Name: Param Somane	**Access ID:** pss5256

Problem 1

Points:

1. $f = \Omega(g)$

2. $f = \Omega(g)$

3. $f = O(g)$

4. $f = \Omega(g)$

5. $f = \Omega(g)$

6. $f = \Omega(g)$

This holds because $\lim_{n\to\infty} \frac{n^2}{(\log n)^{\log n}} = \lim_{n\to\infty} \frac{n^2}{n^{\log\log n}} = 0$

$$\begin{aligned}
&\implies n^2 = o((\log n)^{\log n}) \\
&\implies n^2 = O((\log n)^{\log n}) \\
&\implies (\log n)^{\log n} = \Omega(n^2).
\end{aligned}$$

7. $f = \Theta(g)$

In fact, $f = g$.

8. $f = O(g)$

9. $f = O(g)$

10. $f = \Omega(g)$

11. $f = \Omega(g)$

Proof. By Sterling's approximation, $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ as $n \to \infty$. Thus, $n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$. Now, consider the following limit

$$\begin{aligned}
L &:= \lim_{n\to\infty} \frac{(\log n)^n}{n!} = \frac{1}{\sqrt{2\pi}} \cdot \lim_{n\to\infty} \frac{(e\log n)^n}{n^n\sqrt{n}} \\
\implies \log L' &= \lim_{n\to\infty} n(\log(e\log n) - \log n) - \log(\sqrt{n}) && (\text{where } L' = \sqrt{2\pi}\cdot L) \\
\implies \log L' &= \lim_{n\to\infty} n(1 + \log(\log n) - \log n) - \frac{\log(n)}{2} \\
\implies \log L' &= \infty \cdot (-\infty) - \infty && (\because \log n \text{ dominates over 1 and } \log(\log n)) \\
\implies \log L' &= -\infty \implies L' \to e^{-\infty} \implies L' \to 0 \implies L \to 0
\end{aligned}$$

Hence, $(\log n)^n = o(n!) \implies (\log n)^n = O(n!) \implies n! = \Omega((\log n)^n)$. ■

12. $f = O(g)$

Proof. As $n \to \infty$, we have $0 < \frac{n!}{n^n} = \frac{1\cdot 2\cdots n}{n\cdot n\cdots n} = \left(\frac{1}{n}\right)\cdot\left[\left(\frac{1}{n}\right)\cdots\left(\frac{n-1}{n}\right)\cdot\left(\frac{n}{n}\right)\cdot\right] \le \frac{1}{n} \to 0$. By the squeeze lemma, $\lim_{n\to\infty}\frac{n!}{n^n} = 0 \implies n! = o(n^n) \implies n! = O(n^n)$. ■

13. $f = \Theta(g)$

Proof. We utilize Sterling's approximation again. As $n \to \infty$, we have $\frac{\log(n!)}{\log(n^n)} = \frac{\log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)}{log(n^n)}$ $= \frac{1/2\log(2\pi)+1/2\log(n)+n\cdot\log(n)-n\cdot\log e}{n\cdot\log n} = \frac{\log(2\pi)}{2}\cdot\frac{1}{n\cdot\log n}+\frac{1}{2}\cdot\frac{1}{n}+1-\frac{1}{\log n} \to \frac{\log(2\pi)}{2}\cdot 0+\frac{1}{2}\cdot 0+1-0 = 1$. Thus, $\lim_{n\to\infty}\frac{\log(n!)}{\log(n^n)} = 1 > 0 \implies \log(n!) = \Theta\left(\log(n^n)\right)$. ■

14. $f = \Theta(g)$

Proof. Denote the $n^{th}-$Harmonic number by $H_n := \sum_{k=1}^{n}\frac{1}{k}$. Consider the left and right Riemann sums of $y = \frac{1}{x}$ for any $n \in \mathbb{N}$ as follows

Left Riemann sum = $1 + \frac{1}{2}1 + \frac{1}{3} + \cdots + \frac{1}{n-1} \ge$ Area under $y = \frac{1}{x}$ on $[1,n] = \int_1^n \frac{1}{x}dx = \log n \ge \frac{1}{2}1 + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n} =$ Right Riemann sum

Thus, we have $\forall n \in \mathbb{N}, H_n - \frac{1}{n} \ge \log n \ge H_n - 1$. Hence, we obtain $\lim_{n\to\infty} H_n - 0 \ge \lim_{n\to\infty}\log n \ge \lim_{n\to\infty} H_n - 1 \implies \lim_{n\to\infty}\frac{H_n}{\log n} \ge 1$ from the first inequality and also $\lim_{n\to\infty}\frac{H_n}{\log n} \le 1 + \lim_{n\to\infty}\frac{1}{\log n} = 1$. So $\lim_{n\to\infty}\frac{H_n}{\log n} = 1$ and as a result $H_n = \Theta(\log n)$. ■

15. $f = \Omega(g)$

Follows from the identity $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$.

