

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 i th item weighs w_i and is worth v_i 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 $\frac{1}{2}$ of a necklace or $\frac{1}{4}$ 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 2^n 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 2^n (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) then 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 2^n complexity where $O(2^n)$. 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. Your 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, 1:43 AM* | modified on *Thursday, 25 July 2024, 1:51 AM*

ASYMPTOTIC ANALYSIS:

ALGORITHM CODE -- language: *Java*

```

import java.util.*;

public class Knapsack {
    . . .
    static ArrayList<Item> FillKnapsack(int capacity, Item[] i){
        // store the list of values for items and capacity
        int[][] valueList = new int[i.length + 1][capacity + 1];
        // loop through the items
        for (int j = 1; j <= i.length; j++) { // loop time complexity: O(n)
            // loop through the capacities
            for (int c = 1; c <= capacity; c++) { // loop time complexity: O(W)
                //calculate the value of not including the item
                int withoutItem = valueList[j-1][c];
                Item thisItem = i[j-1]; // since the item list doesn't account for the blank row subtract 1
                if (c < thisItem.weight) {
                    // if we can't even include the item, simply ignore the item
                    valueList[j][c] = withoutItem;
                } else {
                    // otherwise calculate the value of including the item
                    int withItem = valueList[j - 1][c - thisItem.weight] + thisItem.value;
                    // decide whether or not to include the item
                    valueList[j][c] = (withItem > withoutItem) ? withItem : withoutItem;
                }
            }
        }
        // create an array to store the combination of items in the knapsack
        ArrayList<Item> combination = new ArrayList<Item>();
        // identify the total value
        int totalValue = valueList[i.length][capacity];
        // initialize the total weight
        int totalWeight = 0;
        // loop backwards over the items
        for (int j = i.length; j > 0 & totalWeight < capacity; j--) { // loop time complexity: O(n)
            if (valueList[j][capacity-totalWeight] > valueList[j-1][capacity-totalWeight]) {
                // if we included the item (we have a different value than the previous row), then
                // add the current item to our knapsack
                combination.add(i[j-1]);
                // and add the current item weight to our total weight
                totalWeight += i[j-1].weight;
            }
        }
        // display/return the results
        System.out.println("Weight: " + totalWeight);
        System.out.println("Value: " + totalValue);
        return combination;
        // overall time complexity is O(n+nW) -> O(nW)
    }
}

```

The algorithm has two main parts (calculating the best value and reconstructing the list of items).

The first part has two nested loops:

- for (`int j = 1; j <= i.length; j++`) which loops through the items and has a time complexity of $O(n)$
- for (`int c = 1; c <= capacity; c++`) which loops through the capacities and has a time complexity of $O(W)$

This means it has a time complexity of $O(nW)$

The second part has a single loop:

- for (`int j = i.length; j > 0 & totalWeight < capacity; j--`) which loops through the items in reverse and has a time complexity of $O(n)$

So it has a time complexity of $O(n)$

When we combine this we get a time complexity of $O(nW + n)$ and this can be simplified to: $O(nW)$.

So the time complexity of the overall algorithm is $O(nW)$

OUTPUT:

OUTPUT -- language: *Java*

```
Weight: 20
Value: 280
[(v: 100, w:5), (v: 80, w:2), (v: 40, w:4), (v: 50, w:8), (v: 10, w:1)]
```

CODE

FULL CODE -- language: *Java*

```

import java.util.*;

public class Knapsack {
    /**
     * Class for the items
     */
    static class Item {
        // Instance variables
        int value;
        int weight;

        /**
         * Constructor for an item
         * @param v Value of the item
         * @param w Weight of the item
         */
        Item(int v, int w){
            this.value = v;
            this.weight = w;
        }

        /**
         * Overridden toString method represents the item
         */
        @Override
        public String toString() {
            // represent the item
            return "(value: " + this.value + ", weight:" + this.weight + ")";
        }
    }

    /**
     * Main method that runs the knapsack algorithm on the problem
     * @param args
     */
    public static void main(String[] args) {
        // fill a knapsack with items
        ArrayList<Item> itemsSelected = Knapsack.FillKnapsack(20, new Item[] {
            new Knapsack.Item(10, 1),
            new Knapsack.Item(5, 4),
            new Knapsack.Item(30, 6),
            new Knapsack.Item(8, 2),
            new Knapsack.Item(12, 5),
            new Knapsack.Item(30, 10),
            new Knapsack.Item(50, 8),
            new Knapsack.Item(10, 3),
            new Knapsack.Item(2, 9),
            new Knapsack.Item(10, 1),
            new Knapsack.Item(40, 4),
            new Knapsack.Item(80, 2),
            new Knapsack.Item(100, 5),
            new Knapsack.Item(25, 8),
            new Knapsack.Item(10, 9),
            new Knapsack.Item(5, 1)
        });
        // print the result
        System.out.println(itemsSelected);
    }


    /**
     * Function to fill a knapsack using dynamic programming
     * Time complexity: O(nW)
     *
     * @param capacity the capacity of the knapsack
     * @param i the items that we can add
     * @return the best combination of items (maximizes value)
     */
}

```

```

static ArrayList<Item> FillKnapsack(int capacity, Item[] i){
    // store the list of values for items and capacity
    int[][] valueList = new int[i.length + 1][capacity + 1];
    // loop through the items
    for (int j = 1; j <= i.length; j++) { // loop time complexity: O(n)
        // loop through the capacities
        for (int c = 1; c <= capacity; c++) { // loop time complexity: O(w)
            // calculate the value of not including the item
            int withoutItem = valueList[j-1][c];
            Item thisItem = i[j-1]; // since the item list doesn't account for the blank row subtract 1
            if (c < thisItem.weight) {
                // if we can't even include the item, simply ignore the item
                valueList[j][c] = withoutItem;
            } else {
                // otherwise calculate the value of including the item
                int withItem = valueList[j - 1][c - thisItem.weight] + thisItem.value;
                // decide whether or not to include the item
                valueList[j][c] = (withItem > withoutItem) ? withItem : withoutItem;
            }
        }
    }
    // create an array to store the combination of items in the knapsack
    ArrayList<Item> combination = new ArrayList<Item>();
    // identify the total value
    int totalValue = valueList[i.length][capacity];
    // initialize the total weight
    int totalWeight = 0;
    // loop backwards over the items
    for (int j = i.length; j > 0 & totalWeight < capacity; j--) { // loop time complexity: O(n)
        if (valueList[j][capacity-totalWeight] > valueList[j-1][capacity-totalWeight]) {
            // if we included the item (we have a different value than the previous row), then
            // add the current item to our knapsack
            combination.add(i[j-1]);
            // and add the current item weight to our total weight
            totalWeight += i[j-1].weight;
        }
    }
    // display/return the results
    System.out.println("Weight: " + totalWeight);
    System.out.println("Value: " + totalValue);
    return combination;
    // overall time complexity is O(n+nw) -> O(nw)
}
}

```

 **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 should execute in a Java IDE environment, given that all necessary components (such as the Knapsack class and the Item class) are properly defined and the environment is correctly set up.

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 in a Java IDE, the algorithm should produce the appropriate output as detailed above, given the provided set of items and the specified capacity. The output should display the total weight and value of the selected items, along with the list of items that maximize the value within the given capacity.

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.
- The `Item` class has clear constructor and `toString` method documentation.
- The algorithm correctly implements the dynamic programming approach to solve the knapsack problem.
- The logic for calculating the best value and reconstructing the list of items appears correct.

The code includes an asymptotic analysis describing the complexity of the algorithm in terms of Big-O notation, which is accurate and helpful.