Hacking the Coding Interview

Gregory Marton

https://bitbucket.org/gregory_marton/coding-interview/src

Who are you? (until 5:10)

A name to call you by today? Course+Year?

One question / what are you most here to learn?

The agenda:

5:10–5:45 What to expect, prep strategies.

5:45–6:20 Getting un-stuck.

6:20-7pm Non-coding discussions.

Please ask questions throughout.

Purpose

~~Hazing~~

~~GitHub~~

Do we want to work with each other?

Resume is verifiable? Currently qualified for role? Future potential?

What are your strengths?

Do you think systematically, with attention?

Can we have a clear, interesting conversation?

Self Care

Most important asset: confident, positive attitude.

Interviews are asymmetric. Shake it off. Have fun.

Take a break: not a stress test.

Sleep. Eat. Smile. Stretch. Be kind. Be grateful.

Interview Types

Technical phone interview: broad, shallow

On-site calibrated coding: narrower, deep

On-site calibrated design: big-picture, organizing

Standardized knowledge interview: IT

Pair programming or mini-project

Behavioral interview

Lunch "interview"

...

Interview Types

Technical phone interview: broad, shallow

On-site calibrated coding: narrower, deep

On-site calibrated design: big-picture, organizing

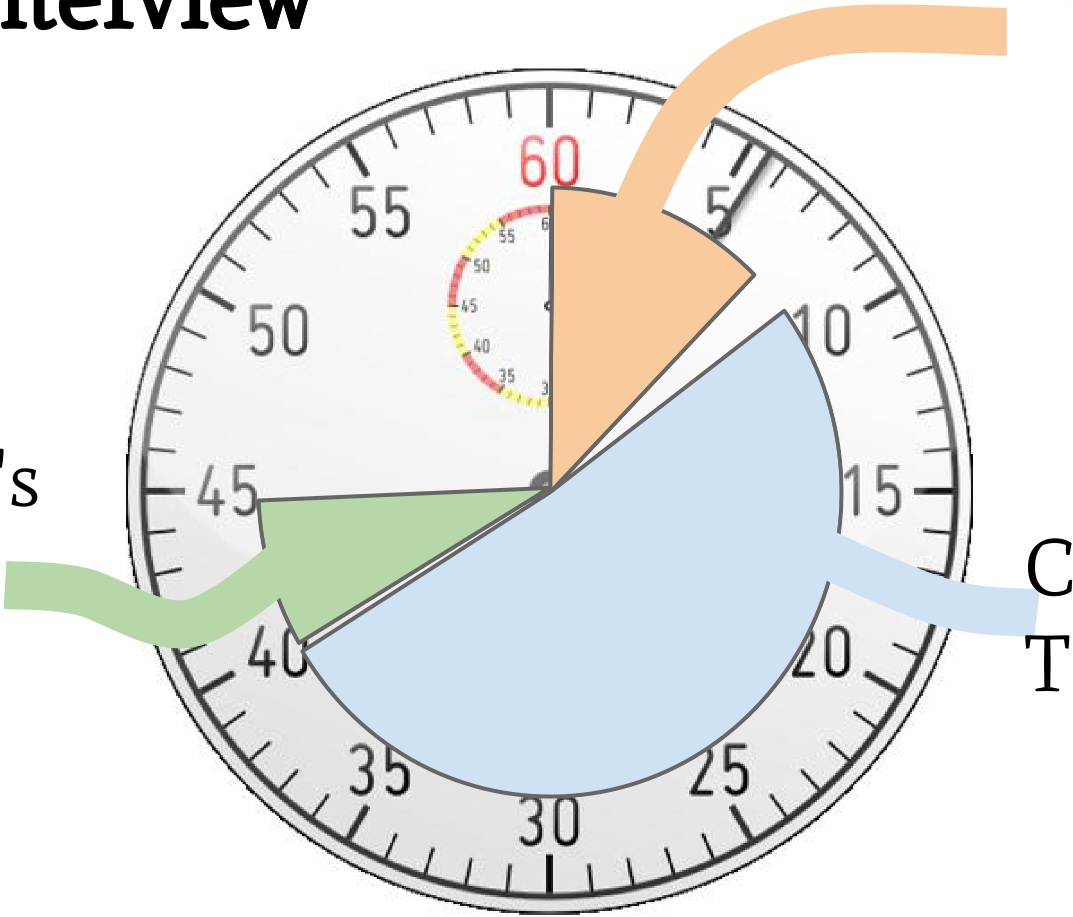
Standardized knowledge interview: IT

Pair programming or mini-project

Behavioral interview

Lunch "interview"

Each Interview



Résumé-
driven

Candidate's
Questions

Calibrated
Technical

The Right Answer™

Thought process

Skill in communicating it

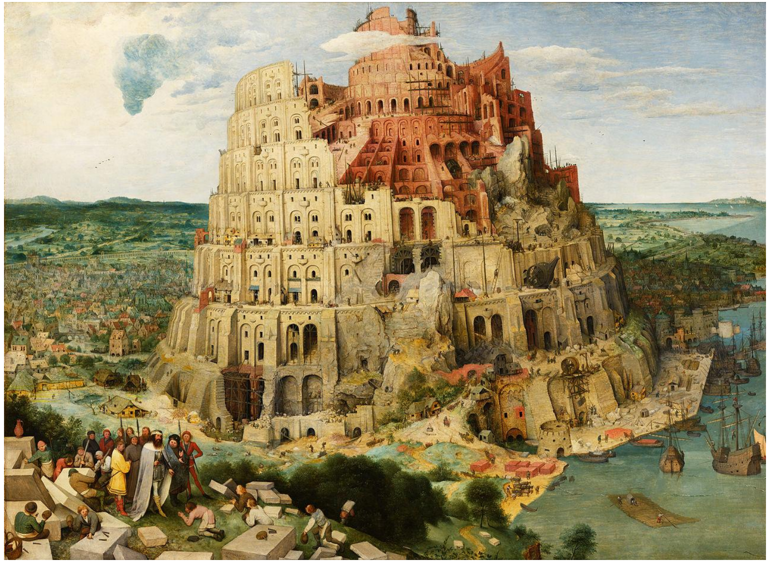
Exploring and Comparing Solutions

Coding!

"Generalist"

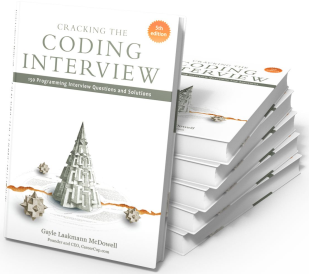
Productive conversations with everyone.

Understands implications of code at many levels.



Practice

Ruby
PROGRAMMING
Language


CRACKING THE
CODING
INTERVIEW
150 Programming Interview Questions and Solutions
5th edition
Gayle Laakmann McDowell
Founder and CEO, CareerCup.com

Getting un-Stuck

What you're thinking → ? → Hints, analogies.

How do you learn something new?

Q&A (until 5:45)

How do interviewers choose questions?

How important is your experience/school/degree?

How to balance work + life + studying?

I'm nervous/shy.

I'm a specialist.

I've heard that question!

I know I'll get rejected.

