Contents

Introduction i

Python Syntax Cheat Sheet ii

1 Time and Space Complexities 3
- 1.1 Efficiency Of an Algorithm . . . 3
- 1.2 Time Complexity and Big O-Notation . . . 5
- 1.3 Worst-Case Scenario and Array Operations . . . 7
- 1.4 Space Complexity . . . 11

2 Hash Tables 13
- 2.1 Take 1 – The Inexperienced Motel Owner . . . 13
- 2.2 What is a Hash Table? . . . 13
- 2.3 Hash Function . . . 14
- 2.4 Collisions and Separate Chaining . . . 15
- 2.5 Time Complexity . . . 15
- 2.6 Hash Sets, Hash Maps and Their Implementation in Python . . . 17
- 2.7 What is Hashable? . . . 19
- 2.8 Pattern: Fast Lookup . . . 21
- 2.9 Pattern: Frequency Tracking . . . 24

3 Stack 27
- 3.1 Take 2 – The Pancake Booth . . . 27
- 3.2 What is a Stack? . . . 27
- 3.3 Implementation in Python . . . 28
- 3.4 Pattern: Monotonic Stack . . . 28
- 3.5 Pattern: Parentheses Matching . . . 31
- 3.6 Pattern: Expression Evaluation . . . 35

4 Binary Search 39
- 4.1 Take 3 – The Weird Guest . . . 39
- 4.2 Linear Search . . . 40
- 4.3 Binary Search . . . 40
- 4.4 Implementation in Python . . . 41
- 4.5 The Half Optimality . . . 42
- 4.6 Time and Space Complexity . . . 42
- 4.7 Pattern: Classic Binary Search . . . 43
- 4.8 Pattern: Binary Search for Boundaries . . . 46

4.9 Pattern: Binary Search on Answers 48

5 More Coming Soon! 51

5.1 If you found this book helpful, please share this book! 51

1 Time and Space Complexities

Before we even start learning how to build algorithms, we first need a metric to evaluate how efficient an algorithm is. Otherwise, we'll be like headless chickens – if we don't even know what qualifies as an efficient algorithm, we won't know how to build one either! In this chapter, we'll put everything we think we know aside for a moment and try to develop methods to evaluate the time and memory efficiency of algorithms. Then, we will use these methods to evaluate some common algorithms.

1.1 Efficiency Of an Algorithm

Method 1: Number of Operations
Let's say we have an algorithm. How do we go about evaluating how time-efficient that algorithm is? One way is to simply count the number of operations the algorithm performs. Let's look at an example.

Assume we want to write an algorithm to print out every number in an array.

```
array = [1, 2, 3, 4, 5]
for num in array:
    print(num)
```

Number of operations:
1. `for num in array` goes through each element: 5 operations
2. `print(num)` prints each element once: 5 operations

Total: 10 operations

Alright, why not try using this algorithm on another array?

```
array = [1, 2, ..., 100]
for num in array:
    print(num)
```

Number of operations:
1. `for num in array` goes through each element: 100 operations
2. `print(num)` prints each element once: 100 operations

Total: 200 operations

Method 2: Number of Operations in Terms of n
Something doesn't quite add up. If we evaluate the efficiency of an algorithm based on the number of operations it performs, then the first algorithm should be more efficient than the second one, since it performed fewer operations. But aren't they the same algorithm, just with different input sizes? Therefore, we need to find a way to evaluate an algorithm's efficiency regardless of the size of the input.

Let's call the size of the input n; then we can evaluate the efficiency by the number of operations performed in terms of n.

```
array = [1, 2, ..., n]
for number in array:
    print(number)
```

Number of operations:
1. `for num in array` goes through each element: n operations
2. `print(num)` prints each each element once: n operations
Total: 2n operations

Now, the efficiencies of an algorithm (2n) for different input sizes are the same, which is what we wanted. Let's use this method to evaluate the efficiency of a slightly more complicated algorithm. This algorithm prints out all ordered pairs of numbers in an array.

```
array = [1, 2, ..., n]
for i in array:
    for j in array:
        print(i, j)
```

