1. The **linear arrangement** problem asks, given an $n$-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \rightarrow v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

**Solution:** Fix an *arbitrary* ordering $v_1, v_2, \ldots, v_{n-1}, v_n$ of the vertices. If at least half the edges point forward, return the ordering $\langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$; otherwise, return the reverse ordering $\langle v_n, v_{n-1}, \ldots, v_2, v_1 \rangle$.

In both cases, at least half of the directed edges agree with the ordering returned by the algorithm. On the other hand, the maximum number of forward edges is at most the number of edges. So we have a 2-approximation algorithm.

The algorithm runs in $O(E + V)$ time.

---

**Rubric:** This is not the only correct solution, but it is the simplest.
2. Let \( G = (V, E) \) be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in \( G \) is boring if its endpoints have the same color, and interesting if its endpoints have different colors. The **most interesting 3-coloring** is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

(a) Let \( \text{wow}(G) \) denote the number of interesting edges in the most interesting 3-coloring of \( G \). Suppose we independently assign each vertex in \( G \) a random color from the set \{red, green, blue\}. Prove that the expected number of interesting edges is at least \( \frac{1}{3} \text{wow}(G) \).

**Solution:** Color the vertices independently and uniformly at random. The probability that any particular edge \( uv \) is interesting is \( \Pr[\text{color}(u) \neq \text{color}(v)] = \frac{2}{3} \). So by linearity of expectation, the expected number of interesting edges is exactly \( 2|E|/3 \). On the other hand, \( \text{wow}(G) \leq |E| \), because every interesting edge in the optimal coloring is an edge.

(b) Prove that with high probability, the expected number of interesting edges is at least \( \frac{1}{4} \text{wow}(G) \). [Hint: Use Chebyshev’s inequality. But wait… How do we know that we can use Chebyshev’s inequality?]

**Solution:** For each edge \( uv \) in \( G \), define an indicator variable \( X_{uv} = [uv \text{ is interesting}] \). I claim that the random variables \( X_{uv} \) are pairwise independent. Consider any two edges \( uv \) and \( wx \); there are two cases to consider:

- If all four vertices \( u, v, w, x \) are distinct, then fixing the colors of \( u \) and \( v \) has absolutely no effect on whether \( xy \) is interesting.
- On the other hand, suppose (without loss of generality) that \( v = x \). No matter how we fix the colors of \( u \) and \( v \), the probability that \( vw \) is interesting is still \( \frac{1}{3} \).

This is precisely the same analysis as we did for tabulation hashing in Homework 4, problem 2(a). However, the random variables \( X_{uv} \) are not necessarily 3-way independent: if any two edges of a triangle are boring, then so is the third.

Thus, the number of interesting edges is \( X = \sum_{uv} X_{uv} \) is a sum of pairwise independent indicator variables with expectation \( 2|E|/3 \). Applying Chebyshev’s inequality \( \Pr[X \leq (1-\delta)\mu] \leq 1/(\delta^2\mu) \) with \( \delta = 1/4 \) gives us

\[
\Pr[X < |E|/2] < \Pr[X \leq 3E[X]/4] < \frac{16}{E[X]} = \frac{24}{|E|}.
\]

We conclude that \( \Pr[X \geq \text{wow}(G)/2] \leq 1 - \frac{24}{|E|} \). ■

(c) Let \( \text{zzz}(G) \) denote the number of boring edges in the most interesting 3-coloring of a graph \( G \). Prove that it is NP-hard to approximate \( \text{zzz}(G) \) within a factor of \( 10^{10^{10^{10}}} \).

**Solution:** Suppose there is an algorithm \( \text{APPROXZZZ} \) that approximates the minimum number of boring edges in a given graph in polynomial time, for any desired approximation ratio. Consider an arbitrary graph \( G \).

- If \( G \) is 3-colorable, then \( \text{zzz}(G) = 0 \), and therefore \( \text{APPROXZZZ}(G) \) must return 0.
- If \( G \) is not 3-colorable, then \( \text{zzz}(G) > 0 \), and therefore \( \text{APPROXZZZ}(G) \) must return a positive number.

Thus, the following algorithm determines whether an arbitrary graph is 3-colorable in polynomial time:

---

2
\begin{tabular}{|l|}
\hline
\textbf{3Color}(G): \\
\hspace{1em} if \text{APPROXZZZ}(G) = 0 \\
\hspace{1.5em} return \text{True} \\
\hspace{1em} else \\
\hspace{2em} return \text{False} \\
\hline
\end{tabular}

But that's ridiculous; 3-colorability is NP-hard!
3. Suppose we want to schedule a set of \( n \) jobs on a machine containing a row of \( p \) identical processors. Our input consists of two arrays \( \text{duration}[1..n] \) and \( \text{width}[1..n] \). A valid schedule consists of two arrays \( \text{start}[1..n] \) and \( \text{first}[1..n] \) that satisfy the following constraints:

- \( \text{start}[j] \geq 0 \) for all \( j \).
- The \( j \)th job runs on processors \( \text{first}[j] \) through \( \text{first}[j]+\text{width}[j]-1 \), starting at time \( \text{start}[j] \) and ending at time \( \text{start}[j]+\text{duration}[j] \).
- No processor can run more than one job simultaneously.

