UCSC Silicon Valley Extension

Advanced C Programming

Recursion

Instructor: Radhika Grover

Overview

- Exhaustive search techniques using recursion and backtracking
- Applications:
  - Maze solving
  - Knapsack problem
  - Variants of minimum spanning trees

Exhaustive search

- Tries all possible solutions - also called brute force.
  - Generates all possible solutions (candidates)
  - Checks if a candidate solution is valid
- Advantage: simple to implement
- Disadvantage: cost to generate all candidate solutions

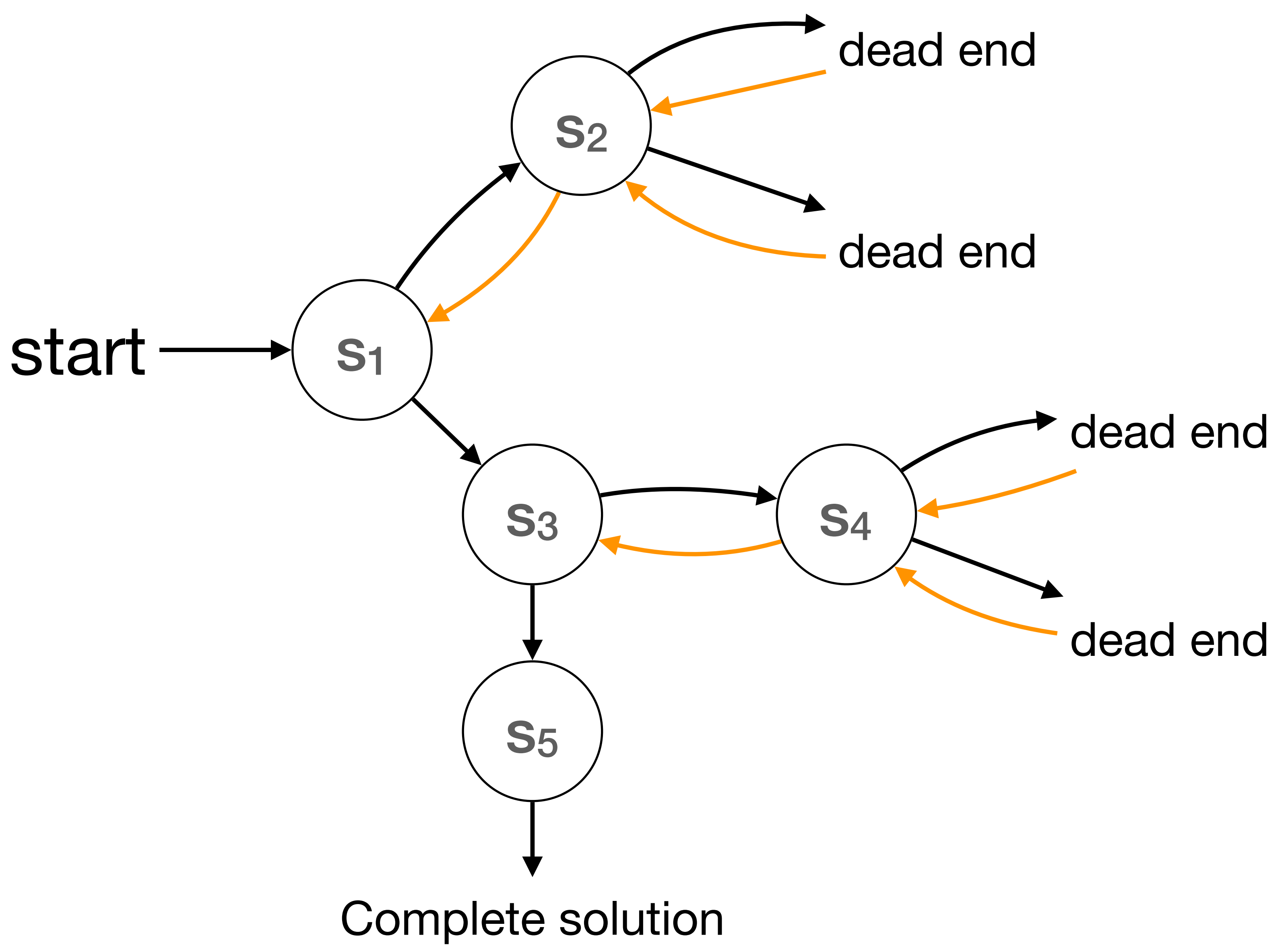
Backtracking

- Abandons a partial solution that cannot be completed to create a complete solution (dead end).
- Returns back to a previous step and chooses a different back.
- Improves exhaustive search

Backtracking

S_1, S_2, S_3, S_4, S_5: partial solutions

backtracking



Knapsack Problem

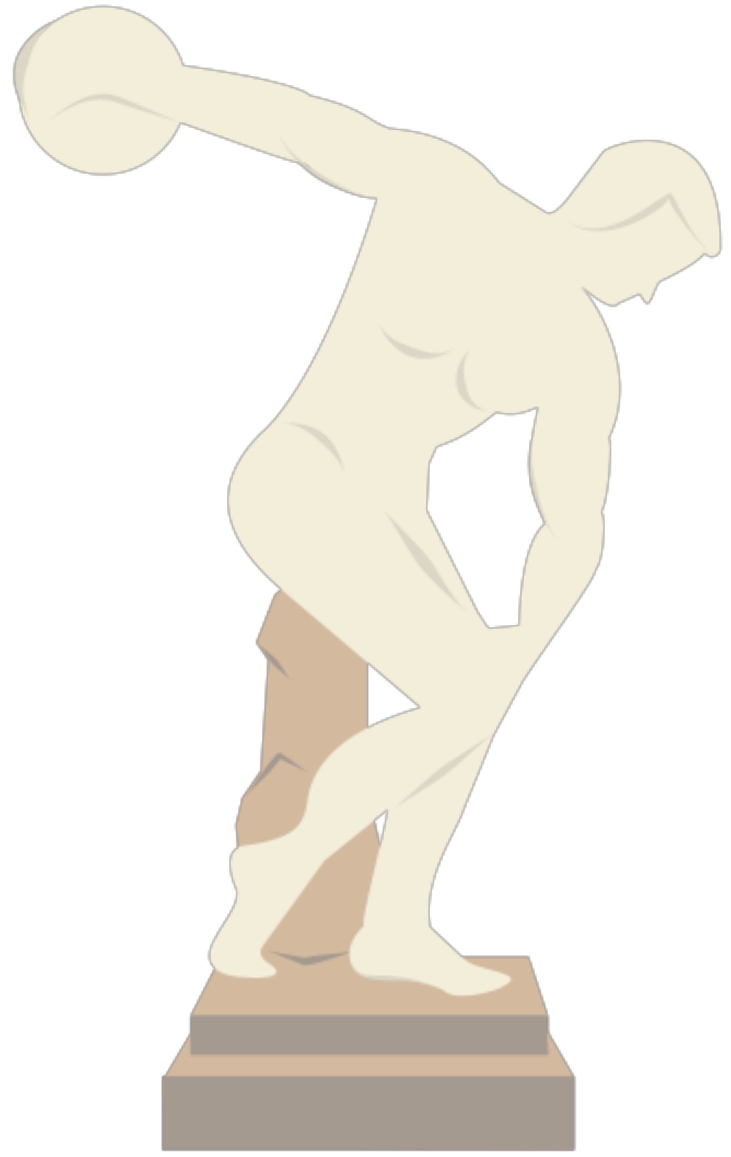
$$Maximize \sum_{j=1}^{n} v_j x_j$$

$$subject\ to \sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \varepsilon \{0, 1\}, j = 1, 2, \ldots, n$$

Cost: $20

Weight: 10 lbs



Cost: $6

Weight: 2 lbs

Cost: $15

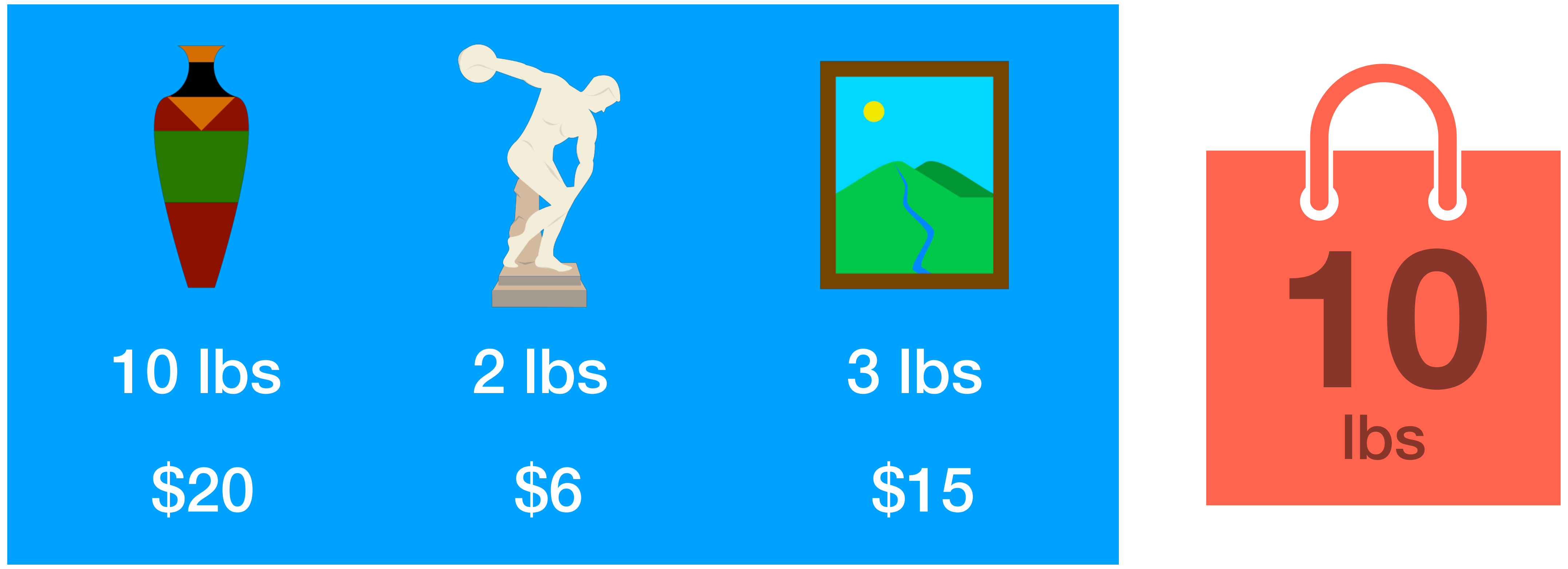
Weight: 3 lbs


Cost: $20
Weight: 10 lbs
Cost: $6
Weight: 2 lbs
Cost: $15
Weight: 3 lbs

0-1 Knapsack Problem

Goal: Maximize Value

Store in bag with a capacity of 10 lbs



Types of Knapsack Problems

- **Bounded**
  - Fixed amount m_j of each type

$$x_j \varepsilon \{0, 1 \ldots, mj\}, j = 1, 2, \ldots, n$$

- **Unbounded**
  - Unlimited amount of each type

$$x_j \geq 0, \mathit{integer}\ j = 1, 2, \ldots, n$$

- **Multiple Choice**
  - Choose exactly 1 item j from each of K classes N_i, i=1,…,k

$$\mathit{Maximize} \sum_{i=1}^{k} \sum_{j \in N_i} v_{ij} x_{ij}$$

$$\mathit{subject\ to} \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \leq c,$$

$$\sum_{j \in N_i} x_{ij} = 1, i = 1, \ldots, k$$

$$x_{ij} \in 0, 1, i = 1, \ldots, k, j \in N_i$$

Types of Knapsack Problems (continued)

- **Subset-sum**
  - Value v_j is equal to weight w_j for each item in 0-1 Knapsack problem.

Maximize $\sum_{j=1}^{n} w_j x_j$
subject to $\sum_{j=1}^{n} w_j x_j \leq c$
$x_{ij} \in 0, 1, i = 1, \ldots, k, j \in N_i$

- **Fractional**

$0 \leq x_i \leq 1, 1 \leq i \leq n$

- **Multi-constraint**
  - Most general form

Maximize $\sum_{j=1}^{n} v_j x_j$
subject to $\sum_{j=1}^{n} w_{ij} x_j \leq c_i, i = 1, \ldots, m$
$x_j \geq 0\ integer, j = 1, \ldots, n$

Applications

- 0-1: Find an optimal investment plan given n projects, the profit from each project is p_j, cost to invest in a project is w_j, and only c dollars are available.

- Multiple-choice: Choose one of N_i dishes in each of k courses in a restaurant without exceeding amount of c calories.

Applications

- Subset-sum
- Fractional
- Multi-constraint

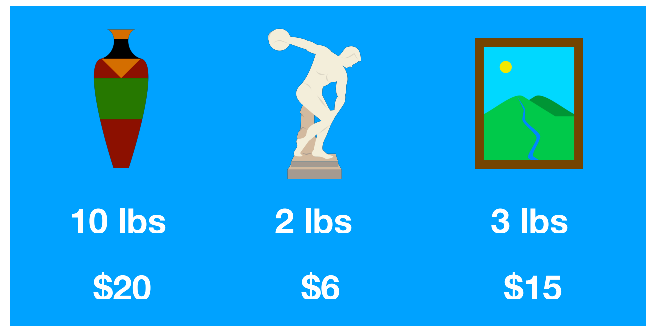
Solutions

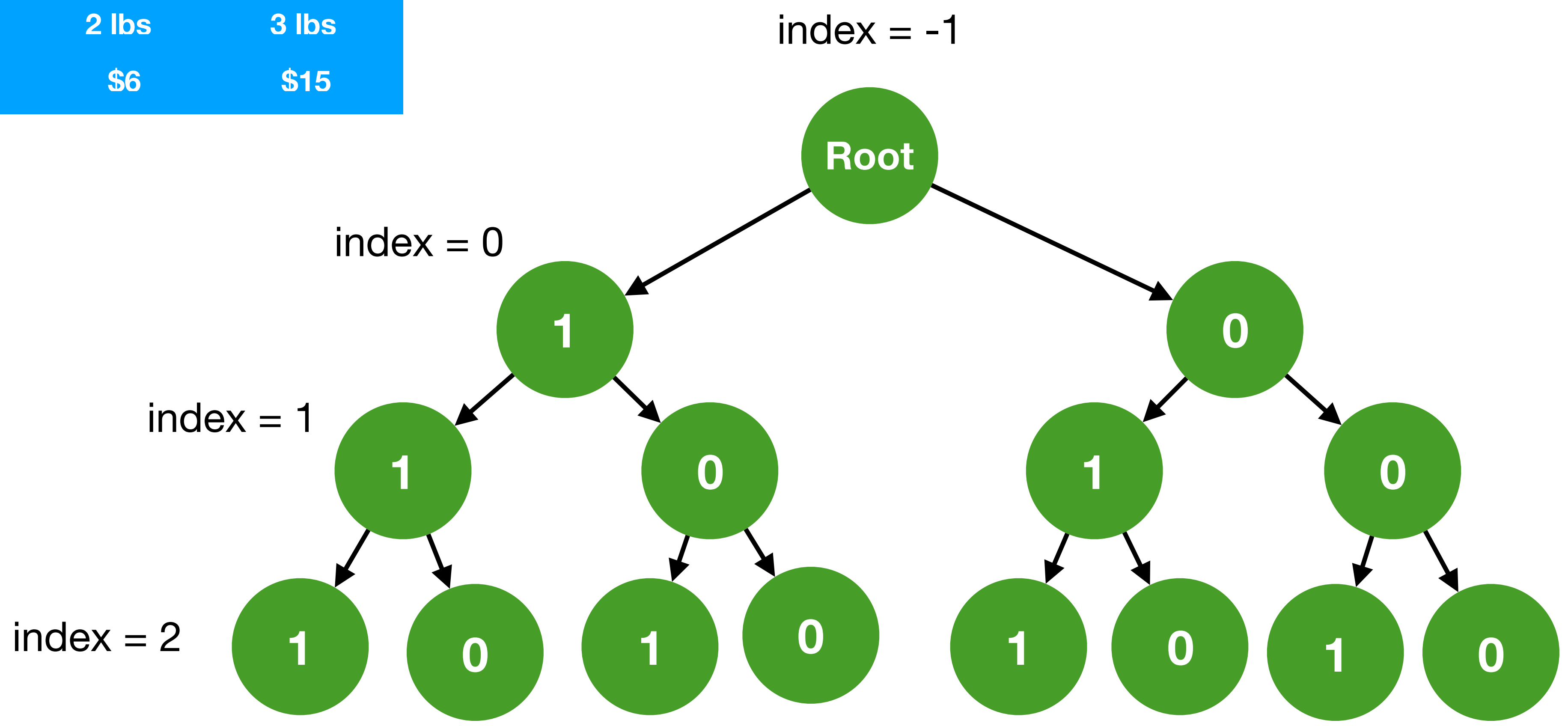
- Exhaustive search
- Greedy algorithm
- Branch and bound
- Dynamic programming
- Approximation algorithms
- Genetic algorithms

Exhaustive Search

- Brute-force search
- Explores all—pick one
- A recursive strategy
- Running time?

Generate binary number

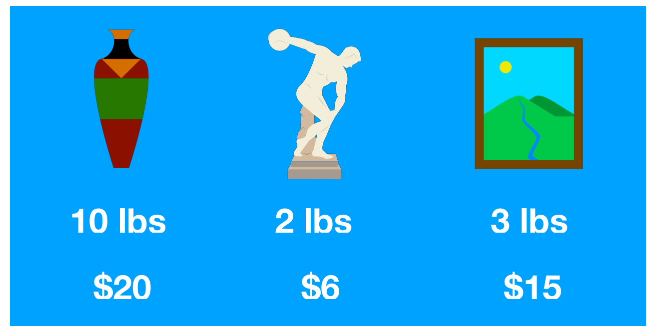


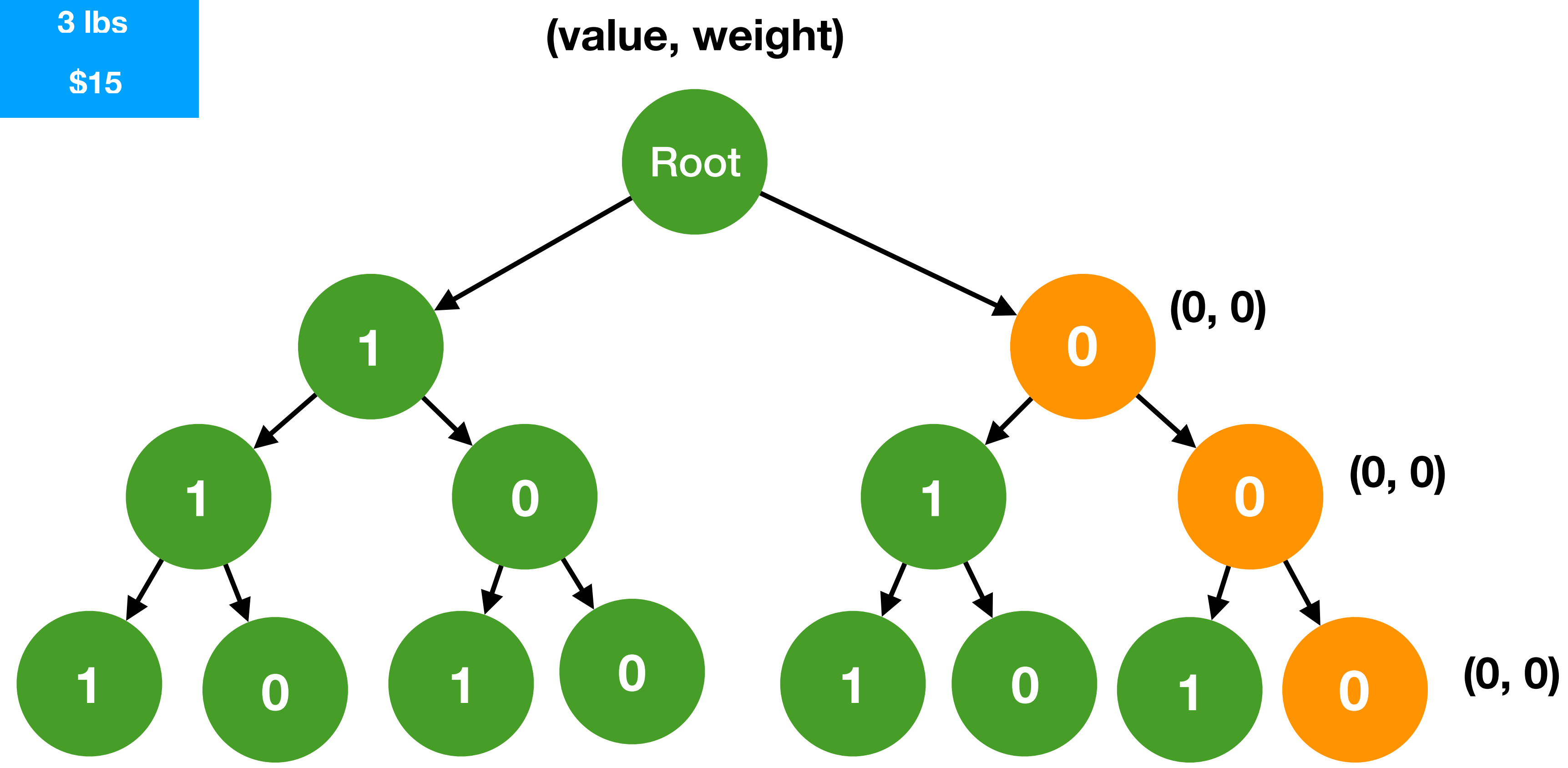


0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

```
current value = value[index] * select + current value
Current weight = weight[index] * select + current weight
```

