

TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)

Personal Member ID#: 117667

Unit 4 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as reference as you solve the breakout problems for Unit 4. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 4. In addition to the material below, you will be expected to know any required concepts from previous units. You may also find advanced concepts and bonus concepts included at the bottom of this page helpful for solving problem set questions, but you are not required to know them for the unit assessment.

Standard Concepts

Asymptotic Analysis (Big O)

Course Scope

For this course, when we ask you to evaluate time and memory usage, **evaluate the worst case performance** unless otherwise stated.

For the Unit 3 standard exam, only **constant, linear, and quadratic** algorithms are in scope. We will discuss other common time complexities as they become more relevant in later units.

Computers usually run programs pretty quickly. When we run our functions, it often takes just fractions of a second to receive the output and takes only a tiny portion of the computer's memory. However, even fractions of a second can quickly add up. Have you ever written or used a piece of software that seems to take forever to run? One of the likely culprits for that lag time is that the software processed and retrieved data in a way that was slow or was using too much of a computer's memory.

When we talk about creating quality, safe code, we want to work towards creating **efficient** functions or algorithms. In other words, we want to write algorithms that execute quickly even on large data sets that have 1 million items or more and minimize the use of computer memory. **Asymptotic Analysis** is a method for measuring the execution time and memory usage of a specific algorithm.

Asymptotic analysis has two components:

- **Time complexity** a measurement of the amount of time an algorithm takes to run as the size of the input (the arguments you pass in to your functions!) changes
- **Space complexity** a measurement of the amount of memory an algorithm uses as the size of the input changes

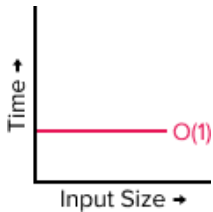
When we evaluate the quality of an algorithm, we generally do not care how an algorithm performs on a specific input size. For example, we are not interested specifically in how long it takes to add a fourth item on to the end of an array of length 3. Instead, we want a more general sense of how the algorithm will perform on both small and large inputs. Time and space complexity allow us to accomplish this by measuring how the runtime and memory usage of an algorithm changes as the input size grows.

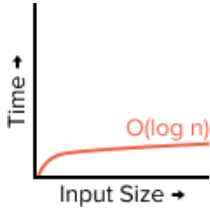
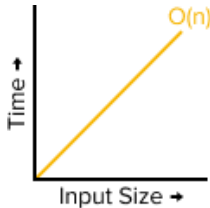
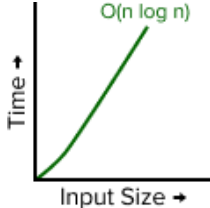
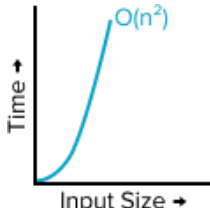
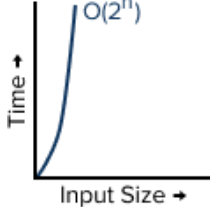
Asymptotic analysis allows us to compare different solutions and select one over another. It also allows us to judge if any algorithm will be able to solve a particular problem within a meaningful amount of time or within the system's limited storage capacity.

Most often, we measure things in terms of *worst-case* performance of an algorithm. This is important when response time and memory usage is critical like when managing a self-driving car or autopilot system. There are also times where the *average-case* performance is important, especially where an algorithm is run often or across many instances, like an analysis program run regularly on a cloud platform; the average runtime (and standard deviation) can be important for a task run thousands of times an hour.

The mathematical notation that we use to describe different trends in time and space complexity is called **Big O** notation. Big O uses the syntax `O(...)` where inside parentheses is some expression that describes the relationship between the size of an algorithm's input and the complexity of the algorithm.

Common Big O complexities are summarized below.

Complexity	Name	Graph	Definition
$O(1)$	Constant	 A graph with 'Time' on the vertical axis and 'Input Size' on the horizontal axis. A horizontal pink line is drawn at a constant level, labeled 'O(1)' at its right end.	The algorithm will take the same amount of time to execute regardless of the size of the input.

Complexity	Name	Graph	Definition
$O(\log n)$	Logarithmic	 A graph showing Time on the y-axis and Input Size on the x-axis. A red curve starts at the origin and increases very slowly, labeled $O(\log n)$.	The algorithm will grow in complexity proportional to the base-2 log of the input size. Logarithmic algorithms increase very slowly as the size of the input increases. They usually involve an algorithm which excludes 1/2 of the input with each iteration of a loop.
$O(n)$	Linear	 A graph showing Time on the y-axis and Input Size on the x-axis. A yellow straight line starts at the origin and increases linearly, labeled $O(n)$.	The algorithm will grow in time or space directly proportional to the input size. The complexity increases at the same rate that the input increases.
$O(n \log n)$	Log Linear	 A graph showing Time on the y-axis and Input Size on the x-axis. A green curve starts at the origin and increases at a rate faster than linear but slower than quadratic, labeled $O(n \log n)$.	A term used to describe an algorithm which will grow in time or space complexity proportional to the $n \log n$ of the input size. "n log n" means that the input size is multiplied by the log of the input size.
$O(n^2)$	Quadratic	 A graph showing Time on the y-axis and Input Size on the x-axis. A blue curve starts at the origin and increases quadratically, labeled $O(n^2)$.	The algorithm will have a runtime or memory usage proportional to the size of the input squared. This often involves 2 nested loops.
$O(2^n)$	Exponential	 A graph showing Time on the y-axis and Input Size on the x-axis. A dark blue curve starts at the origin and increases exponentially, labeled $O(2^n)$.	The algorithm's complexity doubles each time the input size increases by one.

