During an unprecedented Dos-equis-virus outbreak, the local emergency call center receives calls from \( n \) people who each need to be airlifted to one of \( k \) local hospitals. Each patient must be flown to a hospital within 20 miles of their home; however, we do not want to overload any single hospital.

Describe an algorithm that assigns each patient to a hospital while keeping the maximum number of people flown to any single hospital as small as possible. Your input is an array \( D[1..n, 1..k] \), where \( D[i, j] \) is the distance in miles from patient \( i \)'s home to hospital \( j \). Analyze your algorithm as a function of \( n \) (the number of patients) and \( k \) (the number of hospitals).

**Solution:** We build a bipartite graph \( G = (P \sqcup H, E) \), where \( P = [n] \) is the set of patients, \( H = [k] \) is the set of hospitals, and each edge \( i j \) indicates that patient \( i \) lives within 20 miles of hospital \( j \). Building this graph takes \( O(nk) \) time.

Now we can proceed exactly as in Homework 8.2. First consider the decision problem: Can we fly each patient to a close hospital, so that at most \( \ell \) patients are flown to each hospital?

Direct every edge in \( G \) from patient to hospital, and assign each edge capacity 1. Then we add a source node \( s \), edges with capacity 1 from \( s \) to every person node, a target node \( t \), edges with capacity \( k \) from each hospital node to \( t \). The resulting graph has \( O(n + k) \) vertices and at most \( O(nk) \) edges. Finally, compute the maximum flow in the modified graph. If this flow saturates every edge out of \( s \), it assigns each patient to a hospital and assigns at most \( \ell \) patients to each hospital; otherwise, no such assignment is possible. The decision algorithm runs in \( O(VE) = O(nk(n + k)) \) time.

To solve the optimization problem, we perform a binary search over all possible values of \( \ell \) from 1 to \( n \), running the decision problem at each step. The entire algorithm takes \( O(nk(n + k) \log n) \) time.

**Rubric:** 10 points: standard graph reduction rubric. +2 extra credit for \( O(nk(n + k)) \) time; faster algorithms are similar to the faster solutions to HW 8.2. No penalty for assuming \( n > k \) in time analysis.
1. Let $G = (V, E)$ be an arbitrary dag with a unique source $s$ and a unique sink $t$. Suppose we compute a random walk from $s$ to $t$, where at each node $v$ (except $t$), we choose an outgoing edge $v \rightarrow w$ uniformly at random to determine the successor of $v$.

(a) Describe and analyze an algorithm to compute, for every vertex $v$, the probability that the random walk visits $v$.

**Solution:** For any vertex $v$, let $\text{Prob}(v)$ denote the probability that our random walk visits $v$. This function satisfies the following recurrence, where $\text{outdeg}(u)$ denotes the number of edges leaving $u$.

$$\text{Prob}(v) = \begin{cases} 
1 & \text{if } v = s \\
\sum_{u \rightarrow v} \text{Prob}(u)/\text{outdeg}(u) & \text{otherwise}
\end{cases}$$

We need to compute this function for every vertex $v$.

We can memoize this function into the dag itself, adding a field $v.\text{Prob}$ to every node $v$, and we can evaluate the function for all $v$ in forward topological order, in $O(V + E)$ time.

```python
RandomWalkProbabilities(G):
    topologically sort G
    for all vertices v in topological order
        if v = s
            v.Prob ← 1
        else
            v.Prob ← 0
        for all edges u → v
            v.Prob ← v.Prob + u.Prob / u.outdeg
```

Equivalently, as usual, we can evaluate this function for all $v$ by by performing a depth-first search of the reversed graph $G^R$ starting at $t$.

```python
RandomWalkProbabilities(G):
    for all vertices v
        v.Prob ← -1 (Sentinel value)
        t.Prob ← RecRWP(t)

RecRWP(v):
    if v.Prob < 0
        if v = s
            v.Prob ← 1
        else
            v.Prob ← 0
        for all edges u → v
            v.Prob ← v.Prob + RecRWP(u) / u.outdeg
    return v.Prob
```

**Rubric:** 5 points: standard dynamic programming rubric.
(b) Describe and analyze an algorithm to compute the expected number of edges in the random walk.

