Most Utilized Dock: Algorithmic Strategies and Performance Analysis

Halil ABACI
220401025

Hilal AY
220401030

Department of Computer Engineering
İzmir Katip Çelebi University
November 22, 2025

1 Introduction

This project addresses the problem of identifying the most utilized loading dock by modeling daily activity as a binary occupancy matrix $U \in \{0,1\}^{R\times T}$. We implement and compare two algorithmic strategies: a **Sequential Baseline** and a **Divide & Conquer** (D&C) approach.

The primary objective is to determine the dock with the maximum total occupancy, applying a deterministic tie-breaking rule that selects the smallest row index in case of equal totals.

2 Methodology

Our work follows a structured workflow covering data modeling, algorithm design, implementation, complexity reasoning, and experimental evaluation. Both the Sequential and D&Conquer methods were developed and tested under consistent conditions, with responsibilities clearly split between Member A and Member B.

2.1 Modeling the Occupancy Matrix U

We represented dock usage with a binary matrix $U \in \{0,1\}^{R\times T}$. Member A designed the event-log format and generated realistic arrival–departure intervals, while Member B handled time-slot discretization and filled the matrix by marking overlapping intervals. Both members validated correctness using heatmaps and row-sum checks. The detailed construction of U from raw logs is described in Section 3.

3 Data Preparation (Binary Occupancy Construction)

We constructed the binary occupancy matrix $U \in \{0,1\}^{R\times T}$ by converting raw dock event logs into a discretized daily occupancy representation. Member A focused on selecting the working day and preparing the raw event log file, while Member B implemented the Python functions that map these intervals to time slots and build the matrix U.

3.1 Selected Day and Slot Length (Δ)

We selected a single operational day from the log file and used a slot length of

$$\Delta = 5 \text{ minutes},$$

which divides a 24-hour day into

$$T = \frac{24 \times 60}{5} = 288 \text{ slots}.$$

This resolution captures short occupancy events while keeping the matrix efficient.

3.2 Number of Docks (R)

The dataset includes

$$R = 10$$

distinct docks, each represented as a row in matrix U. The row index i corresponds to a specific physical loading dock.

3.3 Interval-to-Binary Conversion Rule

For each dock event interval $[t_{\text{in}}, t_{\text{out}})$ in the log:

- compute time-slot indices t whose $[t, t + \Delta)$ window intersects $[t_{\text{in}}, t_{\text{out}})$,
- if the overlap is positive, set $U[i, t] = 1$,
- otherwise set $U[i, t] = 0$.

Thus, a slot is marked as occupied if the dock is busy for any positive amount of time within that slot.

==

4 Algorithms

4.1 Sequential Algorithm

The Sequential approach was implemented as the baseline. Member A wrote the algorithm that sums each row, identifies the maximum (using smallest index for ties), and returns the selected row[cite: 10]. Member B verified correctness on small test matrices and integrated the method into the experiment script.

The method scans each row of the occupancy matrix U, counting how many 1s appear across all time slots. Each row's total represents the dock's daily occupancy. During the scan, the algorithm tracks two variables: `best_row` and `best_count`[cite: 32]; a row replaces the current best only if its total is strictly larger. If the total is equal, the tie-breaking rule (keep the smallest row index) ensures no update is made. The algorithm visits all R rows and T columns with constant work per cell, giving time complexity $O(RT)$.

Sequential Pseudo-code

Algorithm 1 MOST_UTILIZED_DOCK_SEQUENTIAL(U)

```
Input: U, an R × T binary matrix
Output: (best_row, best_count)
best_row ← −1
best_count ← −1
for i ← 0 to R − 1 do
    current_count ← 0
    for t ← 0 to T − 1 do
        if U[i, t] = 1 then
            current_count ← current_count + 1
        end if
    end for
    if current_count > best_count then
        best_count ← current_count
        best_row ← i
    end if
end for
return (best_row, best_count)
```

Correctness

The Sequential algorithm is correct for two reasons:

- **Full Coverage:** The outer loop visits every row and the inner loop scans all T columns, so `current_count` accurately accumulates the number of 1s in each row.
- **Selecting the Maximum and Ties:** `best_count` stores the largest total seen so far and `best_row` its index. The update condition `current_count > best_count` ensures that the first row with maximum occupancy is kept, satisfying the smallest-index rule.