The makespan of a schedule is the largest finishing time: \( \max_j (\text{start}[j]+\text{duration}[j]) \). Our goal is to compute a valid schedule with the smallest possible makespan.

(a) Prove that this scheduling problem is NP-hard.

**Solution:** We prove that the parallel scheduling problem is NP-hard by a reduction from the NP-hard \textsc{Partition} problem. Let \( X \) be a given set of \( n \) integers, and let \( \Sigma X \) denote the sum of its elements. Define a set of \( n \) jobs, one for each element of \( X \), by setting \( \text{duration}[j] = X[j] \) and \( \text{width}[j] = 1 \) for every index \( j \). This set of jobs can be scheduled on \( p = 2 \) processors with makespan \( \Sigma X/2 \) if and only if the set \( X \) can be partitioned evenly. The reduction requires \( O(n) \) time.

(b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs. That is, if the minimum makespan is \( M \), your algorithm should compute a schedule with makespan at most \( 3M \). You may assume that \( p \) is a power of 2. [Hint: Assume that \( p \) is a power of 2.]

**Solution:** First sort all jobs in non-increasing order by the number of processors they use, so that \( \text{width}[1] \geq \cdots \geq \text{width}[n] \). We schedule the jobs in \( O(\log p) \) phases. In phase \( k \), we divide the \( p \) processors into \( 2^k \) wells, each containing \( p/2^k \) contiguous processors. We then schedule all jobs \( j \) such that \( p/2^k < \text{width}[j] \leq p/2^{k-1} \), in decreasing order of width, by placing each job in the lowest available well. Finally, at the end of the phase, we split each well in two smaller wells for the next phase. The algorithm clearly runs in polynomial time.

Let \( M^* \) denote the optimal makespan, let \( M \) be the makespan of our schedule, and let \( T \) be the first moment in our schedule when at least one processor well is not busy. At all times less than \( T \), at least half the processors are busy, which implies that \( T \leq 2M^* \). On the other hand, because we always schedule the next job in the lowest available well, any job \( j \) that finishes at time \( M \) must have started before \( T \), which implies that \( M - T \leq \text{duration}[j] \leq M^* \). It follows that \( M \leq 3M^* \), as required.
(c) Describe an algorithm that computes a 3-approximation of the minimum makespan of the given set of jobs in \(O(n \log n)\) time. Again, you may assume that \(p\) is a power of 2.

**Solution:** We describe an efficient implementation of the algorithm from part (b). The main idea is to keep the wells in a priority queue. Intuitively, each job requires one \(E\) (to find the best well) and one \(I\) (to change that well’s starting time). Because we have \(n\) jobs to schedule, we only need to keep the best \(n\) wells in the priority queue. So each priority-queue operation should require only \(O(\log n)\) time.

Unfortunately, the previous analysis ignores the time required to split the wells at the end of every phase. Rebuild the priority queue from scratch (from including only the best \(n\) wells) at the end of every phase requires \(O(n)\) per phase, or \(O(n \log p)\) time altogether. To achieve a running time of \(O(n \log n)\), with no dependence on \(p\), we need a way to avoid splitting wells that we never actually use.

So instead of storing wells directly, we actually store contiguous intervals of processors, all with the same available time, in the priority queue. To find the next available well in phase \(i\), we call \(E\) to get the next available interval, take \(p/2^i\) processors from the left end of that interval, and then \(I\) both the new well (which is just a smaller interval) and the rest of the interval (if it is non-empty). During the \(i\)th phase, the size of every interval is guaranteed to be a multiple of \(p/2^i\).

Here is the algorithm in pseudocode. The priority queue stores triples \((t, \ell, w)\) for each cluster of processors, where \(t\) is the next available time (and the priority) for the interval, \(\ell\) is the first (or leftmost) processor in the interval, and \(w\) is the number of processors in the interval (the width).

```plaintext
APPROXIMATE_SCHEDULING(width[1..n], duration[1..n]):
    sort width and permute duration to match
    INSERT(0, 1, p)
    for j ← 1 to n
        w ← [log_2 width[j]]
        well ← 2^w
            \(\langle\text{current well size}\rangle\)
        \((t, [\ell..r])\) ← \text{EXTRACT.MIN}
        Schedule job \(j\) on processors \(\ell .. \ell + width[j] - 1\)
        INSERT\((t + duration[j], \ell, well)\)
        if \(w > \text{well}\)
            INSERT\((t, \ell + \text{well}, w - \text{well})\)
```

The running time of this algorithm is dominated by the time to maintain the priority queue.\(^1\) Initially, the priority queue contains a single interval of size \(p\). Each time we schedule a job, the number of intervals in the priority queue increases by at most one. Thus, there are at most \(n\) intervals in the priority queue at any time, so each priority queue operation requires only \(O(\log n)\) time. Scheduling a job requires at most three priority queue operations, so the overall running time is \(O(n \log n)\), as required.

\(^1\)Actually, we also need \(O(\log p)\) time to maintain the well

---

**Rubric:** This is not the only correct solution.