Quiz 2 Solutions

Problem 1. Trick or Treat (1 Part, 10 Points).
You are given an $m \times n$ Hershey Bar which you are to crack into $mn$ pieces, each sized $1 \times 1$. An elementary move in this model takes a single $j \times k$ piece and cracks it along a single vertical or horizontal edge. How many moves does it take to reduce the bar to a collection of $1 \times 1$ pieces? (By e-mail, the following clarification was sent: We are asking for the tightest possible upper and lower bounds on the number of elementary moves required.) Give a brief argument for your answer.

Solution.
Upper Bound:
Each elementary move increases the total number of pieces by one (after breaking the original bar, you have two pieces; after breaking one of those, you have three; etc.). Thus the total number of moves required to increase the number of pieces from 1 to $mn$ is at most $mn - 1$.

Lower Bound:
Each elementary move increases the total number of pieces by one. Thus the total number of moves required to increase the number of pieces from 1 to $mn$ is at least $mn - 1$.

Tight Bounds:
The total number of moves required is exactly $mn - 1$.

Students were generally successful on this problem. Most students recognized that any way you broke up the Hershey bar, $mn - 1$ moves were required. However, some students got confused by the difference between a lower bound and an upper bound.

A common mistake was to set up an inductive proof, where the base case is a $1 \times 1$ hershey bar, and then in the inductive step, argue that when one row or column is added, the solution can break off this row, solve the rest by induction, then solve the single column/row by brute force. This was valid for an upper bound, since it gives a recursive construction with $mn - 1$ moves. However, it is not a valid lower bound, since there is no way to know that the best solution was the one that broke off this single column/row at this point. One must argue that for all possible breaks (not just the one right at the row or column), the inductive hypothesis holds (which can be made to work, and was probably the most common correct solution).

Another common approach was to describe a dynamic programming algorithm that builds a table where entry $(i,j)$ holds the minimum number of moves required for breaking an $i \times j$ bar into $1 \times 1$ pieces. Some people made this work as a lower bound by deriving a general formula for entry $(i,j)$ in the table. This was given full credit. Some describe the algorithm and claim the $mn - 1$ by simply inspecting the table. This was given partial credit.
Problem 2. A Mysterious Graph Algorithm (3 Parts, 30 Points).
Consider the following algorithm: Given a connected, undirected graph $G$ with distinct edge weights, repeatedly remove the heaviest edge whose removal will not disconnect the graph, until no such edges remain.

A. What does this algorithm produce at termination? Prove the algorithm correct.
We denote the number of vertices and edges in $G$ as $V$ and $E$, respectively.

The algorithm produces a Minimum Spanning Tree (MST) of $G$. To prove this, we want to show that repeatedly removing the heaviest edge that does not disconnect the graph results in an MST. To do this, we show that as we delete edges, we maintain the invariant that the graph still contains an MST of $G$. Then we show that when there is no such edge (i.e., when removing any edge would disconnect the graph), what we are left with is a tree. Proving these two things implies that we are left with an MST.

Claim: Consider a connected graph $G$, and let $e$ be the heaviest edge in $G$ whose removal does not disconnect the graph. Let $G'$ be the graph $G$ with $e$ removed. Then, an MST of $G'$ is also an MST of $G$.

Proof: by contradiction. Assume that an MST $T'$ of $G'$ is not an MST of $G$. Certainly $T'$ spans $G$, since it spans $G'$, and $G$ and $G'$ have the same vertex set. So, $T'$ must have greater cost than an MST $T$ of $G$. To show a contradiction, we will convert $T$ into a spanning tree of $G'$ with cost at most the cost of $T$, thus showing $T'$ is not an MST of $G'$.

If the edge $e$ is not in $T$, then $T$ is already a spanning tree of $G'$, a contradiction. So, $T$ must contain $e$. Consider the two connected groups of vertices $S$ and $\overline{S}$ induced by removing $e$ from $T$. Now consider the cut $(S, \overline{S})$ in $G$. The edge $e$ crosses this cut, and there must also be another edge $e'$ crossing that cut, since removing $e$ from $G$ does not disconnect $G$. Furthermore, $e'$ must have weight at most the weight of $e$, since $e$ is the heaviest edge in $G$ that does not disconnect it. If we remove $e$ from $T$ and add $e'$ we have a spanning tree of $G'$ with cost less than or equal to the cost of $T$. We have created our contradiction, and the claim is proven. □

Repeatedly applying this claim as we remove edges maintains the invariant that an MST of the current graph is also an MST of the original graph $G$. Now, we have to prove that when there are no more edges to delete, we are left with a tree. This would show that we have a tree that contains an MST of $G$, and clearly this MST must be the tree itself.

