PRACTICAL ASSIGNMENT - MARKING REPORT

1. PERSONAL DATA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Group number :** | | | | |
| No | Name | ID | Programme | Total Marks |
| 1. | Sam Heng Zi Sam | 2106458 | SE |  |
| 2. | Lee Lok Yee | 2200488 | SE |  |
| 3. | Koh Win Sing | 2002472 | SE |  |
| 4. | Lee Boon Hao | 2106860 | SE |  |

1. SUBMISSION STATUS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No soft copy/ Upload wrong file(s) | Late submission of softcopy | No hardcopy | Late submission of hardcopy | No issue |
|  |  |  |  |  |

1. COMPILATION AND RUNNING

|  |  |  |
| --- | --- | --- |
| Does not compile/Bytecode & batch file do not work | Compile but no output/ wrong output/ run-time error | Compile and produce output |
|  |  |  |

1. PRESENTATION OF SOURCE CODES(6%)
2. Indent Style (3%) Poor Inconsistent Good
3. Identifier names (3%) Poor choice Meaningful Meaningful and good naming convention
4. PROGRAM COMPONENT (44% + 6%)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Program Components | Missing/ Does not work | Major errors | Minor errors | Not robust | No issue/ Excellent design | Max marks | Marks obtained |
| Framework Design (Use of interfaces and abstract classes) |  |  |  |  |  | 5 |  |
| Classes for storing objects (data structures/containers) |  |  |  |  |  | 10 |  |
| Knapsack Algorithms |  |  |  |  |  | 12 |  |
| Test program (main program, set of given items, weight and value) |  |  |  |  |  | 12 |  |
| Exception and error handling |  |  |  |  |  | 5 |  |
| Presentation of source codes |  |  |  |  |  | 6 |  |
|  |  |  |  |  | Total | 50 |  |

1. REPORT AND OTHER COMPONENT (50%)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Components | Missing | Poor | Average | Good | Excellent | Max marks | Marks obtained |
| Design (data structures and algorithms) and discussion (efficiency and complexities) |  |  |  |  |  | 20 |  |
| Flowchart |  |  |  |  |  | 5 |  |
| UML Diagram |  |  |  |  |  | 5 |  |
| Sample input and test cases |  |  |  |  |  | 5 |  |
| Screenshots |  |  |  |  |  | 5 |  |
|  |  |  |  |  | Total | 40 |  |
| Individual presentation (Involvement, language, confident, preparedness, attitude) |  |  |  |  |  | 10 |  |
| Student 1: |  |  |  |  |  |  |  |
| Student 2: |  |  |  |  |  |  |  |
| Student 3: |  |  |  |  |  |  |  |
| Student 4: |  |  |  |  |  |  |  |
|  |  |  |  |  | Total | 50 |  |

**Design (data structures and algorithms)**

The knapsack problem is a combinatorial optimization problem that involves efficiently allocating resources and managing inventory to maximize returns. The issue is discussed and summarized below:

Assume you plan to go for traveling onboard. You've made a list of items you'd like to bring and packed it in your luggage. The list of items i contains a variety of quantities n, each with its own weight w and value v. You must first select the weight (kg) of the luggage before carrying and fitting the item into it. Each item has a weight, and your luggage is limited to carrying no more than the luggage weight you selected. This is to determine how many of each item to fit into the luggage so that the total weight is less than or equal to the luggage's maximum capacity and the total value of the item is as large as possible. The diagram below depicts a list of items along with their weight, value, and quantity:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No** | **Item** | **Weight(kg)** | **Value** | **Quantities** |
| 1 | laptop | 3 | 3 | 1 |
| 2 | clothes | 1 | 5 | 4 |
| 3 | pillow | 2 | 6 | 2 |
| 4 | shoes | 4 | 5 | 2 |
| 5 | tablet | 2 | 3 | 1 |
| 6 | blanket | 7 | 5 | 1 |
| 7 | towel | 3 | 2 | 2 |
| 8 | book | 5 | 3 | 1 |
| 9 | snacks | 1 | 2 | 2 |
| 10 | toiletries | 2 | 2 | 2 |

We proposed two solutions to solve the knapsack problem: the Greedy algorithm and the Memoization technique, which generate optimal solutions in determining the number of each item to fit into the luggage so that the total weight is less than or equal to the maximum capacity and the total value is as large as possible.

