1. If we restrict the problems we look at, sometimes hard problems like counting the number of independent sets are in a graph become solvable. For instance, consider a graph that is a line on n vertices. (That is, the vertices are labelled 1 to n, and there is an edge from 1 to 2, 2 to 3, etc.) How many independent sets are there on a line graph? Also, how many independent sets are there on a cycle of n vertices? (Hint: In this case, we want to express your answer in terms of a family of numbers – like "For n vertices the number of independent sets is the nth prime." And that's not the answer.)

   Similarly, describe how you could quickly compute the number of independent sets on a complete binary tree. (Here, just explain how to compute this number.) Calculate the number of independent sets on a complete binary tree with 127 nodes. (Warning: it's a pretty big number.)

2. Consider the problem MAX-k-CUT, which is like the MAX CUT algorithm, except that we divide the vertices into k disjoint sets, and we want to maximize the number of edges between sets. Explain how to generalize both the randomized and the local search algorithms for MAX CUT to MAX-k-CUT and prove bounds on their performance.

3. Prove that if there exists a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of 2, then there is a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of $(1+\varepsilon)$ for any constant $\varepsilon > 0$. The degree of the polynomial may depend on ε. Hint: for a starting graph $G = (V,E)$, consider the graph $G \times G = (V',E')$, where the vertex set V' of $G \times G$ is the set of ordered pairs $V' = V \times V$, and $\{(u,v),(w,x)\} \in E'$ if and only if

   $$[\{(u,w)\} \in E \text{ or } u = w] \text{ and } [\{(v,x)\} \in E \text{ or } v = x].$$

   If G has a clique of size k, then how large a clique does G' have?

4. We consider the following scheduling problem, similar to one that we studied before: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. We place a subset of the jobs on each machine. Each job j_i has an associated running time r_i. The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the completion time, sometimes called the *makespan*, which is the maximum load over all machines.

   Consider the following local search algorithm. Start with any arbitrary assignment of jobs to machines. We then repeatedly *swap* a single job from one machine to another, if that swap will *strictly reduce* the completion time. (We won't make a move if the completion time stays the same, and only one job moves in each swap.) If a swap is not possible, we are in a stable state. For example, suppose we had jobs with running times 1,2,3,4, and 5, and we started with the jobs with running times 1,2, and 3 on machine 1, and the jobs with running times 4 and 5 on machine 2. This is a stable state, but it is not optimal; the minimum possible completion time is 8, and this stable state has completion time 9.

   Prove that the local search algorithm always terminates in a stable state, and that the completion time is within a factor of 4/3 of the optimal.