"0-1 Knapsack Problem"

Philip Park

Eastern Mennonite University

1. Abstract

The purpose of this paper is to analyze three different algorithms: brute force, top down memorization, and bottom up, designs applied to a single problem: 0-1 Knapsack Problem. The Knapsack Problem is an optimization-oriented combination problem where the focus is to maximize the benefit of combinations without exceeding the maximum limit. Using the three designs principles, the objective is to compare the results and find the optimal one for the solution.

2. Background

In life, we come across many problems, which can be solved and may not be at the same time. They may be simple enough to find a solution instantly, while it could take ages, or there is no solution at all. Take packing your bag for a vacation as an example. Generally, the carry-on weight limit is up to 26 pounds, with measurements of 45 linear inches (combined length x width x depth) [1]. With that in mind, we try to pack our bags for the trip. However, things do not follow suit in the way you want. What if you are not able to fit everything in that small bag? How do you manage what is important over other things? The knapsack problem is a great example for this issue, it outlines the maximum benefit of the suitcase without exceeding the limit for the carrier.

i. The Knapsack Problem

The Knapsack Problem is one example of an optimization-oriented combinational problem, which looks for the best solution among the rest of the options [3]. The knapsack has a positive integer capacity, (must be positive, otherwise impossible to solve) C. Inside the backpack, there are items, labeled with i for item/value and w for item/weight; these are positive integer values.

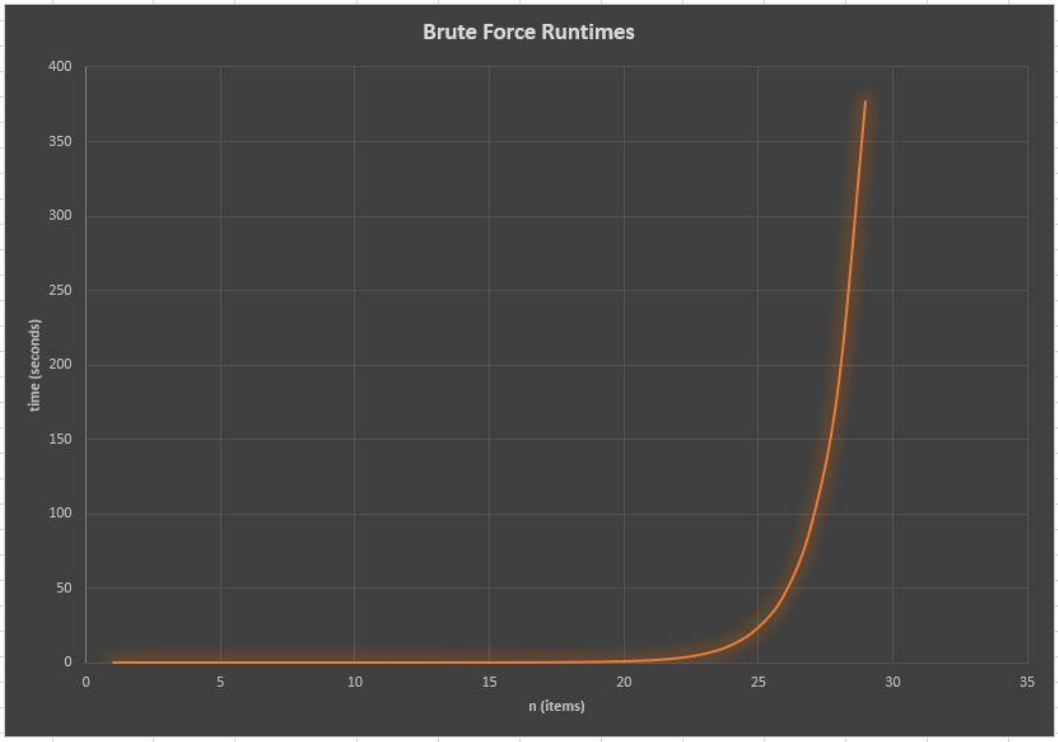
3. Methods

i. Brute Force Algorithm

A greedy algorithm is a simple yet intuitive algorithm that is used in optimization problems. It aims to make the most optimal choice at each stage, while trying to find the overall best way of solving the problem [2]. Brute force algorithms are like greedy algorithms hence they aim for direct, "forcing" through to find the solution. However, the difference between the two is that brute force depends on the CPU's processing power, and does not rely on any optimization, while greedy algorithm tries to find the best way to attain victory without making unnecessary sacrifices.