It is a simple way to solve a problem for Greedy algorithms by selecting the best and optimal option among the available options at each current step. To demonstrate how the Greedy algorithm works, consider the following strategies:

1. Initialize the luggage as empty.
2. Sort the items by value per weight ratio in descending order.
3. Select and put items into the luggage as much as possible with the highest value per weight ratio first.
4. Update the current state.
5. Repeat step 2 and 3 for the selection until a solution is reached and found.

By applying Greedy algorithm, it has low complexity and is easy to understand which simplifies the selection and making optimal choices which will increase the efficiency. The algorithm also provides efficiency in solving problems with large sets of data in a short time and does not require large storage to store results. Aside from that, the algorithm can be used to find approximate solutions to problems that are difficult to solve precisely by providing near-optimal solutions. The algorithm also can be adapted to different variations of a problem by changing the selection criteria. This adaptability enables them to address a wide range of problem constraints and requirements that achieve flexibility. In such cases, the Greedy algorithm is useful because it provides simplicity, efficiency, approximation, and adaptability, all of which lead to a globally optimal solution.

However, it has a few notable limitations on the algorithm itself. The Greedy algorithm does not guarantee that it will always lead to the globally optimal solution in all cases, this means that they may not always result in the best overall solution. The other limitation is that the algorithm does not reverse or backtrack any decision once it is made. This will lead to the production of inaccurate or incorrect results.

Memoization technique is another method to solve the knapsack problem by storing the results of the computation in the table and returning the result when the same computation occurs again for reusation and avoids recomputing the redundant subproblem repeatedly. To demonstrate how the Memoization technique works, consider the following strategies:

1. Construct an array or table to store the results of the subproblem.
2. Check to see if the results of the arguments are already in the table. Return the result if it is. If not, continue with the calculation and store it in the table for future use.
3. Reuse the results from the table if the function is called with the same arguments instead of recalculating the subproblem.

By implementing the Memoization technique, it can avoid duplicated result computation and the same input by ensuring the subproblem is solved only once. Other than that, it can increase efficiency in the way of retrieving and reusing the results instead of recalculating the subproblem and hence improve the performance of the code. Overall, knowing the Memoization technique allows efficiency and improves the performance while avoiding duplicated result computation will result in a globally optimal solution.

In such a scenario, this technique also contains some disadvantages. It utilizes additional memory to store the results of the computed subproblems which will lead to excessive memory usage. In addition, when the memory usage exceeds or the results are too large, it will cause memory overflow as well. Besides that, using Memoization technique will increase code complexity by managing the table, storing, and retrieving the result.

In conclusion, both data structures suit themselves balanced in providing solutions for the knapsack problem. While excel in terms of simplicity and speed, Greedy algorithm is a better choice if considering the time factor to solve a large sets of data . As for the Memoization Technique, it is a better choice if consider the reusable factor to get rid of redundant subproblems with overlapping solutions. We can effectively overcome a wide range of constraints and problems by using these two approaches.

**Discussion (complexities and efficiency)**

Time complexity of Greedy algorithm

Greedy algorithm employs a straightforward and effective strategy. It's a good option when you need a quick solution that's close to optimal for the Knapsack problem.

Sorting: Sorting items according to the value per weight in descending order takes O(n log n) time, where n: number of items. It enables the algorithm to prioritize the items that provide the most value for their weight.

Iteration: Iterate through the sorted items one by one takes O(n) time.

