# Report for Homework 3 on Analysis of Algorithms 2

Taras Demchyna 912310109

I have different output in scheduling algorithm, but priority gain is the same for every case.

**Pseudo code and time complexity for main methods of the program:**

FUNCTION **weighted\_interval\_scheduling**(schedules)

optimal\_schedules <- empty list

FOR EACH schedule IN schedules DO

Sort schedule.rooms by schedule.interval.startTime // O(n log n)

schedule.previousIntervals <- matchInterval(schedule.rooms) // O(n log n)

schedule.weightedIntervals() // O(n^2)

Reverse schedule.scheduleOPT // O(n)

Create temp Schedule object temp with schedule.floorNumber and schedule.scheduleOPT

Add temp to optimal\_schedules

END FOR

RETURN optimal\_schedules

FUNCTION **matchInterval**(rooms)

start <- empty list

finish <- empty list

FOR EACH room IN rooms DO

Add parseTime(room.interval.startTime) to start // O(n)

Add parseTime(room.interval.finishTime) to finish // O(n)

END FOR

result <- empty list

FOR i FROM 0 TO rooms.size() - 1 DO

index <- findIndex(finish, start[i]) // O(log n)

Add index to result

END FOR

RETURN result

FUNCTION **optimal**(i)

IF i EQUALS -1 THEN

RETURN 0

ELSE IF i < optimalRooms.size() THEN

RETURN optimalRooms[i]

ELSE

RETURN MAX(rooms[i].priority + optimal(previousIntervals[i]), optimal(i - 1))

END IF

FUNCTION **weightedIntervals**()

FOR i FROM 0 TO rooms.size() - 1 DO

optimal\_i <- optimal(i)

Add optimal\_i to optimalRooms

END FOR

Scheduling(rooms.size() - 1)

FUNCTION Scheduling(i)

IF i > -1 THEN

IF rooms[i].priority + optimal(previousIntervals[i]) >= optimal(i - 1) THEN

Add rooms[i] to scheduleOPT

Scheduling(previousIntervals[i])

ELSE

Scheduling(i - 1)

END IF

END IF

FUNCTION **parseTime**(time)

item1 <- stoi(time.substr(0, 2)) // O(1)

item2 <- stoi(time.substr(3, 2)) // O(1)

result <- item1 \* 60 + item2 // O(1)

RETURN result

FUNCTION **mergeRooms**(roomsPriority, roomsInterval)

result <- empty list

j <- 0

FOR i FROM 0 TO roomsInterval.size() - 1 DO

IF roomsPriority[j].roomNumber NOT EQUALS roomsInterval[i].roomNumber THEN

j++

END IF

Add roomsInterval[i] to result

result[r++].priority <- roomsPriority[j].priority

END FOR

RETURN result

FUNCTION **transformation**(roomsSchedule)

n <- 0

result <- list containing one empty Schedule object

result[0].floorNumber <- roomsSchedule[0].floorName

FOR i FROM 0 TO roomsSchedule.size() - 1 DO

IF roomsSchedule[i].floorName NOT EQUALS result[n].floorNumber THEN

n++

Add empty Schedule object to result

result[n].floorNumber <- roomsSchedule[i].floorName

END IF

Add roomsSchedule[i] to result[n].rooms

END FOR

RETURN result

FUNCTION **knapsack**(Items, budget)

itemsPrice <- 2D array of size (Items.size() + 1) x (budget + 1) initialized with zeros // O(n \* budget)

FOR i FROM 1 TO Items.size() DO // O(n)

price <- price of Items[i - 1]

value <- value of Items[i - 1]

FOR j FROM 1 TO budget DO // O(budget)

itemsPrice[i][j] <- itemsPrice[i - 1][j]

IF j >= price THEN

itemsPrice[i][j] <- MAX(itemsPrice[i][j], itemsPrice[i - 1][j - price] + value)

END IF

END FOR

END FOR

selected\_items <- empty list

money <- budget

FOR i FROM Items.size() DOWN TO 1 AND money > 0 DO // O(n)

IF itemsPrice[i][money] NOT EQUALS itemsPrice[i - 1][money] THEN

Add Items[i - 1] to selected\_items

money <- money - price of Items[i - 1]

END IF

END FOR

RETURN selected\_items

**What are the factors that affect the performance of the algorithm you developed using the dynamic programming approach?**

1. Number of Rooms: The size of the input, i.e., the number of rooms, directly affects the performance. As the number of rooms increases, the number of subproblems to be solved also increases, leading to longer computation times.
2. Time Complexity of Interval Matching: The match Interval function calculates previous intervals for each room. Its time complexity is O(n\*log(n)), where 𝑛 is the number of rooms. This operation contributes to the overall time complexity of the algorithm.
3. Time Complexity of Optimal Function: The optimal function computes the optimal weight for each room. Its time complexity is O(n), where n is the number of rooms. However, since it's a recursive function, there might be redundant calculations, especially without memorization, which can increase execution time.
4. Sorting: The algorithm sorts the rooms by start time, which has a time complexity of O(n\*log(n)). This sorting operation impacts the overall performance, especially for larger input sizes.
5. Input Data Characteristics: The characteristics of the input data, such as the distribution of start and finish times, can affect the performance. For example, if the intervals are mostly non-overlapping, the algorithm might run faster compared to cases with many overlapping intervals.
6. Memory Usage: The algorithm uses dynamic programming to store intermediate results, which requires additional memory. The space complexity of the algorithm depends on the size of the input and the data structures used for storing solutions to subproblems.
7. Optimization Techniques: The performance can be improved by implementing optimization techniques like memorization to avoid redundant computations and reduce the overall time complexity.

**What are the differences between Dynamic Programming and Greedy Approach? What are the advantages of dynamic programming?**

both Dynamic Programming and Greedy Approach are useful algorithmic techniques, Dynamic Programming shines in situations where finding the optimal solution is essential, even if it comes at the cost of higher time complexity. Greedy Approach, on the other hand, offers simplicity and efficiency but may not always produce the best possible solution.