PRACTICAL ASSIGNMENT - MARKING REPORT

1. PERSONAL DATA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Group number : 19** | | | | |
| No | Name | ID | Programme | Total Marks |
| 1. | Andrew Woon Mun Hong | 2102584 | SE |  |
| 2. | Gan Wei Jia | 2102338 | SE |  |
| 3. | Liew Chun Kin | 2103388 | SE |  |
| 4. | Lim Jun Hau | 2102629 | 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: Andrew Woon Mun Hong |  |  |  |  |  |  |  |
| Student 2: Gan Wei Jia |  |  |  |  |  |  |  |
| Student 3: Liew Chun Kin |  |  |  |  |  |  |  |
| Student 4: Lim Jun Hau |  |  |  |  |  |  |  |
|  |  |  |  |  | Total | 50 |  |

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **No** | **Title** | **Page** |
| 1 | **Solution for knapsack problem** | 3 |
| 2 | **Discussion / Complexities analysis of application** | 4 – 5 |
| 3 | **Flowchart** | 6 – 7 |
| 4 | **Class diagram** | 8 |
| 5 | **Sample input data** | 9 |
| 6 | **Sample output / Screenshots** | 10 – 12 |
| 7 | **Java Program** | 13 – 18 |

|  |
| --- |
| 1. **Solution for knapsack problem** |
| **Greedy algorithm**    We choose the Greedy algorithm solution to solve the knapsack problem because it is a simple and fast way to sort items such as value, weight, capacity, and quantity. The greedy algorithm makes a local and immediate choice at each step in order to lead to a possible optimal outcome. They can also handle large numbers of inputs without any problem. For example, pack the items on the list until the knapsack has reached maximum space and no more items can be added.  **Dynamic programming algorithm**  The second algorithm solution for the knapsack problem is using a dynamic programming algorithm. Dynamic programming can break down a complex problem into several smaller subproblems. For example, a set of items is created as a table and represents the weight and value of each item, then compares the value and weight of each item and makes the final answer. Therefore, the knapsack problem will be solved efficiently and effectively. It can also find the optimal solution by using this algorithm.  **Data Structure and Assumptions**  For the data structure, we use ArrayList for the greedy algorithm and dynamic programming algorithm to store the items because we think that both algorithms will not add an item in between the list.  We have an assumption on both algorithms which is only 1 time for each item can be restocked. For the capacity, we have only up to 6000 capacities for restocking. |

|  |
| --- |
| 1. **Discussion / Complexities analysis of application** |
| **Greedy algorithm**  **Discussion**  Define the parameter needed in the algorithm.  Item and the capacity of the knapsack problem is defined in the main program and put into the Solution class which is the parent class of the GreedyApproach and the parameter can be inherited.  **Create selectedItem Array**  The selectedItem array contains the selected item that has been chosen in the later loop to store it.  **Perform compare the items**  The method will compare the item by value divided by weight and sort it from smallest to biggest in double data type.  **Perform loop to determine the selected item**  The method will read the capacity and load it as currentCapacity.The method will read the item arrayList size and start from the last one of the arraylist and minus it for every item done processed. If the current capacity is greater than or same as the weight of the item that got read, the total value will add the value of the item and the current capacity will minus the weight of the item. The selectedItem will also get the item to add to the arrayList. Else nothing will happen with the item.  **Printing the result**  The method will print the capacity of the knapsack problem and also the maximum total value the method gets in the loop. The method will also print the name of the selectedItem that got selected as part of the result.  **Time Complicatency:**  The sorting step of the Greedy Algorithm for the 0/1 Knapsack issue is frequently the step that takes the longest. It takes O(n log n) time to sort the items using a criterion like value-to-weight ratio, where n is the total number of items.  Following sorting, the algorithm linearly iterates through the sorted list of items to make constant-time decisions about whether to put each one in the knapsack.  The total time complexity of the Greedy Algorithm for the 0/1 Knapsack problem is O(n log n), where n is the number of items.  **Space Complexity:**  The Greedy Algorithm for the 0/1 Knapsack problem normally has a low space complexity. It needs room to store the intermediate variables and input data (such as the weights and values of the items). The complexity of space is generally O(n).  **Dynamic programming algorithm**  **Discussion**  Define the parameter needed in the algorithm.  Item and the capacity of the knapsack problem is defined in the main program and put into the Solution class which is the parent class of the DynamicSolution and the parameter can be inherited.  **Create 2D dp Array**  The 2D dp array contains the size of row of  item.size() +1 and size of column of capacity+1 to store the operation performed.  **Perform nested loop to run all the operations**  The outer loop will loop until all the items are loaded while the inner loop is until the capacity reached. The inner and outer loop starts from 1 because the 0 is used when if(item.get(1-1).getWeight()<= n) == false. If the item weight is less than the capacity the program will find the max between the value of the previous item value in the same column with the current item value sum with the item value that can get after minus out the capacity with the current item weight. The program will choose the one with maximum value to store. If not the system will store the value of the previous item in the same column.  **Find out which one to choose to maximize the value.**  The program first initializes the remaining cap to equal the capacity. The program will loop for the item that should be chosen to maximize the value. The loop will start from the last of the item and the loop will stop when all items are iterated and  the dp[i][remainCap] =0. If the  dp[i][remainCap] does not equal the previous item in the same column, the item  is chosen and printed the name of the item. The remaining capacity is reduced by the item weight. The system print out the total value as the item in the last row and the last column.  **Time and Space Complexity**  The time and space complexity of the dynamic programming algorithm is a polynomial time and space complexity, O(n^W) and linear time complexity of O(n)  n = Number of the items  W = Capacity  For our case, we have the capacity of 5000. The n will keep increasing as the item.size increases which means there are more items. The polynomial space complexity n is determined by the item.size +1 and W is determined by the capacity +1. The polynomial time complexity n is determined by the outer loop of item.size and inner loop of capacity. The dynamic programming algorithm is polynomial time complexity as we implement nested loops that iterate up to the number of items and the capacity. It is also polynomial space complexity as it implements 2D arrays. It is inefficient to use as we have large capacity and the program may run longer. The O(n) linear time complexity is for the operation that retrieve the item that should be chosen to maximize value. |

