Contents

1 Sample Chapter 1 – Binary Search **2**

1.1 The Guessing Game . . . 2
1.1.1 Version 1 . . . 2
1.1.2 Version 2 . . . 3
1.2 Binary Search . . . 4
1.3 The Half Optimality . . . 6
1.4 Implementation in Python . . . 6
1.5 Time and Space Complexity . . . 7
1.6 Pattern: Classic Binary Search . . . 8
1.7 Pattern: Binary Search for Boundaries . . . 12
1.8 Pattern: Binary Search on Answers . . . 16

2 Sample Chapter 2 – Recursion **19**

2.1 What is Recursion? . . . 19
2.2 The Lazy Manager . . . 20
2.3 Building a Countdown Function . . . 21
2.4 Base Case and The Cascading Dominoes . . . 24
2.5 Steps for Constructing Recursive Solutions . . . 27
2.6 Recursion Call Stack . . . 29
2.7 Time Complexity . . . 32
2.7.1 Recursion Tree Method . . . 32
2.7.2 Iterative Method . . . 36
2.8 Space Complexity . . . 37
2.9 Worked Examples . . . 38

1 Sample Chapter 1 – Binary Search

Let's say we have a collection of elements that is sorted. A simple example is the array shown below, where the integers are sorted in ascending order:

[1, 4, 6, 12, 15]

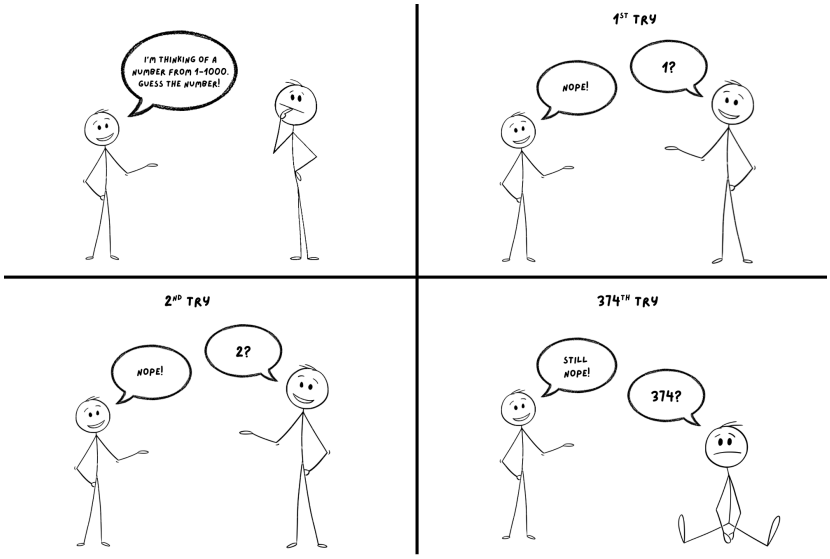
Now let me ask you a question: does the integer 15 exist in the array? In Chapter 1, we said that we can use linear search to answer this question. However, this has a time complexity of O(n) and is quite inefficient. In this chapter, I will introduce you to binary search, a more efficient way of searching in a sorted collection of elements. Then, I will show you how to implement it in Python and, lastly, some common LeetCode patterns involving binary search.

1.1 The Guessing Game

1.1.1 Version 1

One day, your younger brother comes running up to you, practically buzzing with excitement. "I've got a game," he says. "I'm thinking of a number between 1 and 1000. Try to guess it."

Being systematic by nature, you decide to tackle it with a straightforward approach. You guess each number one by one, starting from the beginning. You're confident that you will get there soon enough. But as the guesses drag on – 50, 100, 150 – you start to feel the pace. By the 374th try, you're still nowhere close! This is a long, tedious, and inefficient approach. Clearly, there's something wrong with it.



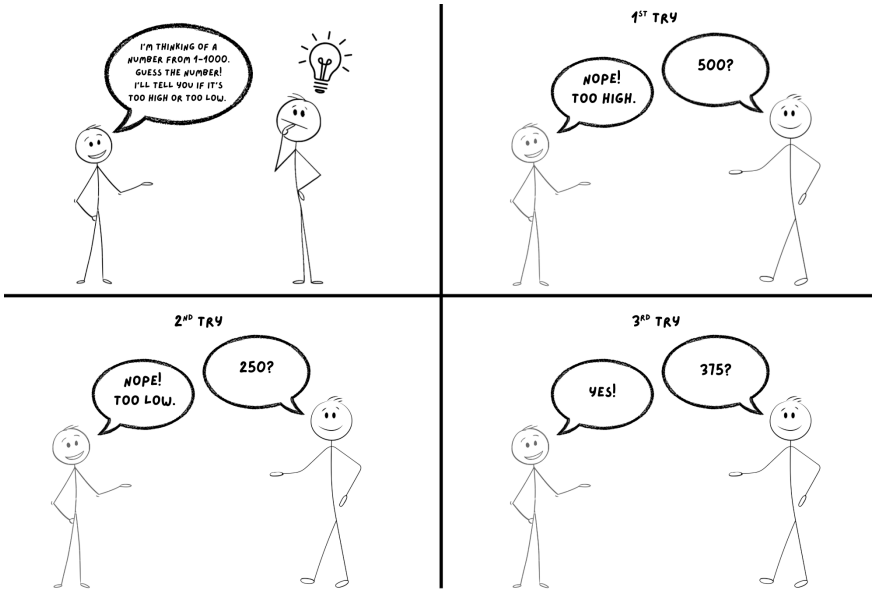
Let's analyze that method. Since we've no clue to which is the correct number, each number essentially has the same probability of being right. Therefore, the best choice would be to start from the beginning and check each number one by one. This is already the most efficient approach given the constraints set by the game. Nothing is wrong with our approach. Instead, it's the conditions of the game that allows us to only eliminate one number at a time, making it so inefficient.

If you think about it, what we were doing is actually linear search. In linear search, we check each element one by one, from the beginning, because we don't know which index the element is in, and this is essentially the same as our approach to guessing. This was already covered in chapter 1 and was shown to have a time complexity of O(n).

1.1.2 Version 2

Sensing your growing frustration, your younger brother lets out a small sigh. "Okay, okay – this isn't very fun like this," he admits. "Let me help you out. From now on, I'll tell you whether your guess is too high or too low."

Just as you're about to guess "375", a stroke of genius hits you. With this new rule in place, a better strategy suddenly comes to mind. You start afresh. You first guess the middle number: 500. If he says "too low", you instantly know that the correct number is not within 1 to 500. If he says "too high", you eliminate the entire range from 500 to 1000. For your next guess, you repeat the same idea – pick the middle of the remaining range. Then again. And again, until you eventually guess the correct number. By the 3rd try, you've already guessed the correct number: 375.



You feel proud with yourself and start bragging to your brother about how quickly you cracked the game and how clever that approach was. But your your brother just smiles, saying nothing. A quiet grin spreads across his face.

This second approach that you used actually has a name. It's known as binary search, where we repeatedly eliminate half of the search space by checking the middle element. If the conditions allow for binary search, it's a much more efficient approach than linear search. In the next section, we'll further develop on the idea of binary search.

1.2 Binary Search

Binary search is a technique commonly used for checking whether a value exists in a sorted array. In the previous section, you saw the rough idea. Now it's time to put that idea to work. In this section, we'll walk through the steps of binary search, clear up a few common points of confusion, and then tie it all together with a simple example.

Steps for Binary Search Method

1. Check the middle number.
2. If it's too high, eliminate the larger half of numbers. If it's too low, eliminate the smaller half of numbers.
3. Repeat steps 1 and 2 on the remaining numbers until you find the target number or realize that it does not exist in the list of numbers.

How to Choose Middle Number?

To clarify how we choose the middle element in step 1, we split it into two cases: when the number of elements is odd and when it's even.

Odd: Choose the exact middle of the array. For example, if `array = [1, 2, 3, 4, 5]`, the middle element is `3`.

Even: There is no single middle element. In this case, choose the element just before the center point. For example, if `array = [1, 2, 3, 4, 5, 6]`, we pick `3`. You could also choose `4`, but for consistency, we will stick with the center-left element.

How many numbers are eliminated with each guess?

Suppose that before a guess, there are `N` remaining numbers. After making a guess, how many numbers are left? To answer this, we need to consider all cases – whether `N` is odd or even, and whether we discard the left half or the right half. Note that the middle element we just checked is also discarded.

Odd and discard left half: $N_{next} = \frac{N-1}{2}$

Odd and discard right half: $N_{next} = \frac{N-1}{2}$

Even and discard left half: $N_{next} = \frac{N}{2}$

Even and discard right half: $N_{next} = \frac{N}{2} - 1$

While the exact number of elements removed can vary slightly from case to case, the key idea is that each guess eliminates roughly half of the remaining range.

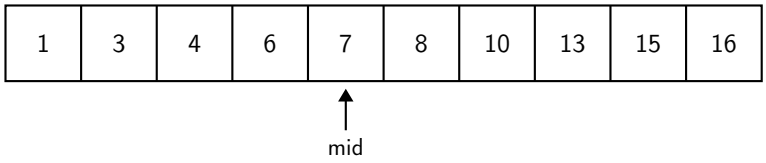
An Example

Suppose we've the array of integers sorted in ascending order

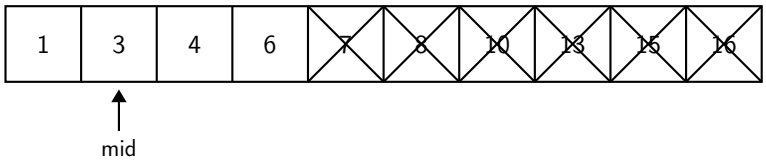
$$[1, 3, 4, 6, 7, 8, 10, 13, 15, 16]$$

We want to check whether the value 6 exists in the array. Let's walk through the binary search process step by step.

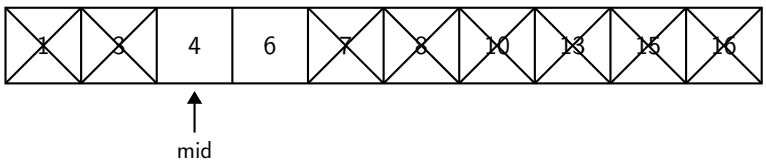
First, we check the middle element, 7. Since it's larger than the target value 6, we eliminate the range from 7 to 16.



