Multiple-choice (35 pt.)

1. **(2 pt.)** Which of the following asymptotic bounds describe the function $f(n) = n^3$? The bounds do **not** necessarily need to be tight. **Circle all that apply.**

   $$(A)\ O(n^2\sqrt{n}) \quad (B)\ O(n^3) \quad (C)\ \Omega(n^2\sqrt{n}) \quad (D)\ \Omega(n^3) \quad (E)\ \Theta(n^3).$$

2. **(4 pt.)** Consider the following iterative algorithm that sorts an input list A.

```
def mystery(A):
  n = len(A)
  # Iterates through the last n-1 elements of the list
  for i in range(1, n):
    cur_value = A[i]
    j = i - 1
    while j >= 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j -= 1
    A[j+1] = cur_value
```

   (a) Which of the following asymptotic bounds describes the best-case runtime of this algorithm (i.e. runtime of the algorithm on a best-case input). The bounds do **not** necessarily need to be tight. **Circle all that apply.**

   $$(A)\ O(n) \quad (B)\ O(n^2) \quad (C)\ \Omega(n) \quad (D)\ \Omega(n^2) \quad (E)\ \Theta(n\log(n)).$$

   (b) Which of the following asymptotic bounds describes a tight bound for the worst-case runtime of this algorithm (i.e. runtime of the algorithm on a worst-case input). **Circle all that apply.**

   $$(A)\ O(n) \quad (B)\ O(n^2) \quad (C)\ \Omega(n) \quad (D)\ \Omega(n^2) \quad (E)\ \Theta(n\log(n)).$$

3. **(3 pt.)** Consider the following recursive implementation of ternary search that splits the input list into thirds instead halves, like binary search.

```
def ternary_search(A, needle):
  if len(A) == 0:
    raise Exception("needle does not exist!")
  n = len(A)
  one_third = n/3
  two_thirds = 2*n/3
  if A[one_third] == needle:
    return one_third
  elif A[two_thirds] == needle:
    return two_thirds
  elif A[one_third] > needle:
    return ternary_search(A[:one_third], needle)
  elif A[two_thirds] > needle:
    return one_third + 1 + ternary_search(A[one_third+1:two_thirds], needle)
  else:
    return two_thirds + 1 + ternary_search(A[two_thirds+1:], needle)
```

Assuming the input list A can always evenly be divided into thirds, which of the following recurrence relations describes the worst-case runtime of this algorithm?

(A) $T(n) = T(n/3) + O(1)$
(B) $T(n) = T(n/3) + O(\log(n))$
(C) $T(n) = T(2n/3) + O(1)$
(D) $T(n) = T(2n/3) + O(\log(n))$
(E) $T(n) = T(n/3) + O(n)$

4. **(4 pt.)** Consider the following recurrence relation: $T(n) = 3T(n/3) + O(n^2)$.

   (a) According to Master method, which of the following describes a closed-form expression for $T(n)$?

   $$(A)\ O(n) \quad (B)\ O(n\log(n)) \quad (C)\ O(n^2) \quad (D)\ O(n^2\log(n)) \quad (E)\ O(n^2\log^2(n)).$$

   (b) Is the recursion tree resulting from this recurrence relation top-heavy, balanced, or bottom-heavy?

   $$(A)\ \text{Top-heavy} \quad (B)\ \text{Balanced} \quad (C)\ \text{Bottom-heavy}.$$

5. **(4 pt.)** Recall the recurrence relation for linear-time selection: $T(n) = T(n/5) + T(7n/10) + \Theta(n)$. Which of the following statements about the runtime of this algorithm are correct? **Circle all that apply.**

   (A) The $T(n/5)$ term in the recurrence relation represents the time required to find the median-of-medians.
   (B) The $T(n/5)$ term in the recurrence relation represents the time required to find the k^{th}-smallest element in the recursed-upon sublist.
   (C) The $\Theta(n)$ term in the recurrence relation represents the time required to find the median-of-medians.
   (D) The recurrence relation can be solved with Master method for $a = 2$, $b = 10/7$, and $d = 1$.
   (E) The $T(7n/10)$ term in the recurrence relation represents the time required to partition the list around the pivot.

