Question 1 (Minimum Spanning Tree, 15 points). Compute the minimum spanning tree of the following graph.

We can obtain this with Kruskal’s algorithm. The edges 1 and 2 can be added. Edge 3 cannot be because it creates a cycle with 1 and 2. Edges 4 and 5 can be added but edge 6 creates a cycle with 2 and 5. Edges 7 and 8 can be added, but 9 would create a cycle with 1, 2, 7 and 8 and 10 creates a cycle with 1, 2, 4, 7, 8. We then add edge 11 completing the tree.
Question 2 (Backtracking, 15 points). Irene is trying to show that the graph below does not have a Hamiltonian cycle. Use backtracking to show that none exists.

We consider cycles starting from vertex A. Considering adding one edge to the path at a time. The tree of possibilities is as follows: We note that none of these paths can be extended without repeating a vertex.

Thus, there is no Hamiltonian cycle.
Question 3 (Lilypad Hopping, 15 points). Jeremiah was a bullfrog. He lives in a pond with lilypads located at coordinates \((x_i, y_i)\) for \(1 \leq i \leq n\). Jeremiah can jump between two lilypads if they are separated by a distance of at most 1 unit. Give an algorithm that with inputs \((x_i, y_i)\) determines whether or not it is possible for Jeremiah to reach the lilypad at location \((x_n, y_n)\) from the one at location \((x_1, y_1)\) by a sequence of such hops, and analyze the runtime. For full credit, your algorithm should run in time \(O(n^2)\).

The algorithm is as follows:

Construct a graph \(G\) with vertices \(v_1..vn\)
For \(i = 1\) to \(n\)
    For \(j = i+1\) to \(n\)
        If \(\text{dist}((x_i, y_i), (x_j, y_j)) \leq 1\)
            Add edge \((v_i, v_j)\) to \(G\)
Run \(\text{explore}(v_1)\)
    If \(\text{visited}(v_n)\)
        return "There is a path"
    Else
        return "There is not a path"

This algorithm works because the graph \(G\) constructed has an edge between \(v_i\) and \(v_j\) exactly when Jeremiah is capable of hopping between lilypads \(i\) and \(j\). Thus, running \(\text{explore}\) on \(v_1\) marks as visited exactly the lilypads that can be reached from the first one.

The runtime of this algorithm is \(O(n^2)\). It takes \(O(n^2)\) time to construct \(G\) (coming from the double loop over \(i\) and \(j\)). Running \(\text{explore}\) takes \(O(|V_G| + |E_G|)\) time. However \(|V_G| = n\) and therefore \(|E_G| < n^2\). So the total runtime is \(O(n^2)\).
Question 4 (Longest Palindromic Subsequence, 15 points). Consider the problem of finding the longest palindromic subsequence of a sequence $x_1, x_2, \ldots, x_n$. In particular, this means to find the longest subsequence $y_1, y_2, \ldots, y_k$ (where $y_i = x_{a_i}$ for some sequence $a_1 < a_2 < \ldots < a_k$) so that $y_1 = y_k, y_2 = y_{k-1}, \ldots, y_i = y_{k+1-i}, \ldots, y_k = y_1$.

Give an algorithm to find the length of the longest palindromic subsequence and prove its correctness. For full credit, your algorithm should have runtime $O(n^2)$.

Our general strategy will be to solve this problem by dynamic programming. For $i < j$, we let $A_{i,j}$ be the length of the longest palindromic subsequence of $x_i, x_{i+1}, \ldots, x_j$. If $j = i$, then the longest subsequence is clearly length 1. We note that if $x_i \neq x_j$ then a palindromic subsequence cannot involve both endpoints, and thus will be a palindromic subsequence of either $x_{i+1}, \ldots, x_j$ or of $x_i, \ldots, x_{j-1}$. Thus if $x_i \neq x_j$, $A_{i,j} = \max(A_{i+1,j}, A_{i,j-1})$. If $x_i = x_j$ and $i > j$, then a palindromic subsequence either does not use $x_i$ or does not use $x_j$ or uses both. In the last case, removing the first and last elements of the subsequence gives a palindromic subsequence of $x_{i+1}, \ldots, x_{j-1}$. Thus, if $x_i = x_j$, we have that $A_{i,j} = \max(A_{i-1,j}, A_{i,j-1}, 2 + A_{i+1,j-1})$ (where $A_{i,j}$ is understood to be 0 if $i > j$). This gives us a solvable recurrence relation for $A$ and implies that the following algorithm works:

Let $A$ be an $n \times n$ array
For $i = n..1$
    For $j = 1..n$
        If $i > j$
            $A[i,j] = 0$
        Else if $i = j$
            $A[i,j] = 1$
        Else if $x_i = x_j$
            $A[i,j] = \max(A[i+1,j], A[i,j-1], 2+A[i+1,j-1])$
        Else
            $A[i,j] = \max(A[i+1,j], A[i,j-1])$
    Return $A[1,n]$

We note that this algorithm works because in any given iteration of the inner loop, $A[i', j']$ has already been assigned the correct value for any $i' > i$ or $i' = i, j' < j$. Thus all of the necessary calls have already been assigned the correct values.

