Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 9 and Homework 9: next Monday 11/21 @ 11:59 pm
  - Assignment 3: ~~Monday 11/28~~ Friday 12/9 @ 11:59 pm
  - Assignment 4: Friday 12/9 @ 11:59 pm

Recap …

- Greedy algorithms
  - elegant but hardly correct
  - optimal substructure
  - greedy choice property
- Without the greedy choice property
  - have to solve all subproblems
  - can be done recursively
- Memoization
  - still recursive
  - avoid solving the same subproblem twice

Recap …

- Dynamic Programming
  - avoid solving the same subproblem twice
  - iterative:
    - start with smaller subproblems then larger subproblems, …
  - sometimes possible to optimize space needed

Recap …

- Fibonaaci
  - inefficient recursive solution
  - memorization solution
  - dynamic programming
    - with space optimization

Solving Dynamic Programming Problems

- Can you solve the problem using subproblems?
  - What is the first decision to make to solve the problem?
  - What subproblem(s) emerge out of the that first decision?
- Can you make the first decision without having to wait for the solution of the subproblems?
  - If yes, that’s a greedy algorithm! Congratulations!

Solving Dynamic Programming Problems

- If you have to wait for subproblem solutions to make the first decision, try the following steps
- start with a recursive solution
- if inefficient, do you have overlapping subproblems?
- identify the unique subproblems
- solve them from smaller to larger
- This is dynamic programming!
- Optimize space if possible

This Lecture

- Dynamic Programming Problems
    - Unbounded Knapsack
    - 0/1 Knapsack
    - Subset Sum
    - Edit Distance
    - Longest Common Subsequence

The unbounded knapsack problem

Given a knapsack that can hold a weight limit L, and a set of n types items that each has a weight (w_i) and value (v_i), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?

