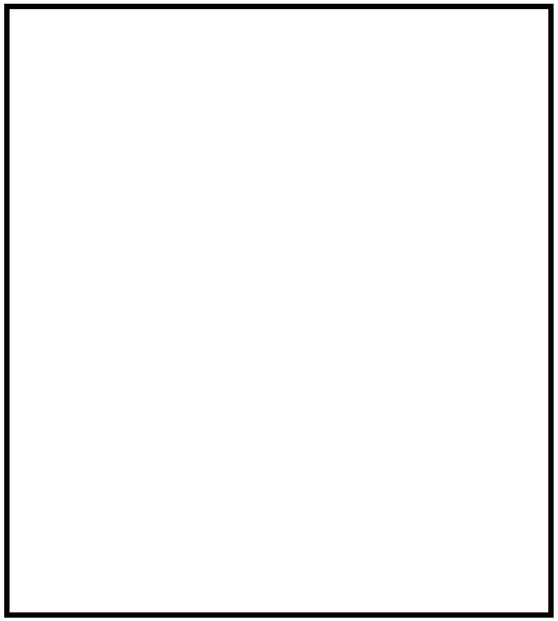
Bucknell University
Computer Science

CSCI 311 - Data Structures

The Knapsack Problem

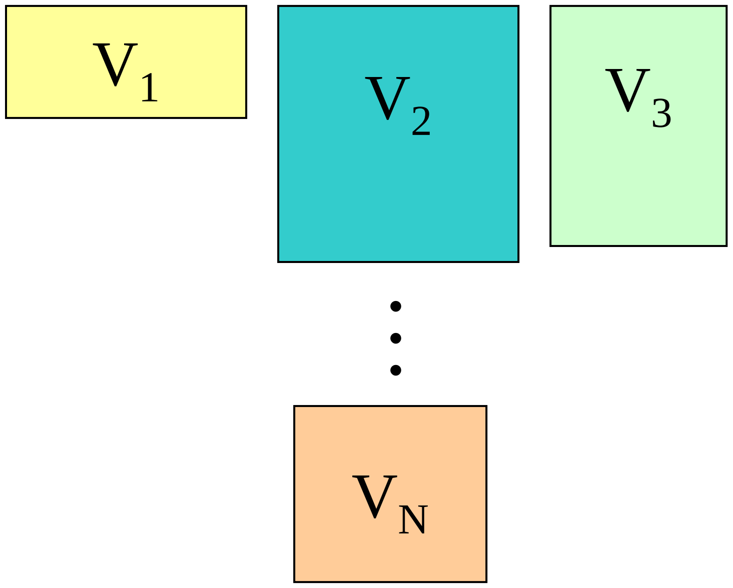
The Knapsack Problem

Knapsack of capacity M



N Types of Indivisible Items

(unlimited number of each type)

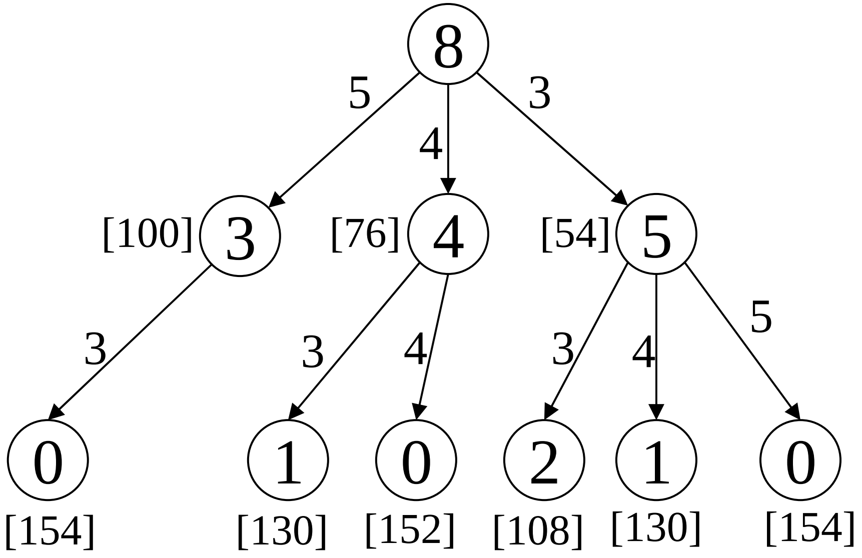


<u>Problem:</u> What is the selection of items that fits in the knapsack maximizing the total value of its contents?

The Knapsack Problem

N=3
M=8

	Type A	Type B	Type C
Value	100	76	54
Weight	5	4	3



Note:

- For each node in this tree, we have a set of possible *decisions*.
- Each decision has a cost (its weight) and leads to an associated yield.
- The goal is to find a sequence of decisions that leads to an optimal solution.
- The number of possible solutions is exponential with M. We'd have to find them all and then choose the very best.