Hence, the overall time complexity of the Greedy algorithm is O(n log n),

Space complexity of Greedy algorithm

The space complexity may differ based on the data structures and specific implementation.

Space for storing the sorted items: Require space to store size of the item, where n: number of items.

Overall, the space complexity of the Greedy algorithm is O(n) .

Efficiency Of Greedy algorithm in Different Scenarios

The Greedy algorithm is efficient by choosing the optimal option from a set of viable solution. There are a few scenarios where Greedy algorithm is used efficiently.

1. Fractional knapsack problem: Fill the knapsack with fractional amount of item to maximize the value or the weight capacity of the knapsack.
2. Coin change problem: Find the smallest number of coins required to make a given amount of change given a set of coins with different denominations.

Time complexity of Memoization Technique

The problem to which a memoized algorithm is applied determines its time complexity.

If no results are memoized when solving problems with n subproblems, the algorithm may have a time complexity of O(2n). The algorithm, on the other hand, effectively reduces time complexity by avoiding redundant computations. Because no value is called more than once, the time complexity of memory is O(N) rather than O(2N), where n is the number of numbers generated.

Hence, the time complexity of Memoization Technique is O(n).

Space complexity of Memoization Technique

The space occupied by the memoization table (cache) determines the overall space complexity. Using memorization, each value will be calculated only once and no additional memory space is required. Hence, the space complexity of Memoization Technique is O(n), where n: number of unique subproblems.

Efficiency Of Memoization Technique in Different Scenarios

Memoization is a technique which the result is stored in the table and reuse the result if the argument is the same. Here are a few scenarios where Memoization Technique is used efficiently:

1. Recursive Calculations: Memoization can greatly speed up recursive calculations. The time complexity of calculating factorial numbers, for example, without memoization can be exponential (O(2n)). Memorization, on the other hand, solves each subproblem only once, reducing the time complexity to linear (O(n)).

Overall, the Greedy algorithm is more efficient in terms of time complexity than the Memoization Technique due to the o(nlogn) time, resulting in a quick and simple decision in making the optimal choice. Aside from that, the Greedy algorithm is more efficient in terms of space complexity than the Memoization Technique because it does not require large amounts of storage. Furthermore, the Greedy algorithm is faster than the Memoization Technique because it can process a large set of data at once in a short period of time. If you need a quick solution and optimality is not critical, a Greedy algorithm may be more efficient. Despite its potential for greater time and space complexity, Memoization is frequently the better choice for problems where optimality is critical or when there are overlapping subproblems. The choice between these two methods depends on the requirement and characteristics of the problem.

**Flowchart**

Greedy Algorithm Flowchart:

public LinkedList<Item> solve (PriorityQueue <Item> queue, int capacity)

A screenshot of a computer screen

Description automatically generated

public int computeWeight(LinkedList<Item> list)

A screenshot of a computer

Description automatically generated

public int computeValue(LinkedList<Item> list)

A screenshot of a computer

Description automatically generated

Memoization Technique Flowchart:

public Memoization()

A close-up of a memory

Description automatically generated

public LinkedList<Item> solve(PriorityQueue<Item> queue, int capacity)

A black background with white rectangles

Description automatically generated

public LinkedList<Item> solve(PriorityQueue<Item> queue, int totalCapacity, int capacityLeft, LinkedList<Item> knapsack)

A black screen with white rectangles

Description automatically generated

public int computeWeight(LinkedList<Item> list)

A screenshot of a computer

Description automatically generated

public int computeValue(LinkedList<Item> list)

A screenshot of a computer

Description automatically generated

public void displayMemory()

