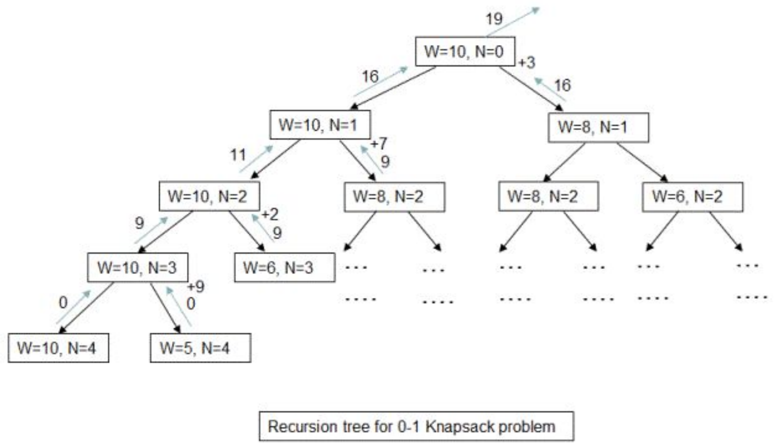
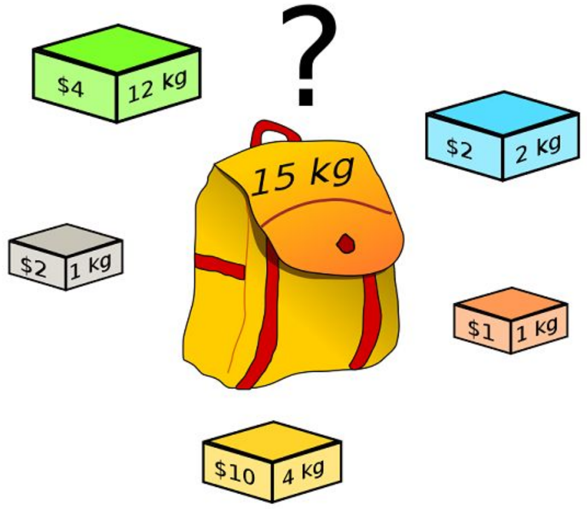
Max Knapsack Presentation

Jonny Lynch, Ethan Crawford, Michael Leek, and Aly Clark





Max Knapsack

My problem consists of whether we take the item or we do not take the item:

Given W, V, L is there a max value V where an item in list L has a weight w that is less than the max weight W?

Optimization Version:

- Dynamic Programming
    - Top Down
    - Optimal in both time and space
    - Stores results of all the subproblems

Applications that use Max Knapsack

- Determine the best programs to run on a limited resource cloud system
- Optimize water distribution across a fixed pipe network
- Optimize the company’s supply chain

Problem Input

Example Input:

First Line: A nonnegative integer weight W

Second Line: A nonnegative integer N

N Lines: Each line n contains a String (name) followed by a String (dollar value) followed by another String (item’s weight)

Example Code input:

```
31
3
HTXRX 25.4 1.8
GIPRI 4.1 2.1
EOFMM 0.7 0.5
```

Reduction

Given weights W and values V, can a subset of items X in {1, 2, …, *n*} be picked that satisfy the following constraints:

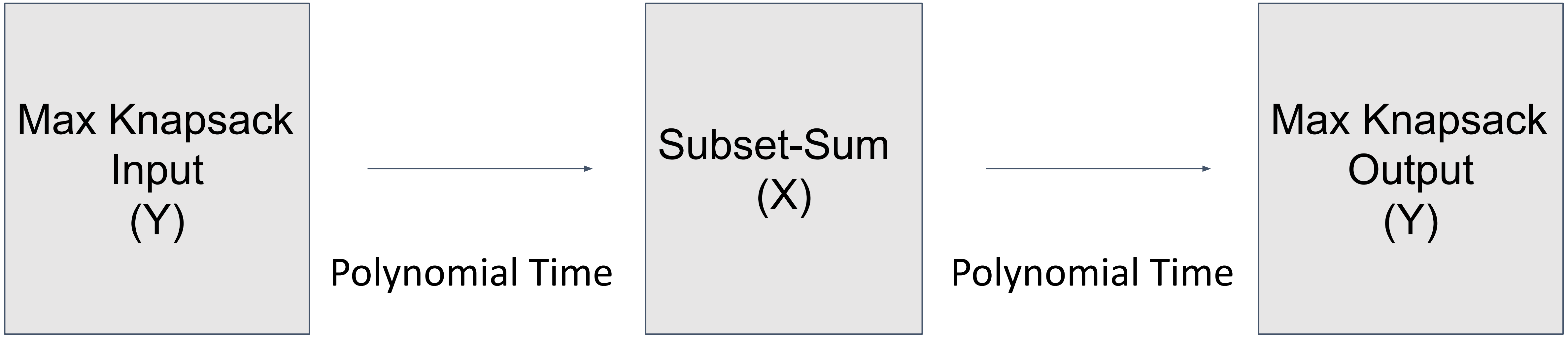
The sum of each item’s value v is greater than or equal to V
The sum of each item’s weight w is less than or equal to W

Solving the above inequalities is the same as solving the Subset-Sum problem, which is already proven to be NP-Complete. Therefore, the max knapsack problem can be reduced to the Subset-Sum problem in polynomial time.