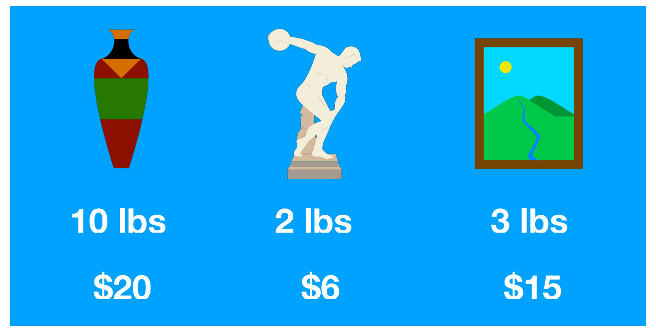
Knapsack : exhaustive search

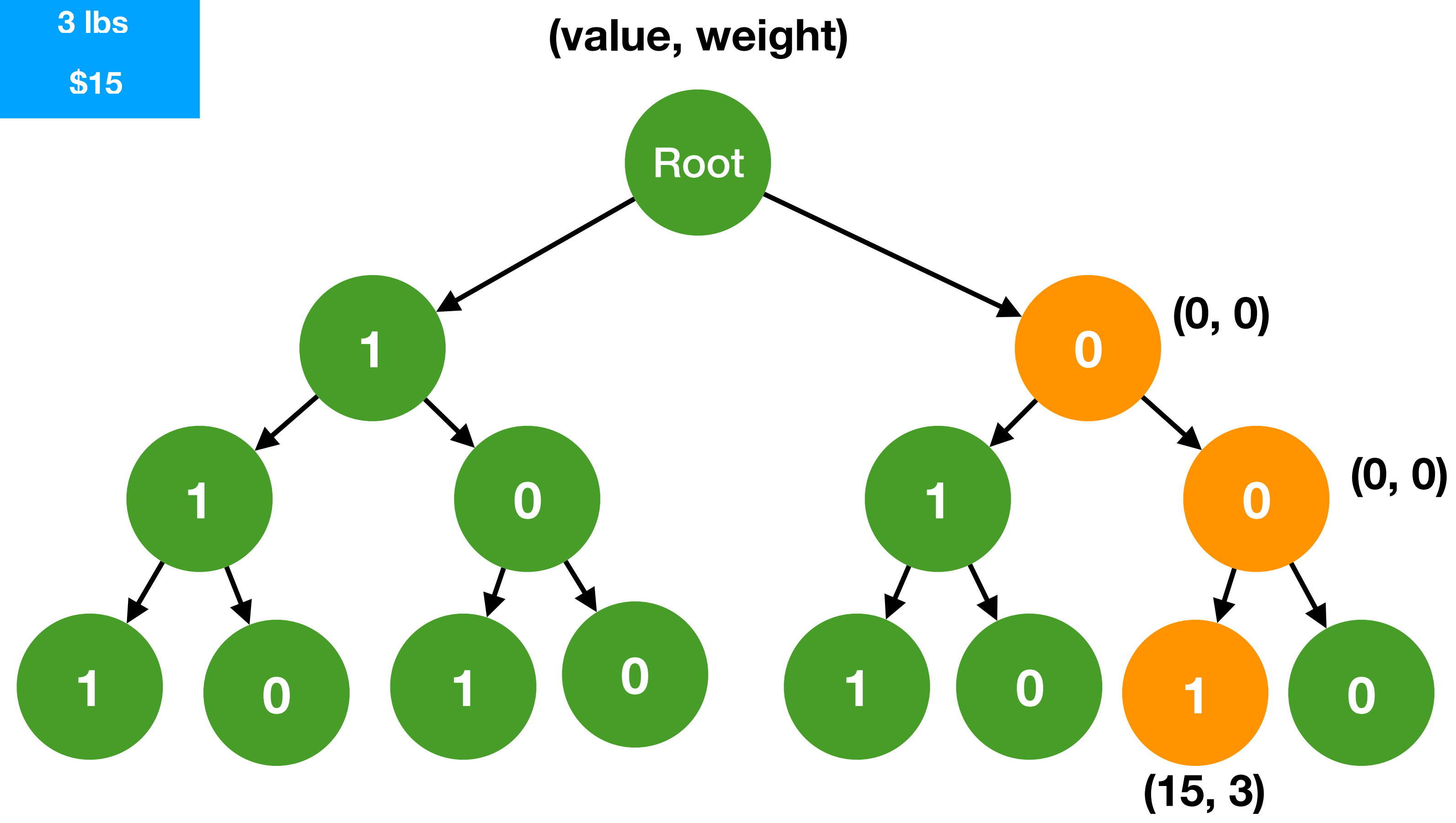




	value	weight
000	0	0
001		
010		
011		
100		
101		
110		
111		

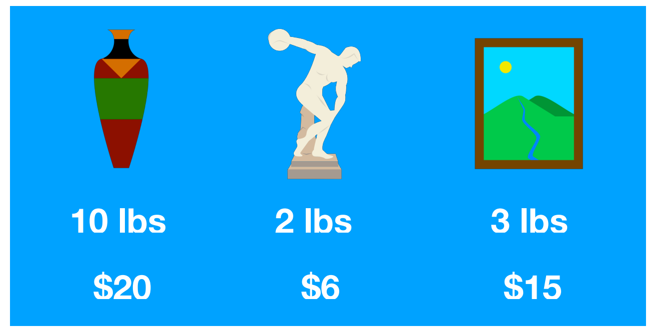
Knapsack exhaustive search

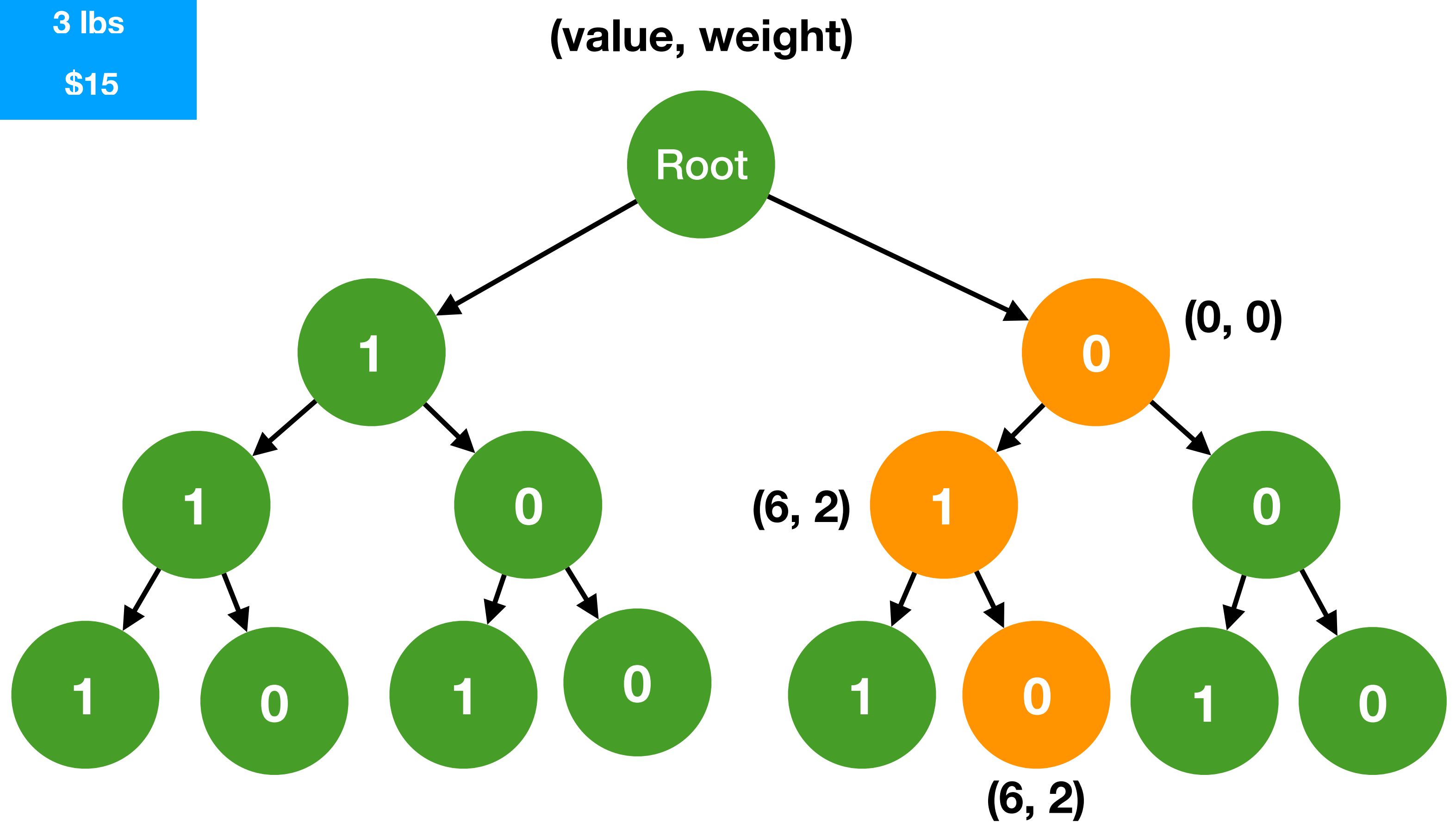




	value	weight
000	0	0
001	15	3
010		
011		
100		
101		
110		
111		

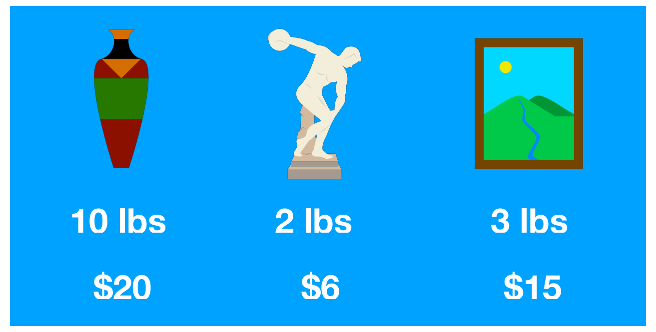
Knapsack : exhaustive search

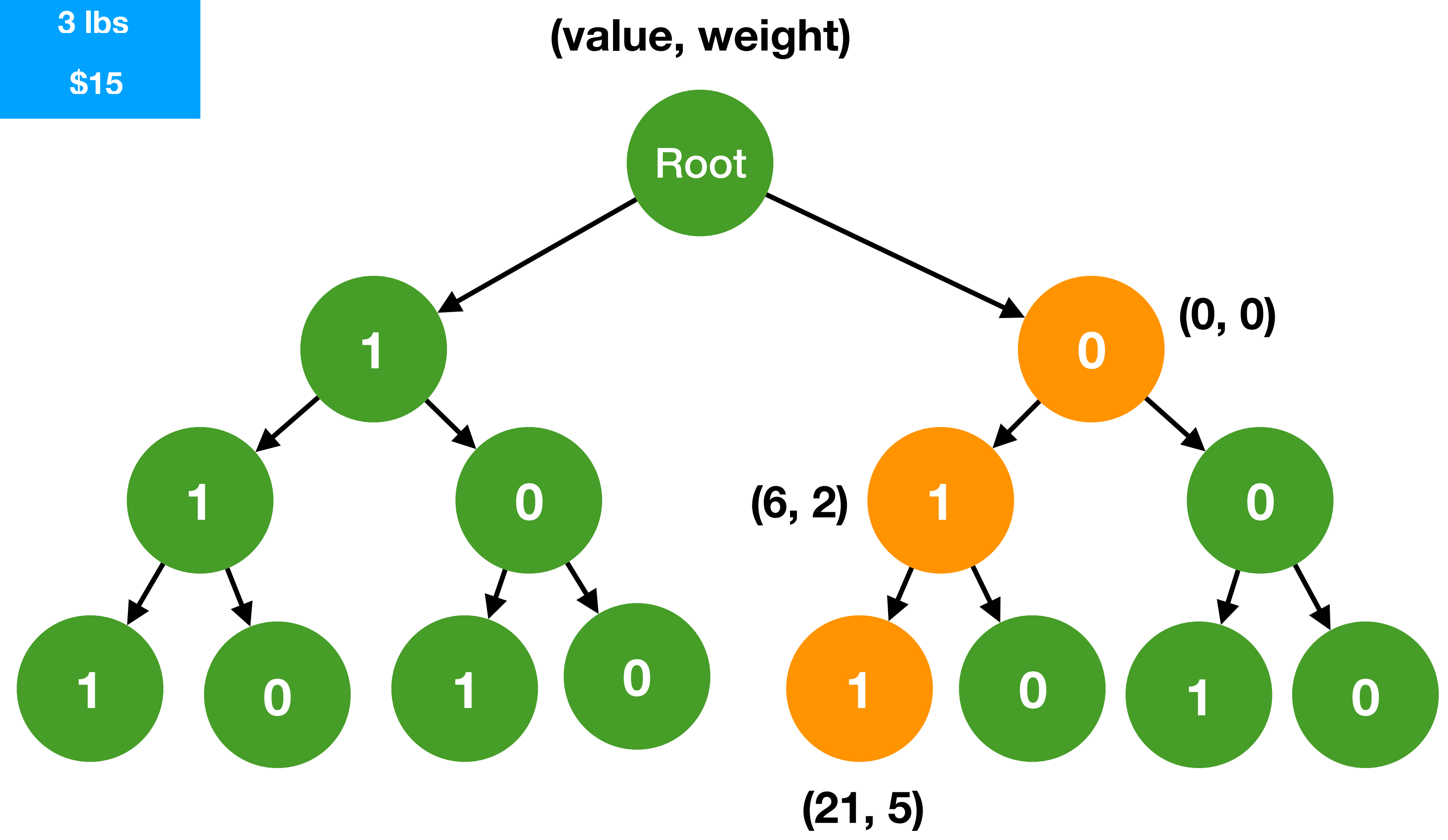




	value	weight
000	0	0
001	15	3
010	6	2
011		
100		
101		
110		
111		

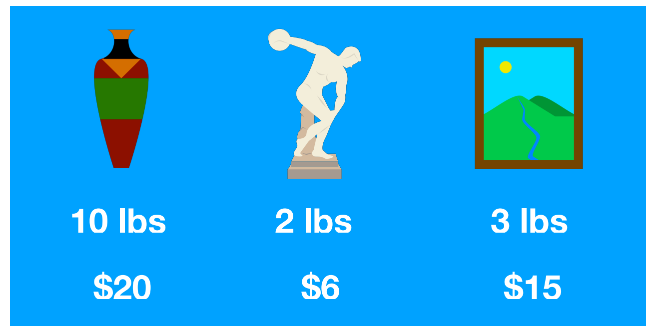
Knapsack : exhaustive search

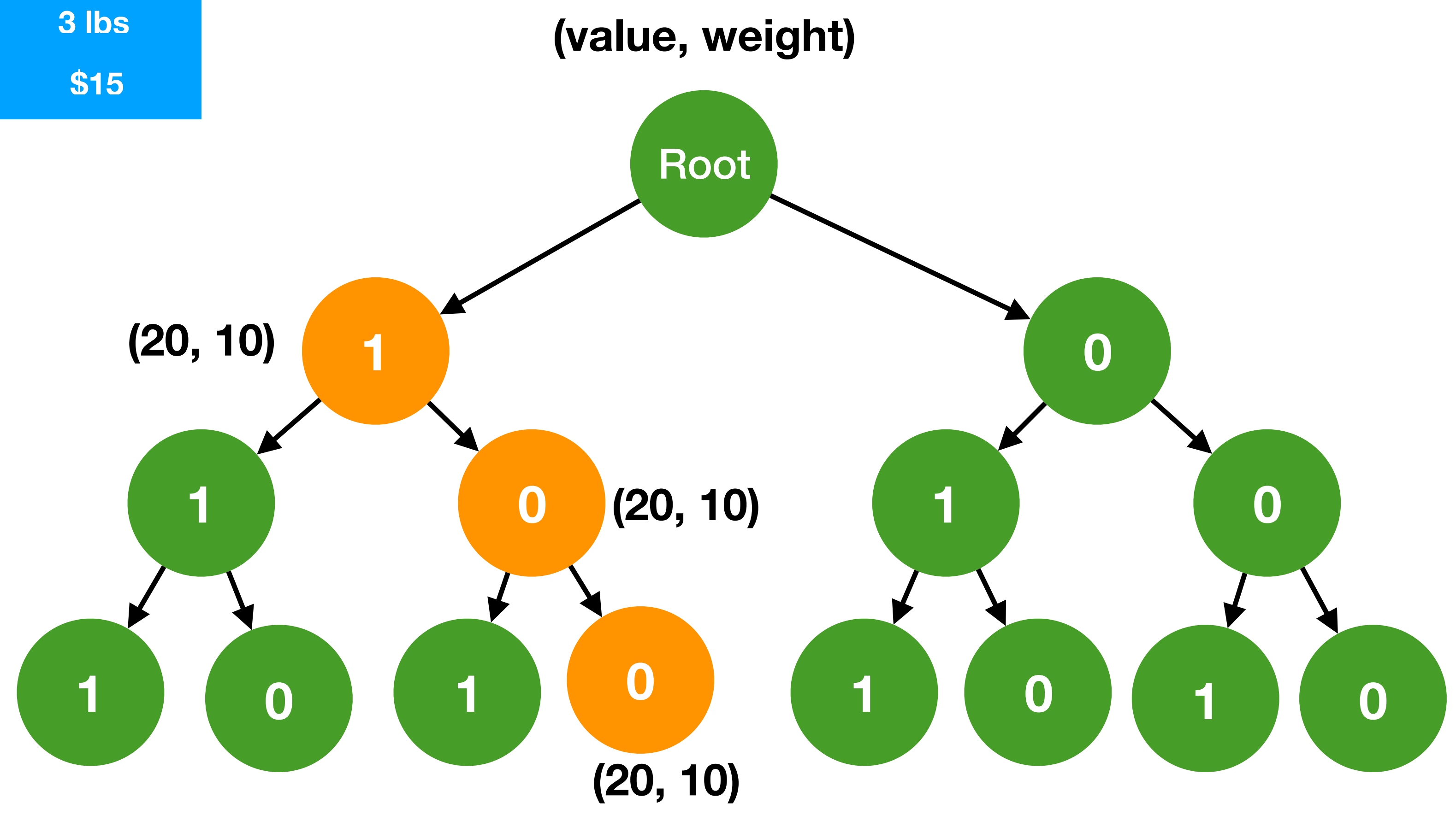




	value	weight
000	0	0
001	15	3
010	6	2
011	21	5
100		
101		
110		
111		

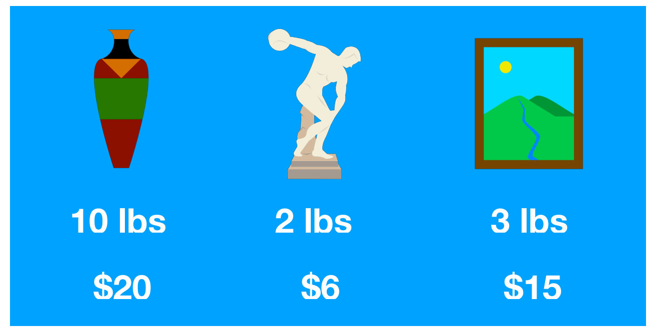
Knapsack : exhaustive search

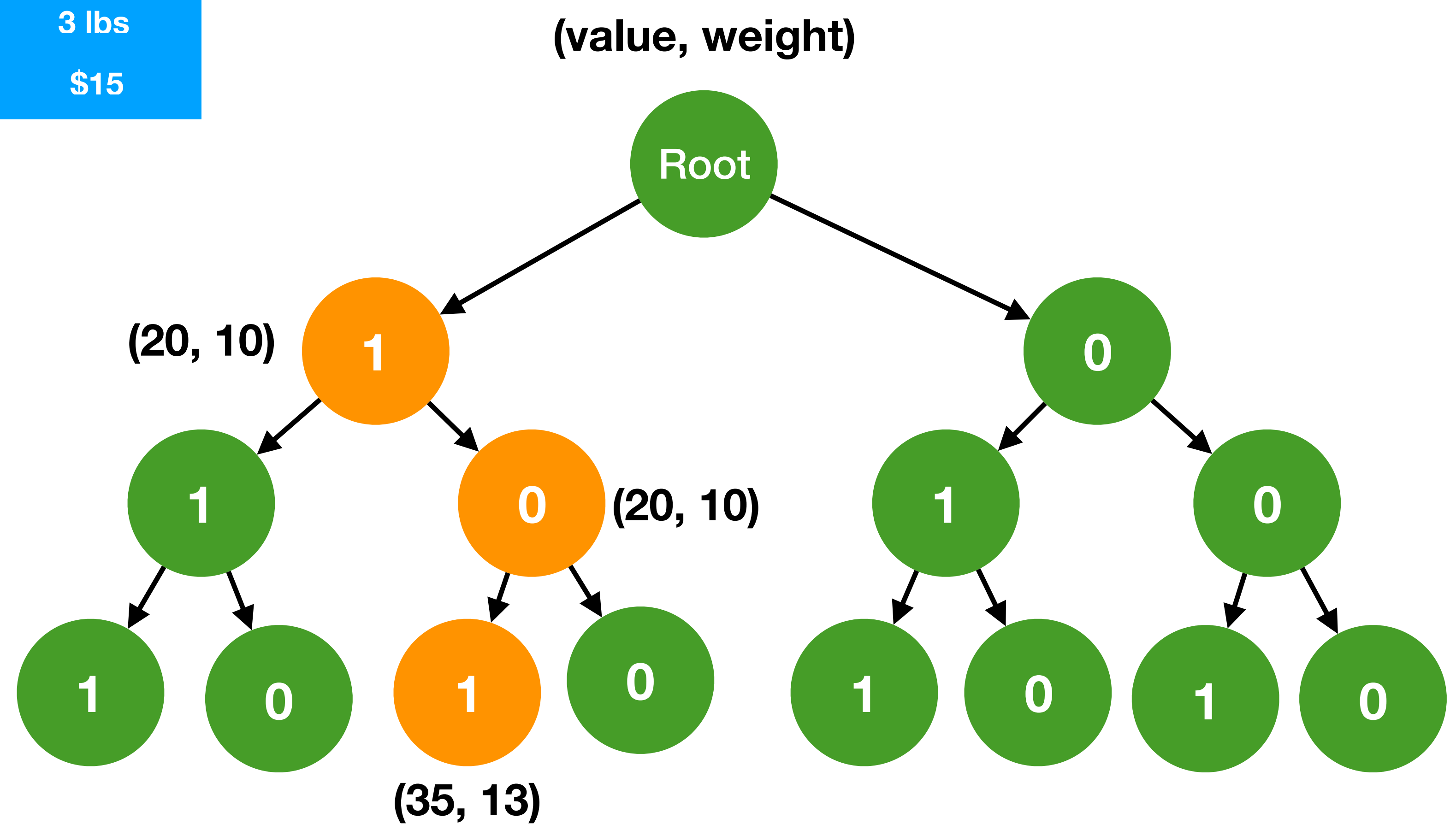




	value	weight
0 0 0	0	0
0 0 1	15	3
0 1 0	6	2
0 1 1	21	5
1 0 0	20	10
1 0 1		
1 1 0		
1 1 1		

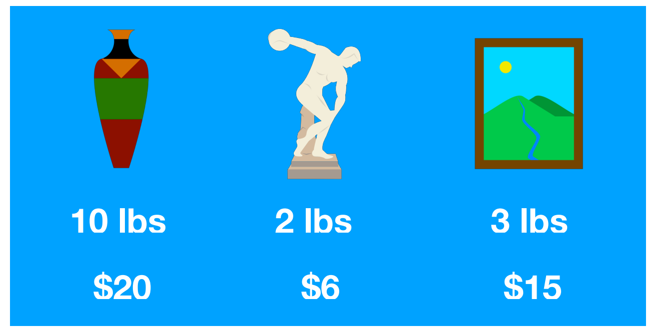
Knapsack : exhaustive search

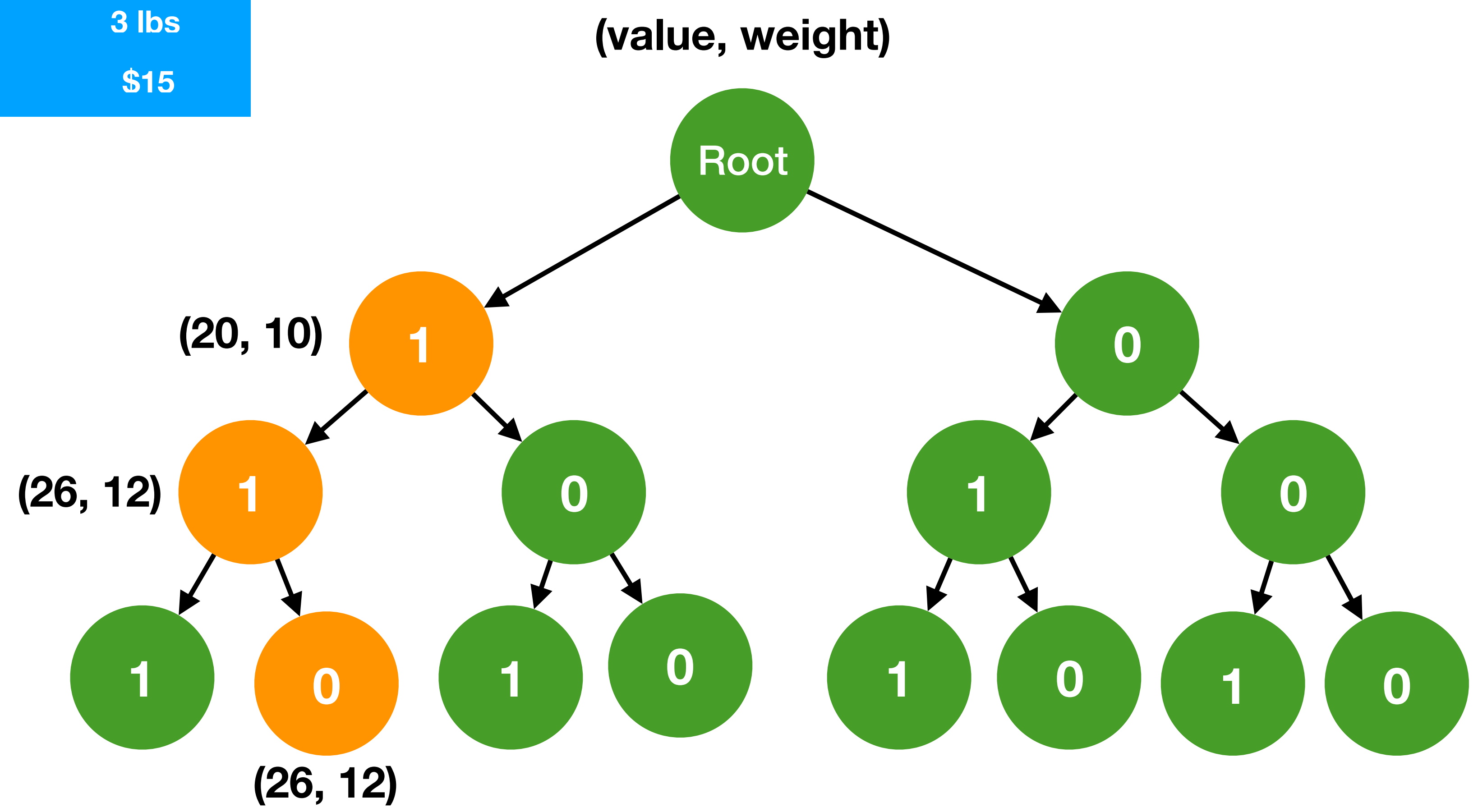




	value	weight
0 0 0	0	0
0 0 1	15	3
0 1 0	6	2
0 1 1	21	5
1 0 0	20	10
1 0 1	35	13
1 1 0		
1 1 1		

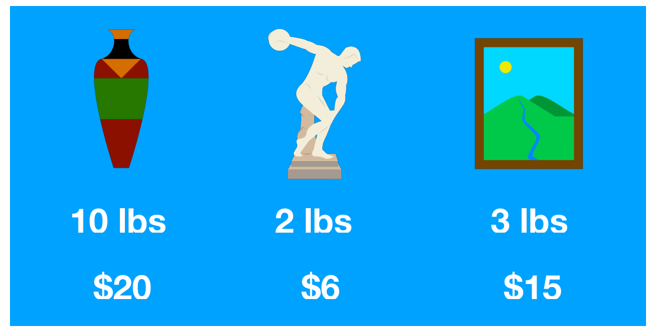
Knapsack : exhaustive search

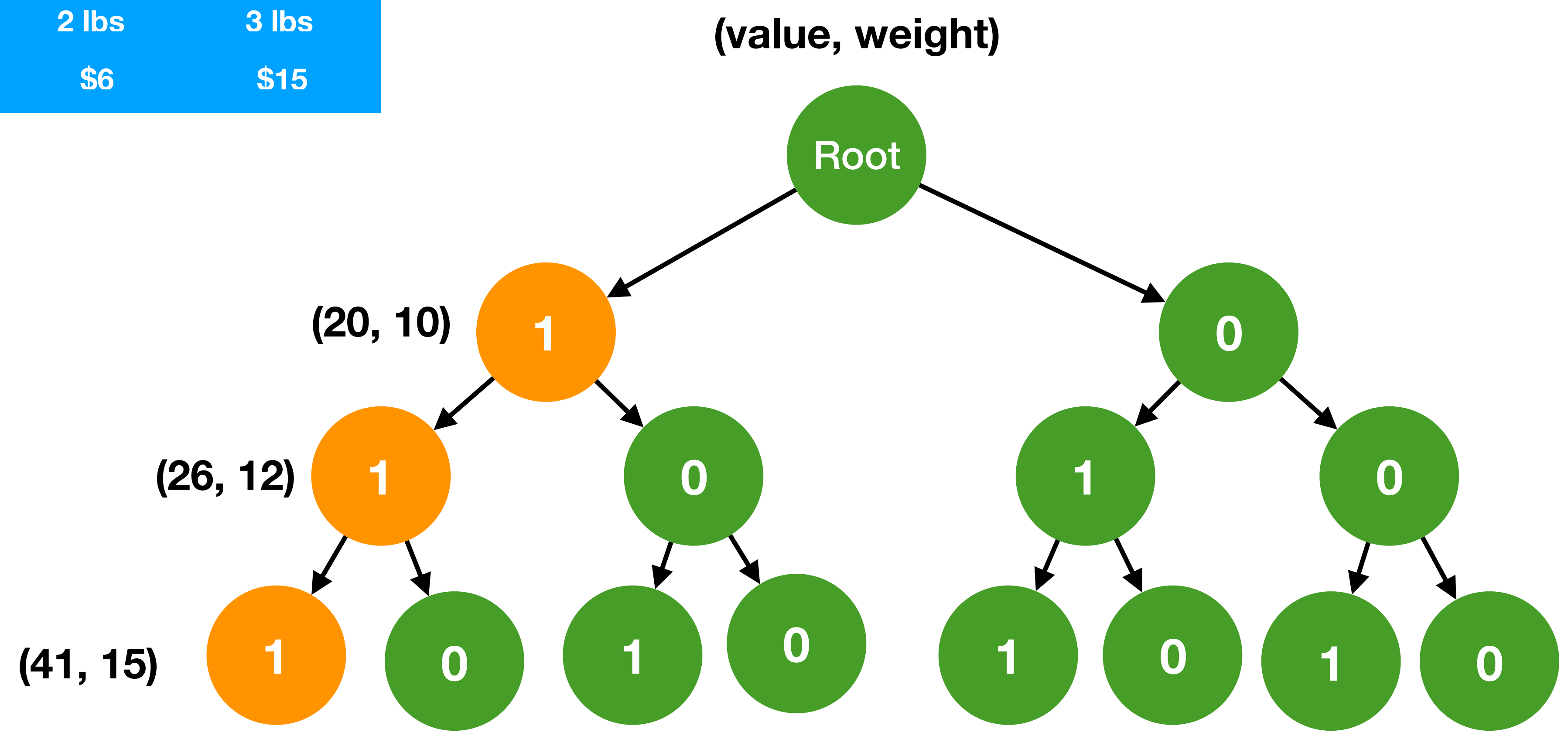




	value	weight
0 0 0	0	0
0 0 1	15	3
0 1 0	6	2
0 1 1	21	5
1 0 0	20	10
1 0 1	35	13
1 1 0	26	12
1 1 1		

