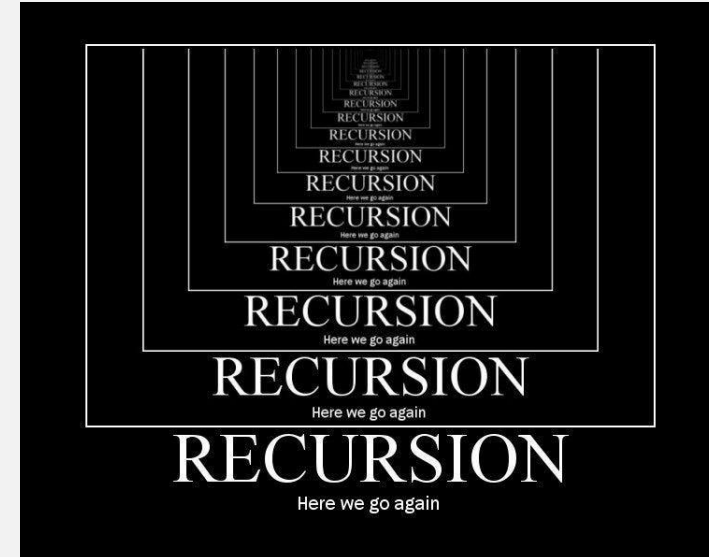UMass Boston Computer Science
CS450 **High Level Languages**
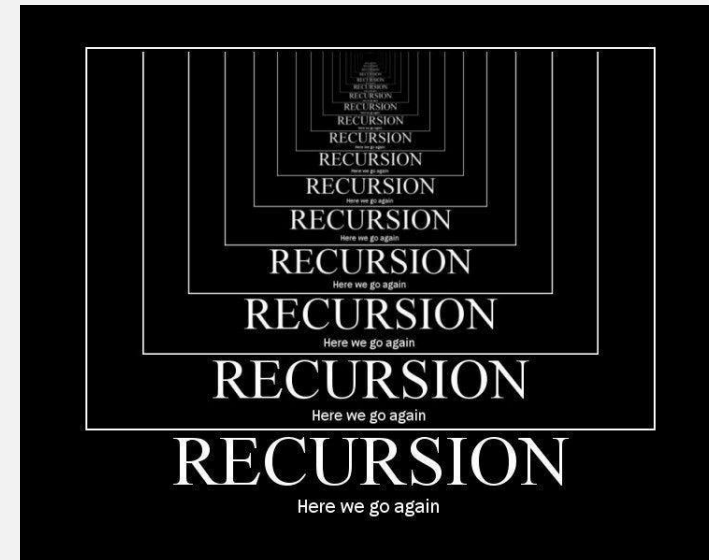# Generative Recursion
Tuesday, March 11, 2025

Logistics

- HW 5 in
  - ~~Due: Tues 3/11 11am EST~~

- HW 6 in
  - Due: Tues 3/25 11am EST (2 weeks)

- Reminder: Spring Break next week!
  - No lecture

(improper base case!)

Two-Argument Templates

- Sometimes … a fn must **process** <u>two</u> **arguments simultaneously**

- **This template** should **combine templates of** <u>both</u> **args**
  - (This is only possible if the data defs are simple enough)

```
;; hit?: Ball Ball -> Boolean?
;; evaluates to true if the two Balls have overlap
```

(2-argument) TEMPLATE

```
(define (hit? b1 b2)
        ...  (ball-x b1) (ball-y b1)  ...
        ...  (ball-x b2) (ball-y b2)  ...
```

Ball **b1** template

Ball **b2** template

Must combine all these pieces
together somehow …

```
;; hit?: Ball Ball -> Boolean?
;; evaluates to true if the two Balls have overlap
```

```
(define (hit? b1 b2)
       ...  (ball-x b1) (ball-y b1)  ...
       ...  (ball-x b2) (ball-y b2)  ...    )
```

Must combine all these pieces
together somehow …

```
;; hit?: Ball Ball -> Boolean?
;; evaluates to true if the two Balls have overlap
```

```
(define (hit? b1 b2)
 ... (dist (ball-x b1) (ball-y b1)
           (ball-x b2) (ball-y b2)) ...    )
```

