# CS 4720/5720 Design and Analysis of Algorithms
# Homework #2
Daniel Frey

## Answers to homework problems:

1. Book Problem: 3.2.3 *(Gadget Testing)* Algorithm to determine highest floor gadget can fall without breaking.

   Drop from each floor sequentially, if gadget broken, return current floor - 1.

```
testGadget(n) (n is total number floors)
    for i = 1 to n :
      if broken :
        return i-1
```

   Best-case: 1 iteration (breaks first floor)
   $T(n) = \sum_{i=1}^{1} 1 = 1 \in \Theta(1)$
   Worst-case: all iterations (doesn't break)
   $T(n) = \sum_{i=1}^{n} 1 = n \in \Theta(n)$

2. Book Problem 3.2.8 *(Counting)* Count all substrings that begin with A and end with B.

   (a) Brute force algorithm, best and worst-case efficiencies.

   Find A, find all subsequent B's and increment counter. Advance to next position, then repeat.

```
countSubstr(S[0,n-1])
    count = 0
    for i = 0 to n-2 :
      if S[i] == 'A' :
        for j = i + 1 to n-1 :
          if S[j] == 'B' :
            count = count + 1
    return count
```

   Best-case: No A's *(Second for loop doesn't execute)*
   $T(n) = \sum_{i=0}^{n-2} 1 = n - 1 \in \Theta(n)$
   Worst-case: All A's *(Iterate through entirety of each for loop)*
   $T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 2 = n(n-1) \in \Theta(n^2)$

   (b) More efficient algorithm. *[In class of $\Theta(n)$]*

   Count A's and B's in one pass. Sum A's and add to running total when B found.

```
countSubstr(S[0,n-1])
    count = 0
    countA = 0
    for i = 0 to n-1 :
      if S[i] == 'A' :
        countA = countA + 1
      if S[i] == 'B' :
        count = count + countA
return count
```

3. *(Knapsack)*

   (a) Exhaustive algorithm, and determine efficiency class.

   Iterate through each permutation checking to see if specific item is present. If present, add value and weight to running total. If current combination of items satisfies and is best, mark as best.

```
exhaustiveKS(maxWeight, value[0,n-1], weight[0,n-1])
    bestVal = 0
    bestComb = 0
    for i = 0 to 2^n :
      currVal = 0
      currWeight = 0
      for j = 0 to n-1 :
        if j is in i :
          currVal = currVal + v[j]
          currWeight = currWeight + w[j]
      if currVal > bestVal and currWeight ≤ maxWeight :
        bestVal = currVal
        bestComb = i
    return bestComb[ ], currVal, currWeight //best is current at end
```