Knapsack : exhaustive search





	value	weight	
0 0 0	0	0	
0 0 1	15	3	
0 1 0	6	2	
0 1 1	21	5	
1 0 0	20	10	
1 0 1	35	13	not feasible
1 1 0	26	12	not feasible
1 1 1	41	15	not feasible

Generate n-digit binary numbers

```
#include <stdio.h>
#include <string.h>

// generates 3 digit binary numbers
#define NUM_DIGITS 10

typedef struct {
   char str[NUM_DIGITS+1];
}Seq;

void enumerate(int index, Seq seq) {
   //printf("index: %d   seq: %s \n", index, seq.str);
    // reached a leaf node, print out the binary sequence and return
    if (index == NUM_DIGITS-1) {
      printf("%s \n", seq.str);
      return;
    }

    index++;

   // create seq1 (with an added 1) and seq0 (with an added 0) to store the new binary sequence.
   Seq seq1, seq0;
   strcpy(seq1.str, seq.str);
   strcpy(seq0.str, seq.str);
   strcat(seq1.str, "1");
   strcat(seq0.str, "0");

   // continue the recursion
   enumerate(index, seq1);
   enumerate(index, seq0);
}
```

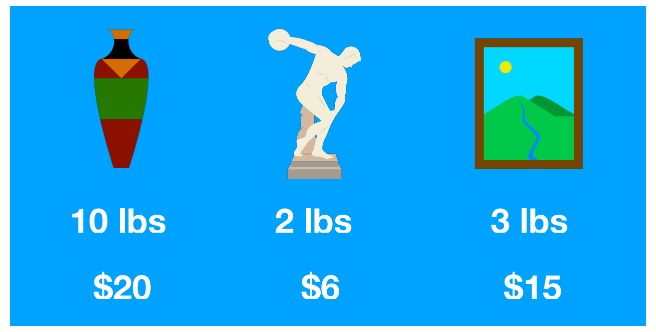

Generate n-digit binary numbers

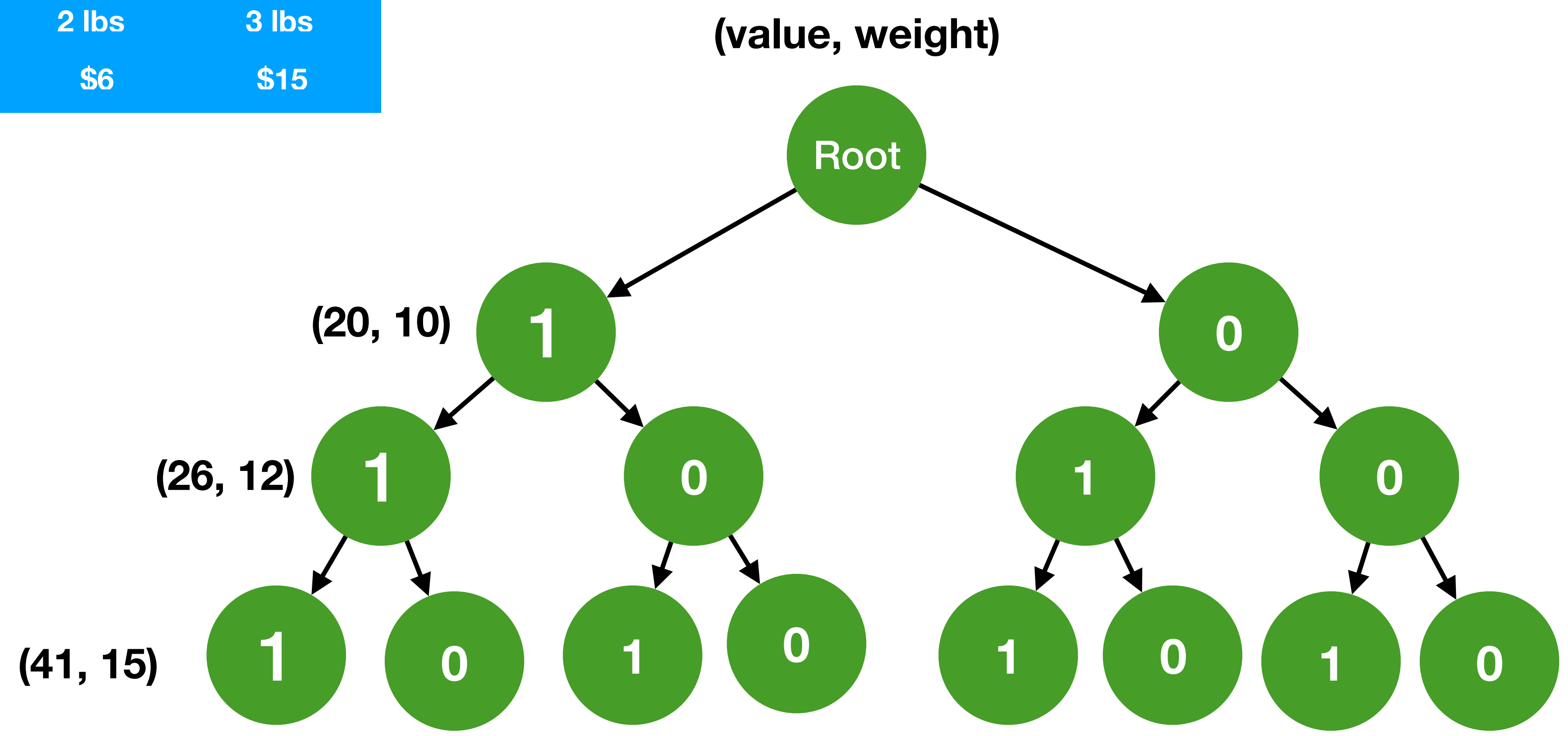
```c
int main(void) {
   Seq seq ;
   strcpy(seq.str,"\0");
   enumerate(-1, seq);
   return 0;
}
```

Program output:

111
110
101
100
011
010
001
000

Knapsack : exhaustive search





	value	weight
0 0 0	0	0
0 0 1	15	3
0 1 0	6	2
0 1 1	**21**	**5**
1 0 0	20	10
1 0 1	35	13
1 1 0	26	12
1 1 1	41	15

Solution = 21

Improve this algorithm using backtracking

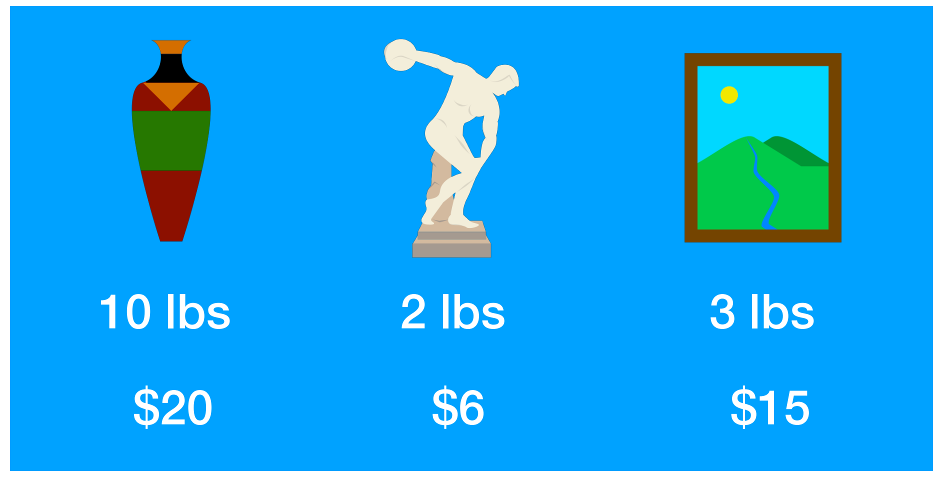
Knapsack: exhaustive search

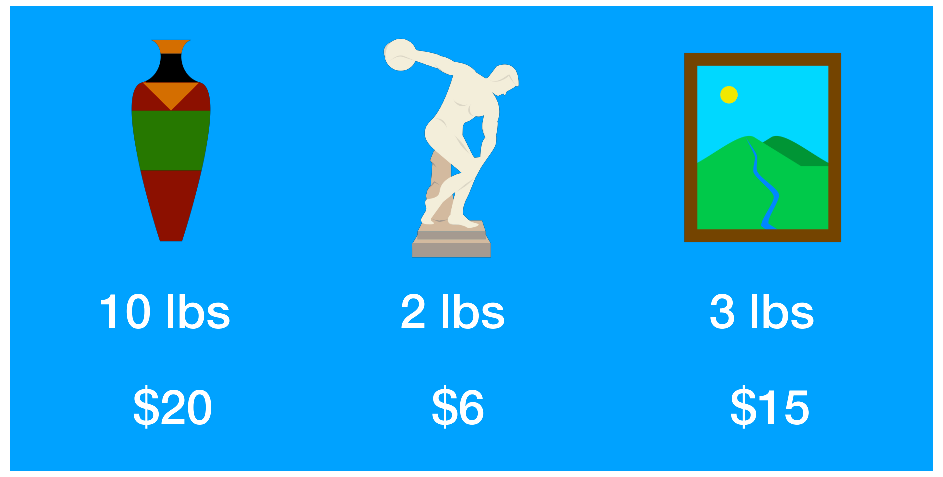
```c
#include <stdio.h>
#include <string.h>

#define SIZE 3

int weight[] = {10, 2, 3};
int value[] = {20, 6, 15};

int maxAllowedWeight = 11;
int maxValue = 0;
```

```c
typedef struct {
    char str[SIZE+1];
}Seq;
```

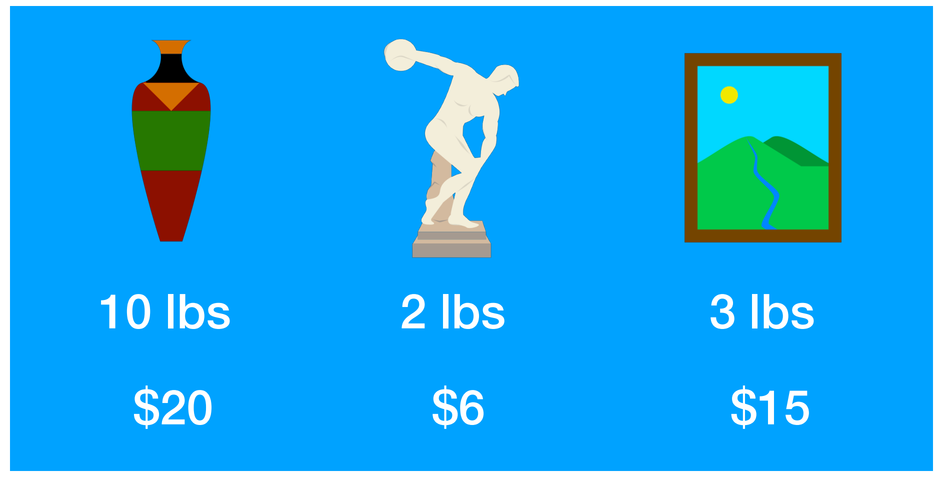
```c
int main(void) {
    Seq seq;
    strcpy(seq.str,"\0");
    knapsack(-1, 0, 0, seq);
    printf(" Max value : %d ", maxValue);
    return 0;
}
```

```c
void knapsack(int index, int currentValue, int currentWeight, Seq seq) {

    printf(" Knapsack :  %s  current value: %d   current weight: %d\n" , seq.str,  currentValue,  currentWeight);

     if (currentWeight > maxAllowedWeight)// weight exceeds maximum weight, backtrack
       return;

     if (currentValue > maxValue)    //record max value found so far
       maxValue = currentValue;

     if (index == SIZE-1)
             return;

   index++; // next item in bag

   Seq seq1, seq0;
   strcpy(seq1.str, seq.str);
   strcpy(seq0.str, seq.str);
   strcat(seq1.str, "1");
   strcat(seq0.str, "0");

   knapsack(index, value[index]+currentValue, weight[index]+currentWeight, seq1);
   knapsack(index, currentValue, currentWeight, seq0);
}
```


10 lbs
$20
2 lbs
$6
3 lbs
$15



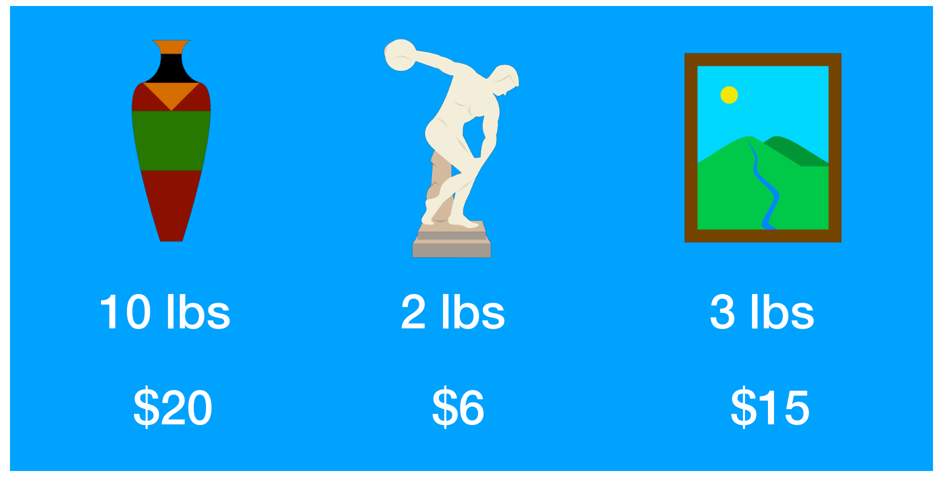
value = KS([$20, $6, $15], 10 lbs)



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)



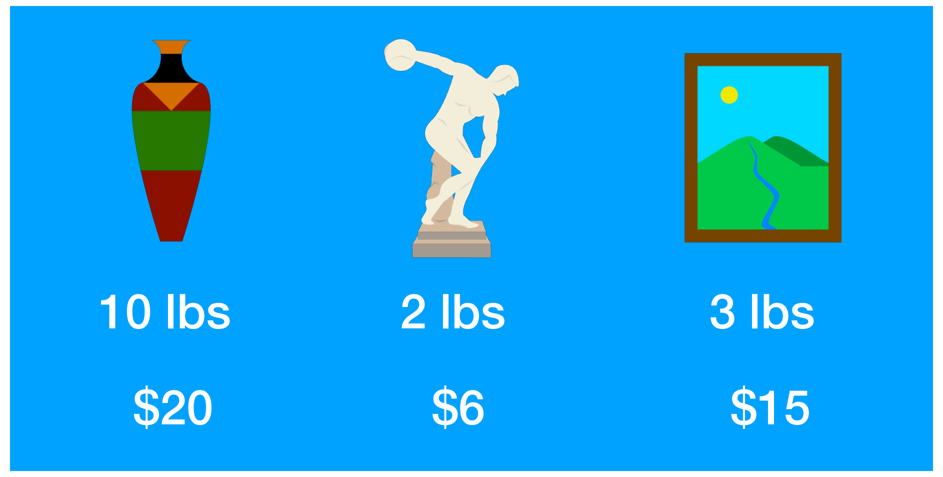
value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

$6 taken

value = $6 + KS([$20], 7 lbs - 2 lbs)



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

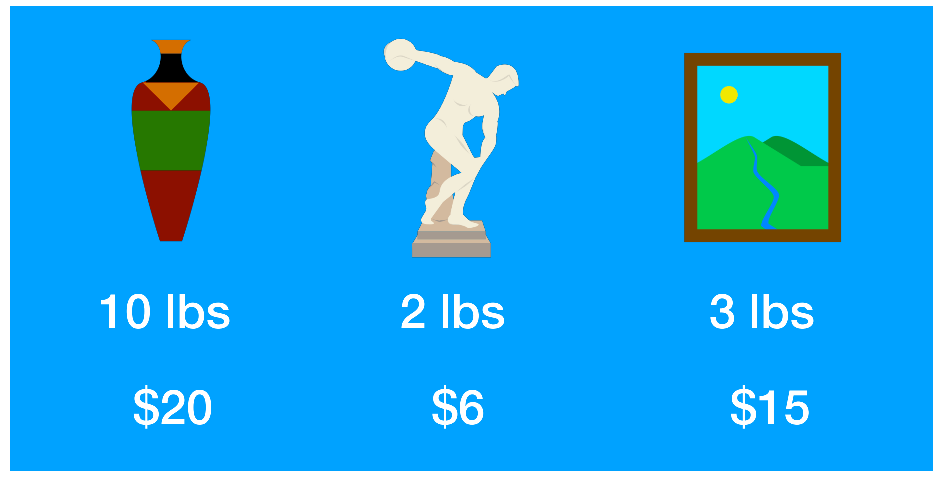
$6 taken

value = $6 + KS([$20], 7 lbs - 2 lbs)

Examine $20

Weight of 10 lbs
greater than capacity.

value = KS([ ] , 5 lbs)



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

$6 taken

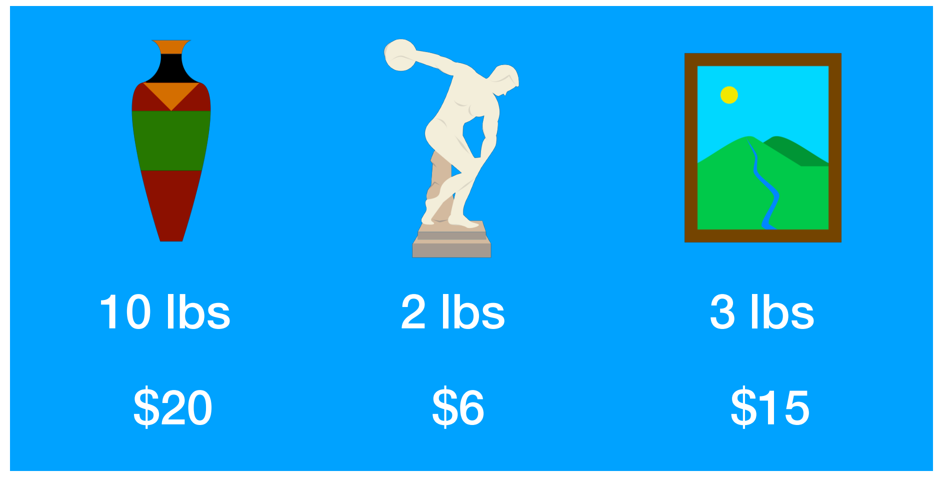
value = $6 + KS([$20], 7 lbs - 2 lbs)

Examine $20

Weight of 10 lbs greater than capacity.

value = KS([ ] , 5 lbs)

Empty list, return 0.



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

$6 taken

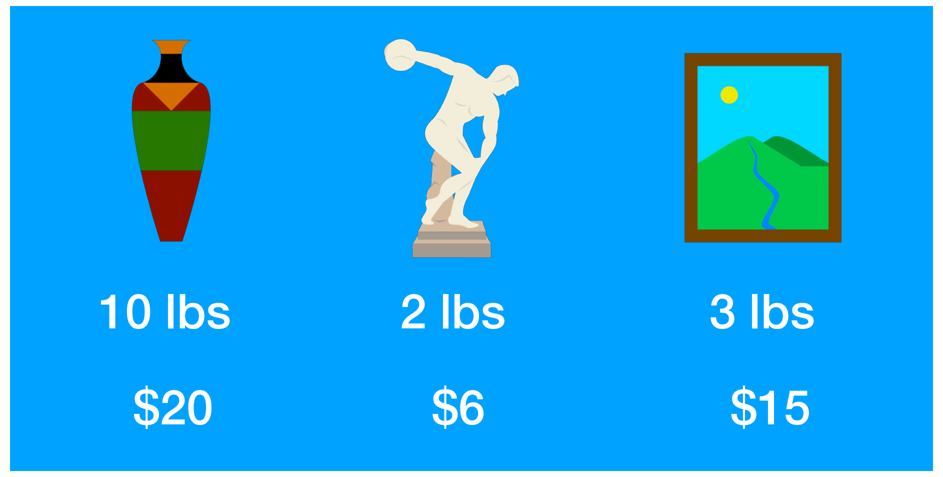
value = $6 + KS([$20], 7 lbs - 2 lbs)

Examine $20

Weight of 10 lbs
greater than capacity.

value = KS([ ] , 5 lbs)

Empty list, return 0.



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

$6 taken

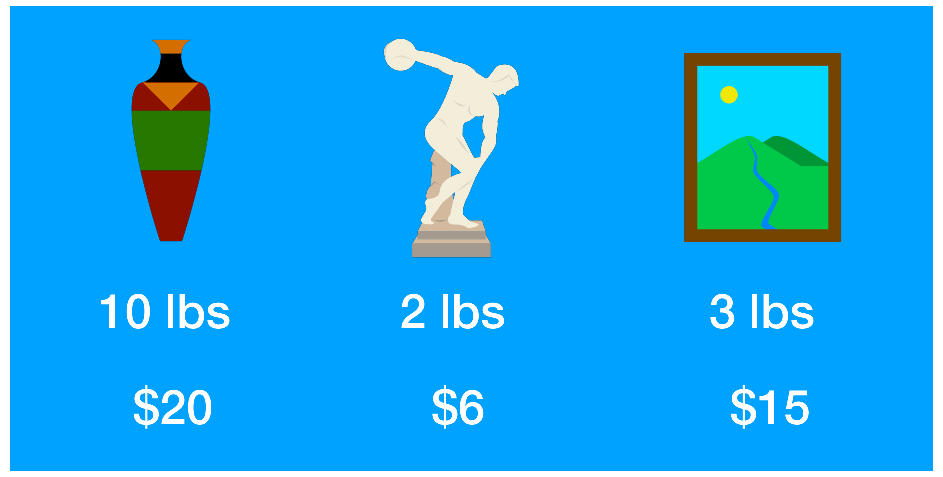
value = $6 + KS([$20], 7 lbs - 2 lbs)

Examine $20

Weight of 10 lbs greater than capacity.

value = KS([ ] , 5 lbs)

Empty list, return 0.



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

$6 taken

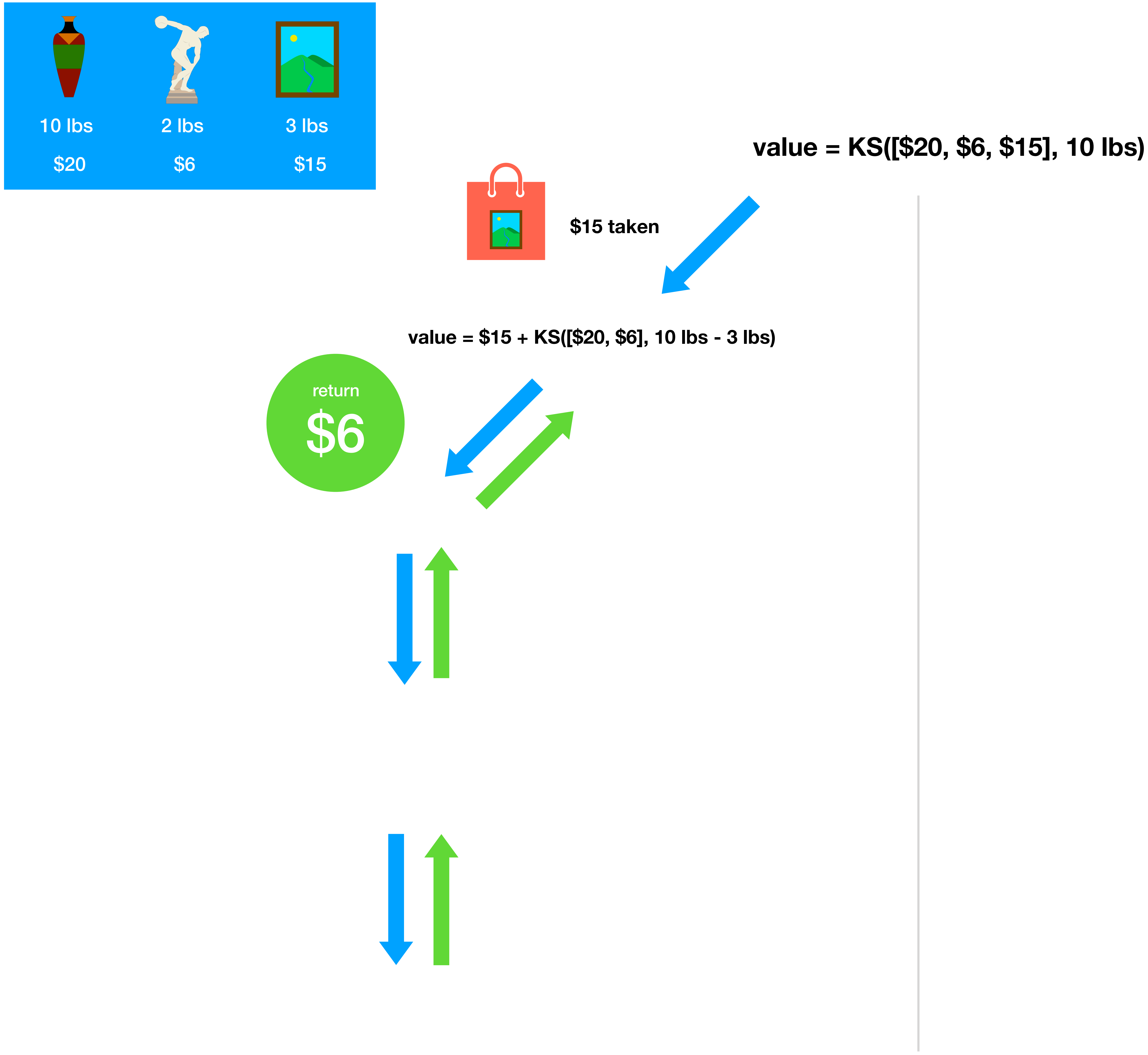
value = $6 + KS([$20], 7 lbs - 2 lbs)

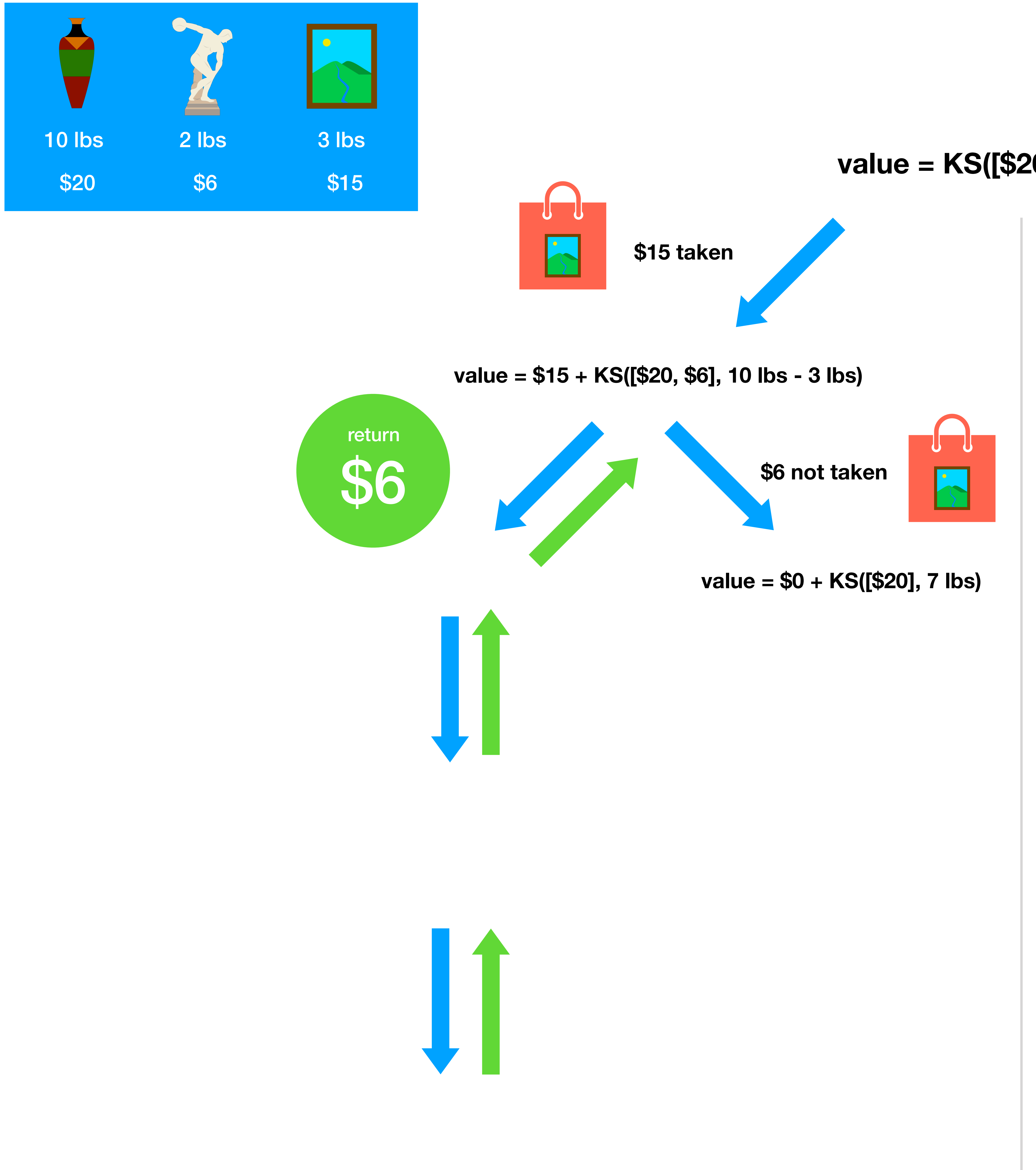
Examine $20

Weight of 10 lbs
greater than capacity.