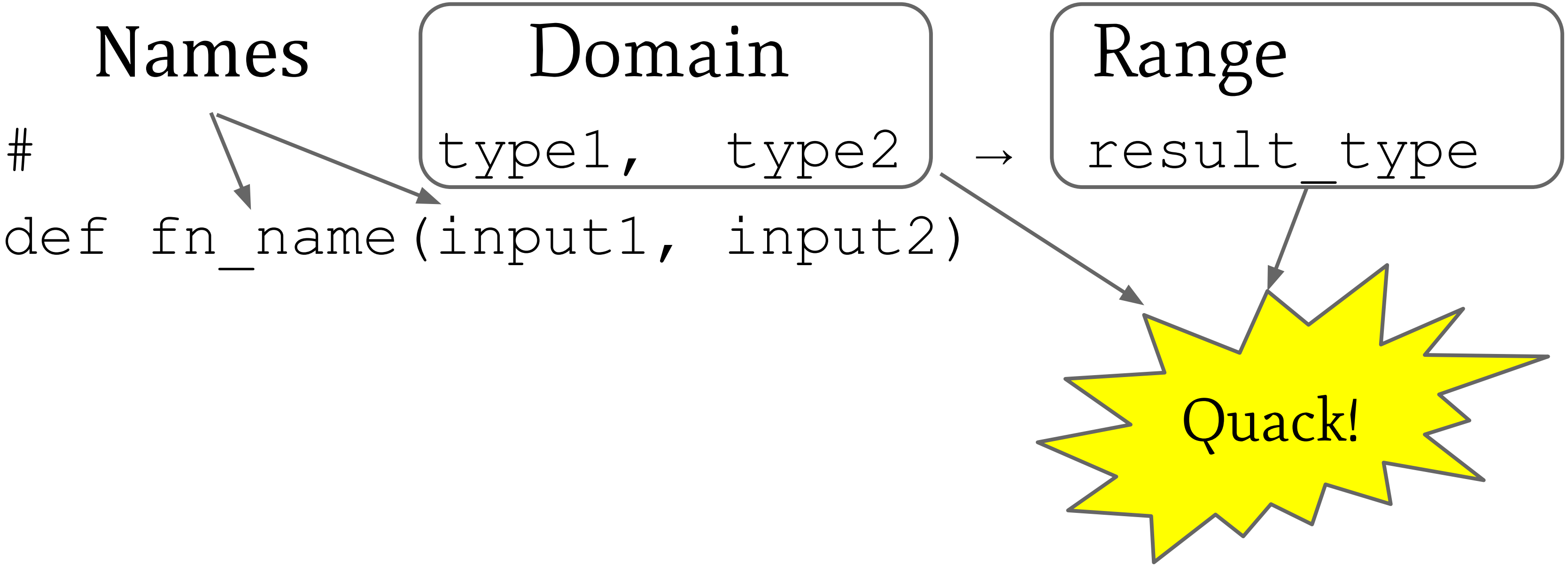
I'm rusty.

Interviewer was rude!

I have a disability.

______________________?

Function Signatures / Contracts



Names

Domain

Range

```
#                type1,  type2  →  result_type
def fn_name(input1, input2)
```

Signatures/Contracts Practice

Problem Statement	Function Name	Input Names	Input Types	Result Type
Is a binary tree full?				
In a list of numbers, find the closest pair.				
Reverse a string, in place.				
Given two sorted arrays, find the common elements.				
Play "24": You get 4 digits; find math operations that get them to 24. E.g. given (2, 3, 8, 4), find (3 * (8 / 2 + 4)).				

Signatures/Contracts Practice

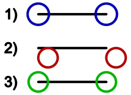
Problem Statement	Function Name	Input Names	Input Types	Result Type
Is a binary tree full?	is_full	tree	Binary Tree	Boolean
In a list of numbers, find the closest pair.	closest_pair	choices	List of Numbers	Pair<Number, Number>
Reverse a string, in place.	reverse	str	String	Modifies input!
Given two sorted arrays, find the common elements.	common_elements	a, b	List, List Items mutually comparable.	List
Play "24": You get 4 digits; find math operations that get them to 24. E.g. given (2, 3, 8, 4), find (3 * (8 / 2 + 4)).	twenty_four	digits	Set of Integers	Tree or Stack of digits and operations

Let's Code!

❑ Volunteers please!

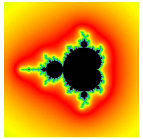
1. Function Signatures

2. Examples



3. Assumptions

4. Algorithms

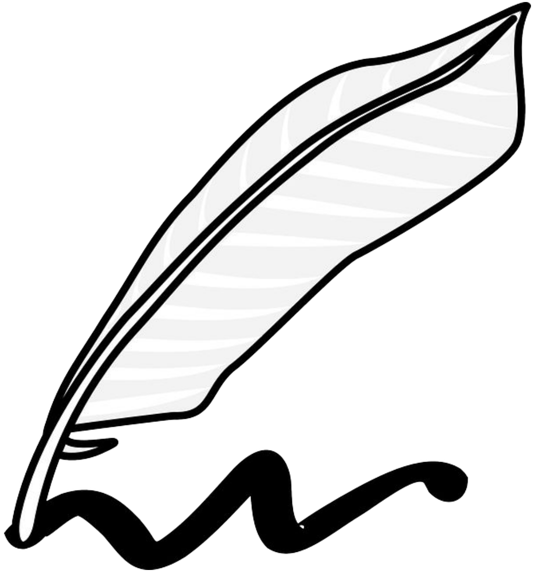


5. Code!
6. Checking back, relaxing assumptions.

Let's Code!

3.2 Make a stack class with push, pop, and min.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

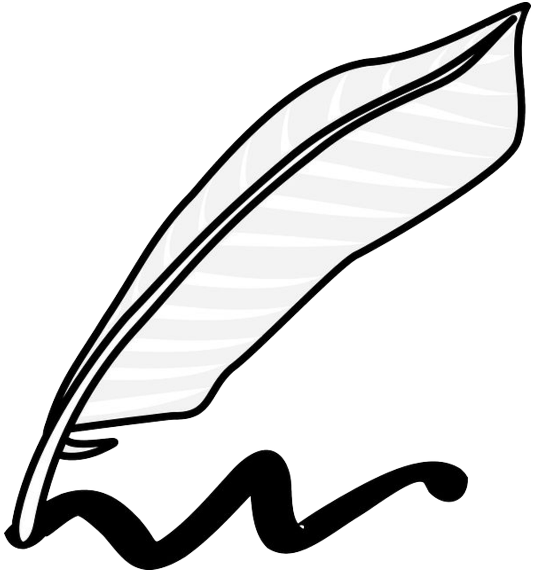


3.2 Make a stack class with push, pop, and min.

```
class MinStack
    # min  : MinStack -> value
    # pop  : MinStack! -> value
    # push : MinStack!, value -> MinStack
end
```

Use or extend existing stack code

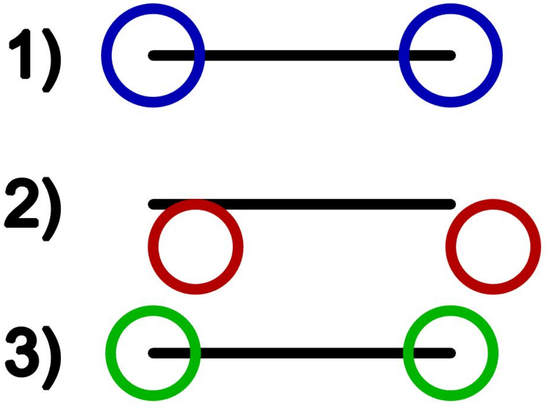
— do not invent your own!



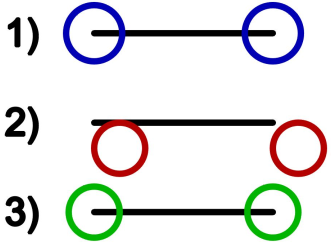
Let's Code!

3.2 Make a stack class with push, pop, and min.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



3.2 Make a stack class with push, pop, and min.



[]	push(3) -> [3] min -> nil pop -> [], nil
[3]	push(3) -> [3, 3] push(3).min -> 3 push(1) -> [3, 1] min -> 3 pop -> [], 3
[3, 1]	push(5) -> [3, 1, 5] min -> 1 pop -> [3], 1 pop then min -> 3
[3, 1, 5]	push(1) -> [3, 1, 5, 1] min -> 1 pop -> [3, 1], 5

Let's Code!