The brute force algorithm runs in $O\ (2^n)$ time. When a value for the max capacity weight is defined, it is sort of setting the maximum space allowed for the algorithm to perform its search for all the solutions. Brute force mainly relies on the branching factor and the depth to find the solution of the problem, defined by the number of items squared by the maximum range of item and weight values. Thus the running time of this algorithm increases exponentially as the defined variables increase.

n	time (s)
1	5.4836E-06
2	1.14444E-05
3	1.62125E-05
4	2.83418E-05
5	4.60147E-05
6	0.000100612
7	0.000198125
8	0.000410318
9	0.000813961
10	0.001668245
11	0.003338881
12	0.006238699
13	0.012343407
14	0.022453308
15	0.038567543
16	0.0634408
17	0.1198771
18	0.240283012
19	0.445997715
20	0.790315208
21	1.466278315
22	2.936762094
23	5.814036565
24	11.6881573
25	23.41374159
26	46.40920019
27	93.56532192
28	187.1233916
29	376.9450187

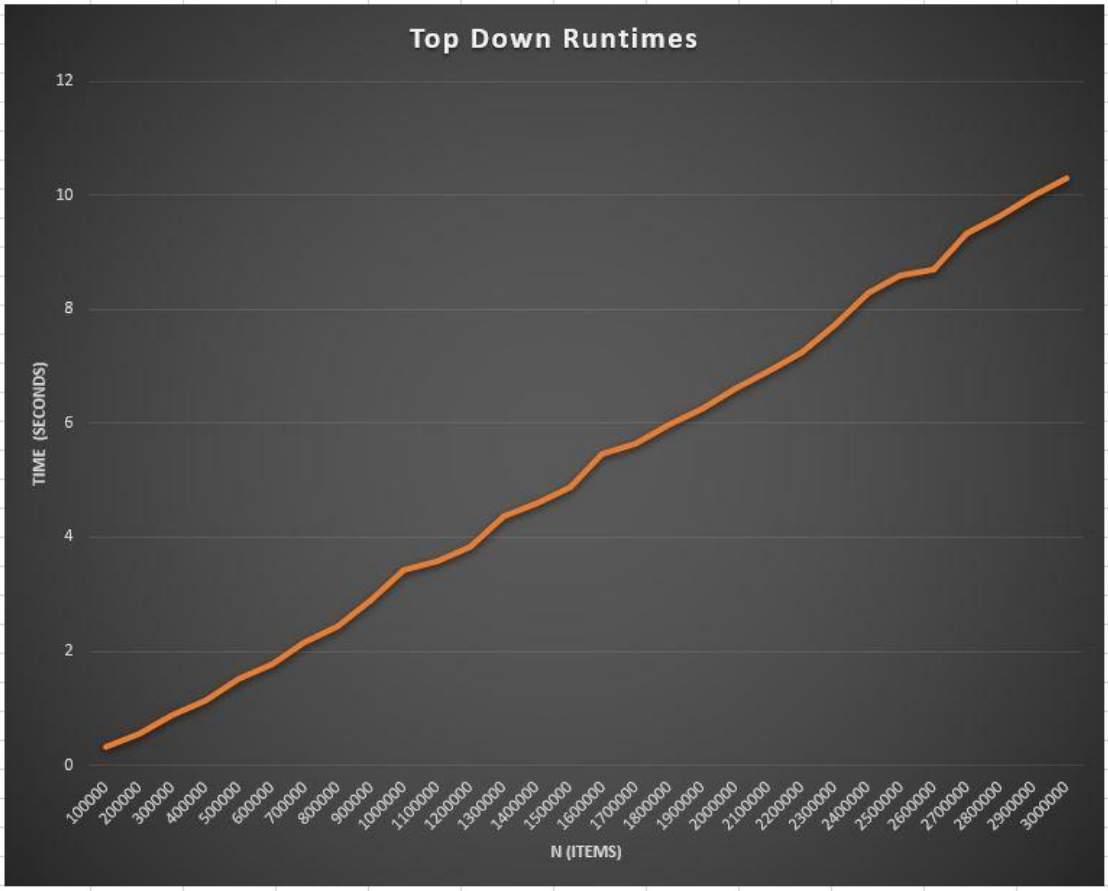


ii. Top-down Approach

The top-down approach is an essential form of breaking down a problem or subsystem to make it easier to solve. A top-down approach starts with the top, the big picture. Then it breaks down into smaller segments, like a tree diagram [4]. For this particular problem, the input starts with defining the three variables: value and weight of the item, and the maximum capacity. Additionally, two sub-variables are created: I and w. They can be viewed as a copy of the value and weight variables; value = I (item value), weight = w (weight of the item), in i-ith term. The function “calculate” takes 5 arguments: value, weight, m (table m), n (number of items), and capacity. Then, it returns the maximum value of the items achieved without exceeding the weight capacity number specified.

This approach runs in $0(n)\ time$, because every item must be created and there are n items. Following that process, the sorting of the array of items runs in O (n log n) time. On appearance, the graph may show a similar lineup to a linear relationship between the time (in seconds) and n (number of items) as n increases, but it follows through with the time of n log n.

n	time (s)
100000	0.320485933
200000	0.554017067
300000	0.881973267
400000	1.125880241
500000	1.519999807
600000	1.77131772
700000	2.141910076
