Name: ____________________________
Student ID: ______________________

This exam is closed book, closed notes, no electronic devices. Show all work. Partial credit given for partial solutions. Presentation counts! Be legible and coherent for full credit.

1. (20 points)
   Here is a weighted graph in the form of the vertex set \( V \) and the edge set \( E \).
   \( V = \{a, b, c, d, e\} \)
   \( E = \{\{a, b\}\text{wgt}1], \{b, c\}\text{wgt}3], \{b, d\}\text{wgt}4], \{b, e\}\text{wgt}2], \{c, d\}\text{wgt}4], \{d, e\}\text{wgt}3]\)

   (a) (3 points)
   Draw a pictorial representation of the graph with dots for vertices and lines for edges labeled with the weights.
   **Solution:**

   ```
   1 3 4 3
   a----b----c----d----e
   | |______| |
   | 4 |
   |________|
   2
   ```

   (b) (5 points)
   Complete the adjacency matrix \( M \) of the graph, where \( M[i, j] = \text{wgt}(i, j) \) if there is a weighted edge between \( i \) and \( j \) and 0 otherwise. Since the graph is not directed, you only need to write the upper half triangle of the matrix.
   **Solution:**

   ```
   \[
   \begin{pmatrix}
   1 & 0 & 0 & 0 \\
   3 & 4 & 2 \\
   4 & 0 \\
   3
   \end{pmatrix}
   ```
(c) (5 points)
Complete the array of adjacency lists $A$ of the graph where each array entry is a list of the vertex’s neighbors with edge weights.

**Solution:**
a $\rightarrow$ b, 1
b $\rightarrow$ a, 1 $\rightarrow$ c, 3 $\rightarrow$ d, 4 $\rightarrow$ e, 2
c $\rightarrow$ b, 3 $\rightarrow$ d, 4
d $\rightarrow$ b, 4 $\rightarrow$ c, 4 $\rightarrow$ e, 3
e $\rightarrow$ b, 2 $\rightarrow$ d, 3

(d) (7 points)
Prim’s algorithm for a minimum spanning tree maintains a partial spanning tree and at each step adds a new edge of minimum possible weight without introducing a cycle, until all vertices are spanned. Assume ties are broken in alphabetical order with the edge containing an alphabetically lower vertex coming first. List the edges in the minimum spanning tree for this graph in the order they are added to the tree by Prim’s algorithm.

**Solution:**
\{a, b\}, \{b, e\}, \{b, c\}, \{d, e\}
2. (20 points)
Here is a directed graph in the form of the vertex set \( V \) and and the directed edge set \( E \).
\[ V = \{a, b, c, d, e, f\} \]
\[ E = \{(a, b), (a, c), (c, e), (d, c), (d, f)\} \]

(a) (3 points)
Draw a pictorial representation of the graph with dots for vertices and arrows for directed edges.

**Solution:**
```
a--->b
|  
|---c--->e
|  
|  
|  
d--->f
```

(b) (5 points)
Complete the adjacency matrix \( M \) of the graph, where \( M[i, j] = E \) if there is a directed edge \((i, j)\) and is blank otherwise.

**Solution:**
```
\[
\begin{bmatrix}
  - & E & E & - & - & - \\
  - & - & - & E & - & - \\
  - & - & E & E & - & - \\
  - & - & - & - & E & - \\
  - & - & - & - & - & - \\
  - & - & - & - & - & - \\
\end{bmatrix}
\]
```

(c) (5 points)
Complete the array of adjacency lists \( A \) of the graph where \( j \) is on \( i \)'s adjacency list if there is a directed edge from \( i \) to \( j \).

**Solution:**
a \(\rightarrow\) b \(\rightarrow\) c
b
(c \(\rightarrow\) e
d \(\rightarrow\) c \(\rightarrow\) f
e
f
(d) (7 points)
The depth-first search topological sort algorithm does depth-first search and appends vertices to the front of a list when all their neighbors have been searched. Then it outputs the list as a linear ordering of the vertices. Assume ties are broken in alphabetical order by visiting alphabetically lower vertices first. Give the list of vertices output by a topological sort of this graph.

**Solution:** d, f, a, c, e, b
3. (10 points)

Let \( f(n) = n^3 \cdot 2^n \) and \( g(n) = n^2 \cdot 2^n \). Compute \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \). Then state whether \( f \) is \( o(g) \), \( \Theta(g) \) or \( \omega(g) \).

**Solution:**

\[
\lim_{n \to \infty} \frac{n^3 \cdot 2^n}{n^2 \cdot 2^n} = \lim_{n \to \infty} \frac{n}{2^n} = \lim_{n \to \infty} \frac{\ln(2)}{2^n \cdot (3 \cdot n^2 - 2 \cdot n \ln(2))} = 0.
\]

We conclude that \( f(n) \) is \( o(g(n)) \).
4. (10 points) The following statement is false: “For all functions \( f, g, r \) and \( s \), if \( f(n) = O(r(n)) \) and \( g(n) = O(s(n)) \), then \( \frac{f(n)}{g(n)} = O\left(\frac{r(n)}{s(n)}\right) \).” Give a counter-example to show it is false.

