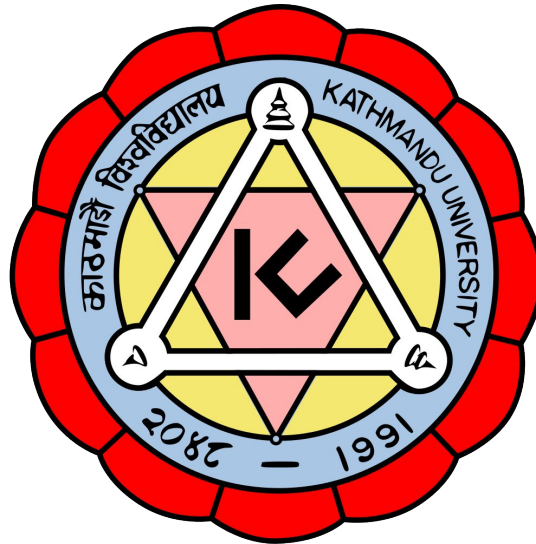


KATHMANDU UNIVERSITY

Department of Computer Science and Engineering
Dhulikhel, Kavre



COMP 314: Algorithms and Complexity
Lab Work #3
“Knapsack Problem”

Submitted By

Mukul Aryal
Roll No. 03
CS III/II

Submitted To

Dr. Rajani Chulyadyo
Assistant Professor
Department of Computer Science and Engineering

22nd January, 2025

Solve the Knapsack problem using the following strategies:

1. Brute-force method (Both fractional and 0/1 Knapsack)

Pseudocode:

```
Algorithm brute_force_01() {  
    // Generate all possible power sets  
    max_binary_string = "1" repeated len(items) times  
    max_number = binary_to_decimal(max_binary_string)  
    power_sets = []  
    for i = max_number down to 0 {  
        binary_string = binary_representation_of(i)  
        add tuple_of(binary_string) to power_sets  
    }  
    max_profit = 0  
    profit_config = null  
    for each subset s in power_sets {  
        profit = 0  
        weight = 0  
        for each index i and item in s {  
            if item == "1" {  
                profit += items[i]  
                weight += weights[i]  
            }  
        }  
        if profit > max_profit and weight <= capacity {  
            max_profit = profit  
            profit_config = s  
        }  
    }  
    return (max_profit, profit_config)  
}
```

```

Algorithm brute_force_fractional() {

    n = length of items

    items_with_ratio = []

    // Calculate value-to-weight ratio and store items with their indices
    for i = 0 to n - 1 {

        items_with_ratio.append((items[i], weights[i], items[i] / weights[i], i))

    }

    // Sort items by value-to-weight ratio in descending order
    sort items_with_ratio by ratio in descending order

    total_value = 0

    remaining_capacity = capacity

    config = ['0'] repeated n times

    for each (value, weight, ratio, idx) in items_with_ratio {

        if remaining_capacity >= weight {

            config[idx] = '1'

            total_value += value

            remaining_capacity -= weight

        } else {

            fraction = remaining_capacity / weight

            config[idx] = string_of(fraction)

            total_value += value * fraction

            remaining_capacity = 0

            break

        }

    }

    return (float_of(total_value), tuple_of(config))

}

```

2. Greedy method (Fractional Knapsack)

```
Algorithm greedy_fractional() {  
  
    n = length of items  
  
    items_with_ratio = []  
  
    // Calculate value-to-weight ratio and store items with their indices  
  
    for i = 0 to n - 1 {  
        items_with_ratio.append((items[i], weights[i], items[i] / weights[i], i))  
    }  
  
    // Sort items by value-to-weight ratio in descending order  
  
    sort items_with_ratio by ratio in descending order  
  
  
    total_value = 0.0  
  
    remaining_capacity = capacity  
  
    config = ['0'] repeated n times  
  
  
    for each (value, weight, ratio, idx) in items_with_ratio {  
        if remaining_capacity >= weight {  
            config[idx] = '1'  
  
            total_value += value  
  
            remaining_capacity -= weight  
        } else {  
            fraction = remaining_capacity / weight  
  
            config[idx] = string_of(fraction)  
  
            total_value += value * fraction  
  
            break  
        }  
    }  
  
    return (float_of(total_value), tuple_of(config))  
}
```

3. Dynamic programming (0/1 Knapsack)

```
Algorithm dynamic_01() {  
  
    n = length of items  
  
    K = 2D array of size (n + 1) x (capacity + 1), initialized to 0  
  
    keep = 2D array of size (n + 1) x (capacity + 1), initialized to False  
  
    // Build the DP table K[][] in bottom-up manner  
    for i = 1 to n {  
        for w = 0 to capacity {  
            if weights[i-1] <= w {  
                without_item = K[i-1][w]  
                with_item = items[i-1] + K[i-1][w - weights[i-1]]  
                if with_item > without_item {  
                    K[i][w] = with_item  
                    keep[i][w] = True  
                } else {  
                    K[i][w] = without_item  
                }  
            } else {  
                K[i][w] = K[i-1][w]  
            }  
        }  
    }  
  
    // Reconstruct the solution  
    w = capacity  
    config = ['0'] repeated n times  
    for i = n down to 1 {  
        if keep[i][w] {  
            config[i-1] = '1'  
            w = w - weights[i-1]  
        }  
    }  
}
```

```
}

max_profit = K[n][capacity]

// Verify the configuration matches the brute force result

test_weight = 0

test_profit = 0

for i = 0 to n - 1 {

    if config[i] == '1' {

        test_weight += weights[i]

        test_profit += items[i]

    }

}

// If weight or profit doesn't match, use brute force result

if test_weight > capacity or test_profit != max_profit {

    (max_profit, config) = brute_force()

}

return (max_profit, tuple_of(config))

}
```