2025

ICP LAB OBSERVATION

PRIMALITY RANGE OF , PRIME FACTORS NAIVE APPROACH

RAJARSHI GHOSH

PRIME NUMBER: BRUTEFORCE METHOD

```
def is_prime_bruteforce(n: int) -> bool:
    if n <= 1:
        return False
    for d in range(2, n):
        if n % d == 0:
            return False
    return True
```

√n (root-n) optimal single-check

```
import math
def is_prime_sqrt(n: int) -> bool:
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    r = int(math.isqrt(n))
    d = 3
    while d <= r:
        if n % d == 0:
            return False
        d += 2  # check only odd divisors: 5, 7, 9,...
    return True
```

Python (√ n, with 6k ± 1)

```
import math


def is_prime_6k1(n: int) -> bool:
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False


    r = int(math.isqrt(n))  # only check up to √n
    d = 5
    while d <= r:
        if n % d == 0 or n % (d + 2) == 0:
            return False
        d += 6
    return True
```

Why It Works

- **Step 1:** Exclude numbers ≤ 3.
- **Step 2:** Exclude multiples of 2 and 3.
- **Step 3:** Only numbers of the form 6k ± 1 can be prime (for k ≥ 1).
- **Step 4:** Check divisibility from 5 to √n using d and d+2.

This reduces the number of checks compared to naive O(n) approach.

Dry Run (Example)

Input: n = 29

- n > 3, not divisible by 2 or 3.
- Check divisors: 5 → 5 and 7 → not divisible

- Next d = 11 → 11 > √29 → stop
- Return True → 29 is prime

Time and Space Complexity

- **Time:** O(√n) — only check divisors up to √n
- **Space:** O(1) — no extra storage

Sieve of Eratosthenes – Unoptimized

```
def sieve_unoptimized(n):
    # Returns a list isprime[0..n], where isprime[i] is True iff i is prime
    isprime = [True] * (n + 1)
    isprime[0] = isprime[1] = False

    if n < 2:
        return isprime

    for num in range(2, n + 1):  # check each number
        if isprime[num]:         # only process if num is prime
            p = num + num        # start marking multiples from 2*num
            while p <= n:
                isprime[p] = False
                p += num

    return isprime

# Example usage:
n = 30
isprime = sieve_unoptimized(n)
primes = [i for i in range(n + 1) if isprime[i]]
print("Primes up to", n, ":", primes)
```

Why It Works

- **Goal:** Identify all primes ≤ n.
- **Approach:**
    1. Initialize all numbers as prime (True) except 0 and 1.
    2. For each number num from 2 to n:
        - If it is still marked prime, mark all multiples 2*num, 3*num, ... as not prime (False).
    3. At the end, indices still marked True are prime numbers.
- **Difference from optimized sieve:**
    - Starts marking multiples from 2*num instead of num*num → extra work.
    - Outer loop goes all the way to n instead of √n.

Example Dry Run

Input: `n = 10`

num	Multiples marked as False
2	4,6,8,10
3	6,9
4	already False
5	10
6-10	already False or multiples handled

Output: `[2, 3, 5, 7]`

Time and Space Complexity

- **Time:** O(n^2) in worst case (because inner loop can run up to n times per prime).
- **Space:** O(n) — store isprime array.

Optimized Sieve – Python and Java Code

```
import math


def sieve_optimized(n):
    """
    Returns a list isprime[0..n], where isprime[i] is True iff i is prime
    """
    isprime = [True] * (n + 1)
    isprime[0] = isprime[1] = False
```

```
    if n < 2:
        return isprime

    rootN = math.isqrt(n)  # only need to check numbers up to √n
    for num in range(2, rootN + 1):
        if isprime[num]:  # process only primes
            p = num * num  # start marking multiples from num^2
            while p <= n:
                isprime[p] = False
                p += num

    return isprime



# Example usage
n = 30
isprime = sieve_optimized(n)
primes = [i for i in range(n + 1) if isprime[i]]
print("Primes up to", n, ":", primes)
```

Why It Works

1. **Start from num*num:** Smaller multiples of num were already marked by smaller primes.
2. **Check only up to √n:** Any composite number > √n has a factor ≤ √n.
3. **Mark multiples as non-prime:** Remaining True indices are primes.