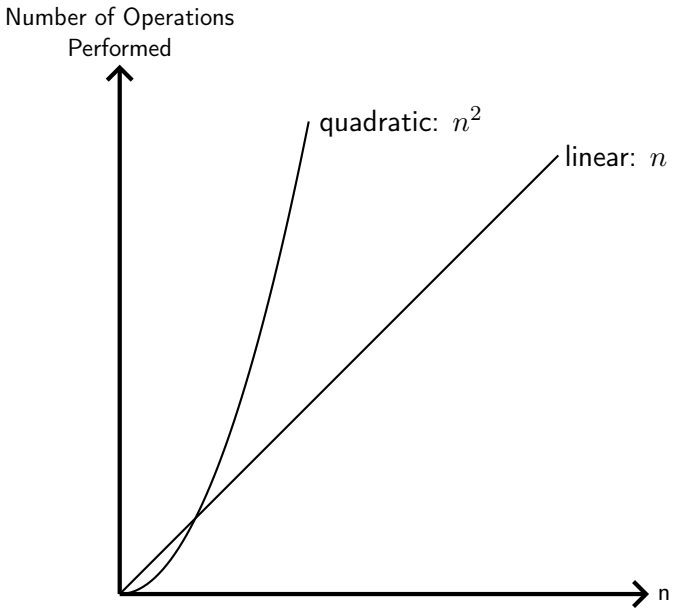
Number of operations:
1. The outer loop (`for i in array`) goes through every element: n operations.
2. The inner loop (`for j in array`) goes through every element of the array for each i: n^2 operations.
3. `print(i, j)` prints each ordered pair of numbers: n^2 operations.
Total operations: $2n^2 + n$

Therefore, in the way that we've previously decided to evaluate efficiency, this algorithm has an efficiency of $2n^2 + n$.

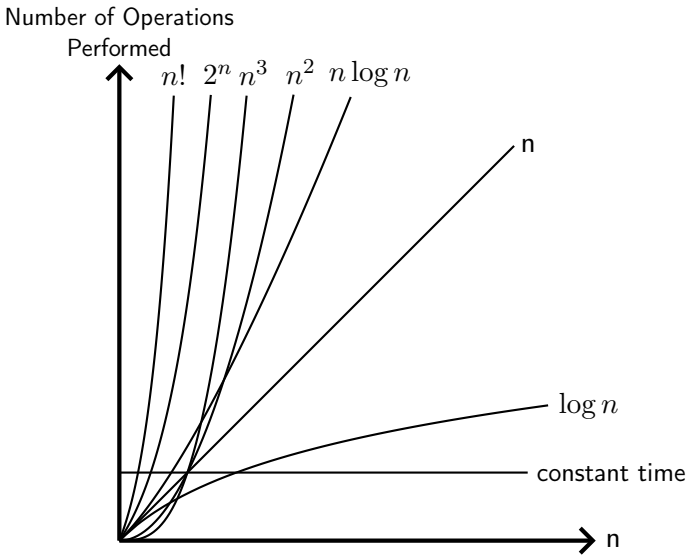
1.2 Time Complexity and Big O-Notation

Best Method: Time Complexity

Let's compare the efficiencies of the two algorithms discussed above: 2n and $2n^2+n$. Because $2n^2 + n$ has a quadratic term, it grows significantly faster than the linear 2n as n increases. In fact, the number of operations performed by any algorithm with quadratic terms in its efficiency grows significantly faster than that of any algorithm with only linear terms.



In reality, the size of real-world data tends to grow dramatically, so we are interested in how the number of operations performed by an algorithm scales with large inputs. With a large input, an algorithm with quadratic efficiency seems to be on a different league from algorithms with linear efficiency. Therefore, computer scientists have categorised the efficiency of algorithms based on how the number of operations performed grows as the input size becomes large. It is called time complexity. It's surprisingly simple. For any efficiency, we simply take the highest power of n (the fastest-growing term). For example, an algorithm that always performs a constant number of operations regardless of input size (say 5) has a time complexity of O(1); an algorithm with an efficiency of $2n$ has a time complexity of O(n); an algorithm with an efficiency of $2n^2 + n$ has a time complexity of O(n^2). Likewise, if an algorithm has an efficiency of $3n^3 - n + 5$, it has a time complexity of O(n^3).



There are many other types of time complexities, each in a separate league in terms of how the number of operations scales with input size. For most algorithms, you will probably only encounter O(n) and O(n^2) time complexities. In future chapters of the book, we will explore algorithms that will have more exotic time complexity. In binary search, we will be exposed to O(log n). In backtracking we will learn algorithms with time complexities O(2^n).

What exactly is "an operation"?
You are probably silently yelling in frustration about what exactly counts as "an operation". However, this question matters less than you might think. Concretely, an operation is any action whose execution requires a constant (independent of input size) number of exact hardware steps. Take the previous example:

```
array = [1, 2, ..., n]
for num in array:
    print(num)
```

A hardware step is a CPU instruction. Let's say going through a number requires 10 CPU instructions, and printing a number requires 20 CPU instructions. Both actions require a constant number of CPU instructions, and therefore, they are each considered an operation.

