## 1 General Premise Knapsack Problem [1, 9]

The Knapsack problem creates the following context to be solved:

You are a thief, smuggler, or petty crook (say Han Solo, Team Rocket, Yuffie, etc.) with a knapsack (smuggling compartments, etc.) and are looking to fill it. You have break into the scene and quickly create a list of weights and values for the items you see.. Assuming you're a smart thief, you realize that you can't possibly take everything because your bag has a finite amount of weight it can hold (and you're a CS/Math major, so you can't carry a lot). The task at hand now is to figure out how exactly to fill the knapsack so that you get the most value in your heist for the limited amount of weight you can carry.

The Knapsack problem is classified as an optimization problem because we are looking to maximize the value within our weight limit. Optimization problems are broken down into a maximization (or minimization) function and a set of constraints. Our problem can be broken down into these categories as follows:

We have have the following vectors:

$v = \{v_1, v_2, ..., v_n\}$, a vector containing each object's respective value
$w = \{w_1, w_2, ..., w_n\}$, a vector containing each object's respective weight

We want to maximize value *V*, which takes the form:

$$V = \sum_{j=1}^{n} v_j x_j$$

Constrained by a weight *W*, such that:

$$\sum_{j=1}^{n} w_j x_j \le W, x_j \in \{0,1\} \ \forall \ j \in \{1, ..., n\}$$

If the maximum weight *W* is greater than the sum of the weights of all available items, everything can be taken. In the case where the maximum weight *W* is less than the sum of all weights, we can clearly see that choosing what items to take and leave is the decision that needs to be made. The problem reduces to one of selecting the optimal object from a finite set of objects, which has been studied in combinatorial optimization.

## 2 The Fractional Knapsack [1, 8, 9]

The Fractional (Continuous) Knapsack problem is a special case of the generalized Knapsack problem. This version of the problem still maintains the general structure that there are three vectors: a value vector *v*, and weight vector *w*. In this case, the thief can opt to only take a portion of an item, meaning that only a fraction of the weight is added to the knapsack. An additional constraint is that the fraction selected of a particular item, $a_i$, cannot be more than is actually available or negative; therefore, $0 \leq \alpha_i \leq 1$.

## 2.1 The Greedy Thief [1, 8, 9]

Let's take the following example. We have a thief who has the ability to carry 6 items (we'll ignore specific units for now) in his/her knapsack. The thief has the following items to choose from:

- A set of 4 items valued at a rating of 140
- A set of 3 items valued at a rating of 90
- A single item rated at value 50
- A single item rated at 25

In the case of the Fractional Knapsack problem, the greedy solution works based on the value per unit weight. The first step is to walk through the items and calculate the unit weight. So our list is transformed into the following list of unit values:

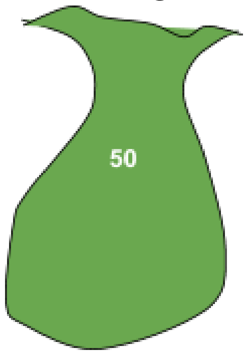
- A set of 4 items valued at 35/unit
- A set of 3 items valued at 30/unit
- A single item valued at 50/unit
- A single item valued at 25/unit

Next, we sort the list by unit value:

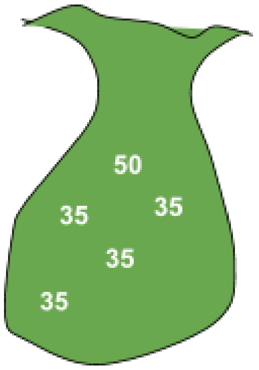
- A single item valued at 50/unit
- A set of 4 items valued at 35/unit
- A set of 3 items valued at 30/unit
- A single item valued at 25/unit

From here, the problem is simply picking the items with the highest value per unit until the knapsack is full. The knapsack will be filled in the following steps:

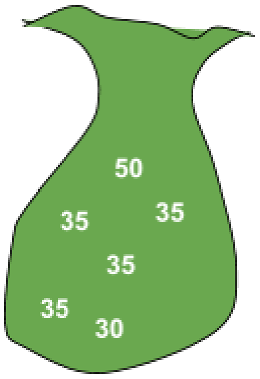
Add the single item of value 50/unit (current weight = 5).



Add the four items of value 35/unit (current weight = 1).



Add one of the three items of value 30/unit (current weight = 0).



