1. We can use competitive analysis to study policies for caching. The cost of a caching policy is how many times we incur a “miss,” i.e., must service a request for data that is not stored in the cache. The problem is, when we get a request for a new address, how do we decide which data to remove (“evict”) to make room for it? Recall that we wish to bound the ratio of the number of misses we incur to the optimal number of misses in hindsight.

(a) The optimal policy with knowledge of the future always evicts the data that will be accessed furthest in the future.

(b) Any deterministic policy can be asked to provide the data it just evicted, causing it to miss on every request, whereas the optimal policy on such a sequence only misses at most once out of every $k + 1$ requests if the cache has size $k$.

2. The following randomized policy will achieve a $O(\log k)$ competitive ratio: values in the cache can be “marked” or “unmarked” (initially the cache is empty and all values are unmarked), and whenever we access an address in the cache we mark it. If the value was not in the cache, if all values were marked then we unmark all values; either way, we then evict an unmarked address at random to make room for the new request.

(a) We unmark all values after $k + 1$ distinct addresses are requested since the last time everything was unmarked. (Call this a “phase” of the request sequence.)

(b) Let $c_j$ denote the number of “fresh” requests in the $j$th phase, i.e., that were not marked during the previous phase. Since in phases $j$ and $j + 1$ there are at least $k + c_{j+1}$ distinct requests, the optimal policy must have at least $c_{j+1}$ misses. Therefore, summing over all phases, it incurs at least $\frac{1}{2} \sum_j c_j$ misses.

(c) As for the number of misses incurred by the randomized policy, there are $k - c_j$ stale requests that the optimal policy may avoid incurring misses on. Since we evict addresses uniformly at random, the probability that we incur a miss on the $i$th request is at most $\frac{c_j}{k - (i-1)}$; summing over all of these misses gives an upper bound of $H_k c_j$ misses in expectation on this phase, where $H_k$ is the $k$th harmonic number (which is $O(\log k)$). Thus, summing over all phases, we find that we are indeed $O(\log k)$-competitive.

3. A simple greedy algorithm is 2-competitive for scheduling to minimize “makespan”: we have $m$ identical machines and a sequence of requests that take arbitrary lengths of time $t_1, t_2, \ldots$. The makespan is the total time of all of the jobs assigned to a single machine.

(a) A simple algorithm assigns an incoming job to an arbitrary machine with the smallest makespan.

(b) Prior to assigning this job, the machine had at most a $1/m$ fraction of the total time assigned to it. Since some machine must have a makespan at least a $1/m$ fraction of the total time of all jobs in the sequence, this machine had at most the optimal makespan assigned to it.
(c) The job we assign to this machine must be completed by some machine, hence the makespan is at least the time of this job.

(d) Therefore, after we assign the job to a machine, that machine still has at most twice the optimal makespan. Since this holds for all assignments we make, the final allocation is indeed 2-competitive.

(e) Note: this problem is \( \text{NP} \)-complete! So it is useful to run this algorithm even if we are given all of the jobs up-front, i.e., as an “offline” algorithm. It does not solve the problem optimally, but it achieves a makespan that is within a factor of 2 of the optimum. It is our first approximation algorithm.

4. If an algorithm always achieves a solution that is at most \( \alpha(n) \) times greater or less than the optimal solution (for minimization or maximization problems, respectively) for some function \( \alpha \), we say that it is an \( \alpha(n) \)-approximation algorithm.

(a) Most \( \text{NP} \)-complete optimization problems have polynomial time approximation algorithms. The approximation factor \( \alpha(n) \) that we achieve may or may not be adequate for practical needs. Nevertheless, approximation algorithms are, unlike average-case algorithms, often a practical way to approach \( \text{NP} \)-complete problems.