Why is an operation so vaguely defined? In this case, doesn't printing a number take longer than going through a number, yet they are both considered "an operation"? To understand why, let us backtrack and start from the beginning. If we define efficiency as the total number of CPU instructions performed in terms of n, instead of the total number of operations performed in terms of n, in the previous example:

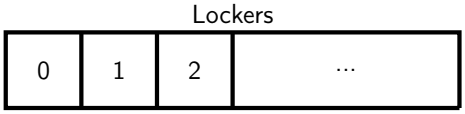
Number of CPU instructions:
1. `for num in array` goes through each element. Let's say going through an element requires 10 CPU instructions. Number of CPU instructions: 10n
2. `print(num)` prints each element once. Let's say printing an element requires 20 CPU instructions. Number of CPU instructions: 20n
Total: 30n CPU instructions

However, as discussed before, we are only interested in how the total number of CPU instructions performed scales with large input. Printing an element might take longer than going through an element, but since both operations require a constant number of CPU instructions, it doesn't matter whether we count the total number of operations or CPU instructions, because they both result in the same time complexity of O(n).

1.3 Worst-Case Scenario and Array Operations

For some algorithms, the number of operations performed varies with the inputs. To determine the time complexity of such algorithms, we usually focus on the worst-case scenario – the input that results in the maximum number of operations. This is because the worst-case scenario represents the upper limit of an algorithm and gives us a guaranteed bound on how slow an algorithm can ever be, regardless of the input. We will explore this idea with a few array operations.

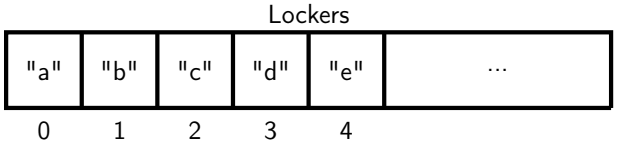
Firsly, how do we even start thinking about arrays?
An array is a contiguous data structure. This means that all the elements are stored in a single, unbroken block of memory, sitting right next to each other. You can think of it using an analogy: imagine your school has a long row of lockers. Each locker can store only one item, and they are labeled from 0, 1, 2, ...



You can think of the following array:

$$\text{array} = [\text{"a"}, \text{"b"}, \text{"c"}, \text{"d"}, \text{"e"}]$$

as:



where locker 6, 7, ... are empty.

Array operation: Checking existence of an element in array – O(n)
Suppose we want to check if a specific element exists in our array, `["a", "b", "c", "d", "e"]`. How do we do that? Since we have no idea which locker the item might be stored in, our only choice is to check each locker, one by one, from the beginning. This is known as linear search and is exactly what the Python built-in function (e.g. `"c" in array`) for checking existence in an array does. To understand this better, let's explicitly write out the code for what the built-in Python function does behind the scenes:

```
# Linear search  built-in Python function
def linearSearch(self, array, target):
    for letter in array:
        if letter == target:
            return True
    return False

# "c" in array
linearSearch(["a", "b", "c", "d", "e"], "c")
```

Now let's look at the best and worst-case scenarios for this algorithm:

Best-case scenario: The item we're trying to check for happens to be stored in the first locker. In just one step, we can say that the item exists in our hallway of lockers. Therefore, in the best-case scenario, the algorithm is O(1).

Worst-case scenario: The item we're trying to check for unfortunately does not exist in any locker, or it is stored in the last non-empty locker. In these cases, it takes n steps because we have to check all n non-empty lockers just to determine whether the item exists. Therefore, in the worst-case scenario, the algorithm is O(n).

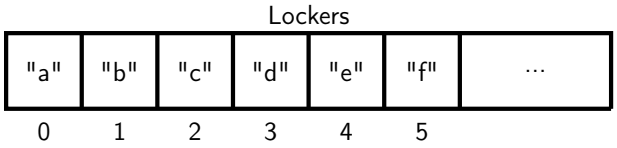
```
# Best case scenario -- O(1)
linearSearch(["a", "b", "c", "d", "e"], "a")

# Worst case scenario -- O(n)
linearSearch(["a", "b", "c", "d", "e"], "e") OR
linearSearch(["a", "b", "c", "d", "e"], 99)
```

Since we consider the worst-case scenario, we conclude that checking whether an element exists in an array has a time complexity of O(n).

Array operation: Appending an element to array – O(1)
Appending an element to an array refers to adding an element to the end of the array. For a quick recap, the Python code for appending an element to an array is:

```
array = ["a", "b", "c", "d", "e"]

# Appending — add element to the end of the array
array.append("f")
# array = ["a", "b", "c", "d", "e", "f"]
```

Think about how this works in our analogy. All you have to do is simply go to the locker after the last non-empty locker and place the item inside it.



