1. (a) SUM = $2^1 + 2^2 + \ldots + 2^{10} = 2046$. By special argument: This is binary 111 1111 1110. If we add 2 to it, it rolls over to 1000 0000 0000 = $2^{11} = 2K = 2048$, so it must be 2046.

By geometric series: SUM = $2^1 + 2^2 + \ldots + 2^{10} = 2^a (1 - r^n)/(1 - r)$ sum formula where here $r = 2, a=2, n=9$

so SUM = $2^a (1 - 2^{10}) / (1 - 2) = 2^a (1 - 1) = 2046$

(b) SUM = $2/3 + 4/9 + 8/27 + 16/81 + \ldots$

Need to subtract the “to infinity ...” part out.

$3/2 \cdot SUM = 1 + 2/3 + 8/27 + 16/81 + \ldots$ and therefore:

$3/2 \cdot SUM - SUM = 1 + (2/3 + 8/27 + \ldots - 2/3 - 8/27 - \ldots)$. The part in parentheses cancels out, we’re left with: 1/2SUM = 1. Thus, SUM = 2

2. A number $N$ has floor($log_2(N)$) + 1 binary digits, where floor(x) denotes the largest integer not greater than x. So:

floor($log_2(2^{100})$) + 1 = floor(100) + 1 = 101,

floor($log_2(5^{100})$) + 1 = floor(100 log_2(5)) + 1 = 232 + 1 = 233

floor($log_2(10^{100})$) + 1 = floor(100 log_2(10)) + 1 = 332 + 1 = 333.

How are these answers related? $2^{100} \times 5^{100} = 10^{100}$ and 101 + 233 = 334 = approx.333

3. We have $log_B(N) = log_2(N)/log_2(B)$ by Weiss, pg. 165, for any base B.

So $log_a(N) = log_2(N)/log_2(a)$ - relate base b to base 2

and $log_a(N) = log_2(N)/log_2(a)$ - relate base a to base 2

and thus $log_a(N)/log_a(N) = log_2(a)/log_2(b) = \text{const.}$

We have $log_{10}(N) = log_2(N)/log_2(10)$, and $1/log_2(10) = 0.3010$, so $log_{10}(N) = 0.3010 \times log_2(N)$

Number of decimal digits = ceiling($log_{10}(N)$) (within 1 of $log_{10}(N)$)

Number of binary digits = ceiling($log_2(N)$) (within 1 of $log_2(N)$)

So numbers in base 10 are about 3/10 length of the same numbers in base 2.

4. (a) Functions ranked in order of increasing growth rate. Most of these are easily decided by looking at the limit of the ratio as N grows. Numerical evidence isn’t really needed.

- $2/N = O(1/N)$ - does not grow at all, it shrinks as $N \rightarrow \infty$
- $38 = O(1)$ - constant
- $\sqrt{N} = N^a = O(N^b)$ when $a \leq b$ covers lots of cases
- $N$
- $N \log \log N$. log log N grows very slowly. It’s just 10 when $N = 2^{1000}$.
- $N \log N$. This and the next are tied
- $N \log (N^2) = 2N \log N$
- $N (\log N)^2$
• $N^{1.5}$
• $N^2$
• $N^2(\log N)$
• $N^3$
• $2^{(N/2)}$
• $2^N$ this is **not** a tie with $2^{(N/2)}$

(b) $\log N$ and $\log(N^2)$ are tied because $\log(N^2) = 2 \cdot \log N$. $\log^2 N = \log N \cdot \log N$ grows faster than $\log N$. You can divide the two and get $\log N >> const$.

5. (a) The outer loop is executed $n$ times, and the inner loop, in the worst case, is also executed $n$ times giving time complexity of $O(n^2)$.

(b) In terms of big-O both functions are $O(N^2)$. When absolute time (in seconds) is considered the second version could be a bit faster since we are doing fewer multiplications. It is possible however that due to compiler optimizations both versions will be equally fast.

