Alternative Solution to International Olympiad in Informatics 2023 'Overtaking' Problem Using Dynamic Programming and Binary Search

Kloce Paul William Saragih - 13524040
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: kloce.edu@gmail.com , 13524040@std.stei.itb.ac.id

***Abstract*—The "Overtaking" problem from the International Olympiad in Informatics (IOI) 2023 was one of the challenging task. While the official solution employs a clever and efficient segment-point decomposition, this paper introduces an alternative approach that combines dynamic programming and binary search. While this solution does not surpass the official solution in time complexity, it offers a more accessible and easier-to-implement alternative. Experimental analysis and comparisons with the official solution highlight the strengths and trade-offs of this alternative approach.**

***Keywords*— International Olympiad in Informatics, Dynamic programming, Binary search, Memoisation.**

I. Introduction

A. Background Information

The International Olympiad in Informatics (IOI) is one of the many international science olympiads in the field of computer science for high school students that is held every year. Each competitor is required to create a program in order to solve each presented task efficiently.

One of the problems featured in IOI 2023 was "Overtaking", authored by Bernard Teo. The official problem statement spans five pages and includes detailed explanations, sample cases, subtasks, and more. The official solution for this problem can be found in Eljakim Schrijvers's youtube channel. However, this paper focuses on presenting an alternative approach to solve the problem with dynamic programming and binary search.

B. Problem Statment

The official problem statement can be found at the following link: IOI Overtaking. While it is recommended to read the full official description, the core idea of the problem can be summarized as follows:

Given $N + 1$ buses traveling along a one-way road from the airport to the hotel, N regular buses depart at specified times and travel at fixed speeds, while one reserve bus's departure time is to be determined. Buses are not allowed to overtake each other except at designated sorting stations located at positions $S[0], S[1], \ldots, S[M]$, where $S[0] = 0$ (the airport) and $S[M]$ is the hotel. At each sorting station, buses may be reordered based on their expected arrival times. The task is to answer Q queries: for each given departure time Y of the reserve bus, determine the time it will arrive at the hotel, accounting for delays caused by slower buses ahead.

C. Overview

Chapter II outlines thereotical basis relevant to the alternative approach. Chapter III presents an indepth analysis of the problem, beginning with a series of naïve and inefficient algorithms and gradually refining them into the final optimized solution. Chapter IV compares the performance of each solution presented in chapter III, including the final optimized solution and the official solution of this problem.

II. Thereotical Basis

A. Graph

A graph G is defined as an ordered pair $G = (V, E)$, where V is the set of vertices and E is the set of edges, where each edge connects two vertices. Based on the relation of its vertices, a graph can be classified as:

a) Undirected graph

An edge that connects A and B can be traversed from A to B and B to A.

b) Directed graph

An edge that connects A and B can only be traversed from A to B.

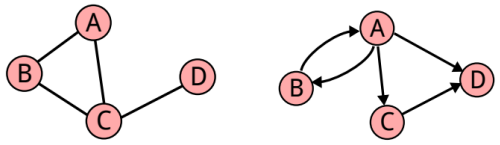


Fig. 1. Undirected graph (left) and directed graph (right) [https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

Based on the weight of it' edges, a graph can also be classified as:

a) Unweighted graph

An unweighted graph consists of edges with uniform edges. An unweighted graph only considers the relation of each vertices.

b) Weighted graph

A weighted graph consists of edges with various weights. Weight of an edge could represent distance or cost.

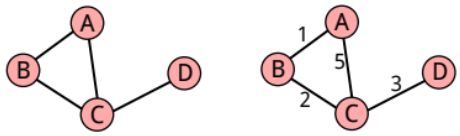


Fig. 2. Unweighted graph (left) and weighted graph (right) [https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

A cycle of a graph is a path that starts and ends at the same vertex with all edges and vertices (except the starting and ending vertex) being distinct.