There is neither a best-case nor a worst-case scenario because appending an element always only takes one step. Therefore, we can conclude that appending has a time complexity of O(1).

Array operation: Removing an element at an index - O(n)
Now, we want to remove the element at a specific index in the array. We can use the built-in Python function to do so. For example, if we wish to remove the element at index 2 of our array:

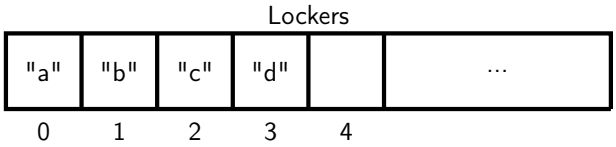
```
array = ["a", "b", "c", "d", "e"]

# Removing element at an index
array.pop(2)
# array = ["a", "b", "d", "e"]
```

How this built-in function works behind the scenes starts to get a bit complicated. Remember I said that an array is a contiguous data structure at the start of this section? Here's how you can think about it: imagine your school has a very demanding, unreasonable, and OCD teacher who made the following rules on how the lockers must be used:
1. The first locker (locker number 0) must be used.
2. There must not be any empty locker where both its neighbours are used.

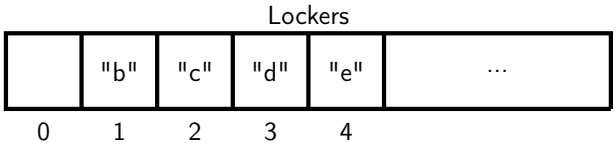
Therefore, in the previous example, after you remove the item from locker number 2, rule 2 is clearly broken. To avoid getting punished, you have to shift the items in each locker after locker number 2, one locker leftward. Now, let's look at the best-case and worst-case scenarios.

Best-case scenario: You want to remove the item from the last non-empty locker. After removing it, rules 1 and 2 are both still satisfied, and you do not need to do anything else. Therefore, the best-case scenario takes only one step and thus has a time complexity of

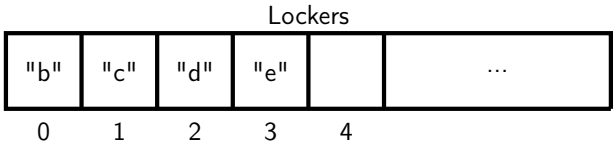
O(1).



Worst-case scenario: You want to remove the item from the very first locker. This takes 1 step. After emptying the first locker, rule 2 is blatantly broken:



To satisfy rule 2, we have to shift the items in each non-empty locker one locker to the left. This takes n-1 steps.



Altogether, the worst-case scenario takes a total of n steps and thus has a time complexity of O(n). Since we consider the worst-case scenario for time complexity, we can conclude that removing an element at a specific index in an array has a time complexity of O(n).

Other array operations

I believe that you are now well-versed in how arrays behave and can determine the time complexities of other array operations. I will now show you a list of common array operations along with their time complexities. Try to understand why they have these time complexities!

```
array = ["a", "b", "c", "d", "e"]

# Checking existence of elements - O(n)
print("d" in array)
# Prints True

# Accessing Elements - O(1)
print(array[2])
# Prints "c"

# Modifying Elements - O(1)
array[2] = "z"
# array == ["a", "b", "z", "d", "e"]
```

```
# Appending Elements - O(1)
array.append("f")
# array == ["a", "b", "c", "d", "e", "f"]

# Inserting Elements - O(n)
array.insert(2, "z")
# array == ["a", "b", "z", "c", "d", "e"]

# Popping Elements - O(1)
array.pop()
# array == [["a", "b", "c", "d"]

# Rermoving element at an index - O(n)
array.pop(1)
# array == ["a", "c", "d", "e"]
```

1.4 Space Complexity

Space complexity is conceptually very similar to time complexity. Space complexity seeks to measure how space-efficient an algorithm is – how much extra memory an algorithm needs, apart from the space used by the input data itself. Just like time complexity, a good way to measure this is by calculating the amount of extra space used in terms of the input size, n. But once again, in reality, the size of real-world data grows dramatically, so we're only interested in how the amount of extra space needed scales with large input – we only care about the fastest-growing term.

Let us look at two simple examples.
1. The following algorithm prints the first element of the array.

```
array = [1, 2, ..., n]
result = array[0]
print(result)
# Prints 1
```

In the above example, the input size is n but it only need one extra space (`result = array[0]`). Therefore, this algorithm has a space complexity of O(1).

