Problem 1. (10 points)

Let $G = (V, E)$ be a weighted undirected graph. We define the bottleneck of a path $p$ in $G$ as the minimum weight of any edge on $p$. We define the maximum bottleneck of any $s, t$-path as the maximum, over all paths $p$ from $s$ to $t$, of the bottleneck of $p$.

Prove or disprove: given any connected, undirected, edge-weighted graph, algorithm Bottleneck below produces a tree $T$ such that the bottleneck of the path $p$ from $s$ to $t$ in $T$ is the maximum bottleneck of any $s, t$-path in the original graph.

Algorithm Bottleneck $(G(V, E) : graph)$

sort the edges $e_1, e_2, \ldots, e_m$ in order of decreasing cost

$T \leftarrow \emptyset$

for $i \leftarrow 1, 2, \ldots, m$ do

Add $e_i$ to $T$ if doing so does not create a cycle in $T$

return $T$

Answer: The algorithm is correct. The algorithm produces a spanning tree $T$ of the graph for the same reason that Kruskal’s algorithm does. At each point in time, each edge considered but not added by the algorithm creates a cycle with the edges added so far. This implies that: (*) if you can get from a vertex $u$ to a vertex $w$ using just the edges considered so far, then you get get from $u$ to $w$ using just the edges added so far. Now consider the path $p$ connecting $s$ to $t$ in $T$. Consider the last edge $e_i$ on $p$ added to $T$ by the algorithm. The property (*) above implies that there is no path from $s$ to $t$ using just the edges $\{e_1, e_2, \ldots, e_{i-1}\}$. In other words, every path from $s$ to $t$ uses some edge in $\{e_i, e_{i+1}, \ldots, e_m\}$. Keeping in mind that the edges are considered in order of decreasing weight, this implies that every path from $s$ to $t$ uses an edge of weight at most $w(e_i)$. This implies that every path from $s$ to $t$ has bottleneck at most $w(e_i)$.

Problem 2. (10 points)

Consider a variation of the Union-Find data structure, with the union-by-rank heuristic but without path compression. That is, we implement Make-Set and Union as usual, but we do not reset the parent pointers in Find-Set. Show that there is some sequence of $n$ calls to Make-Set, some number (at most $n$) of calls to Union, and $m$ calls to Find-Set that require $\Omega(m \log n)$ time from this suboptimal implementation.

Answer:

Start with $n$ Make-sets to create singleton set $\{x_1\}, \{x_2\}, \ldots, \{x_n\}$. Next create a tree of height $\Omega(\log n)$ by a total of $n - 1$ Union. First $n/2$ Unions as follows: Union($x_1$, $x_2$), Union($x_3$, $x_4$), $\ldots$, Union($x_{n-1}$, $x_n$) of size 2; then $n/4$ Unions as follows: Union($x_2$, $x_4$), Union($x_6$, $x_8$), $\ldots$, Union($x_{n-2}$, $x_n$) of size 4; then ... one Union($x_{n/2}$, $x_n$).

This gives a binomial tree which has $\left(\begin{array}{c} k \\ i \end{array}\right)$ of its $2^k$ nodes at depth $i$, for $i = 0, \ldots, k$. Hence at least half of the $n$ nodes are at depth greater than or equal to $(\log n)/2$, so each
**Problem 3.** (10 points)

In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of \( \{d_1, \ldots, d_k\} \) units. They seek an algorithm that will enable them to make change of \( n \) units using the minimum number of coins.

1. The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of \( n \) units using US denominations. Prove its correctness and analyze its time complexity.

2. Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are \( \{1, 6, 10\} \).

3. Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of \( n \) units using denominations \( \{d_1, \ldots, d_k\} \). Analyze its running time.

**Answer:** Here is the greedy algorithm.

**Inputs:** number of units to make change for \( n \)

**Outputs:** number of half dollars, quarter, dimes, nickels, and pennies to use \((c_{50}, c_{25}, c_{10}, c_{5}, c_{1})\).

**Algorithm** \texttt{MakeChange}(\( n \))

\[
\begin{align*}
c_{50} &= n \div 50 \\
n &= n \mod 50 \\
c_{25} &= n \div 25 \\
n &= n \mod 25 \\
c_{10} &= n \div 10 \\
n &= n \mod 10 \\
c_{5} &= n \div 5 \\
n &= n \mod 5 \\
c_{1} &= n
\end{align*}
\]

**return** \((c_{50}, c_{25}, c_{10}, c_{5}, c_{1})\)

Because the algorithm always performs 10 calculations, its worst-case running time is \( O(1) \).

