Problem Set 7 Solutions

(Exercises were not to be turned in, but we’re providing the solutions for your own interest.)

Exercise 7-1. When $n$ is a power of 3, we divide each polynomial into three parts, grouping coefficients for those terms having degrees 0, 1, and 2 mod 3. Formally, $A(x) = A_0(x^3) + x A_1(x^3) + x^2 A_2(x^3)$, where $A_i$ has the coefficients of $A$ for only those terms have degrees that are $i$ mod 3. The recurrence for the new algorithm is $T(n) = 3T(n/3) + \Theta(n)$, which by the Master Theorem solves to $T(n) = \Theta(n \log n)$.

Exercise 7-2. The total running time for the $i$th operations, where $i$ is a power of 2, is $1 + 2 + \cdots + 2^{\lceil \log n \rceil} = 2^{\lceil \log n \rceil + 1} - 1 = \Theta(n)$. The total running time of the other operations is $n - \lceil \log n \rceil$. Therefore the amortized cost per operation is $\Theta(1)$.

Exercise 7-3. The potential function is (a constant multiple $c$ of) the sum of the depths of all the nodes in the heap. We sketch why this works: for \textsc{insert}, the actual amount of work done is $\Theta(\log n)$, and the potential function increases by $\Theta(\log n)$ because a new element is added to the tree. For \textsc{delete-min}, the actual work done is again $\Theta(\log n)$ plus $O(1)$. However, the potential decreases by $c \log n$ because an element is removed. If we choose $c$ to match the constant hidden in the $\Theta(\log n)$, then the decrease in potential cancels out the real work that is done, leaving $\Theta(1)$ amortized cost.

Note that this result is just the result of “clever accounting,” and not anything earth-shattering. In any application of a min-heap, the number of \textsc{insert} operations must be at least the number of \textsc{delete-min} operations, so the running time is dominated by the insertions.

Exercise 7-4. To compute the transpose for an adjacency-list representation, we make a new array of adjacency lists for $G^T$. We walk down each adjacency list of $G$. On the list for node $u$, when encountering a node $v$, we add $u$ to the front of $v$’s list in $G^T$. Each step takes $O(1)$ time, so the total time is $O(V + E)$.

For an adjacency-matrix representation, we merely need to compute the transpose matrix. This can be done in $O(V^2)$ time.

Exercise 7-5. (Trivia: this problem is otherwise known as “testing whether a given graph is bipartite.”) The wrestlers correspond to nodes in a graph, and their rivalries correspond to edges. Pick an arbitrary vertex $s$ and run a breadth-first search from $s$ to produce a vector $d$ of shortest path lengths from $s$. (If the graph is unconnected, run BFS on each of its components.) Then iterate over the edges: if $(u, v)$ is an edge and $d[u]$ and $d[v]$ have the same parity (i.e., both even or both odd), then output “no designation.” If every edge passes this test, output all $u$ such that $d[u]$ is even as the good guys, and all $v$ such that $d[v]$ is odd as the bad guys.

First, note that if all the edge tests are passed, then the designation is a proper one, because every rivalry is between a good and bad guy. Now suppose some test is not passed for an edge $(u, v)$: in
any designation, \( u \) and \( v \) must be of the same type because they are the same number of “hops” from \( s \). But this means the rivalry between \( u \) and \( v \) is not satisfied. Thus, there is no valid designation.

The running time is clear: BFS takes linear time \( O(n + r) \), and iterating over the edges takes \( O(r) \) time, for \( O(n + r) \) total.

**Exercise 7-6.** The graph is on four vertices \( s, t, u, v \), where \( w(s, u) = 4 \), \( w(s, t) = 2 \), \( w(u, t) = -2 \), and \( w(t, v) = 1 \). Starting from \( s \), we set \( d[t] = 2 \) and \( d[u] = 4 \). Therefore \( t \) is extracted, so we set \( d[v] = d[t] + 1 = 3 \). Next \( v \) is extracted, and no changes are made to \( d \). Finally \( u \) is extracted, and we set \( d[t] = d[u] + -2 = 2 \), then the algorithm terminates. Note that the shortest path to \( v \) is \( s, u, t, v \), and has length 3. However, at the end of the algorithm, \( d[v] = 4 \) (corresponding to the path \( s, t, v \)).