Problem 2

Points:

Given a function $f(n) = \Theta(n^d\cdot\log^s n)$ and a recurrence relation of the form $\begin{cases} T(n) = aT(\frac{n}{b}) + f(n) \\ T(1) = 1 \end{cases}$

the generalized master theorem yields the closed form $T(n) = \begin{cases} \Theta(n^d\cdot\log^{s+1} n) & d = \log_b a \\ \Theta(n^{\log_b a}) & d < \log_b a \\ \Theta(n^d\cdot\log^s n) & d > \log_b a \end{cases}$

1. $T(n) = 16\cdot T(n/2) + 100\cdot n^2$. Here, $a = 16, b = 2, d = 2, s = 0,$ and $f(n) = 100\cdot n^2 = \Theta(n^2)$. Since $\log_b a = \log_2 16 = 4 > 2 = d$, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^4)$.

2. $T(n) = 4\cdot T(n/2) + 1000\cdot n^2$. Here, $a = 4, b = 2, d = 2, s = 0,$ and $f(n) = 1000\cdot n^2 = \Theta(n^2)$. Since $\log_b a = \log_2 4 = 2 = d$, $T(n) = \Theta(n^d\cdot\log^{s+1} n) = \Theta(n^2\cdot\log n)$.

3. $T(n) = 8\cdot T(n/2) + 10\cdot n^{3.5}$. Here, $a = 8, b = 2, d = 3.5, s = 0,$ and $f(n) = 10\cdot n^{3.5} = \Theta(n^{3.5})$. Since $\log_b a = \log_2 8 = 3 < 3.5 = d$, $T(n) = \Theta(f(n)) = \Theta(n^{3.5})$.

4. $T(n) = 2\cdot T(n/2) + n\cdot\log n$. Here, $a = 2, b = 1, d = 1, s = 1,$ and $f(n) = n\cdot\log n = \Theta(n\cdot\log n)$. Since $\log_b a = \log_2 2 = 1 = d$, $T(n) = \Theta(n^d\cdot\log^{s+1} n) = \Theta(n\cdot\log^2 n)$.

5. $T(n) = 8 \cdot T(n/2) + n^{1.5} \cdot \log^2 n$. Here, $a = 8$, $b = 2$, $d = 1.5$, $s = 2$, and $f(n) = \Theta(n^{1.5} \cdot \log^2 n)$. Since $\log_b a = \log_2 8 = 3 > 1.5 = d$, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$.

Problem 3 **Points:**

Assume $f(n) = O(g(n))$. Thus, $\exists c > 0, \exists N \geq 0, \forall n \geq N, f(n) \leq c \cdot g(n)$. In addition, we assume that both f and g are non-negative functions.

1. False*

Counterexample. The statement is false in general, but it is true if and only if we assume that $\exists M \in \mathbb{N}$ such that $\forall n \geq M$, $g(n) > 1$. Otherwise, a counterexample can be provided by assigning $g = 1$ and $f = 3$ to be constant functions. Then $3 = O(1) \implies f = O(g)$, but $\log(g) = \log(1) = 0$; thus, we cannot express $\log(f) \leq c \cdot \log(g)$ for any $c > 0$ because $\log(f) = \log(3) > 0$. In general, this counterexample can be extended to all functions g that are eventually constant at 1.