4.2 Divide & Conquer Algorithm

We implemented a D&C method based on splitting the matrix by columns. Member B developed two core functions:

- **dc_column_sum():** Recursively splits the matrix, computes partial row sums, and merges them via vector addition.
- **dc_argmax():** Performs a recursive tournament-style comparison to find the row with the highest occupancy while applying the tie-breaking rule.

The algorithm recursively splits U along the time axis. Instead of scanning all T columns at once, the matrix is divided into left and right halves. Each half is processed to produce a per-row count vector; the two vectors are combined using element-wise addition to obtain the total occupancy. Finally, a recursive argmax tournament selects the row with the highest occupancy while enforcing the tie-breaking rule of choosing the smallest row index.

Split/Combine Justification

The D&C formulation is justified because:

- **Column independence:** Each time slot is independent, so splitting the matrix across columns preserves correctness.
- **Balanced subproblems:** Dividing T columns into two halves creates two $R \times (T/2)$ subproblems with logarithmic recursion depth.
- **Efficient merging:** The combine step is element-wise vector addition in $O(R)$.
- **Natural decomposition and ties:** Total occupancy per row is $left+right$, and the argmax tournament maintains the smallest-index tie rule.

Algorithm 2 D&C Functions

```
Function DC_Column_Sum(U, left, right)
if left = right then
    return column_vector(U, left) {size R}
end if
mid ← ⌊(left + right)/2⌋
L ← DC_Column_Sum(U, left, mid)
R ← DC_Column_Sum(U, mid + 1, right)
return L + R {element-wise sum}

Function DC_Argmax(A, left, right)
if left = right then
    return (left, A[left])
end if
mid ← ⌊(left + right)/2⌋
(i1, v1) ← DC_Argmax(A, left, mid)
(i2, v2) ← DC_Argmax(A, mid + 1, right)
if v1 > v2 then
    return (i1, v1)
else if v2 > v1 then
    return (i2, v2)
else
    return (min(i1, i2), v1) {tie-handling}
end if
```

Correctness of the D&C Method

The D&C algorithm is correct because:

- **Valid decomposition:** Splitting by columns preserves totals since $Total_i = Left_i + Right_i$.
- **Base case and merging:** When $T = 1$, each column already provides the exact 0/1 occupancy, and element-wise addition accurately combines row totals from subproblems.
- **Argmax tournament:** Each merge step preserves the true maximum and consistently applies the tie rule.

5 Complexity Analysis

5.1 Sequential Method

- **Time Complexity:** The algorithm scans all R rows and all T columns once:

$$T_{seq} = \Theta(RT).$$

- **Space Complexity:** Only constant variables are used:

$$S_{seq} = O(1).$$

5.2 Divide & Conquer Method

- **Time Complexity:** Column-sum recursion follows $C(R,T) = 2C(R,T/2) + O(R)$. By Master Theorem:

$$T_{DC} = \Theta(RT).$$

- **Space Complexity:** Recursion depth is $\log T$, storing one R-size vector per level:

$$S_{DC} = O(R \log T).$$

5.3 Comparison

Although both algorithms have the same asymptotic running time $\Theta(RT)$, the Sequential method is faster in practice because it avoids recursion and uses $O(1)$ space.

6 Experimental Framework

The goal of the experimental phase is to compare the Sequential and D&C algorithms in terms of running time while using the same occupancy matrices U[cite: 127]. All experiments were conducted using the generated data from `raw_logs.csv` and the matrix construction function implemented in the project.

6.1 Experimental Setup

We fixed the hardware and software environment to ensure fair comparison.

