1. Exercise 17-3-3

Let D_i be the heap after the ith operation, and let D_i consist of n_i elements. Also, let k be a constant such that each INSERT or EXTRACT-MIN operation takes at most $k \ln n$ time, where $n = \max(n_{i-1}, n_i)$. (We don't want to worry about taking the log of 0, and at least one of n_{i-1} and n_i is at least 1. We'll see later why we use the natural log.)

Define

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0\,, \\ kn_i \ln n_i & \text{if } n_i > 0\,. \end{cases}$$

This function exhibits the characteristics we like in a potential function: if we start with an empty heap, then $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \geq 0$.

Before proving that we achieve the desired amortized times, we show that if $n \geq 2$, then $n \ln \frac{n}{n-1} \leq 2$. We have

$$\begin{aligned} n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\ &= \ln \left(1 + \frac{1}{n-1}\right)^n \\ &\leq \ln \left(e^{\frac{1}{n-1}}\right)^n & \text{(since } 1 + x \leq e^x \text{ for all real } x\text{)} \\ &= \ln e^{\frac{n}{n-1}} \\ &= \frac{n}{n-1} \\ &\leq 2\,, \end{aligned}$$

assuming that $n \geq 2$. (The equation $\ln e^{\frac{n}{n-1}} = \frac{n}{n-1}$ is why we use the natural log.)

If the ith operation is an INSERT, then $n_i = n_{i-1} + 1$. If the ith operation inserts into an empty heap, then $n_i = 1$, $n_{i-1} = 0$, and the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq k \ln 1 + k \cdot 1 \ln 1 - 0 \\ &= 0\,. \end{aligned}$$

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$, and the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq k \ln n_i + kn_i \ln n_i - kn_{i-1} \ln n_{i-1} \\ &= k \ln n_i + kn_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\ &= k \ln n_i + kn_i \ln n_i - kn_i \ln(n_i - 1) + k \ln(n_i - 1) \\ &< 2k \ln n_i + kn_i \ln \frac{n_i}{n_i - 1} \\ &\leq 2k \ln n_i + 2k \\ &= O(\lg n_i)\,. \end{aligned}$$

If the ith operation is an EXTRACT-MIN, then $n_i = n_{i-1} - 1$. If the ith operation extracts the one and only heap item, then $n_i = 0$, $n_{i-1} = 1$, and the amortized cost is

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\le k \ln 1 + 0 - k \cdot 1 \ln 1 \\
&= 0 \,.
\end{aligned}$$

If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \ge 2$, and the amortized cost is

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\le k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\
&< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\
&\le k \ln 2 \qquad (\text{since } n_{i-1} \ge 2) \\
&= O(1) \,.
\end{aligned}$$

A slightly different potential function—which may be easier to work with—is as follows. For each node x in the heap, let $d_i(x)$ be the depth of x in D_i. Define

$$\begin{aligned}
\Phi(D_i) &= \sum_{x \in D_i} k(d_i(x) + 1) \\
&= k \left(n_i + \sum_{x \in D_i} d_i(x) \right) \,,
\end{aligned}$$

where k is defined as before.

Initially, the heap has no items, which means that the sum is over an empty set, and so $\Phi(D_0) = 0$. We always have $\Phi(D_i) \ge 0$, as required.

Observe that after an INSERT, the sum changes only by an amount equal to the depth of the new last node of the heap, which is $\lfloor \lg n_i \rfloor$. Thus, the change in potential due to an INSERT is $k(1 + \lfloor \lg n_i \rfloor)$, and so the amortized cost is $O(\lg n_i) + O(\lg n_i) = O(\lg n_i) = O(\lg n)$.

After an EXTRACT-MIN, the sum changes by the negative of the depth of the old last node in the heap, and so the potential *decreases* by $k(1 + \lfloor \lg n_{i-1} \rfloor)$. The amortized cost is at most $k \lg n_{i-1} - k(1 + \lfloor \lg n_{i-1} \rfloor) = O(1)$.

1.Exercise 17-3-6

Define two stacks of size n each. Lets call them E and D. ENQUEUE is implemented as a push onto the E stack. DEQUEUE is implemented as a pop from the D stack. the queue items are moved from E to D whenever D is empty and DEQUEUE is called.

```
Function:Move-From-E-to-D(E, D)
while TRUE do
  x = POP(E)
  if NOT STACK-EMPTY(E) then
    PUSH(D,x)
  else
    return x
  end if
end while


Function:ENQUEUE(E, x)
PUSH(E,x)


Function:DEQUEUE(E, D)
if STACK-EMPTY(D) then
  return Move-From-E-to-D(E, D)
else
  return POP(D)
end if
```

Analysis using accounting method:
Enqueue is assigned an amortized cost of 4, Dequeue is assigned amortized cost of 0. A sequence of k enqueue operations leaves a credit of $3 * k$. The first Dequeue uses $2 * k$ credits. The subsequent dequeues use 1 credit each. So amortized cost of 4 per operation, or $O(1)$.

