

Solution to CS310 HW1

Thanks to Andrew Zwicker, Ruchi Dubey, Joe Flaherty and Prof. Betty O'Neil for parts of this solution.

Homework problem 1

a) $SUM = 2^1 + 2^2 + \dots + 2^{10} = 2046$

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:

$$SUM = 2^1 + 2^2 + \dots + 2^{10} = 2*(2^0 + \dots + 2^9)$$

$$SUM = a*(1 + r + r^2 + \dots + r^n) = a*(1 - r^{(n+1)})/(1-r) \text{ sum formula}$$

where here $r = 2, a=2, n=9$

$$\text{so } SUM = 2*(1 - 2^{10})/(1-2) = 2*(1K-1) = 2046$$

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

Need to subtract the "to infinity ..." part out.

$$3/2 * SUM = 1 + 2/3 + 8/27 + 16/81 + \dots \text{ and therefore:}$$

$$3/2 * SUM - SUM = 1 + (2/3 + 8/27 + \dots - 2/3 - 8/27 - \dots)$$

Part in parentheses cancels out, left with:

$$1/2 * SUM = 1$$

$$\text{thus, } SUM = 2$$

=====

Homework problem 2

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

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

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

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

How are these answers related?

$$2^{100} * 5^{100} = 10^{100} \quad \text{and } 101 + 233 = 334 \approx 333$$

=====

Homework problem 3

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

So $\log_b(N) = \log_2(N)/\log_2(b)$ --relate base b to base 2

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

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

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

$$\text{so } \log_{10}(N) = 0.3010 * \log_2(N)$$

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

Number of binary digits = $\text{ceiling}(\log_2(N))$ (within 1 of $\log_2(N)$)

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

=====

Homework problem 4 - Weiss, problem 5.14

Functions ranked in order of increasing growth rate - in fact there are no ties on the list.

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 is 2^{1000} .

$N \log N$ this and the next are tied

$N \log(N^2) = 2 N \log N$

$N(\log N)^2$

$N^{1.5}$

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 $\log(N^2) = 2 \log N$. $\log^2 N = \log N \log N$ grows faster than $\log N$. You can divide the two and get $\log N \gg \text{const}$.

=====

Homework problem 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 less multiplications. It is possible however that due to compiler optimizations both versions will be equally fast.

C) `int mysterySum(int n)`

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

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

D) The term is $\sum(i^3) = (n*(n+1)/2)^2$.

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 $\sum(i-1)^3 = (n*(n-1)/2)^2$. 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.

=====
Homework problem 6 – bonus (assume n non-negative)

```
int logPower2(int n)
{
    if (n == 0) return 1;
    // Separate treatment of even and odd numbers
    if(n % 2 == 1) return 2*logPower2(n-1);
    else {
        // if n is even we can calculate power of n/2 and square
        int result = logPower2(n/2);
        return result*result; // Watch out from double recursion!!!
    }
}
```

This is a logarithmic run-time algorithm because when n is even, we call the function with $n/2$, and therefore the runtime analysis would be $T(n) = T(n/2) + C$ (the constant is the if statement and the squaring of the result). This gives a logarithmic runtime analysis. If n is odd, the formula is $T(n) = T(n-1) + C$. This is linear, but the overall algorithm is still logarithmic because it is guaranteed that at least every other call will have an even n . So every 2 recursive calls are bound to reduce the input size by $1/2$. So the number of calls is bound from above by $2*\log(n)$.

Homework problem 7 – fibonacci series

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

We have a double recursion here, so obviously the runtime is bad. The formula is $T(n) = C + T(n-1) + T(n-2)$. So each run almost doubles the number of recursive calls. This is exponential. This is “slightly better” than the exponential power2 shown in class, but still bad. The exact formula is beyond the scope of this class, but it is approximately $T(N) = O(1.6^N)$. A linear time algorithm would be the following:

```
int fib(int n)
```

```

{
    if(n == 0) return 0;
    int prev = 0;
    int cur = 1;
    int sum = 1;
    for(i=2;i<=n;++i) {
        sum = cur+prev;
        prev = cur;
        cur = sum;
    }
    return sum;
}

```

It is linear because the loop runs $O(n)$ times (that is, $n-2$ times) and each iteration performs a constant number of operations.

=====
Weiss 3.10

a) "q" is legal because it is a Person variable defined inside the main function. Accessing p is illegal because you cannot make a reference from a static method (main) to a non-static variable, p. If the name field in the Person class were private line 18 would be illegal as well. We'd have to provide a get function to access a private field.

b) Line 20 is legal: It is ok to access a public constant of q.

Line 21 is not legal: It is not ok to access a private field of p...We need to create a getter method to get the value of SSN.

Line 22 is legal: It is legal because name has "package" visibility the default access.

Line 23 is legal: It is ok to access a public constant inside a class like this...This is similar to how one can use the Math.xxx functions.

Line 24 is not legal: It is not ok to access a private field outside of the scope of the Person class. Person class can not access non-static field.

Sealed-up name case: Now line 22 is not legal.

=====
Weiss 3.13

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.

```

/*
 *CombinationLock.java
 *Class for exercise 3.13 page 88 of textbook
 *@author- Ruchi Dubey
 *ruchi@cs.umb.edu
 */

class CombinationLock
{

```

```

private int a,b,c;
String msg;

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;
    }
}
}

```

=====

Weiss 4.6

Consider the program to test visibility in fig. 4.46

- a) The illegal accesses are: b.bPrivate and d.dPrivate.
- b) Make main a method in Base. Which accesses are illegal? Only d.dPrivate is illegal. b.bPrivate is now accessible since this code is now in the Base class and so is the private member involved, and is properly associated with an object. (We can't access "bPrivate" directly, because we're in static code and bPrivate is not static, so needs an object.)
- c) Make main a method in Derived. Which accesses are illegal? b.bPrivate.
- d) How do these answers change if protected is removed from line 4? No change because these classes are all in the default package. Protected members are accessible to derived classes and all classes in the package. Removing protected just means that the member has package access which is what it had before, so no change. Removing protected does change accessibility from code in another package (none shown here).
- e)

```

public class Base
{
    public Base(int bPublic, int bProtected, int bPrivate)
    {
        this.bPublic=bPublic;
        this.bProtected=bProtected;
        this.bPrivate=bPrivate;
    }
}
public class Derived extends Base
{
    public Derived(int bPublic, int bProtected, int bPrivate,
                  int dPublic, int dPrivate)
    {
        super(bPublic,bProtected,bPrivate);
        this.dPublic=dPublic;
        this.dPrivate=dPrivate;
    }
}

```

f) bPublic, bProtected, dPublic, dPrivate are accessible to the class Derived.

g) bPublic, bProtected.

=====

Weiss 4.7

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.

=====

Weiss 4.9

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.