The complexity of this problem depends on the size of the input values of each item’s weight and value. With this being said, the time to complete the algorithm depends on the size of the input. If the inputs are binary, it’s complexity becomes exponential therefore making this problem NP-Complete.

Reduction (justify its inclusion in NP-Complete)

Illustrated Example:



Max Knapsack wants to maximize the value whereas Subset-Sum wants to maximize the weight

Subset Sum:

$$\text{sum}(j, W) = \max \begin{cases} \text{sum}(j-1, W) & \text{if } j \notin S^* \\ w_j + \text{sum}(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

Max Knapsack:

$$\text{sum}(j, W) = \max \begin{cases} \text{sum}(j-1, W) & \text{if } j \notin S^* \\ v_j + \text{sum}(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

Sketch of Exact Solution (pseudo-code)

```
Knapsack (items, weight):
  Knapsackinator (weights_remain, i_index):
    check if weights_remain is empty or i_index is greater than or equal to the length of
the items
       return a list containing zero and an empty list
    with_item <- Knapsackinator (weights_remain - items's weight, i_index + 1)
    without_item <- Knapsackinator (weights_remain, i_index + 1)
    add item's value to with_item array
    check if with_item's value is greater than without_item
       return with_item
    else:
       return without_item
  return Knapsackinator(weight, 0)
```

Worst Case Example (if possible)

Worst case is when you have to take every item: O(2^n)

Example:

```
40
3
GOLD 5 10
SILVER 5 10
BRONZE 5 10
```

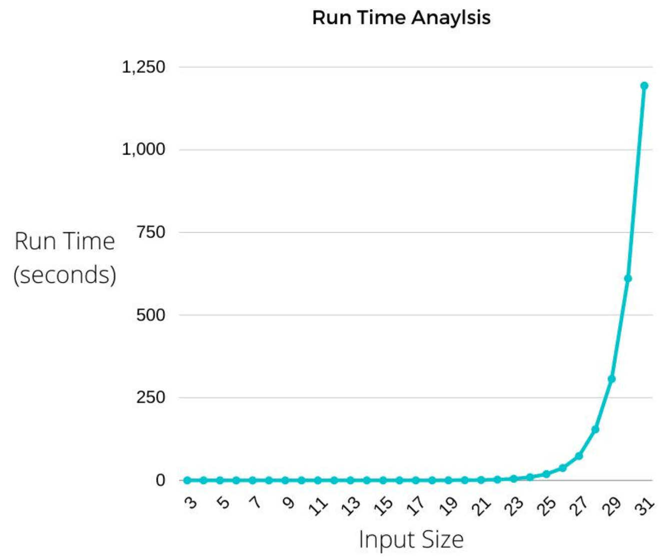
Test Cases

Generated with a python program

29 test with inputs ranging from 3 - 31

Input of 3 items run time: 0.000010s

Input of 31 items run time: 1193.307045s (~20 minutes)

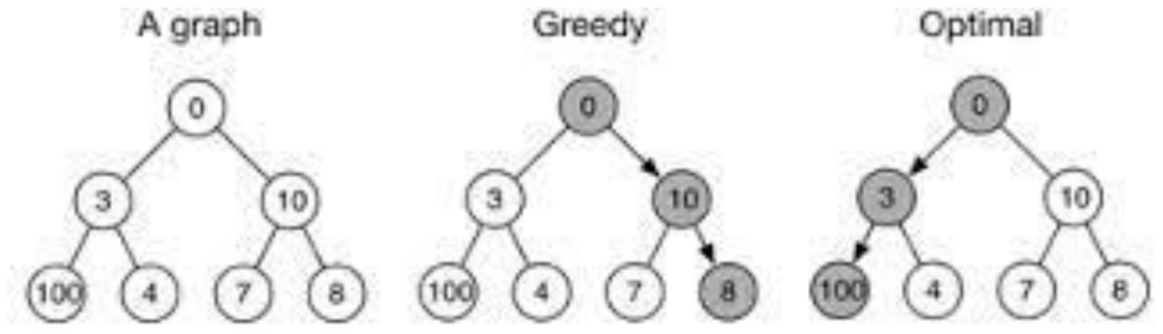


Approximation Portion

Aly Clark and Mike Leek

Approximation

- We chose a greedy algorithm because it drastically speeds up decisions. We sort and choose the most valuable items to put into our knapsack first
- Fractional knapsack was used to show our upper bound
  - It will always be greater than the other two



Pseudocode

```
max_knapsack (items):

   sort items from most valuable per weight
unit

   while there is still more weight allowed
and more items:

      if item weight <= weight left over:
         add to knapsack
         subtract weight item weight from
remaining

   return item list
```

Results