3.2 Make a stack class with push, pop, and min.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

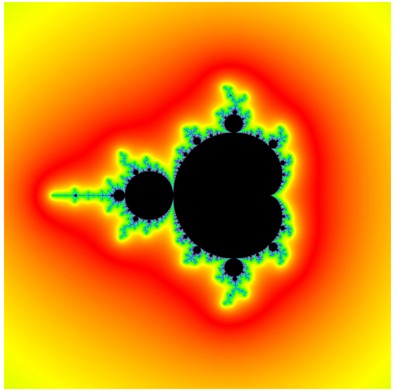
3.2 Make a stack class with push, pop, and min.

Time	Is it okay for min to take O(n) time? **[No! Try for O(1).]** Is it okay for push to take O(log$_2$n) time? **[No! Try for O(1).]**
Space	Can/should we use extra storage? **[Yes, if you want it, take up to O(n) space.]**
Domain and Range	Are values always numbers? [No. Store any value, or describe constraints.] Is there a minimum possible minimum value? [No.]
Special Values	Can the stack be empty? **[Yes.]** Can the stack contain nil? [Up to you. Why or why not?] Are the inputs required to be distinct? [Up to you. Why do you want it?]
Behavior	Do you ever want to pop the minimum value? [No. Why would that be hard?] Do you ever want to pop multiple values? [No. Why would that be hard?]

Let's Code!

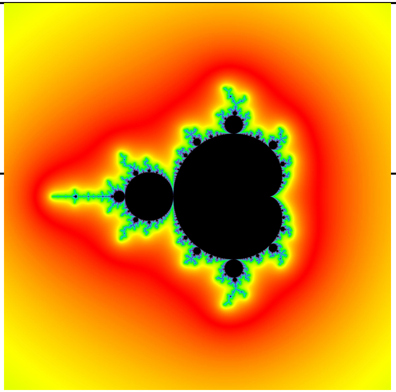
3.2 Make a stack class with push, pop, and min.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



3.2 Make a stack class with push, pop, and min.

[]	push(3) -> [3] min -> nil pop -> [], nil
[0]	push(0) -> [0, 0] push(0).min -> 0 push(1) -> [0, 1] min -> 0 pop -> [], 0
[0, -1]	push(5) -> [0, -1, 5] min -> -1 pop -> [0], -1 pop then min -> 0
[0, -1, 5]	push(1) -> [0, -1, 5, 1] min -> -1 pop -> [0, -1], 5

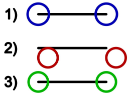


Let's Code!

3.2 Make a stack class with push, pop, and min.

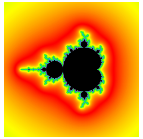
1. Function Signatures

2. Examples



3. Assumptions

4. Algorithms

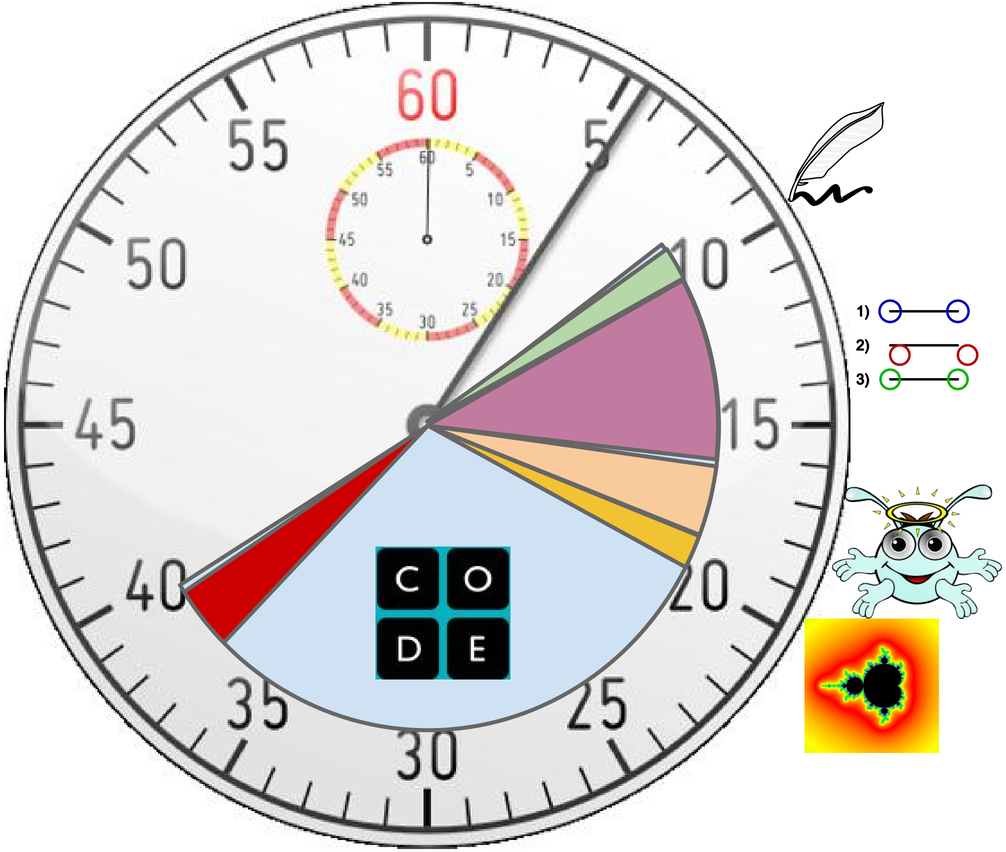


5. Code!
6. Checking back, relaxing assumptions.

```ruby
class MinStack
  # Assumes values are Comparable.

  def initialize()
    @values = []
    @minima = []
  end

  def min()   # -> value (or nil)
    return @minima.last
  end
def push!(value)  # value -> Stack
    prev_min = self.min()
    @values << value
    @minima << ((prev_min and
                 prev_min < value) ?
                 prev_min : value)
    return self
  end

  def pop!()  # -> value (or nil)
    @minima.pop
    return @values.pop
  end
end  # class
```

```ruby
class MinStack
  EntryPair = Struct.new(:value, :stk_min)
  def initialize()
    @entries = []
  end

  def min(next_value = nil)
    return next_value if @entries.empty?
    last_min = @entries.last.stk_min
    return last_min unless next_value
    return [last_min, next_value].min
  end

  def push(value)
    @entries << EntryPair.new(
         value, self.min(value))
    return self
  end

  def pop()
    return @entries.pop.value
        unless @entries.empty?
  end
end  # class
```

Timing



Quick Tutorial: Binary Trees

```
        a
      /   \
    b       c
   /       / \
  d       f   g
```

```
c.class  # → Tree
a.name   # → "a"
a.left   # → b
c.right  # → g
f.parent # → c
d.left   # → nil
b.right  # → nil
a.parent # → nil
```

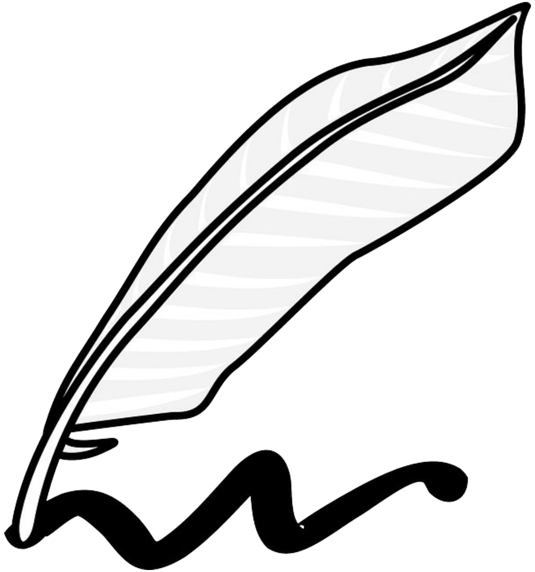
Note: not a binary search tree!

