Abstract

Who among us hasn't felt the pain of using a dictionary or list instead of a more fit-to-purpose data structure because it was easy and available? Have you ever wondered how exactly a hashtable works, or why people sometimes call a heap a treap? Have you ever heard of Big O and Big Θ, and if you have, do you remember what they are?

Many of us are from non-traditional backgrounds or bootcamps and may never have had an algorithms and data structures course, and the rest of us are years removed from the last time we studied the subject formally. As we advance in our careers as software developers, we keep getting further and further away from the fundamentals. That's a shame because the fundamentals are useful in our day jobs, appear often as technical interview questions, and honestly they're also really fun to poke at. It's right there in the name!

This workshop is a refresher, or crash course, on the basic data structures and algorithms that underpin the software we build. We'll focus on techniques for recognizing the structure of a problem, finding a relevant data structure to help solve it, and ballparking the cost of computing answers in terms of time and memory. We won't be implementing structures from scratch. We'll look at:

- Segmenting an image using the color of adjacent pixels to find edges
- The importance of good hashing through the lens of solving a Rubik's cube
- Rapidly testing collision for all sprites in a video game field
- Efficiently finding similar items in a large catalog of items

Attendees can expect to come away with:

- Better familiarity with common data structures
- A process for identifying the correct data structure to solve a problem with
- A (re)-introduction to Big O & Big Θ analysis

Data Structures Crash Course

Jordan Thayer
AI Strategist

Help Me Help You (Where Are We All At)

- How many of you have a formal CS education
    - How many of you had an algorithms & data structures course?
    - That did complexity analysis?
    - And you graduated in the last 5 years?
- How many of you like weird number theory & high performance computing stuff?
    - How many of you are going to do advent of code?
    - Have a leet coding profile?

What to Expect Today

We Will:

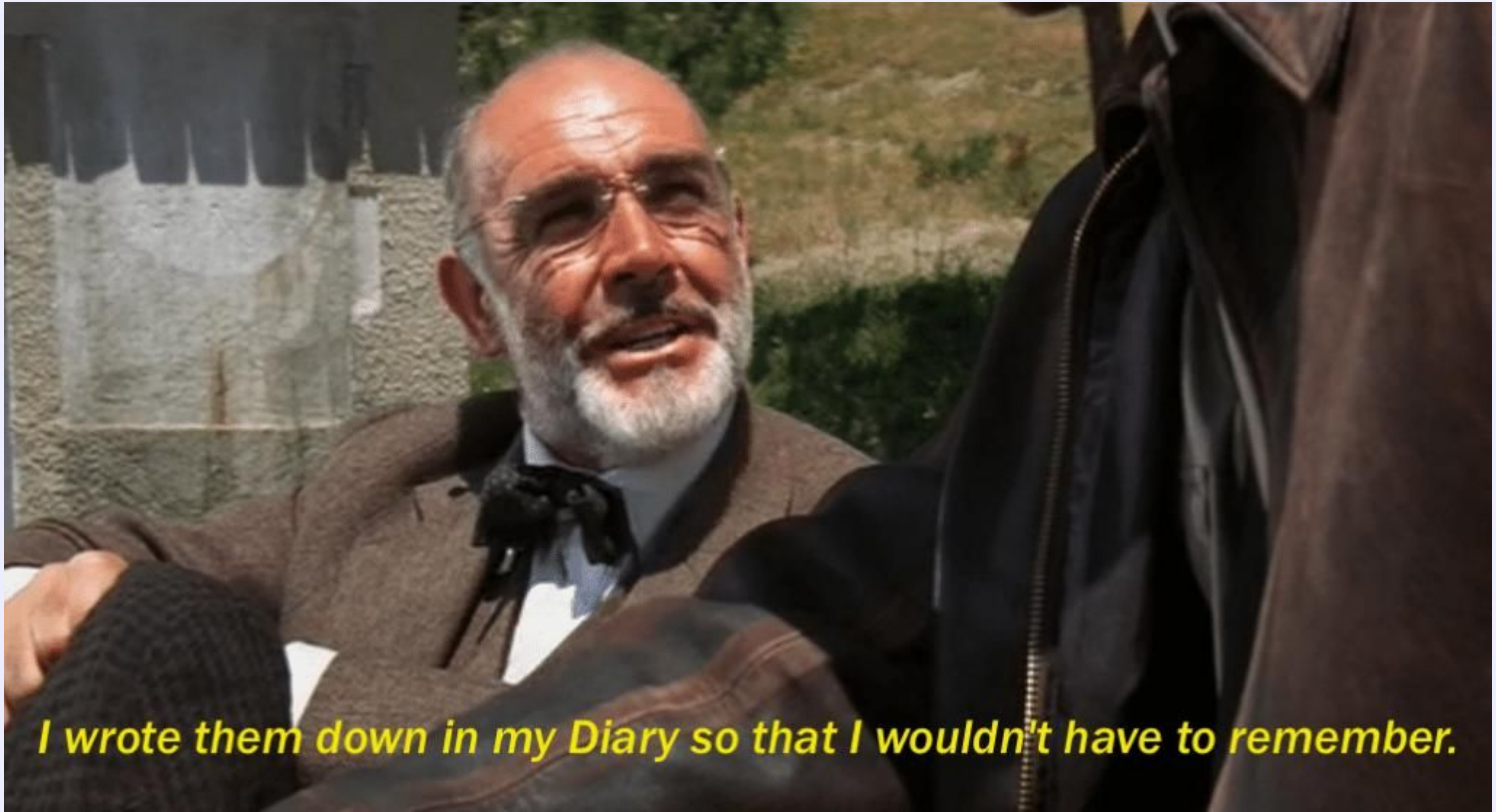
- Map Problems to Algorithms
- Analyze Algorithm Runtime
- Look for patterns in Runtime Analysis
- Focus on Real, Relatable Problems
    - I hope

We Won't:

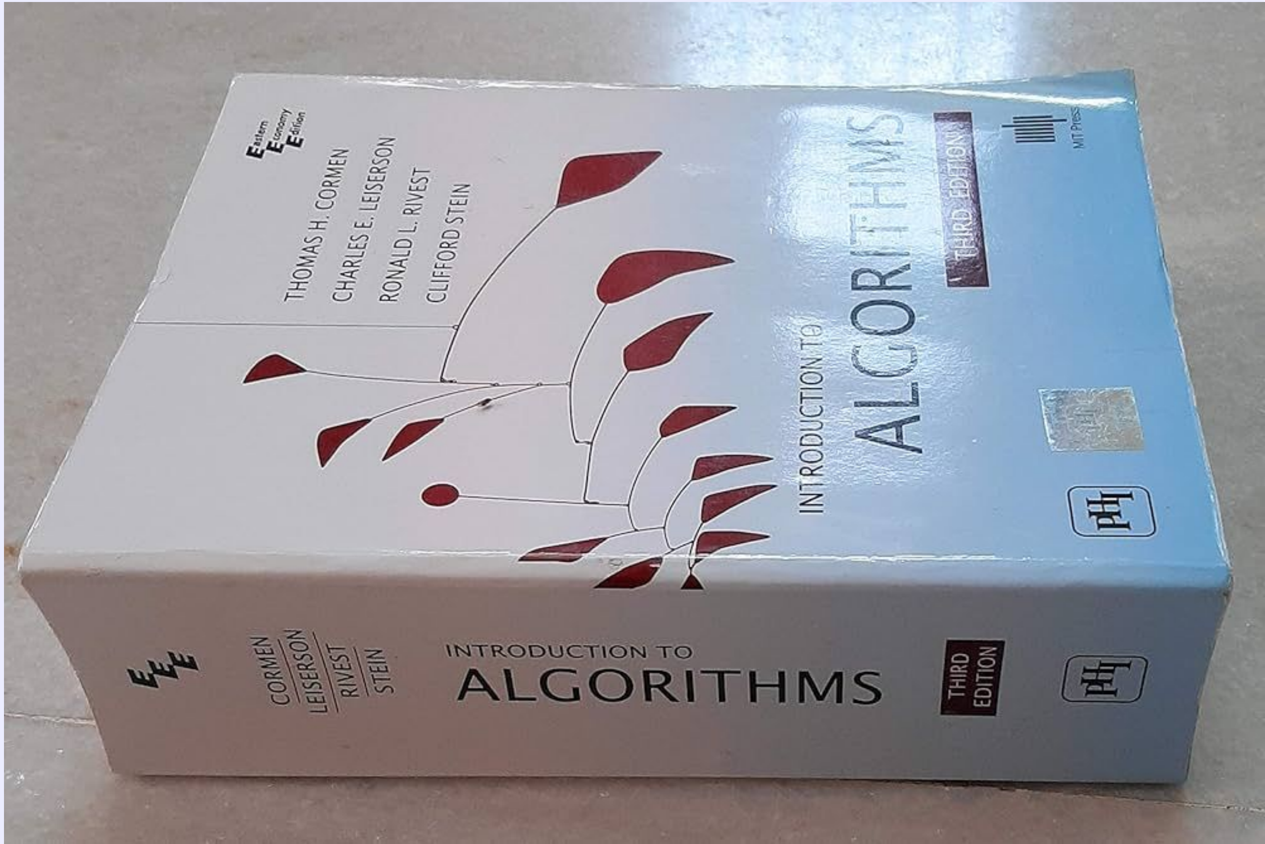
- Belabor the theory behind Big-O Notation
- Write Formal Proofs
- Implement Algorithms from Scratch
- Cover All Algorithms

If Today Isn't Enough

… or if you just want a useful desk reference



BUY ~~MY~~ THIS BOOK



Also MAYBE This One

https://numerical.recipes/

The Rest of Today:

- Introductions, Level & Expectation Setting
- This Outline
- Kicking the Tires by Analyzing Sorting Algorithms
- Segmenting Images for Further Analysis
- The importance of good hashing through the lens of solving a Rubik's cube
- Rapidly testing collision for all sprites in a video game field
- Efficiently finding similar items in a large catalog of items
- Wrap Up
- **TODO: Reorder Outline to Reflect**

Sorting Algorithms

Which sorting algorithms are you familiar with / have you heard of?

Sorting Algorithms

Algorithm	Best	Worst	Average
Bubble Sort			
Merge Sort			
Radix Sort			

What's it mean to be sorted?

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

```
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red (or orange) line happen?

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red (or orange) line happen?

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red line happen?

- Sorted Data

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red line happen?

- Sorted Data
- Reverse Data

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red line happen?

- Sorted Data
- Reverse Data
- Random Data

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red line happen?

- Best Case
- Worst Case
- Average Case

```python
def bubble_sort(input, comparator):
    sorted_this_pass = False
    while not sorted_this_pass:
        prev = 0
        sorted_this_pass = True
        for i in range(1, len(input)):
            tmp_a = input[prev]
            tmp_b = input[i]
            if not comparator(tmp_a, tmp_b):
                input[prev] = tmp_b
                input[i] = tmp_a
                sorted_this_pass = False
            prev = i
```

How often does the red line happen?

- Best Case: $\Omega(n)$
- Worst Case: $O(n)$
- Average Case: $\Theta(n)$

Sorting Algorithms

Algorithm	Best	Worst	Average
Bubble Sort	n	n^2	n^2 / 2
Merge Sort			
Radix Sort			

```
def k
    s
    w
```

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

```
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

```
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

These are our recursive calls, they make a loop too!

We'll analyze Split.merge next, after the loop

```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```

```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```

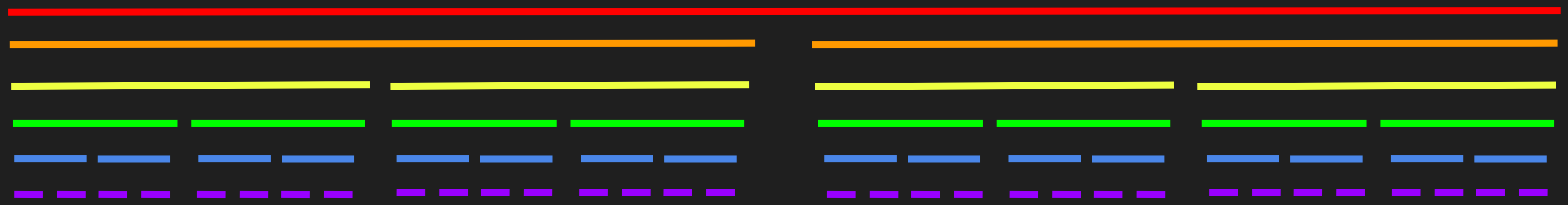
That's it. That's all there is.