Claim: Consider a connected graph $G$. If, for every edge $e$ in $G$, removing $e$ would disconnect $G$, then $G$ must be a tree.

Proof: by contradiction. Assume that $G$ is not a tree. Therefore $G$ contains a cycle $C$. Let $e$ be some edge on the cycle $C$. By assumption, removing $e$ would disconnect the graph, so let $u$ and $v$ be two vertices that become disconnected when $e$ is removed. Now consider a path $P$ from $u$ to $v$ in $G$ (which must be there since $G$ is connected before $e$ is removed). This path $P$ must go through $e$, since $e$ disconnects $u$ and $v$ when it is removed. Since $P$ goes through $e$, it must enter and leave the cycle $C$ at some point. (Perhaps $P$ enters and
leaves $C$ right at $e$, or maybe $u$ and/or $v$ are actually on $C$. These cases are still captured by the argument.) If we reroute $P$ the other direction around the cycle $C$, we have a path from $u$ to $v$ that does not go through $e$. Therefore $u$ and $v$ do not become disconnected when $e$ is removed, and we have a contradiction. □

**Part A** was worth 15 points. Most people recognized the algorithm as producing an MST of $G$. There were a wide variety of proofs for this claim. There were some very elegant solutions that were quite different than the one given here. Many people made too many assumptions that resulted in non-rigorous proofs. A common mistake was to assume that at each step, removing the heaviest edge that does not disconnect the graph immediately implies that what you’re left with in the end is a globally optimal solution. This had to be proven more carefully using an inductive argument with a strong hypothesis like the one given here.

**B.** Give pseudo-code for an efficient implementation of the algorithm. Assume you are given the subroutine \text{REACHABLE}(G, a, b), which returns \text{TRUE} if and only if there is a path from vertex $a$ to vertex $b$ in $G$. We denote the running time of \text{REACHABLE} by $R(E, V)$. (Feel free to write your own version of \text{REACHABLE} if you wish.)

\text{Mysterious}(E, V)
1. Sort edges in $E$ by decreasing weight (Denote the sorted edges by $e_1, e_2, \ldots e_{|E|}$.)
2. for $i \leftarrow 1$ to $|E|$ 
   \hspace{1cm} if \text{REACHABLE}(G - e_i, e_{i1}, e_{i2}) is \text{TRUE} then let $G \leftarrow G - e_i$
   \hspace{1cm} $i \leftarrow i + 1$
3. return $G$

Note that the $G$ returned in line 3 is an MST. The subroutine \text{REACHABLE}(G, a, b) can be implemented as follows.

\text{REACHABLE}(G, a, b)
1. for each vertex $u \in V$
   \hspace{1cm} \text{color}[u] \leftarrow \text{WHITE}
   \hspace{1cm} \pi[u] \leftarrow \text{NIL}
2. $time \leftarrow 0$
3. \text{DFS-Visit}(a)
4. if $b$ is \text{BLACK} 
   \hspace{1cm} return \text{TRUE}
   \hspace{1cm} else return \text{FALSE}

The code for \text{DFS-Visit}(u) is given in CLR (page 478).

**Part B** was worth 10 points. One common error was to terminate the algorithm immediately after encountering an edge whose removal disconnected the graph. Another common error
was to call the \texttt{REACHABLE} subroutine repeatedly on many or all pairs of vertices after removing an edge, instead of just the endpoints of the removed edge.

\textbf{C.} What is your algorithm’s running time in terms of $V$, $E$, and $R(E, V)$?

Step 1 of \texttt{MYSSTEROUS} takes $O(E \lg E)$ time. Step 2 executes the subroutine \texttt{REACHABLE}(\textit{G} \setminus e_i, i_1, i_2) exactly $E$ times, once for each edge. Note that given an adjacency list representation of a graph $G$, we can run \texttt{DFS-Visit}(a) on the graph $G - e_i$ without actually deleting edge $e_i$ from $G$, simply by modifying the code for \texttt{DFS-Visit}(a) so that it ignores edge $e_i$. Thus, we can upper bound the running time by $O(E \lg E + E \cdot R(E, V))$. The implementation of \texttt{REACHABLE}(\textit{G}, i, j) given above runs in $O(E)$ time. The running time of \texttt{MYSSTEROUS} with this implementation of \texttt{REACHABLE} is therefore $O(E \lg E + E^2) = O(E^2)$.