**Proof of Optimality:** Assume that the best non-greedy solution for a given instance of the problem is \((b_{50}, b_{25}, b_{10}, b_{5}, b_{1})\), where \( n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_{5} + b_{1} \). We show that the greedy solution is as good as or better than the best solution. The greedy solution is \((c_{50}, c_{25}, c_{10}, c_{5}, c_{1})\). We want to show that \( c_{50} + c_{25} + c_{10} + c_{5} + c_{1} \leq b_{50} + b_{25} + b_{10} + b_{5} + b_{1} \).
Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If \( b_{50} < c_{50} \) then \( 25b_{25} + 10b_{10} + 5b_{5} + b_{1} \geq 50 \). To satisfy the given inequality these are all the possibilities.

1. if \( b_{25} \geq 2 \), replace with 1 half-dollar
2. if \( b_{25} = 1 \) we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins
3. if \( b_{25} = 0 \) we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If \( b_{50} = c_{50} \) and \( b_{25} < c_{25} \) then \( 10b_{10} + 5b_{5} + b_{1} \geq 25 \). These are the possibilities.

1. if \( b_{10} \geq 3 \), replace with 1 quarter and 1 nickel
2. if \( b_{10} = 2 \) we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter
3. if \( b_{10} = 1 \) we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter
4. if \( b_{10} = 0 \) we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if \( b_{50} = c_{50}, b_{25} = c_{25}, b_{10} = c_{10}, \) and \( b_{5} < c_{5} \).

2) We can show that the greedy algorithm doesn’t work for all possible denominations by giving a counter-example. If \( n = 12 \) and \( (d_{1}, d_{2}, d_{3}) = (1, 6, 10) \), then the greedy algorithm would return \( (c_{10}, c_{6}, c_{1}) = (1, 0, 2) \). However, the optimal solution is \( (c_{10}, c_{6}, c_{1}) = (0, 2, 0) \).

3) Given a list of \( k \) coin values, \( (d_{1}, d_{2}, \ldots, d_{k}) \), and a number \( n \), we want to find the integers \( (c_{d_{1}}, c_{d_{2}}, \ldots, c_{d_{k}}) \) such that \( n = \sum_{i=1}^{k} d_{i}c_{d_{i}} \) and that \( \sum_{i=1}^{k} c_{d_{i}} \) is minimal.

Our subproblems consist of the optimal change set for 1 through \( n \). To keep track of the optimal solution for each subproblem we will use an array called \( \text{sumc} \) which is indexed by subproblem. (i.e. \( \text{sumc}[i] \) contains the least number of coins needed to make change for \( i \)).

\( \text{coin}[i] \) designates which coin denomination was last used when making change for \( i \) units.

\[ \text{sumc}[d_{1}] = 1, \text{sumc}[d_{2}] = 1, \ldots, \text{sumc}[d_{k}] = 1 \]

\[ \text{sumc}[i] = \min_{1 \leq j \leq k} \text{sumc}[i - d_{j}] + 1 \]

**Inputs:** denominations \( (d_{1}, d_{2}, \ldots, d_{k}) \), units \( n \)

**Outputs:** the count of each denomination \( (c_{d_{1}}, c_{d_{2}}, \ldots, c_{d_{k}}) \).
Algorithm MakeChange$(n, (d_1, d_2, \ldots, d_k))$

for $i \leftarrow 1$ to $n$ do
    $sumc[i] \leftarrow \infty$
for $j \leftarrow 1$ to $k$ do
    $sumc[d_j] \leftarrow 1; coin[d_j] \leftarrow j$
// calculate $sumc[i]$ for $1 \leq i \leq n$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $k$ do
        $temp \leftarrow sumc[i - d_j] + 1$
        if $temp < sumc[i]$ then
            $sumc[i] \leftarrow temp; coin[i] \leftarrow j$
// determine if it is possible to make change
if $sumc[n] = 1$ then return impossible
else // generate answer
    for $j \leftarrow 1$ to $sumc[n]$ do
        $c_{d_j} = 0$ // initialization
        // traverse through coins used to make best change
        $total \leftarrow n$
        while $total > 0$ do
            $cd_{coin[total]} \leftarrow cd_{coin[total]} + 1$
            $total \leftarrow total - d_{coin[total]}$
        return $(c_{d_1}, c_{d_2}, \ldots, c_{d_k})$

The running time of the above algorithm is $O(nk)$. Note that this algorithm is pseudo-polynomial.