Proof. *Now suppose that we assume that $\exists M \in \mathbb{N}$ such that $\forall n \geq M, g(n) > 1$. Also suppose that $g(n)$ increases monotonically without bound. Hence, $\exists K \in \mathbb{N}$, such that $\forall n \geq K$, we have $g(n) \geq c > 0$. Let $N_1 = max\{N, M, K\}$. Then $n \geq N_1 \implies n \geq N$, $n \geq M$, and $n \geq K \implies f(n) \leq c \cdot g(n) \leq g(n) \cdot g(n) = (g(n))^2 \implies \ln f(n) \leq 2 \cdot \ln g(n)$. Note that the last implication holds because $\ln(\cdot)$ is an increasing function and both sides of the inequality are **strictly positive**. Thus, with $c' = 2$ and $N' = N_1 \geq 0$, we have $\forall n \geq N'$, $\ln(f(n)) \leq c' \cdot \ln(g(n))$ and this is possible because $g(n) > 1$ for all $n \geq M \implies \ln(g(n)) > 0$. Conclusively, $\ln f(n) = O(\ln g(n))$. ■

2. False

Counterexample. Let $f(n) = \log_2(n^2) = 2 \cdot \log_2(n)$ and $g(n) = \log_2(n)$. Then $f(n) = 2 \cdot g(n)$ and clearly $f = O(g)$ with $c = 2$ and $N = 1$. However, $2^{f(n)} = 2^{\log_2(n^2)} = n^2$ and $2^{g(n)} = 2^{\log_2(n)} = n$. Given any $c' > 0$ and $N' \geq 0$, choosing $n \in \mathbb{N}$ such that $n > c' > 0$ and $n \geq N'$, yields $n^2 > c' \cdot n \implies n^2 \neq O(n) \implies 2^{f(n)} \neq O(2^{g(n)})$.

3. True

Proof. Let $c' = c^2 > 0$. Then, $\forall n \geq N$, we have $f(n) \leq c \cdot g(n) \implies (f(n))^2 \leq c^2 \cdot (g(n))^2 \implies (f(n))^2 \leq c' \cdot (g(n))^2$, because $(\cdot)^2$ is an increasing function. Ergo, $(f(n))^2 = O\big((g(n))^2\big)$. ■

Problem 4 **Points:**

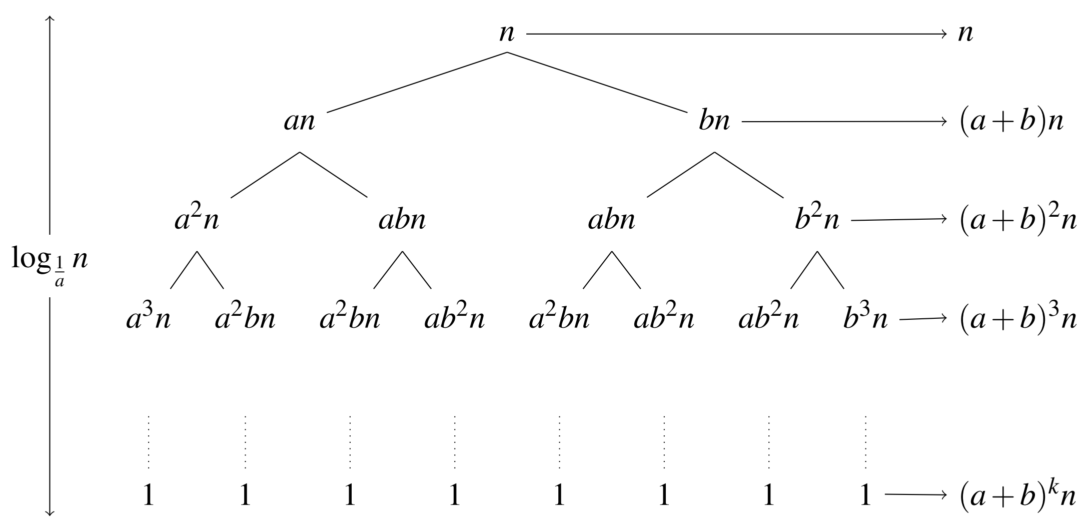
Let $a, b \geq 0$ be given. Consider $T(n) = \begin{cases} \Theta(n) + T(a \cdot n) + T(b \cdot n), & \text{if } n > 1 \\ T(1), & \text{if } n = 1 \end{cases}$

Observe that when $a = 0$ and $b = 0$, we get $T(n) = \Theta(n) + T(0) + T(0) = \Theta(n)$ because $T(0) = 0$. Note that in this case, $a + b = 0 < 1$ and so the claim holds true. The case when $(a, b) \in \{(1,0), (0,1)\}$ is degenerate because $T(n)$ has no solution. Thus, we may assume that $a, b \in (0, 1)$. Let $n \in \mathbb{N}$ be arbitrary. Since $0 < a < 1$, so $\frac{1}{a} > 1 \implies \exists k \in \mathbb{R}^+$, $\left(\frac{1}{a}\right)^k = \frac{1}{a^k} = n$. So $\log_{\frac{1}{a}} n = k$.

