- 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.

Sol: Here,
$$F(n) = \begin{cases} F(\lfloor n/2 \rfloor) + 1 & , n > 1 \\ 1 & , n \leq 1 \end{cases}$$

So,
$$\Rightarrow F(n) = F\left(\left\lfloor \frac{n}{2^1} \right\rfloor\right) + 1$$
$$\Rightarrow F(n) = F\left(\left\lfloor \frac{n}{2^2} \right\rfloor\right) + 1 + 1$$
$$\Rightarrow F(n) = F\left(\left\lfloor \frac{n}{2^3} \right\rfloor\right) + 1 + 1 + 1$$
$$\vdots$$
$$\Rightarrow F(n) = F\left(\left\lfloor \frac{n}{2^k} \right\rfloor\right) + \underbrace{1 + 1 + 1 + \cdots + 1}_{k \text{ times}}$$
$$\Rightarrow F(n) = F\left(\left\lfloor \frac{n}{2^k} \right\rfloor\right) + k$$

For Termination:

$$\frac{n}{2^k} = 1$$

$$\Rightarrow 2^k = n$$
$$\Rightarrow k = log_2 n$$
$$\Rightarrow k = \lfloor log_2 n \rfloor$$

Note: k will be equal to $\lfloor \log_2 n \rfloor$ because division by 2 will take exactly $\log_2 n$ steps iff n is in powers of 2, else it will take $\lfloor \log_2 n \rfloor$ steps. This can easily be proved by solving for $n = 2, 3, 4, 5, 6, 7, 8, \cdots$.

$$\therefore F(n) = \begin{cases} \lfloor log_2 n \rfloor + 1 & , n > 1 \\ 1 & , n \leq 1 \end{cases}$$

Proving Correctness:

Base Case:

For n = 1: Given function prints "Hello World" 1 time.
Also,

$$F(n) = 1$$
$$\therefore \text{For n = 1, } F(1) \text{ holds.}$$

Termination Check:

At each recursive call, we are converting the problem into a sub-problem of almost half the size of the original problem.
$\therefore$ The problem will eventually reach the base case, print "Hello World" once and, henceforth, terminate.

Induction step:

For $n = k$: Assume $F(1), F(2), \ldots, F(k)$ holds true (Induction Hypothesis).

For $n = k + 1$:

$$\begin{aligned} F(k+1) &= F\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) + 1 \\ &= \left\lfloor \log_2 \left(\frac{k+1}{2}\right) + 1 \right\rfloor + 1 \\ &= \lfloor \log_2(k+1) - \log_2 2 + 1 \rfloor + 1 \\ &= \lfloor \log_2(k+1) - 1 + 1 \rfloor + 1 \\ &= \lfloor \log_2(k+1) \rfloor + 1 \end{aligned}$$

$\Rightarrow F(k+1)$ holds for $n = k + 1$.
$\because [F(1) \wedge F(2) \wedge \ldots \wedge F(k)] \Rightarrow F(k+1)$ holds.
$\therefore$ F(n) is true, $\forall$ n ≥ 1 (induction step).

Q.E.D.

(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.

Sol: Here,

Let $G(n, m)$ denote the number of times "Hello World" is printed by our code.

$$G(n,m) = \begin{cases} 0 & ,(n == 0)||(m == 0) \\ G(n-1,m) + G(n,m-1) + 1 & ,otherwise \end{cases}$$

Basically, the number of times "Hello World" will be printed is equal to the number of internal nodes of the recursion tree of G(n, m).
So, let us try to fill a table for G(6, 7) and observe patterns:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	5	9	14	20	27	35
3	0	3	9	19	34	55	83	119
4	0	4	14	34	69	125	209	329
5	0	5	20	55	125	251	461	791
6	0	6	27	83	209	461	923	1715

From the above table, we can clearly observe that the value at position G[i][j] in the above 2-D array is equal to (the number of paths possible from G[0][0] to G[i][j]) − 1.
Now, to reach from G[0][0] to G[n][m], the total number of paths possible (n down moves and m right moves):

$$\begin{aligned} &= \left\{ \underbrace{right, right, right \cdots, right}_{(m\text{-times})}, \underbrace{down, down, down, \cdots, down}_{(n\text{-times})} \right\} \\ &= \frac{(m+n)!}{m!.n!} \\ &= {}^{m+n}C_n \end{aligned}$$

Hence,

$$G(n,m) = \begin{cases} 0 & ,(n == 0)||(m == 0) \\ {}^{m+n}C_n - 1 & ,otherwise \end{cases}$$

Proving Correctness:

Base Case:

For $n = 1, m = 1$: Given function prints "Hello World" 1 time.

