General Input for All Algorithms

1. integer n specifying the N 'limit' per item
2. integer capacity specifying the knapsack weight limit
3. array of Items initialItems specifying the 'base' items
4. array of integers initialCounts specifying the maximum number of items at initialItems(k)

eg. To run N = 4, Capacity = 200, with items (4 reds with weight 10 and value 9, 10 blues with weight 50 and value 12 and 5 greens with weight 5 and value 2) do:

- specify N = 4 and capacity = 200
- make Items with weight and values (and put them in initialItems)
    - r = new Item ("red",9,10)
    - g = new Item ("green",2,5)
    - b = new Item ("red",12,50)
- specify 'pool' initialCounts (item counts)
    - initialCounts having {4,10,5} corresponding to 4 reds, 10 blues and 5 greens
- run as 'Contructor (4, 200, initialItems, initialCounts)' with Constructer corresponding to desired algorithm

Assumptions

1. n >= 1
2. capacity >= 0
3. there must be at least 1 Item in initialItems ('pool') and hence there must also be at least 1 integer in initialCounts
4. $\text{initialItems}_{0..\text{length}}$ does not have a NULL
5. initialItems and initialCounts must have the same size
6. arrays are 0-indexed and hence length of array is array.size - 1

Dynamic Programming

Algorithm

Output

1. number of each Item in optimal solution
2. totalWeight being the optimal weight of the knapsack

3. matrix[length of items][capacity] containing the optimal profit

Function

```
KnapsackONDP (n, capacity, initialItems, initialCounts)
        initialiseDependencies (initialItems, initialCounts)
                items = [ ]
                for i = 0 → length of initialCounts
                        for j = 0 → initialCounts[i] and j < n
                                add initialItems[i] to items

                matrix = [ ] [ ]
                keep = [ ] [ ]    //used for 'backtracking'

        solve (n, capacity)
                for col = 0 → capacity
                        matrix [0][col] = 0;

                for i = 1 → length of items +1
                        for j = 0 → capacity
                                if (items[i – 1].weight <= j) and (items[i – 1].value + matrix[i – 1][j – items[i – 1].weight] > matrix[i – 1][j])
                                        matrix[i][j] = items [i – 1].value + matrix[i - 1][j – items[i – 1].weight]
                                        keep[i][j] = 1
                                else
                                        matrix[i][j] = matrix[i - 1][j];
                                        keep[i][j] = 0;

                totalWeight = 0         //stores computed weight of items in knapsack
                K = capacity
                for i = length of items → 1
                        if keep[i][K] = 1
                                output items[i – 1]
                                add items[i – 1].weight to totalWeight
                                subtract items[i – 1].weight to K
                output totalWeight and matrix[length of items][capacity]
```

Correctness

Assume that for 1 <= i <= length of items, 0 <= j <= capacity, so matrix[i][j] = maximum(matrix[i – 1][j], items.value$_i$ + matrix[i – 1][j – items.weight$_i$]).

To compute matrix[i][j], we note that we only have 2 choices for row i;

1. Leave row i
   The best we can do with rows {1, 2, ……. i – 1} and capacity is matrix[i – 1][capacity]
2. Take row i
   (only possible if items.weight$_i$ <= capacity): Then we gain of items.value$_i$ computing time, but have spent items.weight$_i$ of storage. The best we can do with remaining rows {1, 2, ……. i – 1} and storage (capacity - items.weight$_i$) is matrix[i – 1][j – items.weight$_i$]. In total, we get items.value$_i$ + matrix[i – 1][j – items.weight$_i$].

   Note that if items.weight$_i$ > capacity then matrix[i – 1][j – items.weight$_i$] = – ∞ so the assumption is correct in any case

