1 Short answer (2 questions) - 22 points

Answer the following questions. Briefly justify your answers, but a complete proof is not required.

(a) **[XX points]** Give a *tight* asymptotic bound for the following recurrences :

(i)

$$A(n) = 2A\left(\frac{n}{2}\right) + n^3 \qquad A(0) = A(1) = 1$$

Solution: We can model this recurrence as a recursion tree as follows:

$$n^3$$

$$\frac{n^3}{8} \qquad \frac{n^3}{8}$$

$$\frac{n^3}{64} \quad \frac{n^3}{64} \quad \frac{n^3}{64} \quad \frac{n^3}{64}$$

At the root, the work sums up to n^3. At level 2, the work sums up to $\frac{1}{4}n^3$. At level 3, the work sums up to $\frac{1}{16}n^3$. The amount of work is decreasing at each level. Hence, the overall recurrence is dominated by the root node. Therefore, the asymptotic bound is $O(n^3)$. ■

(ii)

$$B(n) = B(n-2) + n^2 \qquad B(0) = B(1) = 1$$

Solution: Let us unroll the above recurrence:

$$\begin{aligned}
B(n) = {} & B(n-2) + n^2 \\
= {} & B(n-4) + (n-2)^2 + n^2 \\
= {} & B(n-6) + (n-4)^2 + (n-2)^2 + n^2 \\
& \dots \\
= {} & B(n-k) + \sum_{i=0;i+=2}^{k-2} (n-i)^2 \\
= {} & B(0) + \sum_{i=0;i+=2}^{k-2} (n-i)^2 = 1 + \sum_{m=1}^{n} m^2 \\
= {} & 1 + n(n+1)(2n+1)/6
\end{aligned}$$

As we just need the asymptotic upper bound, we simplified the expression in line 5 and considered sum of squares of natural numbers. Therefore, the asymptotic bound is $O(n^3)$. ■

Solution: TLDR explanation:

- Number of levels: n
- Work on each level: n^2

Asymptotic bound: $O\left(n^3\right)$ ■

(b) We developed a new type of algorithm to sort a set of (non-numerical) elements. It sorts a set of size n elements by dividing them into nine sub-problems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time. What is the asymptotic running time of this algorithm?

Solution: The recursion relation for the above can be written as:

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2$$

We can model this recurrence as a recursion tree as follows:

$$n^2$$

$$\frac{n^2}{9} \quad \frac{n^2}{9} \quad \cdots \quad \frac{n^2}{9}$$

$$\frac{n^2}{81} \quad \cdots \quad \frac{n^2}{81} \quad \frac{n^2}{81} \quad \cdots \quad \frac{n^2}{81} \quad \cdots \quad \frac{n^2}{81} \quad \cdots \quad \frac{n^2}{81}$$

At each level, the nodes sum up to n^2. The maximum depth of the tree is $\log n$. Therefore, we have $\log n$ levels with n^2 work per level, so the asymptotic running time is $O(n^2 \log n)$. ■

2 Short answer II (4 questions) - 28 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required. For the following graph problems, use the notation $G = (V, E)$, $n = |V|$ and $m = |E|$

(a) How many strongly connected components does a directed acyclic graph (DAG) have?

Solution: Each node in a DAG is a strongly connected components on its own as there are no cycles in the graph. Hence, for a DAG G=(V,E), it has exactly |V| strongly connected components. ■

(b) In the Floyd-Warshall (found in the cheat sheet), we defined a recurrence $d(i, j, k)$. **Give an English description (no more than 2 sentences) of what $d(i, j, k)$ represents.**

Note: what's in the cheat sheet does not constitute a english description for the recurrence.

Solution: Assuming all the vertices are numbered 1 through n, $d(i, j, k)$ represents the shortest path from i to j using only vertices 1 through k. ■

Solution: For Floyd Warshalls algorithm, the recurrence $d(i, j, k)$ represents the shortest path from node i to node j with k as the intermediate node in between the path from i to j. That is, it represents the shortest path for i → j where you traverse from i → k and then from k → j. ■

(c) Given n vertices, what is the minimum number of edges one would need to create a graph with exactly one topological sort.

Solution: A unique toplogical sort would have all edges directed from a lower level to a higher level in the graph. After designating a source and sink vertex we chose consequent vertices to link each vertices one after the other starting from the source and ending with sink where all the other vertices have exactly an incoming edge and an outgoing edge each. This can be done in eactly n-1 edges. ■

(d) Your friend says he discovered a better way of calculating the shortest path in graphs with negative weight edges. All we need to do is find the minimum edge weight $w^* = \min\{w(u,v) | (u,v) \in E\}$ and add it to all the other edges in the graph $\hat{w} = w(u,v) - w^*$.