(Volunteer to explain?)

Let's Code!

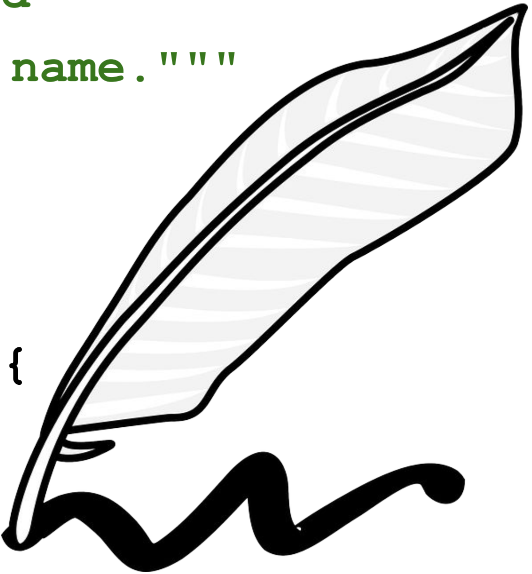
4.7 Given a binary tree and two node ids, find their closest common ancestor.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



Given a binary tree and two node ids, find their first common ancestor.

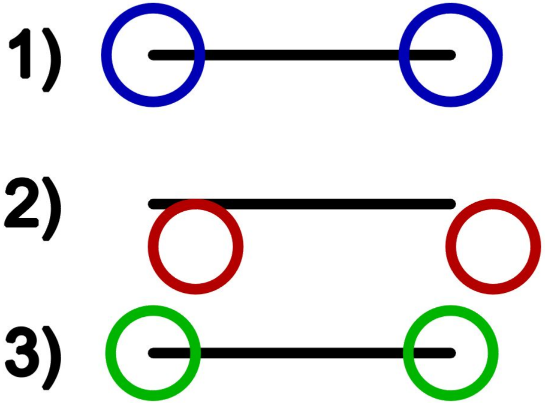
```
def first_common_ancestor(tree, p, q):
    """tree has .left, .right, and .name, and
        p and q are names to find. Returns a name."""

template<typename T>
bool BinaryTree::LCA(
    const pair<T, T>& targets, T* ancestor) {

public static Node ClosestAncestor(
    Node root, Node a, Node b) {
```



Let's Code!

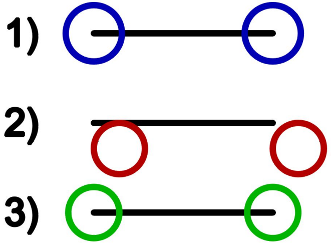
4.7 Given a binary tree and two node ids, find their first common ancestor.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



Given a binary tree and two node ids, find their first common ancestor.

nil	a, a → nil a, b → nil
a	a, a → a a, b → nil b, a → nil
a / \\ b c	a, a → a a, b → a b, c → a b, b → b c, a → a
a / \\ b c / / \\ d f g	a,a → a (systematically all identities) a,b → a a,d → a c,f → c a,g → a c,g → c (descendants) c,d → a b,f → a b,g → a d,f → a (aunts) f, g → c (lower siblings) c, e → nil (not found) (all symmetries by test helper)



Let's Code!

4.7 Given a binary tree and two node ids, find their first common ancestor.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

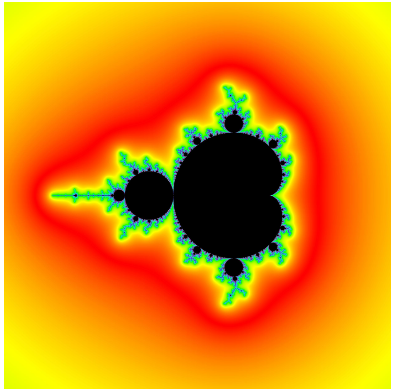
Given a binary tree and two node ids, find their first common ancestor.

Time	Does the tree have parent links? **[If too hard, then yes.]** Do we have an index from id to node reference? **[If too hard, then yes.]** Otherwise this will be O(n) worst case no matter what, right? If it was a search tree, I could do better. Better data structure? Are queries common vs. inserts+deletes?
Space	Can/should we use extra storage? [Do we get any advantage? Use case? Is caching worthwhile? If so, what kind?]
Domain and Range	Like a search tree, can we know anything about the children of a node? Do we only care about ancestors a certain distance away? [Generic nodes.]
Special Values	Can the tree be empty? **Yes.** Guaranteed to find the ids? **No.** Is the tree balanced? **Yes.** What type are node ids? Fast to compare? `nil` legal? **Your choice. Yes. No.** Are the input ids required to be distinct? **No.** Are the ids in the tree all distinct? **[If too easy, then no.]**

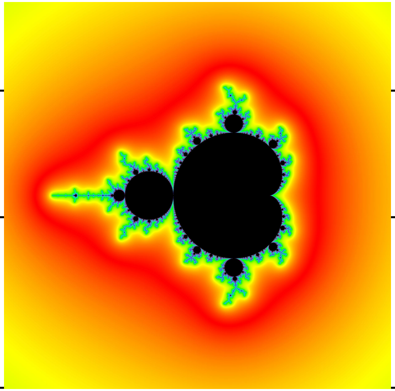
Let's Code!

4.7 Given a binary tree and two node ids, find their first common ancestor.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



Given a binary tree and two node ids, find their first common ancestor.



nil	a, a → nil a, b → nil
a	a, a → a a, b → nil b, a → nil

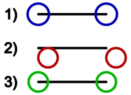
a
   / \\
  b   c
	a, a → a a, b → a b, c → a
	b, b → b c, a → a

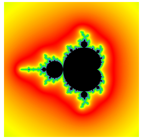
a / \\ b c / / \\ d f g	a,a → a (systematically all identities) a,b → a a,d → a c,f → c a,g → a c,g → c (descendants) c,d → a b,f → a b,g → a d,f → a (aunts) f, g → c (lower siblings) c, e → nil (not found) (all symmetries by test helper)

Let's Code!

4.7 Given a binary tree and two node ids, find their first common ancestor.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