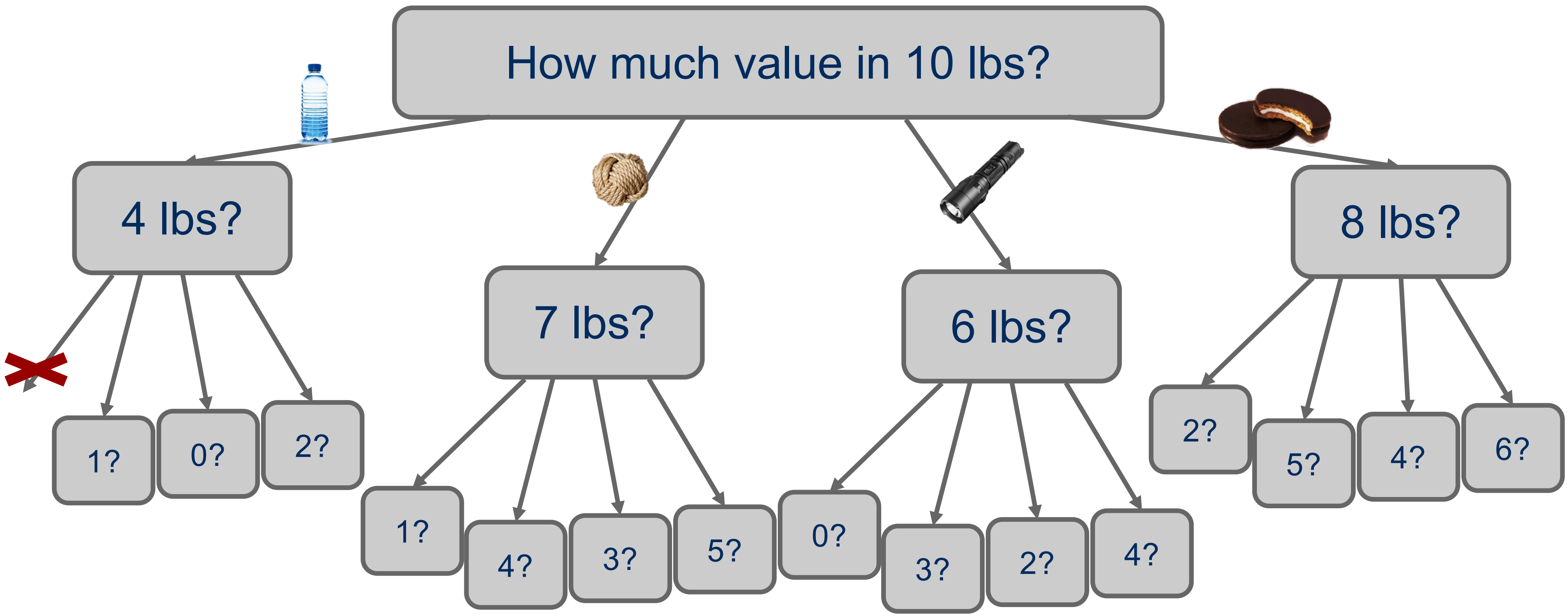
weight:	6	3	4	2
value:	30	14	16	9

10 lb.
capacity

Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

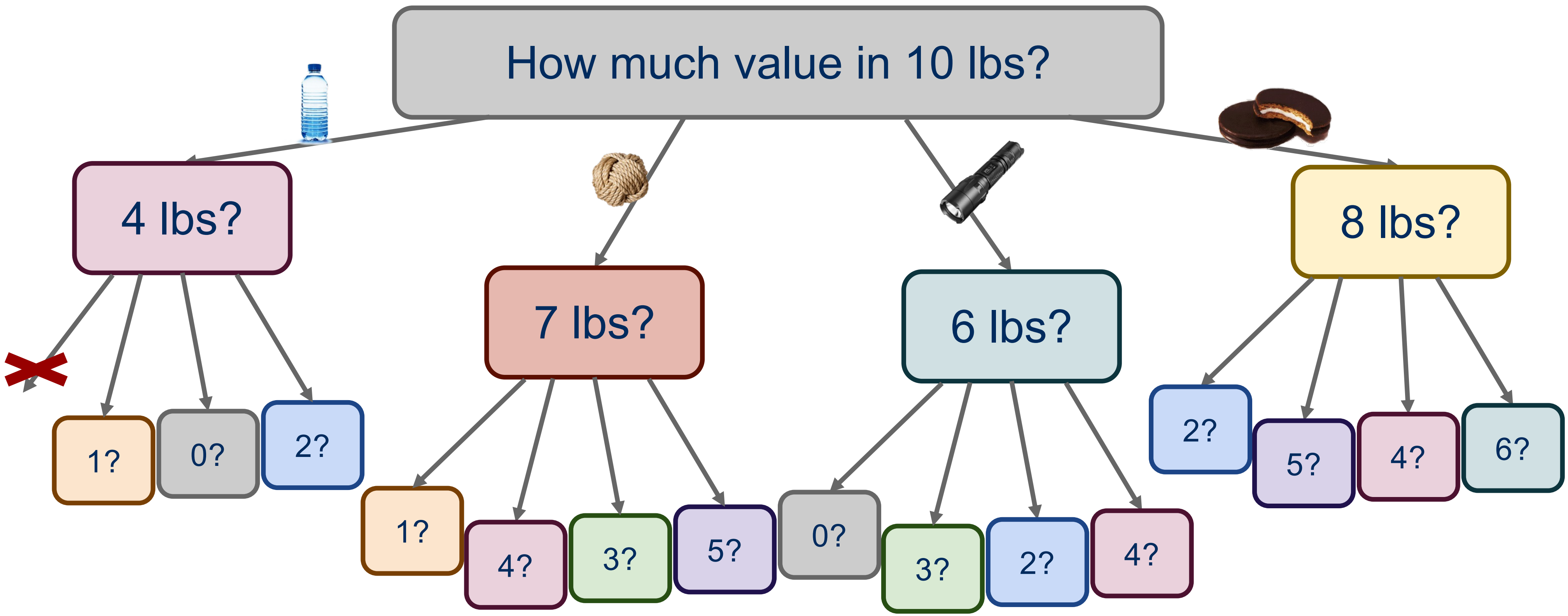
10 lb. capacity



Overlapping Subproblems!

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Bottom-up Solution

weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

Bottom-up solution

```
K[0] = 0
for (l = 1; l <= L; l++) {
      int max = 0;
      for (i = 0; i < n; i++) {
            if (w_i <= l && v_i + K[l - w_i]) > max) {
                  max = v_i + K[l - w_i];
            }
      }
      K[l] = max;
}
```

A greedy algorithm

- Try adding as many copies of highest value per pound item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moonpie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb knapsack?
  - 44
    - Bogus!