To analyze the runtime of this algorithm, note that each iteration of the inner loop takes constant time, and thus the total runtime is $O(n^2)$.
**Question 5** (Local Maximum Search, 20 points). Given a sequence of numbers $a_1, a_2, \ldots, a_n$, we say that $a_i$ is a local maximum if $a_i \geq a_{i+1}, a_{i-1}$ (where the condition is ignored if $a_{i\pm1}$ would be out of range). Give an algorithm that given a sequence $a_1, a_2, \ldots, a_n$ finds a local maximum and analyze its runtime. For full credit, your algorithm should run in time $O(\log(n))$.

**Hint:** Divide the array in two and figure out a way to select one half that is guaranteed to have a local maximum.

We use a recursive procedure $\text{FindLocalMax}(i, j)$ finds a local maximum of the sequence $a_i, \ldots, a_j$. If $i = j$, then $a_i$ is trivially a local maximum of this sequence. Otherwise we select a $k = \lfloor \frac{i+j}{2} \rfloor$ and compare $a_k$ to $a_{k+1}$. If $a_k \leq a_{k+1}$ then a local maximum of $a_{k+1}, \ldots, a_j$ will suffice. This is because if the maximum was not at $k + 1$, it is necessarily larger than its neighbors, and if it’s $a_{k+1}$ we have $a_{k+1} \geq a_{k+2}$ since it’s a local maximum and $a_{k+1} \geq a_k$ by assumption. Similarly, if $a_k \geq a_{k+1}$, a local maximum of $a_i, \ldots, a_k$ suffices. Thus, the following algorithm works:

```plaintext
FindLocalMax(i, n)
FindLocalMax(i, j)
    If i=j
        Return i
    k = floor((i+j)/2)
    If a_k >= a_{k+1}
        Return FindLocalMax(i, k)
    Else
        Return FindLocalMax(k+1, j)
```

To analyze the runtime, we note that this is a divide and conquer algorithm. Each iteration of the problem does constant work to reduce the problem to a single subproblem of nearly half the size. Therefore we have that if $T(n)$ is the runtime for this algorithm on a list of length $n$ that

$$T(n) = T(n/2 + O(1)) + O(1).$$

Therefore, by the Master Theorem, $T(n) = O(\log(n))$. 


Question 6 (Approximation Algorithm for Bin Packing, 20 points). Consider the following problem. Given \( n \) items of weights \( w_1, w_2, \ldots, w_n \leq 1 \) you are tasked with the problem of putting these items into bins so that the total weight of all items in any one bin is at most 1. So for example, if you had items of weight 0.6, 0.5 and 0.3, you could put them all in different bins, or put the first and third in a bin together and the second in a different bin, but you could not put the first and second in the same bin because 0.6 + 0.5 > 1. Your objective is to do this while minimizing the total number of bins used. It turns out that this problem is NP-Hard (there is a relatively simple reduction to Subset Sum).

Give a polynomial time algorithm that provides a 2-approximation for this problem, and prove correctness.

Hint: As long as your bins are full enough that you cannot combine any pair of them, the average weight in each bin is at least 1/2.

We use the following greedy algorithm:

Put each item in a separate bin
While there exist two bins whose total weight is less than 1
   Combine the items in those bins into a single bin

This algorithm clearly takes \( O(n^3) \) time (in fact clever implementations can be made to run in \( O(n \log(n)) \)), since we can combine bins at most \( O(n) \) times, and we can check whether or not there are two combinable bins in \( O(n^2) \) time. The greater difficulty is in showing that this algorithm gives a 2-approximation.

Suppose that this algorithm provides a solution with \( k \) final bins. It must be the case that for any two bins the total weight of all the items in either bin is more than 1, since otherwise we would have combined those bins. Let \( x_1, x_2, \ldots, x_k \) be the total weights of each bin. If \( k = 1 \), then we use only one bin, which clearly cannot be improved upon. Otherwise, \( x_i + x_j > 1 \) for all \( i \neq j \). Therefore, we have that

\[
2(k - 1) \sum_i x_i = \sum_{i \neq j} x_i + x_j > \sum_{i \neq j} 1 = k(k - 1).
\]

Therefore, we have that the total weight of all items is \( \sum_i x_i \), which is at least \( k/2 \). Since no packing is allowed to put more than one unit of weight in any bin, any packing must use at least \( k/2 \) bins. Therefore, the ratio between what our algorithm achieves and the optimal is at most \( k/(k/2) = 2 \). Therefore, we have a 2-approximation.

Alternate Solution: For each item of weight \( w_i \geq 0.5 \) put it in its own bucket. If there are items remaining, take a new bucket and put items in it one at a time until it is filled at to at least half capacity. You will always be able to add another item because all remaining items have weight less than 1/2. Repeat this until there are no items remaining.

This algorithm is clearly linear time, the difficulty comes in proving that it is a 2-approximation. However, it is clear that all buckets that this algorithm produces other than the last one are at least half full. Therefore, if this algorithm uses \( k + 1 \) buckets, the total weights of the items in the first \( k \) buckets is at least \( k/2 \), and therefore, the total weight of all items is strictly more than \( k/2 \). Therefore, the optimal solution must use more than \( k/2 \) buckets, and hence must use at least \( (k + 1)/2 \). Therefore, our algorithm uses at most twice as many buckets as the optimal solution, and therefore is a 2-approximation.