CS310: Advanced Data Structures and Algorithms

Fall 2022 Assignment 1

Due: Friday, Sep. 23, 2022 on Gradescope

Goal

Review of runtime analysis, recursion and Java

Questions

1. You should learn to recognize and sum a geometric series. Try these:
   Note: You don't necessarily have to know exactly how to solve these equations. It's enough if you look it up in your calculus text or online. The important thing for me is that you know how to look these things up and understand how they generalize.

   (a) $\sum_{i=1}^{n=10} 2^i$.

   $$\sum_{i=1}^{n=10} 2^i = \sum_{i=0}^{n=10} 2^i - 1 = 2^{n+1} - 2 = 2^{11} - 2 = 2046$$

   (b) $\sum_{i=1}^{\infty}(2/3)^i$.

   $$\sum_{i=1}^{\infty}(2/3)^i = \frac{1}{1-\frac{2}{3}} = 3$$

2. How many binary digits are there in the numbers 2^{100}, 5^{100} and 10^{100} ? How are the answers to these three questions related? (**Hint:** this is a question about logarithms and change of base.)

   For this question, we can use the logarithm property of change of base to convert each of the values. For a decimal number n, we can find the number of digits x in its base b representation with the following equation:

   $$x + 1 = \frac{\log(n)}{\log(b)}$$

   For all the numbers in equation, we want to find the number of digits in the binary (base b = 2) form. For 2^{100}, we get 101 binary digits. The equation above would result in 100 but we can see that the base of the n value is the same as the binary form base. From this observation, we can say that the binary representation of 2^1 has the 2 binary digits `10` (1 more digit than the exponent). Therefore, we can add 1 to the result of the equation above and get 101 digits. We can use the formula for 5^{100} and 10^{100} after which we get 233 and 333 digits, respectively.

3. (a) Show that $\log_a(x) = c \times \log_b(x)$ for some constant c (expressed only in terms of the constants a and b). **Hint:** This is probably easier than you think... It follows quite directly from the log properties. You should, though, prove that it's true for any a and b and not prove by an example.

   In this proof, we will utilize the change of base property of logarithms.

   $$\log_a(x) = c \times \log_b(x)$$

$$\log_a(x) = c \times \frac{\log_d(x)}{\log_d(b)}$$
$$\frac{\log_d(x)}{\log_d(a)} = c \times \frac{\log_d(x)}{\log_d(b)}$$
$$\frac{\log_d(x)}{\log_d(a)} \times \frac{\log_d(b)}{\log_d(x)} = c$$
$$\frac{\log_d(b)}{\log_d(a)} = c$$

In this proof, d is another arbitrary constant. Because of the properties of logarithms in a change of base, this arbitrary value doesn't affect the value of c (as long as the bases in the logarithms in the numerator and denominator are the same).

(b) Calculate the ratio between the number of digits required to write a number in base 10 and the number of digits required to write the same number in base 2. Notice that this question relates to question 2 and 3a above.

Using the equation derived in the previous question, we can plug in the bases 2 and 10 for a and b respectively. The final ratio is:

$$\frac{\log(2)}{\log(10)} \approx 0.3$$

4. (a) Order the following functions by growth rates:
N, $\sqrt{N}$, $N^{1.5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, 2^N, $2^{N/2}$, 37, N^3, $N^2 \log N$.
Indicate which functions grow at the same rate.

The order of the functions by growth rate is as follows: 2^N, N^3, $2^{\frac{N}{2}}$, $N^2 \log(N)$, N^2, $N^{1.5}$, $N \log^2 N$, $N \log(N^2)$, $N \log N$, N, $N \log \log N$, $\sqrt{N}$, 37, and $2/N$.

(b) Rank the following three functions: $\log N$, $\log(N^2)$, $\log^2 N$. Explain.

The ranking of the three functions are as follows: $\log^2 N$, $\log(N^2)$, and $\log N$. $\log(N^2)$ should have a higher growth rate than $\log N$ due to the exponent on N inside the log. Graphing $\log^2 N$ on a graph shows that its growth rate is higher than $\log(N^2)$ which defined the previously mentioned order.

You should find all the mathematics you need in the class notes and in Kleinberg and Tardos, chapter 2. You may find it useful to remember that one way to compare the relative growth rates of $f(n)$ and $g(n)$ is to look at the ratio $f(n)/g(n)$ as $n \to \infty$. If that ratio approaches 0, then g grows faster than f: $f(n) = O(g(n))$. If it approaches infinity then f grows faster than g. If the ratio approaches a constant different from both 0 and ∞ than f and g grow at the same rate.

