Andrew Doyle

Dr. Andrew Sutton

Algorithms

12/14/16

0-1 Knapsack Problem

**The Problem**

From wikipedia:

The most common problem being solved [of the knapsack problems] is the **0-1 knapsack problem**, which restricts the number *xi* of copies of each kind of item to zero or one. Given a set of *n* items numbered from 1 up to *n*, each with a weight *wi* and a value *vi*, along with a maximum weight capacity *W*,

Maximize  subject to and 

Here *xi* represents the number of instances of item *i* to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

Optimal Substructure

The algorithm has optimal substructure. Each step of the problem can be considered a subproblem that may later relate to a larger subproblem later. The ending maximized subproblem has to be built upon previous subproblems because the prior decisions about items always are always reviewed in conjunction with the newest item to be considered. For example, if the first item examined was worth carrying at 2 pounds (out of a max of 30 let’s say), and the second item examined is worth carrying at 15 pounds, once the algorithm considers carrying 17/30 pounds by including the second item, it would see that the value would increase for a 17 pound bag to include both the 2 pound and the 15 pound item. It knows this by examining the subproblems previously decided, which transitions into checking if this is a dynamic programming problem.

Dynamic Programming

This problem lends itself well to dynamic programming. Each decision about a subproblem is made and recorded so that future subproblems can make use of this information. Maximum values at different weights and numbers of items can be reviewed by future subproblems. Future subproblems need this overlapping work to be done to consider adding more weight to the knapsack. Without remembering the answers to prior subproblems, all possible previous combinations would have to be recalculated for each subproblem, which would dramatically increase the overall runtime of the program.

**The Algorithm**

From wikipedia:

Assume  are strictly positive integers. Define to be the maximum value that can be attained with weight less than or equal to  using items up to  (first  items).

We can define  recursively as follows:

* 
*  if (the new item is more than the current weight limit)
* if 

The solution can then be found by calculating . To do this efficiently we can use a table to store previous computations.

**My Implementation**

Below is the main body of the algorithm.

|  |  |
| --- | --- |
| for(int i=1; i<numItems; i++) //Loop over the worksheet considering all items.  for(int j=0; j<maxWeight+1; j++){ //Consider item at each valid weight as subproblems.  if(weights[i]>j)  //Drop in the value from the row above if item is too heavy for this iteration.  worksheet[i][j] = worksheet[i-1][j];  else if(worksheet[i-1][j-weights[i]] +values[i] > worksheet[i-1][j]){  //If the value of the worksheet would go up by adding this item, include it.  worksheet[i][j] = worksheet[i-1][j-weights[i]] +values[i];  //Note that we have taken this item at this step.  taken[i][j]=true;  }  else //Drop in the value from the row above if item doesn't help.  worksheet[i][j] = worksheet[i-1][j];  if(j>0 && worksheet[i][j]< worksheet[i][j-1])  //Copy value forward if prior entry was greater.  worksheet[i][j]= worksheet[i][j-1];  } | |
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|

Before starting these loops, we set up our vectors of weights and values with randomly generated weights and values, and populate our tables named worksheet and taken with 0’s. I decided to add an extra column of 0’s to the maxWeight number of columns in the tables to represent the case where an item weighs 0. In contrast, the numItems number of rows in the tables did not have an extra +1 row, I found the algorithm had proper behavior without needing the extra “0 item” row. The first row represents the case where we have considered 1 item. To accommodate this, the program considers the first item before the main body of the algorithm using the code below:

|  |  |
| --- | --- |
| //Put our first item on our worksheet to get it started.  //Avoids indexing problems in the main loop.  if(weights[0] < maxWeight){  worksheet[0][weights[0]] = values[0];  taken[0][weights[0]]=true;  for(int i=weights[0]; i<maxWeight+1; i++){  worksheet[0][i+1]=worksheet[0][i];  taken[0][i]=true;  }  }  else if(weights[0] == maxWeight){  worksheet[0][weights[0]] = values[0];  taken[0][weights[0]]=true;  } | |
|
|
|
|
|
|
|
|
|
|
|
|
|

Input

The program generates a fixed number of items and works with a fixed maximum weight. It generates items with random attributes within fixed weight and value ranges by using rand() with the modulus operator. It is possible for our items to have 0 value or 0 weight. I considered changing this behavior, but it seems possible that items could have 0 value or be essentially 0 weight. For a real world example, money or jewelry weighs so little that the scope of the problem would have to be massive before it became an issue of finding huge numbers of “0 weight” items. The items you were considering would have to be skewed that way for it to become a problem quickly. It could be something to consider if you were robbing a bank or a jewelry store for example. For that kind of application, we would have to change the representation from integer weights to doubles or floats.

Output

The outputs of the algorithm are the worksheet table and the taken table. The worksheet table can be used to examine the maximum values possible with different numbers of items and amounts of weight, and the taken table can be used to retrace the steps made by the algorithm to create said maximum values.

I had the program print output to console to show the randomly generated items and the maximum value to be made by combining them given the weight constraint. It prints the indices of the items needed to reach that maximum value. It also prints our worksheet and taken decisions at the end of the program so we can study the output of the algorithm. You can see an example on the next page.

Below is a sample output:

|  |  |
| --- | --- |
| Item 0: weighs 46 pounds, worth $6  Item 1: weighs 11 pounds, worth $4  Item 2: weighs 30 pounds, worth $8  Item 3: weighs 28 pounds, worth $5  Item 4: weighs 27 pounds, worth $8  Item 5: weighs 30 pounds, worth $9  Maximum weight of our knapsack: 30  Maximum value we can carry: $9  Total weight used to carry maximum value: 30/30  Which item(s) to take for this solution: 5.  Ending worksheet:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4  0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 8  0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 8  0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8  0 0 0 0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 9  Ending taken decisions:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 | |
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|

How to use it

The program can be run after you build it with a compiler. If you wanted to change the number of items or maximum weight, those are easily changed. The ranges of random values and weights assigned to the items can be modified by changing the arguments to the mod operator.

Space Requirements

The largest data structures used for the algorithm are the table to keep track of the algorithm’s value outputs and the table to keep track of whether or not a subproblem included an item. These are both vectors of int vectors of

Size = number of items to consider \* maximum weight allowable

And since we need two of them, the upper bound on size constraint would be

2(i\*W) = O(i\*W)

Where i is the number of items to consider and W is maximum weight allowable.

Time Requirements

The biggest time sink of the algorithm would be the main loop body of the algorithm. The loops run for

(number of items+1) \* maximum weight

iterations. Within those loops are only addition and assignment operations. These would not add to the time big-O time complexity of the algorithm. So, the overall complexity would come to:

O(i\*W)

where i is the number of items to consider and W is the maximum weight allowable.