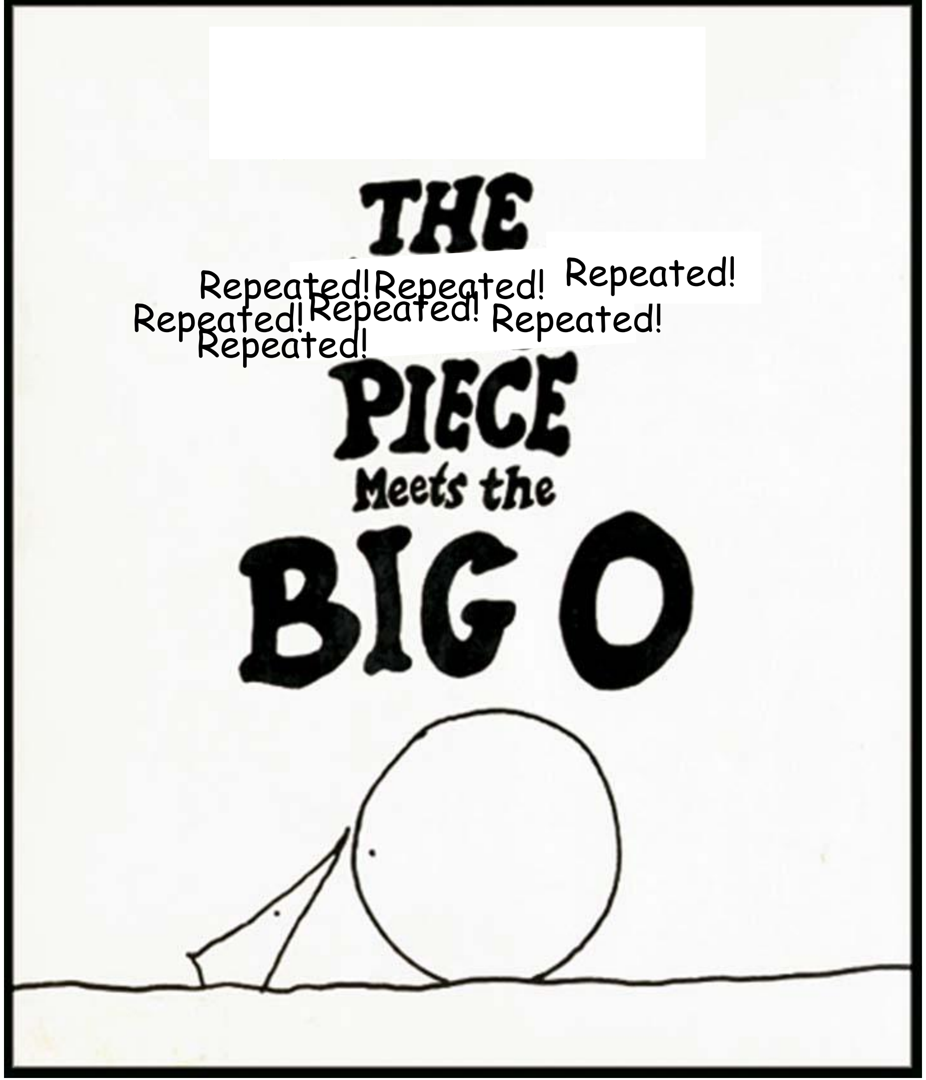




```ruby
def common_prefix_non_solution(p, q)
  p_parents = find_with_parents(p).map{|i| i.name}
  q_parents = find_with_parents(q).map{|i| i.name}
  p_node = p_parents.pop
  q_node = q_parents.pop
  while p_node != q_node
    p_node = p_parents.pop
    q_node = q_parents.pop
  end
  return p_node
end
```

```ruby
def fca_common_suffix(p, q)
  p_parents = find_with_parents(p).map{|i| i.name}
  q_parents = find_with_parents(q).map{|i| i.name}
  common_parent = nil
  p_node = p_parents.shift
  q_node = q_parents.shift
  while (p_node == q_node and
         not p_parents.empty? and
         not q_parents.empty?)
    common_parent = p_node
    p_node = p_parents.shift
    q_node = q_parents.shift
  end
  return p_node == q_node ? p_node : common_parent
end
```

```
def fca_with_a_set(p, q)
  p_parents = Set.new(find_with_parents(p).map{|i| i.name})
  q_parents = find_with_parents(q).map{|i| i.name}
  q_parents.reverse.each do |parent|
    if p_parents.include?(parent)
      return parent
    end
  end
  return nil
end
```



Repeated!Repeated! Repeated!
Repeated! Repeated! Repeated!
Repeated!

With apologies to Shel Silverstein!

Sets: Listlike, Unordered, no Duplicates

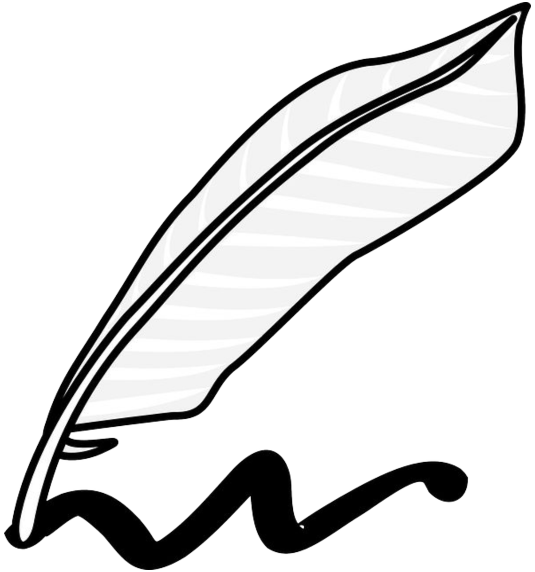
```
s = Set.new(["a", "b"])
s << "c"
s.add("a")              #  no effect.  already there.
s.include?("c")         #  →  true
s.include?(3)           #  →  nil
s - ["b"]               #  →  the set with "a" and "c".
s += ["d"]              #  →  the set {"a", "b", "c", "d"} (math font)
```

`s.each` works as usual   `s.first` returns an element.

Let's Code!

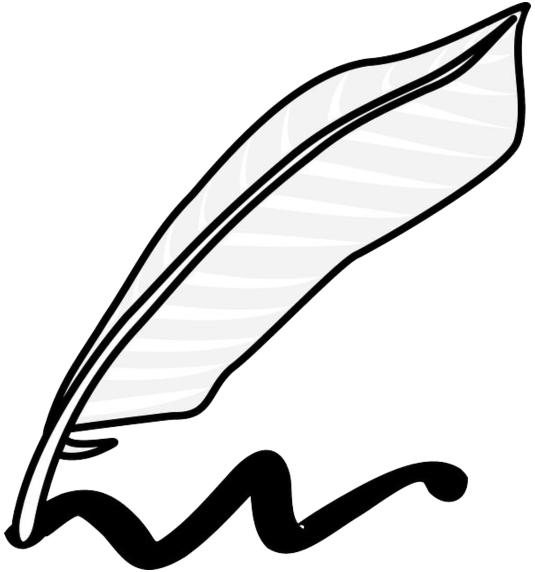
9.4 Return all the subsets of a set.

1. Function Signature
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.



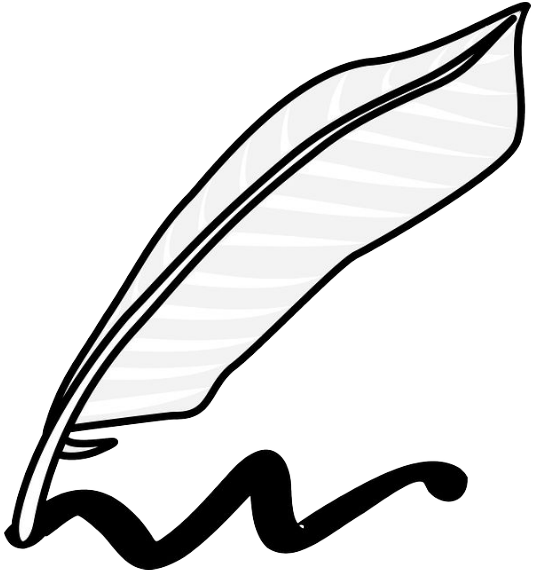
9.4 Return all the subsets of a set.

```
subsets : Set -> Set  ?
```



