Instructions:
Pick any five of the following problems to do and turn in solutions for those. Do not turn in solutions to more than 5 problems (or we will arbitrarily pick 5 and grade only those.)

Some problems are easier than others. Don’t let that bother you :) I do encourage you though to challenge yourself and try your hand at all the problems. Unless specifically forbidden, you are allowed to collaborate with fellow students taking the class in solving problem sets, however you must always write the solutions up on your own. If you do collaborate on solving the problems, you must acknowledge for each problem the people you discussed that problem with (other than the TA and instructor).

The problems have been carefully chosen for their pedagogical value and hence might be similar or identical to those given out in past offerings of this course at UW, or similar courses at other schools. Using any pre-existing solutions from these sources, or any source not explicitly linked to or discussed on our course web page constitutes a violation of the academic integrity expected of you and is strictly prohibited.

When you present an algorithm, you must (briefly) prove that it achieves the guarantees you claim and prove that it runs in polynomial time (unless you are explicitly asked to prove something more specific.)

Concise, clear and precise solutions are appreciated (and expected).

1. Suppose you have a linked list with \( n \) items. Accessing an item at position \( i \) in the list costs \( i \), but immediately after the access, that item can be moved to any of the first \( i \) positions for free. Any pair of adjacent items can also be swapped at a cost of 1. In class we showed that the Move-To-Front algorithm has competitive ratio 2 for this problem.

   Prove that every deterministic algorithm has competitive ratio at least \( 2 - \frac{2}{n+1} \). **Hint:** Fix any deterministic algorithm, and consider a “brutal” request sequence \( \sigma \) in which every single request is to the currently last element in the list. Show that there is an offline algorithm that on a sequence of length \( T \) incurs a cost of at most \( \frac{1}{2} T(n + 1) + C \), where \( C \) is a constant independent of \( T \).

   The simplest way to do this is to use the best static rearrangement of the list (given the request sequence \( \sigma \)).

   (In the previous writeup of this, I proposed the following more complicated way to do it: Consider \( n! \) different “offline” algorithms. Each starts with the \( n \) items in a different permutation (so there may be some paid exchanges at first to get into that permutation), but once there, no rearrangements are ever performed. Then show that one of these \( n! \) algorithms is good.)

2. Same set-up as the previous problem:
   - Show that the online algorithm that after each request moves the requested item 1 position closer to the front (if possible) has competitive ratio \( \Omega(n) \).
• Consider an online algorithm that maintains a counter for each item, so if the items
stored in the list are named \(a_1, \ldots, a_n\), then \(c_i(t)\) is the number of requests there have
been to \(a_i\) up to time \(t\). (Of course \(c_i(0) = 0\) for all \(i\). Whenever an item is accessed,
reorganize the list so that items are kept in order of decreasing value of \(c_i(t)\). (I.e., more
frequently requested items are closer to the front). Ties can be broken arbitrarily. Show
that this algorithm has competitive ratio \(\Omega(n)\). \textbf{Hint:} Consider a request sequence in
which there are first \(2n\) accesses for item \(a_1\), then \(2n - 1\) accesses for item \(a_2\), then \(2n - 2\)
accesses for \(a_3\) and so on.

3. Consider a doubly linked list, with a single external pointer to some position in the list and
suppose that a sequence of Find-Key requests is executed on this list. The cost of executing
this sequence is the number of pointers traversed. The external pointer always points to the
last item visited, and only a pointer from that item can be traversed.

Show that a Find(k) can be performed in time (number of pointers traversed) proportional
to the distance of the item with that key from the location pointed to by the external pointer
immediately before the Find request. What constant factor does your algorithm get?

4. Same setup as previous problem: Consider a cost model in which every pointer traversed
costs 1, and every swap of adjacent elements costs 1, however, either of these can only be
done if the external pointer points to one of the items involved. Assume that on a Find(k),
a little birdie whispers in the ear of the algorithm which direction to go to find it, so that
the time to find it is its distance from the item pointed to by the external pointer. (By the
previous part of this problem, this is within a constant factor of the actual cost.) Suppose the
list starts with \(n\) elements in it, and a sequence of Find requests to items stored in the list is
processed. Show that for any deterministic online algorithm, there is a sequence of requests
on which the algorithm incurs \(\Omega(\log n)\) times the optimal offline cost (where \(n\) is the number
of elements in the list. )

5. Read this description of the bipartite maximum weight matching algorithm presented in class.

Fill in the details (pseudocode and proof of running time) for how to implement the algorithm
on non-dense graphs using a priority queue so as to obtain a running time of \(O(mn \log n)\), and
with linked lists (and an amortized analysis) to get \(O(mn)\). For the priority queue, please
specify your implementation. [This is discussed in the final paragraph before "Additional
comments".]

