EECS 280 – Lecture 18

Recursion and Tail Recursion

3/21/2022

Motivating Example: Factorial

- The factorial of a non-negative integer *n* is

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1, & n > 1 \end{cases}$$

- For example:

$$\begin{aligned} 5! &= 5 * 4 * 3 * 2 * 1 \\ &= 120 \end{aligned}$$

$$1! = 1$$

$$0! = 1$$

Implementing Factorial

- To implement factorial, we could use a loop…

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int fact(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    --n;
  }
  return result;
}
```

- Question: Can we do it without a loop?

Recursive Factorial

- Can we do it without a loop, using **recursion** instead?

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int fact(int n) {
  if (n == 0) {
    return 1;
  }
  else {
    return n * fact(n - 1);
  }
}
```

A recursive function calls itself to implement repetition.

L18.2_fact on Lobster

- Does this work?
  - How does one call set the parameters for the next?
  - How does the recursion know to stop?
  - Where does the multiplication happen?

Solving Problems with Recursion

- A recursive approach needs:

1. A base case
   - Can be solved without recursion

2. Recursive cases:
   - Break down into **subproblems** that are **similar** and **smaller** (closer to a base case)

No base case? Infinite Recursion!

Subproblems that are:

- Similar? **YES**
- Smaller? **NO**

```
void countToInfinity(int x) {
  cout << x << endl;
  // Recursive Case:
  countToInfinity(x + 1);
}

int main() {
  countToInfinity(0);
}
```

Recurrence Relations

- Earlier, we defined factorial iteratively:

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1, & n > 1 \end{cases}$$

- But factorial can also be defined **recursively**:

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

- Factorial is defined in terms of itself!
- This is called a **recurrence relation.**

Writing Recursive Functions

- Start with the recurrence relation:

Base Case

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

Recursive Case

- This often translates directly to code:

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int fact(int n) {
  if (n <= 1) { // BASE CASE
    return 1;
  }
  else {
    return n * fact(n - 1); // RECURSIVE CASE
  }
}
```

The Recursive Leap of Faith

- Here's the trick to writing recursive functions…
- Use the function to write itself. Even before it's "done".
  - It's ok. Trust us.
- Check the "combination" step.

```
// REQUIRES: n >= 0
// EFFECTS: computes and returns n!
int fact(int n) {
  if (n <= 1) { // BASE CASE
    return 1;
  }
  else {
    return n * fact(n - 1); // RECURSIVE CASE
  }
}
```

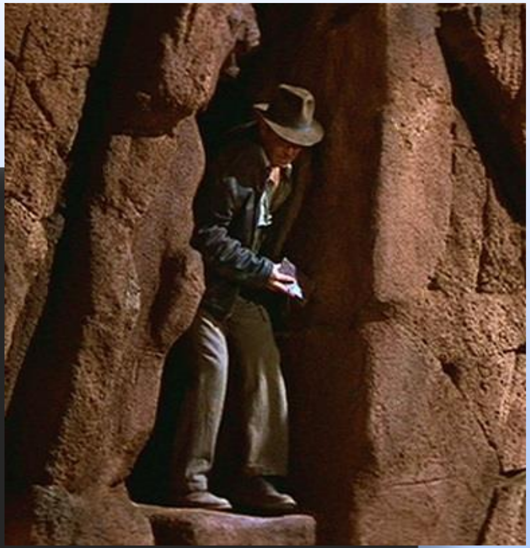


Image: Indiana Jones and The Last Crusade

Exercise: Ducks

- Let's say we want to start a duck farm:
  - We start with 5 baby ducklings.
  - At age 1 month, and **every month** thereafter, each duck lays **3 eggs**.
  - An egg takes 1 month to hatch.
  - All eggs hatch, and ducks never die.

- How many ducks do we have after `n` months?

n	0	1	2	3	4	5	…
`numDucks(n)`	5	5	20	35	95	200	…

- Find a recurrence relation for `numDucks(n)`.
  - Hint: The number of ducks is the number of previous ducks plus the number that have just hatched. (Two subproblems!)

Exercise: Ducks

Question

How many Ducks do we have at month n=6?

A) 425 B) 460 C) 485 D) 755

- Given this recurrence relation:

