Analysis of Algorithms II

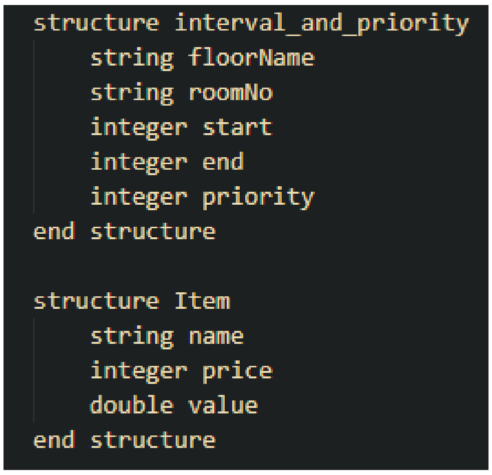
BLG 336E

Project 3 Report

EREN CULHACI
150220763
culhaci22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 14.05.2024

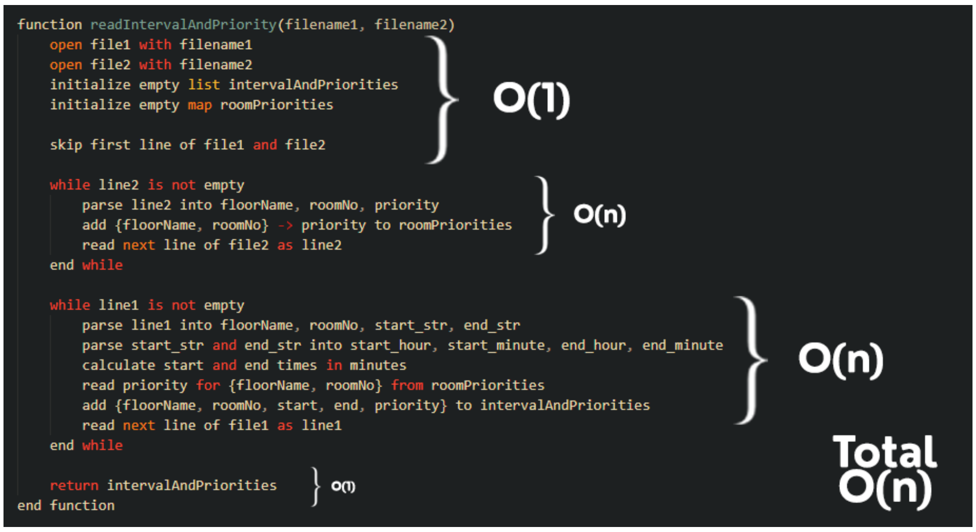
0.1. Implementation Details



```
structure interval_and_priority
    string floorName
    string roomNo
    integer start
    integer end
    integer priority
end structure

structure Item
    string name
    integer price
    double value
end structure
```

Figure 0.1: Structs to store data

- First, I created two structs to store the data coming from the files.
- interval_and_priority keeps priority and interval data including floorName, roomNo, start, end and priority while Item holds name, price and value data.



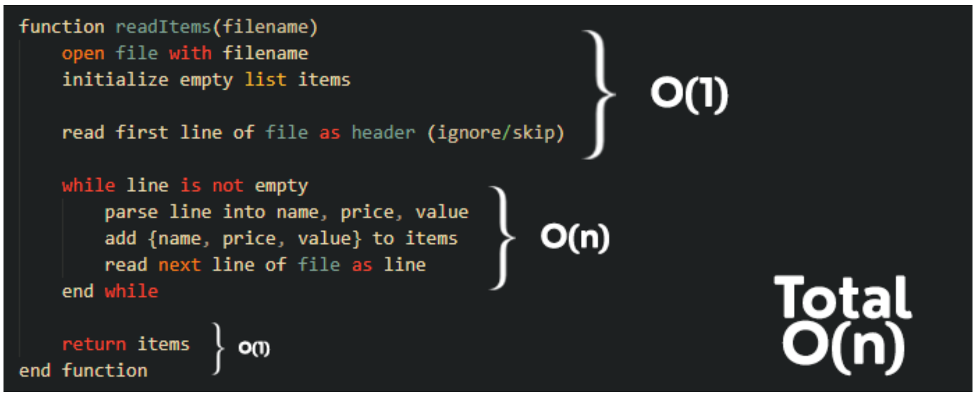
```
function readIntervalAndPriority(filename1, filename2)
    open file1 with filename1
    open file2 with filename2
    initialize empty list intervalAndPriorities
    initialize empty map roomPriorities

    skip first line of file1 and file2

    while line2 is not empty
        parse line2 into floorName, roomNo, priority
        add {floorName, roomNo} -> priority to roomPriorities
        read next line of file2 as line2
    end while

    while line1 is not empty
        parse line1 into floorName, roomNo, start_str, end_str
        parse start_str and end_str into start_hour, start_minute, end_hour, end_minute
        calculate start and end times in minutes
        read priority for {floorName, roomNo} from roomPriorities
        add {floorName, roomNo, start, end, priority} to intervalAndPriorities
        read next line of file1 as line1
    end while

    return intervalAndPriorities
end function
```