Solution: Linearity of expectation implies that the expected number of vertices in the random walk is exactly \( \sum_v \text{Prob}(v) \). So the following algorithm computes the expected number of edges in the random walk in \( O(V + E) \) time.

\[
\text{RandomWalkExpectedLength}(G): \\
\quad \text{RandomWalkProbabilities}(G) \\
\quad \ell \leftarrow 0 \\
\quad \text{for all vertices } v \\
\quad \quad \ell \leftarrow \ell + v.\text{Prob} \\
\quad \text{return } \ell - 1
\]

Rubric: 5 points = 4 for algorithm + 1 for running time. Standard DP rubric for solutions that use DP directly.
2. Consider the following randomized version of mergesort. The input is an unsorted array \( A[1..n] \) of distinct numbers. The Merge subroutine takes two sorted arrays as input and returns a single sorted array, containing the elements of both input arrays, in linear time.

\[
\text{RandomizedMergeSort}(A[1..n]):
\]

\[
\text{if } n \leq 1 \\
\hspace{1cm}\text{return } A \\
\ell \leftarrow 0; \ r \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
\hspace{1cm}\text{with probability } 1/2 \\
\hspace{1cm}\ell \leftarrow \ell + 1 \\
\hspace{1cm}L[\ell] \leftarrow A[i] \\
\hspace{1cm}\text{else} \\
\hspace{1.5cm}r \leftarrow r + 1 \\
\hspace{1.5cm}R[r] \leftarrow A[i] \\
L \leftarrow \text{RandomizedMergeSort}(L[1..\ell]) \\
R \leftarrow \text{RandomizedMergeSort}(R[1..r]) \\
\text{return Merge}(L, R)
\]

(a) Fix two arbitrary indices \( i \neq j \). What is the probability that \( A[i] \) and \( A[j] \) appear in the same recursive subproblem (either \( L \) or \( R \))?

**Solution:** 1/2

**Rubric:** 1 point.

(b) What is the probability that \( A[i] \) and \( A[j] \) appear in the same subproblem for more than \( k \) levels of recursion?

**Solution:** 1/2^k

**Rubric:** 1 point. No penalty for 1/2^{k-1} or 1/2^{k+1} (off-by-one errors in level counting), and similarly for later parts.

(c) What is the expected number of pairs of items that appear in the same subproblem for more than \( k \) levels of recursion?

**Solution:** \( \binom{n}{2}/2^k \)

**Rubric:** 2 points.
(d) Give an upper bound on the probability that at least one pair of items appear in the same subproblem for more than $k$ levels of recursion. Equivalently, upper bound the probability that the recursion tree of `RandomizedMergeSort` has depth greater than $k$.

**Solution:** \( \binom{n}{2}/2^k \)

Let $X$ be the number of pairs that appear together in some level-$k$ subproblem. Then $\Pr[X \geq 1] \leq E[X]/1 = \binom{n}{2}/2^k$ by Markov’s inequality.

**Solution:** \( \binom{n}{2}/2^k \)

Fix $k$. Let $X_{ij}$ indicate whether $A[i]$ and $A[j]$ appear in the same level-$k$ subproblem, let $X = \sum_{i<j} X_{ij}$ be the number of pairs that appear together in some level-$k$ subproblem, and finally let $\mu = E[X] = \binom{n}{2}/2^k$. The variables $X_{ij}$ are pairwise independent, so we can apply Chebyshev’s inequality to bound $\Pr[X \geq 1]$ as follows:

\[
\Pr[X \geq 1] = \Pr[X \geq (1 + \delta)\mu] \quad \text{where} \quad \delta = \frac{1}{\mu} - 1
\]

\[
< \frac{1}{\delta^2\mu} \quad \text{by Chebyshev}
\]

\[
= \frac{1}{(1 - \mu)^2}
\]

This upper bound is slightly larger than $\mu$, but not enough to matter.

**Solution (1 point partial credit):** $1 - (1 - 1/2^k)^\binom{n}{2}$


On the other hand, the World’s Most Useful Approximation ($1 + x \approx e^x$ when $x \approx 0$) implies $1 - (1 - 1/2^k)^\binom{n}{2} \approx 1 - e^{-\binom{n}{2}/2^k} \approx 1 - \left(1 - \frac{n}{2}\right)^2/2^k$, so this is actually pretty close, at least when $k \gg 2\log n$.

**Rubric:** 2 points.

- 1 point for expression equivalent to $\binom{n}{2}/2^k$ after significant simplification
(e) For what value of $k$ is the probability in part (d) at most $1/n$?

**Solution:** $k = 3 \log n$

If $k = 3 \log n$, then $\Pr[X \geq 1] \leq \binom{n}{2}/2^k = \binom{n}{2}/n^3 = n(n-1)/2n^3 < 1/n$. ■

**Rubric:** 2 points = 1 for “$\log n$” + 1 for leading constant $\geq 3$.
- 1½ for expression equivalent to $3 \log n \pm O(1)$ after significant simplification (for example: $k = \log(n^{1/2})$).
- 1 point for $\Theta(\log n)$. The leading constant must be at least 3.