Must combine all these pieces
together somehow …

```
;; hit?: Ball Ball -> Boolean?
;; evaluates to true if the two Balls have overlap
```

```
(define (hit? b1 b2)
  (< (dist (ball-x b1) (ball-y b1)
           (ball-x b2) (ball-y b2))
                              ...                    ))
```
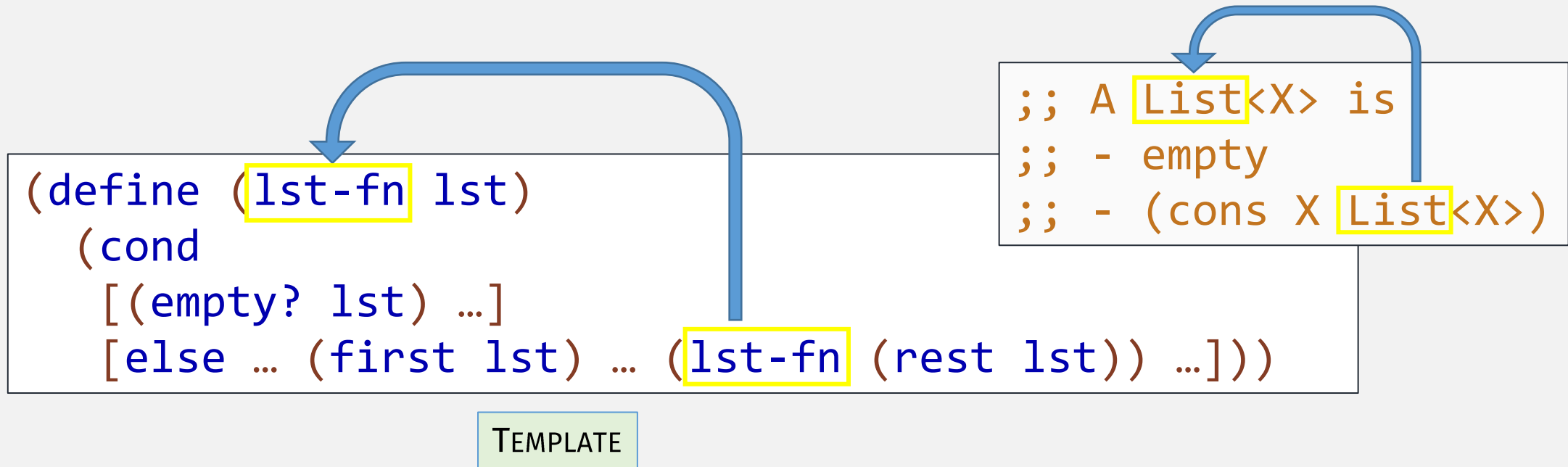
```
;; hit?: Ball Ball -> Boolean?
;; evaluates to true if the two Balls have overlap
```

```
(define (hit? b1 b2)
  (< (dist (ball-x b1) (ball-y b1)
           (ball-x b2) (ball-y b2))
     (+ (ball-radius b1) (ball-radius b2)))))
```

(assuming coordinates are center)

Recursion review

- Most **recursion** is **structural** (i.e., comes from **data definitions**)!

```
(define (lst-fn lst)
  (cond
   [(empty? lst) …]
   [else … (first lst) … (lst-fn (rest lst)) …]))
```

TEMPLATE

```
;; A List<X> is
;; - empty
;; - (cons X List<X>)
```
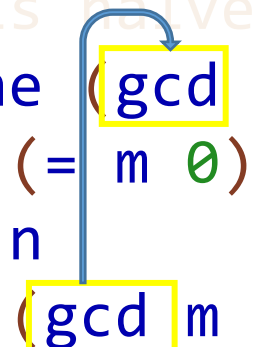
A Different Kind of Recursion!

- <u>Not all</u> recursion is **structural** (i.e., comes from **data definitions**)!

A Different Kind of Recursion!

- **Not all recursion** is **structural** (i.e., comes from **data definitions**)!