```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```

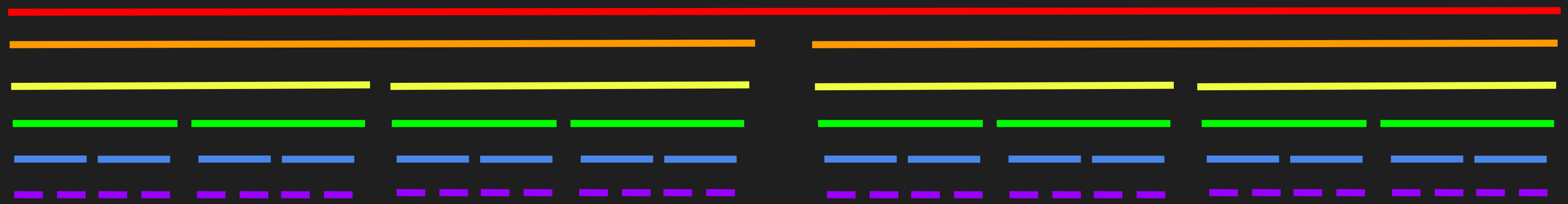
```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```

```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```

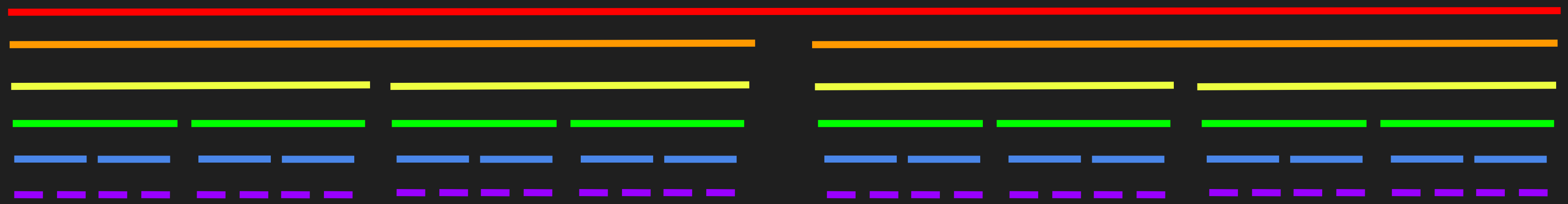
```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```



```python
def split(self):
    if self.length > 1:
        half = self.length // 2
        left = Split(self.start, self.start + half - 1)
        right = Split(self.start + half, self.end)
        return left, right
    else:
        return self, None
```



```
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```



If we halve each time, how deep is the recursive tree?

```python
def merge_sort(input, comparator, split=None):
```

If we halve each time, how deep is the recursive tree?

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

These are our recursive calls, they make a loop too!

We'll analyze Split.merge after them


MR. RENTAL
NOW. YOU'RE LOOKING AT
NOW. EVERYTHING THAT HAPPENS
NOW IS HAPPENING NOW.

```python
def merge(split_1, split_2, input_list=None, comparator=None):
    delta = split_2.start - split_1.end
    if delta != 1:
        raise ValueError(f"Splits must be adjacent, {split_1} and {split_2} are not.")
    if input_list is not None:
        index_1 = split_1.start
        index_2 = split_2.start
        merged = []
        while index_2 <= split_2.end and index_1 <= split_1.end:
            if comparator(input_list[index_1], input_list[index_2]):
                merged.append(input_list[index_1])
                index_1 += 1
            else:
                merged.append(input_list[index_2])
                index_2 += 1
        for ind in range(index_1, split_1.end+1):
            merged.append(input_list[ind])
        for ind in range(index_2, split_2.end+1):
            merged.append(input_list[ind])
        for offset, el in enumerate(merged):
            input_list[split_1.start + offset] = el
    return Split(split_1.start, split_2.end)
```

```python
def merge(split_1, split_2, input_list=None, comparator=None):
    delta = split_2.start - split_1.end
    if delta != 1:
        raise ValueError(f"Splits must be adjacent, {split_1} and {split_2} are not.")
    if input_list is not None:
        index_1 = split_1.start
        index_2 = split_2.start
        merged = []
        while index_2 <= split_2.end and index_1 <= split_1.end:
            if comparator(input_list[index_1], input_list[index_2]):
                merged.append(input_list[index_1])
                index_1 += 1
            else:
                merged.append(input_list[index_2])
                index_2 += 1
        for ind in range(index_1, split_1.end+1):
            merged.append(input_list[ind])
        for ind in range(index_2, split_2.end+1):
            merged.append(input_list[ind])
        for offset, el in enumerate(merged):
            input_list[split_1.start + offset] = el
    return Split(split_1.start, split_2.end)
```

```python
def merge(split_1, split_2, input_list=None, comparator=None):
    delta = split_2.start - split_1.end
    if delta != 1:
        raise ValueError(f"Splits must be adjacent, {split_1} and {split_2} are not.")
    if input_list is not None:
        index_1 = split_1.start
        index_2 = split_2.start
        merged = []
        while index_2 <= split_2.end and index_1 <= split_1.end:
            if comparator(input_list[index_1], input_list[index_2]):
                merged.append(input_list[index_1])
                index_1 += 1
            else:
                merged.append(input_list[index_2])
                index_2 += 1
        for ind in range(index_1, split_1.end+1):
            merged.append(input_list[ind])
        for ind in range(index_2, split_2.end+1):
            merged.append(input_list[ind])
        for offset, el in enumerate(merged):
            input_list[split_1.start + offset] = el
    return Split(split_1.start, split_2.end)
```

```python
def merge(split_1, split_2, input_list=None, comparator=None):
    delta = split_2.start - split_1.end
    if delta != 1:
        raise ValueError(f"Splits must be adjacent, {split_1} and {split_2} are not.")
    if input_list is not None:
        index_1 = split_1.start
        index_2 = split_2.start
        merged = []
        while index_2 <= split_2.end and index_1 <= split_1.end:
            if comparator(input_list[index_1], input_list[index_2]):
                merged.append(input_list[index_1])
                index_1 += 1
            else:
                merged.append(input_list[index_2])
                index_2 += 1
        for ind in range(index_1, split_1.end+1):
            merged.append(input_list[ind])
        for ind in range(index_2, split_2.end+1):
            merged.append(input_list[ind])
        for offset, el in enumerate(merged):
            input_list[split_1.start + offset] = el
    return Split(split_1.start, split_2.end)
```

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

These are our recursive calls, they make a loop too!

We'll analyze them later, after Split.merge

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

log(|input|)

These are our recursive calls, they make a loop too!

We'll analyze them later, after Split.merge

```python
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

What's best case data here?

What about worst case?

```python
def merge(split_1, split_2, input_list=None, comparator=None):
    delta = split_2.start - split_1.end
    if delta != 1:
        raise ValueError(f"Splits must be adjacent, {split_1} and {split_2} are not.")
    if input_list is not None:
        index_1 = split_1.start
        index_2 = split_2.start
        merged = []
        while index_2 <= split_2.end and index_1 <= split_1.end:
            if comparator(input_list[index_1], input_list[index_2]):
                merged.append(input_list[index_1])
                index_1 += 1
            else:
                merged.append(input_list[index_2])
                index_2 += 1
        for ind in range(index_1, split_1.end+1):
            merged.append(input_list[ind])
        for ind in range(index_2, split_2.end+1):
            merged.append(input_list[ind])
        for offset, el in enumerate(merged):
            input_list[split_1.start + offset] = el
    return Split(split_1.start, split_2.end)
```

```
def merge_sort(input, comparator, split=None):
    if split is None:
        split = Split(0, len(input)-1)
    left, right = split.split()
    if right is None:
        return # base case, single element
    merge_sort(input, comparator, split=left)
    merge_sort(input, comparator, split=right, )
    Split.merge(left, right, input_list=input, comparator=comparator)
```

log(|input|)

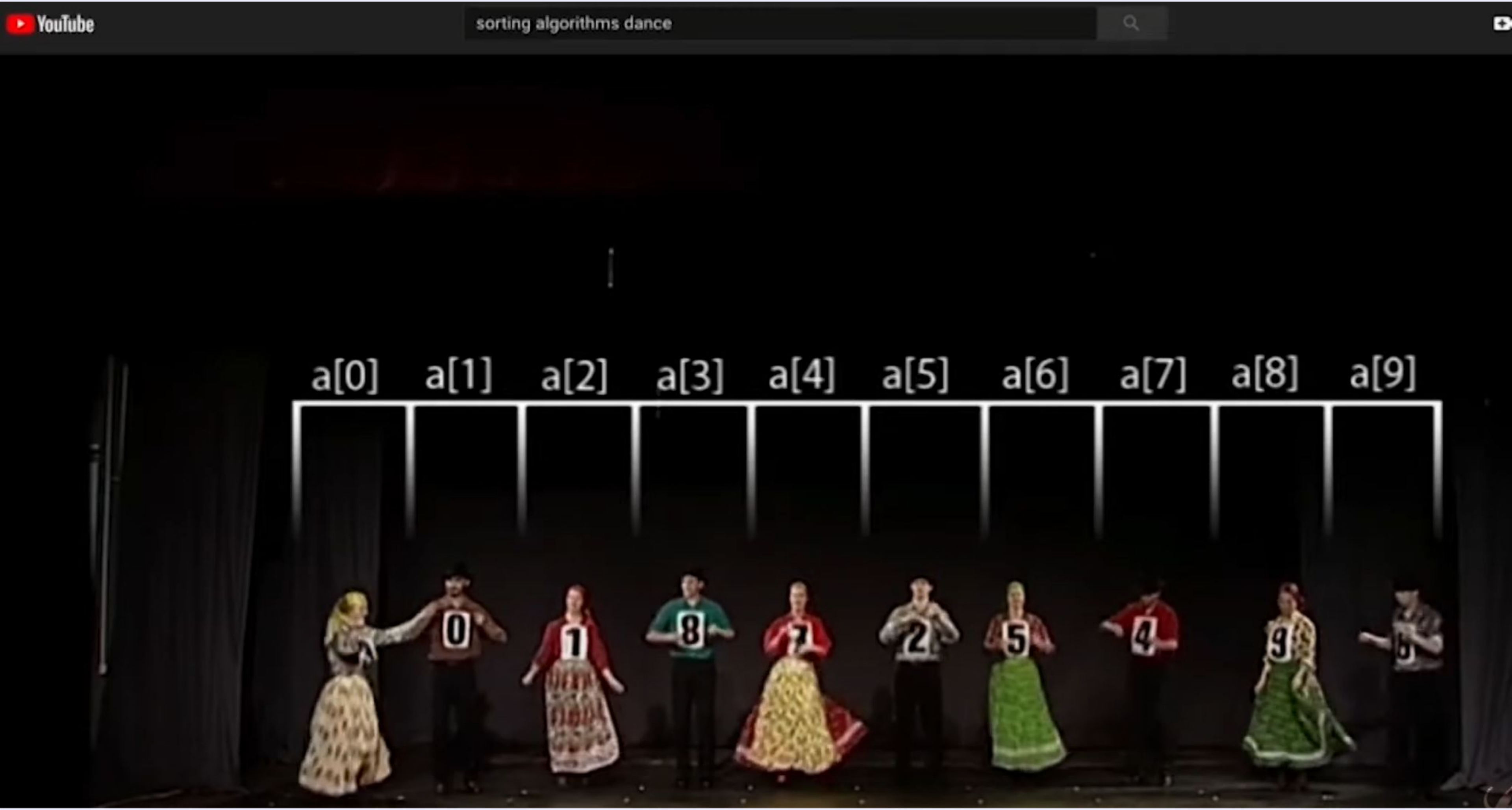
|left| + |right|

There kind of aren't best and worst cases here.

We always touch the data the same-ish way.

Sorting Algorithms

Algorithm	Best	Worst	Average
Bubble Sort	n	n^2	n^2 / 2
Merge Sort	n log(n)	n log(n)	n log(n)
Radix Sort			


YouTube
sorting algorithms dance
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
0
1
8
7
2
5
4
9

```python
def radix_sort(input, comparator):
    radix = 1
    finished = False
    next_list = input
    while not finished:
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

```python
def radix_sort(input, comparator):
    radix = 1
    finished = False
    next_list = input
    while not finished:
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

```python
def radix_sort(input, comparator):
    radix = 1
    finished = False
    next_list = input
    while not finished:
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

```
def radix_sort(input, comparator):
    radix =
    finishe
    next_li
    while n
        nex                                    t, radix)
        rad
        fin
    # The c
    # Our i                                    for this
    # overw                                    ve behavior
    for i i
        inp
```

```python
def radix_sort(input, comparator):
    radix = 1
    finished = False
    next_list = input
    while not finished:
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

```python
def counting_sort(input, radix):
    buckets = [[], [], [], [], [], [], [], [], [], [],]
    # one for each digit in 0 - 9
    # Bucketize input based on radix
    saw_non_zero = False
    for el in input:
        bucket = el % (radix * 10) // radix
        saw_non_zero = saw_non_zero or bucket > 0
        buckets[bucket].append(el)

    # reconstruct output from buckets
    output = []
    for bucket in buckets:
        output.extend(bucket)
    return output, saw_non_zero
```

```python
def counting_sort(input, radix):
    buckets = [[], [], [], [], [], [], [], [], [], [],]
    # one for each digit in 0 - 9
    # Bucketize input based on radix
    saw_non_zero = False
    for el in input:
        bucket = el % (radix * 10) // radix
        saw_non_zero = saw_non_zero or bucket > 0
        buckets[bucket].append(el)

    # reconstruct output from buckets
    output = []
    for bucket in buckets:
        output.extend(bucket)
    return output, saw_non_zero
```

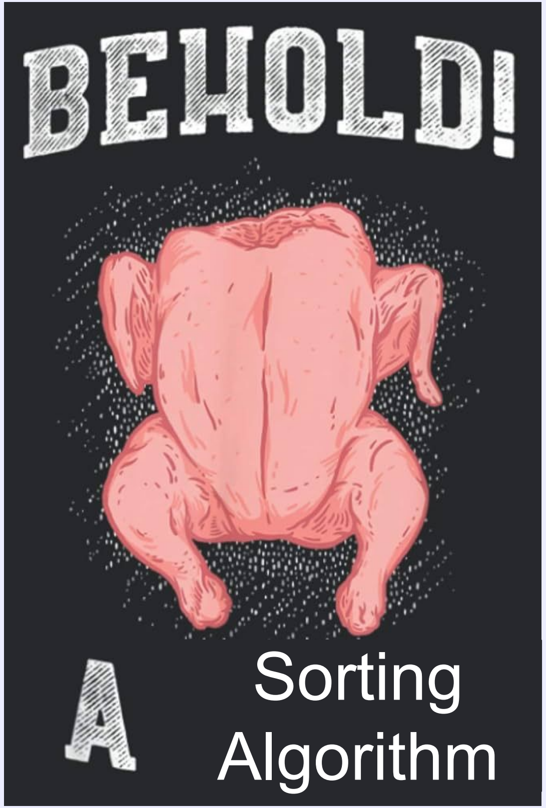
```python
def radix_sort(input, comparator):
    radix = 1
    finished = False
    next_list = input
    while not finished:  radix < all elements in input
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

```
def radix_sort(input, comparator):
    radix = 1                  so, what of best, worst, and expected?
    finished = False
    next_list = input
    while not finished:  radix < all elements in input
        next_list, saw_non_zero = counting_sort(next_list, radix)
        radix *= 10
        finished = not saw_non_zero
    # The other sorts are destructive / in-place
    # Our implementation of counting sort doesn't allow for this
    # overwrite original input to get matching destructive behavior
    for i in range(len(input)):
        input[i] = next_list[i]
```

Sorting Algorithms

Algorithm	Best	Worst	Average
Bubble Sort	n	n^2	n^2 / 2
Merge Sort	n log(n)	n log(n)	n log(n)
Radix Sort	nd	nd	nd

Is Radix Sort Really Sorting?