2. The following algorithm doubles each element of the array and prints the resulting array.

```
array = [1, 2, ..., n]
result = [ ]
for num in array:
```

```
    result.append(num * 2)
print(result)
# Prints [2, 4, ..., 2n]
```

In the above example, the input size is n and the extra space needed (`result = [2, 4, ..., 2n]`) is n. Therefore, this algorithm has a space complexity of O(n).

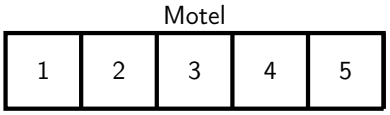
Now, what exactly is a "space"?
An integer stored in a variable in the first example and an integer stored in an array in the second example might not use the exact same amount of memory. Let's say they use 4 bytes and 8 bytes, respectively. However, similar to the discussion on time complexity, we are only interested in how the amount of memory scales with the input size. Therefore, since they both use a constant (independent of the input size n) number of bytes, we consider each of them as "one unit of space."

Do we also use the worst-case scenario to determine space complexity?
Yes. And this is for the same reasons as why we use the worst-case scenario for time complexity – it gives us the upper limit of the space-efficiency of our algorithm.

2 Hash Tables

Let's say we have an array and we want to check whether a specific element is in the array. In the last chapter, we said that we need to loop through the array and individually check each element to do so. This has a time complexity of O(n), which is quite inefficient. Is there a data structure that can do better than that? In this chapter, I will introduce you to hash tables – a data structure that allows us to search for an element in O(1) time. I will explain why hash tables are so efficient, how to implement them in Python, and, lastly, some common LeetCode patterns involving hash tables.

2.1 Take 1 – The Inexperienced Motel Owner



Congratulations, you just opened a motel! It is a small and cozy motel with five rooms. Today is your first day on the job and frankly, you don't really know what you are doing. Each time a guest arrives, you simply give him a vacant room without taking note which room you are assigning them. One day, you wanted to look for a guest and realised you had to check each room, one by one, to find him! In the worst possible twist of luck, the guest was in the very last room you checked. That's 10 minutes of your life you'll never get back.

You learned your lesson. Now, when you assign each guest a room, you remember that room number. Whenever you wish to locate a guest, you can simply recall his room number and go knocking on his door.

2.2 What is a Hash Table?

Think about it. Conducting a linear search on an array is just like finding the guest. Just as you didn't know which room the guest was in, you don't know which index the element is at. Thus, similarly, you have to check each element of the array to find it and in the worst-case scenario, that element is the last element you check. Linear search was already covered in chapter 1 and it has a time complexity of O(n).

In our analogy, you averted the crisis by remembering each guest's room number. Can we learn something from it? Similarly, when we assign each piece of data to be stored at a memory address, can we also "remember" this address? Then, if we wish to access this data, we can simply look up the memory address. This is the basic idea behind hash tables.

2.3 Hash Function

So, the idea behind hash tables is to "remember" where each data item is stored so that we can look it up in O(1) time right? Wait, don't we have to use more memory to store where each data item is stored? This sounds very memory inefficient and perhaps even somewhat self-contradictory. We can solve this problem using the idea of a hash function.

Think of a hash function as a way to assign the address where a data item will be stored in the hash table, based on something "innate" to that data. Then, if we wish to check if this data exists, we can use the hash function on this "something innate to it" to calculate its address in the hash table and check there. Since we are using the hash function on something "innate" to the data, we don't need to "remember" anything and therefore, this solves our paradox.

This probably sounds quite confusing; I get it. So let me explain what I mean using the same analogy. You can think of a guest as a data item and the "something innate to him" is his name. Now, we want to create a hash function to assign him one of the five room. One simple hash function would be to take the remainder of the alphabet sum of their names divided by 5. For example,

$$\text{John} = (10+15+8+14)\,\%\,5 = 47\,\%\,5 = 2$$

considering a=1, b=2, ..., z=26.
Since the remainder of the alphabetical-sum of "John" is 2, you assign him to room 2. You do the same for the other guests, such as Tom and David.

Room	Name
0	David
1	
2	John
3	Tom
4	

Hash Function
John: 47 % 5 = 2
Tom: 48 % 5 = 3
David: 40 % 5 = 0

Now, if you want to check if John is in the motel, we can once again use the hash function on "John" and realise that John should be in room 2. You then proceed to check that room. Since we only have to check one room, this has a time complexity of O(1)

This is essentially how a hash table works. We use a hash function to determine where the data will be stored; and when we wish to look up that data, we again use the hash function to calculate where it is stored and access it in O(1) time. In reality, the hash function is not that simplistic and there can be many different hash functions.