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid's algorithm
;; termination argument:
;; m is halved (at least) every ... via modulo fn)
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m)))
```

What template is this following??

A Different Kind of Recursion!

- **Non-structural recursion** (i.e., doesn't come from **data definitions**) is called **generative recursion**

- no template? … requires **Termination Argument**
  - Explains why the function terminates – because **recursive call is "smaller"!**

```
;; gcd : Nat Nat -> Nat
;; computes greatest common divisor, using Euclid's algorithm
;; termination argument:
;; m is halved (at least) every iteration (via mo
(define (gcd n m)
  (if (= m 0)
      n
      (gcd m (modulo n m))))
```

But how to develop an algorithm like this??

Recursive call must be on "smaller" version of the problem

Generative (non-structural) Recursion Design Recipe

1. Name, Signature
2. Description
   - Must include **Termination Argument**
3. Examples
   - Even more important now!
4. **Code** (No structural template, but can use a "general" template)

5. Tests

Generative (non-structural) Recursion Design Recipe

1. Name, Signature
2. Description
   - Must include **Termination Argument**
3. Examples
   - Even more important now!
4. **Code** (No structural template, but can use a "general" template)
   a) Break problems into <u>smaller</u> problems to (recursively) **solve**
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!
5. Tests

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a "general" template)
   a) Break problems into smaller problems to (recursively) **solve**
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a "general" template)
   a) **Break problems into smaller problems to (recursively) solve**
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!

```
;; genrec-algo: ??? -> ???
;; termination argument: recursive calls are "smaller" bc …

(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
            (genrec-algo (create-smaller-1 problem))
            …
            (genrec-algo (create-smaller-n problem)))]))
```

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a "general" template)
   a) Break problems into smaller problems to (recursively) solve
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!

```
;; genrec-algo: ??? -> ???
;; termination argument: recursive calls are "smaller" bc …
(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
              (genrec-algo (create-smaller-1 problem))
              …
              (genrec-algo (create-smaller-n problem)))]))
```

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a "general" template)
   a) Break problems into smaller problems to (recursively) solve
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!

```
;; genrec-algo: ??? -> ???
;; termination argument: recursive calls are "smaller" bc ...

(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
             (genrec-algo (create-smaller-1 problem))
             ...
             (genrec-algo (create-smaller-n problem)))]))
```

Generative (non-structural) Recursion Design Recipe

4. **Code** (No structural template, but can use a "general" template)
   a) Break problems into smaller problems to (recursively) **solve**
   b) Determine how to combine smaller solutions
   c) "trivially solvable" problem is base case!

```
;; genrec-algo: ??? -> ???
;; termination argument: recursive calls are "smaller" bc …
(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
            (genrec-algo (create-smaller-1 problem))

            …

            (genrec-algo (create-smaller-n problem)))]))
```

GenRec Template Generalizes Structural!

```
(define (lst-fn lst)
  (cond
    [(empty? lst) …]
    [else … (first lst) … (lst-fn (rest lst)) …]))
```

- Trivial solution = data def base case
- Recursive smaller problem = data def smaller piece
- Left to figure out "Combining" pieces

```
;; genrec-algo: ??? -> ???

(define (genrec-algo problem)
  (cond
    [(trivial? problem) (solve-easy problem)] ;; base case
    [else (combine-solutions
            (genrec-algo (create-smaller-1 problem))

            …

            (genrec-algo (create-smaller-n problem)))]))
```

(Functional) Quicksort

```
;; smaller-than: ListofInt Int -> ListofInt
;; Returns a list containing elements of given list
;; that are less than the given int
```

