# Resource Allocation in Online Transportation Using Dynamic Programming

Vanessa Rebecca Wiyono - 13521151
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*13521151@std.stei.itb.ac.id*

***Abstract*— In the era of the 4.0 revolution, the online transportation industry is witnessing significant growth and expansion, driven by rapid technological advancements. To optimize operations, online transportation platforms are leveraging dynamic programming, a robust algorithmic approach, to enhance resource allocation. This research paper aims to analyze the application of dynamic programming in the online transportation industry, specifically focusing on its impact on improving resource allocation. By harnessing the potential of dynamic programming, online transportation services can efficiently allocate resources, resulting in cost reduction, improved efficiency, and enhanced user experiences.**

***Keywords*— Dynamic programming, online transportation, Resource Allocation***

## I. INTRODUCTION

The transformation of conventional motorcycle taxis into technology-based online motorcycle taxis has revolutionized the transportation landscape, offering customers the convenience of booking rides through mobile applications. This integration of technology has not only improved accessibility but also enhanced efficiency and reliability in the online transportation industry.

Online transportation services have emerged as a groundbreaking innovation in the world of m-commerce. These services, commonly referred to as ride-sharing, empower customers to effortlessly book rides through mobile applications, while drivers can efficiently respond to these requests via the same apps. Indonesia, in particular, hosts a range of online transportation platforms, including Uber (no longer exist), Grab, and Gojek as the top providers.

This service is perceived as a responsive, flexible, and user-friendly transportation alternative. As well as offering attractive travel options, it also has the potential to address environmental concerns and reduce dependence on private vehicles. However, it is important to recognize that these benefits may not be accessible to individuals of all income levels. Those who do not own a smartphone or face difficulties in using the service may be left behind, while traditional public transport may experience a decrease in market share. This situation raises questions about the role of government in implementing appropriate regulations and policies. The emergence of an online city shuttle service was enthusiastically welcomed by the Indonesian people, who have long been waiting for an affordable, easy-to-access and high-quality transportation solution.

According to the latest statements and data released by Grab and Gojek as of May 2023, the recruitment of drivers has become increasingly challenging due to the contraction of revenues in the *ojol* (motorcycle taxis) and online taxi sectors. In its financial report, Grab provides insights into the driver statistics on their platform. Previous research conducted by Muhammad Yorga Permana, a doctoral student at the London School of Economics (LSE), has unveiled the inclination among *ojol* drivers to transition to more stable employment positions. A significant factor driving this inclination is the continuous decline in their income, with a notable decrease observed in 2019. Additionally, the previously enticing daily bonuses offered by these platforms are now perceived as less attractive by the drivers.

The aforementioned challenges in driver recruitment and declining revenues in the *ojol* and online taxi sectors have prompted a critical need for a strategic reassessment of resource allocation within online transport platforms. Specifically, attention must be given to optimizing the allocation of drivers and customer orders in order to enhance driver income. This is a crucial aspect that necessitates careful consideration and implementation of effective measures to ensure a fair and equitable distribution of resources. In the subsequent paragraph, we will delve into the various strategies and approaches employed by Grab and Gojek to address this issue and bolster the income of their drivers.

The challenges mentioned above in driver recruitment and declining revenues in the *ojol* and online taxi sectors have driven the critical need for a strategic reassessment of resource allocation in online transportation platforms. In particular, attention should be paid to ensuring a fair and equitable distribution of resources by optimizing the allocation of drivers and customer orders so that the flow of driver assignments for each user is more efficient in terms of time. Therefore, this

paper will discuss further about the resource allocation of public transportation using dynamic programming algorithms.

## II. Basic Theory

### A. Dynamic Programming

Dynamic programming, introduced by Richard Bellman in the 1940s, is a method that combines mathematical optimization and computer programming. The term "dynamic programming" was coined to describe the process of identifying the optimal decision at each stage of a problem. This approach is particularly useful in addressing problems with evolving characteristics over time. The utilization of "programming" in dynamic programming draws parallels with linear programming and mathematical programming, emphasizing the pursuit of optimal solutions.

Dynamic programming operates by decomposing the original problem into a set of subproblems and solving each subproblem only once, storing the solutions in a table. This methodology is commonly employed in optimization problems that exhibit overlapping subproblems.

