

# Vaccine Allocation



VISAGAN RAO J VENKATA RAMANA	217138
RAVINRAJ MATHIALAHAN	216807
MAYNON SELVAM	215548
XIAO FENG	213836

# Introduction



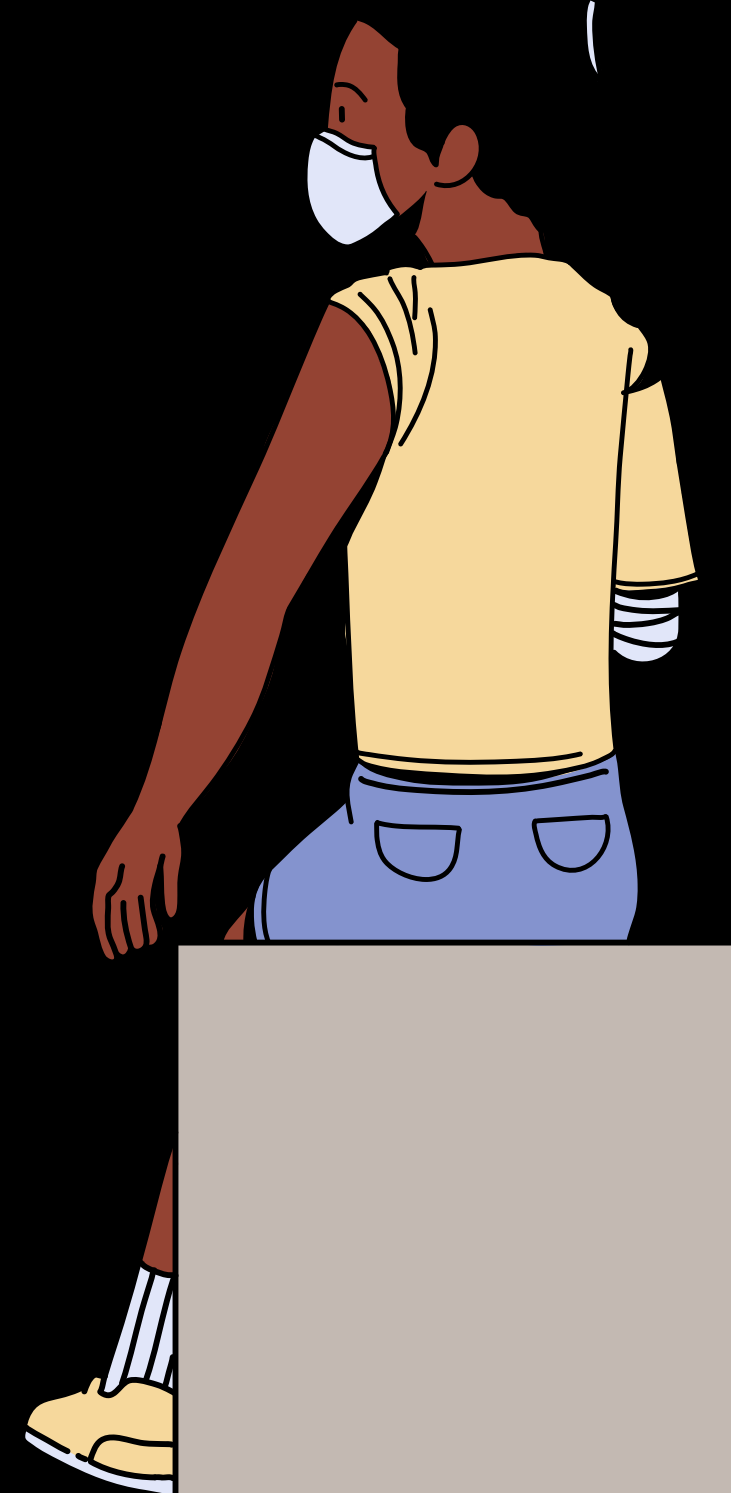
- **Context:** Vaccine distribution during pandemics like COVID-19.
- **Problem:** Limited vaccine supply, high-risk regions—manual decisions are inefficient.
- **Goal:** Use computational optimization to maximize lives saved.
- **Approach:** Use Dynamic Programming (DP) to allocate vaccines optimally under constraints.