(f) Prove that `RANDOMIZEDMERGESORT` runs in $O(n \log n)$ time with probability at least $1 - 1/n$.

**Solution:** The total work at each level of the recursion tree is $O(n)$. We just proved that with probability at least $1 - 1/n$, the recursion tree has depth at most $3 \log n$. We conclude that with probability at least $1 - 1/n$, `RANDOMIZEDMERGESORT` runs in at most $3 \log n \cdot O(n) = O(n \log n)$ time. ■

**Rubric:** 2 points. This is not the only correct proof.
4. Suppose we are given a target string $T[1..n]$ and an list of $k$ fragment strings $F_1[1..m_1]$, $F_2[1..m_2], \ldots, F_k[1..m_k]$. Describe and analyze an algorithm to find the shortest sequence of fragment strings $F_i$ whose concatenation is the target string $T$. You can assume that such a sequence exists. The same fragment $F_i$ can be used multiple times. Express the running time of your algorithm in terms of the parameters $n$, $k$, and $m = \sum_i m_i$.

**Solution (dynamic programming):** This is an example of the text segmentation problem described in class. Let $\text{MinFrag}(i)$ denote the minimum number of fragment strings whose concatenation is $T[i..n]$; we need to compute $\text{MinFrag}(1)$. This function obeys the following recurrence:

$$
\text{MinFrag}(i) = \begin{cases} 
0 & \text{if } i > n \\
\min \{1 + \text{MinFrag}(j) \mid T[i..j-1] = F_\ell \text{ for some index } \ell\} & \text{otherwise}
\end{cases}
$$

(As usual, we define $\min \emptyset = \infty$.)

We can memoize this function into a one-dimensional array $\text{MinFrag}[1..n]$, which we can fill from right to left. To evaluate each subproblem $\text{MinFrag}(i)$, we compare the prefix $T[i..i+m_\ell-1]$ to the fragment $F_\ell$, for each index $\ell$. Performing these string comparisons by brute force requires $O(m)$ time. Thus, the overall algorithm runs in $O(mn)$ time.

However, we can speed up this algorithm using the KMP string-matching algorithm. In a preprocessing phase, for each fragment index $\ell$, we find all indices $i$ such that $T[i..i+m_\ell-1] = F_\ell$, in $O(m_\ell + n)$ time. The total preprocessing time is $O(m + nk)$. Then evaluating each subproblem $\text{MinFrag}(i)$ takes only $O(k)$ time. The overall algorithm runs in $O(m + nk)$ time.

Replacing KMP with more advanced data structures—either suffix trees or Aho-Corasick automata—the running time can be reduced in practice to $O(m + n + z)$, where $z$ is the number of matches between fragments and substrings of $T$. Unfortunately, in the worst case, every fragment matches $\Theta(n)$ substrings of $T$, which means $z = \Theta(kn)$, so the worst-case running time is still $O(m + nk)$.

Directly using the Rabin-Karp rolling hash algorithm instead of KMP does not speed up the naive dynamic programming algorithm. It’s true that each mismatch between a fragment and a substring of $T$ takes $O(1)$ expected time to detect. But whenever a fragment $F_\ell$ matches a substring of $T$, we must spend $O(m_\ell)$ time to explicitly match the strings; matching the hash values is not enough! In the worst case, every fragment matches $\Theta(n)$ substrings of $T$, so the overall worst-case running time is still $O(mn)$.

But in fact we can get a faster algorithm using rolling hashes. Suppose we adjust the parameters of Rabin-Karp algorithm so that the probability of a false match between any fragment $F_\ell$ and any substring of $T$ of length $m_\ell$ is at most $O(1/kn^2)$ instead of $O(1/m_\ell)$. (For example, choose the random prime number $q$ in $\text{RABINKARP}$ between 2 and $[k^2n^4\lg kn]$.) Then the expected number of false matches, summed over all fragments and substrings of $T$, is less than $1/n$. So by Markov’s inequality, the probability of even one false match during the entire algorithm is at most $O(1/n)$. So we can proceed as follows:
• Find all hash-matches between fragments and substrings of $T$ in $O(m + kn)$ time.
• Run the dynamic-programming algorithm, treating hash-matches as true matches, in $O(kn)$ time.
• Finally, verify that the solution to the dynamic program is a sequence of true matches, by brute force, in $O(n)$ time. If we discover a false match in the dynamic programming solution, restart the algorithm from scratch (with new random rolling hash functions).