Also,

$$\begin{aligned} G(1,1) &=^{1+1} C_1 - 1 \\ &=^{2} C_1 - 1 \\ &= 2 - 1 \\ &= 1 \end{aligned}$$

$\therefore$ For n = 1, m = 1, $G(1,1)$ holds.

Termination Check:

At each recursive call, we are converting the problem into two sub-problems, by reducing either of their sizes by 1.
$\therefore$ The recursion will eventually reach the base case, and henceforth, terminate.

Induction step:

For $n = k, m = k$: Assume $G(1,1), G(1,2), G(2,1) \ldots, G(k,k)$ holds true (Induction Hypothesis).

For $n = k+1, m = k+1$:

$$\begin{aligned} G(k+1,k+1) &= G(k,k+1) + G(k+1,k) + 1 \\ &=^{2k+1} C_k - 1 +^{2k+1} C_{k+1} - 1 + 1 \\ &=^{2k+1} C_k +^{2k+1} C_{k+1} - 1 \end{aligned}$$

Using Pascal's Identity for Combinatorial Arguments:

$$^{2k+2}C_{k+1} =^{2k+1} C_k +^{2k+1} C_{k+1}$$

$$\Rightarrow G(k+1,k+1) =^{2k+2} C_{k+1} - 1$$

$\Rightarrow G(k+1,k+1)$ holds for $n = k+1$ and, $m = k+1$.
$\because [G(1,1) \wedge G(1,2) \wedge G(2,1) \wedge \ldots \wedge G(k,k)] \Rightarrow G(k+1,k+1)$ holds.
$\therefore$ G(n, n) is true, $\forall$ n ≥ 1 (induction step).

Q.E.D.

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 true. 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".

To prove: n is odd $\Rightarrow n^2$ is odd.
Proof: Assume the antecedent to be true, *i.e.*, n is odd. So, let $n = 2k + 1, \forall k \in \mathbb{Z}$.

Now,

$$\begin{aligned}(2k+1)^2 &= (4k^2 + 4k + 1) \\ &= 2(2k^2 + 2k) + 1 \\ &= \text{even} + 1 \\ &= \text{odd}\end{aligned}$$

$\therefore$ If n is odd, then n^2 is odd.

Q.E.D.

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

To prove: n^2 is odd $\Rightarrow n$ is odd.
Proof: Assume the consequent to be false, *i.e.*, n is even So, let $n = 2k, \forall k \in \mathbb{Z}$.

Now,

$$\begin{aligned}(2k)^2 &= 4k^2 \\ &= 2(2k^2) \\ &= \text{even}\end{aligned}$$

$\Rightarrow$ If n is even, then n^2 is even
or, If n is not odd, then n^2 is not odd.
$\therefore$ If n^2 is odd, then n is odd.

Q.E.D.

(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."

To prove: At least four (≥ 4) of any 22 days must fall on the same day of the week.
Proof: Assume at most three (< 4) days of 22 days fall on the same day of the week.

So,

$$\text{Maximum total number of days} = 3 * 7 = 21$$

But, it is given that we had 22 days.
$\therefore$ It is a contradiction.
$\Rightarrow$ Our assumption was wrong.
$\therefore$ At least four (>= 4) of any 22 days must fall on the same day of the week.

Q.E.D.

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

To prove: At least one positive integer exists, which is not equal to the sum of squares of two integers.

Proof: Assume $3 \in \mathbb{Z}^+$
Now,

$$3 = a^2 + b^2 \text{ cannot be satisfied for any a, b} \in \mathbb{Z}$$

$$\Rightarrow \text{The given statement is false.}$$
$$\therefore \text{Every positive integer need not be the sum of two integers.}$$

Q.E.D.

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

To prove: $\forall n \geq 0$, $1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^n} = \frac{1-(\frac{1}{2})^{n+1}}{1-\frac{1}{2}}$ holds.