# Scenario

Total vaccines: 100,000 doses.

Multiple cities with:

- **Different populations**
- **Varying infection rates**
- **Varying vaccine efficacies**
- **Fixed dose requirements (no partial allocation)**

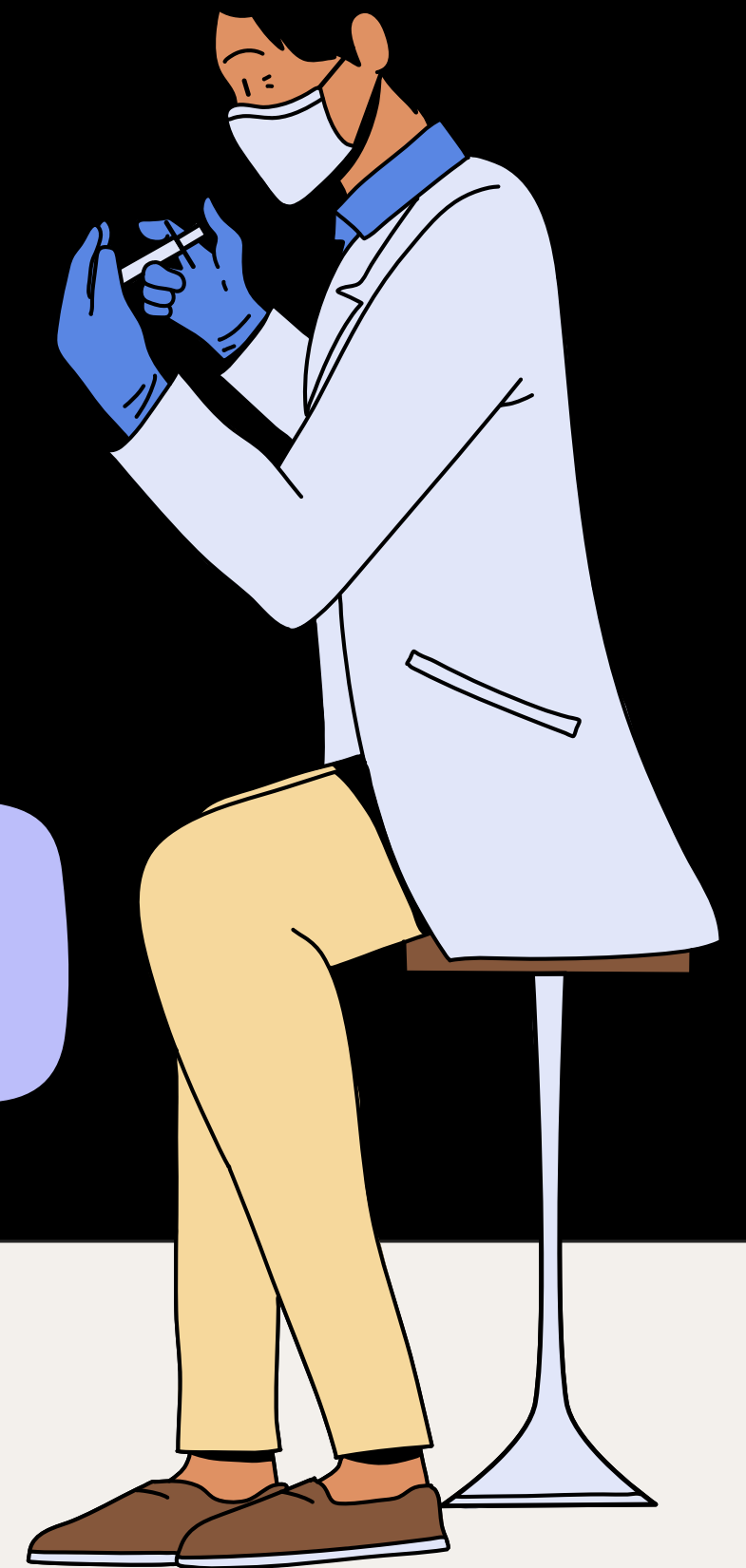


# Objective

- Maximize total lives saved
- Constraints:  
No partial vaccination per city.  
Doses used  $\leq 100,000$ .

