Lab 6: Implement a Knapsack problem using Brute Force Method and Dynamic Programming

In [1]:

```
from itertools import product
from collections import namedtuple
try:
    from itertools import izip
except ImportError:
    izip = zip
```

In [2]:

```
Reward = namedtuple('Reward', 'name value weight volume')

bagpack =   Reward('bagpack',  0, 25.0, 0.25)

items = [Reward('laptop', 3000,  0.3, 0.025),
         Reward('printer',   1800,  0.2, 0.015),
         Reward('headphone',    2500,  2.0, 0.002)]
```

In [3]:

```
def tot_value(items_count):
    """
    Given the count of each item in the sack return -1 if they can't be carried or t

    (also return the negative of the weight and the volume so taking the max of a se
    values will minimise the weight if values tie, and minimise the volume if values
    """
    global items, bagpack
    weight = sum(n * item.weight for n, item in izip(items_count, items))
    volume = sum(n * item.volume for n, item in izip(items_count, items))
    if weight <= bagpack.weight and volume <= bagpack.volume:
        return sum(n * item.value for n, item in izip(items_count, items)), -weight,
    else:
        return -1, 0, 0
```

In [4]:

```
def knapsack():
    global items, bagpack
    # find max of any one item
    max1 = [min(int(bagpack.weight // item.weight), int(bagpack.volume // item.volu

    # Try all combinations of reward items from 0 up to max1
    return max(product(*[range(n + 1) for n in max1]), key=tot_value)
```

In [5]:

```
max_items = knapsack()
maxvalue, max_weight, max_volume = tot_value(max_items)
max_weight = -max_weight
max_volume = -max_volume

print("The maximum value achievable (by exhaustive search) is %g." % maxvalue)
item_names = ", ".join(item.name for item in items)
print("  The number of %s items to achieve this is: %s, respectively." % (item_name
print("  The weight to carry is %.3g, and the volume used is %.3g." % (max_weight, m
```

```
The maximum value achievable (by exhaustive search) is 54500.
  The number of laptop, printer, headphone items to achieve this is:
(9, 0, 11), respectively.
  The weight to carry is 24.7, and the volume used is 0.247.
```

Implement a Knapsack problem using Dynamic Programming. Compare the execution time of brute-force and dynamic programming algorithms

Instead, use a technique known as dynamic programming, which is similar in concept to memoization. Instead of solving a problem outright with a brute-force approach, in dynamic programming one solves subproblems that make up the larger problem, stores those results, and utilizes those stored results to solve the larger problem. As long as the capacity of the knapsack is considered in discrete steps, the problem can be solved with dynamic programming.

In [6]:

```
def knapSack(W, wt, val):
    n=len(val)
    table = [[0 for x in range(W + 1)] for x in range(n + 1)]

    for i in range(n + 1):
        for j in range(W + 1):
            if i == 0 or j == 0:
                table[i][j] = 0
            elif wt[i-1] <= j:
                table[i][j] = max(val[i-1] + table[i-1][j-wt[i-1]],  table[i-1][j])
            else:
                table[i][j] = table[i-1][j]

    return table[n][W]
```

In [7]:

```
val = [50,100,150,200]
wt = [8,16,32,40]
W = 64

print(knapSack(W, wt, val))
```

```
350
```