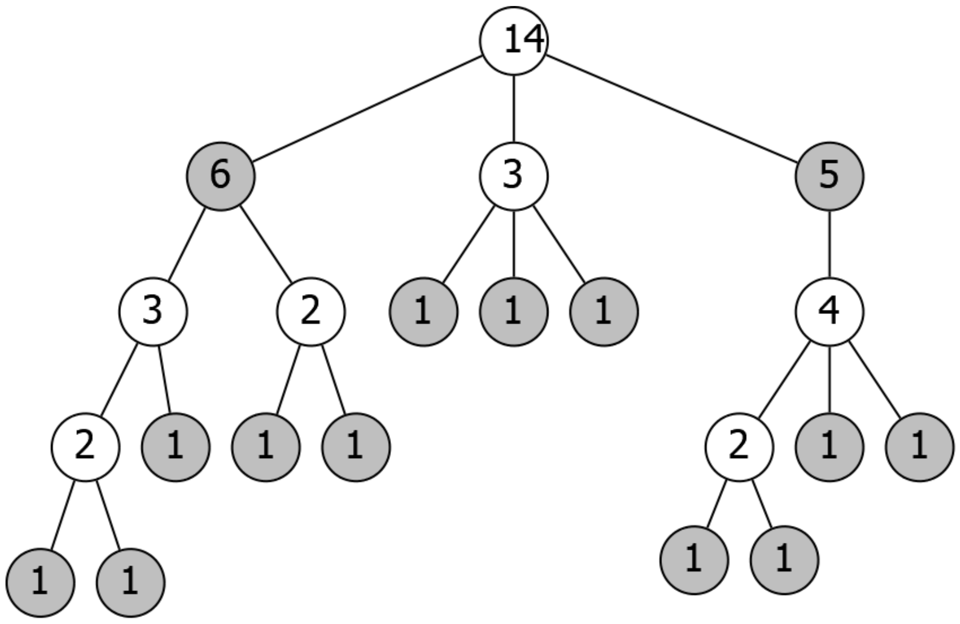
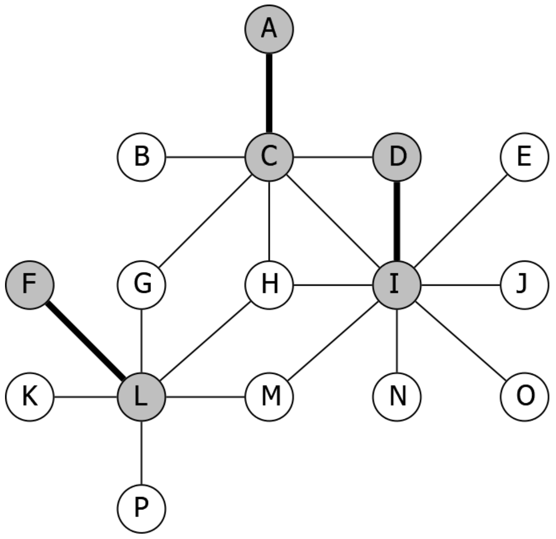
Question 1 (Maximum Independent Set, 30 points)**.** *What is the size of the maximum independent set in the tree below?*

Note: You do not need to provide the set itself, just its size.



The answer is 14. Running the algorithm for maximum independent set in trees, we get the output shown. An example of an independent set of size 14 is given by the shaded vertices.

Question 2 (Vertex Cover, 30 points)**.** *When running the* 2*-approximation algorithm for vertex cover on the following graph, what cover is returned? When the algorithm has multiple possible options of which vertices to add next, please make sure to prioritize adding the alphabetically first valid option.*



The algorithm first adds vertices A and C (as A is the earliest vertex with an uncovered edge and it is connected only to C). Next it adds D and I, since D is the alphabetically first vertex with an uncovered edge. Finally, it adds F and L since F is the first vertex with an uncovered edge. At this point, we have a vertex cover, so the answer is $\{A, C, D, I, F, L\}$. Note that the smallest vertex cover here is $\{C, I, L\}$ of size 3.

Question 3 (Shortest Cycle, 35 points)**.** *Give an algorithm that given an unweighted, undirected graph G computes the length of the shortest cycle in G. Note that a cycle here is not allowed to repeat edges.*

For full credit, your algorithm should run in polynomial time in the number of edges and vertices of G.

