Problem:

- We have a list of products (*products.csv*) and a container with a fixed volume (V = A*B*C):
  - We want to fill the container with the products that are fitting the container (each product has some fixed volume dimensions and them should be equal or smaller that the container 's volume dimensions) in a way that is maximizing the total dollar value (each product has some fixed dollar value):
    - A product is added to the container or not is added (i.e. it is not allowed to add product 's fractions)
    - The product 's volume dimensions and container 's volume dimensions are integers
    - A given product can be added once to the container (i.e. we cannot add a product twice or more times to the container)
    - For equal total dollar value, we want to minimize the total weight (i.e. each product has some fixed weight)

Recurrence:

- The force brute solution (searching for all the possible combinations of products and taking the one with maximum value - and minimum weight for equal value-) has exponential time (2^I), so in order to resolve the problem in polynomial time (I*V) we are using the *Dynamic Programing* meta-technique. It means expressing the problem as a recurrence based in small (and probably overlapping -we are using memoization in order to take advantage of that-) subproblems:
  - Notation:
    - *[DV(i,v) , c]* :
      - *DV(i,v)* = current dollar value
      - *i* = the product at the current index *i*:
        - where the products are indexed from 1 to I:
          - the products were filtered from *products.csv* (in the same order that are appearing there) in order to fit the container:
            - choose i if ($length_i <= A$) and ($width_i <= B$) and ($height_i < C$)
      - *v* = current maximum volume:
        - where it is indexed from *0* to *V* (where V= A*B*C)
      - *c* = the current optimal container with the selected products
  - *[DV(i,v), c]* =:
    - **1.** *[DV(i-1, v), c]* if $volume_i > v$:
      - i.e. we are not adding the *i* to the optimal container because its volume is breaking the current restriction (i.e. $volume_i > v$):
        - So we are exploring only the subproblem where *i* is not included in the optimal container
    - **2.** max_{ *[DV[(i-1, v), c]* , *[DV(i-1, v' = v - $volume_i$) + DV_i, c = c+i]* }:
      - i.e. the current *i* can or could not be in the optimal container (i.e. there

isn't a problem of breaking the current restriction choosing *i*), so we need to explore both subproblems, and select the one with maximum *DV* (and the minimum weight in the case of equal *DV*):

- if $DV(i-1, v) = DV(i-1, v' = v - volume_i)$:

  if $total\_weight(c) <= total\_weight(c+i)$:
  choose $[DV[(i-1, v), c]$
  else
  choose $[DV(i-1, v' = v - volume_i) + DV_i, c = c+i]$

Recurrence with memoization:

- As there is overlapping in the subproblems, if we are solving the whole problem using *naive recursion*, then we are solving the same subproblem several times and the recursion will take a lot of time to complete, so the best is to use recurrence with memoization:
  - In this way we are filling a structure (i.e. we are storing the previous computed subproblems) from the bottom-up in way that it is always consistent with the recurrence:
    - It means that we should have computed earlier everything on what is dependent the current subproblem we are solving, so solving the current subproblem means we are only doing a constant amount of operation (i.e. we compute some simple numbers from the previous solved subproblems, and compare them to know which one is the best)