800000	2.427339797
900000	2.882771992
1000000	3.420078942
1100000	3.574099064
1200000	3.8370893
1300000	4.362854958
1400000	4.58849883
1500000	4.8698567
1600000	5.468298456
1700000	5.630125761
1800000	5.975736865
1900000	6.244175341
2000000	6.614526438
2100000	6.926522017
2200000	7.241564751
2300000	7.727230077
2400000	8.282039925
2500000	8.585542339
2600000	8.705816507
2700000	9.332063199
2800000	9.643348651
2900000	10.00641521
3000000	10.30006409



iii. Bottom-Up Approach

The dynamic programming algorithm, also known as the "bottom up approach" runs in $O(n^2)\ time$. It is an optimization strategy as it stores the results of the subproblems the first time they are called, so they do not have to be recalculated the times they are called in the

future. Bottom-up algorithms are similar to working from the leaves of a family tree to the top, in order. In the algorithm solution, the function knapsack takes 3 arguments: value, weight, and the maximum capacity, along with the number of items. Then it returns the maximum value of items that has not exceeded the total weight capacity.

4. Complexity

The complexity of a brute force algorithm is usually O (n2), growing exponentially. In small test cases, this approach may be used well as there is not much time and power required to process the sorts. However, since this is dependent on the optimization of the CPU, larger tasks would take significantly longer to process. With an instance of a table test case used in the algorithm, with 45 items and a maximum weight volume of 750, the average running time was between 0.02 – 0.04 seconds. In cases of the bottom-up approach, as the time complexity is O (N*capacity), it requires a two-dimensional array with rows = number of items + columns = volume of weight. Keeping the test cases the same, the average running time for this solution was between 0.04 – 0.056 seconds. The last method of the three is the top-down approach, with the time complexity being O(NlogN) + O(N) = O(NlogN). From the same test cases used, the approximate average running time was between 0.12 – 0.149 seconds. For each time of executions were measured through the time module of python, built in. However, this does not account for the running background processes and tasks, therefore will not be entirely accurate.