Proof: Let P(n) $= 1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^n} = \frac{1-(\frac{1}{2})^{n+1}}{1-\frac{1}{2}}$

For n = 0 (Base Case):

$$\text{L.H.S.} = 1$$
$$\text{R.H.S.} = \frac{1-(\frac{1}{2})^{(0+1)}}{1-\frac{1}{2}}$$
$$= \frac{1-\frac{1}{2}}{1-\frac{1}{2}}$$
$$= 1$$

$$\because \text{L.H.S.} = \text{R.H.S.}$$
$$\therefore \text{For n} = 0, \text{P(0) holds.}$$

For n = k: Assume P(0), P(1), P(2), ..., P(k) holds true (Induction Hypothesis).
i.e.,

$$1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^{k-1}} = \frac{1-(\frac{1}{2})^k}{1-\frac{1}{2}}$$

For n = k+1:

$$\begin{aligned}
\text{L.H.S.} &= 1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^{k-1}} + \frac{1}{2^k} \\
&= \frac{1 - (\frac{1}{2})^k}{1 - \frac{1}{2}} + \frac{1}{2^k} \\
&= \frac{\frac{(2^k - 1)}{2^k}}{\frac{1}{2}} + \frac{1}{2^k} \\
&= \frac{2^{k+1} - 2}{2^k} + \frac{1}{2^k} \\
&= \frac{1}{2^k}(2^{k+1} - 1) \\
&= 2 - \frac{1}{2^k} \\
&= 2(1 - \frac{1}{2^{k+1}}) \\
&= \frac{(1 - \frac{1}{2^{k+1}})}{\frac{1}{2}} \\
&= \frac{(1 - \frac{1}{2^{k+1}})}{(1 - 1/2)}
\end{aligned}$$

$\because$ L.H.S. = R.H.S.
$\therefore$ [P(0) $\wedge$ P(1)$\wedge$ $\ldots \wedge$ P(k)] $\Rightarrow$ P(k + 1)
$\therefore$ P(n) is true, $\forall$ n $\geq$ 0 (induction step).

Q.E.D.

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 exptonential-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 $O(n)$.

Proof:

$$2^{\sqrt{\log n}} \leq c.n$$

Let $c = 1(> 0)$

$$\Rightarrow 2^{\sqrt{\log n}} \leq n$$
$$\Rightarrow \log 2^{\sqrt{\log n}} \leq \log n$$
$$\Rightarrow \sqrt{\log n} \leq \log n$$
$$\Rightarrow \sqrt{\log n} \geq 0$$
$$\Rightarrow logn \geq 0$$
$$\Rightarrow n \geq 2^0$$
$$\Rightarrow n \geq 1$$

$2^{\sqrt{\log n}} \in O(n)$ as: $2^{\sqrt{\log n}} \leq c.n$ holds ***TRUE*** for $c = 1$, $\forall n \geq n_0$ and, $n_0 = 1$

Q.E.D.

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

Disproof: Assume $(9n2^n + 3^n)$ is $\Omega(n3^n)$.
So,

$$9n2^n + 3^n \geq c.n3^n$$

Let $c = 3(> 0)$

$$\Rightarrow 9n2^n + 3^n \geq 3.n.3^n$$
$$\Rightarrow 9n2^n + 3^n \geq 3^{n+1}.n$$
$$\Rightarrow 9n2^n \geq 3^{n+1}.n - 3^n$$
$$\Rightarrow 9n2^n \geq 3^n(3n - 1)$$
$$\Rightarrow 9n\left(\frac{2}{3}\right)^n \geq (3n - 1)$$
$$\Rightarrow \left(\frac{2}{3}\right)^n \geq \frac{1}{3} - \frac{1}{9n}$$

$\because$ For $n > 3$, this inequality does not holds.
$\Rightarrow$ Our Assumption was wrong.
$\therefore (9n2^n + 3^n) \notin \Omega(n3^n)$.

Q.E.D.

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 $O(g(n))$.

Proof: Here,

$$d(n) \in O(f(n))$$
$$\Rightarrow d(n) \leq c1.f(n)\ ,\ c1 > 0 \quad (1)$$

Similarly,

$$f(n) \in O(g(n))$$

$$\Rightarrow f(n) \leq c2.g(n) \ , \ c2 > 0 \tag{2}$$

From eq. 1 and 2:

$$\Rightarrow d(n) \leq c1.c2.g(n)$$
$$\Rightarrow d(n) \leq c3.g(n) \ , \ c3 > 0$$

$\therefore d(n) \in O(n)$ as: $d(n) \leq c3.g(n)$ holds ***TRUE*** for $c3 > 0, \forall n \geq n_0$ and, $n_0 > 0$.