2.4 Collisions and Separate Chaining

It turns out that two guests with different names can have the same remainder when divided by 5. For example, using our previous hash function, Mary and John are both assigned to the same room! Surely, the solution to such "collisions" is not to make them share a room right? But hey, a solution is still a solution!

Since the hash function can cause multiple elements to be stored at the same address of a hash table, this results in collisions. One common approach to deal with this is called separate chaining, which links the multiple elements together using a linked list (which we will learn in future chapters) and then stores them at that address.

Room	Name
0	David
1	
2	John → Mary
3	Tom
4	

Hash Function
John: 47 % 5 = 2
Tom: 48 % 5 = 3
David: 40 % 5 = 0
Mary: 57 % 5 = 2

2.5 Time Complexity

In the previous section, we said that if our hash function assigns two different data items to the same address in our hash table, we store both of them at the same address using a linked list. To recap, you can think of John and Mary sharing the same room if the alphabetical-sum of their names each have a remainder of 2 when divided by 5 as shown below.

Room	Name
0	David
1	
2	John → Mary
3	Tom
4	

How do collisions affect the time complexity? Is the time complexity still O(1)? In the analogy, you would probably have guessed that it is still O(1) time. After all, you just have to use the hash function to calculate that Mary is staying at room 2 and then check if Mary is there. However, this is when the analogy breaks down. As you will learn in future chapters, to access the a specific element of a linked list, you have to traverse through the linked list from the beginning until you reach it. Sure, looking up John takes one step, because he is at the beginning of the linked list. But to look up Mary, it would take two steps, because you have to "traverse" through John to "access" her. Okay, so what is the time complexity of searching for an element if there are collisions? To answer that, we look to the worst-case scenario. Let's assume we used another hash function instead, and this function is really bad (or our luck is just really bad), causing all our items to be stored in the same address of our hash table like the one shown below.

Room	Name
0	
1	
2	John → Mary → Tom → David
3	
4	

In the worst-case scenario, you want to search for David and have to "traverse" through all n guests before "accessing" him. This takes n steps. Therefore, in the worst-case, searching has a time complexity of O(n). Let's now look at the time complexity of other operations, such as adding and removing data from a hash table, in the worst-case scenario.

In the worst case scenario – every data are stored in the same address:

Searching: O(n) as explained earlier.
Adding: We don't want duplicates of any key at an address. Therefore, you have to traverse the entirety of the linked list to verify that the key is unique before inserting it at the end. This results in a time complexity of O(n).
Removing: If the data we wish to remove happens to be at the end of the linked list, we need to traverse to the end of the linked list before removing it. Therefore, this has a time complexity of O(n).

In reality, however, hash functions are usually very good at distributing items evenly, resulting in few collisions. This is known as the average-case scenario, and let us see how the time complexity of those operations changes.

In the average case scenario – very few addresses have multiple data items:

Searching: O(1) since each address usually stores only one item.
Adding: O(1) since you usually add items to empty addresses.
Removing: O(1) since you usually remove the one and only item from an address.

This is why hash tables are so powerful. It allows us to search, add and remove items in O(1) time, whereas an array may require O(n) time to perform the same operations.

Why do we not use the worst-case scenario for time complexity?
In Chapter 1, we said that we should analyze the worst-case scenario to determine the time complexity of an algorithm. Why, then, do we consider the average case instead in this context? The reason is that, for a good hash function, the probability of encountering the worst case is extremely low – so low that it is reasonable to ignore it entirely in practical analysis.

2.6 Hash Sets, Hash Maps and Their Implementation in Python

Hash Sets – Sets
In Python, hash sets are called sets. A hash set is simply a hash table where each address only stores keys. This is something we're already familiar with. To determine which address each key is stored at, we apply a hash function to the key. For example, we want to create the following hash set:

Hash table

0	"orange"
1	
2	"apple"
3	"banana"
4	

This is how we can implement the hash set in Python and also some of its common operations.

```python
# Creating an empty hashset
mySet = set()

# Or with initial values
mySet = {"apple", "banana", "orange"}

# Check existence of elements
print("banana" in mySet)
# Prints True

# Adding elements
mySet.add("grape")
# mySet = {"apple", "banana", "orange", "grape"}

# Removing elements
mySet.remove("apple")
# mySet = {"banana", "orange"}
```

Hash map – Dictionaries

In Python, hash maps are called dictionaries. The idea of hash maps differs slightly from hash sets but it is actually quite simple. Hash maps store key:value pairs at addresses of the hash table, instead of just keys. Note that, to determine which address to place a key:value pair at, we apply the hash function only to the key; so only the key determine which address of the hash table the pair would be stored at. For example, we want to create the following hash map with fruit:number pairs. Notice that each key is stored at the same address as in the hash set earlier (assuming the same hash function was used).