Figure 0.2: Read Interval and Priority from files

- Next, I read the data from the priority.txt and room_time_interval.txt files in the same function called readIntervalAndPriority.
- First, I opened the files and initialized a list and a map to map priorities with rooms.
- Then I skipped the headers in the file.
- Then I parsed the priority.txt file and read the priorities, and after that I parsed the room_time_interval.txt file and parsed the time intervals. Also while parsing the

time intervals, I created start_hour, start_minute, end_hour, end_minute integers and parsed the interval time clocks as integers. Then i changed this information into minute metric. So i changed the time clocks into integers and stored it as integers as start and end in interval_and_priority structure. So I mapped priorities with corresponding rooms and time intervals and added all of this information into the vector I created intervalAndPriorities. Then I returned it.

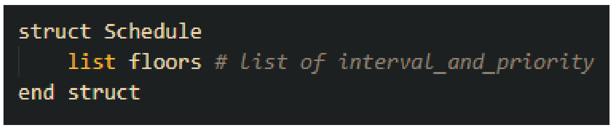
- Since I read all the file, the time complexity of each while section is $O(n)$. Time complexity of this function is also same.



```
function readItems(filename)
    open file with filename
    initialize empty list items

    read first line of file as header (ignore/skip)

    while line is not empty
        parse line into name, price, value
        add {name, price, value} to items
        read next line of file as line
    end while

    return items
end function
```

Figure 0.3: readItems function

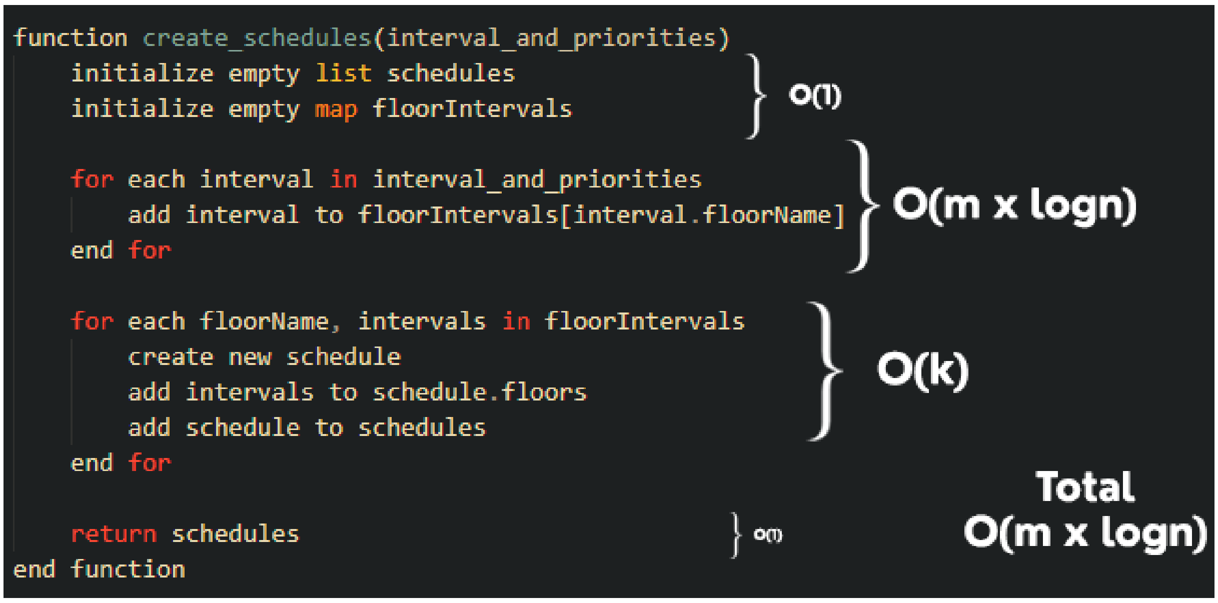
- In readItems function, First I opened the file and initialized and empty Item vector called items.
- Then I skipped the header line.
- Next, I parsed name, price and value data into the vector I created and returned it.
- Time complexity is $O(n)$ since I read all the lines one by one to the end of the file.



```
struct Schedule
    list floors # List of interval_and_priority
end struct
```

