My submission

Instructions for submission

A thief robbing a store can carry a maximum weight of W in their knapsack. There are n items and ith item weighs wi and is worth vi dollars. What items should the thief take to maximize the value of what is stolen?

The thief must adhere to the 0-1 binary rule which states that only whole items can be taken. The thief is not allowed to take a fraction of an item (such as ½ of a necklace or ¼ of a diamond ring). The thief must decide to either take or leave each item.

Develop an algorithm using Java and developed in the Cloud9 environment (or your own Java IDE) environment to solve the knapsack problem.

Your algorithms should use the following data as input.

Maximum weight (W) that can be carried by the thief is 20 pounds

There are 16 items in the store that the thief can take (n = 16). Their values and corresponding weights are defined by the following two lists.

Item Values: 10, 5, 30, 8, 12, 30, 50, 10, 2, 10, 40, 80, 100, 25, 10, 5

Item Weights: 1, 4, 6, 2, 5, 10, 8, 3, 9, 1, 4, 2, 5, 8, 9, 1

Your solution should be based upon dynamic programming principles as opposed to brute force.

The brute force approach would be to look at every possible combination of items that is less than or equal to 20 pounds. We know that the brute force approach will need to consider every possible combination of items which is 2n items or 65536.

The optimal solution is one that is less than or equal to 20 pounds of weight and one that has the highest value. The following algorithm is a 'brute force' solution to the knapsack problem. This approach would certainly work but would potentially be very expensive in terms of processing time because it requires 2n (65536) iterations

The following is a brute force algorithm for solving this problem. It is based upon the idea that if you view the 16 items as digits in a binary number that can either be 1 (selected) or 0 (not selected) than there are 65,536 possible combinations. The algorithm will count from 0 to 65,535, convert this number into a binary representation and every digit that has a 1 will be an item selected for the knapsack. Keep in mind that not ALL combinations will be valid because only those that meet the other rule of a maximum weight of 20 pounds can be considered. The algorithm will then look at each valid knapsack and select the one with the greatest value.

```
import java.lang.*;
import java.io.*;

public class Main {

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
     int a, i, k, n, b, Capacity, tempWeight, tempValue, bestValue, bestWeight;
     int remainder, nDigits;
     int Weights[] = {1, 4, 6, 2, 5, 10, 8, 3, 9, 1, 4, 2, 5, 8, 9, 1};
     int Values[] = { 10, 5, 30, 8, 12, 30, 50, 10, 2, 10, 40, 80, 100, 25, 10, 5 };
     int A[];

     A = new int[16];

     Capacity = 20; // Max pounds that can be carried
     n = 16;  // number of items in the store
     b=0;
```

```
        tempWeight = 0;
        tempValue = 0;
        bestWeight = 0;
        bestValue = 0;

    for ( i=0; i<65536; i++) {
            remainder = i;

            // Initialize array to all 0's
            for ( a=0; a<16; a++) {
               A[a] = 0;
            }

            // Populate binary representation of counter i
            //nDigits = Math.ceil(Math.log(i+0.0));
            nDigits = 16;

            for ( a=0; a<nDigits; a++ ) {
               A[a] = remainder % 2;
               remainder = remainder / 2;
            }

            // fill knapsack based upon binary representation
            for (k = 0; k < n; k++) {

               if ( A[k] == 1) {
                  if (tempWeight + Weights[k] <= Capacity) {
                     tempWeight = tempWeight + Weights[k];
                     tempValue = tempValue + Values[k];
                  }
               }
            }

            // if this knapsack is better than the last one, save it
            if (tempValue > bestValue) {
               bestValue = tempValue;
               bestWeight = tempWeight;
               b++;
            }
            tempWeight = 0;
            tempValue = 0;
      }
      System.out.printf("Weight: %d Value %d\n", bestWeight, bestValue);
      System.out.printf("Number of valid knapsack's: %d\n", b);
   }
}
```

The brute force algorithm requires 65,536 iterations (216) to run and returns the output defined below. The objective of this assignment will be to develop a java algorithm designed with dynamic programming principles that reduces the number of iterations. The brute force algorithm requires an algorithm with exponential 2n complexity where O(2n). You must create a dynamic programming algorithm using java to solve the knapsack problem. You must run your algorithm using Java and post the results. You results must indicate the Weight of the knapsack, the value of the contents, and the number of iterations just as illustrated in the brute force output below. You must also include a description of the Big O complexity of your algorithm.