Dry Run Example

Input: n = 10

num	Multiples marked
2	4,6,8,10

num Multiples marked

3 9

4 skipped

Output primes: [2,3,5,7]

Complexity

- **Time:** O(n log log n) — highly efficient
- **Space:** O(n) — store boolean array

Prime Factorisation

```
def primefactorization_N(n):
    original_n = n  # Keep the original number for reference
    for f in range(2, n + 1):
        cf = 0  # count factors or exponents
        while n % f == 0:
            cf += 1
            n //= f  # integer division
        if cf > 0:
            print(f"{f}^{cf}", end=", ")
    print()  # newline after printing all factors


if __name__ == "__main__":
    primefactorization_N(100)  # 2^2, 5^2
    primefactorization_N(97)   # 97^1
    primefactorization_N(360)  # 2^3, 3^2, 5^1
```

BRUTE FORCE:

```
import math


def primeFactors_SqrtN(n):
    for f in range(2, math.isqrt(n) + 1):
        cf = 0  # count factors or exponents
        while n % f == 0:
            cf += 1
            n //= f
        if cf > 0:
            print(f"{f}^{cf}", end=", ")
    # If remaining n > 1, it's a prime factor larger than sqrt(n)
    if n > 1:
        print(f"{n}^1", end=", ")
    print()  # newline after printing all factors

if __name__ == "__main__":
    primeFactors_SqrtN(100)  # prints 2^2, 5^2
    primeFactors_SqrtN(55)   # prints 5^1, 11^1
    primeFactors_SqrtN(360)  # prints 2^3, 3^2, 5^1
```

Unit3:

70. Climbing Stairs – Dynamic Programming (Optimized Approach)

🧩 Problem Statement

You are climbing a staircase. It takes n steps to reach the top.
Each time you can climb either 1 step or 2 steps.
Find the number of distinct ways to reach the top.

✅ Python Code (Optimized O(1) Space)

```
class Solution:
    def climbStairs(self, n: int) -> int:
        one, two = 1, 1  # Base cases for 1 step and 0 step

        for _ in range(n - 1):
            temp = one
            one = one + two
            two = temp

        return one
```

🧠 Explanation

- To reach step n, you can either come from:
  - step n - 1 (1 step jump), or
  - step n - 2 (2 step jump)
- Hence,

$$f(n) = f(n-1) + f(n-2)$$

- This follows the Fibonacci pattern.

🔍 Example Dry Run (n = 5)

Iteration	one	two	Explanation
Start	1	1	Base values

1	2	1	one = 1 + 1
2	3	2	one = 2 + 1
3	5	3	one = 3 + 2
4	8	5	one = 5 + 3

✅ Output: 8 ways

⏱️ Time and Space Complexity

- Time Complexity: O(n)
- Space Complexity: O(1)

🧩 Alternate DP Approach

```
class Solution:
    def climbStairs(self, n: int) -> int:
        dp = [0] * (n + 1)
        dp[0], dp[1] = 1, 1
        for i in range(2, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]
        return dp[n]
```

✅ Final Answer: The number of distinct ways to climb n stairs = f(n) = f(n−1) + f(n−2)

263. Ugly Number — Prime Factor Division Method

🧩 Problem Statement

An ugly number is a positive integer whose prime factors are limited to 2, 3, and 5.
Given an integer n, determine whether it is an ugly number.

✅ Python Code

```
class Solution:
    def isUgly(self, n: int) -> bool:
        if n <= 0:
            return False
```

```
    for p in (2, 3, 5):
        while n % p == 0:
            n //= p
    return n == 1
```

🧠 **Explanation**

- If n is non-positive, it cannot be ugly → return False.
- Otherwise, divide n by 2, 3, and 5 as long as it's divisible.
- After removing all these factors, if what's left is 1, then n has no other prime factors and is therefore ugly.

🔍 **Example Dry Runs**

Example 1:
n = 6
→ divide by 2 → n = 3
→ divide by 3 → n = 1 ✅
Output: True

Example 2:
n = 14
→ divide by 2 → n = 7
→ cannot divide by 3 or 5 → remains 7 ❌
Output: False

Example 3:
n = 30
→ divide by 2 → 15
→ divide by 3 → 5
→ divide by 5 → 1 ✅
Output: True

⏱️ **Complexity Analysis**

- Time Complexity: O(log n) — each division reduces n quickly
- Space Complexity: O(1)

✅ **Key Idea**

Keep stripping off the prime factors 2, 3, and 5 until the number either becomes 1 (ugly) or contains another factor (not ugly).