Hardware

- CPU: AMD Ryzen 7 7735HS. (4 cores, 8 threads, up to 4.2 GHz)
- RAM: 16 GB
- Storage: 512 GB SSD
- Operating System: Windows 10 (64-bit)

Software

- Programming Language: Python 3.11 (CPython)
- Libraries: NumPy, Matplotlib, `time`
- IDE: Visual Studio Code
- Execution script: `run_experiment.py`

Member A defined the tested (R,T) configurations and generated the corresponding occupancy matrices, while Member B implemented the experiment loop and result logging.

6.2 Experimental Protocol

Correctness Verification

Both algorithms were executed using the same occupancy matrix U. For all tested configurations, both methods produced identical output pairs:

$$\text{Sequential}(U) = \text{D\&C}(U),$$

confirming correctness and tie-breaking behavior.

Runtime Measurement

Runtime was measured with the high-resolution timer `time.perf_counter()`. Each experiment was repeated

$$N = 10$$

times. For each configuration, we report mean runtime and standard deviation, for example

$$0.00421 \text{ s} \pm 0.00031.$$

Input Scaling

We tested multiple input sizes:

$$R \in \{10, 50\}, \quad T \in \{100, 500, 1000\},$$

to observe linear growth in runtime and the recursion overhead of the D&C method.

6.3 Results Summary

For all tested (R, T) values, the **Sequential method consistently ran faster** than the D&C method. The running-time curves for both methods increased roughly linearly with respect to T, consistent with their $\Theta(RT)$ theoretical complexity. D&C exhibited higher runtime due to recursive overhead, intermediate vectors, and Python's function call cost The gap between methods widened for larger matrix sizes.

6.4 Visualization

Both members produced the required plots using Matplotlib and saved them under `/figures/`:

- occupancy matrix heatmap,
- row totals bar chart,
- runtime comparison chart.

These figures validated both correctness and performance trends.

7 Figures

7.1 Occupancy Matrix Heatmap

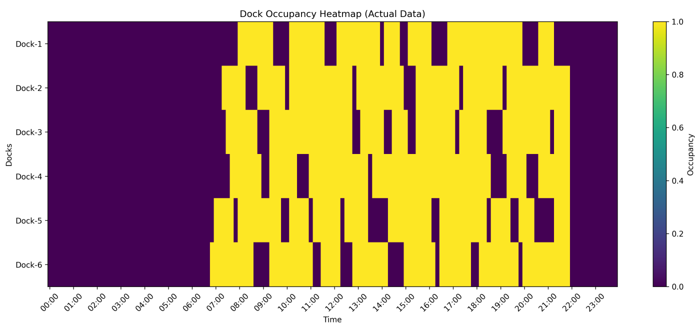

Figure 1: Binary occupancy matrix U (rows: docks, columns: time slots).

7.2 Row Totals (Bar Chart)

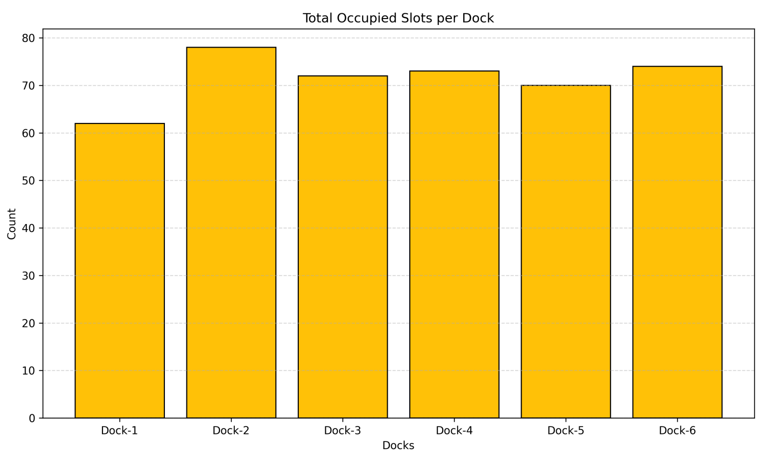

Figure 2: Total occupancy count per dock; the highest bar corresponds to the most utilized dock.