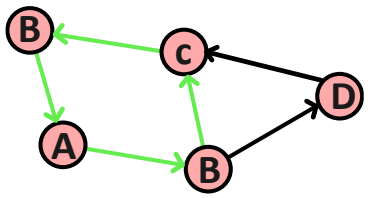


Fig. 3. A cycle of an undirected graph (A-B-C-D)

A directed acyclic graph (DAG) is a special form of directed graph. A directed acyclic graph does not form any cycles.

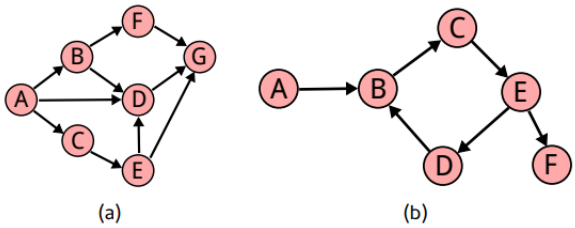


Fig. 4. Figure (a) forms a DAG, however figure (b) isn't since it consist of a cycle (B-C-E-D) [https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

B. Time Complexity

The time complexity of an algorithm estimates how much time the algorithm will use for some input. We denote the amount of time an algorithm takes in relation N as $T(N)$. We can classify time complexities in three different categories:

a) $T_{max}(n)$: worst case time complexity

b) $T_{min}(n)$: best case time complexity

c) $T_{avg}(n)$: average case time complexity

In this paper, each algorithm will be measured by its worst-case time complexity.

We will express the running time of each algorithm in this paper with an *asymptotic notation*. There exists three different asymptotic function, namely big-O, big-Omega, and big-Theta. Let f and g be a non-negative function. We define each asymptotic function as follows:

a) If there exists positive constants n_0 and c such that $f(n) \leq cg(n)$ whenever $n >= n_0$, we say that $f(n) = O(g(n))$ (big-O of $g(n)$).

b) If there exists positive constants n_0 and c such that $f(n) \geq cg(n)$ whenever $n \geq n_0$, we say that $f(n) = \Omega(g(n))$ (big-Omega of $g(n)$).

c) If $f(n) = O(n)$ and $f(n) = \Omega(n)$, we say that $f(n) = \Theta(g(n))$ (big-Theta of $g(n)$).

The following are the properties of big-O notation:

1) $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$.
2) $O(f(n)) . O(g(n)) = O(f(n) . g(n))$

C. Binary Search

The standard usage of binary search is to find an item in a sorted array. The process involves checking the middle element to see if it matches the target item. If it does, we stop. If not, we decide whether we should continue to search the left or right half of the array based on the comparison. The maximum amount of operation required until the target item is equal to the length of sequence [*N,N/2,N/4,…, 2,1*] which is $\lceil log_2 N \rceil$, so the time complexity is $O(\log n)$. There also exist a slight modification of binary search to find the lower bound and upper bound of an element in a sorted array.

Fig. 5. A simple ilustration of binary search.

D. Dynamic Programming

Dynamic programming (DP) is a problem-solving method that utilizes information from solving smaller subproblems. The solution to each subproblem is only calculated once and stored

in memory. DP is effective when a problem exhibits overlapping subproblems, allowing us to build solutions while storing the previous results. By restoring the results of subproblems, DP reduces redundancy of computation and lowering time complexity. However, this trade-off involves increased space usage.

III. Problem Analysis

A. Notation

In this section, we develop some notations and structures to analyze the problem. We will use the pseudocode notation to represent the properties of each bus.

```
type Bus <
      departureTime : integer,
      pace          : integer,
      id            : integer
>
bus : array [0..N] of Bus
```

Each element bus_i holds the following properties:

- *departureTime:* the time (in seconds) when the bus leaves the previous station. This value will be updated in each station.
- *pace:* time in seconds (in seconds) to travel one kilometer. (denote the pace of the reserve bus as *X*).
- *id:* the unique identifier of each bus. This is used to track and reorder the bus efficiently.