The algorithm is as follows:

```
ShortestCycle(G)
  Best <- Infinity
  For each edge e = (u,v) in G
    Let G_e = G with e removed
    Use BFS to compute length of the shortest path
      from u to v in G_e and call it L
    Best <- min(Best,L+1)
  Return Best
```

To analyze the runtime of this algorithm, we note that the main loop is executed $O(|E|)$ times and that the runtime of each iteration is dominated by the `BFS` call, which takes $O(|V| + |E|)$ time. Therefore, the final runtime is $O(|E|(|V| + |E|))$, which is polynomial.

To show correctness, we note that a cycle must contain some edge $e = (u, v)$. Such a cycle must consist of e plus a path from u to v that does not use e (i.e. a path from u to v in G_e). The shortest length of such a path is computed by `BFS`, and the length of the shortest such cycle is one more than that. The answer is just the minimum value of this over all edges e, which our algorithm computes.

Note: There is a slightly more complicated algorithm with runtime $O(|V|(|V| + |E|))$. Essentially, using the fact that one can compute the length of the shortest cycle through a given vertex in time $O(|V| + |E|)$. Repeating this for each vertex in G and taking a minimum gives the final answer.

Question 4 (Minimal Effort Academics, 35 points)**.** *Gamzee needs to take $2n$ classes in school. The i^{th} class requires a_i hours of study to pass, but b_i hours of study to get an A in it (for some $b_i \geq a_i \geq 0$). In order to satisfactorily complete his schooling, Gamzee must pass all $2n$ classes and must get As in at least n of them. Give an algorithm that given a_i, b_i determines the fewest number of hours of study necessary to accomplish this goal.*

For full credit, your algorithm must run in expected time $O(n)$ or better.

We note that Gamzee's total workload is equal to the sum of all of the a_i's plus the sum of the values of $b_i - a_i$ in the classes that he wishes to get an A in. In order to minimize this, he will want to get an A in the n classes with $b_i - a_i$ as small as possible. The algorithm for this is as follows:

```
MinimalEffort(A,B)
  Let PassEffort = Sum of elements of A
  Let ExtraEffort be a list of length 2n
  For i = 1 to 2n
    ExtraEffort[i] = B[i] - A[i]
  Let Bound = OrderStatistics(ExtraEffort,n+1)
  Let S be the set of elements of ExtraEffort less than or equal to Bound
  Let ExtraSum be the sum of the elements of S plus Bound*(n-|S|)
  Return PassEffort + ExtraSum
```

Note that the order statistics computation takes $O(n)$ time and the rest of this algorithm is clearly linear time. `Bound` computes the n^{th} smallest value of $b_i - a_i$, so the values of $b_i - a_i$ we want are all the values less than `Bound`, plus enough values equal to `Bound` (usually just one) to make there by n in total.

Question 5 (Gameshow Finale, 35 points)**.** *Dirk returns to the gameshow one more time. As before there are n challenges available to him. Again, each challenge has a probability p_i that he will pass the challenge and a (non-negative) reward R_i that he will win if he does pass it. Dirk may attempt any number of challenges in any order, but may not attempt the same challenge more than once. He may quit at any time and leave with all of the rewards of challenges that he has completed so far, however if Dirk fails any challenge he attempts, he will forfeit any rewards won thus far, and leave with nothing.*

Dirk would like to find an algorithm that given p_i and R_i determines the maximum possible expected reward that he can achieve. Prove that this problem is NP-Hard.

Hint: Consider instances where $p_i = e^{-R_i}$ for each i. You may use the fact that the function $f(x) = xe^{-x}$ achieves its maximum value of e^{-1} at $x = 1$ and nowhere else.

We first note that Dirk's problem is an NP Optimization problem. If Dirk's strategy is to attempt all challenges in some set S (in any order) before quitting, he has a probability of $\prod_{i\in S} p_i$ of passing all challenges and winning $\sum_{i\in R} R_i$ reward. He otherwise wins nothing. Thus, Dirk's expected reward is $\left(\prod_{i\in S} p_i\right)\left(\sum_{i\in S} R_i\right)$. He wants to find a set S maximizing this easily computable objective function.

