Ruby Coding Interview.

- Complexity Theory

└ Computational complexity is about measuring how well your algorithm scales with regards to its input size.

└ eg. comparing 2 arrays. & return same items.

└ sol 1) nested loop

```
Intersection(a,b)
  foreach(x:a)      } Nested Loop
    foreach(y:b)
      if x==y then add(intersection, x)
```

↳ CPU = $O(a*b)$; a & b = sizes of the arrays.

$= O(n^2)$, if $a=b=n$

↳ an algorithm with a runtime complexity of big O n-squared, or ~~this~~ this algo is quadratic

↳ one of complexity classes

Constant class : $O(1)$ offers the best scalability.

Exponential class : $O(k^n)$ " worst scalability.

Log $O(\log n)$

Linear $O(n)$

Log Linear $O(n \log n)$

Quadratic $O(n^2)$

better ↓ worse

sol 2 : sorted

└ Intersection (a,b)

```
  sort(a) ⟶ $O(a \log a)$
  sort(b) ⟶ $O(b \log b)$
  merge Intersection(a,b) → $O(a+b)$
```

↳ CPU : $O(n \log n)$ * choose the worse among

or $O(n \log n + n \log n + n) = O(2(n \log n) + n) = O(n \log n)$

★ Start by thinking about

The easiest & quickest solution

↓

work out the runtime & space complexity

↓

Stop. Take a step back, think of ways to improve it.

Ask yourself 3 Qs :

1) Can I re-encode the inputs to make the algorithm faster? (eg sort(a), sort(b))

2) Can I use a better data structure?

3) Can I fit the solution in a common design paradigm? (eg. divide & conquer)

4) Equilibrium Problem

- Given an array, get a pivot pt which split the array into 2 ~~equal~~ parts with minimum $\overset{\text{absolute}}{\vee}$ differences.

- eg. $\overset{\longleftarrow \quad n \quad \longrightarrow}{[\underset{0}{3}, \underset{1}{1}, \underset{2}{2}, \underset{3}{4}, \underset{4}{3}]}$, where $n \in [2, 100{,}000]$, item $\in [-1000, 1000]$

  if P = 3, $|\text{sum(left)} - \text{sum(right)}| = |(3+1+2) - (4+3)| = |6-7| = 1$

- Write. solution (a), return absdiff, where abs diff. between left & right (split at P) is at minimum.
  a → array of numbers

- Hint 1:

  7. for each value of P
     abs (sum(subList(0, P)) - sum(subList(P, end))) → $O(n^2)$

  ↳ the easiest solution is often not the correct one because it will perform quite poorly.

  ↳ Think abt from a diff angle, re-encode ur problem. scales

- Hint 2: How to move an item from right to left without re-summing everything?

- Solution: P=1, left=3 : right=10 → 7
  P=2, =4 " $\underset{\downarrow\, 5}{9}$; $\underline{\underline{3}}$:$\underset{\downarrow\, 1}{(6, 7)}$; 4 : $\underset{\downarrow\, 7}{(10, 3)}$

  - Keep min diff & update it if there's lower.
  - return min-diff

  ↳ ~~O(n)~~ CPU = O(n) Memory = O(1)

└ Implementation :

```
class TapeEquilibrium
  def solution(a)
    sum_left = a[0]
    sum_right = a.inject(0) { |sum, x| sum + x } - a[0]
    diff = (sum_left - sum_right).abs
    for i in (1 .. a.length - 2) do
      sum_left += a[i]
      sum_right -= a[i]
      current_diff = (sum_left - sum_right).abs
      diff = current_diff if (diff > current_diff)
    end
    diff
  end
end

puts TapeEquilibrium.new.solution([3,1,2,4,3])
  ↳ 1
```

9) Arrays.