1.Exercise 17-4-3

Notice that the i-th operation cannot cause the table to contract since contract only occurs when $\alpha_i < 1/3$. We need to consider cases for the load factor α_i. For both cases $num_i = num_{i-1} - 1$.

And $size_i = 2/3 * size_{i-1}$.

There fore, $size_i/2 = size_{i-1}/3 = num_{i-1} = num_i + 1$.

Assume $\alpha_i \geq 1/3$. Then $c_i = 1$

$$\begin{aligned}\hat{c}_i &= c_i + \phi_i - \phi_{i-1}\\ &= 1 + |(2 * num_I - size_i)| - |(2 * num_{i-1} - size_{i-1})|\\ &= 1 + |(2 * num_I - size_i)| - |(2 * (num_i + 1) - size_i)|\\ &= 1 + |(2 * num_I - size_i)| - |(2 * num_i + 2 - size_i)|\\ &\leq 1 + |(2 * num_I - size_i)| - |(2 * num_i - size_i)| + 2\\ &= 3\end{aligned}$$

Then consider $\alpha_i < 1/3$, Then $c_i = num_i + 1$,

$$\begin{aligned}\hat{c}_i &= c_i + \phi_i - \phi_{i-1}\\ &= num_i + 1 + |2 * num_i - size_i| - |2 * num_{i-1} - size_{i-1}|\\ &= num_i + 1 - 2 * num_i + size_i + 2 * num_{i-1} - size_{i-1}\\ &= num_i + 1 - 2 * num_{i-1} + 2 + size_i - 2 * num_{i-1} - size_{i-1}\\ &= num_i + 3 + size_i - size_{i-1}\\ &= num_i + 3 - size_{i-1}/3\\ &= size_{i-1}/3 - 1 + 3 - size_{i-1}/3\\ &= 2\end{aligned}$$

2. Exercises 22-1-6

We start by observing that if $a_{ij} = 1$, so that $(i, j) \in E$, then vertex i cannot be a universal sink, for it has an outgoing edge. Thus, if row i contains a 1, then vertex i cannot be a universal sink. This observation also means that if there is a self-loop (i,i), then vertex i is not a universal sink. Now suppose that $a_{ij} = 0$, so that $(i, j) \notin E$, and also that $i \neq j$. Then vertex j cannot be a universal sink, for either its in-degree must be strictly less than $|V| - 1$ or it has a self-loop. Thus if column j contains a 0 in any position other than the diagonal entry (j, j), then vertex j cannot be a universal sink.

Using the above observations, the following procedure returns TRUE if vertex k is a universal sink, and FALSE otherwise. It takes as input a$|V| \times |V|$ adjacency matrix $A = (a_i$

j).

```
IS-SINK($A$,$k$)
let $A$ be $|V| \times |V|$
for $j \leftarrow 1$ to $|V|$
    do if $a_{kj} = 1$
        then return FALSE
for $i \leftarrow 1$ to $|V|$
    do if $a_{ik} = 0$ and $i \neq k$
        then return FALSE
return TRUE
```

Because this procedure runs in $O(V)$ time, we may call it only $O(1)$ times in order to achieve our $O(V)$-time bound for determining whether directed graph G contains a universal sink.

Observe also that a directed graph can have at most one universal sink. This property holds because if vertex j is a universal sink, then we would have $(i, j) \in E$ for all $i \neq j$ and so no other vertex i could be a universal sink.

The following procedure takes an adjacency matrix A as input and returns either a message that there is no universal sink or a message containing the identity of the universal sink. It works by eliminating all but one vertex as a potential universal sink and then checking the remaining candidate vertex by a single call to IS-SINK.

```
let $A$ be $|V| \times |V|$
$i \leftarrow j \leftarrow 1$
while $i \leq |V|$ and $j \leq |V|$
    do if $a_{ij} = 1$
        then $i \leftarrow i + 1$
        else $j \leftarrow j + 1$
$s \leftarrow 0$
if $i > |V|$
    then return “there is no universal sink”
elseif IS-SINK($A$,$i$) = FALSE
    then return ìthere is no universal sinkî
```

else return i "is a universal sink"

UNIVERSAL-SINK walks through the adjacency matrix, starting at the upper left corner and always moving either right or down by one position, depending on whether the current entry a_{ij} it is examining is 0 or 1. It stops once either i or j exceeds $|V|$.