6. **(5 pt.)** Consider the following algorithm that outputs a random number from 0 to $n-1$.

```
def pick(n):
  A = random.shuffle(list(range(n)))
  max_so_far = -1
  counter = 0
  for a in A:
    if a > max_so_far:
      counter += 1
      max_so_far = a
  return counter
```

(a) What is the expected runtime of the algorithm?

$$(A)\ \Theta(\log(n)) \quad (B)\ \Theta(\sqrt{n}) \quad (C)\ \Theta(n) \quad (D)\ \Theta(n\sqrt{n}) \quad (E)\ \Theta(n^2).$$

(b) What is the expected value returned from the algorithm?

$$(A)\ \Theta(\log(n)) \quad (B)\ \Theta(\sqrt{n}) \quad (C)\ \Theta(n) \quad (D)\ \Theta(n\sqrt{n}) \quad (E)\ \Theta(n^2).$$

7. **(4 pt.)** Consider a hash family consisting of a single hash function that hashes a universe $\mathcal{U}$ of whole numbers less than 1 million to one of $n = 3$ buckets according to: h(key) is the sum of the digits of the key mod 3. Which of the following **must** be true about this hash family? **Circle all that apply.**

(A) This hash family is universal for $\mathcal{U}$.

(B) This hash family is universal for the following subset of $\mathcal{U}$: $\{0, 1, 2, 10, 11, 12\}$

(C) There exists a specific worst-case input of $n = 3$ elements from $\mathcal{U}$, such that a randomly-drawn hash function from this hash family hashes all of the elements to the same bucket.

(D) For a randomly selected hash function h from the hash family, we have that $P(h(123) = h(321)) = 1$.

(E) For a randomly selected hash function h from the hash family, we have that $P(h(1) = h(2)) = 1$.

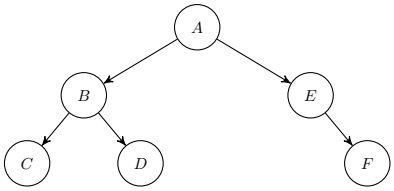
8. **(3 pt.)** Recall

$$h_{a,b}(x) = ax + b \mod p \mod n$$

where $a \in [1, p-1]$ and $b \in [0, p-1]$ with prime $p \geq |\mathcal{U}|$ describes a universal family of hash functions mapping a universe of keys $\mathcal{U}$ to values {1, 2, ..., n}. Let's call this hash family $\mathcal{H}_{\text{mod}}$. For some $u_x, u_y \in \mathcal{U}$, such that $u_x \neq u_y$, the number of hash functions from this family for which $u_x = u_y$ is at most:

$$(A)\ |\mathcal{U}|/n \qquad (B)\ n/|\mathcal{H}_{\text{mod}}| \qquad (C)\ n/|U| \qquad (D)\ 1/n \qquad (E)\ |\mathcal{H}_{\text{mod}}|/n.$$

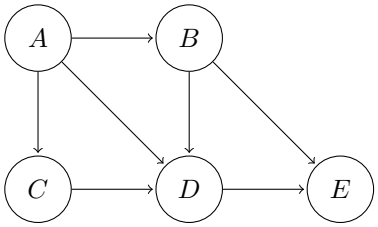
9. **(3 pt.)** Consider the following binary search tree.



Which of the following vertices must be black in **any** valid red-black coloring? **Circle all that apply.**

(A) A (B) B (C) C (D) D (E) E (F) F.

10. **(3 pt.)** You run `dfs` from A. Assuming a vertex's neighbors will be visited in alphabetical order, what will be the ordering of end times from lowest to highest.



(A) A, B, C, D, E

(B) A, B, E, C, D

(C) A, B, D, E, C

(D) E, D, B, C, A

(E) E, B, D, C, A

Short-answers (15 pt.)

---

1. **(6 pt.)** Using the definition of Big-Ω, formally prove the following statement:

   $$n \text{ is } \textbf{not } \Omega(n^2)$$

   [We are expecting: A short but rigorous proof, using the definition of Big-Ω.]

2. **(9 pt.)** Using substitution method, formally prove the recurrence relation

   $$T(n) = T(n/3) + T(2n/3) + O(n)$$

   evaluates to $O(n \log(n))$.

   [We are expecting: A short but rigorous proof, using substitution method.]

Problems (50 pt.)

1. **(Exponentiator) (12 pt.)**

   The function `exponentiator(x, n)` takes as input positive integers x and n and returns the value x^n. For example, given $x = 4$ and $n = 3$, your algorithm should return $4^3 = 64$.

   (a) **(6 pt.)** Design an $O(\log(n))$-time **divide-and-conquer algorithm** for `exponentiator`. Note that basic arithmetic operations (addition, subtraction, multiplication, division) require $O(1)$-time.

   [We are expecting: Pseudocode for the algorithm.]