Figure 0.4: Schedule Struct

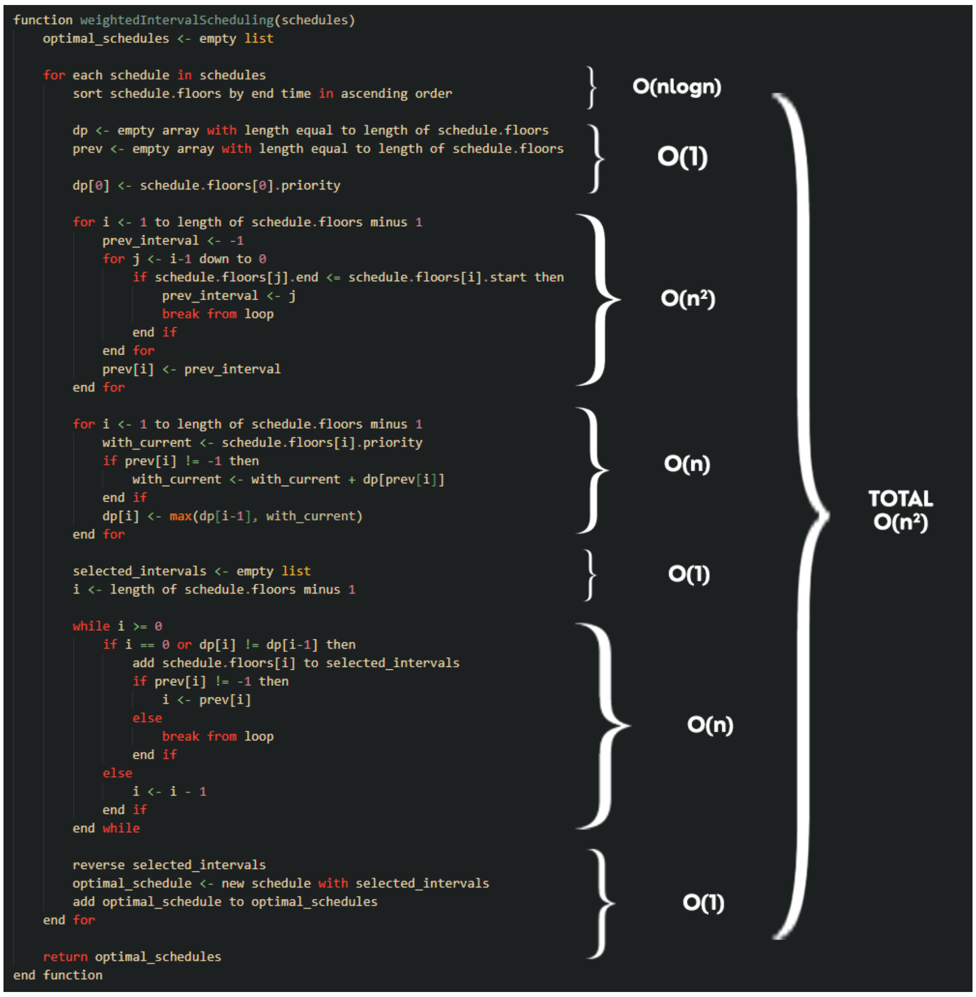
- I created a Schedule struct which holds a interval_and_priority vector called floors. It represents the schedules for floors.



```
function create_schedules(interval_and_priorities)
    initialize empty list schedules
    initialize empty map floorIntervals

    for each interval in interval_and_priorities
        add interval to floorIntervals[interval.floorName]
    end for

    for each floorName, intervals in floorIntervals
        create new schedule
        add intervals to schedule.floors
        add schedule to schedules
    end for

    return schedules
end function
```

