1A. $3^{n+2} = 3^2 \cdot 3^n = 9 \cdot 3^n \leq 9 \cdot 3^n$ for $n > 0$

Here, $3^{n+2} \in \Theta(3^n)$

$3^{2n} = 3^n \cdot 3^n$. Suppose $3^n \cdot 3^n \leq c \cdot 3^n$ for some $c$ and $n > n_0$. That would mean $3^n \leq c$ for $n > n_0$. However, note that $3^k > k$ for $k \geq 0$. So choose $j \geq \max(c, n_0)$, and $3^j > j \geq c$. So no such $c$ exists and $3^{2n} \notin \Theta(3^n)$.

1B. 

1C. The strategy is to swap the deleted element with the last element in the heap, then remake the heap. A critical observation is that the moved element may need to move upward in the heap to get to the correct position.
There is an existing procedure, heap-Increase-Key that will "bubble" an element upward. If the new element at \( A[i] \) is greater than the previous element, we call that procedure. If not, the new element may need to move downward to restore the heap, which Max-Heapify will do.

```
Heap-Delete (A, i)
```

```
V = A[i]
A[i] = \text{heapsize}[A] - 1
if A[i] > V
   then Heap-Increase-Key (A, i, \text{Al}[i])
else Max-Heapify (A, i)
```

Note that whichever branch of the if is taken, the amount of work is bounded by the height of the heap, which is \( O(\log n) \).

Example: Suppose we call Heap-Delete (A, 3) where A is as shown in \( V \) of the answer to 1B. Then after the swap and shortening of the heap, we have:

\( V = 20 \) since 10 \( \geq \) 20, we call Max-Heapify (A, 3) to get:
1D. (a) \( f(n) \) in \( O(g(n)) \) does not imply \( g(n) \in O(f(n)) \)

Counter-example: \( f(n) = n \), \( g(n) = n^2 \).

(b) \( f(n) \) in \( O(g(n)) \) implies \( O(f(n) + g(n)) = O(g(n)) \)

- Clearly \( O(g(n)) \subseteq O(f(n) + g(n)) \), since any function \( h(n) \leq c \cdot g(n) \) for \( n > n_0 \), we also have \( h(n) \leq c \cdot (f(n) + g(n)) \) for \( n > n_0 \).

- Suppose \( h(n) \in O(f(n) + g(n)) \). Then there is a \( d > 0 \) and an \( n_1 \), such that \( h(n) \leq d \cdot (f(n) + g(n)) \), \( n > n_1 \).

Since \( f(n) \in O(g(n)) \), there is a \( e > 0 \) and an \( n_2 \) such that \( f(n) \leq e \cdot g(n) \) for \( n > n_2 \).

- Let \( b = d \cdot (e+1) \) and \( n_3 = \max(n_1, n_2) \). Then

\[ b \cdot g(n) = d \cdot (e \cdot g(n) + g(n)) \geq d \cdot (f(n) + g(n)) \geq h(n). \]

So \( h(n) \in O(g(n)) \). Since \( h(n) \) was arbitrary,

\[ O(f(n) + g(n)) \subseteq O(g(n)). \]

Hence \( O(f(n) + g(n)) = O(g(n)) \).

(c) \( f(n) \) in \( O(g(n)) \) does not imply \( 2^f(n) \) in \( O(2^g(n)) \).

Counter-example \( f(n) = 2n \), \( g(n) = n \). Similarly to 1-A, we know that \( 2^{2n} \notin O(2^n) \).

(d) \( f(n) \in O(g(n)) \) implies \( \log f(n) \in O(\log g(n)) \).

If \( f(n) \in O(g(n)) \), then there are \( c > 0 \) and \( n_0 \) with \( f(n) \leq c \cdot g(n) \) for \( n > n_0 \). Since \( g \) is an increasing function, the inequality is preserved if we take logs of both sides: \( \log f(n) \leq \log (c \cdot g(n)) = \log c + \log g(n) \) for \( n > n_0 \).

