COMP 285 Practice Midterm Questions

The following are questions meant to help you practice, and cannot be submitted for a grade.

Important Notes

- *It is meant to give you a chance to do some practice questions after having reviewed the slides, quizzes, in-class exercises, homeworks, etc.*
- *It should give a rough sense of some ways questions might be posed, though there's no guarantee that the actual midterm will have the exact same format (at a minimum, one difference is that the actual midterm will show the point values associated with the questions).*
- *It should give a rough idea of the level of mastery expected generally, though more/less mastery may be expected for any given topic.*

Thanks for reading the notes above - the big picture thing is that I want to be sure you use this resource appropriately, while at the same time **do not neglect the many other more comprehensive resources**!

Asymptotic Analysis

1. $O(n/100 + \log(n) + 200)$ can be simplified to $O(n)$. True or False?
   True

2. $2x + x^2/2 = \Theta(x^2 + 2x + x \log(x))$. True or False?
   True

3. $x + 20 = \Omega(999)$. True or False?
   True

For questions 4 - 6, refer to the containsDuplicates pseudocode.

```
algorithm containsDuplicates
  input: size n vector of ints called vec
  output: true if vec contains duplicates, false otherwise

for i = 0...n-1
  for j = i + 1...n-1  // Notice we start at i + 1, not j
    if vec[i] == vec[j]
      return true
return false
```

4. What is the **best-case runtime** of containsDuplicates? Define n, provide a tight upper bound with Big-O, and justify your answer.
   $O(1)$, where n is the size of vec.If the first two elements of vec are the same, then we will return after a constant number of operations.

5. What is the **worst-case runtime** of containsDuplicates? Define n, provide a tight upper bound with Big-O, and justify your answer.
   $O(n^2)$, where n is the size of vec. The inner for loop will run roughly n/2 times, while the outer for loop will run roughly n times. The innermost body is $O(1)$ work, so we get $O(n * n/2 * 1) = O(n^2)$.

6. What is the **worst-case space complexity** of containsDuplicates? Define n, provide a tight upper bound with Big-O, and justify your answer.
   $O(1)$, where n is the size of vec. We are creating no new data structures that would take up more than constant space. There is also no recursion (i.e. there are no stack frames to account for).

Using the Right Tools

7. Which of the data structure implementations below have O(1) runtime on average for element insertions? Select **ALL** that apply.
   - ☐ A stack (C++: std::stack)
   - ☐ A queue (C++: std::queue)
   - ☐ A hash set (C++: std::unordered_set)
   - ☐ A hash map (C++: std::unordered_map)
   - ☐ A priority queue (C++: std::priority_queue)

8. Given a vector of n integers, where each integer is at most d away from its correct position in the sorted vector, complete the pseudocode in the box below that returns a sorted array in O(n log(d)) time.

```
algorithm sort
  input: d and an almost sorted vector vec of ints as described above
  output: the sorted vector

m = new min priority queue of size d + 1
for i = 0...d
  m.push(vec[i])
ret = new empty vector to be returned
i = d + 1
while !m.empty()
  ret.push_back(m.pop())
  if i < vec.size()
    m.push(vec[i])
    i++
return ret
```

Sorting

9. Which array of the following will RadixSort take the most number of steps on? Select **ONE.**
    a. [1, 2, 3, 4, 5, 6]
    b. [5, 43, 3, 11, 6, 9]
    c. [3, 1, 34, 3, 4, 81]
    d. [4, 4754, 4, 24, 1, 33]

10. For each of the below, explain in 1 - 2 sentences what they mean with respect to sorting.
    - Adaptive

    If a sorting algorithm is adaptive, it will run more efficiently if the array is more sorted.

    - Stable

    If a sorting algorithm is stable, elements of the same value will stay ordered relative to each other in the output. For example {1, 4, 1*, 2} → {1, 1*, 2, 4} would be a stable sort, because the star 1 is to the right of the non-starred 1 in both the input and output.

