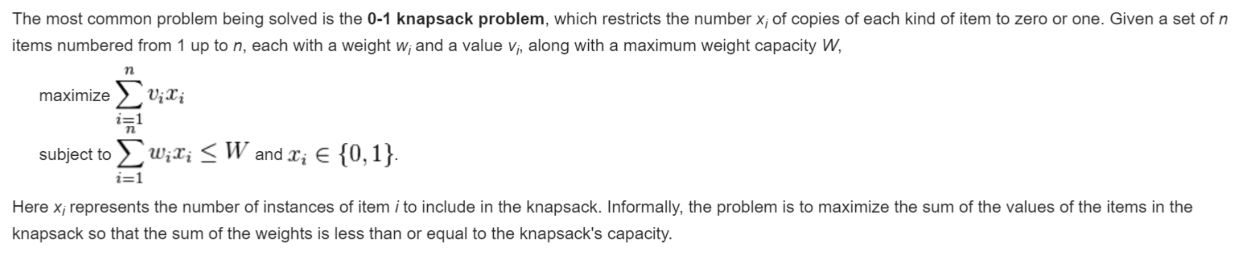
Discrete Optimisation

Report: Assignment 2 – Knapsack

The task:

Solve the 0/1 knapsack problem for 5 different problem sets.



The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i 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 w_i and a value v_i, along with a maximum weight capacity W,

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}.$$

Here x_i 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.

Figure 1: source https://en.wikipedia.org/wiki/Knapsack_problem

The different algorithms

Three different methods have been used:

- Dynamic programming with a bottom up approach
- Dynamic programming with a top down approach (memoization)
- Branch and bound (with depth-first and best-first)

An explanation of the different algorithms can be found here:
http://www.micsymposium.org/mics_2005/papers/paper102.pdf

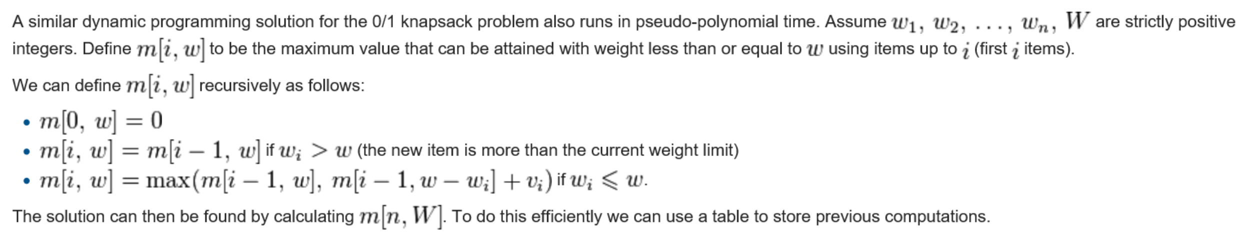
The following tables sums up the complexity of the different tasks and the performance of the algorithms. (Dell xps 13 intel i5 8gb ram)

Method	Complexity	Task1	Task2	Task3	Task4	Task5
num_items and size		n=30 S= 10^5	n=200 S=10^5	n=400 S=10^7	n=100 S=10^5	n=10000 S=10^6
Dynamic programming	O(n*S)	3 10^6	2 10^7	4 10^9	10^7	10^10
Time BU (s)		9.452	69.029	NA	NA (time)	NA
BU sol		99798	100236	NA	NA	NA
Time TD (s)		**0.048**	**4.611**	NA	NA(memory)	NA
TD sol		99798	100236	NA	NA	NA
Branch and bound	O(2^n) (Worst case)	2^30	2^200	2^400	2^100	2^10000
BB time (s)		0.388	824.506	**9.463**	5.616	140.681

BB sol		99798	100236	3967180	109899	1099893
BBPQ time(s)		1.786	NA	23.25	**0.872**	**31.227**

Dynamic Programming – Bottom Up

We use the following recursion relation:



A similar dynamic programming solution for the 0/1 knapsack problem also runs in pseudo-polynomial time. Assume $w_1,\ w_2,\ \ldots,\ w_n,\ W$ are strictly positive integers. Define $m[i,w]$ to be the maximum value that can be attained with weight less than or equal to w using items up to i (first i items).

