CSE 101 Homework 3
Speeding up algorithms with restructuring, preprocessing and data structures.
Due Tuesday, April 23
80 points total %

Solve each problem. For algorithm problems, if the problem only specifies
that you need to give a proof of correctness, then no time analysis is required.
If it specifies that you need to give an efficient implementation, then you do
not need to give a correctness proof for the basic strategy (just explain why
your version actually carries out the strategy). If it says to do both, or doesn’t
specify what parts you need, you need to give both a proof of correctness and
time analysis.

Short answers (10 points each: explain your answer)
1. If Alg1 takes
time $\Theta(n^2)$ and produces an output of length $n \log n$, and Alg2 takes
time $\Theta(n^2)$, what is the time complexity of an algorithm that runs
Alg1 and then runs Alg2 on the output of Alg1? Explain your answer.
Alg 1 will run for $O(n^2)$ time, and then produce an output of size
$n' = O(n \log n)$. Alg2 will then run for $O((n')^2) = O(n^2 (\log n)^2)$
time. Thus, the total time will be $O(n^2 + n^2 (\log n)^2) = O(n^2 (\log n)^2)$.

2. If on a graph with $n$ nodes and $m$ edges, Alg1 takes time $O(n m)$ and
Alg2 takes time $O(n^2 \log n)$, when is it better to use Alg1? Explain.
Alg1 takes time at most $c n m$ for some constant $c$, and Alg2 at most
$c_2 n^2 \log n$ time. Thus, Alg1 is faster when $m < c_2 / c_1 n \log n$. If $m$ is
$o(n \log n)$, then Alg1 is asymptotically faster, if $m = \omega(n \log n)$ then
Alg2 is faster, and if $m = \Theta(n \log n)$, which is faster depends on the
hidden constants, and needs to be determined empirically.

Merging $k$ sorted lists We wish to merge $k$ sorted lists, $L_1, \ldots, L_k$. We use
the same basic strategy as heap sort: find the smallest element in the $k$
lists, put it in the first position on the output array, delete it from the set
of elements to be sorted, and repeat. But we observe that the smallest
element will be the head of one of the lists. So we will only have $k < n$
possibilities to pick from at any time.

More precisely, the strategy is

1. Let \textit{Candidates} be the heads of each list (together with the name of
   the list it is from).
2. FOR $l = 1$ to $n$ do:
3. \quad Let $(V, J)$ be the element of \textit{Candidates} with smallest value $V$.
5. \quad Delete $(V, J)$ from \textit{Candidates}
6. \quad Delete the head of list $J$
7. IF List J is not empty, insert a pair (V', J) into Candidates, where V' is the new head of list J.

8. Return A.

We prove by induction the invariant that, after looping t times, A[1..t] contain the t smallest elements of the lists in sorted order, the lists contain all remaining elements, and Candidates contains the head of each non-empty list, together with the list name. This invariant is true at the start, since there are no output elements, the lists are entire, and Candidates is initialized to the heads of each list, together with the list name. Assume the invariant is true after t loops. The t + 1st smallest element V is in some list J.

Since by the invariant, the t smaller elements have been deleted from the lists, it must be the smallest remaining element in list J. Since the list is sorted, it must be at the head of list J. Therefore, by the invariant, the pair (V, J) is in candidates. Since all elements of Candidates are still in the lists, they must be at least as large as V. Therefore, the smallest element of Candidates is V, and we’ll make that A[t + 1] so A[1..t + 1] will contain the first t + 1 elements in sorted order. We will then delete V from some list J' where it is the head, and from candidates. (If there are ties, we might have J' ≠ J.) We delete V from list j', so we have only the remaining elements in the lists. Then we replace the pair (V', J') for the current value of the head of list J', so Candidates contains exactly the head of each non-empty list. Thus, after t + 1 iterations, the invariant still holds.

Thus, by induction it holds after any number t of iterations.

In particular, for t = n, the array A[1..n] contains all n elements of the lists in sorted order.

To get an efficient implementation of this strategy, we use a Minheap of pairs (V, j) keyed by value V to store Candidates. In terms of the Heap operations Insert, DeleteMin, FindMin, the algorithm becomes:

1. FOR J = 1 to k do:
2.   V ← HeadoflistJ
3.   Insert((V, J)).
4. FOR I = 1 to n do:
5.   Let (V, J) ← FindMin
7.   DeleteMin.
8. Delete the head of list J
9. IF List $J$ is not empty, $Insert(list(J).head, J)$.
10. Return $A$.

Note that the heap Candidates starts with $k$ elements, and each iteration of the loop, we insert at most once and delete exactly one element. Therefore, the heap stays at size at most $k$. So the operations $DeleteMin$ and $Insert$ take time $O(\log k)$. Thus, the total time is $O(n \log k)$.

**Implementation-20 points**  Implement a naive $O(n^2)$ time sorting algorithm (such as bubble sort) and heap-sort. You can use heaps from a standard library to implement heap-sort. Plot their performance on random arrays of $n$ integers with values between 1 and $n$, for $n = 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$. Plot their performance on a log-log scale. Is heap-sort always better than bubble-sort? Why or why not?

This is in a separate file.