Problem 1

(a) Let $M_1, M_2, \ldots$ be an enumeration of polynomial-time Turing Machines. Since $L \notin \text{P}$, for each machine $M_i$ there exist infinitely many $x$ such that $M_i$ fails to solve $x$ correctly for $L$. The distribution $\mu_{L,n}$ will be designed in a way so that it gives substantial probability to such $x$. Then if we think of $M_i$ as a heuristic, it will fail with non-negligible probability.

Let’s look at a particular instance length $n$ and the first $n$ machines $M_1, \ldots, M_n$. If the machine $M_i$ fails to solve some $x$ of length $n$ correctly, $\mu_{L,n}$ will assign probability about $1/n$ to this $x$. This will ensure that for every machine $M_i$, a lot of probability will fall on instances that $M_i$ does not solve correctly.

More formally, we have

$$\mu_{L,n}(x) = \begin{cases} p_n, & \text{if } x \text{ is the first string of length } n \text{ such that } M_i(x; 1/n^2) \neq L(x) \text{ for some } i \leq n; \\ 0, & \text{otherwise.} \end{cases}$$

The number $p_n$ is chosen so that $\mu_{L,n}$ is a probability distribution, namely the probabilities are distributed evenly among all the instances of the first type. Note that $p_n \geq 1/n$ since at most $n$ strings are ”covered” by nonzero probability in the above definition.

Now, for every potential heuristic algorithm $M_i$ for $L$, let $x^*$ be the first $x$ of length $n \geq i$ such that $M_i(x^*; 1/n^2) \neq L(x^*)$. But $\mu_{L,n}(x^*) = p_n \geq 1/n$, therefore

$$\Pr_{x \sim \mu_{L,n}}[M_i(x; 1/n^2) \neq L(x)] \geq 1/n^2$$

so $M_i$ cannot be a heuristic algorithm for $L$.

(b) The “if” direction is true for every ensemble $\mu$. For the “only if” part we need to come up with a $\mu$ such that if $L \in \text{NP} - \text{P}$ then $(L, \mu)$ doesn’t have a polynomial-time heuristic. Let $N_1, N_2, \ldots$ be an enumeration of nondeterministic polynomial-time turing machines. Define $\mu$ as follows.

$$\mu_n(x) = \frac{\mu_{L(N_1),n}(x) + \cdots + \mu_{L(N_n),n}(x)}{n},$$

where $L(N_i)$ is the language defined by machine $N_i$ and $\mu_{L(N_i),n}$ is defined as in part (a).

Now suppose $M_i$ is a potential heuristic algorithm for $L$. Let $x^*$ be the first string of length $n \geq i$ such that $M_i(x^*; 1/n^2) \neq L(x^*)$. Then $\mu_{L,n}(x^*) \geq 1/n$ and therefore $\mu_n(x^*) \geq 1/n^2$. However,

$$\Pr_{x \sim \mu_n}[M_i(x; 1/n^2) \neq L(x)] \geq 1/n^2$$

so $M_i$ is not a heuristic algorithm for $L$. 

1
Problem 2

(a) Suppose, by way of contradiction, that \( \mu \) is polynomial time computable. Therefore, there is an efficient procedure that on input \( x \) computes \( \mu_n(x) \). Let \( \nu \) be the uniform distribution. To distinguish \( \mu \) from \( \nu \), consider the following test \( T(\cdot) \). On input \( x \), if \( \mu_n(x) > 0 \) then output 1, otherwise output 0. Since for at least half the strings we have \( \mu_n(x) = 0 \), it follows that \( \Pr_{X \sim \{0,1\}^n}[T(G_n(X)) = 1] - \Pr_{Y \sim \{0,1\}^m}[T(Y) = 1] \geq 1/2 \). This contradicts the assumption that \( G_n \) is a pseudorandom generator.

(b) To prove that \( \text{PComp} = \text{PSamp} \) implies \( \text{P} = \text{P}^{\#P} \), recall that there is a randomized algorithm \( R \) which given a DNF formula uniformly samples a satisfying assignment in expected polynomial time. Consider now an algorithm that first picks a random formula \( \varphi \) of length \( n \), and then runs \( R \) to produce \(( \varphi, R(\varphi) \)) . This algorithm can be viewed as a polynomial-time sampler for pairs \(( \varphi, a )\) (for simplicity assume \( |\varphi| = |a| = n \)) from the distribution

\[
\mu_{2n}(\varphi, a) = \begin{cases} 
1/(2^n \cdot \#\text{SAT}(\varphi)), & \text{if } a \text{ is a satisfying assignment for } \varphi; \\
0, & \text{otherwise};
\end{cases}
\]

Under the assumption \( \text{PComp} = \text{PSamp} \), there is a polynomial-time algorithm that on input \(( \varphi, a )\) computes the value \( \mu_{2n}(\varphi, a) \). We can use this algorithm to solve \( \#\text{DNF} \) exactly as follows: On input \( \varphi \), first find an arbitrary satisfying assignment \( a \) for \( \varphi \) (this can be done in linear time), then output the value \( 1/(2^n \cdot \mu_{2n}(\varphi, a)) = \#\text{SAT}(\varphi) \). Since \( \#\text{DNF} \) is \( \text{P} \)-complete it follows that \( \text{P} = \text{P}^{\#P} \).

