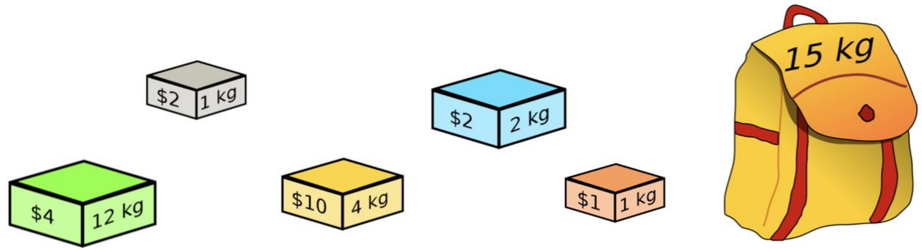
Data Structures and Algorithms

Lab 5
Dynamic Programming & Knapsack Problem

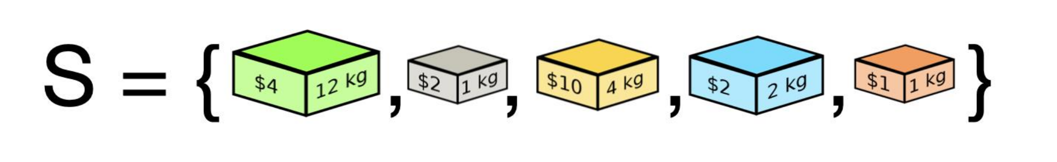
Agenda

- Recap
  - Dynamic Programming
- Knapsack Problem
- Coding exercises

Knapsack Problem

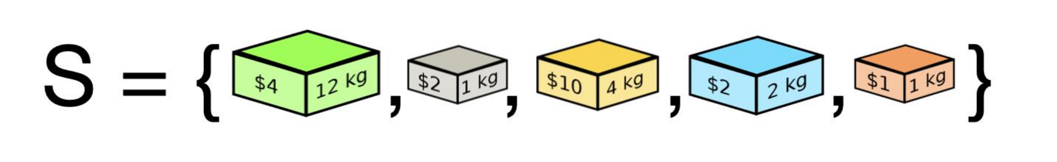


Knapsack Problem



Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Knapsack Problem



1. Total value should be maximized.
2. Total weight should not exceed the limit.
3. Each item can be taken (once) or not taken.

Note: this variation is commonly known as «0-1 Knapsack problem».

Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

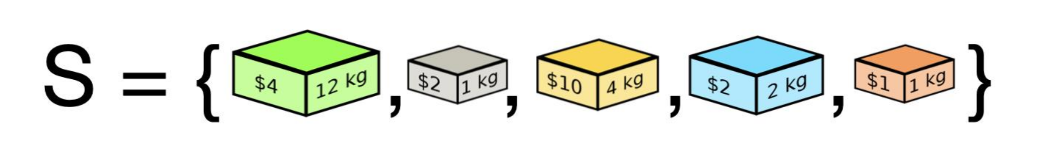
Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Recap: Algorithmic Strategies

- Brute Force
- Greedy
- Divide-and-Conquer
- Dynamic Programming

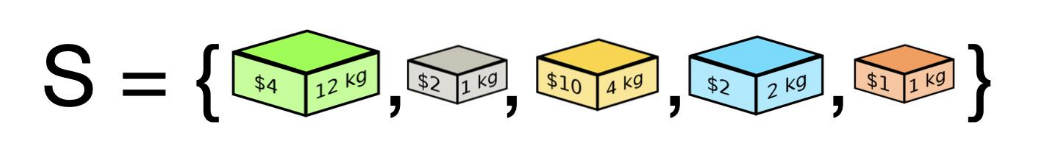
Knapsack Problem: Brute Force



Exercise 5.1. Suggest a brute-force algorithm for Knapsack Problem.

Exercise 5.2. Determine the time complexity of the brute force algorithm.

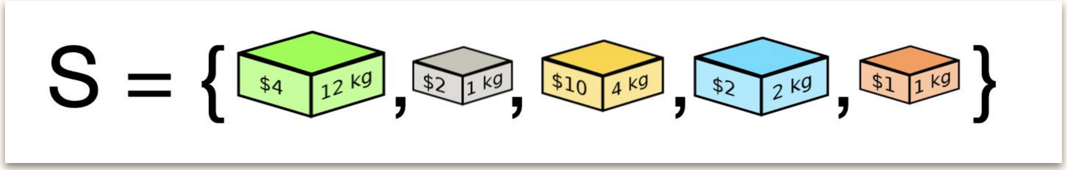
Knapsack Problem: Greedy



Exercise 5.3. Suggest a greedy algorithm for Knapsack Problem.

Exercise 5.4. Determine the time complexity of the greedy algorithm.

Knapsack Problem: Greedy

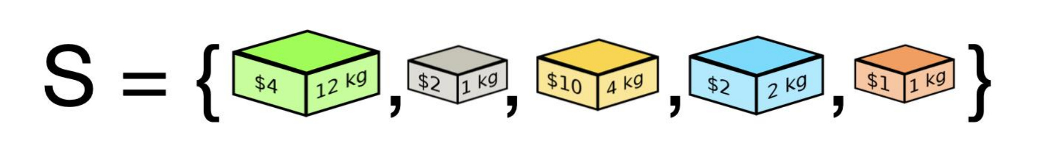


Exercise 5.3. Suggest a greedy algorithm for Knapsack Problem.