A black and white background with white squares

Description automatically generated

**UML Diagram ( Class Diagram)**

A diagram of a computer

Description automatically generated with medium confidence

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Project Name:** | Knapsack problem | **Test Designed by:** |  |  | | | | | |
| **Module Name:** | Implement Greedy algorithm and Memoization technique | **Test Designed date:** |  |
| **Release Version:** | | **Test Executed by:** |  |
|  |  | **Test Execution date:** |  |
|  |  | | | | | | | | |
| **Pre-condition** | The items comes with its own weight, value and quantity | | | | | | | |  |
| **Dependencies:** | none | | | | | | | |
| **Test Priority** | High | | | | | | | |
|  |  | | | | | | | | |
| **Test Case#** | **Test Title** | **Test Summary** | **Test Steps** | **Test Data** | **Expected Result** | **Post-condition** | **Actual Result** | **Status** | **Notes** |
| 1 | To verify the correctness and effectiveness of solving knapsack problem. | If Greedy Algorithm is implemented. | User enter the maximum capacity(kg) of the luggage to calculate the total value, quantity, weight and time spent. | Enter the maximum capacity(kg) of your luggage: 0 | No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of 0kg has weight of 0kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. | No of items found in the knapsack is shown.  Total value is shown.  Total weight is shown.  The time spent to solve the knapsack problem using Greedy Algorithm is shown. | No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of 0kg has weight of 0kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. | Pass |  |
| Enter the maximum capacity(kg) of your luggage: -1 | No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of -1kg has weight of 0kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. | No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of -1kg has weight of 0kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. |
| Enter the maximum capacity(kg) of your luggage: 10 | Items include in knapsack:  1. clothes  2. clothes  3. clothes  4. clothes  5. pillow  6. pillow  7. snacks  8. snacks  Total value are 36.  Knapsack with maximum of 10kg has weight of 10kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. | Items include in knapsack:  1. clothes  2. clothes  3. clothes  4. clothes  5. pillow  6. pillow  7. snacks  8. snacks  Total value are 36.  Knapsack with maximum of 10kg has weight of 10kg.  The time spent to solve the knapsack problem using Greedy Algorithm is 0ms. |
| 2 | To verify the correctness and effectiveness of solving knapsack problem. | If Memoization Technique is implemented. | User enter the maximum capacity(kg) of the luggage to calculate the total value, quantity, weight and time spent. | Enter the maximum capacity(kg) of your luggage: 0 | No solution is stored in the memory yet.  No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of 0kg has weight of 0kg.  The time spent to solve the knapsack problem using Memoization is 0ms. | Items added, current capacity, capacity left and solution for knapsack with its respectively weight is shown.  No of items found in the knapsack is shown.  Total value is shown.  Total weight is shown.  The time spent to solve the knapsack problem using Memoization is shown. | No solution is stored in the memory yet.  No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of 0kg has weight of 0kg.  The time spent to solve the knapsack problem using Memoization is 0ms. | Pass |  |