To understand why UNIVERSAL-SINK works, we need to show that after the **while** loop terminates, the only vertex that might be a universal sink is vertex i. The call to IS-SINK then determines whether vertex i is indeed a universal sink.

Let us fix i and j to be values of these variables at the termination of the **while** loop. We claim that every vertex k such that $1 \leq k < i$ cannot be a universal sink. That is because the way that i achieved its final value at loop termination was by finding a 1 in each row k for which $1 \leq k < i$. As we observed above, any vertex k whose row contains a 1 cannot be a universal sink.

If $i > |V|$ at loop termination, then we have eliminated all vertices from consid- eration, and so there is no universal sink. If, on the other hand, $i \leq |V|$ at loop termination, we need to show that every vertex k such that $i < k \leq |V|$ cannot be a universal sink. If $i \leq |V|$ at loop termination, then the **while** loop terminated because $j > |V|$. That means that we found a 0 in every column. Recall our earlier observation that if column k contains a 0 in an off-diagonal position, then vertex k cannot be a universal sink. Since we found a 0 in every column, we found a 0 in every column k such that $i < k \leq |V|$. Moreover, we never examined any matrix entries in rows greater than i, and so we never examined the diagonal entry in any column k such that $i < k \leq |V|$. Therefore, all the 0s that we found in columns k such that $i < k \leq |V|$ were off-diagonal. We conclude that every vertex k such that $i < k \leq |V|$ cannot be a universal sink.

Thus, we have shown that every vertex less than i and every vertex greater than i cannot be a universal sink. The only remaining possibility is that vertex i might be a universal sink, and the call to IS-SINK checks whether it is.

To see that UNIVERSAL-SINK runs in $O(V)$ time, observe that either i or j is incremented in each iteration of the **while** loop. Thus, the **while** loop makes at most $2\,|V| - 1$ iterations. Each iteration takes $O(1)$ time, for a total **while** loop time of $O(V)$ and, combined with the $O(V)$-time call to IS-SINK, we get a total running time of $O(V)$.

3. Exercise 22-2-9

Using a variant of depth-first search. Every edge is marked the first and second time it is traversed with unique marks for each traversal. Edges that have been traversed twice may not be taken again.

Our depth-first search algorithm must ensure that all edges all explored, and that each edge is taken in both directions. To ensure that all edges are explored, the algorithm must ensure that unexplored edges are always taken before edges that are explored once. To ensure that edges are taken in each direction, we simply backtrack every time the depth-first search reaches a dead-end. The search keeps backtracking until a new unexplored edge is found. This way, edges are only explored in the reverse direction during the backtracking.

Complexity: This algorithm is based on depth-first search, which has time complexity O(V+ E).

3. (Problem 23.3)
(a) Suppose that an MST T of a graph G is not a bottle-neck spanning tree. Say that the maximum weight edge in T is (u,v).Imaging removing (u,v), the resulting disconnected set of vertices can be viewed as a cut of the graph. It must be the case that (u,v) must be a light edge for this cut otherwise T would not be an MST. Because every possible MST select one edge for this particular cut we guaranteed that every possible MST must contain (u,v) or another edge for this cut with no less weight than (u,v). Thus by contradiction, we cannot construct another spanning tree in which the heaviest edge is lighter than (u,v).

(b) Remove all edges from the graph G. Now for each edge, see if it's weight is less than or equal to b. If so, then add it back to the graph. If the resulting graph G' is connected then we know that it is possible to construct a spanning tree while using no edge with weight larger than b. This algorithm runs in linear time in O(E).
(C)
Run the algorithm in part (b) repeatedly for a dressing value of b each time. Start with the value of b set to the heaviest edge in the set of edges, and repeatedly run the algorithm for a smaller and smaller value of b until the algorithm returns a NO as the answer. We then know the bottleneck value b for this graph.
Obtain the set of all edges with weight less than or equal to b. From this set randomly pick an edge and similar to KRUSKAL's algorithm, run two instances of FIND and one instance of UNION for that edge. If we use an implementation of FIND-UNION data structure that uses pth-compression and union-by-rank heuristics. Thus we perform O(E) disjoint set operations in the worst case (each costing O(1)) and obtain a bottle-neck spanning tree.

4. (Problem 24-6)
Observe that a bitonic sequence can increase, then decrease, then increase, or it can decrease, then increase, then decrease. That is, there can be at most two changes of direction in a bitonic sequence. Any sequence that increases, then decreases, then increases, then decreases has a bitonic sequence as a subsequence.