The final answer to each query is the departureTime of the bus whose id equals the query value N. We will also define the following variables and notations:

- *N* : the number of non-reserved bus. (where bus_N is the reserve bus).
- *M* : the number of sorting stations.
- S_i : the position in km of the i'th sorting station. (where S_{M-1} being the destination).
- $t_{i,j}$: the arrival time of bus *i* to station *j*.
- $e_{i,j}$: the expected arrival time of bus i to station j.

$$e_{i,j} = T_{i,j-1} + pace_i * (S[j] - S[j-1])$$

Let *computeExpArrival(bus, idx)* return the expected arrival time of bus at station *idx*.

```
function computeExpArrival(b : Bus, idx : integer) → integer
   distance ← S[idx] - S[idx-1]
   → b.departureTime + distance * b.pace
```

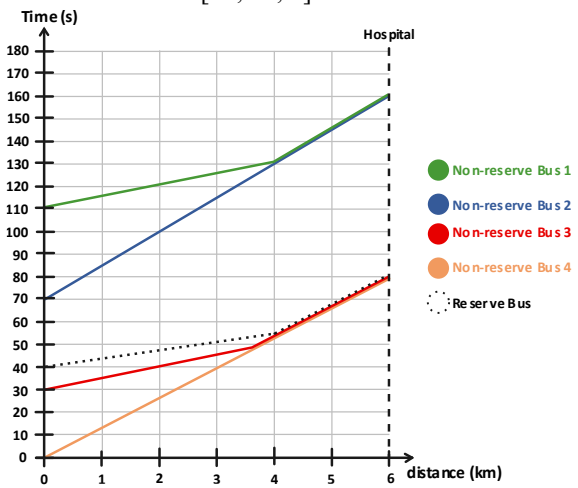
B. Solution I – Naïve Approach

We denote each bus as a tuple [*a*,*b*,*c*], where:

- *a represents the departure time (in seconds),*
- *b is the pace (time per kilometer),*
- *c is the ID of the bus.*

Let's try to solve the problem for $M = 2$ and $N = 4$, where initially:

- Non-reserve bus 1: [110, 5, 1]
- Non-reserve bus 2: [70, 15, 2]
- Non-reserve bus 3: [30, 5, 3]
- Non-reserve bus 4: [0, 50, 4]
- Reserve bus : [40, 50, 5]



For each $0 \leq i \leq N$ and $0 \leq j \leq M$, we denote the actual arrival time of bus *i* at time *j* as $t_{i,j}$. Observe that the value of $t_{i,j}$ is determined by taking the maximum value between:

- $e_{i,j}$, and
- $e_{k,j}$ such that bus *k* arrived at station *j*-1 before the bus *i*, for all $0 <= k <= N$ where $t_{k,j-1} < t_{i,j-1}$

We will sort the buses by its departure time in ascending order. If there are two buses with the same departure time, we will sort them by their pace in ascending order.

```
procedure sortByDeparture(input/output B : array of Bus)
{ Sorts B by its departure time non-decreasingly
  if two bus has the same departureTime, they will be
  sorted by their pace (non-decreasing) }
```

In order to fully compute the arrival time of each bus to the first sorting station $t_{i,1}$. We can sort the bus based on the rules mention before, then we will calculate the values in the sorted order.

```
curMax ← 0
sortByDeparture(bus)
i traversal [0..N]
   exp ← computeExpArrival(busᵢ, 1)
   busᵢ.departureTime ← max(exp, curMax)
   curMax ← max(curMax, busᵢ.departureTime)
→ busfindID(N).departureTime
```

Once the arrival times $t_{i,1}$ at the first sorting station have been computed using the sorted order, we can extend this method to support $M > 2$ by applying the same logic repeatedly for each station from $j = 1$ to $j = M - 1$.

```
repeat M-1 times
    curMax ← 0
    sortByDeparture(bus)
    i traversal [0..N]
      exp ← computeExpArrival(bus_i, 1)
      bus_i.departureTime ← max(exp, curMax)
      curMax ← max(curMax, bus_i.departureTime)
→ bus_findID(N).departureTime
```

Since sorting has a time complexity of $O(N\ log\ N)$, the final complexity of the code above is $O(M \cdot N\ log\ N)$. Repeating this to each Q queries grant us a final time complexity of $O(Q \cdot M \cdot N \log N)$.

