- Homework solutions should be neatly written or typed and turned in through **Gradescope** by 11:59 pm on the due date. No late homework will be accepted for any reason. You will be able to look at your scanned work before submitting it. Please ensure that your submission is legible (neatly written and not too faint), or your homework may not be graded.
- Students should consult their textbook, class notes, lecture slides, instructor, and TAs when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the internet. Only post about graded homework questions on Piazza if you suspect a typo in the assignment or if you don't understand what the question is asking you to do.
- Your assignments in this class will be evaluated not only on the correctness of your answers but on your ability to present your ideas clearly and logically. You should always explain how you arrived at your conclusions using mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to convince the reader that your results and methods are sound.
- For questions requiring pseudocode, you can follow the same format as we do in class or write pseudocode in your style, as long as you specify your notation. For example, are you using "=" to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using `InsertionSort`, you can call `InsertionSort(`A`)` instead of writing out the pseudocode for `InsertionSort`.

There are 6 questions for a total of 100 points.

---

1. (*Analysing recursive functions*) In this question, you are asked to analyse two recursive functions.
   (a) (10 points) Consider the following recursive function that takes as input a positive integer.

   ```
   F(n)
     · If (n > 1) F(⌊n/2⌋)
     · print("Hello World")
   ```

   Give the **exact** expression, in terms of n, for the number of times "Hello World" is printed when a call to $F(n)$ is made. Argue the correctness of your expression using mathematical induction.

   (b) (15 points) Consider the following recursive function that takes as input two positive integers n and m.

   ```
   G(n, m)
     · if (n = 0 OR m = 0)return;
     · print("Hello World")
     · G(n − 1, m)
     · G(n, m − 1)
   ```

   Give the **exact** expression, in terms of n and m, for the number of times "Hello World" is printed when a call to `G(`n, m`)` is made. Argue the correctness of your expression using mathematical induction.

2. (*Proof techniques review*) Let us quickly review a few proof techniques you must have studied in your introductory Mathematics course.

- Direct proof: *Used for showing statements of the form p implies q. We assume that p is true and use axioms, definitions, and previously proven theorems, together with rules of inference, to show that q must also be true.*
- Proof by contraposition: *Used for proving statements of the form p implies q. We take $\neg q$ as a premise, and using axioms, definitions, and previously proven theorems, together with rules of inference, we show that $\neg p$ must follow.*
- Proof by contradiction: *Suppose we want to prove that a statement p is true and suppose we can find a contradiction q such that $\neg p$ implies q. Since q is false, but $\neg p$ implies q, we can conclude that $\neg p$ is false, which means that p is true. The contradiction q is usually of the form $r \wedge \neg r$ for some proposition r.*
- Counterexample: *Suppose we want to show that the statement for all x, $P(x)$ is false. Then we only need to find a counterexample: an example x for which $P(x)$ is false.*
- Mathematical induction: Used for proving statements of the form $\forall n, P(n)$. This has already been discussed in class.

Solve the proof problems that follow:

(a) (3 points) Give a direct proof of the statement: "If n is odd, then n^2 is odd".

(b) (3 points) Prove by contraposition that "if n^2 is odd, then n is odd".

(c) (3 points) Give proof by contradiction of the statement: "at least four of any 22 days must fall on the same day of the week."

(d) (3 points) Use a counterexample to show that the statement "Every positive integer is the sum of squares of two integers" is false.

(e) (3 points) Show using mathematical induction that for all $n \geq 0$, $1+\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}+...+\frac{1}{2^n} = \frac{1-(\frac{1}{2})^{n+1}}{1-\frac{1}{2}}$.

3. (*Working with the definition of big-O*)

Big-O notation requires practice to become comfortable. There is a bit of mathematical jugglery since the definition involves a quantified statement. However, the mathematics involved should not be new to you. All that is required is to work with the quantified statement. Let us consider an example. Suppose there is an exponential-time algorithm with a running time expression 2^n. Would it be correct to say that the running time is $O(3^n)$? Yes, this would be correct. Let us work with the definition to verify this. To show that 2^n is $O(3^n)$, all we need to do is to give constants $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0, 2^n \leq c \cdot 3^n$. It is easy to check that this holds for constants $c = 1$ and $n_0 = 1$.