To show hardness, we use a reduction from `SubsetSum`. Consider the subset sum problem with a set T of positive numbers and a target sum value of C. We produce an instance of Dirk's problem where for each element $x_i \in T$ there is a challenge with $p_i = e^{-x_i/C}$ and $R_i = x_i/C$. We note that if Dirk selects a set S of challenges, his expected reward will be:

$$\left(\prod_{i\in S} p_i\right)\left(\sum_{i\in S} R_i\right) = \left(\prod_{i\in S} e^{-x_i/C}\right)\left(\sum_{i\in S} x_i/C\right) = xe^{-x},$$

where $x = \sum_{i\in S} x_i/C$. We note that the function $f(x) = xe^{-x}$ takes a unique maximum value of e^{-1} at $x = 1$. Thus, Dirk can achieve an expected reward of e^{-1} or better if and only if there is a set S of challenges so that $x = \sum_{i\in S} x_i/C = 1$, or in other words if there is a set S of challenges so that $\sum_{i\in S} x_i = C$. In particular, Dirk's optimal reward is at least e^{-1} if and only if there is a solution to the original Subset Sum problem. This gives a reduction from Subset Sum to Dirk's problem, and thus Dirk's problem is NP-Hard.

Question 6 (Palindrome Detection, 35 points)**.** *A* palindrome *is a collection of words whose letters (ignoring spaces) form the same sequence if read forwards as they would if read backwards. For example, "`A MAN A PLAN A CANAL PANAMA`" is a famous palindrome. As in the above example, a palindrome is allowed to use the same word more than once.*

Give an algorithm that given a list of k words, each of which is provided as a string of at most n letters, determines whether or not there is a (non-empty) palindrome consisting only of words from that list.

For full credit your algorithm should run in time polynomial in k and n.

Hint: Try constructing the solution by building up both ends of the palindrome, adding one word at a time to either end, trying to keep the two sides roughly balanced.

Our strategy will be to construct the palindrome by adding words to both the start and the end of it so that at any point one side is longer than the other by at most part of the last word added to that side. For example, in the palindrome given above we might have `A MAN A PLAN` from the start and `CANAL PANAMA` at the end. The words at the start are the reverse of the last 9 letters of the words at the end, leaving the first two letters `CA` in `CANAL` left over.

While doing this, we keep track of the letters that are unmatched. This is either a suffix of a word at the beginning or a prefix of a word at the end. If we have an unmatched word at the beginning, we must add a word at the end that matches its reverse as much as possible. This leaves two possibilities either we have the leftover suffix $W^R A$ (where W^R is the reverse of a word W) and add W to the end and are left with the remaining string A at the start, or we have the leftover suffix A, add a word BA^R to the end, and then are left with an overhanging prefix B. Similarly, if we have leftover letters at the end of the word, we can either go from having prefix AW^R to prefix A by adding the word W to the beginning or go from prefix A to suffix B by adding the word $A^R B$ to the beginning.

This continues until the unmatched section is itself a palindrome.

It is not hard to see that we can produce a palindrome using words from our vocabulary if and only if there is a sequence of such moves that can be capped off by a final word as described above. This can be determined using a graph exploration algorithm. In particular,

Define a directed graph G where the set of vertices consist of either:

- A prefix of a word in the vocabulary
- A suffix of a word in the vocabulary

the edges of G connect either:

- The suffix AB to the suffix B if A^R is a word
- The suffix A to the prefix B if BA^R is a word
- The prefix AB to the prefix A if B^R is a word
- the prefix A to the suffix B if $A^R B$ is a word

After creating G, run explore on the vertices corresponding to each word (as either a prefix or suffix of itself). If any of them can reach any vertex which is a palindrome, return `YES`, otherwise return `NO`.

Correctness of this algorithm is due to the discussion above. As for runtime, G consists of at most $O(kn)$ vertices, which can be produced in linear time. For each vertex, we can find the edges coming from that vertex by checking every possible word that could be added to the other side and spending $O(n)$ time to see if they match appropriately and if so which vertex they should connect to. Thus G can be computed in $O(k^2n^2)$ time and has $O(k^2n)$ edges. Running explore from each vertex of G takes time $O(k^3n^2)$ time and checking whether or not any vertex we have found is a palindrome takes $O(kn^2)$ time. Thus, this algorithm runs in time $O(k^3n^2)$.

Note: A more optimized version of this algorithm runs in time $O(k^2n^2)$.