```
numDucks(n) = numDucks(n-1) + 3 * numDucks(n-2)
```

Number of existing ducks.

Number of newly hatched ducks.

- Write code to implement `numDucks`:

```
// REQUIRES: n >= 0
// EFFECTS: computes the number of ducks at month n
int numDucks(int n) {

}
```

Solution: Ducks

- Given this recurrence relation:

```
numDucks(n) = numDucks(n-1) + 3 * numDucks(n-2)
```

Number of existing ducks.

Number of newly hatched ducks.

- Write code to implement `numDucks`:

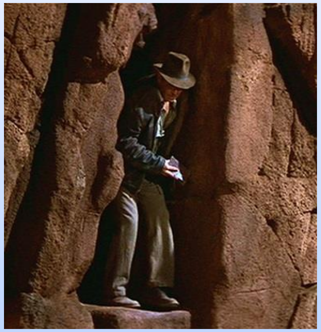
```
// REQUIRES: n >= 0
// EFFECTS: computes the number of ducks at month n
int numDucks(int n) {
  if (n <= 1) { // BASE CASE
    return 5;
  }
  else {
    // RECURSIVE CASE
    return numDucks(n - 1) + 3 * numDucks(n - 2);
  }
}
```

Exercise: Recursive Reverse

- How can we reverse an array using recursion?

1	2	3	4	5	6	7	8

- You're allowed to reverse any smaller array. (Or a "subarray" within the original.)



- Brainstorm an algorithm and write pseudocode for it.
  - What's the recurrence?
  - What's the base case?

Exercise: Recursive Reverse

- Let's assume you're working with this algorithm:
- If the array size <= 0:
  1. Do nothing.
- If the array is not empty:
  1. Reverse the "middle" of the array (i.e. the recursive call).
  2. Swap the first/last elements of the array.

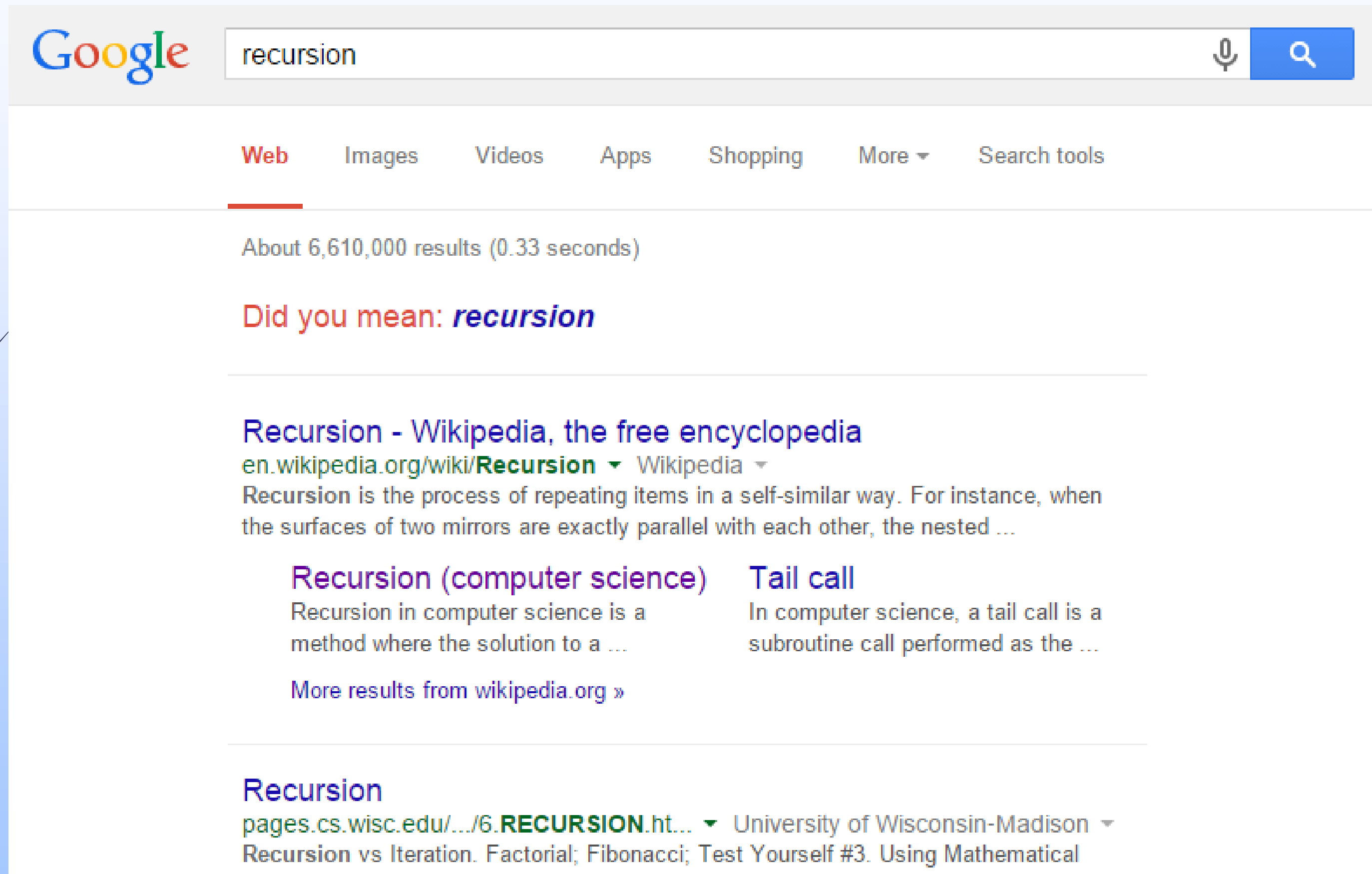
```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {






}
```

Solution: Recursive Reverse

- Let's assume you're working with this algorithm:
- If the array size <= 0:
  1. Do nothing.
- If the array is not empty:
  1. Reverse the "middle" of the array (i.e. the recursive call).
  2. Swap the first/last elements of the array.

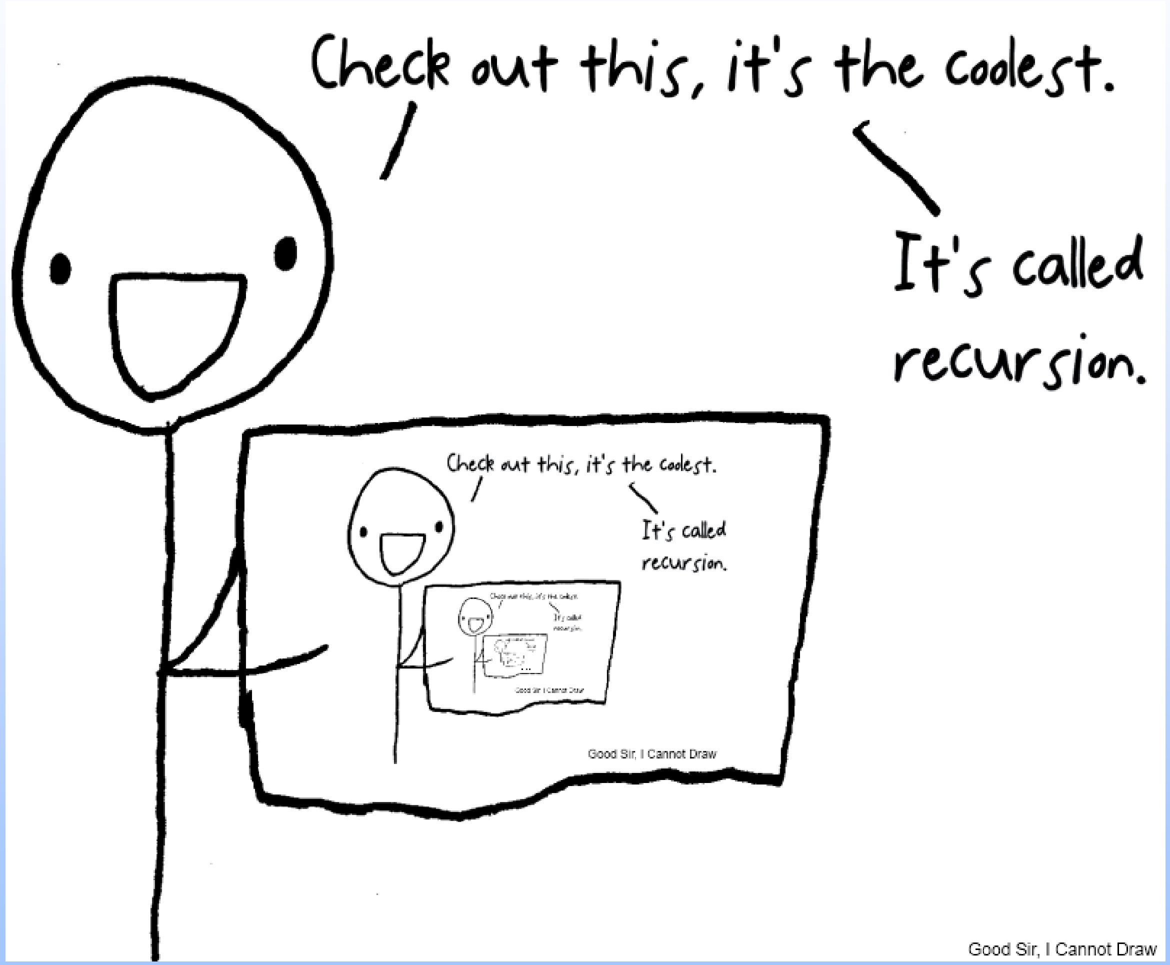
```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {
  if (left < right) {
    reverse(left + 1, right - 1);
    int temp = *left;
    *left = *right;
    *right = temp;
  }
}
```

Break Time!



Google
recursion
Web
Images
Videos
Apps
Shopping
More
Search tools
About 6,610,000 results (0.33 seconds)
Did you mean: recursion
Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion
Wikipedia
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other, the nested ...
Recursion (computer science)
Recursion in computer science is a method where the solution to a ...
Tail call
In computer science, a tail call is a subroutine call performed as the ...
More results from wikipedia.org »
Recursion
pages.cs.wisc.edu/.../6.RECURSION.ht...
University of Wisconsin-Madison
Recursion vs Iteration. Factorial; Fibonacci; Test Yourself #3. Using Mathematical

Break Time!


Check out this, it's the coolest.
It's called recursion.
Check out this, it's the coolest.
It's called recursion.
Good Sir, I Cannot Draw
Good Sir, I Cannot Draw

Break Time!