Hash table

0	"orange":30
1	
2	"apple":10
3	"banana":20
4	

Below is how we can implement the above hash map and some of its common operations.

```
# Create an empty dictionary
d = {}

# Or with initial values
d = {"apple": 10, "banana": 20, "orange": 30}

# Checking existence of keys
print("apple" in d)
# Prints True

# Adding/updating new/existing key-value pair
d["grape"] = 40
d["banana"] = 99
# d = {"apple": 10, "banana": 99, "orange": 30, "grape": 40}

# Access value by key
print(d["apple"])
# Prints 10

# Using get to avoid KeyError if key doesn't exist
print(d.get("pineapple", "Not found"))
# Prints Not found
```

2.7 What is Hashable?

So, what type of elements are hashable? By this, I mean: what types of elements can be used as keys in a hash set/hash map? Generally, immutable objects are hashable, whereas mutable objects are not.

Note: An immutable object means that its content cannot be changed after it is created (e.g. `int`, `float`, `bool`). Vice versa, a mutable object means that its content can be changed after it is created (e.g. `list`, `set`, `dict`)

Hashable	Not Hashable
`int`: 4	`list`: [1, 2, 3]
`float`: 3.14	`set`: 1, 2, 3
`bool`: True	`dict`: "a": 1
`str`: "hello"	
`tuple`: (1, 2, 3)	

If you put a non-hashable item as a key in a set or dictionary, you will get an error:

```
# Attempting to add a list into a set
aList = [1, 2]
mySet = set()
mySet.add(aList)
# Output: TypeError: unhashable type: 'list'

# Attempting to add a set as a key into a dictionary
aEmptySet = set()
myDict = {}
myDict[aEmptySet] = 2
# Output: TypeError: unhashable type: 'set'
```

These restrictions only apply to keys, and not values. Because if a key were mutable and its data changed, its hash code would also change. Consequently, when you later try to retrieve the value, the hash function would calculate a different address and fail to locate the data, even though it is still in memory. Values have no such restriction because they are not used to calculate the storage location; they are simply the data stored at the address the key provides.

2.8 Pattern: Fast Lookup

For algorithms that require us to repeatedly check the existence of specific elements in an array, it would be wise to first add the elements to a hash table. Then, whenever we wish to check for a specific element, we can do that in O(1) time, instead of O(n).

Problem: Contains Duplicates:
Given an integer array `nums`, return `True` if any value appears more than once. Otherwise, return `False`.

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

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

Solution 1: Brute Force

```python
class Solution(object):
    def containsDuplicate(self, nums):
        for i in range(len(nums)-1):
            for j in range(i+1, len(nums)):
                if nums[i] == nums[j]:
                    return True
        return False
```

How this works in broad strokes:
Using a nested for loops, we check every pair of numbers once.

Time complexity: O(n^2)
Space complexity: O(1)

Solution 2: Hash Set

```python
class Solution(object):
    def containsDuplicate(self, nums):
        seen = set()
        for num in nums:
            # Checking if num was seen before
            if num in seen:
                return True
            # Remembering that num was seen
```

```
            seen.add(num)
        return False
```

How this works in broad strokes:
We first create an empty set to store seen values. As we iterate through `nums`, we check if we have seen the number before by checking its existence in `seen`. We also add `num` to the `seen`, so that in the future, we can "remember" that we've seen this number.

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

Problem: Two Sum
Given an array of integers `nums` and an integer `target`, return the indices (i and j) of the two numbers such that they add up to `target`. Return the indices as an array `[i, j]`, in any order. Note that `i != j` and assume that each input have exactly one solution.

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

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

Solution:

```python
class Solution(object):
    def twoSum(self, nums, target):
        # Use hash map to store complement:index pairs
        complement = {}
        # Adding to hash map
        for i, number in enumerate(nums):
            complement[target-number] = i
        # Checking existence in hash map
        for i, number in enumerate(nums):
            if number in diff and i != complement[number]:
                return [i, complement[number]]
```

How this works in broad strokes:
For a `num`, if `target – num` also exists, they make up valid pair of solution. We can say that the complement of `num` is `target – num`. Now, we iterate through the array to store all `complement of nums : index of nums` pairs in a hash map. Then, we iterate through array

again and use the hash map to check if any `num` is a complement to any other elements. Note that we also check if `i != complement[number]` because `i != j`.

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

Problem: Longest Consecutive Sequence
You are given an array of integers `nums`. Return the length of the longest consecutive sequence of numbers that can be formed.

Example 1:
Input: `nums = [0, -1, 10, 1, 2, 9, 8]`
Output: `4`
Explanation: `[-1, 0, 1, 2]` is the longest consecutive sequence.

