- The solutions are provided to add to your understanding of the concepts. Sometimes, we may provide extra intuition or ideas that the question does not explicitly ask.
- We may not provide solutions to all questions. For multiple questions of similar nature, we may provide a solution for only one. For straightforward questions, we may avoid giving a solution altogether. Please discuss solutions for such questions on Piazza in case needed.
- The solution may contain some typos. Feel free to report these to the course staff.

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.

Solution: We will show that $T(n) = \lfloor \log_2 n \rfloor + 1$. In order to argue that $T(n) = \lfloor \log_2 n \rfloor + 1$, we will argue that for all n such that $2^k \leq n < 2^{k+1}$, $T(n) = k$ for all values of k. This we show using induction.

Let $P(i)$ denote the statement that for all n such that $2^i \leq n < 2^{i+1}, T(n) = i + 1$.

Basis step: $P(0)$ is true since for all n, $1 \leq n < 2$, the function prints "Hello World" once.

Induction step: Assume that $P(0), P(1), ..., P(k)$ are true. We will argue that $P(k+1)$ is true. Consider any number n, $2^{k+1} \leq n < 2^{k+2}$. When the function F is called with such a number n as input, it makes a recursive call to $F(\lfloor n/2 \rfloor)$. So the total number of times "Hello World" is printed will be $T(\lfloor n/2 \rfloor) + 1$. Note that $2^k \leq \lfloor n/2 \rfloor < 2^{k+1}$. So, from the induction hypothesis, we get that $T(\lfloor n/2 \rfloor) = k + 1$. So, $T(n) = T(\lfloor n/2 \rfloor) + 1 = k + 2$. This shows that $P(k+1)$ is true.

Using the principle of induction, we get that $P(i)$ is true for all i.

   (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.

Solution: Let $L(n, m)$ denote the number of times "Hello World" is printed when `G(n, m)` is called. We can say the following about $L(n, m)$:

1. $L(0, m) = 0$ for any $m \geq 0$
2. $L(n, 0) = 0$ for any $n \geq 0$
3. $L(n, m) = L(n-1, m) + L(n, m-1) + 1$ for $n, m > 0$

The main challenge is finding a closed-form solution for this 2-Dimensional recurrence relation. You can get some intuition by filling a 2-D table with the value of $L(n, m)$. The figure below shows such a table.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	5	9	14	20
3	0	3	9	19	34	55
4	0	4	14	34	69	125
5	0	5	20	55	125	251

A bit of creative thinking reveals that $L(n, m)$ corresponds roughly to the number of ways of traversing from one corner of a 2-D $n \times m$ grid to the diagonally opposite corner along grid lines that either go left-to-right or top-to-down. Using this intuition, we make our guess for $L(n, m)$ as $\binom{n+m}{n} - 1$. Note that this is consistent with the table above. Now we need to argue that this is indeed the expression. We try to prove this by induction. One bottleneck is that this is not a typical one-dimensional induction that we are used to. So, we need to set it up carefully. We will work with the following propositional function $P(.)$

$P(i)$: For every positive integer pair (n, m) for which $n + m = i$, $L(n, m) = \binom{n+m}{n} - 1$.

Base case: $P(1)$ holds since the the only positive integer pairs that sum to 1 are $(1, 0)$ and $(0, 1)$. The number of times "Hello World" is printed on these inputs is 0 which is $\binom{1+0}{1} - 1 = \binom{1+0}{0} - 1 = 0$.

Inductive step: Assume that $P(1), ..., P(k)$ holds for an arbitrary $k \geq 1$. We will show that $P(k+1)$ holds. Consider any positive integer pair (n, m) such that $n + m = k + 1$. We have:

$$\begin{aligned} L(n, m) &= L(n-1, m) + L(n, m-1) + 1 \\ &= \binom{n+m-1}{n-1} - 1 + \binom{n+m-1}{n} - 1 + 1 \quad \text{(using Induction hypothesis)} \\ &= \binom{n+m}{n} - 1 \end{aligned}$$

This shows that $P(k+1)$ holds.

From the principle of mathematical induction, we get that $P(i)$ holds for all i, which in turn shows that the number of times "Hello World" is printed for any positive integer pair (n, m) is $\binom{n+m}{n} - 1$.

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 $O(n)$.

Solution: Since n can be written as $2^{\log_2 n}$, we have that $2^{\sqrt{\log_2 n}} \leq 1 \cdot n$ for all $n > 1$. So, for $c = 1$ and $n_0 = 1$, we have $\forall n \geq n_0, 2^{\sqrt{\log_2 n}} \leq c \cdot n$. So, from the definition of big-O, we get that $2^{\sqrt{\log_2 n}}$ is $O(n)$.