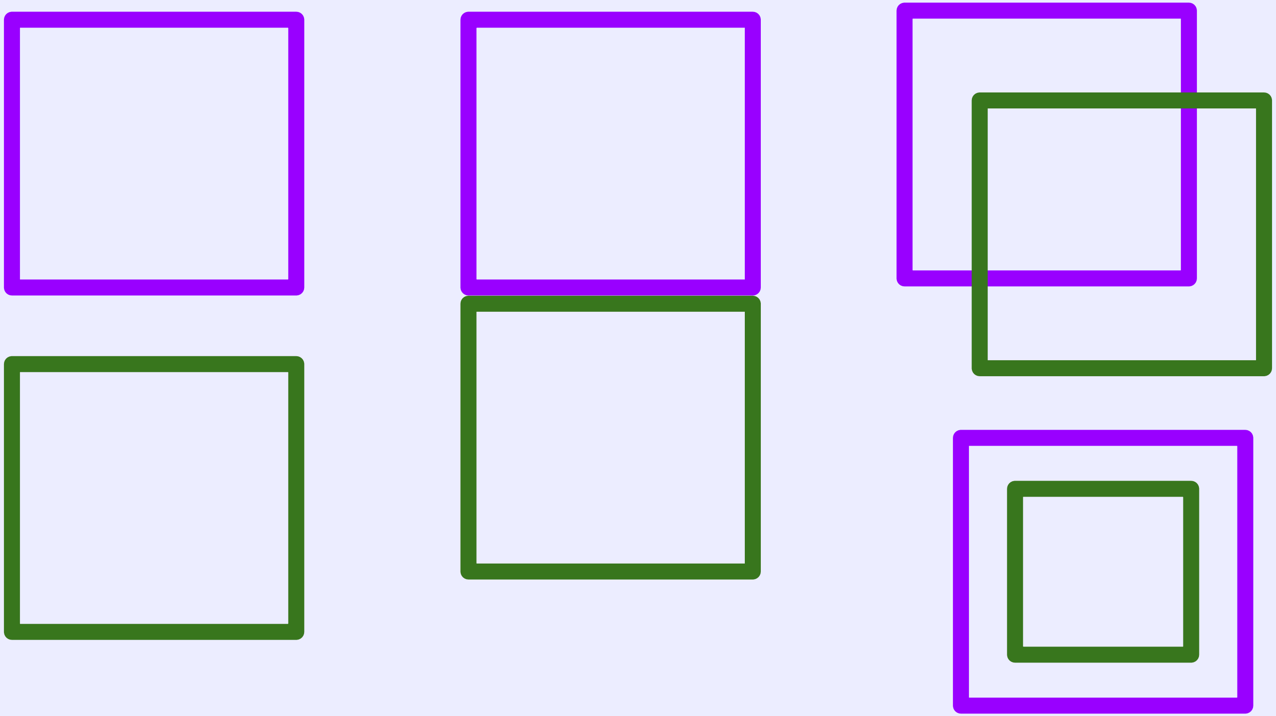
A Brief Respite

Some Common Algorithm / Data Structure Strategies

- Divide & Conquer (usually with trees)
  - You saw this already (Merge Sort)
- Recursive Substructure
- Dynamic Programming
- Some Other Ones I'm not Thinking Of

What commonality do platformers, shmups, and fighting games have?

Rectangle Intersection Testing (How Do We Do This?)



```python
def contains(self, p : Point) -> bool:
    Rect.test_counts += 1
    return (
        p.x >= self.west and
        p.x <= self.east and
        p.y >= self.south and
        p.y <= self.north
    )

def intersects(self, Rect_2) -> bool:
    Rect.test_counts += 1
    return (
        # self contains Rect_2
        (self.north >= Rect_2.north and self.south <= Rect_2.south
             and self.west <= Rect_2.west and self.east >= Rect_2.east) or
        # Rect_2 contains self
        (self.north <= Rect_2.north and self.south >= Rect_2.south
             and self.west >= Rect_2.west and self.east <= Rect_2.east) or
        # self contains one of the corners of Rect_2 (previous cases handle abutment)
        self.contains(Rect_2.corners[0]) or
        self.contains(Rect_2.corners[1]) or
        self.contains(Rect_2.corners[2]) or
        self.contains(Rect_2.corners[3])
    )
```

```python
def contains(self, p : Point) -> bool:
    Rect.test_counts += 1
    return (
        p.x >= self.west and
        p.x <= self.east and
        p.y >= self.south and
        p.y <= self.north
    )

def intersects(self, Rect_2) -> bool:
    Rect.test_counts += 1
    return (
        # self contains Rect_2
        (self.north >= Rect_2.north and self.south <= Rect_2.south
             and self.west <= Rect_2.west and self.east >= Rect_2.east) or
        # Rect_2 contains self
        (self.north <= Rect_2.north and self.south >= Rect_2.south
             and self.west >= Rect_2.west and self.east <= Rect_2.east) or
        # self contains one of the corners of Rect_2 (previous cases handle abutment)
        self.contains(Rect_2.corners[0]) or
        self.contains(Rect_2.corners[1]) or
        self.contains(Rect_2.corners[2]) or
        self.contains(Rect_2.corners[3])
    )
```

So, what's the naive approach?

```python
def contains(self, p : Point) -> bool:
    Rect.test_counts += 1
    return (
        p.x >= self.west and
        p.x <= self.east and
        p.y >= self.south and
        p.y <= self.north
    )

def intersects(self, Rect_2) -> bool:
    Rect.test_counts += 1
    return (
        # self contains Rect_2
        (self.north >= Rect_2.north and self.south <= Rect_2.south
             and self.west <= Rect_2.west and self.east >= Rect_2.east) or
        # Rect_2 contains self
        (self.north <= Rect_2.north and self.south >= Rect_2.south
             and self.west >= Rect_2.west and self.east <= Rect_2.east) or
        # self contains one of the corners of Rect_2 (previous cases handle abutment)
        self.contains(Rect_2.corners[0]) or
        self.contains(Rect_2.corners[1]) or
        self.contains(Rect_2.corners[2]) or
        self.contains(Rect_2.corners[3])
    )
```

So, what's the naive approach?

Test every rectangle against each other

```python
def naive_approach_one_rect(rect_of_interest, rectangles):
    intersecting = []
    for rect in rectangles:
        if rect is rect_of_interest:
            continue
        if rect_of_interest.intersects(rect):
            intersecting.append(rect)
    return rect


def naive_approach_all_rects(rects):
    intersecting = []
    for rect in rects:
        intersect_this_rect = naive_approach_one_rect(rect, rects)
        intersecting.extend(intersect_this_rect)
    return intersecting
```

```python
def naive_approach_one_rect(rect_of_interest, rectangles):
    intersecting = []
    for rect in rectangles:
        if rect is rect_of_interest:
            continue
        if rect_of_interest.intersects(rect):
            intersecting.append(rect)
    return rect


def naive_approach_all_rects(rects):
    intersecting = []
    for rect in rects:
        intersect_this_rect = naive_approach_one_rect(rect, rects)
        intersecting.extend(intersect_this_rect)
    return intersecting
```

```python
def contains(self, p : Point) -> bool:
    Rect.test_counts += 1
    return (
        p.x >= self.west and
        p.x <= self.east and
        p.y >= self.south and
        p.y <= self.north
    )

def intersects(self, Rect_2) -> bool:
    Rect.test_counts += 1
    return (
        # self contains Rect_2
        (self.north >= Rect_2.north and self.south <= Rect_2.south
             and self.west <= Rect_2.west and self.east >= Rect_2.east) or
        # Rect_2 contains self
        (self.north <= Rect_2.north and self.south >= Rect_2.south
             and self.west >= Rect_2.west and self.east <= Rect_2.east) or
        # self contains one of the corners of Rect_2 (previous cases handle abutment)
        self.contains(Rect_2.corners[0]) or
        self.contains(Rect_2.corners[1]) or
        self.contains(Rect_2.corners[2]) or
        self.contains(Rect_2.corners[3])
    )
```

So, what's the naive approach?

Test every rectangle against each other

Best? **Worst?** **Expected?**

```python
def contains(self, p : Point) -> bool:
    Rect.test_counts += 1
    return (
        p.x >= self.west and
        p.x <= self.east and
        p.y >= self.south and
        p.y <= self.north
    )

def intersects(self, Rect_2) -> bool:
    Rect.test_counts += 1
    return (
        # self contains Rect_2
        (self.north >= Rect_2.north and self.south <= Rect_2.south
             and self.west <= Rect_2.west and self.east >= Rect_2.east) or
        # Rect_2 contains self
        (self.north <= Rect_2.north and self.south >= Rect_2.south
             and self.west >= Rect_2.west and self.east <= Rect_2.east) or
        # self contains one of the corners of Rect_2 (previous cases handle abutment)
        self.contains(Rect_2.corners[0]) or
        self.contains(Rect_2.corners[1]) or
        self.contains(Rect_2.corners[2]) or
        self.contains(Rect_2.corners[3])
    )
```

So, what's the naive approach?

Test every rectangle against each other

Best? n^2 Worst? n^2 Expected? n^2

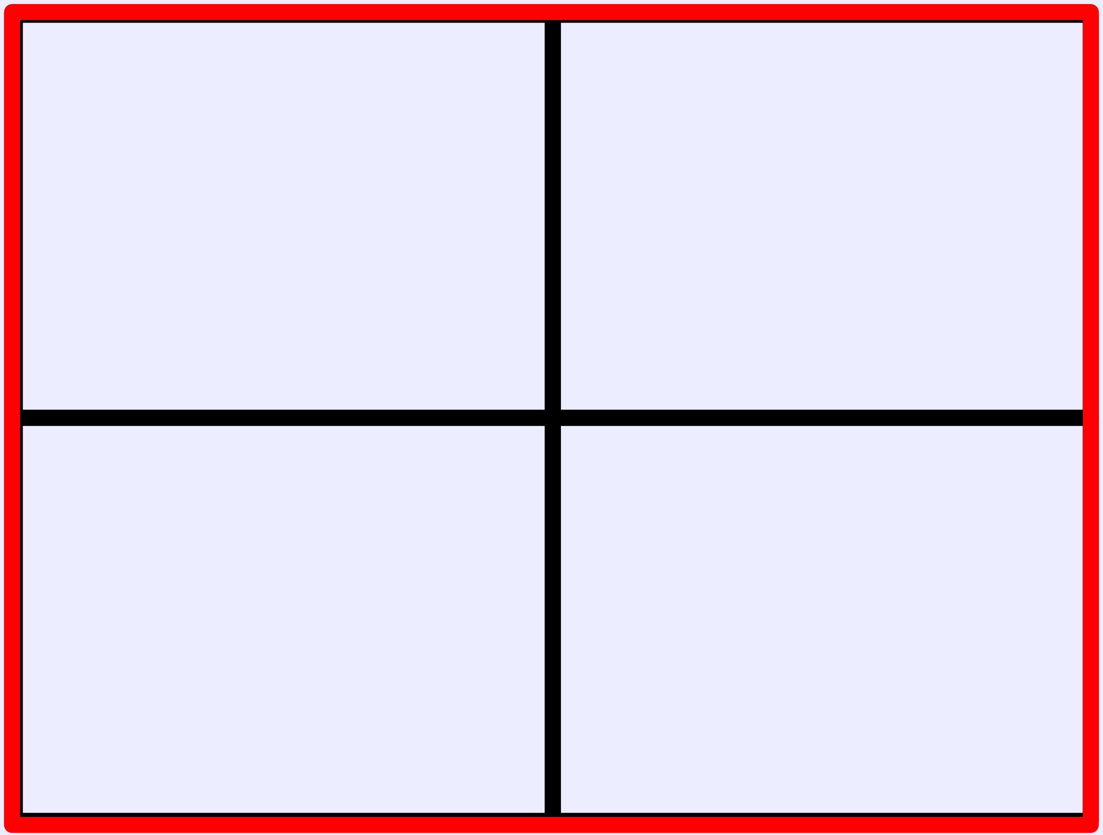
Find Intersecting

Algorithm	Best	Worst	Average
naive	n^2	n^2	n^2
????			