Lives Saved Formula:

$$\text{LivesSaved}_i = (\text{Population} \times \text{InfectionRate} \times \text{Efficacy}) / 10,000$$



# Algorithm Comparison

Paradigm	Strengths	Weaknesses	Suitability
Sorting	Simple, rank by efficiency	Ignores constraints	Not suitable
Divide & Conquer	Solves subproblems independently	Can't enforce dose limits globally	Not suitable
Greedy	Fast, easy to implement	May skip better combinations	Suboptimal
Graph (Max Flow)	Good for flow constraints	Overcomplicated for this problem	Overkill
Dynamic Programming	Handles constraints, optimal	Higher memory/time cost	Best choice



# Why Choose Dynamic Programming ?

- Models the problem as 0/1 Knapsack:
- Cities = Items, Doses = Weights, Lives Saved = Values
- Explores all feasible combinations.
- Handles strict constraints (e.g. no partial city doses).
- Guarantees global optimality.
- Efficient due to memoization (no redundant calculations).

# Algorithm Design & Recurrence

DP Table:

$dp[d][c]$  = max lives saved using  $d$  doses and first  $c$  cities

Recurrence Relation:

$$dp[d][c] = \max(\begin{array}{l} dp[d][c - 1], \quad // \text{Skip city} \\ dp[d - \text{doses}[c]][c - 1] + \text{livesSaved}[c] // \text{Include city} \end{array})$$

Base Cases:

- $dp[0][c] = 0 \rightarrow 0 \text{ doses} = 0 \text{ lives saved}$
- $dp[d][0] = 0 \rightarrow 0 \text{ cities} = 0 \text{ lives saved}$

# Algorithm Correctness

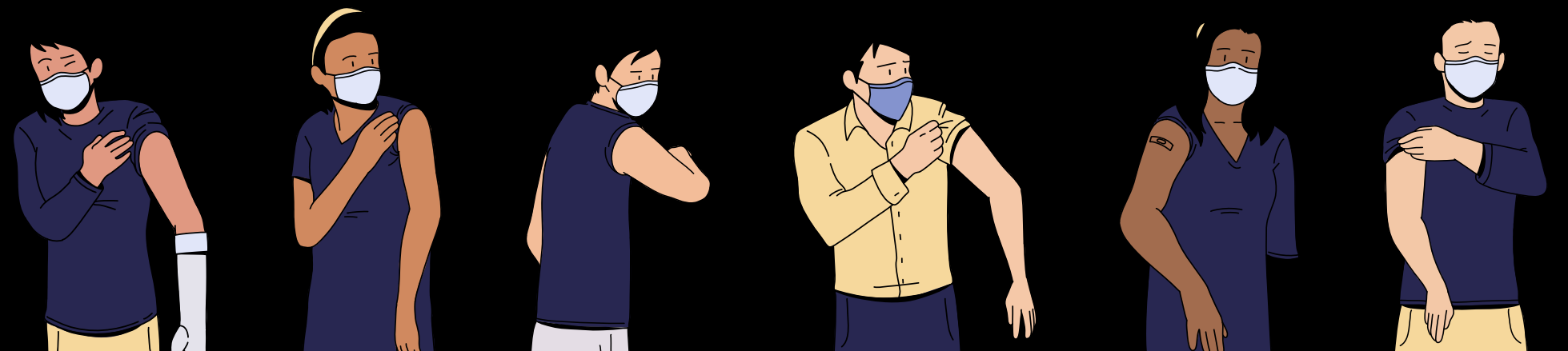
✓ **Base Cases:** Initialized correctly to reflect 0 doses or 0 cities.

✓ **Subproblem Overlap:** Solved efficiently using memoization.

✓ **Optimality:** Explores all feasible city sets for max lives saved.

✓ **Valid Recurrence:** Considers both including and excluding each city.

✓ **Backtracking:** `selected[][]` array traces chosen cities.