- Most basic data structure; bread & butter of most algorithms; often used as a base to build more complex data structure. many algo. asked in interview require you to perform operations on array, it's important that you're comfortable using these data structure. Arrays are usually modeled as continuous chunks of memory. eg:
  - An array storing 3× 32bit integers, typically requires 12 bytes of continuous memory. An item in an array can be accessed in a random direct manner by multiplying the size of the item by its index.
- Arrays are static structures. You allocate space in the bgn. once & that's all you've got After that you're not allowed to resize that space. If you need more space, the common strategy is to allocate a bigger chunk of memory & copy over the contents of ur old array

Some languages provide array libraries that resize arrays in this manner automatically whenever you need more spaces → amortized array / growable arrays. (eg. Java, Ruby)

└ Resizing arrays has a performance penalty. Arrays might not be the right data structure for you if you're not certain how much data you have.

10) Cyclic Rotation Problem.

└ solution(a,k) ← number of rotations.
← array with length n.

return an array, with contents rotated by k times to the right.

∴ write an efficient solution to solve this. (there exists an algo solving this problem in linear runtime & memory time complexity.
|CPU| |MEM|

11) Hint 1 - The remainder trick. (%)

12) Solution:

eg. [7, 2, 8, 3, 5] K=2, Size=5
0 1 2 3 4

- (index + K) % size → new position.

* ≈ pacman → use %

* Used in Circular Buffers, Hashing Algorithms.

13) Code:

```
class CyclicRotation
  def solution(a,k)
    result = Array.new(a.length)
    for i in 0..(a.length - 1)
      result[(i+k)%a.length] = a[i]
    end

    result
  end
end
puts CyclicRotation.new.solution([3,8,9,7,6], 3).join(",") → 9,7,6,3,8
puts CyclicRotation.new.solution([1,2,3,4], 4).join(",") → 1,2,3,4
```

14) Counting elements.

- counting the number of repetitions of each element in a list.

&

a	4
b	3
c	2

→ frequency table → to come up with a histogram.

- Use hash table, on average has a great preformance, in the worst case it can perform poorly & have O(N) for inserts & searches. Some strict interviewers might be looking for better ways case performance
- Using array instead of a hash table is possible depending on type of element on the list., but memory-consuming.

15) Max Counters Problem.

- can be computed in order ~~O(n+m)~~ n+M (runtime) (CPU : O(n+m)), MEM : O(n)

space time

size of array

- solution (n, a) ← array ~~size~~ of instructions.

n of counter

Hint 1: move starting line whenever 'max-counter'

16) Max Counter Solution.

- move every left behind to the starting line.

&

```
array counters
var startLine
var currentMax
for each (instruction IN a) {...}
for each (counter IN counters) {...}
return counters
```

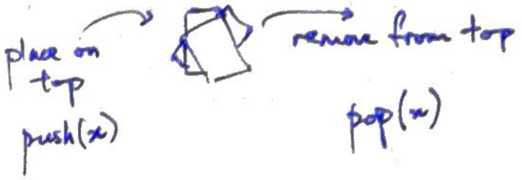
~~def solution (n,a)~~
~~counters = Array (n, 0)~~
~~start_line = 0~~
~~current_max = 0~~
~~for i in a do~~
~~x = i - 1~~
~~case~~
~~when~~

19) Max Counters Code Walkthrough

```
def solution(n,a)
    counters = Array.new(n,0)
    start_line = 0
    current_max = 0
    for i in a do
      x = i-1
      case
      when i > n
         start_line = current_max
      when counters[x] < start_line
         counters[x] = start_line + 1
      else
         counters[x] += 1
      end
      current_max = counters[x] if i<=n && counters[x] > current_max
    end
    for i in 0..counters.length -1 do
      counters[i] = start_line if counters[i] < start_line
    end
    counters
end
                              max counter.
                              ↓
puts solution(5, [3,4,4,6,1,4,4]).join(",")
    ↳ [3, 2, 2, 4, 2]
```

20) Stack & Queues Data Structures.