   ```
   def exponentiator(x, n):
   ```

   (b) **(3 pt.)** Give the recurrence relation that describes the worst-case runtime of your algorithm.

   [We are expecting: A recurrence relation.]

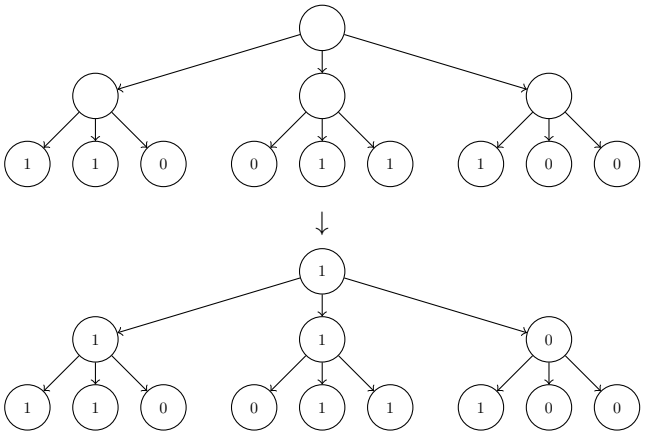
   (c) **(3 pt.)** Prove that your algorithm takes $O(\log(n))$-time.

   [We are expecting: A runtime analysis using recursion tree, substitution method, or Master method.]

2. **(Majority Tree) (22 pt.)**

Consider a complete ternary tree T i.e. a tree comprised of two types of vertices: (1) internal (not-leaf) vertices with exactly **three** children and (2) leaf vertices distance h from the root, where h represents the height of the tree. Note that no vertices can be added to this tree without increasing its height.

The function `majority_tree_root_value(T, M)` takes as input the complete ternary tree and a map `M` of its $n = 3^h$ leaves mapped to $n = 3^h$ boolean values, and returns the boolean value associated with the root. To determine the boolean value of a leaf `v`, index into the map `M[v]`. To determine the boolean value associated with an internal vertex or the root, take the majority of its children.



Given the boolean values specified for the top complete ternary tree above, the function should return 1 for the boolean value associated with the root. To determine the boolean value associated with the root, notice that 1 is the majority element of the root's children (its left and middle children are 1 while only its right child is 0), each of which was determined by taking the majority elements of their respective children.

(a) **(6 pt.)** Design a **divide-and-conquer** algorithm that implements this function that always inspects all of the leaves exactly once (i.e. for each leaf, your algorithm should index into `M` exactly once).

[We are expecting pseudocode.]

(b) **(4 pt.)** Sometimes it's possible to short-circuit recursive calls to avoid inspecting all of the leaves. For example, in the example tree, after obtaining the values for its left and middle children, the root need not recurse to its right child since its majority element is guaranteed to be 1. Design a **divide-and-conquer** algorithm that implements this short-circuiting.
[We are expecting pseudocode.]

(c) **(6 pt.)** Design a **randomized algorithm** that inspects $O(n^{0.9})$ leaves on worst-case inputs in expectation.
[We are expecting pseudocode.]

(d) **(2 pt.)** Justify that the algorithm in part (c) inspects $O(n^{0.9})$ leaves in expectation.
[We are expecting a convincing mathematical argument.]

(e) **(2 pt.)** For any implementation of `majority_tree_root_value`, including your algorithms from parts (a)-(c), there exists a set of boolean values for the leaves (in part (c), this is not a deterministic input) that requires the algorithm to inspect the boolean values of all 3^h leaves. Why must this be the case?
[We are expecting two to three sentences of explanation.]

(f) **(2 pt.)** Describe the advantage of your randomized algorithm from part (c), compared to your divide-and-conquer algorithm from part (b).
[We are expecting two to three sentences of explanation.]

3. **(Detecting Odd Cycles) (16 pt.)**

An odd cycle of a graph is a cycle with an odd number of vertices.

(a) **(12 pt.)** Design an $O(|V|+|E|)$-time algorithm that takes as input a connected undirected graph G and returns true if the graph contains an odd cycle and false otherwise. Given a vertex v, you can access its list of neighbors with `v.neighbors` in $O(1)$-time. You can also add meta-information to the vertices.

[We are expecting: Either an English description or pseudocode of the algorithm.]

```
def contains_odd_cycle(G):
  # You can call G.vertices to get the set of vertices; for each vertex v,
  # you can call v.neighbors to get its list of neighbors.
```

(b) **(4 pt.)** Prove that your algorithm takes $O(|V| + |E|)$-time.

[We are expecting: An English justification of why it takes $O(|V| + |E|)$-time.]