But why doesn’t the greedy algorithm work for this problem?

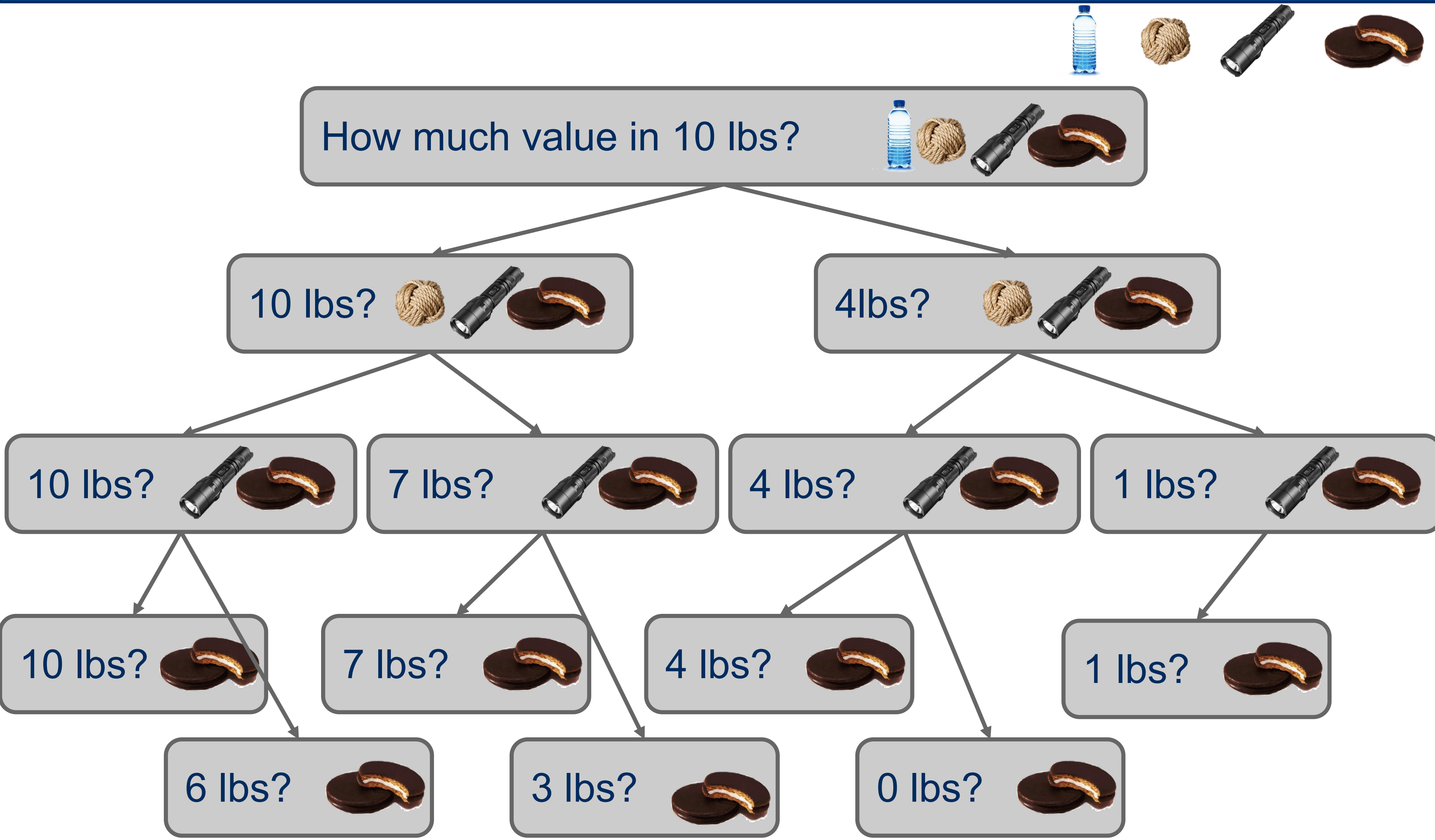
The greedy choice property is missing!

The 0/1 knapsack problem

- What if we have a finite set of items that each has a weight and value?
  - Two choices for each item:
    - Goes in the knapsack
    - Is left out
- What would be our first decision?
- What suproblems emerge?