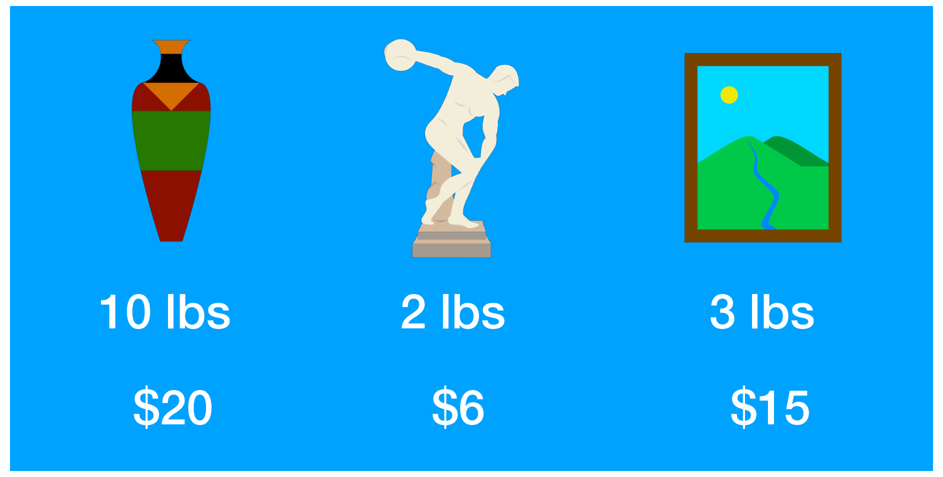
value = KS([ ] , 5 lbs)

Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6



10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 7 lbs)



value = KS([$20, $6, $15], 10 lbs)

$15 taken

value = $15 + KS([$20, $6], 10 lbs - 3 lbs)

return
$6

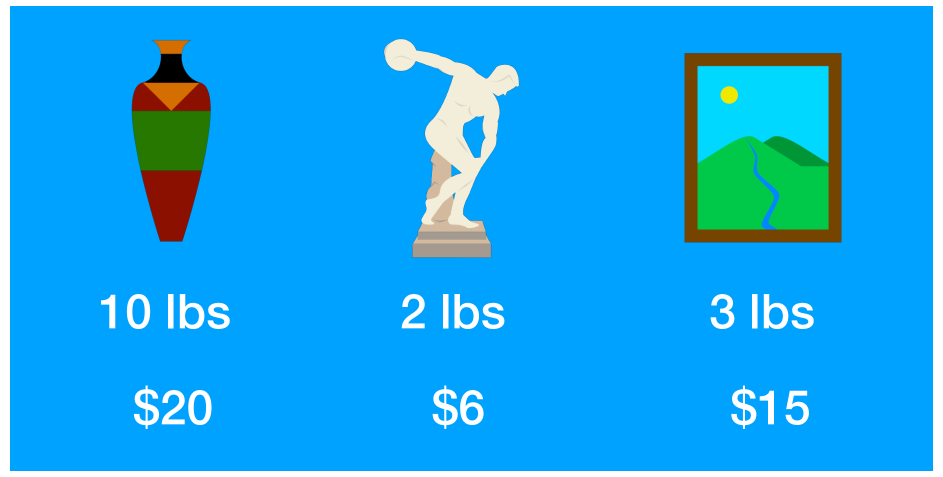
$6 not taken

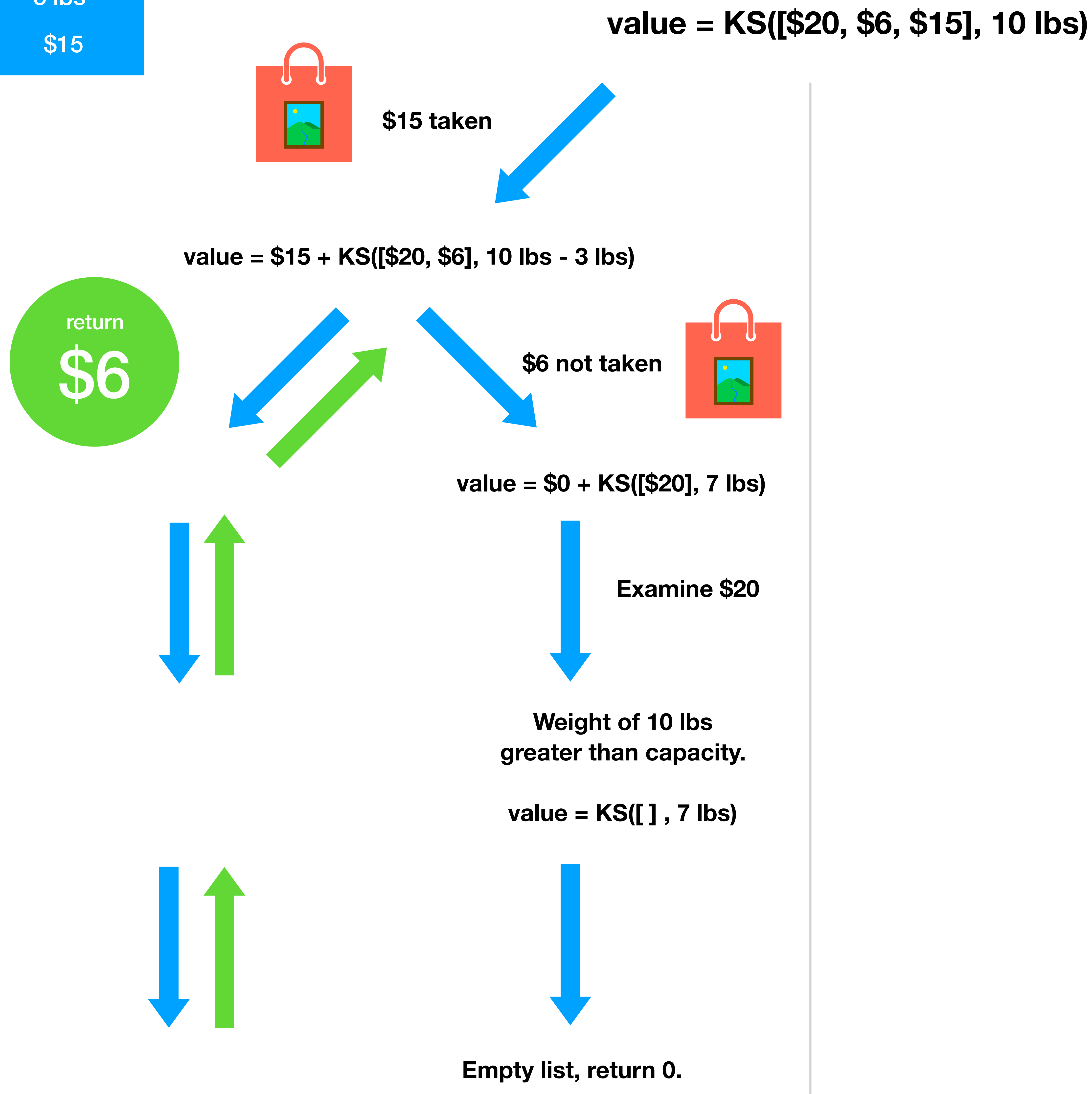
value = $0 + KS([$20], 7 lbs)

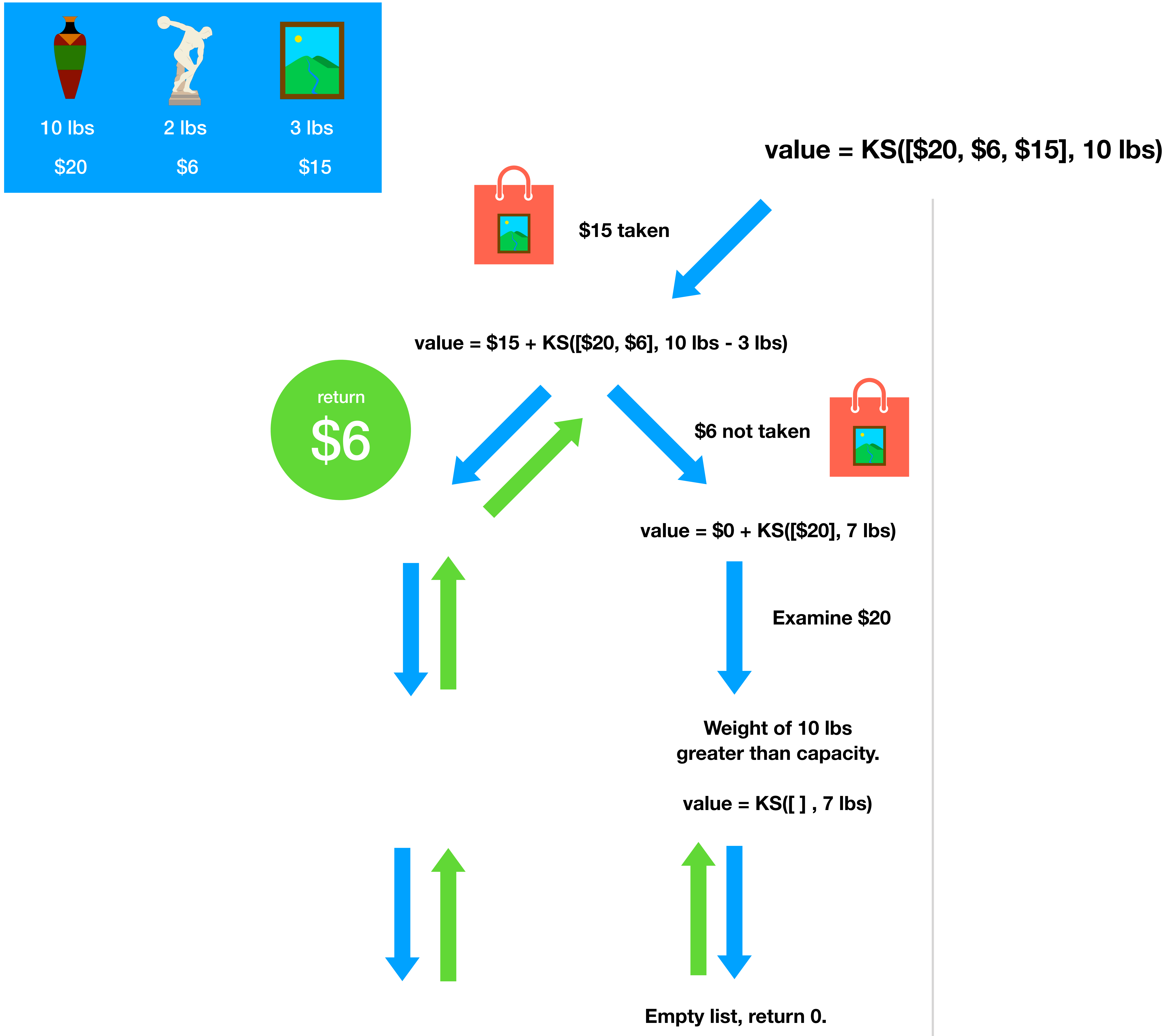
Examine $20

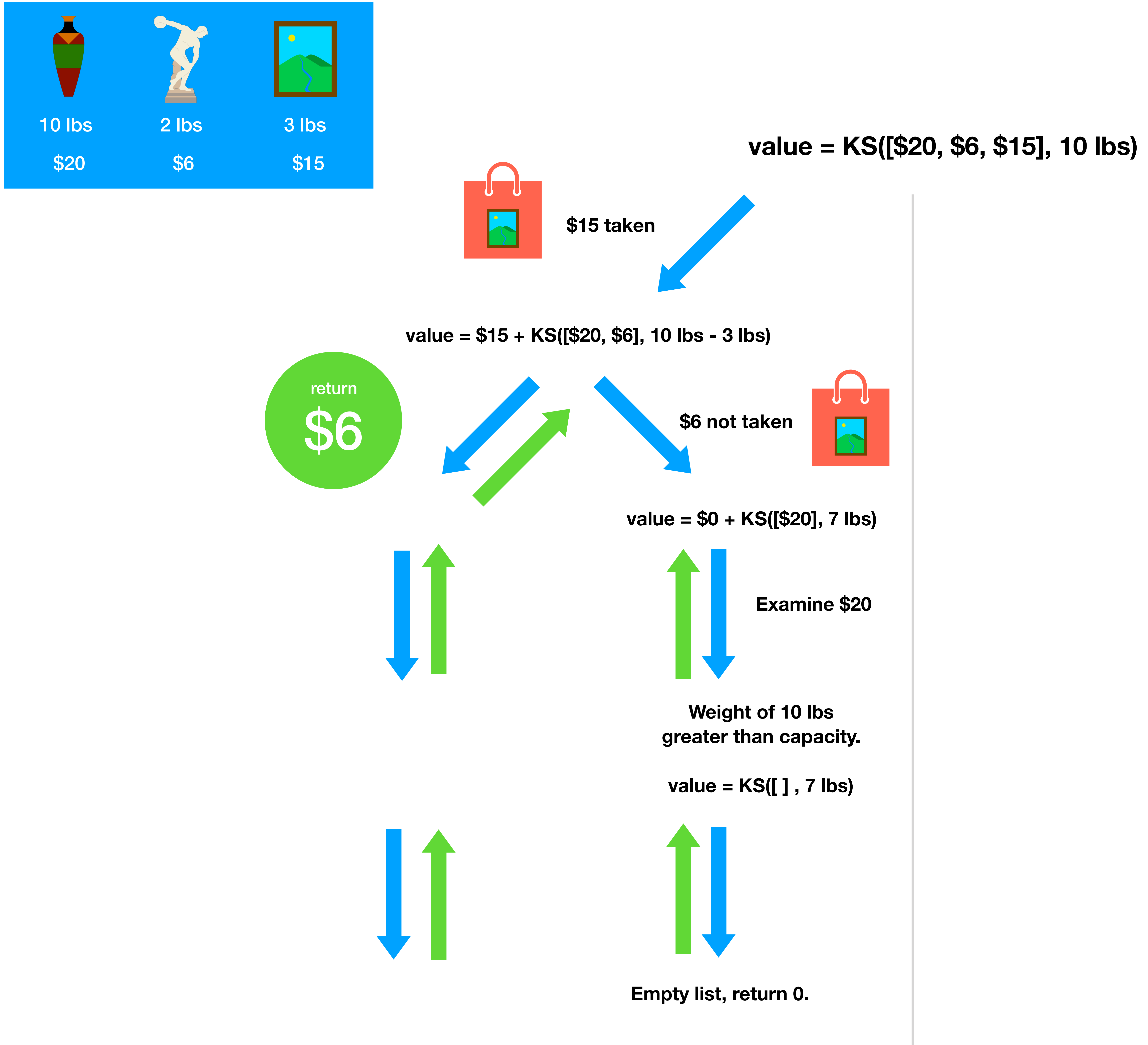
Weight of 10 lbs greater than capacity.

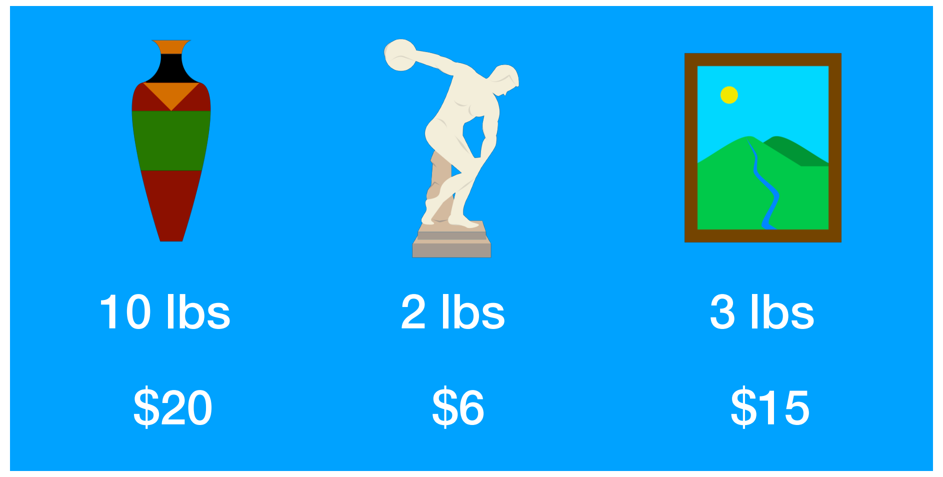
value = KS([ ] , 7 lbs)

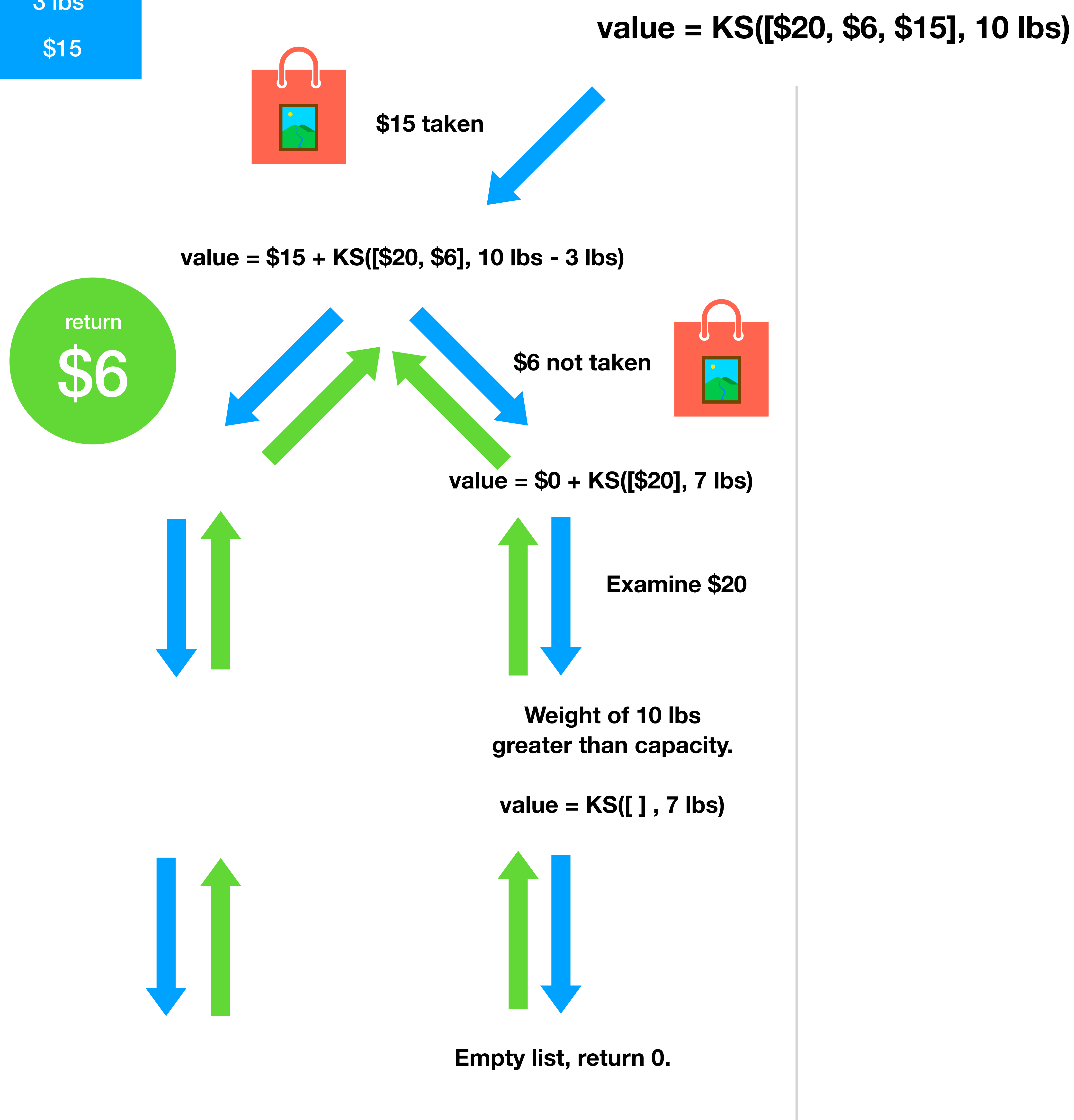

10 lbs
$20
2 lbs
$6
3 lbs
$15

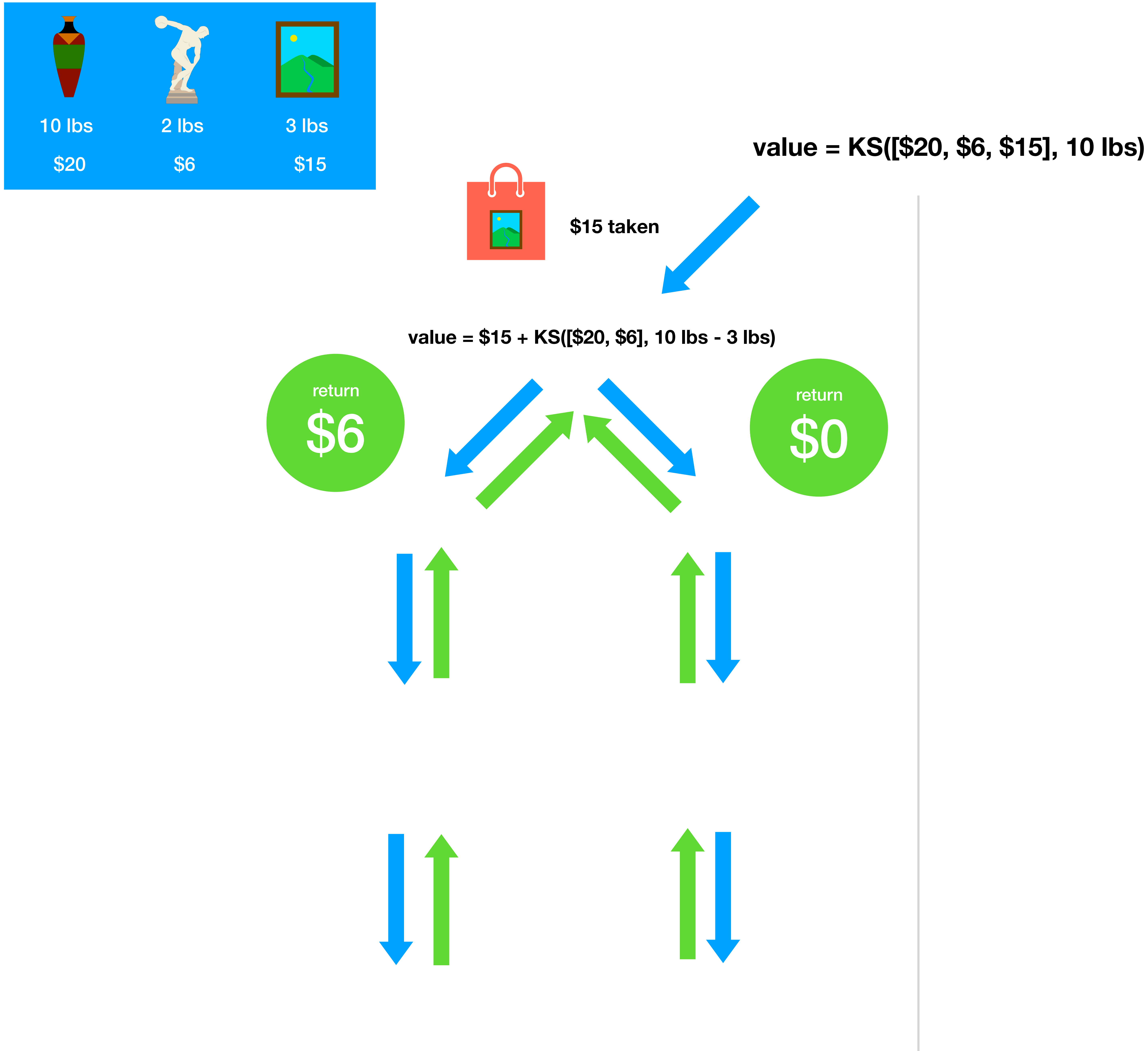

value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 7 lbs)
Examine $20
Weight of 10 lbs
greater than capacity.
value = KS([ ] , 7 lbs)
Empty list, return 0.