Observing the side of optimization problems, the presence of overlapping subproblems is observed when a recursive algorithm repeatedly solves the same subproblem. Dynamic programming offers a key advantage in such cases by solving these overlapping subproblems only once and storing their solutions in a table for future use.

This concept can be illustrated through the example of computing the Fibonacci sequence, where each Fibonacci number is computed recursively. When computing the nth Fibonacci number (Fn), it requires calculating Fn-1 and Fn-2 and adding them together. However, in computing Fn-1, we encounter the need to compute Fn-2 again, even though it has already been computed to find Fn. By computing and storing Fn-2 for Fn-1, we can avoid the repetition of the computation, thereby improving efficiency.

In the application, there are certain characteristics that can be solved by implementing dynamic programming. First of all, it is important that these problems can be divided into multiple stages, with each stage requiring a single decision to be made. Secondly, each stage comprises various states, representing the possible inputs or conditions specific to that stage. There are two methods commonly used to solve problems. The first is the top-down approach with memorization and the second one which is the bottom up approach.

Top down approach involves breaking down a complex problem into smaller subproblems and solving them recursively, starting from the top and working towards the base cases and is often combined with memoizaiton.

Memoization is a technique employed when a problem can be solved recursively, leveraging the solutions to its subproblems, especially when these subproblems overlap. This method involves storing the computed values in a table, following a top-down approach. Memoization does not necessitate the filling of all cells in the table unless it is explicitly needed. Each cell is filled on-demand, as required by the computation process. In other words, memoization does not mandate the population of all table cells to reach the final answer.

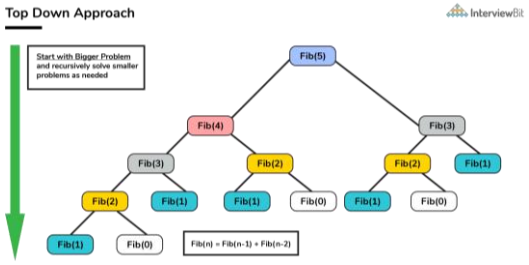



Figure 1. Top down approach (source: https://ibpublicimages.s3-us-west-2.amazonaws.com/tutorial/dp2.png )

On the other hand, the second method is the bottom-up approach with tabulation. This method differs from the top-down approach as it involves filling in a multidimensional table. In the bottom-up approach, a table is utilized to store the computed values representing the optimal solutions to the subproblems. Unlike memorization, tabulation requires the filling of all cells in the table. The solution to a given problem is formulated recursively by using the solutions to the corresponding subproblems. The subproblems are solved first, and their solutions are combined to obtain the solution to the original problem. This process is typically organized in a tabular format, where the computed solutions for the subproblems are stored and utilized from the table.

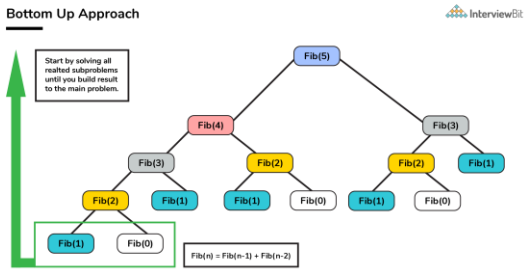



Figure 2. Bottom up approach (source: https://ibpublicimages.s3-us-west-2.amazonaws.com/tutorial/dp1.png )

In such cases, dynamic programming offers a more efficient alternative to naive approaches, requiring less time for computation. It proves especially beneficial when confronted with problems that would otherwise necessitate exponential time using a straightforward approach. Dynamic programming enables the resolution of these problems within polynomial time complexity.

### B. Resource Allocation

In the context of strategic management, resource allocation refers to efforts to minimize fluctuations in resource utilization across projects. This involves the careful management and distribution of resources to ensure their efficient and effective use. The goal is to maintain a balance and avoid over or under use of resources, which can lead to sub-optimal project performance.

In project management, resource allocation involves scheduling activities and determining the resources required for each activity while considering the availability of resources and the project timeline. Taking into account the availability of resources and the time frame of the project, it is expected that the utilization of resources can be optimized and the timely completion of tasks is ensured. Effective allocation of resources in both strategic management and project management plays an important role in achieving the desired results and project success.

