Jason Caplan
Design and Analysis of Algorithms
Professor Leff
HW9

1. Since all items are all of the same weight, we just want to take the items with the highest values. Therefore the optimal solution will be to take the top $p = \frac{W}{wn}$% of the items with respect to highest v_i, since this is the ratio of $\frac{allowed\ weight}{available\ weight}$. Practically, this means taking the top $t = \lfloor pn \rfloor$ items, and whatever fraction of the next-highest-valued item remaining that we can fit. But we don't actually need an $O(nlogn)$ sort to do this – we can simply use the $O(n)$ DSelect algorithm discussed in class to select the "pivot" item such that $\lfloor pn \rfloor$ items are valued higher than it, and then trapse through the item list and add items to our knapsack that have a higher value than that "pivot" item (finally adding whatever fraction of the "pivot" item can fit in the remaining space in the knapsack).

2. Pseudo-code:

```
fractional_knapsack_same_weight(values = array of values v_i,
                                W = total allowed weight,
                                w = weight of a given item):
    // Step 1 -> find our "pivot item" k, and save its value
    top_percent = W divided by w // top percentage (highest values) we desire to
                                 // take (rounded down to nearest integer)
    k = values.length – top_percent
    value_k = values[ DSelect(values, k) ]  // using 1-indexing for simplicity

    // Step 2 -> run through the items, adding those that are in the top%
    weight_remaining = W
    for v_i in values:
       if v_i > value_k:
          add item i to knapsack
          weight_remaining -= w

    // Step 3 -> fill in the remaining space in the knapsack
    //              with a fraction of the pivot item
    add (weight_remaining divided by w) of item k
```

3. Performance and Correctness:
    a. Performance is $O(n)$:
        i. Step 1 is two $O(1)$ arithmetic calculations, and one $O(n)$ call to DSelect
        ii. Step 2 is $O(n)$ traversal of the values array
        iii. Step 3 is an $O(1)$ calculation
    b. Correctness
        i. In the classical knapsack problem, the optimal solution is to add the items in descending order of $\frac{value}{weight}$ ratio. Since here the weights are the same, this translates to descending order of $value$. We accomplish the same additions (see answer to #1), albeit out of order, but this does not matter because the knapsack does not "care" in which order its optimal items are added.