C. Solution 2 – Optimization of solution I

Observe that any bus with a pace greater than X (the pace of the reserve bus) will never affect the final answer to each query. This is because a bus can only be delayed by other, slower buses ahead of it—never by faster ones behind it. Therefore, before processing each query, we can safely discard all buses whose pace exceeds X. After filtering, we compute and **store** the arrival times $t_{i,j}$ for all remaining buses, where $0 \leq i < N$ and $0 \leq j < M$. Note that for each station j, the array $t_{x,j}$ must be sorted. This will help us process each query efficiently. We will also use this setup for solution 3.

Consider two bus bus_i and bus_j $(0 \leq i, j \leq N)$, and suppose at some station x $(0 \leq x < M\text{-}1)$, The arrival time of i is less than or equal to that of bus j. i.e. $t_{i,x} \leq t_{j,x}$. It follows that their arrival order will be preserved at the next station as well, meaning:

$$t_{i,x+1} \leq t_{j,x+1}$$

This implies that the sequence of arrival times from one station to the next is **non-decreasing** for each bus relative to others. Once a bus is ahead of another at a station, it will not be overtaken in the following stations under this simulation model.

Using this information, we can eliminate the calculation needed to find *curMax* at each station. Instead, we perform **binary search** to find the nearest bus ahead of the reserve bus in each station. We then compare its arrival time with the expected arrival time of the reserve bus to determine whether a delay should occur.

Since binary search has a time complexity of $O(\log N)$, the total complexity to answer Q queries is:

$$O_{query} = O(Q) \cdot O(M \log N) = O(Q \cdot M \log N).$$

Since we have to sort the array $t_{x,j}$ for each station j before processing each query (to ensure we can apply binary search), the time complexity for initialization before processing each query is calculated as follows:

$$O_{init} = O(M) \cdot O(N \log N) = O(M \cdot N \log N)$$

The total time complexity can be calculated as follows:

$$O_{total} = O_{init} + O_{query}$$

$$O_{total} = O(M \cdot N \log N) + O(Q \cdot M \log N)$$

$$O_{total} = O(M \log N \cdot (Q + N))$$

D. Solution 3 – DP approach

Let's define a special point as the intersection between the station line and the bus line in the illustration. If we start at any point and end up at a special point, we will have the same result. Because of this, we can actually pre-calculate the result if we start at all the special points. Since there N stations and M buses, we will have at most NM special points.

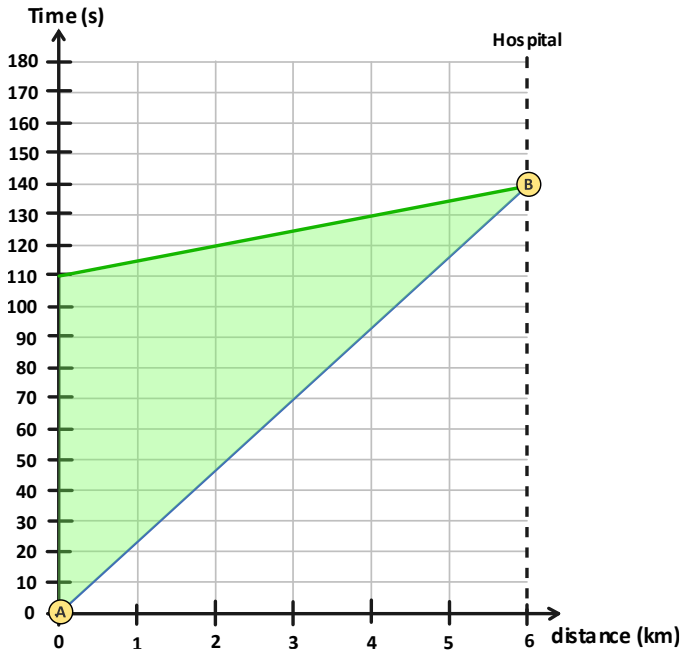


Fig. 6. Point A and B is a special point. For X = 5, if we start at any point within the green area, we will end up at point B and get a final answer of 140.