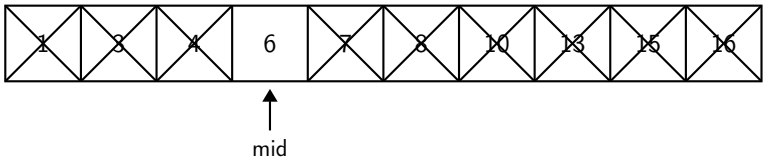
Next, we check the middle of the remaining elements, 3. Since it's smaller than 6, we eliminate the range from 1 to 3



We then check the middle again, which is 4. Since it's still smaller than 6, we eliminate 4.



Finally, we are left with a single element, 6. Since it matches the target, we conclude that 6 exists in the array.



1.3 The Half Optimality

It turns out that checking the middle element in binary search is optimal. In other words, it's more efficient than, say, checking the $\frac{3}{4}$th element at each guess. At first glance, this might not be immediately obvious why – after all, picking the $\frac{3}{4}$th element could eliminate up to $\frac{3}{4}$ of the remaining numbers in one guess.

To understand why the middle is more efficient, we first need to be clear about what "efficient" means. Here, we're talking about time complexity, where we consider the performance in the worst-case scenario. This was already covered in Chapter 1.

When we choose the middle element, each guess guarantees that we eliminate about half of the remaining numbers. No matter what happens, the search space is cut down by $\frac{1}{2}$ every time. Now consider choosing the $\frac{3}{4}$-th element instead. In some cases, you might eliminate $\frac{3}{4}$ of the numbers. But in the worst case, you only eliminate $\frac{1}{4}$. If lady luck is not on your side, you might repeatedly only eliminate $\frac{1}{4}$ of the numbers at each guess.

You can think of choosing the $\frac{1}{4}$ element as a high-risk, high-reward strategy, while choosing the middle is steady and reliable. Since we care about the worst-case scenario, the guaranteed $\frac{1}{2}$ reduction at every step makes the middle element the optimal choice.

This is why we pick the middle element in binary search – to exploit the optimality of cutting the search space in half!

1.4 Implementation in Python

Now we want to translate the binary search method into working code. Consider the problem below. We'll try to use binary search to return the index of a target element in a sorted array.

Problem

You are given an array of distinct integers `nums` which is sorted in ascending order, and an integer `target`. Using binary search, determine whether `target` is in `nums`. If it is, return its index. Otherwise, return `-1`.

Example 1:
Input: `nums = [-2, -1, 1, 4, 5, 8]`, `target = 1`

Output: `2`

Example 2:
Input: `nums = [-1, 3, 4, 6]`, `target = 5`
Output: `-1`

Solution

We want to translate the steps of binary search into code. To repeat steps 2–3, a `for` loop is not suitable, since the number of iterations depends on the values in `nums` and the `target`. Instead, we use a `while` loop, which allows us to continue repeating until a specific condition is met.

```
class Solution:
    def binarySearch(self, nums, target):
        ❶ l, r = 0, len(nums)-1
        ❷ while l <= r:
            ❸ mid = (l+r)//2
                if nums[mid] < target:
                    ❹ l = mid + 1
                elif nums[mid] > target:
                    ❺ r = mid - 1
            ❻ else:
                ❼ return mid
        ❽ return -1
```

1. First, initialize the left and right pointers, `l` and `r`, to the indices of the first and last elements of `nums`, respectively ❶. This defines the range in which the `target` could be in.
2. Compute the index of the middle element ❷ and check whether it equals the `target` ❻.
3. If the middle element is smaller than the `target`, eliminate the left half of the array ❹. If it's larger than the `target`, eliminate the right half ❺.
4. Repeat steps 2–3 using a `while` loop ❷ until either the middle element equals the `target` ❻ (in which case we `return` its index ❼ and the loop terminates), or the search space becomes empty – when `l > r`, causing the `while` loop to terminate based on its condition ❷ – meaning that `target` is not in the array. In the latter case, we `return -1` ❽.

1.5 Time and Space Complexity

In binary search, since we are eliminating about half of the input each time, we can intuitively guess that the time complexity is O(log n). But let's prove it mathematically.

Each time you guess a number, if you get it correct, you end there. If you get it wrong, in all cases, you eliminate at least half of the remaining numbers. Let's put this mathematically.

Let N_k be the number of remaining numbers after k guesses. Then

$$N_{k+1} \leq \frac{N_k}{2}$$

By repeated substitution and noting that $N_0 = n$, the equation becomes

$$N_k \leq \frac{N_0}{2^k} = \frac{n}{2^k}$$

Binary search ends when there is at most one element to be check ($N_k \leq 1$). So termination occurs when

$$\frac{n}{2^k} \leq 1$$

Now, we try to find the number of guesses for binary search to terminate, n. Multiplying both sides by 2^k,

$$n \leq 2^k$$

Taking logarithm base 2 to both sides,

$$\log_2 n \leq k$$

Therefore, the maximum number of guesses is

$$k = \lceil \log_2 n \rceil$$

Finally, note that each iteration of guessing and eliminating half of the remaining numbers can be accomplished in O(1) time. And since the maximum number of such iterations is O(log n) as just proven, the time complexity of binary search is

$$\text{time complexity} = O(\log\ n)$$

1.6 Pattern: Classic Binary Search

At its core, classic binary search is about eliminating half of the remaining elements in each move. Binary search isn't merely restricted to a sorted collection of elements. In some

problems, we can use logic to eliminate half of the remaining elements even if they're not sorted in any kind of order. This is perfectly exemplified by the second problem in this section.

Problem: Binary Search in 2D Matrix

You are given a list of integer list `matrix` which represents a `m x n` 2D matrix, and an integer `target`. The matrix is sorted in ascending order from left to right, top to bottom. Return `True` if `target` exists in the `matrix`. Otherwise, return `False`.

Example 1:

1	4	5	9
13	15	16	20
24	27	32	38

Input: `matrix = [[1, 4, 5, 9], [13, 15, 16, 20], [24, 27, 32, 38]]`, `target = 16`
Output: `True`

Example 2:

1	4	5	9
13	15	16	20
24	27	32	38

Input: `matrix = [[1, 4, 5, 9], [13, 15, 16, 20], [24, 27, 32, 38]]`, `target = 23`
Output: `False`

Solution

We first conduct binary search on the rows of the matrix to isolate the row where `target` could be at. Then, we conduct binary search on this row to check if `target` is in it.

```
class Solution:
    def searchMatrix(self, matrix, target):
        m, n = len(matrix), len(matrix[0])-1
        # Binary search on rows
    ❶ L, R = 0, len(matrix)-1
```

```
❷ while L <= R:
        Mid = (L+R)//2
        if target > matrix[Mid][n]:
            L = M + 1
        elif target < matrix[Mid][0]:
            R = Mid - 1
        else:
            # Binary search on columns
        ❸ l, r = 0, n
        ❹ while l <= r:
                mid = (l+r)//2
                if target > matrix[M][mid]:
                    l = mid + 1
                elif target < matrix[M][mid]:
                    r = mid - 1
                else:
                ❺ return True
        ❻ return False
❼ return False
```

We first initialize `L` and `R` ❶ to be the index of the first and last row respectively. They represent the rows that `target` could be in. Then using a `while` loop ❷, we conduct binary search to isolate which row `target` could be in. If we're unable to find such a row, `L > R` will be true, and the `while` ❷ loop terminates. This also means `target` is not in the matrix and we `return False` ❼. If we manage to isolate a row that `target` could be in, we initialize `l` and `r` ❸ to be the index of the first and last column respectively, and we conduct binary search on this row using a `while` loop ❹. Then, we `return` `True` ❺ or `False` ❻ depending on whether we find `target` in this row.

Time complexity: O(log(mn))
Space complexity: O(1)
where m is the number of rows and n is the number of columns.

Problem: Finding Peak Element

A peak element of an array is defined as an element greater than both its neighbours. For elements at the ends of an array, it is considered a peak element if it is greater than its one (and only) neighbour and you can think of this as `nums[-1] = nums[n] = -∞`. You are given an integer array `nums`, which have at least one peak element. Return the index of any one of the peak element.

Write an algorithm with time complexity of O(log n). Also assume that each element cannot have the same value as its neighbour(s).

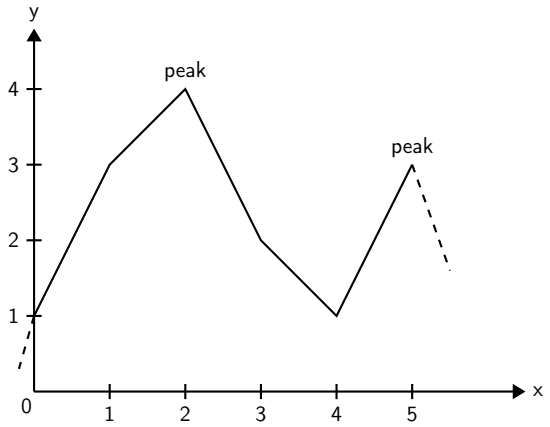
Example 1:

Input: `nums = [1, 3, 4, 2, 1, 3]`
Output: `2` or `5`

Example 2:
Input: `nums = [1, 4, 1]`
Output: `1`

Solution

At first glance, it's not obvious how this problem relates to binary search. However, if we experiment with a few examples, a useful pattern begins to emerge. First, plot the array as a graph, where the index and value of each element is the x-value and y-value respectively. For example, when `nums = [1, 3, 4, 2, 1, 3]`, its corresponding graph is



Since `nums[-1] = nums[n] = -∞`, we depict this by drawing dashed lines at both ends of the graph.

Next, define the gradient at index `i` as `nums[i + 1] - nums[i]`. Now consider the behavior at the boundaries. If neither endpoint is a peak element, then the `gradient at first index > 0` and the `gradient at last index < 0`. Under this constraint, the graph can only take on certain shapes. In particular, it has the following key property: if the gradient at some index is positive, then there must be a peak element somewhere to its right. Conversely, if the gradient at an index is negative, then there must be a peak element somewhere to its left.

