BLG 336E- Analysis of Algorithms II Homework 2 Report

Özkan Gezmiş 150200033

a) Pseudo-codes:

```
non_conflicting_bs(schedules, i):
    low = 0
    high = i - 1
    while low <= high:
        mid = (high + low) / 2
        if schedules[mid].interval.finish <= schedules[i].interval.start:
            if schedules[mid + 1].interval.finish <= schedules[i].interval.start:
                low = mid + 1
            else:
                return mid
        else:
            high = mid - 1
    return -1

weighted_interval_scheduling(schedules):
    sort schedules by finish time in ascending order
    n = size(schedules)
    initialize dp as an array of size n with all values set to 0
    initialize non_conflicting as an array of size n with all values set to -1

    for i from 1 to n-1:
        non_conflicting[i] = non_conflicting_bs(schedules, i)

    for i from 1 to n-1:
        select = schedule[i].priority
        if non_conflicting[i] != -1:
            select = select + dp[non_conflicting[i]]
        dp[i] = max(select, dp[i-1])

    initialize optimal_schedules as an empty array

    for i from n-1 to 0:
        if dp[i] != dp[i-1]:
            add schedules[i] to optimal_schedules
            i = non_conflicting[i]

    reverse optimal_schedules
    return optimal_schedules
knapsack(items, budget):
```

```
n = size(items)
initialize dp as a 2D array of size (budget+1) x (n+1) with all values set to 0

for x from 1 to budget:
  for j from 1 to n:
      dp[x][j] = dp[x][j-1]
      if items[j-1].weight <= x:
         dp[x][j] = max(dp[x][j], item[j].value + dp[x - items[j-1].weight][j-1]

initialize selected_items as an empty array
set x to budget, j to n

while x > 0 and j > 0:
   if dp[x][j] != dp[x][j-1]:
      add item j to selected_items
      x - = items[j-1].weight
   j--

return selected_items
```

b) Explanation of Functions and Complexity Analysis:

Non-Conflicting_BS:

- Binary search is used to efficiently find the index of the last non-conflicting schedule for a given schedule.
- This optimization reduces the time complexity of finding non-conflicting intervals within the Weighted Interval Scheduling function.

Complexity:

- Time Complexity: O(log n) where n is the number of schedules.
- Space Complexity: O(1) as it uses constant space for storing variables.

Weighted Interval Scheduling:

- Sorting schedules by finish time ensures efficient selection of compatible intervals.
- Dynamic programming is used to maximize priority gain by considering both including and excluding each interval.

Complexity:

- Time Complexity:
  - Sorting: O(n log n), where n is the number of schedules.
  - Binary search for finding non-conflicting intervals for each schedule: O(n log n).
  - Dynamic programming: O(n) for both filling the DP table and backtracking.

- Space Complexity:
  - O(n) for storing schedules, DP table, and auxiliary arrays.

Knapsack:

- Dynamic programming is used to maximize the total value of selected items within the budget constraint.
- Filling the DP table iterates over each item and each possible capacity, considering both including and excluding each item.

Complexity:

- Time Complexity:
  - Filling the DP table: O(n * budget), where n is the number of items and budget is the maximum budget constraint.
  - Backtracking: O(n) to determine the selected items.
- Space Complexity:
  - O(n * budget) for storing the DP table.

main:

- Reads input files, calls the weighted interval scheduling and knapsack functions, and prints the results.
- I used a map instead of a custom struct for Priorities since it simplifies the code significantly. With a map, I can directly associate each floor-room pair with its priority value, eliminating the need for defining and managing a separate data structure for priorities. This approach streamlines the code by providing a more intuitive and concise representation of the relationship between floors, rooms, and their corresponding priorities. Additionally, maps in C++ offer efficient key-value lookup operations, making it easy to retrieve priority values for any given floor-room pair. Overall, using a map enhances readability and reduces complexity, while also ensuring efficient access to priority information.

