Lab 2 - Solution

Problem 1

Look at the following function:

```
def countdown(n):
    if n == 0:
        return
    print(n)
    countdown(n - 1)
```

If we call countdown(3), what will be printed? Explain step by step how the function works.

Answer

Step-by-step explanation when calling countdown(3):

1. **countdown(3)**
    - Prints: 3
    - Calls countdown(2)
2. **countdown(2)**
    - Prints: 2
    - Calls countdown(1)
3. **countdown(1)**
    - Prints: 1
    - Calls countdown(0)
4. **countdown(0)**
    - Hits base case (n == 0), stops recursion, returns to previous call.

Final output printed:

```
3
2
1
```

Problem 2

What will be the output of calling sum_numbers(3) in the following function?

```
def sum_numbers(n):
    if n == 0:
        return 0
    return n + sum_numbers(n - 1)
```

Show the calculation step by step.

Answer

Step-by-step calculation for sum_numbers(3):

1. **sum_numbers(3)** = 3 + sum_numbers(2)
2. **sum_numbers(2)** = 2 + sum_numbers(1)
3. **sum_numbers(1)** = 1 + sum_numbers(0)
4. **sum_numbers(0)** = 0 (base case)

Now, substituting back:

- sum_numbers(1) = 1 + 0 = 1
- sum_numbers(2) = 2 + 1 = 3
- sum_numbers(3) = 3 + 3 = 6

Final output: 6

Problem 3

What is the Big-O complexity of the following function?

```
def find_max(arr):
    max_value = arr[0]  # Line 1
    for num in arr:  # Line 2
        if num > max_value:  # Line 3
            max_value = num  # Line 4
    return max_value  # Line 5
```

Write T(n) (the exact count of operations) and then simplify it to find O(n).

Answer

Step-by-step complexity analysis:

- **Line 1:** executes **once** → 1 operation.
- **Lines 2-4:** loop through the entire array of size n:
    - The loop itself runs n + 1 times (including the final check that exits the loop).
    - Checking condition (**Line 3**): runs n times.
    - Assignment (**Line 4**): in worst case, runs n times.
- **Line 5:** executes **once** → 1 operation.

Exact operation count T(n):

$T(n)=1+(n+1)+n+n+1=3n+3$

Simplified Big-O complexity:

- Drop constants and lower-order terms: O(n)

Final Answer: O(n)

Problem 4

Both functions below calculate the sum of numbers from 1 to n. Analyze their time complexity and determine which one is faster.

Code 1: Using a Loop

```
def sum_loop(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total
```

Code 2: Using a Formula

```
def sum_formula(n):
    return (n * (n + 1)) // 2
```

1. Determine which function is faster.

Answer: The **formula-based function (sum_formula)** is faster

2. Explain your reasoning based on time complexity.

Answer

The **sum_loop function** runs in **linear time complexity O(n)**, meaning its execution time increases as n grows.
The **sum_formula function** operates in **constant time complexity O(1)**, meaning its execution time stays the same regardless of the size of n.

Thus, the formula function is more efficient, especially for larger inputs.

Problem 5

Compare the growth strategies of arrays and linked lists when storing large amounts of data. Which one is better for frequent insertions and deletions?

Answer

When storing large amounts of data:

- **Arrays**:

    - **Growth Strategy**: Must allocate large contiguous memory blocks as they grow.
    - **Insertions/Deletions**:
        - Fast at the end: O(1)
        - Slow in the middle or beginning: O(n), because shifting elements is required.
- **Linked Lists**:
    - **Growth Strategy**: Can allocate nodes individually, non-contiguously.
    - **Insertions/Deletions**:
        - Fast at any position (if position is known): O(1)
        - Finding positions can still take O(n).

Conclusion:

- For frequent insertions/deletions, **linked lists** are generally better because they avoid the costly shifting of elements required by arrays.

Problem 6

Consider the following queue implementation using a list:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        return "Queue is empty"

    def is_empty(self):
        return len(self.queue) == 0
```

What is the time complexity of the `dequeue()` operation?

Answer

- The operation pop(0) removes the first element in a Python list.
- Removing the first element requires shifting all subsequent elements one position forward.
- If the queue has n elements, this shifting takes **O(n)** time.

Conclusion:
The time complexity of the dequeue() operation is **O(n)**.

Problem 7

Consider the following stack implementation using an array:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return "Stack is empty"

    def is_empty(self):
        return len(self.stack) == 0
```

1. What happens when we call pop() on an empty stack?

Answer

When pop() is called:

- The method first checks if the stack is empty by calling self.is_empty().
- If the stack is indeed empty (len(self.stack) == 0), it returns the string "Stack is empty".