This makes intuitive sense if you visualize the graph: a rising slope indicates that a peak has not yet been reached, while a falling slope indicates that the peak must already have been passed. By looking at the gradient at the middle index, we can determine whether a peak lies to the left or to the right. This allows us to discard half of the search space at each step.

```
class Solution:
    def peakElement(self, nums):
(1)     l, r = 0, len(nums)-1

        # Checking edge cases
(2)     if len(nums) == 1:
            return 0
(3)     if nums[l+1] < nums[l]:
            return l
(4)     if nums[r-1] < nums[r]:
            return r

        # Binary search
(5)     while l <= r:
            mid = (l+r)//2
(6)         if nums[mid] > nums[mid+1] and nums[mid] > nums[mid-1]:
                return mid
(7)         elif nums[mid+1]-nums[mid] > 0:
                l = mid + 1
(8)         elif nums[mid+1]-nums[mid] < 0:
                r = mid - 1
```

We first conduct "binary search" on the array. With `l` and `r` (1) initialized to represent the range of index that must contain at least one peak element, we use a `while` (1) loop to conduct binary search on this range. Depending on whether the gradient of the middle element is positive (7) or negative (8), we shrink the search space accordingly, until we find a peak element (6).

However, we are not done yet. We have to handle some edge cases that would be overlooked by our binary search method. In the case where there is only one element in `nums`, the `while` loop will not activate since `l = r`, and therefore we handle it separately in line (1). Also, notice that our logic for binary search explained earlier depends on the end elements not being peak elements. Therefore, we also have to separately handle the edge cases of the end elements being peak elements in line (2) and (3).

Time complexity: O(log n)
Space complexity: O(1)

1.7 Pattern: Binary Search for Boundaries

In some problems, we aren't just looking for a specific value, but rather a transition point where some conditions are satisfied. Below are some of such problems and I'll also explain the logic you can use to develop algorithms for such problems.

Problem: First Occurrence Index

You are given an array of integers `nums` sorted in ascending order and an integer `target`. Return the index of the first occurrence of `target` in `nums`.
Assume that `target` is in `nums`

Example 1:
Input: `nums = [2, 4, 7, 7, 7, 10]`, `target = 7`
Output: `2`

Example 2:
Input: `nums = [1, 2, 3]`, `target = 3`
Output: `2`

Solution

When implementing the binary search in previous problems using `while` loops, we write `if` statement for `target > arr[mid]` and `target < arr[mid]` to shrink the search space. This would not work for this problem because now an element can appear multiple times. For example, if `nums = [2, 2, 2, 2, 2]` the previous implementation will be useless at shrinking the search space. So to solve this problem, we modify the `if` statements to become `target >= arr[mid]` and `target < arr[mid]` instead.

```
class Solution:
    def firstOccurrence(self, nums, target):
        l, r = 0, len(nums)-1
    ❶   while l <= r:
            mid = (l+r)//2
        ❷   if nums[mid] < target:
            ❸   l = mid + 1
        ❹   else:  # nums[mid] >= target
            ❺   r = mid - 1
    ❻   return l
```

In the code above, the `while` loop ❶ is used to half the search space (`r - l`) at each iteration. After the last iteration of the loop, when `l > r`, `l` ends up at the index of the first occurrence of `target`, and so we `return` it ❻. This is not by coincidence. The `if` statements ❷ and ❹ were designed so that the following bounds always hold true.

With each iteration of the `while` loop ❶, either `l` shifts rightward ❸ or `r` shifts leftward ❻. We want a bound on how far rightward and leftward `l` and `r` can each reach respectively. We first focus on `l`. Observe that `l` shifts rightward ❸ only when `nums[mid] < target`

❷. Since all indices at or beyond the first occurrence of target satisfy `nums[i] ≥ target`, if we consider `mid` now to be the index of any element in the array, no comparison of a `nums[mid]` to `target` ❷ can ever cause `l` to move past the first occurrence of `target`. Hence, this gives the bound: `(0 ≤ l ≤ index of first occurrence)`. Using the same logic, we realize that the bound of `r` is `(index of first occurrence - 1 ≤ r ≤ lens(nums) - 1)`. These bounds must always hold true.

After the last iteration of the `while` loop ❷, `l > r`. Under the constraint of the bounds of `l` and `r` that we have just determined, `l > r` is only possible when `l = index of first occurrence` and `r = index of first occurrence-1`. Therefore, after the last iteration, `l` will be at the `first occurrence of target`.

For instance, in example 1, the bounds are `0 ≤ l ≤ 2` and `1 ≤ r ≤ 5` and after the last iteration of the while loop, `l = 2` and `r = 1`.

Time complexity: O(log n)
Space complexity: O(1)

Problem: Finding Minimum in Rotated Sorted Array

An array of integers is originally sorted in ascending order, where every integer is unique. This array is now rotated and given to you as `nums`. An array can be rotated between `1` to `n` times, where `n` is the number of elements in the array. For example, an array `[1, 2, 3, 4, 5]` is:
`[4, 5, 1, 2, 3]` if it is rotated 2 times.
`[1, 2, 3, 4, 5]` if it is rotated 5 times.

Return the minimum element of the array. Write an algorithm that runs in `O(log n)` time.

Example 1:
Input: `nums = [1, 2, 3]`
Output: `1`

Example 2:
Input: `nums = [3, 4, 0, 1, 2]`
Output: `0`

Solution

Since the problem requires our solution to run in `O(log n)` time, it likely means that we have to eliminate half of the search space as candidates for the minimum element each time. We need to determine what information we can extract out from the middle element would allow us to do so. After experimenting with some examples, we realize that if the middle element is larger than the last element of `nums`, the minimum element is somewhere to its right. Otherwise, if the middle element is smaller, the minimum element can be itself, or somewhere to its left. We can use this observation to conduct a boundary binary search – halving the search space at each iteration while, more importantly, maintaining the invariant bounds.

```
class Solution:
    def findMin(self, nums):
        l, r = 0, len(nums)-1
        last = nums[-1]

    ❶ while l <= r:
            mid = (l+r)//2
        ❷ if nums[mid] > last:
                l = mid + 1
        ❸ else:  # nums[mid] <= last
            ❹ r = mid - 1

    ❺ return nums[l]
```

With the insights gained, we can use a `while` loop ❶ to design the `if` statements ❷ ❸ to maintain the bounds to be invariant while halving the search space at each iteration. In the code above, upon some iterations of the loop, the search range might not contain the minimum element anymore. This occurs when `mid` is the index of the minimum element. This will activate line ❸ and `r` gets shifted to the element before the minimum element ❹. However, this does not matter as our goal is only to maintain the bounds.

Using the logic in the previous problem, the bound of `l` and `r` are `(0 ≤ l ≤ index of minimum element)` and `(index of minimum element - 1 ≤ r ≤ lens(nums) - 1)` respectively. These bounds must always hold true. After the last iteration of the loop, to satisfy `l > r`, `l = index of minimum element` and `r = index of minimum element - 1`. Therefore, we `return nums[l]` ❺.

Time complexity: O(log n)
Space complexity: O(1)

1.8 Pattern: Binary Search on Answers

Till now, we've been performing binary searches only on collections that are "physical" – given to you in the input. However, we can also perform binary search on a range of possible answers that we determine ourselves using logic. In some sense this range is "imaginary" because it is not given to you as an input.

Problem: Square Root

You are given a non-negative integer x. Return the value of $\lfloor \sqrt{x} \rfloor$, where $\lfloor \ \rfloor$ is the floor function. The value you return must also be non-negative. You cannot use any built in square root or exponentiation functions.

Example 1:
Input: $x = 16$
Output: 4

Example 2:
Input: $x = 10$
Output: 3
Explanation: $\sqrt{10} = 3.16227...$, which rounds down to 3.

Solution

It's not obvious how binary search is related to this. But to start out, we can try out small values of x. If $x = 0$ or 1, $\sqrt{x} = x$. If integer x is more than 2 $(x \geq 3)$, then realise that, mathematically, $0 \leq \lfloor \sqrt{x} \rfloor \leq \lfloor \frac{x}{2} \rfloor$. We can think of this range as the range in which the answer, $\lfloor \sqrt{x} \rfloor$, is in. The answer is the largest integer k, such that $k \leq \sqrt{x}$, or equivalently, the largest integer k, such that $k^2 \leq x$. Therefore, we can conduct boundary binary search to find the answer!

Let's call the middle number of the answer range mid. If $mid^2 \leq x$, the answer might be itself, or somewhere to its right. Otherwise, if $mid^2 > x$, the answer is somewhere to its left.

```
class Solution:
    def sqrt(self, x):
        # When x = 0 or 1
    ❶   if x < 2:
            return x

        # Boundary binary search on answer range
```

```
    ❷ l, r = 0, x//2
    ❸ while l <= r:
          mid = (l+r)//2
      ❹ if mid * mid <= x:
          ❺ l = mid + 1
        else:  # mid * mid > x
            r = mid - 1
    ❻ return r
```

We first handle the edge cases where `x = 0 or 1` ❶. Then, initializing `l` and `r` ❷ to represent the answer range, we use a `while` loop ❸ to half this range at each iteration. Do note that upon some iterations of the loop, the search range might not contain the answer anymore. This occurs when `mid` is the answer. This will activate line ❹ and `l` gets shifted to `answer + 1` ❺.

However, again, this does not matter. The bounds $(0 \leq l \leq answer + 1)$ and $(answer - 1 \leq r \leq \lfloor \frac{x}{2} \rfloor)$ will always hold true at every iteration of the loop. After the last iteration when $l > r$, the only way this is true is when $r = answer$ and $l = answer + 1$. Therefore, we return `r` ❻.

Time complexity: O(log x)
Space complexity: O(1)

Problem: Scooping Water:

You are given an integer array `bucket`, where `bucket[i]` is the volume of water in the `ith` bucket. Your job is to use a scooper to remove all water from every bucket. Each time, you may choose a bucket and use the scooper to scoop out water from the bucket.