Example 2:
Input: `nums = [ ]`
Output: `0`

Solution:

```python
class Solution:
    def longestConsecutive(self, nums):
        s = set()
        longest = 0
        # Adding all num to set
        for num in nums:
            s.add(num)
        # Finding length of longest sequence with num as the start
        for num in nums:
            length = 1
            if num-1 not in s:
                while num+length in s:
                    length += 1
            # Comparing this length to the longest length encountered thus far
            longest = max(longest, length)
        return longest
```

How this works in broad strokes:
We first create a hash set and add every `num` into it. Then, for each `num` in the array, we want to find the length of the longest consecutive sequence with `num` as the start of the sequence. We can find this length by using a hash set to continuously check if the next number exists in the set. But realise we can ignore those `num` if `num-1` is also in the set, because starting the sequence with `num-1` yields a longer sequence. Therefore, we find the lengths of the longest sequences that we can make with each of the remaining `num` (`num-1`

is not in the set) and return the length of the longest sequence among them.

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

2.9 Pattern: Frequency Tracking

For solutions that require us to track the frequency of appearance of elements, a hash map is the perfect data structure for it! We can store key:value pairs, where the keys are the elements and the values are the frequency of each element. Since the value can be mutable, we can continuously update it throughout. We can also look up specific elements, along with their frequency in O(1) time.

Problem: Anagram
You are given two strings `s1` and `s2`. Two strings are anagrams of each other if they each can be rearranged to become the other. Return `True` if `s1` and `s2` are anagrams of each other. Otherwise, return `False`.

Note: `s1` and `s2` can only contain lowercase English letters.

Example 1:
Inputs: `s1 = "true"`, `s2 = "rteu"`
Output: `True`

Example 2:
Inputs: `s1 = "cat"`, `s2 = "dog"`
Output: `False`

Solution:

```
class Solution:
    def isAnagram(self, s1, s2):
        # Use hash maps to store characters:frequency pairs
        s1Freq = {}
        s2Freq = {}
        # Track frequency of each letter for s1 and s2 in hash maps
        for letter in s1:
            s1Freq[letter] = 1 + s1Freq.get(letter, 0)
        for letter in s2:
            s2Freq[letter] = 1 + s2Freq.get(letter, 0)
        # Checking if s1 and s2 are anagrams
        return s1Freq == s2Freq
```

How this works in broad strokes:
To check if two strings are anagrams of each other, we have to check that they contain the same characters and same frequencies of those characters. Therefore, for `s1` and `s2`, we create two hash maps, `s1Freq` and `s2Freq`, to store character:frequency pairs of each string. To check if the two strings are anagrams of each other, we simply check if the two hash maps are equal.

Time complexity: O(n+m) where n and m are the lengths of `s1` and `s2` respectively.
Space complexity: O(1) since there are at most 26 unique characters.

Problem: Group Anagrams
You are given an array of strings, `strs`, group all anagrams with each other into subarrays. Return the resultant array and the subarrays can be arranged in any order. `s` can only contain lowercase English letters.

Example 1:
Input: `strs = ["Acre", "Hello", "Art", "Care", "Rat", "Tar", ]`
Output: `[ ["Acre", "Care"], ["Art", "Rat", "Tar"], ["Hello"] ]`

Example 2:
Input: `strs = ["j"]`
Output: `[["j"]]`

Solution:

```
class Solution:
    def groupAnagrams(self, strs):
        # Use hash map to store blueprint: array of words with that blueprint pairs
        d = defaultdict( list )

        for s in strs:
            # Use array to store frequency for each 26 lowercase letter
            count = [0] * 26
            # frequency of "a" is store at count[0], frequency of "b" at count[1], ...
            for char in s:
                count[ord(char) - ord("a")] += 1
            # Convert list to immutable tuple so it can be used as a dictionary key
            d[tuple(count)].append(s)

        res = [ ]
        for value in d.values():
            res.append(value)
        return res
```

How this works in broad strokes:
For each word, we can use an array with a fixed size of 26 to track the frequency of each lowercase English letter. I will call this the blueprint of a word. Using a dictionary to store `blueprint : array of words with that blueprint`, we iterate through `strs` to append words to the array (value) of its corresponding blueprint (key). This allows us to group anagrams together as the value for their blueprint key.

Note: `defaultdict(list)` automatically initialise a default value of `[ ]` for keys that don't exist yet. More can be found in Python Syntax Cheat Sheet.

Time complexity: O(m∗n)
Space complexity: O(m∗n)
where m is the number of strings and n is the length of the longest string