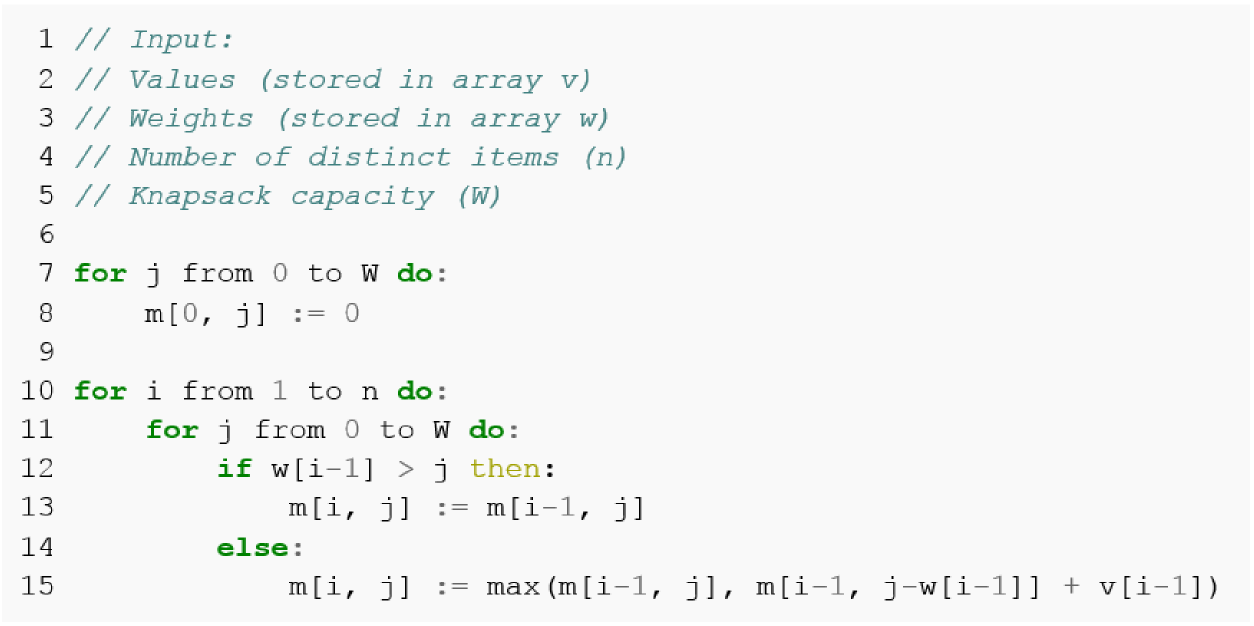
We can define $m[i,w]$ recursively as follows:

- $m[0,\ w] = 0$
- $m[i,\ w] = m[i-1,\ w]$ if $w_i > w$ (the new item is more than the current weight limit)
- $m[i,\ w] = \max(m[i-1,\ w],\ m[i-1, w-w_i] + v_i)$ if $w_i \leqslant w$.

The solution can then be found by calculating $m[n,W]$. To do this efficiently we can use a table to store previous computations.

Figure 2 source https://en.wikipedia.org/wiki/Knapsack_problem

We start from the bottom and work our way up, using a table to store previous values so as to be able to call them without recalculating. The solution is based on the following pseudo-code:



```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)

for j from 0 to W do:
    m[0, j] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i-1] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i-1]] + v[i-1])
```