9.4 Return all the subsets of a set.

```
~~subsets : Set -> Set  ?~~

subsets : Set -> List<Set> ?
```

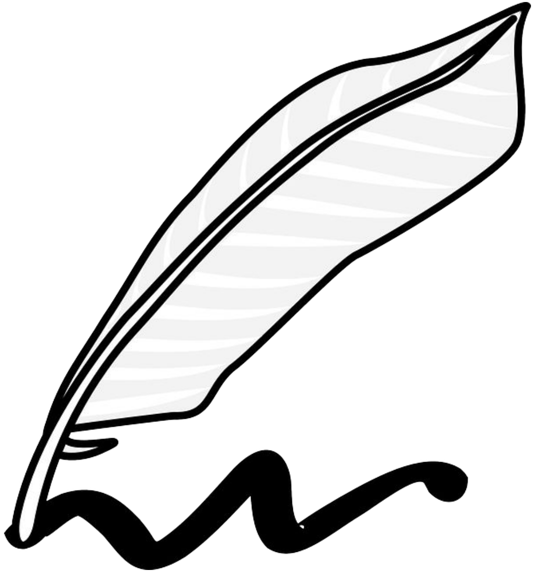


9.4 Return all the subsets of a set.

~~subsets : Set -> Set ?~~

~~subsets : Set -> List<Set> ?~~

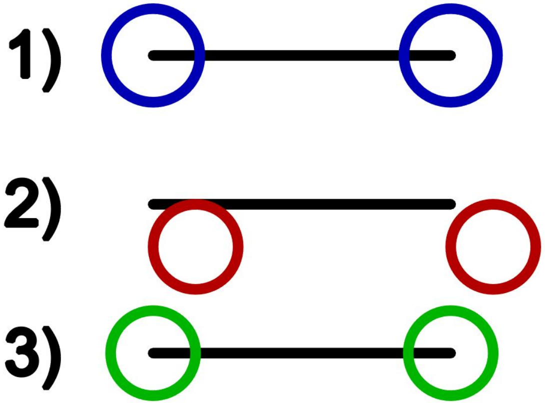
subsets : Set -> Set<Set> ?



Let's Code!

9.4 Return all the subsets of a set.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

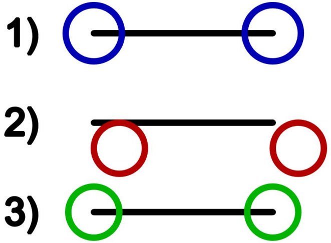


9.4 Return all the subsets of a set.

{}	{{}}
{a}	{{}, {a}}
{a,b}	{ {}, {a}, {b}, {a,b} }

{a,b,c}	{ {}, {a}, {b}, {a,b}
	{c}, {a,c}, {b,c}, {a,b,c} }

{a,b,c,d}	{ {}, {a}, {b}, {a,b}, {c}, {a,c}, {b,c}, {a,b,c}
	{d}, {a,d}, {b,d}, {a,b,d}, {c,d}, {a,c,d}, {b,c,d}, {a,b,c,d} }



Let's Code!

9.4 Return all the subsets of a set.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

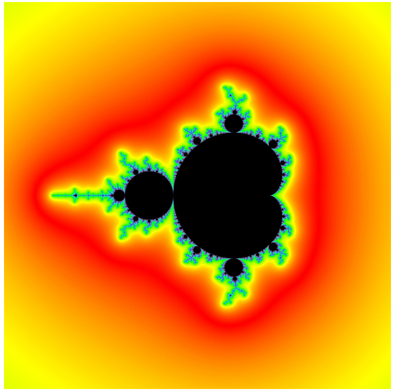
9.4 Return all the subsets of a set.

Time	Can we assume O(1) set operations? [**Yes**, wherever that's reasonable.]
Space	Can/should we use extra storage? **[Be careful of extra copying.]**
Domain and Range	Should the empty set always be in the result? [Yes, that's fine.]
Special Values	Can the initial set be empty? [Yes.] Can the initial set be nil? [It's fine to assume that it's a set.]

Let's Code!

9.4 Return all the subsets of a set.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

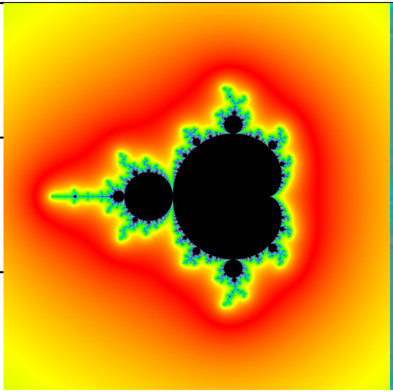


9.4 Return all the subsets of a set.

{}	{{}}
{a}	{{}, {a}}
{a,b}	{ {}, {a}, {b}, {a,b} }

{a,b,c}	{ {}, {a}, {b}, {a,b}
	{c}, {a,c}, {b,c}, {a,b,c} }

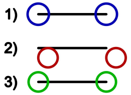
{a,b,c,d}	{ {}, {a}, {b}, {a,b}, {c}, {a,c}, {b,c}, {a,b,c}
	{d}, {a,d}, {b,d}, {a,b,d}, {c,d}, {a,c,d}, {b,c,d}, {a,b,c,d} }

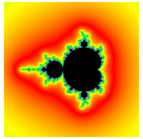


Let's Code!

9.4 Return all the subsets of a set.

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.

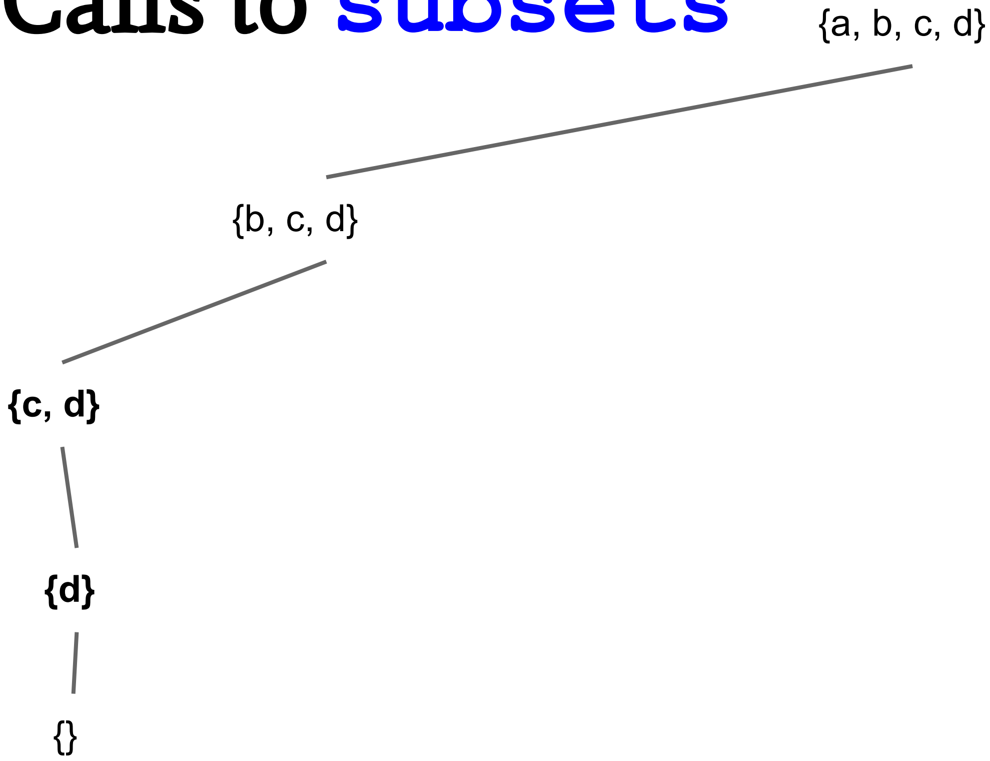




```ruby
def subsets(source)
  # Set -> Set of Sets
  result = Set.new([Set.new])
  if source.size > 0
    # Set has #first from Enumerable.
    some_element = source.first
    without = subsets(source - [some_element])
    result.merge(without)
    without.each do |subset|
      result.add(subset + [some_element])
    end
  end
  return result
end
```

```ruby
def subsets(source)
  # Set -> Set of Sets
  result = Set.new([Set.new])
  # Taking away each possible way to make it smaller:
  source.each do |some_element|

    without = subsets(source - [some_element])
    result.merge(without)
    without.each do |subset|
      result.add(subset + [some_element])
    end
  end
  return result
end
```

