A Knapsack Problem
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm
Room A202 Langley Hall

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/knapsack/knapsack.pdf

---

Take what you can and leave the rest!

The 0/1 Knapsack Problem

- *The knapsack problem is an abstract version of many realistic scheduling and packing problems;*
- *A museum contains many items worth stealing;*
- *Each item has a weight and value;*
- *A thief has a knapsack with limited weight capacity;*
- *The thief's goal is to maximize the value of the items he fits into his knapsack;*
- *So for each item, he needs to decide to take (1) or not take (0);*
- *The choices can be represented by a 0/1 vector;*
- *A brute force algorithm allows us to determine the best choice;*
- *A greedy algorithm may be much faster, though inaccurate*
- *A dynamic algorithm should give a better estimate of the best plan.*
- *Python's* `itertools()` *will help us carry out the brute force algorithm.*

1 Optimal Thievery

The knapsack problem begins with a set of n portable items of different weights and values, which we can think of as residing in a home or museum. Item `i` has weight w_i and value v_i. A thief breaks in, and wants

to make the best haul possible. Because the thief's knapsack has a weight limit of k, not all items can be stolen. The thief must make a list or choice vector, called `s`, so that if item i is to be taken, `s[i]` will be 1, otherwise it will be 0. The thief's gain from this robbery will be the sum of the values of the items that were stolen.

The total value of stolen items is known as the *objective function*. The weight limit k on the knapsack is known as a *constraint*.
Mathematically, the knapsack problem seeks a choice vector `s` which maximizes the objective function subject to the constraint.

2 Practice: Rob a Home

The thief breaks into a private home, with a knapsack that can hold no more than $k = 20$ pounds, and faces the following set of portable items:

Index	Item	Value ($)	Weight (lb)	Density ($/lb)
0	Clock	175	10	17.5
1	Painting	90	9	10
2	Radio	20	4	5
3	Vase	50	2	25
4	Book	10	1	10
5	Computer	200	20	10

The burglar might start by ranking all the items, perhaps by

1. Highest value: Computer, Clock, Painting, Vase, Radio, Book.
2. Lowest weight: Book, Vase, Radio, Painting, Clock, Computer.
3. Highest density: Vase, Clock, Painting, Book, Computer, Radio.

The burglar chooses items according to the ranking, stopping before the weight limit is exceeded:

Ranking	Selection S	Total Value	Total Weight
Value	(0,0,0,0,0,1)	$200	20
Weight	(0,1,1,1,1,0)	$170	16
Density	(1,0,1,1,1,0)	$255	17

If the values are in the vector v, and the weights in w, and the chosen items are listed in the 0/1 vector s, then

- The total number of chosen items is $\sum_{0 \le i < n} s_i$;
- The total value of chosen items is $\sum_{0 \le i < n} s_i v_i$ or `np.dot(s,v)`;
- The total weight of the chosen items is $\sum_{0 \le i < n} s_i w_i$ or `np.dot(s,w)`;

so the problem can be described computationally as

Problem 1 (Knapsack) *Find a 0/1 vector s maximizing $V = \sum_{0 \le i < n} s_i v_i$ subject to $W = \sum_{0 \le i < n} s_i w_i \le k$.*

3 Algorithm: Random sampling

Of course, our sample problem is very small, but we can already see that there are many options. If we are faced with a big problem, one approach would be to randomly select a choice vector s, and if it satisfies the weight constraint, compute the resulting value. Repeat this many times, and remember the selection vector s corresponding to the highest value. It's not very mathematical, but it does give a start on the problem.

```
Initialize Vmax to 0.
Initialize smax to 0.

Repeat
  Pick a random 0/1 vector $s$
  Compute W = s*w
  If W <= k
    Compute V = s*v.
    if Vmax < V
      set Vmax = V
      set smax = s.

If more cases to run, go back to Repeat
Report Vmax, smax.
```

How do we generate a random n-vector s of 0's and 1's? The `numpy.random()` library provides a function `choice()` that randomly selects n values from any given set.

