The whole point of the 0-1 knapsack problem is that when given a set of weights that correspond to specific benefits, get the most optimal benefit without exceeding the maximum weight of your knapsack. There are multiple ways you can go about doing this, and being greedy is one of them.

The greedy solution to the 0-1 knapsack problem will evaluate how much the item is worth or valued at based off the benefit to weight ratio. It will take the items and do their benefit over weight to get their “values” and then take items into the knapsack based off the highest values. In this project, we did not do any fractional knapsacks per the instructions; therefore, this approach just fills the knapsack to the weight limit without going over.

My experiments yielded the following results by using the set of weights and benefit below:

int[] weights = {-1, 60, 50, 60, 50, 70, 70, 45};  
int W = 100;  
int[] benefits = {-1, 180, 95, 40, 95, 40, 40, 105};

**double divv;  
int[] positions = new int[w.length];  
for (int i = 0; i < values.length; i++) {  
 divv = (double) b[i] / w[i];  
 values[i] = divv;  
}**

The bolded code above creates a new array of values which my greedy algorithm will later go through to find the highest valued item to place in the knapsack first. The code for the traversal of the values array can be found below and it shows how the highest value is founded. The code is truncated as the other lines did not pertain to the current matter at hand.

**int count = 0;  
for (int v = 0; v < values.length; v++) {**

**//other code here that is currently irrelevant  
 if (values[v] > maxValue) {  
 maxValue = values[v];**

**//other code here that is currently irrelevant  
 }  
}**

Now that the highest value is found, the algorithm can find the index at which the highest value was obtained and use that index value to find the weight and benefit that led to the “most optimal” value in my values array.

After back tracking a little bit, it finds itself with the following results:

Greedy Approximate Solution

Optimal set= {1} weight sum = 60 benefit sum = 180

Now you might think, well that isn’t the most optimal set? It should have taken the combination of either indexes {2,7} or {4,7}. With either have a weight combination of 50 and 45 and their benefits adding to up to 200, that must be the most optimal set, right? Not to mention that each of their values is approximately 1.9 each so combined it has a value of 3.8. True, but that’s not the only thing we are looking for so it is wrong. This algorithm ran on this specific set will find that at index 1, there is a value of 3. That is the largest value found and it has corresponded with a weight of 60 and a benefit of 180. Taking that into the knapsack first will not allow it to take anything else into the knapsack because anything else from the set will make it exceed the maximum weight limit of the knapsack (which is 100). The same is true for all the following test cases:

**Testcase #1**  
      int n = 7;  
      int[] weights = {-1, 60, 50, 60, 50, 70, 70, 45};   
      int W = 100;  
      int[] benefits = {-1, 180, 95, 40, 95, 40, 40, 105};  
              
Greedy Approximate Solution

Optimal set= {1} weight sum = 60 benefit sum = 180

      **Testcase #2**  
      int n2 = 18;  
      int[] weights2 = {-1,25,4,2,5,6, 2,7,8,2,1, 1,3,5,8,9,  6,3,2};  
      int W2 = 39;  
      int[] benefits2 = {-1,75,7,4,3,2,  6,8,7,9,6,  5,4,8,10,8,  1,2,2};  
     
Greedy Approximate Solution

Optimal set= {1} weight sum = 25 benefit sum = 75

      **Testcase #3**  
      int n3 = 20;  
      int[] weights3 = {-1, 10,14,35,12,16, 20,13,7,2,4, 3,10,5,6,17,

                         7,9,3,4,3};   
      int W3 = 29;  
      int[] benefits3 = {-1, 2,13,41,1,12, 5,31,2,41,16,2,12,1,13,4, 51,6,12,1,9};   
        
Greedy Approximate Solution

Optimal set= {9} weight sum = 2 benefit sum = 41  
  
      **Testcase #4**     
      int n4 = 7;  
      int[] weights4 = {-1, 2,5,3,2,5,3,7 };   
      int W4 = 10;  
      int[] benefits4 = {-1, 5,10,5,20,15,5,10};

Greedy Approximate Solution

Optimal set= {4} weight sum = 2 benefit sum = 20

The greedy method is interesting as it is not the best way to find the most optimal set and it will not give you your best bang for your buck for the items in your knapsack.