```
void func() {
  if (true) {
    func();
  }
  else {
    return; // can't stop the func
  }
}
```

```
void aboutThatBase(int x) {
  if (x == 0) {
    cout << "BASE CASE!!!" << endl;
  }
  else {
    // Call aboutThatBase (recursively)
    return aboutThatBase(x - 1);
  }
}
```

Break Time!

We'll start again soon.

Minute Exercise: The Cost of Recursion

- Let's say we want to reverse an array of size `N`
- For this **recursive** implementation of `reverse`…
  - **Time**: Find the number of swaps performed.
  - **Memory**: Find the *largest* number of stack frames for `reverse` on the stack at any given time.

```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {
  if (left < right) {
    reverse(left + 1, right - 1);
    int temp = *left;
    *left = *right;
    *right = temp;
  }
}
```

Solution: The Cost of Recursion

- Let's say we want to reverse an array of size **N**
- For this **recursive** implementation of `reverse`…
  - **N/2** swaps are performed. **O(N) - linear time**
  - **N/2+1** stack frames are needed. **O(N) - linear space**

```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {
  if (left < right) {
    reverse(left + 1, right - 1);
    int temp = *left;
    *left = *right;
    *right = temp;
  }
}
```

Minute Exercise: The Cost of Iteration

- Let's say we want to reverse an array of size `N`
- For this **iterative** implementation of `reverse`…
  - **Time**: Find the number of swaps performed.
  - **Memory**: Find the *largest* number of stack frames for `reverse` on the stack at any given time.