Recursive solution

weight:	6	3	4	2
value:	30	14	16	9

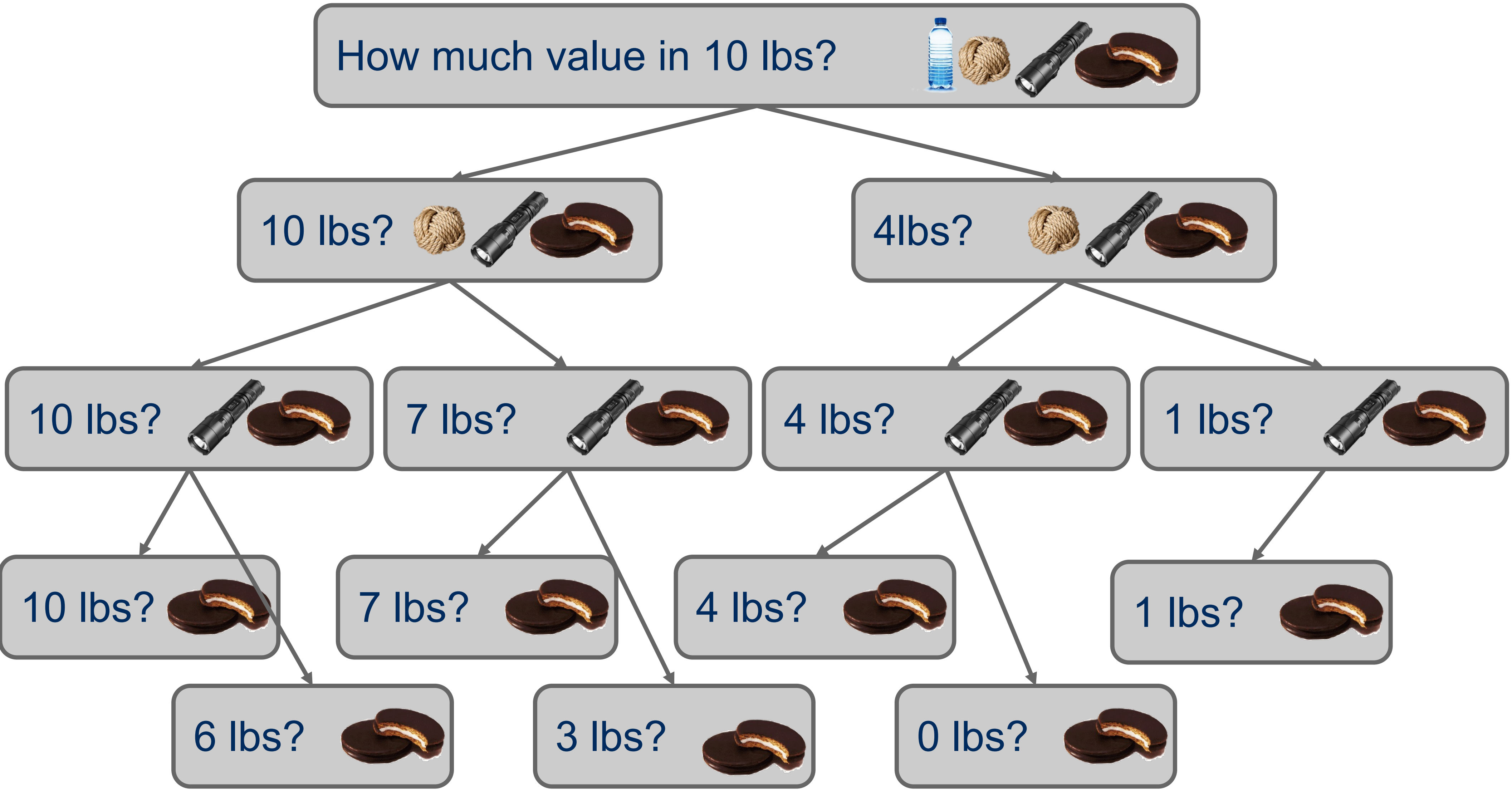


Recursive solution

```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0 };
    //try placing the n-1 item
    if (wt[n-1] > L) {
        return knapSack(wt, val, L, n-1)
    }
    else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1)
                   );
    }
}
```