In other words, give examples of functions \( f, g, r \) and \( s \) such that \( f(n) = O(r(n)) \), \( g(n) = O(s(n)) \), but \( \frac{f(n)}{g(n)} \) is not \( O\left(\frac{r(n)}{s(n)}\right) \).

**Hint:** You can play with constants because big-\( O \) is indifferent to constant factors. Think of functions that cancel each other under division.

**Solution:**
Many counter-examples are possible. Here is one of them. \( f(n) = n^2, g(n) = n, r(n) = n^3, s(n) = n^3 \). We have that \( n^2 = O(n^3) \) and \( n = O(n^3) \) because a higher degree polynomial is always an asymptotic upper bound for a lower degree one. But \( \frac{n^2}{n} = n \) is not \( O\left(\frac{n^3}{n^3}\right) = 1 \).

**Comment:** If we use big-\( \Theta \) instead of big-\( O \), then the statement becomes true. In other words, it is true that if \( f = \Theta(r) \) and \( g = \Theta(s) \), then \( \frac{f}{g} \) is \( \Theta\left(\frac{r}{s}\right) \). Here is a proof of the statement. If \( f = \Theta(r) \), then there is an \( N \) such that for all \( n > N \), \( c_1 \cdot f(n) \leq r(n) \leq c_2 \cdot f(n) \) for some pair of constants \( c_1 \) and \( c_2 \). And if \( g = \Theta(s) \), then there is an \( \bar{N} \) such that for all \( n > \bar{N} \), \( k_1 \cdot g(n) \leq s(n) \leq k_2 \cdot g(n) \) for some pair of constants \( k_1 \) and \( k_2 \). Let \( M = \max(N, \bar{N}) \). Then for all \( n > M \), we will have \( \frac{c_1 \cdot f(n)}{k_1 \cdot g(n)} \leq \frac{r(n)}{s(n)} \leq \frac{c_2 \cdot f(n)}{k_2 \cdot g(n)} \). Let \( \tilde{c}_1 = \frac{c_1}{k_1} \) and \( \tilde{c}_2 = \frac{c_2}{k_2} \), then we have that for all \( n > M \), \( \tilde{c}_1 \cdot \frac{f(n)}{g(n)} \leq \frac{r(n)}{s(n)} \leq \tilde{c}_2 \cdot \frac{f(n)}{g(n)} \) which means that \( \frac{f}{g} \) is \( \Theta\left(\frac{r}{s}\right) \).
5. (15 points) **Finding a matching weight:** You have a coin of unknown weight. You know its weight is one of \( n \) standard coin weights. You have a scale and \( n \) reference coins ordered from 1 (lightest) to \( n \) (heaviest).

(a) (5 points) Give an efficient algorithm for finding the reference coin that matches the unknown coin. The algorithm must be asymptotically better than trying every coin.

**Solution:**
We first weigh the unknown coin against the coin number \( \frac{n}{2} \). If the coins are equal, we have found the answer. If not, we have a new problem of matching the coin against a reference set of size \( \frac{n}{2} \) (the upper half of the original reference set if the unknown coin was heavier, or the lower half of the set if the unknown coin was lighter). We recursively solve this problem. In the worst case we must succeed when there is only one coin left in the reference set.

(b) (5 points) Give a recurrence for the number of weighings used by your algorithm in the worst case as a function of the number of coins in the reference set.

**Solution:** Let \( W(n) \) be the number of weighings required with a reference set of \( n \) coins. Then with 1 weighing we reduce to a similar problem of size \( n/2 \). A recurrence for this situation is therefore \( W(n) = 1 + W(n/2) \).

(c) (5 points) Give a big-\( \Theta \) bound for the solution to your recurrence. Give a justification for your solution using any method.

**Solution:** \( W(n) \) is \( \Theta(\log(n)) \). We verify this by the method of substitution:
\[
\log(n) = 1 + \log(n/2) \\
= 1 + \log(n) - \log(2) \\
= 1 + \log(n) - 1 \\
= \log(n)
\]
so the substitution works.
6. (15 points) **Project integration:** A computer company is using divide-and-conquer to manage its programming projects.

- Every team splits into 2 equal subteams, the subteams split into sub-subteams, and so on until split into teams of 1.
- 1 person takes 1 person-hour to do a task.
- The 2 subteams join back together to integrate their code. It takes $2^k$ person-hours for a group of $k$ people to integrate their code.

(a) (5 points) Write a recurrence for the number of person-hours it takes for $n$ people to do a project.