2. Modify the pop() method to return None instead of a string when the stack is empty.

Answer

```
    def pop(self):
        if not self.is_empty():
```

```
        return self.stack.pop()
    return None # return None if the stack is empty
```

Updated implementation: Returns Python's built-in value None, clearly and concisely signaling that no element is available without needing additional text.

Problem 8

Write a recursive function to compute the product of the first n natural numbers (n!), i.e., factorial of n. Then, analyze its time complexity.

Example Cases:

- factorial(5) → 120
- factorial(3) → 6
- factorial(0) → 1

Answer

```
def factorial(n):
    if n == 0:
        return 1 # Base case: factorial(0) = 1
    return n * factorial(n - 1)
```

Complexity Analysis:

- The recursion calls itself n times.
- Each call does a constant amount of work.
- Hence, the total complexity is proportional to the number of recursive calls.

Time complexity:

- The recursion depth is n.
- Thus, time complexity is **O(n)**.

Problem 9

In a hospital emergency room, patients are generally treated in the order they arrive. However, patients come in with varying degrees of severity. Patients with critical conditions (urgent) must be treated before those with less severe conditions (regular). You need to design a system that always processes patients with higher severity first. Which data structure should you use?

Answer

Analysis of the Situation:

In this scenario, we have two considerations for patient treatment:

- **Arrival Order**: Normally treated as FIFO (First-In-First-Out).
- **Severity of Condition**: Higher severity patients should be prioritized, even if they arrive later.

A standard queue (FIFO) will not effectively handle severity-based prioritization because it strictly follows arrival order without considering priority.

Suitable Data Structure: Priority Queue

To handle such scenarios, the best-suited data structure is a **Priority Queue**.

Priority Queue Explained:

- A priority queue is a data structure that orders elements based on their **priority** rather than just arrival time.
- Elements with higher priority (e.g., higher severity) are processed before elements with lower priority.

How a Priority Queue works in the hospital scenario:

- Each patient entering the ER is assigned a **priority number** based on severity.
    - Higher number → Higher severity → Higher priority.
- Patients are stored in the queue based on their priority:
    - When a patient arrives, they're inserted into the queue according to their priority level.
    - The next patient to be treated is always the one with the highest severity (highest priority).

Problem 10

The newest huge maths company, South Pacific Computation Corporation (SPPC), is here! SPPC has shares that they need to distribute to their employees. Assume that you are the CEO for SPPC, and you will receive your shares over days. On day one, you receive *n* shares. On day two, you receive *n/2* shares (rounded down). On day three, you receive *n/3* shares (rounded down). One day *i*, you receive *n/i* shares (rounded down). On the final day (day n), you receive *n/n = 1* share. For example, if *n = 3*, then you would receive 3+1+1 = 5 shares.

How many shares in total do you receive?

Input

The input consists of a single line containing one integer ***n (1 <= n <= 10^12)***, which is the number of days over which you receive your shares.

Output

Display the number of shares in total that you receive.

Sample Input	Output
1	1
3	5
4	8
5	10
10	27

Answer

Step-by-step logic:

Step 1: Understand the issue
Calculating shares day-by-day from 1 to n is inefficient (linear complexity O(n)), especially when n is very large (10^12).

Step 2: Efficient method using grouping
We observe that several consecutive days can yield the same number of shares. Thus, we group them to calculate faster.

Example: For n=10:

Day	**Shares (n//day)**
1	10 (10//1)
2	5 (10//2)
3	3 (10//3)
4	2 (10//4)
5	2 (10//5)
6	1 (10//6)
7	1 (10//7)

Day	Shares (n//day)
8	1 (10//8)
9	1 (10//9)
10	1 (10//10)

Notice: day 1 (1 day): 10 shares/day
day 2 (1 day): 5 shares/day
day 3 (1 day): 3 shares/day
days 4–5 (2 days): 2 shares/day
days 6–10 (5 days): 1 share/day

But when n is huge, calculating day-by-day would be inefficient.

Step 2: Group days receiving the same number of shares

- On day i, you get n//i shares.
- To efficiently calculate, we find a range of days that yield the same shares.
- The last day that you receive the same shares as day i is: j=n // (n//i)

Formula breakdown clearly explained:

- On day i, shares received = n//i.
- Last day receiving (n//i) shares: last_day = n // shares
- **Days in group**: days_in_group= last_day − i + 1
- **Total shares added in each group**: shares per day * days_in_group
- **Next day with fewer shares: last_day + 1**

Step 3: Python Implementation

```
def total_shares(n):
    total = 0
    i = 1
    while i <= n:
        shares = n // i
        last_day = n // shares
        days_in_group = last_day - i +1
        total += shares * days_in_group
```

```
        i = last_day + 1
    return total
```