If the reserve bus start at a special point x, it may eventually reach another special point y. This relationship can be represented using a directed graph, where each node corresponds to a special point, and a directed edge from node x to node y indicates that a departure from x leads to an arrival at y.

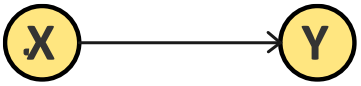


Fig. 7. Departuring from a special point x leads to an arrival at special point y, adj(X) = {Y}.

Suppose we have figured out the relation of each special point within the illustration. Observe that the graph created from such relation is a directed acyclic graph.

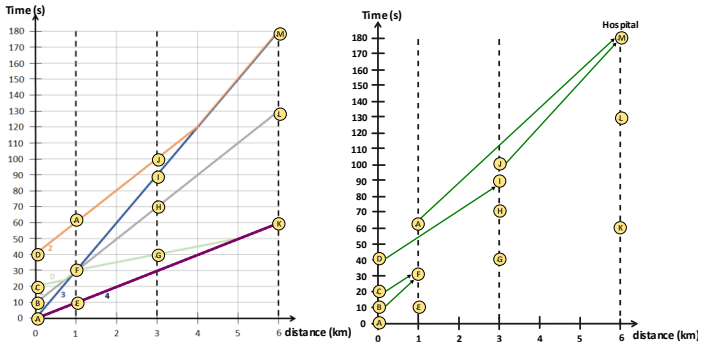


Fig. 8. Position of each special point (left), DAG form of the illustration [assuming X = 1] (right).

Let's define *DP(x)* where *x* is a node from the DAG as the final answer if we start at point *x*. The value of *DP(x)* is defined as follows:

$$DP(x) = \begin{cases} 0, & \text{if } x = \{\} \\ DP(y), \quad y \in x, & \text{if } x \neq \{\} \end{cases}$$

We would like to precompute the value of *DP* for all node, however we are required to construct the DAG first. A straightforward simulation would take $O(M)$ per vertex, which is quite inefficient.

One key observation is that, after filtering out unnecessary buses (i.e., those with pace greater than the reserve bus), the number of buses ahead of the reserve bus is **non-increasing** at each station. This monotonicity allows us to perform **"long jumps"** and avoid recomputing from scratch at each station a. After precomputing and storing the arrival times of the other buses at all stations, we will do the following:

- Use binary search to efficiently determine the last station *p* that can be reach without being interrupted by other bus (i.e, the number of bus ahead of it does not increase),
- Jump directly to station *p*, updating the arrival time,
- Then perform another binary search to find the next state (the next node in the DAG) to transition to.
- Save (memoize) the value of each *DP*.

This strategy significantly reduces the number of steps per query from linear in *M* to logarithmic in *M*. Since the number of special points are at most *NM,* the time complexity to compute the value of *DP*(*x*) for all node x is $O(NM \cdot \log M \cdot \log N)$.

This approach models the reserve bus's journey as a path through a DAG where each node represents a state (station + number of buses ahead), and transitions represent "long jumps" followed by one "normal jump" to the next relevant node.

Hence, the total complexity for precomputation and initialization is:

$$O_{init} = O(NM \cdot \log N \cdot \log M)$$

We will apply this process for each query, replacing the naïve simulation of one station at a time with an optimized sequence of **binary searches and long jumps** to check whether the reserve bus can reach the last station uninterrupted. Two cases might happen:

1) *The reserve bus arrived at some station p at the same time with other bus.* If this happens, we can return the precomputed *DP* value at the corresponding vertex *x* where both bus arrived the same time.

