1. In general, if we have a graph with 1 vertex, then it has two independent sets which are the vertex itself or the empty set. If a graph has no vertices, then it has one independent set, the empty set. The recurrences to count the number of independent sets for line, cyclical, and complete binary tree graphs will rely on whether or not we include the first or top (which be made precise) node in the graph.

   Let L_n be the number of independent sets on a line graph and so we know that $L_0 = 1$ and that $L_1 = 2$ per the explanation above.
   Since this grpah is a line we can proceed from left to right and either include or exclude each node has we go along.
   $L_n = L_{n-1} + L_{n-2}$ because if the leftmost node is excluded, then the number of indendent sets on a line graph with the leftmost node exlcuded is the same as the number of independent sets on a line of length $n-1$. If we include the leftmost node, then the second leftmost vertex cannot be in the independent set by definition and so the number if independent sets with the leftmost included is the number of independent sets on $n-2$ vertices. If we say that the first and second Fibonacci numbers, F_1 and F_2, are both 1, then a line graph of length n has F_{n+1} independent sets.

   For a cycle graph we do something similar. Likewise let C_n be the number of independent sets on a graph of n vertices and $C_0 = 1$ and $C_1 = 2$. Also, for cycle graphs, we will add $C_2 = 3$ as a base case (either the independent set is empty or it contains one of the two vertices but not both). To determine the recurrence, let's pick some vertex and label it 1. If vertex 1 is in the independent set, then neither of its two neighbors can be and so we can remove vertex 1 and its neighbors and we are left with a line graph with $n-3$ vertices. Thus if vertex 1 (which could be any vertex by symmetry) is in the independent set, then there are L_{n-3} ways to do this. If vertex 1 is not, then remove it and we have a line graph with $n-1$ vertices and so the number of independent sets without vertex 1 is L_{n-1}. Therefore $C_n = L_{n-1} + L_{n-3}$. For $n \geq 2$, C_n equals the n^{th} Lucas number.

   For the complete binary trees, let T_n be the number of independent sets on a complete binary tree of height n. Simlarly, $T_0 = 1$ and $T_1 = 2$. Suppose we have a complete binary tree of height n, if we have an independent set on this tree, the root is either in the set or not. If it's not, the number of independent sets that do not include it is the square of the number of independent sets on a complete binary tree $n-1$ deep because the root has two distinct complete binary trees $n-1$ deep as its children. If the root is in the independent set, then the vertices that are its direct children cannot be in the independent set, but its four 'grandchildren' (should they exist) which are the roots of trees $n-2$ deep, could be and if the root is in the independent set then there are the number of independent sets for a tree $n-2$ high raised to the fourth power. So, $T_n = T_{n-1}^2 + T_{n-2}^4$.

   If there are 127 nodes in a complete binary tree, then it has a height of 7.
   $T_7 = 13345346031444632841427643906$

2. For randomized k-cut, run thorugh all the vertices and assign each vertex to one of the k groups with probability $\frac{1}{k}$. The probability that a given edges crosses one of the cuts is the probability that the two vertices at either end of the edge are in different cuts. This happens with probability $\frac{k-1}{k}$. Therefore the expected size of the cut is $\frac{k-1}{k}|E|$, which for k equals 2, reduces to a 2-approximation as we showed in class. This process takes $O(|V|)$ time to assign every vertex a cut and then to actually calculate the size of the cut we have to traverse over the entire graph which will take $O(|V| + |E|)$ time.

   The local search approximation algorithm for k cut runs similarly to the one presented in class. First, split the vertices randomly into k sets, $S_1, S_2, ..., S_k$. Then, so long as there is a vertex that will strictly increase the size of the k cut by moving it to another set, move it. This algorithm will terminate because the largest value could even be is $|E|$ and since the algorithm increases the cut by exactly

1 each time it finds an improvement, we will do $|E|$ updates at most and so it will terminate. This process will run in $O(|E|(|V|+|E|))$ time because we will have to do at most $|E|$ update steps and at each step we have to check all the $|V|$ vertices to see if moving them to a different partition will increase the size of the cut and at each vertex we have to check all of its neighbors and as we iterate over the entire graph we will go down each edge twice which is $O(|V|+|E|)$ amount of work and so we get the total runtime is $O(|E| \cdot (|V|+|E|))$.