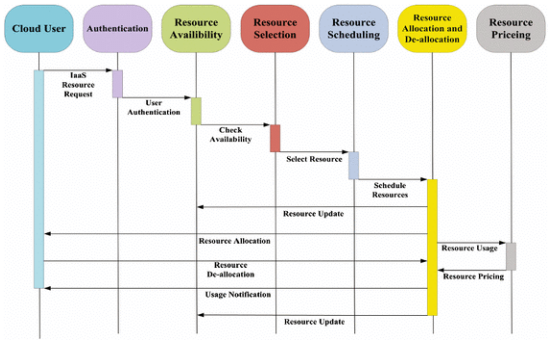



Figure 3. Process of resource allocation (source: https://www.researchgate.net/profile/Hamid_Madni/publication/311771195/figure/fig2/AS:779409946578965@1562837220380/Process-of-resource-allocation.gif)

With the increasing popularity of online transportation platforms, such as ride-hailing services, it becomes essential to allocate resources appropriately to ensure smooth operations and customer satisfaction. In this context, resources primarily refer to the available pool of drivers and the demand for transportation services.

Efficient resource allocation involves dynamically matching the supply of drivers with the fluctuating demand from passengers. This process requires sophisticated algorithms and real-time data analysis to optimize driver availability and minimize passenger waiting times.

By intelligently allocating drivers to areas with high demand and adjusting their distribution based on real-time traffic conditions and passenger requests, transportation platforms can enhance service quality and reduce customer dissatisfaction. Moreover, proper resource allocation enables a fair distribution of ride requests among drivers, promoting a balanced workload and maximizing their income potential.

However, challenges arise in resource allocation due to the inherent uncertainty and variability in passenger demand and driver availability. Balancing supply and demand can be a complex task, especially during peak hours or special events when demand surges. Transport companies and platforms must continually monitor and analyze data to make informed decisions regarding driver recruitment, incentives, and dispatching strategies. Additionally, factors such as driver preferences, driver retention, and driver satisfaction should be considered to maintain a stable and motivated driver pool.

## III. Dynamic Programming in Resource Allocation

To optimize the allocation of drivers and effectively address fluctuating passenger demand in online-based transportation platforms like Uber, Gojek, and Grab, dynamic programming can be utilized. The allocation of resources, particularly matching the available pool of drivers with incoming ride requests in real-time, is a crucial aspect of resource allocation. By leveraging historical and real-time data, dynamic programming enables these platforms to analyze demand patterns, identify peak hours, and allocate drivers efficiently. Factors such as driver proximity, availability, and efficiency are considered to optimize the dispatching process, minimizing passenger wait times and maximizing driver utilization.

Another important application of dynamic programming in resource allocation is addressing surge pricing during periods of high demand. By leveraging historical data on price surges and passenger willingness to pay, these platforms can dynamically adjust fare rates to balance supply and demand. This approach incentivizes more drivers to come online during peak hours, ensuring adequate availability of transportation services while enhancing the overall passenger experience.

Continuously monitoring and analyzing data allows these platforms to refine their resource allocation algorithms. Considerations such as driver ratings, preferences, and availability patterns are taken into account to ensure a fair distribution of rides among drivers. This approach not only contributes to a sustainable and motivated driver workforce but also maintains high levels of driver satisfaction.

Steps for applying dynamic programming in resource allocation for online-based transportation platforms:

1. Demand Prediction: By analyzing historical and real-time data, dynamic programming algorithms can predict demand patterns and identify peak hours, which will helps in allocating drivers more efficiently, ensuring sufficient coverage during periods of high demand.

2. Driver Dispatching: Dynamic programming algorithms consider factors such as driver proximity, availability, and efficiency to optimize the dispatching process. By matching drivers with ride requests in real-time, these algorithms minimize passenger wait times and maximize driver utilization.

3. Surge Pricing: Leveraging historical data on price surges and passenger willingness to pay, dynamic programming allows platforms to dynamically adjust fare rates during high-demand periods to balance the supply and demand, incentivizing more drivers to come online and ensuring adequate availability of transportation services.

4. Continuous Optimization: Continuous monitor and data analyze refine dynamic programming algorithms' resource allocation strategies with considered factors such as driver

ratings, preferences, and availability patterns to ensure a fair distribution of rides among drivers

5. Resource Management: Dynamic programming helps platforms make efficient use of resources like driver time, vehicle capacity, and operational costs. By optimizing resource allocation, platforms can achieve better operational efficiency and cost-effectiveness.