Exercise 5.4. Determine the time complexity of the greedy algorithm.

Exercise 5.5. Prove that greedy algorithm is not always correct or demonstrate a counterexample.

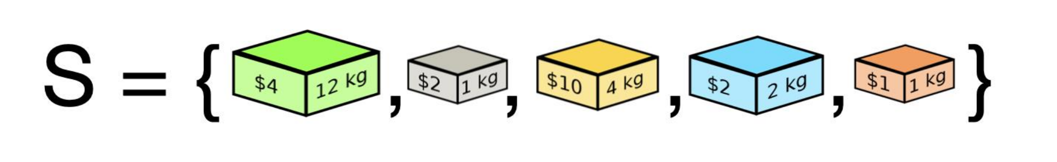
Knapsack Problem: Divide and Conquer



Exercise 5.6. Suggest divide-and-conquer algorithm for Knapsack Problem.

Exercise 5.7. Determine the time complexity of the divide-and-conquer algorithm.

Knapsack Problem: Divide and Conquer

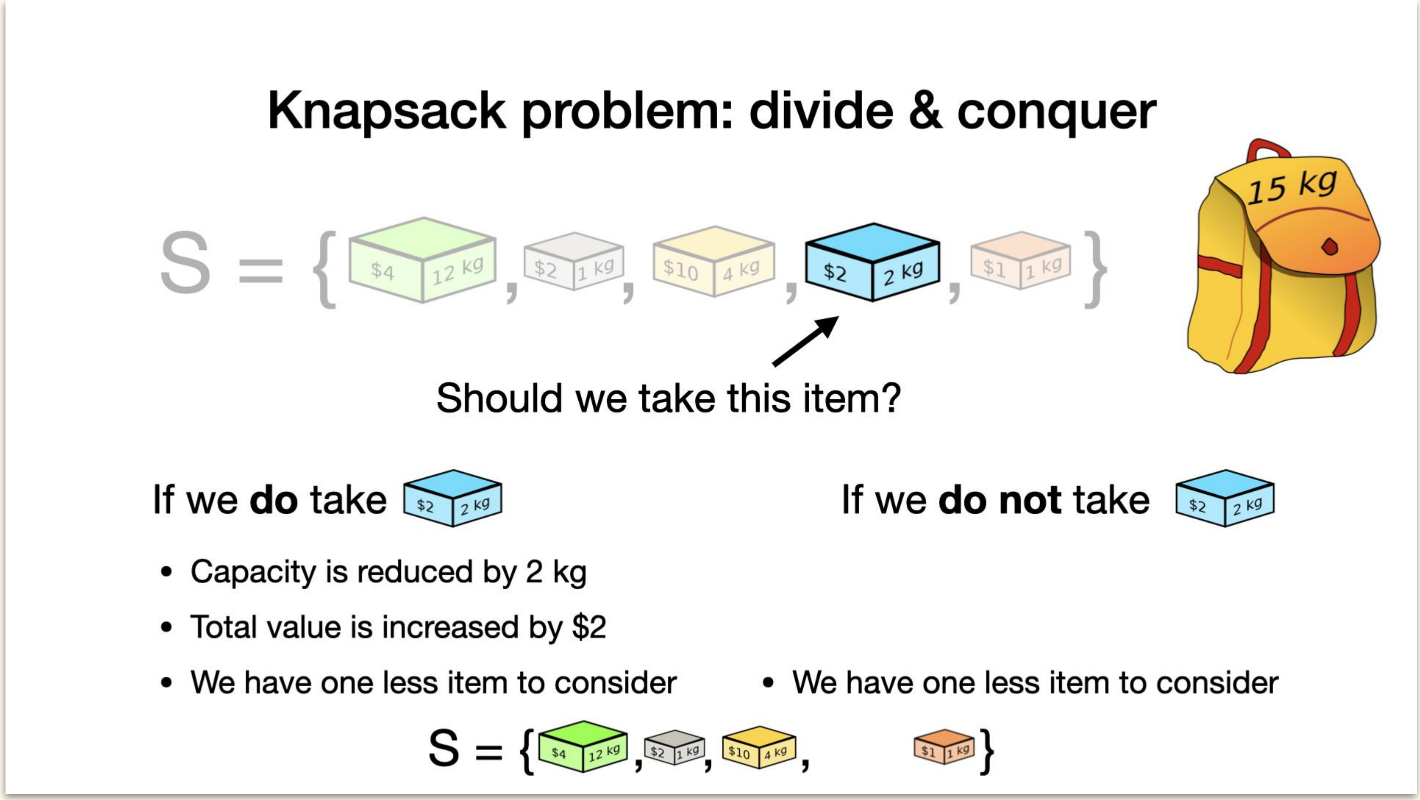


Exercise 5.6. Suggest divide-and-conquer algorithm for Knapsack Problem.

