HW #20

1)

```
randomsolution(items, W)   # W is max weight of knapsack
 for all item in items:
  int random = random(0 or 1)
  if random == 1:
   if (totalweight + item.weight <= W)
     totalWeight += item.weight        > Add item to knapsack
     knapsack[itemindex] = 1
   else: Don't add item -> knapsack[itemindex] = 0
return knapsack

localsearch(items, W)
randomknapsack = randomsolution(items, weight)
currentmax = sum of values of items stored in knapsack
itemindex = 0
while true:
   while true:                                   # get neighbor
     flip bit of item in randomknapsack at itemindex (if =1, remove, if 0, add)
     if (randomknapsack.totalweight > W)
       revert bit change back, to before
       if itemindex = items.size then itemindex = 0, else itemindex ++
     else: break
   if (randomknapsack.totalvalue > currentmax)
     currentmax = randomknapsack.totalvalue
     if itemindex = items.size then itemindex=0, else, itemindex++
   else: revert knapsack to prev state then break
return randomknapsack
```

My algorithm works using a bit array of 0 & 1s representing each item. The random solution loops through the size of items, & uses a random number generator to decide add or not for the initial solution. The weight of the knapsack is checked to ensure a valid solution is made, & returned to the main local search function.

Once the random solution is made, the value of the items with 1 in the bit array is saved as the current max. The main while true loop starts, only stopping when the neighbor solution is < the current max, which indicates a local max has been found for total value, since we are trying to max that.

As for finding neighbors, I chose the processes of flipping 1 bit each iteration of the loop, using an item index iterator to track which bit to change. (Ex. if random solution = 0101 for 4 item knapsack, 1st iteration neighbor = 1101, 2nd iteration = 1001, 3rd 1011, etc) It is not a perfect method to traverse all neighbors, but should be enough to find a local max.

2) The whole concept of a local search is greedy in the sense that it takes the first max/min it sees. Graphically speaking, an absolute max/min comes from the set of local max/mins, but a local max/min is not guaranteed to be the absolute max/min. Without trying all possible solutions, there is no way to know if it is an optimal solution. Therefore, there's a chance it is the optimal answer, but we can't guarantee it with out testing the rest of the solutions, which goes against the greedy design of this algorithm.