Now, observe that

$$
\begin{aligned}
T(n) &= \Theta(n) + T(a\cdot n) + T(b\cdot n) \\
&= \Theta(n) + [\Theta(a\cdot n) + T(a^2\cdot n) + T(a\cdot b\cdot n)] + [\Theta(b\cdot n) + T(a\cdot b\cdot n) + T(b^2\cdot n)] \\
&= \Theta((a+b)\cdot n + n) + T(a^2\cdot n) + 2\cdot T(a\cdot b\cdot n) + T(b^2\cdot n) \\
&= \Theta([(a+b)^2 + (a+b) + 1]\cdot n)) + T(a^3\cdot n) + 3\cdot T(a^2\cdot b\cdot n) + 3\cdot T(a\cdot b^2\cdot n) + T(b^3\cdot n) \\
&\quad\vdots \\
&= \Theta\left(\sum_{i=0}^{k}(a+b)^i\cdot n\right) + \sum_{i=0}^{k+1}\binom{k+1}{i}T(a^{k+1-i}\cdot b^i\cdot n)
\end{aligned}
$$

Figure 1: Recursion tree for $T(n)$



Proof. When the base case $T(a^k\cdot n) = T(1) = 1$ is reached, the accumulated combining cost is $\Theta(\sum_{i=0}^{k}(a+b)^i\cdot n)$. This is illustrated in Figure 1 above.

1. Suppose $a+b<1$. Then the first term in the sum $\sum_{i=0}^{k}(a+b)^i\cdot n$ dominates over the rest of the terms and so we have $T(n) = \Theta((a+b)^0\cdot n) = \Theta(n)$.

2. Suppose $a+b=1$. So $T(n) = \Theta(\sum_{i=0}^{k}(a+b)^i\cdot n) = \Theta(n\cdot\sum_{i=0}^{k}1) = \Theta(n\cdot(k+1))$ $= \Theta\left(n\cdot(\log_{\frac{1}{a}} n + 1)\right) = \Theta(n\cdot\log n + n) = \Theta(n\cdot\log n)$. (Since $n = o(n\cdot\log n)$).

Hence, $T(n) = \begin{cases}\Theta(n), & \text{if } a+b<1\\ \Theta(n\cdot\log n), & \text{if } a+b=1\end{cases}$ ∎

Problem 5

Points:

1. $\Theta(n^2)$ because the inner loop runs $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ times.

```
for i := 1 to n do
    j := i;
    while j < n do //runs (n-1)+(n-2)+...+1 times
        j := j + 5; //runs in constant time
    end
end
```

2. $\Theta(n^2)$ because the inner loop runs $\sum_{i=1}^{\lfloor \frac{n}{4} \rfloor}(n-4i+1) \approx \frac{n^2}{8} - \frac{n}{4}$ times, assuming WLOG that $\frac{n}{4} \in \mathbb{N}$.

```
for i := 1 to n do
    for j := 4i to n do //runs (n-3)+(n-7)+...(n-4i+1)+...+(n-4.[n/4]+1) times
        s := s + 2; //runs in constant time
    end
end
```

3. $\Theta\left(n^{\frac{6}{5}}\right) = \Theta(n^{1.2})$ because the inner loop runs $\sum_{i=1}^{\lfloor \sqrt[5]{n} \rfloor}(n - i^5) \approx n \cdot \sqrt[5]{n} - p(\sqrt[5]{n}) \approx \frac{5}{6}n^{1.2}$ times, where $p(n) = \sum_{i=1}^{n} i^5 = \frac{1}{12} \cdot n^2 \cdot (n+1)^2 \cdot (2n^2 + 2n - 1)$.

```
for i := 1 to n do
    j := n;
    while i^5 < j do //runs (n-1^5)+(n-2^5)+...+(n-[n^(1/5)]^5) times
        j := j - 1; //runs in constant time
    end
end
```

