Exercise 7

1. Compare the two computing models: mainframe and cloud. Why did it change so dramatically over time?

Mainframe Computing:

- Centralized computing model
- High upfront costs: proprietary hardware, long provisioning times
- Fixed capacity: scaling requires physical upgrades
- Employed by large institutions (e.g., banks, government)
- Users accessed via terminals; tightly controlled environments

Cloud Computing:

- Decentralized and distributed over the Internet
- Pay-as-you-go: minimal upfront investment
- Elastic scalability: infrastructure adjusts to demand
- Services (e.g., compute, storage, DB) abstracted via APIs
- Encourages agility, experimentation, and rapid deployment

Why the Shift Happened:

- **Economic Efficiency:** Cloud removes infrastructure CAPEX and shifts to OPEX
- **Elasticity & Scalability:** Businesses can dynamically scale without overprovisioning
- **DevOps & Automation:** Cloud APIs enable automated provisioning, deployment, and scaling
- **Global Accessibility:** Cloud resources can be accessed globally with low latency
- **Innovation & Competition:** Cloud providers rapidly innovate with new services, making legacy systems less competitive

2. Advantages of the cloud architectures from Varia's paper compared to Exercise 6 implementation

- Elastic scaling via EC2 and Hadoop
- Loose coupling using Amazon SQS queues
- High resilience with status tracking in SimpleDB and reprocessing logic
- Parallel processing using Hadoop for performance
- Ephemeral infrastructure (zero idle resources)
- Automated deployment and cleanup phases

Varia’s architecture offers superior **scalability, cost-efficiency, and resilience**, making it suitable for large-scale and on-demand workloads.

- Benefits of cloud architectures

4.

Problem Definition:
You are given:
A set of n items, each with:
a weight w[i], a value v[i], a knapsack with maximum weight capacity W
Goal: Select a subset of the items such that:
The total weight is ≤ W
The total value is maximized

Types of Knapsack Problems:
0/1 Knapsack:
Each item can be either included once or not at all
Example: You can't cut a gold bar in half
Fractional Knapsack (Greedy solution works):
You can take fractions of items
Example: Taking part of a liquid or grain
Multiple Knapsack or Bounded Knapsack:
Multiple containers or limited quantity of items

Why It’s Difficult (Computationally):
For n items, there are 2^n possible combinations.
Brute-force approach checks all subsets — exponential time
Dynamic Programming can solve it in O(nW) time, but this is pseudo-polynomial, meaning it's efficient only for small W

Java Code:

```
class GFG {
    static int knapSack(int W, int wt[], int val[], int n)
    {
        // Making and initializing dp array
        int[] dp = new int[W + 1];

        for (int i = 1; i < n + 1; i++) {
            for (int w = W; w >= 0; w--) {

                if (wt[i - 1] <= w)

                    // Finding the maximum value
                    dp[w]
                        = Math.max(dp[w], dp[w - wt[i - 1]]
                                              + val[i - 1]);
            }
        }
```

```
        // Returning the maximum value of knapsack
        return dp[W];
    }

    // Driver code
    public static void main(String[] args)
    {
        int profit[] = { 60, 100, 120 };
        int weight[] = { 10, 20, 30 };
        int W = 50;
        int n = profit.length;
        System.out.print(knapSack(W, weight, profit, n));
    }
}
```

AI solution:

```
public class Knapsack {

  // Function to solve 0/1 Knapsack problem using DP
  public static int knapsack(int[] weights, int[] values, int capacity) {
    int n = weights.length;
    int[][] dp = new int[n + 1][capacity + 1];

    // Build table dp[][] in bottom up manner
    for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
        if (i == 0 || w == 0) {
          dp[i][w] = 0;
        } else if (weights[i - 1] <= w) {
          dp[i][w] = Math.max(
            values[i - 1] + dp[i - 1][w - weights[i - 1]],
            dp[i - 1][w]
          );
        } else {
          dp[i][w] = dp[i - 1][w];
        }
      }
    }

    return dp[n][capacity]; // Maximum value
  }

  public static void main(String[] args) {
    int[] weights = {2, 3, 4, 5};
    int[] values = {3, 4, 5, 6};
    int capacity = 5;

    int maxValue = knapsack(weights, values, capacity);
    System.out.println("Maximum value in knapsack: " + maxValue);
  }
}
```

- This **greedy approach** takes items in input order without considering value-to-weight ratio.
- It **doesn’t explore all combinations** nor ensure maximum value.
- It **misses optimal solutions** in many standard test cases.