Complexity

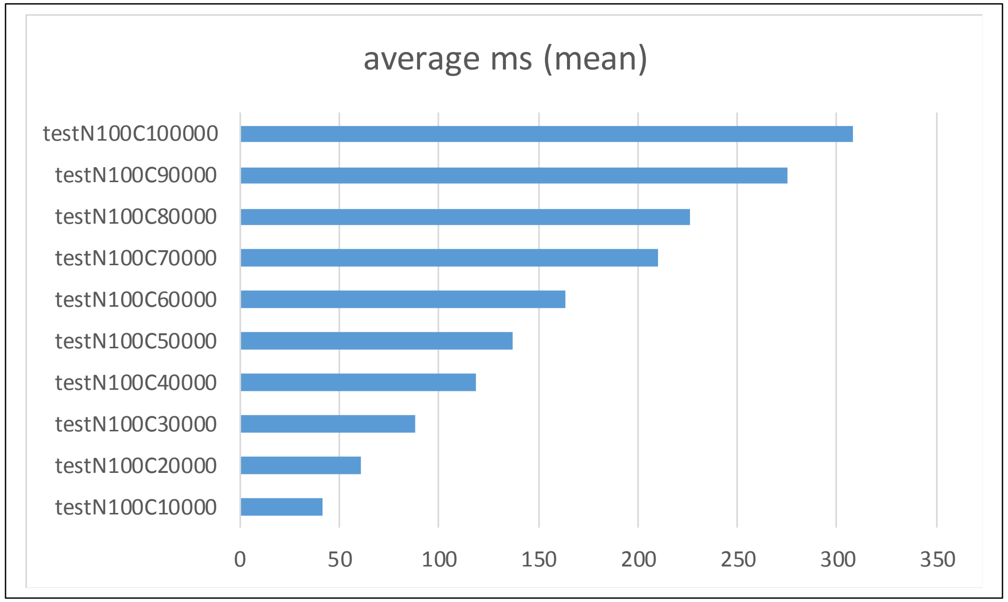
$$\sum_{i=0}^{number\ of\ items}\ \sum_{j=0}^{capacity} 1 = \sum_{i=0}^{number\ of\ items} [1+1+1+\cdots \ldots +1]\ \text{(capacity times)}$$

= capacity * [1+1+1+……+1] (repeat number of ‘items’ times)
= capacity * number of ‘items’
= Θ (capacity * number of ‘items’)
Thus, the complexity of the algorithm is Θ(length of ‘items’ * capacity). In terms of memory, this requires a 2 two-dimensional arrays with rows equal to the number of ‘items’ and columns equal to the capacity of the knapsack (one for matrix and one for keep).

Results

name	average ms (mean)
testN100Capacity10000	41.2
testN100Capacity20000	61.13333
testN100Capacity30000	88.26667
testN100Capacity40000	118.4
testN100Capacity50000	137.0667
testN100Capacity60000	163.8
testN100Capacity70000	210.1333
testN100Capacity80000	225.8667
testN100Capacity90000	274.9333
testN100Capacity100000	307.7333

Actual complexity seems to match predicted complexity. It linearly increases proportional to number of items and capacity



Brute Force, Enumerate All

Algorithm

Output

1. all other feasible solution 'sets'
2. number of each Item in optimal solution
3. maxWeight of the optimal weight of the knapsack
4. maxProfit of the optimal profit of items in the knapsack

Function

```
KnapsackONBF (n, capacity, initialItems, initialCounts)
        initialise (initialItems, initialCounts)
                solutions = [ ]
                items = initialItems;
                counts = initialCounts;

        solve (n, capacity)
                //for each combination of x,y,z
                for x = 0 → counts[0] and x <= n
                        for y = 0 → counts[1] and y <= n
                                for z = 0 → counts[2] and z <= n
                                        combi = computeValueAndWeight (items, itemCounts)     //value and weight of x,y,z
                                        if combi.weight <= capacity
                                                add combi to solutions
                                                output combi
                sort solutions in terms of combi.profit      //descending
                output solutions(first)         //get top combi in solutions

        computeValueAndWeight(items, itemCounts)
                value = 0, weight = 0
                for i = 0 → length of items
                        add 'items[i].value*itemCounts[i]' to value
                        add 'items[i].weight*itemCounts[i]' to weight
                return value and weight
```

Correctness

By virtue, because we would examine all combinations <= n then we know that somehow we would reach an optimal combination and the loop terminates because of n as upper bound.

Complexity

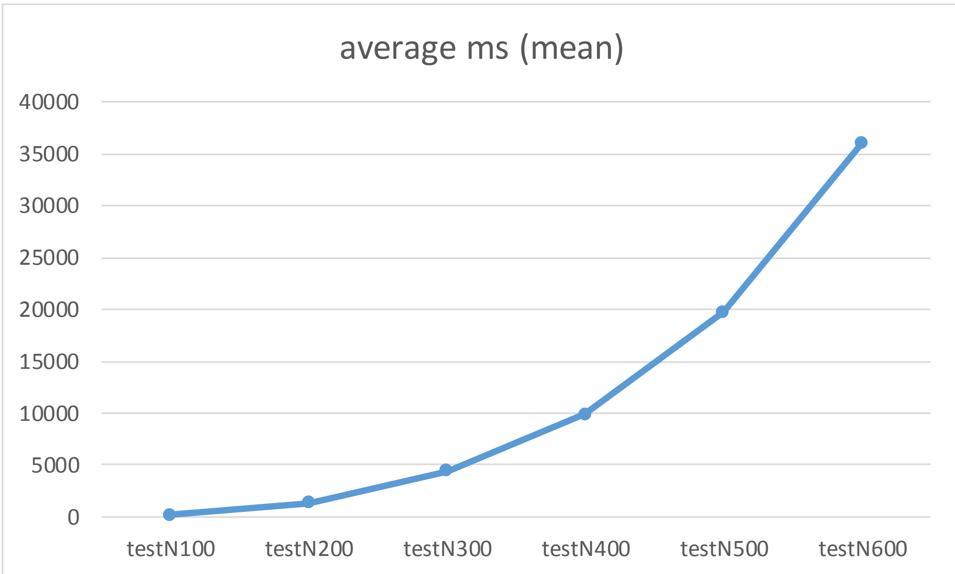
The algorithm is reducible to:

$$\sum_{i=1}^{2^n}\left[\sum_{j=n}^{1} + \sum_{k=1}^{n}\right] = \sum_{i=1}^{2^n}[\{1+..+1\}(\text{n times}) + \{1+..+1\}(\text{n times})] \text{ with n } = \sum_{i=0}^{2}\sum_{j=0}^{counts_i} j \times items_i \; such\ that\ j \le N$$

= (2n) * [1+1+1…+1] (2^n times)
= Θ ($2n*2^n$)
= Θ ($n*2^n$)
Therefore, the complexity of this algorithm is Θ ($n2^n$). Since the complexity of this algorithm grows exponentially, it can only be used for small instances. Otherwise, it does not require much programming effort in order to be implemented. The memory used to store all the feasible solutions of this algorithm requires an array (solutions)

Results

name	average ms (mean)
testN100	187.2667
testN200	1334.067
testN300	4406.733
testN400	9876.667
testN500	19705.8
testN600	36033.27



The actual complexity is as expected. It exponentially grows in proportion to the N component

Graph Search

Algorithm

Output

1. number of each Item in optimal solution
2. maxWeight of the optimal weight of the knapsack
3. maxProfit of the optimal profit of items in the knapsack

Function

```
Knapsack0NGS (n, capacity, initialItems, initialCounts)
    initialiseDependencies (initialItems, initialCounts)
        items = [ ]
        for i = 0 → length of initialCounts
            for j = 0 → initialCounts[i] and j < n
                add initialItems[i] to items

    solve (n, capacity)
        sort items according to value-to-weight ratio
        best = new node
        root = new node
        computeBound (items, capacity) for root
        pq = new priority queue
        enqueue root to pq

        while pq has a node in it
            current = dequeue node from pq
            if bound of current < value of best node and height of current < length of items
                //set the left child of the current node to include the next item
                left = new child node with current node as its parent
                item = items [height of current]
                add weight of item to weight of left child
                if weight of left child <= capacity
                    add items [height of current] to best solution
                    add item value to value of left child
                    computeBound (items, capacity) for left child
```

```
                    if value of left child > value of best node
                        //update best solution
                        best = left
                    if bound of left child > value of best node
                        enqueue left to pq

                right = new child node with current node as parent          //does NOT include next item in items
                computeBound (items, capacity) for right
                if bound of right child > value of best node
                    enqueue right to pq

        return best node containing the optimal items subset, weight and profit

Node ()
    computeBound (items, capacity)
        i = height of this node
        w = weight of this node
        bound = value of this node

        //loop at least once
        for count = 0 → 1 or i < length of items
            item = items[i]
            if w + weight of item > capacity
                stop loop
            add weight of item to w
            add value of item to bound
            increment i by 1
```

add $(capacity - w) \times \frac{value\ of\ item}{weight\ of\ item}$ to bound

Correctness

The algorithm constructs candidate solutions one component at a time and evaluates the partly constructed solutions. If no potential values of the remaining components can lead to a solution, the remaining components are not generated at all.

It is based on the construction of a 'state space tree'. A state space tree is a rooted tree where each level represents a choice in the solution space that depends on the level above and any possible solution is represented by some path starting out at the root and ending at a leaf. The root, by definition, has level zero and represents the state where no partial solution has been made. A leaf has no children and represents the state where all choices making up a solution have been made. In this context there are "$length\ of\ items$" possible items to choose from, then the kth level represents the state where it has been decided which of the first k items have or have not been included in the knapsack. In this case, there are 2^k nodes on the kth level and the state space tree's leaves are all on level "$length\ of\ items$".

In the state space tree, a branch going to the left indicates the inclusion of the next item while a branch to the right indicates its exclusion. The upper bound is computed by adding the cumulative value of the items already selected in the subset, and the product of the remaining capacity of the knapsack and the best per unit payoff among the remaining items.