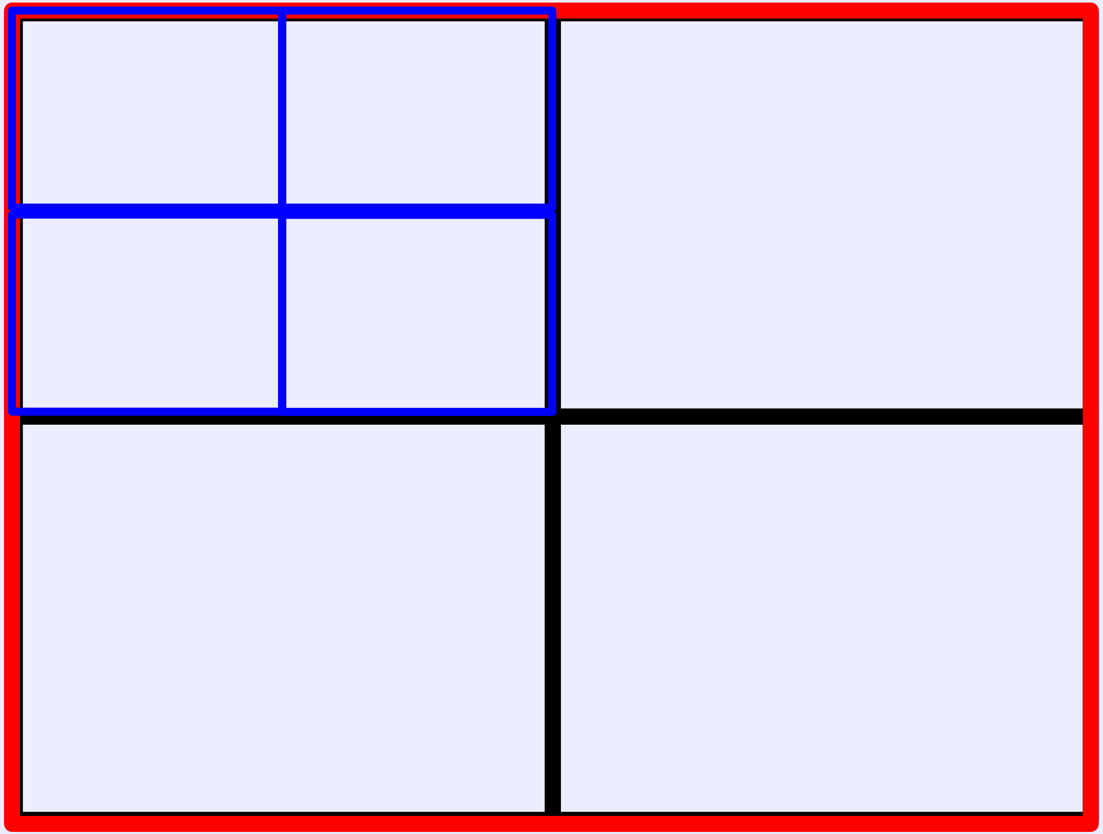

LV 31
15:25
2940
5625

Recursive Space Division

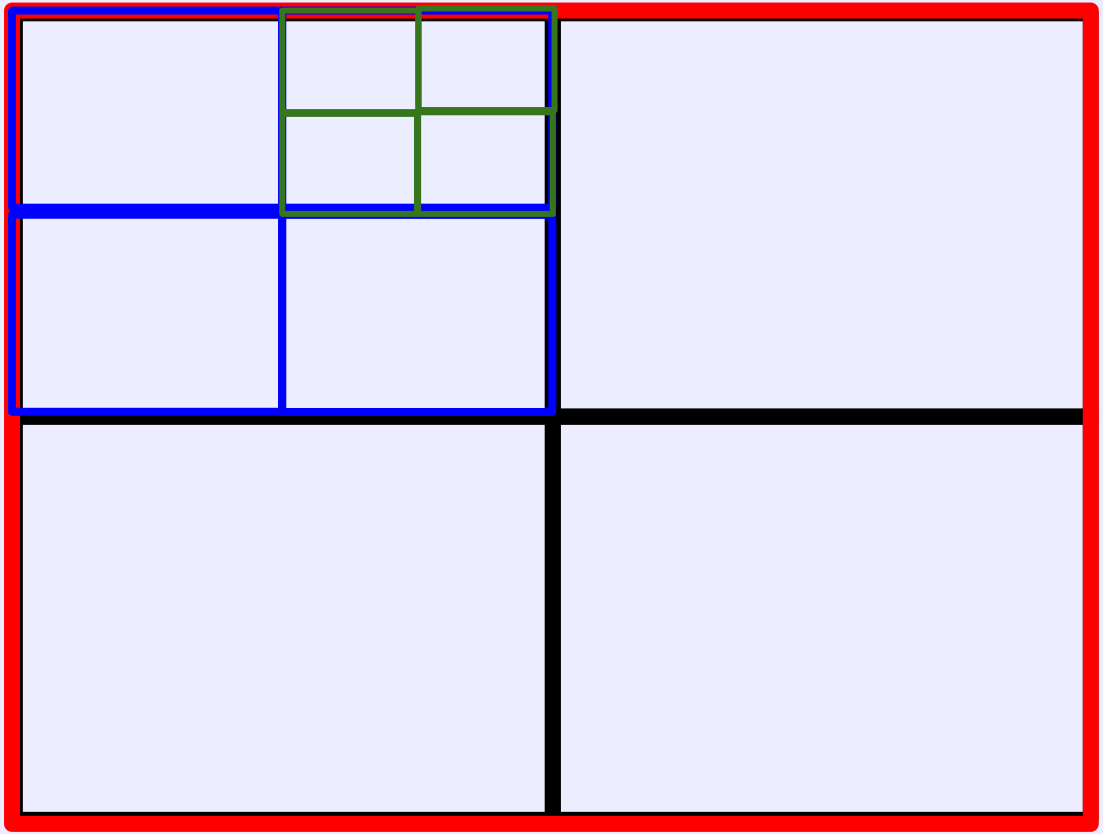
Recursive Space Division



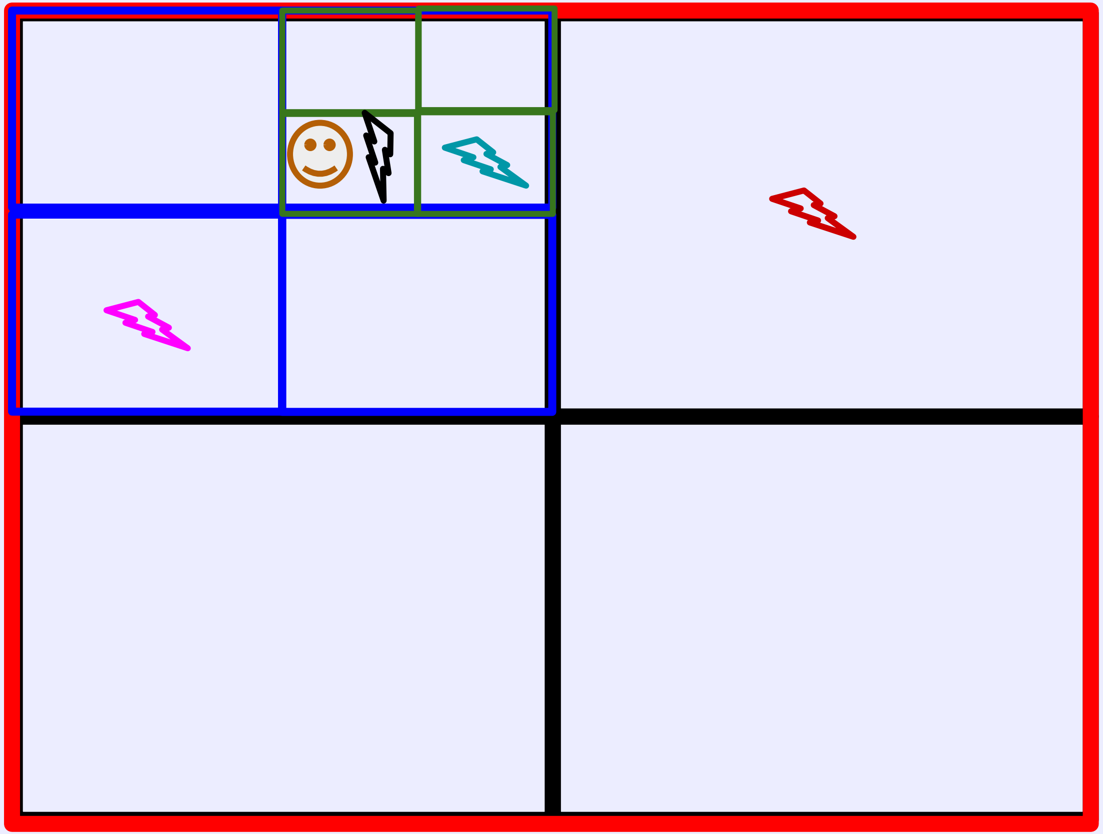
Recursive Space Division



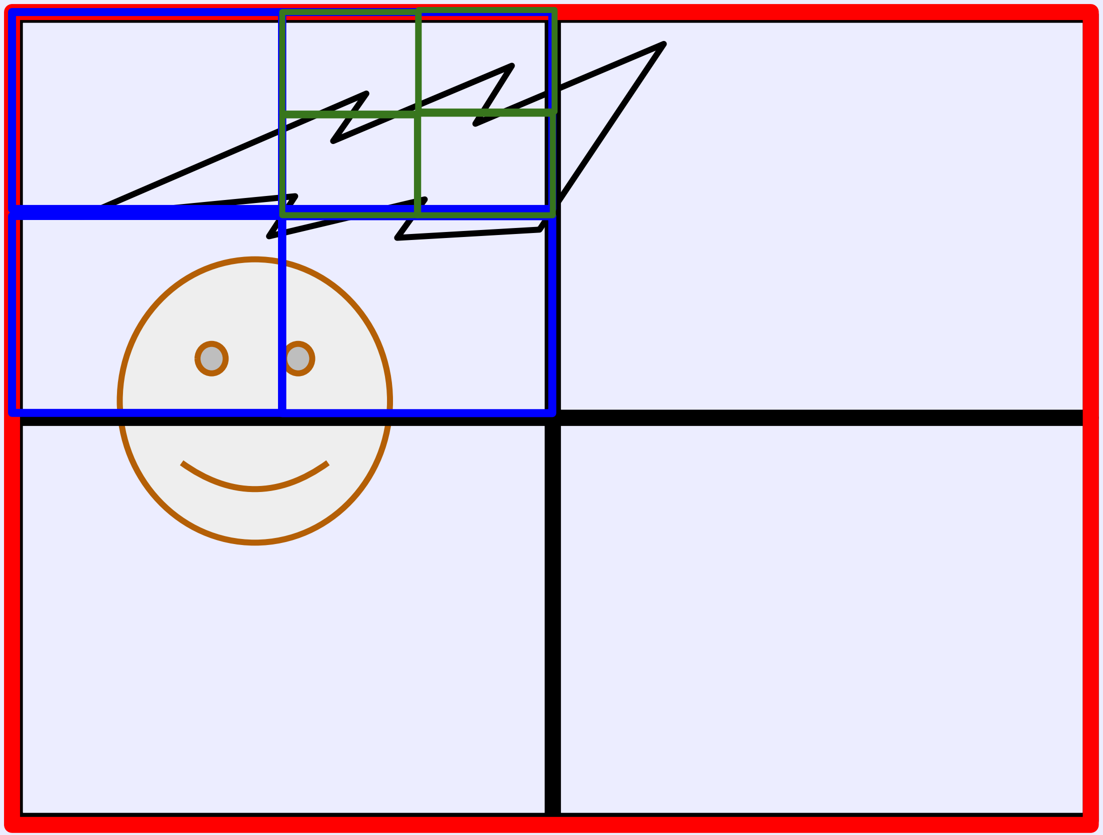
Recursive Space Division



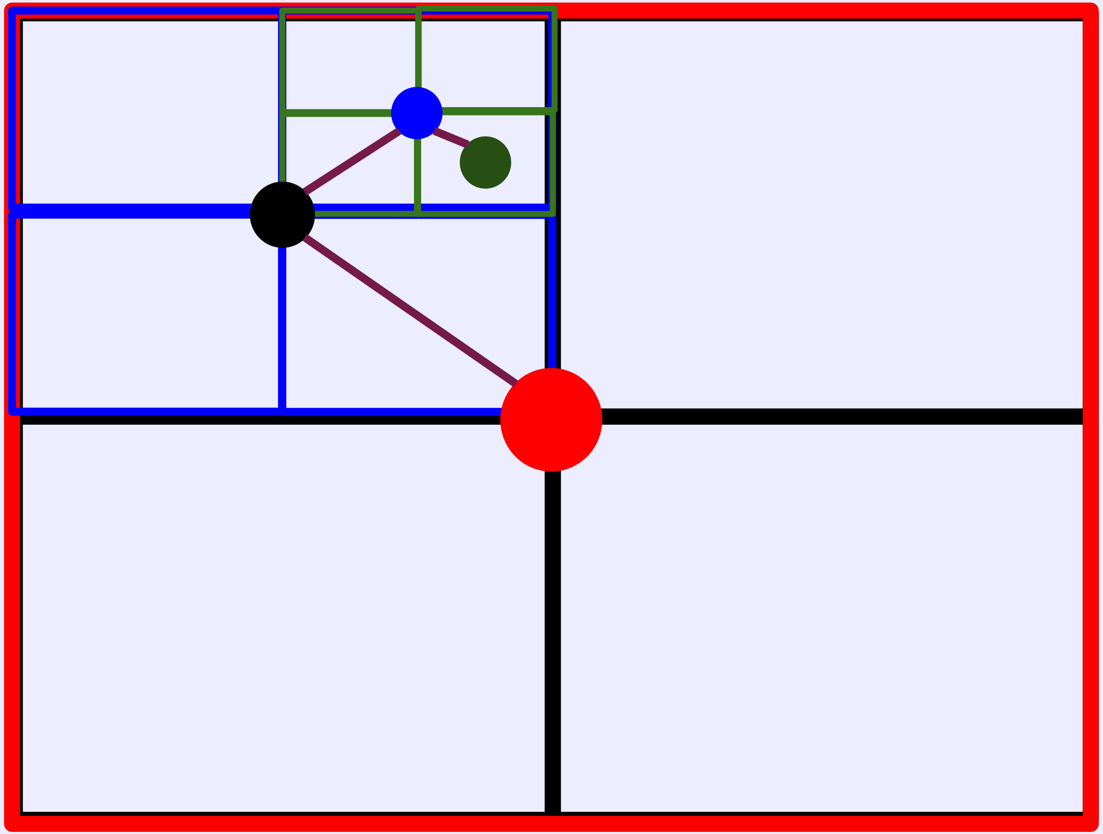
Quad Trees for Fast Intersection Tests



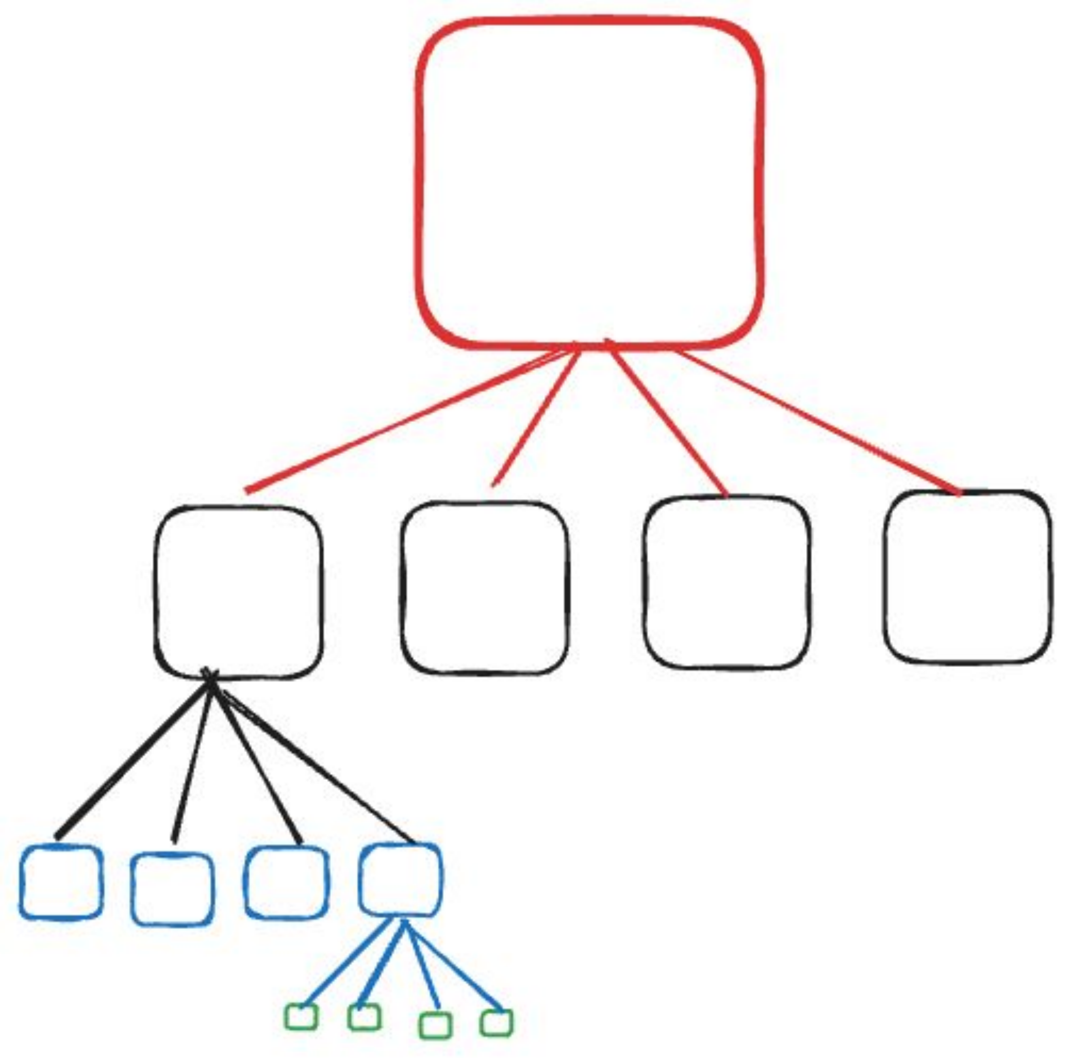
Quad Trees for Fast Intersection Tests



Holy Shit, It’s a Tree!

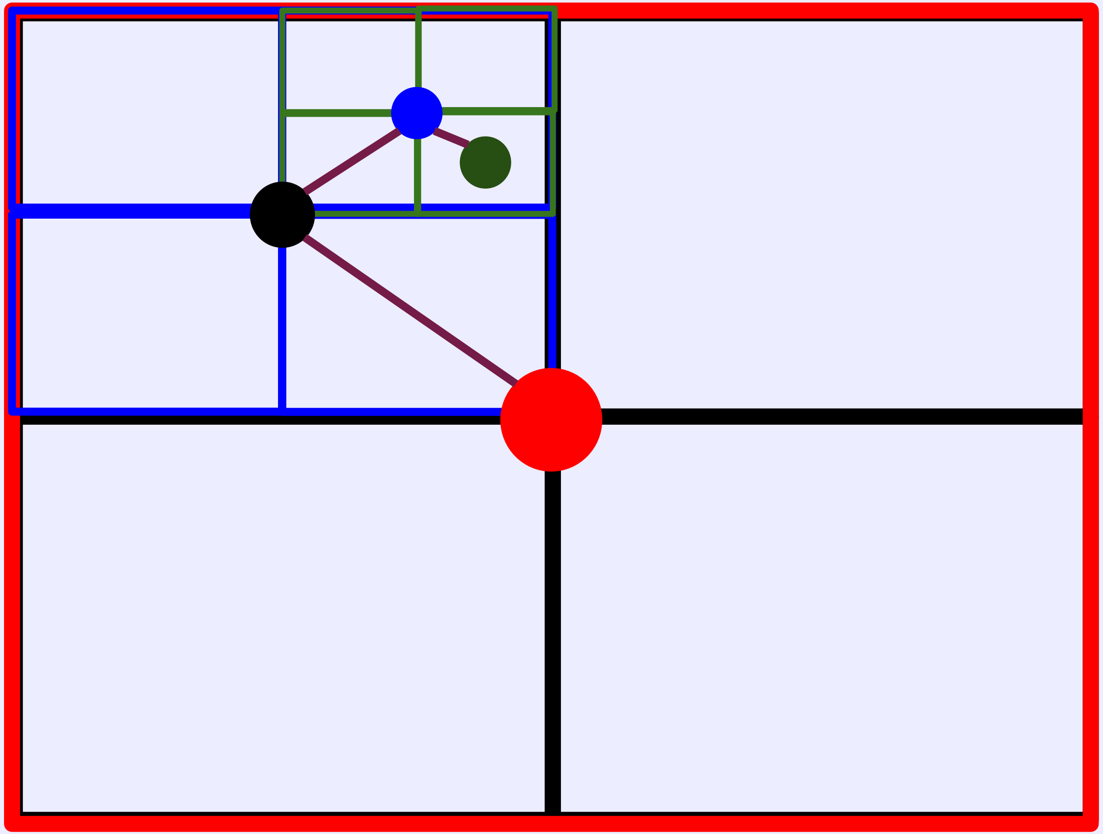


Holy Shit, It’s a Tree!



It’s Actually A Tree Pretty Often

Quad Trees for Fast Intersection Tests



Quad Trees Pseudo Code

```
def find_intersecting(self, q1 : Rect) -> list:
    to_ret = []
    if self.contained is not None:
        for q2 in self.contained:
            if q2.intersects(q1):
                to_ret.append(q2)
    else:
        if self.northwest.contains(q1):
            to_ret += self.northwest.find_intersecting(q1)
        if self.southwest.contains(q1):
            to_ret += self.southwest.find_intersecting(q1)
