The Egg Drop Problem

- How strong are egg shells? Let's do some science!
- There is a building with 10000 floors
- Assume that:
  - The egg stays intact when dropped from floor 0
  - The egg breaks when dropped from floor 9999
  - An egg breaks when dropped from floor k or greater
  - But stays intact when dropped from floor less than k
- What is k?

The Egg Drop Problem: Ideas

- Try dropping at floor 0
  - If it breaks, we know that $k = 0$
  - If it stays intact, we know that $k > 0$
- Try dropping at floor 1
  - If it breaks, we know that $k = 1$
  - If it stays intact, we know that $k > 1$
- Try dropping at floor 2
  - If it breaks, we know that $k = 2$
  - If it stays intact, we know that $k > 2$
- Etc.

The Egg Drop Problem: Ideas

- Slow in the worst case!
- Too tiring to go up and down (even with an elevator)!
- We want to know k in as few drops as possible

The Egg Drop Problem: Ideas

- We know k is between 0 and 9999
- Try some floor m (between 0 and 9999)
- If it breaks, we know that k is between 0 and m
  - Why?
- If it stays intact, we know that k is between $m + 1$ and 9999
- What m should we choose?

The Egg Drop Problem: Ideas

- Try the middle floor $m \sim 4999$
- If it breaks, we know that k is between 0 and 4999
  - Why?
- If it stays intact, we know that k is between 5000 and 9999

The Egg Drop Problem: Solution

- In general, suppose we know k is between L and R
- Try floor $\frac{L+R}{2}$ (between L and R)
- If it breaks, we know that k is between L and $\frac{L+R}{2}$
  - Why?
- If it stays intact, we know that k is between $\frac{L+R}{2} + 1$ and R
- Repeat until k is in a range with exactly one number
- Then k has to be that number!

Will This Be Fast?

- At every step, the number of floors being searched is halved
- How many times can a number n be halved before it reaches 1?
- We call this value $\log n$
- https://www.desmos.com/calculator/tbm5cuxxoy
  - $\log n$ grows *very* slowly relative to n
  - Even for large n, $\log n$ is small

n	$\log_2 n$
2	1
10	3.3219…
100	6.6439…
1000	9.9658…
$10{,}000$	13.2877…
$100{,}000$	16.6096…
10^6	19.9315…
10^9	29.8973…
10^{12}	39.8631…
10^{15}	49.8289…

Let’s Call This Strategy…

- The Thanos Strategy

Searching

- New problem: find the index of an item in a list
  - Assume item to find always exist: easy to modify later to handle case where item might not exist
- In general, $\Theta(n)$ time required
- But if list is sorted, there's a faster way

Thanos Search

1. Look at the middle element
2. If target item = middle element, we found it
3. If target item < middle element, recursively search left
4. If target item > middle element, recursively search right

Find **17** from the array of 10 elements

2	5	7	8	11	13	17	19	23	29

2	5	7	8	11	13	17	19	23	29

2	5	7	8	11	13	17	19	23	29

2	5	7	8	11	13	17	19	23	29

2	5	7	8	11	13	17	19	23	29

Will This Be Fast?

- At every step, the size of the list being searched is halved
- How many times can a number n be halved before it reaches 1?
- Thanos search is $\Theta(\log n)$

Let's Try...

```
def tsearch(a, target):
    if len(a) == 1:
        return ???
    else:
        # something with a[:mid] and a[mid+1:] here
```

Thanos Search Implementation

- Since we need to return the index, we need to "remember" more info
- Idea: maintain the true starting index of the list we are searching

Thanos Search Implementation (in Python)

```python
def tsearch(a, start, target):
    if len(a) == 1:
        return start
    else:
        mid = ???
        if target == a[mid]:
            return ???
        elif target < a[mid]:
            return tsearch(a[:mid], ???, target)
        else: # target is surely > a[mid]
            return tsearch(a[mid+1:], ???, target)
```

Thanos Search Implementation (in Python)

```python
def tsearch(a, start, target):
    if len(a) == 1:
        return start
    else:
        mid = len(a) // 2
        if target == a[mid]:
            return start + mid
        elif target < a[mid]:
            return tsearch(a[:mid], start, target)
        else: # target is surely > a[mid]
            return tsearch(a[mid+1:], start + mid+1, target)
```

Now Let's Test Our Code

```python
# put tsearch definition here

n = 100000
a = [i for i in range(n)]
it_works = all(tsearch(a, 0, i) == i for i in range(n))
print(it_works)
```

It's Too Slow?

