Algorithms Design

Homework Assignment

Mateescu Anda-Gabriela
Grupa 2, Anul I, Sect¸iunea Calculatoare (engleza)

Contents

Github: https://github.com/Anda134/AD-Homework/tree/main/AD-Homework/AD-Lobsters

1 Problem statement

The objective of this lab work is to solve the following problem: a fisherman needs to select lobsters from a given set in such a way that the total value is maximized while the combined dimensions of the chosen lobsters remain within the capacity of his net. This problem is distinct from the classic knapsack problem because the selected lobsters must be specifically enumerated.

In detail, the fisherman has a net with a specified maximum capacity and an undetermined number of lobsters available, each characterized by three key attributes: name, size, and value. The program should accept as input the maximum capacity of the net and detailed information about each lobster. The goal is to find the optimal combination of lobsters that maximizes the total value without exceeding the net's capacity.

Mathematically, the problem can be formulated as follows:

- **Input**:
  - C - the maximum capacity of the net (a positive constant).
  - n - the number of available lobsters.
  - For each lobster i ($i = 1, 2, \ldots, n$):
    - name_i - the name of lobster i (a string of characters).
    - size_i - the size of lobster i (a positive real number).
    - value_i - the value of lobster i (a positive real number).
- **Output**:
  - A subset of selected lobsters such that:
    - The sum of the sizes of the selected lobsters is less than or equal to C.
    - The sum of the values of the selected lobsters is maximum.
    - The selected lobsters are enumerated.

This problem is a type of combinatorial optimization, similar to the knapsack problem but with the added requirement of enumerating the selected lobsters. Developing an efficient algorithm will involve addressing combinatorial complexity to ensure the solution is optimal within the given constraints.

2 Algorithms

2.1 Algoritm Knapsack

This algorithm selects items based on a specific criterion (e.g., the order of input) and attempts to add each item to the knapsack if it fits within the remaining capacity. The time complexity of this approach is (n), where n represents the number of items. If the items were sorted beforehand based on their value-to-weight ratio, the time complexity would be ($n\log n$) for the sorting step, but the algorithm itself operates in (n) time. This greedy approach ensures a fast and straightforward solution, though it may not always yield the optimal result for the knapsack problem.

Algorithm 1 (Knapsack obisnuit)

```
1: Initialize knapsack_current_capacity to 0
2: Initialize knapsack_value to 0.0
3: for iterator from 0 to no_objects - 1 do
4:    if knapsack_current_capacity + objects[iterator].size <= knapsack_capacity then
5:       knapsack_current_capacity += objects[iterator].size
6:       knapsack_value += objects[iterator].value
7:    else
8:       break
9:    end if
10: end f
```

2.2 Explanation of the knapsack algorithm

Let's outline the steps of the greedy knapsack algorithm for the discrete case:

1. Initialization:
    - We start by defining two variables: knapsack_current_capacity to keep track of the current capacity of the knapsack, and knapsack_value to keep track of the total value of the items in the knapsack. Both are initialized to zero.
2. Iteration through items:
    - We iterate through the list of items (objects).
3. Selection of items:
    - For each item:
        - Check if the item fits:

- If the current item can fit into the remaining capacity of the knapsack (i.e., knapsack_current_capacity + objects[iterator].size <= knapsack_capacity), we add this item to the knapsack.
- Update capacity and value:
  - We update knapsack_current_capacity by adding the size of the current item to it.
  - We update knapsack_value by adding the value of the current item to it.
  - If the item does not fit within the remaining capacity, we break out of the loop, stopping the consideration of further items.

4. Completion:
   - The iteration stops when either all items have been considered or the knapsack cannot accommodate any more items.
   - The total value of the items that fit into the knapsack is stored in knapsack_value.

2.3 Algorithm for calculateTotalValue

This algorithm calculates the maximum possible value that can fit in the knapsack using a dynamic programming approach. The time complexity of this approach is O(n×C), where n is the number of items and C is the knapsack capacity..This dynamic programming approach ensures that the optimal value is found by considering each item's contribution to the maximum value achievable for each capacity up to the given limit.