|  |
| --- |
| 1. **Flow Chart** |
| **Greedy algorithm**    **Dynamic algorithm** |

|  |
| --- |
| 1. **Class diagram** |
|  |

|  |  |  |
| --- | --- | --- |
| 1. **Sample input data** | | |
| **Item name** | **Weight** | **Value** |
| **Cable** | **2000** | **100** |
| **Casing** | **3000** | **200** |
| **Charger** | **1500** | **50** |
| **Screen Protector** | **1000** | **100** |
| **Earphone** | **1000** | **50** |

|  |
| --- |
| 1. **Sample output / Screenshots** |
| 1. **Main Menu**      1. **Add item menu from Main menu**      * 1. **Add an item on Add item menu**      * 1. **Continue add item after one item is added**      * 1. **Quit from Add item menu**      * 1. **Error occurs during wrong input**      1. **Dynamic solution from Main menu**      1. **Greedy solution from Main menu**      1. **Error when input letter**      1. **Error when input wrong number**      1. **Exit System** |

|  |
| --- |
| 1. **Java Program** |
| **TestKnapsack**  import java.util.\*;  public class TestKnapsack {  public static void main(String[]args) {  Scanner sc= new Scanner(System.***in***);  String mainMenu = ("WELCOME TO STORAGE RESTOCK FOR PHONE ACCESSORIES SYSTEM...\n"  + "CAPACITY OF THE STORAGE IS 6000\n"  + "1. Add item\n"  + "2. Dynamic solution\n"  + "3. Greedy solution\n"  + "4. Exit\n"  + "\nOption: ");    ArrayList <Item> items = new ArrayList <Item>();    int choice = 0;  int capacity = 6000;  int size = items.size();  boolean cdone = false;  do {  size = items.size();  System.***out***.print(mainMenu);  try  {  choice = Integer.*parseInt*(sc.nextLine());    while(choice < 1 || choice >4) {  System.***out***.print("\nError! Incorrect choice.\n");  System.***out***.print(mainMenu);  choice = Integer.*parseInt*(sc.nextLine());  }  cdone= true;  }  catch(NumberFormatException exception)  {  System.***out***.print("\nOnly Numbers ! Please try again");  }  System.***out***.println();    switch(choice) {  case 1:{  boolean con = true;  do {  con = false;  System.***out***.print("PLEASE ENTER THE ITEM THAT YOU WOULD LIKE TO ADD FOR RESTOCK...\n");  System.***out***.print("Item name: ");  String itemName = sc.nextLine();  try {  System.***out***.print("Enter the weights of " + itemName +" : ");  int weight = Integer.*parseInt*(sc.nextLine());  System.***out***.print("Enter the values of " + itemName +" : ");  int value = Integer.*parseInt*(sc.nextLine());  items.add(new Item(itemName,weight, value));  System.***out***.println("SUCCESSFULLY ADDED...");  System.***out***.print("DO YOU WISH TO CONTINUE ADD ITEM?('Y' OR 'y' TO CONTINUE): ");  String cadd = sc.nextLine();  if(cadd.equals("Y") || cadd.equals("y"))  con = true;  else  System.***out***.println("Returning to main menu...");  }  catch(NumberFormatException exception)  {  System.***out***.print("\nOnly Number! Please try again\n");  }  System.***out***.println();  }while(con == true);  break;  }  case 2:{  Solution s= new DynamicApproach(items, capacity);  s.disCurrentItem();  s.solve();  break;  }  case 3:{  ArrayList<Item> selectedItems = new ArrayList<>();    Solution s= new GreedyApproach(items, capacity,selectedItems);  s.disCurrentItem();  s.solve();  break;  }    }    }while(choice != 4);  System.***out***.println("THANK YOU FOR USING THE SYSTEM...");  }  }  **GreedyApproach**  import java.util.