Let \( d = \log c + 1 \). Then \( d \cdot (\log g(n)) = (\log c) \cdot (\log g(n)) + \log g(n) \geq \log c + \log g(n) \geq \log f(n) \) for \( n > n_0 \). So \( \log f(n) \in O(\log g(n)) \). [I guess this assumes \( g(n) \neq 2 \) for \( n > n_0 \).]
1. The list elements need to be turned into (key, list#) pairs. [There can also be a pointer to the record, if it is longer than just the key.] Note that each component comes from a bounded range: \(1 \leq \text{key} \leq m, 1 \leq \text{list#} \leq k\), which should suggest something like counting sort or bucket sort, which are \(O(k + r)\) for \(r\) elements in the range \(1..r\).

**Pseudo code**

1. Create a list \(L\) of pairs \((i, j)\) where \(i\) is the key of a record in list \(j\). \(O(kn)\)

2. Bucket sort \(L\) on the key component \(O(kn + m)\)

3. Bucket sort \(L\) on the list# component \(O(kn + k) = O(kn)\)

4. Each bucket contains a sorted list in its first components \(O(kn)\)

* The sort used needs to be stable: it keeps elements with the same list# in the same order on key.

**Complexity:** \(O(kn + m)\) is the dominant cost.

**Example:** \(m = 6, k = 3, n = 4\)

\(L_1 = 1, 6, 4, 4 \quad L_2 = 2, 5, 2, 4 \quad L_3 = 3, 2, 4, 6\)

\(L = (1,1), (6,1), (4,1), (4,1), (2,2), (5,2), (2,2), (4,2), (3,3), (2,3), (4,3), (6,3)\)

**Bucket sort 1:**

\[\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
\end{array} \rightarrow \begin{array}{c}
(1,1) \\
(2,2) \rightarrow (2,2) \rightarrow (2,3) \\
(3,3) \\
(4,1) \rightarrow (4,1) \rightarrow (4,2) \rightarrow (4,3) \\
(5,2) \\
(6,1) \rightarrow (6,3) \\
\end{array}\]

**Bucket sort 2:**

\[\begin{array}{c}
1 \\
2 \\
3 \\
\end{array} \rightarrow \begin{array}{c}
(1,1) \rightarrow (1,1) \rightarrow (1,1) \rightarrow (6,1) \\
(2,2) \rightarrow (2,2) \rightarrow (4,2) \rightarrow (5,2) \\
(3,3) \rightarrow (3,3) \rightarrow (4,3) \rightarrow (6,3) \\
\end{array}\]

**Sorted lists**

\[\begin{array}{c}
L_1 = 1, 4, 4, 6 \\
L_2 = 2, 2, 4, 5 \\
L_3 = 3, 3, 4, 6 \\
\end{array}\]
There are two changes to the Freq algorithm. One is that we always use \( \text{Median}(S) \) to choose \( c \) for splitting \( S \) into \( S_{-\text{low}} \) and \( S_{-\text{high}} \). With just this change, we are guaranteed an \( O(n \lg n) \) algorithm, since the depth of recursion is at most \( \lg n \), and the combined sizes of subproblems at each level is \( < n \).

The second change is to always work on the largest remaining subproblem first, and never call a subproblem of size less than the last most frequent item found so far. This policy guarantees that we will discover \( m \) before we call any problems of size \( \leq k \). We can use a priority queue (say, implemented as a heap) to find the largest remaining subproblem at any point. The policy cuts off any recursive calls of size \( < k \), hence the bottom \( \lg k \) levels of the call tree are avoided.

Also note that there are at most \( \frac{n}{k} \) subproblems of size \( \geq k \), so the heap never gets deeper than \( \lg \frac{n}{k} \) levels. So any \textit{Extract-Min} or \textit{Insert} is \( O(\lg \frac{n}{k}) \).

Since the combined size of problems at each level is still \( < n \), we get the \( O(n \lg \frac{n}{k}) \) bound on time.
Example 1 = order of subproblems

1. \(\langle 4, 2, 1, 6, 6, 1, 2, 2, 8, 4 \rangle\)  Median = 4
2. \(\langle 2, 1, 1, 2, 2 \rangle\)  \(e = 1\), \(f = 2\)  Median = 2
3. \(\langle 6, 6, 8 \rangle\)  Median = 6
4. \(\langle 2, 2, 2 \rangle\)  Median = 2
5. \(\langle 8 \rangle\)

This subproblem is never called.