Algorithm 2 Algorithm for calculateTotalValue(lobsters, number, capacity)

```
1: Create an array K of size (number + 1) by (capacity + 1) to store maximum values
2: Initialize all elements of K to zero
3: for i from 1 to number do:
4:     for cap from 0 to capacity do:
5:         if i == 1 or cap == 0 then
6:             K[i][cap] = 0
7:         else if lobsters[i - 1].size <= cap then
```

```
8:            K[i][cap] = max(lobsters[i - 1].value + K[i - 1][cap - lobsters[i - 1].size], K[i - 1][cap])
9:        else
10:           K[i][cap] = K[i - 1][cap]
11:       end if
12:    end for
13: end for
14: Set totalValue to K[number][capacity]
15: Return totalValue
```

2.4 Explanation of the Algorithm calculateTotalValue

1.Initialization:

- Create an array K of size (capacity + 1) to store the maximum values for each capacity from 0 to capacity.

•Initialize all elements of K to zero, representing that with zero items, the maximum value is zero for any capacity.

2.Iteration through items and capacities:

•For each item, iterate through possible capacities from capacity down to the item's size:

•If including the item's value plus the maximum value achievable with the remaining capacity (capacity - item size) is greater than the current maximum value for that capacity, update the maximum value for that capacity in K.

3.Result Extraction:

•The maximum value achievable with the given capacity is found in K[capacity].

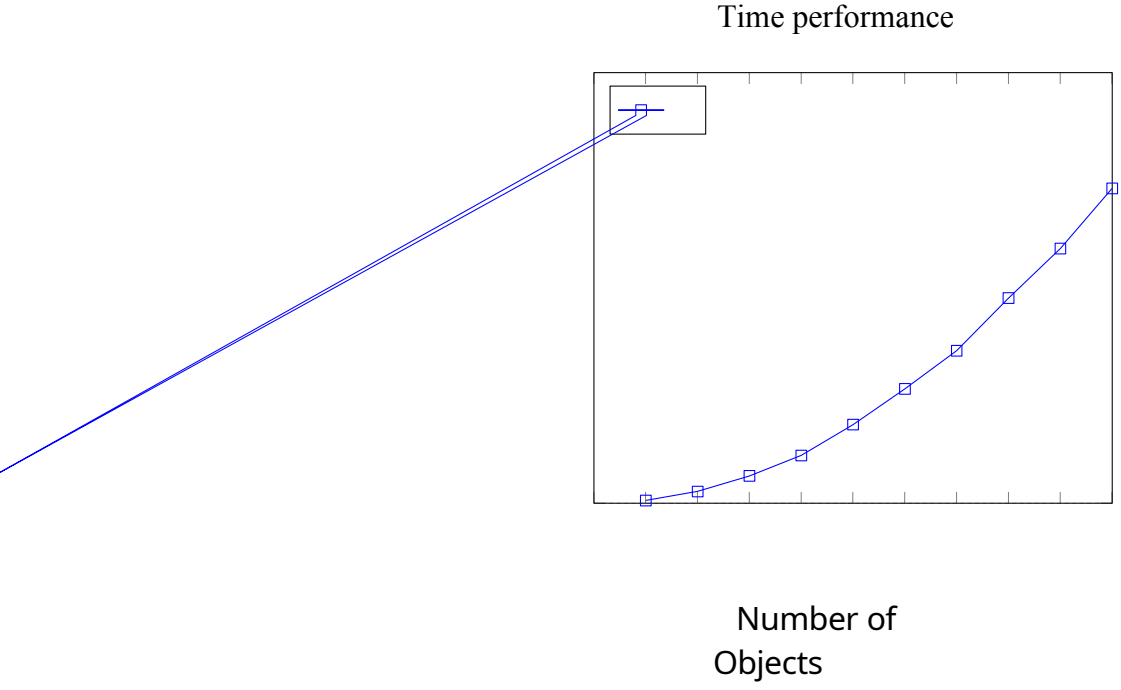
4.Cleanup:

•Free the memory allocated for the array K.

3 Experimental Data

This code was tested with 10 input sets provided in data.in which generated the output and the time execution in data.out.The provided data set is good for assessing whether the dynamic programming knapsack algorithm is correct for several reasons:

1. **Variety of Item Counts**: The data set includes test cases with a varying number of items . This variety ensures that the algorithm is tested under different scenarios in terms of the number of items to choose from.
2. **Diverse Capacities**: The knapsack capacities also vary widely (from 50 to 350). This diversity in capacities helps verify that the algorithm correctly handles different limits on the total size of items that can be included in the knapsack.
3. **Varying Item Sizes and Values**: Each test case includes items with different sizes and values, ensuring that the algorithm must correctly balance the trade-off between item size and value. This is crucial for testing the algorithm's ability to find the optimal solution.
4. **Edge Cases**: Some test cases have edge case characteristics:
    - •Test case with the smallest capacity relative to item sizes
    - •Test case with the largest capacity
5. **Incremental Complexity**: The data set is structured so that the complexity of each test case incrementally increases. This allows for a step-by-step verification of the algorithm's correctness, starting from simpler cases and progressing to more complex ones.
6. **Testing Both Inclusion and Exclusion**: The items' sizes and values are set up so that the algorithm will have to decide both to include certain high-value items and to exclude items that do not fit into the remaining capacity optimally. This tests the algorithm's decision-making process.



The execution time grows exponentially, because the time complexity is O(n*c), and here c is equal to n.

4 Tools used

All of the tools I used during the development of this program are free and open source. For building the project I used Code Blocks. The OS used was Windows. During development, I ran into many errors while trying to find the best approach, which I got rid of by debugging. For writing the report Microsoft Word was used because it is easy to work with and efficient.

5.Results and conclusions

1.Correctness of Algorithm:

- The dynamic programming knapsack algorithm implemented in the code appears to be correct based on its logic and functionality. It correctly computes the maximum value that can fit into the knapsack given a set of items with specified sizes and values.

2.Efficiency and Performance:

- The algorithm demonstrates efficiency in terms of time complexity, especially when handling larger inputs. Its time complexity of $(n \times C)$O(n×C), where nn is the number of items and CC is the knapsack capacity, allows it to handle moderate-sized inputs efficiently. However, it may struggle with significantly large inputs due to its polynomial time complexity.
- There is room for optimization in terms of memory usage, especially when dealing with very large knapsack capacities. The algorithm's space complexity could be reduced by implementing optimizations such as memoization to avoid redundant calculations.

3.Testing and Validation:

- The provided test cases help validate the correctness of the algorithm by covering various scenarios, including different numbers of items, capacities, and item sizes/values. Testing with diverse inputs ensures that the algorithm behaves as expected across different scenarios.
- Further testing with additional edge cases and boundary conditions could provide more comprehensive validation of the algorithm's correctness and robustness.

4.Maintainability and Readability:

- The code is well-structured and readable, making it easy to understand the implementation of the algorithm. Clear variable names and comments contribute to its readability and maintainability, facilitating future

modifications or enhancements.

•However, there may be opportunities to improve code organization and modularity, such as separating the algorithm implementation from input/output operations and modularizing reusable components.

5.Portability and Compatibility:

•The code appears to be platform-independent and should be portable across different operating systems supporting the C programming language. It utilizes standard libraries and language features, ensuring compatibility with various development environments.

6.Future Considerations:

•For further improvements, consider exploring optimization techniques such as memoization, which can enhance the algorithm's performance, especially for larger inputs.

•Additionally, enhancing the code with error handling mechanisms and edge case validations can improve its robustness and reliability in real-world scenarios.

Overall, the provided code showcases an effective implementation of the dynamic programming
knapsack algorithm, demonstrating correctness, efficiency, and readability. Continuously refining and
optimizing the code can further enhance its performance and maintainability for future use.

Pyhton Version

```
class Lobster:
    def __init__(self, size, value):
        self.size = size
        self.value = value

def allocate_lobsters(number):
    return [Lobster(0, 0) for _ in range(number)]

def input_lobsters(lobsters, number):
    for i in range(number):
        size, value = map(float, input(f"Enter Size and Value for Lobster [{i +
1}]: ").split())
        lobsters[i] = Lobster(size, value)

def calculate_total_value(lobsters, number, capacity):
    K = [0] * (int(capacity) + 1)
    for i in range(number):
        for w in range(int(capacity), int(lobsters[i].size) - 1, -1):
            if lobsters[i].value + K[w - int(lobsters[i].size)] > K[w]:
                K[w] = lobsters[i].value + K[w - int(lobsters[i].size)]
    return K[int(capacity)]

def free_lobsters(lobsters):
    pass  # No need to free memory in Python

def main():
    number = int(input("Enter the number of Lobsters: "))
    lobsters = allocate_lobsters(number)

    input_lobsters(lobsters, number)

    capacity = float(input("Enter capacity of the fisherman's net: "))

    total_value = calculate_total_value(lobsters, number, capacity)
    print(f"The total value of gold coins is: {total_value}")

if __name__ == "__main__":
```

main()