Homework 3

1. Gadget Testing
    a. Brute Force

This algorithm would be to simply starting at level 1 and going up 1 level at a time, drop the gadget and stopping when it breaks.

Pseudo Code:

Floor = 0

While (notBroken) :

    Floor += 1

    dropGadget(floor)

Return floor

Worst Case: Gadget is unbreakable and you need to drop from every floor of building during testing (time = n)

    b. Optimization (better than n (ie log n or constant)

The help of a second gadget at first thought would be to begin cutting n in half. Then we could start at $n/2^{nd}$ floor, drop gadget, if breaks then brute force bottom half. If not, then try the middle of the top half and so on.

Pseudo Code:

NumFloors = n // number of floors to be checked

G1Floor = 0

G2Floor = numFloors /2 // half way point

While (g2 != broken):

dropGadget(g2, g2Floor)

if (g2 != broken):

g1Floor = g2Floor

temp = (numFloor – g2Floor) / 2

g2Floor += temp // only need to go up if unbroken

if g2Floor = numFloor

unbreakable gadget

Reapeat first Algo from g1 -g2 floor once broken

Time Complexity:

Average would be (log n) since we are potentially cutting number of trials in half for each attempt with g2 gadget, with worst case still being n (though n/2, still linear time)

T(n) = T(n/2) + 4 x=n/2 t(n) = t(n/4) +8 x= n/4 t(n) = t(n/8) + 12
T(n) = T(n/2^k) + 4k T(1) = c : n = 2^k k = log(n)
T(n) = T(n/n) + 4log(n) = c + 4 log(n)

2. Counting Context
    a. Brute Force

My brute force algorithm would find an A in the string then add the number of B's after to a counter, then find the next A and do it again and again.

Pseudo Code:

Counter = 0

For letter in InputString:

If letter == A:

From current to end of list

If current == B:

Counter += 1

Worst Case

T(n) = $\sum_{i=0}^{n} \sum_{j=i}^{n} 1 = \sum_{i=0}^{n} n - i + 1$ = (n+1) (n(n+1)/2) in Θ(n^3)

Best Case (no A found)

Θ(n)

b. Optimization

Thoughts on more efficient: indexes (no, still nested for loop), Find B's and look Back (Would be same as A's but Best Case would be no B's instead), Many languages could use predefined func to get all combos of A (*) B then just return the length but not our algorithm.

Psuedo Code:

Not sure what would be better.

3. Graph (figure 1)
   a. Adjacency Matrix and list (assuming no self loops)

Adjacency Matrix

	A	B	C	D	E	F	G
A	0	1	1	1	1	0	0
B	1	0	0	1	0	1	0
C	1	0	0	0	0	0	1
D	1	1	0	0	0	1	0
E	1	0	0	0	0	0	1
F	0	1	0	1	0	0	0
G	0	0	1	0	1	0	0

Adjacency List

A -> B, C, D, E B -> A, D, F C -> A, G D-> A, B, F

E -> A, G F -> B, D G -> C, E

b. Start at A, DFS

Push A, Push B, Push D, Push F, Pop F, Pop D, Pop B, Push C, Push G, Push E, Pop E, Pop G, Pop A

c. Start at A, BFS

Queue:

A, B, C, D, E, F, G

A adds b,c,d,e then B adds f, then C adds g and all nodes are seen

Tree: (Assume arrows going down the tree)

A

B C D E (all added from parent node A)

F (added by B)

G (added by C)

4. Knapsack
    a. Write pseudo code and determine efficiency

Thoughts: Like in class, brute force and try every subset but knowing we are basically given them all on their own

Other funcs:

AddWeights // takes in array and adds up weights at indexes in that array

AddValue // same as above but for values

Value = {v1, … , vn}

```
Weight = {w1, … , wn}
Possibilities = {}
For i in n:
    Tester = {i}
    If w(i) <= W :
        Possiblilites.append(i)  //append the index of single values
    For j = i+1 to n: //build bigger and bigger arrays until reaches end
        Tester.append(j)
        If addWeight(tester) <= W
            Possibilities.append(tester) // something like python could put array into another array
        Else:
            Tester.pop // remove element that made too heavy
bestValue = {}
// finally, find best value from weight restricted list of arrays
for i in possibilities:
    if addValue(possibilities(i)) > addValue(bestValue)
        bestValue = possibilities(i)
```

This kind of works though it would not do something like {1,3} as an option but does well to cover most sequential attempts?

b. Consider given scenario

Other funcs:

AddWeights // takes in array and adds up weights at indexes in that array

AddValue // same as above but for values

valueOrder // reorganizes BOTH lists into order from highest to smallest

Value = {v1, … , vn}

Weight = {w1, … , wn}

Possibilities = {0}

valueOrder(value, weight)

// now similar to above attempt

For i=1 to n:

    Possibilities.append(i)

    If addWeight(tester) > W

        possibilities.pop // remove element that made too heavy

In this one, no need to have the single arrays and can just start at highest value, so no second for loop to find in possibilities.

Looks nicer than what I did but I feel like I am missing something and so not sure if I implemented correctly.

c. Prove greedy works

Ignoring possibility that reordering list may cause the slowdown, will look at the rest of the written above.

T(n) = $\sum_{i=1}^{n} 2 + 1 = 2n + 1$ which is in $\Theta(n)$??

(that doesn't seem right but see comment at bottom of part b and top of c)

d. See Code from HW3 Code assignment