# Performance Analysis

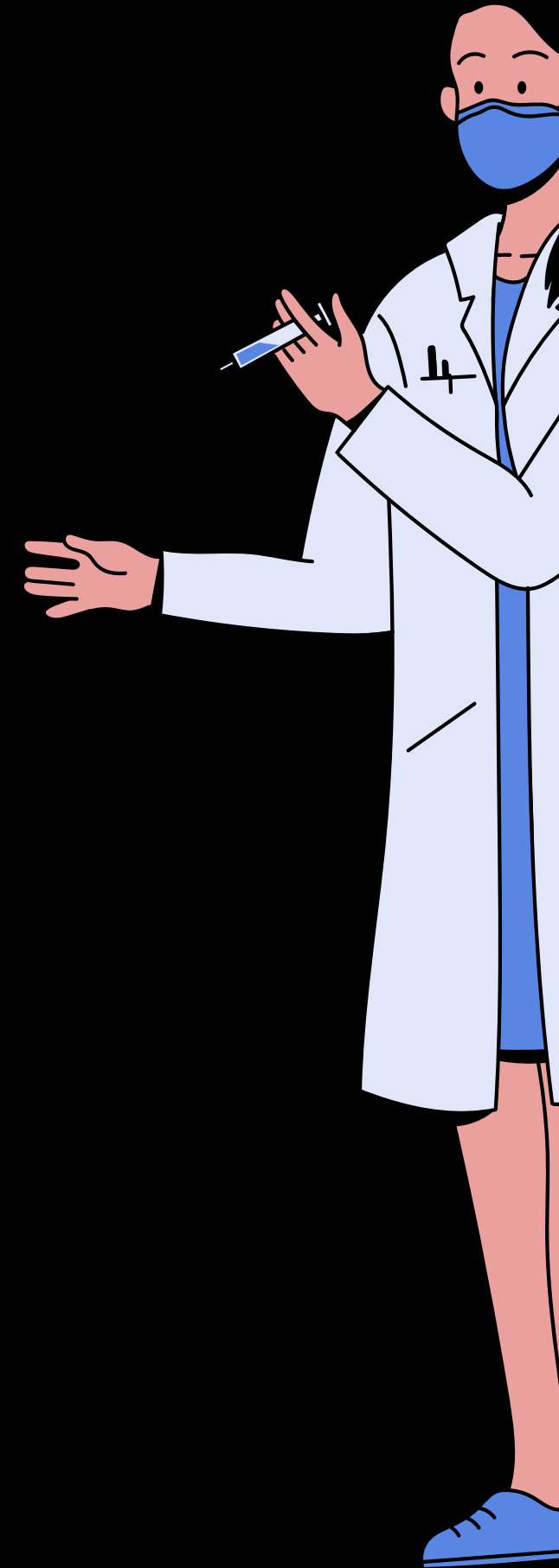
Time Complexity:

Let:

- $n$  = number of cities
- $d$  = total doses

Best / Avg / Worst =  $O(n \times d)$   
(All DP cells filled)

The algorithm has a **time complexity of  $O(n \times d)$** , where  $n$  is the number of cities and  $d$  is the total number of available vaccine doses. This is because every combination of city and dose must be evaluated in the dynamic programming table.