Then the resulting value V will be:

$$V = \sum_{j=1}^{n} v_j \alpha_j$$

where $v_j$ is the total value of the *full* weight of the $j^{th}$ item, and $\alpha_j$ is the fraction of the $j^{th}$ item taken (constrained as mentioned above). In our example, we would take (from the sorted list) all of the first two items and only one of the third item. The resulting value $v$ would be:

$$v = 50(1/1) + 140(4/4) + 90(1/3) + 20(0) = 220$$

This algorithm can be written in pseudocode as follows:

```
FRACTIONAL-KNAPSACK(v, w, n, W)
  let c[0..n] be a new array
  let x[0..n] be a new array
  for i = 1 up to n
      c[i] = v[i]/w[i]
      x[i] = 0
  Sort(c, v, w)
  for i = 1 up to n and W > 0
```

```
        if (W >= w[i])
            x[i] = w[i]
        else
            x[i] = W/w[i]
        W = W - w[i]
    return x
```

If we look at the two **for** loops we see that they run at worst *O(n)* times. The **Sort** function runs in *O(n log n)* time, resulting in a total runtime of *O(n + n log n)*. After factoring, the runtime expression can be reduced to *O(n log n)*. The space complexity of the greedy fractional algorithm is *O(n)* because only two arrays of size *n* are allocated.

## 2.2 Why Greedy Works [1, 8, 9]

In order to prove that a greedy solution is valid, there are two things that need to be shown:

1. That the greedy choice is always the best choice, such that once the first choice is made that an optimal solution can be reached.
2. Optimal substructure, or that once a choice is made the remaining problem is a smaller version of the full problem.

To prove the greedy choice property in this instance, let us suppose there is an optimal ratio of value to weight, $v'/w'$. Since $v'/w'$ is optimal, it is implied that $v'/w' > v/w$ for all other pairs $(v, w)$. Assume that the current solution does not contain the full weight $w'$ in the knapsack. Then by replacing part of any other $w$ with the remaining portion of $w'$, the solution will be transformed into a more optimal solution because the value will be improved.

The latter property of optimal substructure is simple to prove. When we choose something to put into the knapsack, the item no longer is up for consideration and its weight is subtracted from the remaining weight. It is clear that we now have an *n - 1* knapsack problem with some maximum weight $W' < W$.

# 3 The 0-1 Knapsack Problem [3, 4, 5]

We now explore the *0-1* knapsack problem. Again, assume you are a thief. But this time, instead of stealing gold dust (or something else that you can take a fraction of), you are now stealing gold bricks (or something that you must take the whole item or none of it), but you can only steal the entire item, or nothing at all. You're pretty smart though and you watched a couple of good thief movies before you decided to steal anything.

This problem can thus be defined as follows:
We have have the following vectors:

$v = \{v_1, v_2, ..., v_n\}$, a vector containing each object's respective value

$w = \{w_1, w_2, ..., w_n\}$, a vector containing each object's respective weight

We want to maximize value, which takes the form:

$$\sum_{j=1}^{n} v_j x_j$$

Constrained by a weight *W*, such that:

$$\sum_{j=1}^{n} w_j x_j \leq M, x_j \in \{0,1\} \; \forall \; j \in \{1, ..., n\}$$

## 3.1 The Greedy Thief, or Why the Greedy Approach Fails

In the first movie, you watched a really greedy thief. He/she had a knapsack that can hold weight *W = 8 lbs*. There were five items to steal. The respective vectors look like:

*v = {2, 2, 5.5, 2, 2}*
*w = {2, 2, 5, 2, 2}*

Since the thief was greedy, he sorted the same way as in the fractional case, and he wound up with a knapsack of *{5,2}*, for a total value of 7.5 and weight of 7.0. This, of course, is not optimum, as the optimum solution would be *{2,2,2,2}*, for a total value of 8.0. The greedy thief is incapable of seeing beyond *unit* value, which causes him/her to pick items which may not optimally use the available weight. The thief makes decisions based on a single unit value, pounds for example, but the base units change based on item (since taking all or none is required). He did, however, escape and live to steal another day.

## 3.2 The Brute Thief, or Why the Enumeration Approach Fails

In the second movie, you watched a real brute thief. Besides being a brute, he was also incredibly smart, and he wanted to think about what he could take. In the movie, he also had a knapsack that was able to hold weight *W = 8 lbs*. This time though, there were also 8 items to hold in the knapsack, each which weighed various pounds, with various values.