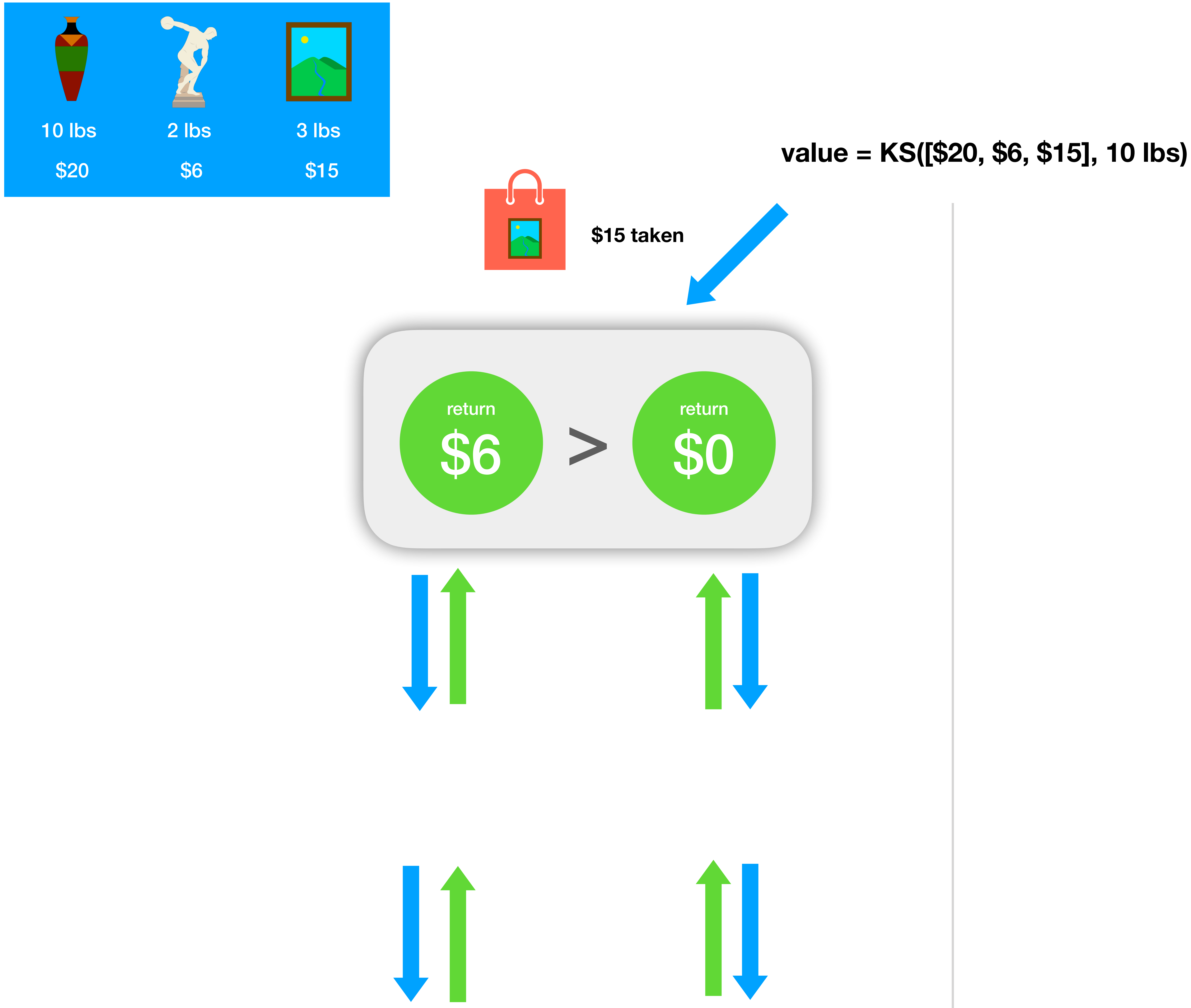

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 7 lbs)
Examine $20
Weight of 10 lbs
greater than capacity.
value = KS([ ] , 7 lbs)
Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 7 lbs)
Examine $20
Weight of 10 lbs
greater than capacity.
value = KS([ ] , 7 lbs)
Empty list, return 0.

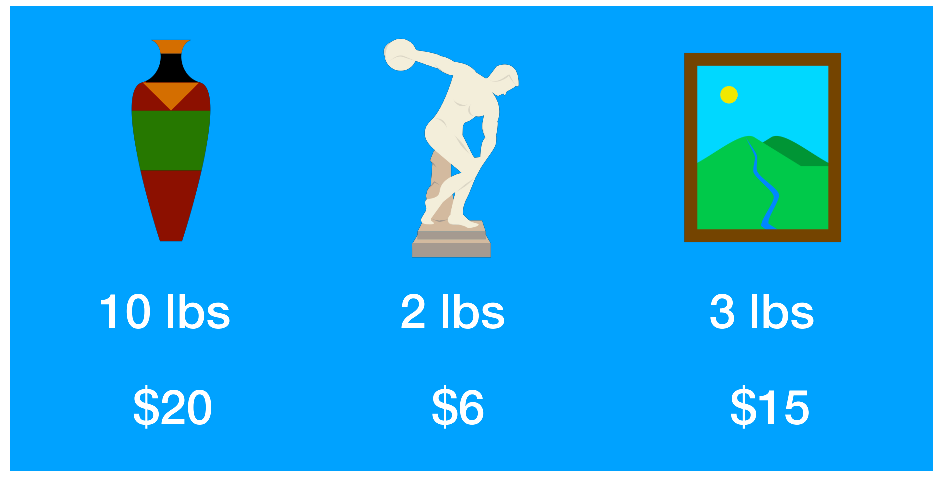

10 lbs
$20
2 lbs
$6
3 lbs
$15


value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 7 lbs)
Examine $20
Weight of 10 lbs
greater than capacity.
value = KS([ ] , 7 lbs)
Empty list, return 0.

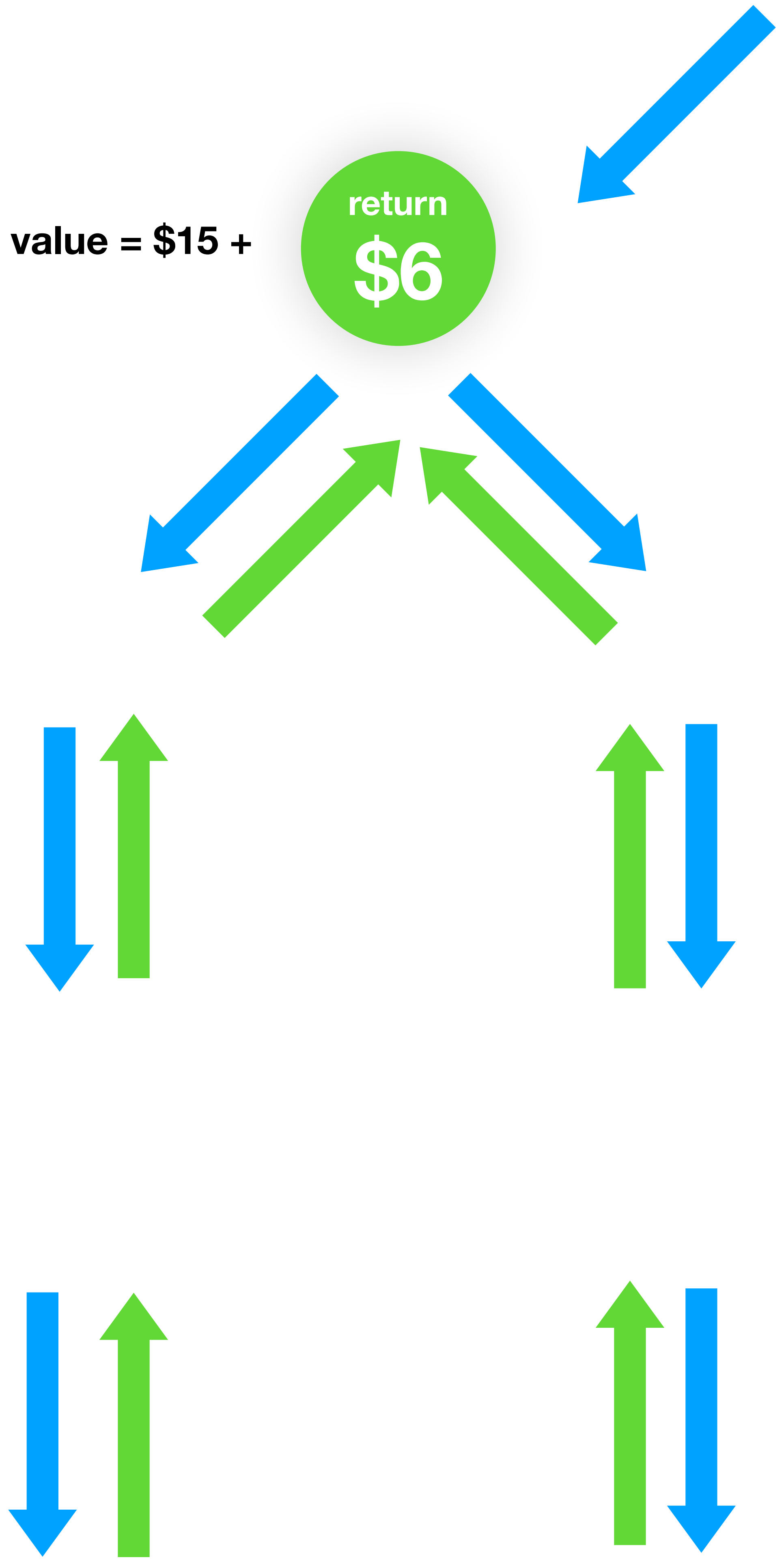

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
value = $15 + KS([$20, $6], 10 lbs - 3 lbs)
return
$6
return
$0

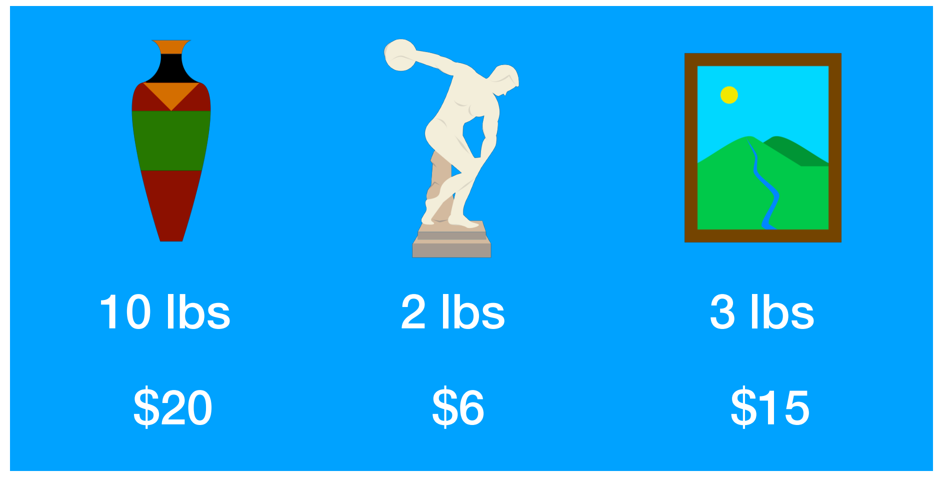


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 taken
return
$6
>
return
$0

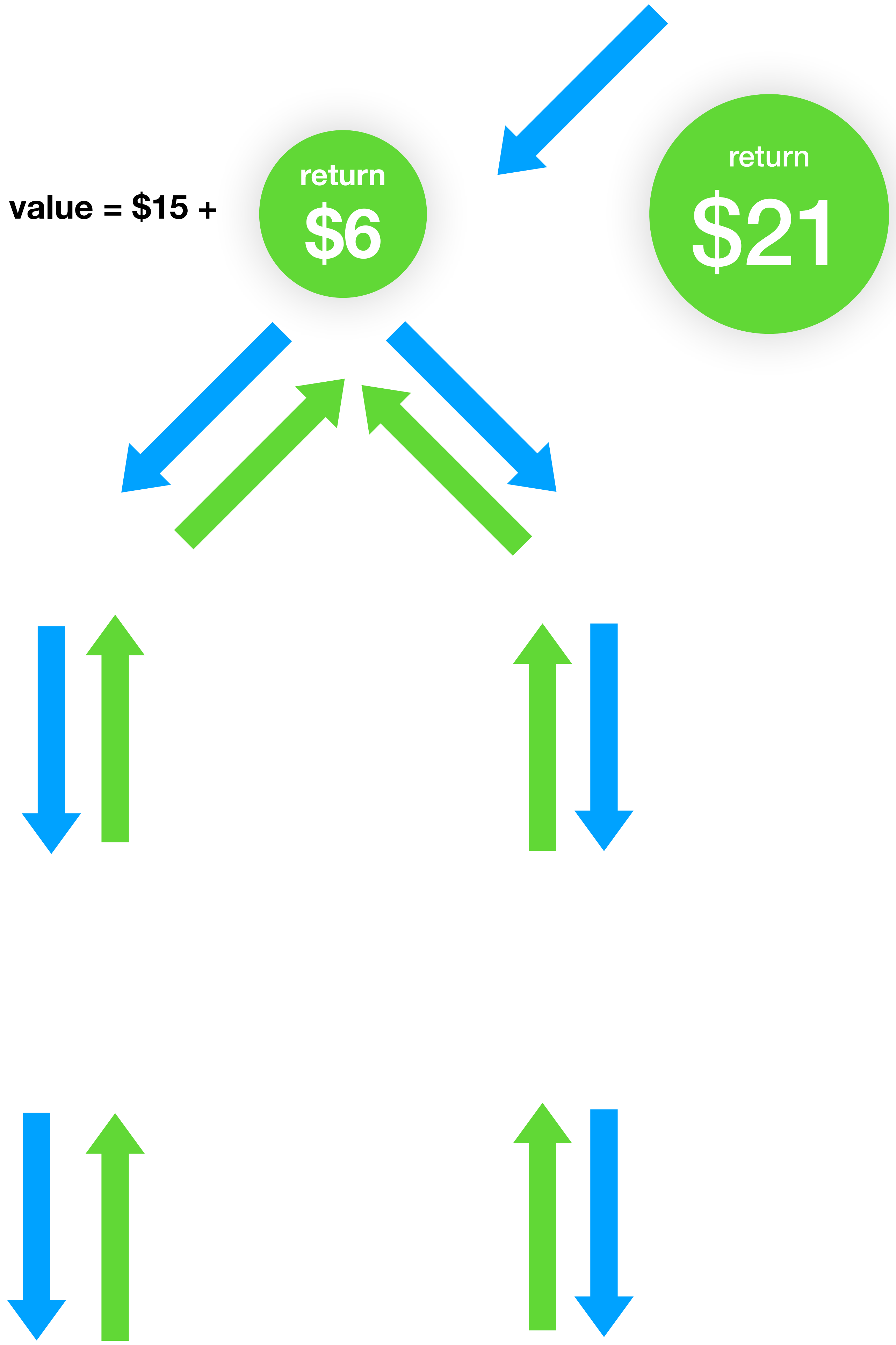


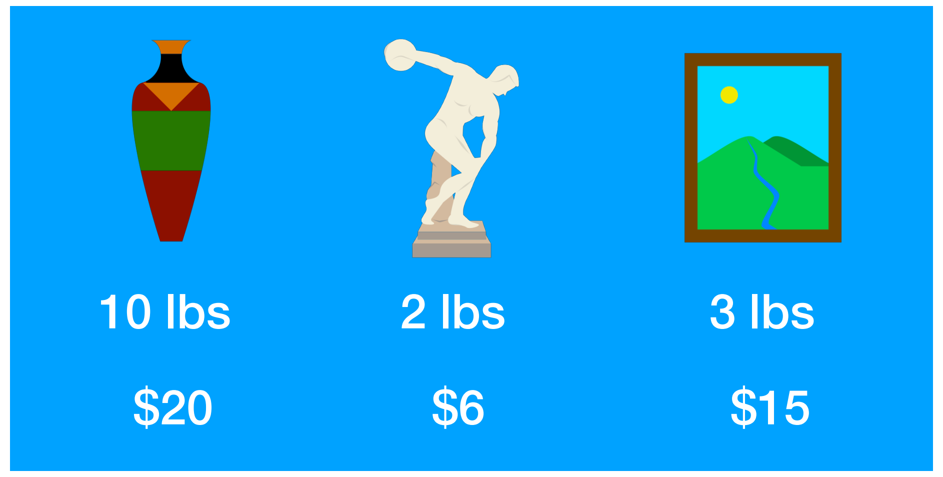
value = KS([$20, $6, $15], 10 lbs)

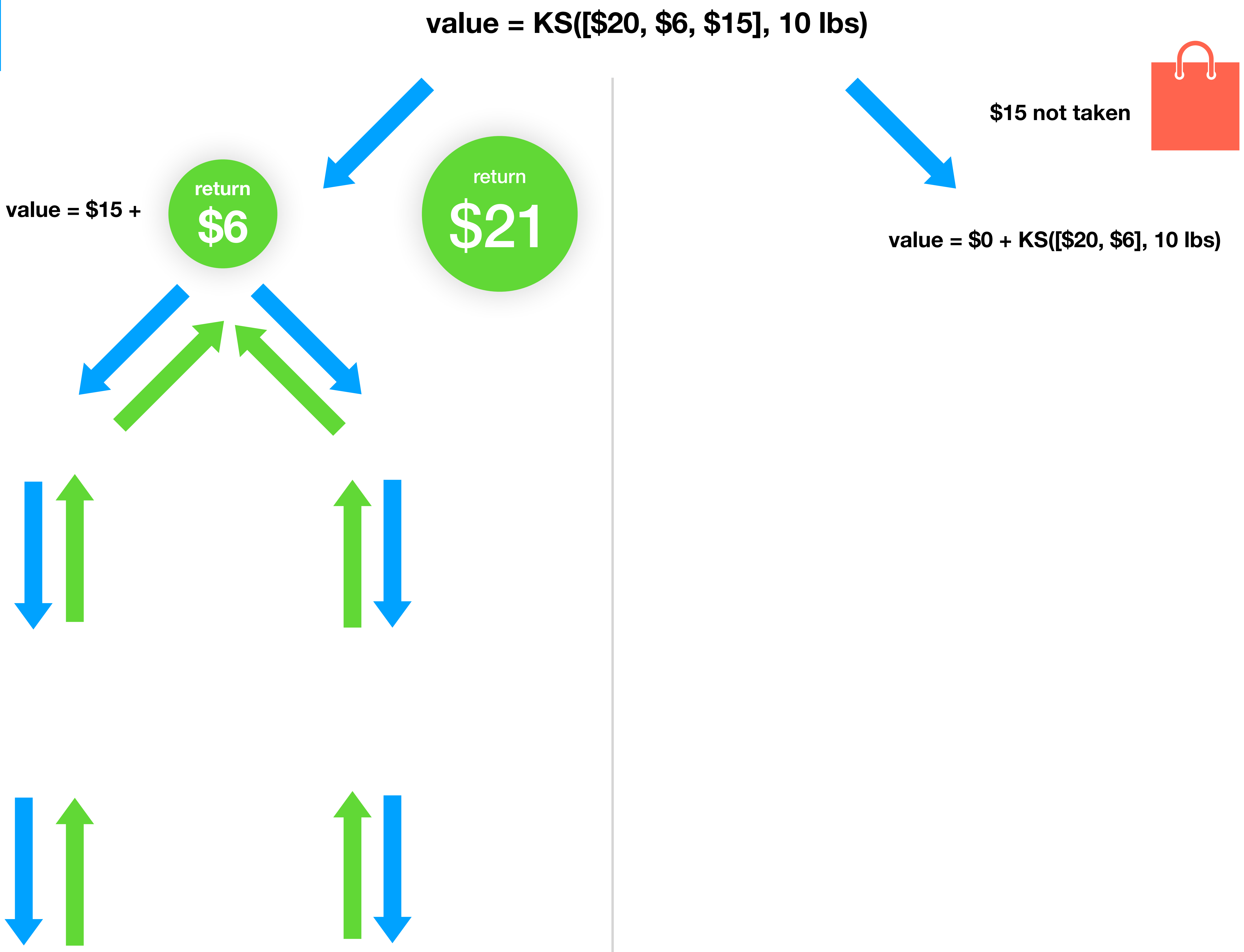


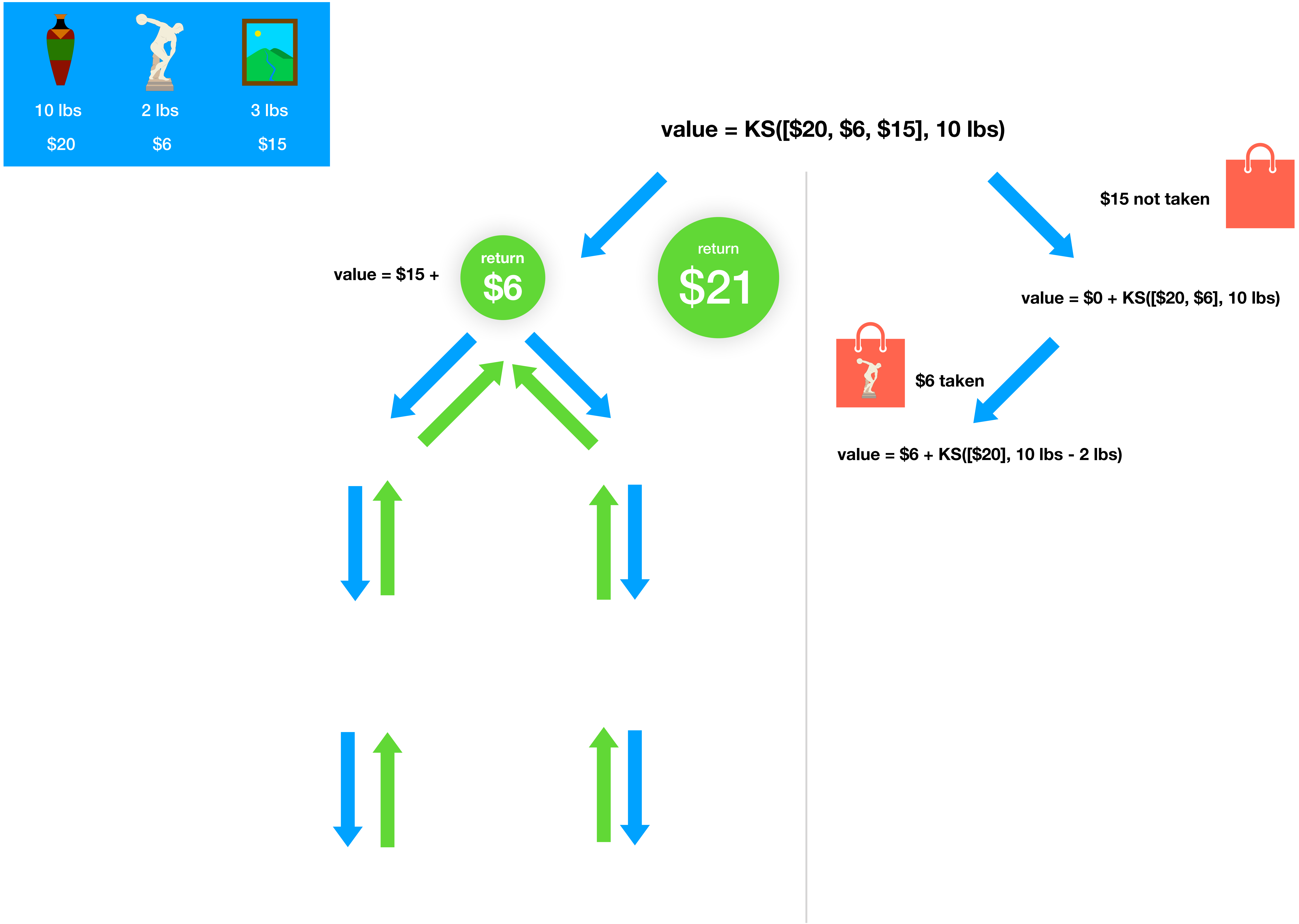


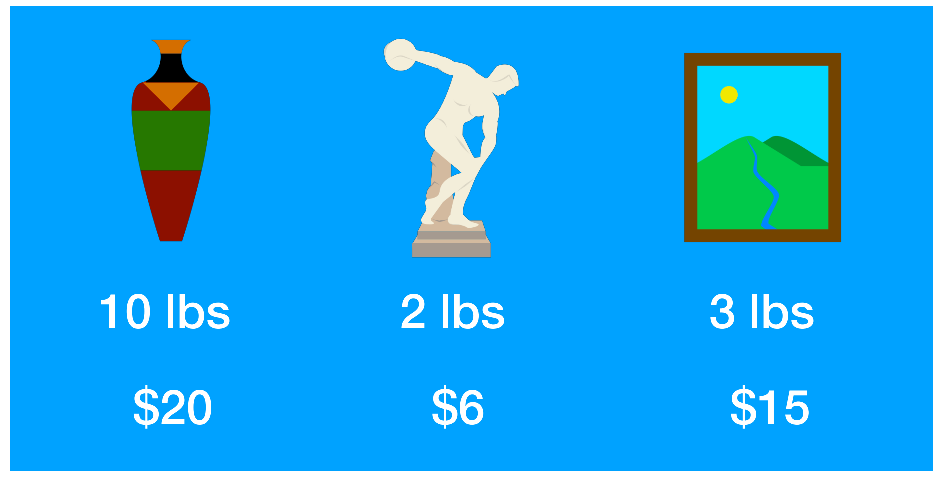
value = KS([$20, $6, $15], 10 lbs)