4. $\Theta(n \cdot \log(\log n))$. First, observe that the inner while loop is entered only when $i \geq 3$. Given some $i \in \{1, 2, \cdots, n\}$, the inner while loop runs k times, where k is the smallest non-negative integer satisfying $j = 2^{4^k} \geq i$. Now, $2^{4^k} \geq i \implies 4^k \geq \log_2 i \implies k \geq \log_4(\log_2 i)$. Ergo, the iterations performed by the inner loop are approximately $\sum_{i=3}^{n} \lceil \log_4(\log_2(i)) \rceil$. But this expression is bounded by $\lceil \log_4(\log_2(n)) \rceil \leq \sum_{i=3}^{n} \lceil \log_4(\log_2(i)) \rceil \leq n \cdot \lceil \log_4(\log_2(n)) \rceil$. Since $\lim\limits_{n\to\infty} \frac{\lceil \log_4(\log_2(n)) \rceil}{n \cdot \log\log n} = 0$ and $\lim\limits_{n\to\infty} \frac{n \cdot \lceil \log_4(\log_2(n)) \rceil}{n \cdot \log\log n} = \frac{1}{\log(4)}$; thus, $\Theta(\log\log n) \leq \Theta\left(\sum_{i=3}^{n} \lceil \log_4(\log_2(i)) \rceil\right) \leq \Theta(n \cdot \log\log n)$ by the squeeze lemma for sequences. By choosing the worst case, we get that the running time of the code snippet is $\Theta(n \cdot \log\log n)$.

Alternative way to see this: Given $k = \log_4(\log_2(n))$, or equivalently $2^{4^n} = k$, observe that the total iterations of the inner while loop are $1 \cdot (2^{4^1} - 2^{4^0}) + 2 \cdot (2^{4^2} - 2^{4^1}) + \cdots + k \cdot (2^{4^k} - 2^{4^{k-1}})$ $= k \cdot 2^{4^k} - \left(2^{4^0} + 2^{4^1} + \cdots + 2^{4^{k-1}}\right)$. Since $\forall i \in \{0, 1, \cdots, k-1\}$, $2^{4^i} < 2^{4^k} = n$, we deduce that the term $k \cdot 2^{4^k} = n \cdot \log_4(\log_2(n))$ dominates in the sum above as $n \to \infty$.

```
for i := 1 to n do
    j := 2;
    while j < i do //runs in log_4(log_2(i))) time for each i
        j := j^4; //runs in constant time
        //NOTE: j = {2, 2^4, (2^4)^4 = 2^16, ..., 2^4^k, ...}
    end
end
```

Problem 6

Points:

Scheme: Exponentiation can be performed through recursive multiplication, whereby a given positive integer x can be raised to the power n, for some $n \in \mathbb{N}$, by the formula $x^n = x^{n/2} \cdot x^{n/2}$; thus, dividing the original problem of size n into two sub-problems each of size $n/2$.

Algorithm 1: Algorithm for computing x^n in $O(\log n)$ time

Function Exp-dc ($x \in \mathbb{R}$, $n \in \mathbb{N}$) **:**
- **if** $n = 1$ **then**
  - **return** x;
- **else**
  - **if** *n is divisible by* 2 **then**
    - $n' := \frac{n}{2}$ // $\Theta(1)$
    - $x' :=$Exp-dc(x, n')
    - **return** $x' * x'$ // $\Theta(1)$
  - **else**
    - $n' := \frac{n-1}{2}$ // $\Theta(1)$
    - $x' :=$Exp-dc(x, n')
    - **return** $x' * x' * x$ // $\Theta(1)$
  - **end**
- **end**

Correctness: Let $x \in \mathbb{R}$ be arbitrary. We employ induction on n.

[Base case] When $n = 0$, Exp-dc$(x, 0) = 1 = x^0$. When $n = 1$, Exp-dc$(x, 1) = x = x^1$.

[Inductive step] Let $n \in \mathbb{N}$. Suppose that $\forall k < n$, Exp-dc$(x, k) = x^k$. If n is odd, then Exp-dc$(x, n) =$ Exp-dc$(x, \frac{n-1}{2}) \times$ Exp-dc$(x, \frac{n-1}{2}) \times x = x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}} \times x$ (from induction hypothesis) $= x^n$. If n is even, then Exp-dc$(x, n) =$Exp-dc$(x, \frac{n}{2}) \times$ Exp-dc$(x, \frac{n}{2}) = x^{\frac{n}{2}} \times x^{\frac{n}{2}} = x^n$. So $\forall n \in \mathbb{N}$, Exp-dc$(x, n) = x^n$; thus, the algorithm is correct. ■

