Time & Space Complexity Cheat Sheet

BIG O NOTATION HIERARCHY (Fastest to Slowest)

```
O(1) < O(log n) < O(√n) < O(n) < O(n log n) < O(n²) < O(n³) < O(2^n) < O(n!)

Constant < Logarithmic < Root < Linear < Linearithmic < Quadratic < Cubic < Expon
```

ACCEPTABLE TIME COMPLEXITIES FOR HACKERRANK

For n = 10^5 (typical Citadel constraint):

Complexity	Max n	Example
O(1)	Any	Hash lookup, array access
O(log n)	10^18	Binary search
O(√n)	10^14	Prime checking
O(n)	10^8	Single loop
O(n log n)	10^6	Sorting, heap operations
$O(n^2)$	10^4	Nested loops (DANGER for n=10^5)
$O(n^3)$	500	Triple nested loops
O(2^n)	20	Subset generation

For 75-minute test with n=10^5:

- **SAFE:** O(n), O(n log n)
- **RISKY:** $O(n^2)$ will likely timeout
- **NO GO:** $O(n^3)$ or worse

PYTHON DATA STRUCTURES COMPLEXITY

List (Array)

Operation	Average	Worst	Notes
`arr[i]`	O(1)	O(1)	Index access
`arr.append(x)`	O(1)	O(1)	Add to end
`arr.insert(i, x)`	O(n)	O(n)	Insert at position
`arr.pop()`	O(1)	O(1)	Remove last
`arr.pop(i)`	O(n)	O(n)	Remove at position
`arr.remove(x)`	O(n)	O(n)	Remove by value
`x in arr`	O(n)	O(n)	Search
`arr.sort()`	O(n log n)	O(n log n)	In-place sort
`sorted(arr)`	O(n log n)	O(n log n)	New sorted list
`arr.reverse()`	O(n)	O(n)	In-place reverse
`arr[::-1]`	O(n)	O(n)	New reversed list
`min(arr)` , `max(arr)`	O(n)	O(n)	Find min/max
Slicing `arr[a:b]`	O(b-a)	O(b-a)	Create sublist

Dictionary (Hash Table)

Operation	Average	Worst	Notes
`d[key]`	O(1)	O(n)	Access
`d[key] = value`	O(1)	O(n)	Insert/Update
`del d[key]`	O(1)	O(n)	Delete
`key in d`	O(1)	O(n)	Check existence
Iteration	O(n)	O(n)	All keys/values

Set

Operation	Average	Worst	Notes
`s.add(x)`	O(1)	O(n)	Add element
`s.remove(x)`	O(1)	O(n)	Remove element
`x in s`	O(1)	O(n)	Check membership
`s1 | s2`	O(len(s1) + len(s2))		Union
`s1 & s2`	O(min(len(s1), len(s2)))		Intersection
`s1 - s2`	O(len(s1))		Difference

Deque (Double-ended Queue)

Operation	Complexity	Notes
`append(x)`	O(1)	Add to right
`appendleft(x)`	O(1)	Add to left
`pop()`	O(1)	Remove from right
`popleft()`	O(1)	Remove from left
`d[i]`	O(n)	Random access (slow!)

Heap (Priority Queue)

Operation	Complexity	Notes
`heappush(h, x)`	O(log n)	Insert
`heappop(h)`	O(log n)	Remove min
`h[0]`	O(1)	Peek min
`heapify(arr)`	O(n)	Build heap from list

Counter

Operation	Complexity	Notes

Creation	O(n)	Count all elements
`count[x]`	O(1)	Get count
`most_common(k)`	O(n log k)	Top k elements

COMMON ALGORITHM COMPLEXITIES

Sorting Algorithms

```
# Built-in sort - O(n log n) time, O(n) space
arr.sort()  # Timsort
sorted(arr)

# Counting sort - O(n + k) where k is range
# Only for integers in limited range
def counting_sort(arr, max_val):
    count = [0] * (max_val + 1)
    for num in arr:
        count[num] += 1

    result = []
    for num, freq in enumerate(count):
        result.extend([num] * freq)
    return result
```

Search Algorithms

```
# Linear search - O(n)
def linear_search(arr, target):
    for i, val in enumerate(arr):
        if val == target:
            return i
    return -1

# Binary search - O(log n)
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
```

```
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Graph Algorithms

```
# BFS - O(V + E) time, O(V) space
# DFS - O(V + E) time, O(V) space
# Dijkstra - O((V + E) log V) with heap
# Bellman-Ford - O(VE)
# Floyd-Warshall - O(V³)
# Kruskal's MST - O(E log E)
# Prim's MST - O(E log V) with heap
# Topological Sort - O(V + E)
```

Tree Algorithms

```
# Tree traversal - O(n) time, O(h) space
# BST search - O(h) average, O(n) worst
# BST insert - O(h) average, O(n) worst
# Balanced tree operations - O(log n)
```

String Algorithms

```
# Pattern matching (naive) - O(nm)
# KMP pattern matching - O(n + m)
# Rabin-Karp - O(n + m) average
# String comparison - O(min(n, m))
# Substring search - O(n)
```

SPACE COMPLEXITY

Common Space Patterns

```
# O(1) - Constant space
def constant_space(arr):
    result = 0
    for num in arr:
        result += num
    return result
```

```
# O(n) - Linear space
def linear_space(arr):
    return arr[:]  # Copy array

# O(n) - Hash table for frequency
def frequency_count(arr):
    freq = {}  # O(n) space
    for num in arr:
        freq[num] = freq.get(num, 0) + 1
    return freq