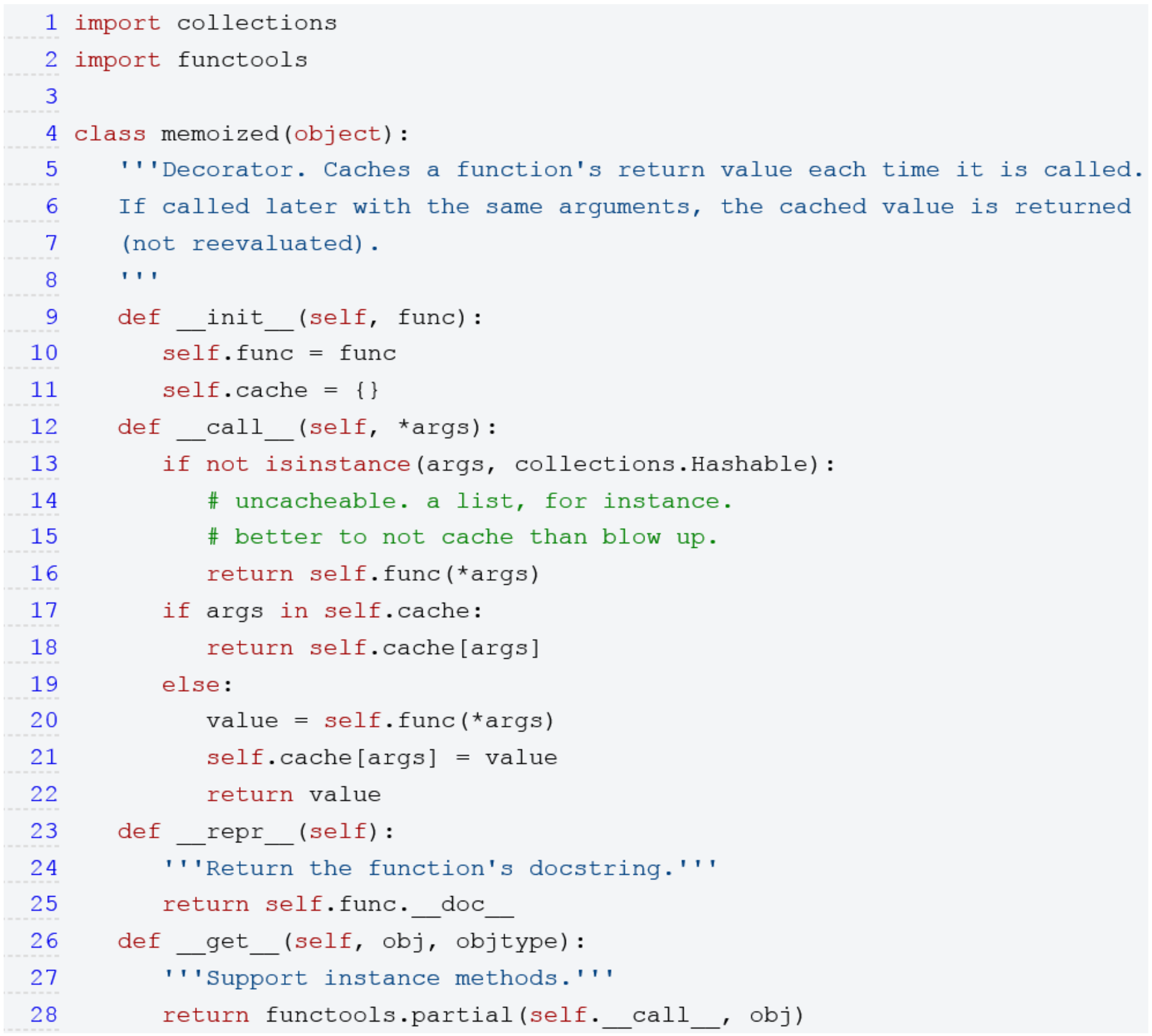
Figure 3 source https://en.wikipedia.org/wiki/Knapsack_problem

The complexity of the problem is O(n*S), with n the number of items, and S the size of the knapsack. It can thus be solved in pseudo polynomial time (polynomial in the numeric value of the input, but exponential in the length of the input).