5. (a) Find a big-O estimate for the running time (in terms of n) of the following function (with explanation)

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

This function's runtime is $O(n^2)$. The outer loop will run n times while the inner loop would also run n times. Therefore, combining these two runtimes results in a $O(n^2)$ time complexity.

(b) Is this version of mysterySum faster? Is the big-O analysis different?

```
int mysterySum1(int n) {
  int i, j, s=0;
  for(i=0; i < n; i++) {
    int i2 = i*i;
    for(j=0; j < i; j++) {
      s += i2;
    }
  }
}
```

This method does the same thing as the previous method with the only difference being that the mathematical operation is done separately to the increment of the sum. This operation runs at constant runtime so it must be the same as the previous method.

(c) Replace the inner loop in mysterySum by an O(1) expression and compute the running time of the new program.

Replacing the inner loop with a constant runtime expression would simplify the function to a linear runtime, $O(n)$ time complexity. Running sample code in the IDE and running it, we see that the function is a sum of cubes. So we can replace the inner loop with `s += i*i*i`.

(d) Find a single O(1) expression giving the same result. **Hint:** Evaluate the function by hand (or compile and run the code) for a view values of n and try to see the pattern
Notice: You have to find the mathematical formula, not a piece of code. Start with the expression you derived in part c above (hint: It is a series) and find the sum of the series any way you want (including looking it up).

As previously mentioned, running sample values on this method in an IDE shows that the method is a sum of cubes. The formula for the sum of cubes is:

$$\sum_{i=0}^{n} i^3 = \frac{n^2 (n+1)^2}{4}$$

6. The following program computes 2^n:

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

(a) Find a recurrence formula as we learned in class. Find the runtime. What is the big problem with this function? (hint: We discussed something similar in class).

The runtime for this method is 2^n. We can see that every recurring call creates 2 different equivalent calls. The biggest problem with this method is that we are doing a lot of redundant calls because of the equivalent calls.

Deriving the recurrence formula, we will denote the function's runtime with $T(n)$. Then we can denote the first recursive call's runtime with $T(n-1)$. So after the first recursive call, the whole function's runtime is $T(n) = c + 2 \times T(n-1)$. Then we can denote the second recursive call's runtime as $T(n-2)$. The first recursive call's runtime with respect to the second recursive call's runtime is $T(n-1) = c + 2 \times T(n-2)$. Furthermore, we can then say that the whole method's runtime with respect to the second recursive call is $T(n) = c + 4 \times T(n-2)$. Doing this for the third recursive call results in $T(n) = c + 8 \times T(n-3)$. As you can see, the coefficient for the current recursive call's runtime doubles each time (2 to 4 to 8). Therefore, we can say that the runtime for this method is 2^n.

(b) Introduce a small modification that makes the function run in linear time. Show why the runtime is linear.

All we need to do to change the function's runtime to linear is change the expression in the method to be `return 2*power2(n-1).`

Deriving the recurrence formula, the function's runtime is $T(n)$. The first recursive call's runtime is $T(n-1)$. The method's runtime $T(n)$ with respect to the first recursive call is $T(n) = c + T(n-1)$. Similarly, we can find the runtime of the second recursive call's runtime is $T(n-1) = c + T(n-2)$. Therefore, the method's runtime $T(n)$ with respect to the second recursive call is $T(n) = 2c + T(n-2)$. After n calls and reaching the if statement where $T(1) = k$, we can see that the runtime is $T(n) = (n-1) \times c + k = O(n)$. This is linear runtime.