2) *The reserve bus reaches the last station uninterrupted.* In this case, we simply return the expected arrival time of the bus to the destination.

This reduces the time complexity to process all queries from $O(Q \cdot M \log N)$ to $O(Q \cdot \log \mathrm{M} \log N)$.

The total complexity of this solution is computed as follows:

$$O_{total} = O_{init} + O_{query}$$

$$O_{total} = O(M \cdot N \log N) + O(Q \cdot \log \mathrm{M} \log N)$$

$$O_{total} = O(MN \log N + Q \log \mathrm{M} \log N)$$

IV. IMPLEMENTATION

The implementation of each solution will be implemented in C++. There are two main parts of the code:

- *init(...)*: this part will consist of initialization and precomputation needed before processing each query. This procedure will only be called once.
- *arrival_time(Y)*: this part will consist of code requires to process each query. This function will be called *Q* times.

The following is a snippet of declarations and utility functions used in each solution:

```cpp
typedef long long ll;

struct Bus {
    ll arrivalTime; // updated as the bus progresses
    ll pace;
    int id;
};

bool byArrivalThenPace(const Bus &a, const Bus &b) {
    if (a.arrivalTime == b.arrivalTime)
        return a.pace < b.pace;
    return a.arrivalTime < b.arrivalTime;
}

vector<Bus> buses;        // current simulation state
vector<Bus> buses_init;   // initial state
vector<ll> stations;
int N, M;

ll computeExpectedArrival(const Bus& b, int j) {
    ll dist = stations[j] - stations[j - 1];
    return b.arrivalTime + dist * b.pace;
}
```

Fig. 9.Declarations and utility functions implementations

A. Solution I Implementation

```cpp
void init(int Lp, int Np, vector<ll> Tp, vector<int> Wp, int Xp, int Mp, vector<int> Sp) {
    buses.clear();
    stations.assign(Sp.begin(), Sp.end());

    for (int i = 0; i < Np; i++) {
        buses.push_back({Tp[i], Wp[i], i});
    }

    // Add reserve bus
    buses.push_back({0, Xp, Np});

    N = (int)buses.size();
    M = Mp;

    // Save initial state for resetting on each query
    buses_init = buses;
}

ll arrival_time(ll Y) {
    // Reset buses to initial state for this query
    buses = buses_init;

    // Set reserve bus departure time
    for (auto &b : buses) {
        if (b.id == N - 1) {
            b.arrivalTime = Y;
            break;
        }
    }

    // Simulate each station
    for (int j = 1; j < M; j++) {
        sort(buses.begin(), buses.end(), byArrivalThenPace);

        ll curMax = 0;
        for (int i = 0; i < N; i++) {
            ll expected = computeExpectedArrival(buses[i], j);
            buses[i].arrivalTime = max(expected, curMax);
            curMax = max(curMax, buses[i].arrivalTime);
        }
    }

    // Find and return reserve bus's final arrival time
    ll ret = 0;
    for (const auto &b : buses) {
        if (b.id == N - 1)
            ret = b.arrivalTime;
    }

    return ret;
}
```

Fig. 10. Implementation of solution I

The previous code highlights the main functionality of both functions. First, we save the departure time, pace, and ID of each bus in another array of bus *buses_init*. Each time a query is called, we reset the attributes of each regular bus before proceeding to other calculations.