- some of the simplest forms of DS in Computer Science.
- Used in many diff. algo
- Knowing when & how to use both will help you solve many interview problem.
  - A combination of technical knowledge & experience.
    - Tech : To acquire technical knowledge, you'll need to know how stacks & Queues are implemented & how they're used
    - Experience : work through coding puzzles & try to find patterns in the types of problems.
- Stack?



  - 2 main stack implementations :
    - array base. - pointer (top-most) ↑↓ push pop. - simple. limited by size
    - use a linked list based stack which allows the stack to grow dynamically.
- Queue?
  - 2 main operations
    - Tail - Enqueue : + node on the tail of the linked list.
    - Head - Dequeue : removes 1st item at the head of the linked list.
- Diff between Stack & Queue?
  - The mode in which data is consumed from the data structure.
    - Stack : FIFO ; Queue : LIFO .
- If there's easy a way to implement a queue using an array instead of a linked list?
  - Yes, visit cutajarjames.com

21) Brackets Problem

	return
eg. "[()()]" ✓	1 → correctly nested
"[(]()]" ✗	0 → otherwise.
"[{()}]" ✓	1
"][" ✗	0
"()[]{}()[]()" ✓	1

- Hint 1: Use Stacks // Queues.
  - Initialize an empty stack (eg. stack=[])
    - stack.push("6m")
    - stack.pop() == "m"
  - Process the string input 1 character at a time & depending on the character, decide to push/pop on this stack.
- Solution:
  - Rule:
    - left bracket → Push to stack ; Right bracket → Pop to stack & check whatever the item is equivalent bracket of the right one we have.
  - eg. "[({})]" → (, [
    - stack.push("[") ×3
    - stack.pop() == "{" ? ×3 → (, [
    - stack.empty?
  - eg. "[{(})]"
    - stack.push ×3
    - stack.pop()=="{"? → ✗ ⇒ fail
  - eg. "[({})"
    - ~~stack.pop() == "["? → ✗~~
    - stack.empty? → ✗

Brackets Code Walkthrough.

```
class Brackets
  def solution(s)                                → string.
    valid = true                                 → initialize 'valid' variable
    stack = []                                   → initialize an empty stack.
    s.each_char do |c|                           → loop over each_char of string as 'c'
      case
      when c == "[" || c == "{" || c == "("
        stack.push(c)                            → push to stack for left-bracket.
      when c == ")"
        left = stack.pop
        valid = false if left != "("
      when c == "]"                              → pop & set valid to false if any failed.
        left = stack.pop
        valid = false if left != "["
      when c == "}"
        left = stack.pop
        valid = false if left != "{"
      end
    end                                          → end the string loop.
    valid && stack.empty? ? 1 : 0                → if valid is true and the stack is empty.
  end
end

puts Brackets.new.solution("()[]{}()[]{}") → 1
```

25) Fish (voracious) Problem.

- left : 0 ; right : 1 ; 1st input : [0, 1, 0, 0, 0] ; 2nd input (fish weight) : [4, 3, 2, 1, 5]
  - an array of direction
  - elements are unique as fishes are unique
- solution (a, b)
  - direction [0, 1, 0, 0, 0]
  - array of weight [4, 3, 2, 1, 5]
- return integer - number of survivors (eg. 2)
- Efficient algo that is linear of both space & runtime complexity (#CPU : O(n) ; #MEM : O(n))
- Hint 1 : use stack datastructure
  - initialize an empty stack
  - Rule : When we meet a fish moving to the left, compare it with whatever we have on our stack;
    - if empty . survivors++
    - if !empty, compare again
    - → if smaller : dispose fish swimming left
    - → if larger : dispose fish right swimming
  - When right : store that on top of our stack.
  - Once every items in the list (stack) is processed, count the number of items & + size (stack) to survivors (count)

```
class Fish
  def solution (a, b)                    → a : weight ; b : direction.
    stack = []                           → initialize a stack
    survivors = 0
    for i in 0..a.length-1 do            → loop over each item of a array
      weight = a[i]
      if b[i] == 1                       → swimming right
        stack.push(weight)               → push weight to stack.
      else
        weightdown = stack.pop
        white !weightdown nil? && weightdown < weight
          weightdown = stack.pop
        end
        if weightdown.nil?
          survivors += 1
        else
          stack.push(weightdown)
        end
      end
    end
    survivors + stack.length
  end
end
```

puts Fish.new.solution([4, 8, 2, 6, 7], [0, 1, 1, 0, 0]) => 2

29) Leader definition & the Denominator Problem (Find leader of an array).