Time Complexity

Whenever we run an algorithm or function, our computer runs a series of **operations** or a set of instructions. Time complexity is the measurement of the number of operations an algorithm executes in comparison to a given input size. It examines the following:

- Given a specific input size, how many operations does the algorithm do?
- When the input size increases by one, how many additional operations are required?
- When we increase the input size by 1 million, how many additional operations does the algorithm do?

Oftentimes, as the size of the input `n` increases, the number of operations performed by the algorithm often also increases. **We care about how the trend in the number of operations changes with the size of the input.**

Constant Time Complexity

Some algorithms take approximately the same or *constant* amount of time to run regardless of the size of the input.

Consider the example below which returns the sum of numbers `1` through `n`.

```
def add_from_1_to_n(n):  
    return (n * (n + 1)) / 2
```

► How does this solution work?

Notice that regardless of the size of `n`, we perform the same number of operations: Whether `n` is 1 or one million, we multiply its value by itself plus one, then divide the resulting product in half.

In Big O terms, we can say an algorithm that perform essentially the same number of operations regardless of the size of the input has a **constant time complexity** or is `O(1)`. Note that `1` is a constant (not variable) value.

Linear Time Complexity

On the other hand, for some algorithms, the amount of time it takes to run the algorithm is proportional to changes in the size of the input. A good example is a loop based version of our function that calculates the sum of numbers `1` through `n`.

```
def add_from_1_to_n(n):  
    total = 0  
    for i in range(1, n + 1):  
        total += i  
  
    return total
```

If the input size doubles, our function will take roughly double the time to run. We can see this in action by estimating the runtime on specific inputs.

Imagine that assigning `total` to 0 takes 1 second, incrementing `i` takes 1 second, and adding `i` to the total also takes 1 second. Regardless of whether the `n` is 10 or 20, initializing `total` to 0 will take the same amount of time (1 second). But if `n` is

`10`, we will need to increment `i` 10 times (10 seconds) and add `i` to `total`

10 times (10 seconds) resulting in an overall runtime of `1 + 10 + 10 = 21` seconds.

If `n` is 20, we will need to increment `i` 20 times (20 seconds) and add `i` to `total` 20 times (20 seconds) resulting in an overall runtime of `1 + 20 + 20 = 41` seconds.

A function where the runtime increases in direct proportion to the size of the input is a function with **linear time complexity** or $O(n)$.

If we altered our algorithm slightly to instead use a loop to print all the numbers from 1 to n and then down from n to 1 it would take $O(n + n) = O(2n)$ time to execute. For Big O, we drop any coefficients or constant terms. Thus $O(2n) \Rightarrow O(n)$.

```
def count_up_and_down(n):  
    # Takes `n` time to run  
    for i in range(1, n + 1):  
        print(i)  
  
    # Takes `n` time to run  
    for i in range(n, 0, -1):  
        print(i)
```

So although `count_up_and_down()` may be doing more operations than `add_from_1_to_n()` it also has a **linear time complexity** or $O(n)$.

In general, whenever we do operations that are sequential (one after another) we add their time complexities together. If the time complexities being added are different, say an $O(n)$ and $O(1)$ operation (as opposed to the $O(n) + O(n)$ operation in the example above), the less dominant (faster/better) time complexity is dropped. Thus $O(n + 1) \Rightarrow O(n)$.

Quadratic Time Complexity

Often we encounter algorithms which involve nested loops. For example the code below contains a for loop which runs n times and a loop nested inside of it which runs, worst-case, $n-1$ times where n is the length of `numbers`.

```
def duplicates_within_k(numbers, k):  
    lst_length = len(numbers)  
  
    if lst_length < 2 or k == 0:  
        return False  
  
    for i in range(lst_length):  
        j = i + 1  
        dist_remaining = k  
        while dist_remaining > 0 and j < lst_length:  
            if numbers[i] == numbers[j]:  
                return True  
            j += 1  
            dist_remaining -= 1  
    return False
```

When we encounter nested loops, the total number of operations performed is the product of the number of iterations of both the outer and inner loops.

in this case:

- If we multiply the number of iterations the outer loop does by the number of iterations the inner loop does in the worst case we have $(n * (n-1))$ which is equivalent to $(n^2 - n)$.
- We drop the linear term n because, for large values of n , the quadratic term n^2 grows much faster and dominates the overall complexity.
- We end up with a time complexity of $O(n^2)$.

This is called a **quadratic time complexity** or $O(n^2)$.

