Homework #2

2.4 1.b. $x(n) = 3x(n-1)$ for $n > 1$ $\quad x(1) = 4$

$$= 3[3x(n-2)] = 3^2 x(n-2)$$
$$= 3^2[3x(n-3)] = 3^3 x(n-3)$$
$$\vdots$$
$$3^i x(n-i)$$
$$\vdots$$
$$3^{n-1} x(n-1)$$
$$3^{n-1} x(1)$$
$$4 \; 3^{n-1}$$

1d. $X(n) = x(n/2) + n$ for $n > 1$ $\quad x(1) = 1$ solve for $n = 2^k$

$$= x(2^{k-1}) + 2^k$$
$$= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k$$
$$= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k$$

$$= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \dots + 2^k$$
$$= x(2^{k-k}) + 2^{k-k+1} + 2^{k-k+2} + \dots + 2^k$$
$$= x(2^{k-k}) + 2^1 + 2^2 + \dots + 2^k$$
$$= 1 + 2^1 + 2^2 + \dots + 2^k$$
$$= 2^{k+1} - 1$$
$$= 2 \cdot 2^k - 1$$
$$= 2n - 1$$

3.a. $S(n) = 1^3 + 2^3 + \dots + n^3$

basic operation: multiplication count per run is 2

$M(n) = M(n-1) + 2 = M(n-2) + 2 + 2 = M(n-3) + 2 + 4$

thus $M(n-i) + 2i$

$$M(1) + 2(n-1) = (1) + 2(n-1) = \boxed{2(n-1)}$$

b. $\sum_{i=2}^{n} 2 = 2\sum_{i=2}^{n} 1 = 2(n-2+1) = 2(n-1)$

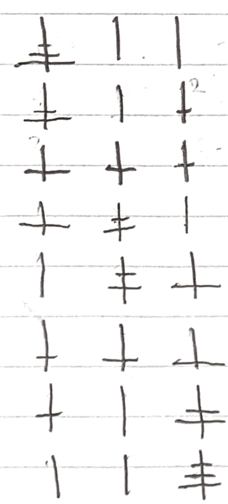
Thus the non recursive algorithm does the same number of basic operation executions, but doesn't carry the time and space overhead as the recursion algorithm.

5a. From slides: $M(n) = 2M(n-1) + 1$, $M(1) = 1$ 64 disks   1 disk move = 1 minute

$$M(n) = 2M(n-1) + 1$$
$$= 2(2M(n-2)+1)+1$$
$$= 2(2(2M(n-3)+1)+1)+1 = 2^3 M(n-3) + 7$$
$$\vdots$$
$$2^i M(n-i) + 2^i + 1 \quad \text{but } i = n-1$$

$$2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1}(1) + 2^{n-1} - 1$$
$$= 2^n - 1$$

So 64 disks will take $2^{64} - 1$ minutes

b. 

disk 1: ||||
disk 2: ||
disk 3: |

1, 2, 4 $i=3$

Thus 2^{i-1} moves are made by the ith largest disk

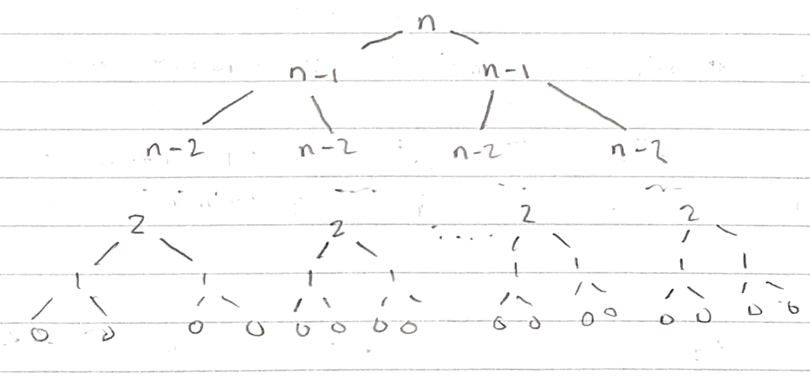
8a. Algorithm pow2 (n)

```
Algorithm pow2 (n)
    if n = 0
        return 1
    else
        return pow2 (n-1) + pow2 (n-1)
```

b. $A(n) = 2A(n-1) + 1$

$$2A(n-1)+1$$
$$2[2A(n-2)+1]+1$$
$$2^2 A(n-2) + 2 + 1$$
$$2^3 A(n-3) + 2^2 + 2 + 1$$
$$\vdots$$
$$2^i A(n-i) + 2^{i-1} + 2^{i-2} + \dots + 1$$
$$2^n A(n-n) + 2^{n-1} + 2^{n-2} + \dots + 1$$
$$2^n A(0) + 2^{n-1} + 2n-2 + \dots + 1$$
$$0 + 2^{n-1} + 2^{n-2} + \dots + 1$$
$$2^n - 1$$
$$= \Theta(2^n)$$

c.