- count(c) > $\frac{n}{2}$; element c is a leader if the count of occurrences in that element is more than half the size of the list.
  - With this def, we could only have 1 leader because 2 items can't both be occurring more than half the size of the list.
- eg. [3, 4, 2, 3, 3, 2, 3] ; $\underset{\text{count(3)}}{4} > \frac{7}{2}$, 3 is the leader.

  eg. [5, 1, 5, 3, 1, 5, 4] ; $3 \not> \frac{6}{2}$; no leader in this list.
- Solution 1 : count the occurrences of each item & return the first one that occurs more than half the size of the array.
  - count = null ; leader = null.
  - Problem : Slow runtime performance ; In the worst case (no leader), the runtime complexity is quadratic.
- Solution 2 : Sorting the array first ; if the item occurs for more than half of the size of the array, the middle element in a sorted list has to be that item. Pick the middle element of a sorted list & count the item to check if it's the leader.
  - Runtime performance (#CPU) : O(n logn)
- Solution 3 : Denominator (Codility Puzzle)
  - ★ Return any index of that leader if there's one
    - eg. [2, 4, 3, 3, 3, 2, 3] → output : 2, 3, 4, 6
  - Require : #CPU : O(n), linear runtime & space complexity.
  - solution(a) [2, 4, 3, 3, 3, 2, 3]
    - return any index of the leader , -1 if none.
  - Hint : If we remove 2 non-equal elements from the array, the leader will still be the same
    - Initialise a stack, loop over the array. if stack is empty → insert
      - not empty → ~~pop~~ & diff, pop both pop & throw
        - same, ~~pop~~ throw & push back.
    - Take the topmost entry & count the occurrences in the original array, > $\frac{n}{2}$ → leader
      - #O(n) for CPU & MEM → consume the same amt of memory as input array.
      - The worst case : if every entry is the same.

Solution - Store a variable representing that entry, named 'candidate.'

└ candidate counter
    wolf 3

└ diff. item, ~~subtract~~ counter ~~→~~ -1 ~~++~~

└ same. item, counter ++

→ got a candidate leader represented by ...

→ O(n) linear time, constant space

```
┌ class Denominator
    def solution(a)
      consecutive_size = 0
      candidate = 0
      for item in a do
        case
        when consecutive_size == 0
          candidate = item
          consecutive_size += 1
        when candidate == item
          consecutive_size += 1
        else
          consecutive_size -= 1
        end
      end
      occurrence = a.inject(0){ |sum, x| sum + if x == candidate then 1 else 0 end}
      if occurrence > (a.length/2) then a.find_index(candidate) else -1 end
    end
  end
  puts Denominator.new.solution([3,0,1,1,4,1,1])
```

↳ 2 ← the first index of 1
↖ or 3, 5, 6

33) (Max Sub Array Problem) Maximum Slice Problem

- continuous, have +ve & -ve values.
- 1) Find all sub array method $o(n^2) / o(n^3)$
  2) Divide & Conquer method $o(n \log n)$
  3) Kadane method $O(n)$ linear

34) Max Profit Problem

solution (a)

- return maxProfit, the profit on the best buying opportunity
- #CPU $o(n)$ #MEM $o(1)$

35) Max Profit Hint.