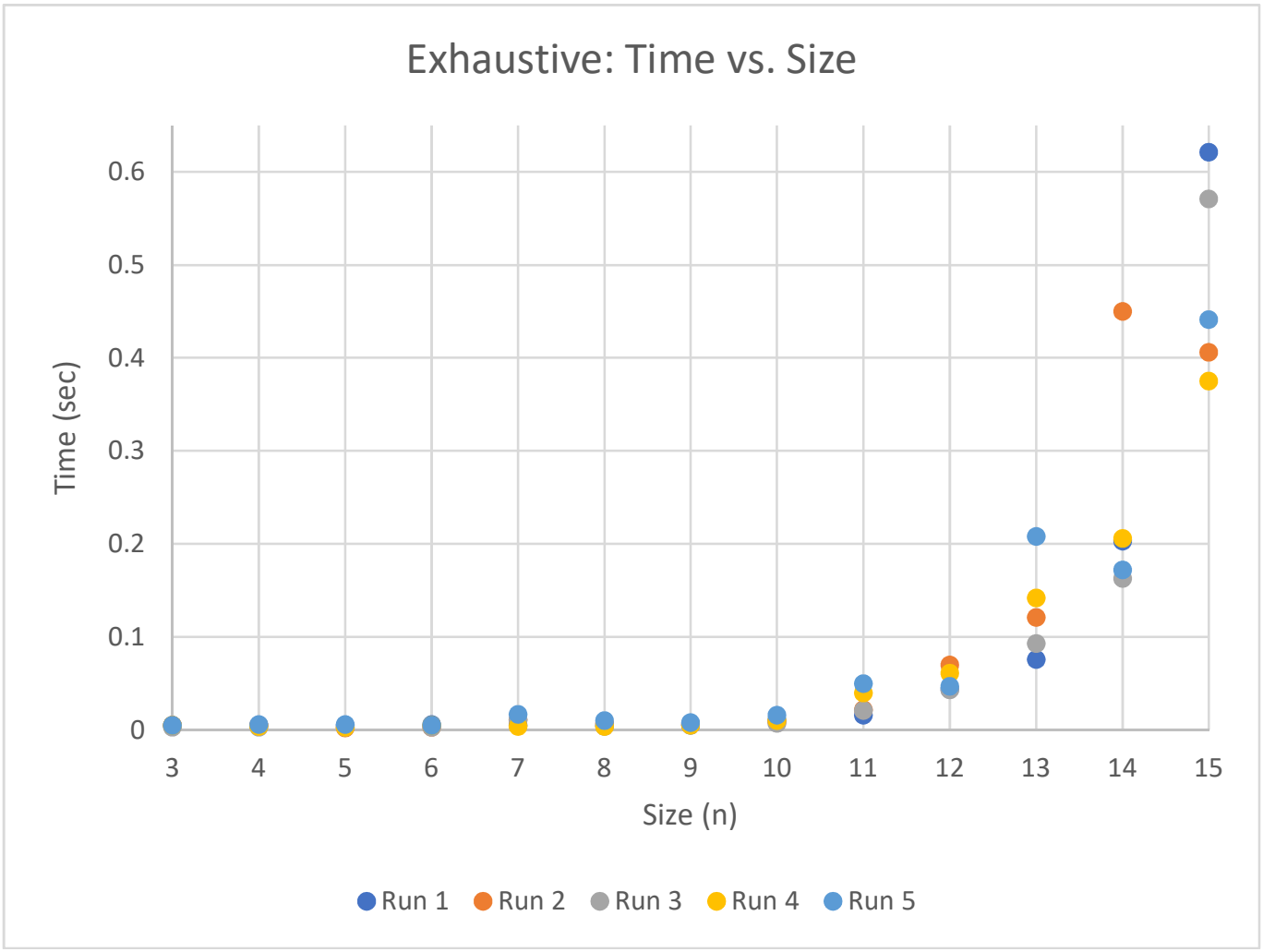
   Efficiency class:
   $T(n) = \sum_{i=0}^{2^n} \sum_{j=0}^{n-1} 1 = n(2^n + 1) \in \Theta(n2^n)$