Q.E.D.

(b) (5 points) $\max\{f(n), g(n)\}$ is $O(f(n) + g(n))$.

Proof: Here, Let's say:

$$max(f(n), g(n)) = \begin{cases} f(n) & , f(n) >= g(n) \\ g(n) & , f(n) < g(n) \end{cases}$$

As we know, asymptotic analysis is performed only on non-negative functions:
So,

$$f(n) \geq 0 \tag{3}$$

and,

$$g(n) \geq 0 \tag{4}$$

Also,

$$f(n) \leq f(n)$$
$$\Rightarrow f(n) \leq f(n) + g(n) \tag{5}$$

and,

$$g(n) \leq g(n)$$
$$\Rightarrow g(n) \leq f(n) + g(n) \tag{6}$$

From eq. 3, 4, 5 and, 6:

$$\Rightarrow max(f(n), g(n)) \leq f(n) + g(n)$$
$$\Rightarrow max(f(n), g(n)) \leq 1.(f(n) + g(n))$$

$\therefore max(f(n), g(n)) \in O(f(n) + g(n))$ as $max(f(n), g(n)) \leq c.(f(n) + g(n))$ holds ***TRUE*** for $c = 1$, $\forall n \geq n_0$ and, $n_0 > 0$.

Q.E.D.

(c) (5 points) If $a(n)$ is $O(f(n))$ and $b(n)$ is $O(g(n))$, then $a(n) + b(n)$ is $O(f(n) + g(n))$.

Proof: Here,

$$a(n) <= c1.f(n) \quad , c1 > 0 \tag{7}$$

$$b(n) <= c2.g(n) \quad , c2 > 0 \tag{8}$$

From eq. 7 and eq. 8:

$$a(n) + b(n) \leq c1.f(n) + c2.g(n)$$
$$\Rightarrow a(n) + b(n) \leq (c1 + c2).f(n) + (c1 + c2).g(n)$$
$$\Rightarrow a(n) + b(n) \leq (c1 + c2).(f(n) + g(n))$$

Let $c = c1 + c2$, for some $c > 0$.

$$\Rightarrow a(n) + b(n) \leq c.\,(f(n) + g(n))$$

$\therefore a(n) + b(n) \in O\,(f(n) + g(n))$ holds ***TRUE*** for $c > 0$, $\forall n \geq n_0$ and $n_0 > 0$.

Q.E.D.

(d) (5 points) If $f(n)$ is $O(g(n))$, then $2^{f(n)}$ is $O(2^{g(n)})$.

Disproof: We can easily disproof this via a counter-example.

Assume $f(n) = 2n$ and $g(n) = n$.

We know that,

$f(n) \in O(g(n))$ as $2n \leq c \cdot n$ holds true for $c \geq 2$ and $n \geq 1$

But, $2^{2n} \notin O(2^n)$ as $2^{2n} > c.2^n$ holds for $c < 2^n$ which is true for $c = 1$ and $n = 1$

$\therefore$ If $f(n)$ is $O(g(n))$, then $2^{f(n)}$ need not be $O(2^{g(n)})$.

Q.E.D.

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

Proof: Assume,

If $f(n) \in O(g(n))$, then $f(2n) \in \omega(g(2n))$.
Let $f(n) = n$ and $g(n) = n^2$.
$\Rightarrow f(2n) = 2n$ and $g(2n) = 4n^2$.

But,

$2n \in O(4n^2)$ as $2n \leq c.(4n^2)$ holds **TRUE** for $c = 1, \forall n \geq n_0$ and, $n_0 = \dfrac{1}{2}$

$\Rightarrow$ Our assumption was wrong. *(contradiction)*
$\therefore$ If $f(n)$ is $O(g(n))$, then $f(2n)$ is $O(g(2n))$.

Q.E.D.