Now that the edges are all positive weight, you can use Djikstra and find the shortest path. Does this method of re-weighting work? Either prove the correctness of the method or provide a counter example (and briefly explain the counter example).

Circle one: Yes(re-weighting works) No (re-weighting does not work)

Solution: This would NOT work as the paths that have more edges would essentially have more weight per edge added to it. Let's say there were two paths from a to c : $1.a->b->c = -3+5 = 2$ and $2.a->d->e->c = -3+1+3 = 1$ and path 1 is the shortest. Now adding the smallest edge weight to all edges would result in the path 1 length becoming $2+2(3) = 8$ and path 2 length as $1+3(3) = 10$ thereby changing the shortest path to path 1 which is wrong. (Rough solution; will be adding a diagram later if needed) ■

3 Finding a plurality - 15 points

Given an arbitrary array $A[1..n]$, describe an algorithm to determine in $O(n)$ time whether A contains more than $n/4$ copies of any value. **Do not use hashing, or radix sort, or any other method that depends on the precise input values.**

Solution: This problem was a direct duplicate of a HW and lab problem. Only a brief explanation was required:

- For a array to have an element that appears n/4 times, then that element must appear in the rank $n/4$, $n/2$ or $3n/4$ spot.
- we can use Linear time selection (QuickSelect + MoM) to find the elements of rank 0, $n/4$, $n/2$, $3n/4$ and n.
- Then we loop over the array once for each of those values and see how many times each of those elements appears. If one of those elements appears $> n/4$ times, we return yes, otherwise no.

The average for this problem was 62% and that's giving substantial credit for sub-optimal (sort-based) solutions. Only 31% of students used anything resembling linear time selection. I can't help but think that maybe certain course policies are doing more harm than good? Looking at that 31% number (and the fact that groups of three are allowed for homeworks), suggests that the vast majority of groups are simply divvying assignments which is sad.... ■

Solution: Longer explanation: The algorithm is formally described below. We use the fact that the selection problem can be solved in linear time. That is, given an unsorted array A of n values and an index j between 1 and n, we can find the j-th ranked element in A in $O(n)$ time. We denote this black box algorithm as SELECT(A[1..N], j) which returns the value of the j-th ranked element in A. To determine whether an element appears more than $n/4$ times, we select values with rank $n/4$, $2n/4$, and $3n/4$. If an element x appears more than $n/4$ times, it follows that at least one of these selected values is equal to x. Thus, we can scan and count the number of occurrences of each of these selected values.

```
Contains4Duplicates($A[1..N]$)
    $x_1 \leftarrow$ SELECT(A, $\lceil N/4 \rceil$)
    $x_2 \leftarrow$ SELECT(A, $\lceil 2N/4 \rceil$)
    $x_3 \leftarrow$ SELECT(A, $\lceil 3N/4 \rceil$)
    for ($i \leftarrow 1 : 3$)
        count $\leftarrow 0$ for ($j \leftarrow 1 : N$)
            if ($A[j] = x_i$) then count$++1$
        if (count $> N/4$) then return True
    return False
```

Since SELECT runs in $O(n)$ time, finding x_1, x_2, and x_3 also takes $O(n)$ time. Looping over the array of length n a total of 3 times takes $O(n)$ time. Thus, this algorithm runs in the required $O(n)$ time.

To prove correctness of the algorithm, we must show that if an element appears more than $n/4$ times, it must be at least one of the selected values with rank $\lceil n/4 \rceil$, $\lceil 2n/4 \rceil$, or

$\lceil 3n/4 \rceil$. Assume an element x appears $i > n/4$ times. Then, there must be consecutive ranks $j, \dots, j+i-1$ with value x. Without loss of generality, consider the number of values of rank between $\lceil n/4 \rceil$ and $\lceil 2n/4 \rceil$ (excluding the outside values). Since $\lceil n/4 \rceil \geq n/4$ and $\lceil 2n/4 \rceil \leq 2n/4+1$, the maximum number of values is given by $(2n/4+1)-(n/4)-1 = n/4$. Thus, there are at most only $n/4$ spots for more than $n/4$ values. By pigeonhole principle, one of the selected values must be equal to x. ■

4 Dynamic programming - 15 points

A common subsequence of three strings X, Y, Z is a string that is a subsequence of each of X, Y, and Z. Describe a DP algorithm that returns the length of the longest common subsequence of $X[1..n]$, $Y[1..n]$, and $Z[1..n]$ by providing the following.