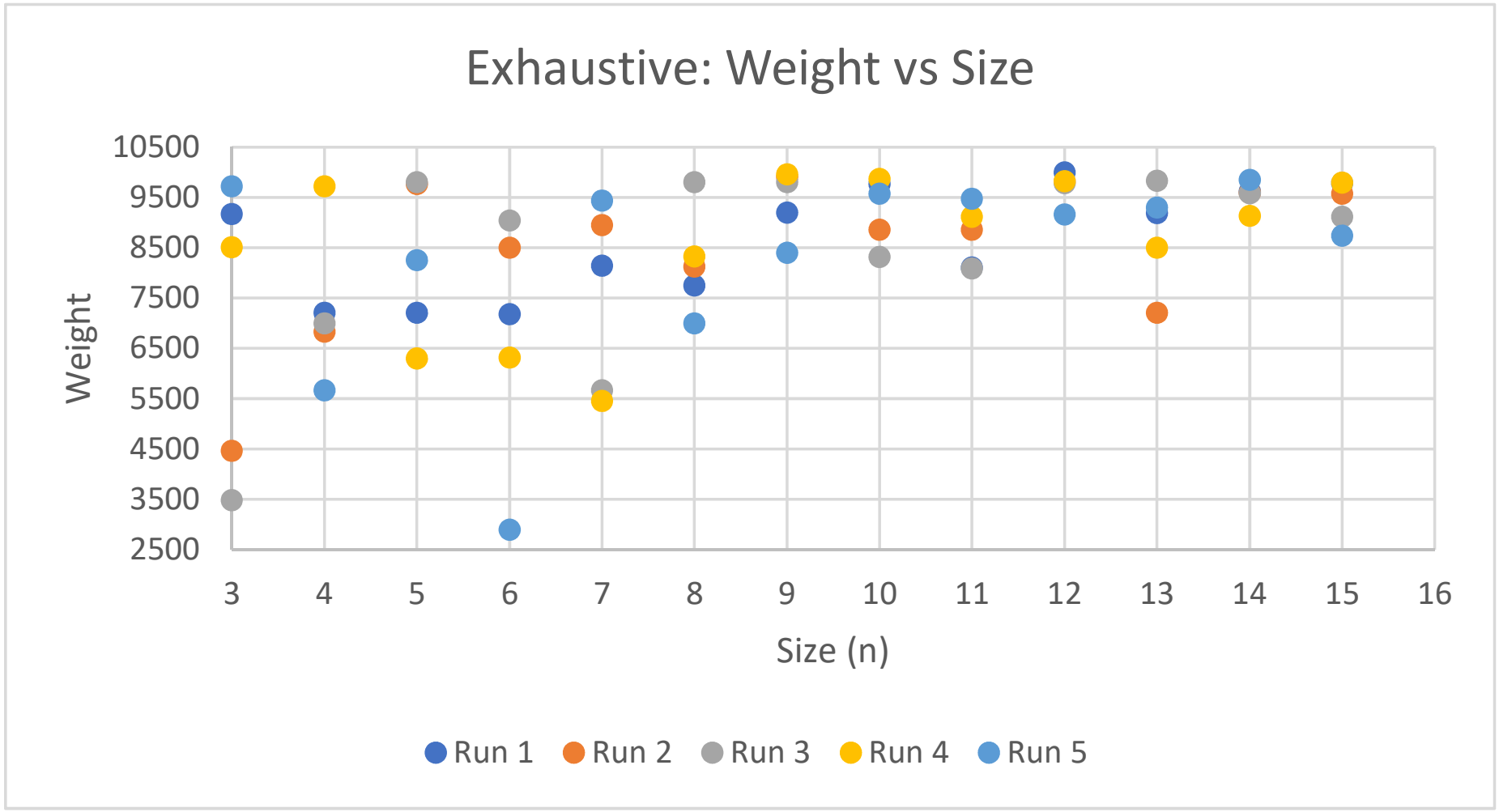
   (b) Greedy algorithm that selects next highest value item that fits in capacity left, and adds to K.