- $\log 1{,}000{,}000 \times 1{,}000 \leq 10^8$
- We expected it to finish instantly

The Issue

- Creating list slices takes too long
  - The original list must be copied
- Every step of the search must be $O(1)$ so that entire search is $O(\log n)$
- Idea: don't literally cut lists in half
  - Keep track of sublist being considered by keeping start and *end* indices

Thanos Search Implementation (in Python)

```python
def tsearch(a, start, end, target):
    if end - start == 1:
        return start
    else:
        mid = (start + end) // 2
        if target == a[mid]:
            return mid
        elif target < a[mid]:
            return tsearch(a, start, mid, target)
        else: # target is surely > a[mid]
            return tsearch(a, mid+1, end, target)
```

Now, Try It Again

```
# put tsearch definition here

n = 100000
a = [i for i in range(n)]
it_works = all(tsearch(a, 0, n, i) == i for i in range(n))
print(it_works)
```

Thanos Search Implementation (in C++)

- Passing vectors to function calls makes a copy of the vector
  - Declare the vector globally and avoid passing it around
  - Or, use pass by reference (if you know it)

Thanos Search Implementation (in C++)

```
int tsearch(int start, int end, int target) {
    if(end - start == 1) {
        return start;
    } else {
        int mid = (start + end) / 2;
        if(target == a[mid])
            return mid;
        else if(target < a[mid])
            return tsearch(start, mid, target);
        else
            return tsearch(mid+1, end, target);
    }
}
```

Practice Problems

- https://progvar.fun/problemsets/binary-search

Sorting

- Selection sort: to sort a list of items, take the smallest item, put it in front, and *sort the remaining items*
- How fast is it?
  - At every step, need to search for the smallest item
  - Takes time proportional to length of remaining list
  - $n + n - 1 + n - 2 + \cdots + 1 = O(n^2)$

Is There a Faster Way?

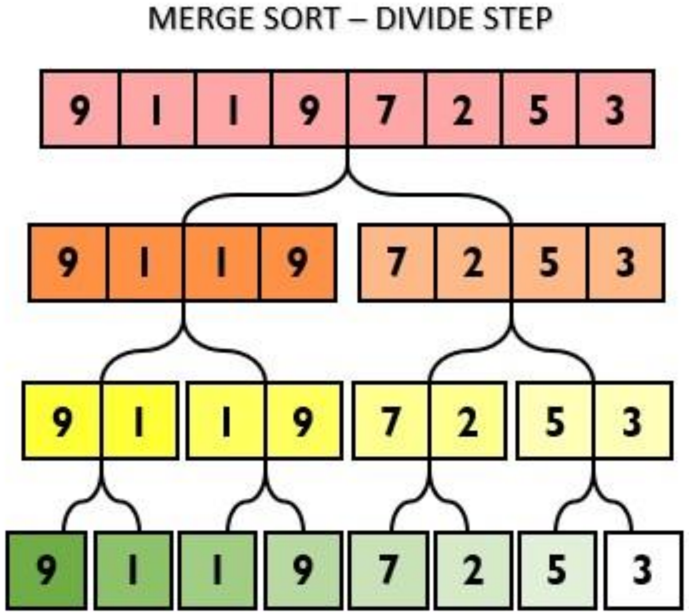
Inspiration: Copy Thanos Search Strategy

