Complete Time and Space Complexity Guide (With Examples)

O(1) Constant Time

```
Accessing an element by index:
  arr = [10, 20, 30]
  print(arr[1])  # Output: 20
```

O(log n) Logarithmic Time

```
Binary search on sorted list:
  def binary_search(arr, target):
      low, high = 0, len(arr) - 1
      while low <= high:
          mid = (low + high) // 2
          if arr[mid] == target:
              return mid
          elif arr[mid] < target:
              low = mid + 1
          else:
              high = mid - 1
      return -1
```

O(n) Linear Time

```
Traversing a list:
  for item in [1, 2, 3, 4]:
      print(item)
```

O(n log n) Linearithmic Time

```
Merge sort (simplified structure):
  def merge_sort(arr):
      if len(arr) <= 1:
          return arr
      mid = len(arr) // 2
      left = merge_sort(arr[:mid])
      right = merge_sort(arr[mid:])
      return merge(left, right)
```

O(n^2) Quadratic Time

```
Nested loop:
  for i in range(n):
      for j in range(n):
          print(i, j)
```

Complete Time and Space Complexity Guide (With Examples)

O(2^n) Exponential Time

```
Fibonacci (naive recursion):
  def fib(n):
      if n <= 1:
          return n
      return fib(n-1) + fib(n-2)
```

O(n!) Factorial Time

```
All permutations:
  def permute(nums):
      if len(nums) <= 1:
          return [nums]
      res = []
      for i in range(len(nums)):
          for p in permute(nums[:i] + nums[i+1:]):
              res.append([nums[i]] + p)
      return res
```

Amortized Time Complexity

```
Append in dynamic arrays:
  arr = []
  for i in range(1000):
      arr.append(i)  # O(1) amortized, O(n) when resized

Hash table insert:
  my_map = {}
  my_map["key"] = "value"  # O(1) avg, O(n) worst-case
```

Rare Complexities

```
O(n): Sieve of Eratosthenes
  is_prime = [True] * (n+1)
  for i in range(2, int(n**0.5) + 1):
      if is_prime[i]:
          for j in range(i*i, n+1, i):
              is_prime[j] = False

O(b^d): BFS
  Use BFS in a tree/graph with branching factor b and depth d
```

Data Structures Time & Space

```
Array: O(1) access, O(n) insert/delete
```

Complete Time and Space Complexity Guide (With Examples)

```
Stack/Queue: O(1) push/pop
HashMap: O(1) avg, O(n) worst
Set: Same as HashMap
Linked List: O(n) access, O(1) insert/delete
BST: O(log n) avg, O(n) worst
Heap: O(log n) insert/delete
Trie: O(L), Space O(ALPHABET_SIZE * L)
Segment Tree: O(log n) query/update
Union Find (with path compression): O((n))  constant
```

Space Complexity Examples

```
O(1): Swap in place
  a, b = 1, 2
  a, b = b, a

O(n): Copy list
  copy = arr[:]

O(log n): Recursive binary search call stack
```

Space Optimization Techniques

```
Sliding window (O(1) space):
  def max_subarray(arr, k):
      max_sum = window = sum(arr[:k])
      for i in range(k, len(arr)):
          window += arr[i] - arr[i - k]
          max_sum = max(max_sum, window)
      return max_sum

Bit manipulation:
  x = x ^ y
  y = x ^ y
  x = x ^ y
```

Tips and Pitfalls

- Avoid ignoring recursion stack space
- Hashmaps are not always O(1) in worst-case
- Consider amortized cost for dynamic arrays
- Understand worst vs. average vs. best case
- Input size growth can destroy performance