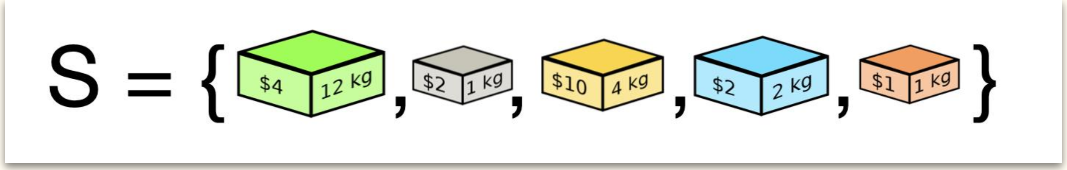
Exercise 5.7. Determine the time complexity of the divide-and-conquer algorithm.

Exercise 5.8. Determine opportunities for optimisation by identifying overlapping subproblems in the divide-and-conquer approach.

Knapsack Problem: Divide and Conquer



Knapsack Problem: Divide and conquer



1. Consider all subsets of items and calculate the total weight and value of all subsets.
2. pick the maximum value subset:
    a. Maximum value obtained by n-1 items and W weight (excluding nth item).
    b. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

Knapsack Problem: Recursive Function

$$f(items, W_{max})$$

maximum total value for a specific instance of Knapsack problem

Knapsack Problem: Recursive Function

$$f(items, W_{max})$$

maximum total value for a specific instance of Knapsack problem

$$f(\varnothing, W) = 0$$

$$f(\{item\} \cup items, W) = \max(\underline{f(items, W)}, \underline{item.value + f(items, W - item.weight)})$$

Should we take **this** item?

What if we **do not** take it?

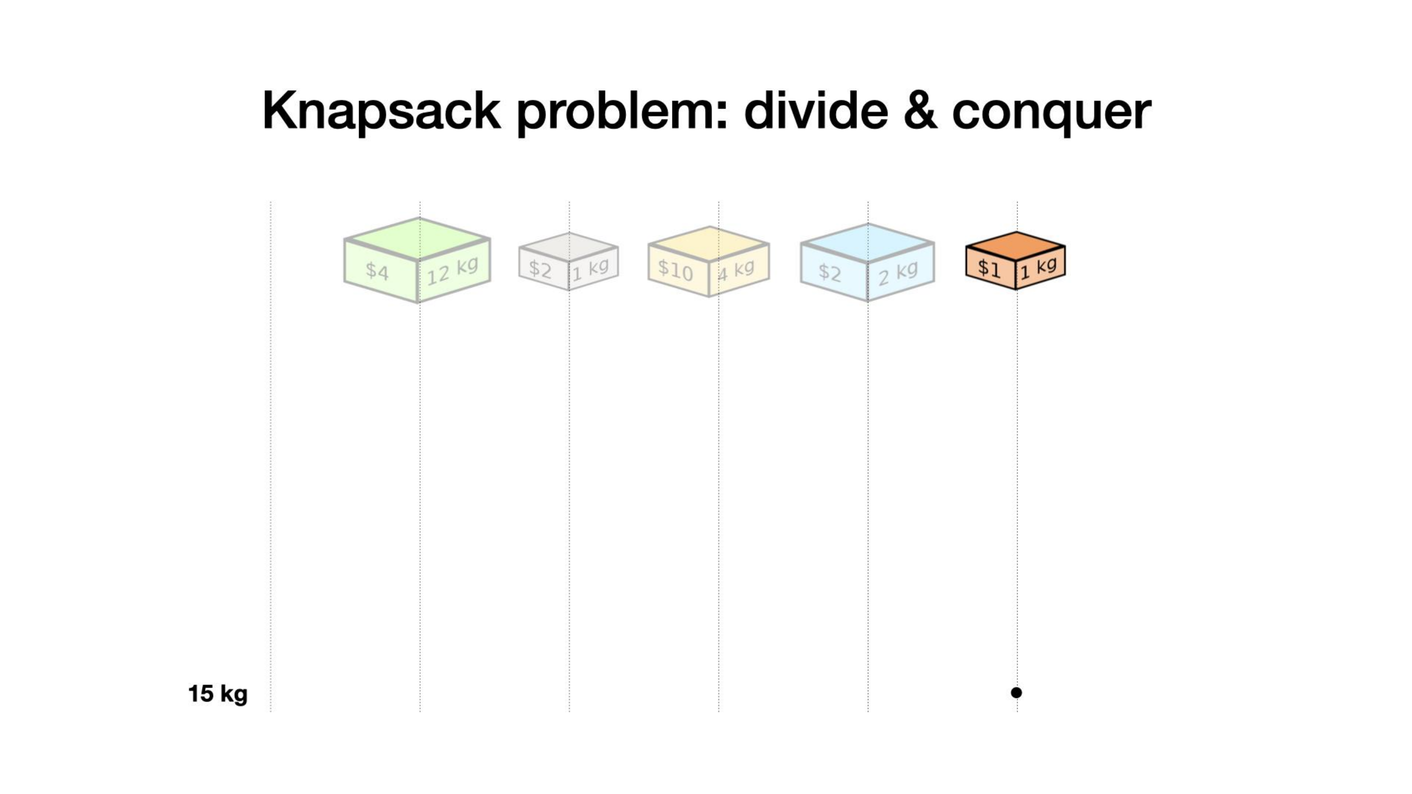
What if we **do** take it?