\*;  public class DynamicApproach extends Solution {    public DynamicApproach(ArrayList <Item> items, int capacity)  {  super(items, capacity);  }  public void solve()  {  ArrayList <Item> items = this.getItem();  int capacity = this.getCapacity();    int [][] dp = new int [items.size()+1][capacity+1]; //5 //5001    for (int i =1; i< items.size()+1 ; i++) { //size =5  for (int n = 1; n < capacity+1; n++)  {  if (items.get(i-1).getWeight() <= n)  dp[i][n] = Math.*max*(dp[i-1][n], items.get(i-1).getValue()+  dp[i-1][n - items.get(i-1).getWeight()]);  else  dp[i][n] = dp [i-1][n];  }  }  System.***out***.println("\nCAPACITY\t\t" + capacity);  System.***out***.println("MAXIMUM VALUE THAT CAN BE OBTAINED: " + dp[items.size()][capacity]);    int remainCap = capacity;  System.***out***.println("\nSELECTED ITEMS: ");  for (int i = items.size(); i > 0 && dp[i][remainCap] > 0; i--) {  if (dp[i][remainCap] != dp[i - 1][remainCap]) {  System.***out***.println(items.get(i-1).getName());  remainCap -= items.get(i-1).getWeight();  }  }  System.***out***.println();  }  }  **DynamicApproach**  import java.util.\*;  public class GreedyApproach extends Solution {    private ArrayList <Item> selectedItem;    public GreedyApproach(ArrayList <Item> items, int capacity, ArrayList<Item> selectedItem)  {  super(items, capacity);  this.selectedItem = selectedItem;  }    public void solve()  {  ArrayList <Item> items = this.getItem();  int capacity = this.getCapacity();    Collections.*sort*(items, Comparator.*comparingDouble*(item -> (double) item.getValue() / item.getWeight()));  double totalValue = 0;  int currentCapacity = capacity;    for (int i = items.size() - 1; i >= 0; i--) {  if (currentCapacity >= items.get(i).getWeight()) {  totalValue += items.get(i).getValue();  currentCapacity -= items.get(i).getWeight();  selectedItem.add(items.get(i));  }  }  double maxValue = totalValue;    System.***out***.println("\nCAPACITY\t\t" + capacity);  System.***out***.println("MAXIMUM VALUE THAT CAN BE OBTAINED: " + maxValue);    System.***out***.println("\nSELECTED ITEMSs:");  for (Item item : selectedItem) {  System.***out***.println(item.getName());  }  System.***out***.println();  }  }  **Item**  public class 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 void setName(String name)  {  this.name = name;  }  public void setWeight(int weight)  {  this.weight = weight;  }  public void setValues(int value)  {  this.value = value;  }  public String getName()  {  return name;  }  public int getWeight()  {  return weight;  }  public int getValue()  {  return value;  }  }  **Solution** import java.util.\*;  public abstract class Solution implements DisplayItem {  private ArrayList <Item> items;  private int capacity;    public Solution (ArrayList <Item> items, int capacity) {  this.items = items;  this.capacity = capacity;  }    public abstract void solve();    public ArrayList<Item> getItem()  {  return items;  }    public int getCapacity()  {  return capacity;  }    public void disCurrentItem() {  System.***out***.println("NAME\t\t\tWEIGHT\t\t VALUE\n");  for(int i = 0; i < items.size(); i++)  {  System.***out***.printf("%-16s %13d %15d",items.get(i).getName(),items.get(i).getWeight(),items.get(i).getValue());  System.***out***.print("\n");  }  }  }  **DisplayItem**  public interface DisplayItem{  public abstract void disCurrentItem();  } |