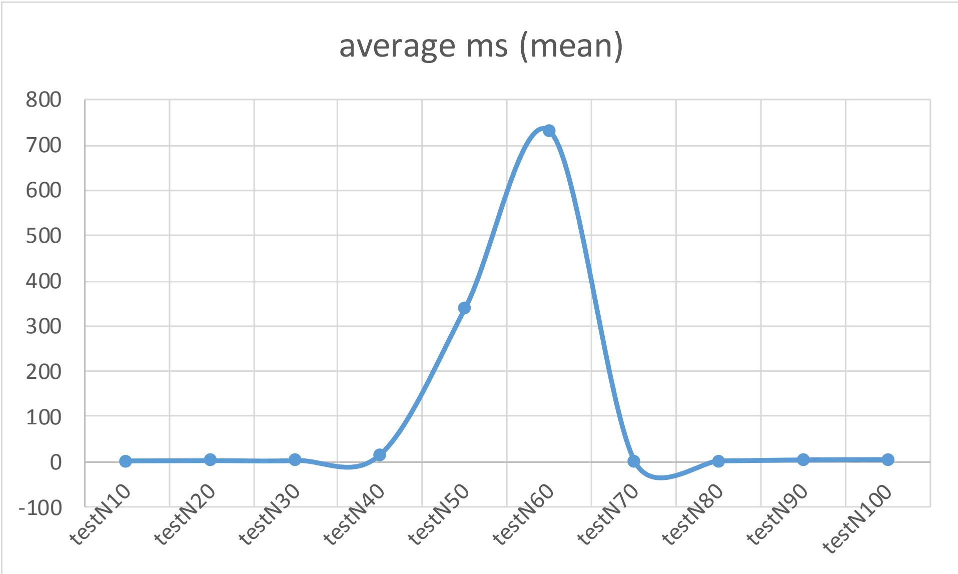
Complexity

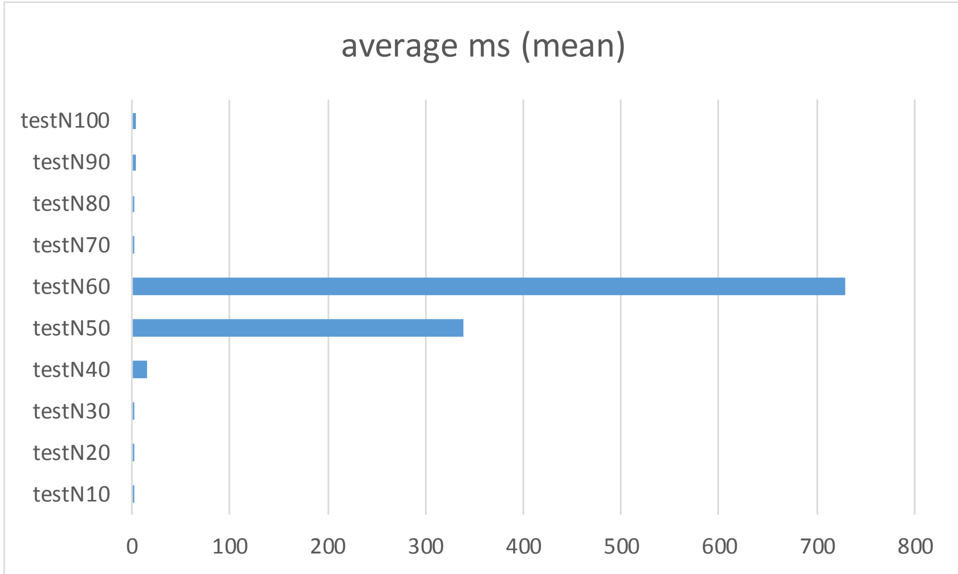
In the worst case, the branch and bound algorithm will generate all intermediate stages and all leaves. Therefore, the tree will be complete and will have $2^{length\ of\ items} - 1$ nodes i.e. will have an exponential complexity. However, it is still better than the dynamic programming algorithm because on average it will not generate all possible solutions. The required memory depends on the length of the priority queue.

Results

name	average ms (mean)
testN10	0.8
testN20	1.333333333
testN30	1.666666667
testN40	15.13333333
testN50	338.8
testN60	730
testN70	0.133333333
testN80	0.8
testN90	3.133333333
testN100	3.8

Complexity seems to be parabolic in nature which would match the predicted complexity


average ms (mean)
800
700
600
500
400
300
200
100
0
-100
testN10
testN20
testN30
testN40
testN50
testN60
testN70
testN80
testN90
testN100


average ms (mean)
testN100
testN90
testN80
testN70
testN60
testN50
testN40
testN30
testN20
testN10
0
100
200
300
400
500
600
700
800