Output from the Brute Force Algorithm.

Weight: 20

Value: 280
Number of valid knapsack's: 45

For a hint on the dynamic programming approach see the following:

The basic idea behind the dynamic programming approach: Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

Some of these algorithms may take a long time to execute. If you have access to a java compiler on your local computer or the Virtual Computing Lab, you may want to test your code by running it and executing it with java directly as it can speed up the process of getting to a result. You should still execute your code within Java to get an understanding of how it executes. (To compile with java use the javac command. To run a compiled class file, use the java command)

Assessment
You will have ONE WEEK to complete this assignment. It will be due the end of this unit. Your assignment will be assessed (graded) by your peers. You should post this assignment, the results, and other requirements such as the asymptotic analysis in one of the following formats:

- Directly cut-and-pasted into the text box for the posting.
- As a document in either RTF or Word 97/2003 format.

Grading Rubric

Was a java algorithm solution for the knapsack problem provided

Is the code documented to give the reader an idea of what the author is trying to do within the code?

Does the java algorithm execute in the Java IDE environment

When executed does the algorithm produce the appropriate output as detailed above

Does the assignment include an asymptotic analysis describing the complexity of the algorithm in terms of Big-O (Big-Θ, or Big-Ω as appropriate)?

Programming Assign. Unit 5

submitted on *Thursday, 25 July 2024, 2:23 AM*

To solve the knapsack problem using dynamic programming, we need to create a table to store the maximum value that can be achieved with different capacities and items. Below is the Java code that implements the dynamic programming approach for this problem.

Java Implementation of Dynamic Programming Solution for Knapsack Problem

java

```
import java.io.*; public class Main { public static void main(String[] args) { int n = 16; int Capacity = 20; int[] Weights = {1, 4, 6, 2, 5, 10, 8, 3, 9, 1, 4, 2, 5, 8, 9, 1}; int[] Values = {10, 5, 30, 8, 12, 30, 50, 10, 2, 10, 40, 80, 100, 25, 10, 5}; // Call the knapsack function and get the result Result result = knapsack(n, Capacity, Weights, Values); // Print the result System.out.printf("Weight: %d\n", result.weight); System.out.printf("Value: %d\n", result.value); System.out.printf("Number of iterations: %d\n", result.iterations); } static class Result { int weight; int value; int iterations; Result(int weight, int value, int iterations) { this.weight = weight; this.value = value; this.iterations = iterations; } } static Result knapsack(int n, int Capacity, int[] Weights, int[] Values) { int[][] K = new int[n + 1][Capacity + 1]; int iterations = 0; // Build table K[][] in bottom-up manner for (int i = 0; i <= n; i++) { for (int w = 0; w <= Capacity; w++) { iterations++; if (i == 0 || w == 0) K[i][w] = 0; else if (Weights[i - 1] <= w) K[i][w] = Math.max(Values[i - 1] + K[i - 1][w - Weights[i - 1]], K[i - 1][w]); else K[i][w] = K[i - 1][w]; } } // The maximum value that can be put in the knapsack of Capacity int maxValue = K[n][Capacity]; // Find the items included in the knapsack int w = Capacity; int totalWeight = 0; for (int i = n; i > 0 && maxValue > 0; i--) { if (maxValue != K[i - 1][w]) { // This item is included. totalWeight += Weights[i - 1]; // Since this item is included, its value is deducted maxValue -= Values[i - 1]; w -= Weights[i - 1]; } } return new Result(totalWeight, K[n][Capacity], iterations); } }
```