| Enter the maximum capacity(kg) of your luggage: -5 | No solution is stored in the memory yet.  No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of -5kg has weight of 0kg.  The time spent to solve the knapsack problem using Memoization is 0ms. | No solution is stored in the memory yet.  No items are found in the knapsack.  Total value are 0.  Knapsack with maximum of -5kg has weight of 0kg.  The time spent to solve the knapsack problem using Memoization is 0ms. |
| Enter the maximum capacity(kg) of your luggage: 15 | No solution is stored in the memory yet.  Item: clothes(1kg)  Added.  Current capacity: 1kg  Capacity left: 14kg  Solution for knapsack with 1kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 2kg  Capacity left: 13kg  Solution for knapsack with 2kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 3kg  Capacity left: 12kg  Solution for knapsack with 3kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 4kg  Capacity left: 11kg  Solution for knapsack with 4kg is added into memory table.  Item: pillow(2kg)  Added.  Current capacity: 6kg  Capacity left: 9kg  Solution for knapsack with 6kg is added into memory table.  Item: pillow(2kg)  Added.  Current capacity: 8kg  Capacity left: 7kg  Solution for knapsack with 8kg is added into memory table.  Item: snacks(1kg)  Added.  Current capacity: 9kg  Capacity left: 6kg  Solution for knapsack with 9kg is added into memory table.  Item: snacks(1kg)  Added.  Current capacity: 10kg  Capacity left: 5kg  Solution for knapsack with 10kg is added into memory table.  Item: tablet(2kg)  Added.  Current capacity: 12kg  Capacity left: 3kg  Solution for knapsack with 12kg is added into memory table.  Item: shoes(4kg)  NOT added.  Item: shoes(4kg)  NOT added.  Item: laptop(3kg)  Added.  Current capacity: 15kg  Capacity left: 0kg  Solution for knapsack with 15kg is added into memory table.  Solution for knapsack with capacity of 15kg generated.  Items include in knapsack:  1. clothes  2. clothes  3. clothes  4. clothes  5. pillow  6. pillow  7. snacks  8. snacks  9. tablet  10. laptop  Total value are 42.  Knapsack with maximum of 15kg has weight of 15kg.  The time spent to solve the knapsack problem using Memoization is 2ms. | No solution is stored in the memory yet.  Item: clothes(1kg)  Added.  Current capacity: 1kg  Capacity left: 14kg  Solution for knapsack with 1kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 2kg  Capacity left: 13kg  Solution for knapsack with 2kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 3kg  Capacity left: 12kg  Solution for knapsack with 3kg is added into memory table.  Item: clothes(1kg)  Added.  Current capacity: 4kg  Capacity left: 11kg  Solution for knapsack with 4kg is added into memory table.  Item: pillow(2kg)  Added.  Current capacity: 6kg  Capacity left: 9kg  Solution for knapsack with 6kg is added into memory table.  Item: pillow(2kg)  Added.  Current capacity: 8kg  Capacity left: 7kg  Solution for knapsack with 8kg is added into memory table.  Item: snacks(1kg)  Added.  Current capacity: 9kg  Capacity left: 6kg  Solution for knapsack with 9kg is added into memory table.  Item: snacks(1kg)  Added.  Current capacity: 10kg  Capacity left: 5kg  Solution for knapsack with 10kg is added into memory table.  Item: tablet(2kg)  Added.  Current capacity: 12kg  Capacity left: 3kg  Solution for knapsack with 12kg is added into memory table.  Item: shoes(4kg)  NOT added.  Item: shoes(4kg)  NOT added.  Item: laptop(3kg)  Added.  Current capacity: 15kg  Capacity left: 0kg  Solution for knapsack with 15kg is added into memory table.  Solution for knapsack with capacity of 15kg generated.  Items include in knapsack:  1. clothes  2. clothes  3. clothes  4. clothes  5. pillow  6. pillow  7. snacks  8. snacks  9. tablet  10. laptop  Total value are 42.  Knapsack with maximum of 15kg has weight of 15kg.  The time spent to solve the knapsack problem using Memoization is 2ms. |