Knapsack Problem: Divide and conquer

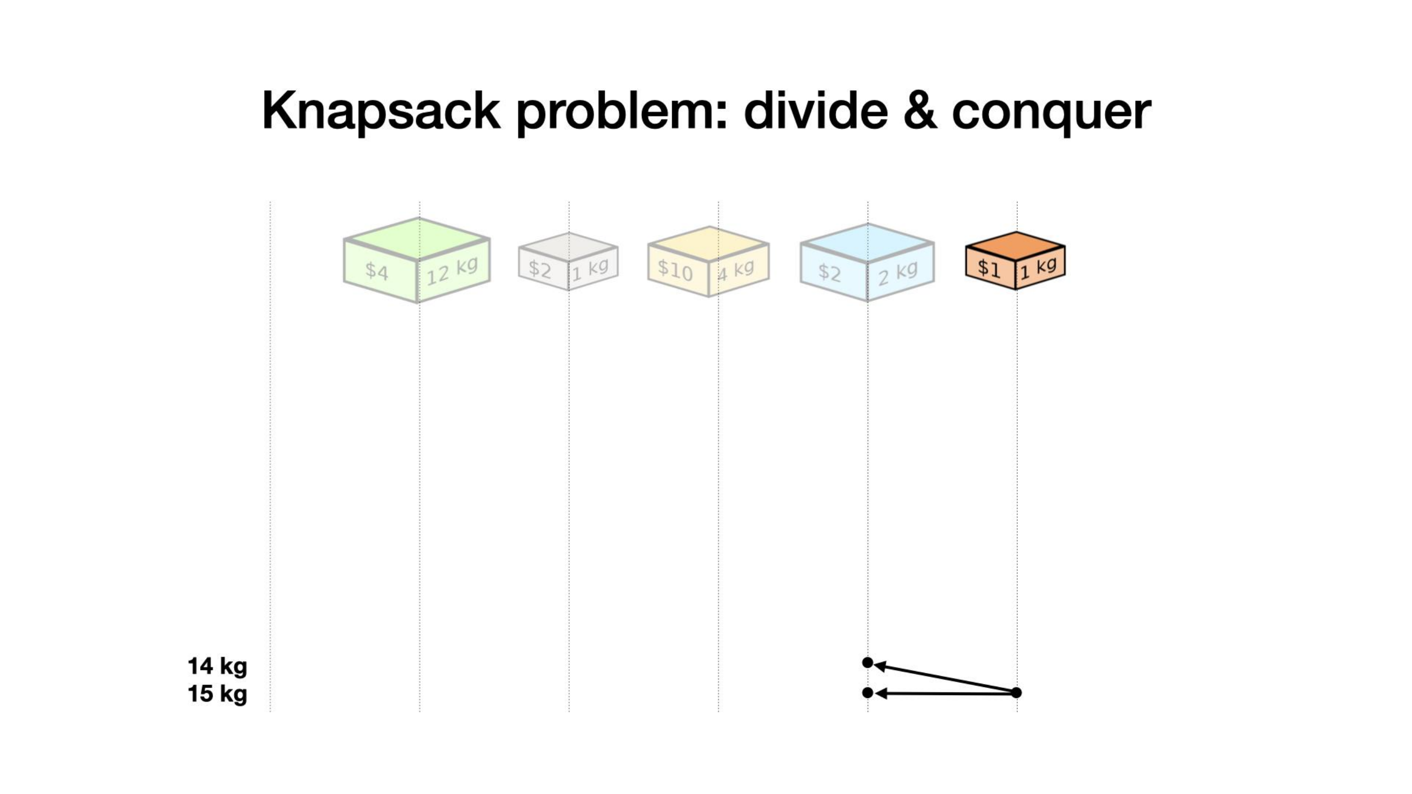
- This approach computes the same sub-problems again and again.
- In the recursion tree, K(1, 1) is being evaluated twice.
- The time complexity of this naive recursive solution is exponential (2^n).

```
In the following recursion tree, K() refers
to knapSack(). The two parameters indicated in the
following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}
                       K(n, W)
                       K(3, 2)
                   /            \
                 /                \
            K(2, 2)                  K(2, 1)
          /       \                  /    \
        /           \              /        \
       K(1, 2)      K(1, 1)        K(1, 1)     K(1, 0)
       /  \         /   \              /          \
     /      \     /       \          /              \
K(0, 2)  K(0, 1)  K(0, 1)  K(0, 0)  K(0, 1)   K(0, 0)
Recursion tree for Knapsack capacity 2
units and 3 items of 1 unit weight.
```