One can prove a statement in the opposite direction if the sampling algorithm \( S \) always runs in polynomial time. Then there is a polynomial-time verifier \( A \) that takes input \( x \) of length \( n \) and potential witness \( r \) and accepts when \( S(1^n, r) \leq x \) (meaning that when the sampling algorithm uses \( r \) as its randomness, it outputs a string that is lexicographically at most \( r \)). Then

\[
\overline{\mu}_n(x) = |\{ r, |r| = p(n) : M(x, r) \text{ accepts} \}|/2^{p(n)}.
\]

where \( S(1^n) \) uses \( p(n) \) bits of randomness. If \( \text{P} = \text{P}^{\#P} \) this quantity is clearly computable in polynomial time.

Problem 3

Let \( A' \) be an average polynomial-time algorithm with running time \( t_{A'}(x) \) on input \( x \), which for some constant \( c \) satisfies \( E_{x \sim \mu_n}[t_{A'}(x)^{1/c}] = O(n) \). By Markov’s inequality for every \( \varepsilon > 0 \) we have

\[
\Pr[t_{A'}(x)^{1/c} > O(n/\varepsilon)] < \varepsilon.
\]

To construct an algorithm \( A \) with the desired properties, we run \( A' \) for \( O((n/\varepsilon)^c) \) steps, and if it halts we output the answer, otherwise we output “fail”. We have

\[
\Pr[A(x, \varepsilon) = \text{“fail”}] = \Pr[t_{A'}(x) > O((n/\varepsilon)^c)] = \Pr[t_{A'}(x)^{1/c} > O(n/\varepsilon)] < \varepsilon
\]

as desired.
For the converse, suppose that
\[ \Pr_{x \sim \mu_n} [A(x; \varepsilon) = \text{“fail”}] < (n/\varepsilon)^c \]
for every \( \varepsilon > 0 \). We use \( A \) to construct an average polynomial-time algorithm \( A' \) as follows: On input \( x \), first try running \( A(x; 1/2) \). This should take care of half the inputs. If \( A \) fails, try running \( A(x; 1/4) \). This should take care of half the remaining inputs, and so on. More formally,

\[
A'(x)
\]
1 \( k \leftarrow 0 \)
2 \( \textbf{repeat} \ k \leftarrow k + 1 \)
3 \hspace{1em} \text{answer} \leftarrow A(x, 2^{-k}) \)
4 \hspace{1em} \textbf{until} \ \text{answer} \neq \text{“fail”} \)
5 \( \textbf{return} \ \text{answer} \)

Let \( S_k \) be the set of all inputs of length \( n \) that are solved in the \( k \)th iteration of this algorithm. Then \( \Pr_{x \sim \mu_n}[x \in S_k] \leq 2^{-k+1} \), because iteration \( k - 1 \) has solved all but a \( 2^{-(k+1)} \) fraction of inputs. Also, if \( x \in S_k \) then the running time \( t_{A'}(x) \) is at most \( \sum_{i=1}^{k} ((n \cdot 2^i)^c + O(1)) = O((n \cdot 2^k)^c) \).

\( A' \) is an average polynomial-time algorithm since

\[
\mathbb{E}_{x \sim \mu_n}[t_{A'}(x)^{1/2}] = \sum_{k=1}^{\infty} \mathbb{E}_{x \sim \mu_n}[t_{A'}(x)^{1/2} | x \in S_k] \cdot \Pr[x \in S_k]
\leq \sum_{k=1}^{\infty} O((n \cdot 2^k)^{c/2}) \cdot 2^{-k+1}
= \sum_{k=1}^{\infty} O(n^{1/2} \cdot 2^{-k/2}) = O(n^{1/2}).
\]

Now let \( R \) be a reduction from \((L, \mu)\) to \((L', \mu')\), and let \( p(n) \) be the polynomial associated with \( R \). If \( A' \) is an algorithm for \((L', \mu')\), define the algorithm \( A \) for \((L, \mu)\) as \( A(x; \varepsilon) = A'(R(x); \varepsilon/p(n)) \). It can be shown (see the proof of theorem 7 in the notes) that \( \Pr[A(x; \varepsilon) = \text{“fail”}] \leq \varepsilon \).

**Problem 4**

Observe that a graph \( G \) has a cycle of odd length if and only if there is an edge \((u, v)\) for which there is also a path of even length between \( u \) and \( v \). Furthermore, there is a path of even length between two nodes \( u, v \in V(G) \) if and only if \((G^2, u, v) \in USTCON \). Consider now the following algorithm.

\[
S(G)
\]
1 \( \textbf{for} \ \text{each edge} \ (u, v) \ \text{in} \ G \)
2 \hspace{1em} \textbf{do if} \ \( (G^2, u, v) \in USTCON \)
3 \hspace{2em} \textbf{then} \ \textbf{reject} \)
4 \( \textbf{accept} \)
By the above discussion, this algorithm accepts if and only if $G$ is bipartite. The algorithm can also be made to use logarithmic space. The only problem is that we cannot afford to construct $G^2$ and feed its description to our subroutine for $USTCON$. However, we can decide if there is a path of length two between two nodes $u, v \in G(V)$, i.e. if $(u, v)$ is an edge in $G^2$, just by using the description of $G$ and logarithmic space (check if there is a $w$ such that $(u, w)$ and $(w, v)$ are both edges of $G$). Hence, every time the subroutine $USTCON$ needs to know if $(u, v)$ is an edge of $G^2$, we can answer in logarithmic space.