        if self.northeast.contains( q1):
            to_ret += self.northeast.find_intersecting(q1)
        if self.southeast.contains(q1):
            to_ret += self.southeast.find_intersecting(q1)
    return to_ret
```

Quad Trees Pseudo Code

```
def find_intersecting(self, q1 : Rect) -> list:
    to_ret = []
    if self.contained is not None:
        for q2 in self.contained:
            if q2.intersects(q1):
                to_ret.append(q2)
    else:
        if self.northwest.contains(q1):
            to_ret += self.northwest.find_intersecting(q1)
        if self.southwest.contains(q1):
            to_ret += self.southwest.find_intersecting(q1)
        if self.northeast.contains( q1):
            to_ret += self.northeast.find_intersecting(q1)
        if self.southeast.contains(q1):
            to_ret += self.southeast.find_intersecting(q1)
    return to_ret
```

Find Intersecting

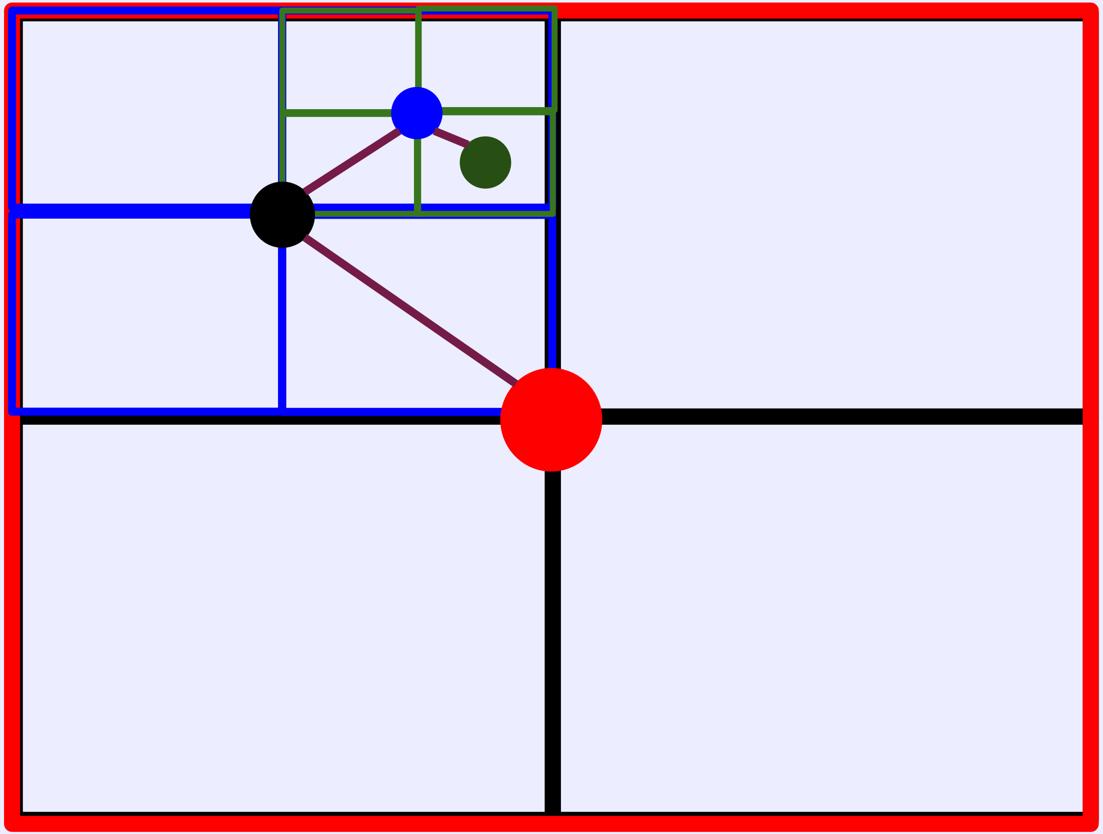
Algorithm	Best	Worst	Average
naive	n^2	n^2	n^2
Quad trees	log(n)	n log(n)	