Figure 0.5: create_schedules function

- In this function I basically divide the schedules into floors to be solved by weighted interval scheduling in next function.
- In previous step, I created a Schedule struct which includes a interval_and_priority vector, now here I start by creating a Schedule vector called schedules.
- Then I created a map called floorIntervals and filled it.
- Then in the for loops, I basically add all the intervals that are at same floor into same schedule and fill the schedules vector. Then I return that vector.
- In the first loop, I iterate through all elements of interval_and_priorities and insert them into a map floorIntervals based on floor names. Inserting into a map is O(log n) time complexity and then the time complexity of this step is O(m * log n), where ’m’ is the number of intervals and ’n’ is the number of unique floor names. In the second loop, I iterate through each floor in floorIntervals and create a Schedule object for each floor, setting its floors member to the intervals corresponding to that floor. Since this loop iterates through each floor once, and there are ’k’ unique floor names, the time complexity of this step is O(k), where ’k’ is the number of unique floor names.



```
function weightedIntervalScheduling(schedules)
    optimal_schedules <- empty list

    for each schedule in schedules
        sort schedule.floors by end time in ascending order

        dp <- empty array with length equal to length of schedule.floors
        prev <- empty array with length equal to length of schedule.floors

        dp[0] <- schedule.floors[0].priority

        for i <- 1 to length of schedule.floors minus 1
            prev_interval <- -1
            for j <- i-1 down to 0
                if schedule.floors[j].end <= schedule.floors[i].start then
                    prev_interval <- j
                    break from loop
                end if
            end for
            prev[i] <- prev_interval
        end for

        for i <- 1 to length of schedule.floors minus 1
            with_current <- schedule.floors[i].priority
            if prev[i] != -1 then
                with_current <- with_current + dp[prev[i]]
            end if
            dp[i] <- max(dp[i-1], with_current)
        end for

        selected_intervals <- empty list
        i <- length of schedule.floors minus 1

        while i >= 0
            if i == 0 or dp[i] != dp[i-1] then
                add schedule.floors[i] to selected_intervals
                if prev[i] != -1 then
                    i <- prev[i]
                else
                    break from loop
                end if
            else
                i <- i - 1
            end if
        end while

        reverse selected_intervals
        optimal_schedule <- new schedule with selected_intervals
        add optimal_schedule to optimal_schedules
    end for

    return optimal_schedules
end function
```

Figure 0.6: Weighted Interval Scheduling

- In this function, function takes vector of schedules as input parameter. My goal is to select a subset from each schedule so no two interval is overlapped and the priorities are maximized.

- First, for each schedule, I sort the intervals based on their end times in ascending order. This takes $O(nlogn)$ time.