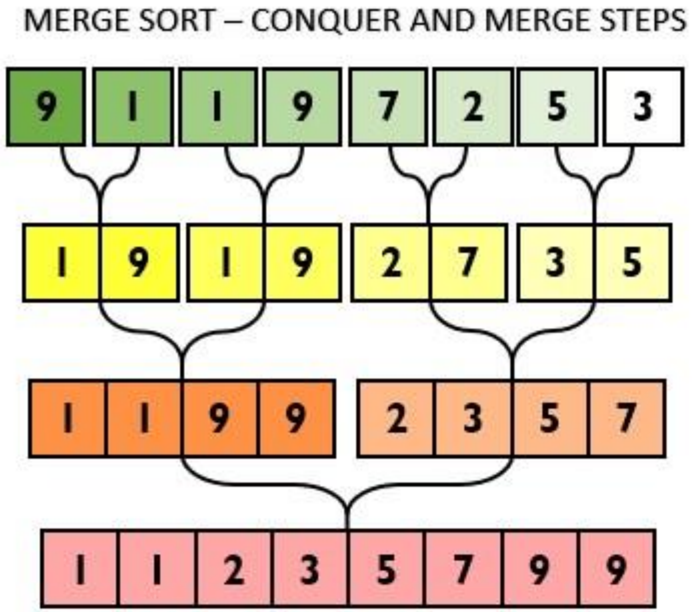
- Right now: split list into 1 and $n-1$
  - Feels like linear search
- Let's try: split list into $\frac{n}{2}$ and $\frac{n}{2}$

Sorting Fast

- **Divide** list into two halves
- **Recursively** sort each half
- **Combine** the two sorted halves
- Also known as **merge sort**

Sorting Fast: Merge Sort





Sorting Fast: Merge Sort

```
def sort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a) // 2
        return merge(sort(a[:mid]), sort(a[mid:]))
```

How to Combine Two Sorted Halves?

- Boys of a class are lined up by height
- Girls are separately lined up by height
- Form a line with everyone sorted by height
- Who can possibly be at the front of the line?
- At any point in time, which people do you need to look at to know who goes next in line?

How to Combine Two Sorted Halves?

- Compare *front elements* of two halves
- Remove the smaller element and put it in the next unoccupied position of sorted list
- Then, *repeat*

How to Combine Two Sorted Halves?

```
def merge(L, R):
    if min(L[0], R[0]) == L[0]:
        return [L[0]] + merge(L[1:], R)
    else:
        return [R[0]] + merge(L, R[1:])
```