The Knapsack Problem

The recursive nature of the problem jumps out at us when we observe the decision tree.

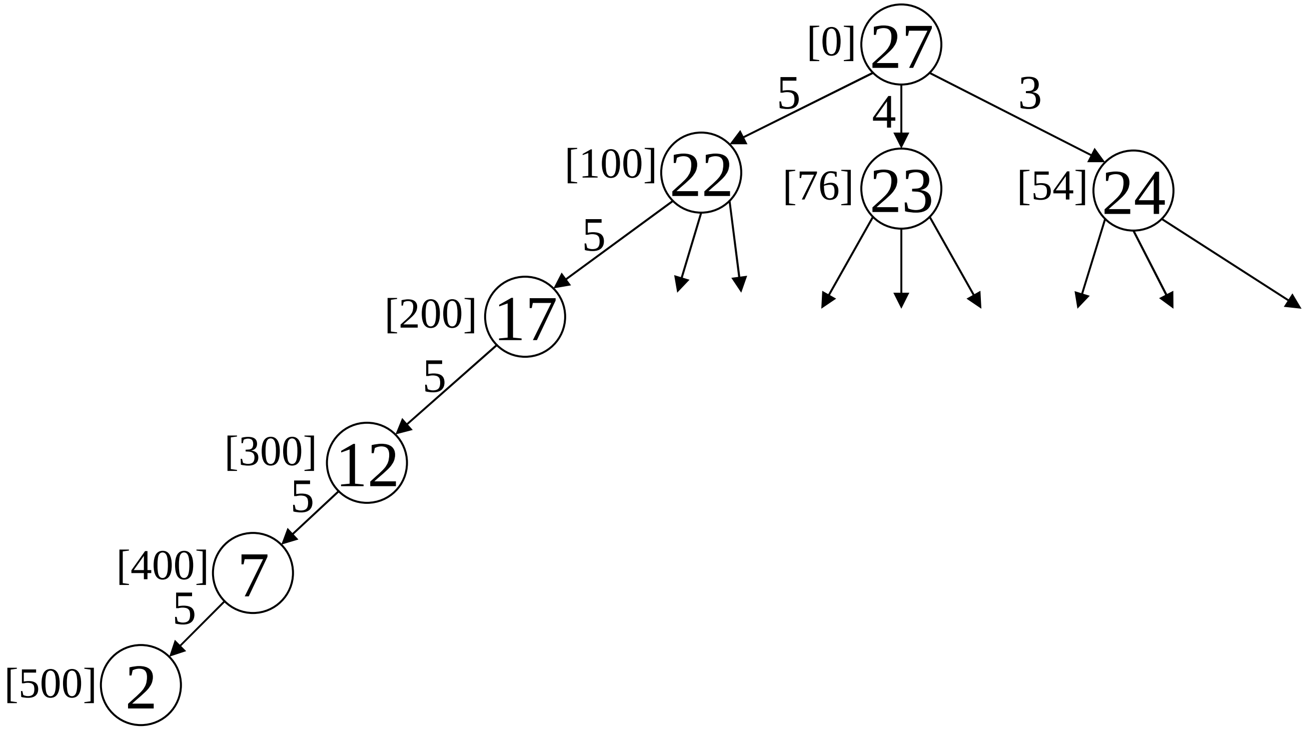
The problem has **optimal substructure** and **overlapping sub-problems**, so it is solvable with *dynamic programming*.

What we have to figure out is how to map the problem onto some kind of data structure to store solutions to each sub-problem as the tree is traversed.

The Knapsack Problem

N=3
M=27

	Type A	Type B	Type C
Value	100	76	54
Weight	5	4	3



Question: What kind of data structure is needed to apply DP to this problem?

The Knapsack Problem

(recursive solution)

```
knap(M)
  max = 0;
  for i = 1 to N // Loop through item types
    // Solve problem assuming we include
    // an item of type i
    do spaceLeft = M – size[i]
       if spaceLeft >= 0 // if type i fits
          then // Compute candidate sol’n t
               t = knap(spaceLeft)+val[i]
               if t > max
                  then max = t
  return max;
```

The Knapsack Problem

(DP solution)

```
knap(M)
   if maxKnown[M]!= unknown
     then return maxKnown[M];

  // Otherwise, result not yet known:
  max = 0
  for i = 1 to N // Try each item type
    do spaceLeft = M – size[i]
       if spaceLeft >= 0 // If item type i fits
          then // Compute candidate solution t
               t = knap(spaceLeft) + val[i]
               if t > max
                   then max = t;
                        maxi = i;

  maxKnown[M] = max  // memoize result
  return max
```