```
(check-equal?
  (smaller-than (list 1 3 4 5 9) 4)
                (list 1 3))
```
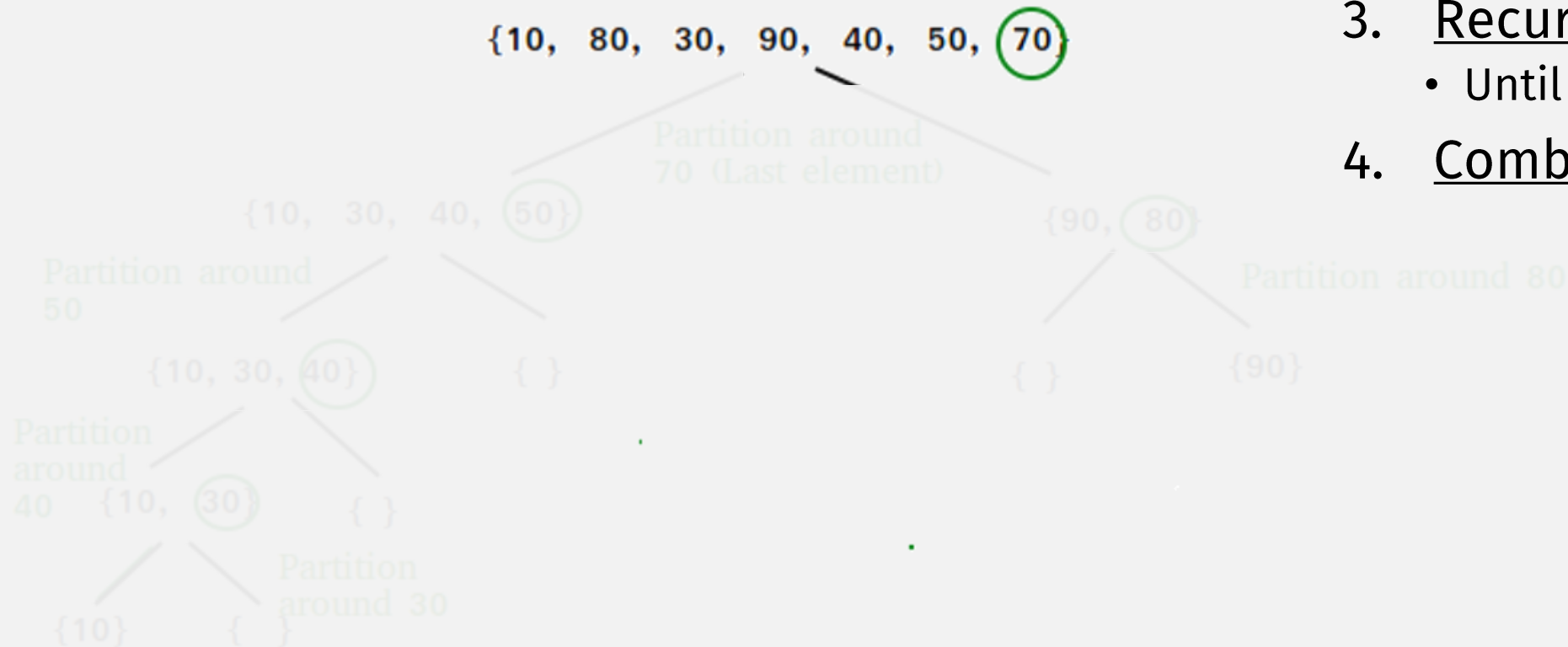
```
;; larger-than: ListofInt Int -> ListofInt
;; Returns a list containing elements of given list
;; that are greater than the given int
```

```
(check-equal?
  (greater-than (list 1 3 4 5 9) 4)
                (list 5 9))
```

```
;; qsort: ListofInt -> ListofInt
;; sorts the given list of ints in ascending order
(define (qsort lst)
  (define pivot (random lst))
  (append (qsort (smaller-than lst pivot))
          (list pivot)
          (qsort (greater-than lst pivot)))))
```

Quicksort overview ("divide and conquer")

1. <u>Choose</u> "pivot" element

2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot

3. <u>Recurse</u> on smaller lists
   - Until base case

4. <u>Combine</u> small solutions

{10, 80, 30, 90, 40, 50, 70}

Partition around 70 (Last element)

{10, 30, 40, 50}   {90, 80}

Partition around 50   Partition around 80

{10, 30, 40}   { }   { }   {90}

Partition around 40

{10, 30}   { }

Partition around 30

{10}   { }

Gen Rec Example: (functional) quicksort

1. <u>Choose</u> "pivot" element
2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot
3. <u>Recurse</u> until base case
4. <u>Combine</u> small solutions

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
   [(trivial? problem) (solve-easy lst)] ;; base case
   [else
    (define pivot (first lst))
    (combine-solutions
      (qsort (smaller-problem-1 lst))
               …
            (qsort (smaller-problem-n lst)))]))
```