\textbf{Part C} was worth 5 points. Full credit was generally given for this part if the pseudo-code from Part B – whatever its efficiency – was analyzed correctly.

\textbf{For your interest.} It turns out that you can do better than time $O(|E|)$ for the \texttt{REACHABLE} function, thus speeding up the algorithm. To do this, you need a \textit{dynamic connectivity} data structure. This is a data structure that maintains an underlying graph, and supports the following operations: Insert($e$), Delete($e$) and Reachable($u, v$). Insert($e$) adds edge $e$ to the graph, Delete($e$) removes $e$ from the graph, and Reachable($u, v$) returns yes if $u$ and $v$ are connected in the current graph, and no if they are not. Our data structure in the solution above was simply the graph itself; we performed Inserts and Deletes in constant time (by just adding or removing the edge), and we performed \texttt{REACHABLE} in time $O(|E|)$ using DFS.

There are advanced data structures that support all operations in sub-linear amortized time per operation (and would therefore speed up the algorithm, since we perform at most one of each kind of operation per loop iteration), but they are beyond the scope of this class. David Karger’s Advanced Algorithms course (6.854) has covered some of these data structures in years past. For your interest, the first such data structure was created in 1985 by G. N. Frederickson, and it supports Inserts and Deletes in time $O(\sqrt{|E|})$, and \texttt{REACHABLE} in constant time. This has been repeatedly improved by many researchers, and in fact research is still active in this area. The latest result (Thorup, 2000) supports Insert and Delete in time $O(\frac{\log |E|}{\log \log \log |E|})$, and supports \texttt{REACHABLE} in time $O(\log |E| (\log \log |E|)^3)$.
Problem 3. A Queue from Two Stacks (3 Parts, 30 Points).
Suppose we wish to realize a first-in-first-out queue, using only two stacks and the (constant-time) primitive stack operations Push and Pop.
Consider this implementation of Enqueue and Dequeue (ignoring error-handling):

Stack P, Q; \triangleright Initialy empty.

Enqueue( Key k )

Push( k, P );

Dequeue( )

if ( Q is Empty )

while ( P is not Empty )

Push( Pop( P ), Q );

return Pop( Q );

To understand this data structure, it might help you to make a table with four columns labeled “Operation,” “State of P”, “State of Q”, and “Result of D”, and fill it in for an interesting sequence of operations (say, E1, E2, E3, D, D, E4, D, E5, D, D, where Ex means “Enqueue(x)” and D means “Dequeue”). Convince yourself that keys are dequeued in exactly the same order that they were enqueued.

Now consider an initially empty queue, and any sequence of n operations, each of which is either Enqueue or Dequeue.

A. What is the worst-case (non-amortized) running time of Enqueue? Of Dequeue?
B. Use the Accounting Method to bound the amortized cost per operation.
C. Use the Potential Method to bound the amortized cost per operation.

Solution.
We will count each Pop and Push as a unit operation. Let s(P) denote the number of elements in P at a given time.

A. Enqueue always does one Push and therefore its worst-case running time is 1.

Dequeue does one Pop if Q is non-empty, and s(P) + 1 Pops and s(P) Pushes if Q is empty. Thus, the worst-case running time of Dequeue is 2 \cdot s(P) + 1.

Note that s(P) grows by one with each Enqueue, and does not grow with each Dequeue. Thus, if we are considering n operations, at least one of which is Dequeue, then s(P) \leq n - 1, so the worst-case running time of Dequeue in this case is at most 2(n - 1) + 1 = 2n - 1. This is indeed achieved if the first n - 1 operations are Enqueue and the last one is Dequeue.
Most students got this part right. One point was taken off for the right answer without explanation, if the rest of the write-up showed some basic understanding of this construction.

B. Note that each element in the queue is:

- Pushed once onto \( P \), only when Enqueue is called for that element;
- Moved at most once from \( P \) to \( Q \), at the cost of one Pop and one Push;
- Popped at most once from \( Q \), only when Dequeue is called for that element.

Thus, we shall charge the amortized cost of 3 for each Enqueue and 1 for each Dequeue. It is then easy to see that our balance will always be \( 2 \cdot s(P) \), since it costs us 1 to Push each element onto \( P \), so we have 2 leftover for each element in \( P \), which we do not spend until that element is moved onto \( Q \). Thus, our balance is always positive, so our analysis works.

Most students got this part right. Many found the nice analogy of viewing the stack as a stack of plates, and leaving money on each plate (thus, each plate in \( P \) has \$2 on top of it). Another common solution charged \$4 for Enqueue and \$0 for Dequeue, in which case every plate in \( P \) had \$3 on it, and every plate in \( Q \) had \$1 on it.