    - In-Place

    If a sorting algorithm is in-place, we only use O(1) additional space.

11. Given an array is already sorted, which sort will take the least time? Select **ONE.**
    a. Insertion Sort
    b. Quick Sort
    c. Merge Sort
    d. Selection Sort

For questions 12 - 13, refer to quickSort provided.

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = findPivot(vec)
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)

(Note: pseudocode from lecture, but the pivot is selected with “findPivot”)
```

12. Suppose findPivot is a function which finds the element that will partition the list in two (nearly) equal halves in linear time while using constant space. What is the **worst-case runtime** of quickSort in this case? Justify your answer.
    If the pivot splits the list roughly into half, it can be represented as T(n) = 2 T(n/2) + X. To find X, we note that there is O(n) work happening at each level: T(n) = 2 T(n/2) + O(n). Using Master Theorem, we see the runtime is O(n log(n)).
    OR
    If the pivot splits the list roughly into half, we will have ~log(n) levels of work, with the total amount of work happening at each level adding up to n (n on the first level, n/2 + n/2 on the second level, n/4 + n/4 + n/4 + n/4 on the third level, etc). So we can multiply n and log(n) to get O(n log(n)).

13. Challenge: what is the **worst-case space complexity** of quickSort in this case? Justify your answer.
    We create ~O(n) space at each level, but we need to keep track of how many stack frames build up. At most, we’ll have roughly n + n/2 + n/4 + … space total pending on the call stack (adding up the one pending stack frame from each of the levels when the base case is reached). Simplifying, n (1 + 1/2 + 1/4 + …) is roughly O(2n) = O(n) space.

Master Theorem

14. Find the tight upper-bound of an algorithm with the following recurrence relation: $T(n) = T(n/2) + O(1)$. You may show your work for partial credit.

    $a = 1$, $b = 2$, $k = 0$, so $\log_a b = \log_2 1 = 0$
    $k = 0$
    We see k and the log are are equal, meaning this is the case of Master Theorem where the runtime is $n^0\log(n)$ which simplifies to $O(\log(n))$

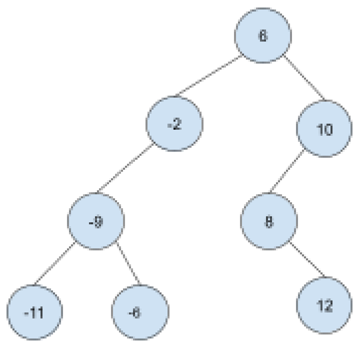
15. Write a recurrence relation for MergeSort. You may show your work for partial credit.

    We split the list in half ($b=2$) and call on each half ($a = 2$). In each function call outside of the recursion, we're doing $O(n)$ work. So we have
    $T(n) = 2\,T(n/2) + O(n)$

Trees

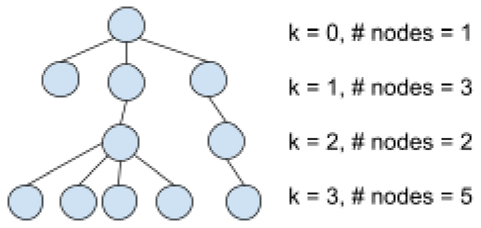
16. Is the tree on the right a Binary Search Tree? Explain.
    No. The left subtree of 10 contains 12, which is greater than it. If the 12 were a 9, for example, then it would be a Binary Search Tree.

17. What would a post-order traversal of this tree print out?
    -11, -6, -9, -2, 12, 8, 10, 6



---

18. Complete the recursive case of countAtLevel in the box, which counts the number of nodes at each level in a Tree.