Gen Rec Example: (functional) quicksort

1. <u>Choose</u> "pivot" element
2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot
3. <u>Recurse</u> until base case
4. <u>Combine</u> small solutions

```
;; qsort: List<Int> -> List<Int>
;; terminati
;; r                              vot
  (c
[          problem) (solve-easy lst)] ;
[else
                      (first lst))
    (combine-solutions
      (qsort (filter (curry > pivot) lst))
        …
      (qsort (filter (curry <= pivot) lst)))])))
```

Function "arithmetic"!

Result is a function!

```
(curry f arg1)
      =
(lambda (arg2) (f arg1 arg2))
```

Curry = "partial apply"

```
(curry > pivot)
       =
(lambda (x) (> pivot x))
```

"less than"

"greater than"

Gen Rec Example: (functional) quicksort

1. <u>Choose</u> "pivot" element
2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot
3. <u>Recurse</u> until base case
4. <u>Combine</u> small solutions

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
   [(trivial? problem) (solve-easy lst)] ;; base case
   [else
    (define pivot (first lst))
    (combine-solutions
      (qsort (filter (curry > pivot) lst))
                  …
           (qsort (filter (curry <= pivot) lst)))]))
```

"less than"

"greater than"

Gen Rec Example: (functional) quicksort

1. <u>Choose</u> "pivot" element
2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot
3. <u>Recurse</u> until base case
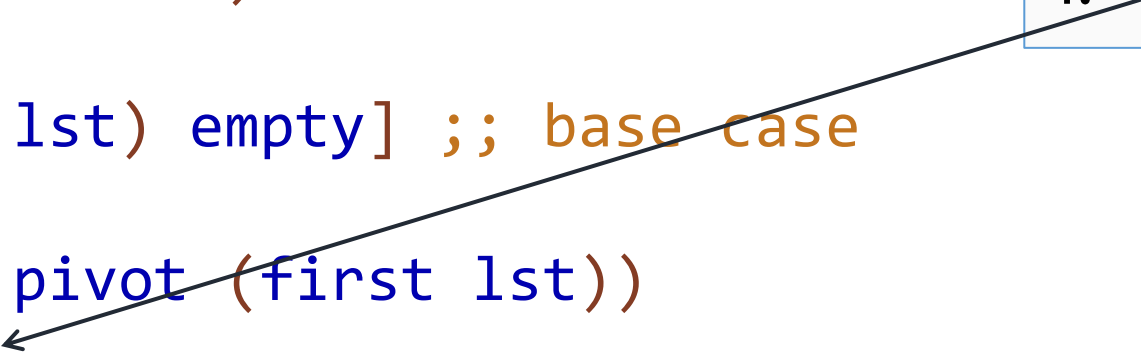4. <u>Combine</u> small solutions

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
    [(empty? lst) empty] ;; base case
    [else
      (define pivot (first lst))
      (combine-solutions
        (qsort (filter (curry > pivot) lst))
                    …
                (qsort (filter (curry <= pivot) lst)))]))
```

Gen Rec Example: (functional) quicksort

1. <u>Choose</u> "pivot" element
2. <u>Partition</u> into smaller lsts:
   - < pivot
   - >= pivot
3. <u>Recurse</u> until base case
4. <u>Combine</u> small solutions

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls drop at least pivot
(define (qsort lst)
  (cond
  [(empty? lst) empty] ;; base case
  [else
   (define pivot (first lst))
   (append
     (qsort (filter (curry > pivot) lst))
     (cons pivot
            (qsort (filter (curry <= pivot) lst)))]))
```

Gen Rec Example: (functional) quicksort

```
;; qsort: List<Int> -> List<Int>
;; termination argument:
;; recursive calls "smaller" bc at least one item dropped (pivot)
(define (qsort lst)
  (cond
   [(empty? lst) empty] ;; base case
   [else
    (define pivot (first lst))
    (append
     (qsort (filter (curry > pivot) lst))
     (cons pivot
           (qsort (filter (curry <= pivot) lst))))]))
```

Interlude: Recursion vs Iteration

- **Recursive** functions have a self-reference

