Data Structures and Algorithms Homework 1

Due Wednesday Sept 4; Joseph Sepich (jps6444)

1 Problem 1

I understand the course policies.

2 Problem 2

2.1 Part a. Prove that if a and b are even, then gcd(a, b) = 2gcd(a/2, b/2)

1. By the definition of even we can state that a = 2n and b = 2m, where n and m are both integers.
2. Using the definition in step 1 we can write gcd(a, b) = gcd(2n, 2m).
3. Since 2 is a common divisor we can also write gcd(2n, 2m) = 2gcd(n, m).
4. Using step 1, we also know that n = $\frac{a}{2}$ and m = $\frac{b}{2}$.
5. Plugging step 4 into the equalities in step 2 and 3 we get gcd(a, b) = gcd(2n, 2m) = 2gcd(n, m) = 2gcd($\frac{a}{2}$, $\frac{b}{2}$).

Therefore if a and b are both even, gcd(a, b) = 2gcd($\frac{a}{2}$, $\frac{b}{2}$).

2.2 Part b. Prove that if a is even and b is odd, then gcd(a, b) = gcd(a/2, b)

1. By the definition of even we can state that a = 2n, where n is an integer.
2. Using the definition in step 1 we can write gcd(a, b) = gcd(2n, b).
3. Since b is odd, it cannot be divided by 2, so the 2 in the term 2n is unnecessary information (cannot contribute to the gcd). We can then write gcd(a, b) = gcd(n, b).
4. Using step 1, we also know that n = $\frac{a}{2}$.
5. Plugging step 4 into the equalities in step 2 and 3 we get gcd(a, b) = gcd(2n, b) = gcd(n, b) = gcd($\frac{a}{2}$, b).

Therefor if a is even abd b is odd, then gcd(a, b) = gcd($\frac{a}{2}$, b).

2.3 Part c. Prove that if a and b are both odd and a >= b, then gcd(a, b) = gcd((a-b)/2, b)

1. By the definition of odd we can state that a = 2n + 1 and b = 2m + 1, where n and m are both integers.
2. a - b = (2n + 1) - (2m + 1) = 2n - 2m + 1 - 1 = 2(n-m). This must be greater than 0, since a >= b.
3. a - b is therefore by definition an even number since 2(n - m) can be written as 2x = (a-b) where x is the integer n - m.
4. Definition of gcd means that d | a and d | b, where d is an integer. This means $a = d * z$ and $b = d * y $ where z and y are integers.
5. Through step 4 b - a = d(z - y), so d must also divide the integer (z - y). This means gcd(a, b) = gcd(a-b, b).
6. Since we determined in step 3 a-b is even (and b is even) and in step 5 gcd(a, b) = gcd(a-b, b), then through the proof in part b of the problem we can conclude that gcd(a, b) = gcd($\frac{a-b}{2}$, b).

2.4 Part d.

We know that by the defintion of gcd

```
int gcd(int a, int b) {
    // input a >= b
    int d = 1;
    if (a == b) return a;
    bool a = isEven(a);  // test parity of variables
    bool b = isEven(b); // unit time
    while (a > 0 && b > 0) {
      if (a is odd and b is even) { // 2 is not a common divisor (part b)
        a = a / 2;
      } else if (a is even and b is odd) { // 2 is not a common divisor (part b)
        b = b / 2;
      } else if (a is odd and b is odd) { // part c
        a = (a - b) / 2; // our input requires a > b
      } else { // both a and b are even, so both can be divded by 2 (as in part a)
        a = a / 2;
        b = b / 2;
        d += 1;
      }
    }
    // d is how many times divided by 2
    // a is non even part of gcd
    return a * 2^d;
}
```

Now let us asses running time. We are assuming testing parity and halving are in unit time, so let's focus on subtraction. As we can recall a positive integer a has at most log(a) bits, and here a is the larger number. Subtraction two n bit integer taskes O(n) time, so here it would take us O(log(a)) time. Therefore the algorithm meets the requirements of the problem.

3 Problem 3

a) $f(n) = \Omega(g(n))$, this is true because we know with polynomials, the higher degree will always grow faster.

b) $f(n) = \Theta(g(n))$, this is true because $2^{n-1} = 2 * 2^n$ and we know that coefficients do not make a difference in larger values of n.

c) $f(n) = \Omega(g(n))$, this is true because f(n) has an exponent which grows, while g(n) has a constant exponent, therefore f(n) must grow faster.

d) $f(n) = \Omega(g(n))$, this is true because in g(n) the 2^{n} term is dominant. While 2^{n} and 3^{n} both have the same exponent term of n, the integer that has the exponent is greater in f(n).

e) $f(n) = \Theta(g(n))$, this is true because you can transform the exponent on each into a coefficient, due to properties of logarithms, then change the c you put in front of g(n) to obtain an identical function.

f) $f(n) = \Omega(g(n))$, this is true because f(n) is growing a constant rate, while g(n) is growing at less than a constant rate.