You are also given an integer `s`, which represents the maximum total number of scoops that you are allowed with, so as to empty all the bucket. You may decide on `k`, which represents the capacity of the scooper. Return the minimum integer `k`, such that you can empty all buckets within `s` number of scoops.

Example 1:
Inputs: `bucket = [1, 2, 3, 4]`, `s = 9`
Output: `2`

Example 2:
Input: `bucket = [0, 0, 0]`, `s = 3`
Output: `0`

Solution

First, realize that for all $k \geq \max(buckets)$, the minimum number of scoops needed to empty all the buckets remains constant even if k increases. This is because we can only scoop from one bucket at a time. Therefore, the minimum integer k such that you can remove all water within s number of scoops must be in the range $1 \leq k \leq \max(buckets)$. We conduct a boundary binary search on this range of answers to find it.

```python
class Solution:
    def minScoops(self, buckets, s):
        l, r = 1, max(buckets)                          # ❶
        while l <= r:                                   # ❷
            m = (l + r) // 2
            count = sum(math.ceil(buckets / m) for bucket in buckets)
            if count > s:
                l = m + 1
            else:  # count <= s
                r = m - 1
        return l                                        # ❸
```

Initializing l and r ❶ to represent the answer range, we use the `while` loop ❷ to half the search range at each iteration. We keep in mind of the bounds that always hold true: $1 \leq l \leq k$ and $k-1 \leq r \leq \max(buckets)$. To satisfy the inequality $l > r$ after the last iteration of the loop, $l = k$ and $r = k-1$. Therefore, we return l ❸.

Time complexity: O(n log m)
Space complexity: O(1)
where n = len(buckets) and m = max(buckets).

2 Sample Chapter 2 – Recursion

Recursion is my favorite topic. When I first encountered it, I felt completely lost in its dizzying logic. But once I pushed through that initial confusion, I found it to be elegant and even strangely addictive. By the end of this chapter, I hope you will feel the same way I do. In this chapter, I will introduce the idea of recursion and explain how they work. Then, I will walk through the framework for constructing recursive functions, along with methods for analysing their time and space complexity. Finally, we will work through several problems step by step. By the end of this chapter, you will not only gain a logically rigorous yet intuitive understanding of recursion, but also be able to confidently construct your own recursive solutions.

2.1 What is Recursion?

In Python, a function can call another function and use its results. The code below shows an example:

```
def introduce():
    greet()                          ❶
    print("I love recursion!")       ❷

def greet():
    print("Hello!")

introduce()
# Prints "Hello!" "I love recursion!"   ❸
```

The `introduce()` function calls the `greet()` function ❶ before printing `"I love recursion!"` ❷. Together, calling `introduce()` will print line ❸. This should be quite familiar to us, and therefore, let me challenge you with a question: What does the following function do?

```
def function():
    print("This is a recursive function.")
    function()                       ❶
```

Instead of calling another function, `function()` calls itself ❶ within itself! This is known as recursion, where a function calls itself. It is hard to wrap our heads around what calling `function()` would do. But to answer the question: it turns out that calling `function` would

print the sentence `"This is a recursive function."` an infinite number of times, resulting in an error.

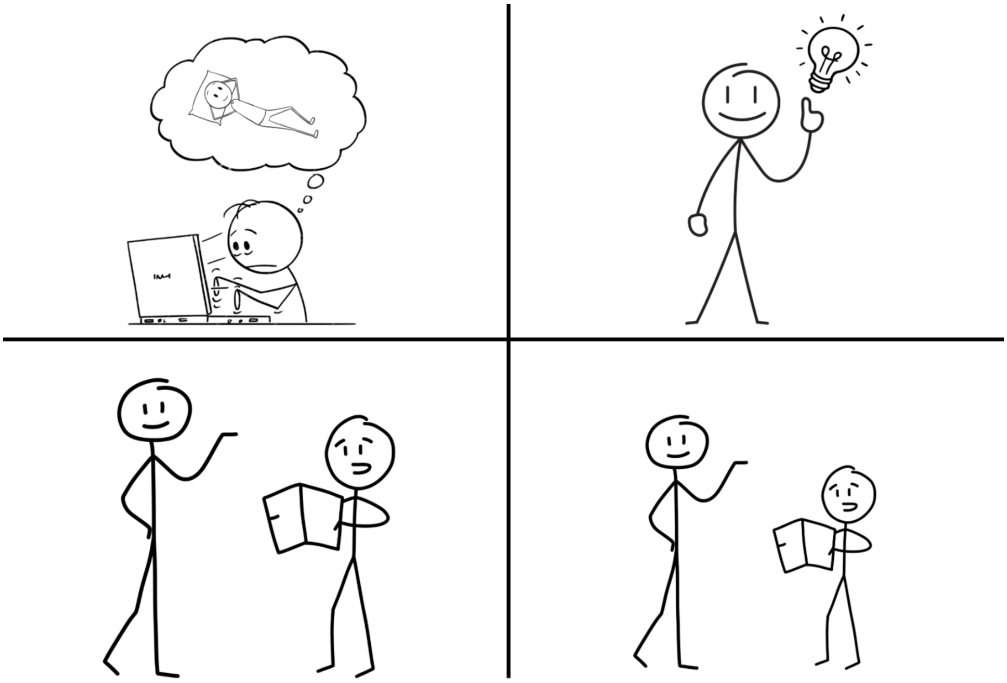
Unfortunately, our first ever recursive function turned out to be wrong. But don't get discouraged – in this chapter, we'll understand how and why recursion works, and learn how to construct recursive functions that actually work. As we journey through this chapter, you'll transition from confusion to intuition, from intuition to logic, and ultimately, to seeing recursion as a work of art. Let's start off with a story.

2.2 The Lazy Manager

Imagine you're a lazy manager who has just been assigned a massive, daunting project – one that would take weeks to complete on your own. Then an idea strikes: instead of doing everything yourself, you can delegate the work to your subordinate! Therefore, you decided to complete a small task, and delegate the remaining to your subordinate.

Unexpectedly, your subordinate turns out to be just as lazy. He follows the same strategy – completing only a small portion and delegating the rest to his subordinate. This chain of delegation continues, with each person doing a small part of the work and passing along the remainder.

Eventually, the task becomes so small that the last person in the chain can complete it immediately without further delegation. Once finished, the completed work is passed back up the chain, step by step, until it finally returns to you as a fully completed project.



At the start of this chapter, we introduced recursion as a function calling itself. That's a good starting point, but it's not quite the full picture. In practice, a recursive function does not just call itself – it calls a smaller version of itself. Therefore, this analogy is a more accurate way of thinking about recursion. Think of a recursive function as the lazy manager. It does a small share of work, then hands the rest of the project over to his subordinate by calling a smaller version of himself. This smaller version also does its share and passes the rest to its subordinate – an even smaller version of the function – and so on. This process continues until the remaining work becomes small enough that a sufficiently small version of the function handles its part, and the project is complete.

All this sounds really confusing. What does a smaller version of a function even means? What is the chain of delegation supposed to represent? I get it; but trust me, this idea will become much clearer when we attempt to build a recursive countdown function in the next section.

2.3 Building a Countdown Function

We want to build the countdown function defined below:

Countdown function:

Given an integer `i`, construct a function that prints each number from `i to 1` in that order. Note that `i >= 1`.

Using For Loops

A simple method to do so would be to use a `For` loop.

```
def countdown(i):
    for i in range(i, 0, -1):
        print(i)
# Prints i i-1 i-2 ... 1
```

Using Recursion?

That was an easy solution. Let's challenge ourselves by attempting to construct a recursive solution with the heuristics learned from the lazy manager analogy.

Let's recap. We want to construct a function `countdown(i)` that prints each number from `i` to `1` in that order. How do use the idea of recursion to construct that function?

Think of the lazy manager as `countdown(i)` and he has been tasked with the project of

printing every number from `i` to `1`. However, this would cost him too much effort. Perhaps he can complete some of the work and delegate the rest to his subordinate, a smaller version of `countdown(i)`? Maybe `countdown(i−1)`? Let's naively use the insights from this analogy and try to construct a recursive solution.

```
# An attempt to code a function which prints each number from i to 1
def countdown(i):

    # Completing a small task — Print number i
❶   print(i)

    # Delegating the rest — Hopefully print numbers from i−1 to 1?
❷   countdown(i−1)
```

The massive project to accomplish is to print each number from `i` to `1`. The function `countdown(i)` does a small share of the work by printing `i` ❶, before delegating the responsibility of printing `i−1` to `1` to `countdown(i−1)` ❷. Let's check if our function works by testing a small case of the function, `countdown(3)`.

```
❶ countdown(3)
  # Prints 3 2 1 −1 −2 −3 ...
```

Calling `countdown(3)` ❶ prints each number from `3` to `-∞`. Our function unfortunately did not work. To understand why, we can try tracing every step after calling `countdown(3)`:

Step 1: `countdown(3)` is called and it prints `3`.

Step 2: Before `countdown(3)` is complete, it calls `countdown(2)`.

Step 3: `countdown(2)` is called and it prints `2`.

Step 4: Before `countdown(2)` is complete, it calls `countdown(1)`.

Step 5: `countdown(1)` is called and it prints `1`.

Step 6: Before `countdown(1)` is complete, it calls `countdown(0)`.

Step 7: `countdown(0)` is called and it prints `0`.

Step 8: Before `countdown(0)` is complete, it calls `countdown(-1)`.

⋮

The lazy manager `countdown(3)` does its share of printing `3` and delegates the rest to `countdown(2)`. `countdown(2)` does its share of printing `2` and delegates the rest to

`countdown(1)`. This process carries on. Altogether, this sequence of steps prints each number from `3` to `-∞`, causing an error.

Each person in the delegation chain completes a small part of the task and passes the remaining work to his subordinate. The issue, however, is that no one in the chain recognizes when the project is actually complete – that is, when all numbers from `3` down to `1` have been printed. Consequently, when `countdown(0)` is called, it fails to realize that the project is already completed and continues to unnecessarily add on to the project!