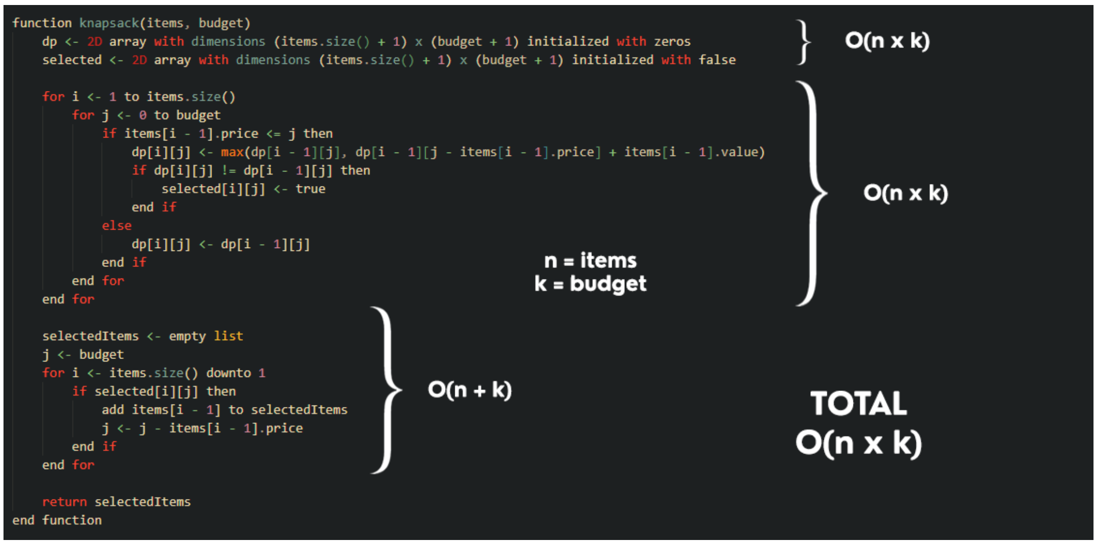
- Then I initialize two arrays dp and prev both sized n where n is the number of intervals in that schedule. dp[i] Represents the maximum total priority achievable by considering intervals up to index i and prev[i] stores the index of the interval just before i that is compatible with interval i.

- And in DP (Dynamic Programming) Process, for each interval i, I find the latest non-overlapping interval prev[i] before i and calculate the max priority can be achieved by including the current interval or excluding it.

- Than after calculating dp, I backtrack to find the selected intervals that gives optimal solution. I start from the last interval and move back, adding the intervals to selected_intervals until the beginning of the schedule or until there is no more intervals.

- Then I return the vector of schedules that contain schedules for floors with non-overlapping intervals with max priority gain.

- For the time complexity, I didn't include the first for loop, because the floor number in this example will be always a small constant, since even Burj Khalifa has at most 163 floors. So if we add the first for loop, the time complexity will be multiplied with m which is the floor count but I haven't included it for the sake of simplicity. But for the other steps, sorting intervals take $O(nlogn)$ time, DP Process involves two nested loops, each running for n iterations, resulting in $O(n^2)$ time complexity and backtrack operation takes $O(n)$ time complexity, so in total, the time complexity is $O(n^2)$. If we take the floor count into consideration, it is $O(m * n^2)$.



```
function knapsack(items, budget)
    dp <- 2D array with dimensions (items.size() + 1) x (budget + 1) initialized with zeros
    selected <- 2D array with dimensions (items.size() + 1) x (budget + 1) initialized with false

    for i <- 1 to items.size()
        for j <- 0 to budget
            if items[i - 1].price <= j then
                dp[i][j] <- max(dp[i - 1][j], dp[i - 1][j - items[i - 1].price] + items[i - 1].value)
                if dp[i][j] != dp[i - 1][j] then
                    selected[i][j] <- true
                end if
            else
                dp[i][j] <- dp[i - 1][j]
            end if
        end for
    end for

    selectedItems <- empty list
    j <- budget
    for i <- items.size() downto 1
        if selected[i][j] then
            add items[i - 1] to selectedItems
            j <- j - items[i - 1].price
        end if
    end for

    return selectedItems
end function
```

Figure 0.7: Knapsack Function

- This function takes two parameters, Item vector and a budget integer.