Let us consider the reverse case. That is, suppose the running time of an algorithm is 3^n. Would it be correct to say that the running time is $O(2^n)$? No, this would not be correct. The way to argue that 3^n is **not** $O(2^n)$ is to show the negation of the quantified statement for this example. That is, we need to show that for **any** constants $c > 0, n_0 > 0$, there exists $n \geq n_0$ such that $3^n > c \cdot 2^n$. Note that $3^n > c \cdot 2^n \Leftrightarrow (3/2)^n > c \Leftrightarrow n > \log_{3/2} c$. So, for any constant c, $3^n > c \cdot 2^n$ when $n > \log_{3/2} c$. So, for any constants $c > 0$ and $n_0 > 0$, let us try $n' = \max(\lceil \log_{3/2} c + 1 \rceil, n_0)$. We find that $3^{n'} > c \cdot 2^{n'}$ and furthermore $n' \geq n_0$. From this, we conclude that 3^n is not $O(2^n)$.

Prove or disprove the following statements:

(a) (3 points) $2^{\sqrt{\log n}}$ is $O(n)$.

(b) (7 points) $(9n2^n + 3^n)$ is $\Omega(n3^n)$.

4. (*More practice with big-O definition*) Prove or disprove the following statements:
   (a) (5 points) If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
   (b) (5 points) $\max\{f(n), g(n)\}$ is $O(f(n) + g(n))$.
   (c) (5 points) If $a(n)$ is $O(f(n))$ and $b(n)$ is $O(g(n))$, then $a(n) + b(n)$ is $O(f(n) + g(n))$.
   (d) (5 points) If $f(n)$ is $O(g(n))$, then $2^{f(n)}$ is $O(2^{g(n)})$.
   (e) (5 points) If $f(n)$ is $O(g(n))$, then $f(2n)$ is $O(g(2n))$.

5. (15 points) (*Arguing correctness using induction*) Argue using induction that the following algorithm correctly computes the value of a^n when given positive integers a and n as input.

```
Exponentiate(a, n)
  · if (n = 0) return(1)
  · t = Exponentiate(a, ⌊n/2⌋)
  · if (n is even) return(t · t)
  · else return(t · t · a)
```

6. (*Analysing running time*) We start with a brief discussion.

As discussed in class, the big-O notation allows us to ignore the hairy details we may need to keep track of otherwise. Let us consider the example of the Insertion Sort algorithm below for this discussion.

```
InsertionSort(A, n)
 · for i = 1 to n
   · j = i − 1
   · while(j > 0 and A[j] > A[i]) j- -
   · for k = j+1 to i-1
     · Swap A[i] and A[k]
```

The first line of the algorithm is a 'for' statement. How many basic operations does this 'for' statement contribute? Getting a precise count will require some deeper analysis of its implementation mechanism. It should involve a comparison operation in every iteration since one needs to check if the variable i has exceeded n. It should include arithmetic operation since the variable i needs to be incremented by 1 at the end of each iteration. Are these all the operations? Not really. The increment operation involves loading the variable and storing it after the increment. Keeping track of all these hairy details may be too cumbersome. A quick way to account for all possible basic operations is to say that the contribution to the number of operations coming from every iteration of the outer 'for' statement is a constant. So, the number of operations contributed by the outer 'for' statement is $an + b$ for some constants a, b. This is the level of granularity at which we shall do the counting to keep things simple.

Answer the questions that follow:

   (a) (2 points) What is the worst-case number of operations contributed by the while-loop in the i^{th} iteration of the outer for-loop as a function of i? (*You can give an expression in terms of symbols for constants as we did for the outer 'for' statement in the discussion. This will be sufficient since the big-O will allow us to ignore the specific constants.*)

(b) (2 points) Similarly, what is the worst-case number of operations contributed by the inner for-loop in the i^{th} iteration of the outer for-loop as a function of i? Again, express using symbolic constants.

(c) (2 points) Use expressions of the previous two parts to give the worst-case number of operations executed by the algorithm. Use big-O notation. Give a brief explanation.

(d) (4 points) Show that your running time analysis in the previous part is tight.