Complexity:

- Time Complexity:
  - Reading input files: O(m), where m is the size of the input data. The time taken to read the input files depends on the size of the files and the efficiency of the file reading operations.
  - Weighted Interval Scheduling: O(n log n), where n is the number of schedules. This dominates the overall time complexity if the number of schedules is large.
  - Knapsack: O(n * budget), where n is the number of items and budget is the maximum budget constraint. This dominates the overall time complexity if the number of items or the budget is large.
  - Overall time complexity is determined by the most dominant factor among reading input files, weighted interval scheduling, and knapsack algorithms.

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

The performance of the algorithm developed using dynamic programming approach can be influenced by several factors:

1. **Input Size**: The size of the input data directly affects the performance. Larger input sizes may increase the time and space complexity of the algorithm.

2. **Number of Schedules** (Weighted Interval Scheduling): In the context of weighted interval scheduling, the number of schedules plays a significant role. As the number of schedules increases, the time complexity of sorting and finding non-conflicting intervals also increases.

3. **Number of Items** (Knapsack): For the knapsack problem, the number of items affects the time and space complexity of the algorithm. More items require more iterations to fill the dynamic programming table, leading to increased computational overhead.

4. **Budget Constraint** (Knapsack): The maximum budget constraint in the knapsack problem influences the time and space complexity. A higher budget limit may lead to a larger dynamic programming table and increased computation time.

5. **Choice of Algorithms**: The choice of algorithms for specific subproblems within dynamic programming (e.g., sorting algorithms, binary search) can affect the overall performance. Using more efficient algorithms can improve runtime efficiency.

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

Dynamic Programming and Greedy Approach are both algorithmic paradigms used to solve optimization problems, but they differ in their strategies and characteristics.

1. **Optimality:**
   - Dynamic Programming: Provides an optimal solution by considering all possible subproblems and choosing the best solution among them.
   - Greedy Approach: Makes a series of locally optimal choices at each step, aiming to find a globally optimal solution. However, it doesn't always guarantee the optimal solution.
2. **Subproblems**:
   - Dynamic Programming: Solves each subproblem only once and stores the solution to avoid redundant calculations. It relies on the principle of optimal substructure.

- Greedy Approach: Solves each subproblem greedily, making the best choice at each step without considering the consequences of future decisions. It doesn't necessarily involve memorization or reuse of solutions.

3. **Decision Making:**
   - Dynamic Programming: Makes decisions by considering all possible choices and evaluating their consequences. It may involve exploring multiple paths to find the optimal solution.
   - Greedy Approach: Makes decisions by selecting the best available option at each step, without revisiting previous choices. It aims to maximize immediate gain without considering future implications.
4. **Problem Types:**
   - Dynamic Programming: Well-suited for problems with overlapping subproblems and optimal substructure, such as shortest path problems, knapsack problem, and sequence alignment.
   - Greedy Approach: Effective for problems where a locally optimal choice leads to a globally optimal solution, such as minimum spanning tree, Huffman coding, and interval scheduling.

Advantages of Dynamic Programming:

- **Optimality**: Dynamic programming guarantees the optimal solution for problems with optimal substructure.
- **Versatility**: It can be applied to a wide range of problems, including those with complex dependencies and constraints.
- **Efficiency**: By memorizing solutions to subproblems, dynamic programming avoids redundant calculations, leading to efficient computation.
- **Flexibility**: Dynamic programming allows for the incorporation of various optimization criteria and constraints, making it adaptable to different problem scenarios.
- **Scalability**: It can handle larger problem instances by breaking them down into smaller, more manageable subproblems and solving them iteratively.

In summary, both Dynamic Programming and Greedy Approach offer solutions to optimization problems. Dynamic Programming ensures optimality and versatility, though with potentially higher computational complexity due to its exhaustive approach. On the other hand, Greedy Approach provides simplicity and efficiency, but it may sacrifice optimality and could exhibit exponential time complexity in certain scenarios.