The proof of Theorem 24.6 fails where (on page 598, end of second paragraph) it claims that \( \delta(s, y) \leq \delta(s, u) \) “because \( y \) occurs before \( u \) on a shortest path from \( s \) to \( u \) and all edge weights are nonnegative.” In fact, we see in the above example that this is not the case: the shortest path from \( s \) to \( t \) is \( s, u, t \) and has length 2, but the shortest path from \( s \) to \( u \) has length 4. Therefore the proof of correctness is no longer sound.

**Problem 7-1. Maximum Spanning Tree**

We note that this problem is very similar to the minimum spanning tree problem. One correct solution involves a direct transformation, by negating all the edge weights of \( G \) and running Prim’s (or Kruskal’s) algorithm on the resulting graph \( G’ \). (These algorithms work properly even with negative edge weights.) A minimum spanning tree on \( G’ \) is a maximum spanning tree on \( G \), because a tree in \( G’ \) is a tree in \( G \) and vice versa, and because the weight of a tree in \( G’ \) is negated in \( G \).

Another way to solve this problem is by noticing a greedy-choice property, similar to that of the minimum spanning tree (and proven in a very similar way): in any maximum spanning tree \( T \), if we remove an edge \( (u, v) \) to yield two trees \( R, S \), then \( R \) and \( S \) are maximum spanning trees on their respective vertices, and \( (u, v) \) is a heaviest edge crossing between those sets of vertices. With this in mind, we can use Prim’s algorithm with a max-heap, or Kruskal’s algorithm with the edges sorted in descending order of weights, to find a maximum spanning tree. The running times remain unchanged.

**Problem 7-2. Toeplitz Matrices**

(a) The sum is Toeplitz. If we are adding matrices \( A \) and \( B \) (with entries \( a_{i,j} \) and \( b_{k,j} \), respectively), then the sum \( C \) (with entries \( c_{i,j} \)) has

\[
c_{i,j} = a_{i,j} + b_{k,j} = a_{i-1,j-1} + b_{i-1,j-1} = c_{i-1,j-1}
\]

as desired.
The product is not necessarily Toeplitz. Here is a counterexample:

\[
\begin{pmatrix}
1 & 2 \\
0 & 1
\end{pmatrix} \begin{pmatrix}
1 & 0 \\
2 & 1
\end{pmatrix} = \begin{pmatrix}
5 & 2 \\
2 & 1
\end{pmatrix}
\]

(b) Note that there are only $2n - 1$ diagonals in an $n \times n$ matrix, and the values on a diagonal are all the same. Therefore we need only a $(2n - 1)$-coordinate vector to represent an $n \times n$ Toeplitz matrix. Specifically, the vector is a tuple of the elements $a_{1,n}, a_{1,n-1}, \ldots, a_{1,1}, a_{2,1}, \ldots, a_{n,1}$. Adding two matrices is done by adding their representative vectors, entry-by-entry. This takes only $O(n)$ time (and space).

(c) Let the input vector be a column vector $\vec{b} = (b_1, \ldots, b_n)^T$, and call the product $c^r = (c_1, \ldots, c_n)^T$. Suppose also that we are representing the Toeplitz matrix $A$ by the vector $\vec{a}$ described above. Then by the definition of Toeplitz and matrix multiplication, we have

\[
c_i = \sum_{j=1}^{n} a_{n+i-j}b_j = \sum_{j=1}^{2n-1} a_{n+i-j}b_j,
\]

where we adopt the convention that $b_j = 0$ when $j > n$, and $a_j = 0$ when $j \leq 0$. But now we see that the coefficient $c_i$ is just the coefficient of the degree-$(n + i)$ term of the product of polynomials $a$ and $\vec{b}$, whose representations are given in coefficient form by the vectors $\vec{a}, \vec{b}$. These polynomials have degree $O(n)$, so we can multiply them in $O(n \log n)$ time, as desired.

**Problem 7.3. Amortized Queues**

(a) The total work is $3 + (6 + 2) + 3 + (1 + 6 + 1) = 22$. At the end, $S_1$ has 0 elements, and $S_2$ has 2.

(b) An insertion always takes 1 unit, so our worst-case cost must be caused by a removal. No more that $n$ elements can ever be in $S_1$, and no fewer than 0 elements can be in $S_2$. Therefore the worst-case cost is $2n + 1$: $2n$ units to dump, and one extra to pop from $S_2$. This bound is tight, as seen by the following sequence: perform $n$ insertions, then $n$ removals. The first removal will cause a dump of $n$ elements plus a pop, for $2n + 1$ work.

(c) The tightest amortized upper bounds are 3 units per insertion, and 1 unit per removal. We will prove this 2 ways (using the accounting and potential methods; the aggregate method seems too weak to employ elegantly in this case). (We would also accept valid proofs of 4 units per insertion and 0 per removal, although this answer is looser than the one we give here.)