6. Scalability: Dynamic programming algorithms can handle large-scale resource allocation problems by breaking them down into smaller subproblems. This enables efficient computation and scalability as the platform expands and handles increasing volumes of ride requests.

## IV. IMPLEMENTATION

dynamic programming is used to find the most efficient driver for each passenger based on the time efficiency between their locations, with the breakdowns:

Basic calculation for distamce between a passenger and a driver using euclidean distance and fare calculation with surge multiplier that will be activated during rushour

```
def calculate_distance(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

def calculate_fare(base_fare, distance, surge_multiplier):
    return base_fare + (distance * surge_multiplier)
```

Dynamic Proogramming Approach:

```
def allocate_resources(passenger_location, driver_locations, rush_hour):
    n = len(passenger_location)  # Number of passengers
    m = len(driver_locations)  # Number of available drivers
    base_fare = 5.0  # Base fare for the trip

    time_efficiency_table = [[0] * m for _ in range(n)]

    # Calculate the time efficiency
    for i in range(n):
        for j in range(m):
            x1, y1 = passenger_location[i]
            x2, y2 = driver_locations[j]
            distance = calculate_distance(x1, y1, x2, y2)
            time_efficiency_table[i][j] = distance * 1000    # in ms

    # Dynamic programming
    chosen_drivers = []
    total_fares = []

hour
    if rush_hour:
        surge_multipliers = [1.5, 2.0, 1.8]  # Surge multipliers for rush hour
        fare_type = "Rush Hour"
    else:
        surge_multipliers = [1.0, 1.0, 1.0]  # Surge multipliers for non-rush hour
        fare_type = "Non-Rush Hour"

    for i in range(n):
        min_time_efficiency = float('inf')
        chosen_driver = None
        for j in range(m):
            time_efficiency = time_efficiency_table[i][j]
            if time_efficiency < min_time_efficiency:
                min_time_efficiency = time_efficiency
                chosen_driver = j
        chosen_drivers.append(chosen_driver)

        distance = calculate_distance(
            passenger_location[i][0], passenger_location[i][1],
            driver_locations[chosen_driver][0],
driver_locations[chosen_driver][1]
        )
        surge_multiplier = surge_multipliers[chosen_driver]

        fare = calculate_fare(base_fare, distance, surge_multiplier)
        total_fares.append(fare)

        print(f"Passenger {i+1}: Chosen Driver {chosen_driver+1} |
Total Fare: ${fare:.2f}")

    # Print the time efficiency table in milliseconds
    print("\n")
    print("Efficiency Table (ms):")
    for row in time_efficiency_table:
        print([f"{time:.2f}" for time in row])
    print()
```

The calculation begins by initializing some variables such as the number of passengers (n), the number of available drivers (m), and the base fare for the trip (base_fare). It also creates an empty table called time_efficiency_table to store the time efficiency values in milliseconds between each passenger and driver.

It continues to calculate the time efficiency between each passenger and driver by iterating over the passengers and drivers. For each combination, it retrieves the coordinates of the passenger and driver, calculates the Euclidean distance using the calculate_distance function, and stores the result in the time_efficiency_table after converting it to milliseconds.

Empty lists are initialized to store the chosen drivers (chosen_drivers) and the total fares for each passenger (total_fares).

The surge multipliers are determined based on whether it is rush hour or non-rush hour, and the corresponding fare type (fare_type) is set accordingly.

Minimum time efficiency will be searched for each passenger respectively. First, it initializes variables for the minimum time efficiency (min_time_efficiency) and the chosen driver (chosen_driver) followed by iterating over the drivers and compares the time efficiency from the time_efficiency_table for each combination. If a lower time efficiency is found, the minimum time efficiency and chosen driver are updated.

Once the most efficient driver is determined for a passenger, the code calculates the distance between the passenger and the chosen driver using the calculate_distance function. It retrieves the surge multiplier corresponding to the chosen driver and calculates the fare using the calculate_fare function with the base fare, distance, and surge multiplier. The fare is then added to the total_fares list.

For each passenger, the passenger number, chosen driver number, and the total fare will be print. At the end, time efficiency table, displaying time efficiency values between each passenger and driver in milliseconds will be printed.