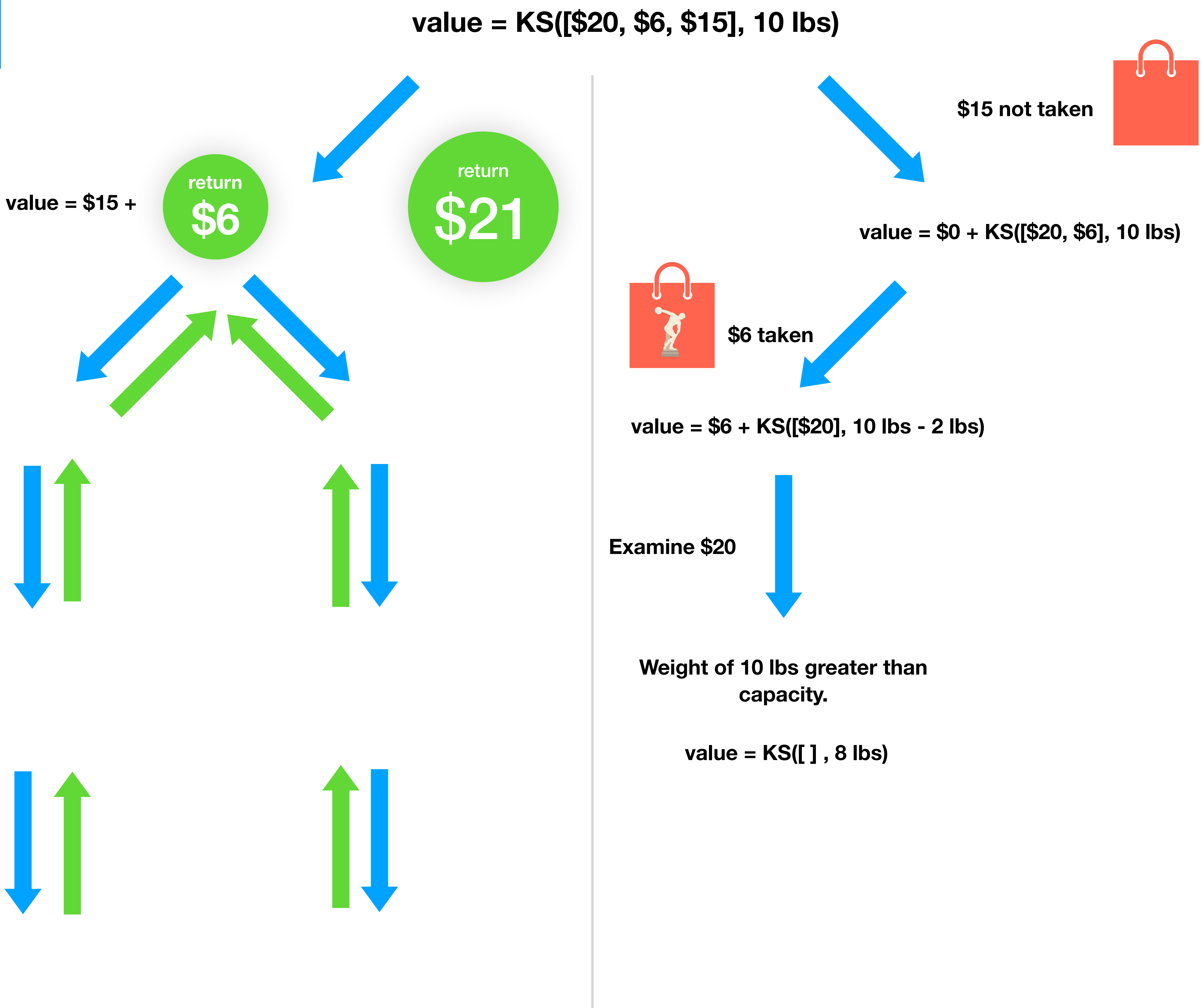


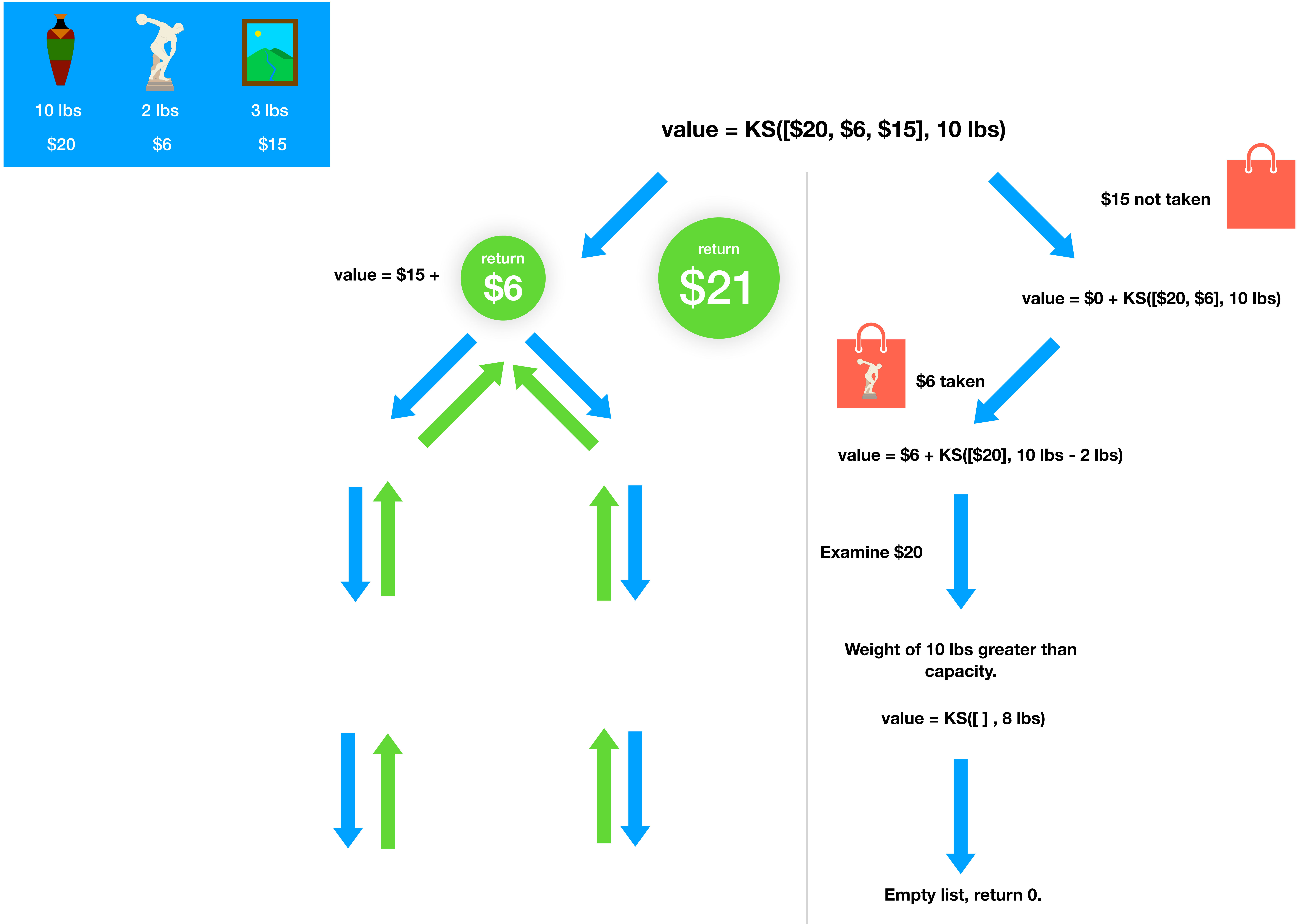

10 lbs
$20
2 lbs
$6
3 lbs
$15

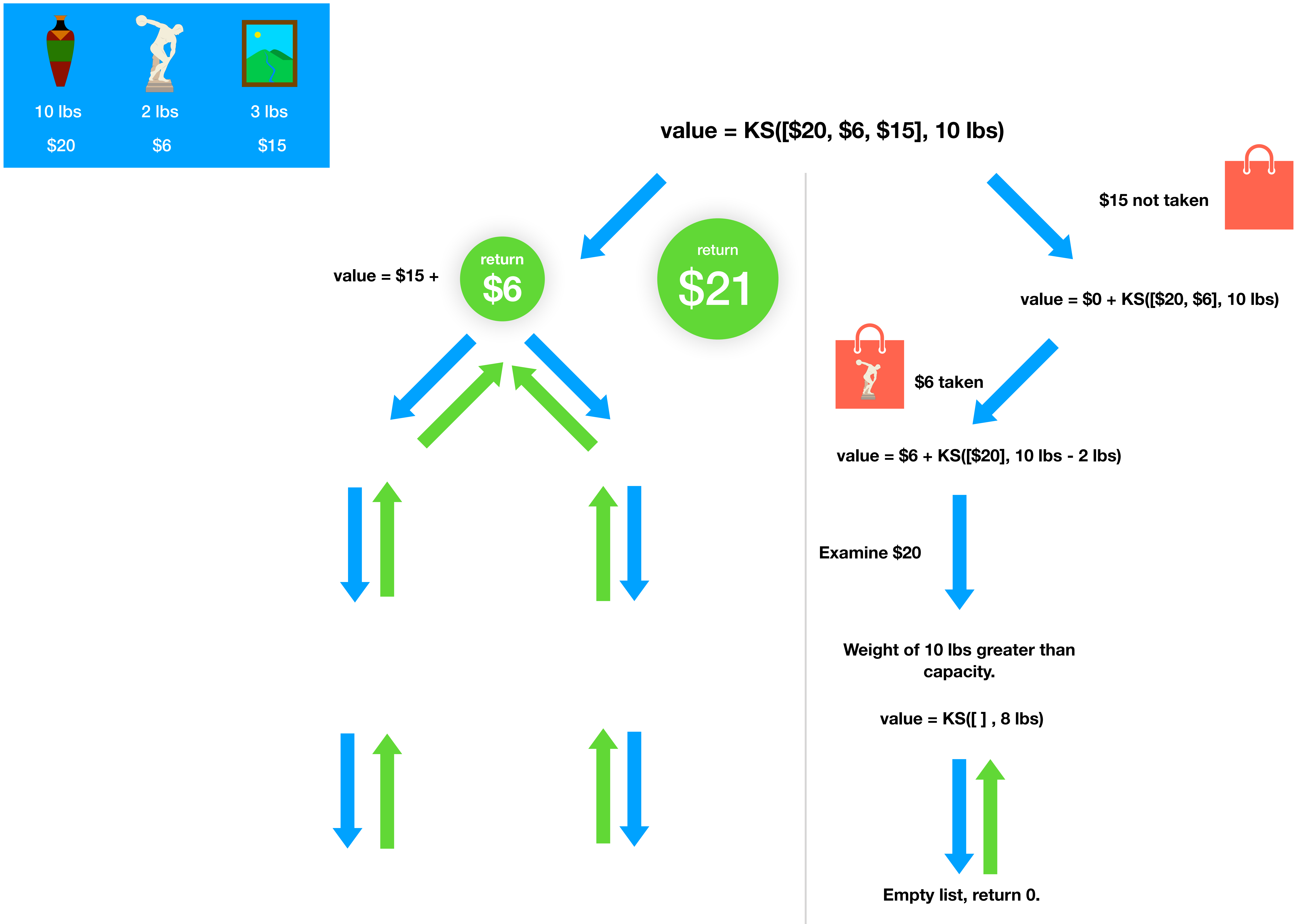

value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)

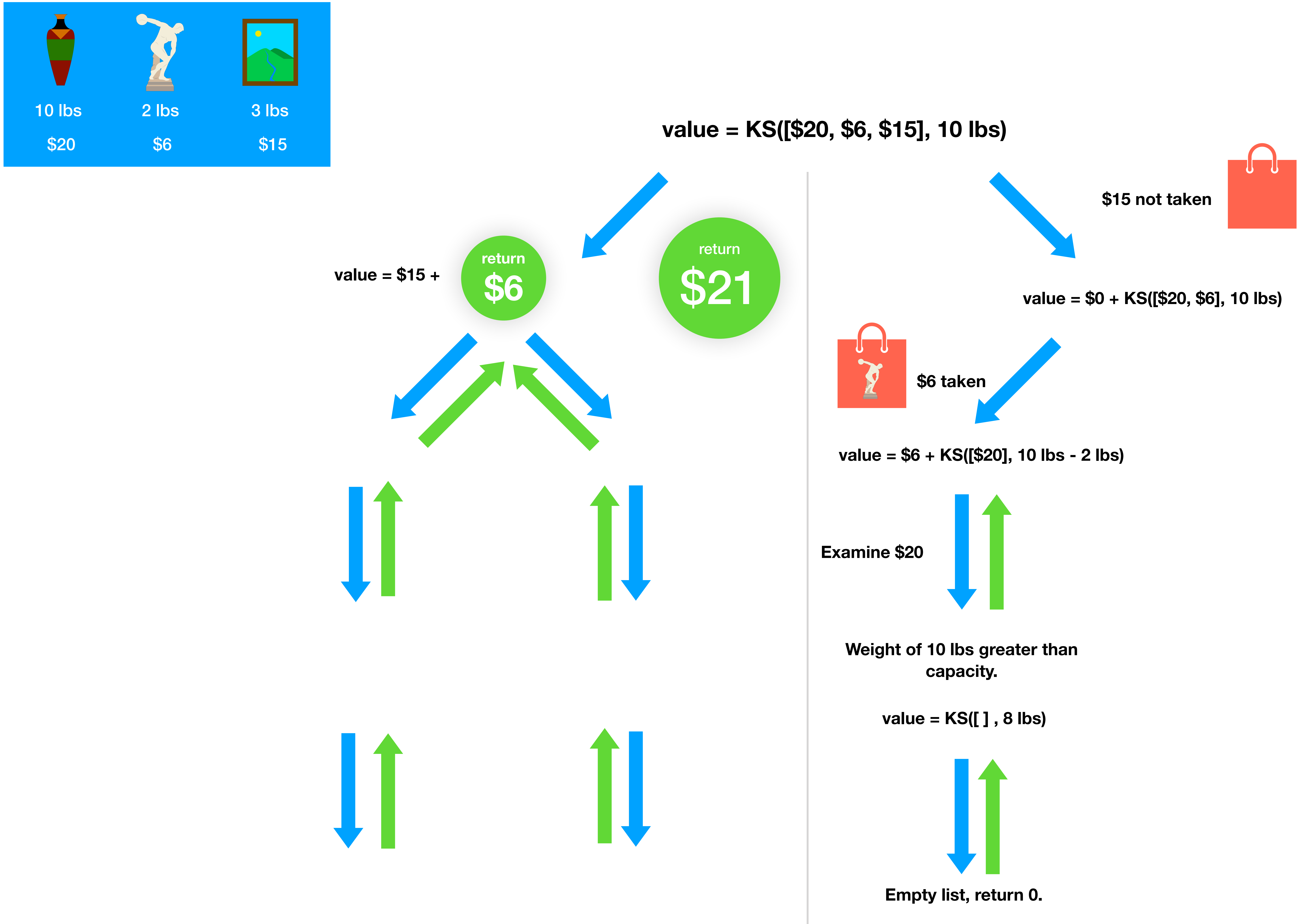

10 lbs
2 lbs
3 lbs
$20
$6
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)

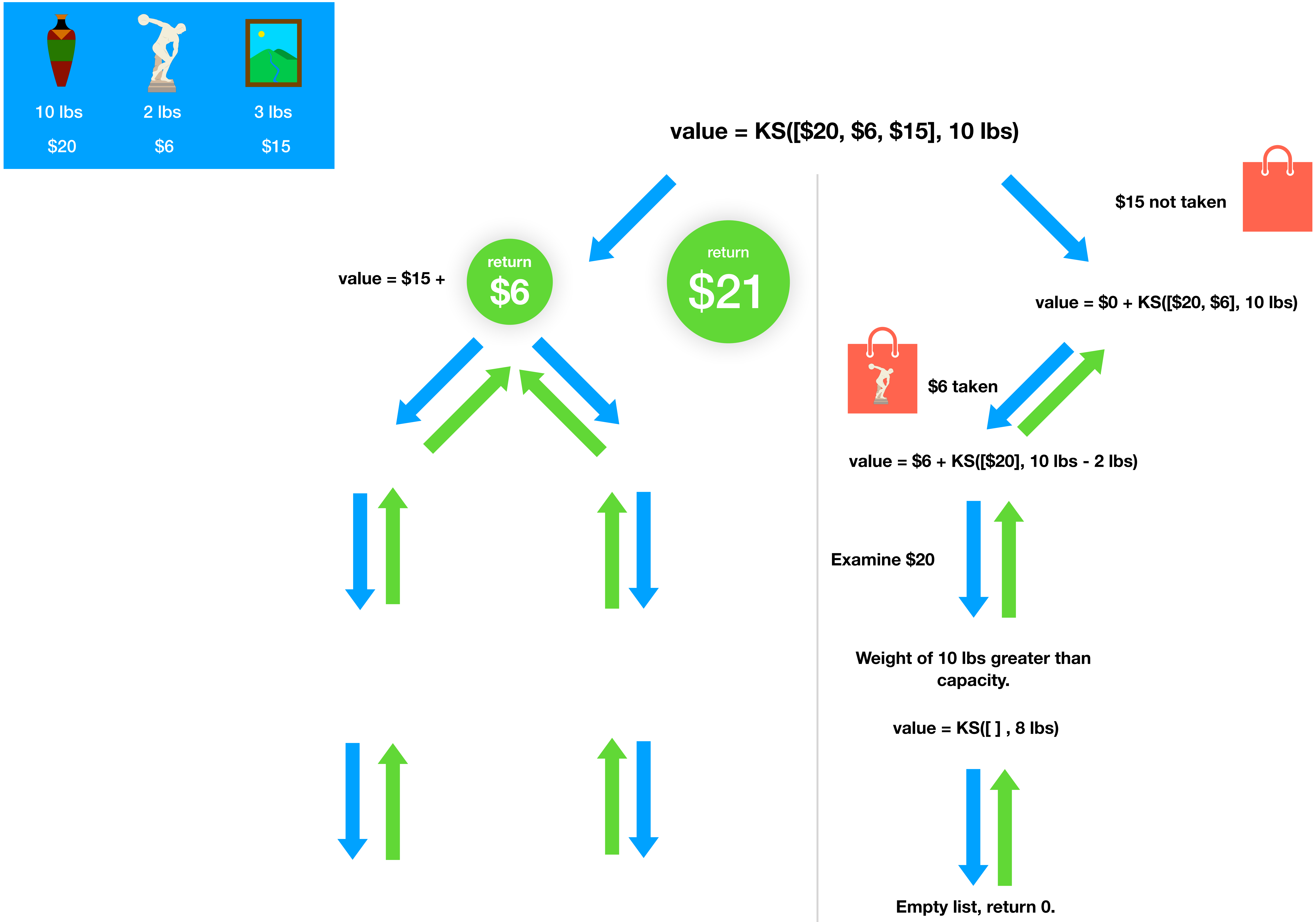

10 lbs
$20
2 lbs
$6
3 lbs
$15

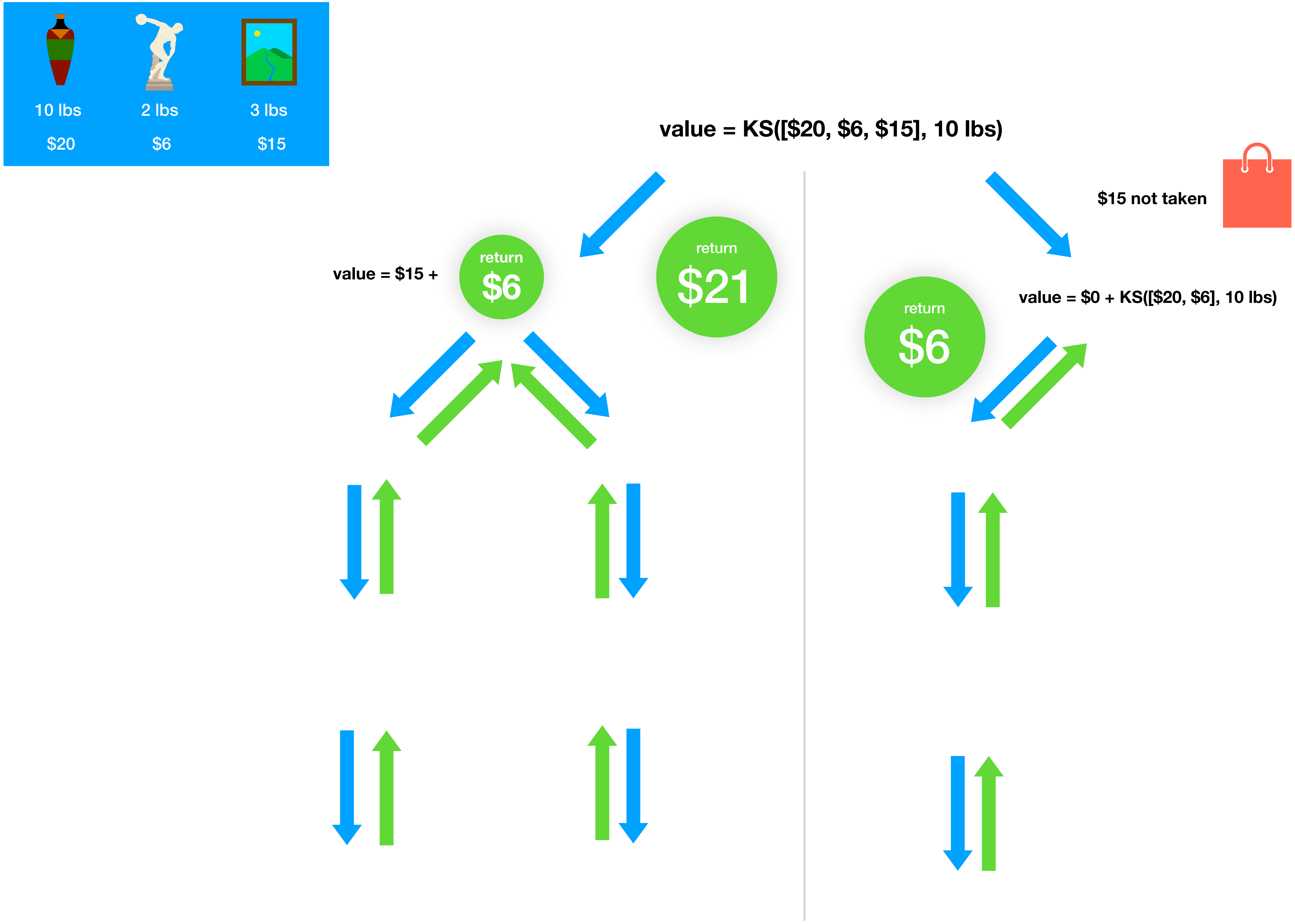

value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)
Examine $20
Weight of 10 lbs greater than
capacity.
value = KS([ ] , 8 lbs)

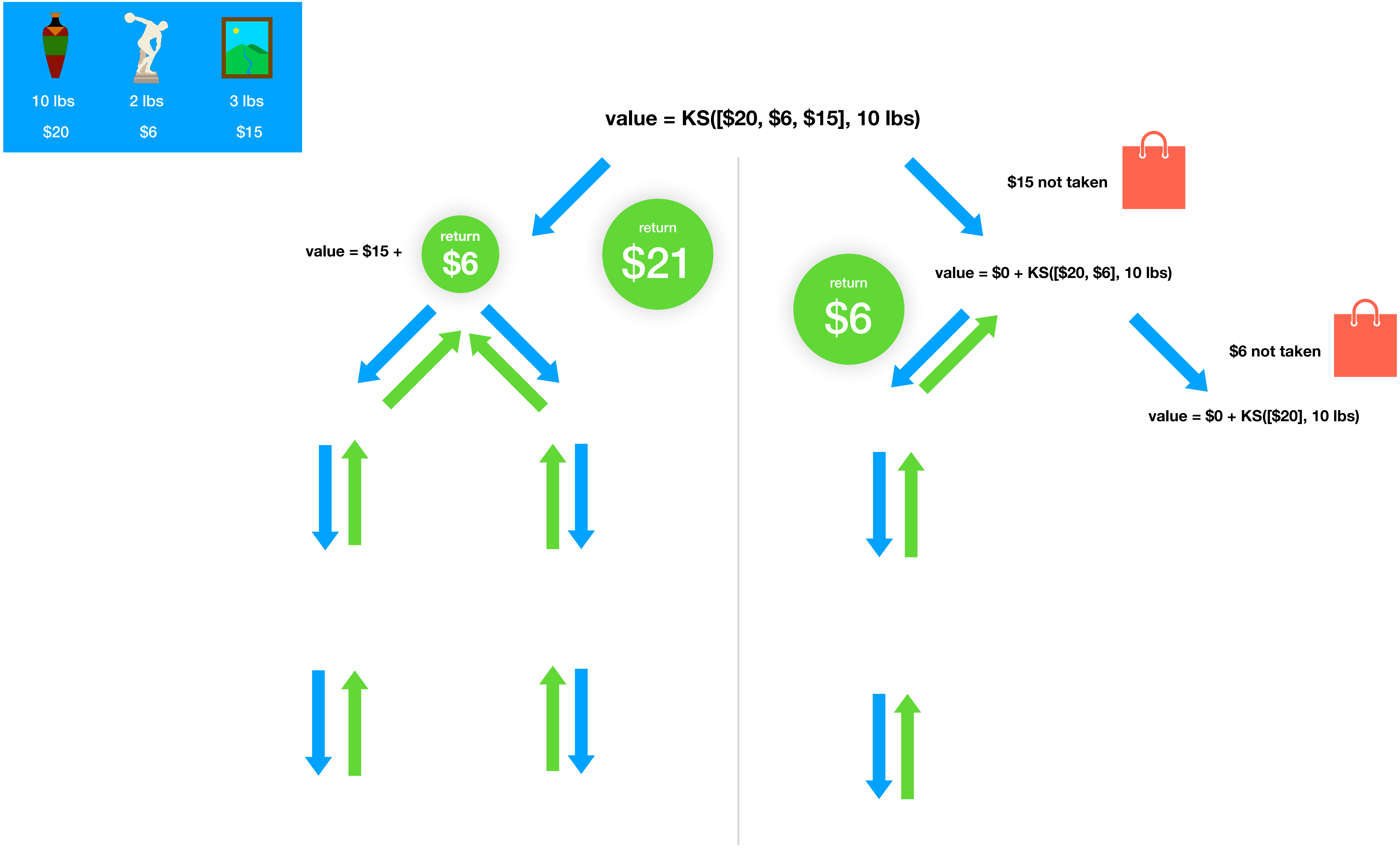

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)
Examine $20
Weight of 10 lbs greater than
capacity.
value = KS([ ] , 8 lbs)
Empty list, return 0.

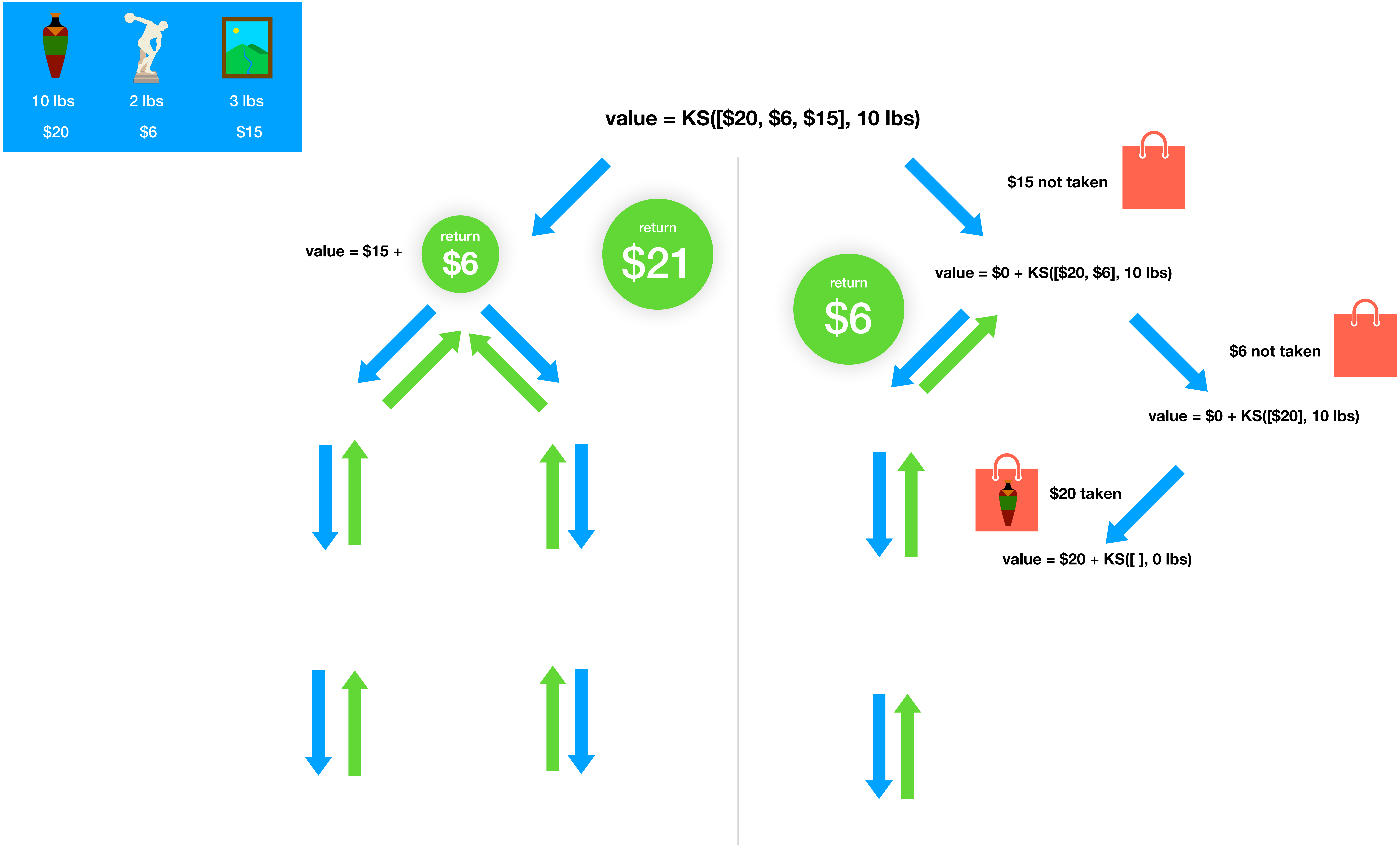

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 not taken
value = $15 +
return
$6
return
$21
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)
Examine $20
Weight of 10 lbs greater than
capacity.
value = KS([ ] , 8 lbs)
Empty list, return 0.