Just as in class I'll count every edge that crosses a cut as $\frac{1}{2}$ because we'll double count them in the summation.

$$\begin{aligned} C &= \frac{1}{2}\sum_{j=1}^{k}\sum_{v\in S_j} |\{w : (v,w)\in E, w \notin S_j\}| \\ &\geq \frac{1}{2}\sum_{j=1}^{k}\sum_{v\in S_j} \frac{k-1}{k}\delta(v) \\ &= \frac{1}{2}\frac{k-1}{k}\cdot 2 \cdot |E| \\ &= \frac{k-1}{k}\cdot |E| \end{aligned}$$

which is again analogous to the result we found in class for the max 2-cut. The inequality in the second line comes from the fact that we know that at least $\frac{k-1}{k}$ of a vertex's edges must belong to other sets, because otherwise we could improve the max-cut. $\delta(v)$ gives the degree of a vertex and when we sum over all the degrees of vertices we get twice the number of edges, because every edge is counted twice.

3. Before showing that assuming we had this polynomial 2-approximation algorithm that we coudl then derive a $(1+\epsilon)$-approximation algorithm, I'll show that G' (as defined in the problem) as a max-clique of k^2.
If G has a clique of size k, number the vertices in that clique $1, 2, ..., k$. For every i, j pair with $1 \leq i, j, \leq k$, edge (i, j) is in the graph. In G' we will have k^2 nodes that will correspond to pairs of these nodes in this k-clique. Let (a, b) and (c, d) be 2 nodes in these k^2 nodes. There must be an edge between them in G' because either there was an edge between a and c in G or $a = c$ and there was either an edge between b and d in G (because they were in the same clique) or $b = d$. Therefore the clique in G' is at least as large as k^2 because every pair of nodes that is a pair of two nodes which were in G are connected to every other pair. Now to show the G' has a clique that is exactly k^2 in size, suppose there were some vertex g not in the original k-clique but is in this clique in G'. Because g was not in the original k clique there must have be some node i in G such that there was no edge (g, i) in G. And so for any $1 \leq v, x \leq k$, $\{(g, v), (i, x)\}$ cannot be an edge and so the clique in G' is exactly k^2 in size.
Now let's say that G has a max-clique of size k, then G' has a max-clique of size k^2. If we run this hypothetical 2-approximation polynomial time algorithm on G', it will find a clique that is at least as large as $\frac{k^2}{2}$ and so we know that the max clique in G is at least as large as $\frac{k}{\sqrt{2}}$. Thus by computing G' and running this algorithm, we end up with a $\sqrt{2}$ approximation algorithm for max-clique. If we compute $G^{(n)}$ that is we use n-tuples for the vertices and an analogous rule for the edges, we can see inductively that this will have a k^n clique (G^2 would correspond to G' in this problem). Therefore, by generating $G^{(n)}$ (which will only take a polynomial amount of time because the number of vertices is equal to $|V|^n$ and the number of edges is no more than this value squared and so both are polynomial), we can then run this polynomial 2-approximation algorithm on $G^{(n)}$, which will take an amount of time that is polynomial in the size of $G^{(n)}$ and thus polynomial in the size of G and we can conlcude that the n^{th} root of whatever the algorithm returns is between $\frac{k}{\sqrt[n]{2}}$ and k. As $n \to \infty$, $\sqrt[n]{2} \to 1+\epsilon$.

4. To show that this is a $\frac{4}{3}$ approximation algorithm, let's suppose that it's not and that it has found some configuration such that the load on machine 1 is m_1, the load on machine 2 is m_2, $m_1 > m_2$, and

$m_1 > \frac{4}{3}OPT$ where OPT is the optimal time on the same set of jobs. We can assume that $m_1 > m_2$ because if they were equal then our solution would be optimal.

We know that $m_1 + m_2 < 2OPT$ because $m_1 + m_2$ is the same as the total sum of the runtimes jobs and OPT must be at least half the sum of the runtimes of all the jobs. From that we can conclude that $m_2 < \frac{2}{3}OPT < \frac{1}{2}m_1$.

So if machine 1 has one job, then we must actually be in the optimal arrangement and so the assumption that $m_1 > \frac{4}{3}OPT$ doesn't hold. If machine 1 has 2 or more jobs then the smallest job must have a runtime less than or equal to $\frac{1}{2}m_1$ and so we can move this job to machine 2 and so the new makespan on m_2 will be less than m_1 because m_2 was strictly less than $\frac{1}{2}m_1$. If the resulting state is not less than $\frac{4}{3}$ the optimal then we would have to be able to continue this same process by the exact same argument until the difference between the two machines's makespans is less than the size of the smallest job on machine 1 and we'll be in a stable state. Thus it is not possible to find a stable arrangment using this algorithm that is greater that has a makespan greater than $\frac{4}{3}$ of the optimal makespan because we could take the smallest job on m_1 and move it to m_2.

I collaborated on this assignment with Lauren Kim.