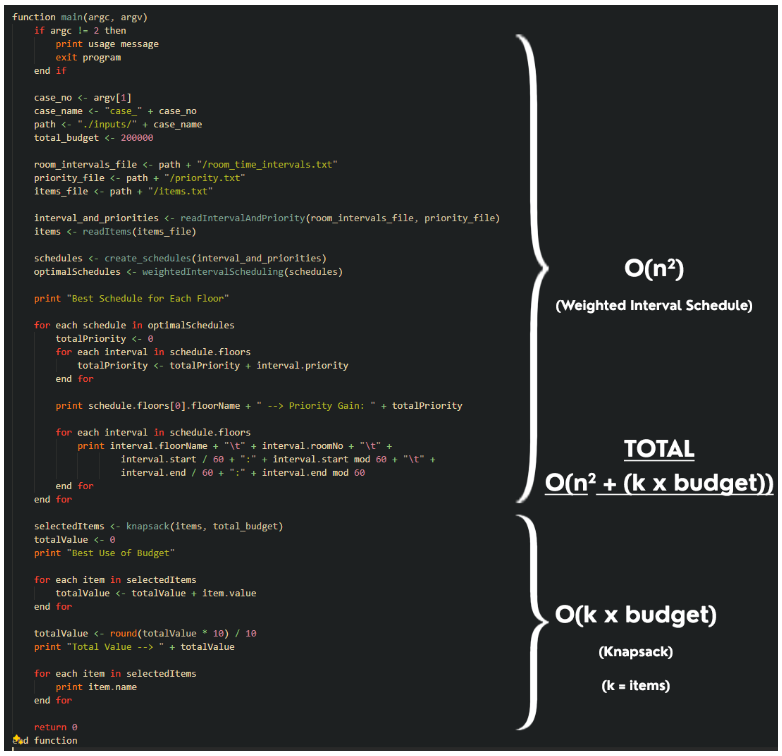
- To initialize Dynamic Programming (DP) I created two 2D vectors called dp and selected with dimensions (items.size() + 1) x (budget + 1). dp[i][j] represents the maximum total value can be achieved with the first i items and a budget of j. selected[i][j] indicates if item i was selected to achieve the maximum value at budget j.

- Then for the DP Process, for each item i from 1 to items.size(), and for each budget j from 0 to budget, I controlled if the current item's price (items[i - 1].price) is less than or equal to current budget.

- If it is, then there are two options:

1. Do not include the current item. For this case, the maximum value stays the same as the maximum value achieved with the previous items at same budget, i.e., dp[i][j] = dp[i - 1][j].
2. Include the current item. In this case, the maximum value is the sum of the value of the current item and the maximum value achieved with the remaining budget after subtracting the price for current item, i.e., dp[i][j] = dp[i - 1][j - items[i - 1].price] + items[i - 1].value.

- Then I update selected[i][j] to true if I chose the current option.
- I update the dp[i][j] with max value between these two options.
- Then in backtracking operation, I try to find the selected items that contribute to max value. I start from the last item and the max budget and go back:
  1. If selected[i][j] is true, then i was selected
  2. Add the selected item to selectedItems vector and decrease its price from the current budget j.
  3. Go to prev. item and the remaining budget.
- Lastly, I return the selected items vector.
- The time complexity is $O(n * budget)$ where n is number of items since we iterate through all items for each budget value. Also since we used 2D vectors for dynamic programming the space complexity is also the same.



```
function main(argc, argv)
    if argc != 2 then
        print usage message
        exit program
    end if

    case_no <- argv[1]
    case_name <- "case_" + case_no
    path <- "./inputs/" + case_name
    total_budget <- 200000

    room_intervals_file <- path + "/room_time_intervals.txt"
    priority_file <- path + "/priority.txt"
    items_file <- path + "/items.txt"

    interval_and_priorities <- readIntervalAndPriority(room_intervals_file, priority_file)
    items <- readItems(items_file)

    schedules <- create_schedules(interval_and_priorities)
    optimalSchedules <- weightedIntervalScheduling(schedules)

    print "Best Schedule for Each Floor"

    for each schedule in optimalSchedules
        totalPriority <- 0
        for each interval in schedule.floors
            totalPriority <- totalPriority + interval.priority
        end for

        print schedule.floors[0].floorName + " --> Priority Gain: " + totalPriority

        for each interval in schedule.floors
            print interval.floorName + "\t" + interval.roomNo + "\t" +
                    interval.start / 60 + ":" + interval.start mod 60 + "\t" +
                    interval.end / 60 + ":" + interval.end mod 60
        end for
    end for

    selectedItems <- knapsack(items, total_budget)
    totalValue <- 0
    print "Best Use of Budget"

    for each item in selectedItems
        totalValue <- totalValue + item.value
    end for

    totalValue <- round(totalValue * 10) / 10
    print "Total Value --> " + totalValue

    for each item in selectedItems
        print item.name
    end for

    return 0
end function
```