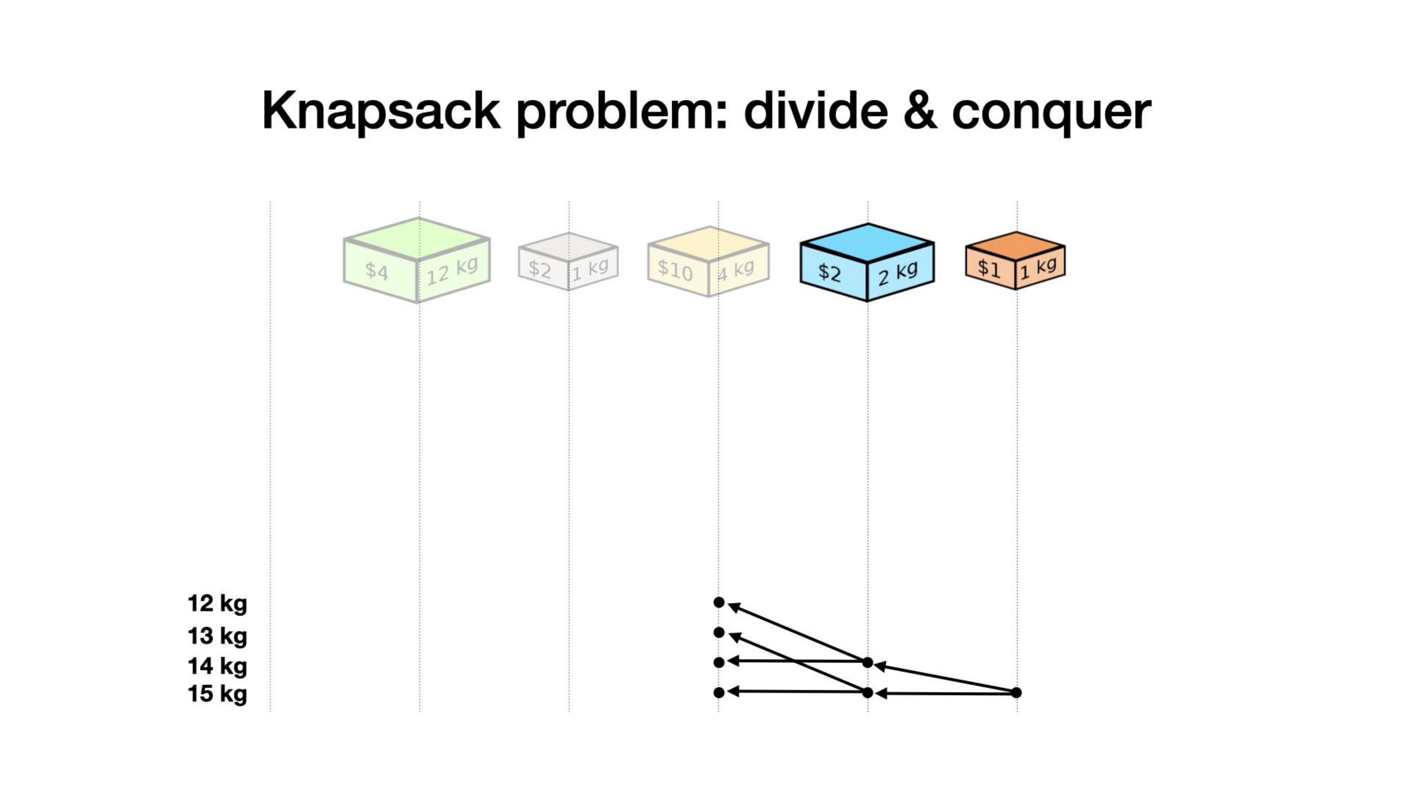
Knapsack Problem: Subproblem Space



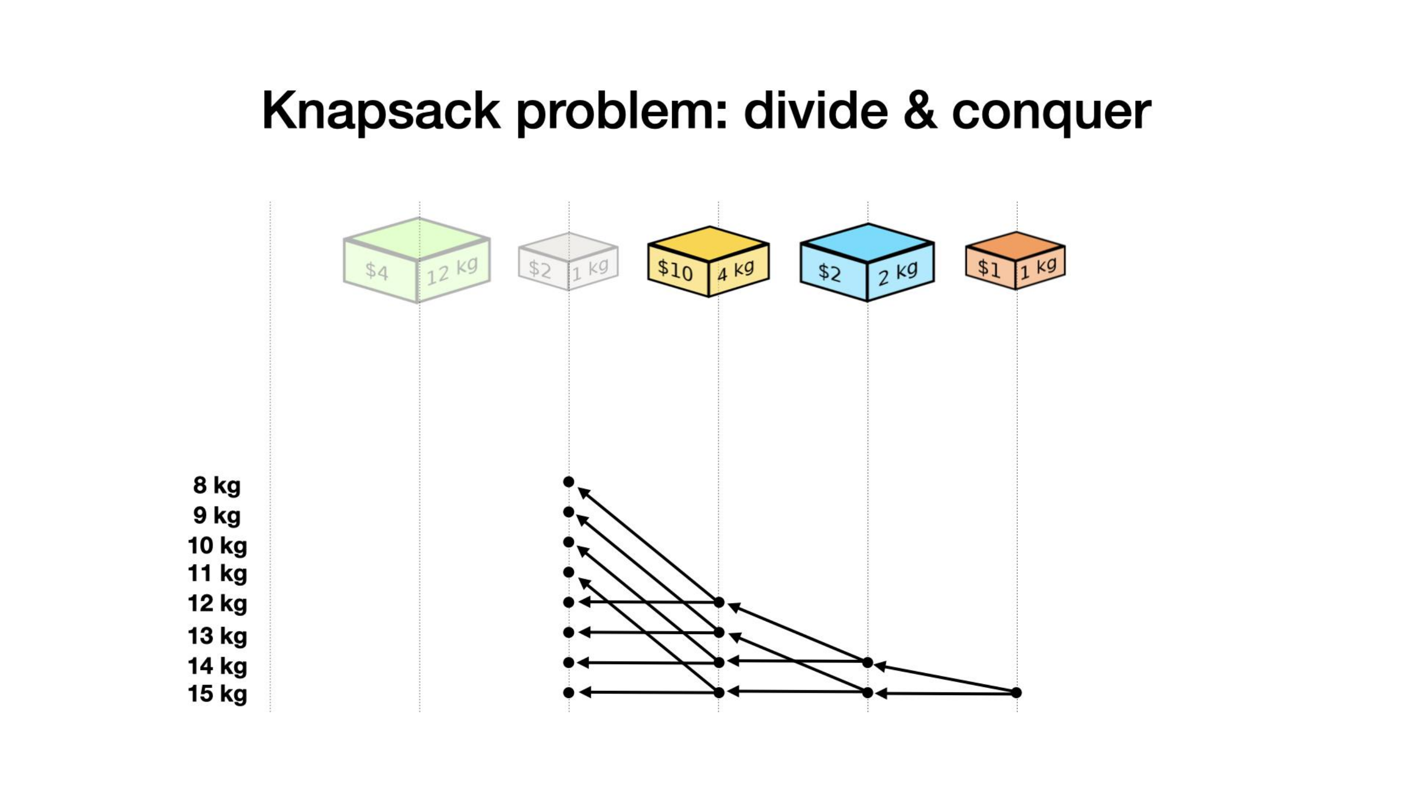
Knapsack Problem: Subproblem Space



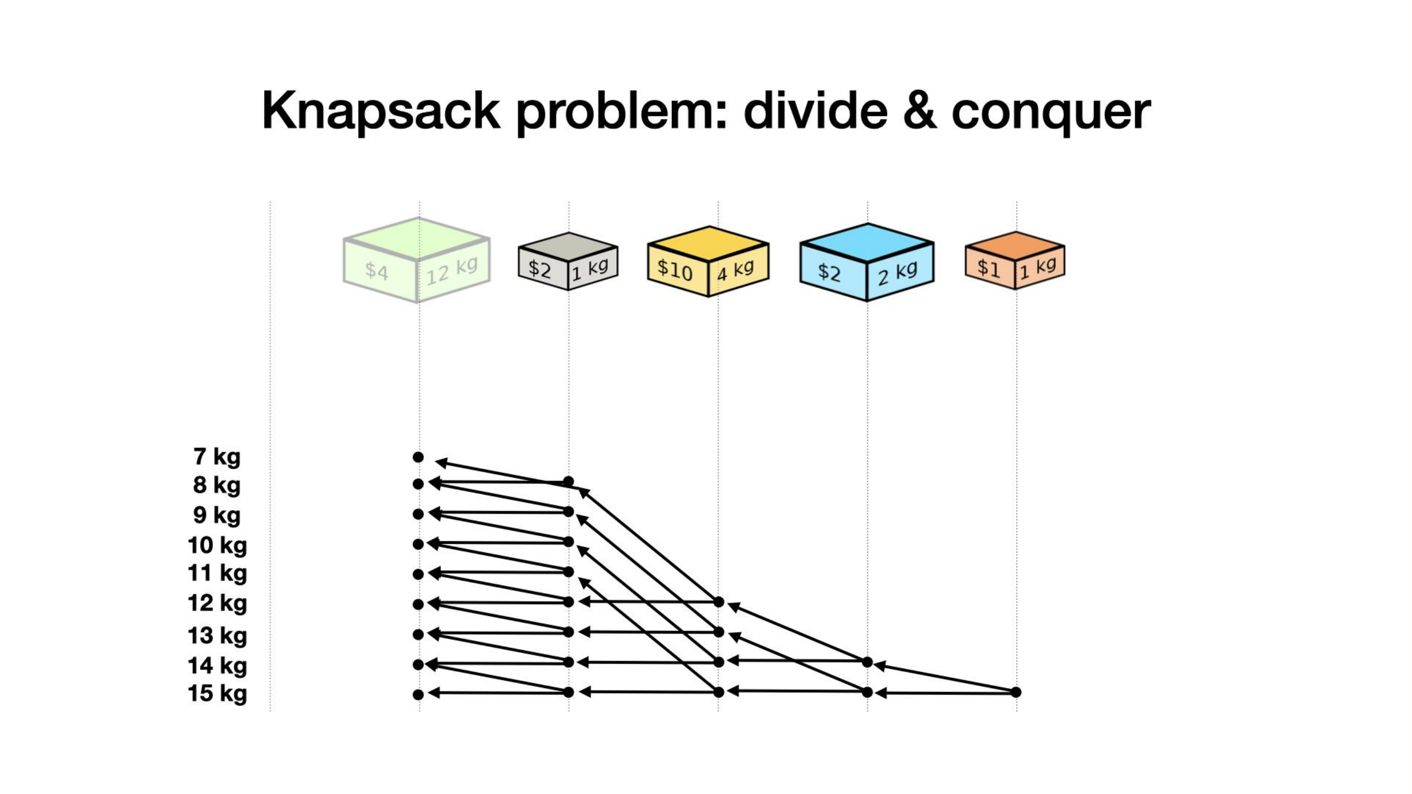
Knapsack Problem: Subproblem Space



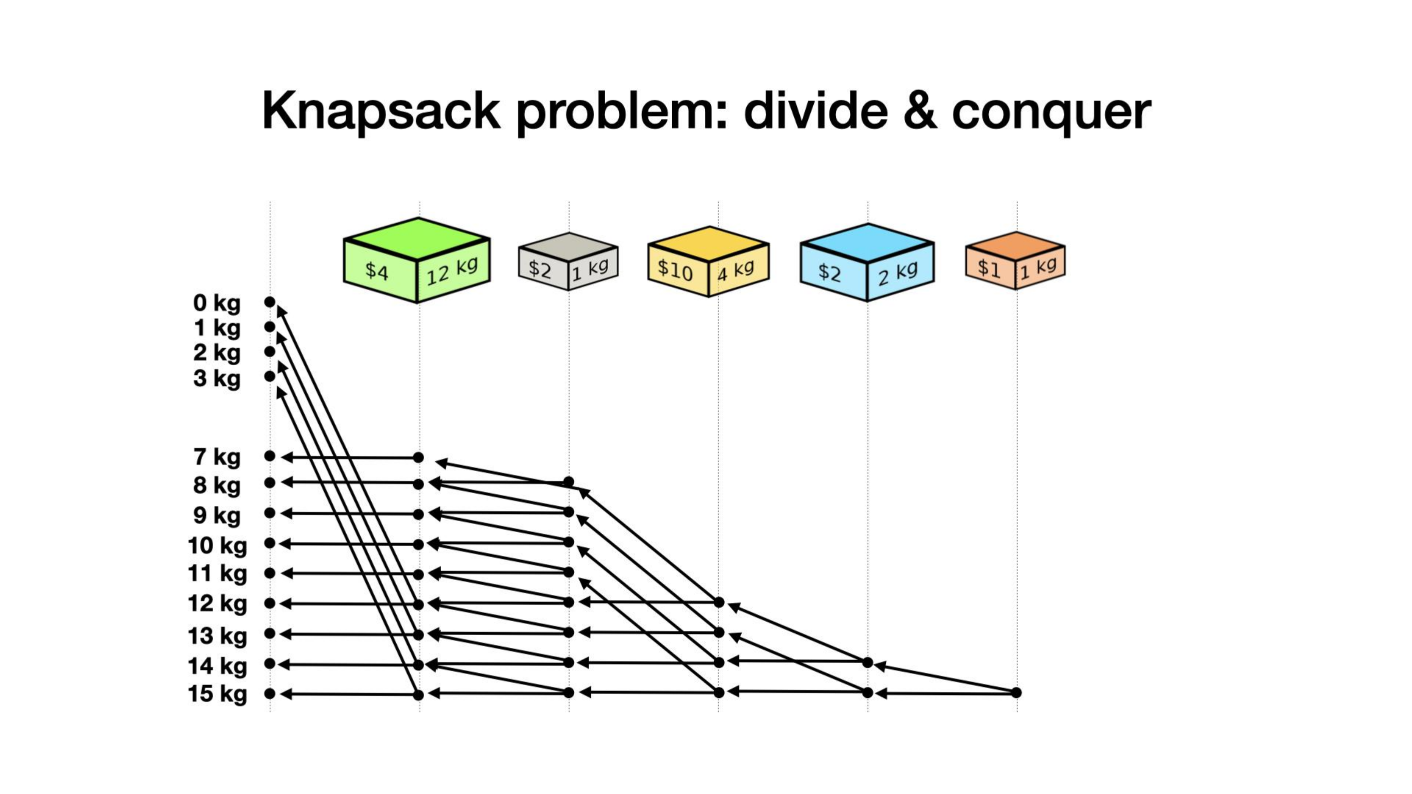
Knapsack Problem: Subproblem Space



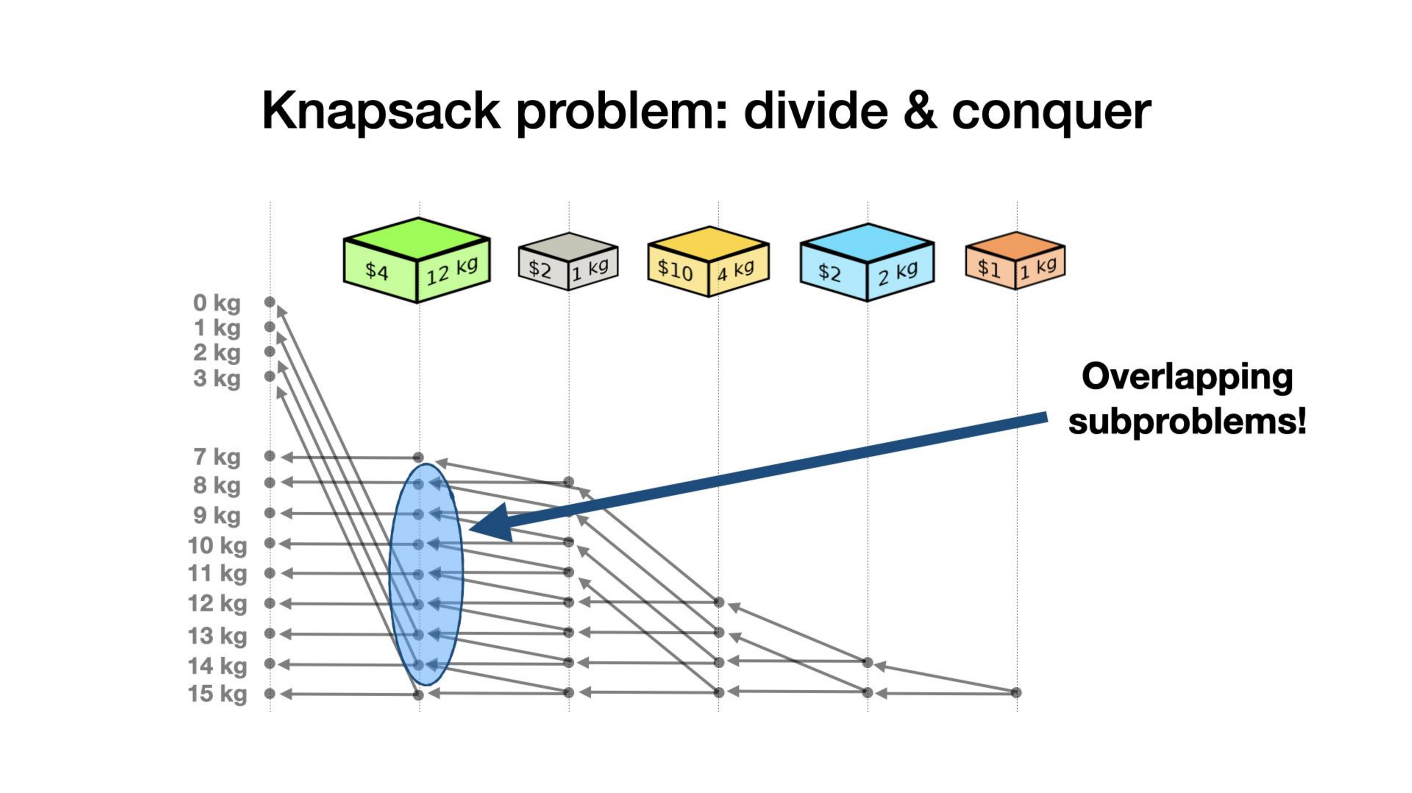
Knapsack Problem: Subproblem Space



Knapsack Problem: Subproblem Space



Knapsack Problem: Subproblem Space



Dynamic Programming

Top-down approach

- Use recursive algorithm
- Memoize solutions for subproblems

Bottom-up approach

- Solve smaller subproblems first
- Then build up on those to solve bigger subproblems