14. Longest Common Prefix — Prefix Shrinking Method

🧩 Problem Statement

Given an array of strings strs, find the **longest common prefix** (LCP) shared among all strings.
If there is **no common prefix**, return an **empty string** "".

✅ Python Code

```
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if not strs:
            return ""
        prefix = strs[0]
        for s in strs[1:]:
            while not s.startswith(prefix):
                prefix = prefix[:-1]
                if not prefix:
                    return ""
        return prefix
```

🧠 Explanation

1. **Start** with the first string as the initial prefix.
2. **Iterate** through each string:
    - While the current string doesn't start with the current prefix, **shrink** the prefix by one character from the end.
3. If the prefix becomes empty, return "".
4. The remaining prefix at the end is the **longest common prefix**.

🔍 Example Dry Runs

Example 1:
strs = ["flower", "flow", "flight"]

- prefix = "flower"
- "flow" doesn't start with "flower" → shrink → "flow" ✅
- "flight" doesn't start with "flow" → shrink → "flo" → "fl" ✅
  Output: "fl"

Example 2:
strs = ["dog", "racecar", "car"]

- prefix = "dog"
- "racecar" doesn't start with "dog" → shrink to "" ❌
  Output: ""

⏱️ **Complexity Analysis**

- **Time Complexity:** O(S) — total number of characters across all strings
- **Space Complexity:** O(1)

✅ **Key Idea**

Continuously shrink the prefix until all strings share it as their starting substring.

Common Elements Between Two Arrays — Set Intersection Method

🧩 **Problem Statement**

Given two integer arrays nums1 and nums2, return a list containing two values:

1. The number of elements in nums1 that also appear in nums2.
2. The number of elements in nums2 that also appear in nums1.

✅ **Python Code**

```
class Solution:
    def findIntersectionValues(self, nums1: List[int], nums2: List[int]) -> List[int]:
        set1, set2 = set(nums1), set(nums2)
        answer1 = sum(1 for x in nums1 if x in set2)
        answer2 = sum(1 for x in nums2 if x in set1)
        return [answer1, answer2]
```

🧠 Explanation

1. Convert both lists to **sets** for O(1) membership lookup.
    - set1 = unique elements of nums1
    - set2 = unique elements of nums2
2. For each element in nums1, count if it's also in set2.
3. For each element in nums2, count if it's also in set1.
4. Return the counts as a list [count1, count2].

🔍 Example Dry Run

Input:
nums1 = [1, 2, 3]
nums2 = [2, 3, 4]

Process:

- set1 = {1, 2, 3}
- set2 = {2, 3, 4}
- From nums1, common = [2, 3] → count = 2
- From nums2, common = [2, 3] → count = 2

Output:
[2, 2]

⏱️ Complexity Analysis

- **Time Complexity:** O(n + m) — linear scan of both arrays
- **Space Complexity:** O(n + m) — storage for sets

✅ Key Idea

Using **sets** drastically reduces lookup time, allowing efficient detection of overlapping elements between two arrays.

Unit4:

● Valid Palindrome

125. Valid Palindrome — Two-Pointer String Cleaning Approach

🧩 Problem Statement

Given a string s, determine if it reads the same backward and forward **after**:

- Converting all uppercase letters to lowercase.
- Removing all non-alphanumeric characters (anything except letters and digits).

Return True if the cleaned string is a palindrome, otherwise False.

✅ Python Solution

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        l, r = 0, len(s) - 1

        while l < r:
            while l < r and not s[l].isalnum():
                l += 1
            while l < r and not s[r].isalnum():
                r -= 1

            if s[l].lower() != s[r].lower():
                return False
            l += 1
            r -= 1

        return True
```

🧠 Explanation

1. **Two-pointer approach:**

- One pointer l starts from the left, and another r starts from the right.

2. **Skip non-alphanumeric characters:**
   - Move l forward until it points to an alphanumeric character.
   - Move r backward until it points to an alphanumeric character.
3. **Compare characters:**
   - Convert both characters to lowercase and compare.
   - If any mismatch occurs, return False.
4. **If loop completes, return True.**

🔍 Example Dry Run

Example 1:
s = "A man, a plan, a canal: Panama"

- Cleaned string = "amanaplanacanalpanama"
- Reads same forward and backward → ✅ **True**

Example 2:
s = "race a car"

- Cleaned string = "raceacar"
- Not same forward/backward → ❌ **False**

Example 3:
s = " "

- Cleaned string = "" (empty string)
- An empty string is considered palindrome → ✅ **True**

⏱️ Complexity Analysis

- **Time Complexity:** O(n) — single pass through the string.
- **Space Complexity:** O(1) — no extra data structures used.

💡 Key Idea

Use two pointers and isalnum() to efficiently check for palindrome without creating extra strings.

⚡ 643. Maximum Average Subarray I (Sliding Window – O(n))