We can solve this problem by stopping the chain at step 7. Right after `countdown(0)` is called, the process stops, and it does not print `0` or call `countdown(-1)`. This can be done by introducing an `if` statement:

```
def countdown(i):
 ❶ if i == 0:
        return
    print(i)
    countdown(i−1)
```

The addition of the `if` statement above ❶ will stop the chain at step 7. To understand why, let's take a look at what happens when we call `countdown(0)`:

```
def countdown(0):
    # Gets activated only for countdown(0)
 ❶ if i == 0:
     ❷ return
    print(−1)
    countdown(−1)

❸ countdown(0)
```

When `countdown(0)` ❸ is called, the `if` statement ❶ gets activated and it immediately `returns` ❷ without printing `0` or calling `countdown(-1)`. Therefore, calling `countdown(0)` breaks the chain and essentially does nothing.

This can be thought of as allowing `countdown(0)` to recognise that its project is already completed and it can simply hand the project back to its superior, `countdown(1)`, without doing anything. Let's try calling this new function.

```
countdown(3)
# Prints 3 2 1
```

It works!

We can trace out the steps again to understand why it works:

Step 1: `countdown(3)` is called and it prints `3`.

Step 2: Before `countdown(3)` is complete, it calls `countdown(2)`.

Step 3: `countdown(2)` is called and it prints `2`.

Step 4: Before `countdown(2)` is complete, it calls `countdown(1)`.

Step 5: `countdown(1)` is called and it prints `1`.

Step 6: Before `countdown(1)` is complete, it calls `countdown(0)`.

Step 7: `countdown(0)` is called and it `return` and does nothing. The chain ends here.

Altogether, this new sequence of steps only prints `3` to `1`, which is what we wanted.

In sum, recursive functions can be thought of as the lazy manager. When we call a recursive function, it kick-starts the chain of delegation, where each function calls a smaller version of the itself. However, this would result in an infinite chain. An addition of an appropriate `if` statement would stop this chain at where we wish to, and therefore, the `if` statement can be thought of as defining what would constitute as the completion of the project.

2.4 Base Case and The Cascading Dominoes

In the previous section, we understood how recursive functions worked by tracing every step after calling them. However, understanding recursion through the tracing method diminishes recursion into an overly mechanical process, when it in fact has such conceptual richness. In this chapter, I'll present to you an elegant way to understand recursion. I'll explain this using the example of our countdown function. To recap, this is the countdown function that we have constructed:

```
def countdown(i):
    if i == 0:
        return
    # Prints i
❶   print(i)
    # Hopefully prints i-1 to 1
❷   countdown(i-1)
```

Let's forget everything we have discussed for the time being pretend we are looking at

`countdown(i)` for the very first time. We do not know whether the `countdown(i)` works, and all we do know is that we want it to print `i` to `1`. Now let's look at `countdown(i)`. `countdown(i)` first prints `i` ❶ before calling `countdown(i−1)` ❷. If calling `countdown(i−1)` do indeed prints `i−1 to 1`, then calling `countdown(i)` will print `i to 1`. We should not attempt to trace out each step of the recursive call, but instead, focus only on this logical statement: if calling `countdown(i−1)` works, then `countdown(i)` works. Now, let's take a closer look on `countdown(i−1)`:

```
def countdown(i−1):
    if i == 0:
        return
    # Prints i−1
❶   print(i−1)
    # Hopefully prints i−2 to 1
❷   countdown(i−2)
```

It first prints `i−1` ❶ and then hopes calling `countdown(i−2)` ❷ prints `i−2 to 1`. Therefore we can make the logical statement: if calling `countdown(i−2)` works, then `countdown(i−1)` works. Let's now focus on `countdown(i−2)`:

```
def countdown(i−2):
    if i == 0:
        return
    # Prints i−2
❶   print(i−2)
    # Hopefully print i−3 to 1
❷   countdown(i−3)
```

It prints `i−2` ❶, and then hopes calling `countdown(i−2)` ❷ prints `i−2 to 1`. Once again, we can make the logical statement: if calling `countdown(i−3)` works, then `countdown(i−2)` works.

This pattern continues. Let's list all the statements we would make:

Statement 1: If `countdown(i−1)` works, then `countdown(i)` works.

Statement 2: If `countdown(i−2)` works, then `countdown(i−1)` works.

Statement 3: If `countdown(i−3)` works, then `countdown(i−2)` works.

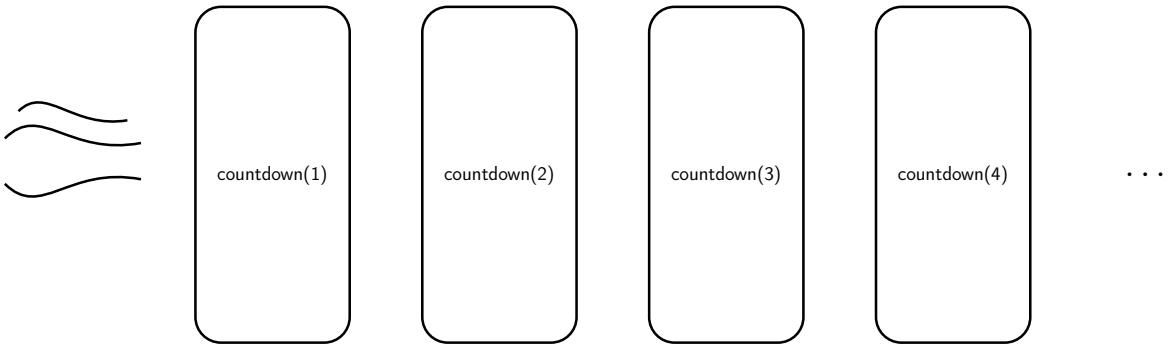
⋮

Statement i−2: If `countdown(2)` works, then `countdown(3)` works.

Statement i−1: If `countdown(1)` works, then `countdown(2)` works.

Here comes the essence of recursion: since each function hinges on the success of a smaller version of the function, if we make `countdown(1)` work, then `countdown(2)` works, then `countdown(3)` works, then `countdown(4)` works, and so on. The base case is the smallest version of the function, such that if it works, all larger versions of the function works.

Think of recursion as a chain of cascading dominoes and the base case as the first block. Once we make the base case work, it topples, triggering a chain of cascading dominoes, such that all larger versions of the function works. In this context, the base case is `countdown(1)`.



All that's left to do is to make the base case – `countdown(1)` – work. Does our solution already accomplish that?

```
def countdown(1):
❶  if  i  == 0:
        return
    # Prints 1
❷  print (1)
    # Does nothing
❸  countdown(0)
```

The function `countdown(1)` first prints `i` ❷ before calling `countdown(0)` ❸. Calling `countdown(0)` activates the `if` statement ❶, and therefore it does nothing. As a result, calling `countdown(1)` only prints `1`. Our base case works!

Do note, however, an alternative solution would be the following:

```
def countdown(i):
```

```
❶ if i == 1:
        print(1)
        return
    print(i)
    countdown(i-1)

❷ countdown(1)
  # Prints 1
```

Look at the alternative solution. When we call `countdown(1)` ❷, it prints `1`. Therefore, our base case also works for this function! The reason is that when we call `countdown(1)` ❷, it activates the `if` statement ❶ and prints `1`. The working of the base case triggers the cascading dominoes effect and results in `countdown(i)`, for all `i >= 2`, to work.

This idea about the base case triggering the cascading domino effect gives us fresh insights and meaning to the `if` statement in our countdown function. Previously, it merely served as a means to stop the chain when we deem the project to be completed. Now it's about making the base case work! This widens our perspective to a whole new conceptual dimension, and this is powerful. Understanding this idea will serve us well in the next section, where I'll outline the steps for constructing recursive solutions

2.5 Steps for Constructing Recursive Solutions

We've learned how recursion work. However, the motivation for each step of constructing a recursive function might still remain elusive and perhaps seems out of nowhere.

As a beginner, you might be tempted to construct recursive functions by tracing each step after the initial function call. This method might work, but when the problem gets much more complicated, unless you've a really large working memory, it's impractical to construct a working recursive function using it. Therefore, I'll introduce to you a much easier and elegant way of constructing recursive functions – taking the leap of faith:

1. **Figure out exactly what we want the function to do**. What are the range of inputs this function accepts. What are their corresponding outputs?

```
# For i >= 1, print every number from i to 1 in that order.
def countdown(i):
```

For our countdown function, the range of inputs it takes is any integer `i`, such that `i` ≥ 1. Its output would be to print every number from `i` to `1` in that order.

2. **Build the function with blind faith**. Assume that all smaller versions of your func-

tion is somehow already working as what was defined in step 1 and build the function. In this context, when constructing `countdown(i)`, we assume that `countdown(i−1)` magically works. This means assuming that if we call `countdown(i−1)`, it will print every number from `i−1` to `1` in that order, just as it was defined in step 1. Make use of this assumption to make `countdown(i)` print `i` to `1`.

```
def countdown(i):
    # This prints i.
    print(i)

    # Assuming countdown(i−1) works, this prints i−1 to i.
    countdown(i−1)

    # Togther, they print each number from i to 1 in that order.
```

3. **Find the base case**. Determine the smallest version of the function, such that if it works, all larger versions of the function will work. In this context, since we want to make `countdown(i)` work for all $i \geq 1$, `countdown(1)` is the base case.

4. **Make the base case work**. We typically do that by adding an `if` statement to the function. We need to add a `if` statement specifically to make the base case `countdown(1)` work.

```
def countdown(i):
    # Making the base case work
❶   if i == 1:
        print(1)
        return
    print(i)
    countdown(i−1)
```

The added `if` statement ❶ would only be affected for `countdown(1)`. Now when we call `countdown(1)`, it will print `1`. We can try checking by calling `countdown(1)`.

```
def countdown(1):
❶   if i == 0:
        return
    print(1)
    countdown(0)

❷ countdown(1)
  # Prints 1
```

Now when we call `countdown(0)` ❷, it activates the `if` statement ❶ and prints `1`. The base case works!

2.6 Recursion Call Stack

It's essential for us to understand how a computer treat recursive functions as it would allow us to be able to analyse its time and space complexity. We need to understand two main ideas – call stack and stack frame.