5. Conclusion

With the analysis of brute force, dynamic programming, and top-down memoization algorithms, each has their own strengths and weaknesses. While the time complexities of the algorithms are to each of their own, the nature of the problem they are applied to would possibly make some of them more suitable than others; more efficient for their purpose. Between the three algorithms, my personal preferences would most likely be the dynamic programming and brute force algorithms. In comparison to each other, the bottom-up process is straightforward and easier to code in some sense, as it does not require additional structures. As for brute force, in simple tasks, it is best suited for the job as it does not require many resources, but is not as efficient as the other algorithms once it passes the boundary of "simple tasks."

References:

[1]. Kleinberg, J., & Tardos, E. (2006). Algorithm design. Boston: Pearson/Addison-Wesley.

[2]. Brute Force algorithm. (n.d.). Retrieved March 26, 2019, from http://www-igm.univ-mlv.fr/~lecroq/string/node3.html

[3]. 0-1 Knapsack Problem | DP-10. {2018, September 28}. Retrieved March 26, 2019, from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

[4]. DP - Top down VS Bottom up - Competitive Programming. (n.d.). Retrieved March 26, 2019, from https://inginious.org/course/competitive-programming/dp-topdown-bottomup

[5]. Manivannan, A. (2014, July 05). The Knapsack problem. Retrieved March 26, 2019, from https://rerun.me/2014/05/27/the-knapsack-problem/

Appendix A: Python Code

Bruteforce.py

```
import operator
import sys
import time

#Table of items, weight capacity
#items = [ [20, 70], [25, 45], [60, 35], [50, 35], [65, 30], [90, 30], [10, 20],
[35, 15], [55, 10], [10, 10] ] #10
#max_weight = 300

items = [ [20, 70], [25, 45], [60, 35], [35, 35], [50, 30], [35, 30], [65, 20],
[90, 15], [10, 10], [35, 10], [55, 30], [10, 100], [25, 78], [10, 98], [34, 5],
[90, 67], [24, 90], [12, 3], [34, 3], [54, 23], [55, 20], [64, 69], [74, 25],
[95, 45], [99, 52], [64, 22], [121, 36], [31, 35], [66, 75], [96, 1], [468, 74],
[1, 25], [20, 65], [96, 66], [345, 87], [87, 30], [74, 95], [15, 56], [26, 23],
[36, 42], [95, 35], [45, 45], [111, 34], [212, 3], [313, 3]] #45
max_weight = 750

"""Defining variables"""
def value(item): return item[0]
def weight(item): return item[1]

def weight_items(items):
    return sum(weight(item) for item in items)

def value_items(items):
    return sum(value(item) for item in items)

#Creating a chart with the contents of the knapsack
def vwr(item):
    return float(value(item)) / weight(item)

def items_r(items):
    return [item + [vwr(item)] for item in items]
#Sorting the items by maximum value/weight
def chart(items):
    return sorted(items, key=operator.itemgetter(2), reverse=True)


"""Main Function"""
def knapsack(items, max_weight):
```

```
    current_w = 0
    current_v = 0
    items_sorted = chart(items_r(items))

    print('Greedy Solution/Brute Force Algorithm\n')

    print('Maximum Weight Capacity = %d' % max_weight)

    print('\n*** Contents of the Knapsack ***')
    print('(v    wt  val/wt)') #spacing for simple-table format
    for item in items_sorted: sys.stdout.write( '(%-4d %-4d %-4.2f)\n' %
(value(item), weight(item), vwr(item)) )
    sys.stdout.write('\n')

    items_taken = []
    for item in items_sorted:
        if current_w + weight(item) < max_weight:
            current_w += weight(item)
            current_v += value(item)
            items_taken.append(item)

        else:
            sub = (max_weight - current_w) / float(weight(item))
            current_w += sub*weight(item)
            current_v += sub*value(item)

            print('Current weight of the Knapsack is %d (Max)' % current_w)
            print('Current value of the Knapsack is', current_v)
            print('')
            break

def main():

    start = time.time()#Measurement of code process

    knapsack(items, max_weight)

    end = time.time()
    print("Time to run: ",end - start, "seconds")
    print('-----------------------------\n')


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

topdown.py