Each attempted run of the algorithm takes $O(m + kn)$ time and succeeds with probability at least $1 - O(1/n)$. It follows that the overall algorithm runs in $O(m + kn)$ expected time!

Rubric: 10 points = 8 for dynamic programming + 2 for KMP optimization

Solution (dag traversal): We define a directed acyclic graph $G = (V, E)$ as follows:

• $V = \{1, 2, \ldots, n + 1\}$
• $E = \{i \to j \mid T[i..j-1] = F_{\ell} \text{ for some index } \ell\}$.

For example, here is the graph corresponding to the text ABRACADABRA and the fragments A, ABRA, ARC, BRAC, CAD, DAB, RA:

![Graph Diagram]

For all indices $i$ and $\ell$, we can check whether fragment $F_{\ell}$ is a prefix of $T[i..n]$ in $O(m_{\ell})$ time by brute force. Each such matching fragment gives us an edge leaving vertex $i$. Thus, we can construct $G$ in $O(mn)$ time.

However, we can actually find all substrings of $T$ that match any fragment $F_{\ell}$ in $O(m_{\ell} + n)$ time using the KMP string matching algorithm. This optimization reduces the time to construct $G$ to $O(m + nk)$.

Every path in $G$ from 1 to $n + 1$ corresponds to a sequence of fragments whose concatenation is the string $T$. Thus, we need to compute the shortest path from vertex 1 to vertex $n + 1$ in $G$. Because $G$ is unweighted, we can compute this shortest path in $O(V + E) = O(kn)$ time.

The overall algorithm runs in $O(mn)$ time if we construct $G$ by brute force, or $O(m + kn)$ time if we use KMP optimization.

Comments on the previous solution also apply to this one.

Rubric: 10 points = 8 for graph reduction (standard rubrics) + 2 for KMP optimization
5. Suppose we are given a standard flow network \( G = (V, E) \), with a source vertex \( s \), a target vertex \( t \), and capacities \( c(e) \geq 0 \) for every edge \( e \). Suppose each edge in \( G \) also has a color. A flow \( f \) in \( G \) is color-consistent if \( f(e) = f(e') \) for every pair of edges \( e \) and \( e' \) with the same color. The maximum color-consistent flow problem asks for a color-consistent flow with maximum value. The standard maximum flow problem is the special case where every edge has a different color.

Describe a linear program whose solution is the maximum color-restricted flow in \( G \). [Hint: Modify the standard linear program for maximum flow. Don’t try to actually compute this flow.]

**Solution:** We modify the standard maxflow LP by adding one equality constraint for each pair of edges with the same color. Let \( \text{color}(u \rightarrow v) \) denote the color of edge \( u \rightarrow v \).

\[
\begin{align*}
\text{max} & \quad \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s) \\
\text{s.t.} & \quad \sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) = 0 \quad \text{for each vertex } v \neq s, t \\
& \quad f(u \rightarrow v) \leq c(u \rightarrow v) \quad \text{for each edge } u \rightarrow v \\
& \quad f(u \rightarrow v) \geq 0 \quad \text{for each edge } u \rightarrow v \\
& \quad f(u \rightarrow v) - f(u' \rightarrow v') = 0 \quad \text{for each pair of edges } u \rightarrow v \text{ and } u' \rightarrow v' \\
& \quad \text{such that } \text{color}(u \rightarrow v) = \text{color}(u' \rightarrow v')
\end{align*}
\]

**Solution:** We introduce a new variable \( F(x) \) for each color \( x \), storing the flow value of all edges with that color. We also add equality constraints connecting edge flow values to colors. Let \( \text{color}(u \rightarrow v) \) denote the color of edge \( u \rightarrow v \).

\[
\begin{align*}
\text{maximize} & \quad \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s) \\
\text{subject to} & \quad \sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) = 0 \quad \text{for each vertex } v \neq s, t \\
& \quad f(u \rightarrow v) \leq c(u \rightarrow v) \quad \text{for each edge } u \rightarrow v \\
& \quad f(u \rightarrow v) \geq 0 \quad \text{for each edge } u \rightarrow v \\
& \quad f(u \rightarrow v) - F(\text{color}(u \rightarrow v)) = 0 \quad \text{for each edge } u \rightarrow v
\end{align*}
\]
Solution: Let $\text{color}(u \rightarrow v)$ denote the color of edge $u \rightarrow v$. We modify the standard maxflow LP by associating flow values with colors instead of individual edges.