Stack Frame

Whenever a function is called, whether recursive or not, it creates a stack frame for that function. Think of stack frame as a temporary "storage box" that a computer creates each time a function is called, containing information data required for the function call and this includes:

1. Return Address: where to jump back to after this current function is completed. In the context where the function is called recursively, return address is the line of code to jump back to in the caller function.
2. Input Parameters: the values passed into the function
3. Local Variables: any variables declared inside the function

Call Stack

Whenever any function is called, recursive or not, its stack frame gets pushed onto the call stack. Think of a call stack as a structure to store stack frames, using the Last-In, First-Out property. Just like in chapter 3, this call stack uses two main operations – Push and Pop – to manage the stack frame. This are the two main operations of the call stack:

Push: Whenever you call a function, it creates a stack frame for it and pushes it onto the top of the call stack.
Pop: Whenever a function reaches a `return` statement or finishes it last line, its stack frame is then popped off from the top of the call stack.

The Stack of Books

I like to think of stack frames and the call stack using the analogy of a stack of books.

Each book represents the stack frame of a function. Whenever we call a function, we write its return address, input parameters and local variables into a book and place it on the top

of the stack of books. The stack of books represents the call stack. Whenever the function, represented by the stack frame at the top of the stack, reaches a `return` statement or finishes it last line, we remove the book at the top of the stack.

At any point in time, you can only be reading the top book, accessing its information. While you are reading the top book, every book beneath it is completely inaccessible. This means that a function can only see and utilise its own variables and parameters, and never another function's. This explains why you can repeatedly define the variable `x` in a recursive function without conflicting with the same variable name in another call.

Let us use the countdown function as an example. To recap, our countdown function is

```
def countdown(i):
    if i == 0:
        return
    print(i)
    countdown(i-1)
```

Recall what happens when we call `countdown(3)`.

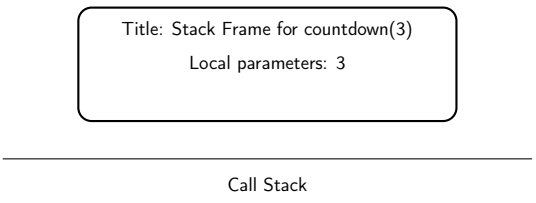
Step 1: `countdown(3)` is called. It prints `3` and calls `countdown(2)`.

Step 2: `countdown(2)` is called. It prints `2` and calls `countdown(1)`.

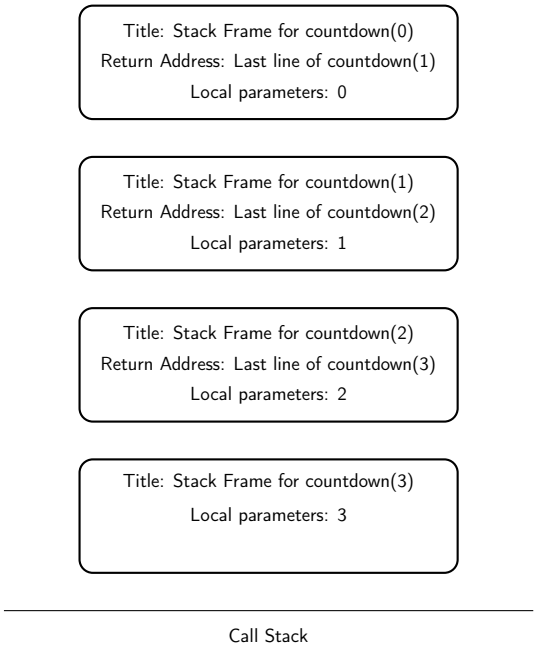
Step 3: `countdown(1)` is called. It prints `1` and calls `countdown(0)`.

Step 4: `countdown(0)` is called. It `return` and does nothing. The chain ends here.

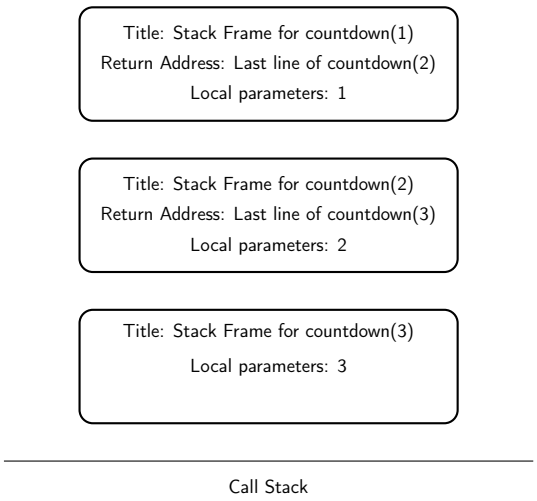
When we called `countdown(3)` in step 1, a stack frame for it gets created and is pushed onto the top of the stack:



Then, `countdown(3)` calls `countdown(2)`, creating `countdown(2)`'s stackframe and pushing it onto the stack. This patterns continues, resulting in the call stack as illustrated below:



Now, after `countdown(0)` finishes its last line, its stack frame gets pops off the call stack as illustrated below.



One by one, each countdown function in the call stack finishes its last line and gets pops off the call stack. At the end, all stack frames have been popped and the call stack is empty.

Call Stack

2.7 Time Complexity

You can think of the time complexity of a recursive function as:

time complexity = O(number of times the function is called x cost per call)

This is the value we wish to determine and I will elaborate on two main methods to do so – recursion tree and iterative.

2.7.1 Recursion Tree Method

The recursion tree method is a visual way to figure out the time complexity of a recursive function. It involves drawing out the recursive calls as a tree, where each node represents a function call. Then, the time complexity is determined by summing up the cost contribution of every node in the tree. I will outline the steps for using the recursion tree method. We shall explore the method using two examples.

Example 1

Let us start off by using the recursion tree method to determine the time complexity of our good, old countdown function.

1. **Write the cost recurrence relation**. `T(n)` represents the cost of the recursive function with input size n. For our countdown function, `T(i)` refers to the cost of `countdown(i)`. To write the cost recurrence relation, we sum the contributions from each line of code in the recursive function:

```
def countdown(i):
    if i <= 0:
        return
    # Contribute constant cost c.
    print(i)
    # Contribute T(i-1) cost.
    countdown(i-1)
```

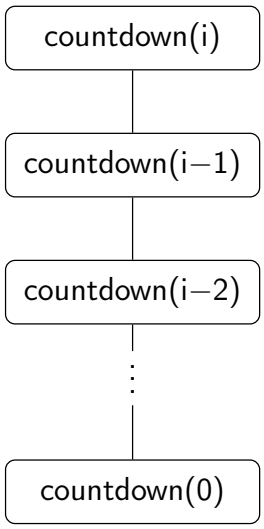
The recurrence relation is given by the sum of contributions:

$$T(i) = T(i-1) + c$$

This tells us that `T(i)`, the cost of calling `countdown(i)`, is equal to the sum of T(i−1), the cost of calling `countdown(i-1)`, and `c`, the cost of `print(i)`.

NOTE: `print(i)` contributing a constant cost `c` simply means that that line of code does not depend on the input size `i`, and has a time complexity of `O(1)`.

2. **Draw the recursion tree**. A recursion tree is a tree, where each node represents a function call. Edges connect a parent node at a higher level to its child nodes at the next level, representing the parent function calling its child functions. Using the recurrence relation as a guide, draw the recursion tree until you see a clear pattern:



3. **Determine the cost of each node in the tree and sum them up**. The cost contribution of a node is the recurrence relation but excluding the recursive call terms. For example, the node `countdown(i)` contribute `c` instead of `T(i−1) + c`, the node `countdown(i−1)` also contribute `c` instead of `T(i−2) + c`, and so on. This is so that we do not double count the costs because the cost of recursive call terms of a node is already accounted for by summing the cost contributions of all of its descendant nodes. This is better understood by tracing out each steps after the initial function call:

Step 1: `countdown(i)` is called. It prints `i`. and calls `countdown(i−1)`.

Step 2: `countdown(i−1)` is called. It prints `i−1` and calls `countdown(i−2)`.

Step 3: `countdown(i−2)` is called. It prints `i−2` and calls `countdown(i−3)`.

⋮

Step i: `countdown(1)` is called. It prints `1` and calls `countdown(0)`.

Step i+1: `countdown(0)` is called. It `return` and does nothing. The chain ends here.

`T(i)` is the cost of steps `1 to i+1`, `T(i−1)` is the cost of steps `2 to i+1`, and `c` is

the cost of step `1`. `T(i−1)` is the cost of steps `2 to i+1`, `T(i−2)` is the cost of steps `3 to i+1`, and `c` is the cost of step `2`. Therefore, a `countdown(i)` node for a `i >= 1` contributing a constant cost `c` can be thought of as the cost for completing step `i`. Additionally, the `countdown(0)` node also contribute a constant cost c, which is the cost of step `i+1`.

We want to sum the cost contribution of every node in the recursion tree, which is the cost of completing steps `1 to i+1`. Since there is a total of `i+1` nodes in our recursion tree, and each node contributes a constant cost `c`, `T(i)` is given by:

$$T(i) = \text{total number of nodes} \times c = (i+1) \times c$$

Therefore, we can conclude that the time complexity of the countdown function is

$$\text{time complexity} = O(i)$$

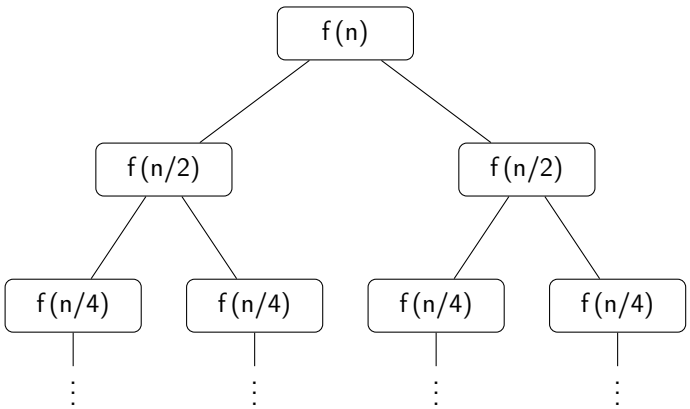
Example 2

Assume an algorithm has a cost recurrence relation of $T(n) = 2T\left(\frac{n}{2}\right) + n$, and that its base case occurs when $n = 1$. Use the recursion tree method to determine its time complexity.