g) $f(n) = O(g(n))$, this is true because for each additional n, f(n) is multiplied by 2, but g(n) is multiplied by n, so it must be growing faster.

h) $f(n) = O(g(n))$, this is true because by the logarithm properties you are comparing $nlog(e)$ and $nlog(n)$, and since we don't look at coefficients for growth rates you are comparing n and $nlog(n)$, so while f(n) grows at a constant rate, g(n) is a function that increases at an increasing rate.

i) $f(n) = \Theta(g(n))$, this is true because n in each equation is the dominating term. This makes both equations $\Theta(n)$, since log(n) grows more slowly than n.

j) $f(n) = \Theta(g(n))$, this is true because of the same concept in the last part. n grows faster than both log(n) and $\text{n}^{0.5}$. If you chose c to be 5, then they would be identical equations.

4 Problem 4

4.1 Part a.

1. Recall we talked about multiplication in lecture. Using our standard multiplication algorithm it takes $\Theta(n^2)$ time. This is because for each new digit, another row is added from having to multiply another digit with the second number. You also then have to add up the new parts in the addition part of the algorithm.
2. Since we are summing up 1 through n, we are doing the multiplcation (squaring step) from step 1 n times. This means we are going to be performing n*n^2 operations.

Therefore the running time must be $\Theta(n^3)$.

4.2 Part b.

1. For this statement we can expand on the previous proof.
2. Agains we are doing n operations of k^{j}.
3. k^{j} must be $\Theta(n^j)$, because performing a multiplication of x and x (to be y) will be n^2.
4. Once you have the answer to x andx from step 3, you can say that if you have k^{j} where j is a multiple of two, you can dececrease it to (x and x)$^{\mathrm{j}/2}$, and so on. You can do this until you have to multiply by x again, if the exponent is odd. This would make it so you have n^2 running time, making it j operations.
5. If we combine step 4 and 2 we can conclude that the summation will take $\Theta(n^{j+1})$ running time.

4.3 Part c.

1. The summation in this problem is also known as the harmonic series.
2. We know that the integral of $\frac{1}{x} = ln(n)$, so we can use this to show that the partial sum of the series is bound by ln(n).
3. The sum of $\frac{1}{x+1} \leq \int \frac{1}{x} dx = ln(n) \leq \Sigma \frac{1}{x}$.

Therefore the harmonic series must be bound as $\Theta(log(n))$.

4.4 Part d.

1. By definition $log(n!) = log(1) + log(2) + ... + log(n)$.
2. Upper bound we know that $log(n!)$ as written above $\leq log(n) + log(n) + log(n) = n * log(n)$.
3. The partial sum of $log(n!) = log(n/2) + ... + log(n)$ is then $\leq log(n!)$, which we know by definition of partial sum.
4. We can also say the given partial sum $\geq log(n/2) + log(n/2) + ... + log(n/2) = n/2 * log(n/2)$, which follows the $n * log(n)$ structure.

Therefore since $log(n!)$ is upper and lower bounded by $nlog(n)$ it must be $\Theta(n * log(n))$.

StackOverflow was referenced to completed this problem.

4.5 Part e.

1. In the summation, c is a constant, where i (its exponent) is being incremented in the summation.
2. If c $>$ 1, the series is increasing and follows the proof in part b of this problem. By the statement in part b the running time must be $\Theta(c^k)$, with no + 1, since the series starts at 0.
3. If c == 1, the running time is linear, since one is merely summing up 1, k times.
4. If c $<$ 1, then c is will create a similar series to the harmonic series, but the series will be decreasing and convergent, so one knows the summation equation to sum the series, making it unit time.

5 Problem 5

5.1 1.

- An outer loop does n iterations
- An inner loop does (n-i) / 5 iterations

The second bullet point is true, since j is initialized to be i, and is incremented by 5 each time. These two statements imply a running time of n * (n-i) / 5 = $\frac{1}{5}n^2 - ni$. We know that in this running time the n^2 term dominates, so it is $\Theta(n^2)$.

5.2 2.

- An outer loop does n iterations
- An inner loop does (n-4i) iterations

The second bullet point is true, since j is intialized to 4i, but still increments only 1 every loop. Since it is intialized to 4i, it will not run those 4i times. These two statements imply a running time of $n * (n - 4i) = n^2 - 4ni$, which is a running time of $\Theta(n^2)$, with the dominant n^2 term.

5.3 3.

- An outer loop does n iterations
- An inner loop does n-i^5 iterations

You could flip the algorithm to have the same running time where the inner loops says:

```
j := i^5;
while j < n do
  j := j + 1;
end
```

This means we once again have $\Theta(n^2)$, since the running time is $n * (n - i^5) = n^2 - ni^5$.

5.4 4.

- An outer loop does n iterations
- An inner loop does $\frac{log_2(n)}{4}$ iterations.

The second bullet point is true, because j starts at 2, increments by itself to the 4th, but only goes until i. This means it will go from 2 to 16 to 256 to 4096 and so on. These two statements imply $n * \frac{log_2(n)}{4} = \Theta(nlog(n))$