$$\begin{align*}
\text{max} & \quad \sum_w f(\text{color}(s \rightarrow w)) - \sum_u f(\text{color}(u \rightarrow s)) \\
\text{s.t.} & \quad \sum_w f(\text{color}(v \rightarrow w)) - \sum_u f(\text{color}(u \rightarrow v)) = 0 \quad \text{for each vertex } v \neq s, t \\
& \quad f(\text{color}(u \rightarrow v)) \leq c(u \rightarrow v) \quad \text{for each edge } u \rightarrow v \\
& \quad f(\text{color}(u \rightarrow v)) \geq 0 \quad \text{for each edge } u \rightarrow v
\end{align*}$$

We can further reduce the number of constraints by replacing the capacity constraints for each edge with a capacity constraint $f(x) \leq c_{\min}(x)$ for each color $x$, where

$$c_{\min}(x) := \min \{ c(u \rightarrow v) \mid \text{color}(u \rightarrow v) = x \}$$

Rubric: 10 points = 2 for objective + 2 for conservation + 2 for capacity + 2 for non-negativity + 2 for color constraints. These are not the only correct solutions.
6. Suppose we need to distribute a message to all the nodes in a given binary tree. Initially, only the root node knows the message. In a single round, each node that knows the message is allowed (but not required) to forward it to at most one of its children. Describe and analyze an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in the tree.

**Solution:** Let $Rounds(v)$ denote the minimum number of rounds need to spread a message from $v$ to all its descendants. We need to compute $Rounds(root)$. This function obeys the following recurrence:

\[
Rounds(v) = \begin{cases} 
0 & \text{if } v \text{ is a leaf} \\
1 + Rounds(w) & \text{if } w \text{ is the only child of } v \\
\min \left\{ \begin{array}{l} 
1 + Rounds(left(v)) \\
2 + Rounds(right(v)) \\
1 + Rounds(right(v)) \\
\end{array} \right. & \text{otherwise} 
\end{cases}
\]

We can memoize this function into the tree itself. Since the value at every node depends only on the values at its children, we can evaluate this function at all nodes by a postorder traversal of the tree. The resulting algorithm runs in $O(n)$ time.

**Rubric:** 10 points: standard dynamic programming rubric.

Sorry, it was either going to be three questions about dynamic programming or three questions with flows. I flipped a coin.
Non-solution: Let $\text{depth}(v)$ denote the depth of the subtree rooted at $v$, that is, the maximum distance from $v$ to a leaf descendant of $v$. We can compute $\text{depth}(v)$ for every vertex $v$ in $O(n)$ time by evaluating the following recurrence with a postorder traversal:

$$
\text{depth}(v) = \begin{cases} 
0 & \text{if } v \text{ is a leaf} \\
1 + \text{depth}(w) & \text{if } w \text{ is the only child of } v \\
1 + \max\{\text{depth(left}(v)), \text{depth(right}(v))\} & \text{otherwise}
\end{cases}
$$

Now let $\text{Rounds}(v)$ denote the minimum number of rounds need to spread a message from $v$ to all its descendants. We need to compute $\text{Rounds(root)}$. Since we obviously want to send the message into the deeper subtree first, this function obeys the following recurrence:

$$
\text{Rounds}(v) = \begin{cases} 
0 & \text{if } v \text{ is a leaf} \\
1 + \text{Rounds}(w) & \text{if } w \text{ is the only child of } v \\
\max\left\{ \begin{array}{l} 
1 + \text{Rounds(left}(v)) \\
2 + \text{Rounds(right}(v))
\end{array} \right\} & \text{if } \text{depth(left}(v)) > \text{depth(right}(v)) \\
\max\left\{ \begin{array}{l} 
2 + \text{Rounds(left}(v)) \\
1 + \text{Rounds(right}(v))
\end{array} \right\} & \text{otherwise}
\end{cases}
$$

We can memoize this function into the tree itself. Since the value at every node depends only on the values at its children, we can evaluate this function at all nodes by a postorder traversal of the tree. The resulting algorithm runs in $O(n)$ time.

Alas, this doesn't work. The number of rounds is always at least the depth of $T$, but it could be significantly larger, depending on the shape of the tree. In particular, distributing a message through a complete binary tree with depth $d$ requires $2d$ rounds. So there's no way to tell just from the depths which subtree to send the message into first. Consider the following example; numbers above each node indicate the minimum number of rounds to distribute a message through that node's subtree.