Space Complexity

Whenever we run an algorithm, our computer uses memory. Every variable, literal, and object in our program takes up some amount of memory. Space complexity is the measurement of the number of memory units an algorithm uses while executing in comparison to a given input size. It examines the following:

- Given a specific input size, how many variables do we store in memory?
- When the input size increases by one, how many additional variables does the algorithm store in memory?
- When we increase the input size by 1 million, how many more variables are stored in memory?

The rules with regard to space complexity are:

- Most single-value variables take up $O(1)$ space.
 - booleans, integers, floats, etc.
 - Sometimes these are called *primitives*
- Strings take up $O(n)$ space where n is the string length
- Lists take up $O(n)$ space where n is the list length
- Dictionaries take up $O(2n)$ space which reduces to $O(n)$ where n is the number of key-value pairs

Constant Space Complexity

Many functions have constant space complexity. Consider the following function which sums the integer elements of an array `arr`.

```
def sum_array(arr):
    total = 0
    for num in arr:
        total += num
    return total
```

While our function does create two additional variables `total` and `num`, it creates these two variables *regardless* of the size of the input list `arr`.

Further, the size of the input list `arr` is not considered in this calculation, as it exists outside the function. It was created outside of the `sum_array()` algorithm and then passed in as an argument. When calculating space complexity, only the space taken by variables created within the function affects the space complexity.

Linear Space Complexity

Linear space complexity is common when we create new data structures like lists or dictionaries inside of our function. Consider the following function `copy_array()` which accepts an `arr` of length `n` as input.

```
def copy_array(arr):
    copy = []
    for num in arr:
        copy.append(num)
    return copy
```

As the function name implies, the algorithm creates a *new* array `copy`, whose size directly correlates to that of our input list `arr`. If `arr` has size `1`, `copy` also has size `1`. If `arr` has size `n`, `copy` also has size `n`. Therefore, the space complexity of the function is $O(n)$.

Quadratic Space Complexity

Quadratic space complexity is most often seen when building new, nested data structures within an algorithm. For example, consider the following function which creates an `n x n` matrix.

```
def init_matrix(n):
    matrix = []
    for row in range(n):
        matrix.append([])
        for col in range(n):
            matrix[r].append(None)
    return matrix
```

We essentially create `n` lists of size length `n` (one for each row), plus the additional outer list to hold each row. Thus, we have $O(n^2)$ space complexity.

Quadratic space complexity is also possible when building a 1D array or data structure. For example `all_pairs()` generates and stores all possible pairs `(i, j)` from a list of `n` elements. The space complexity is quadratic because the number of pairs is proportional to n^2 .

```
def all_pairs(arr):
    n = len(arr)
    pairs = []
    for i in range(n):
        for j in range(n):
            pairs.append((arr[i], arr[j]))
    return pairs
```

Accounting for Big O of Built-In and Helper Functions.

When evaluating time and space complexity, we also need to consider the Big O of any built-in or helper functions our primary algorithm calls while executing.

For example, consider the following function that accepts a list `arr` and an integer `k` and returns the `kth` smallest element in the list.

```
def kth_smallest_element(arr, k):  
    arr.sort()  
    return arr[k-1]
```

The list `sort()` method actually has $O(n \log n)$ time complexity. Therefore the overall time complexity of the function is $O(n \log n)$.

Common Built-In Time Complexities

⚠️ We do not expect you to memorize the Big O of these functions for exams, but it is a good idea to gradually learn them so you know them when you are in an interview setting.

Function	Time Complexity
<code>len()</code>	$O(1)$
<code>sequence[idx]</code>	$O(1)$
<code>sequence[idx: idx + k]</code>	$O(k)$
<code>sequence.sort()</code>	$O(n \log n)$
<code>sorted(sequence)</code>	$O(n \log n)$
<code>sequence.copy()</code>	$O(n)$
<code>lst.append()</code>	$O(1)$
<code>lst.pop()</code>	$O(1)$
<code>lst.insert()</code>	$O(n)$
<code>dict.get()</code>	$O(1)$
<code>dict.pop()</code>	$O(1)$

If you are ever unsure of a built-in or library function's Big O, the best way to learn it is to search online!

Advanced Concepts

Complexities with Multiple Variables

We may also encounter time complexities which use variables other than `n`. One typical scenario for this is when we have multiple inputs, each with variable size, that both affect time complexity.

For example, say we have the following function which will make a 2D matrix with `rows` number of rows and `columns` number of columns where all values inside the matrix are initialized to `None`.

```
def init_matrix(rows, columns):  
    matrix = []  
    for r in range(rows):  
        matrix.append([])  
        for c in range(columns):  
            matrix[r].append(None)  
    return matrix
```

We could say this algorithm has an **`O(m*n)` time complexity** where `m` is the number of rows and `n` is the number of columns. When discussing time complexity, we want to use separate variables to represent the number of rows and number of columns because their respective values could be drastically different. We could be trying to create a matrix with just 1 row and 1000 columns.

Multiple variables is one reason it is very important that we clearly define what our Big O variables represent when discussing time and space complexity. When multiple variables are at play, we want to communicate clearly which ones are affecting the overall runtime and/or memory usage.