The brute, being the thinker that he is, wanted to determine the optimal solution to steal. So he laid out his bag and decided to see which combinations of items to take. In this example, let *0* define not taking an item, and *1* define taking an item. After putting in an item, he can check the weight and value. He can check the weight to make sure it is within the constraint, and he can check to see if this is his highest stored value.

Brute's Knapsack before selecting any items:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Brute's Knapsack with selecting the first item:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Brute's Knapsack with selecting the first two items:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Brute's Knapsack with selecting the second item:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

And so on. Obviously, this thief was super smart, so we wanted to enumerate through every single possible combination to make sure he maximized his money. In the movie, the thief ends up getting arrested because it took way too long for him to enumerate all the possibilities for what to steal. In fact, in the example, there are $2^n$ possibilities, which results in $O(2^n)$ time... not quite a walk in the park to solve, especially as n increases.

But you're smart, and you know about Dynamic Programming.

## 3.3 Dynamic Programming [1, 2]

So what is this dynamic programming and why is it so awesome? Dynamic programming, similarly to divide-and-conquer, is a method to solve problems by combining solutions of subproblems. Unlike divide-and-conquer, which solves disjoint subproblems, dynamic programming is used when the subproblems overlap. Dynamic programming is generally used to solve optimization problems, as noted above.

In order to use dynamic programming on a problem, a problem must first have two attributes: **Overlapping subproblems**, and **Optimal Substructure**.

### Overlapping Subproblems

A problem has overlapping subproblems if it can be broken down into subproblems which are reused multiple times, and are reused more than once. An example of this would be recursive Fibonacci sequence, but not factorial.

### Optimal Substructure

A problem has optimal substructure if the globally optimal solution can be constructed from locally optimal solutions to subproblems, which can be solved independently.

In general, dynamic programming is used to solve subproblems only once, saving the solution. Then, when that subproblem answer is needed, it can be looked up, rather than recomputed. Dynamic programming uses extra memory in order to reduce computation time (it is a *time-memory trade off*). This time-memory trade off can result in exponential-time solutions being changed to near polynomial-time.

Dynamic problems are generally solved in one of two ways: top down or bottom up. In top down, the problem is solved recursively with each recursive solution being saved, or **memoized**. Then, within the method, the code first checks to see if whether that particular subproblem has been solved already or not. If it is, then use it. If not, then compute it. With bottom up, a "size" of the subproblem has to be established. With size established, we first sort the subproblems by size and then solve them in order, from smallest to largest. As larger problems are solved, their solutions are the combinations of previously solved subproblems, which can be looked up. Both bottom-up and top-down result in the same asymptotic times.

### Developing a Dynamic Programming Algorithm

There are four basic steps to developing a dynamic programming algorithm.
*Step 1. Structure*: Characterize the structure of an optimal solution: decompose the problem into smaller problems.
*Step 2. Principle of Optimality*: Recursively define the value of an optimal solution in terms of solutions to smaller problems.
*Step 3. Bottom-up (or top-down) Computation*: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.
*Step 4. Construction of Optimal Solution*: Construct an optimal solution from computed data.

## 3.4 Thieving like a Boss with Dynamic Programming [1, 6, 7]

We can now follow the development steps above to create a dynamic programming algorithm for the Knapsack problem.

### Step 1: Structure

We construct an array $c[0..n][0..W]$.
For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry *c[i][w]* will store the maximum (combined) of any subset of items *{1, 2, ..., i}* of combined size at most *w*.

If it is possible to compute every entry in this array, then the array entry *c[n][W]* will contain the maximum value of items that can fit in the knapsack, which is the solution to our problem.

### Step 2: Principle of Optimality

Initially, we want to set:
*c[0][w]* = *0* for $0 \leq w \leq W$, which represents no item is present
*c[i][w]* = $-\infty$ for *w* < *0*, which represents an item with negative weight (not allowed)

Our recursive step is:
$c[i][w] = max(v_i + c[i-1][w-w_i], c[i-1][w])$
For:
$1 \leq i \leq n, 0 \leq w \leq W.$

*Proof of correctness for computing c[i][w]:*

*Lemma:*
For:
$1 \leq i \leq n, 0 \leq w \leq W,$
$c[i][w] = max(v_i + c[i-1][w-w_i], c[i-1][w])$

Proof:
There are only two things that we can do with item i:
**Leave** it or **take** it.
The "take it or leave it" principle is really the key to understanding why the dynamic programming solution is going to work, which is the optimal substructure. That is, we reduce the problem of filling the knapsack to one of deciding for each item whether to take it or leave it, and we can build up the final solution by solving this problem incrementally for each item.