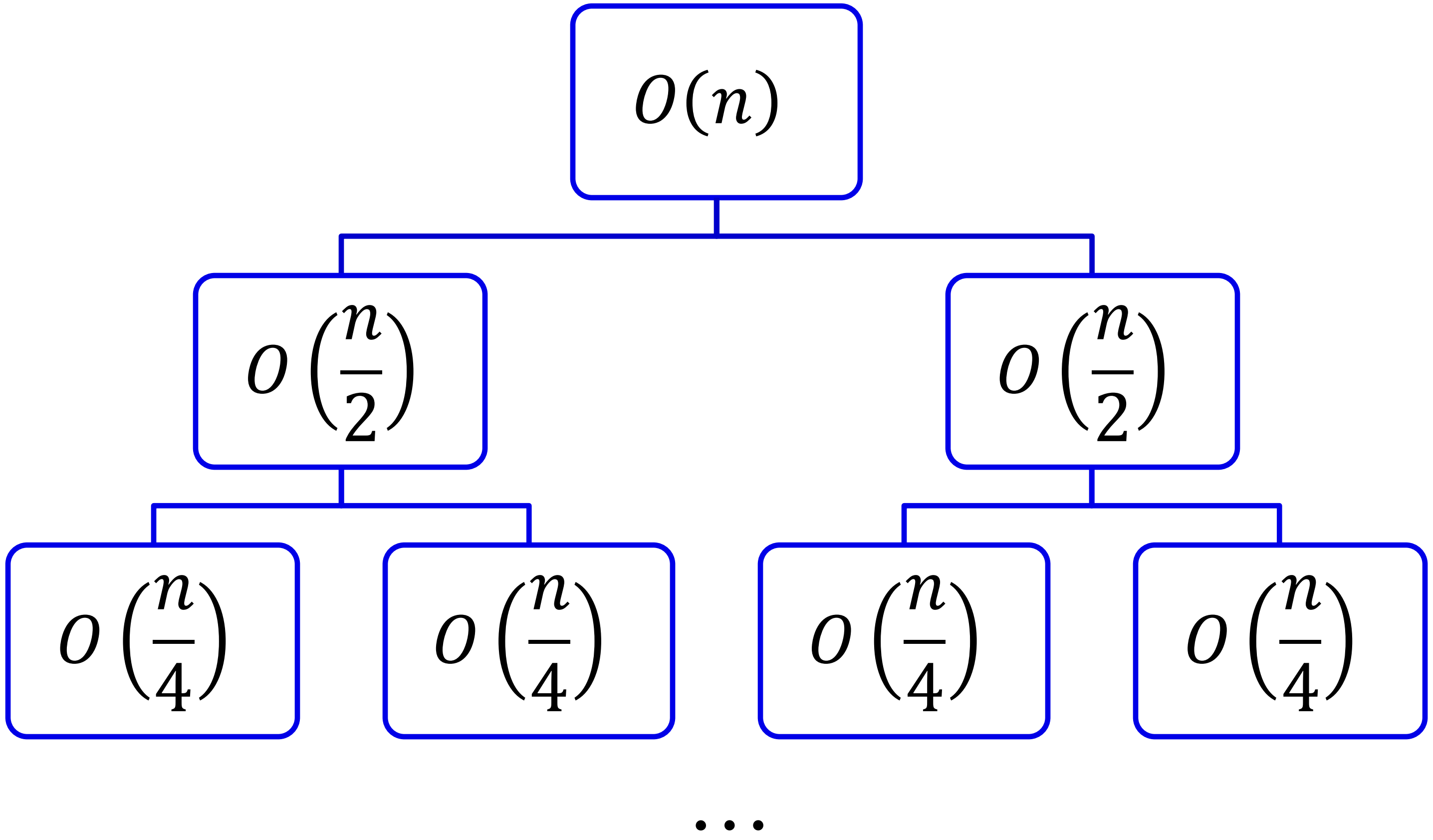
Improving the Implementation

- Handle cases when L or R becomes empty
- Slicing and copying lists is expensive
  - Like thanos search implementation, there's a way to implement merge sort using start and end indices
  - Enables `merge` to run in $O\big(len(L) + len(R)\big)$ time
  - Enables cutting list in half to be done in $O(1)$ time
- You can try this at home by yourself

Analysis

- Every recursive step requires $O\left(len(a)\right)$ non-recursive work to merge

Diagram the Amount of Non-Recursive Work at Each Call



Analysis

- Every level of this tree has the same total. What is it?
  - $O(n)$
- So the total time is just $O(n)$ times the number of levels
- How many levels are there?
  - This is just the number of times you divide n by 2 until you reach the base case $n = 1$
  - AKA $O(\log n)$
- $O(n \log n)$ total time

Note

- You almost never have to write your own sort function
  - C++: Just do `sort(a.begin(), a.end())`
  - Python: `a.sort()` or `sorted(a)`
- Works in $O(n \log n)$ time using similar ideas
- But knowing how it works is still important!

The General Idea: Divide-and-Conquer

- **Divide**: split problem into parts
- **Conquer**: (recursively) solve each part independently
  - Some parts may be skipped if they don't contribute to the answer
- (Optional) **Combine**: combine answers from each part

Thanos Search

- **Divide:** look at the middle element
- **Conquer:**
  - If target item = middle element, we found it
  - If target item < middle element, recursively search left
  - If target item > middle element, recursively search right

Merge Sort

- **Divide:** list into two halves
- **Conquer:** recursively sort each half
- **Combine:** the two sorted halves

Exponentiation

- x^n is just repeated multiplication
- But doing it this way takes $O(n)$ time, slow when n is large
- Is there a faster way?

