Lab Session 1

Algorithms, Complexity & Divide-and-Conquer

The Art of Computer Programming 2 — Week 1 — February 21, 2026

Objectives

By the end of this lab, you should be able to:

- Formally prove asymptotic bounds using the O, Ω, Θ definitions
- Apply the master theorem to characterize divide-and-conquer recurrences
- Implement merge sort and binary search from scratch in Python
- Design and analyze a divide-and-conquer algorithm for a new problem
- Empirically verify theoretical complexity through benchmarking

1 Warm-up: Complexity Quick-fire (20 min)

Pen and paper — no computer needed.

1.1 Rank by growth rate

Rank the following functions from **slowest** to **fastest** growing:

$$n^2, \quad \log n, \quad n!, \quad 2^n, \quad n \log n, \quad 1, \quad n, \quad \sqrt{n}, \quad n^3$$

1.2 True or false?

For each statement, answer **True** or **False** and briefly justify.

#	Statement	T/F?	Justification
a	$n^2 = O(n^3)$		
b	$n^3 = O(n^2)$		
c	$2^{n+1} = O(2^n)$		
d	$2^{2n} = O(2^n)$		
e	$\log_2 n = \Theta(\log_{10} n)$		
f	$n^2 + 10^6\, n = \Theta(n^2)$		

1.3 What is the complexity?

Determine the time complexity (in Θ notation) of each code snippet.

(a) Simple nested loop:

```python
def snippet_a(n):
    count = 0
    for i in range(n):
        for j in range(n):
            count += 1
    return count
```

(b) Triangular loop:

```python
def snippet_b(n):
    count = 0
    for i in range(n):
        for j in range(i):
            count += 1
    return count
```

(c) Logarithmic loop:

```python
def snippet_c(n):
    count = 0
    i = n
    while i > 1:
        count += 1
        i = i // 2
    return count
```

(d) Nested with logarithmic inner:

```python
def snippet_d(n):
    count = 0
    for i in range(n):
        j = 1
        while j < n:
            count += 1
            j *= 2
    return count
```

(e) Challenge:

```python
def snippet_e(n):
    count = 0
    i = 1
    while i < n:
        j = 0
        while j < i:
            count += 1
            j += 1
        i *= 2
    return count
```

2 Proving Asymptotic Bounds (15 min)

Using the **formal definitions** from the lecture, prove each of the following.

Recall

$f(n) = O(g(n))$ means $\exists\, c > 0,\ n_0 > 0$ such that $0 \leqslant f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

2.1 Prove that $3n^2 + 5n + 2 = O(n^2)$

Hint

Find explicit constants $c > 0$ and $n_0 > 0$ such that $3n^2 + 5n + 2 \leqslant c \cdot n^2$ for all $n \geqslant n_0$.

2.2 Prove that $n^2 \neq O(n)$

Hint

Use proof by contradiction. Assume $\exists\, c, n_0$ such that $n^2 \leqslant c \cdot n$ for all $n \geqslant n_0$, and derive a contradiction.

2.3 Prove that $n \log n = O(n^2)$ and $n \log n = \Omega(n)$

Conclude: what can you say about $n \log n$ in terms of Θ?

2.4 (Bonus) Prove that $\log(n!) = \Theta(n \log n)$

Hint

Use Stirling's approximation, or bound $n!$ between $(n/2)^{n/2}$ and n^n.

3 Master Theorem (15 min)

Recall

Master Theorem. For $T(n) = aT(n/b) + f(n)$ with $a \geqslant 1$, $b > 1$, compare $f(n)$ with $n^{\log_b a}$:

- **Case 1:** $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
- **Case 2:** $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
- **Case 3:** $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ and regularity condition $\Rightarrow T(n) = \Theta(f(n))$

3.1 Apply the master theorem

For each recurrence, fill in the table. If the master theorem does not apply, explain why.

#	Recurrence	a	b	$n^{\log_b a}$	Case	Solution
1	$T(n) = 4T(n/2) + n$					
2	$T(n) = 2T(n/2) + n^2$					
3	$T(n) = 3T(n/4) + n \log n$					
4	$T(n) = 2T(n/2) + n \log n$					
5	$T(n) = 9T(n/3) + n^2$					
6	$T(n) = T(n/2) + 1$					

3.2 Recursion tree for #4

The master theorem does **not** directly apply to $T(n) = 2T(n/2) + n \log n$.

Why?

There is no $\varepsilon > 0$ such that $n \log n = \Omega(n^{1+\varepsilon})$. The gap between $f(n) = n \log n$ and $n^{\log_2 2} = n$ is only logarithmic, not polynomial.

Draw the recursion tree and answer:

(a) How many levels does the tree have?

(b) What is the work at each level k?

(c) What is the total work? *(Expected: $T(n) = \Theta(n \log^2 n)$)*

4 Implement Merge Sort (15 min)

4.1 Implement from scratch

Write a complete implementation of merge sort in Python:

```python
def merge(left: list, right: list) -> list:
    """Merge two sorted lists into a single sorted list."""
    # YOUR CODE HERE
    pass

def merge_sort(arr: list) -> list:
    """Sort a list using the merge sort algorithm."""
    # YOUR CODE HERE
    pass
```

Requirements:

- `merge_sort` must return a **new** sorted list (do not modify the input).
- `merge` must run in $O(n)$ where n is the combined length.
- Handle edge cases: empty list, single element.