(c) ```c
int mysterySum(int n)
{
    int i, j, s=0;
    for(i=0; i<n; i++)
        s += i*i*i;
}
```

And the time complexity is $O(n)$.

(d) The term is $\text{sum}(i^3) = (n * (n+1)/2)^2$. Proof by induction: True for 1 (result is 1) and 2 (result is $9 = (2 * 3/2)^2$). Suppose it’s true for $n-1$, that is - $\text{sum}(i-1)^3 = (n * (n-1)/2)^2$. Let’s show that by adding $i^3$ we’ll get $(n * (n+1)/2)^2$. $(n * (n-1)/2)^2$ opens to $(n^4 - 2n^3 + n^2)/4$. $(n * (n+1)/2)^2$ opens to $(n^4 + 2n^3 + n^2)/4$. Subtracting the former from the latter gives us $4 * n^3/4 = n^3$. Exactly what we said we had to add. QED.

6. (a) Calculating $\text{power2}$

```c
int power2(int n)
{
    if (n==0) return 1;
    return power2(n-1)+power2(n-1);
}
```

We have a double recursion here, so obviously the runtime is bad. The formula is $T(n) = C + 2 * T(n-1)$. So each run doubles the number of recursive calls while doing constant time operations outside the recursions. This is exponential, so $T(N) = O(2^N)$. It can be shown easily by expanding the next expression $T(n-1) = C + 2T(n-2)$, etc. The factors of 2 accumulate so that at the $k^{th}$ stage we have $2^k \cdot C$. Overall we have $n$ stages.

A linear time algorithm would be the following:

```c
int fib(int n)
{
    if (n==0) return 1;
    return 2*power2(n-1);
}
```

We simply run the recursion once and multiply by 2... You can easily show that the recurrence formula is linear, since it’s the same as the factorial function we showed in class: $T(n) = C + T(n-1)$.

(b) **Bonus:**
```java
int power2New(int n)
{
    if (n==0) return 1;
    if (n % 2 == 0) {
        int result = power2New(n/2);
        return result*result;
    }
    else
        return 2*power2New(n-1);
}
```

Explanation: If n is even, then $2^n = (2^{n/2})^2$, so we can calculate $2^{n/2}$ recursively and square, cutting half of n in one move, which is similar to what we do in a binary search. So if we could always do that, the runtime formula would be the same as binary search: $T(n) = T(n/2) + C$, which is logarithmic. However, if n is odd we resort to the previous method by subtracting one from n. We showed it is linear... So why is the overall runtime logarithmic despite the fact that we may have to run some “linear” stages? The answer lies in the following observation: If n is even, we divide it by 2. The next stage we get $n/2$ which may be even or odd. If n is odd, we subtract 1 and get $n-1$ which is always even! In other words – we have an odd n at most one half of the times, so the runtime is at most twice as much as the “optimal” scenario when n is always even, hence $2\cdot O(\log n) = O(\log n)$.

7. Code for combinational lock. Note there are no getters or setters for a, b, c, since the combination is “secret”. To change the combination, you need to supply the old combination.

```java
/*
* CombinationLock.java
* @author -- Ruchi Dubey
* ruchi@cs.umb.edu
*/
class CombinationLock {
    private int a,b,c;

    CombinationLock(int a1,int a2, int a3)
    {
        a = a1;
        b = a2;
        c = a3;
    }

    public boolean open(int x,int y, int z)
    {
        return x == a && y == b && z == c;
    }

    /* The old combination is xyz and the new to be changed is pqr */
    public boolean changeCombo(int x, int y, int z, int p, int q, int r)
    {
        if(this.open(x, y, z))
        {
            a = p;
            b = q;
            c = r;
            return true;
        }
        else
            return false;
}
```
8. Here is a suggested class:

```java
public class WordUsage {
    private final String word;
    private int count;

    public WordUsage(String x, int count) {
        word = x;
        this.count = count;
    }

    public WordUsage(String x) {
        word = x;
        this.count = 1;
    }

    public void setCount(int x) {
        count = x;
    }

    public String getWord() {
        return word;
    }

    public int getCount() {
        return count;
    }

    public void increment() {
        count = count + 1;
    }
}
```

9. (a) A final class is a class that cannot be extended or subclass-ed. Final classes are used for several reasons: a. Speedup (can be more efficient code). b. To prevent accidental overriding of classes that the programmer wishes to leave as is.

(b) An interface provides an API, a set of method descriptions, but no implementation whatsoever of that API. It also can specify constants and interfaces (see Map.java, pp 237-238 for an example of interface inside an interface). An abstract class may provide a default implementation by specifying some non-abstract methods, and can have fields. The interface is not allowed to provide any implementation details either in the form of data fields or implemented methods. Only public final fields and abstract public members may be in an interface, plus nested interfaces such as Entry in Map. The big advantage of interface over abstract class is that another class X can implement several interfaces but can extend only one class, abstract or not. Thus we use abstract classes only as base classes for important concrete classes we want.