(c) (bonus) The following function also calculates 2^n:

```
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 $2^n = (2^{\frac{n}{2}})^2$, so we can calculate $2^{\frac{n}{2}}$ recursively and square, cutting half of n in one move. Otherwise, we resort to the previous method. Show that the runtime of power2New is logarithmic in n. Hint: It's easy to show it when n is even, but sometimes n is odd... The trick is to show that the entire function is logarithmic nonetheless.

7. **The stable matching algorithm:** Given the following rankings of students and hospitals like the one given in class:

Atlanta	Xavier	Yolanda	Zeus
Boston	Yolanda	Xavier	Zeus
Chicago	Xavier	Yolanda	Zeus

Xavier	Boston	Atlanta	Chicago
Yolanda	Atlanta	Boston	Chicago
Zeus	Atlanta	Boston	Chicago

(a) Show that any stable match will match Zeus and Chicago (very easy. Almost copy-paste from the notes).

We can determine that Zeus and Chicago will be paired together no matter the order of hospitals that offer admission. For example, Atlanta would offer to Xavier and he would accept because Atlanta is at the top of his list. Boston would then offer Yolanda as their top applicant. Yolanda would accept as her top choice, Atlanta, is already matched with Xavier. Chicago would then offer Xavier and Yolanda, but they would refuse because they are at better hospitals in their rankings. Finally, Chicago would be matched up the Zeus. Now that all hospitals and students are paired, the algorithm is complete. Doing this twice more with the 2 other different order of hospitals all result in Chicago and Zeus being paired together.

(b) Show that it is true in any setting for any number of students/hospitals. In other words, if a student s is last on all of the hospital's list, and a hospital h is last on every student's list, they will end up matched

to each other. This is also quite simple: Show that any other match would result in instability. Note that the proof should be for **any match**, not just the one produced by Gale-Shapley's algorithm.

We can say that for any unstable pairing results in either a hospital that wants a better candidate, or a student that wants a better location, or both. Then for any matching algorithm, if it comes across an unstable pair, the hospital will be replaced. In this way, if a hospital is last on all students' lists, the student whose last on that hospital's list will be paired together. All other combinations would result in another student wanting to go to any of the other locations (therefore unstable).

8. **Designing a Java class:** A combination lock has the following basic properties: the combination (a sequence of three numbers) is hidden; the lock can be opened by providing the combination; and the combination can be changed, but only by someone who knows the current combination. Design a class with public methods `open` and `changeCombo` and private data fields that store the combination. The combination should be set in the constructor. Provide the java code as part of your submission, not as a separate file (just copy-paste the code to your text file). You may compile and run your code for testing but I will not- I will just look for overall design and understanding of the concept.

CombinationLock.java

```
public class CombinationLock{
    private int a, b, c; // The lock's combination

    // Constructor
    public CombinationLock(int a1, int a2, int a3) {
        this.a = a1;
        this.b = a2;
        this.c = a3;
    }

    // Opens the lock if the attempted combination matches the lock's combination
    public boolean open(int x, int y, int z) {
        return x == a && y == b && z == c;
    }

    // Changes the lock's combination (old combination followed by new combination)
    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;
        }
        return false;
    }
}
```

9. **Designing an encapsulated Java class:** In the example `FrequencyCounter.java` (https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/FrequencyCounter.java), the word with highest frequency is found, along with its count. This result could be packaged up in one object with two instance variables "word" and "count". For example if "the" showed up 2034 times in the text, this object would have word "the" and count 2034. Create such a class, named WordUsage, with a constructor taking both the word and the count, and another constructor taking only the word and using count 1. Make the instance variables private for proper encapsulation of the objects. Give the class getters for word and count (i.e. getWord and getCount) and a setter for count but not word (setCount) and a mutator method named increment that adds one to the count. Don't implement equals or other Object methods here: this is a simple object meant for just carrying data from one place to another in our code. Note the strong Java convention that class names are

capitalized, but method names and variables start with lower case. See pp. 84-85 of S&W for discussion of a similar class implementation. Here we can also mark the String instance variable as final because we can't change it once the object is created, since there is no setter for the word, and the instance variables are private, protected against direct access.

WordUsage.java

```
public class WordUsage {
    private final String word; // the current word being tracked
    private int count; // the number of times this word has been used

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

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

    // Returns this tracked word
    public String getWord() {
        return this.word;
    }

    // Returns this word's count
    public int getCount() {
        return this.count;
    }

    // Sets this word's count
    public void setCount(int count) {
        this.count = count;
    }

    // Increments the count for this word
    public void increment() {
        this.count++;
    }
}
```

10. **Java questions:**

    (a) What is the difference between a final class and other classes? Why are final classes used?

    Final classes are a way for preventing the possibility of inheritance on that class. Meaning that no other classes can extend the final class. Final classes can be used to ensure that the program doesn't tamper with the object.

    (b) What is an interface? How does the interface differ from an abstract class? What members may be in an interface?

    An interface can be thought of an outline that other classes can implement. Interfaces can method signatures with no bodies (also known as abstract methods). Abstract classes can have abstract methods of any type as well as declare fields. You can extends multiple interfaces but one one abstract class.