Try It!

- Compute 13^{1024} “by hand”
- You may use Python interactive shell and the $*$ operator to help you, but nothing else

Exponentiation by Squaring

- Let's assume n is a power of 2
- Then instead of doing $x \cdot x \cdot x \cdot \dots \cdot x$
- We can do $\left(\left(\left((x^2)^2\right)^2\right)^2 \dots\right)^2$

Another Way to Look At It

- **Divide** the exponent n by 2
- **Conquer**: recursively solve for $x^{\frac{n}{2}}$ (only once!)
- **Combine**: reuse same "half" of the answer by squaring it

Analysis

- Each step cuts the exponent in half
- After $\log n$ steps, base case is reached
- $O(\log n)$ time

What If n Not a Power of 2?

- We need to handle even and odd cases

```
def power(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(power(x, n // 2))
    else:
        return square(power(x, n // 2)) * x
```

Fast Exponentation in Python

- `base ** exp` and `pow(base, exp)` and `pow(base, exp, mod)` do something like this behind the scenes to compute the answer fast

A New Problem

- Given a list of integers (can be negative), find a sublist of consecutive elements with the highest sum, and output this sum

Sum = -3

3 -1 4 -1 5 -9 2 -6 5 -3 5 -8

Sum = 1

A New Problem

- Given a list of integers (can be negative), find a sublist of consecutive elements with the highest sum, and output this sum
- There's a complete search solution. What and how fast?
  - $O(n^3)$ by trying all $O(n^2)$ possible sublists and computing the sum of each one in $O(n)$
- Can we do better?

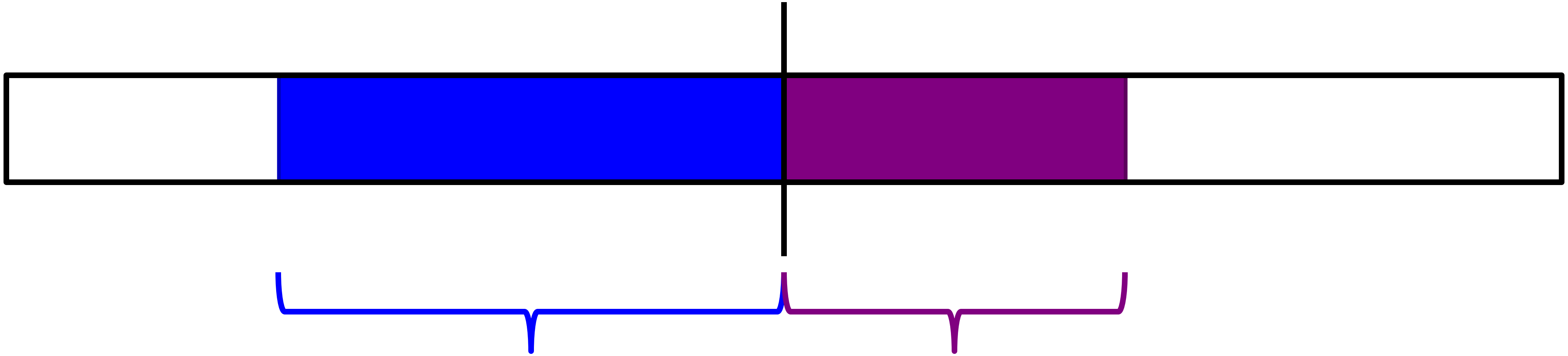
Let's Try Divide and Conquer!

- **Divide:** input list into two halves
- **Conquer:** get the maximum sublist sum from each of the two halves
- **Combine:** the maximum sum for the entire list is one of the following:
  - Maximum of the left half
  - Maximum of the right half
  - ???

Let's Try Divide and Conquer!

- **Divide:** input list into two halves
- **Conquer:** get the maximum sublist sum from each of the two halves
- **Combine:** the maximum sum for the entire list is one of the following:
  - Maximum of the left half
  - Maximum of the right half
  - A sublist that crosses the boundary between the two halves
    - We can try all possible sublists, but this is just fancy brute force and degrades to $O(n^3)$

What Does a Boundary-Crossing Sublist Look Like?