The following code shows how to request a vector of length 6 containing a random choice of values from the set [0,1]:

```
from numpy.random import default_rng
rng = default_rng()
s = rng.choice ( [0,1], size = 6 )
```

So with these ideas, you should have enough information to write a program which generates a random choice vector s to take some of the 6 items, determines if s satisfies the weight constraint, and if so computes the value. Do this 20 times, and report the highest value you observe, and the corresponding choice vector s.

4 Algorithm: An upper bound for the profit

If we run random sampling, we can expect that from time to time, our best profit increases. In a sense, we have a lower limit on the very best profit. But can we know whether our random samples have gotten close to the upper limit, the very best profit?

If we are willing to relax the rules slightly, there is a way to get a reliable upper bound on the best profit that is achievable. To do this, we will pretend for the moment that every item is divisible, somewhat like a powder. Instead of the 0/1 choice (nothing or everything) we will suppose we could take any fractional amount r of each item, where, of course, $0 \leq r \leq 1$. This version of our problem is known as the *rational knapsack problem*. Luckily, there is a known solution and it is easy to compute.

Now let's think of our items in terms of their density, that is, the ration of value to weight. If the items were actually divisible, it must be true that the best strategy is to take as many whole items as possible of the highest density, until no more whole items fit. Then the remaining space in the knapsack is to be completely filled by whatever fraction of the next densest item.

Why is this guaranteed to be the best selection? If we were to replace x pounds of an item in the knapsack by x pounds of something not in the knapsack, we are replacing something of high density by something of lower density, and our profit would go down.

Because we are breaking the 0/1 constraint, this does not represent a solution of the original problem. And under the original rules (only whole items) the preference for highest density first is not guaranteed to achieve the highest profit. Can you come up with a simple example that shows this?

Stiil, no solution of the original problem can have a higher profit than what we see with the rational knapsack problem. This result is worth computing so that we know in advance the ceiling on our profit.

To solve the rational knapsack problem, a code returns the choice vector s and the maximum profit `Vmax`, doing something like this:

```
Given: a weight limit k
Given: n values v[] and weights w[]

Compute the density d = v / w
Sort the objects in decreasing order of d
Initialize wmax = 0
Initialize s = zeros(n)

For 0 <= i < n
  wmax = s * w
  If wmax + w[i] <= k
    s[i] = 1
  else
    s[i] = ( k - wmax ) / w[i]
    break

Vmax = s * v

Output: s and Vmax
```

5 Algorithm: Check Every Possible List

With random sampling, we're never sure we've checked every possibility. For any knapsack problem, there are only finitely many different choice vectors s and so one option might be simply to try every possible choice. In this case, we have given up thinking of clever ways to find a solution!

The first question is: *can we generate every possible choice vector s?*

The generation of every possible choice vector is actually something that you can program. It's just like counting.

```
Initialize s = zeros(n)
Initialize p = n - 1 (the last index in s)
We want to add 1 to s[p].

Begin Loop

  If s[p] is 0,
    s[p] = 1
    our new vector s is done, exit loop

  Otherwise,
    s[p] = 0
    if p == 0,
      s is entirely 0 again,
      there are no more vectors.
      exit loop
    Otherwise
      set p = p - 1
      loop again
```

Try to program this algorithm, and verify that it can generate all the choice vectors for $n = 6$.

The second question is: *is it practical to do this for moderately large values of n?*

The answer to this question is, sadly, no! For n items, the number of different choice vectors is 2^n. For $n = 10$ items, this is $2^10 = 1,024$, and for 20 it is $2^20 = 1,048,576$. Each increase of 10 items multiplies the number of choices by 1,000. If we want to consider problems with, say, $n = 100$ items, we will never be able to handle the necessary work. So a brute force method gives certainty, but we can only afford this for a limited range of values of n.

Use your counting algorithm to write a program that uses brute force to find the maximum profit for the 6 item problem.