$$\sum_{k=0}^{n-1} 2^k = \frac{1 \times 2^n - 1}{2-1} = 2n-1$$

$2^n - 1$ calls made by the algorithm

d. This is not a good algorithm for solving 2^n because exponential algorithms are really slow and this algorithm has $\in \Theta(2^n)$.

2.6

1. The comparison counter is in the wrong place is not inserted into the right place. It should be outside the while loop but in the for loop in order for it to properly count the number of comparisons. Right now it only counts if $A[j] > v$ which is when $A[j] > A[i]$, thus improper counting. The line should be if $j \geq 0$, count ← count + 1.

2a.

size of array	# of comparisons		
1000	257344	11 000	30072760
2000	999270	12 000	35864008
3000	2260493	13 000	41781465
4000	3987428	14 000	48767568
5000	6289971	15 000	56389520
6 000	9110606	16 000	63987275
7 000	12317141	17 000	71690311
8 000	15984220	18 000	81216637
9 000	20084416	19 000	90261674
10 000	24708232	20 000	99781418

b. After analyzing the data by pointing the data on an xy graph, the algorithm's average-case efficiency appears to be $\Theta(n^2)$ since the number of computation looks like an^2 with a being a constant. Using the graph tools, the constant a is approximately 0.25 so number of computations is roughly $0.25n^2$.

c. Using the estimate that the number of computations is about $0.25n^2$, an array of size 25,000 should have about 156274047 key comparisons.

3a.

size of array	milliseconds						
1000	117	6000	5901	11000	14605	16000	31429
2000	475	7000	13528	12000	18137	17000	33800
3000	1047	8000	19946	13000	21514	18000	38872
4000	2289	9000	22849	14000	25857	19000	46221
5000	3982	10000	18872	15000	28194	20000	65190

b. After analyzing the graph, the algorithm seems to have an order of growth of n^2. To be specific, the algorithm's run time for n size of array appears to be $t(n) = 1.31 \cdot 10^{-4} n^2$.

c. Using the estimate, an array of size 25000 should take about 81,875 ms $\approx$ 82 s to run.

4. $\frac{M(2n)}{M(n)}$

$$\frac{M(2000)}{M(1000)} = \frac{24303}{11966} \approx 2.031 \approx 2$$

$$\frac{M(4000)}{M(2000)} = \frac{53010}{24303} \approx 2.181 \approx 2$$

$$\frac{M(8000)}{M(4000)} = \frac{113063}{53010} \approx 2.133 \approx 2$$

$$\frac{M(6000)}{M(3000)} = \frac{78692}{39992} \approx 1.968 \approx 2$$

$$\frac{M(10000)}{M(5000)} = \frac{140538}{67272} \approx 2.089 \approx 2$$

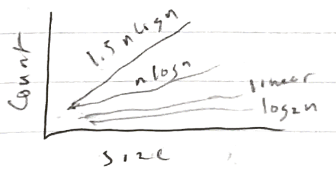
Right now we can assume the algorithm's efficiency class is either logarithmic or linear since it only slightly changes in ratio and the ratio converges to 2.

Checking by $\frac{M(n)}{g(n)} = \frac{M(1000)}{g(1000)} = \frac{11966}{\log 1000} \approx \frac{11966}{6.908} \approx 1732.194 \approx 1.5(1000) \approx 1.5n$

$$\frac{M(n)}{g(n)} = \frac{M(3000)}{g(3000)} = \frac{39992}{\log 3000} = \frac{39992}{8.003} \approx 4997.126 \approx 1.5(3000) \approx 1.5n$$

$$\frac{M(n)}{g(n)} = \frac{M(7000)}{g(7000)} = \frac{91274}{\log(7000)} = \frac{91274}{8.854} \approx 10308.787 \approx 1.5(7000) \approx 1.5n$$

After graphing the points, it is observed that the efficiency class appears to be roughly $1.5\, n \log n$ thus $\in \Theta(n \log n)$



3.1 4a. Algorithm Brute Force Polynomials $(P[0...n], x)$

```
p ← 0.0
for i ← n down to 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x
    p ← p + P[i] * power
return p
```

$$\sum_{i=0}^{n} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

b. Algorithm Linear Polynomial $(P[0...n], x)$

```
p ← P[0]
power ← 1
for i ← 1 to n do
    power ← power * x
    p ← p + P[i] * power
return p
```

$$\sum_{i=1}^{n} 2 = 2n \in \Theta(n)$$

c. It is not possible to design an algorithm with a better efficiency than linear efficiency for evaluating an arbitrary polynomial because it needs to process n+1 coefficients.