Dynamic Programming – Top Down

The top down algorithm is similar to the bottom up in principle, but as it starts from the top and uses recursion, it only calculates the values of m required. It is therefore computationally more efficient. However the recursion is memory intensive and can quickly lead to recursion limit errors.

The technique used is memorization. When a value of m is calculated, it is cached, and therefore can quickly be retrieved if it is needed in another branch of the recursion calculation. To implement this we have used a python decorator called memoize. Here is the source code:



```python
import collections
import functools

class memoized(object):
   '''Decorator. Caches a function's return value each time it is called.
   If called later with the same arguments, the cached value is returned
   (not reevaluated).
   '''
   def __init__(self, func):
      self.func = func
      self.cache = {}
   def __call__(self, *args):
      if not isinstance(args, collections.Hashable):
         # uncacheable. a list, for instance.
         # better to not cache than blow up.
         return self.func(*args)
      if args in self.cache:
         return self.cache[args]
      else:
         value = self.func(*args)
         self.cache[args] = value
         return value
   def __repr__(self):
      '''Return the function's docstring.'''
      return self.func.__doc__
   def __get__(self, obj, objtype):
      '''Support instance methods.'''
      return functools.partial(self.__call__, obj)
```

Figure 4 source https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize

Branch and bound

Branching is essentially iterating through all the possible solutions. Each item constitutes a choice: either we take it or we don't. This results in 2^n possibilities, with n the number of items. To avoid an algorithm that grows exponentially in complexity, we bound the solution. This means pruning branches of the tree of solutions that cannot hold the optimal solution

The idea is to define an upper bound for a node and all of its successors. Then when exploring the tree of solutions, if the upper bound for a node is inferior to the value of the current best found solution, we can prune the node and the branch it forms, thus avoiding exploring it.

The heuristic for the upper bound comes from relaxing the constraint on the integrality of taking an item or not. By converting the problem into a continuous one, i.e. allowing to take fractions of an item, we make the problem much easier. The solution to the continuous problem is an upper bound to the integer one, because the set of solutions of the integer problem is included in the set of problems of the continuous one.

To solve the continuous problem, we sort the items by value to size ratio, then we fill the knapsack as much as possible. We then complete the knapsack by taking the exact fraction of the next item in the list needed to reach the total capacity. This gives us an upper bound for the value of a node and its sons. We can round this value down to the closest integer because the problem we are solving is an integer one.

The main algorithm is for exploring the possible solutions. We sort the items by their value to size ratio to start. We use a queue to store the node to explore. On each iteration, we pop the first item of the queue. We then check its left and right sons (i.e. taking the next item and not taking it). If their value is better than the value of the current best solution, we update the latter. Then we calculate their upper bounds (N.B. the upper bound of the left node is the same as its parent), and if the upper bounds are higher than the value of the current best node, we add the nodes to the queue. If not, the nodes and the branches they form are pruned.

By using a priority queue, we can implement a best first search, and by using a normal queue (LIFO) we can implement a depth first search.

Also note that is the upper bound is an integer, then we have solved the integer problem for that particular node. There is therefore no need to explore the offspring of the node as the optimal solution has already been found for the given configuration common to the parent node and its offspring.

Conclusions:

Each different task required a different method for optimal performance. The dynamic programming approaches were particularly quick on tasks 1 and 2 (where n*S was smallest), but were too slow for the other tasks. Additionally, the memorization method, although very fast, is actually limited by the maximum recursion limit authorized by the system.

The branch and bound techniques were more efficient on large data sets. However it should be noted that branch and bound is partly luck (and choice of heuristic and search pattern). In the worst case it is still very much of exponential complexity. It should be noted that for problem 2, the resolution was particularly slow, even though the problem size wasn't that big.

As for branch and bound, depth first search and best first search give different run times, best first being faster on tests 4 and 5 while depth first outperforming on tasks 1, 2 and 3. Again this is dependent on the problem configuration and is partially luck of the draw.