**Algorithm Study Template**

**Algorithm**: Greedy Solution to the Fractional Knapsack Problem

**aka**: Greedy Solution to the Continuous Knapsack Problem

**Techniques**: Greedy

**Categories**: Searching, Sorting

**Problem**: This algorithm solves the Fractional Knapsack Problem, a variation of the classical Knapsack Problem. Given a knapsack with a fixed capacity and materials of different values per unit volume, it will find the most valuable combination of materials that fit inside the knapsack. The difference from the classical problem is that the Fractional Knapsack Problem allows fractions of materials inside the knapsack, rather than enforcing whole materials.

**Applications**: Via Wikipedia: “Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials, selection of investments and portfolios, selection of assets for asset-backed securitization, and generating keys for the Merkle-Hellman Knapsack Cryptosystem.” In particular, the Fractional Knapsack problem can arise in situations where liquid materials are used. It can also appear in Langrangian relaxation methods for facility location problems.

**References**:

* <http://en.wikipedia.org/wiki/Fractional_knapsack_problem>
* <http://xlinux.nist.gov/dads/HTML/fractionalKnapsack.html>
* <http://xlinux.nist.gov/dads/HTML/knapsackProblem.html>
* <http://www.cse.ust.hk/~dekai/271/notes/L14/L14.pdf>

**Implementation details**:

* **Big Idea**: Place materials in the knapsack in descending order according to their value/cost ratio until the knapsack is full. If the last material into the knapsack costs more than the remaining volume of the knapsack, only the percent of the material that the remaining volume can contain will be added. Once the knapsack is full, no more materials can be added, regardless of whether or not every material made it into the bag.
* **Description**: While the knapsack is not full, iterate through materials and look for the one with the highest value/cost ratio. Once found, add that material to the knapsack and flag it so it won’t be found and added again. Continue to find materials with the best ratio in descending order and adding them to the knapsack until it is full. If the last material going into the knapsack costs more than the remaining volume of the knapsack, add only the percent which the knapsack can contain. Once the knapsack is full, iteration ceases and the total value of the knapsack’s contents is reported.
* **Pseudo-code**:

for(all elements of inserted){

inserted[i] = 0

}

//set space left in knapsack

current space = value passed in as knapsack size

while(current space > 0){

//largest ratio not yet found

index of material with largest ratio = -1

for(all materials){

//if largest ratio not set or smaller than current

if(current index == 0 && ((index of material with largest ratio == -1) || ratio of current index > largest ratio)){

//set current ratio as largest

index of material with largest ratio = current index

}

}

//flag material as added

inserted[index of material with largest ratio] = 1

//update remaining knapsack space

current space -= cost of material with largest ratio

//update value of knapsack contents

total value += value of material with largest ratio

if(current space >= 0){

//if bag is exactly filled

print object added, value, cost, and remaining space

}else{

//if last material is partial/fractional

print percent of last material added, value, and cost

}

}

print total value of materials in knapsack

* **Specific implementation**: (see FractionalKnapsack.java)

**Correctness**:

**Theoretical**: If we start by finding the material with the largest value/cost ratio and placing it in the knapsack, we can then keep finding and adding the materials with the largest ratios (excluding the materials already in the knapsack) until the knapsack becomes full. If the last material going into the knapsack does not fit as a whole, we only add the percent of the material which will fit. At that point, the knapsack is full and no more items are added. Please note that since the tests I run on this program are all specific cases, it is assumed that input is always correct and will not break the program.

**Empirical**: I used six different sets of test data. They are described below. Since the tests are all specific cases, it is assumed that input is always correct and will not break the program. Please note that “values” refers to the value of the materials themselves. For example, the value might represent what the material would cost in a store. “Cost”, by contrast, is the amount of space that material takes up in the knapsack. The ratios determine the order in which the materials are added to the knapsack (in descending order). (Additional test information can be seen in the output of FractionalKnapsack.java)

Test A:

* 5 materials
* Costs were 12, 1, 2, 1, and 4 units
* Values were 4, 3, 3, 2, and 10 dollars
* Ratios were 4/12, 3/1, 3/2, 2/1, and 10/4
* Size of knapsack was 16 units
* The total cost of materials was 20 units, but the knapsack could only hold 16 units
* So, only 67% of the last material was put into the knapsack

Test B:

* 5 materials
* Costs were 12, 1, 2, 1, and 4 units
* Values were 4, 3, 3, 2, and 10 dollars
* Ratios were 4/12, 3/1, 3/2, 2/1, and 10/4
* Size of knapsack was 20 units
* The total cost of the materials was 20 units, and the knapsack held exactly 20 units
* So, no partial materials were put into the knapsack

Test C:

* 5 materials
* Costs were 15, 3, 2, 6, and 4 units
* Values were 4,5,3,3, and 10 dollars
* Ratios were 4/15, 5/3, 3/2, 3/6, and 10/4
* Size of knapsack was 28 units
* The total cost of the materials was 30 units, but the knapsack could only hold 28 units
* So, only 87% of the last material was put into the knapsack

Test D:

* 5 materials
* Costs were 15, 3, 2, 6, and 4 units
* Values were 4, 5, 3, 3, and 10 dollars
* Ratios were 4/15, 5/3, 3/2, 3/6, and 10/4
* Size of knapsack was 30 units
* The total cost of the materials was 30 units, and the knapsack held exactly 30 units
* So, no partial materials were put into the knapsack

Test E:

* 10 materials
* Costs were 15, 3, 2, 10, 5, 20, 2, 3, 3, and 2 units
* Values were 4, 2, 3, 1, 10, 20, 5, 7, 2, and 8 dollars
* Ratios were 4/15, 2/3, 3/2, 1/10, 10/5, 20/20, 5/2, 7/3, 2/3, and 8/2
* Size of knapsack was 50 units
* The total cost of the materials was 65 units, but the knapsack only held 50 units
* So, only 67% of the last material was put into the knapsack
* It is also noteworthy that because of the large cost of some materials, not all of them made it into the knapsack. Material #4 (ratio of 1/10) was not put into the knapsack because by the time it (with the lowest ratio) was reached at the end, the knapsack was already full. Thus, material #4 was excluded and only the most valuable materials were placed in the knapsack.

Test F:

* 10 materials
* Costs were 15, 3, 2, 6, 4, 10, 2, 3, 3, and 2 units
* Values were 4, 2, 3, 1, 10, 20, 5, 7, 2, and 8 dollars
* Ratios were 4/15, 2/3, 3/2, 1/6, 10/4, 20/10, 5/2, 7/3, 2/3, and 8/2
* Size of knapsack was 50 units
* The total cost of the materials was 50 units, and the knapsack held exactly 50 units
* So, no partial materials were put into the knapsack

**Performance**:

**Theoretical**: Big O is **O(n2). Because this algorithm has a fairly large Big O, it is not intended for use with very large sets of materials.**

**Empirical**: I timed the knapsack-filling method for each test. Times vary from one run to the next, so below are two instances of times I collected. I believe the spikes in test A are due to print statements being run within the algorithm.

Run 1:

* Test A completed in 29.048 milliseconds
* Test B completed in 0.564 milliseconds
* Test C completed in 0.609 milliseconds
* Test D completed in 0.585 milliseconds
* Test E completed in 0.787 milliseconds
* Test F completed in 1.011 milliseconds

Run 2:

* Test A completed in 38.43 milliseconds
* Test B completed in 0.816 milliseconds
* Test C completed in 0.554 milliseconds
* Test D completed in 0.457 milliseconds
* Test E completed in 0.717 milliseconds
* Test F completed in 1.007 milliseconds

**Anecdotes**: <none>

**History**: George Dantzig was the first to propose a greedy solution to a knapsack problem. His version was specifically proposed to solve the Unbounded Knapsack Problem, but it can be extended to other knapsack problems as well.

**Variations**: There is another way to implement the greedy solution. In it, the materials are sorted in descending order of value/cost ratio prior to insertion. This eliminates the need to repeatedly search for the largest ratio during insertion. That version has a Big O of **O(n\*lg(n)).**

**Alternatives**: The classical Knapsack Problem has many other solutions (dynamic programming, meet-in-the-middle, fully polynomial time approximation scheme, and dominance relations), but the Fractional Knapsack Problem does not appear to have any good alternatives.

**Credits:**

* <http://compprog.wordpress.com/2007/11/20/the-fractional-knapsack-problem/>
* <http://rosettacode.org/wiki/Knapsack_problem/Continuous#Java>
* <http://en.wikipedia.org/wiki/Knapsack_problem>