# 0-1 Knapsack Problem + Solution

CA318 CA3

Student Name: David Weir

Student ID: 19433086

# What is the Knapsack Problem?

The Knapsack problem is a problem in combinatorial optimisation / combinatorics.

The problem states: “Given a set of items, each with a weight and value, determine the number of each item to include in a collection so that the total weight is less than or equal to the a given weight limit and the total value is as large as possible.”

There are 2 types of Knapsack problem:

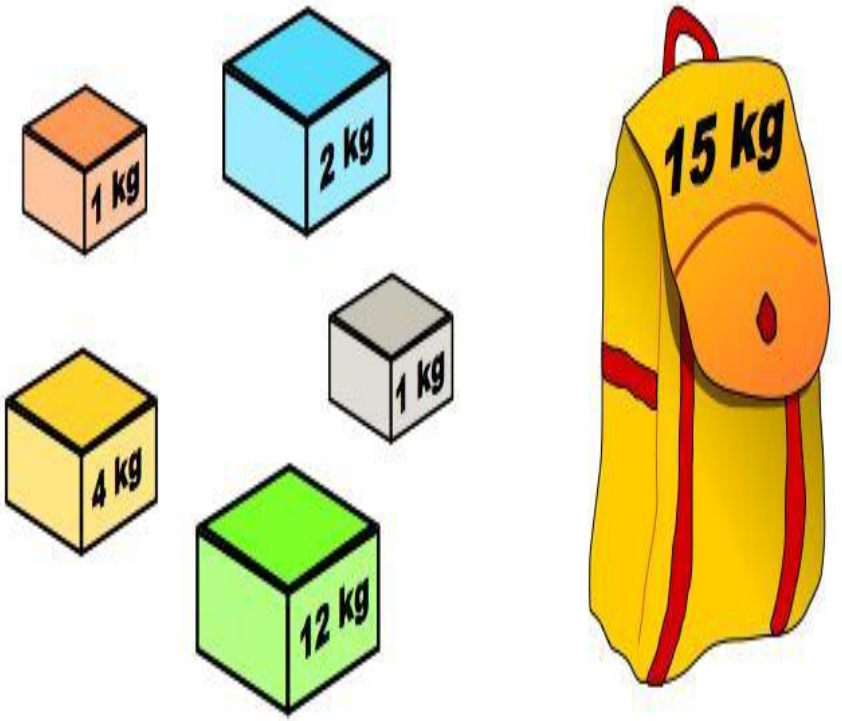
- 0/1 Knapsack problem
- Fractional Knapsack Problem

In this lecture we will focus on the 0/1 Knapsack problem

# 0-1 Knapsack Problem Explained

Under the 0/1 problem items are indivisible, you must take the whole weight of the item or not at all. The steps are as follows:

1. An item can either be added to the knapsack or not, we cannot divide the weight.
2. We are given a knapsack with a maximum weight (W) and a set (S) of n number of items with given values and weights.
3. Our goal is to pack the knapsack in a way where we achieve the highest total value of items while still under the maximum weight.



# A Written Example

Suppose we have 2 lists:

Vals = [50,100,150,200], contains the values of each item that we can take.

Weights = [8,16,32,40], contains these items corresponding weights.

W = 64, this is the maximum weight we can carry in the knapsack.

We as humans can easily determine the the best total value we can achieve under the maximum weight is 350.

We get this by taking items 1, 2 and 4 with values 50 + 100 + 200 = 350 and a total weight of 64 which is <= 64.

# Implementation

For the purpose of this lecture we will explore 4 different implementations to solve the Knapsack problem.

1. Brute Force Method
2. Bottom-up Method
3. Memoization & Recursion
4. Memoization & Recursion using Optimized Space Complexity

Each of these implementations will improve upon the last in both time complexity and space efficiency.

# Method 1: Brute Force with Recursion

This simple method will calculate the total weight and value of all subsets that can make up a collection.

Of these subsets only those with a total weight less than the max weight (W) will be considered and of these the subset with the maximum value will be picked.

For each item in a subset that item is either added or not added to the optimal set.

For each item n we check that its weight is < the current value W. If it is <W then we add its value to the current max value and subtract its weight from W to update the current W.

If it is >W then it is not added.

# Method 1: Brute Force with Recursion

- Base Case: checks that the max weight is not 0 and that there are items to choose from.
- If the weight of the current item is > W, we cannot include it in the solution and call the function again
- If the weight is < W then the max value is found by finding the max of:
  - (1) The max value obtained by items-1 and W
  - (2) The value of the current item added to the max value of items-1 and W- the weight of this item

Time Complexity: O(2^n) - redundant subproblems

Space Complexity: O(1) - no data structure is used for storing values

```python
def knapsack(W, vals, weights, items):

    # Base Case: there is no max weight or no items
    if items == 0 or W == 0:
        return 0

    # if the items weight is > W (max weight) then it cannot be added
    # call knapsack to move to the next item
    if weights[items-1] > W:
        return knapsack(W, vals, weights, items-1)

    # find the max value between
    #       (1) the max value obtained by the next item and W (excluding the current item)
    #       (2) the value of the curr item and the max value obtained by the next item and W - the weight of the curr item (including the current item)
    else:
        return max(vals[items-1] + knapsack(W-weights[items-1], vals, weights, items-1), knapsack(W, vals, weights, items-1))

# driver code
def main():
    vals = [50,100,150,200]
    weights = [8,16,32,40]
    w = 64
    items = len(vals)

    sol = knapsack(w, vals, weights, items)

    print(sol)

if __name__ == "__main__":
    main()
```

# Method 2: Bottom-Up

A dynamic programming Bottom-up Method can be used as an alternative to recursion.