A common mistake was to try to charge \$2 for Push and \$0 for Pop, which doesn’t give you a constant amortized cost, because Dequeue can do a non-constant number of Pushes.

C. Let \( D_i \) be the state of our data structure after some \( i \) operations, and let the potential \( \Phi(D_i) = 2 \cdot s(P) \). To find the amortized cost of an operation, we will compute the actual cost of the operation plus the difference in potential \( \Delta \Phi \) that the operation produces.

We have three cases to consider:

1. For Enqueue, \( c = 1 \) (for 1 Push) and the potential difference \( \Delta \Phi = 2 \) (because \( s(P) \) changes by 1), so the amortized cost \( \hat{c} = 3 \).
2. For Dequeue, if \( Q \) is non-empty, then \( c = 1 \) (for 1 Pop) and the potential difference \( \Delta \Phi = 0 \) (because \( s(P) \) doesn’t change), so \( \hat{c} = 1 \).
3. For Dequeue, if \( Q \) is empty, then \( c = 2 \cdot s(P) + 1 \) (for \( s(P) \) Pushes and Pops in the loop and one final Pop) and the potential difference \( \Delta \Phi = -2 \cdot s(P) \) (because the new potential is 0, because \( P \) is now empty), so \( \hat{c} = 1 \).

This was the hardest part of the problem. Many couldn’t find the right potential function, even though they did the accounting method right (and, of course, the accounting method’s bank balance would work as the potential function). Another good potential function was \( 3 \cdot s(P) + s(Q) \) (this corresponds to paying \$4 for Enqueue and \$0 for Dequeue in the accounting method). A common bad potential function was \( s(P) - s(Q) \), which doesn’t work, because it becomes negative.

If you had the right \( \Phi \), but forgot to check all three cases, you got 7-8 points. If you couldn’t find the right \( \Phi \), you got 0-3 points, depending on how much understanding of the potential method you showed.
Problem 4. Converting Between Graph Representations (3 Parts, 30 Points).

Consider an undirected graph $G$ embedded in the plane such that no two edges of the graph meet except at vertices. Such a graph partitions the plane into a set of $F$ polygonal “faces,” each of which is bounded by three or more edges. If we include also the “infinite face,” then Euler’s relation holds that $V - E + F = 2$.

Suppose we are given the graph described not in the customary way, but as a list of independent faces. Each finite face is specified as a list of vertices in counter-clockwise order (the infinite face is specified in clockwise order). Each vertex is specified as two non-negative, integer-valued coordinates $x, y$ (no bound on the range of $x$ and $y$ is known in advance). Here is an example graph and its independent-faces description:

```
F1: [(2, 1), (5, 3), (1, 5)]
F2: [(5, 3), (8, 6), (5, 7), (1, 5)]
F3: [(9, 2), (8, 6), (5, 3)]
F4: [(5, 3), (2, 1), (6, 1)]
F5: [(6, 1), (9, 2), (5, 3)]
Finf: [(9, 2), (6, 1), (2, 1), (1, 5), (5, 7), (8, 6)]
```

The independent-faces description above is well-suited for some operations (e.g., real-estate tax assessment), but not for others. Consider, for example, running depth-first or breadth-first search, or finding a minimum spanning tree, on an independent-faces description. Thus the need arises to convert graphs from one representation to another.

A. Give an algorithm which takes as input an independent-faces description of a graph $G$, and produces as output an adjacency-list representation of $G$. Make your algorithm as efficient as you can; it is possible to achieve worst-case space usage, and expected running time, linear in $V+E+F$. To do so, you will probably have to use data structures we have seen in lecture. Slower or less space-efficient solutions will receive partial credit. Assume $G$ is connected, and all edges and vertices are part of at least one finite face.

(Hint 1: as your algorithm scans the input, how can it efficiently “recognize” vertices and/or edges that it has previously encountered? Ignore parsing details; assume that your parser correctly delimits faces and vertices in the input. Hint 2: note that every edge appears exactly twice in the input, once in each direction.)

B. Argue for the correctness of your algorithm.
C. Analyze your algorithm’s space usage and running time.

Solution.
The basic idea here was to use a hash function to recognize vertices (and therefore edges) that had been seen previously in the input. Our data structure is a hash table containing linked lists, each element of which represents a vertex in the graph. Hanging off each vertex \( u \) is a linked list of pointers to vertices found to be adjacent to \( u \). Whenever an edge \((u, v)\) is seen in the input, the vertex \( u \) is inserted into the hash table (or simply located if it is already there), and \( v \) is inserted onto \( u \)’s adjacency list. When the algorithm terminates, all adjacency lists are correct.