```
class Solution:
    def findMaxAverage(self, nums: List[int], k: int) -> float:
        n = len(nums)
        cur_sum = sum(nums[:k])
        max_sum = cur_sum

        for i in range(k, n):
            cur_sum += nums[i] - nums[i - k]
            if cur_sum > max_sum:
                max_sum = cur_sum

        return max_sum / k
```

✅ Explanation

- Uses **sliding window** to maintain a running sum of length k.
- Updates the window sum in **O(1)** each iteration.
- Keeps track of the maximum sum seen.
- Divides only once at the end → faster than recomputing average each step.

🧩 Example

```
nums = [1, 12, -5, -6, 50, 3]
k = 4
print(Solution().findMaxAverage(nums, k))  # Output: 12.75
```

Best Time to Buy and Sell Stock

121. Best Time to Buy and Sell Stock (O(n), 0 ms Style)

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price, max_profit = float('inf'), 0
        for p in prices:
            if p < min_price:
                min_price = p
            elif p - min_price > max_profit:
                max_profit = p - min_price
        return max_profit
```

✅ **Why It Works**

- Tracks the **minimum price so far** (best day to buy).
- For each day, checks potential profit (p - min_price).
- Updates max_profit only if a better sell opportunity is found.
- Single pass → **O(n)** time, **O(1)** space.

🧩 **Example**

```
prices = [7, 1, 5, 3, 6, 4]
print(Solution().maxProfit(prices))  # Output: 5
```

Explanation:
Buy at **1**, sell at **6**, profit = **5**.

Unit5:

● Binary Search / 704

704. Binary Search (Iterative, O(log n))

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            m = (l + r) // 2
            if nums[m] == target:
                return m
            if nums[m] < target:
                l = m + 1
            else:
                r = m - 1
        return -1
```

✅ **Why It Works**

- Binary search repeatedly halves the search space.
- Checks mid element and narrows down to left or right half.
- Guaranteed **O(log n)** time and **O(1)** space.

🧩 **Example**

```
nums = [-1, 0, 3, 5, 9, 12]
target = 9
print(Solution().search(nums, target))  # Output: 4
```

ChatGPT said:

Here's the clean, **O(log n)** binary search solution for
153. Find Minimum in Rotated Sorted Array — minimal and optimal:

⚡ 153. Find Minimum in Rotated Sorted Array

```
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1
        while l < r:
            m = (l + r) // 2
            if nums[m] > nums[r]:
                l = m + 1
            else:
                r = m
        return nums[l]
```

✅ Logic

- If nums[m] > nums[r], the minimum is **to the right** (since rotation happened past m).
- Else, the minimum is **at m or to the left**.
- Continue narrowing until l == r → position of the smallest element.

🧩 Example

```
nums = [4,5,6,7,0,1,2]
print(Solution().findMin(nums))  # Output: 0
```

Time Complexity: O(log n)
Space Complexity: O(1)

33. Search in Rotated Sorted Array

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            m = (l + r) // 2
            if nums[m] == target:
                return m
            if nums[l] <= nums[m]:  # left half sorted
                if nums[l] <= target < nums[m]:
                    r = m - 1
                else:
                    l = m + 1
            else:  # right half sorted
                if nums[m] < target <= nums[r]:
                    l = m + 1
                else:
                    r = m - 1
        return -1
```

✅ **Logic Summary**

- Use modified binary search.
- Determine which half ([l, m] or [m, r]) is **sorted**.
- If target lies inside that sorted half → narrow into it.
  Else → search the other half.
- Continue until l > r.

🧩 **Example**

```
nums = [4,5,6,7,0,1,2]
print(Solution().search(nums, 0))  # Output: 4
```

Time Complexity: O(log n)
Space Complexity: O(1)