Calls to `subsets`



{a, b, c, d}

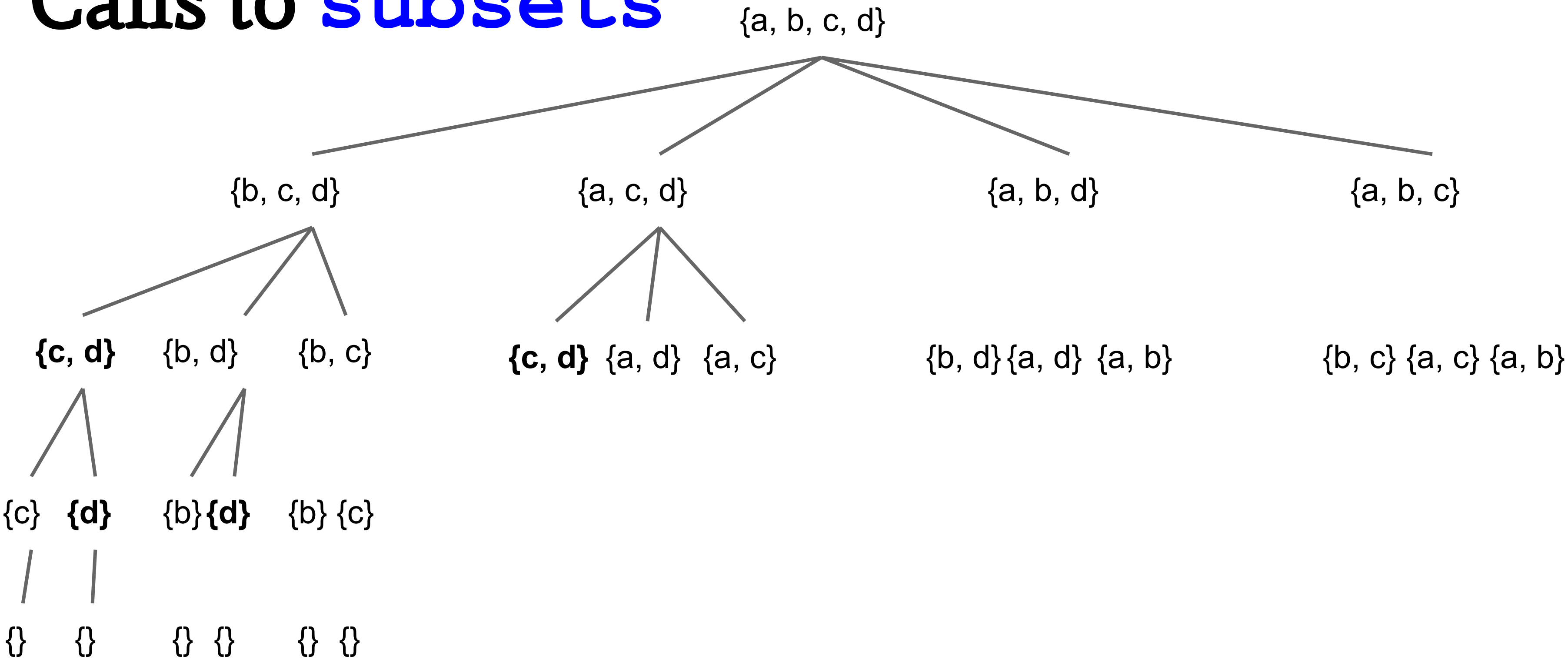
{b, c, d}

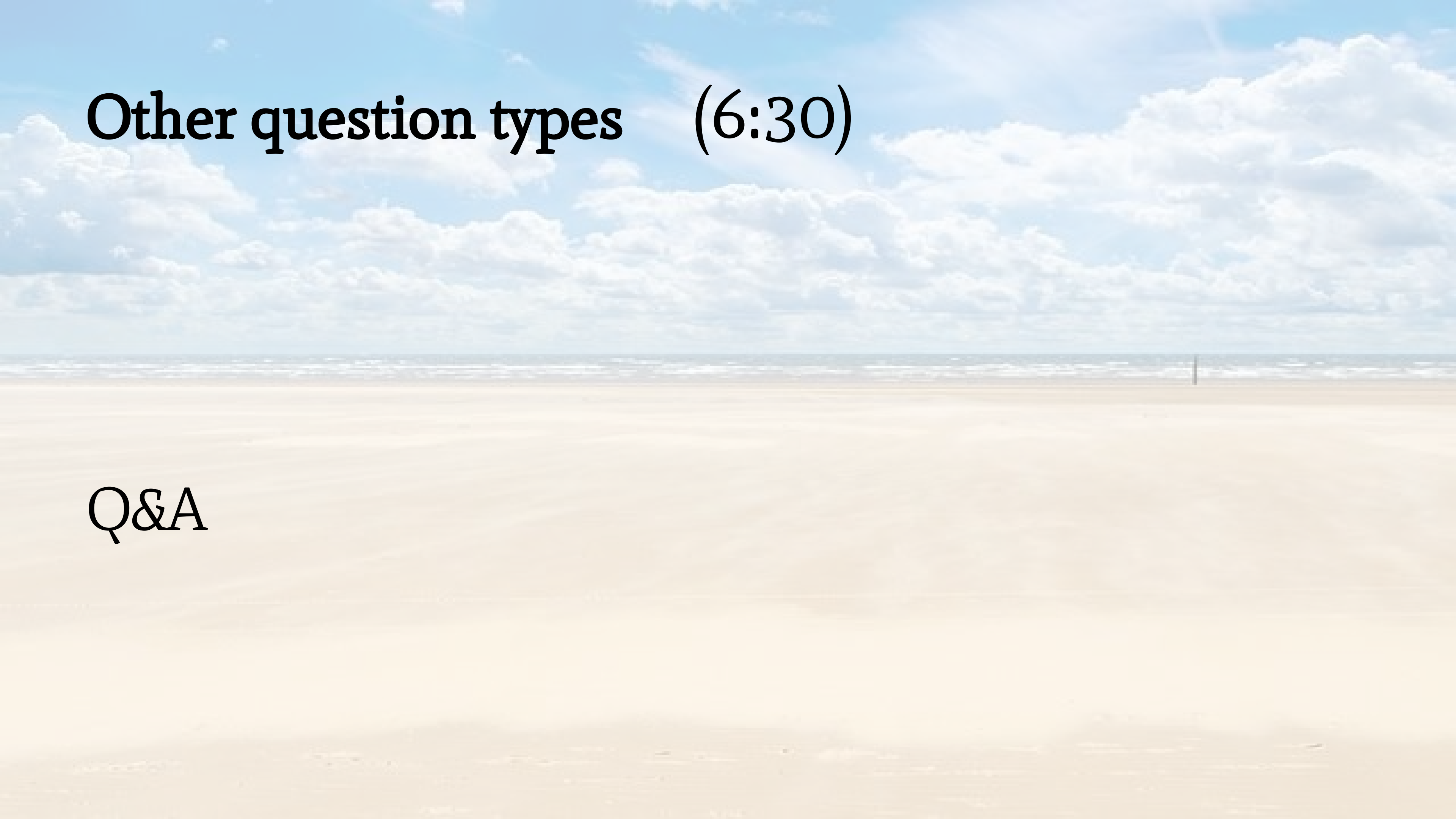
{c, d}

{d}

{}

Calls to `subsets`





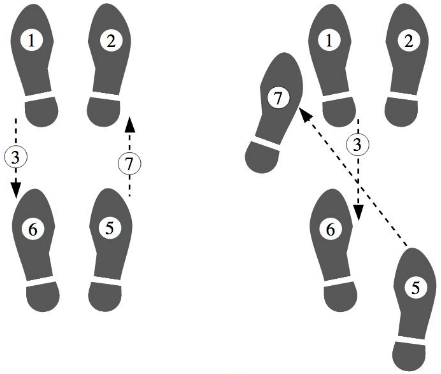
Other question types (6:30)

Q&A

Tell me about this project.