**Sample outputs of the program**

1. **Data input (input.txt)**

A screenshot of a computer program

Description automatically generated

**A screenshot of a computer

Description automatically generated**

1. **Program output for each test case 1( Greedy Algorithm)**

**A screenshot of a computer program

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

1. **Program output for each test case 2( Memoization Technique)**



**A screenshot of a computer program

Description automatically generated**

****

**A black screen with white text

Description automatically generated**

****

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**Print out of the Java program**

**GreedyMethod.java**

import java.util.Iterator;

import java.util.LinkedList;

import java.util.PriorityQueue;

public class GreedyMethod implements Algorithm{

public LinkedList<Item> solve (PriorityQueue <Item> queue, int capacity){

//Create a new priority queue and sort the items according to the descending value-per-weight ratio

PriorityQueue <Item> queue2 = new PriorityQueue<Item> ();

LinkedList<Item> knapsackSet = new LinkedList<Item>();

//Add the items in queue to the new priority queue

while(queue.size() > 0)

queue2.add(queue.remove());

while (queue2.size() > 0 && capacity > 0) {

Item currentItem = queue2.poll();

if (currentItem.getWeight() <= capacity) {

capacity -= currentItem.getWeight();

knapsackSet.add(currentItem);

}

}

return knapsackSet;

}

public int computeWeight(LinkedList<Item> list)

{

if(list!=null)

{

Iterator<Item> iterator = list.iterator();

int sum=0;

while(iterator.hasNext())

sum+=iterator.next().getWeight();

return sum;

}

else return 0; //return 0 when list == null

}

public int computeValue(LinkedList<Item> list)

{

if(list!=null)

{

Iterator<Item> iterator = list.iterator();

int sum=0;

while(iterator.hasNext())

sum+=iterator.next().getValue();

return sum;

}

else return 0;

}

}

**Memoization.java**

import java.util.Iterator;

import java.util.LinkedList;

import java.util.PriorityQueue;

import java.util.SortedMap;

import java.util.TreeMap;

public class Memoization implements Algorithm{

private SortedMap<Integer,LinkedList<Item>> memory;

public Memoization()

{

memory=new TreeMap<Integer,LinkedList<Item>>();

memory.put(0, null);

}

public LinkedList<Item> solve(PriorityQueue<Item> queue, int capacity)

{

displayMemory();

LinkedList<Item> knapsack=new LinkedList<Item>();

if(capacity==0)

{

return knapsack;

}

else if(capacity>memory.lastKey())

{

if(memory.lastKey()==0)

return solve(queue,capacity,capacity,knapsack);

else

{

PriorityQueue<Item> queueToCompare = new PriorityQueue<Item>(queue);

LinkedList<Item> result = memory.get(memory.lastKey());

System.out.println("Solution for knapsack with capacity = "+memory.lastKey()+" is found in the memory table.");

System.out.println("Items added into the knapsack: ");

for(Item item:result)

{

Item i=queueToCompare.poll();

if(item.equals(i))

{

queue.remove(i);

knapsack.add(i);

System.out.println(i.getName()+"("+i.getWeight()+"kg)");

}

}

System.out.println();

int currentCapacity=computeWeight(knapsack);

System.out.println("Current capacity: "+currentCapacity+"kg");

int capacityLeft=capacity-currentCapacity;

System.out.println("Capacity left: "+capacityLeft+"kg");

System.out.println();

return solve(queue,capacity,capacityLeft,knapsack);

}

}

else

{

int memoIndex=capacity;

while(!memory.containsKey(memoIndex))

memoIndex--;

LinkedList<Item> results= new LinkedList<Item>(memory.get(memoIndex));

Iterator<Item> iterator= results.iterator();

System.out.println("Solution for knapsack with capacity = "+memoIndex+" is found in the memory table.");

System.out.println("Items added into the knapsack: ");

while(iterator.hasNext())

{

Item i=iterator.next();

queue.remove(i);

knapsack.add(i);

System.out.println(i.getName()+"("+i.getWeight()+"kg)");

}

System.out.println();

int currentCapacity=computeWeight(knapsack);

System.out.println("Current capacity: "+currentCapacity+"kg");

int capacityLeft=capacity-currentCapacity;

System.out.println("Capacity left: "+capacityLeft+"kg");

System.out.println();

return solve(queue,capacity,capacityLeft,knapsack);

}

}

//recursive helper method for adding items

public LinkedList<Item> solve(PriorityQueue<Item> queue, int totalCapacity, int capacityLeft, LinkedList<Item> knapsack)

{

if(capacityLeft==0)

{

System.out.println("Solution for knapsack with capacity of "+totalCapacity+"kg generated.");

System.out.println();

return knapsack;

}

//perform recursive calculation

else

{

if(!queue.isEmpty())

{

Item i=queue.peek();