4.2 Test your implementation

```python
assert merge_sort([]) == []
assert merge_sort([42]) == [42]
assert merge_sort([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]
assert merge_sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5]
assert merge_sort([38, 27, 43, 3, 9, 82, 10]) == [3, 9, 10, 27, 38, 43, 82]
assert merge_sort([3, 3, 1, 1, 2, 2]) == [1, 1, 2, 2, 3, 3]
```

4.3 Count comparisons

Modify your merge sort to also return the number of comparisons performed:

```python
def merge_sort_count(arr: list) -> tuple[list, int]:
    """Return (sorted_list, num_comparisons)."""
    pass
```

Run it on random arrays of size $n = 100,\ 1000,\ 10\,000$ and verify that the number of comparisons is approximately $n \log_2 n$.

```python
import random

for n in [100, 1000, 10000]:
    arr = [random.randint(0, 100000) for _ in range(n)]
    sorted_arr, comparisons = merge_sort_count(arr)
    theoretical = n * (n.bit_length() - 1)   # n * floor(log2(n))
    print(f"n={n:>6}: comparisons={comparisons:>8}, "
          f"n*floor(log2(n))={theoretical:>8}, "
          f"ratio={comparisons/theoretical:.2f}")
```

5 Divide-and-Conquer Challenges (15 min)

*Choose **5a** or **5b** (or both if you have time).*

5.1 Fast Exponentiation

Implement a fast power function using divide-and-conquer:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ \left(x^{n/2}\right)^2 & \text{if } n \text{ is even} \\ x \cdot x^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

```python
def fast_power(base: float, exp: int) -> float:
    """Compute base^exp using O(log exp) multiplications."""
    # YOUR CODE HERE
    pass
```

Questions:

1. What is the recurrence for the number of multiplications?
2. Apply the master theorem. What is the complexity?
3. Compare with the naïve approach (a simple loop multiplying `base` n times — $O(n)$ multiplications).
4. Test: compute 2^{100}, 3^{50}; verify against Python's built-in `**`.

5.2 Counting Inversions ⋆

An **inversion** in an array is a pair of indices (i, j) with $i < j$ and $a[i] > a[j]$.

Example: `[2, 4, 1, 3, 5]` has 3 inversions: $(2, 1)$, $(4, 1)$, $(4, 3)$.

Why it matters: Counting inversions measures how "far" a list is from being sorted. This is used in recommendation systems to compare two rankings (Kendall tau distance).

1. **Naïve approach:** Write `count_inversions_naive(arr)` in $O(n^2)$.
2. **D&C approach:** Modify merge sort to count inversions during the merge step.

   > **Hint**
   >
   > When merging, if an element from the right half is placed before elements remaining in the left half, each of those remaining left elements forms an inversion.

3. Verify: `[1,2,3,4,5]` → 0 inversions; `[5,4,3,2,1]` → 10 inversions.

4. Test on a random array of size 1000: compare the naïve count with the D&C count.

```python
def count_inversions(arr: list) -> tuple[list, int]:
    """Return (sorted_array, inversion_count) using modified merge sort."""
    # YOUR CODE HERE
    pass
```

6 Benchmarking — Theory vs Practice (10 min)

6.1 Setup

```python
import time, random
import matplotlib.pyplot as plt

def insertion_sort(arr):
    """Insertion sort (from S1)."""
    a = arr.copy()
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
    return a
```

6.2 Benchmark

For each n in `[100, 500, 1000, 2000, 5000, 10000]`:

1. Generate a random array of size n.
2. Time `insertion_sort(arr)` and `merge_sort(arr)` (use `time.perf_counter()`).
3. Store and print the results.

6.3 Plot

Plot the results using `matplotlib`: insertion sort in red, merge sort in blue. Label axes and add a legend.

Questions:

1. At what value of n does merge sort start clearly outperforming insertion sort?
2. For $n = 10\,000$, roughly what is the ratio of execution times? How does it compare to the theoretical ratio $\frac{n}{\log_2 n} \approx \frac{10\,000}{13} \approx 769$?
3. *(Bonus)* Also benchmark Python's built-in `sorted()` (Timsort). How does it compare?

Bonus Exercises

For students who finish early.

Bonus 1 — Binary Search Variants

Implement `binary_search(arr, target)`, `lower_bound(arr, target)`, and `upper_bound(arr, target)`. Use `lower_bound` and `upper_bound` to count occurrences of a value in a sorted array in $O(\log n)$.

Bonus 2 — Merge Sort in C++

Implement merge sort using `std::vector<int>`. Compare its performance with your Python implementation, `std::sort` (introsort), and `std::stable_sort` (merge sort variant).

Bonus 3 — Karatsuba Multiplication

Implement the Karatsuba algorithm for multiplying large integers: $x \cdot y = 10^n \cdot ac + 10^{n/2}((a+b)(c+d) - ac - bd) + bd$
Recurrence: $T(n) = 3T(n/2) + O(n) \Rightarrow \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Summary

$O(g(n))$	Upper bound: $f(n) \leqslant c \cdot g(n)$ for $n \geqslant n_0$
$\Omega(g(n))$	Lower bound: $f(n) \geqslant c \cdot g(n)$ for $n \geqslant n_0$
$\Theta(g(n))$	Tight bound: both O and Ω
Divide-and-conquer	Divide $\rightarrow$ Conquer (recurse) $\rightarrow$ Combine
Merge sort	$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
Binary search	$T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$
Master theorem	$T(n) = aT(n/b) + f(n)$: compare $f(n)$ vs $n^{\log_b a}$

Next week: Advanced OOP — Inheritance, Polymorphism, Design Patterns. Review your S1 notes on classes!