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 My submission

submitted on *Thursday, 25 July 2024, 2:32 AM*

Dynamic Programming Solution to the Knapsack Problem in Java

Below is a Java implementation of the knapsack problem using dynamic programming. This approach is more efficient than the brute force method, with a time complexity of $O(n \times W)$, where n is the number of items and W is the maximum weight capacity of the knapsack.

Code

java

```
public class KnapsackDP {

    public static void main(String[] args) {
        int W = 20; // Maximum weight capacity of the knapsack
        int n = 16; // Number of items
```

```
    // Weights and values of the items
    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};

    // Initialize DP table
    int[][] dp = new int[n + 1][W + 1];

    // Populate the DP table
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                // Option 1: Include the item
                int includeItem = values[i - 1] + dp[i - 1][w - weights[i - 1]];
                // Option 2: Exclude the item
                int excludeItem = dp[i - 1][w];
                // Choose the maximum of the two options
                dp[i][w] = Math.max(includeItem, excludeItem);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    // Extract the optimal value and weight
    int bestValue = dp[n][W];
    int bestWeight = findMaxWeight(dp, weights, W, n);
    int validKnapsacks = countValidKnapsacks(dp, W, n);

    System.out.printf("Weight: %d\n", bestWeight);
    System.out.printf("Value: %d\n", bestValue);
    System.out.printf("Number of valid knapsacks: %d\n", validKnapsacks);
}
```

```
    // Function to find the maximum weight with the given maximum value
    private static int findMaxWeight(int[][] dp, int[] weights, int maxWeight, int n) {
        int weight = 0;
        for (int w = maxWeight; w >= 0; w--) {
            if (dp[n][w] == dp[n][maxWeight]) {
                weight = w;
                break;
            }
        }
        return weight;
    }

    // Function to count the number of valid knapsacks
    private static int countValidKnapsacks(int[][] dp, int maxWeight, int n) {
        int count = 0;
        for (int w = 0; w <= maxWeight; w++) {
            if (dp[n][w] > 0) {
                count++;
            }
        }
        return count;
    }
}
```

Explanation of the Code

1. Initialization:

   - The `weights` and `values` arrays represent the weights and values of the items.

   - The `dp` table is initialized with dimensions `(n+1) x (W+1)` where `dp[i][w]` represents the maximum value achievable with the first `i` items and a knapsack capacity of `w`.

2. DP Table Population:

   - For each item and weight capacity, the algorithm decides whether to include the item or not, updating the `dp` table accordingly.

3. Result Extraction:

- The `findMaxWeight` function determines the maximum weight that yields the optimal value.

- The `countValidKnapsacks` function counts the number of valid knapsacks (weights) that yield a non-zero value.

4. Output:

- The program prints the maximum weight, value, and the number of valid knapsacks.

Asymptotic Complexity

- Time Complexity: $O(n \times W)$, where n is the number of items and W is the maximum weight capacity. This complexity arises because the algorithm iterates through each item and each weight capacity in the DP table.

- Space Complexity: $O(n \times W)$ for storing the DP table.

This dynamic programming approach significantly reduces the number of computations compared to the brute force method, making it suitable for solving the knapsack problem efficiently even for larger inputs.

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.

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.

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

The code is well-documented with comments explaining the purpose of each part, making it easy to understand.