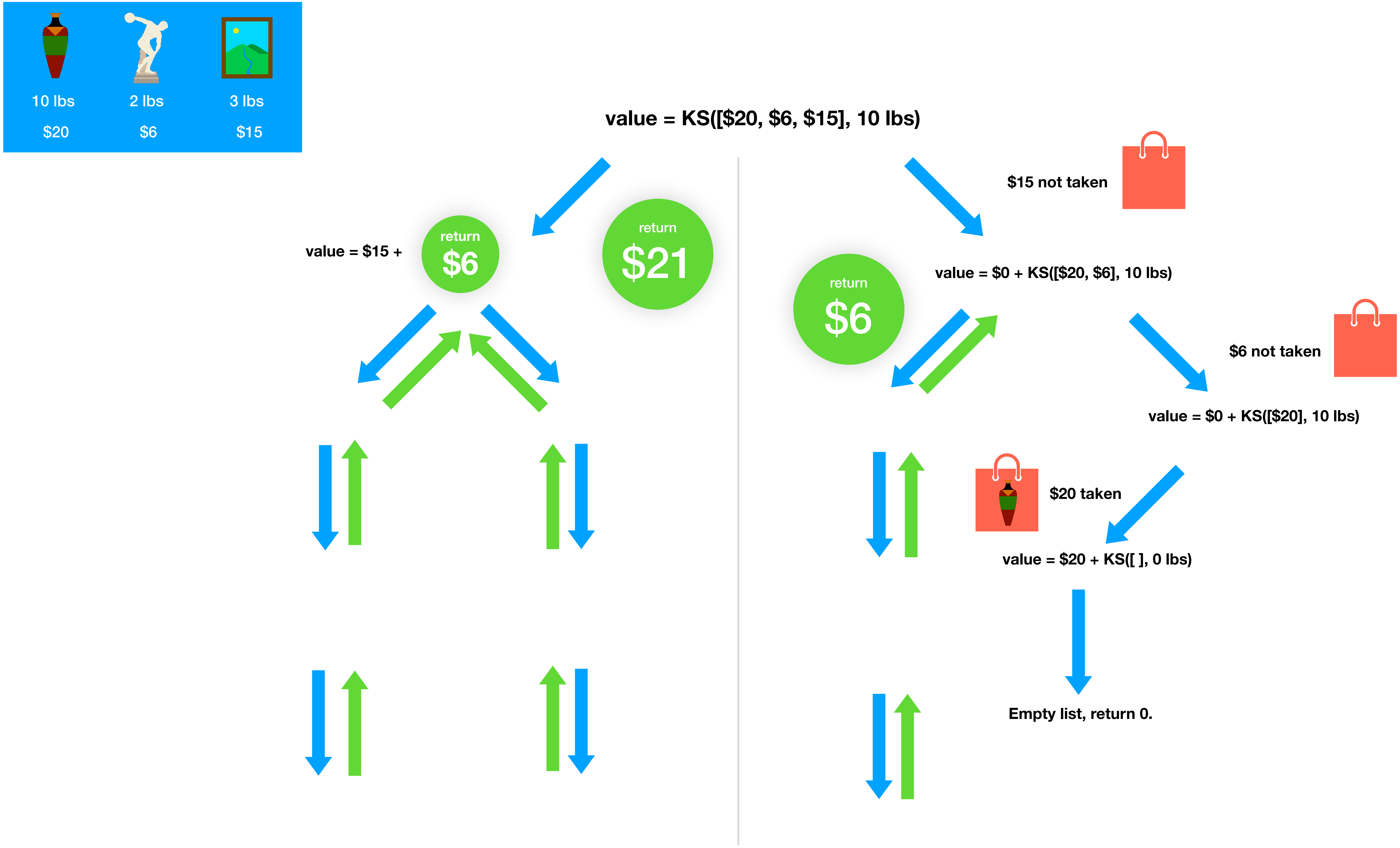

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)
Examine $20
Weight of 10 lbs greater than capacity.
value = KS([ ] , 8 lbs)
Empty list, return 0.

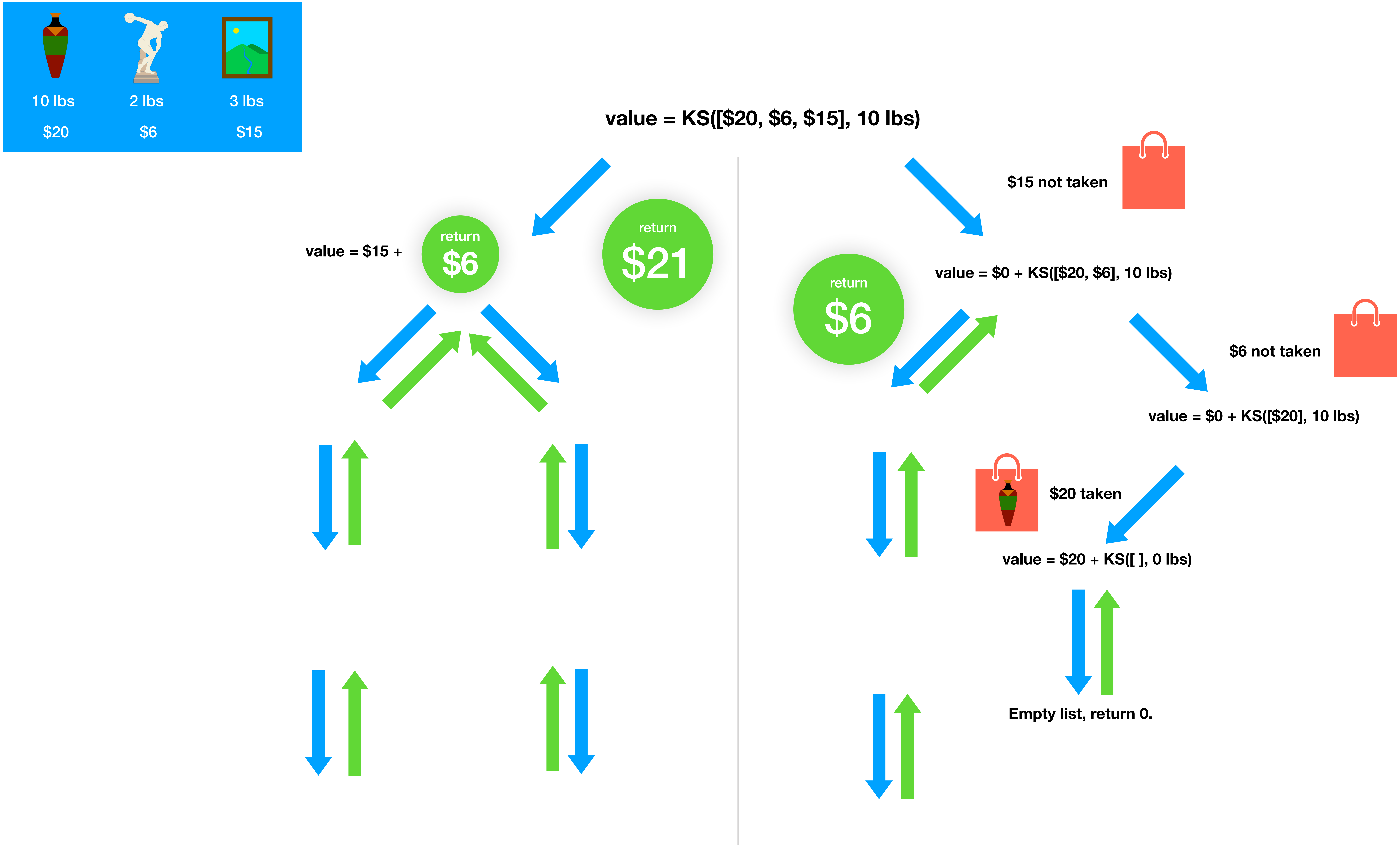

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
$6 taken
value = $6 + KS([$20], 10 lbs - 2 lbs)
Examine $20
Weight of 10 lbs greater than capacity.
value = KS([ ] , 8 lbs)
Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
return
$6

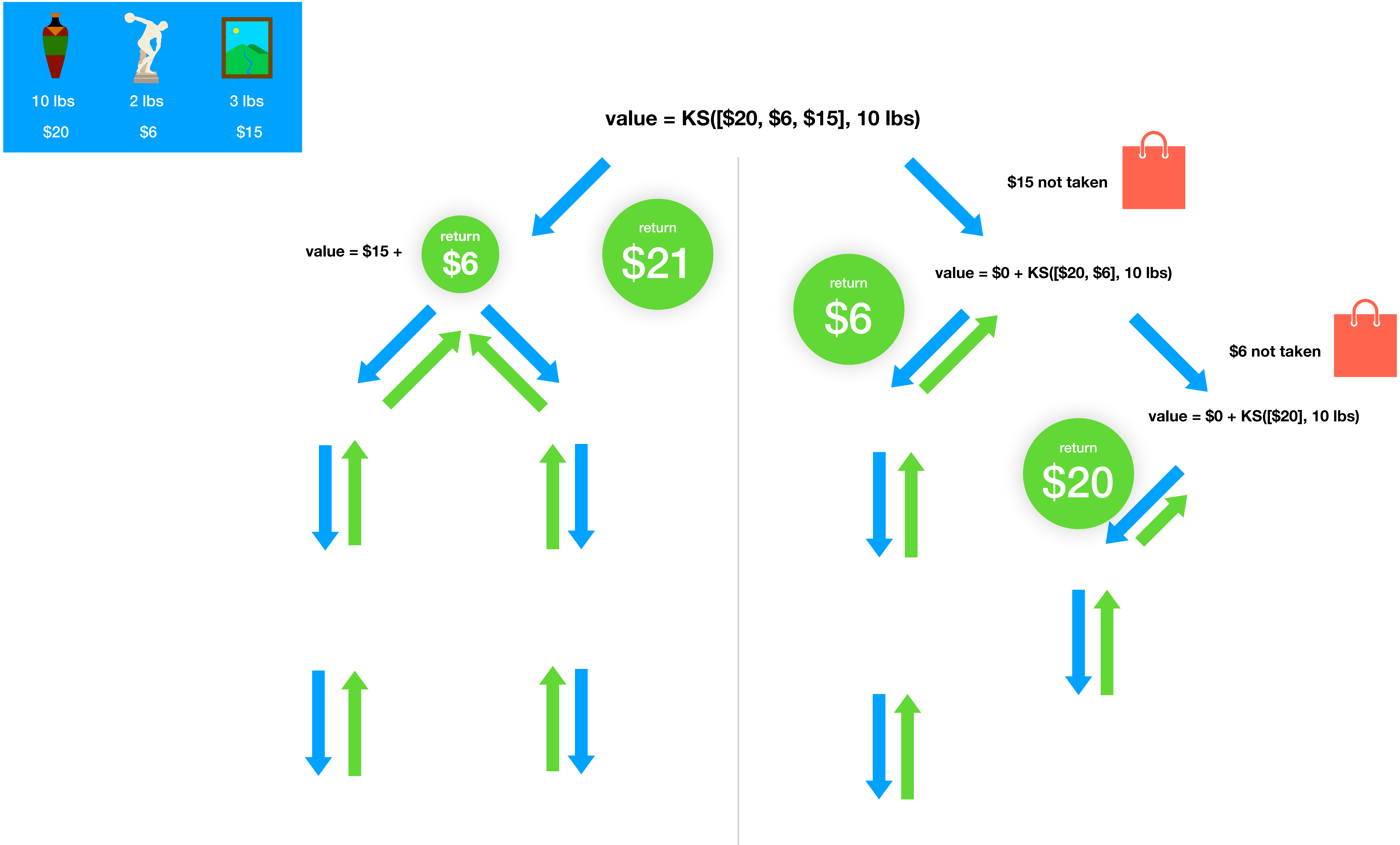

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)

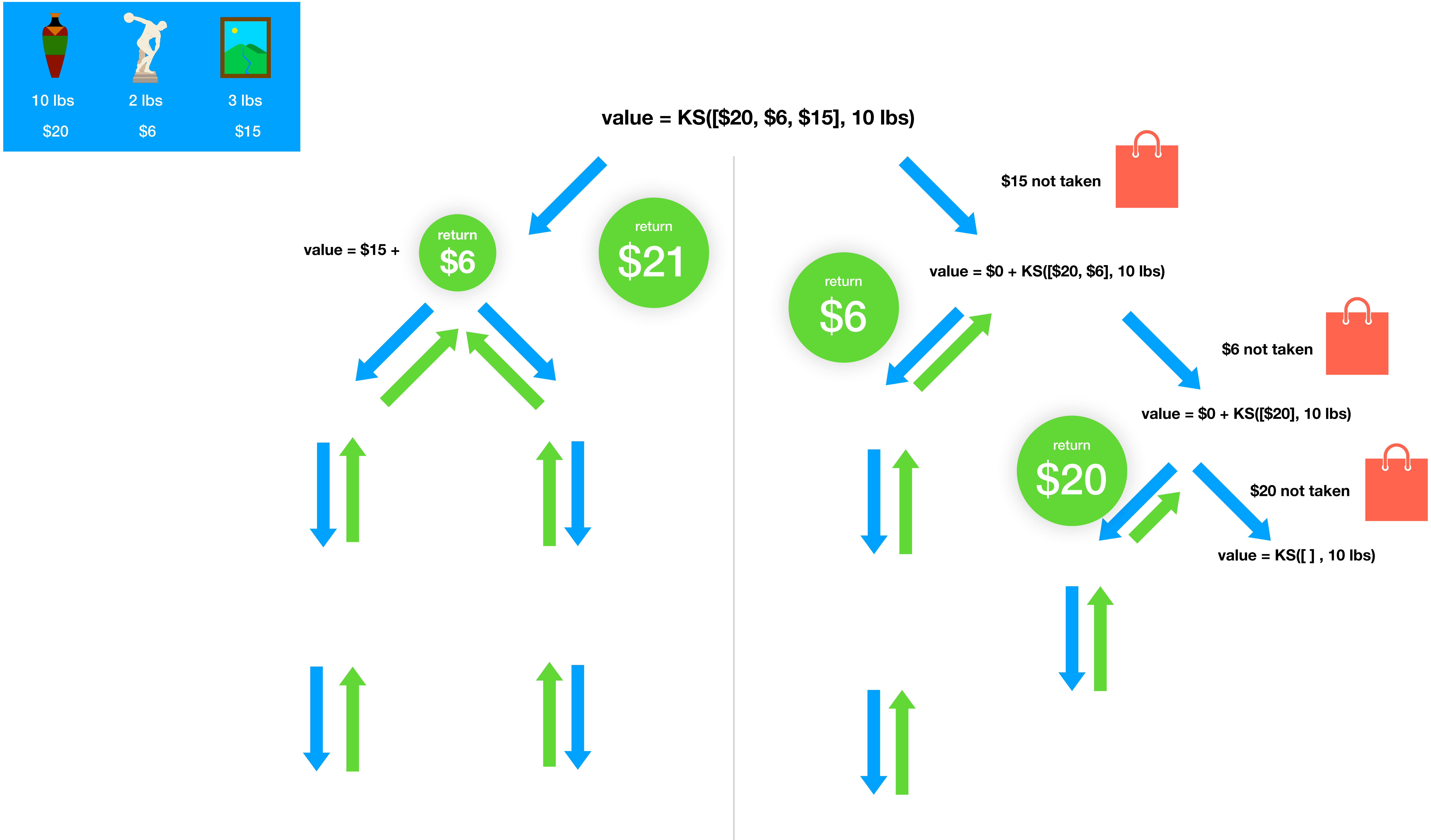

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
$20 taken
value = $20 + KS([ ], 0 lbs)

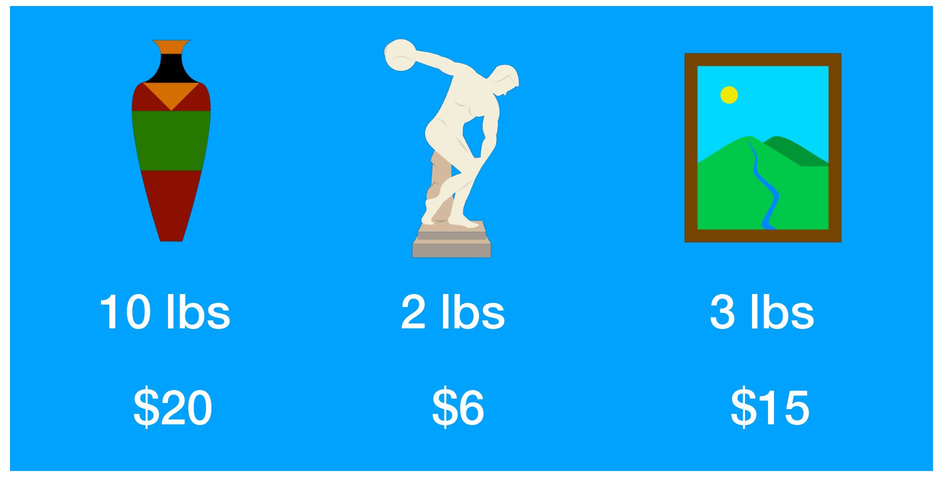

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
$20 taken
value = $20 + KS([ ], 0 lbs)
Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
$20 taken
value = $20 + KS([ ], 0 lbs)
Empty list, return 0.

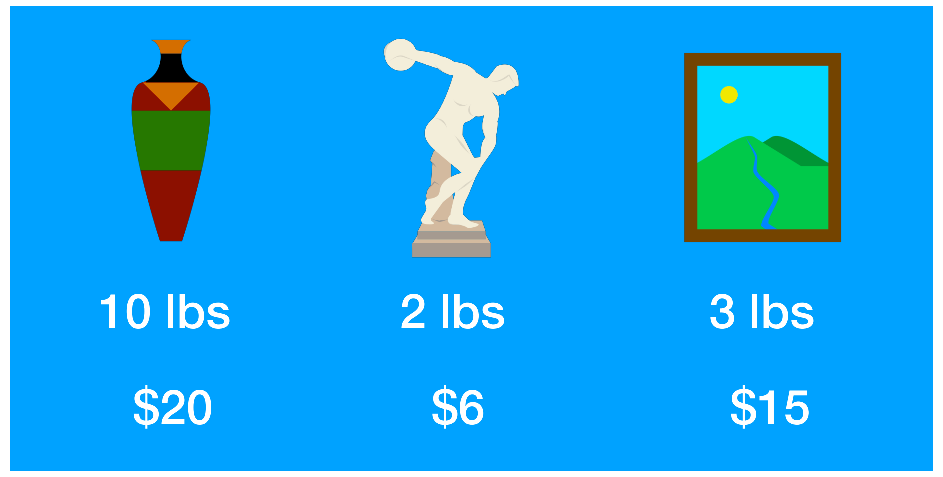
10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
$20 taken
value = $20 + KS([ ], 0 lbs)
Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20
$20 not taken
value = KS([ ] , 10 lbs)

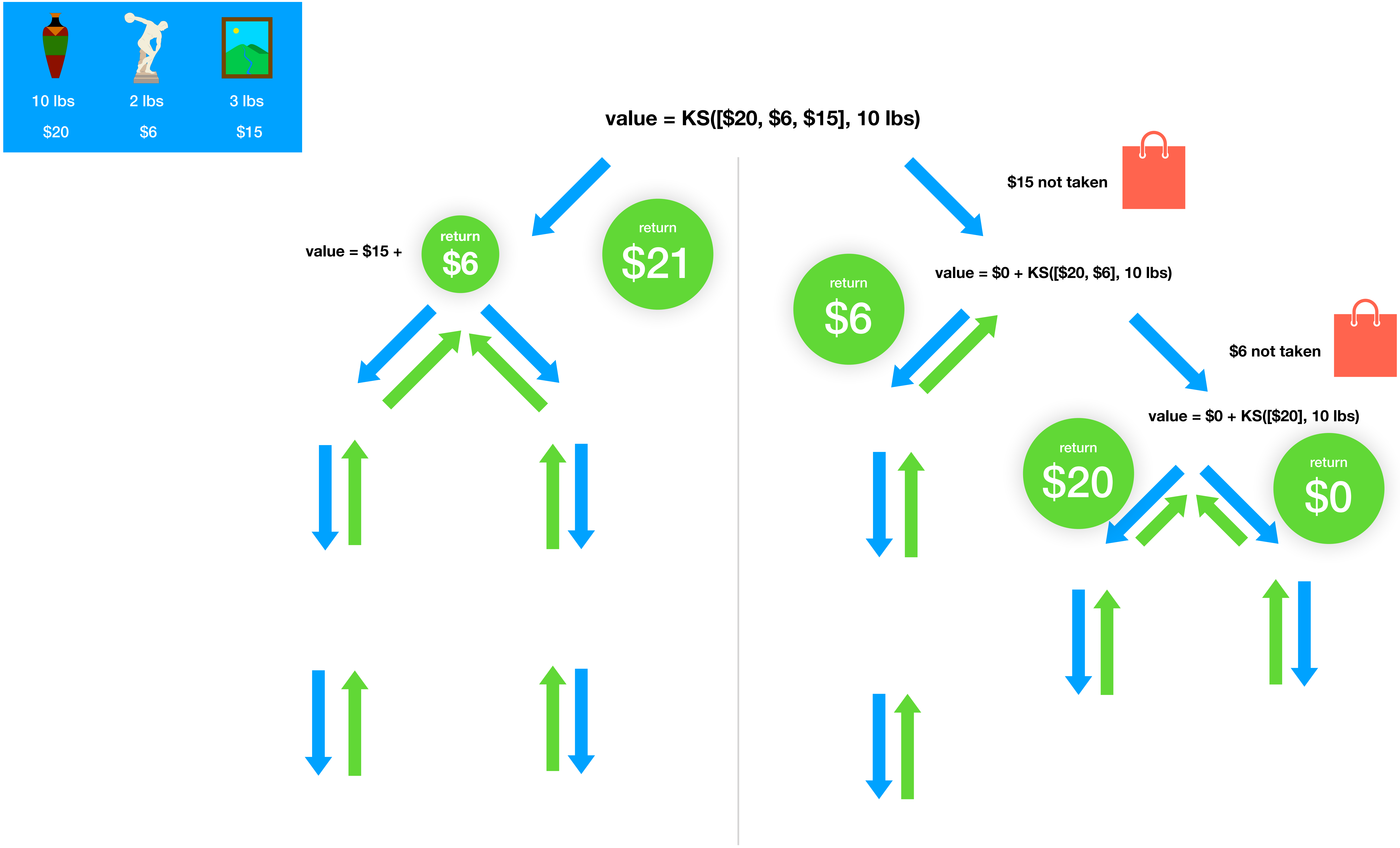

10 lbs
$20
2 lbs
$6
3 lbs
$15

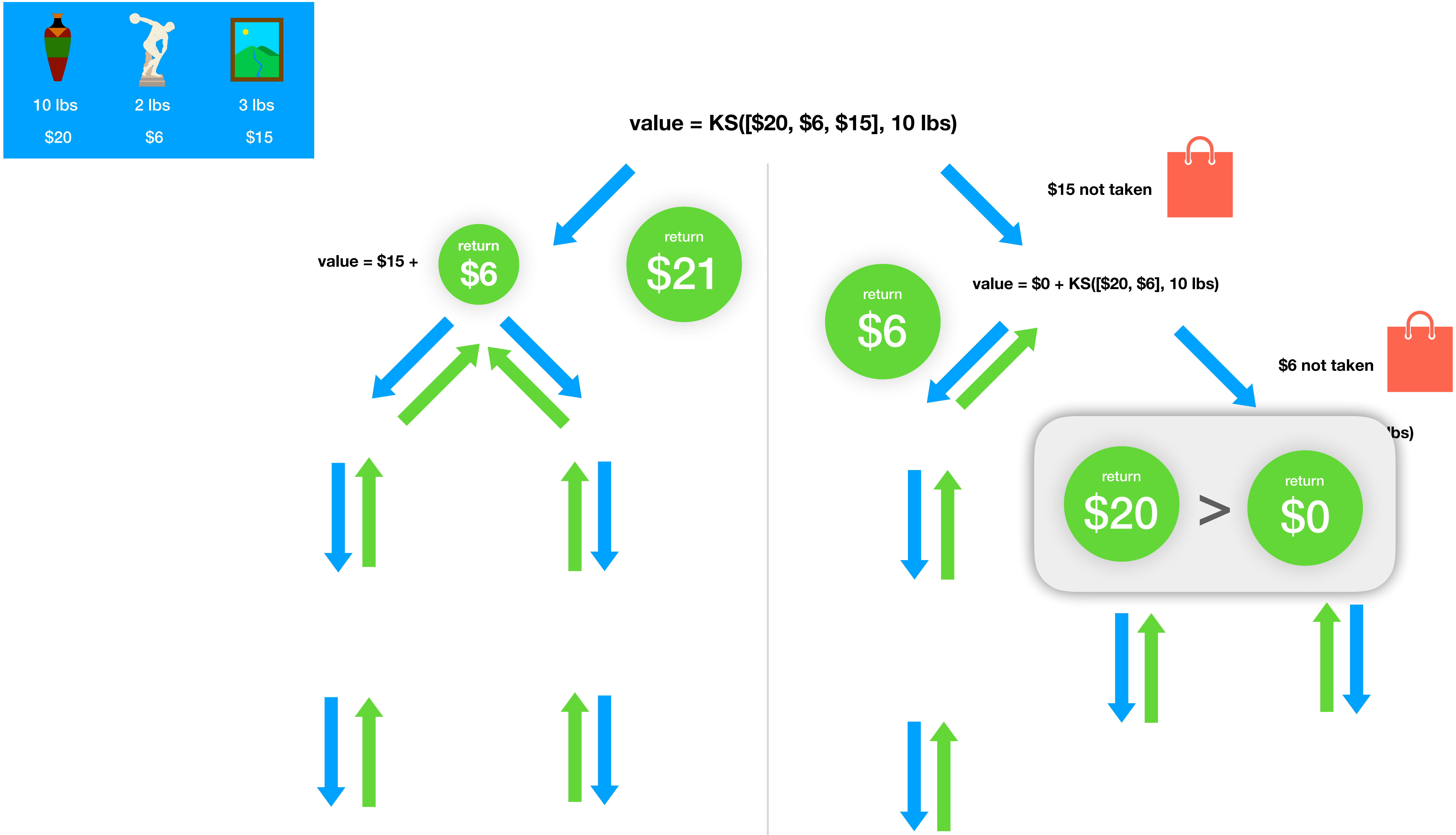

value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20
$20 not taken
value = KS([ ] , 10 lbs)
Empty list, return 0.