Solution: This is simply the edit-distance/longest-common-subsequence problem I spent Lecture 14 describing.

Recurrence and short English description(in terms of the parameters):

$$LCS(i,j,k) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \text{ or } k=0 \\ 1 + LCS(i-1,j-1,j-1) & \text{if } X[i]=Y[j]=Z[k] \\ \max\begin{cases} LCS(i-1,j,k) \\ LCS(i,j-1,k) \\ LCS(i,j,k-1) \end{cases} & \text{otherwise} \end{cases}$$

Where $LCS(i,j,k)$ denotes the length of the longest common subsequence of $X[1..i]$, $Y[1..j]$, $Z[1..k]$.

Memoization data structure and evaluation order: The data structure would be a 3-dimensional n by n by n array. We evaluate in increasing order(from index 1 to n) for all three dimensions.

Return value: $LCS(n,n,n)$, which is the longest common subsequence of $X[1..n]$, $Y[1..n]$, $Z[1..n]$.

Time Complexity: $O(n^3)$, because of the nested loops over i, j, k.

■

5 Graph algorithms (2 questions) - 20 points

For the graph problems, assume that the graph is represented by adjacency lists with outgoing edges only – that is, for each vertex u in the graph, you know Out(u), which stores outgoing edges from vertex u.

Assume you had a directed acyclic graph with one edge marked as **important**. A *important path* is a path that contains this one important edge.

Assume all the edges have the same weight $\ell(e) = 1$.

(a) Describe an algorithm that finds the shortest *important* path (**not just path length**) from s to t.

Solution: To find the shortest important path we find the shortest path from s to the beginning of the important edge, node u. Then the shortest path from the end of the important edge, node v, to t. When these 2 paths are added together with the important edge the result is the shortest important path from s to t.

To find the shortest path from s to u we use a modified BFS. This modified BFS keeps track of the parent node and as soon as u is reached it ends the while loop. Then starting with u it adds parent nodes until it reaches s. BFS finds the shortest path because all edges have the same weight.

Do the same for the shortest path from v to t.

Then we concatenate these 2 paths together to get the shortest important path.

Let e be the important edge, V the node list, Out be the adjacency list, s be the start node, and t be the destination node.

```
ImportantPath($G(V, Out), e, s, t$):
    $u$ ← outNode($e$)
    $v$ ← inNode($e$)
    pathsu ← PathBFS($G(V, Out), s, u$)
    pathvt ← PathBFS($G(V, Out), v, t$)
    pathst ← empty
    if pathsu and pathvt are non-empty
        pathst ← concatenate(pathsu,pathvt)
    return pathst
```

```
PathBFS($G(V, Out), x, y$):
  for $v$ in $V$
      visited(v) ← false
  N ← x
  visited(x) ← true
  while N is non-empty and visited(y) is false
      remove w from N
      for z in Out(w)
          if visited(z) is false
              visited(z) ← true
              add z to N
              parent(z) ← w
  path ← empty
  if visited(y) is true
      t ← y
      path ← y
      while t is not x
          t ← parent(t)
          path ← concatenate(t,path)
  return path
```

Note N is a queue.

ModifiedBFS is $O(V+E)$ because it is essentially normal BFS plus a while loop that is at most $O(V)$. So the total running time is $O(V+E)$

■

(continued from previous page)

(b) Describe an algorithm that finds all the vertices that can reach t using an *important* path.

Solution: There are 2 components to this algorithm. First verifying that t can be reached from the endpoint of the important edge, node v. Then determining which nodes can reach the beginning of the important edge, node u.

To verify that t can be reached from the endpoint we run BFS on the graph from node v then check that t is reached.

To determine which nodes can reach the beginning of the important edge we flip the direction of all the edges then run BFS on this new graph from node u. All nodes that can reach node u in the original graph will be reachable from node u in the reverse graph.

Let e be the important edge, V the node list, Out be the adjacency list, and t be the destination node.

```
ImportantVertices($G(V, Out), e, t$):
  $u \leftarrow outNode(e)$
  $v \leftarrow inNode(e)$
  Tv $\leftarrow$ BFS($G(V, Out), v$)
  if $t$ in Tv
      for $x$ in $V$
          for $y$ in $Out(x)$
              $x$ in $Out2(y)$
      Tu $\leftarrow$ BFS($G(V, Out2), u$)
      list $\leftarrow$ vertices(Tu)
  else
      list $\leftarrow$ empty
  return list
```

BFS is $O(V+E)$, reversing the edges in the graph is $O(E)$. So the total running time is $O(V+E)$.

■