B. Solution II Implementation

The main difference between solution I and II is the precomputation. Solution II filters, simulates, and save the arrival time of all necessary buses. The main implementation difference between solution I and solution II is highlighted in yellow.

```cpp
void init(int Lp, int Np, vector<ll> Tp, vector<int> Wp, int Xp, int Mp, vector<int> Sp) {
    // Load Data
    L = Lp; reservePace = Xp; M = Mp;
    stations = Sp;
    N = Np;

    // Load regular bus
    for (int i = 0; i < N; i++) {
        if (Wp[i] >= reservePace)
            buses.push_back({Tp[i], Wp[i], i});
    }
    N = (int)buses.size();

    // Initialize vectors before accessing
    arrivalRanges.assign(M, vector<pair<ll,ll>>());
    arrivalStarts.assign(M, vector<ll>());
    prefixMaxArrival.assign(M, vector<ll>());

    // Simulate the process and store the arrival times in each station
    ll prevPos = 0;
    for (int i = 1; i < M; i++) {
        sort(buses.begin(), buses.end(), byArrivalThenPace);

        ll maxArrival = 0;
        arrivalRanges[i].clear();
        arrivalStarts[i].clear();

        for (int j = 0; j < N; j++) {
            ll start = buses[j].arrivalTime;
            ll expected = start + dist(i) * buses[j].pace;

            if (expected <= maxArrival) {
                arrivalRanges[i].emplace_back(start, maxArrival);
                buses[j].arrivalTime = maxArrival;
            } else {
                arrivalRanges[i].emplace_back(start, expected);
                buses[j].arrivalTime = expected;
                maxArrival = expected;
            }
            arrivalStarts[i].push_back(start);
        }
        prevPos = stations[i];
    }


    for (int i = 0; i < M; i++) {
        sort(arrivalRanges[i].begin(), arrivalRanges[i].end());
        sort(arrivalStarts[i].begin(), arrivalStarts[i].end());

        prefixMaxArrival[i].resize(arrivalRanges[i].size() + 1);
        prefixMaxArrival[i][0] = 0;
        for (int j = 1; j <= (int)arrivalRanges[i].size(); j++)
            prefixMaxArrival[i][j] = max(prefixMaxArrival[i][j-1], arrivalRanges[i][j-1].second);
    }
}
```

Fig. 11. Implementation of initialization function for solution II

Additionally, we store additional information from each station:

- *arrivalRanges:* pairs of (start arrival, adjusted arrival) times of buses
- *arrivalStarts*: sorted list of start arrival times
- *prefixMaxArrival*: prefix maximum of adjusted arrivals

After pre-calculating the required information, we can process each query by simulating the process efficiently with

binary search. The following code utilizes the built-in function *upper_bound()* to find the corresponding the queries.

```
ll arrival_time(ll Y) {
    ll prevPos = 0, arrival = Y;

    for (int i = 1; i < M; i++) {
        auto &v =arrivalStarts[i];
        int idx = (upper_bound(v.begin(), v.end(), arrival - 1) - v.begin());
        ll d = stations[i] - prevPos;
        if (arrival + d * reservePace < prefixMaxArrival[i][idx])
            arrival = prefixMaxArrival[i][idx];
        else
            arrival += d * reservePace;
        prevPos = stations[i];
    }
    return arrival;
}
```

Fig. 12. Implementation of query function for solution II

C. *Solution III Implementation*

Due to space constraints, the implementation of Solution III is available at the following link: Implementation-of-Solution-III-Overtaking.

D. *Runtime Comparison*

This section will provide the run-time comparison of solution I, II, III, and the official solution. It is important to note that the official solution works in:

$$O_{total} = O(MN \log N + Q \log MN)$$

While the implementation of the official solution is not available, the compared solution follows the same core idea as the official one.

Each algorithm will be tested with fixed parameters $N = 100$ and $M = 100$, while varying the number of queries Q.

num. of queries(Q)	Runtime (ms)			
	Sol. I	Sol. II	Sol. III	Official
10	10	0	4	3
100	85	0	5	3
1000	826	34	7	4
10000	8124	128	22	9
100000	83503	1282	174	59
10000000	-	12571	1415	566
100000000	-	126368	13904	5416

Fig. 13. Runtime comparison of each solution. All algorithms were run on an AMD Ryzen™ 7 4800H Mobile Processor (8-core/16-thread, 12MB Cache, up to 4.2 GHz boost).