Running time: Recursive formula for running time of Exp-dc is $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$. By the master theorem, $d = 0 = \log_2 1 \implies T(n) = \Theta(n^d \log n)$. Since $d = 0$, so $T(n) = \Theta(\log n)$. (Note: Divisibility of n by 2 can be checked in $\Theta(1)$ time.) In addition, $T(1) = 1$ because x is merely returned when $n = 1$. Hence, the running time of Exp-dc is $\begin{cases} T(n) = \Theta(\log n) & \text{if } n > 1 \\ T(1) = 1 & \text{if } n = 1 \end{cases}$

Problem 7

Points:

Scheme: The most intuitive way to approach this problem is to merge the two sorted arrays $A[1\cdots m]$ and $B[1\cdots n]$, of sizes m and n respectively, into a single array $A\cup B$ of size $m+n$ as this allows us to return the median as $(A\cup B)[\lceil \frac{m+n}{2} \rceil]$ if $m+n$ is odd and the average of the middle two elements when $m+n$ is even. However, merging two sorted arrays takes $O(m+n)$ time; ergo, we try to devise a different, more efficient algorithm. Recall that the BinarySearch(A,k) algorithm, which returns the index of element k in array A, if it is present, runs in $O(\log |A|)$ time. Thus, it is natural for us to tailor an algorithm that implements BinarySearch to some extent since we seek an algorithm that runs in $O(\log(m+n))$ time.

First, observe that 'finding the median' of $A\cup B$ is the same notion as partitioning it into two, disjoint left and right halves of equal sizes. Now, a partition of $A\cup B$ 'induces' a partition onto each of A and B (since $A, B \subseteq A\cup B$). Explicitly, if we have $A = \{a_1, a_2, \cdots a_m\}$ and $B = \{b_1, b_2, \cdots b_n\}$, then a partition of $A\cup B$ can be visualized as the two disjoint sets $X = \{a_1, a_2, \cdots a_i, b_1, b_2, \cdots b_j\}$ and $Y = \{a_{i+1}, \cdots a_m, b_{j+1}, \cdots b_n\}$ for some $1 \le i \le m$ and $1 \le j \le n$. In the case of the median, this partition has the special property that $\forall x \in X, \forall y \in Y, x \le y$ and also that $|X|$ and $|Y|$ differ by at most one (0 if $m+n$ is even and 1 if $m+n$ is odd). In particular, we are only concerned with $a_i \le b_{j+1}$ and $b_j \le a_{i+1}$ because A and B are sorted $\mathbf{(1)}$. Now, if $m+n$ is even, then we only need to look at the middle 4 elements $\{a_i, a_{i+1}, b_j, b_{j+1}\}$. In this case, $max\{a_i, b_j\}$ and $min\{a_{i+1}, b_{j+1}\}$ would be the middle two elements and so the median would be $\frac{1}{2}\cdot(max\{a_i, b_j\} + min\{a_{i+1}\})$. If $m+n$ is odd, then the median is clearly $max\{a_i, b_j\}$.

Hence, the problem of finding the median of $A\cup B$ now reduces to finding appropriate values of i and j such that the above two inequalities in $\mathbf{(1)}$ hold true, because if they do, then the median can be computed in $O(1)$ time using the formulae given above. To find i and j, we apply BinarySearch on one of A or B, preferably on the one with smaller cardinality as this will reduce the number of iterations needed to find the required pair $(i,\ j)$. The following pseudo-code illustrates this scheme.

Algorithm 2: Algorithm for finding the median of the union of two sorted arrays A and B, of sizes m and n respectively, in $O(\log(m+n))$ time

Function `MedianSortedArrays(`$A[1\cdots m]$, $B[1\cdots n]$`)`**:**

- **if** $|A| > |B|$ **then**
  - **return** *MedianSortedArrays*(B, A) `// This will ensure that` $|A| \le |B|$
- **end**
- init $m = |A|$
- init $n = |B|$
- `/*` *start* `and` *end* `are the bounds of our current search range on` A `*/`
- init $start = 1$
- init $end = m$
- **while** $start \le end$ **do**
  - `/* Partition` A `at the midpoint of the current search range */`
  - $i := \frac{(start+end)}{2} - 1$
  - `/* Add 1 to` $m+n$ `to account for both the even and odd cases. This formula partitions` B `at` j `such that the cardinalities of the two 'sides' of the partition differ by at most 1 */`
  - $j := \frac{(m+n+1)}{2} - i - 1$
  - `/* Checking if the formulae from` **(1)** `hold and accounting for even and odd cases */`
  - **if** $A[i] \le B[j+1]$ *and* $B[j] \le A[i+1]$ **then**
    - **if** 2 *divides* $m+n$ **then**
      - **return** $\frac{1}{2} \cdot (max\{A[i], B[j]\} + min\{A[i+1], B[j+1]\})$
    - **else**
      - **return** $max\{a_i, b_j\}$
    - **end**
  - `/* Need to shift search region to the left or right in` A `if` **(1)** `is not satisfied */`
  - **else if** $A[i] > B[j+1]$ **then**
    - $end := i - 1$
  - **else**
    - $start := i + 1$
  - **end**
  - `/* Note: The case when the search region becomes empty, either in` A `or in` B`, should be dealt with separately. */`
- **end**