Goal

Breakdown into major pieces,



Your Contribution,

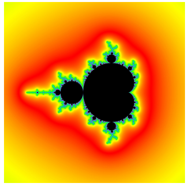
Results.

Exploratory / Design

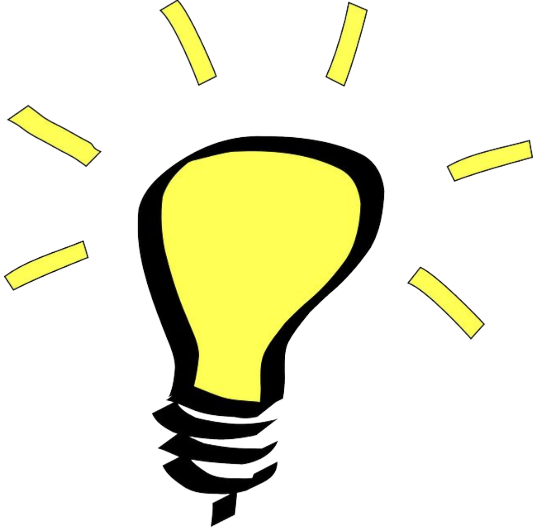
A good way to sort a million numbers?

Copy a file to a million machines.

Design {interviewer's fav app/site/feature]



If you could do anything...



What are you most excited to do?

A fresh idea? Teach me something!

Where does your motivation come from?

Do you know our company's goals & pain points?

Prior Challenges

A difficult bug to track down.

Details → Experience.

A tough interpersonal situation.

How do you handle adversity?

Resilience? Integrity? Creativity? Leadership?

Problematic

How did you pay for college?*

If I were to look at the web history section of your browser, what would I learn about you that isn't on your resume?*

* From How Google Works by Eric Schmidt, Jonathan Rosenberg, and Alan Eagle, 2014.

What do you enjoy doing in your free time?

Problematic → Turn it Around

What do you hope to learn by asking that?
Why is that important to you?

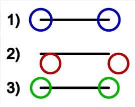
Are you looking for examples of initiative?

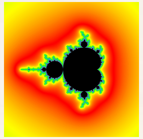
Imagine with them: I think some of your colleagues might answer … because …

Q&A

The Design Recipe

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.





```ruby
def fca_anydist_helper(p, q)
  empty = FcaIntermediateResult.new(false, false, nil)
  found_p = @name == p
  found_q = @name == q
  return FcaIntermediateResult.new(true, true, @name) if found_p and found_q
  left_result = @left ? @left.fca_anydist_helper(p, q) : empty
  return left_result if left_result.fca
  right_result = @right ? @right.fca_anydist_helper(p, q) : empty
  return right_result if right_result.fca
  found_p |= (left_result.found_p or right_result.found_p)
  found_q |= (left_result.found_q or right_result.found_q)
  return FcaIntermediateResult.new(true, true, @name) if found_p and found_q
  return FcaIntermediateResult.new(found_p, found_q, nil)
end
def fca_anydist(p, q)
  return fca_anydist_helper(p, q).fca
end
```

Résumés

~~History~~ → Ad

Recruiter: impressiveness, clarity of fit + purpose.

Interviewer: deep, interesting conversation.

1 page great conversations interesting specifics

(+ a10tion to detail)

Peer Review — Your time.

7-7:30 What to expect, prep strategies.

7:30-8 Guided coding practice.

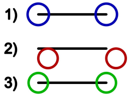
8:05-8:30

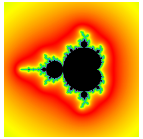
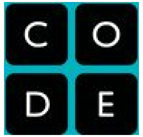
- Other question types
- Fielding bad questions
- q&a

8:30-9 Peer review of resumes and code.

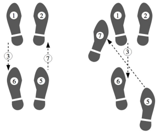
Peer Review — Your time.

Résumés: 1 page ad, recruiter+coworker

Programs: 

, names!

Your story: 

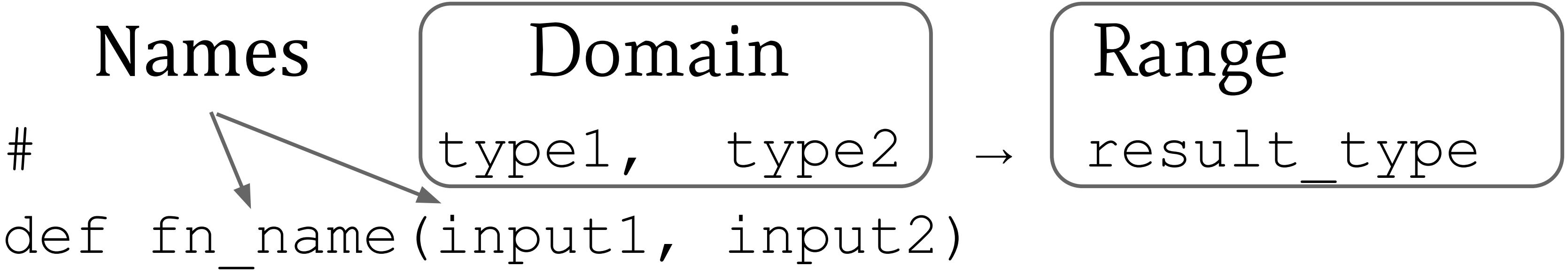
Handout

Hacking the Coding Interview

Gregory Marton

https://bitbucket.org/gregory_marton/coding-interview/src

Function Signatures / Contracts



Names

Domain

Range

```
#           type1,  type2  →  result_type
def fn_name(input1, input2)
```

Code a contract/signature on the board quickly.

Types help you talk about constraints.

Signatures/Contracts Practice

Problem Statement	Function Name	Input Names	Input Types	Result Type
Is a binary tree full?				
In a list of numbers, find the closest pair.				
Reverse a string, in place.				
Given two sorted arrays, find the common elements.				
Play "24": You get 4 digits; find math operations that get them to 24. E.g. given (2, 3, 8, 4), find (3 * (8 / 2 + 4)).				

Quick Tutorial: Binary Trees

```
     a
   /   \
  b     c
 /     / \
d     f   g
```

```
c.class  # → Tree
a.name   # → "a"
a.left   # → b
c.right  # → g
f.parent # → c
d.left   # → nil
b.right  # → nil
a.parent # → nil
```

Note: not a binary search tree!

(Volunteer to explain?)

Sets: Listlike, Unordered, no Duplicates

```
s = Set.new(["a", "b"])
s << "c"
s.add("a")           #  no effect.  already there.
s.include?("c")        #  →  true
s.include?(3)          #  →  nil
s - ["b"]          #  →  the set with "a" and "c".
s += ["d"]         #  →  the set {"a", "b", "c", "d"} (math font)
```

`s.each` works as usual   `s.first` returns an element.

Practice Problems

http://www.careercup.com/

Levels: http://projecteuler.net/ http://www.rankk.org/

Help people: http://stackoverflow.com/

Daily/Weekly:

http://programmingpraxis.com/

http://www.reddit.com/r/dailyprogrammer

The Design Recipe

http://htdp.org/ Book: How To Design Programs

1. Function Signatures
2. Examples
3. Assumptions
4. Algorithms
5. Code!
6. Checking back, relaxing assumptions.