Test examples:
Input:

```
passenger_location = [(0, 0), (2, 3), (5, 1)]  # Passenger locations (x, y coordinates)
driver_locations = [(1, 1), (3, 2), (4, 0)]  # Driver locations (x, y coordinates)

print("Rush Hour")
allocate_resources(passenger_location, driver_locations, rush_hour=True)

print("\nNon-Rush Hour")
allocate_resources(passenger_location, driver_locations, rush_hour=False)
```

This test aims to show the difference between case during rush hour and non-rush hour

Output example:

```
Rush Hour
Passenger 1: Chosen Driver 1 | Total Fare: $7.12
Passenger 2: Chosen Driver 2 | Total Fare: $7.83
Passenger 3: Chosen Driver 3 | Total Fare: $7.55

Efficiency Table (ms):
['1414.21', '3605.55', '4000.00']
['2236.07', '1414.21', '3605.55']
['4000.00', '2236.07', '1414.21'

Non-Rush Hour
Passenger 1: Chosen Driver 1 | Total Fare: $6.41
Passenger 2: Chosen Driver 2 | Total Fare: $6.41
Passenger 3: Chosen Driver 3 | Total Fare: $6.41

Efficiency Table (ms):
['1414.21', '3605.55', '4000.00']
['2236.07', '1414.21', '3605.55']

Non-Rush Hour
Passenger 1: Chosen Driver 1 | Total Fare: $6.41
Passenger 2: Chosen Driver 2 | Total Fare: $6.41
Passenger 3: Chosen Driver 3 | Total Fare: $6.41

Efficiency Table (ms):
['1414.21', '3605.55', '4000.00']
['2236.07', '1414.21', '3605.55']
['4000.00', '2236.07', '1414.21']
```

During the rush hour, the algorithm determines the most efficient driver for each passenger based on the calculated time efficiency values. The efficiency table shows the distances between passengers and drivers in milliseconds. The algorithm considers the surge multipliers specific to rush hour conditions (1.5, 2.0, and 1.8).

For the rush hour scenario, the algorithm assigns the passengers to the drivers with the lowest time efficiency values, resulting in the following allocations: Passenger 1 is assigned to Driver 1, Passenger 2 is assigned to Driver 2, and Passenger 3 is assigned to Driver 3. The calculated total fares for each passenger are $7.12, $7.83, and $7.55, respectively.

On the other hand, during non-rush hour, the algorithm performs the same calculations, but with surge multipliers set to 1.0 for all drivers. The efficiency table remains the same, representing the distances between passengers and drivers.

In the non-rush hour scenario, the algorithm assigns the passengers to the drivers with the lowest time efficiency values, resulting in the same driver allocations for all passengers: Driver 1 for Passenger 1, Driver 2 for Passenger 2, and Driver 3 for Passenger 3. The calculated total fares for all passengers are $6.41.

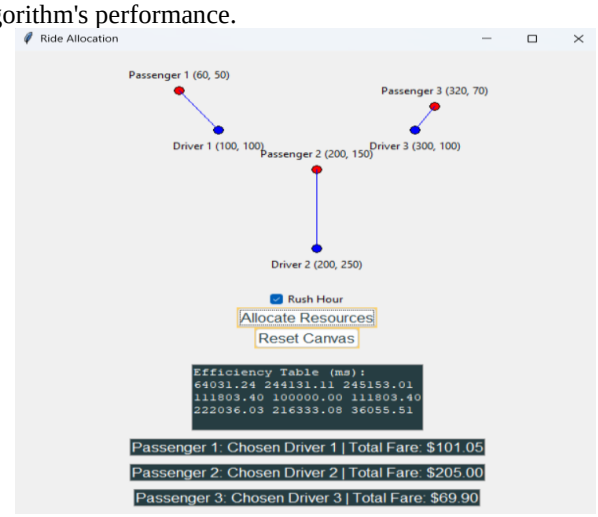
The efficiency table is computed based on the distances between each passenger and drive that are calculated using the calculate_distance function, employing the Euclidean distance formula. Since the efficiency table is derived solely from the distances, it remains unchanged regardless of whether it is a rush hour or non-rush hour.