Correctness: Can be verified by fixing m and an induction proof on n. The other case follows by symmetry.

Running time: Running time of MedianSortedArrays is $T(n) = O(min\{\log_2|A|, \log_2|B|\}) = O(\log(m+n))$ because we are essentially performing a modified version of BinarySearch on the array with the smaller cardinality; instead of merely computing the new search regions or returning the index of the found element as in BinarySearch, we are performing a finite number of computations, all of which are clearly $O(1)$. Also, observe that at each iteration of the while-loop, the search region is halved similar to BinarySearch. Lastly, $min\{\log_2|A|, \log_2|B|\} = O(\log(|A|+|B|)) = O(\log(m+n))$ since $|A| \leq |A|+|B| = m+n$, and similarly for $|B|$.

Problem 8

Points:

Scheme: One way to approach this problem would be to sort the array of points P in ascending order of $\|p_i\|$, where the *norm* $\|p_i\|$ of point p_i is defined to be its distance $\sqrt{x_i^2+y_i^2}$ from the origin in the Euclidean metric on $\mathbb{R}^2$. However, sorting, in particular the merge-sort algorithm, takes $O(n \cdot \log n)$ time, while we seek an algorithm that runs in linear time. Recall that given an array A and an integer $m \in [1, |A|]$, the Selection(A, m) algorithm returns the m^{th} smallest element of A, by partitioning A into sub-arrays of length 5, and its running time is $\Theta(n)$.

Now, in the setting of the current problem, let $Dist[\ ] = \{\|p_i\| \mid p_i \in P\}$ be an array containing the distances of the points p_i from the origin that is in one-to-one correspondence with P. However, to utilize the Selection algorithm here, we **need to work under the assumption that all the points in P are distinct, and more importantly, the n elements of $Dist[\ ]$ are also distinct**. We first find $N =$ Selection(P, k), the k^{th} closest point to the origin in P, where the ordering on the points in P is given by their norm. Then, we form a list L consisting of N and all points $p_i \in P$ such that $Dist[p_i] < Dist[N]$ (inequality is strict by the distinctness of distances). By construction, we are guaranteed to have the first k closest points in L.

Algorithm 3: Algorithm for finding the closest k points to the origin in an array P of points in $\mathbb{R}^2$, in $O(n)$ time

Function `ClosestKPoints`$(P = [p_1, p_2, ..., p_n] \subset \mathbb{R}^2,\ k \in [1, n])$**:**

    init $L = \emptyset$ // $|L| = k$
    init $Dist = \emptyset$ // $|Dist| = n$
    /* $O(n)$ (∵ computing $\|p_i\|$ is $O(1)$) */
    for $i = 1$ **to** n **do**
        Dist[i]=$\|p_i\|$
    end
    init $kMaxDist$:=Selection$(Dist, k)$ // $O(|Dist|) = O(n)$
    init $counter = 1$
    /* $O(n)$ */
    for $i = 1$ **to** n **do**
        if $Dist[i] \leq kMaxDist$ **then**
            L[$counter$] = P[i]
            $counter = counter + 1$
            continue
        end
    end
    return L

Note that in the above algorithm, we are guaranteed to have $counter \leq k+1$ inside the second for-loop, during all iterations, because there are exactly k points in P which are at least as close to the origin as the k^{th} closest point, the latter of which is at a distance of *kMaxDist* from the origin. $counter = k+1$ when we have found all the required points and all the following points in P are farther from the origin than we want.

Correctness: Obvious by construction of algorithm.

Running time: Running time of ClosestKPoints is $T(n) = O(n)$ because it does not involve any recursive calls to itself, but only calls the Selection algorithm which has running time $O(n)$ and involves a for-loop which also runs in $O(n)$.