6 Algorithm: Greedy Grabbing

Surely we can do better than random sampling! Let's suppose we rank the items, either by high value, low weight, or high density. In that case, a greedy strategy would be to start by taking the first ranked item, and then the next and the next, continuing until the weight constraint stops us or we run out of items.

We start with data v, w, and $d = v/w$. We need to sort one of these arrays, and then rearrange all of them accordingly. The way to do this is with an index vector. To get a ranking of all the values v, from lowest to highest,

```
index = np.argsort ( v )
```

To reverse the index vector so it goes from highest to lowest,

```
index = np.flip ( index )
```

To rearrange all three arrays according to this ordering

```
  v = v[index]
  w = w[index]
  d = d[index]
```

We would handle density the same way. But if we were ranking by weights, we want to choose the smallest weight first, so we would omit the call to `np.flip()`.

Once we have our data reordered, we simply choose each item in order until we hit the weight limit. That's our greedy estimate of the solution.

7 Algorithm: Dynamic Programming

Dynamic programming is a technique which gradually builds an optimal solution by working first with smaller problems, and then cleverly connecting them. In the knapsack case, the dynamic programming approach requires that the weight constraint k is an integer. Then it constructs a table $M(*,*)$ whose rows correspond to weight constraints $0, 1, ..., k$, and whose columns correspond to no items, item 1, items 1 and 2, ..., all items. Entry $M(i, j)$ represents the maximum profit we expect to obtain, assuming we use items 0 through i, and have weight constraint equal to j. If we can fill out this table correctly, then $M(n, k)$ contains our desired maximum profit.

To keep things simple, let's just look at how a dynamic programming code would work for our knapsack problem.

```
def knapsack_dynamic_table ( v, w, k ):

  import numpy as np
```

```
  n = len ( v )

  m = np.zeros ( [ n + 1, k + 1 ] )
#
#  Consider object i from 0 to n-1
#
  for i in range ( 0, n ):
#
#  Consider the weight limit j from 0 through k.
#
    for j in range ( 0, k + 1 ):
#
#  If item i weighs more than j, then it can't be used.
#
      if ( j < w[i] ):
        m[i+1,j] = m[i,j]
#
#  If item i weighs no more than j,
#  then it could be used along with the best solution for weight j-w[i].
#  If this gives a better result, update.
#
      else:
        m[i+1,j] = max ( m[i,j], m[i,j-w[i]] + v[i] )
#
#  The value in m[n,k] is the highest profit using n items for a total
#  weight no more than k.
#
  return m
```

Notice that the value of $M(n,k)$ tells us the maximum profikt, but doesn't tell us how to get it! We would have to do more work to figure out the actual selection of items that was used. This is possible, but requires a little more programming than we want to look at now!

8 Test Data

Here is the data for a few small knapsack problems than you can test your code on.

- n is the number of items;
- v is the value of each item;
- w is the weight of each item;
- k is the weight limit on the knapsack;

```
n = 5
v = [  24,  13,  23,  15,  16 ]
w = [  12,   7,  11,   8,   9 ]
k = 26

n = 6
v = [  50,  50,  64,  46,  50,   5 ]
w = [  56,  59,  80,  64,  75,  17 ]
k = 190

n = 7
v = [  70,  20,  39,  37,   7,   5,  10 ]
w = [  31,  10,  20,  19,   4,   3,   6 ]
W = 50

n = 8
v = [ 350, 400, 450,  20,  70,   8,   5,   5 ]
w = [  25,  35,  45,   5,  25,   3,   2,   2 ]
```

```
k = 104

n = 10
v = [ 505, 352, 458, 220, 354, 414, 498, 545, 473, 543 ]
w = [  23,  26,  20,  18,  32,  27,  29,  26,  30,  27 ]
k = 67

n = 10
v = [  92,  57,  49,  68,  60,  43,  67,  84,  87,  72 ]
w = [  23,  31,  29,  44,  53,  38,  63,  85,  89,  82 ]
k = 165
```