From a dynamic programming perspective, the algorithm uses the concept of "optimal substructure" to solve the problem efficiently. It breaks down the problem of allocating passengers to drivers into smaller subproblems of finding the most efficient driver for each passenger individually. By iteratively calculating the time efficiency between passengers and drivers and updating the minimum time efficiency and chosen driver, the algorithm ensures that the chosen drivers collectively result in the most efficient allocation for all passengers.

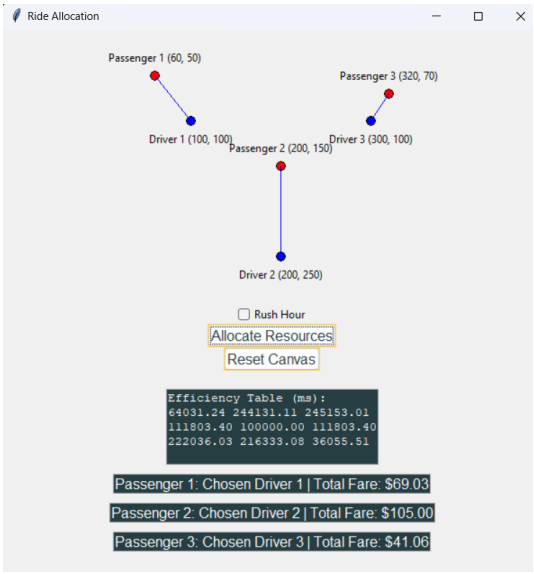
application of dynamic programming efficiently allocate passengers to drivers based on time efficiency, considering different surge multipliers for rush hour and non-rush hour scenarios. Therefore, fare efficiency can be maximized while ensuring fair distribution of passengers among available drivers.

A graphical user interface (GUI) was implemented using Tkinter to visualize and simulate different scenarios for the resource allocation problem. The GUI displayed passenger and driver locations on a graphical map and calculated the most efficient driver for each passenger based on distance and surge multipliers. The results were shown in the GUI, providing a clear representation of the assigned drivers and total fares. This

visualization enhanced understanding and evaluation of the algorithm's performance.




Rush hour




Non Rush hour

Comparison with brute force:

```
def allocate_resources_brute_force(passenger_location, driver_locations):
    n = len(passenger_location)
    m = len(driver_locations)

    min_total_time_efficiency = float('inf')
    best_allocation = None

    # all possible permutations of drivers
    driver_permutations = permutations(range(m))

    # Iterate through each permutation
    for permutation in driver_permutations:
        total_time_efficiency = 0
        for i in range(n):
            passenger_index = i
            driver_index = permutation[i]
            x1, y1 = passenger_location[passenger_index]
            x2, y2 = driver_locations[driver_index]
            distance = calculate_distance(x1, y1, x2, y2)
            time_efficiency = distance * 1000
            total_time_efficiency += time_efficiency

        if total_time_efficiency < min_total_time_efficiency:
            min_total_time_efficiency = total_time_efficiency
            best_allocation = list(permutation)

    return best_allocation
```

In contrast to the dynamic programming, brute force solution exhaustively generates all possible permutations of the drivers and iterates through each permutation to calculate the total time efficiency for resource allocation. By considering all combinations, it ensures finding the allocation with the minimum total time efficiency. However, this approach has a time complexity of O(n!), where n is the number of passengers or drivers.

On the other hand, the solution that utilizes dynamic programming efficiently select the best allocation based on the minimum time efficiency. This approach exhibits a more efficient time complexity of O(n * m), where n is the number of passengers and m is the number of drivers.

The significant difference in time complexity between the brute force solution and the original code highlights the advantage of dynamic programming in tackling resource allocation problems. By avoiding redundant calculations and reusing solutions to subproblems, the original code achieves a more efficient allocation process, particularly when dealing with larger datasets.

## V. ANALYSIS

Problem Description:
This problem revolves around efficiently allocating passengers to drivers to maximize fare efficiency. The allocation depends on factors such as distance, surge multipliers, and time efficiency between passenger and driver locations.

Solution Approach:
The implemented solution follows a dynamic programming approach to find the most efficient driver for each passenger. The algorithm iterates through the passenger and driver locations, calculating time efficiency values and selecting the driver with the minimum time efficiency for each passenger.