- First we construct a table with rows of items (i) and columns of weight limits (j).
- The index table[i][j] is a cell containing the max value possible considering all items 0-i and the limit j.

To fill the table we set the 0th row & column to 0. Here we have no items to consider and our max weight is 0.

- For each cell [i][j] we either include the value of object i in the solution or exclude it.
- If we include the item it satisfies the same 2 conditions as before:
  - The total weight after inclusion does not exceed the weight limit W.
  - The value after inclusion is greater than when it is excluded.

# Method 2: Bottom-Up

- Create a table using list comprehension
- We use nested for loops to traverse the table and fill the cells in a bottom-up manner
- Set the 0th row and column to 0
- We check that the item's weight is <= to weight limit of that cell (j).
- If it is, we either include to exclude the item based on the max of the 2 terms.

Time Complexity: O(N*W)

Space Complexity: O(N*W), as we use a 2D Array

```python
def knapsack(W, weights, vals):
    n = len(vals) # number of items

    table = [[0 for x in range(W+1)] for x in range(n+1)] # bottom up table (unfilled)

    # nested loops traverse the table and fill entires in each cell in a bottom-up manner
    for i in range(n+1):
        for j in range(W+1):

            # set the 0th row and column to 0
            if i == 0 or j == 0:
                table[i][j] = 0

            # weight of the i object is <= the max weight for that cell (j)
            elif weights[i-1] <= j:
                # include or exclude the object by finding the max of 2 terms
                # table[i-1][j] means the i item is not included
                # vals[i-1] + table[i-1][j-weights[i-1]] means the item i is included in the optimal solution
                table[i][j] = max(vals[i-1] + table[i-1][j-weights[i-1]], table[i-1][j])

            # if the weight of i is greater than the weight limit j
            else:
                table[i][j] = table[i-1][j]

    # when the table is filled return the last cell containing the final solution
    return table[n][W]
```

# Method 3: Recursion and Memoization

Our previous recursive method can be improved using memoization. In this implementation we will use a data structure to store already computed results.

Using memoization we can solve our previous problem of computing redundant cases which increased our complexity.

In this case we will use a 2D array to store a cell result so that we do not calculate the same value multiple times instead we can access it in O(1) time from the 2D array.

This method is an improvement upon Method 1 using the dynamic programming concept memoization and improves our time and space complexity to rival that of our Bottom-up method.

# Method 3: Recursion and Memoization

- This method is an improvement upon our previous recursive method and so our logic and much of the structure remains the same.
- We have the same base case and the same code to decide if we include or exclude an item
- The obvious difference being the 2D array called table that allows us to store a value (n,w) so that if we come across it again we can find it in the array in constant time

Time Complexity: O(N*W), removes redundant calculations

Space Complexity: O(N*W), as we use a 2D Array

```python
def knapsack(W, weights, vals, n):

    # create a table populated with 0
    table = [[0 for i in range(W+1)] for j in range(n+1)]

    # Base cases
    if n == 0 or W == 0:
        return 0

    if table[n][W] != 0:
        return table[n][W]

    # choose whether to include or exclude the item

    # if the item's weight is <= W
    if weights[n-1] <= W:
        table[n][W] = max(vals[n-1] + knapsack(W-weights[n-1], weights, vals, n-1), knapsack(W, weights, vals, n-1))

        return table[n][W]

    # if it is greater than
    elif weights[n-1] > W:
        table[n][W] = knapsack(W, weights, vals, n-1)

        return table[n][W]
```

# Method 4: Optimised Space

We can improve our space efficiency of method 3 by using a 1D array as opposed to a 2D array. The 1D array improves our space complexity.

In this method we construct our table using bottom-up and memoization.

Time Complexity: O(N*W), removes redundant calculations

Space Complexity: O(W), as we use a 1D Array

```python
def knapsack(W, weights, vals):

    n = len(vals) # number of items
    dp_array = [0 for i in range(W+1)]

    for i in range(n+1):
        for w in range(W, 0, -1): # start from the back allows us to use data of previous computations

            # if the item weight is less than the weight limit W
            # choose to include to exclude the item in the solution
            if weights[i-1] <= w:
                dp_array[w] = max(dp_array[w], vals[i-1] + dp_array[w-weights[i-1]])

    return dp_array[W] # return the max value
```

# Conclusion

The Knapsack Problem appears often in real-world decision making situations in many fields .

For our purposes however, the fluidity of the Knapsack solution allows us to demonstrate and practice many different implementations of the same solution.

As we have seen here in this presentation, it has allowed us to look into and use several different dynamic programming techniques to gain a better understanding of dynamic programming in action.

Thank you for listening!