As Q increases, the runtime of each solution reveal its scalability. Significantly, solution I becomes infeasible for large Q, while the official and solution III remain efficient.

We can also plot the performance of each solution in a graph as following:

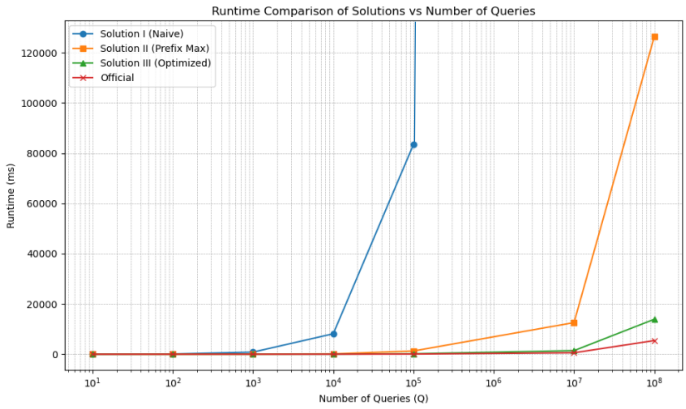


Fig. 14. Line graph plotting of each solution's runtime

V. Conclusion

This paper presents an alternative approach to the "Overtaking" problem from IOI 2023. The solution combines dynamic programming and binary to search to address the limitation of a pure simulation method. Three solutions were discussed:

- Solution I, a naïve simulation, straighforward and easy to implement, but has a poor scalability. This solution becomes infeasible for large inputs
- Solution II, an optimized version of solution I by utilizing prefix information and binary search, significantly reduces the time required to answer each query by eliminating unncessary and redundant computation with the help of precomputed structures.
- Solution III, models the problem as a DAG over "special points", applying dynamic programming (memoization) and binary search techniques to further reduce the computation required to answer each query, achieving a complexity competitive with the official solution.

Experimental analysis confirmed the thereotical time complexities (in big-O notation). Especially solution III that scales well even for $Q = 10^8$. The third solution presented in this paper demonstrates a compareable performance to the official solution idea.

The solution provided in this paper offers a creative and clear alternative to the official solution,. The findings demonstrates how dynamic programming and binary search can be effectively combined to handle complex problems efficiently.

The implementation of each problem has been tested and judged on many online judges such as DMOJ and OJ.UZ, in which solution III able to pass the whole test cases achieving a score of 100.

VI. Acknowledgement

I would like to express my gratitude to the organizers and problem authors of the International Olympiad in Informatics (IOI) 2023 for providing such an interesting problem. I would like to thank Eljakim Schrijvers and the IOI scientific commitee for sharing insightful explanation of the official solution. Additionally, I am thankful to Dr. Rinaldi Munir for his valuable lecture materials in IF1220 discrete mathematics course.

VII. References

[1] Rinaldi Munir, *Kompleksitas Algoritma (Bagian 2)*, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf [Accessed: June. 17, 2025]

[2] R. Munir, "Pohon (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: June. 17, 2025].

[3] TOKI, "Pemrograman Kompetitif Dasar," OSN TOKI. [Online]. Available: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf. [Accessed: June. 18, 2025]

[4] E. Schrijvers, *IOI 2023 Overtaking - Solution*, YouTube, 2023. [Online]. Available: https://www.youtube.com/watch?v=MZ9MMvnzhO4 [Accessed: June. 15, 2025]

[5] J. Sannemo, *Principles of Algorithmic Problem Solving*, October 24, 2018. [Online]. Available: https://usaco.guide/PAPS.pdf [Accessed: June. 19, 2025]

VIII. Appendix

Every implementation mentioned in this paper can be found at the following link:
https://github.com/YeyThePotatoMan/Makalah-Matdis

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Kloce Paul William Saragih - 13524040