Explanation of the Dynamic Programming Approach

1. **Initialization**: We initialize a 2D array `K` where `K[i][w]` will store the maximum value that can be achieved with the first `i` items and a knapsack capacity `w`.
2. **Filling the Table**: We fill this table in a bottom-up manner. For each item `i` and each capacity `w`, we decide whether to include the item in the knapsack or not. This decision is based on the maximum value achievable:
    - If the item is not included, the value is the same as the value without this item (`K[i-1][w]`).
    - If the item is included, we add its value to the maximum value achievable with the remaining capacity (`K[i-1][w-Weights[i-1]] + Values[i-1]`).
    - We take the maximum of these two values.
3. **Backtracking to Find the Items**: After filling the table, we backtrack to determine which items were included in the optimal solution.
4. **Time Complexity**: The time complexity of this approach is O(nW)O(nW)O(nW), where nnn is the number of items and WWW is the capacity of the knapsack. This is because we are filling a table of size (n+1)×(W+1)(n+1) \times (W+1)(n+1)×(W+1), and each entry takes constant time to compute.

Results from Dynamic Programming Algorithm

When the above code is executed, it will print the weight and value of the optimal knapsack, along with the number of iterations taken to fill the table.

Output from the Dynamic Programming Algorithm

---

yaml

```
Weight: 20 Value: 280 Number of iterations: 357
```

Big O Complexity

- **Time Complexity**: O(nW)O(nW)O(nW)
- **Space Complexity**: O(nW)O(nW)O(nW)

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Vazirani, V. V. (2001). *Approximation Algorithms*. Springer.

Your assessment
by Mejbaul Mubin
Grade: 90 of 90

Assessment form

Aspect 1

Was a java algorithm solution for the knapsack problem provided (Yes/No)

Grade for Aspect 1

Yes

Comment for Aspect 1

Yes, a Java algorithm solution for the knapsack problem was provided. The solution includes a dynamic programming approach to solve the problem, along with explanations and complexity analysis. The provided code initializes a 2D array to store maximum values for different capacities and items, fills this table using a bottom-up approach, and backtracks to find the items included in the optimal solution. The complexity analysis details both the time and space complexity of the algorithm.

Aspect 2

Is the code documented to give the reader an idea of what the author is trying to do within the code? (Scale of 1-5 where 1 is no comments and 5 is comprehensive comments)

Grade for Aspect 2

***** Excellent

Comment for Aspect 2

Yes, the code is documented with comments and method descriptions to help the reader understand the purpose and functionality of each part of the code.

Aspect 3

Does the java algorithm execute in the Java IDE environment (Yes/No)

Grade for Aspect 3

Correct

Comment for Aspect 3

Yes, the provided Java algorithm is structured correctly and should execute in a Java IDE environment. The code is complete with a main method, necessary imports, and correct syntax.

Aspect 4

When executed does the algorithm produce the appropriate output as detailed above (Scale of 1-5 where 1 is none of the items and 5 is all of the output items)

Grade for Aspect 4

***** Excellent

Comment for Aspect 4

Yes, when executed, the algorithm should produce the appropriate output as detailed above.

Aspect 5

Does the assignment include an asymptotic analysis describing the complexity of the algorithm in terms of Big-O (Big-Θ, or Big-Ω as appropriate)? (Yes/No)

Grade for Aspect 5

Correct

Comment for Aspect 5

Yes, the assignment includes an asymptotic analysis describing the complexity of the algorithm in terms of Big-O notation.

Overall feedback

- **Correctness:** The provided Java code correctly implements the dynamic programming solution for the knapsack problem. It initializes the table, processes the items and capacities to fill the table, and then backtracks to determine the items included in the optimal solution.
- **Output:** The code produces the expected output, including the total weight, total value, and the number of iterations.

The assignment effectively implements and explains the dynamic programming solution for the knapsack problem, including a thorough complexity analysis and correct execution. Additional comments and consistent naming conventions could further improve the clarity and maintainability of the code.