CS 180 Homework 4

Jeremy Cristobal 604982952

n = number of cristaline chunks
k = max pounds that can be loaded
$A = \{a_1, a_2, \dots, a_n\}$ (weight of each cristaline chunk)
M = 2D array that is $(n+1)$ by $(k+1)$

```
for (w = 0 to k) {
    M[0, w] = 0
}
for (i = 1 to n) {
    for (w = 1 to k) {
        if (a_i > w) {
            M[i, w] = M[i-1, w]
        }
        else {
            M[i, w] = max{M[i-1, w], a_i + M[i-1, w-a_i]}
        }
    }
}
B = empty set
x = n
while (M[n, k] == M[n, k-1]) {
    k = k-1
}
while (x > 0) {
    if ((M[n, k] - a_x) == (M[n, k-a_x])) {
        add a_x to set B
        k = k - a_x
    }
    x = x-1
}
Return B
```

CS 180 Homework 4 Part II

Time complexity:

The most complex part of my algorithm is the Subset Sum/Knapsack portion. Because of the nested loops of different lengths, the time complexity is $O(nk)$.

Proof:

The first part of my algorithm is very influenced by the Knapsack Problem. However, the notable difference is that these cristaline chunks have the same value as weight, so we can eliminate one of the sets the Knapsack Problem requires.

Once we have acquired our 2D array, we take the bottom-right value. We move left until the array value is equal to the value of the second aspect of the array, if it is necessary at all.

Our final loop does the tricky stuff. We take our last value (a_x) and subtract it from our value $M[n, k]$. If our difference exists in its correct spot, we add a_x to our new set B because this means a_x is a possible choice to achieve the maximum weight we calculated with our Knapsack algorithm. We then subtract the weight from k, and repeat this whole process until $k = 0$, which means that we have found a set of chunks that equals exactly our maximum calculated weight.

And once we have found this set, we return it.