Rush Hour Scenario:
In the rush hour scenario, the surge multipliers are set to specific values (1.5, 2.0, and 1.8) to reflect increased demand and fares during peak hours. The algorithm correctly assigns passengers to drivers based on the calculated time efficiency values. The resulting fare efficiency is demonstrated by the total fares of $7.12, $7.83, and $7.55 for each passenger.

Non-Rush Hour Scenario:
In the non-rush hour scenario, surge multipliers are set to 1.0 for all drivers, representing normal fare conditions. The algorithm assigns passengers to drivers based on time

efficiency, resulting in the same driver allocations for all passengers. The total fares for all passengers are $6.41, indicating fair distribution and similar fare efficiency.

Efficiency and Optimization:
The algorithm optimizes the allocation process by using dynamic programming principles. It breaks down the problem into smaller subproblems of finding the most efficient driver for each passenger individually. By iteratively updating the minimum time efficiency and chosen driver, the algorithm ensures an efficient allocation that maximizes fare efficiency.

Performance and Scalability:
Based on the provided test cases, the algorithm performs well and produces the expected results. However, it's important to consider the scalability of the solution. The algorithm has a time complexity of O(n * m), where n is the number of passengers and m is the number of available drivers. For larger input sizes, the performance may be impacted, and further optimizations may be required.

General:
Dynamic programming approach allows for efficient computation of the most efficient driver for each passenger individually. By breaking down the problem into smaller subproblems and iteratively updating the minimum time efficiency and chosen driver, the algorithm achieves an optimal allocation strategy.

This algorithm demonstrates good performance and accuracy based on the provided test cases. However, it's essential to consider the scalability of the solution for larger input sizes. As the algorithm has a time complexity of O(n * m), where n is the number of passengers and m is the number of available drivers, further optimizations may be necessary to handle significant increases in passenger and driver counts efficiently.

The implemented solution provides a solid foundation for solving the online transportation resource allocation problem. It showcases the benefits of dynamic programming in achieving optimal results and balancing fare efficiency and fairness among drivers. With proper performance optimizations, the solution can be scaled to handle larger scenarios effectively.

## VI. CONCLUSION

In conclusion, the use of dynamic programming is proved effective in breaking down the resource allocation problem into smaller subproblems and finding optimal solutions. By considering all possible combinations of drivers and passengers, we achieved fair fares and improved overall system efficiency. It is important to note that our algorithm assumes a simplified scenario and does not consider additional factors such as traffic conditions or real-time updates. Future research could focus on incorporating these elements to enhance real-world applicability.

By optimizing assignments and fare calculations, the system efficiently allocates resources, providing a satisfactory experience for passengers and drivers. As the algorithm has a time complexity of O(n * m), where n is the number of passengers and m is the number of available drivers, further optimizations may be necessary to handle significant increases in passenger and driver counts efficiently.

VIDEO LINK YOUTUBE
https://youtu.be/sb3xtnzhMvE

PROJECT LINK: GITHUB
https://github.com/vanessrw/DynamicProg_ResourceAllocation

## ACKNOWLEDGMENT

The author expresses heartfelt gratitude to God Almighty for providing guidance and support throughout the completion of this paper. Special appreciation is extended to Dr. Ir. Rinaldi Munir, MT., the author's lecturer in the IF2211 Algorithm Strategies course K-01, fellow friends and colleagues, and author's family who have supported the journey of writing this paper.

## REFERENCES

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf, accesed on May 20th 2023

[2] Program Dinamis (Dynamic Programming) Bagian 1 (itb.ac.id) , accesed on May 20th 2023

[3] https://iopscience.iop.org/article/10.1088/1755-1315/286/1/012034/pdf, accesed on May 21th 2023

[4] https://www.cnbcindonesia.com/tech/20230519132705-37-438752/isu-krisis-ojol-hantam-aplikasi-grab-buka-bukaan-data-driver, accesed on May 21th 2023

[5] https://www.researchgate.net/publication/329390711_Dynamic_programming, accesed on May 21th 2023

[6] https://myrobin.id/untuk-bisnis/resource-allocation/, accesed on May 21th 2023

[7] https://www.sciencedirect.com/science/article/abs/pii/S0191261523000243, accesed on May 21th 2023

## STATEMENT

I hereby declare that the paper I wrote is my writing, not an adaptation, or a translation of someone else's paper, and not plagiarism

Bandung, May 20th 2023

Vanessa Rebecca Wiyono
13521151