1. **Write the cost recurrence relation.** In this problem, we assume we begin by already knowing the cost recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

2. **Draw the recursion tree.** With reference to the cost recurrence relation, each function $f(n)$ recursively calls two smaller function $f\left(\frac{n}{2}\right)$. Therefore, we draw the recursion tree until we notice a pattern:



However, we would like the extract more information about our recursion tree through the cost recurrence relation. Mainly, we would like to find the height and the number of leaves

our recursion tree has. The height is simply the total number of levels in our recursion tree.

Height of recursion tree

Let k be the number of recursive calls f(n) have to call to reach the base case f(1).

$$f(1) = f\left(\frac{n}{2^k}\right)$$

$$1 = \frac{n}{2^k}$$

$$\Rightarrow k = \log_2 n$$

However, since k does not include the very first level f(n). Therefore, the height of the recursion tree is k+1:

$$h = \log_2 n + 1$$

Number of leaves

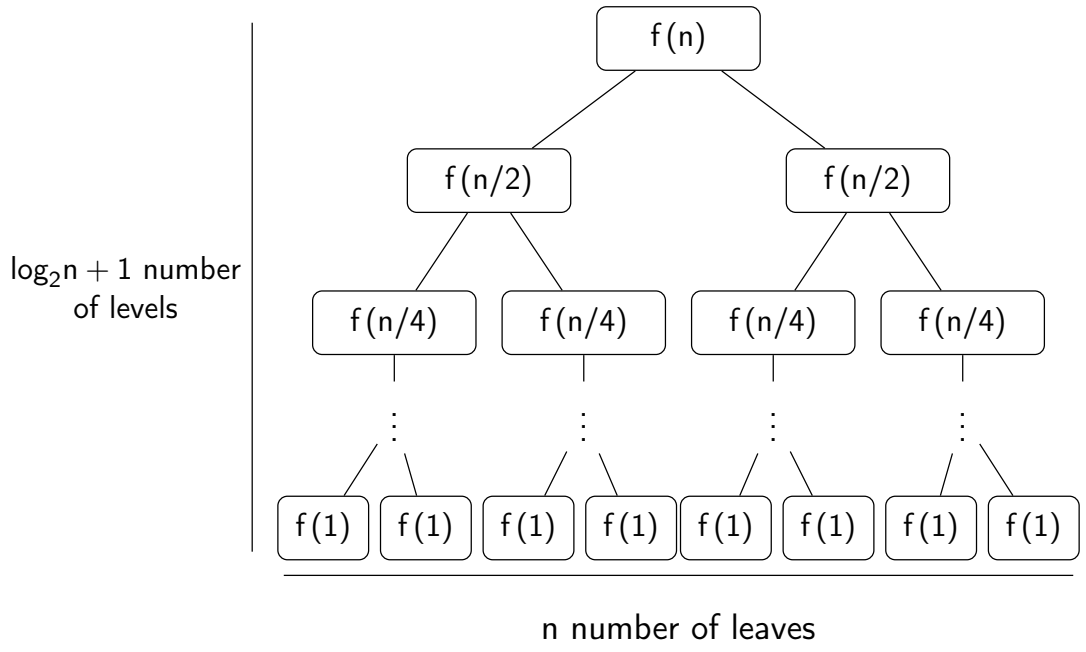
Notice that level 1 have 2^0 node, level 2 have 2^1 nodes, level 3 have 2^2 nodes – the i'th level have 2^{i-1} number of nodes. Therefore, the number of leaves are is:

$$\text{number of leaves} = 2^{h-1} = 2^{\log_2 n}$$

Using logarithm rules and simplifying,

$$\text{number of leaves} = n^{\log_2 2} = n$$

We can now add in the height and number of leaves into our drawing of the recursion tree:



3. **Determine the cost of each node in the tree and sum them up.** As explained before, to prevent double counting, each node contribute a cost of n, instead of $2T\left(\frac{n}{2}\right) + n$. Of course, this does not include the leaves. Since the leaves are the base cases, they each contribute a constant cost c. We sum up the cost contributions from every node to determine the time complexity of a recursive function.

level	Cost of each node	Cost of each level
1	n	n
2	$\frac{n}{2^1}$	$2^1\left(\frac{n}{2^1}\right) = n$
3	$\frac{n}{2^2}$	$2^2\left(\frac{n}{2^2}\right) = n$
$\vdots$	$\vdots$	$\vdots$
k	$\frac{n}{2^k}$	$2^k\left(\frac{n}{2^k}\right) = n$
$\vdots$	$\vdots$	$\vdots$
$\log_2 n + 1$	c	nc

Note: Since every node of a level contribute the same cost, the cost contributed by each level is simply the number of nodes at that level multiplied by the cost of each node.

The time complexity is given by summing up the contributions from every level of the recursion tree.

$$T(n) = n\log_2 n + nc$$

Therefore, the time complexity of the recursive function is

$$\text{time complexity} = O(n\log n)$$

2.7.2 Iterative Method

Instead of drawing the recursion tree, another way to determine the time complexity is to solve the recurrence relation algebraically. This is known as the iterative method and it involves substituting the recursive term with its own definition until a clear pattern is observed. I will outline the steps for doing so below:

1. **Write out the recurrence relation**. As explained in the recursion tree method above, the recurrence relation for `countdown(i)` is

$$T(i) = T(i-1) + c \tag{1}$$

2. **Use the recurrence relation for repeated substitution** We write out the recurrence relation for `countdown(i-1)` and substitute it into the equation above.

$$T(i-1) = T(i-2) + c \tag{2}$$

Substituting (2) into (1),

$$T(i) = T(i-2) + 2c$$

Repeatedly substituting,

$$T(i) = T(i-3) + 3c$$
$$T(i) = T(i-4) + 4c$$
$$T(i) = T(i-5) + 5c$$
$$\vdots$$

3. **Identify the pattern and generalise**. It is quite easy to notice the pattern. Noting that $T(0) = c$,

$$T(i) = T(0) + ic = (i+1)\,c$$

Therefore, we conclude that the time complexity of `countdown(i)` is:

$$\text{time complexity} = O(i)$$

2.8 Space Complexity

To determine the space complexities of recursive solutions, we need to consider the worst case scenario. This occurs when the call stack is at its maximum depth. More precisely,

$$\text{space complexity} = O(\text{max call stack depth x memory per stack frame})$$

For the recursive countdown function developed at the start of this chapter:

```
def countdown(i):
    if i <= 0:
        return
    print(i)
    countdown(i-1)
```

The maximum call stack depth is `i + 1`, and not `i` since `countdown(0)` was also called. The max stack depth is O(i). The memory stored per stack frame is constant. Therefore, the max stack depth is O(i) and memory stored per stack frame is O(1). Consequently,

$$\text{space complexity} = O(i \times 1) = O(i)$$

2.9 Worked Examples

We have come a long way. We began by understanding how recursion works through building the countdown function, then moved on to the general steps for constructing recursive solutions. From there, we explored the recursion call stack, along with methods for analysing the time and space complexity of recursive functions.

As promised at the start, by the end of this chapter, you would be able to confidently construct your own recursive solutions. To wrap up this chapter, we will work through three problems step by step, bringing everything we have learned together.

Problem: Fibonacci Number

The nth term of the Fibonacci sequences is denoted by `F(n)`, such that

1. `F(0) = 0`, `F(1) = 1`
2. `F(n) = F(n−1) + F(n−2)`

Given `n`, find `F(n)` using a recursive solution.

Example 1:
Input: `n = 2`
Output: `1`

Example 2:
Input: `n = 3`
Output: `2`

Solution

1. **Figure out exactly what we want the function to do**.

```
class Solution:
    # If n >= 0, this function takes in an integer n and returns F(n)
    def F(n):
```

We decide that the range of input `f(n)` accepts will be any integer `n`, such that `n ≥ 0`. The output is that it `returns` the `nth` fibonacci number.

2. **Build the recursive function with blind faith**. We assume that, somehow, all smaller versions of the function than `F(n)` – `F(n−1)`, `F(n−2)`, `F(n−3)`, and so on – are already working correctly. Use this assumption to build `F(n)`.

```
class Solution:
    def F(n):
     ❶ return self.F(n−1) + self.F(n−2)
```

We assume calling `F(n−1)` and `F(n−2)` will return the `(n−1)th` and `(n−2)th` fibonacci number respectively. To make `F(n)` return the `nth` fibonacci number, we simply `return F(n−1) + F(n−2)` ❶.

3. **Find the base case**. For `F(n)` to work, it depends on the success of `F(n−1)` and `F(n−2)`. Consequently, if `F(0)` and `F(1)` work, this ensures that all `F(n)` for `n > 1` will also work. Therefore, the function requires two base cases instead of just one. This is because each `F(n)` relies on two smaller instances of itself, so a single base case would be insufficient to guarantee that the larger values of `F(n)` can work.

4. **Make the base cases work**. We seek to make `F(0)` and `F(1)` work.

```
class Solution:
    def F(self, n):
        # Making F(0) work
        if n == 0:
            return 0
        # Making F(1) work
        if n == 1:
            return 1

        return self.F(n−1) + self.F(n−2)
```

We can test out our base cases to check if they indeed work.

```
print F(0)
# Prints 0

print F(1)
# Prints 1
```

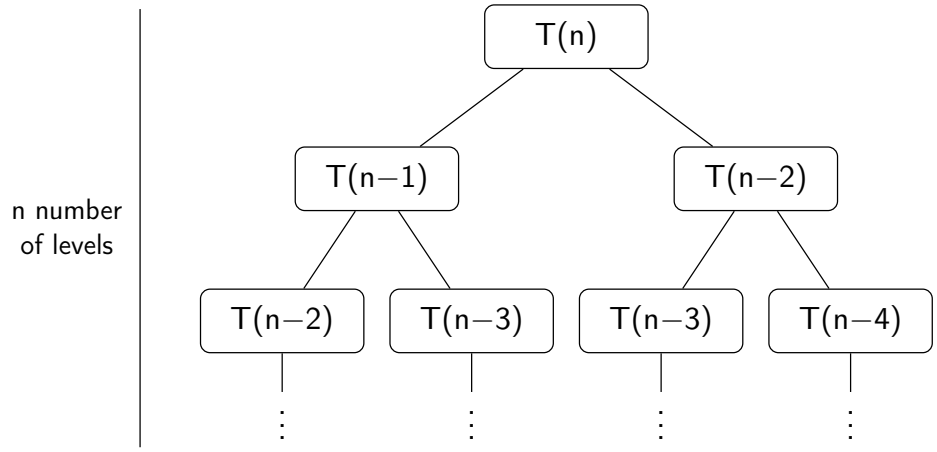
They work! Now we have finished constructing a working recursive solution to calculate Fibonacci numbers. How do we determine the time complexity for our solution?