Must be a maximum sublist of left half
BUT must also end at the tail

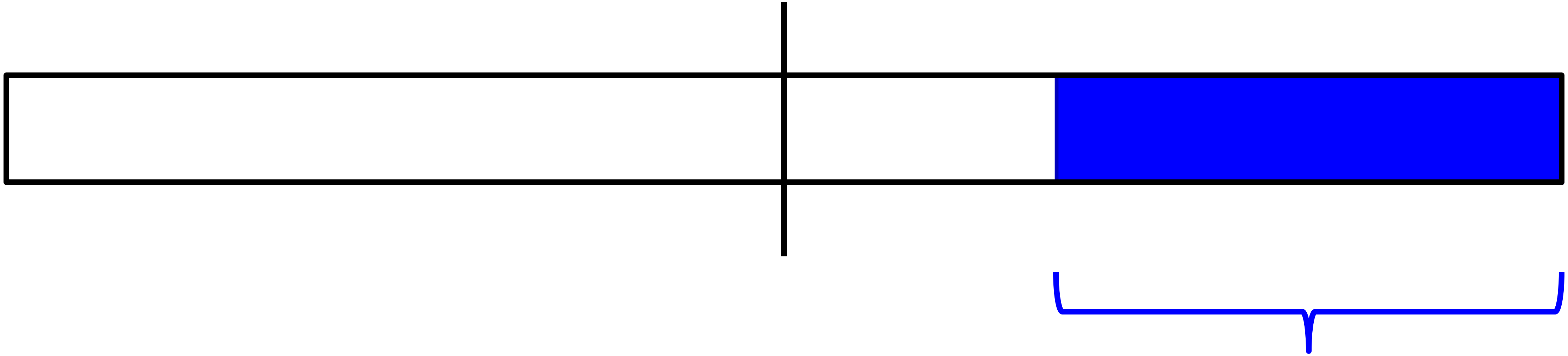
Must be a maximum sublist of right half
BUT must also start at the head

We Need More *Subproblems*

```python
def max_subsum(a):
    mid = len(a) // 2
    return max(
        max_subsum(a[:mid]),
        max_subsum(a[mid:]),
        max_tailsum(a[:mid]) + max_headsum(a[mid:])
    )
```

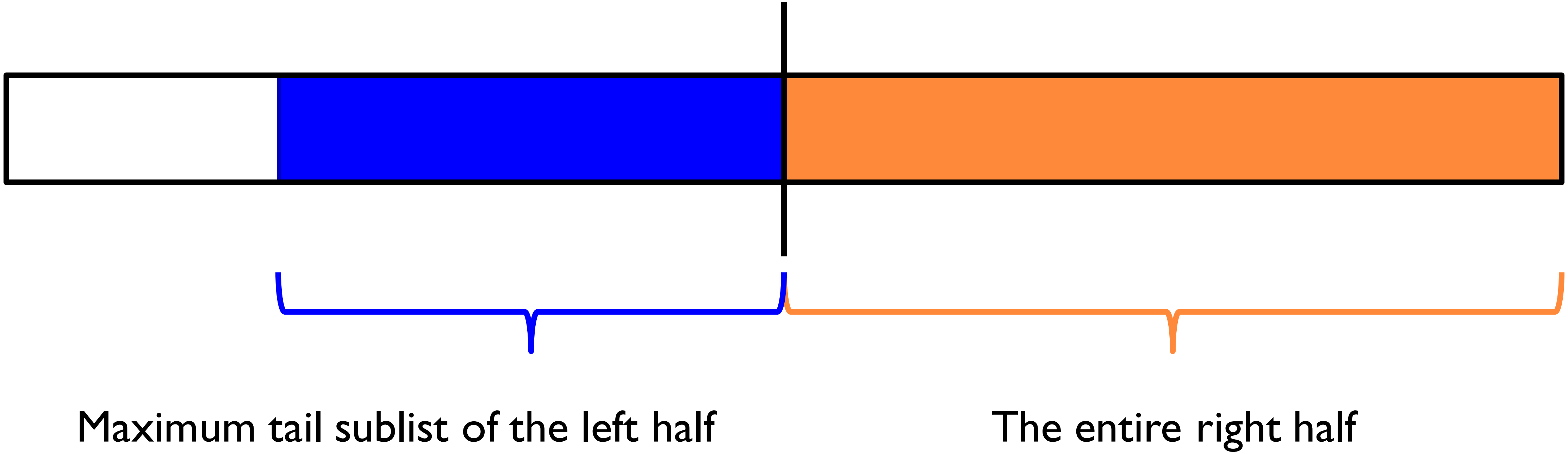
What Does a Maximum Tail Sublist Look Like?