```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {
  while (left < right) {
    int temp = *left;
    *left = *right;
    *right = temp;
    ++left;
    --right;
  }
}
```

Solution: The Cost of Iteration

- Let's say we want to reverse an array of size `N`
- For this **iterative** implementation of `reverse`…
  - `N/2` swaps are performed. **O(N) - linear time**
  - `1` stack frame is needed. **O(1) - constant space**

```
// EFFECTS: Reverses the array starting at 'left'
//          and ending at (and including) 'right'.
void reverse(int *left, int *right) {
  while (left < right) {
    int temp = *left;
    *left = *right;
    *right = temp;
    ++left;
    --right;
  }
}
```

Tail Recursion

- Consider these two implementations of reverse:

```
void reverse(int *left, int *right) {
  if (left < right) {
    reverse(left + 1, right - 1); // reverse middle
    int temp = *left;
    *left = *right; // swap first/last elements
    *right = temp;
  }
}
```

```
void reverse(int *left, int *right) {
  if (left < right) {
    int temp = *left;
    *left = *right; // swap first/last elements
    *right = temp;
    reverse(left + 1, right - 1); // reverse middle
  }
}
```

This version is tail recursive, because the recursion comes at the end. But why do we care?

Review: Function Calls

- To call a function, the computer must…
  1. Evaluate the actual **arguments** to the function.
  2. Make a **new and unique** invocation of the called function
     - Push a **stack frame** (space for formal parameters and locals)
     - Pass **parameters** (actual → formal)
  3. Pause the **original** function

Active flow

  4. Run the **called** function
  5. Return some value (optional)

Passive flow

  6. Start the **original** function where it left off
  7. Destroy the stack frame. (In simple cases, do nothing.)

Tail Call Optimization (TCO)

- A function call is a **tail call** if it is the very last thing in its containing function.
- The calling function has no pending work to do after a tail call (in the passive flow), **so its stack frame isn't needed anymore.**

- Some compilers are able to recognize tail calls and optimize them.
  - Just overlay the new stack frame over the memory used for the old one.
  - In g++, **-02** includes TCO

- TCO usually has a much bigger impact for recursive functions.
  - Why?

Another Version of Factorial

```
int fact(int n, int resultSoFar) {
  if (n == 0) { // BASE CASE
    return resultSoFar;
  }
  else { // RECURSIVE CASE
    return fact(n - 1, n * resultSoFar);
  }
}

int main() {
  return fact(5, 1); // Seed result with identity
}
```

- Simulate the code in Lobster:
  - Where does the multiplication happen now?
  - Why do we call the extra parameter `resultSoFar`?
  - How is the base case different from before?
  - Why is `1` passed in for the `resultSoFar` from `main`?

Two Implementations of `fact`

Recursive

```
int fact(int n) {
  if (n == 0) { // BASE
    return 1;
  }
  else { // RECURSIVE
    return n * fact(n - 1);
  }
}
```

Linear Space

- Computation is done **after** the "repetition".
- Multiplication happens during **passive flow**.
- We need to keep track of each stack frame with each value of n.

Tail Recursive

```
int fact(int n, int soFar) {
  if (n == 0) { // BASE
    return soFar;
  }
  else { // RECURSIVE
    return fact(n - 1,
                n * soFar);
  }
}
```

Constant Space

- Computation is done **before** the "repetition".
- Multiplication happens in **active flow**.
- Nothing happens in the passive flow, so we do not need the stack frame to stick around.

Making fact Tail Recursive…

- TCO can take a recursive algorithm from linear to constant space complexity as long as it only makes tail calls.
  - We say a function is "**tail recursive**" if and only if ALL the recursive calls it makes are tail calls.
- From some (but not all) tail-recursive functions, we need to seed the recursion with an initial value.
  - We use a helper function to retain the same interface.

```
static int fact_helper(int n, int resultSoFar) {
  if (n == 0) { // BASE
    return resultSoFar;
  } else { // RECURSIVE
    return fact_helper(n - 1, n * resultSoFar);
  }
}
int fact(int n) {
  return fact_helper(n, 1); // Seed with identity
}
```