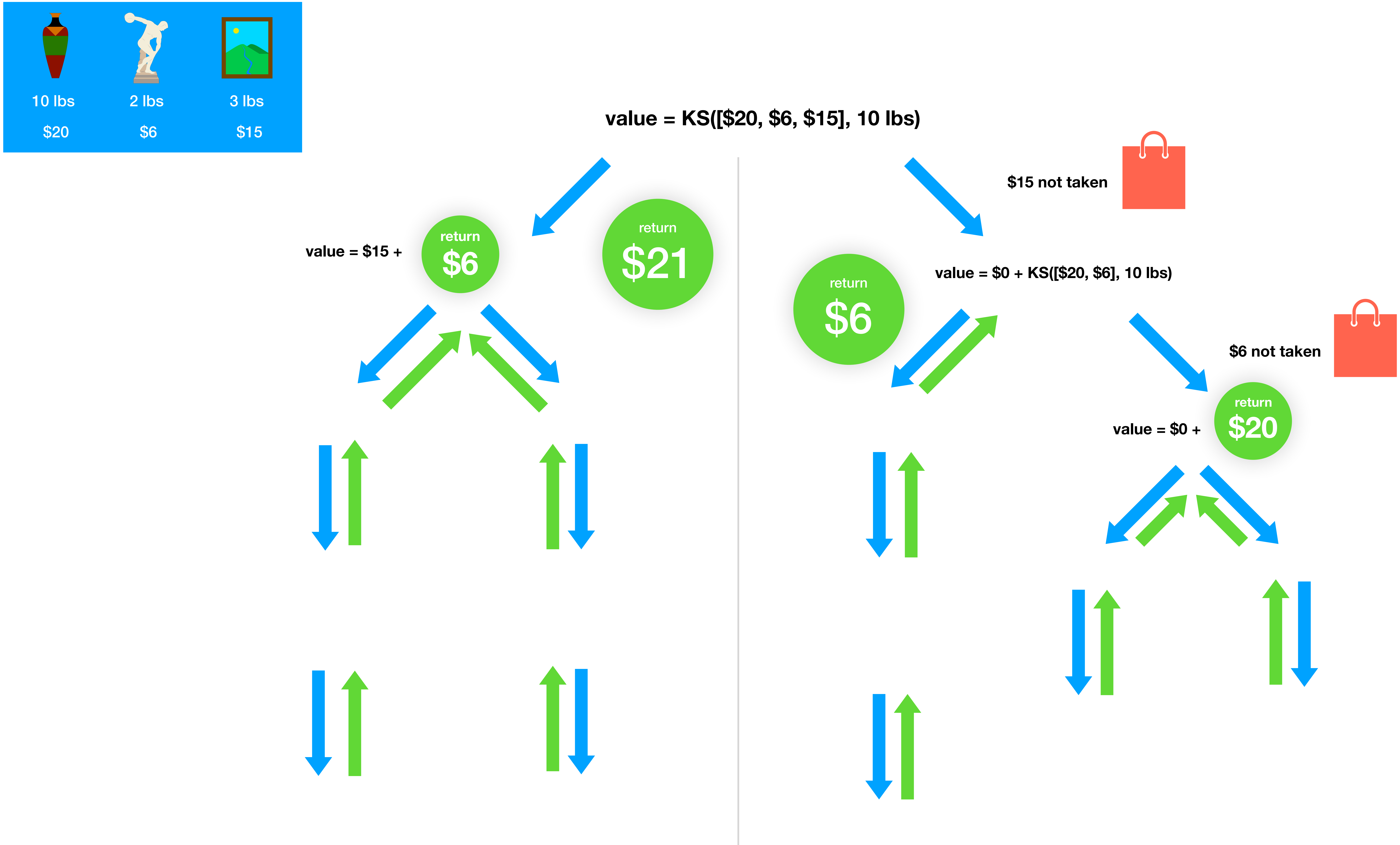

10 lbs
$20
2 lbs
$6
3 lbs
$15


value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
value = $0 + KS([$20, $6], 10 lbs)
return
$6
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20
$20 not taken
value = KS([ ] , 10 lbs)
Empty list, return 0.

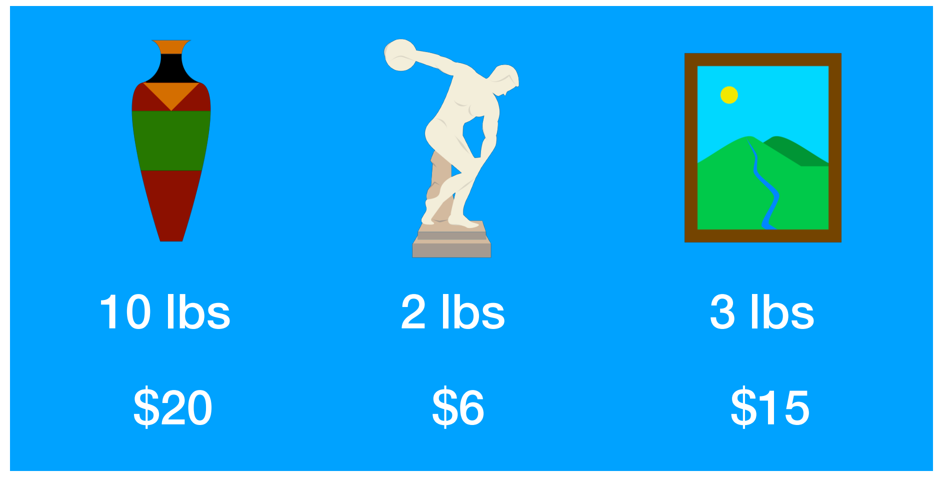
10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20
$20 not taken
value = KS([ ] , 10 lbs)
Empty list, return 0.


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 + KS([$20], 10 lbs)
return
$20
return
$0

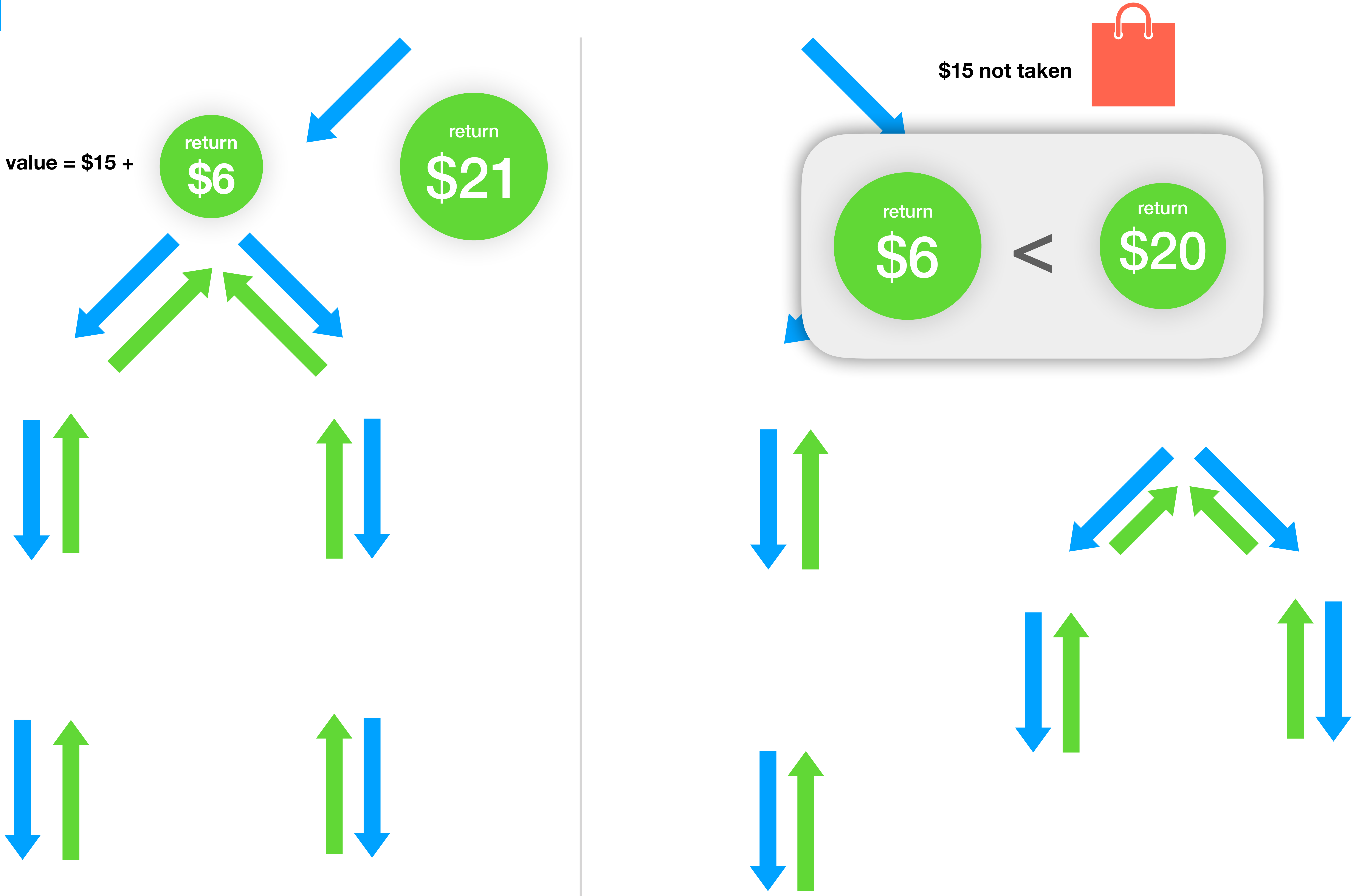

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
lbs)
return
$20
>
return
$0

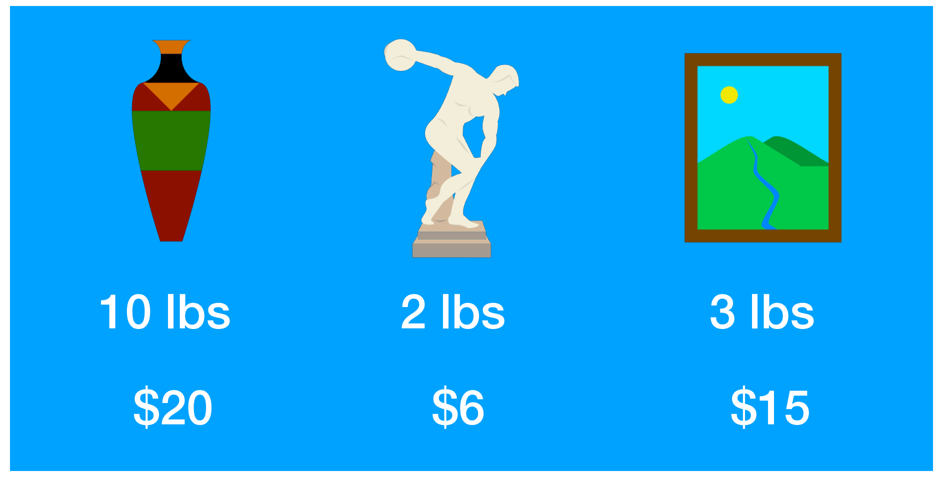

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
$15 not taken
return
$6
value = $0 + KS([$20, $6], 10 lbs)
$6 not taken
value = $0 +
return
$20

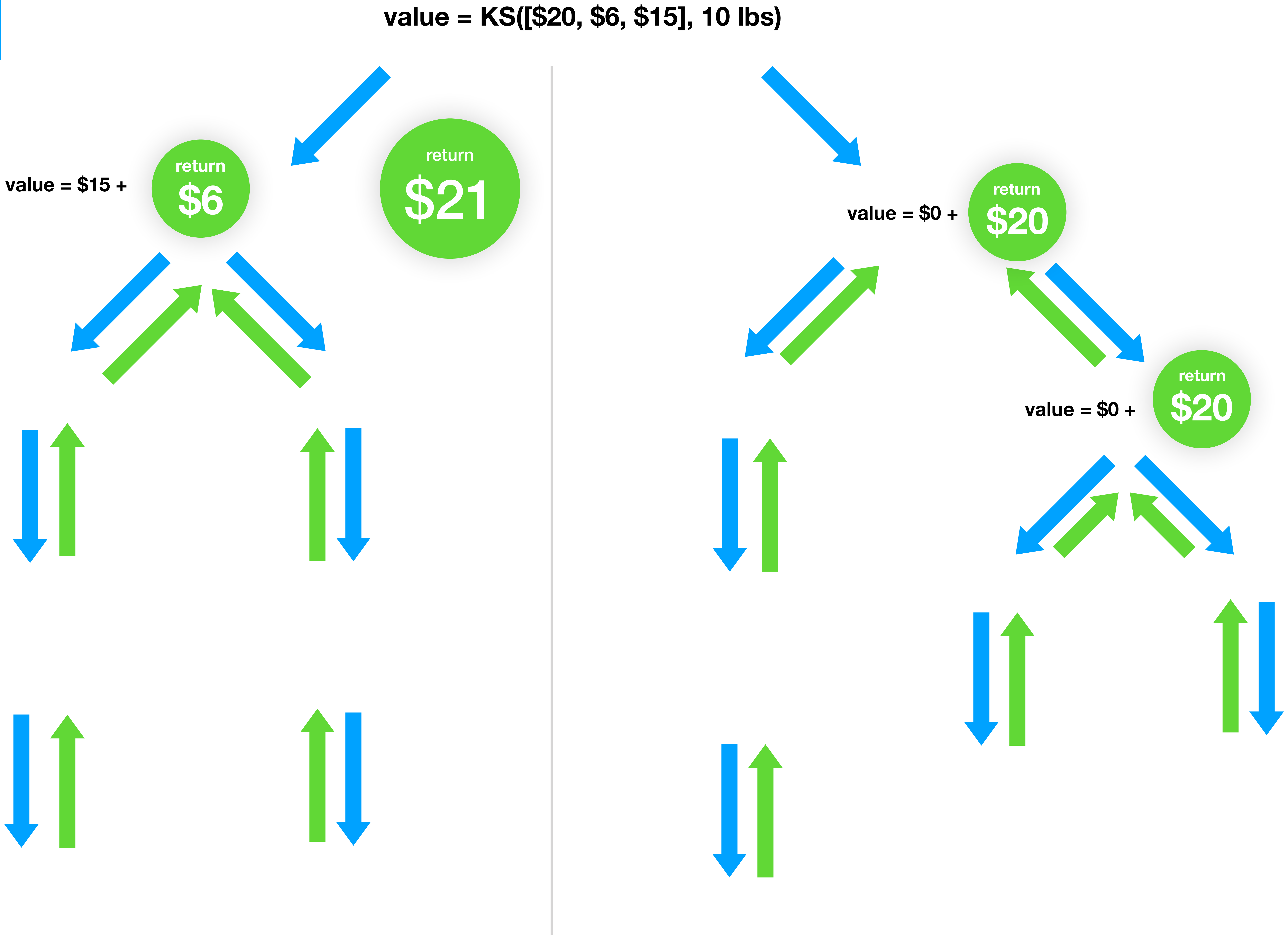

10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
$15 not taken
value = $15 +
return
$6
return
$21
return
$6
value = $0 + KS([$20, $6], 10 lbs)
return
$20
value = $0 +
return
$20



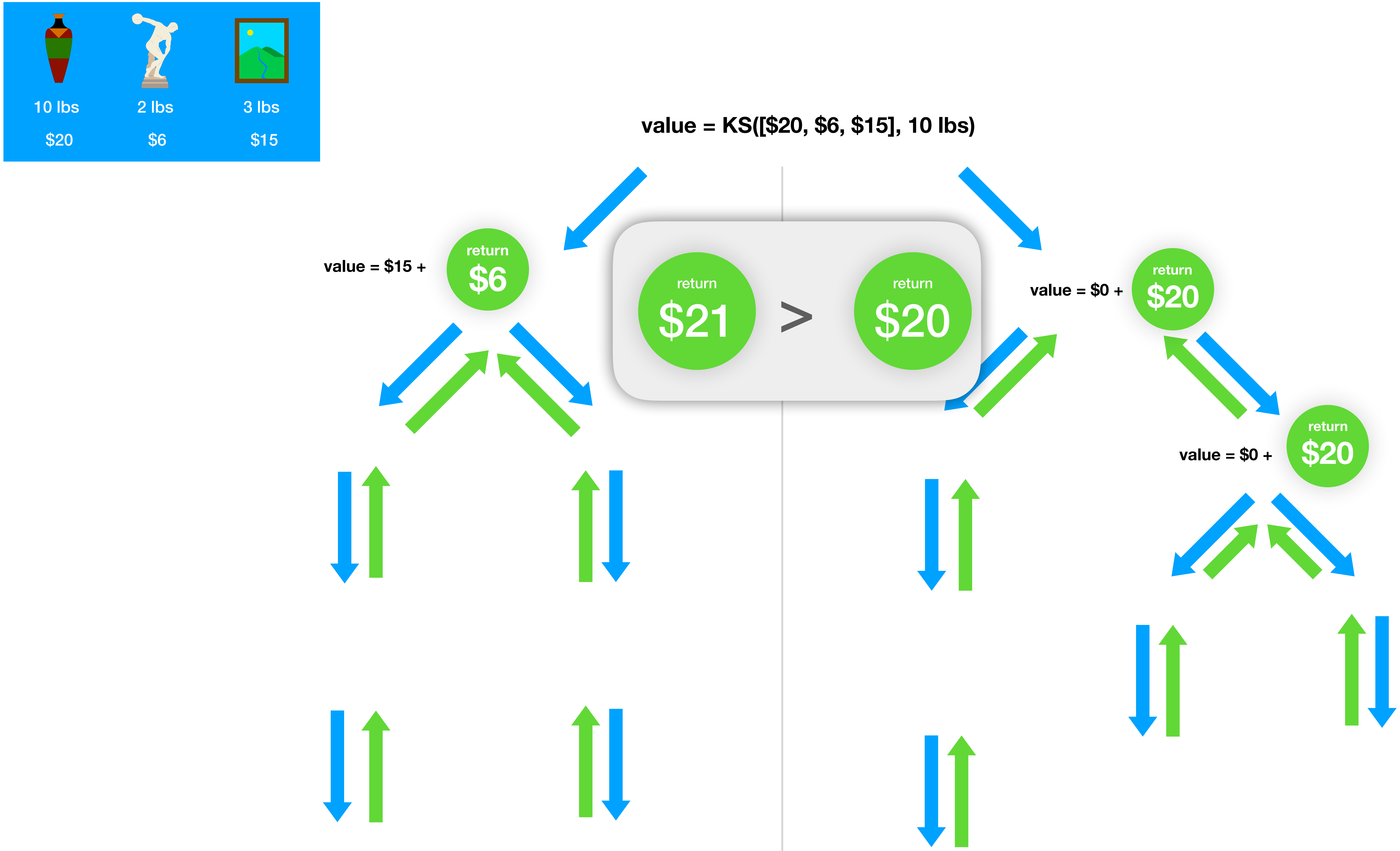
value = KS([$20, $6, $15], 10 lbs)



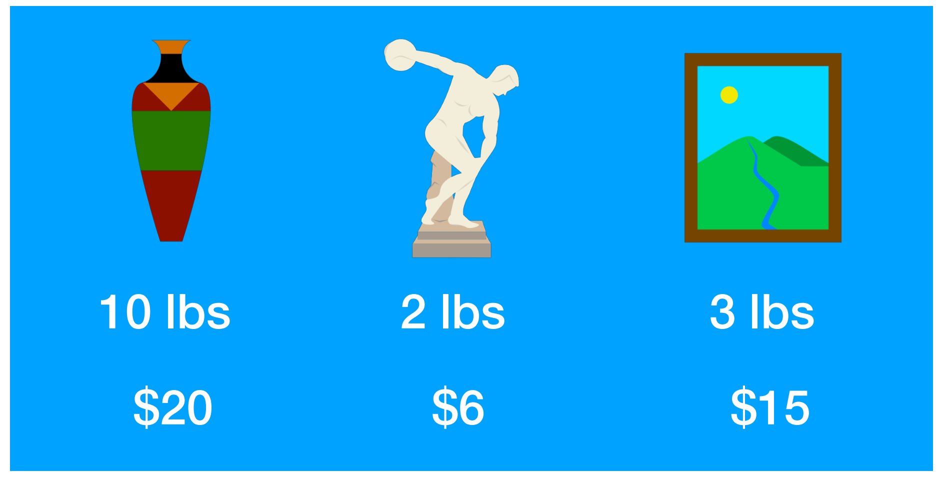

10 lbs
$20
2 lbs
$6
3 lbs
$15

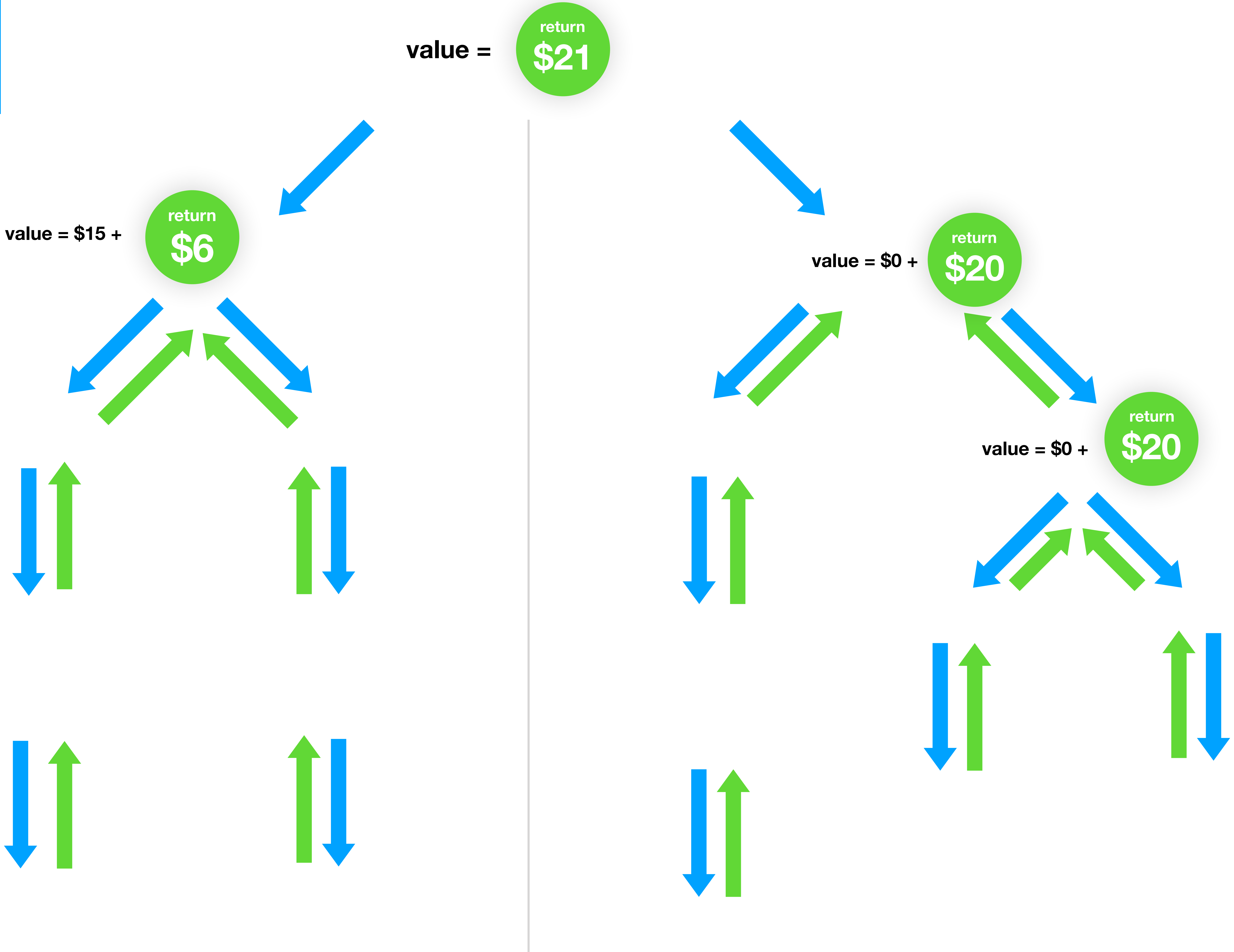

value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
value = $0 +
return
$20
value = $0 +
return
$20


10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
return
$20
value = $0 +
return
$20
value = $0 +
return
$20



10 lbs
$20
2 lbs
$6
3 lbs
$15
value = KS([$20, $6, $15], 10 lbs)
value = $15 +
return
$6
return
$21
>
return
$20
value = $0 +
return
$20
value = $0 +
return
$20


10 lbs
$20
2 lbs
$6
3 lbs
$15


value =
return
$21
value = $15 +
return
$6
value = $0 +
return
$20
value = $0 +
return
$20

Exercise: Write the code for the recursive Solution to 0-1 Knapsack

Greedy Strategies

- Choose item with maximum value
- Choose item with lightest weight
- Choose item with highest value/weight ratio.

Greedy Algorithm

- Compute the value-to weight ratios:
  - $r_i = v_i / w_i$
- Sort the items in non-increasing order of value to weight ratios
- For all items do:
  - If current item fits into the knapsack, add it to knapsack

Running Time for Greedy Approach

1. Sorting takes $O(N \log N)$, where N is the number of items.

2. The for loop takes $O(N)$

Total time is $O(N \log N)$

Requires a one-dimensional array to store the solution.

Fractional Knapsack

- Greedy approach
  - Sort in the ratio value/weight
  - Continue adding items with highest ratios, add as much of last item as possible
  - Optimal

References

Cormen, Thomas H., et al. *Introduction to Algorithms*. The MIT Press, 2014

https://en.wikipedia.org/wiki/Knapsack_problem