What Does a Maximum Tail Sublist Look Like?



A maximum tail sublist of the right half

What Does a Maximum Tail Sublist Look Like?



What Does a Maximum Tail Sublist Look Like?

```
def max_tailsum(a):
    mid = len(a) // 2
    return max(
        max_tailsum(a[mid:]),
        max_tailsum(a[:mid]) + sum(a[mid:])
    )
```

Similarly

```
def max_headsum(a):
    mid = len(a) // 2
    return max(
        max_headsum(a[:mid]),
        sum(a[:mid]) + max_headsum(a[mid:])
    )
```

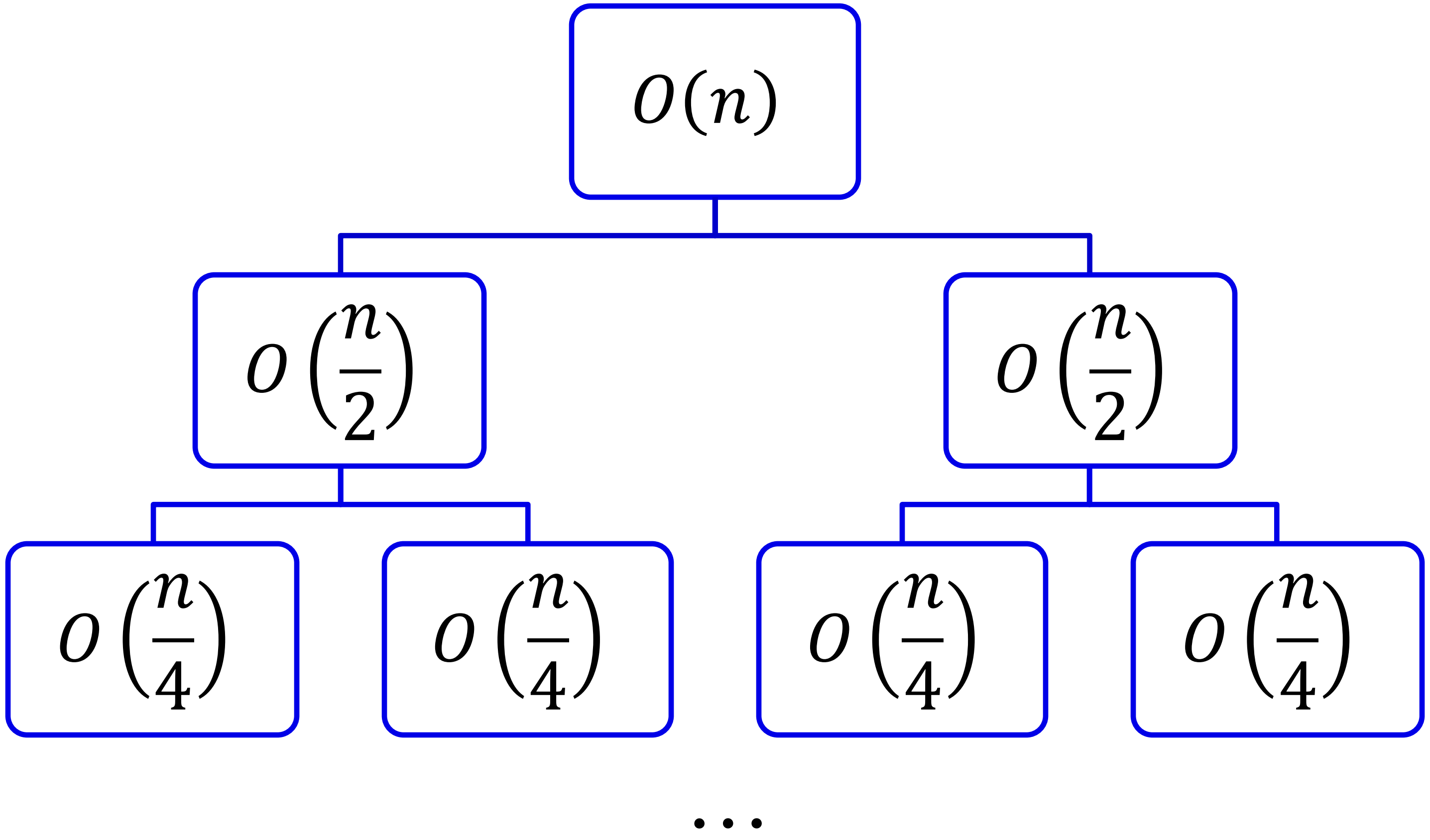
Improving the Implementation

- Slicing and copying lists is expensive: implement everything using start and end indices
  - This makes cutting the list in half doable in $O(1)$
- You can try this at home by yourself

Analysis

- `max_tailsum` and `max_headsum` both require $O\left(len(a)\right)$ non-recursive work to compute `sum`

Analysis



$O(n)$
$O\left(\frac{n}{2}\right)$
$O\left(\frac{n}{2}\right)$
$O\left(\frac{n}{4}\right)$
$O\left(\frac{n}{4}\right)$
$O\left(\frac{n}{4}\right)$
$O\left(\frac{n}{4}\right)$
$\dots$

Analysis

- Every level needs $O(n)$ time
- There are $O(\log n)$ levels until the base case
- $O(n \log n)$ total time for one call of `max_tailsum` or `max_headsum`

Analysis