Now, let us suppose that we had an even stronger condition than the bitonic prop- erty given in the problem: for each vertex v $\in V$, the weights of the edges along any shortest path from s to v are increasing. Then we could call INITIALIZE- SINGLE-SOURCE and then just relax all edges one time, going in increasing order of weight. Then the edges along every shortest path would be relaxed in order of their appearance on the path. (We rely on the uniqueness of edge weights to en- sure that the ordering is correct. [Note that the uniqueness assumption was added in the fifth printing of the text.]) The path-relaxation property (Lemma 24.15) would guarantee that we would have computed correct shortest paths from *s* to each vertex.

If we weaken the condition so that the weights of the edges along any shortest path increase and then decrease, we could relax all edges one time, in increasing order of weight, and then one more time, in decreasing order of weight. That order, along with uniqueness of edge weights, would ensure that we had relaxed the edges of every shortest path in order, and again the path-relaxation property would guarantee that we would have computed correct shortest paths.

To make sure that we handle all bitonic sequences, we do as suggested above. That is, we perform four passes, relaxing each edge once in each pass. The first and third passes relax edges in increasing order of weight, and the second and fourth passes in decreasing order. Again, by the path-relaxation property and the uniqueness of edge weights, we have computed correct shortest paths.

The total time is $O(V + E \lg V)$, as follows. The time to sort $|E|$ edges by weight is $O(E \lg E) = O(E \lg V)$ (since $|E| = O(V^2)$). INITIALIZE-SINGLE-SOURCE takes $O(V)$ time. Each of the four passes takes $O(E)$ time. Thus, the total time is $O(E \lg V + V + E) = O(V + E \lg V)$.

5. (Problem 25-1)

Let T be the $|V| \times |V|$ matrix representing the transitive closure, such that $T[i,j]$ is1 if there is a path from I to j, and 0 if not.

Initialize T (when there are no edges in G) as follows:

$T[i,j]=$ 1 if i=j

0 otherwise

T can be updated as follows when an edge (u, v) is added to G:

TRANSITIVE-CLOSURE-UPDATE(u,v)

```
for $i \leftarrow 1$ to $|V|$ do for $j \leftarrow 1$ to $|V|$
```

do if $T[i,u] = 1$ and $T[v, j] = 1$ **then** $T[i, j] \leftarrow 1$

- This says that the effect of adding edge (u, v) is to create a path (via the new edge) from every vertex that could already reach u to every vertex that could already be reached from v.
- Note that the procedure sets $T[u,v] \leftarrow 1$, because of the initial values $T[u,u] = T[v,v] = 1$.
- This takes teta(V^2) time because of the two nested loops.

b. Consider inserting the edge (v_n , v_1) into the straight-line graph $v_1 \to v_2 \to \cdots \to v_n$, where n = |V|. Before this edge is inserted, only n(n + 1)/2 entries in T are 1 (the entries on and above the main diagonal). After the edge is inserted, the graph is a cycle in which every vertex can reach every other vertex, so all n^2 entries in T are 1. Hence $n^2 - (n(n + 1)/2)$ = teta(n^2) =teta(V^2) entries must be changed in T , so any algorithm to update the transitive closure must take omega(V^2) time on this graph.

c. The algorithm in part (a) would take teta(V^4) time to insert all possible teta(V^2) edges, so we need a more efficient algorithm in order for any sequence of in- sertions to take only O(V^3) total time. To improve the algorithm, notice that the loop over j is pointless when T[i,v] = 1. That is, if there is already a path i ; v, then adding the edge (u,v) canít make any new vertices reachable from i . The loop to set T [i, j] to 1 for j such that thereís a path v ; j is just setting entries that are already 1. Eliminate this redundant processing as follows: TRANSITIVE-CLOSURE-UPDATE(u, v) for i ← 1 to |V | do if T [i, u] = 1 and T [i, v] = 0 then for j ← 1 to |V | do if T[v, j] = 1 then T[i, j] ← 1 We show that this procedure takes O(V^3) time to update the transitive closure for any sequence of n insertions:

- Therecanítbemorethan$|V|^2$ edgesinG,son$\leq|V|^2$.
- Summed over n insertions, time for the first two lines is O(nV) = O(V^3).
- The last three lines, which take teta(V) time, are executed only O(V^2) times for n insertions. To see this, notice that the last three lines are executed only when T[i,v] = 0, and in that case, the last line sets T[i,v] ← 1. Thus, the number of 0 entries in T is reduced by at least 1 each time the last three lines run. Since there are only $|V|^2$ entries in T , these lines can run at most $|V|^2$ times.
- Hence the total running time over n insertions is O(V^3).