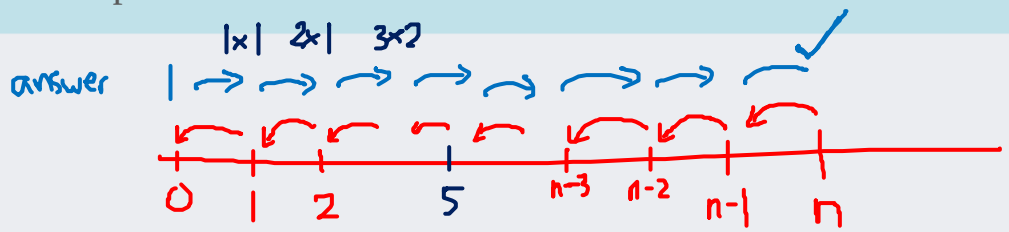


```
openPresent(pkg) {
  if you can see the actual gift
    say "thank you"
  else
    open the box to reveal spkg
    openPresent(spkg)
}
```

Designing recursive functions

Example: factorial

- $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
- $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
- $6! = 6 \cdot 5!$
- $0! = 1$
- $n! = (n) \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$
- or, $n! = n \cdot (n-1)!$



$(n-1)!$

call stack

fact(4) n=4

fact(5) n=5

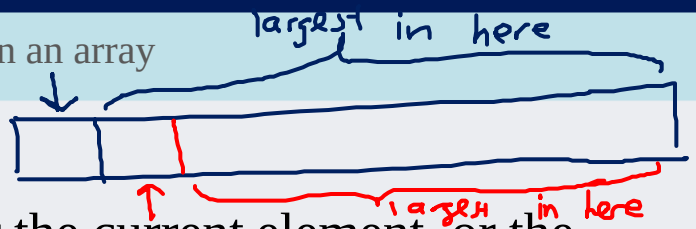
$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

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

```
else {
  int f_n_minus_1 = factorial(n-1);
  int f_n = n * f_n_minus_1
  return f_n;
```

Designing recursive functions

Example: maximum value in an array



- Intuition: inspect a single element
  - The maximum value is the array is either the current element, or the largest value in the rest of the array, whichever is larger
- When do we know for sure we have the largest element?
  - When the (sub)array contains just a single element

```
int arrayMax(int arr[], int size, int start) {
  if (start == size - 1)
    return arr[start];
  else
    return max(arr[start], arrayMax(arr, size, start+1));
}
```

current index

last index

current element

largest element in the rest of the array

Designing recursive functions

Example: summation

- $\sum_{i=0}^{n} i = n + (n-1) + (n-2) + \cdots + 2 + 1 + 0$
- $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$
- $\sum_{i=0}^{0} i = 0$

```
int summation(int n) {
  if (n <= 0)
    return 0;
  else
    return (n + summation(n-1));
}
```

```
else {
    int sum_n_minus_1 = summation(n-1);
    int sum_n = n + sum_n_minus_1;
    return sum_n;
}
```

Stack overflow

It's not just a useful website

- By default, program stack space is extremely limited
  - If many function invocations are placed on the stack without returning, a stack overflow can result
  - You encountered this in lab 1 already

```
int summation(int n) {
  if (n <= 0)
    return 0;
  else
    return (n + summation(n-1));
}
```

```
summation(1000000);
```

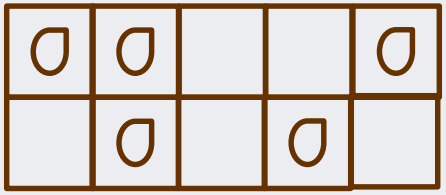
```
int factorial(int n) {
  if (n <= 0)
    return 1;
  else
    return (n * factorial(n-1));
}
```

↑ likely will produce integer overflow before stack overflow

More recursive function design

Eating a chocolate bar

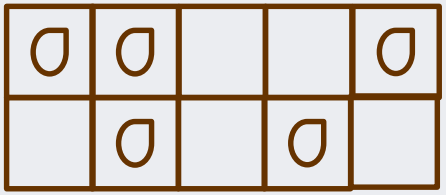
- Suppose I have a (mild) nut allergy, and a chocolate bar with nuts. I can only eat the squares that do not contain nuts. Write a recursive algorithm that lets me eat the chocolate bar.



```
eatChocolateBar(b)
{
  if (b is a single square)
    if (b does not contain a nut)
      eat it
  else
    break the bar into two pieces
    eatChocolateBar(piece1)
    eatChocolateBar(piece2)
}
```