# Performance Analysis

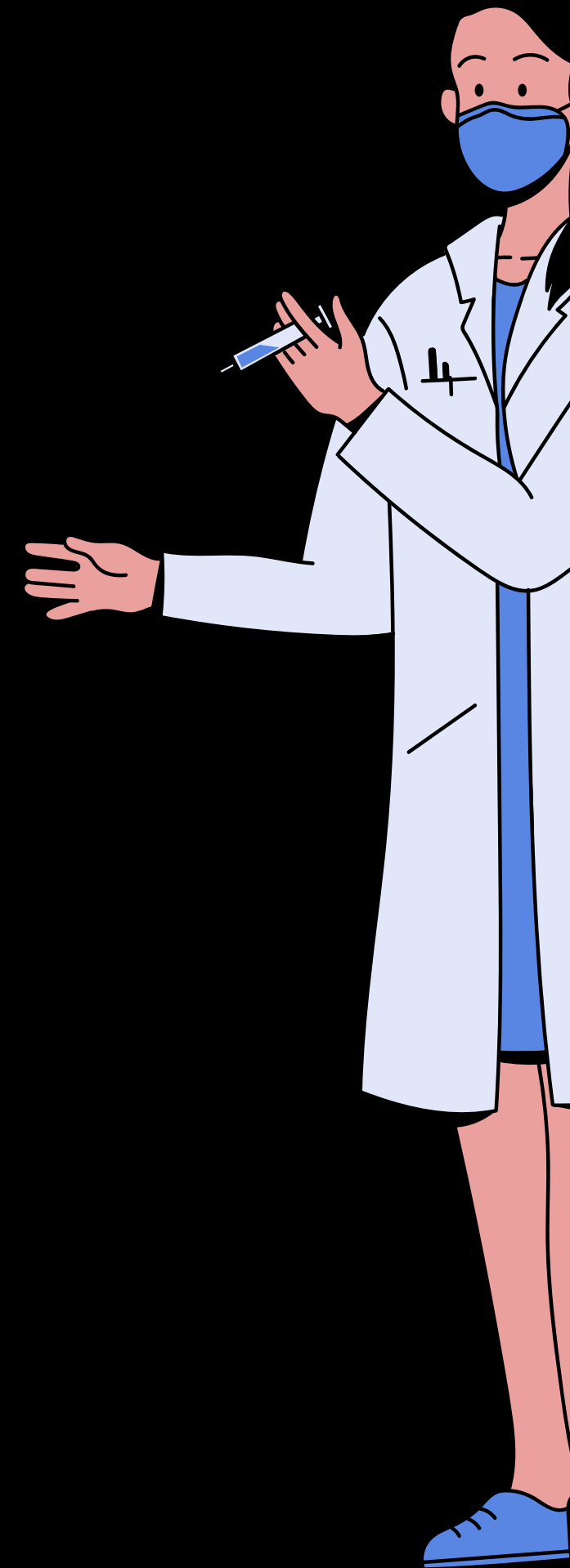
Let:

- $n$  = number of cities
- $d$  = total doses

Space Complexity:

- Full table:  $O(n \times d)$
- Can reduce to  $O(d)$  if backtracking not required

The space complexity is also  $O(n \times d)$  due to the 2D DP array, but it can be optimized to  $O(d)$  if only the maximum lives saved is needed without tracking selected cities. Despite this cost, the algorithm remains efficient and scalable for moderate input sizes



# CODE

```
public class VaccineAllocation {
    static class City {
        String name;
        int population, infectionRate, efficacy, doses;

        public City(String name, int population, int infectionRate, int efficacy, int doses) {
            this.name = name;
            this.population = population;
            this.infectionRate = infectionRate;
            this.efficacy = efficacy;
            this.doses = doses;
        }

        public int livesSaved() {
            return (population * infectionRate * efficacy) / 10000;
        }
    }

    public static int allocateVaccines(City[] cities, int totalDoses) {
        int n = cities.length;
        int[][] dp = new int[totalDoses + 1][n + 1];
        boolean[][] selected = new boolean[totalDoses + 1][n + 1];

        long startTime = System.currentTimeMillis();
```

```
        for (int d = 0; d <= totalDoses; d++) {
            for (int c = 1; c <= n; c++) {
                City city = cities[c - 1];
                if (city.doses <= d) {
                    int include = dp[d - city.doses][c - 1] + city.livesSaved();
                    int exclude = dp[d][c - 1];
                    if (include > exclude) {
                        dp[d][c] = include;
                        selected[d][c] = true;
                    } else {
                        dp[d][c] = exclude;
                    }
                } else {
                    dp[d][c] = dp[d][c - 1];
                }
            }
        }
    }
```

```
    int d = totalDoses, c = n, totalUsed = 0;
```

```
    System.out.println("Selected Cities:");
    while (c > 0) {
        if (selected[d][c]) {
            City city = cities[c - 1];
            System.out.println("- " + city.name + ": " + city.doses + " doses → " +
city.livesSaved() + " lives saved");
            d -= city.doses;
            totalUsed += city.doses;
        }
        c--;
    }
}
```

# CODE

```
        System.out.println("\nTotal vaccines used (DP): " + totalUsed);
        System.out.println("Max lives saved: " + maxLivesSaved);
        System.out.println("Execution time: " + (endTime - startTime) + " ms");

        return maxLivesSaved;
    }

    public static void main(String[] args) {
        City[] cities = {
            new City("City A", 1000000, 10, 90, 50000),
            new City("City B", 500000, 20, 70, 30000)
        };

        int totalDoses = 100000;
        allocateVaccines(cities, totalDoses);
    }
}
```