```python
def factorialUsingRecursion(n):
    if (n == 0):
        return 1;


    # recursion call
    return n * factorialUsingRecursion(n - 1);
```

- **Iterative** code typically use a loop

```python
def factorialUsingIteration(n):
    res = 1;


    # using iteration
    for i in range(2, n + 1):
        res *= i;


    return res;
```

Recursion vs Iteration: Which is "Better"?

▶ **Recursive vs. Iterative Solutions**

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least O(n) memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the tradeoffs with your interviewer.

Cracking the Coding Interview, Ch8

r/ **r/learnprogramming** • 11 yr. ago

[Best Practices] Recursion. Why is it generally avoided and when is it acceptable?

≡ stack**overflow**

Are recursive methods always better than iterative methods in Java?

Recursion vs Iteration: Conventional Wisdom

Strengths:

- Iteration can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory.
- Iteration is faster and more efficient than recursion.
- It's easier to optimize iterative codes, and they generally have polynomial time complexity.
- They are used to iterate over the elements present in data structures like an array, set, map, etc.
- If the iteration count is known, we can use *for* loops; else, we can use *while* loops, which terminate when the controlling condition becomes false.

Iteration

Iteration is <u>good</u> with *non-recursive data*

Weaknesses:

- In loops, we can go only in one direction, i.e., we can't go or transfer data from the current state to the previous state that has already been executed.
- It's difficult to traverse trees/graphs using loops.
- Only limited information can be passed from one iteration to another, while in recursion, we can pass as many parameters as we need.

Iteration is <u>bad</u> with *recursive data*!

Recursion better when **accumulators** are needed

Recursion vs Iteration: Conventional Wisdom

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
    - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand. ☑
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- It requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- Is difficult to optimize a recursive code, and they generally have higher time complexity an iterative codes due to overlapping subproblems.

Recursion better when **accumulators** are needed

Use recursion with *recursive data*!

Recursion is slow

Recursion is slow

Recursion is slow

Recursion is slow

Investigate:
Is recursion is slower??

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Recursion vs Iteration: In Racket

```
;; sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (sum-to x)
  (if (zero? x)
      x
      (+ x (sum-to (sub1 x))))))
```

Conclusion?
Recursion is slower?

WAIT!
Racket does not have "for" loops

```
(define BIG-NUMBER 999999)

(time (sum-to BIG-NUMBER))
; cpu time: 202 real time: 201 gc time: 156
```

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Recursion vs Iteration: In Racket

Conclusion?
Recursion is <u>not</u> slower than iteration?

equivalent

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result)))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in Racket is just a **macro** (i.e., "syntactic sugar") for a recursive function

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Tail Calls

In computer science, a **tail call** is a subroutine call performed as the final action of a procedure. If the target of a tail is the same subroutine, the subroutine is said to be **tail recursive**, which is a special case of direct recursion. **Tail recursion** (or **tail-end recursion**) is particularly useful, and is often easy to optimize in implementations.

Tail calls can be implemented without adding a new stack frame to the call stack.

Recursion vs Iteration: In Racket

Racket Recursion

Conclusion?
Recursion is <u>not</u> slower than iteration?

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
   (if (zero? x)
        result
        (iterative-sum-to (sub1 x) (+ x result)))))
```

Tail-recursive function

Tail-call (does not add to stack)

(Tail) recursion <u>is</u> iteration!

Recursion vs Iteration: Under the Hood

- It makes sense that **recursion** and **iteration** are **equivalent …**
  - Recursive call compiles to:
    - **JUMP** instruction
  - Loop compiles to:
    - **JUMP** instruction!

