This is a mock of what an interview cheat sheet could look like. The key is to personalize it with what's valuable to you.

[Python] Interview Cheat Sheet

Big-O Complexities

TIME COMPLEXITY	MAX N (LOW END)	MAX N (HIGH END)
$O(n!)$	11	9
$O(2^n)$	27	20
$O(n^3)$	584	125
$O(n^2)$	14,142	1,414
$O(n \log n)$	8,677,239	118,649
$O(n)$	200,000,000	2,000,000
$O(\log n)$	Super high	Super high
$O(1)$	Infinite	Infinite

COMMON COMPLEXITY CLASSES

COMPLEXITY CLASS	NOTATION	DESCRIPTION	EXAMPLES OF FUNCTIONS IN THE CLASS
Constant	$O(1)$	The growth rate does not depend on the input.[1]	$18, 3 * 3 * 3, 10!$ $\min(n, 10)$
Logarithmic	$O(\log n)$	Very slow growth rate. When the input doubles, the value only increases by a constant.	$\log_2(n), \log_3(n)$ $6*\log_2(2n+1)$
Linear	$O(n)$	When the input doubles, the value at most doubles.	$n, n/10, 3n + 5$ $n + \log_2(n)$
Linearithmic	$O(n \log n)$	Slightly faster growth than linear. When the input doubles, the value grows by a bit more than double.	$7n * 3\log_2(n)$ $n * \log_2(n) + n$
Quadratic	$O(n^2)$	When the input doubles, the value at most quadruples.	$n(n-1)/2$ $1 + 2 + 3 + ... + n$
Cubic	$O(n^3)$	When the input doubles, the value grows at most eight-fold.	$n^3 + n^2*\log_2(n)$
Exponential (with base 2)	$O(2^n)$	Extremely fast growth rate. When the input grows by one, the value can double.	$2^n, 2^{(n+1)}, 2^n + n^2$
Exponential (with base 3)	$O(3^n)$	Extremely fast growth rate. When the input grows by one, the value can triple.	$3^n, 3^{(n+4)}, 3^n + 2^n$
Factorial	$O(n!)$	Even faster growth rate. When the input grows by one, the value gets multiplied by a factor that increases each time.	$n! + n!$ $1 * 2 * 3 * ... * n$

Technical Interviewing Rubrics

Problem solving:

- How much difficulty did they have finding an algorithm and optimizing it, relative to the difficulty of the problem?
- How much help did they need, relative to other candidates?
- Did they discuss tradeoffs in terms of the time and space complexity?
- Did they use appropriate data structures and algorithms to solve the problem?

Coding:

- Could they translate their ideas into correct code?
- Did they overcomplicate the logic?
- Did they demonstrate good coding hygiene (clear variable names, abstracting relevant code into reusable helper functions instead of copy/pasting, consistent and reasonable style, etc.)?
- Is the code well organized and idiomatic to the language used?
- Did they demonstrate internalized software engineering principles?

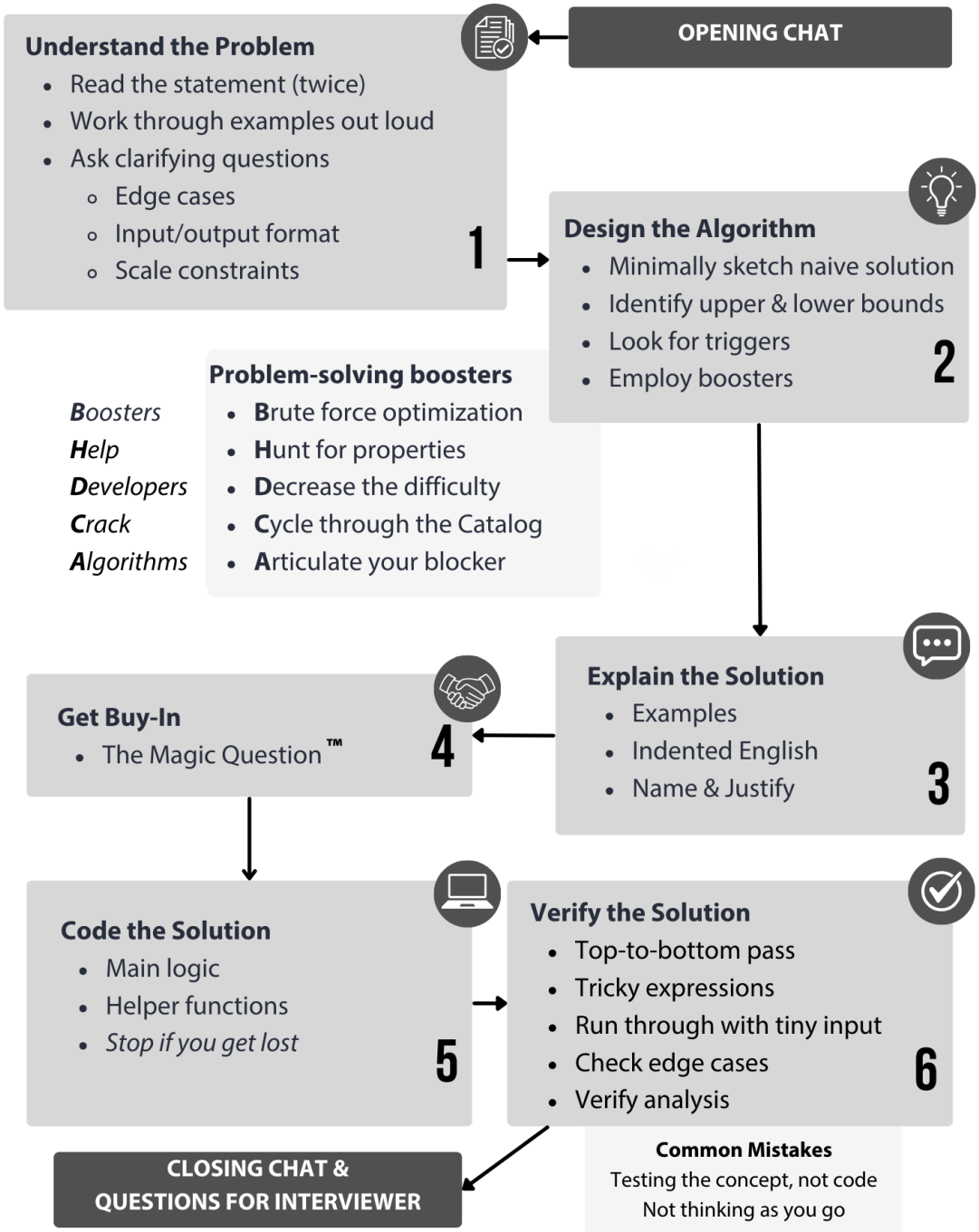
Verification:

- Did they ask good clarifying questions (that weren't already answered in the question description or examples)?
- Did they proactively consider edge cases?
- Did they test their code or provide a good argument for its correctness?
- Were they able to spot and fix any bugs?

Communication:

- Were they able to clearly communicate their thoughts, even when the conversation got technical?
- Were they able to answer technical questions?
- Were they open to feedback?
- If they were confused, were they able to articulate why?

Interview checklist



M.I.K.E. Problem-Solving Framework

- **[M]**inimally sketch the brute force
- **[I]**nfer boundaries
- **[K]**eywords
- **[E]**mploy boosters

Boosters

If boundary & trigger thinking don't point you to the right approach, start with the brute force

Brute Force Optimization

- Preprocessing Pattern
- Data Structure Pattern
- Skip Unnecessary Work

1

If you need a new approach...

Hunt for Properties

- DIY
- Case Analysis
- Reverse Engineer the Output Pattern
- Sketch a Diagram
- Reframe the Problem

2

If you can't find any approach...

Decrease the Difficulty

- Tackle an Easier Version
- Break Down the Problem

3

Solution might be in your blindspot

Cycle Through the Catalog

- *Think: Could ___ be useful?*

4

If you are still stuck...

Articulate Your Blocker

- Don't Say "Hint"
- Show Your Work

5

Technical Topics

String Manipulation

Strings in Python are ***immutable***!
Don't forget about useful methods like:

- `"".join(arr)`
- `string.split(",")`

Binary Search

- **Triggers:** The input is a sorted array/string. The brute force involves repeated linear scans. We are given an optimization problem that's hard to optimize directly.
- **Keywords:** sorted, threshold, range, boundary, find, search, minimum/maximum, first/last, smallest/largest

Transition Point Recipe

```
transition_point_recipe()

# define is_before(val) to return whether val is 'before' initialize l and r to the first and last values in the range to handle the three edge cases:

1. the range is empty
2. l is 'after' (the whole range is 'after')
3. r is 'before' (the whole range is 'before')

while l and r are not next to each other (r - l > 1)
  mid = (l + r) / 2
  if is_before(mid)
    l = mid
  else
    r = mid

return l (last 'before'), r (first 'after'), or something else, depending on the problem
```

Sorting

- Optimal sorts like mergesort, quicksort, heapsort take $O(n \log n)$ time.
- Naive sorts like bubble sort, selection sort, insertion sort take $O(n^2)$ time.
- **Python** uses a variation of mergesort (called Timsort) that takes $O(n \log n)$ time.

Sorting details to remember in Python:

```python
# sort in increasing order
arr.sort() # in-place
res = sorted(arr) # sorted copy of arr

# sort in decreasing order
arr.sort(reverse=True) # in-place
res = sorted(arr, reverse=True) # sorted copy of arr

# sort array of things by an index
things.sort(key=lambda thing: thing[1])

# sort array of objects by an attribute
objects.sort(key=lambda obj: obj.attr)

# sort a custom class without a key by defining your own __lt__
class CustomClass:
    def __init__(self, val):
        self.val = val

    def __lt__(self, other):
        return self.val > other.val # can be much more complicated than this!
```

Python syntax

Syntax:

- **Swap**: `a, b = b, a`
- **Ternary operator**: `a if cond else b`
- **Multiple comparison**: `if a <= b <= c: …`
- **Integer (floor) division**: `//`
- **Exponentiation operator**: `x**y`
- **Walrus operator**: `variable := expression`
- **Duplicate removal**: `arr = list(set(arr))`