Quad Trees for Fast Intersection Tests

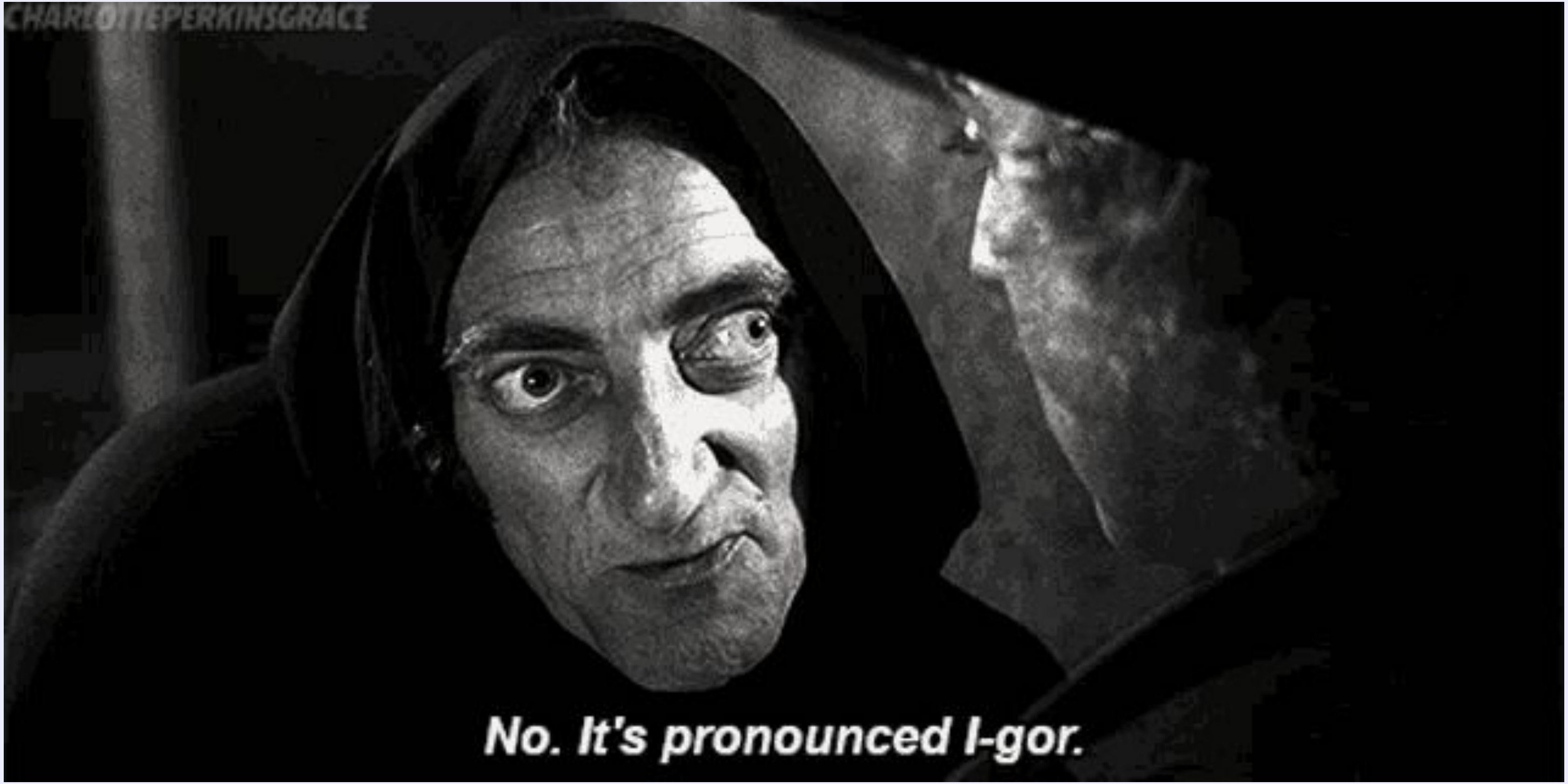
Best?

Worst?

Expected?

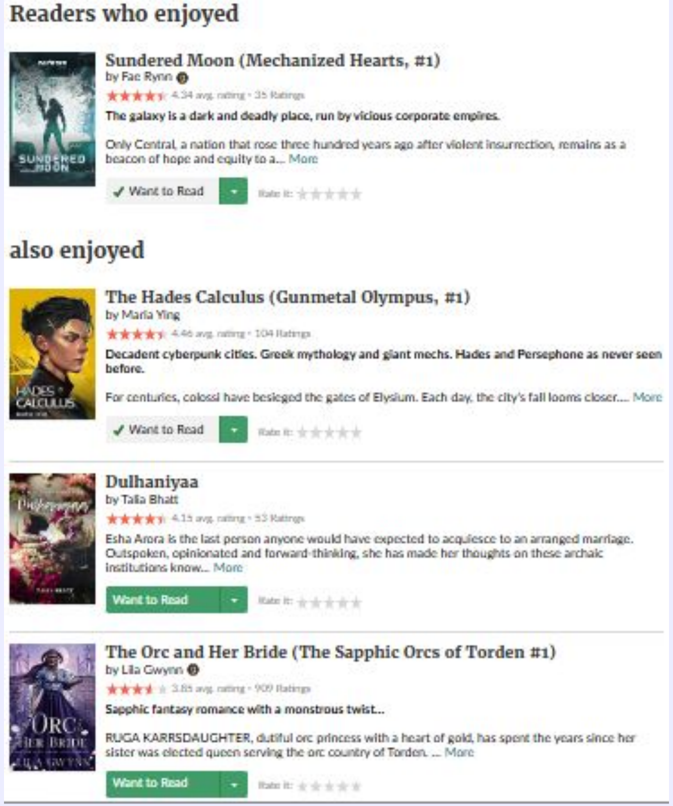


Quad Trees Runtime Analysis



Pause & Reflect

Finding Similar Things in Large Sets



K-Dimension Trees (KD Trees)

K-Dimension Trees (KD Trees)

```
def distance(point1, point2):
    Node.test_count += 1
    sum = 0
    for feat1, feat2 in zip(point1, point2):
        sum += (feat1 - feat2)**2
    return sqrt(sum)
```

```python
def distance(point1, point2):
    Node.test_count += 1
    sum = 0
    for feat1, feat2 in zip(point1, point2):
        sum += (feat1 - feat2)**2
    return sqrt(sum)
```

```python
def find_nearest_neightbor(self, point, current_best=None):
    _, current_best_distance = current_best
    dist = self.dimm_distance(self.median, point)
    if self.leaf:
        if current_best is None:
            return (self.median,  dist)
        else:
            if current_best_distance > dist:
                return (self.median, dist)
            else:
                return current_best
    else:
        if dist < current_best_distance:
            return self.left.find_nearest_neightbor(point,
                                        current_best=(self.median, dist))
        else:
            return self.right.find_nearest_neightbor(point,
                                            current_best=current_best)
```

```python
def find_nearest_neighbtor(self, point, current_best=None):
    _, current_best_distance = current_best
    dist = self.dimm_distance(self.median, point)
    if self.leaf:
        if current_best is None:
            return (self.median,  dist)
        else:
            if current_best_distance > dist:
                return (self.median, dist)
            else:
                return current_best
    else:
        if dist < current_best_distance:
            return self.left.find_nearest_neightbor(point,
                                    current_best=(self.median, dist))
        else:
            return self.right.find_nearest_neightbor(point,
                                    current_best=current_best)
```

```python
def dimm_distance(self, point1, point2):
    Node.test_count += 1
    dimm = self.max_axis % self.depth
    return sqrt((point1[dimm] - point2[dimm])**2)
```

KD Tree Run Time Analysis

Best	Worst	Average

KD Tree Run Time Analysis

Best	Worst	Average
1		

KD Tree Run Time Analysis

Best	Worst	Average
1	n	

KD Tree Run Time Analysis

Best	Worst	Average
1	n	log(n)

A Brief Pause

Hash Tables Are Pretty Neat, Actually

Hash Tables Are Pretty Neat, Actually

Hash Tables Are Pretty Neat, Actually

- Hashing
- Table
- Buckets
- The Stored Elements

Hash Tables Are Pretty Neat, Actually

- Hashing
  - How do we map the object to some sortable, lookupable thing (probably an integer)
- Table
  - The Storage Area
- Buckets
  - A drawer in our vertical filing cabinets
- The Stored Elements
  - Files, in the cabinets

```python
class Bucket():
    test_count = 0
    def __init__(self):
        self.contents = []

    def add(self, key, element):
        self.contents.append(Container(key, element))

    def find(self, key):
        for container in self.contents:
            Bucket.test_count += 1
            if container.key == key:
                return container.element
        return None
```

```python
class Bucket():
    test_count = 0
    def __init__(self):
        self.contents = []

    def add(self, key, element):
        self.contents.append(Container(key, element))

    def find(self, key):
        for container in self.contents:
            Bucket.test_count += 1
            if container.key == key:
                return container.element
        return None
```

```python
class HashTable():

    def __init__(self, hash_function, max_bucket=100):
        self.max_bucket = max_bucket
        self.buckets = []
        for _ in range(self.max_bucket):
            self.buckets.append(Bucket())
        self.hash = hash_function
```

```python
def retrieve(self, key):
    HashTable.test_count += 1
    bucket = self.buckets[key % self.max_bucket]
    return bucket.find(key)

def add(self, element, key=None):
    HashTable.test_count += 1
    if key is None:
        key = self.hash(element)
    bucket = self.buckets[key % self.max_bucket]
    bucket.add(key, element)
    return key
```

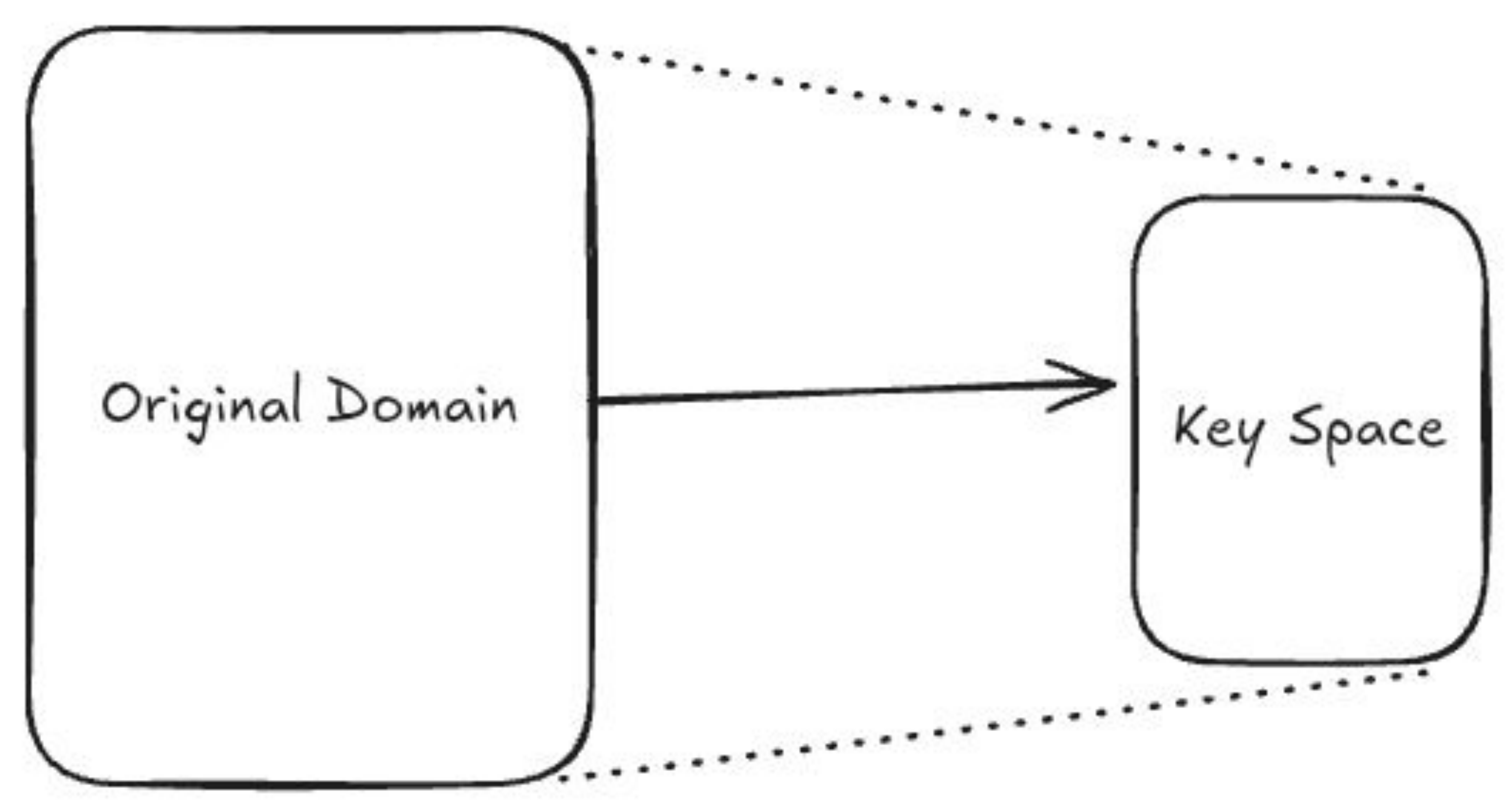
```python
def retrieve(self, key):
    HashTable.test_count += 1
    bucket = self.buckets[key % self.max_bucket]
    return bucket.find(key)

def add(self, element, key=None):
    HashTable.test_count += 1
    if key is None:
        key = self.hash(element)
    bucket = self.buckets[key % self.max_bucket]
    bucket.add(key, element)
    return key
```