- Kadane's Algo.
  - 2 variables = global max, local max → if local max > global max, reset global max
  - ~~1st or 1st~~ nth or [illegible] (n_{th} + n_{+1})
    ↓
    restart local max subarray ~~[illegible]~~ when choosing left side.

```
class MaxProfit
  def solution(a)
    global_max_sum = 0
    local_max_sum = 0
    for i in 1.. a.size -1
      d = a[i] - a[i-1]                 → delta = ΔP
      local_max_sum = [d, local_max_sum + d].max
      global_max_sum = [local_max_sum, global_max_sum].max
    end
    global_max_sum
  end
end
puts MaxProfit.new.solution([23171, 21011, 21123, 21366, 21013, 21367])
        → 356                                  +356
```

38) Overview of diff. sorting Algorithms

- Selection sort : Insert unsorted card to sorted card
- Bubble sort : Swapping adjacent cards a number of time
  - & perform poorly with runtime complexity of $O(n^2)$

~~→ sorting function →~~

→ well-known efficient sorting function : merge, quick, heapsort

quadratic.

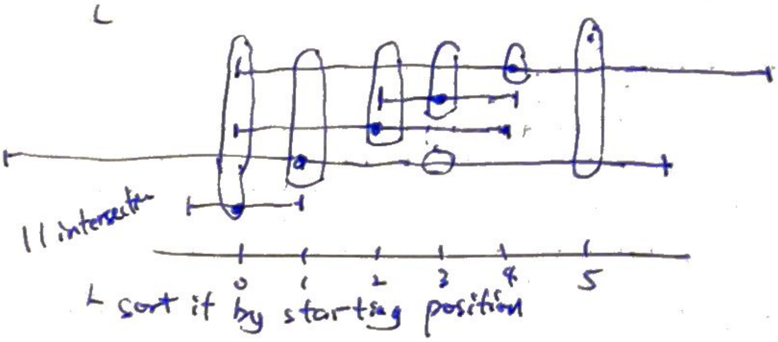
& log-linear worst case runtime complexity.

- ~~merge sort $O(n \log n)$~~
- ~~quick sort $O(n^2)$; $O(n \log n)$, avg~~
- ~~heap sort~~ $O(n \log n)$

	CPU	MEM	others.
L merge sort	$O(n \log n)$	$O(n)$	
L quick sort	$O(n^2)$; $O(n \log n)$, avg	$O(\log n)$	
L heap sort	$O(n \log n)$	$O(1)$	- algo unstable

39) Disc Intersection Problem.

- given an array of disc ~~position &~~ radius, find number of intersections.
- ~~eg.~~ input : eg [1, 5, 2, 1, 4, 0] → radius.
  0 1 2 3 4 5 ← position.
- solution(a)
  - return count of intersection pairs, or -1 if count > 10 000 000
- runtime complexity : #CPU = $O(n \log n)$
- Hint : Look at them from the top. eg. [1, 5, 2, 1, 4, 0]
  -

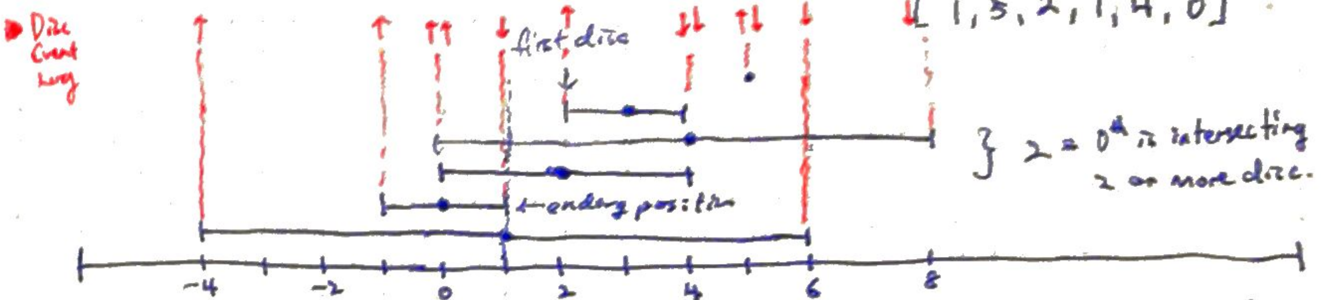


0: {1, 2, 4}
1: {2, 4}
2: {3, 4}
3: {4, 1}
4: {}
5: {1, 4}

- sort it by starting position

41) Disc Intersection Solution

- In Hint ; - Looking from the top & sort according to the starting positions, allow us to use binary search technique.
- Process each disc in the order that it was sorted.