- Repeat

Dynamic Programming

1. In a DP[i][j] table, consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows.
2. The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’.
3. Now we have to take a maximum of two possibilities,
    a. if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j]
    b. but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row.
    c. So we take the maximum of these two possibilities to fill the current state.

- **Time Complexity: O(N*W)**.

  where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.

Dynamic Programming (example)

```
Let weight elements = {1, 2, 3}
Let weight values = {10, 15, 40}
Capacity=6

   0   1   2   3   4   5   6

0  0   0   0   0   0   0   0

1  0  10  10  10  10  10  10

2  0  10  15  25  25  25  25

3  0
```

Explanation:

```
For filling 'weight = 2' we come
across 'j = 3' in which
we take maximum of
(10, 15 + DP[1][3-2]) = 25
  |        |
'2'       '2 filled'
not filled
```

```
   0   1   2   3   4   5   6

0  0   0   0   0   0   0   0

1  0  10  10  10  10  10  10

2  0  10  15  25  25  25  25

3  0  10  15  40  50  55  65
```

Explanation:

```
For filling 'weight=3',
we come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3'
we come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3'
we come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 65
```

Knapsack Problem: Dynamic Programming

	$4, 12 kg	$2, 1 kg	$10, 4 kg	$2, 2 kg	$1, 1 kg
0 kg					
1 kg					
2 kg					
3 kg					
4 kg					
5 kg					
6 kg					
7 kg					
8 kg					
9 kg					
10 kg					
11 kg					
12 kg					
13 kg					
14 kg					
15 kg					

Knapsack Problem: CodeForces

PROBLEMS SUBMIT CODE MY SUBMISSIONS STATUS STANDINGS CUSTOM INVOCATION

Problems

#	Name			
A	Knapsack (Brute Force)	standard input/output 2 s, 256 MB		
B	Knapsack Problem	standard input/output 0.25 s, 256 MB		

Complete problemset

https://codeforces.com/group/M5kRwzPJIU/contest/369148

See You next week!