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)
```

To prove: `Exponentiate(a, n)` produces correct output $\forall a, b \in \mathbb{Z}^+$
Proof: Let P(n) = value returned by `Exponentiate(a, n)`.
Base Case:

For n = 1:

`Exponentiate(a, 1)` returns $1 * 1 * a = a$.

$\therefore$ For $n = 1$, $P(1)$ holds.

Termination Check:

At each recursive call, we are converting the problem into a sub-problem of almost half the size of the original problem *(optimal substructures)*.
$\therefore$ The recursion will eventually reach the base case, return 1 and, henceforth, terminate.

Induction step:

For $n = k$: Assume $P(1), P(2), \ldots, P(k)$ holds true (Induction Hypothesis).

case 1: If k is even

For $n = k + 1$: $\Rightarrow (k + 1)$ is odd

So,

$$\begin{aligned} &\Rightarrow t = Exponentiate(a, \lfloor (k+1)/2 \rfloor) \\ &\because P(\lfloor (k+1)/2 \rfloor) \text{ holds.} \\ &\Rightarrow t = a^{k/2} \\ &\therefore P(k+1) = t.t.a \\ &\quad = a^{k/2}.a^{k/2}.a \\ &\quad = a^{k+1} \end{aligned}$$

$\therefore P(k + 1)$ holds for $n = k + 1$.

Similarly,

case 2: If k is odd

For $n = k + 1$: $\Rightarrow (k + 1)$ is even

So,

$$\begin{aligned} &t = Exponentiate(a, \lfloor (k+1)/2 \rfloor) \\ &\because P(\lfloor (k+1)/2 \rfloor) holds. \\ &\Rightarrow t = a^{(k+1)/2} \\ &\therefore P(k+1) = t.t \\ &\quad = a^{(k+1)/2}.a^{(k+1)/2} \\ &\quad = a^{k+1} \end{aligned}$$

$\Rightarrow P(k+1)$ holds for $n = k+1$.
$\because [P(1) \wedge P(2) \wedge \ldots \wedge P(k)] \Rightarrow P(k+1)$ holds.
$\therefore$ P(n) is true, $\forall$ n ≥ 0 (induction step).

Q.E.D.

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.)

Sol: Assume, a worst-case input of size n containing first n natural numbers as:

$$\underbrace{2, 3, 4, 5, 6, \ldots, (n-1), \underbrace{n}_{j} \underbrace{\vdots}_{} \underbrace{1}_{i}}_{\text{sorted part}} \quad \text{unsorted part}$$

So,

Consider the n^{th} iteration when i points to 1 and j points to n. Now, in the while loop, 1 will be first compared with n, then with $(n-1)$, then with $(n-2)$ $\cdots$ and, so on, until $(j > 0)$ condition becomes false.
So, it will take $(n-1)$ total comparisons to find appropriate place for 1 in the sorted part of the

array.
$\therefore$ Worst-case number of comparisons by the while loop = $(cn + d)$ 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.

Sol: Again, consider the same worst-case input as:

$$\underbrace{\underbrace{\underbrace{}_{j}\underbrace{2}_{k}, 3, 4, 5, 6, \ldots, (n-1), n}_{\text{sorted part}}\underbrace{\vdots\, 1}_{i}}_{\text{unsorted part}}$$

So,

Consider the n^{th} iteration when i points to 1 and j points at index 0 *(outside the array)*. Now, k will iterate from elements 2 to n, and in each iteration swap elements with elements at i^{th} position to create a sorted array of size n.
$\therefore$ Worst-case number of comparisons by the inner for-loop = $(en + f)$ 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.

Sol: Consider a worst-case input as:

$$n, (n-1), (n-2), \cdots, 3, 2, 1$$

Here,

The outer for-loop will iterate on all elements from 1 to n. So, it will take $(an+b)$ comparisons, for some constant a and b, say.
In each iteration, first, the while-loop will find the appropriate position for the element in $(cn + d)$ comparisons. Then, the inner for-loop will place the element in its correct position by performing array shift operations, in $(en + f)$ comparisons.
$\therefore$ Worst-case number of operations performed by the algorithm:

$$\begin{aligned}
&= (an + b).((cn + d) + (en + f)) \text{ for some constants a, b, c, d, e, f.} \\
&= (an + b).((c + e).n + d + f) \\
&= (an + b).((c + e).n + d + f) \\
&= (ac + ae)n^2 + adn + afn + (bc + be)n + bd + bf \\
&= (ac + ae)n^2 + (ad + af + bc + be)n + bd + bf \\
&= O(n^2)
\end{aligned}$$

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

Sol: The running time analysis in previous parts is tight, because, out of all possible inputs *(input space)*, the worst-case input $[n, (n-1), (n-2), \cdots, 3, 2, 1]$ is taking, at max, $O(n^2)$ comparisons.
$\therefore$ The worst case time complexity of above algorithm is $O(n^2)$.