# O(h) - Recursion depth for tree
def tree_height(root):
    if not root:
        return 0
    return 1 + max(tree_height(root.left), tree_height(root.right))

# O(2^n) - All subsets
def all_subsets(arr):
    result = []

    def backtrack(index, current):
        if index == len(arr):
            result.append(current[:])
            return

        # Include current element
        current.append(arr[index])
        backtrack(index + 1, current)
        current.pop()

        # Exclude current element
        backtrack(index + 1, current)

    backtrack(0, [])
    return result
```

OPTIMIZATION TECHNIQUES

1. Use Hash Table for O(1) Lookup

```
# SLOW - O(n²)
for num in arr1:
    if num in arr2:  # O(n) search
```

```
        result.append(num)

# FAST - O(n)
set2 = set(arr2)  # O(n) to build
for num in arr1:  # O(n)
    if num in set2:  # O(1) lookup
        result.append(num)
```

2. Avoid Repeated Calculations

```
# SLOW - O(n²)
for i in range(n):
    total = sum(arr[:i])  # Recalculates sum each time

# FAST - O(n)
prefix = [0]
for num in arr:
    prefix.append(prefix[-1] + num)
```

3. Two Pointers Instead of Nested Loops

```
# SLOW - O(n²)
for i in range(n):
    for j in range(i+1, n):
        if arr[i] + arr[j] == target:
            return [i, j]

# FAST - O(n) with sorted array
left, right = 0, n-1
while left < right:
    total = arr[left] + arr[right]
    if total == target:
        return [left, right]
    elif total < target:
        left += 1
    else:
        right -= 1
```

4. Sliding Window Instead of Recalculating

```
# SLOW - O(n²)
for i in range(n - k + 1):
    window_sum = sum(arr[i:i+k])  # O(k) each time
```

```python
# FAST - O(n)
window_sum = sum(arr[:k])
for i in range(k, n):
    window_sum += arr[i] - arr[i-k]  # O(1) update
```

5. Use Deque for Queue Operations

```python
# SLOW - O(n²) because list.pop(0) is O(n)
queue = []
queue.append(x)
x = queue.pop(0)  # O(n)

# FAST - O(1) for all operations
from collections import deque
queue = deque()
queue.append(x)
x = queue.popleft()  # O(1)
```

6. Binary Search Instead of Linear Search

```python
# SLOW - O(n)
for i, val in enumerate(sorted_arr):
    if val == target:
        return i

# FAST - O(log n)
left, right = 0, len(sorted_arr) - 1
while left <= right:
    mid = (left + right) // 2
    if sorted_arr[mid] == target:
        return mid
    elif sorted_arr[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
```

COMMON MISTAKES TO AVOID

1. Nested Loops on Large Input

```python
# TIMEOUT for n=10^5
```

```python
for i in range(n):
    for j in range(n):  # O(n²)
        process(i, j)
```

2. Sorting Inside Loop

```python
# TIMEOUT - O(n² log n)
for i in range(n):
    sorted_sub = sorted(arr[:i])  # Sort every iteration
```

3. String Concatenation in Loop

```python
# SLOW - O(n²) because strings are immutable
result = ""
for char in chars:
    result += char  # Creates new string each time

# FAST - O(n)
result = ''.join(chars)
```

4. Using List for Frequent Membership Testing

```python
# SLOW - O(n) per lookup
seen = []
for num in arr:
    if num not in seen:  # O(n)
        seen.append(num)

# FAST - O(1) per lookup
seen = set()
for num in arr:
    if num not in seen:  # O(1)
        seen.add(num)
```

5. Unnecessary Deep Copies

```python
# SLOW - Copies entire array each time
def backtrack(arr):
    if condition:
        result.append(arr[:])  # OK - need copy here

    for i in range(len(arr)):
```

```
        new_arr = arr[:]  # BAD - unnecessary copy
        new_arr[i] = x
        backtrack(new_arr)

# FAST - Modify in place
def backtrack(arr):
    if condition:
        result.append(arr[:])

    for i in range(len(arr)):
        old_val = arr[i]
        arr[i] = x  # Modify
        backtrack(arr)
        arr[i] = old_val  # Restore
```

QUICK COMPLEXITY CHECKS

Before implementing, ask:

1. **What's n?** (array length, string length, etc.)
2. **How many nested loops?** Each adds O(n)
3. **Am I sorting?** That's O(n log n)
4. **Am I using hash table?** Lookups are O(1)
5. **Am I searching unsorted array?** That's O(n)
6. **Will this timeout?**
    - n=10^5 and $O(n^2)$ → YES
    - n=10^5 and O(n log n) → NO
    - n=10^3 and $O(n^2)$ → NO

RULE OF THUMB FOR CITADEL

- **n ≤ 10:** O(n!) is acceptable (brute force permutations)
- **n ≤ 20:** O(2^n) is acceptable (subset enumeration)
- **n ≤ 500:** $O(n^3)$ is acceptable
- **n ≤ 10^4:** $O(n^2)$ is acceptable
- **n ≤ 10^5:** O(n log n) or O(n) required

- **n ≤ 10^6:** O(n) or O(log n) required

For n=10^5 (most Citadel problems): ✅ O(n), O(n log n) ⚠️ $O(n^2)$ - likely timeout ❌ $O(n^3)$ or worse - guaranteed timeout