1. Theory of Computation

(A) If **PLem** is true, then by applying it to infinite regular languages, we know that **PLem** cannot be vacuously satisfied. In other words, whatever N is chosen by **PLem**, there must exist strings x, y, z satisfying the conditions of **PLem**. We can now show that **PLem'** holds by choosing $N_1 = |x| + |z|$ and $N_2 = |y|$ where |..| denotes cardinality of a string.

(B) We can use the Pumping Lemma here (the conventional one i.e. **PLem**). Let the language be regular, and let M be the constant of pumping lemma. We assume that the function func is not bounded so there should exists at least one integer n such that $func(n) > M$. Now, consider the string $\sigma = 0^{func(n)}\ 1^n$. Using the pumping lemma, we have $\sigma = xyz$ with $y = 0^j$, $j > 0$. Using the pumping lemma, we should have $0^{func(n)+k*j}1^n$ in the language, for all $k > 0$ — leading to a contradiction since $func(n) \neq func(n) + k * j$ since $k * j$ is positive.

(C) Consider a string $\sigma \in L$ where $\sigma = 0\ \sigma'\ 0$. The proof for $\sigma = 1\ \sigma'\ 1$ is similar and is not shown. Clearly, $num(1, \sigma') - num(0, \sigma') = 2$. This difference between number of 1s and 0s starts with 0 at the beginning of σ' and grows to 2 at the end of σ'. Since this difference changes by 1 on reading each symbol of σ' it can only grow from 0 to 2 by going through 1. Let the smallest prefix of σ' where the difference between number of 1s and 0s is 1, be σ''. Then $\sigma = xy$ where $x = 0\sigma'' \in L$, $\sigma' = \sigma'' \circ \sigma'_{rest}$ and $y = \sigma'_{rest}0 \in L$. This completes the proof for part 1.

For part 2, the grammar is given by the following rules. S is the start symbol (and the only non-terminal).

$$S \rightarrow SS \mid 0S1 \mid 1S0 \mid \epsilon$$

2. Logic and Artificial Intelligence

(A.1) We give the FOL sentences below.

1. All blogs are webpages.
   $\forall x Blog(x) \Rightarrow Webpage(x)$
2. All webpages that have dated entries and are linked to by a blog are blogs.
   $\forall x(Webpage(x) \wedge HasDatedEntries(x) \wedge (\exists y Linked(y,x) \wedge Blog(y))) \Rightarrow Blog(x)$

(A.2) We first translate the four sentences provided in the question into CNF clauses. Redundant clauses are omitted.

1. $Webpage(Steve's)$
2. $Webpage(Mary's)$
3. $HasDatedEntries(Steve's)$
4. $HasDatedEntries(Mary's)$
5. $Blog(Jim's)$
6. $Linked(Jim's, Mary's)$
7. $Linked(Mary's, Steve's)$

We also convert the relevant sentences from **A.1** into CNF

8. $\neg Blog(x) \vee Webpage(x)$
9. $(\neg Webpage(x) \vee \neg HasDatedEntries(x) \vee \neg(\exists y(Linked(y,x) \wedge Blog(y)))) \vee Blog(x)$
   $(\neg Webpage(x) \vee \neg HasDatedEntries(x) \vee (\neg Linked(y,x) \vee \neg Blog(y))) \vee Blog(x)$
   $\neg Webpage(x) \vee \neg HasDatedEntries(x) \vee \neg Linked(y,x) \vee \neg Blog(y) \vee Blog(x)$

We then add the negation of the goal to the KB as:

10. $\neg Blog(Steve's)$

and resolve to yield *nil*, proving $Blog(Steve's)$:

11. (10+9)$\neg Webpage(Steve's) \vee \neg HasDatedEntries(Steve's) \vee \neg Linked(y_1, Steve's) \vee \neg Blog(y_1)$
    $\theta = \{x_1/Steve's\}$
12. $(11+1)\neg HasDatedEntries(Steve's) \vee \neg Linked(y_1, Steve's) \vee \neg Blog(y_1)$
    $\theta = \{x_1/Steve's\}$
13. $(12+3)\neg Linked(y_1, Steve's) \vee \neg Blog(y_1)$
    $\theta = \{x_1/Steve's\}$
14. $(13+7)\neg Blog(Mary's)$
    $\theta = \{x_1/Steve's, y_1/Mary's\}$
15. (14+9)$\neg Webpage(Mary's) \vee \neg HasDatedEntries(Mary's) \vee \neg Linked(y_2, Mary's) \vee \neg Blog(y_2)$
    $\theta = \{x_1/Steve's, y_1/Mary's, y_1/x_2\}$
16. $(15+2)\neg HasDatedEntries(Mary's) \vee \neg Linked(y_2, Mary's) \vee \neg Blog(y_2)$
    $\theta = \{x_1/Steve's, y_1/Mary's, y_1/x_2\}$
17. $(16+4)\neg Linked(y_2, Mary's) \vee \neg Blog(y_2)$
    $\theta = \{x_1/Steve's, y_1/Mary's, y_1/x_2\}$
18. $(17+6)\neg Blog(Jim's)$
    $\theta = \{x_1/Steve's, y_1/Mary's, y_1/x_2, y_2/Jim's\}$

19. $(18+5) nil$
    $\theta = \{x_1/Steve's, y_1/Mary's, y_1/x_2, y_2/Jim's\}$

Note that variable renaming is needed and in this answer shown by subscripts. We resolve to nil, therefore $Blog(Steve's)$ must be true.

(B) Propositional logic requires a separate ruleset for each and every square and type of piece. FOL is more appropriate by allowing variables to substitute for explicit x, y coordinates, significantly reducing the size of the KB.

(C) Propositionalizing FOL logic means instantiating the FOL in the KB as propositional logic. For any variable in a sentence we have to create all possible instantiations. If the FOL KB contains one or more functions, a full listing of propositions would be infinitely large as a function works on any symbol to produce another unique symbol. This phenomenon causes problems in trying to decide whether a sentence is not entailed as there is no algorithmic way to tell whether the sentence is about to be entailed by the inference by instantiating terms at a deeper depth, or whether it is truly not entailed by the KB. This states that proof of an FOL KB by reduction to propositional logic is *semi-decidable.*

3. Algorithms

(A) Sorting.

Best time: $O(n)$ time if the array is already sorted in non-decreasing order.

The algorithm sorts the array in $O(n^3)$ time

In the worst case scenario (descending input), for the largest number, the swap with a[i] is followed by i consecutive comparisons before the next swap with a[i+1] is executed: $1 + 2 + ... + (n-2) = O(n^2)$ consecutive comparisons is needed to put it in place. Since the ith largest number required $O(i^2)$ consecutive comparisons in order to be put in place , the running time of mystery is $= 0^2 + 1^2 + 2^2 + ... + (n-2)^2 = O(n^3)$.

(B) Put all the vertices in a double link list as a set storage. While the double link list is not empty, extract the first vertex and do a DFS from that. During each DFS, delete every traversed vertex in the double link list. After each DFS, a component is found. Repeat this until the double link list is empty. Each vertex deletion can be done in $O(1)$ time by recording the pointer of the node containing that vertex in the double link list.

(C) Sort the disks by their radii in an ascending order. ($O(n \log n)$) Insert each disk in the increasing order of radii. Each circle can be divided as the left and right halves by the highest and lowest points. The goal is to find the intersection of each half. For each half circle, we only have to consider the half of the half, meaning the quarter circle if it is intersecting the boundary of the exisiting union. We know that the highest point is always higher than and out of the existing union. If the other end of this quarter circle is also out of the existing union, then the entire quarter circle does not have intersection. Otherwise, to find the intersection, we can use a binary search on the slab of the existing circle. Namely, putting vertical line in each "corner" (the intersection between two circles) of the existing union. They can be organized as an interval tree, then just have to find the intersection by a binary search.

(D) The situation is exactly the same if we modify the idea of vertical slab into wedges that end at the center. Apply the same technique as above, in the same time complexity.

4. Principles of Programming Languages

(A) No, function `F` is not doing an iterative computation as the associated semantic stack increases with one at each recursive call. So, the semantic stack size depends on the number of parameters. For example, the call `{F 5 3}` consists of the following computations:

```
{F 5 3}          =
5*{F 5 2}        =
5*(5*{F 5 1})        =
5*(5*(5*{F 5 0})))  =
5*(5*(5*1)))        =
5*(5*5)          =
5*25             =
125
```

(B) There are many ways to prove the equivalency of functions `F` and `G`. One way of proving this is to use invariants. We consider the state of the program: `(X, N, Acc)`. For example, here is an illustration of recursive calls and the state evolution:

```
{F 5 3}* 1       (5, 3, 1)
{F 5 2}* (5*1)       (5, 2, 5)
{F 5 1}* (5*5)       (5, 1, 25)
{F 5 0}* (5*25)      (5, 0, 125)
```

We observe that from the state `(X, N, A)`, we can go to `(X, N-1, X*A)`. In general, $\forall i$, $0 \leq i \leq$ `N`, `(X,` i`, A)` $\rightarrow$ `(X,` $i-1$`, X*A)`. So, `(X,` i`, A)` $\rightarrow$ `(X,` $i-1$ `, X*A)` $\rightarrow$ `(X,` $i-2$`,` X^2`*A)` $\rightarrow \ldots \rightarrow$ `(X,` $i-$`N,` X^N`*A)`. Now, taking i=`N`, we get that from state `(X, N, A)`, we can go to `(X, 0,` X^N`*A)`.

The invariant of `G` is: At the `M`-th recursive call, we have `A=`X^{N-M}. Here there are some recursive calls that illustrate the new version perform an iterative computation:

```
{G1 5 3   1}    =
{G1 5 2   5}    =
{G1 5 1  25}    =
{G1 5 0 125}    =
125
```

Therefore, by taking `A=1`, function `G` returns X^N as the initial call is `{G1 X N 1}`.

(C) For example, the call `{H 2 6}` corresponds to `{H1 2 6 1}`, that leads to only fours recursive calls `{H1` 2^2 `3 1}`, `{H1` 2^2 `2` 2^2`}`, `{H1` 2^4 `1` 2^2`}`, and `{H1` 2^4`0` 2^6`}`. This returns 2^6.

Let $a_k \ldots a_1$ be the binary representation of `N`. The following relationships hold:

(1) `{H X N}`=`{H1 X` $a_k \ldots a_1$ `1}`

(2) $\forall i$, $0 \leq i \leq k-1$, `{H1 X` $a_k \ldots a_1$ X^0`}`= `{H1` X^{2^i} $a_k \ldots a_{i+1}$ $X^{a_i \cdots a_1}$`}`

Relation (2) can be easily proved by induction on i.

Taking $i = k-1$ in (2), we get `{H1 X` $a_k \ldots a_1$ `1}`= `{H1` $X^{2^{k-1}}$ 0 $X^{a_k \cdots a_1}$`}`

But `{H1` $X^{2^{k-1}}$ 0 $X^{a_k \cdots a_1}$`}` $=$ $X^{a_k \cdots a_1}$ $=$ X^N

Combining all these relationships, we get the conclusion that the call `{H X N}` returns X^N.

The time complexity is simply the number of recursive calls, that is, k + number of `1`s from the binary representation of `N`. But k is $[\log_2 N]$ and the number of `1`s from the binary representation of `N` is less than $[\log_2 N]$. Therefore, the time complexity of `H` is $\mathcal{O}([\log_2 N])$.

It is easy to see that function `G` has $\mathcal{O}(N)$ time complexity. In conclusion, `H` is more efficient than `G` in terms of time complexity.