Problem 1. Prove that our algorithm for the “stack-with-array problem” takes $O(n)$ time to process any sequence of $n$ operations where each operation can be either a push or a pop.

Solution. Refer to an array expansion or shrinking collectively as an overhaul. If an overhaul happens when the stack has $m$ elements, the overhaul incurs $O(m)$ cost. It suffices to prove that we can always charge the cost on $\Omega(m)$ operations, such that each operation is charged only once.

Suppose that the overhaul is an expansion at $m$. Consider the moment right after the current array (with length $m$) was created. As a new array is always half full, we know that the array contained exactly $m/2$ elements at that moment. We charge the $O(m)$ overhaul cost on the (at least) $m/2$ pushes that must have occurred since that moment.

Suppose that the overhaul is a shrinking at $m$, which means there are only $m/4$ elements left in the array. Consider the moment right after the current array (with length $m$) was created. As before, the array contained exactly $m/2$ elements at that moment. We charge the $O(m)$ overhaul cost on the (at least) $m/2 - m/4 = m/4$ pops that must have occurred since that moment.

This completes the proof.

Problem 2. Let $S$ be a multi-set of $n$ integers. Define the frequency of an integer $x$ as the number of occurrences of $x$ in $S$. Design an algorithm to produce an array that sorts the distinct integers in $S$ by frequency. Your algorithm must terminate in $O(n)$ expected time. For example, suppose that $S = \{75, 123, 65, 75, 9, 9, 65, 9, 93\}$. Then you should output $(123, 93, 65, 75, 9)$. Note that if two integers have the same frequency, their relative ordering is unimportant. For example, $(93, 123, 75, 65, 9)$ is another legal output.

Solution. We can collect the set $T$ of distinct integers in $S$ by hashing as follows. For every integer $x \in S$, check whether the hash table has already contained a copy of $x$. This takes $O(1)$ in expectation. If so, ignore $x$; otherwise, insert $x$ into the hash table in $O(1)$ time. The collection requires $O(n)$ time overall.

We can then obtain the frequency of every distinct integer as follows. For each integer $x \in S$, find its copy in the hash table, and increase the counter of the copy by 1 (the counter initially set to 0). This takes $O(1)$ time per integer, and hence, $O(n)$ time overall.

Now we simply sort all the distinct integers by frequency. Note that the frequencies are in the domain from 1 to $n$. Hence, counting sort gets this done in $O(n)$ time.

Problem 3* (Textbook Exercise 17.3-7). Suppose that we want to implement the following two operations on a set $S$ of integers ($S$ is empty at the beginning):

- Insert($e$): Add a new integer $e$ into $S$ (you are assured that $e$ is not already in $S$).
- Delete-Half: Delete the $\lceil |S|/2 \rceil$ smallest elements from $S$. 
Describe a data structure that consumes $O(|S|)$ space, and supports each operation in $O(\log |S|)$ time amortized.

For students in the training camp: improve the operation bound to $O(1)$ amortized!

**Solution.** Simply store all the elements of $S$ in a linked list. Every insertion takes $O(1)$ time. To perform a delete-half operation, move all the elements in $S$ to an array of length $|S|$, and sort them in $O(|S| \log |S|)$ time. Then, remove the smallest $\lfloor |S|/2 \rfloor$ elements in $O(|S|)$ time. Charge the $O(|S| \log |S|)$ time on the insertions that added those elements. Each insertion bears only $O(\log |S|)$ extra cost. Every insertion can be charged only once.

For the training camp, replace sorting with median selection.

**Problem 4.** Recall that our algorithm for the “dynamic array problem” (i.e., insertion only) ensures that if the current set $S$ has $n$ elements, the array has a length of at most $2n$. Modify the algorithm to ensure that our array has length at most $1.5n$. You will still need to process an insertion in $O(1)$ amortized time.

**Solution.** When the current array is full, simply make the length of the new array $\lfloor 1.5n \rfloor$. Charge the $O(n)$ expansion cost on the $\lfloor 0.5n \rfloor$ elements inserted since the previous expansion. The amortized insertion time is therefore still $O(1)$.

**Problem 5.** Let $S$ be a set of $n$ key-value pairs of the form $(k, v)$, where $k$ is the key and $v$ is the value. Preprocess $S$ into a data structure so that the following queries can be answered efficiently. Given a pair $(q_k, q_v)$, a query

- Returns nothing if $S$ contains no pair with key $q_k$;
- Otherwise, it returns the number of pairs $(k, v) \in S$ such that $k = q_k$ and $v \leq q_v$.

Define the frequency of a key $k$ as the number of pairs in $S$ with key $k$. Define $f$ as the maximum frequency of all keys. Your structure must use $O(n)$ space, and answer a query in $O(\log f)$ expected time. Furthermore, it must be possible to construct the structure $O(n \log f)$ time.

For example, suppose that $S = \{(75, 35), (123, 6), (65, 32), (75, 22), (9, 1), (9, 10), (65, 74), (9, 8), (93, 23)\}$. Then, given $(63, 33)$, the query returns nothing. Given $(65, 33)$, the query returns 1. Given $(65, 2)$, the query returns 0. In this example, $f = 3$.

**Solution.** Collect the set $T$ of distinct keys in $S$, and obtain their frequencies in $O(n)$ time (see the solution of Problem 2). Create a hash table on $T$ in $O(n)$ time. For every key $k \in T$, create an array $A_k$ whose length is equal to the frequency of $k$. Store in $A_k$ all the values $v$ such that $(k, v)$ is a pair in $S$. Sort $A_k$ in ascending order. The sorting takes $O(|A_k| \log |A_k|) = O(|A_k| \log f)$ time. Store the beginning address of $A_k$ at the copy of $k$ in the hash table. The overall construction time is $O(\sum_k |A_k| \log f) = O(n \log f)$. The space consumption is obviously $O(n)$.

To answer a query $(q_k, q_v)$, first probe the hash table to see if $q_k \in T$. If not, terminate the algorithm. Otherwise, perform binary search in $A_k$ in $O(\log f)$ time. The overall query time is $O(1)$ expected plus $O(\log f)$ worst case, which is $O(\log f)$ expected.