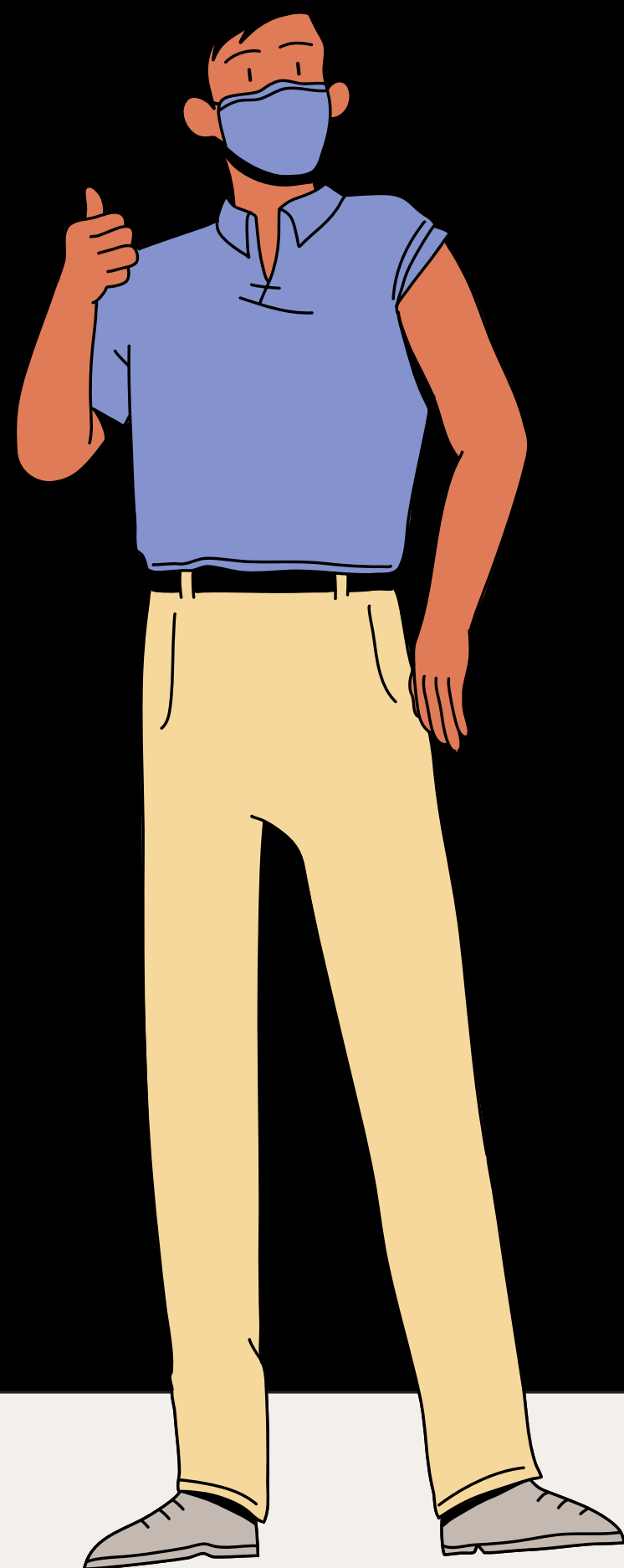
# OUTPUT

```
<terminated> VaccineAllocation [Java Application] C:\Users\visar\.p2\pool\
Selected Cities:
- City B: 30000 doses → 70000 lives saved
- City A: 50000 doses → 90000 lives saved

Total vaccines used (DP): 80000
Max lives saved: 160000
Execution time: 9 ms
```

# CONCLUSION

Dynamic Programming ensures an optimal, fair, and efficient approach to vaccine allocation by considering all possible combinations under strict constraints. Unlike Greedy or Divide and Conquer methods, DP guarantees the best outcome by exploring feasible solutions comprehensively. It is also flexible enough to be adapted for real-world enhancements, such as prioritizing high-mortality regions, incorporating financial or logistical limitations, and supporting partial vaccination strategies. This makes DP a powerful and reliable tool for solving complex public health resource distribution problems.



THANK YOU !!