**Leave** it: The best we can do with item {1, 2, ..., i-1} and the weight $w$ is $c[i-1][w]$.
**Take** it: So long as $w_i \leq w$, then we are able to gain value $v_i$, but we have used $w_i$ weight of the knapsack. The best we can do with the remaining items {1, 2, ..., i-1} and weight ($w-w_i$) is $c[i-1][w-w_i]$. In total we result in $v_i + c[i-1][w-w_i]$.

## Step 3: Bottom-up (top-down) Computation *c[i][w]* using iteration, not recursion

The bottom is:
$c[0][w] = 0, \forall\, 0 \leq w \leq W$

The bottom-up (top-down) computational table is calculated using the recursive step from step 2:
$c[i][w] = max(v_i + c[i-1][w-w_i], c[i-1][w])$
calculating each row by row.

Let's check out an example:
Let $W = 7$
and we have 3 items:

| ***i*** | 1 | 2 | 3 |
|---|---|---|---|
| $v_i$ | 10 | 40 | 30 |
| $w_i$ | 3 | 2 | 4 |

Initially, our table is filled with nothing (all *0*'s):

| ***c[i][w]*** | ***w = 0*** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|---|
| *i = 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

We then increment to *i = 1* for the first item:

| ***c[i][w]*** | ***w = 0*** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|---|
| ***i = 0*** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 |

Recall that *c[i][w]* is the maximum (combined) of any subset of items *{1, 2, ..., i}* of combined size at most *w*. Once we get to *w* of 3, we are able to hold one item, which is number 1, for a value of 10.

Next, we move to the second item.

| ***c[i][w]*** | ***w = 0*** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|---|
| ***i = 0*** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 |
| **2** | 0 | 0 | 40 | 40 | 40 | 50 | 50 | 50 |

Item 2 weighs 2, so we add it as soon as we get to *w* = 2. Once we get to *w* = 5, we are able to also add item 1, which weights an additional 3, for a total of value of 50.

Finally, we add the last item.

| ***c[i][w]*** | ***w = 0*** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|---|
| ***i = 0*** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 |
| **2** | 0 | 0 | 40 | 40 | 40 | 50 | 50 | 50 |
| **3** | 0 | 0 | 40 | 40 | 40 | 50 | 70 | 70 |

In this scenario, item 2 is still worth the most, at the smallest weight. So we add it at *w = 2*. We then continue on until we hit *w = 5*, in which we add item 1 again. However, when we get to *w = 6*, we remove item 1 and add item 3, which is worth more for its value. In total, we end up with value = 70, in the location *c[n][W]*.

It's worth noting that this table is *n* by *W* in size, which will be the additional space required for this dynamic programming algorithm.

We can choose to stop here if wanted. However, this only calculates the maximum value the thief can take. We can continue with step 4 if we want to determine what items to keep in order to achieve that optimal solution.

Pseudocode (for maximum value):

```
DYNAMIC-0-1-KNAPSACK(v, w, n, W)
```

```
  let c[0..n][0..W] be a new matrix
  for w = 0 to W
    c[0][w] = 0
  for i = 1 to n
    c[i][0] = 0
    for w = 1 to W
      if w_i ≤ w
        if v_i + c[i-1][w-w_i] > c[i-1][w]
          c[i][w] = v_i + c[i-1][w-w_i]
        else
          c[i][w] = c[i-1][w]
      else
        c[i][w] = c[i-1][w]
  return c[n][W]
```

Complexity is *O(nW)*, where *n* is number of items and *W* is the knapsack weight. This is, of course, due to the fact that the outer loop loops *n* times and the inner loop loops *W* times. This in interesting because the complexity is *O(nW)*, which is not solely based on the input size *n*, but also the weight. Many dynamic programming algorithms do not depend only on input size, although some do. The important point is that *W*, the value of an argument to the problem, appears in the running time, which is different from what we've seen in most algorithms so far.

## Step 4: Construction of Optimal Solution

In order to determine the subset of items that we need to keep, we can keep a boolean array with 1 meaning take the item or 0 meaning don't take the item.

Once we have the values of keep, we can then use them to determine the subset *T* of items which has the maximum value.

If *keep[n,W]* == *1*, then $n \in T$. We can then repeat this with *keep[n-1,W-w$_n$]*.

If *keep[n,W]* == *0*, then $n \notin T$. We can then repeat this with *keep[n-1,W]*.