```
algorithm countAtLevel
  input: TreeNode root and a level k
  output: the number of nodes at level k in root

if level == 0  // base case
  return 1

total = 0
for each element child in root->getChildren()
   total += countAtLevel(child, level-1)
return total
```

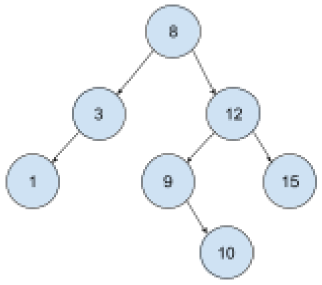
19. What is the **best-case runtime** of **removing** a node for a “**regular**” (i.e. not AVL) BST. Give an example of when the best-case happens.
    O(1). Example: if we are removing the minimum value, and the minimum value has no children (e.g. removing the root of a tree that looks like a linked list by only having “right” children).

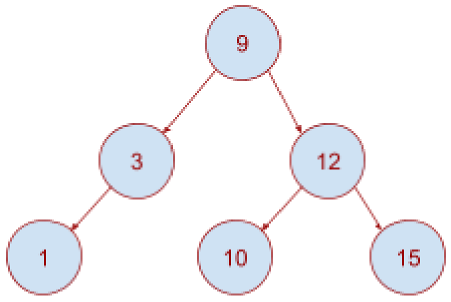
20. What is the **worst-case runtime** of **searching** for a node in a **balanced** (e.g. AVL) BST. Give an example of when the worst-case happens.
O(log(n)) because searching in a tree is O(height of the tree) and the height of a balanced tree is O(log(n)). This happens when we are searching for one of the leaves.

For question 21, use the following pseudocode

```
BSTremove(t, v) // from visualgo.net
  search for v
  if v is a leaf
    delete leaf v
  else if v has 1 child
    bypass v
  else replace v with successor
```

21. Draw what the BST t below will look like after BSTremove(t, 8)





Graphs

Use the following adjacency list, to answer questions 22 - 23

```
0: {1}
1: {3, 4}
2: {0}
3: {1, 2}
4: {}
```

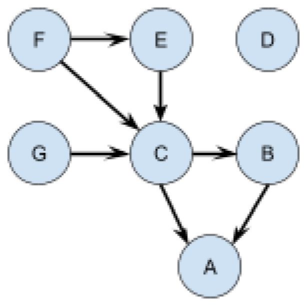
22. Represent this Graph as an adjacency matrix. Recall that `m[a][b] = 1` means that there's a directed edge from Node(a) to Node(b)

    {
     {0, 1, 0, 0, 0},
     {0, 0, 0, 1, 1},
     {1, 0, 0, 0, 0},
     {0, 1, 1, 0, 0},
     {0, 0, 0, 0, 0}
    }

23. Does this graph have cycles? If yes, identify them.

    Yes, 1-3 and 0-1-3-2

Use the following DAG to answer questions 24 - 26.



24. How many source nodes are there?

    3 (F, D, G)

25. How many sink nodes are there?
    2 (A, D)

26. Provide one valid topological sort for this DAG
    G, D, F, E, C, B, A

For questions 27 - 29, use the below.

Wildlife scientists observe elephants in Serengeti National Park. Although the elephant herds may rarely be seen altogether, the scientists want to understand the **average herd size**, so they record all elephant "interactions" they observe over 3 months. Assume:

- The scientists can uniquely identify each elephant.
- Elephants will only ever "interact" with other elephants in their same herd.
- Elephants can only belong to one herd.

27. In order to solve this problem, we can represent this as a graph. What are the nodes and edges?
    Nodes are each elephant (which can be uniquely identified). Edges are interactions. There is an edge between elephant A and B if a scientist has observed an "interaction" between them.

28. Which graph properties apply to this graph? Select **ALL** that apply.
    - [✓] **Undirected**
    - ☐ Acyclic
    - ☐ Weighted

29. How would you solve this problem leveraging graph algorithms we've covered? Explain which algorithm you would use in words AND how you would use it to produce the **average herd size** amongst all observed elephants**.**

    - The key insight is to realize that all connected components are a herd.
    - Do a BFS (or DFS) starting from any elephant and find all elephants reachable from that elephant. This represents one herd, so we increment our herd count.
    - Repeat the process until all elephants have been reached.
    - Take the total number of elephants divided by the number of connected components.