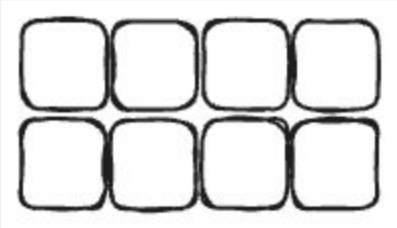
```
    def extend(self):
        new_max = self.max_bucket * 2
        new_buckets = []
        for i in range(new_max):
            new_buckets.append(Bucket())
        for bucket in self.buckets:
            for container in bucket.contents:
                # We can end-run the insertion function
                # because we already have the wrapped object
                HashTable.test_count += 1
                new_buckets[container.key %
new_max].contents.append(container)
        self.buckets = new_buckets
        self.max_bucket = new_max
```

```
    def extend(self):
        new_max = self.max_bucket * 2
        new_buckets = []
        for i in range(new_max):
            new_buckets.append(Bucket())
        for bucket in self.buckets:
            for container in bucket.contents:
                # We can end-run the insertion function
               # because we already have the wrapped object
                HashTable.test_count += 1
                new_buckets[container.key %
new_max].contents.append(container)
        self.buckets = new_buckets
        self.max_bucket = new_max
```

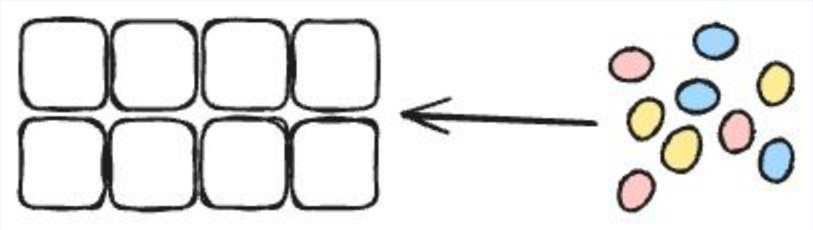
Hashing



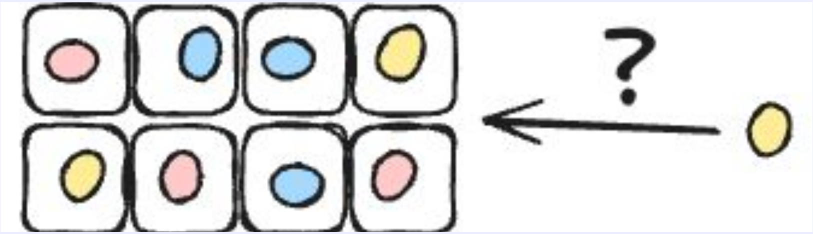
What’s a Hash Collision?



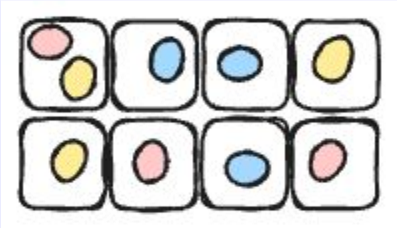
What’s a Hash Collision?



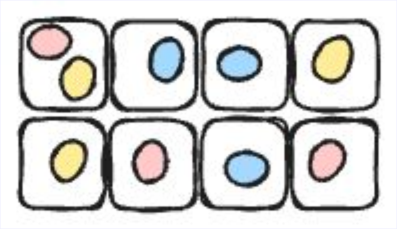
What’s a Hash Collision?



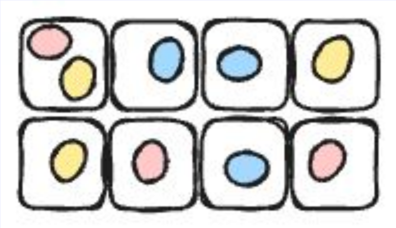
What's a Hash Collision?