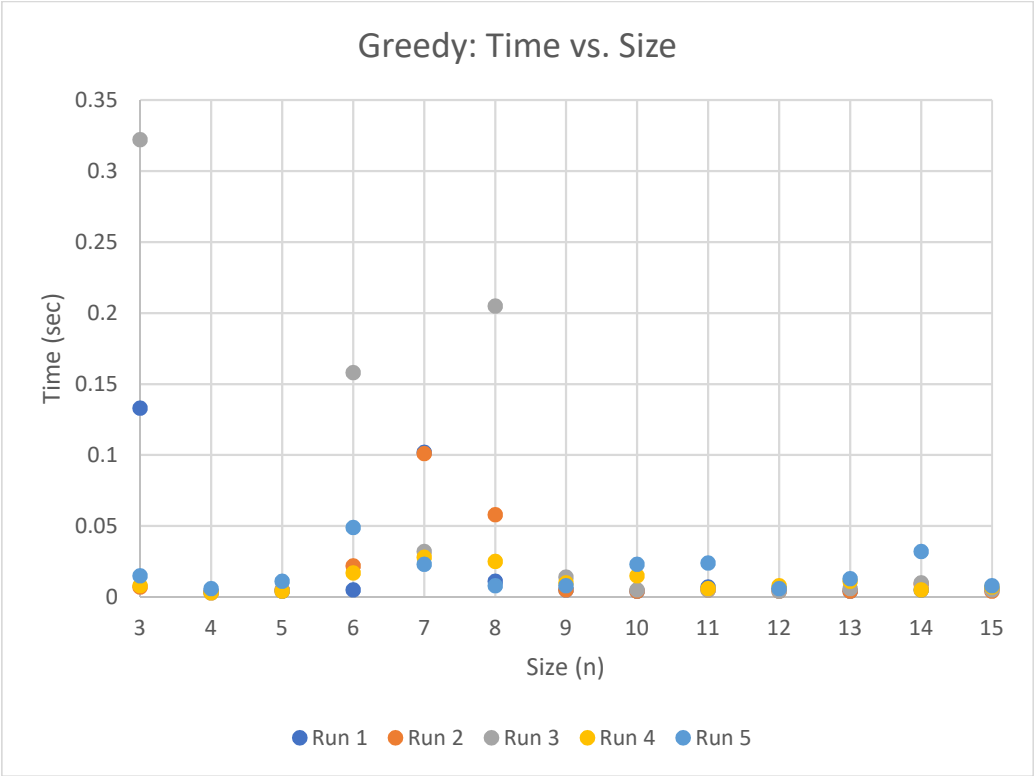
```
greedyKS(maxWeight, value[0,n-1], weight[0,n-1])
    currVal = 0
    currWeight = 0
    numVisited = 0
    highIndx = 0
    while currWeight < maxWeight and numVisited < n :
      highIndx = nextHighest(value[0,n-1], value[highIndx])
      tempWeight = currWeight + weight[highIndx]
      if tempWeight ≤ maxWeight
        currVal = currVal + value[highIndx]
        currWeight = currWeight + weight[highIndx]
        add highIndx to K
      numVisited = numVisited + 1
```

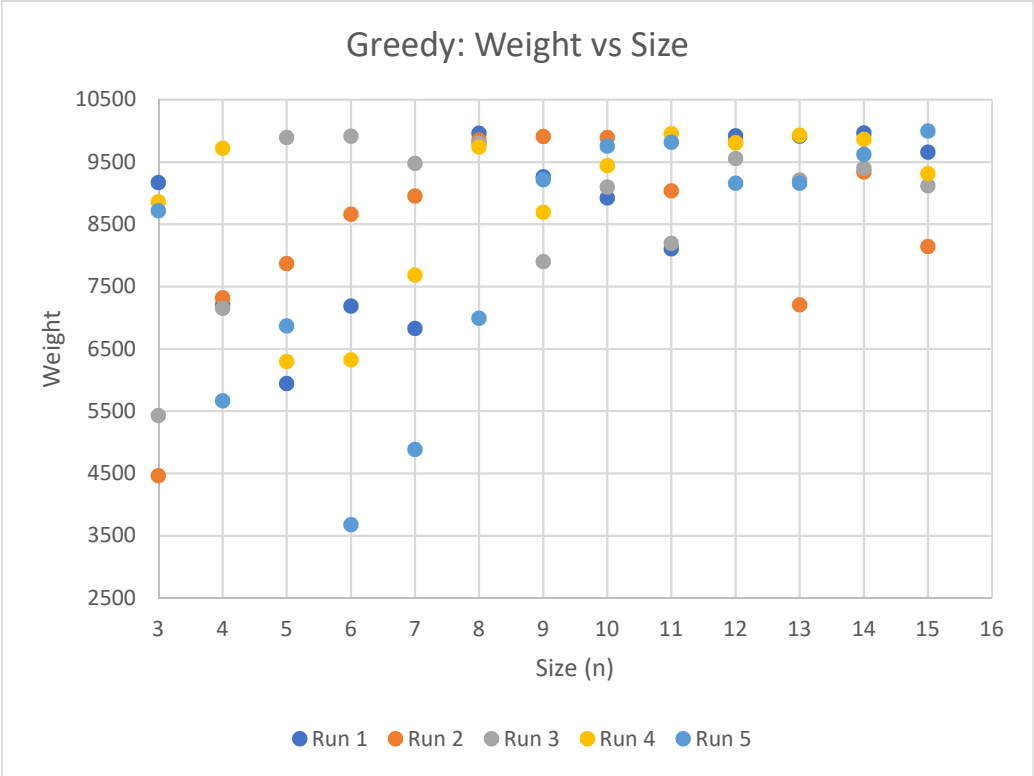
   Efficiency class:
   (nextHighest() $\in \Theta(n)$)
   $T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = n^2 \in \Theta(n^2)$