Figure 0.8: Main Function

- In t he main function, first I utilize how the command line arguments should work and how the error message will be displayed if it is done in wrong way.
- Then I created the file name strings and initialized the total budget as 200000.
- Then I called readIntervalAndPriority to read the room_time_interval.txt and priority.txt and stored data in vector interval and priorities.
- Then I read the items from the items.txt file by calling readItems function.
- Then I created schedules for each floor by calling create_schedules and solved the weighted interval scheduling problem by calling the function weightedIntervalScheduling().
- Then I calculated total priority gain for each floor and printed all the info as wanted in the calico file.
- Since I turned the time data in clock format into integer format in readIntervalAndPriority function to store the intervals as integers and make it easier to compute, I turned them back into time format in main and printed them in given format.
- Then I solved the knapsack problem by calling the knapsack function and I printed the necessary information.
- To get only 1 value after the decimal point, I used setprecision(1) and for example if I got 36.65 I printed that as 36.7.
- After printing I ended the program, so total time complexity of the main function depends on the functions inside it, so dominant parts are knapsack and weighted time interval, Since knapsack has $O(n * budget)$ and weighted time interval has $O(n^2)$, our main (also our whole program) has $O(n^2 + (k * budget))$ time complexity.

0.2. Questions

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

    - For knapsack, since time complexity is directly affected by item number, the input size affects the performance. Also budget size affects performance of the knapsack algorithms since bigger budget may result in a longer computing time according to our time complexity $O(n * budget)$. Also the values and weights of the items can affect performance, if there is a wide range of values or weights, it may affect the performance of the dynamic programming method. Also size of the dp table is proportional to budget and items, so larger tables require more memory and may slow down our program.

- For weighted interval scheduling, similar to knapsack, the time complexity of the algorithm is heavily influenced by the number of intervals. So lower performance if the number of intervals are higher due to time complexity $O(n^2)$. Also the lengths of the intervals can impact the performance. If there are many short intervals or if intervals have similar lengths, it may have bad impact on the performance of the algorithm. Also the priorities of the intervals can heavily affect the performance of the algorithm, if there is wide range of priorities or priorities are closely clustered, it can have bad impact on performance. Moreover, the complexity of the interval overlapping process can have a bad impact on performance. So highly overlapping intervals needs more computing. Furthermore, since we sort the intervals based on their end times, if they are already sorted or it is expensive to sort, it can impact the performance. Also similar to knapsack, table size for the dp affects memory usage of the program. So it also can have an effect on the performance.

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

   - Greedy algorithms make optimal choices locally in each step and they hope that these choices will end with a globally optimal solution. They don't take into consideration that their decisions can have future consequences or not. Also they do not backtrack the decision they made after it's done. Once a decision is made, it is final, so no reevaluating the choices they made even a new information comes. They are often simpler to implement and understand compared to Dynamic Programming algorithms. They generally involve straightforward iterative processes. However, Dynamic Programming algorithms guarantee finding a globally optimal solution and provide the best possible solution. Also DP algorithms can be applied to broad range of problems, including problems with complex dependencies and subproblems. Moreover, DP efficiently handles problems with overlapping subproblems by avoiding unnecessary calculations with memorization or tabulation. Also, DP algorithms can solve problems which require multiple subproblem solutions and these solutions to subproblems may be reused to solve many others either. So new information affects the decisions.
   - So in briefly, dynamic programming guarantees the globally optimal solution but in greedy methods there is no assurance getting the optimal solution. And while dynamic programming uses memorization, greedy methods never changes its previous decisions. Greedy methods are often faster than dynamic programming since they do not change their earlier decisions and their decisions are final but dynamic programming gives assurance to get the best possible optimal solution.