More recursive function design

Eating a chocolate bar



```
eatChocolateBar(b)
{
  if (b is a single square)
    if (b does not contain a nut)
      eat it
  else
    break the bar into two pieces
    eatChocolateBar(piece1)
    eatChocolateBar(piece2)
}
```

Tail recursion

- A function is tail-recursive if the recursive call is the absolute last thing the function needs to do before returning
- No need to wait for a return from a deeper recursive call to compute a result
  - Why bother pushing a new stack frame? There is nothing to remember
  - Just use the old stack frame
  - Most compilers will do this

Ordinary vs tail recursion

- How are these functions similar/different?

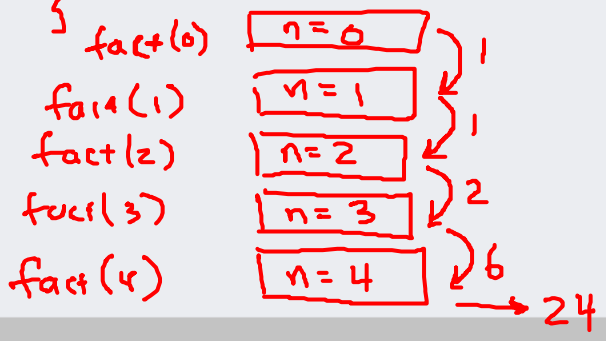
```
int factorial(int n) {
  if (n <= 0)
    return 1;
  else
    return (n * factorial(n-1));
}
```

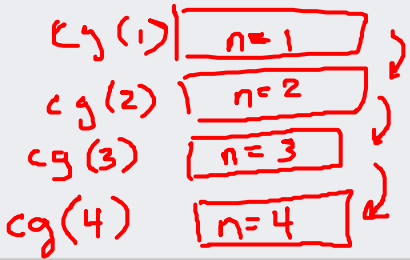
```
else {
   int f_nminus1 = factorial(n-1);
   int result = n * f_nminus1;
   return result;
}
```

```
void countingGreet(int n) {
  if (n <= 1)
    return;
  printf("Hello!\n");
  countingGreet(n-1);
}
```

no computation occurs after recursive call

Think about the program flow of a call e.g. `factorial(4)` vs `countingGreet(4)`





Tail recursive factorial

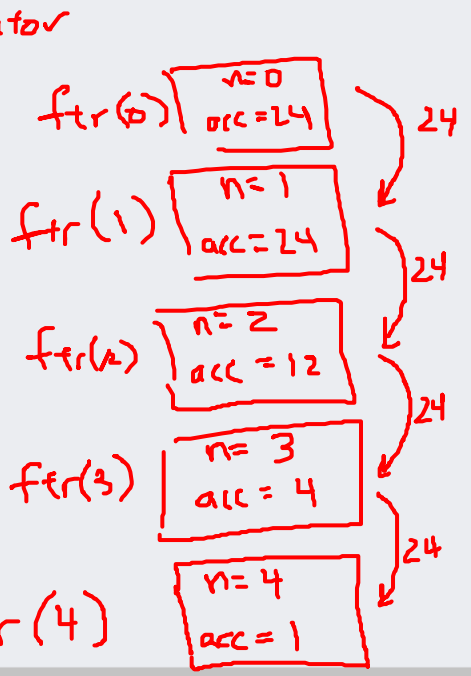
- Use an additional parameter (and a recursive helper function) to keep track of the computed factorial so far

```
int factTail(int n) {
  int result = factTailRec(n, 1);
  return result;
}
```

```
int factTailRec(int n, int acc) {
  if (n == 0)
    return acc;
  else
    return factTailRec(n-1, n*acc);
}
```

accumulator

else {
int subproblem_size = n-1;
int accumulated_result = n*acc;
return factTailRec(subproblem_size, a_r);



Tail recursive Fibonacci

```
int fibTail(int n) {
  return fibTailRec(n, 1, 1);
}
```

```
int fibTailRec(int n, int next, int result) {
  if (n == 1)
    return result;
  else
    return fibTailRec(n-1, result + next, next);
}
```

This runs almost as quickly as an iterative implementation, and uses about the same amount of stack space as well