7.3 Runtime Comparison (Sequential vs D&C)

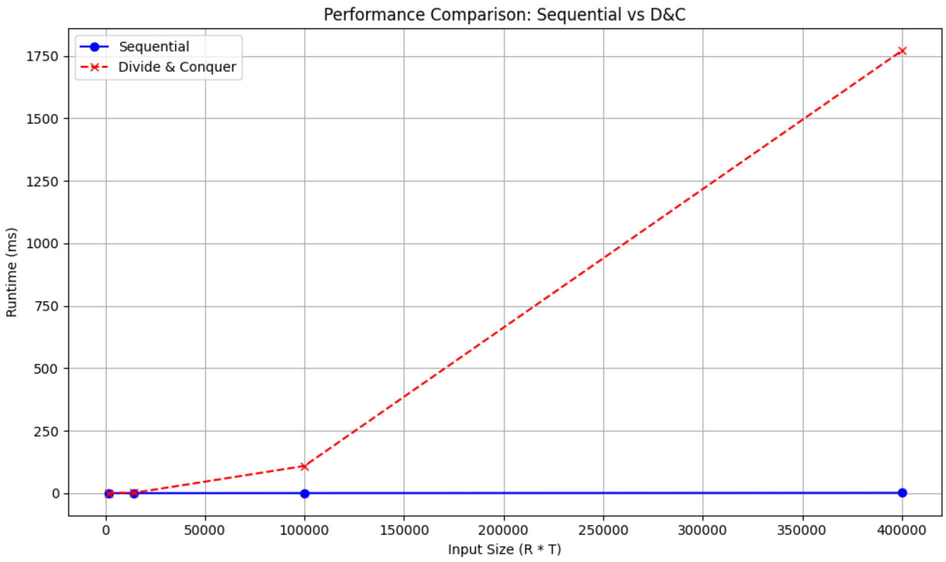

Figure 3: Runtime comparison of Sequential and D&C algorithms over different T values.

8 Conclusion

The matrix employed in our analysis displays high sparsity, which realistically represents short and intermittent dock usage patterns and directly affects both runtime and memory performance as R and T increase. Within this framework, we modeled dock activity through a binary occupancy matrix and applied two distinct approaches—Sequential scanning and a Divide-and-Conquer strategy—to determine the most utilized dock. Both algorithms reliably produced correct results and adhered to the assignment's tie-breaking rules. Theoretical examination demonstrates that the two methods share the same asymptotic time complexity, $\Theta(RT)$, yet differ significantly in memory requirements: the Sequential approach operates with constant extra space, whereas the D&C method incurs an $O(R \log T)$ overhead due to its recursive structure. Experimental evaluation further reinforces these insights; although both methods scale linearly with respect to the number of time slots, the Sequential algorithm consistently outperforms the D&C approach thanks to its lower overhead and simpler memory usage.

9 Reproducibility and File Organization

We kept the project organized in a single repository named `GroupID_MostUtilizedDock`. The root folder contains the final report (`report.pdf`) and a short `README.md`. All data files are stored under `data/` (original `logs.csv` and the derived `generated_U.csv`), while all source code lives in `src/`, including `sequential.py`, `divide_and_conquer.py`, `matrix_builder.py`, and the driver script `run_experiment.py`. The plots used in the report (heatmap, row-totals bar chart, and runtime comparison) are saved in the `figures/` directory as PNG files.

All experiments can be reproduced by executing `python src/run_experiment.py` in the project root. This script constructs the occupancy matrix U, runs both algorithms, measures their running times for the selected (R, T) configurations, and regenerates all required figures in the expected format. This script:

- constructs the occupancy matrix U,
- runs both algorithms,
- measures runtime,
- generates all required figures,
- prints output in the expected format.