Pseudocode (including which to keep):

```
DYNAMIC-0-1-KNAPSACK-FULL(v, w, n, W)
  let c[0..n][0..W] be a new matrix
  let k[0..n][0..W] be a new matrix
  for w = 0 to W
    c[0][w] = 0
  for i = 1 to n
    c[i][0] = 0
    for w = 1 to W
      if w_i ≤ w
        if v_i + c[i-1][w-w_i] > c[i-1][w]
          c[i][w] = v_i + c[i-1][w-w_i]
          keep[i][w] = 1
        else
          c[i][w] = c[i-1][w]
          keep[i][w] = 0
      else
        c[i][w] = c[i-1][w]
        keep[i][w] = 0
  K = W
  for (i = n down to 1)
    if (keep[i][K] == 1)
      output i
      K = K - w[i]
  return c[n][W]
```

Again, the complexity is *O(nW)*, where *n* is number of items and *W* is the knapsack weight (no differences here).

## 5 Alternative Knapsack Problems [12]

- Bounded knapsack problem:
  - upper bound on number of times j can be selected
  - OR lower bound on number of times j can be selected

- Unbounded (integer) knapsack problem:
  - no upper bound on the number of times an item can be selected
  - NP-Complete
    - 1975, Lueker

- Multiple-choice KS problem:
  - subdivide items into classes and one item from each class must be taken

- Subset-Sum problem is special case:
  - profits and weights are identical

- Multiple KS problem:
  - n items and m KS's with capacities Wi

- Multiple-constraints KS problem:
  - Multiple constraints (such as volume)
  - multiple variants
  - Some of these may be NP-complete

## References

[1] Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. Introduction to Algorithms. MIT Press, Third Edition, 2009.

[2] Farmer, Jesse. *Introduction to Dynamic Programming*. Accessed 21 March - 1 April 2012. Available from: http://20bits.com/articles/introduction-to-dynamic-programming/.

[3] Grimson, Eric and John Guttag. *Debugging, Knapsack Problem, Introduction to Dynamic Programming*. Video lecture from Introduction to Computer Science and Programming, MIT, Fall 2008. Available from: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-12/.

[4] Grimson, Eric and Professor John Guttag. *Dynamic Programming: Overlapping Subproblems, Optimal Substructure*. Video lecture from Introduction to Computer Science and Programming, MIT, Fall 2008. Available from: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-13/.

[5] Grimson, Eric and Professor John Guttag. *Analysis of Knapsack Problem, Introduction to Object-Oriented Programming*. Video lecture from Introduction to Computer Science and Programming, MIT, Fall 2008. Available from: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-14/.

[6] Kleinberg, Joe and Eva Tardos. Algorithm Design. Addison-Wesley, 2006.

[7] Otten, Ralph. *The Knapsack Problem*. Lecture notes from Combinatorial Algorithms, Technische Universiteit. Accessed 21 March - 1 April 2012. Available from: http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf.

[8] Subramani, K. *Fractional Knapsack*. Lecture notes from Analysis of Algorithms, West Virginia University, Fall 2005. Accessed 21 March - 1 April 2012. Available from: http://www.csee.wvu.edu/~ksmani/courses/fa05/aoa/qen/fk.pdf.

[9] Wu, Dekai. *Greedy Algorithms*. Lecture notes from Design and Analysis of Algorithms, The Honk Kong University of Science and Technology, Spring 2005. Available from: http://www.cse.ust.hk/~dekai/271/notes/L14/L14.pdf.

[10] Wikipedia. *Combinatorial Optimization*. Accessed 21 March - 1 April 2012. Available from: http://en.wikipedia.org/wiki/Combinatorial_optimization.

[11] Wikipedia. *Dynamic Programming*. Accessed 21 March - 1 April 2012. Available from:

http://en.wikipedia.org/wiki/Dynamic_programming.

[12] Wikipedia. *List of Knapsack Problems*. Accessed 21 March - 1 April 2012. Available from: http://en.wikipedia.org/wiki/List_of_knapsack_problems.

[13] Wikipedia. *Knapsack Problem*. Accessed 21 March - 1 April 2012. Available from: http://en.wikipedia.org/wiki/Knapsack_problem.

[14] Wikipedia. *Memoization*. Accessed 21 March - 1 April 2012. Available from: http://en.wikipedia.org/wiki/Memoization.

[15] Wikipedia. *Subset Sum Problem*. Accessed 21 March - 1 April 2012. Available from: http://en.wikipedia.org/wiki/Subset_sum_problem.