📚 Day 9 Study Notes

1. Aptitude: Combinations

Goal: Understand selection logic where the order of selection does not matter.

Key Concepts

- **Definition:** A combination is a selection of items from a collection, such that the order of selection does not matter (unlike Permutations, where order matters).
- **Formula:**

$$nC_r = \frac{n!}{r!(n-r)!}$$

  - n = Total number of distinct items.
  - r = Number of items to be selected.
  - $!$ = Factorial (e.g., $5! = 5 \times 4 \times 3 \times 2 \times 1$).

Common Scenarios

1. **Selection of a Team:** Selecting 11 players from a squad of 15.
2. **Committee Formation:** Choosing 2 men and 2 women from a group.
3. **Drawing Cards/Balls:** Picking 3 red balls from a bag of 10.

Example Problem

Question: In how many ways can a committee of 3 people be chosen from a group of 10?
Solution:

$$10C_3 = \frac{10!}{3!(10-3)!} = \frac{10!}{3! \times 7!} = \frac{10 \times 9 \times 8}{3 \times 2 \times 1} = 120 \text{ ways}$$

2. Programming: Second Largest Element

Goal: Find the second largest number in an array efficiently.

Logic (Single Pass Approach - $O(N)$)

Using sorting takes $O(N \log N)$, but a linear scan is more efficient.

1. Initialize `largest` and `second_largest` to negative infinity (or the smallest possible integer).
2. Iterate through the array:
   - If the current element is **greater than** `largest` :

        - Update `second_largest` to be the current `largest` .
        - Update `largest` to be the current element.
    - Else if the current element is **greater than** `second_largest` AND **not equal** to `largest` :
        - Update `second_largest` to be the current element.
3. **Edge Case:** If the array has fewer than 2 elements or all elements are identical, a second largest does not exist.

Code Snippet (Python)

```
def find_second_largest(arr):
    if len(arr) < 2:
        return None

    largest = float('-inf')
    second = float('-inf')

    for num in arr:
        if num > largest:
            second = largest
            largest = num
        elif num > second and num != largest:
            second = num

    return second if second != float('-inf') else None
```

3. Concept: Python Dictionaries

Goal: Understand hash-map based data structures for fast lookups.

Key Features

- **Structure:** Stores data in `key: value` pairs.
- **Unordered:** (Mostly) Prior to Python 3.7, order wasn't guaranteed. Now insertion order is preserved, but they are conceptually unordered mappings.
- **Keys:** Must be unique and immutable (strings, numbers, tuples).
- **Values:** Can be any data type and duplicated.
- **Performance:** Average time complexity for lookups, inserts, and deletes is $O(1)$.

Common Operations

Operation	Syntax	Description
Creation	`my_dict = {"name": "Alice", "age": 25}`	Creates a new dictionary.

Access	`my_dict["name"]` or `my_dict.get("name")`	Access value. `.get()` avoids errors if key is missing.
Update	`my_dict["age"] = 26`	Updates existing key or adds new one.
Deletion	`val = my_dict.pop("age")`	Removes key and returns value.
Keys	`my_dict.keys()`	Returns view of all keys.
Items	`my_dict.items()`	Returns view of (key, value) tuples.

4. C/C++ Concept: Structures (`struct`)

Goal: Grouping variables of different types under a single name.

Why use Structures?

Arrays can only hold data of the *same* type. Structures allow you to bundle mixed data types (e.g., an `int` ID, a `string` name, and a `float` salary) representing a single entity.

Syntax & Usage

Definition:

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

Declaration & Access:

```
int main() {
    struct Student s1; // Declaration

    // assigning values
    s1.id = 101;
    strcpy(s1.name, "John"); // String copy for C
    s1.marks = 85.5;

    // Accessing
    printf("ID: %d, Name: %s", s1.id, s1.name);
    return 0;
}
```

Note: In C++, `typedef` is not strictly required to use the struct name as a type, unlike in older C standards.

5. SQL: Logical Operators (AND / OR / NOT)

Goal: Filter data based on multiple conditions.

Operators

1. **AND:** Returns a record if **ALL** conditions separated by AND are TRUE.
2. **OR:** Returns a record if **ANY** of the conditions separated by OR is TRUE.
3. **NOT:** Displays a record if the condition(s) is NOT TRUE.

Operator Precedence

When combining these, SQL processes them in this order:

1. `NOT`
2. `AND`
3. `OR` *(Use parentheses `()` to enforce specific logic!)*

Example Queries

Table: Employees

ID	Name	Dept	Salary
1	A	HR	50000
2	B	IT	60000
3	C	IT	45000

1. AND: Find IT employees earning more than 50k.

```
SELECT * FROM Employees
WHERE Dept = 'IT' AND Salary > 50000;
-- Result: Row 2 (B)
```

2. OR: Find employees who are in HR *or* earn less than 48k.

```
SELECT * FROM Employees
WHERE Dept = 'HR' OR Salary < 48000;
-- Result: Row 1 (A - HR), Row 3 (C - Salary < 48k)
```

3. NOT: Find employees who are NOT in IT.

```
SELECT * FROM Employees
WHERE NOT Dept = 'IT';
```

```
          -- Result: Row 1 (A)
```