- … except in languages that make them not equivalent!
  - i.e., languages that push extra stack frames that are not needed

Tail-Calls in Other Languages

- **Most functional languages** (RACKET, HASKELL, ERLANG, F#) **implement proper tail calls**
  (no extra stack frame)

- **Some languages require an explicit annotation**
  - CLOJURE: `recur`
  - SCALA: `@tailrec`

- **Some languages** (JAVASCRIPT) **have it** (ECMASCRIPT 6)**, but don't have it**

- **Most imperative languages don't properly implement tail calls**
  (they add an unnecessary stack frame)
  - PYTHON, JAVA, C#, GO

Guido Got It Backwards

Wednesday, April 22, 2009

Tail Recursion Elimination

I recently posted an entry in my Python History blog on the origins of Python's functional features. A side remark about not supporting tail recursion elimination (TRE) immediately sparked several comments about what a pity it is that Python doesn't do this, including links to recent blog entries by others trying to "prove" that TRE can be added to Python easily. So let me defend my position (which is that I don't *want* TRE in the language). If you want a short answer, it's simply unpythonic. Here's the long answer:

First, as one commenter remarked, TRE is incompatible with nice stack traces: when a tail recursion is eliminated, there's no stack frame left to use to print a traceback when something goes wrong later. This will confuse ... ently wrote something recursive (the recursion isn't obviou... ce printed), and makes debugging hard. Providing an optio... eems wrong to me: Python's default is and should always b... be maximally helpful for debugging. This also brings me to the next issu...

About Me

B Guido van Rossum

Python's BDFL

View my complete profile

Blog Archive

▶ 2022 (2)

▶ 2019 (1)

Wrong!

Equivalent to saying:
"every **for** loop iteration should push a stack frame!"

Proper tail calls is about eliminating stack frames that <u>shouldn't be there in the first place!</u> (because it's just iteration!)

Tail Calls as Loops

C

```c
// this function calculates factorial modulo prime
int factorial(int store, int num, int prime) {
    if (num < 1) {
        // Base case
        return store;
    }

    return factorial((store%prime * num%prime)%prime, num - 1, prime);
}
```
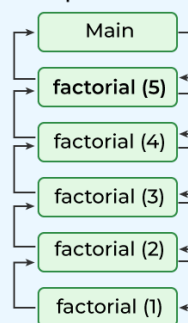
C

```c
// Program to find factorial of a number n modulo p
int factorial(int n, int prime)
{
    if (n <= 1) {
        // base case
        return 1;
    }

    return (n * factorial(n - 1, prime) % prime) % prime;
}
```

Tail call

Non-tail-call

```c
int main()
{
    // Calling for the factorial of 5
    int result = factorial(5, 1000000007);
    printf("%d\n", result);
    return 0;
}
```
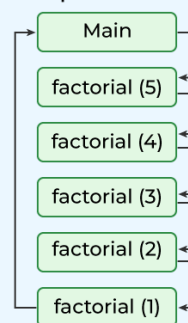
Tail Call Optimisation in C

Without tail call Optimisation

- Main
- factorial (5)
- factorial (4)
- factorial (3)
- factorial (2)
- factorial (1)

With tail call Optimisation

- Main
- factorial (5)
- factorial (4)
- factorial (3)
- factorial (2)
- factorial (1)

```c
int main()
{
    // Calling for the factorial of 5
    int result = factorial(5, 1000000007);
    printf("%d\n", result);
    return 0;
}
```

Faster, less memory

Slower, more memory

https://www.geeksforgeeks.org/tail-call-optimisation-in-c/

51

Proper Tail Calls in JavaScript

Proper Tail Calls (PTC) is a new feature in the ECMAScript 6 language. This feature was added to facilitate recursive programming patterns, both for direct and indirect recursion. Various other design patterns can benefit from PTC as well, such as code that wraps some functionality where the wrapping code directly returns the result of what it wraps. Through the use of PTC, the amount of memory needed to run code is reduced. In deeply recursive code, PTC enables code to run that would otherwise throw a stack overflow exception.

https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/

Feature name

proper tail calls (tail call optimisation)

Not supported in V8 (Chrome) or SpiderMonkey (Firefox)! 😡

https://compat-table.github.io/compat-table/es6/

Recursion vs Iteration: Conclusion

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
    - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand.
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- It requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- It is difficult to optimize a recursive code, and they generally have higher time complexity than iterative codes due to overlapping subproblems.

Recursion better when **accumulators** are needed

Recursion is (usually) easier to read

Use recursion with *recursive data*!

Recursion is slower …

… in languages that <u>choose</u> to make it slower!

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration