Practice Midterm I Solutions
7/21/18

Multiple Choice

1. B, C, D, E
2. (a) A, B, C; (b) B, D
3. A
4. (a) C; (b) A
5. A
6. (a) C; (b) A
7. C, D
8. E
9. A, E
10. D

Short Answers

1. There are a few valid solutions.

Soln. 1 We proceed by contradiction; suppose n is $\Omega(n^2)$. Then there exist constants $c > 0$ and $n_0 > 0$ such that $n \geq c \cdot n^2$ for all $n \geq n_0$. Dividing both sides of the equation by n gives: $1 \geq c \cdot n$ for all $n \geq n_0$. Consider $n = \max\{n_0, 1/c\} + 1$. We have that $n \geq n_0$ but this choice of n causes $1 < c \cdot n$. We have reached a contradiction; thus, our original assumption must be false.

Soln. 2 Consider the ratio $\lim_{n\to\infty} \frac{n^2}{n}$. In order for n to be $\Omega(n^2)$, this ratio in the limit must be some constant $c \geq 0$, less than infinity. However,

$$\lim_{n\to\infty} \frac{n^2}{n} = \lim_{n\to\infty} n = \infty \neq c$$

2. We proceed by induction.

   - **Inductive Hypothesis** $T(n) \leq c_1 \cdot n\log(n) + c_2$ for all n': $1 \leq n' < n$.
   - **Base Case** Suppose $T(1) = 1$. (This is a valid assumption unless stated otherwise in the problem statement.) Then, we have $T(1) = 1 \leq c_1 \cdot (1)\log(1) + c_2 = c_2$.
   - **Inductive Case** Suppose $T(n) \leq c_1 \cdot n\log(n) + c_2$ for all n': $1 \leq n' < n$. We prove it for n:

     $$\begin{aligned} T(n) &= T(n/3) + T(2n/3) + c \cdot n \\ &\leq c_1(n/3)\log(n/3) + c_2 + c_1(2n/3)\log(2n/3) + c_2 + cn \\ &= c_1(n/3)\log(n) - c_1(n/3)\log(3) + c_1(2n/3)\log(n) + c_1(2n/3)\log(2) - c_1(2n/3)\log(3) + 2c_2 + cn \\ &= c_1 n\log(n) - c_1 n\log(3) + c_1(2n/3)\log(2) + 2c_2 + cn \end{aligned}$$

     This expression is less than or equal to $c_1 \cdot n\log(n) + c_2$ as long as $n > \frac{c_2}{dc_1\log(3) - c}$ where $d = 1 - \frac{2}{3}\frac{\log(2)}{\log(3)}$. (The last statement follows from solving the inequality for n.)
   - Setting c_1 and c_2 to satisfy the inequalities required for the base case and inductive case in terms of c (the linear cost for each problem size) guarantees $T(n) = O(n\log(n))$.

Problems

---

1. (a)
```
def exponentiator(x, n):
  if n == 1:
  return x
  if n % 2 == 0:
    return exponentiator(x, n/2) ** 2
  else:
    return x * (exponentiator(x, (n-1)/2) ** 2)
```

   (b) Most answers remotely close to the following would earn full credit (even those with only a single case).

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n\%2 = 0 \\ T((n-1)/2) + 2 & \text{otherwise} \end{cases}$$

   (c) Notte that $T(n) \leq T(n/2) + 2$ for all $n > 1$. We then use Master method on this new upper bound recurrence relation: $a = 1, b = 2, d = 0$, so the algorithm is $O(\log(n))$.

2. (a)
```
def majority_tree (root, M):
  if root.left is nil:
  return M[root]
  return majority_tree(root.left, M)
         + majority_tree(root.middle, M)
         + majority_tree(root.right, M) >= 2
```

   (b)
```
def majority_tree(root, M):
  if root.left is nil:
  return M[root]
  left = majority_tree(root.left, M)
  middle = majority_tree(root.middle, M)
  if left == middle:
  return left
  return majority_tree(root.right, M)
```

(c) Same as part (b), except choose a random set of two children to explore first instead of deterministically choosing the left and middle children. Part (b) was intended to be a strong hint for part (c).

```
def majority_tree(root, M):
    if root.left is nil:
    return M[root]
    first_child, second_child, third_child = random_order({root.left, root.middle, root.right})
    first_result = majority_tree(first_child, M)
    second_result = majority_tree(second_child, M)
    if first_result == second_result:
    return first_result
    return majority_tree(third_child, M)
```

(d) In the worst case, at level i, we have inputs with 2 of one value and 1 of the other. In these cases, we have a $1/3$ chance of picking the correct two children as `first_child` and `second_child` and we only need to recurse twice on trees at level $i-1$. Otherwise, we have a $2/3$ chance of needing to recurse three times on trees at level $i-1$. This produces the following recurrence relation:

$$T(i) = 1/3 * 2 * T(i-1) + 2/3 * 3 * T(i-1) = 8/3T(i-1)$$

To determine the value of the root, we expect to inspect at most $8/3$ of its children, and $8/3$ of their children, etc. Since $n = 3^h$, we have $h = log_3(n)$. So starting at level h, we expect to inspect at most $(8/3)^h = (8/3)^{log_3(n)} = n^{log_3(8/3)} = O(n^0.9)$ leaves. This is an upper bound since we assumed a worst-case scenario for our tree.

(e) Suppose an algorithm decides not to check one of the leaves. Its possible to construct an assignment of the other leaves that requires the value of this leaf to be known, such that all of the internal vertices along the path from the root to this leaf are "split decisions."

(f) Same as the advantage of randomized quicksort vs. quicksort; an adversary cannot construct a worst-case input i.e. the randomized algorithm reduces the expected number of leaves checked by the algorithm for a worst-case input.

3. Run BFS, color every other vertex a different color, and return true if we every try to color a vertex different colors. BFS runs in $O(|V| + |E|)$-time, so this algorithm runs in $O(|V| + |E|)$-time.