```
import time

#Top-down memoization algorithm for Knapsack 0-1 Problem

start_time = time.clock()
def knapsack(value, weight, capacity):
    '''
    Value = Value of the item(s)
    Weight = Weight of the item(s)
    Capacity = Maximum Weight Possible
    '''
    n = len(value) - 1
    """
    m = maximum weight
    i = item
    w = weight
    """
    m = [[-1]*(capacity + 1) for _ in range(n + 1)]

    return calculate(value, weight, m, n, capacity)

def calculate(value, weight, m, i, w):

    if m[i][w] >= 0:
        return m[i][w]

    if i == 0:
        q = 0
    elif weight[i] <= w:
        q = max(calculate(value, weight,
                                m, i - 1 , w - weight[i])
                + value[i],
                calculate(value, weight,
                                m, i - 1 , w))
    else:
        q = calculate(value, weight,
                            m, i - 1 , w)
    m[i][w] = q
    return q
```

```
value = [20, 25, 60, 35, 50, 35, 65, 90, 10, 35, 55, 10, 25, 10, 34, 90, 24, 12,
34, 54, 55, 64, 74, 95, 99, 64, 121, 31, 66, 96, 468, 1, 20, 96, 345, 87, 74, 15,
26, 36, 95, 45, 111, 212, 313] #45
weight = [70, 45, 35, 35, 30, 30, 20, 15, 10, 10, 30, 100, 78, 98, 5, 67, 90, 3,
3, 23, 20, 69, 25, 45, 52, 22, 36, 35, 75, 1, 30, 74, 25, 65, 66, 87, 30, 95, 56,
23, 42, 35, 45, 34, 3] #45
m = 750
i = len(value)
w = len(weight)
print("Current Maximum Weight:",m)
print("Current value of the knapsack is",(knapsack(value, weight, m)))
print("Time to Run:",time.clock() - start_time, "seconds")
```

Bottomup.py

```
import time

start_time = time.clock()

#bottom-up implementation of knapsack 0-1 problem

def knapsack(C, wt, val, n):
    contents = [[0 for x in range(C+1)] for x in range(n+1)]

    # building in bottom up fashion?
    for i in range(n+1):
        for w in range(C+1):
            if i == 0 or w == 0:
                contents[i][w] = 0
            elif wt[i-1] <= w:
                contents[i][w] = max(val[i-1] + contents[i-1][w-wt[i-1]],
contents[i-1][w])
            else:
                contents[i][w] = contents[i-1][w]

    return contents[n][C]

# Table of values/weights for testing
# ***Values used are same from the brute force.py algorithm***
'''
    val = value of elements
    wt  = weight of elements (of the values)
    C   = maximum weight capacity
'''
```

```
val = [20, 25, 60, 35, 50, 35, 65, 90, 10, 35, 55, 10, 25, 10, 34, 90, 24, 12,
34, 54, 55, 64, 74, 95, 99, 64, 121, 31, 66, 96, 468, 1, 20, 96, 345, 87, 74, 15,
26, 36, 95, 45, 111, 212, 313] #45
wt = [70, 45, 35, 35, 30, 30, 20, 15, 10, 10, 30, 100, 78, 98, 5, 67, 90, 3, 3,
23, 20, 69, 25, 45, 52, 22, 36, 35, 75, 1, 30, 74, 25, 65, 66, 87, 30, 95, 56,
23, 42, 35, 45, 34, 3] #45
C = 750
n = len(val)
print("Current Maximum Weight:",C)
print("Current value of the knapsack is",(knapsack(C, wt, val, n)))
print("Time to Run:",time.clock() - start_time, "seconds")
```