System.out.println("Item: "+i.getName()+"("+i.getWeight()+"kg)");

if(i.getWeight()<=capacityLeft)

{

queue.poll();

knapsack.add(i);

System.out.println("Added.");

capacityLeft-=i.getWeight();

int memoIndex=totalCapacity-capacityLeft;

System.out.println("Current capacity: "+memoIndex+"kg");

System.out.println("Capacity left: "+capacityLeft+"kg");

if(!memory.containsKey(memoIndex))

{

LinkedList<Item> result=new LinkedList<Item>(knapsack);

memory.put(memoIndex, result); //add the result into memory for later use

System.out.println("Solution for knapsack with "+memoIndex+"kg is added into memory table.");

}

System.out.println();

return solve(queue,totalCapacity,capacityLeft,knapsack);

}

else

{

queue.poll();

System.out.println("NOT added.");

System.out.println();

return solve(queue,totalCapacity,capacityLeft,knapsack);

}

}

else

{

System.out.println("Knapsack solution for "+totalCapacity+"kg generated.");

if(!memory.containsKey(totalCapacity))

{

LinkedList<Item> result=new LinkedList<Item>(knapsack);

memory.put(totalCapacity, result); //add the result into memory for later use

System.out.println("Solution for knapsack with "+totalCapacity+"kg is added into memory table.");

}

System.out.println();

return knapsack;

}

}

}

public int computeWeight(LinkedList<Item> list)

{

if(list!=null)

{

Iterator<Item> iterator = list.iterator();

int sum=0;

while(iterator.hasNext())

sum+=iterator.next().getWeight();

return sum;

}

else return 0; //return 0 when list == null

}

public int computeValue(LinkedList<Item> list)

{

if(list!=null)

{

Iterator<Item> iterator = list.iterator();

int sum=0;

while(iterator.hasNext())

sum+=iterator.next().getValue();

return sum;

}

else return 0;

}

public void displayMemory()

{

if(memory.lastKey()==0)

{

System.out.println("No solution is stored in the memory yet.");

System.out.println();

}

else

{

System.out.println("------------------------------");

System.out.println(" Solutions stored in memory ");

System.out.println("------------------------------");

for(int i=1;i<=memory.lastKey();i++)

{

if(memory.containsKey(i))

{

Iterator<Item> iterator=memory.get(i).iterator();

System.out.print(i+"kg knapsack: ");

while(iterator.hasNext())

{

System.out.print(iterator.next().getName()+", ");

}

System.out.println();

}

}

System.out.println();

}

}

}

**Algorithm.java**

import java.util.LinkedList;

import java.util.PriorityQueue;

public interface Algorithm {

public LinkedList<Item> solve(PriorityQueue<Item> queue, int capacity);

public int computeWeight(LinkedList<Item> list);

public int computeValue(LinkedList<Item> list);

}

**AlgorithmTest.java**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.LinkedList;

import java.util.PriorityQueue;

import java.util.Scanner;

public class AlgorithmTest {

public static void main(String[] args)

{

PriorityQueue <Item> queue = new PriorityQueue<Item>();

Algorithm gm = new GreedyMethod();

Algorithm memo=new Memoization();

Scanner scanner=new Scanner(System.in);

String filename = "C:\\Users\\Joanne Lee\\Downloads\\Data Structures Assignment\\input.txt";

String data;

Scanner input;

try {

input = new Scanner(new File(filename));

while(input.hasNext()) {

data = input.nextLine();

String[] words = data.split(",");

Item item=new Item(words[0], Integer.parseInt(words[1]), Integer.parseInt(words[2]));

for(int i=0;i<Integer.parseInt(words[3]);i++)

queue.offer(item);

}

} catch (FileNotFoundException e) {

System.out.println("ERROR : input.txt file is missing.");

e.printStackTrace();

} catch (NumberFormatException e) {

System.out.println("ERROR : input.txt file data have incorrect format");

e.printStackTrace();