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

Solution: We will disprove the statement. $f(n)$ is not $\Omega(g(n))$ if and only if for all constants $c > 0, n_0 \geq 0$, there exists $n \geq n_0$ such that $f(n) < c \cdot g(n)$. Note that $(c/2)n3^n > 3^n$ when $n > 2/c$ for any constant c. Moreover, note that $(c/2)n3^n > 9n2^n \Leftrightarrow (3/2)^n > 18/c \Leftrightarrow n > \log_{3/2}(18/c)$.
Combining the previous two statements, we get that for any constant $c > 0$, $cn3^n > (10n2^n + 3^n)$ when $n > \max(2/c, \log_{3/2}(18/c))$. This further implies that for any constants $c > 0, n_0 \geq 0$, $cn'3^{n'} > (10n'2^{n'} + 3^{n'})$ for $n' = \max(2/c, \log_{3/2}(18/c), n_0)$. Note that $n' \geq n_0$. So, for any constants $c > 0, n_0 \geq 0$, there is a number $n \geq n_0$ (n' above is such a number) such that $cn3^n > (10n2^n + 3^n)$. This implies that $f(n)$ is not $\Omega(g(n))$.

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 $O(2^{g(n)})$.

Solution: We will disprove the statement. Consider $f(n) = 2n$ and $g(n) = n$. For these functions we can show that $f(n) = O(g(n))$ since for all $n \geq 1$, $f(n) \leq 2 \cdot g(n)$. However, $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$. For any constant c, we can show that for all $n \geq \lceil \log_2 c \rceil + 1$, $2^{f(n)} > c \cdot 2^{g(n)}$. This is because if $n \geq \lceil \log_2 c \rceil + 1$, then $2^n > c$, which further implies $2^{2n} > c \cdot 2^n$.

   (e) (5 points) If $f(n)$ is $O(g(n))$, then $f(2n)$ is $O(g(2n))$.

Solution: Since $f(n)$ is $O(g(n))$, there exists constants c, n_0 such that for all $n \geq n_0, f(n) \leq c \cdot g(n)$. This means that for all $n \geq \lceil n_0/2 \rceil, f(2n) \leq c \cdot g(2n)$. This means that $f(2n)$ 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)
```

Solution: We will show correctness using mathematical induction on n. We will work with the following proposition:
$P(i)$: For any value of a, the algorithm on input (a, i) returns the correct value of a^i.

Base case: $P(0)$ holds since the algorithm on input $(a, 0)$ returns 1 which is the correct value of a^1.

Inductive step: Assume that $P(0), P(1), ..., P(k)$ holds for an arbitrary $k \geq 0$. We will show that $P(k+1)$ holds. Let us examine what the algorithm does on input $(a, k+1)$. The algorithm makes a call to `Exponentiate`$(a, \lfloor (k+1)/2 \rfloor)$ and the result is stored in variable t. From the induction hypothesis, we get that $t = a^{\lfloor (k+1)/2 \rfloor}$. If $k+1$ is even, then $\lfloor (k+1)/2 \rfloor = (k+1)/2$ and in this case, the algorithm returns $t \cdot t = a^{(k+1)/2} \cdot a^{(k+1)/2} = a^{k+1}$. On the other hand, if $k+1$ is even, then $\lfloor (k+1)/2 \rfloor = k/2$ and in this case, the algorithm returns $t \cdot t \cdot a = a^{k/2} \cdot a^{k/2} \cdot a = a^{k+1}$. In either case, it returns the correct value a^{k+1}. So, $P(k+1)$ holds.

From the principle of Mathematical induction, we get that $P(n)$ holds for all $n \geq 0$. This proves the correctness of our algorithm.

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.*)

Solution: $ci + d$, for some constants c, d.

(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.

Solution: $ei + f$, for some constants e, f.

(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.

Solution: The time spent within in the i^{th} iteration of the outer for-loop is $(ci + d) + (ei + f)$ plus the operations in the statement $j = i - 1$ which is some constant g. So, the overall running time is $(an + b) + \sum_{i=1}^{n}(ci + d + ei + f + g) = b + (a + d + f + g)n + (c + e)n(n + 1)/2$ which is $O(n^2)$.

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

Solution: As discussed in class, to show the tightness of the bound, it is sufficient to give one input example of size n on which the algorithm spends $\Omega(n^2)$ time. Consider an input array containing n distinct numbers sorted in decreasing order. In this case, the expressions for the running time obtained in parts (a) and (b) are tight since $A[i]$ is smaller than all $A[1], ..., A[i-1]$ and so the inner while and for loops run for $\Omega(i)$ steps in the i^{th} iteration of the outer for-loop.