- `max_subsum` requires $O(len(a)\log len(a))$ non-recursive work to compute `max_tailsum` and `max_headsum`

Analysis

$$O\left(n \log \frac{n}{2}\right)$$

$$O\left(\frac{n}{2} \log \frac{n}{4}\right) \qquad O\left(\frac{n}{2} \log \frac{n}{4}\right)$$

$$O\left(\frac{n}{4} \log \frac{n}{8}\right) \qquad O\left(\frac{n}{4} \log \frac{n}{8}\right) \qquad O\left(\frac{n}{4} \log \frac{n}{8}\right) \qquad O\left(\frac{n}{4} \log \frac{n}{8}\right)$$

$$\cdots$$

Or, Since $O\left(\frac{n}{c}\log\frac{n}{c/2}\right) = O\left(\frac{n}{c}\log n\right)$

$O(n \log n)$

$O\left(\frac{n}{2}\log n\right)$ $O\left(\frac{n}{2}\log n\right)$

$O\left(\frac{n}{4}\log n\right)$ $O\left(\frac{n}{4}\log n\right)$ $O\left(\frac{n}{4}\log n\right)$ $O\left(\frac{n}{4}\log n\right)$

$\cdots$

Analysis

- Every level needs $O(n \log n)$ time
- There are $O(\log n)$ levels until the base case
- $O(n \log^2 n)$ total time for `max_subsum`
- $O(n \log n)$ doable if we limit the amount of work done in one call of `max_tailsum` or `max_headsum` to $O(n)$
  - By analysis similar to merge sort
  - There are simple non-divide-and-conquer $O(n)$ solutions for `max_tailsum` or `max_headsum`
  - Try thinking about it at home!

Practice Problems and Further Study

- https://progvar.fun/problemsets/divide-and-conquer