**Solution:** A team of $n$ people will split into 2 teams of $n/2$ people and perform a similar split-and-integrate procedure, then take $2n$ person-hours to integrate their work. Thus, a recurrence for the number of person-hours is $T(n) = 2 \cdot T(n/2) + 2n$.

(b) (5 points) Fill in the following recursion tree diagram for this recurrence, where the number in the brackets stands for the nonrecursive cost at each level. The nonrecursive cost at the root is already filled in for you as $2n$.

**Solution:**

```
   n[2n]
  /\  \
 /  \  \
/  n/2[n]  n/2[n]
/\  /\  /\  \
/   n/4 n/4 n/4
/\  /\  /\  \
/   n/2 [n/2] [n/2] [n/2]
```

(c) (2 points) Give a formula for the nonrecursive cost of the $k$th level of the recurrence.

**Solution:** $2n$ in the general case (in the final level it will be $n$ since each of the $n$ people spends 1 person-hour).

(d) (1 point) Solve for $k$ in $n/2^k = 1$ to find the number of levels in the tree diagram.

**Solution:**

$$\frac{n}{2^k} = 1, n = 2^k, \log(n) = k$$
(e) (2 points) Take the summation over all the levels of the nonrecursive cost at each level to solve the recurrence.

\[
\sum_{k=0}^{\log_2(n)-1} 2^n + n = 2n \cdot \log(n) + n \text{ or } \Theta(n \cdot \log(n)).
\]
7. (10 points) The graph $K_n$, called the complete graph on $n$ vertices, is a graph in which every pair of vertices is connected by an edge. Here is a recursive procedure for constructing $K_n$.

If $n = 1$, construct a single vertex $v_1$.
If $n > 1$, construct $K_{n-1}$. Add a new vertex $v_n$. Add an edge from $v_n$ to every vertex in $K_{n-1}$.

Based on this recursive procedure, write a recurrence for the number of edges in $K_n$.

**Solution:**
Since $K_{n-1}$ is a graph with $n - 1$ vertices, there are $n - 1$ new edges to add from $v_n$ to every vertex in $K_{n-1}$. Therefore the recurrence is $E(n) = E(n - 1) + n - 1$. 


8. (10 points) Suppose we have a 4-way branching tree, such that every vertex except the leaves has 4 children. Suppose also that each vertex contains three keys \( k_1, k_2 \) and \( k_3 \) and the vertices satisfy the 4-way search property: all keys in the first subtree are less than or equal to \( k_1 \), all keys in the second subtree are greater than \( k_1 \) and less than or equal to \( k_2 \), all keys in the third subtree are greater than \( k_2 \) and less than or equal to \( k_3 \), and all keys in the fourth subtree are greater than \( k_3 \) at every level in the tree.

(a) (5 points) Describe an algorithm for retrieving a key from such a 4-way branching search tree.

**Solution:**
The basic idea is to compare the key with \( k_2 \), then if necessary with either \( k_1 \) or \( k_3 \), and use the results to decide in which subtree to search recursively. The recursion stops either when the key is found, when it returns with success, or when a leaf is reached, when it returns with failure. Here is pseudocode for this idea:

```plaintext
Algorithm search(tree T, key k)
    if T is an empty tree return with failure
    otherwise
        \([k_1, k_2, k_3]\) = list of keys in root of T
        compare k with k2
        if k < k2
            compare k with k1
            if k < k1
                search(1st subtree of T, k)
            if k = k1
                return with success
            if k > k1
                search(2nd subtree of T, k)
        if k = k2
            return with success
        if k > k2
            compare k with k3
            if k < k3
                search(3rd subtree of T, k)
            if k = k3
                return with success
            if k > k3
                search(4th subtree of T, k)
    end algorithm
```
(b) (5 points) Assuming the tree is perfectly balanced (every non-leaf always has 4 children), give a big-$\Theta$ bound for the number of comparisons required for your algorithm.

**Solution:**
In the worst case, the algorithm does two comparisons at each level and reaches a leaf, in which case the number of comparisons is twice the height of the tree. Intuitively, the height of the tree is logarithmic in the number of vertices, since it is an exponentially branching structure. A constant number of elements are stored in each vertex. So the number of comparisons for a set of $n$ elements is $\Theta(\log(n))$.

More rigorously, there are 3 elements in every vertex, and $4^k$ vertices on the $k$th level, so if the tree has $k$ levels it will have \( \sum_{i=0}^{k} 3 \cdot 4^k = \frac{3 \cdot (4^{k+1} - 1)}{4 - 1} = 4^{k+1} - 1 \) elements which is bounded by $4^{k+1}$. Taking logs, a bound for the height of the tree for $n$ elements is $k+1 = \log_4(n)$ or $k = \log_4(n) - 1$. We have to do comparisons at the root too, so the number of levels on which we do comparisons is bounded by $\log_4(n)$, and the number of comparisons is bounded by $2 \cdot (\log_4(n))$. So the big-$\Theta$ bound of $\Theta(\log(n))$ is tight.