7a. Algorithm Stack of Fake Coins $(S[0...n])$

```
for i ← 0 to n do
    if (weight of coin from S[i] == 11 grams)    // pop one coin off stack [i]
        print S[i] contains counterfeit coins.
        break
```

$\sum_{i=0}^{n} 1 = n \in \Theta(n)$ is worst class efficiency since worse case is that the last stack is the one with counterfeits.

b. The minimum number of weightings needed to identify the stack with fake coins using the algorithm from a is n but you can get it down to $\log_2(n)$ by using binary divide and conquer. You divide the stack into two stacks. From there you measure the stack to see if it aligns with expected weight which should be $n/2$ times 10 grams. If one stack weighs more then you divide and conquer that stack till you find your self with one coin left. Keep track of number of divides so you can figure out which stack it came from.

8.
| E X A M P L E
A | X E M P L E
A E | X M P L E
A E E | M P L X
A E E L | P M X
A E E L M | P X
A E E L M P | X
A E E L M P X

9. Selection sort is not stable, because it does not preserve the relative order of any two equal elements due to the swapping of the min element with the first unsorted element.

11.
E ↔ X ↔ A M P L E
E A X ↔ M P L E
E A M X ↔ P L E
E A M P X ↔ L E
E A M P L X ↔ E
E A M P L E | X
E ↔ A M P L E | X
A E ↔ M ↔ P ↔ L E | X
A E M L P ↔ E | X
A E M L E | P X

A ↔ E ↔ M ↔ L E | P X
A E L M ↔ E | P X
A E L E | M P X
A ↔ E ↔ L ↔ E | M P X
A E E | L M P X
A ↔ E ↔ E L M P X
A E | E L M P X
A ↔ E | E L M P X
A | E E L M P X
| A E E L M P X

13. Bubble sort is a stable algorithm because it preserves the relative order of two equal elements by only swapping adjacent elements if a[j+1] < a[j]. The less than keeps it stable.

3.2 1a. $C_{worst}(n) = n+1$

b. $C_{avg}(n) = [1\frac{p}{n} + 2\frac{p}{n} + \ldots + i\frac{p}{n} + \ldots + n\frac{p}{n}] + (n+1)(1-p)$

$= \frac{p}{n}[1+2+\ldots+i+\ldots+n] + (n+1)(1-p)$

$= \frac{p}{\cancel{n}}\left(\frac{\cancel{n}(n+1)}{2}\right) + (n+1)(1-p)$

$\boxed{\frac{(2-p)(n+1)}{2}}$

4. n: 47 chars m: 6 chars

Worst case number of trials $= n - m + 1$

$= 47 - 6 + 1$

$= 42$

Total number of char comparisons $= (42 - 1$ (failed comparison) $\times 1 + (1 \times 2$ (successful comparison))

$= 41 + 2$

$= 43$

total number of comparison for GANDHI is $\boxed{43}$

8a. Algorithm Bruteforce_algorithm (S[0..n-1])

```
count ← 0
for i ← 0 to n-2 do
    if (T[i] = A) do
        for j ← i+1 to n-1
            if (T[j] = B) do
                count ← count + 1
return count
```

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} n-1-i+1$$

$$= \sum_{i=0}^{n-1} n-i$$

$$= (n + n-1 + n-2 + \dots + 2)$$

$$= (n + n-1 + n-2 + \dots + 1) - 1$$

$$= \frac{n(n+1)}{2} - 1$$

$$\frac{n(n+1)}{2} - 1 \in \Theta(n^2)$$

b. Algorithm Linear Brute Force (T[0,.. n-1])

```
count1 ← 0
count2 ← 0
for i ← 0 to n-1 do
    If (T[i] = A) do
        count1 ← count2 + 1
    IF (T[i] = B) do
        count1 ← count 1 + count2
return count
```