# ALGORITHMS, FALL 2019, HOMEWORK 8

– This assignment is worth 1 unit.
– Due on Thursday, November 21, at noon.
– No credit will be given if you include the problem statement in your submission. All other formatting rules still apply.

1. You are given $k$ plates, arranged in $k$ columns, each containing one plate. A "move" is defined as choosing a column and redistributing all of its plates into other columns, but when doing this each of the columns can receive at most one plate. Also, during the move, you are allowed to make new columns (in fact this is unavoidable if you don't have enough existing columns in which to place all the plates), but again only one plate can go in each new column.
   The *reward* of one move is precisely the number of plates in the column that you choose to redistribute. Suppose that you get to make some huge number of moves (say, $n$, far greater than $k$).

   (a) Develop a strategy to maximize your average reward per move (equivalent to maximizing total reward over $n$ moves). Express this as a function of $k$, using $\Theta$-notation. In other words your maximization doesn't have to be entirely precise; you may assume that $k$ is any convenient number that will make the math easier for your strategy, but you cannot assume that $k = O(1)$. Notice that any strategy that you come up with provides a *lower bound* on reward optimality. The better the strategy, the better (higher) the lower bound. It's trivial to get $\Omega(1)$ per move, so you must get $\omega(1)$.

   (b) Now it's time to figure out how good your strategy actually is. We will get an upper bound on the optimal reward for this game: Use amortization (specifically, the potential method) to obtain an upper bound on the best possible average reward. Ideally this upper bound should match what you get in part (a), using $\Theta$-notation. If it doesn't, then either your algorithm isn't optimal, or your analysis in part (b) isn't tight enough. If it helps to think in terms of cost instead of reward, pretend that your friend is playing this game and it will cost you because you must pay their reward. In this context, you want to find an upper bound on the total cost of your friend's $n$ moves, without having the slightest idea of what their strategy is. In other words you will get an upper bound on what you will pay your friend, before your friend even starts to think about their strategy.
   For amortization, as usual, you want to take advantage of the fact that "expensive" moves do not happen that often. Your $\Phi$ should create a $\Delta\Phi$ that offsets expensive moves, making their amortized cost relatively smaller. So, first think about defining what an expensive move is, and what a non-expensive move is. It will help to have a good strategy in part (a) to determine what the cutoff should be. Then think about something that changes a lot in the configuration of columns whenever you have an expensive move, and use that as your $\Delta\Phi$, then reverse-engineer $\Phi$. Always remember that when you evaluate $\Phi_i$ at any given time, you are taking a snapshot of the current state of the world. $\Phi$ cannot be a function of "what changed" between two iterations. That is the job of $\Delta\Phi$.

   (c) Instead of the potential method, use the accounting method.