Here is pseudo-code for the process. It assumes that the input is parsed once (with no hashing) to determine the maximum \( y \) coordinate \( YMAX \) and the number of vertices \( V \) (using Euler’s relation on the known quantities \( F \) and \( E \)), then parsed again with hashing as shown below.

The function \texttt{UniversalHash} maps a single integer key to a valid index into the hash table. A \texttt{HashTable} entry contains a \texttt{Head} pointer, which points to a linked list of \texttt{HashList} records. A \texttt{HashList} record contains a pointer to a \texttt{Vertex} record, and a \texttt{Next} pointer to the next \texttt{HashList} record (or \texttt{NIL}). A \texttt{Vertex} record contains three fields: an \( x \) coordinate, an \( y \) coordinate, and a linked list of vertex \texttt{Adjacencies}.

The procedure \texttt{LinkedListSearch} scans a linked list for a vertex record containing \( x, y \), and returns it if one is found, otherwise returning \texttt{NIL}.

\begin{verbatim}
VertexPtr LinkedListSearch ( LinkedListHead h, Integer x, y )
    NodePointer L = h;
    while ( L != NIL )
        if ( L.VertexPtr.x == x and L.VertexPtr.y == y )
            return L.VertexPtr;
        L = L.Next
    return NIL;
\end{verbatim}

The procedure \texttt{HashVertex} converts the coordinates \( x, y \) to an integer key, which it then hashes with \texttt{UniversalHash}. If a vertex record for this vertex already exists, a pointer to it is returned. If not, a record is created and filled in, and a pointer to it is returned.

\begin{verbatim}
VertexPtr HashVertex ( Integer x, y )
    Integer h = UniversalHash ( x * YMAX + y );
    // Search attached linked list for vertex x,y
    VertexPtr = LinkedListSearch ( HashTable[h].Head, x, y );
    if ( VertexPtr == NIL )
        VertexPtr = NewVertexRecord; // allocate vertex record
        VertexPtr.x = x;
\end{verbatim}
VertexPtr.y = y;
VertexPtr.Adjacencies = NIL;
Push (HashTable[h].Head, VertexPtr ); // insert into HashTable
return VertexPtr;

The procedure InsertEdge takes a pair of vertices, inserts both vertices into the hashtable, and adds the second vertex to the adjacency list of the first vertex.

InsertEdge ( Integer x1, y1, x2, y2 )
VertexPtr u = HashVertex ( x1, y1 );
VertexPtr v = HashVertex ( x2, y2 );
Insert ( u.Adjacencies, v ); // at front of linked list

Finally, we scan through all faces and edges in the input, and insert each edge \((u, v)\). Note that to insert a face with vertices \(A, B\) and \(C\), we have to insert three edges: \((A, B), (B, C)\) and \((C, A)\).

For each input face \(F\)
// let number of vertices in this face == n
// vertices are numbered \(v_1, v_2, \ldots, v_n\)
For \(i = 1 \) to \(n\)
   \(j = i \mod n + 1\)
   InsertEdge ( \(x_i, y_i, x_j, y_j\) );

Most people realized that hashing vertices was the optimal solution method. Hashing solutions got full credit, or nearly so, with a few points taken off if people were excessively vague, for example if they didn’t spell out which type of hashing to use, or how to determine \(V\), or how many hash table entries to allocate \((O(V))\), or how to construct the hash key from \(x\) and \(y\). Sub-optimal but correct solutions lost 5 to 10 points. Incorrect solutions lost more.

B. Whenever edge \((u, v)\) is encountered, the adjacency \((u, v)\) will is recorded, since \(v\) is inserted onto the adjacency list of \(u\). Moreover, the structure of the input guarantees that if edge \((u, v)\) is seen in the input, then eventually edge \((v, u)\) will be seen as well (therefore the adjacency \((v, u)\) will be recorded – \(u\) will be inserted onto the adjacency list of \(v\)). Thus, after all edge insertions are completed, all adjacency lists will be correct.

Most people got full credit on this part.

C. Each search of the hash table takes \(O(1)\) expected time. There are two hash table searches per edge. The number of edges \(E\) is linear in \(F + V\), by Euler’s relation. Thus the algorithm runs in time linear in \(E\), and linear in the size of the input.

Most people got full credit on this part, or nearly so if they mis-stated the running time of hashing operations. People who chose a sub-optimal method in part A did not lose points in part C; they were graded on whether their analysis of their own proposed solution was correct.