Time Complexity

We can first write out its cost recurrence relation. Note that summing `F(n−1)` and `F(n−2)` contribute a constant cost c.

$$T(n) = T(n-1) + T(n-2) + c$$

To visualise this, we draw its recursion tree. The longest path from the root to a leaf is n (following the n−1 branch all the way down). So, the height is h = n.



Notice that in our recursion tree, some branches have fewer levels than others. At the extremes, the rightmost branch have the least number of levels while the leftmost branch have the most number of levels. However, for the sake of simplicity, we can assume that it is a complete binary tree, where the i'th level has 2^i number of nodes. This would represent a loose upper bound on the time complexity because we are overcounting the number of nodes. With this assumption, the total number of nodes can be calculated as a geometric series:

$$\text{total number of nodes} = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Since every node in the recursion tree contribute a constant cost, the time complexity is simply:

$$\text{time complexity} = O(2^n)$$

Note that if we would like calculate the time complexity precisely by calculating the exact number of node, we would get that the time complexity = $O(\varphi^n)$, where φ is the golden ratio. However, it does not really matter whether we use $O(\varphi^n)$ or $O(2^n)$, since they both belong to the same exponential complexity class.

Space Complexity

The maximum recursion depth occurs in the leftmost branch of the recursion tree, which has n levels. Since our solution uses constant memory per stack frame, the space complexity is

$$\text{space complexity} = O(n)$$

Problem: Josephus and his game

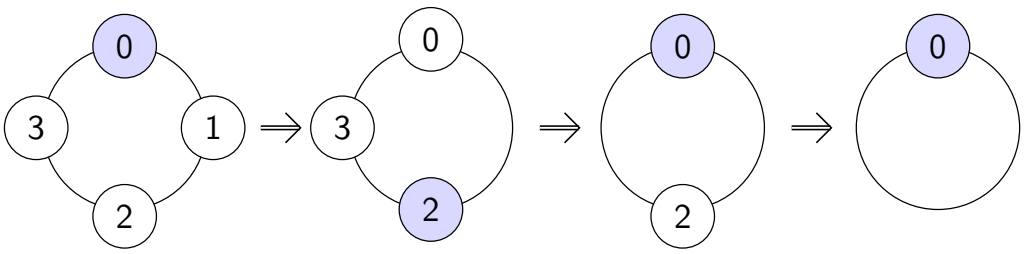
In 67 CE, during the First Jewish–Roman War, Josephus was a Jewish general defending the city of Yotapata against the Roman army. After a brutal 47-day siege, the city fell. Josephus and a group of soldiers escaped the slaughter and hid in a cave, but the Romans soon discovered them and demanded their surrender. Josephus wanted to give up. His men did not. They preferred death to capture and threatened to kill him if he tried to surrender. So Josephus proposed a grim alternative: a game that would decide their fate. The soldiers agreed.

You are given the total number of men (including Josephus) `n` and an integer `k`. These are the rules to the game:

1. Josephus and the men sit in a circle and number themselves from `0 to n−1`.
2. Begin with the `0th` man. Starting from himself, count the next `k` men in the clockwise direction and eliminate him.
3. If there is still more than one men in the circle, return to step 2 starting from the man immediately clockwise of the man who was just eliminated and repeat.
4. The last man to remain alive is the winner.

Return the position number Josephus should stand at so that he will win the game.

Example 1:



Inputs: `n = 4`, `k = 2`
Outputs: `0`

Example 2:
Inputs: `n = 5`, `k = 2`

Outputs: `2`

Solution

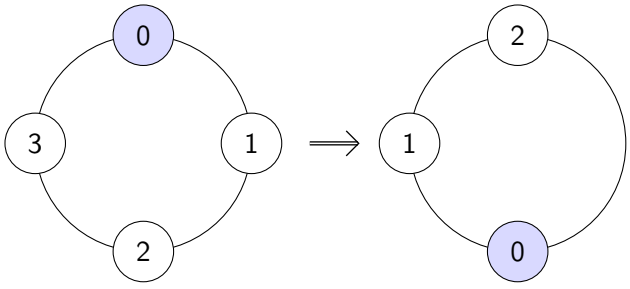
1. **Figure out exactly what we want the function to do.**

```
# For an integer n >= 1, this function will return the index of
  the survivor of the game.
class Solution:
    def survivor(self, n, k):
```

2. **Take the leap of faith.** We assume that `survivor(self, n−1, k)` is already working. Then, we make use of this assumption to construct our function, `survivor(self, n, k)`.

```
class Solution:
    def survivor(self, n, k):
        return (self.survivor(n - 1, k) + k) % n
```

To see how this works, take example 1. Starting with `n = 4`, after the `0th` man eliminated the `k−1th` man, there are now `3` men left. The problem, though of a smaller size, still remains the same, and thus we can call `survivor(self, 3, k)` to find the index of the survivor in the remaining `3` men. However, do note that we return `(survivor(self, 3, k) + k) % n` instead of simply `survivor(self, 3, k)` because when we call `survivor(self, 3, k)`, it changes the index of each of the men as shown below. `(survivor(self, 3, k) + k) % n` therefore re-indexes each man to match his original index.



3. **Find the base case.** Since the success of each `survivor(self, n, k)` hinges the success of `survivor(self, n−1, k)`, the base case is `survivor(self, 1, k)`, where `n = 1`. As long as this base case works, all cases larger than itself also works.

4. **Make the base case work.** When $n = 1$, there is just one men left. This men is therefore the survivor and we return his index, 0.

```python
class Solution:
    def survivor(self, n, k):
        # Make the base case work
        if n == 1:
            return 0
        return (self.survivor(n - 1, k) + k) % n
```

Time Complexity

The cost recurrence relation is

$$T(n) = T(n-1) + c$$

This is a familiar result, which we have already shown to have time complexity of $O(n)$.

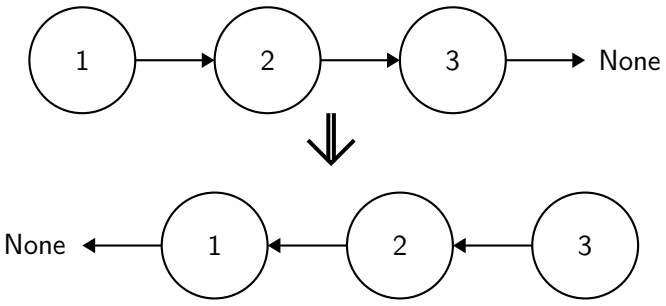
Space Complexity

Since the maximum recursion depth is n and each stack frame uses constant memory, the space complexity is O(n).

Problem: Recursive Reversal of Linked List

You are given the `head` of a linked list. Reverse the linked list and return the head of it. Return `None` if the linked list have no node. Use a recursive algorithm.

Example 1:



Input: `head = [1, 2, 3]`
Output: `head = [3, 2, 1]`

Example 2:
Input: `head = [ ]`
Output: `[ ]`

Solution

1. **Figure out exactly what we want the function to do**

```
class Solution:
    # If the head is a node, this function reverses the linked list
      and return the new head.
    # If the head == None, this function return None
    def reverseList (self, head):
```

2. **Take the leap of faith**. If the inputted linked list have n number of nodes, we assume that the function works for all smaller cases – the function is able to successfully reverse and return the new head of a any linked list with x number of nodes, for all x <= n. Then, make use of this assumption to write a recursive function:

```
class Solution:
    def reverseList (self, head):
        # Fist, reverse the section of the linked list, starting from head.next to tail
        # Also, store the original tail node as newHead.
        newHead = self.reverseList(head.next)

        # Change the direction of arrow between the original head and head.next
        head.next.next = head
        head.next = None

        # The linked list is now fully reversed.
        # Return the new head of the reversed linked list.
        return newHead
```

3. **Find the base case**. The base case is a linked list of one node, that is, if `reverseList(self, head)` works for a linked list with one node, it will also work for a linked list with n number of nodes, where n > 1. However, notice that there is a case smaller than the base case – a linked list with no nodes, `reverseList(self, None)`. Why is this smaller case not the base case instead? If this smaller case is the base case instead, the function `reverseList(self, head)` for a head of a linked list with one node will be correct if `reverseList(self, None)` is correct. Note that `reverseList(self, None)` is correct if it returns `None`. However, even if

`reverseList(self, None)` is correct, the code in step 2 still result in an error for the line `head.next.next = head`, since `None` type does not have a next value. In sum, even if `reverseList(self, None)` works, `reverseList(self, head)` for a linked list of one node will result in an error and will not work. Therefore, the base case is a linked list with one node, instead of a linked list with no node. We must make both the base case and its smaller case work.

4. **Make the base case (and the case smaller than it) work.**

```
class Solution:
    # This function reverses the linked list from the inputed head to end of linked list.
    # Returns the new beginning of the reversed list
    def reverseList(self, head):
        # Making the case smaller than the base case work
        if not head:
            return None
        # Making the base case work
        if not head.next:
            return head

        newHead = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return newHead
```

Time Complexity

We first write out the cost recurrence relation:

$$T(n) = T(n-1) + c$$

Notice that this is the the exact same recurrence relation as the countdown function. Therefore, just as in section 2.5.2, the time complexity of this solution is O(n).

Space Complexity

Since the maximum recursion depth is n and each stack frame uses constant memory, the space complexity is

$$\text{space complexity} = O(n)$$