} catch (IllegalArgumentException e) {

System.out.println("ERROR : Argument is invalid");

e.printStackTrace();

}

printAllItems(queue);

int choice;

do

{

//queueForGreedy and queueForDynamic have the same items

PriorityQueue <Item> queueForGreedy = new PriorityQueue<Item>(queue);

PriorityQueue <Item> queueForMemoization = new PriorityQueue<Item>(queue);

System.out.println("--------------------------------------------------------------------------------------");

System.out.println();

//Requires maximum capacity from user

System.out.print("Enter the maximum capacity(kg) of your luggage: ");

int knapsackCapacity=scanner.nextInt();

System.out.println();

System.out.println("Using greedy algorithm: ");

System.out.println();

long start = System.currentTimeMillis();

LinkedList<Item> knapsack = gm.solve(queueForGreedy, knapsackCapacity);

long end = System.currentTimeMillis();

printKnapsackItem(knapsack);

System.out.println();

System.out.println("Total value are " + gm.computeValue(knapsack)+".");

System.out.println("Knapsack with maximum of "+knapsackCapacity+"kg has weight of " + gm.computeWeight(knapsack) + "kg.");

System.out.println();

System.out.println("The time spent to solve the knapsack problem using Greedy Algorithm is "+calculateTimeSpent(start,end)+"ms.");

System.out.println("--------------------------------------------------------------------------------------");

System.out.println();

System.out.println("Using Memoization algorithm: ");

System.out.println();

start=System.currentTimeMillis();

knapsack = memo.solve(queueForMemoization, knapsackCapacity);

end=System.currentTimeMillis();

printKnapsackItem(knapsack);

System.out.println();

System.out.println("Total value are " + memo.computeValue(knapsack)+".");

System.out.println("Knapsack with maximum of "+knapsackCapacity+"kg has weight of " + gm.computeWeight(knapsack) + "kg.");

System.out.println();

System.out.println("The time spent to solve the knapsack problem using Memoization Algorithm is "+calculateTimeSpent(start,end)+"ms.");

System.out.println();

System.out.println("Do you want to calculate using another knapsack capacity ?");

System.out.println("1. Yes");

System.out.println("2. No");

choice=scanner.nextInt();

while(choice!=1&&choice!=2)

{

System.out.print("Enter 1 or 2: ");

choice=scanner.nextInt();

}

}while(choice==1);

scanner.close();

}

public static void printAllItems(PriorityQueue <Item> queue) {

PriorityQueue<Item> printItemQueue = new PriorityQueue<Item>(queue);

System.out.println("Item\tWeight(kg)\tValue");

while(printItemQueue.size() > 0) {

System.out.println(printItemQueue.remove().toString());

}

}

public static void printKnapsackItem(LinkedList<Item> list) {

if(!list.isEmpty())

{

System.out.println("Items include in knapsack: ");

int i = 1;

for (Item b: list) {

System.out.println(i + ". " + b.getName());

i++;

}

}

else

System.out.println("No items are found in the knapsack.");

}

public static long calculateTimeSpent(long startTime, long endTime)

{

return endTime-startTime;

}

}

**Item.java**

public class Item implements Comparable<Item>{

private String name;

private int weight;

private int value;

public Item(String name, int weight, int value)

{

this.name=name;

this.weight=weight;

this.value=value;

}

public String getName()

{

return name;

}

public int getWeight()

{

return weight;

}

public int getValue()

{

return value;

}

public int compareTo(Item i)

{

double ratio1=(double)this.value/(double)this.weight;

double ratio2=(double)i.value/(double)i.weight;

if(ratio1<ratio2)

return 1;

else if(ratio1>ratio2)

return -1;

else

if(this.value<i.value)

return 1;

else if(this.value>i.value)

return -1;

else

return 0;

}

public String toString() {

return name + "\t" + weight + "\t\t" + value;

}

}