- pick the ending position of a disc & find out the first disc that starts after that ending position. (eg for 0th, 3rd disc is the first disc that starts ~)
- subtract its index from the current index that we're processing.
- Runtime Complexity : ~~O(nlogn) for sorting & another O(nlogn) for~~ runtime complexity.
  - O(nlogn) Sorting
  - O(nlogn) Binary Search. } #CPU = O(nlogn)

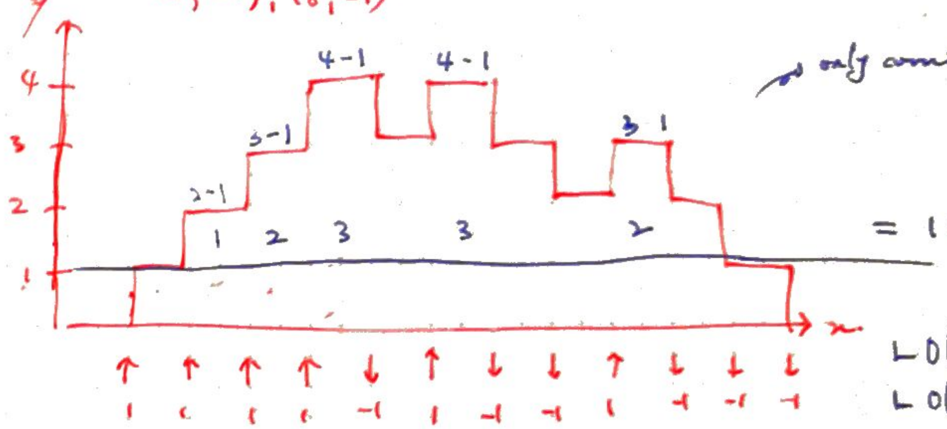
Solution 2 : Disc Event Log. (Difficult to understand but easier to implement)

↳ ~~(-4, +1)~~ +1 : -4, -1, 0, 0, 2, 5

−1 : 1, 4, 4, 5, 6, 8

↳ sort ~~by~~ ~~(-4,~~ (asc, desc)

↳ (-4, +1), (-1, 1), (0, 1), (0, 1), (1, -1), (2, +1), (4, -1), (4, -1), (5, 1), (5, -1), (6, -1), (8, -1)



↗ only count +1

= 11

- O(nlogn) Sorting
- O(n) log Processing

↳ O(nlogn)

#CPU =

42) Disc Intersection Code Walkthrough

```
class NumberOfDiscIntersections
  Disc = Struct.new(:x, :start_end)
  def solution(a)
    disc_history = Array.new(a.length * 2)   → start & finish / disc
    j = 0                                    → pointer to disc history area.
    for i in 0..a.length-1 do                → starting pt.
      disc_history[j] = Disc.new(i - a[i], 1)     → +1 = start ↑
      disc_history[j+1] = Disc.new(i + a[i], -1)  → -1 = ending ↓
      j += 2                                 → ending pt
    end
    disc_history = disc_history.sort_by{|a| a.x*10 - a.start_end}
    intersections = 0                        → sort by x coordinate, then marker desc.
    active_intersections = 0
    for log in disc_history do
      active_intersections += log.start_end
      intersections += active_intersections -1 if log.start_end > 0   ]...
      if intersection > 1000000
        intersection = -1
        break
      end
    end
    intersections
  end
end
puts NumberOfDiscIntersections.new.solution([1,5,2,1,4,0])
  ↳ 11
```