Here the graph is \(n\) by \(n\), and there are \(m\) edges of value 1. (All non-edges have value 0.)

6. Consider the following generalized matching problem. There are two sets: \(B\), a set of buyers,
and \(I\), a set of items being sold by a single seller, with a bipartite graph connecting them.
Each buyer \(i\) has an associated budget \(B_i\), and a value \(v_{ij}\) for each item \(j\), representing what
he will pay for that item. A generalized matching associates each item \(j\) with one buyer. A
buyer, on the other hand, may be matched to (i.e., he can purchase) multiple items, as long
as the items he is allocated respect his budget. If they go over his budget, he will only pay
up to his budget. Thus, if \(S_i\) is the set of items sold to buyer \(i\), then the profit obtained from
\(i\) is \(\min(\sum_{j \in S_i} v_{ij}, B_i)\). A maximum generalized matching assigns the items to the buyers so
as to maximize the profit of the seller. Give a polynomial time algorithm for this problem that is guaranteed to obtain at least half of the optimal total profit.

7. Consider the following online and simplified version of the previous problem. There are a set of \( n \) distinct items available for sale. \( n \) buyers arrive one at a time. When buyer \( i \) arrives, you learn which items \( S_i \) he is interested in. (Some of them may already be sold.) At that point you can pick one of the items he is interested in that is still unsold and sell it to him. (Obviously, if all the items he is interested in have already been sold, then you cannot sell anything to him.) This is the only chance you have to sell anything to this buyer. The goal is to come up with an online selling algorithm that maximizes the number of items sold.

- Give a 2-competitive deterministic algorithm for this problem. (In a profit problem, like this one, a competitive ratio of 2 means that optimal offline profit is at most twice that of the online algorithm profit.)
- Show that every deterministic algorithm for this problem has competitive ratio at least 2.
- Now consider the randomized algorithm that, upon the arrival of a buyer, sells him a uniformly random unsold item among \( S_i \) (so if \( k \) of the items in \( S_i \) are still unsold, a random one of these \( k \) is selected and sold to him). Show that the competitive ratio of this algorithm is \( 2 - o(1) \). (Here the \( o(1) \) refers to the fact that \( n \) is tending to infinity.)

8. You might want to first read through Section 2.2 of [http://courses.cs.washington.edu/courses/cse521/15sp/refs/thorup1.pdf](http://courses.cs.washington.edu/courses/cse521/15sp/refs/thorup1.pdf). Suppose you are implementing a dictionary via hashing with chaining. For simplicity, assume that your hash function is completely random. However, you do not know how many items will be stored in your dictionary, so you don’t know what size hash table to use. Describe an adaptive scheme that at all times has a hash table of size proportional to the current number of items in the dictionary, and has the property that the total expected cost of doing \( T \) dictionary operations is \( O(T) \). Thus, for example, if \( T \) times you insert and delete the same element, then your table should never have more than constant size. On the other hand, if you insert \( T \) different elements one after the other, then after the first \( k \) of those operations, your table should have size \( O(k) \) and the expected time to perform the first \( k \) operations should be \( O(k) \).

9. In Section 2.2 of [http://courses.cs.washington.edu/courses/cse521/15sp/refs/thorup1.pdf](http://courses.cs.washington.edu/courses/cse521/15sp/refs/thorup1.pdf) it is proved that the set of hash functions

\[
\mathcal{H} = \{ (ax + b) \bmod{p} \bmod{m} \mid a \in \{1, \ldots, p-1\}, b \in \{0, \ldots, p-1\} \}
\]

is universal (i.e. \( \Pr(h(x) = h(y)) \leq 1/m \), for \( x \neq y \)). This class has \( p(p-1) \) functions in it. Suppose instead that we also allow \( a = 0 \), that is, we consider the class

\[
\mathcal{H}' = \{ (ax + b) \bmod{p} \bmod{m} \mid a \in \{0, \ldots, p-1\}, b \in \{0, \ldots, p-1\} \}.
\]

Show that \( \mathcal{H}' \) may not be universal, but that for any \( x \neq y \in [p] \),

\[
\Pr(h(x) = h(y)) < \frac{2}{m}.
\]

As usual, this probability is taken over the uniformly random choice of \( a \) and \( b \).