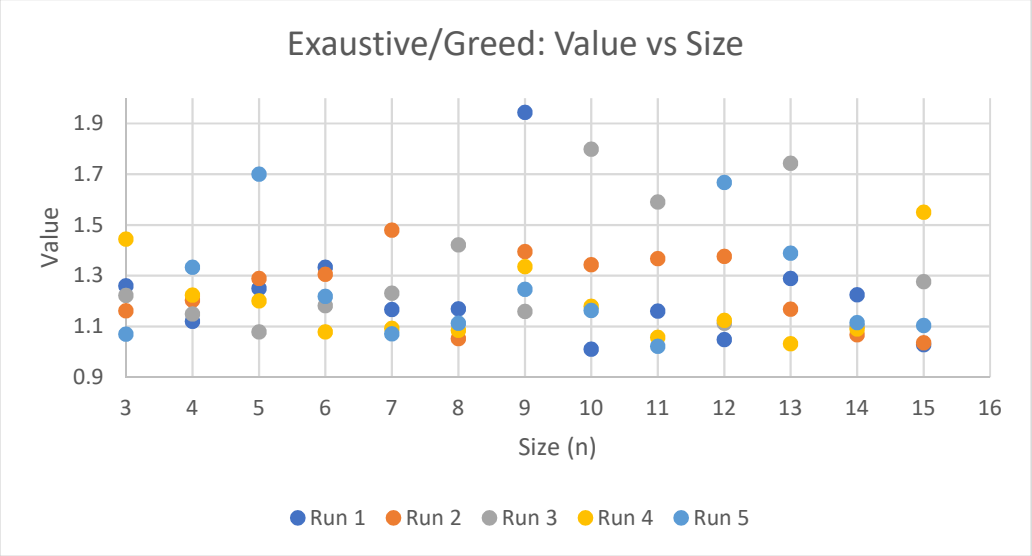
   (c) Greedy algorithm does not solve the problem by always finding the best set of items. For example, consider n=3: v[100, 75, 75], and w=[9000, 4500, 4500]. Exhaustive would pick the two 75 items each weighing 4500, for a total of 150 value and 9000 weight. The greedy algorithm would choose the 100 value for a total weight of 9000. They each have the same weight, but different values. Exhaustive would be better in this case.


Exhaustive: Time vs. Size
Time (sec)
0
0.1
0.2
0.3
0.4
0.5
0.6
3
4
5
6
7
8
9
10
11
12
13
14
15
Size (n)
Run 1
Run 2
Run 3
Run 4
Run 5


Exhaustive: Weight vs Size
Weight
10500
9500
8500
7500
6500
5500
4500
3500
2500
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Size (n)
Run 1
Run 2
Run 3
Run 4
Run 5


Greedy: Time vs. Size
Time (sec)
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
3
4
5
6
7
8
9
10
11
12
13
14
15
Size (n)
Run 1
Run 2
Run 3
Run 4
Run 5


Greedy: Weight vs Size
Weight
10500
9500
8500
7500
6500
5500
4500
3500
2500
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Size (n)
Run 1
Run 2
Run 3
Run 4
Run 5


Exaustive/Greed: Value vs Size
Value
1.9
1.7
1.5
1.3
1.1
0.9
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Size (n)
Run 1
Run 2
Run 3
Run 4
Run 5