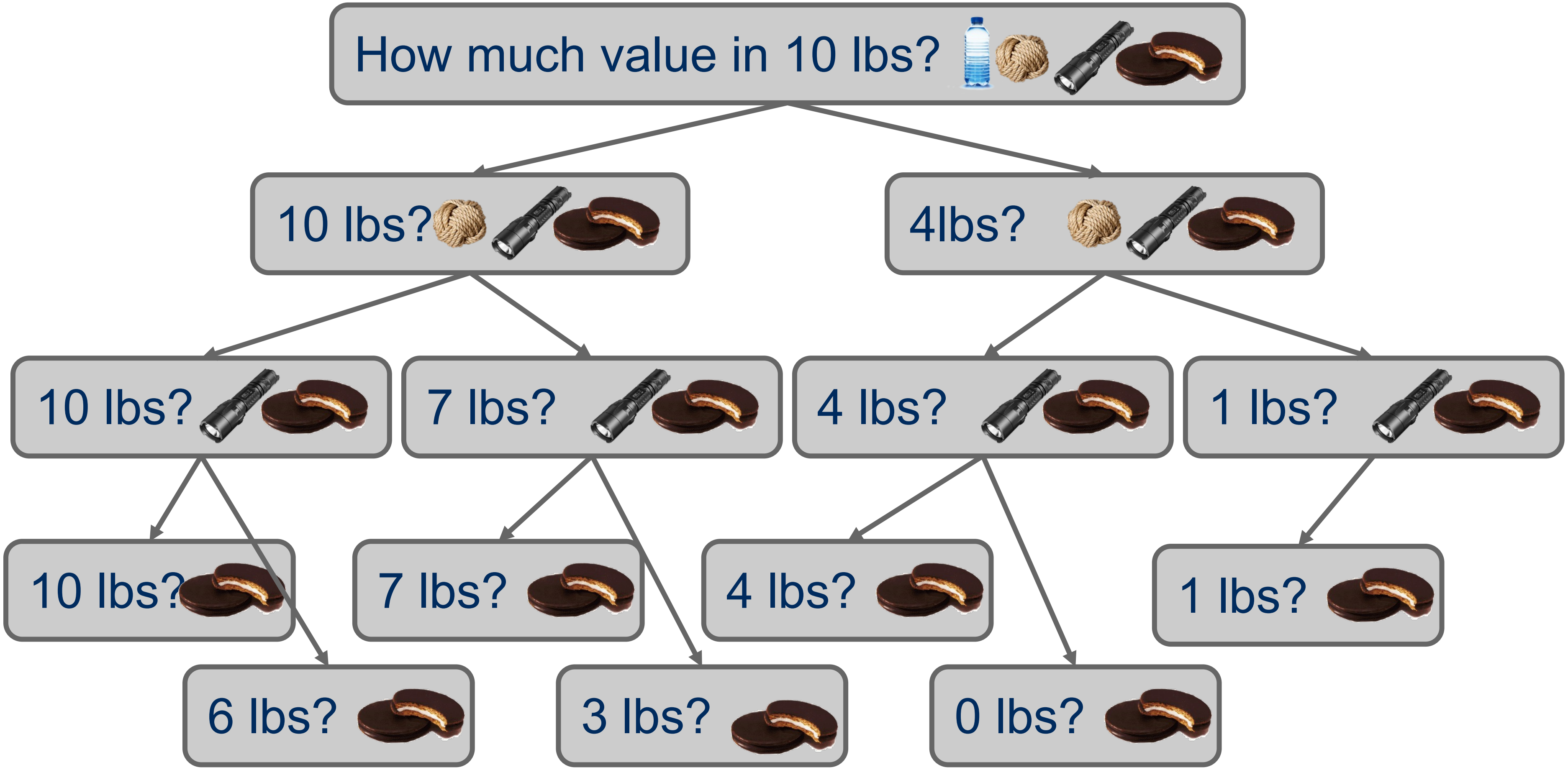
Recursive solution

weight:	6	3	4	2
value:	30	14	16	9



Subproblems

- What are the unique subproblems?



The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
K[n+1][L+1]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0								
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16						
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0									

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	16	30	30	39	44	46

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
            //try to add item i-1
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                                          K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```