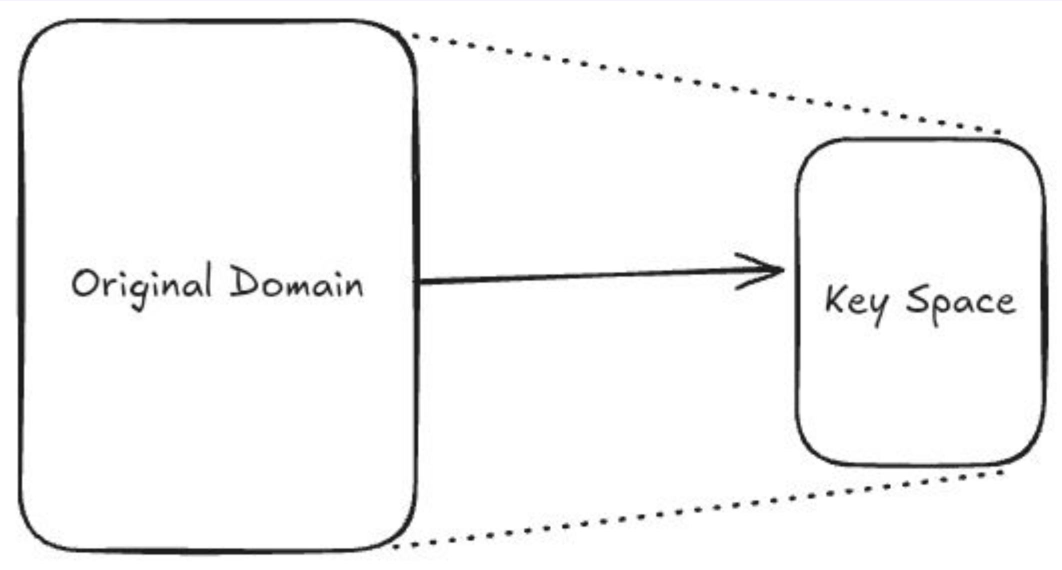


Pigeon Hole Principle

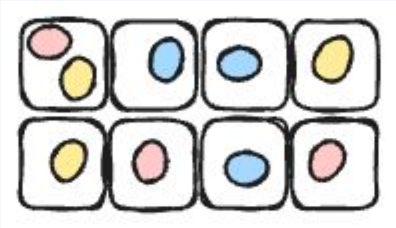


Pigeon Hole Principle

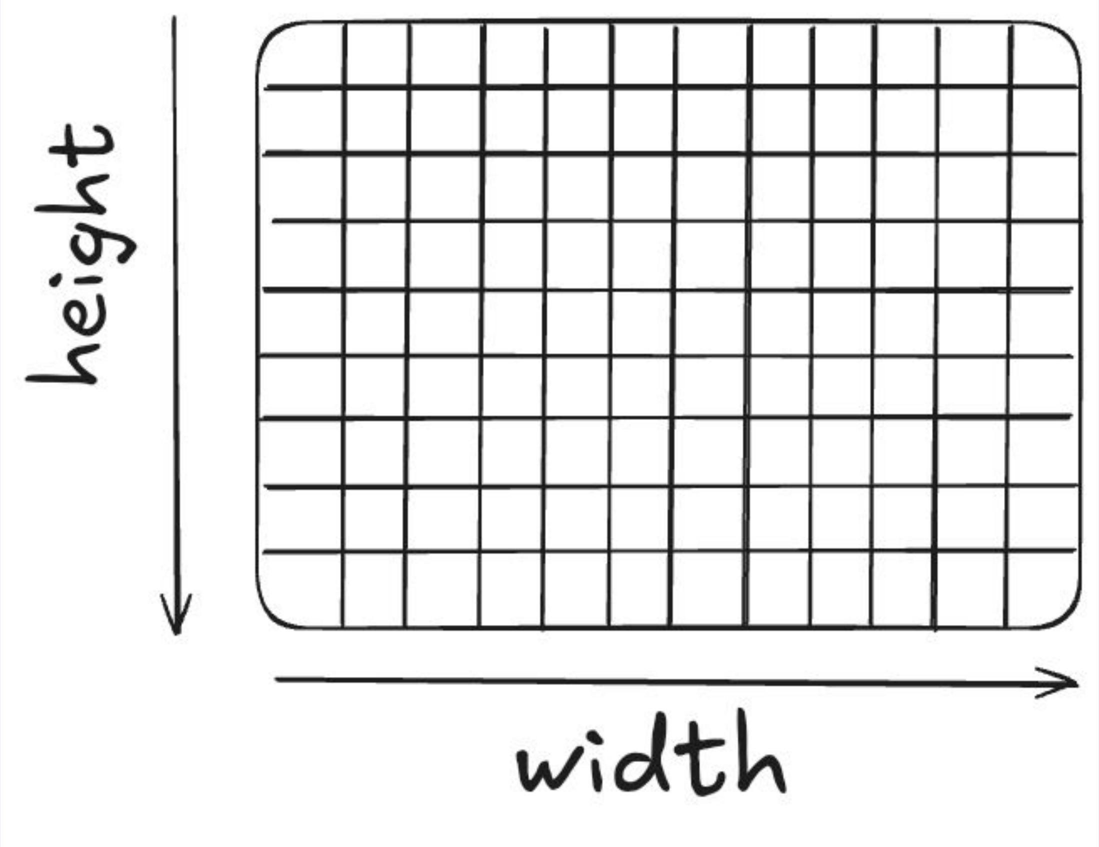




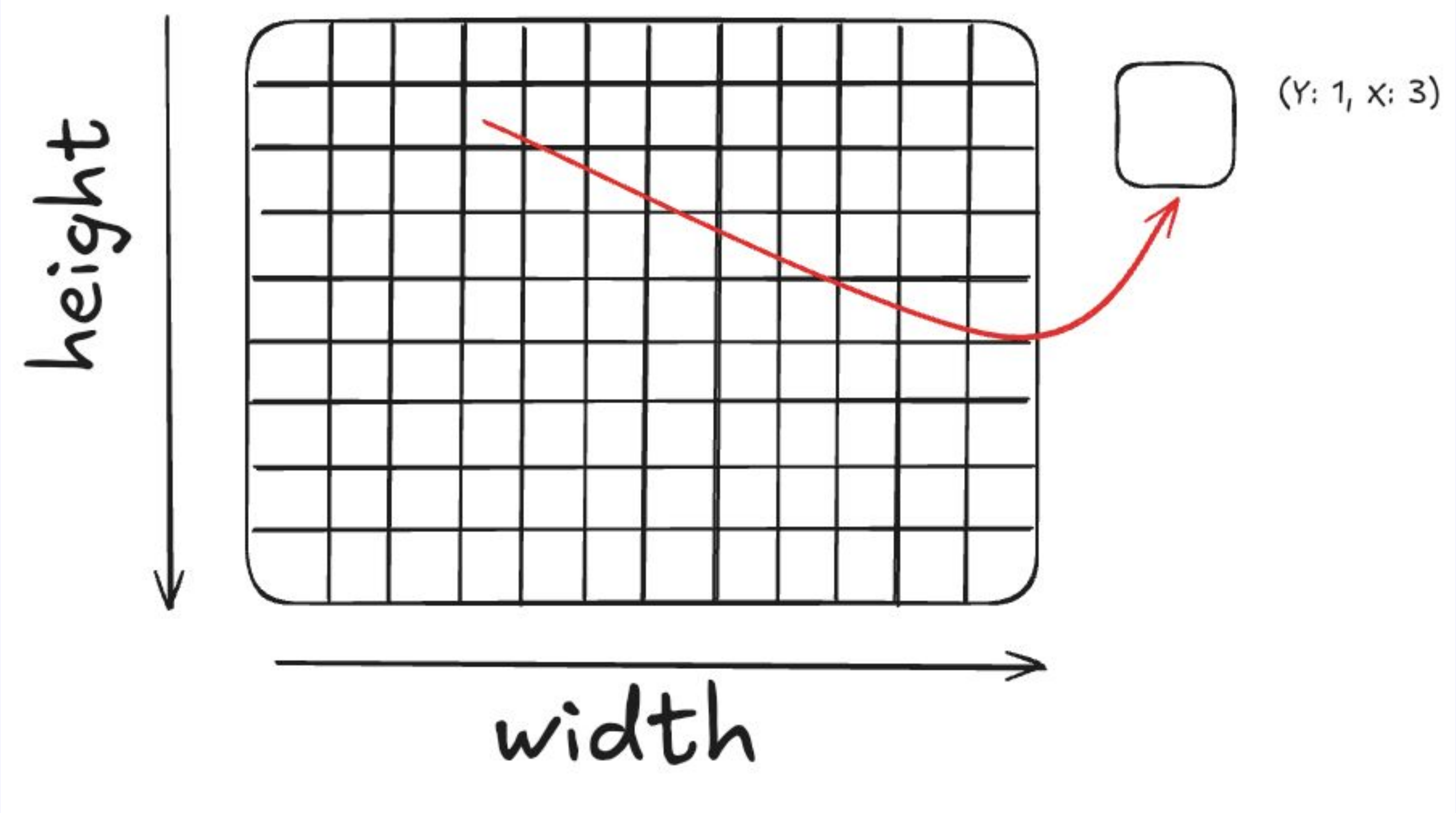
Pigeon Hole Principle



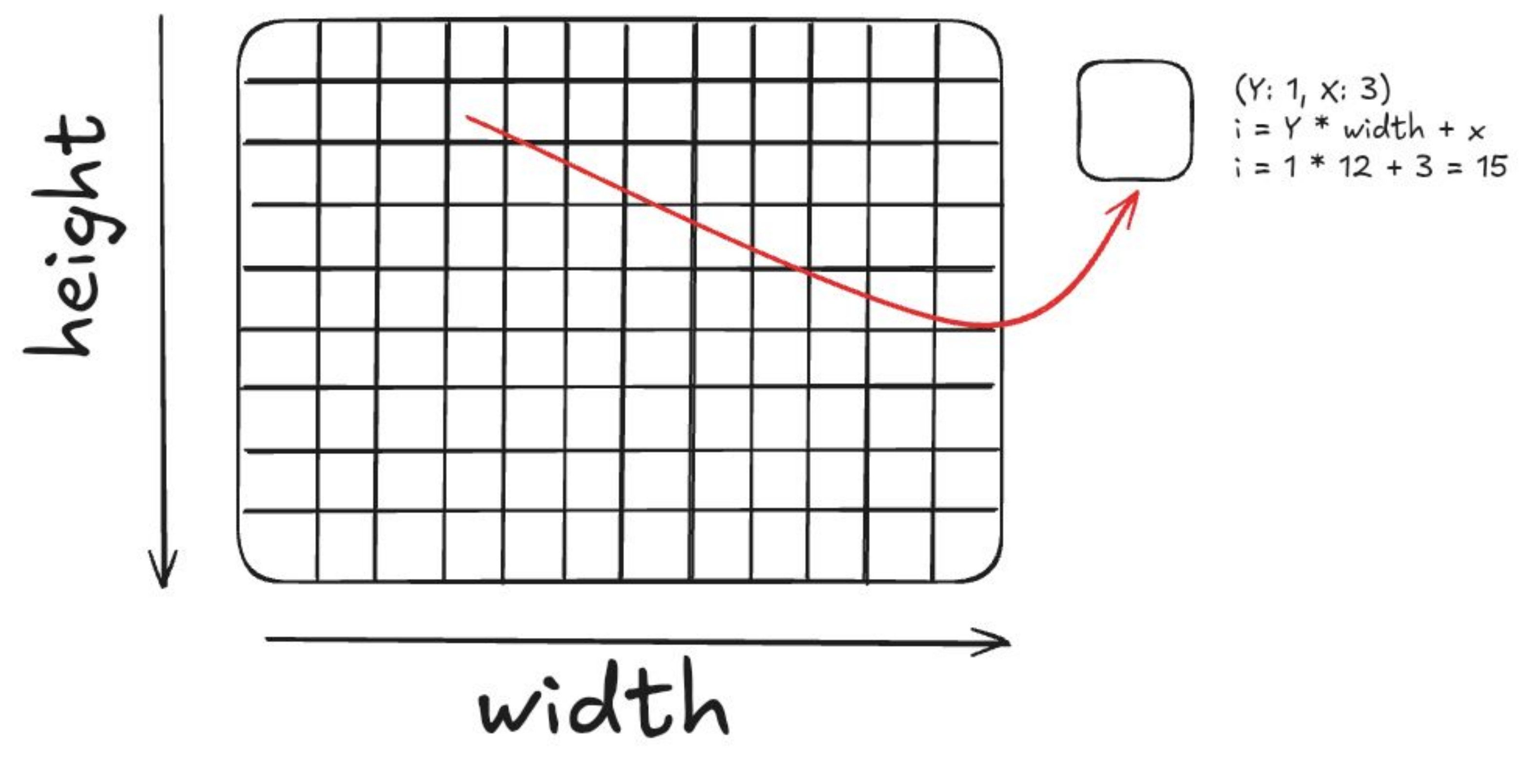
How Do We Hash?



How Do We Hash?



How Do We Hash?



What’s Differt Between Perfect & Imperfect Hashes?

That’s Good Intuition, How Do We Evaluate It?

```python
class Bucket():
    test_count = 0
    def __init__(self):
        self.contents = []

    def add(self, key, element):
        self.contents.append(Container(key, element))

    def find(self, key):
        for container in self.contents:
            Bucket.test_count += 1
            if container.key == key:
                return container.element
        return None
```

```
    def extend(self):
        new_max = self.max_bucket * 2
        new_buckets = []
        for i in range(new_max):
            new_buckets.append(Bucket())
        for bucket in self.buckets:
            for container in bucket.contents:
                # We can end-run the insertion function
                # because we already have the wrapped object
                HashTable.test_count += 1
                new_buckets[container.key %
new_max].contents.append(container)
        self.buckets = new_buckets
        self.max_bucket = new_max
```

Well, What Does That Mean?

- Ideally, the hash function “works”
- Buckets have a small number (1-ish) elements
- Lookup and Removal Are O(1)
- What About Extend though?
  - It should be vanishingly rare

Hashtable Run Time Analysis

- What Kind of Hashing Performance Should We Expect?
- Best Case Analysis
- Worst Case Analysis
- Expected Case Analysis
- Parameterized Analysis

The Final Stretch

Image Segmentation (Graph-Cut)

- How does the lasso tool work?
- How can we segment foreground and background?
- What pixels form a coherent shape in this image?

Image Segmentation (Graph-Cut)

How do you think that works?

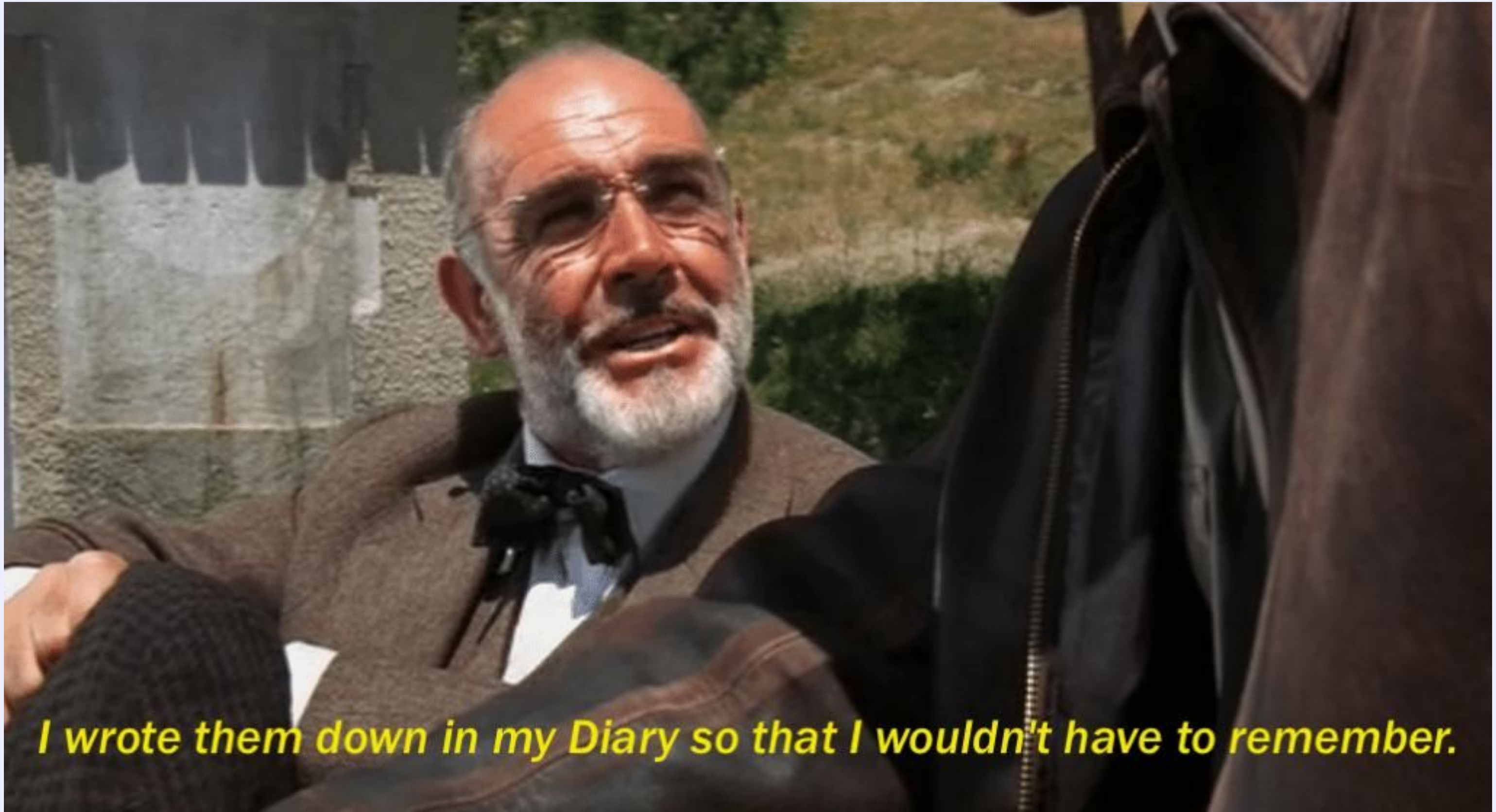
https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm

Wrap-Up

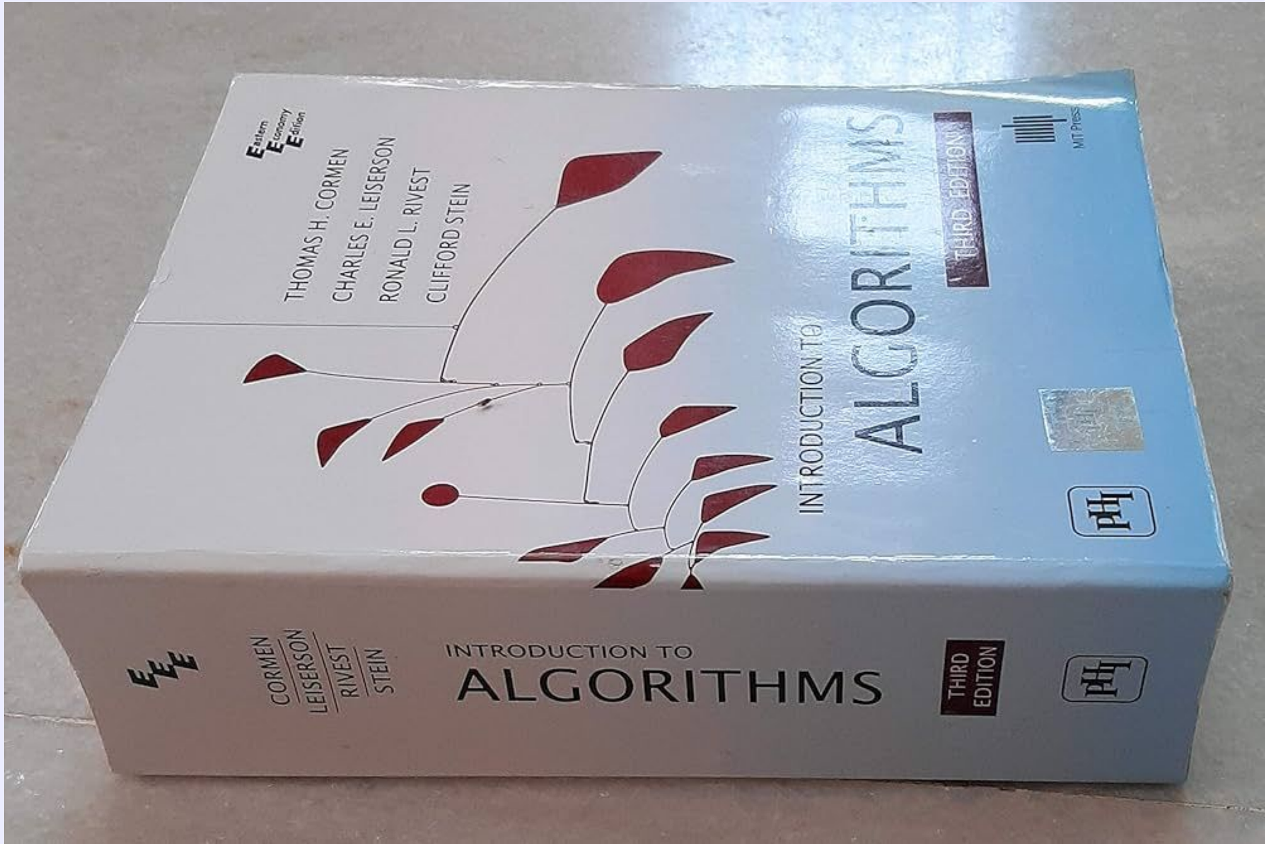
- We looked at a bunch of algorithms and datastructures
- We talked about how to analyze them
- We looked at the ‘grammar’ of algorithms and data structures
- We slogged through some of the bigger data structures and algorithms you use regularly

If Today Wasn’t Enough

… or if you just want a useful desk reference



BUY ~~MY~~ THIS BOOK



Also MAYBE This One

https://numerical.recipes/