Here is an analysis using the accounting method: with every insertion we pay $3$: $1$ is used to push onto $S_1$, and the remaining $2$ remain attached to the element just inserted. Therefore every element in $S_1$ has $2$ attached to it. With every removal we pay $1$, which will (eventually) be used to pop the desired element off of $S_2$. Before
that, however, we may need to dump $S_1$ into $S_2$; this involves popping each element off of $S_1$ and pushing it onto $S_2$. We can pay for these pairs of operations with the $2$ attached to each element in $S_1$.

Now we analyze the structure using the potential method: let $|S_i|$ denote the number of elements in $S_1$ after the $i$th operation. Then the potential function $\phi$ on our structure $Q_i$ (the state of the queue after the $i$th operation) is defined to be $\phi(Q_i) = 2|S_i|$. Note that $|S_i| \geq 0$ at all times, so $\phi(Q_i) \geq 0$. Also, $|S_0| = 0$ initially, so $\phi(Q_0) = 0$ as desired.

Now we compute the amortized costs: for an insertion, we have $S_{i+1}^i = S_i^i + 1$, and the actual cost $c_i = 1$, so

$$\hat{c}_i = c_i + \phi(Q_{i+1}) - \phi(Q_i) = 1 + 2(S_i^i + 1) - 2(S_i^i) = 3.$$

For a removal, we have two cases. First, when there is no dump from $S_1$ to $S_2$, the actual cost is $1$, and $S_{i+1}^i = S_i^i$. Therefore $\hat{c}_i = 1$. When there is a dump, the actual cost is $2|S_i| + 1$, and we have $S_{i+1}^i = 0$. Therefore we get

$$\hat{c}_i = (2|S_i| + 1) + 0 - 2|S_i| = 1$$

as desired.

**Problem 7-4. Shortest-Path Special Cases**

(a) We make the following observation about Dijkstra’s algorithm in this case: if $i$ is the value returned by the most recent DELETE-MIN, then the priority queue only contains keys $i, i + 1, \ldots, i + C, \infty$. This is because each element in the queue has key at least $i$, and is either not a neighbor of any vertex that has been removed from the queue (in which case its key is still $\infty$), or it is a neighbor of a vertex that has been removed. Such a neighbor is within $i$ of the source vertex, so the vertex in question would have key at most $i + C$. Therefore by keeping an array as our priority queue (with $CV = O(V)$ entries), we can implement DELETE-MIN in $O(1)$ time by straightforward search in the array, for a new total running time of $O(V + E)$.

We can also make a direct transformation to a BFS problem, in the following way: split each edge with weight $w > 0$ into $w$ edges (by adding $w - 1$ nodes in between). Contract (i.e., merge) vertices connected by edges of weight 0. This transformation increases the size of the graph by a factor of at most $C$ (a constant), so the number of nodes in the new graph is still $O(V)$, and the number of edges $O(E)$. Therefore we can run a breadth-first search in time $O(V + E)$.

(b) (Note the correction to the original problem set: the desired time is $O((V + E)\log \log u)$.) Note that the priorities in the queue are the lengths of paths, so they may be up to length $uV$. Use a van Emde Boas queue, with universe $\{0 \ldots uV\}$, in Dijkstra’s algorithm. Because $u > V$, the running time of a vEB operation is $O(\log \log uV) = O(\log \log u)^2 = O(\log \log u)$. Instead of decreasing keys (which we don’t know how to
do for vEB queues), we simply remove the old key and insert the new one. This is done at most $|E|$ times, so by modifying the analysis of the algorithm, we get a $O((V + E) \lg \lg u)$ running time.

(c) Store a bit vector of length $u$, initially all zeros. To insert an element with key $x$, set bit $x$ to 1 (and update any pointers to auxiliary data). Maintain an index to which key the last DELETE-MIN returned. The DELETE-MIN procedure works as follows: starting from the current index, find the smallest key that exists in the queue (i.e., the index of the first non-zero bit) and return its element. Update the index accordingly. The total time over a sequence of $k$ operations is $O(u)$ to make at most one full pass over the bit vector, plus $O(k)$ to do the deletions, for $O(u + k)$ as desired.

(d) We can use the monotone priority queue exactly as described above in Dijkstra’s algorithm. We perform $O(|V|)$ DELETE-MIN operations, so the running time becomes $O(|V| + |E| + u)$. 